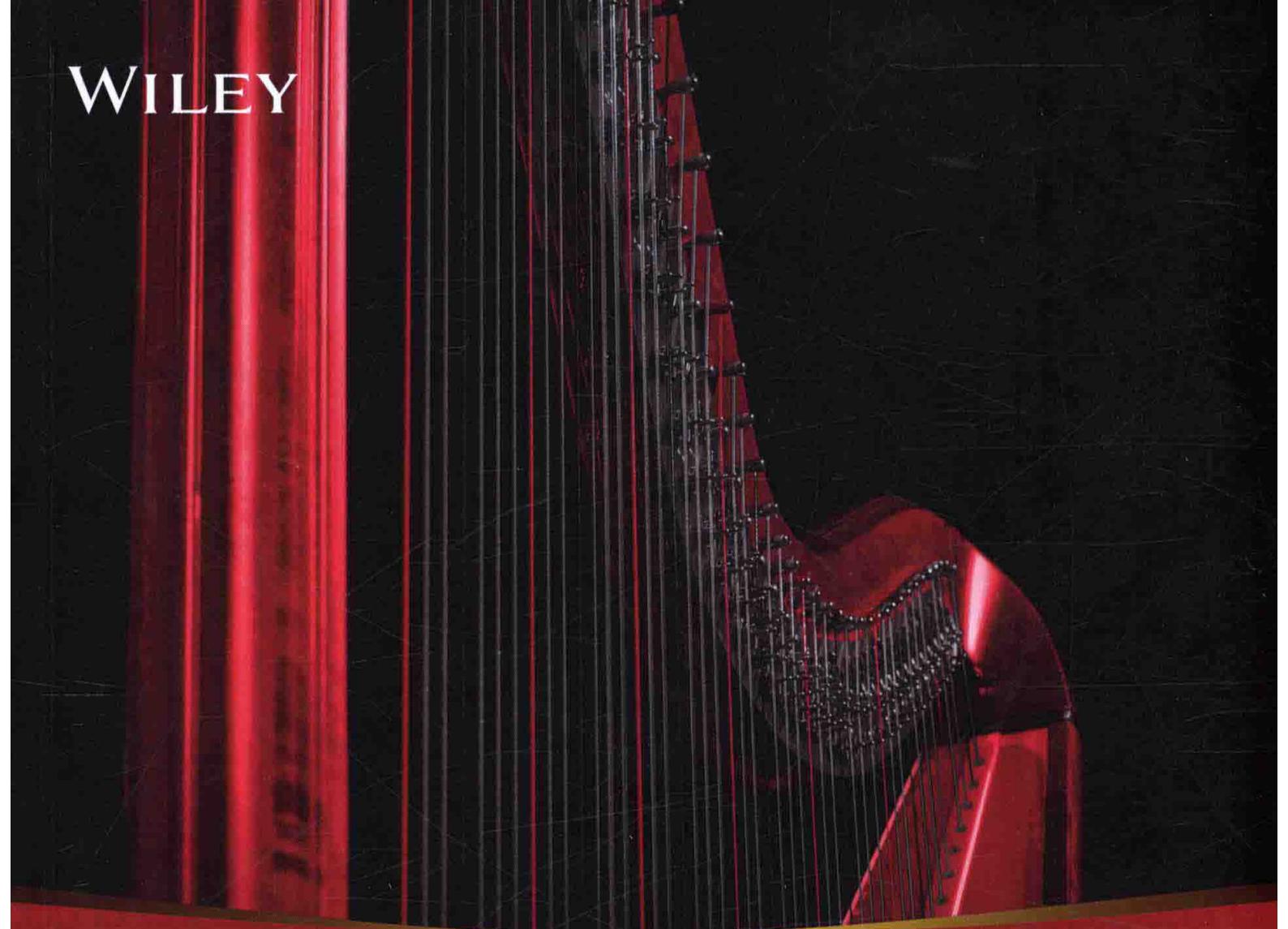


WILEY



Professional C++, Fifth Edition

# C++20高级编程

(第5版) (上册)

[比] 马克·格雷戈勒(Marc Gregoire) 著  
程序喵大人 惠惠 墨梵 译



清华大学出版社

# C++20 高级编程

## (第5版)

### (上册)

[比] 马克·格雷戈勒(Marc Gregoire) 著  
程序喵大人 惠惠 墨梵 译

清华大学出版社  
北京

北京市版权局著作权合同登记号 图字：01-2021-3641

All Rights Reserved. This translation published under license. Authorized translation from the English language edition, entitled Professional C++, Fifth Edition, ISBN 9781119695400, by Marc Gregoire, Published by John Wiley & Sons. Copyright © 2021 by Marc Gregoire. No part of this book may be reproduced in any form without the written permission of the original copyrights holder.

Copies of this book sold without a Wiley sticker on the cover are unauthorized and illegal.

本书中文简体字版由 Wiley Publishing, Inc. 授权清华大学出版社出版。未经出版者书面许可，不得以任何方式复制或传播本书内容。

本书封面贴有 Wiley 公司防伪标签，无标签者不得销售。

版权所有，侵权必究。举报：010-62782989，beiqinquan@tup.tsinghua.edu.cn。

#### 图书在版编目(CIP)数据

C++20高级编程：第5版 / (比)马克·格雷戈勒(Marc Gregoire)著；程序喵大人，惠惠，墨梵译. —北京：清华大学出版社，2022.3

书名原文：Professional C++, Fifth Edition

ISBN 978-7-302-60213-2

I. ①C… II. ①马… ②程… ③惠… ④墨… III. ①C++语言—程序设计 IV. ①TP312.8

中国版本图书馆 CIP 数据核字(2022)第 033304 号

责任编辑：王军

装帧设计：孔祥峰

责任校对：成凤进

责任印制：宋林

出版发行：清华大学出版社

网    址：<http://www.tup.com.cn>, <http://www.wqbook.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](mailto:c-service@tup.tsinghua.edu.cn)

质量反馈：010-62772015, [zhiliang@tup.tsinghua.edu.cn](mailto:zhiliang@tup.tsinghua.edu.cn)

印装者：北京同文印刷有限责任公司

经    销：全国新华书店

开    本：170mm×240mm        印    张：62.5        字    数：1805 千字

版    次：2022 年 4 月第 1 版        印    次：2022 年 4 月第 1 次印刷

定    价：228.00 元(全二册)

---

产品编号：091909-01

## 作者简介

Marc Gregoire是微软Visual C++的MVP、软件架构师和开发人员，比利时C++用户组的创始人。他曾为西门子和诺基亚西门子网络公司完成关键的2G和3G通信软件，目前在Nikon Metrology公司工作，负责开发X射线、CT和3D几何检测软件。Marc是该系列书第2版到第4版的作者，并担任多本IT图书的技术编辑。

## 拥抱C++的深度和复杂性，挖掘更多可能

众所周知，C++难以掌握，但其广泛的功能使其成为游戏和商业软件应用程序中最常用的语言。即使是有经验的用户通常也不熟悉许多高级特性，但C++20的发布提供了探索该语言全部功能的绝佳机会。《C++20高级编程(第5版)》为C++的必要内容提供了一个代码密集型、面向解决方案的指南，包括最新版本中的最新工具和功能。本书包含面向现实世界编程的实用指导，是程序员深入研究C++的理想机会。第5版涵盖了C++20的内容。

### 主要内容

- 演示如何用C++思考，以最大限度地发挥语言的深  
远能力，并开发有效的解决方案
- 解释难以理解的原理，进行陷阱警告，分享提高  
效率和性能的提示、技巧和变通方法
- 呈现各种具有挑战性的、真实世界的程序，其用  
途广泛，足以融入任何项目
- 重点介绍C++20的新特性，包括模块、概念、三  
向比较、立即执行函数等
- 深入讨论新的C++20标准库功能，例如文本格式  
化、范围、原子智能指针、同步原语、日期、时  
区等

# 译 者 序

本书经典，且内容丰富，不仅有 C++ 基础知识，也有很多 C++ 高级功能，特别是 C++20 的新特性。目前介绍 C++ 基础知识的书籍很多，但介绍 C++20 新特性的书籍却不多，而既介绍 C++ 基础知识又介绍 C++20 新特性的书可以说几乎没有。另外本书还重点介绍了很多编程哲学，包括 C++ 的设计方法论，还从专业角度分析了 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进阶。

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，近期又更新了 C++20 新特性。作为一名 C++ 程序员，很有必要了解语言最新的变革。而且 C++20 标准新引入了很多有用的内容，例如模块化、协程、`std::format`、`std::jthread` 等。读者在学习本书 C++20 新特性的时候，可以多做一层思考，思考为什么标准委员会要引入此新特性。

要成为资深 C++ 开发人员，必须扎实理解 C++ 语言的底层原理，了解不同的编程哲学和软件工程方法论，如何设计和编码，如何测试，如何调试，如何优化等。巧了，这些知识，本书都有介绍。

本书包括 5 部分。第 I 部分是 C++ 基础速成教程，第 II 部分介绍 C++ 设计方法论，第 III 部分从专业角度分析 C++ 编程技术，第 IV 部分讲解如何真正掌握 C++ 的高级功能，第 V 部分重点介绍 C++ 软件工程技术。最后，附录 A 阐述了在 C++ 技术面试中取得成功的指南，附录 B 总结标准的 C++ 头文件，附录 C 则简要介绍 UML。

对于这本经典之作，译者在翻译过程中力求忠于原文，再现原文风格，但是鉴于译者水平有限，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，感谢妻子和父母家人在各方面的支持和理解。

最后，希望读者通过阅读本书能在 C++ 领域有更深的造诣，领略 C++ 语言之美，“精通” C++。

译者

# 作者简介

Marc Gregoire 是一名软件工程师，毕业于比利时鲁汶大学，拥有计算机科学工程硕士学位。之后，他在鲁汶大学又获得人工智能专业的优等硕士学位。完成学业后，他开始为软件咨询公司 Ordina Belgium 工作。他曾在 Siemens 和 Nokia Siemens Networks 为大型电信运营商提供有关在 Solaris 上运行关键 2G 和 3G 软件的咨询服务。这份工作要求与来自南美、美国、欧洲、中东、非洲和亚洲的国际团队合作。Marc 目前担任 Nikon Metrology([www.nikonmetrology.com](http://www.nikonmetrology.com))的软件架构师；Nikon Metrology 是 Nikon 的一个部门，是精密光学仪器、X 光机等设备和 X 光、CT 和 3D 扫描解决方案的领先供应商。

Marc 的主要技术专长是 C/C++，特别是 Microsoft VC++ 和 MFC 框架。他还擅长在 Windows 和 Linux 平台上开发 24×7 运行的 C++ 程序，例如 KNX/EIB 家庭自动化监控软件。除了 C/C++ 之外，Marc 还喜欢 C#。

2007 年 4 月，他凭借 Visual C++ 方面的专业技能，获得了微软年度 MVP 称号。

Marc 还是比利时 C++ 用户组([www.becpp.org](http://www.becpp.org))的创始人，是 *C++ Standard Library Quick Reference* 第 1 版和第 2 版(Apress)的共同作者，以及多家出版社出版的多种书籍的技术编辑，是 C++ 大会 CppCon 的活跃演讲者。

# 前 言

作为带有类的 C 的继承者，丹麦计算机科学家 Bjarne Stroustrup 于 1982 年发明了 C++。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 年的周期发布更新的版本，出现了 C++14、C++17 以及现在的 C++20。总之，2020 年发布了 C++20 之后，C++ 已经将近 40 年了，并且仍然很强大。在 2020 年的大多数编程语言排名中，C++ 都排在前四位。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++ 还可以用来解决几乎任何编程工作，如移动平台上的游戏、对性能要求严格的人工智能(AI)和机器学习(ML)软件、实时三维图形引擎、底层硬件驱动程序、整个操作系统等。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。尽管 C++ 语言已经风靡全球，但这种语言却难以完全掌握。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，你也仍可能不熟悉 C++ 的一些高级特性，或仍不具有使用这门语言的完整能力。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++，但有很丰富的 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(例如队列)以及有用的算法(例如排序和搜索)。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++20 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 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，他可能会认为所有源代码都要放在程序的 `main()` 函数中，还有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以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。

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的重复工作。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 5 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，这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。这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。这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。这部分还解释了 C++ 标准库，包括容器、迭代器、范围和算法。在这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++。这部分揭示 C++ 中神秘的部分，并描述如何使用这些更高级的特性。在这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使用的工程实践，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。

本书最后是四个附录。附录 A 列出在 C++技术面试中取得成功的指南，附录 B 总结 C++标准中的头文件，附录 C 简要介绍 UML(Unified Modeling Language，统一建模语言)，附录 D 是带注解的参考文献列表(附录 B~D 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 D 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](http://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/](http://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](https://github.com/isocpp/CppCoreGuidelines)

C++核心指南是由 C++语言发明人 Bjarne Stroustrup 亲自领导的协作工作。它们是许多组织多年讨论和设计的结果。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相对较高级别的问题，如接口、资源管理、内存管理和并发。

[github.com/Microsoft/GSL](https://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](https://isocpp.org/faq)

这是一个频繁被提问的 C++问题的庞大集合。

[stackoverflow.com](https://stackoverflow.com)

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++20 特性支持

本书包含 C++20 标准引入的新特性。在撰写本书时，还没有编译器可以完全支持 C++20 的所有新特性。某些新特性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特性，我相信不久就会有完全符合 C++20 标准的编译器可用。可以在 [en.cppreference.com/w/cpp/compiler\\_support](http://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++20 模块支持

在撰写本书时，还没有编译器可以完全支持 C++20 的模块。有些编译器提供了实验性支持，但仍然不完整。本书到处都使用模块。我们尽了最大努力确保编译器完全支持模块后，所有示例代码都可以编译，但由于我们无法编译和测试所有示例，因此可能会出现一些错误。当使用支持模块的编译器时，如果遇到任何代码示例的问题，请在 [www.wiley.com/go/proc++5e](http://www.wiley.com/go/proc++5e) 上仔细检查本书的勘误表，以查看是否存在已知问题。如果你的编译器还不支持模块，可以将模块化代码转换为非模块化代码，第 11 章中有简要说明。

## 示例：Microsoft Visual C++ 2019

首先需要创建一个项目。启动 Visual C++ 2019，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，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(从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++ 2019 尚未自动启用 C++20 特性。要启用 C++20 特性，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 | C/C++ | Language，根据使用的 Visual C++ 版本，将 C++ Language Standard 选项设置为 ISO C++20 Standard 或 Preview | Features from the Latest C++ Working Draft。仅当项目至少包含一个.cpp 文件时，才能访问这些选项。

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。

### 模块支持

在撰写本书期间，Visual C++ 2019 尚未完全支持模块。编写或使用自己的模块通常没有问题，但是，导入标准库头文件(如以下内容)还不能立即生效：

```
import <iostream>;
```

要使此类导入声明生效，目前需要向项目中添加一个单独的头文件，例如 HeaderUnits.h，其中包含要导入的每个标准库头文件的导入声明。例如：

```
// HeaderUnits.h
```

```
#pragma once
import <iostream>;
import <vector>;
import <optional>;
import <utility>;
// ...
```

接下来，在 Solution Explorer 窗口中右击 HeaderUnits.h 文件，然后单击 Properties。选择 Configuration Properties | General，设置 Item Type 为 C/C++ Compiler，然后单击 Apply 按钮。下一步，选择 Configuration Properties | C/C++ | Advanced，将 Compile As 设置为 Compile as C++ Header Unit(/exportHeader)，然后单击 OK 按钮。

现在重新编译你的项目，在 HeaderUnits.h 文件中具有相应导入声明的所有导入声明都应该可以正常编译。

如果你正在使用模块实现分区(见第 11 章)，也称为内部分区，那么右键单击包含此类实现分区的所有文件，单击 Properties，转到 Configuration Properties | C/C++ | Advanced，将 Compile As 设置为 Compile as C++ Module Internal Partition(/internalPartition)，然后单击 OK 按钮。

## 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：

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

`-std=c++2a` 用于告诉 GCC 启用 C++20 支持。当 GCC 完全兼容 C++20 后，这个选项将会改为 `-std=C++20`。

### 模块支持

在撰写本书期间，GCC 对模块仅有实验性的支持，通过一个特殊的版本(分支 `devel/c++-modules`)。当你在使用这个版本的 GCC 时，可以通过 `-fmodules-ts` 开启 module 支持，这个选项未来可能会改成 `-fmodules`。

但是，像下面这样对标准库头文件的导入声明还未被很好地支持：

```
import <iostream>;
```

如果遇到这种情况，将导入声明简单替换为相应的`#include` 指令即可：

```
#include <iostream>;
```

例如，第 1 章中的 AirlineTicket 示例使用了模块。将标准库头文件的导入替换为`#include` 指令后，可以通过更改包含代码的目录并运行以下命令来编译 AirlineTicket 示例：

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

当其通过编译后，你可以这样运行它：

```
./AirlineTicket
```

## std::format 支持

本书中的许多代码示例都使用了第 1 章介绍的 std::format()。在撰写本书时,还没有支持 std::format() 的编译器。但是,只要编译器还不支持 std::format(),就可以使用免费提供的{fmt}库作为替换:

- (1) 从 <https://fmt.dev/> 下载{fmt}库的最新版本并解压代码到你的计算机上。
- (2) 将 include/fmt 和 src 目录复制到你的项目目录中的 fmt 和 src 子目录,然后将 fmt/core.h、fmt/format.h、fmt/format.inl.h 和 src/format.cc 添加到项目中。
- (3) 将名为 format(无扩展名)的文件添加到项目的根目录,并向其中添加以下代码:

```
#pragma once
#define FMT_HEADER_ONLY
#include "fmt/format.h"
namespace std
{
    using fmt::format;
    using fmt::format_error;
    using fmt::formatter;
}
```

- (4) 最后,添加项目根目录(包含 format 文件的目录)作为项目的附加 include 目录。例如,在 Visual C++ 中,在 Solution Explorer 窗口中右击你的项目,然后单击 Properties,选择 Configuration Properties | C/C++ | General, 将\$(ProjectDir);添加到 Additional Include Directories 选项的前面。

### 注意:

当编译器支持了标准的 std::format()之后,不要忘记取消这些操作。

## 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,这对于学习过程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。

### 提示:

由于许多图书的书名都十分类似,因此按 ISBN 搜索是最简单的,本书英文版的 ISBN 是 978-1-119-69540-0。

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。

附录 B~D 和本书习题答案可扫描封底二维码下载。

# 目 录

## 第1部分 专业的 C++简介

<b>第1章 C++和标准库速成</b>	<b>3</b>
1.1 C++速成	3
1.1.1 小程序“Hello World”	4
1.1.2 名称空间	7
1.1.3 字面量	9
1.1.4 变量	9
1.1.5 运算符	12
1.1.6 枚举类型	14
1.1.7 结构体	16
1.1.8 条件语句	17
1.1.9 条件运算符	19
1.1.10 逻辑比较运算符	20
1.1.11 三向比较运算符	21
1.1.12 函数	22
1.1.13 属性	23
1.1.14 C 风格的数组	26
1.1.15 std::array	27
1.1.16 std::vector	28
1.1.17 std::pair	28
1.1.18 std::optional	29
1.1.19 结构化绑定	30
1.1.20 循环	30
1.1.21 初始化列表	31
1.1.22 C++中的字符串	32
1.1.23 作为面向对象语言的 C++	32
1.1.24 作用域解析	35
1.1.25 统一初始化	36
1.1.26 指针和动态内存	39
1.1.27 const 的用法	43
1.1.28 constexpr 关键字	45
1.1.29 consteval 关键字	46
1.1.30 引用	47
1.1.31 const_cast()	55

1.1.32 异常	56
1.1.33 类型别名	56
1.1.34 类型定义	57
1.1.35 类型推断	58
1.1.36 标准库	60
<b>1.2 第一个大型的 C++程序</b>	<b>61</b>
1.2.1 雇员记录系统	61
1.2.2 Employee 类	61
1.2.3 Database 类	64
1.2.4 用户界面	67
1.2.5 评估程序	69
1.3 本章小结	69
1.4 练习	69
<b>第2章 使用 string 和 string_view</b>	<b>71</b>
2.1 动态字符串	71
2.1.1 C 风格字符串	71
2.1.2 字符串字面量	73
2.1.3 C++ std::string 类	75
2.1.4 数值转换	78
2.1.5 std::string_view 类	81
2.1.6 非标准字符串	84
2.2 字符串格式化	84
2.2.1 格式说明符	85
2.2.2 格式说明符错误	87
2.2.3 支持自定义类型	87
2.3 本章小结	90
2.4 练习	90
<b>第3章 编码风格</b>	<b>91</b>
3.1 良好外观的重要性	91
3.1.1 事先考虑	91
3.1.2 良好风格的元素	92
3.2 为代码编写文档	92
3.2.1 使用注释的原因	92
3.2.2 注释的风格	96

3.3	分解	99
3.3.1	通过重构分解	100
3.3.2	通过设计分解	101
3.3.3	本书中的分解	101
3.4	命名	101
3.4.1	选择恰当的名称	101
3.4.2	命名约定	102
3.5	使用具有风格的语言特性	103
3.5.1	使用常量	104
3.5.2	使用引用代替指针	104
3.5.3	使用自定义异常	104
3.6	格式	105
3.6.1	关于大括号对齐的争论	105
3.6.2	关于空格和圆括号的争论	106
3.6.3	空格、制表符、换行符	106
3.7	风格的挑战	107
3.8	本章小结	107
3.9	练习	107

## 第 II 部分 专业的 C++ 软件设计

第 4 章	设计专业的 C++ 程序	113
4.1	程序设计概述	113
4.2	程序设计的重要性	114
4.3	C++ 设计	116
4.4	C++ 设计的两个原则	116
4.4.1	抽象	116
4.4.2	重用	118
4.5	重用现有代码	119
4.5.1	关于术语的说明	119
4.5.2	决定是否重用代码	120
4.5.3	重用代码的指导原则	121
4.6	设计一个国际象棋程序	127
4.6.1	需求	127
4.6.2	设计步骤	127
4.7	本章小结	132
4.8	练习	133

第 5 章	面向对象设计	135
5.1	过程化的思考方式	135
5.2	面向对象思想	136
5.2.1	类	136
5.2.2	组件	136

5.2.3	属性	136
5.2.4	行为	137
5.2.5	综合考虑	137
5.3	生活在类的世界里	138
5.3.1	过度使用类	138
5.3.2	过于通用的类	139
5.4	类之间的关系	139
5.4.1	“有一个”关系	139
5.4.2	“是一个”关系(继承)	140
5.4.3	“有一个”与“是一个”的区别	142
5.4.4	not-a 关系	144
5.4.5	层次结构	145
5.4.6	多重继承	146
5.4.7	混入类	147
5.5	本章小结	147
5.6	练习	148

## 第 6 章 设计可重用代码

6.1	重用哲学	149
6.2	如何设计可重用代码	150
6.2.1	使用抽象	150
6.2.2	构建理想的重用代码	151
6.2.3	设计有用的接口	157
6.2.4	设计成功的抽象	162
6.2.5	SOLID 原则	162
6.3	本章小结	163
6.4	练习	163

## 第 III 部分 C++ 编码方法

第 7 章	内存管理	167
7.1	使用动态内存	167
7.1.1	如何描绘内存	168
7.1.2	分配和释放	169
7.1.3	数组	170
7.1.4	使用指针	177
7.2	数组-指针的对偶性	178
7.2.1	数组就是指针	178
7.2.2	并非所有指针都是数组	180
7.3	底层内存操作	180
7.3.1	指针运算	180
7.3.2	自定义内存管理	181

7.3.3 垃圾回收.....	181	9.3 与方法有关的更多内容.....	246
7.3.4 对象池.....	182	9.3.1 static 方法.....	246
<b>7.4 常见的内存陷阱.....</b>	<b>182</b>	9.3.2 const 方法.....	247
7.4.1 数据缓冲区分配不足以及 内存访问越界.....	182	9.3.3 方法重载.....	248
7.4.2 内存泄漏.....	183	9.3.4 内联方法.....	251
7.4.3 双重释放和无效指针.....	186	9.3.5 默认参数.....	252
<b>7.5 智能指针.....</b>	<b>186</b>	<b>9.4 不同的数据成员类型.....</b>	<b>252</b>
7.5.1 unique_ptr.....	187	9.4.1 静态数据成员.....	253
7.5.2 shared_ptr.....	190	9.4.2 const static 数据成员.....	254
7.5.3 weak_ptr.....	193	9.4.3 引用数据成员.....	255
7.5.4 向函数传递参数.....	193	<b>9.5 嵌套类.....</b>	<b>256</b>
7.5.5 从函数中返回.....	194	<b>9.6 类内的枚举类型.....</b>	<b>257</b>
7.5.6 enable_shared_from_this.....	194	<b>9.7 运算符重载.....</b>	<b>258</b>
7.5.7 过时的、移除的 auto_ptr.....	195	9.7.1 示例：为 SpreadsheetCell 实现加法.....	258
<b>7.6 本章小结.....</b>	<b>195</b>	9.7.2 重载算术运算符.....	261
<b>7.7 练习.....</b>	<b>195</b>	9.7.3 重载比较运算符.....	262
<b>第 8 章 类和对象.....</b>	<b>197</b>	9.7.4 创建具有运算符重载的类型.....	266
8.1 电子表格示例介绍.....	197	<b>9.8 创建稳定的接口.....</b>	<b>266</b>
8.2 编写类.....	198	<b>9.9 本章小结.....</b>	<b>270</b>
8.2.1 类定义.....	198	<b>9.10 练习.....</b>	<b>270</b>
8.2.2 定义方法.....	200	<b>第 10 章 揭秘继承技术.....</b>	<b>271</b>
8.2.3 使用对象.....	203	10.1 使用继承构建类.....	271
8.3 对象的生命周期.....	205	10.1.1 扩展类.....	272
8.3.1 创建对象.....	205	10.1.2 重写方法.....	275
8.3.2 销毁对象.....	219	10.2 使用继承重用代码.....	282
8.3.3 对象赋值.....	220	10.2.1 WeatherPrediction 类.....	282
8.3.4 编译器生成的拷贝构造函数和 拷贝赋值运算符.....	223	10.2.2 在派生类中添加功能.....	283
8.3.5 复制和赋值的区别.....	223	10.2.3 在派生类中替换功能.....	284
8.4 本章小结.....	224	10.3 利用父类.....	285
8.5 练习.....	225	10.3.1 父类构造函数.....	285
<b>第 9 章 精通类和对象.....</b>	<b>227</b>	10.3.2 父类的析构函数.....	286
9.1 友元.....	227	10.3.3 使用父类方法.....	287
9.2 对象中的动态内存分配.....	228	10.3.4 向上转型和向下转型.....	289
9.2.1 Spreadsheet 类.....	228	10.4 继承与多态性.....	290
9.2.2 使用析构函数释放内存.....	231	10.4.1 回到电子表格.....	290
9.2.3 处理复制和赋值.....	231	10.4.2 设计多态性的电子表格 单元格.....	291
9.2.4 使用移动语义处理移动.....	237	10.4.3 SpreadsheetCell 基类.....	291
9.2.5 零规则.....	246	10.4.4 独立的派生类.....	293
		10.4.5 利用多态性.....	294

10.4.6 考虑将来.....	295	11.4.4 非局部变量的初始化顺序.....	335
10.5 多重继承.....	296	11.4.5 非局部变量的销毁顺序.....	335
10.5.1 从多个类继承.....	296	11.5 C 的实用工具.....	335
10.5.2 名称冲突和歧义基类.....	297	11.5.1 变长参数列表.....	336
10.6 有趣而晦涩的继承问题.....	300	11.5.2 预处理器宏.....	337
10.6.1 修改重写方法的返回类型.....	300	11.6 本章小结.....	338
10.6.2 派生类中添加虚基类方法的 重载.....	301	11.7 练习.....	338
10.6.3 继承的构造函数.....	302		
10.6.4 重写方法时的特殊情况.....	306		
10.6.5 派生类中的复制构造函数和 赋值运算符.....	312		
10.6.6 运行期类型工具.....	313		
10.6.7 非 public 继承.....	314		
10.6.8 虚基类.....	315		
10.7 类型转换.....	316		
10.7.1 static_cast().....	316		
10.7.2 reinterpret_cast().....	317		
10.7.3 std::bit_cast().....	318		
10.7.4 dynamic_cast().....	318		
10.7.5 类型转换小结.....	319		
10.8 本章小结.....	319		
10.9 练习.....	320		
<b>第 11 章 零碎的工作.....</b>	<b>321</b>		
11.1 模块.....	321		
11.1.1 模块接口文件.....	322		
11.1.2 模块实现文件.....	324		
11.1.3 从实现中分离接口.....	325		
11.1.4 可见性和可访问性.....	326		
11.1.5 子模块.....	326		
11.1.6 模块划分.....	327		
11.1.7 头文件单元.....	329		
11.2 头文件.....	330		
11.2.1 重复定义.....	330		
11.2.2 循环依赖.....	330		
11.2.3 查询头文件是否存在.....	331		
11.3 核心语言特性的特性测试宏.....	331		
11.4 STATIC 关键字.....	332		
11.4.1 静态数据成员和方法.....	332		
11.4.2 静态链接.....	332		
11.4.3 函数中的静态变量.....	334		
<b>第 12 章 利用模板编写泛型代码.....</b>	<b>341</b>		
12.1 模板概述.....	341		
12.2 类模板.....	342		
12.2.1 编写类模板.....	342		
12.2.2 编译器处理模板的原理.....	349		
12.2.3 将模板代码分布到多个 文件中.....	350		
12.2.4 模板参数.....	351		
12.2.5 方法模板.....	355		
12.2.6 类模板的特化.....	359		
12.2.7 从类模板派生.....	361		
12.2.8 继承还是特化.....	362		
12.2.9 模板别名.....	362		
12.3 函数模板.....	363		
12.3.1 函数模板的重载.....	364		
12.3.2 类模板的友元函数模板.....	365		
12.3.3 对模板参数推导的更多介绍.....	366		
12.3.4 函数模板的返回类型.....	367		
12.4 简化函数模板的语法.....	368		
12.5 变量模板.....	369		
12.6 概念.....	369		
12.6.1 语法.....	369		
12.6.2 约束表达式.....	370		
12.6.3 预定义的标准概念.....	372		
12.6.4 类型约束的 auto.....	372		
12.6.5 类型约束和函数模板.....	373		
12.6.6 类型约束和类模板.....	375		
12.6.7 类型约束和类方法.....	375		
12.6.8 类型约束和模板特化.....	376		
12.7 本章小结.....	376		
12.8 练习.....	377		
<b>第 13 章 C++ I/O 揭秘.....</b>	<b>379</b>		
13.1 使用流.....	379		

13.1.1 流的含义	380	14.3.6 异常的源码位置	422
13.1.2 流的来源和目的地	381	14.3.7 嵌套异常	423
13.1.3 流式输出	381	14.4 重新抛出异常	425
13.1.4 流式输入	386	14.5 堆栈的释放与清理	426
13.1.5 对象的输入输出	392	14.5.1 使用智能指针	427
13.1.6 自定义的操作算子	393	14.5.2 捕获、清理并重新抛出	428
13.2 字符串流	393	14.6 常见的错误处理问题	428
13.3 文件流	394	14.6.1 内存分配错误	428
13.3.1 文本模式与二进制模式	395	14.6.2 构造函数中的错误	430
13.3.2 通过 seek() 和 tell() 在文件中 转移	395	14.6.3 构造函数的function-try-blocks	432
13.3.3 将流链接在一起	397	14.6.4 析构函数中的错误	435
13.4 双向 I/O	398	14.7 本章小结	435
13.5 文件系统支持库	399	14.8 练习	435
13.5.1 路径	399		
13.5.2 目录条目	401		
13.5.3 辅助函数	401		
13.5.4 目录遍历	401		
13.6 本章小结	402		
13.7 练习	403		
<b>第 14 章 错误处理</b>	<b>405</b>		
14.1 错误与异常	405		
14.1.1 异常的含义	405		
14.1.2 C++ 中异常的优点	406		
14.1.3 建议	407		
14.2 异常机制	407		
14.2.1 抛出和捕获异常	408		
14.2.2 异常类型	410		
14.2.3 按 const 引用捕获异常对象	411		
14.2.4 抛出并捕获多个异常	411		
14.2.5 未捕获的异常	414		
14.2.6 noexcept 说明符	415		
14.2.7 noexcept(expression) 说明符	415		
14.2.8 noexcept(expression) 运算符	415		
14.2.9 抛出列表	416		
14.3 异常与多态性	416		
14.3.1 标准异常层次结构	416		
14.3.2 在类层次结构中捕获异常	418		
14.3.3 编写自己的异常类	419		
14.3.4 源码位置	421		
14.3.5 日志记录的源码位置	422		
		14.3.6 异常的源码位置	422
		14.3.7 嵌套异常	423
		14.4 重新抛出异常	425
		14.5 堆栈的释放与清理	426
		14.5.1 使用智能指针	427
		14.5.2 捕获、清理并重新抛出	428
		14.6 常见的错误处理问题	428
		14.6.1 内存分配错误	428
		14.6.2 构造函数中的错误	430
		14.6.3 构造函数的function-try-blocks	432
		14.6.4 析构函数中的错误	435
		14.7 本章小结	435
		14.8 练习	435
<b>第 15 章 C++ 运算符重载</b>	<b>437</b>		
15.1 运算符重载概述	437		
15.1.1 重载运算符的原因	438		
15.1.2 运算符重载的限制	438		
15.1.3 运算符重载的选择	438		
15.1.4 不应重载的运算符	440		
15.1.5 可重载运算符小结	440		
15.1.6 右值引用	443		
15.1.7 优先级和结合性	444		
15.1.8 关系运算符	444		
15.2 重载算术运算符	445		
15.2.1 重载一元负号和一元正号 运算符	445		
15.2.2 重载递增和递减运算符	446		
15.3 重载按位运算符和二元逻辑 运算符	446		
15.4 重载插入运算符和提取运算符	447		
15.5 重载下标运算符	448		
15.5.1 通过 operator[] 提供只读访问	451		
15.5.2 非整数数组索引	452		
15.6 重载函数调用运算符	452		
15.7 重载解除引用运算符	453		
15.7.1 实现 operator*	454		
15.7.2 实现 operator->	455		
15.7.3 operator.* 和 operator ->* 的 含义	455		
15.8 编写转换运算符	456		
15.8.1 auto 运算符	456		

15.8.2 使用显式转换运算符解决多义性问题	457	16.2.16 标准整数类型	475
15.8.3 用于布尔表达式的转换	457	16.2.17 标准库特性测试宏	475
<b>15.9 重载内存分配和内存释放运算符</b>	<b>459</b>	16.2.18 <version>	476
15.9.1 new 和 delete 的工作原理	459	16.2.19 源位置	476
15.9.2 重载 operator new 和 operator delete	461	16.2.20 容器	476
15.9.3 显式地删除/默认化 operator new 和 operator delete	463	16.2.21 算法	482
15.9.4 重载带有额外参数的 operator new 和 operator delete	463	16.2.22 范围库	488
15.9.5 重载带有内存大小参数的 operator delete	464	16.2.23 标准库中还缺什么	488
15.9.6 重载用户定义的字面量运算符	464	<b>16.3 本章小结</b>	489
15.9.7 cooked 模式字面量运算符	465	<b>16.4 练习</b>	489
15.9.8 raw 模式字面量运算符	465		
15.9.9 标准用户定义的字面量	466		
<b>15.10 本章小结</b>	<b>466</b>		
<b>15.11 练习</b>	<b>466</b>		
<b>第 16 章 C++ 标准库概述</b>	<b>469</b>		
16.1 编码原则	470		
16.1.1 使用模板	470		
16.1.2 使用运算符重载	470		
<b>16.2 C++ 标准库概述</b>	<b>470</b>		
16.2.1 字符串	470		
16.2.2 正则表达式	471		
16.2.3 I/O 流	471		
16.2.4 智能指针	471		
16.2.5 异常	471		
16.2.6 数学工具	472		
16.2.7 时间和日期工具	473		
16.2.8 随机数	473		
16.2.9 初始化列表	474		
16.2.10 Pair 和 Tuple	474		
16.2.11 词汇类型	474		
16.2.12 函数对象	474		
16.2.13 文件系统	474		
16.2.14 多线程	475		
16.2.15 类型萃取	475		
<b>第 17 章 理解迭代器与范围库</b>	<b>491</b>		
17.1 迭代器	491		
17.1.1 获取容器的迭代器	494		
17.1.2 迭代器萃取	495		
17.1.3 示例	495		
<b>17.2 流迭代器</b>	<b>496</b>		
17.2.1 输出流迭代器	497		
17.2.2 输入流迭代器	497		
<b>17.3 迭代器适配器</b>	<b>498</b>		
17.3.1 插入迭代器	498		
17.3.2 逆向迭代器	499		
17.3.3 移动迭代器	500		
<b>17.4 范围</b>	<b>502</b>		
17.4.1 基于范围的算法	502		
17.4.2 视图	504		
17.4.3 范围工厂	508		
<b>17.5 本章小结</b>	<b>509</b>		
<b>17.6 练习</b>	<b>509</b>		
<b>第 18 章 标准库容器</b>	<b>511</b>		
<b>18.1 容器概述</b>	<b>511</b>		
18.1.1 对元素的要求	512		
18.1.2 异常和错误检查	513		
<b>18.2 顺序容器</b>	<b>514</b>		
18.2.1 vector	514		
18.2.2 vector<bool>特化	531		
18.2.3 deque	532		
18.2.4 list	532		
18.2.5 forward_list	535		
18.2.6 array	537		
18.2.7 span	538		

18.3 容器适配器.....	540	19.5 调用.....	590
18.3.1 queue.....	540	19.6 本章小结.....	590
18.3.2 priority_queue.....	542	19.7 练习.....	590
18.3.3 stack.....	545		
18.4 有序关联容器.....	545	<b>第 20 章 掌握标准库算法 .....</b>	<b>593</b>
18.4.1 pair 工具类.....	545	20.1 算法概述.....	593
18.4.2 map.....	546	20.1.1 find()和 find_if()算法.....	594
18.4.3 multimap .....	554	20.1.2 accumulate()算法.....	596
18.4.4 set .....	556	20.1.3 在算法中使用移动语义.....	597
18.4.5 multiset.....	558	20.1.4 算法回调.....	597
18.5 无序关联容器/哈希表 .....	558	20.2 算法详解.....	598
18.5.1 哈希函数.....	559	20.2.1 非修改序列算法 .....	598
18.5.2 unordered_map.....	560	20.2.2 修改序列算法 .....	603
18.5.3 unordered_multimap .....	563	20.2.3 操作算法 .....	611
18.5.4 unordered_set/ unordered_multiset .....	564	20.2.4 分区算法 .....	613
18.6 其他容器 .....	564	20.2.5 排序算法 .....	614
18.6.1 标准 C 风格数组 .....	564	20.2.6 二分查找算法 .....	615
18.6.2 string .....	565	20.2.7 集合算法 .....	616
18.6.3 流 .....	566	20.2.8 最小/最大算法 .....	618
18.6.4 bitset.....	566	20.2.9 并行算法 .....	619
18.7 本章小结 .....	570	20.2.10 约束算法 .....	620
18.8 练习 .....	570	20.2.11 数值处理算法 .....	621
<b>第 19 章 函数指针, 函数对象, lambda 表达式 .....</b>	<b>571</b>	20.3 本章小结 .....	622
19.1 函数指针 .....	571	20.4 练习 .....	622
19.2 指向方法(和数据成员)的指针.....	573		
19.3 函数对象 .....	576	<b>第 21 章 字符串的本地化与正则表达式 .....</b>	<b>625</b>
19.3.1 编写第一个函数对象.....	576	21.1 本地化 .....	625
19.3.2 标准库中的函数对象.....	576	21.1.1 宽字符 .....	625
19.4 lambda 表达式 .....	582	21.1.2 本地化字符串字面量 .....	626
19.4.1 语法 .....	583	21.1.3 非西方字符集 .....	626
19.4.2 lambda 表达式作为参数 .....	587	21.1.4 locale 和 facet .....	628
19.4.3 泛型 lambda 表达式 .....	587	21.1.5 转换 .....	631
19.4.4 lambda 捕获表达式 .....	587	21.2 正则表达式 .....	632
19.4.5 模板化 lambda 表达式 .....	588	21.2.1 ECMAScript 语法 .....	632
19.4.6 lambda 表达式作为返回类型 .....	589	21.2.2 regex 库 .....	637
19.4.7 未计算上下文中的 lambda 表达式 .....	589	21.2.3 regex_match() .....	638
19.4.8 默认构造、拷贝和赋值 .....	589	21.2.4 regex_search() .....	640

<b>第 22 章</b>	<b>日期和时间工具</b>	647	25.2.5	添加分配器支持	712
22.1	编译期有理数	647	25.2.6	改善 graph_node	716
22.2	持续时间	649	25.2.7	附加的标准库类似功能	717
22.3	时钟	653	25.2.8	进一步改善	719
22.4	时间点	655	25.2.9	其他容器类型	719
22.5	日期	656	25.3	本章小结	720
22.6	时区	658	25.4	练习	720
22.7	本章小结	659			
22.8	练习	659			
<b>第 23 章</b>	<b>随机数工具</b>	661	<b>第 26 章</b>	<b>高级模板</b>	721
23.1	C 风格随机数生成器	661	26.1	深入了解模板参数	721
23.1.1	随机数引擎	662	26.1.1	深入了解模板类型参数	721
23.1.2	随机数引擎适配器	663	26.1.2	template template 参数介绍	724
23.1.3	预定义的随机数引擎和 引擎适配器	664	26.1.3	深入了解非类型模板参数	725
23.1.4	生成随机数	664	26.2	类模板部分特例化	727
23.1.5	随机数分布	666	26.3	通过重载模拟函数部分特例化	730
23.2	本章小结	668	26.4	模板递归	731
23.3	练习	669	26.4.1	N 维网格：初次尝试	731
<b>第 24 章</b>	<b>其他库工具</b>	671	26.4.2	真正的 N 维网格	732
24.1	variant	671	26.5	可变参数模板	734
24.2	any	673	26.5.1	类型安全的变长参数列表	734
24.3	元组	674	26.5.2	可变数目的混入类	736
24.3.1	分解元组	676	26.5.3	折叠表达式	737
24.3.2	串联	677	26.6	模板元编程	739
24.3.3	比较	677	26.6.1	编译时阶乘	739
24.3.4	make_from_tuple()	678	26.6.2	循环展开	740
24.3.5	apply()	678	26.6.3	打印元组	741
24.4	本章小结	678	26.6.4	类型 trait	744
24.5	练习	678	26.6.5	模板元编程结论	752
<b>第 IV 部分 掌握 C++ 的高级特性</b>			26.7	本章小结	752
26.8	练习	752			
<b>第 25 章</b>	<b>自定义和扩展标准库</b>	683	<b>第 27 章</b>	<b>C++ 多线程编程</b>	753
25.1	分配器	683	27.1	多线程编程概述	754
25.2	扩展标准库	684	27.1.1	争用条件	755
25.2.1	扩展标准库的原因	685	27.1.2	撕裂	756
25.2.2	编写标准库算法	685	27.1.3	死锁	756
25.2.3	编写标准库容器	686	27.1.4	伪共享	757
25.2.4	将 directed_graph 实现为 标准库容器	696	27.2	线程	757
			27.2.1	通过函数指针创建线程	758
			27.2.2	通过函数对象创建线程	759
			27.2.3	通过 lambda 创建线程	760
			27.2.4	通过成员函数创建线程	760

27.2.5 线程本地存储 .....	761	28.2.3 螺旋类模型 .....	802
27.2.6 取消线程 .....	761	28.2.4 敏捷 .....	804
27.2.7 自动 join 线程 .....	761	28.3 软件工程方法论 .....	805
27.2.8 从线程获得结果 .....	762	28.3.1 UP .....	805
27.2.9 复制和重新抛出异常 .....	762	28.3.2 RUP .....	806
<b>27.3 原子操作库 .....</b>	<b>764</b>	28.3.3 Scrum .....	806
27.3.1 原子操作 .....	766	28.3.4 极限编程 .....	808
27.3.2 原子智能指针 .....	767	28.3.5 软件分流 .....	812
27.3.3 原子引用 .....	767	<b>28.4 构建自己的过程和方法 .....</b>	<b>812</b>
27.3.4 使用原子类型 .....	767	28.4.1 对新思想采取开放态度 .....	812
27.3.5 等待原子变量 .....	769	28.4.2 提出新想法 .....	812
<b>27.4 互斥 .....</b>	<b>770</b>	28.4.3 知道什么行得通、什么 行不通 .....	812
27.4.1 互斥体类 .....	770	28.4.4 不要逃避 .....	813
27.4.2 锁 .....	772	<b>28.5 源代码控制 .....</b>	<b>813</b>
27.4.3 std::call_once .....	774	<b>28.6 本章小结 .....</b>	<b>814</b>
27.4.4 互斥体对象的用法示例 .....	776	<b>28.7 练习 .....</b>	<b>814</b>
<b>27.5 条件变量 .....</b>	<b>779</b>	<b>第 29 章 编写高效的 C++ 程序 .....</b>	<b>817</b>
27.5.1 虚假唤醒 .....	780	<b>29.1 性能和效率概述 .....</b>	<b>817</b>
27.5.2 使用条件变量 .....	780	29.1.1 提升效率的两种方式 .....	818
<b>27.6 latch .....</b>	<b>781</b>	29.1.2 两种程序 .....	818
<b>27.7 barrier .....</b>	<b>782</b>	29.1.3 C++ 是不是低效的语言 .....	818
<b>27.8 semaphore .....</b>	<b>782</b>	<b>29.2 语言层次的效率 .....</b>	<b>818</b>
<b>27.9 future .....</b>	<b>783</b>	29.2.1 高效地操纵对象 .....	819
27.9.1 std::promise 和 std::future .....	784	29.2.2 预分配内存 .....	823
27.9.2 std::packaged_task .....	784	29.2.3 使用内联方法和函数 .....	823
27.9.3 std::async .....	785	<b>29.3 设计层次的效率 .....</b>	<b>823</b>
27.9.4 异常处理 .....	786	29.3.1 尽可能多地缓存 .....	823
27.9.5 std::shared_future .....	786	29.3.2 使用对象池 .....	824
<b>27.10 示例：多线程的 Logger 类 .....</b>	<b>787</b>	<b>29.4 剖析 .....</b>	<b>829</b>
<b>27.11 线程池 .....</b>	<b>791</b>	29.4.1 使用 gprof 的剖析示例 .....	829
<b>27.12 协程 .....</b>	<b>792</b>	29.4.2 使用 Visual C++ 2019 的 剖析示例 .....	836
<b>27.13 线程设计和最佳实践 .....</b>	<b>793</b>	<b>29.5 本章小结 .....</b>	<b>838</b>
<b>27.14 本章小结 .....</b>	<b>794</b>	<b>29.6 练习 .....</b>	<b>838</b>
<b>27.15 练习 .....</b>	<b>794</b>	<b>第 30 章 熟练掌握测试技术 .....</b>	<b>841</b>
<b>第 V 部分 C++ 软件工程</b>		<b>30.1 质量控制 .....</b>	<b>841</b>
<b>第 28 章 充分利用软件工程方法 .....</b>	<b>799</b>	30.1.1 谁负责测试 .....	842
28.1 过程的必要性 .....	799	30.1.2 bug 的生命周期 .....	842
28.2 软件生命周期模型 .....	800	30.1.3 bug 跟踪工具 .....	843
28.2.1 瀑布模型 .....	800		
28.2.2 生鱼片模型 .....	802		

30.2	单元测试.....	844	32.1.5	抛出和捕捉异常.....	893
30.2.1	单元测试方法.....	844	32.1.6	写入文件.....	894
30.2.2	单元测试过程.....	845	32.1.7	读取文件.....	894
30.2.3	实际中的单元测试.....	848	32.1.8	写入类模板.....	895
30.3	模糊测试.....	855	32.1.9	约束模板参数.....	895
30.4	高级测试.....	855	32.2	始终存在更好的方法.....	896
30.4.1	集成测试.....	855	32.2.1	RAII.....	896
30.4.2	系统测试.....	856	32.2.2	双分派.....	898
30.4.3	回归测试.....	857	32.2.3	混入类.....	902
30.5	用于成功测试的建议.....	857	32.3	面向对象的框架.....	904
30.6	本章小结.....	858	32.3.1	使用框架.....	904
30.7	练习.....	858	32.3.2	MVC 范型.....	905
<b>第 31 章</b>	<b>熟练掌握调试技术.....</b>	<b>859</b>	32.4	本章小结.....	906
31.1	调试的基本定律.....	859	32.5	练习.....	906
31.2	bug 分类学.....	860	<b>第 33 章</b>	<b>应用设计模式.....</b>	<b>907</b>
31.3	避免 bug.....	860	33.1	依赖注入.....	908
31.4	为 bug 做好规划.....	861	33.1.1	示例：日志机制.....	908
31.4.1	错误日志.....	861	33.1.2	依赖注入 logger 的实现.....	908
31.4.2	调试跟踪.....	862	33.1.3	使用依赖注入.....	909
31.4.3	断言.....	869	33.2	抽象工厂模式.....	910
31.4.4	崩溃转储.....	870	33.2.1	示例：模拟汽车工厂.....	910
31.5	调试技术.....	870	33.2.2	实现抽象工厂.....	911
31.5.1	重现 bug.....	870	33.2.3	使用抽象工厂.....	912
31.5.2	调试可重复的 bug.....	871	33.3	工厂方法模式.....	913
31.5.3	调试不可重现的 bug.....	871	33.3.1	示例：模拟第二个汽车工厂.....	913
31.5.4	调试退化.....	872	33.3.2	实现工厂.....	914
31.5.5	调试内存问题.....	872	33.3.3	使用工厂.....	915
31.5.6	调试多线程程序.....	876	33.3.4	工厂的其他类型.....	917
31.5.7	调试示例：文章引用.....	876	33.3.5	工厂的其他用法.....	917
31.5.8	从 ArticleCitations 示例中 总结出的教训.....	887	33.4	适配器模式.....	918
31.6	本章小结.....	887	33.4.1	示例：适配 Logger 类.....	918
31.7	练习.....	887	33.4.2	实现适配器.....	919
<b>第 32 章</b>	<b>使用设计技术和框架.....</b>	<b>889</b>	33.4.3	使用适配器.....	920
32.1	容易忘记的语法.....	890	33.5	代理模式.....	920
32.1.1	编写类.....	890	33.5.1	示例：隐藏网络连接问题.....	920
32.1.2	派生类.....	891	33.5.2	实现代理.....	921
32.1.3	编写 lambda 表达式.....	892	33.5.3	使用代理.....	922
32.1.4	使用“复制和交换”惯用 语法.....	892	33.6	迭代器模式.....	922

33.7.3 使用观察者 .....	924	34.2.2 改变范型 .....	941
33.8 装饰器模式 .....	925	34.2.3 链接 C 代码 .....	944
33.8.1 示例：在网页中定义样式 .....	926	34.2.4 从 C# 调用 C++ 代码 .....	946
33.8.2 装饰器的实现 .....	926	34.2.5 C++/CLI 在 C++ 中使用 C# 代码 和在 C# 中使用 C++ 代码 .....	947
33.8.3 使用装饰器 .....	927	34.2.6 在 Java 中使用 JNI 调用 C++ 代码 .....	948
33.9 责任链模式 .....	928	34.2.7 从 C++ 代码调用脚本 .....	950
33.9.1 示例：事件处理 .....	928	34.2.8 从脚本调用 C++ 代码 .....	950
33.9.2 实现责任链 .....	928	34.2.9 从 C++ 调用汇编代码 .....	952
33.9.3 使用责任链 .....	929	34.3 本章小结 .....	953
33.10 单例模式 .....	930	34.4 练习 .....	953
33.10.1 日志记录机制 .....	931		
33.10.2 实现单例 .....	931		
33.10.3 使用单例 .....	933		
33.11 本章小结 .....	933		
33.12 练习 .....	933		
<b>第 34 章 开发跨平台和跨语言的应用 程序 .....</b>	<b>935</b>	<b>在线资源(扫描封底二维码下载)</b>	
34.1 跨平台开发 .....	935	附录 B 标准库头文件 .....	977
34.1.1 架构问题 .....	935	附录 C UML 简介 .....	983
34.1.2 实现问题 .....	938	附录 D 带注解的参考文献 .....	989
34.1.3 平台专用功能 .....	940		
34.2 跨语言开发 .....	940		
34.2.1 混用 C 和 C++ .....	941		

**第 VI 部分 附录**

附录 A C++面试 .....	957
<hr/>	
在线资源(扫描封底二维码下载)	
附录 B 标准库头文件 .....	977
附录 C UML 简介 .....	983
附录 D 带注解的参考文献 .....	989

# 第I部分

## 专业的 C++ 简介

---

- ▶ 第 1 章 C++ 和标准库速成
- ▶ 第 2 章 使用 string 和 string\_view
- ▶ 第 3 章 编码风格



# 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)最重要的部分以及语法
- 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、异常以及类型别名
- 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。此外还忽略了与 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那部分 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,C++ 语言之父 Bjarne Stroustrup 撰写的 *The C++ Programming Language*,

*Forth Edition*(Addison-Wesley Professional, 2013)共有 1368 页, 而 Kernighan 和 Ritchie 撰写的 *The C Programming Language, Second Edition*(Prentice Hall, 1988)只有 274 页。因此, 如果你是一位 C 程序员, 请关注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:

```
// helloworld.cpp
import <iostream>

int main()
{
    std::cout << "Hello, World!" << std::endl;
    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
- 输入输出流

下面简要描述这些概念(如果你的编译器还不支持 C++20 的模块, 请使用包含头文件的方式替代模块)。

#### 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。在前面以及下面的示例中, 用两条斜杠表明这一行中在它后面的内容都是注释。

```
// helloworld.cpp
```

使用多行注释也可以实现这个行为(也就是说, 二者没什么不同)。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“编码风格”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。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 <iostream>;
```

如果此程序没有导入该模块, 它就无法实现输出文字这项唯一的功能。

由于本书是关于 C++20 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模块中。你自定义的类型和函数也可以通过模块提供,你将在本书中学习如何做到这一点。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,我们将在下一节讨论。

### 3. 预处理指令

如果你的编译器还不支持 C++20 的模块,需要编写如下的预处理指令而不是像 import<iostream>这样的 import 声明。

```
#include <iostream>
```

简单来说,生成一个 C++ 程序共有 3 个步骤。首先,代码经过预处理器,预处理器会识别代码中的元信息。其次,代码被编译或转换为计算机可识别的目标文件。最后,独立的目标文件被连接在一起生成一个应用程序。

预处理指令以#字符开始,前面示例中的#include <iostream>就是如此。在此例中,include 指令告诉预处理器:提取<iostream>头文件中的所有内容并提供给当前文件。<iostream>头文件声明了 C++ 提供的输入输出机制。

头文件最常见的用途是声明在其他地方定义的函数。函数声明会通知编译器如何调用这个函数,并声明函数中参数的个数和类型,以及函数的返回类型。函数定义包含了这个函数的实际代码。在 C++20 引入模块之前,声明通常放在扩展名为.h 的文件中,称为头文件,其定义通常包含在扩展名为.cpp 的文件中,称为源文件。有了模块,我们不再需要将声明与定义分离,但是之后你将会看到,这样的写法依然是可行的。

#### 注意:

在 C 中,标准库头文件的名称通常以.h 结尾,如<stdio.h>,不使用名称空间。

在 C++ 中,标准库头文件的名称省略了.h 后缀,如<iostream>;所有文件都在 std 名称空间和 std 的子名称空间中定义。

C 中的标准库头文件在 C++ 中依然存在,但使用以下两个版本。

- 不使用.h 后缀,改用前缀 c;这是推荐使用的版本,这些头文件将所有内容放在 std 名称空间中(例如<cstdio>)。
- 使用.h 后缀,这是旧版本。这些版本不使用名称空间(例如<stdio.h>)。

注意,我们不能保证这些 C 标准库头文件是可以通过 import 声明导入的。为安全起见,使用 #include <cxyz> 而不是 import <cxyz>。

表 1-1 给出了最常用的预处理指令。

表 1-1 最常用的预处理指令

预处理指令	功能	常见用法
#include [file]	将指定的文件插入代码中指令所在的位置	几乎总是用来包含头文件,使代码可使用在其他位置定义的功能
#define [id] [value]	每个指定的标识符都被替换为指定的值	在 C 中,常用来定义常数值或宏。C++ 提供了常数和大多数宏类型的更好机制。宏的使用具有风险,因此在 C++ 中使用它们要谨慎,更多内容详见第 11 章“零碎的工作”

(续表)

预处理指令	功能	常见用法
#ifdef [id] #endif	ifdef("if defined")块或 ifndef("if not defined")块中的代码被有条件地包含或舍弃, 这取决于是否使用#define 定义了特定的标识符	经常用来防止循环包含。每个头文件都以#ifndef 开头, 以保证某个标识符还未被定义, 然后用一条#define 指令定义该标识符。头文件以#endif 结束, 这样这个头文件就不会被多次包含。参见该表之后列举的示例
#pragma [xyz]	xyz 因编译器而异。如果在预处理期间执行到这一指令, 通常会显示一条警告或错误信息	参见该表之后列举的示例

下面是使用预处理指令避免重复包含的一个示例:

```
#ifndef MYHEADER_H
#define MYHEADER_H
// ... the contents of this header file
#endif
```

如果编译器支持#pragma once 指令(大多数现代编译器都支持), 可采用以下方法重写上面的代码。

```
#pragma once
// ... the contents of this header file
```

更多内容参见第 11 章。但是, 正如我们提到的, 本书使用 C++20 的模块而不是旧式风格的头文件。

#### 4. 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。在 main()函数中, 可以忽略显式的 return 语句, 这种情况下会自动返回 0。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。

#### 5. 输入输出流

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地输出。std::cout 就是对应于用户控制台或标准输出的滑槽, 此外还有其他滑槽, 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()函数的更多细节会在第 2 章讨论, 但是我们可以使用它的基本形式重写之前的代码。

```
std::cout << std::format("There are {} ways I love you.", 219) << 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字符是一个转义字符(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1-2列出了最常用的转义字符。

表1-2 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。不建议过度地使用 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。

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：

```
int value;
std::cin >> value;
```

需要慎重对待用户输入，因为永远都不会知道用户会输入什么类型的数据。第13章将全面介绍如何使用输入流。

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format()和流库，主要原因是printf()和scanf()未提供类型安全。

### 1.1.2 名称空间

名称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名称空间，从而指定定义名称的环境。为将某段代码加入名称空间，可用namespace块将其包含。下面是一个例子：

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

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

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

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

```
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名称空间，实际上就是完全取消了名称空间。如果使用了两个同名的名称空间，将再次出现名称冲突问题。另外，应该知道代码在哪个名称空间内运行，这样就不会无意中调用错误版本的函数。

前面已经看到了名称空间的语法——在 Hello World 程序中已经使用过名称空间，`cout` 和 `endl` 实际上是定义在 `std` 名称空间中的名称。可使用 `using` 指令重新编写 Hello World 程序，如下所示。

```
import <iostream>

using namespace std;

int main()
{
    cout << "Hello, World! " << endl;
}
```

#### 注意：

本书的大多数代码片段都假定已经对 `std` 名称空间使用了 `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` 指令引用名称空间内的特定项。例如，如果只想使用 `std` 名称空间中的 `cout`，可以使用如下的 `using` 声明。

```
using std::cout;
```

后面的代码可使用 `cout` 而不需要预先指明这个名称空间，但仍然需要显式说明 `std` 名称空间中的其他项。

```
using std::cout;
cout << "Hello, World!" << std::endl;
```

#### 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，例如名称空间或类作用域中，是可以的，甚至是在文件头部。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。

### 1. 嵌套的名称空间

嵌套的名称空间，即将一个名称空间放在另一个名称空间中。各个名称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名称空间。

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

## 2. 名称空间别名

可使用名称空间别名，为另一个名称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
- 十六进制字面量0x7B
- 二进制字面量0b1111011

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")

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

可以在数值字面量中使用数字分隔符。数字分隔符是一个单引号。例如：

- 23'456'789
- 0.123'456f

### 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 };
cout << format("{} is a random value", uninitializedInt) << endl;
cout << format("{} was assigned an initial value", initializedInt) << endl;
```

**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：

```
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以及推荐它的原因。

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 1-3 列出了 C++ 中最常见的变量类型。

表 1-3 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)	<code>int i {-7};</code> <code>signed int i {-6};</code> <code>signed i {-5};</code>
(signed) short (int)	短整型整数(通常是 2 字节)	<code>short s {13};</code> <code>short int s {14};</code> <code>signed short s {15};</code> <code>signed short int s {16};</code>
(signed) long (int)	长整型整数(通常是 4 字节)	<code>long l {-7L};</code>
(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)	<code>long long ll {14LL};</code>
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 $\geq 0$	<code>unsigned int i {2U};</code> <code>unsigned j {5U};</code> <code>unsigned short s {23U};</code> <code>unsigned long l {5400UL};</code> <code>unsigned long long ll {140ULL};</code>
float	浮点型数字	<code>float f {7.2f};</code>
double	双精度数字，精度不低于 float	<code>double d {7.2};</code>
long double	长双精度数字，精度不低于 double	<code>long double d {16.98L};</code>
char unsigned char signed char	单个字符	<code>char ch {'m'};</code>
char8_t(从 C++20 开始) char16_t char32_t	单个 $n$ 位 UTF- $n$ 编码的 Unicode 字符， $n$ 可以是 8, 16, 32	<code>char8_t c8 {u8'm'};</code> <code>char16_t c16 {u'm'};</code> <code>char32_t c32 {U'm'};</code>
wchar_t	单个宽字符，大小取决于编译器	<code>wchar_t w {L'm'};</code>
bool	布尔类型，取值为 true 或 false	<code>bool b {true};</code>

`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。

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。

下面是一些例子：

```
cout << "int:\n";
cout << format("Max int value: {}\n", numeric_limits<int>::max());
cout << format("Min int value: {}\n", numeric_limits<int>::min());
cout << format("Lowest int value: {}\n", numeric_limits<int>::lowest());

cout << "\ndouble:\n";
cout << format("Max double value: {}\n", numeric_limits<double>::max());
cout << format("Min double value: {}\n", numeric_limits<double>::min());
cout << format("Lowest double value: {}\n", 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: 1.79769e+308
Min double value: 2.22507e-308
Lowest double value: -1.79769e+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

可以用一个 `{0}` 的统一初始化器将变量初始化为 0，0 在这里是可选的。一对空的花括号组成的统一初始化器，`{}`，称为零初始化器。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以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 0 除以非零数得到的结果。
- *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`。

## 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 1-4 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 章将介绍如何将这些运算符用到自定义类型上。

表 1-4 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 true/false(非零或零)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 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 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 += j;
-=	i=i-j; 的简写	i -= j;
*=	i=i*j; 的简写	i *= j;
/=	i=i/j; 的简写	i /= j;
%=	i=i%j; 的简写	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;

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;
someShort = static_cast<short>(someInteger);
someLong = someShort * 10000;
someFloat = someLong + 0.785f;
someDouble = static_cast<double>(someFloat) / 100000;
cout << someDouble << endl;

```

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++ 运算符的优先级表格，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单独求值。

规范地说，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 1-3 中出现的运算符的优先级，越靠上的运算符优先级越高，它们也将先执行。

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

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

### 1.1.6 枚举类型

整数代表某个数字序列中的值。枚举类型允许你定义自己的序列，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。更糟糕的是，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，PieceTypeKing 具有整型值 1，编译器为 PieceTypeQueen 赋予整型值 2，PieceTypeRook 的值为 10，编译器自动为 PieceTypePawn 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：

```
if (PieceType::Queen == 2) {...}
```

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字冲突。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，像下文描述的那样。

从 C++20 开始，可以使用 using enum 声明来避免使用枚举值的全名。这是一个例子：

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



另外，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++20 允许不使用枚举值的全名，仍然建议审慎地使用这个特性。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。

#### 旧式风格枚举类型

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被使用。Visual C++ 2019 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旧式风格枚举。

### 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。请注意，导入自定义模块时，不得使用尖括号。

```
import <iostream>;
import <format>;
import employee;
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.
    cout << format("Employee: {}{}", anEmployee.firstInitial,
                  anEmployee.lastInitial) << endl;
    cout << format("Number: {}", anEmployee.employeeNumber) << endl;
    cout << format("Salary: ${}", anEmployee.salary) << endl;
}
```

### 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) { ... }
```

在这个示例中，初始化器获得一个雇员，并且判断被检索雇员的薪水是否超出 1000。只有满足条件才执行 if 语句体。本书将穿插列举具体示例。

## 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。此外还可提供 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 更改为 84,然后执行与 Default 和 Standard 相同的语句。换句话说,“Custom”情况跌落进了“Standard/Default”案例。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;
}
```

#### 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。

```
cout << ((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。

```
cout << (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 1-5 列出了常见的逻辑比较运算符。

表 1-5 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) { std::cout << "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) { std::cout << "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) { std::cout << "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) { std::cout << "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) { std::cout << "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) { std::cout << "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，因此不会对其他部分求值。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以及包含短路的指针。



### 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是 true 或 false，所以它不能返回布尔类型。相反，它返回类枚举(enum-like)<sup>①</sup>类型，定义在<compare>和 std 名称空间中。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

如果操作数是浮点类型，结果是一个偏序(partial ordering)。

- 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: 第一个操作数等于第二个
- 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

以下是它的用法的示例：

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

还有一种弱排序，这是可以选择的另一种排序类型，以针对你自己的类型实现三向比较。

- 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
- weak\_ordering::equivalent: 第一个操作数等于第二个

对于原始类型，与仅使用=、<和>运算符进行单个比较相比，使用三元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

<sup>①</sup> 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。

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 (result == strong_ordering::less) { cout << "less" << endl; }
if (result == strong_ordering::greater) { cout << "greater" << endl; }
if (result == strong_ordering::equal) { cout << "equal" << endl; }
```

### 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。下面的函数定义输出了两个参数的值：

```
void myFunction(int i, char c)
{
    cout << format("the value of i is {}", i) << endl;
    cout << format("the value of c is {}", c) << endl;
}
```

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：

```
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，但是它们必须全部解析为同一类型。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)
{
    cout << format("Entering function {}", __func__) << endl;
    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，参数的数量和/或类型必须不同。例如，以下代码段定义了两个名为 `addNumbers()` 的函数，一个函数为整数定义，另一个函数为 `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。

```
cout << addNumbers(1, 2) << endl;      // Calls the integer version
cout << 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。



从 C++ 20 开始，可以字符串形式为`[[nodiscard]]` 属性提供一个原因，例如：

```
[[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，则此函数定义可能会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、枚举以及枚举值。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。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。这是一个例子：

```
[[noreturn]] void forceProgramTermination()
{
    std::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 {
        bool isLicensed { featureId == 42 };
        // Dongle available, perform license check of the given feature...
        return isLicensed;
    }
}

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

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，该参数可用于解释不赞成使用的原因，如以下示例所示。

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

如果使用了已弃用的函数，你将会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 method, please use xyz
```

### 5. [[likely]]和[[unlikely]]

这些可能性属性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。请注意，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。但在某些情况下，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;
}
```

### 1.1.14 C 风格的数组

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。常量表达式将在本章稍后讨论。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：

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

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：

```
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。三维数组和更高维的数组更难描绘，也极少使用。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

#### 注意：

在 C++ 中，最好避免使用本节讨论的 C 风格的数组，而改用标准库功能，如 `std::array` 和 `std::vector`，如以下两节所述。

### 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。在 `array<int, 3>` 中，`array` 后面的尖括号的用法在第 12 章中讨论。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 };
cout << format("Array size = {}", arr.size()) << endl;
cout << format("2nd element = {}", arr[1]) << endl;
```

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，在编译时必须知道这一点。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` 会自动增加其大小。

### 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.
cout << format("1st element: {}", myVector[0]) << endl;
```

`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 };
```

```
cout << format("{} {}", myPair.first, myPair.second);
```

pair 也支持 CTAD，所以你可以按下列方式定义 myPair：

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

### 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 nullptr、end()、-1、EOF 之类的。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。

```
cout << "data1.has_value = " << data1.has_value() << endl;
if (data2) {
    cout << "data2 has a value." << endl;
}
```

如果 optional 有值，可以使用 value()或解引用运算符访问它。

```
cout << "data1.value = " << data1.value() << endl;
cout << "data1.value = " << *data1 << endl;
```

如果你对一个空的 optional 使用 value()，将会抛出 std::bad\_optional\_access 异常。异常将会在本章稍后介绍。

value\_or()可以用来返回 optional 的值，如果 optional 为空，则返回指定的值。

```
cout << "data2.value = " << data2.value_or(0) << endl;
```

请注意，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.
cout << format("theString: {}", theString) << endl;
cout << format("theInt: {}", theInt) << endl;
```

通过使用 auto& 或 const auto& 代替 auto，还可以使用结构化绑定语法创建一组对非 const 的引用或 const 引用。本章稍后将讨论对非 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) {
    cout << "This is silly." << endl;
    ++i;
}
```

在循环中可使用 break 关键字立刻跳出循环并继续执行程序。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 {
    cout << "This is silly." << endl;
    ++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) {
    cout << "This is silly." << endl; [1]
}
```

### 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 以及第 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) { cout << i << endl; }
```

#### 基于范围的 for 循环的初始化器

从 C++ 20 开始，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>; <for-range-declaration> : <for-range-initializer>) { <body> }
```

任何在<initializer>中引入的变量只能被用于<for-range-initializer>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) { cout << i << endl; }
```

#### 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 中存储的对象类型。下例演示如何使用初始化列表：

```
import <initializer_list>;
```

```

using namespace std;

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

```

`makeSum()`函数接收一个整型类型的初始化列表作为参数。函数体使用“基于范围的 for 循环”累加总数。可按如下方式使用该函数：

```

int a { makeSum({ 1, 2, 3 }) };
int b { makeSum({ 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 `makeSum()` 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，如下所示：

```

int c { makeSum({ 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和更安全的 `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" };
cout << format("The value of myString is {}", myString) << endl;
cout << format("The second letter is {}", myString[1]) << endl;

```

## 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所需的所有背景知识。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)以及方法(行为)。每个数据成员以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，推荐把所有的数据成员都声明为 private，在需要时，可通过 public 或 protected 的获取器(getter)和设置器(setter)访问它们。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。关于 protected 的用法，将在第 5 章和第 10 章介绍“继承”时讲解。

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，同时也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 <string>

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在模块实现文件(.cpp)中。源文件以如下的模块声明开头，告诉编译器这是 airline\_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; }
// Other get and set methods have a similar implementation.
```

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() };
cout << format("This ticket will cost ${}", cost) << endl;

```

上面的示例代码显示了创建和使用类的一般语法。当然，还有许多内容需要学习。第 8~10 章将更深入地讲述 C++ 定义类的特定机制。

### 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。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。可以使用名称空间、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，并且在 `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，然后在下一个范围内查找，以此类推，直到全局范围。不在名称空间、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范围内。如果在全局范围内未找到，则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` 名称空间里的 `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;
    cout << d.get() << endl;           // prints 5
    cout << NS::get() << endl;         // prints 20
    cout << ::get() << endl;          // prints 10
    cout << get() << endl;            // prints 10
}

```

请注意，如果将名为 `NS` 的名称空间定义为未命名/匿名的，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名称空间中的 `get()`。

```
cout << get() << endl;
```

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<sup>①</sup>, 只需要指定一对空的大括号。例如:

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

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:

```
void func(int i) { /* ... */ }

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

在 main() 的两行代码中, C++ 在对 x 赋值或调用 func() 之前, 会自动将 3.14 截断为 3。注意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

---

<sup>①</sup> 使用默认构造函数构造对象, 将基本整数类型(如 char 和 int 等)初始化为 0, 将浮点类型初始化为 0.0, 将指针类型初始化为 nullptr。

视，因为它们可能会引起细微的错误。使用统一初始化，如果编译器完全支持 C++11 标准，`x` 的赋值和 `func()` 的调用都会生成编译错误。

```
int x { 3.14 }; // Error because narrowing
func({ 3.14 }); // Error because narrowing
```

如果你需要窄化转换，建议使用准则支持库(GSL)<sup>①</sup>中提供的 `gsl::narrow_cast()` 函数。

统一初始化还可用来初始化动态分配的数组：

```
int* myArray = new int[4] { 0, 1, 2, 3 };
```

从 C++20 开始，可以省略数组的大小 4，像下面这样：

```
int* myArray = new int[] { 0, 1, 2, 3 };
```

统一初始化还可在构造函数初始化器中初始化类成员数组：

```
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

C++20

C++20 引入了指派初始化器，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或类的对象：仅 `public` 数据成员、无用户声明或继承的构造函数、无虚函数和无虚基类、`private` 或 `protected` 的基类(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，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` 结构是使用如下的统一初始化语法初始化的：

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

<sup>①</sup> GSL 的仅头文件实现可以在 [github.com/Microsoft/GSL](https://github.com/Microsoft/GSL) 中找到。

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，在这种情况下，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，这是不可以的，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时大小可变的数据，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。

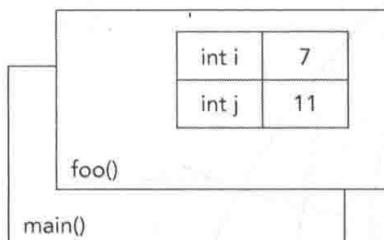


图 1-2 foo()函数中声明了两个整型值

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，可以将它当作一堆比特。程序可在任何时候向其中添加新的比特或修改已有的比特。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，可以将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，最终会导致程序崩溃。

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。从技术上讲，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，在此假定存在一个名为 `getEmployee()` 的函数，它返回一个指向 `Employee` 实例的指针。

```
Employee* anEmployee { getEmployee() };
cout << (*anEmployee).salary << endl;
```

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。

```
Employee* anEmployee { getEmployee() };
cout << anEmployee->salary << endl;
```

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。

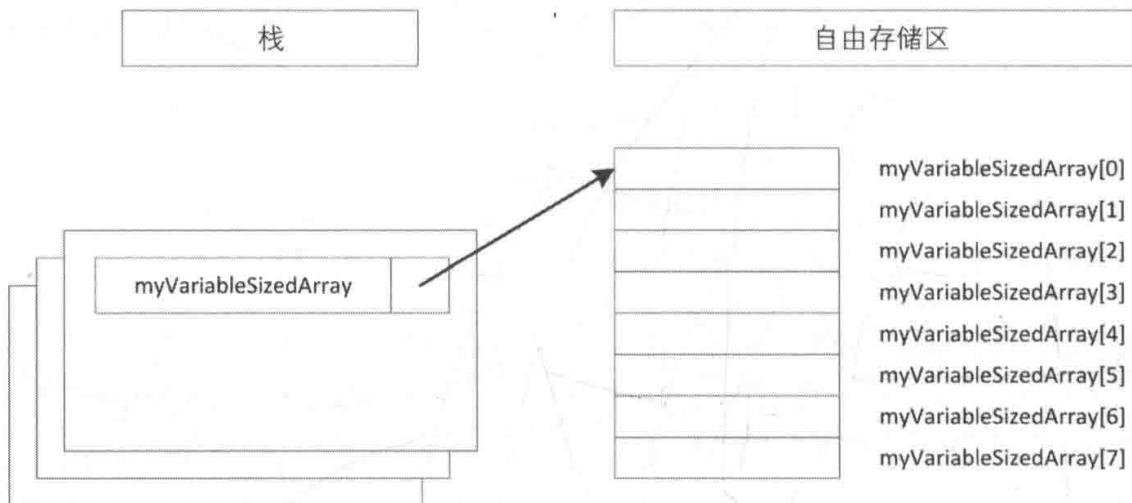


图 1-3 自由存储区

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：

避免使用 C 中的 `malloc()` 和 `free()`，而使用 `new` 和 `delete`，或者使用 `new[]` 和 `delete[]`。

#### 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。`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：

```
void func(int i) { cout << "func(int)" << endl; }

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，用来保证此要求。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，就正确地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 { 3.141592653589793238462 };
```

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，你要阻止第二行还是第三行？

为了防止指向的值被修改(如第三行所示)，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` 关键字作用于其直接左侧的内容。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` 可以放在变量之前，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，正如以下示例：

```
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);
}

```

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` 方法

`const` 关键字的第二个用途是将类方法标记为 `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, 以将所有只读方法标记为 `const`。如果任何 `const` 方法尝试修改 `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 };
};

string AirlineTicket::getPassengerName() const
{
    return m_passengerName;
}
// Other methods 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` 成员函数也被称为赋值函数(mutator)相对, 这些成员函数也称为检查器(inspector)。

## 1.1.28 `constexpr` 关键字

C++一直有常量表达式的概念, 即在编译器求值的表达式。在某些情况下, 必须使用常量表达式。例如, 定义数组时, 数组的大小需要为常量表达式。由于此限制, 以下代码在 C++ 中无效。

```

constexpr int getArraySize() { return 32; }

int main()
{
    int myArray[getArraySize()]; // Invalid in C++
}

```

使用 `constexpr` 关键字，`getArraySize()` 函数可以被重定义，允许在常量表达式中调用它。

```
constexpr int getArraySize() { return 32; }

int main()
{
    int myArray[getArraySize()]; // OK
}
```

你甚至可以这样做：

```
int myArray[getArraySize() + 1]; // OK
```

将函数声明为 `constexpr` 对函数的功能施加了很多限制，因为编译器必须能够在编译时对函数求值。例如，允许 `constexpr` 函数调用其他 `constexpr` 函数，但不允许调用任何非 `constexpr` 函数。这样的函数不允许有任何副作用，也不能引发任何异常。`constexpr` 函数是 C++ 的高级功能，因此在本书中不再详细讨论。

通过定义 `constexpr` 构造函数，可以创建用户自定义类型的常量表达式变量。与 `constexpr` 函数一样，`constexpr` 类也有很多限制，这在本书中也不再赘述。但是，为了让你对可能发生的事情有一个了解，下面是一个示例。以下 `Rect` 类定义了 `constexpr` 构造函数。它还定义了执行一些计算的 `constexpr` `getArea()` 方法。

```
class Rect
{
public:
    constexpr Rect(size_t width, size_t height)
        : m_width{width}, m_height{height} {}

    constexpr size_t getArea() const { return m_width * m_height; }

private:
    size_t m_width{0}, m_height{0};
};
```

使用这个类声明 `constexpr` 对象是非常容易的。

```
constexpr Rect r{8, 2};
int myArray[r.getArea()]; // OK
```

C++20

### 1.1.29 `consteval` 关键字

上一节中讨论的 `constexpr` 关键字指定函数可以在编译期执行，但不能保证一定在编译期执行。采用以下 `constexpr` 函数：

```
constexpr double inchToMm(double inch) { return inch * 25.4; }
```

如果按以下方式调用，则会在编译时满足需要对函数求值。

```
constexpr double const_inch{6.0};
constexpr double mm1{inchToMm(const_inch)}; // at compile time
```

然而，如果按以下方式调用，函数将不会在编译期被求值，而是在运行时！

```
double dynamic_inch{8.0};
double mm2{inchToMm(dynamic_inch)}; // at run time
```

如果确实希望保证始终在编译时对函数进行求值，则需要使用 C++20 的 `consteval` 关键字将函数转换为所谓的立即函数(immediate function)。可以按照如下方式更改 `inchToMm()` 函数：

```
consteval double inchToMm(double inch) { return inch * 25.4; }
```

现在，对 `inchToMm()` 的第一次调用仍然可以正常编译，并且可以在编译期进行求值。但是，第二个调用现在会导致编译错误，因为无法在编译期对其进行求值。

### 1.1.30 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，可以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，并从函数和方法返回引用。

#### 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。这是一个代码示例：

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

可以尝试使用 `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目的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，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，因此不能编写例如 unnamedRef2 = 7 这样的代码。

```
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;
```

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，后面的“引用参数”一节会讨论。

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，必须在所谓的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对 const 引用的参数解决该问题。

这是另一个引用派上用场的例子，这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。

当你有一个指针但函数或方法只能接收引用时，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。

因此，我们有以下规则：

### 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效率。当将值传递给函数时，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：

```
void printString(const string& myString)
{
    cout << myString << endl;
}

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，但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，应该只在对于简单的内置类型，例如 `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，则通过对非 `const` 的引用将其传递。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是，能够直接把返回值作为左值(赋值语句的左侧)对其进行赋值。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。

## 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，该函数接受对 `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所谓的多态性，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，而使用所谓的智能指针(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。总共有 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];
    }
}
}
}

```

在这种情况下，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_ts` 的引用。使用此版本的函数，不再需要传递指针或 `size_ts` 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干净得多，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()` 函数重写为更安全、更短、更美观并且更具可读性，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。

使用向量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！如果 `object` 是局部变量、函数参数或临时值，`return object` 格式的声明将会触发返回值优化(RVO)。此外，如果对象是局部变量，命名返回值优化(NRVO)将会生效。RVO 和 NRVO 都是复制省略(copy elision)的形式，使从函数中返回对象非常高效。使用复制省略功能，编译器可以避免复制从函数返回的对象，这构成零复制值传递语义。

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向量作为参数。它也使用了 C++ 20 的指派初始化器。

```
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。

#### 1.1.31 `const_cast()`

在 C++ 中，每个变量都有特定的类型。在某些情况下，有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()`（自 C++20 起），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：一个函数指定接收 `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 ThirdPartyLibraryMethod(char* str);

void f(const char* str)
{
    ThirdPartyLibraryMethod(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) };

```

### 1.1.32 异常

C++是一种非常灵活的语言，但并不是非常安全。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。

异常是一种预料之外的情形。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“处理错误”详细介绍了异常处理，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 {
    cout << divideNumbers(2.5, 0.5) << endl;
    cout << divideNumbers(2.3, 0) << endl;
    cout << divideNumbers(4.5, 2.5) << endl;
} catch (const invalid_argument& exception) {
    cout << format("Exception caught: {}", exception.what()) << endl;
}
```

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
An exception was 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。

### 1.1.33 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用于为现有类型声明引入同义词而无须创建新类型的语法。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本来的名称。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可方便管理的名称。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>` 来表示字符串。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，指定函数参数等等，必须要写 `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);
}
```

### 1.1.34 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，但是要复杂得多。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*;
```

可以用 `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详细信息。

**警告：**

总是优先选择类型别名而不是 `typedef`。

### 1.1.35 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。
- 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。
- `decltype(auto)`，见第 12 章。
- 其他函数语法，见第 12 章。
- 泛型 `lambda` 表达式，见第 19 章。

`auto` 可用于告诉编译器，在编译时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。由于自动去除引用和 `const` 限定符，因此以下结果变量的类型为 `string`，而不是

const string&类型，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 };
```

当将 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 };
```

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 C++17 引入的直接列表初始化之间就存在重要区别。

从 C++17 之后，你会得到如下结果(需要<initializer\_list>)。

```
// 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.
```

请注意，对于拷贝列表初始化，带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
```

在早期的标准版本(C++11/14)中，拷贝列表和直接列表初始化都将推断出 initializer\_list<>。

```
// Copy list initialization
auto a = { 11 };           // initializer_list<int>
auto b = { 11, 22 };        // initializer_list<int>

// Direct list initialization
auto c { 11 };             // initializer_list<int>
auto d { 11, 22 };          // initializer_list<int>
```

## 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() };
```

刚开始会觉得 decltype 有多大价值。但在模板环境中，decltype 会变得十分强大，详见第 12 和 26 章。

### 1.1.36 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某些类，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。从本书的开始，在示例中就会使用这些代码，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、名称空间、引用以及其他语言特性。

### 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，包括过去以及现在的雇员
- 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 提供程序的交互接口。

### 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。此外还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 <string>;
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<string>功能。此代码还声明花括号中包含的后续代码位于 Records 名称空间中，为使用特定代码，整个程序都会用到 Records 名称空间。

接下来，在 Records 名称空间内定义以下两个常量。请注意，本书使用约定不以任何特殊字母作为常量的前缀。

```
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 访问它。

第二个常量是用于晋升或降级雇员的默认薪资涨幅或跌幅。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，以默认涨薪的两倍将雇员提拔。

接下来，声明了 Employee 类及其 public 方法。

```

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; // Outputs 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;
}

```

提供了一个接受名字和姓氏的构造函数。promote()和demote()方法都具有整数参数，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 <iostream>;
import <format>;
using namespace std;

```

第一行指定了此源文件的模块，接下来是<iostream>和<format>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()`方法只是用一些新值调用`setSalary()`方法。注意，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()`方法正确设置了`m_hired`数据成员。

```
void Employee::hire() { m_hired = true; }
void Employee::fire() { m_hired = false; }
```

`display()`方法使用控制台输出流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
{
    cout << format("Employee: {}, {}", getLastName(), getFirstName()) << endl;
    cout << "-----" << endl;
    cout << (isHired() ? "Current Employee" : "Former Employee") << endl;
    cout << format("Employee Number: {}", getEmployeeNumber()) << endl;
    cout << format("Salary: ${}", getSalary()) << endl;
    cout << endl;
}
```

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;
}
// ... other getters and setters omitted for brevity
}
```

即使这些方法看起来微不足道，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()`方法中执行边界检查，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 <iostream>;
import employee;

using namespace std;
using namespace Records;

int main()
{
    cout << "Testing the Employee class." << endl;
    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();
}
```

另一种测试各个类的方法是使用单元测试，详见第 30 章。

## 1.2.3 Database 类

`Database` 类使用标准库中的 `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 <string>;
import <vector>;
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()` 方法来获得雇员的引用。为这个方法声明了两个版本，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，因此具有输出所有雇员、当前在职雇员以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()` 方法的实现如下：

```
module database;
import <stdexcept>

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()` 方法创建一个新的 `Employee` 对象，在其中填充信息并将其添加到 `vector` 中。注意当使用了这个方法后，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()` 方法返回 `vector` 中最后一个元素的引用，即最新添加的雇员。

`getEmployee()` 方法之一的实现如下。第二版本以类似的方式实现，因此未示出。他们都使用基于范围的 `for` 循环遍历 `m_employees` 中的所有雇员，并检查 `Employee` 是否与传递给该方法的信息匹配。如果找不到匹配项，则会引发异常。

```
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。`displayFormer()`类似于`displayCurrent()`。

```

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(); }
    }
}

```

### 3. DatabaseTest.cpp

用于数据库基本功能的简单测试如下所示：

```

import <iostream>;
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();

    cout << "all employees: " << endl << endl;
    myDB.displayAll();

    cout << endl << "current employees: " << endl << endl;
    myDB.displayCurrent();

    cout << endl << "former employees: " << endl << endl;
    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。

main()函数是一个显示菜单的循环，执行被选中的操作，然后重新开始循环。对于大多数的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 <iostream>;
import <stdexcept>;
import <exception>;
import <format>;
import <string>;
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:
                cerr << "Unknown command." << endl;
                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;
    cout << endl;
    cout << "Employee Database" << endl;
    cout << "-----" << endl;
    cout << "1) Hire a new employee" << endl;
    cout << "2) Fire an employee" << endl;
    cout << "3) Promote an employee" << endl;
    cout << "4) List all employees" << endl;
    cout << "5) List all current employees" << endl;
    cout << "6) List all former employees" << endl;
    cout << "0) Quit" << endl;
    cout << endl;
    cout << "---> ";
    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;

    cout << "First name? ";
    cin >> firstName;

    cout << "Last name? ";
    cin >> lastName;

    auto& employee { db.addEmployee(firstName, lastName) };
    cout << format("Hired employee {} {} with employee number {}.", 
        firstName, lastName, employee.getEmployeeNumber()) << endl;
}
```

`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 方法进行修改。

```
void doFire(Database& db)
{
    int employeeNumber;
    cout << "Employee number? ";
    cin >> employeeNumber;

    try {
        auto& emp { db.getEmployee(employeeNumber) };
        emp.fire();
        cout << format("Employee {} terminated.", employeeNumber) << endl;
    } catch (const std::logic_error& exception) {
        cerr << format("Unable to terminate employee: {}",
```

```

        exception.what()) << endl;
    }

}

void doPromote(Database& db)
{
    int employeeNumber;
    cout << "Employee number? ";
    cin >> employeeNumber;

    int raiseAmount;
    cout << "How much of a raise? ";
    cin >> raiseAmount;

    try {
        auto& emp { db.getEmployee(employeeNumber) };
        emp.promote(raiseAmount);
    } catch (const std::logic_error& exception) {
        cerr << format("Unable to promote employee: {}", exception.what()) << endl;
    }
}

```

### 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)没有暴露出 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，然后再从网站上查看解决方案。

**练习 1-1** 修改本章开始的 Employee 结构体，将其放在一个名为 HR 的名称空间中。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 C++ 20 的指派初始化器。

**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 名称空间中。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** 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，以尽可能地使用引用，并正确使用 const。

**练习 1-6** 修改练习 1-5 中的 AirlineTicket 类，使其包含一个可选的常旅客号码。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常旅客号码。修改 main() 函数来测试你的实现。

# 第2章

## 使用 `string` 和 `string_view`

### 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。“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。

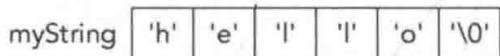


图 2-1 6 个字符的空间

C++包含一些来自 C 语言的字符串操作函数，它们在<cstring>头文件中定义。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，函数通过 `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 位模式下编译时，`s3` 的值为 4；而在 64 位模式下编译时，`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废弃了。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。

```
cout << "hello" << endl;
```

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，编译器可重用等效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程序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。可能会导致程序崩溃；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*`，因此编译器会捕捉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，也不会进行字面量的池操作。

```
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。

```
const char* str { "Hello "World"!" }; // Error!
```

对于普通字符串，必须转义双引号，如下所示。

```
const char* str { "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。

```
const char* str { R"(Hello "World"!)" };
```

如果需要一个包含多行的字符串，不使用原始字符串字面量的话，就需要在字符串中新行的开始位置嵌入\n 转义序列。例如：

```
const char* str { "Line 1\nLine 2" };
```

如果将这个字符串输出到控制台，将看到以下结果。

```
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。

```
const char* str { 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)。

```
const char* str { R"(Is the following a tab character? \t)" };
```

因此，如果将此字符串输出到控制台，将看到以下结果。

```
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，因为在这个字符串的中间包含)"。

```
const char* str { 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。

```
const char* str { 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 名称空间的<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, 从而可让人思考应该有更好的方式。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 风格字符串相比一等数据类型的那些优势实际上无关紧要的。

#### 2. 使用 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。例如, 给 string 重新定义+运算符, 以表示“字符串串联”。下面的例子会得到 1234:

```
string a { "12" };
string b { "34" };
string c;
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-指针的双重性。

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。单独的字符可通过方括号运算符`[]`访问。

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) { cout << "less" << endl; }
if (result > 0) { cout << "greater" << endl; }
if (result == 0) { cout << "equal" << endl; }
```

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)) { cout << "equal" << endl; }
```

`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`==`。

C++20 通过第 1 章介绍的三向比较运算符改进了这一切。字符串类完全支持这个运算符，下面是一个例子：

```
auto result { a <= b };
if (is_lt(result)) { cout << "less" << endl; }
if (is_gt(result)) { cout << "greater" << endl; }
if (is_eq(result)) { cout << "equal" << endl; }
```

### 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。

```
string myString { "hello" };
myString += ", there";
string myOtherString { myString };
if (myString == myOtherString) {
    myOtherString[0] = 'H';
}
cout << myString << endl;
cout << myOtherString << endl;
```

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` 对象都创建为堆中的变量。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，以下是一部分：

- `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`。

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));
}
cout << strHello << endl;
```

输出如下：

```
Hello World
```

可参阅标准库参考资源，查看可在 `string` 对象上执行的所有受支持操作。

### 注意：

从 C++20 开始，`std::string` 是第 1 章介绍的 `constexpr` 类。这意味着 `string` 可用于在编译时执行操作，并可用于 `constexpr` 函数和类的实现。

## 3. `std::string` 字面量

源代码中的字符串字面量通常解释为 `const char*`。使用用户定义的标准字面量 `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"s };     // string2 is an std::string.
```

标准用户定义字面量 `s` 在 `std::literals::string_literals` 名称空间中定义。但是，`string_literals` 和 `literals` 名称空间都是所谓的内联名称空间。因此，使用以下选项可以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，在内联名称空间中声明的所有内容都会自动在父名称空间中可用。要自己定义内联名称空间，可以使用 `inline` 关键字。例如，`string_literals` 内联名称空间定义如下：

```
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*>`，而不是 `vector<string>`！这是一个很容易犯的错误，可能会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，下面一节将详细解释。

#### 1. 高级数值转换函数

`std` 名称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 以及 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) };
```

### 字符串转换为数值

通过下面这组同样在 std 名称空间中定义的函数，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，idx 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。idx 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 \*idx=0, int base=10);
- long stol(const string& str, size\_t \*idx=0, int base=10);
- unsigned long stoul(const string& str, size\_t \*idx=0, int base=10);
- long long stoll(const string& str, size\_t \*idx=0, int base=10);
- unsigned long long stoull(const string& str, size\_t \*idx=0, int base=10);
- float stof(const string& str, size\_t \*idx=0);
- double stod(const string& str, size\_t \*idx=0);
- long double stold(const string& str, size\_t \*idx=0);

下面是一个示例：

```
const string toParse { " 123USD" };
size_t index { 0 };
int value { stoi(toParse, &index) };
cout << format("Parsed value: {}", value) << endl;
cout << format("First non-parsed character: '{}'", toParse[index]) << endl;
```

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)。最终的结果是，这些函数可以比其他高级数值转换函数快几个数量级。这些函数也是为所谓的完美往返而设计的，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`。

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{}) { cout << out << endl; /* 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{}) { cout << out << endl; /* 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 `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.hhhpf+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{}) { cout << out << endl; /* 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。

```
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 `from_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{}) { cout << out << endl; /* Conversion successful. */ }

double value2;
auto [ptr2, error2] { from_chars(out.data(), out.data() + out.size(), value2) };
if (error2 == errc{}) {
    if (value1 == value2) {
        cout << "Perfect roundtrip" << endl;
    } else {
        cout << "No perfect roundtrip?!?" << endl;
    }
}
```

## 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*`。更糟糕的是，函数将失去 `std::string` 良好的面向对象的方面及其良好的辅助方法。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` 支持与 `std::string` 类似的接口。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)` 方法, 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。

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。

```
string_view extractExtension(string_view filename)
{
    return filename.substr(filename.rfind('.'));
}
```

该函数可用于所有类型的不同字符串:

```
string filename { R"(c:\temp\my file.ext)" };
cout << format("C++ string: {}", extractExtension(filename)) << endl;

const char* cString { R"(c:\temp\my file.ext)" };
cout << format("C string: {}", extractExtension(cString)) << endl;

cout << format("Literal: {}", extractExtension(R"(c:\temp\my file.ext)")) << endl;
```

在对 `extractExtension()` 的所有这些调用中, 没有进行一次复制。`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 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, 这也是有用的。

```
const char* raw { /* ... */ };
size_t length { /* ... */ };
cout << format("Raw: {}", extractExtension({ raw, length })) << endl;
```

最后一行代码也可以写成这样:

```
cout << format("Raw: {}", extractExtension(string_view { raw, length })) << endl;
```

### 注意:

在每当函数需要将只读字符串作为一个参数时, 可使用 `std::string_view` 替代 `const std::string&` 或 `const char*`。

无法从 `string_view` 隐式构建一个 `string`。要么使用一个显式的 `string` 构造函数, 要么使用 `string_view::data()` 成员。例如, 假设有以下接收 `const string&` 的函数。

```
void handleExtension(const string& extension) { /* ... */ }
```

不能采用如下方式调用该函数:

```
handleExtension(extractExtension("my file.ext"));
```

下面是两个可供使用的选项：

```
handleExtension(extractExtension("my file.ext").data()); // data() method
handleExtension( string { extractExtension("my file.ext") } ); // explicit ctor
```

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 };
```

你可以对 `string_view` 使用 `data()` 方法，如下所示：

```
auto result1 { str + sv.data() };
```

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。存储 `std::string` 更安全。

### 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!" };
cout << 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

直到 C++20 之前，字符串的格式化一般是通过 `printf()` 之类的 C 风格函数或是 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。其基本形式在前一章介绍，并已在示例中使用。现在是时候看看 `format()` 的强大之处了。

`format()` 的第一个参数是待格式化的字符串，后续参数是用于填充待格式化字符串中占位符的值。到目前为止，使用 `format()` 时的占位符一般都是一对花括号：`{}`。在这些花括号内可以是格式为 `[index][:specifier]` 的字符串。可以省略所有占位符中的 `index`，也可以为所有占位符指定从零开始的索引，以指明应用于此占位符的第二个和后续参数。如果省略 `index`，则 `format()` 的第二个和后续参数传递的值将按给定顺序用于所有占位符。`specifier` 是一种格式说明符，用于更改值在输出中格式化的方式，将在下一节详细说明。如果需要输出 `{or}` 字符，则需要将其转义为 `\{or\}`。

先来看看如何使用 index 部分。以下对 format()的调用省略了占位符中的显式索引：

```
auto s1 { format("Read {} bytes from {}", n, "file1.txt") };
```

可以按以下方式手动指定索引：

```
auto s2 { format("Read {0} bytes from {1}", n, "file1.txt") };
```

不允许混合使用手动和自动索引。下例使用了一个无效的字符串格式化：

```
auto s2 { format("Read {0} bytes from {} ", n, "file1.txt") };
```

可以更改输出字符串中格式化值的顺序，而不需要更改传递到 format()的实参的实际顺序。如果要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，前面的格式化字符串可以翻译成中文，如下所示。在中文中，句子中占位符的顺序是颠倒的，但由于格式化字符串中的占位符是有索引的，format()函数的参数顺序保持不变。(注意，有关L前缀的用法可参见第21章)。

```
auto s3 { format(L"从{1}中读取{0}个字节。", n, L"file1.txt") };
```

下一节将深入探讨如何使用格式说明符控制输出。

## 2.2.1 格式说明符

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，本处不作进一步讨论)：

```
[[fill][align][sign][#][0][width][.precision][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，例如 5。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 format()的实参列表中的下一个参数。

示例如下：

```
int i { 42 };
cout << format("|{:5}|", i) << endl;      // | 42|
cout << format("|{:{}|", i, 7) << endl;    // | 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 };
cout << format("|{:7}|", i) << endl;      // | 42|
cout << format("|{:<7}|", i) << endl;    // |42|
```

```
cout << format("{:_>7}", i) << endl; // |____42|
cout << format("{:_^7}", i) << endl; // |_42_|
```

### 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 };
cout << format("{:<5}", i) << endl; // |42 |
cout << format("{:<+5}", i) << endl; // |+42 |
cout << format("{:< 5}", i) << endl; // | 42 |
cout << format("{:< 5}", -i) << endl; // |-42 |
```

### 4. #

#启用所谓的备用格式(alternate formatting)规则。如果为整型启用，并且还指定了十六进制、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 的十六进制)，X(大写字母 A,B,C,D,E 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)，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)。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。

整型的示例如下：

```
int i { 42 };
cout << format("{:10d}", i) << endl; // |        42|
cout << format("{:10b}", i) << endl; // |      101010|
cout << format("{:#10b}", i) << endl; // | 0b101010|
```

```
cout << format("|\{:10X\}|", i) << endl; // | 2A|
cout << format("|\{:#10X\}|", i) << endl; // | 0X2A|
```

字符串的示例如下：

```
string s { "ProCpp" };
cout << format("|\{:^10\}|", s) << endl; // |_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。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 };
cout << format("|\{:12g\}|", d) << endl; // | 1.365870|
cout << format("|\{:12.2\}|", d) << endl; // | 1.37|
cout << format("|\{:12e\}|", d) << endl; // | 1.365870e+00|
```

```
int width { 12 };
int precision { 3 };
cout << format("|\{2:{0}.{1}f\}|", width, precision, d) << endl; // | 1.366|
```

## 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 插入在数值的前面，但在符号以及任何 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 };
cout << format("|\{:06d\}|", i) << endl; // |000042|
cout << format("|\{:+06d\}|", i) << endl; // |+00042|
cout << format("|\{:06X\}|", i) << endl; // |00002A|
cout << format("|\{:#06X\}|", i) << endl; // |0X002A|
```

### 2.2.2 格式说明符错误

如前所述，格式说明符需要遵循严格的规则。如果格式说明符包含错误，将抛出 `std::format_error` 异常。

```
try {
    cout << format("An integer: {:.}", 5);
} catch (const format_error& caught_exception) {
    cout << caught_exception.what(); // "missing precision specifier"
}
```

### 2.2.3 支持自定义类型

可以扩展 C++20 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()。模板特化和方法模板将在第 12 章进行解释。不过，为了完整起见，我们来看看掌握模板特化的情况下将如何实现自定义 formatter。在读到此处时，你还能理解这个例子中的所有语法，因为你需要等到第 12 章，但它让你知道当你在书中进一步深入时可能发生的事情，到那时，你可以回头温习这个例子。

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;
};
```

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：{:a}只输出键，{:b}只输出值，{:c}和{}同时输出键和值。

```
template<>
class formatter<KeyValue>
{
public:
    constexpr auto parse(auto& context)
    {
        auto iter = context.begin();
        const auto end = context.end();
        if (iter == end || *iter == '}') { // {} format specifier
            m_outputType = OutputType::KeyAndValue;
            return iter;
        }

        switch (*iter) {
            case 'a': // {:a} format specifier
                m_outputType = OutputType::KeyOnly;
                break;
            case 'b': // {:b} format specifier
                m_outputType = OutputType::ValueOnly;
                break;
            case 'c': // {:c} format specifier
                m_outputType = OutputType::KeyAndValue;
                break;
            default:
                throw format_error{ "Invalid KeyValue format specifier." };
        }

        ++iter;
        if (iter != end && *iter != '}') {
            throw format_error{ "Invalid KeyValue format specifier." };
        }
        return iter;
    }
}
```

```

auto format(const KeyValue& kv, auto& context)
{
    switch (m_outputType) {
        using enum OutputType;

        case KeyOnly:
            return format_to(context.out(), "{}", kv.getKey());
        case ValueOnly:
            return format_to(context.out(), "{}", kv.getValue());
        default:
            return format_to(context.out(), "{} - {}",
                            kv.getKey(), kv.getValue());
    }
}

private:
    enum class OutputType
    {
        KeyOnly,
        ValueOnly,
        KeyAndValue
    };
    OutputType m_outputType { OutputType::KeyAndValue };
};

```

`parse()`方法负责解析范围[`context.begin()`, `context.end()`)内的格式说明符。它应该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。

`format()`方法根据 `parse()` 解析的格式规范格式化第一个实参，将结果写入 `context.out()`，并返回一个指向输出末尾的迭代器。在本例中，通过将工作转发到 `std::format_to()` 来执行实际的格式化。`format_to()` 函数接收预先分配的缓冲区作为第一个参数，并将结果字符串写入其中，而 `format()` 则创建一个新的字符串对象以返回。

可以这样测试自定义 `formatter`:

```

KeyValue keyValue { "Key1", 11 };
cout << format("{}", keyValue) << endl;
cout << format("{:a}", keyValue) << endl;
cout << format("{:b}", keyValue) << endl;
cout << format("{:c}", keyValue) << endl;
try { cout << format("{:cd}", keyValue) << endl; }
catch (const format_error& caught_exception) { cout << caught_exception.what(); }

```

输出如下:

```

Key1 - 11
Key1
11
Key1 - 11
Invalid KeyValue format specifier.

```

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 C++20 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，然后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, s);
```

**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。

# 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，让他的团队假想一名一年后加入的实习生。如果没有文档而函数有好几张长，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？良好的代码由于便于阅读和理解，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找到有用的技巧。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)以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、返回值以及可能抛出的异常。

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() 方法只能在 openDatabase() 方法之后调用，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 the openDatabase() method has not  
// been called yet.  
int saveRecord(Record& record);
```

saveRecord() 方法接收一个对 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 does not yet have a database ID, then the method  
// modifies the record object to store the ID assigned by the database.  
// Throws:
```

```
// DatabaseNotOpenedException if the openDatabase() method 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()`方法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 does not yet have a database ID, then the method
// 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 the openDatabase() method 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()` 方法的建议：

```
// Parameters:
// record: If the given record does not yet have a database ID, then the method
// modifies the record object to store the ID assigned by the database.
// Throws:
// DatabaseNotOpenedException if the openDatabase() method 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、返回值类型以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，描述生成单个随机数的调用的公开文档应指定它不带参数，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[], 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所使用的算法，并说明(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，顶部的注释在较高层次说明了这个算法，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. The 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.
void sort(int data[], size_t size)
{
    // 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

使用注释的另一个原因是在高于代码的层次提供信息。元信息提供创建代码的详细信息，但是不涉及代码的特定行为。例如，某组织可能想使用元信息跟踪每个方法的原始作者。还可以使用元信息引用外部文档或其他代码。

下例给出了元信息的几个实例，包括文件的作者、创建日期、提供的特性。此外还包括表示元数据的行内注释，例如对应某行代码的 bug 编号，以及提醒以后重新访问时代码中某个可能的问题。

```

// 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) return -1; // Added to address bug #142 - jsmith 110428
    record.setId(id);
    // TODO: What if setId() throws an exception? - akshayr 110501
    return id;
}

```

每个文件的开头还可包含修改日志。下面给出一个修改日志的示例。

```

// Date      | Change
//-----+-----
// 110413   | REQ #005: <marcg> Do not normalize values.
// 110417   | REQ #006: <marcg> use nullptr instead of NULL.

```

#### 警告：

使用第 28 章讲述的源代码控制方案(也应当使用该方案)，前几个示例中就不必使用所有元信息(TODO 注释除外)。源代码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，你不必手动跟踪元信息。

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。

注释很容易走向极端。最好与团队成员讨论哪种类型的注释最有用，并制定约定。例如，如果团队的某个成员使用 TODO 注释表明代码仍然需要加工，但是其他人不知道这个约定，需要注意的代码就可能被忽略。

### 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，为了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。下例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.
}
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(kStart, kEnd, kOffset);
// 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 + kMarigoldOffset) / 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
- 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\*
- 未完成的特性\*\*
- 已知的 bug\*\*

对于标有单星号的条目，通常由源代码控制解决方案自动处理(见第 28 章)。

对于标有双星号的条目，通常由 bug 和特性追踪系统处理(见第 30 章)。

开发环境可能允许创建模板，以自动建立具有前置注释的新文件。某些源代码控制系统(例如 Subversion(SVN))甚至可以帮助填写元数据。例如，如果注释包含了字符串\$Id\$，SVN 可以自动扩展注释，包含作者、文件名、版本和日期。

下面给出了一个前置注释示例：

```
// $Id: Watermelon.cpp, 123 2004/03/10 12:52:33 marcg $
//
// 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 2017, 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(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(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。

```

void someFunction(Args... args, OtherArgs... otherArgs) {
    args...;
    otherArgs...;
}

if (args & defaultArg1) {
    defaultArg1 = true;
    args...;
    otherArgs...;
} else {
    defaultArg1 = false;
    args...;
    otherArgs...;
}

// now do something else
const char* message = args -> defaultArg1 ? "Hello" : "World";
const char* message = args -> defaultArg1 ? "Hello" : "World";
const char* message = args -> defaultArg1 ? "Hello" : "World";
}

```

图 3-2 代码的分解(1)

```

void someFunction(Args... args, OtherArgs... otherArgs) {
    args...;
    otherArgs...;
}

if (args & defaultArg1) {
    defaultArg1 = true;
    args...;
    otherArgs...;
} else {
    defaultArg1 = false;
    args...;
    otherArgs...;
}

const char* message = args -> defaultArg1 ? "Hello" : "World";
const char* message = args -> defaultArg1 ? "Hello" : "World";
const char* message = args -> defaultArg1 ? "Hello" : "World";
}

```

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。下面给出了一些可用来重构代码的技术，更全面的内容请参考附录 B 列出的重构书籍。

- 增强抽象的技术
  - 封装字段：将字段设置为私有，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
  - 提取方法：将较大方法的一部分转换成便于理解的新方法。
  - 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
  - 移动方法或字段：移到更合适的类或源文件中。
  - 重命名方法或字段：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、方法或函数，程序通常不会像把全部功能放在一起的代码那样密集，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)是保留名称，不应当使用。
- 全局名称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、类或名称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
g_Settings 表明全局身份	globalUserSpecificSettingsAndPreferences 太长
m_nameCounter 表明了数据成员身份	mNC 太简单，太模糊
calculateMarigoldOffset() 简单，明确	doAction() 太宽泛，不准确
mTypeString 赏心悦目	typeSTR256 只有计算机才会喜欢的名称
	mlIHateLarry 不恰当的内部玩笑

(续表)

适当的名称	不当的名称
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	mData	“成员”	类的数据成员
m_	m_data		
s	sLookupTable	“静态”	静态变量或数据成员
ms	msLookupTable		
ms_	ms_lookupTable		
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	bCompleted	“布尔值”	表示布尔值
is	isCompleted		

#### 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访谈法，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. 把常量放到名称空间中

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名称空间中，名称空间参见第 1 章。可以创建两个名称空间：`Menu` 和 `Button`。每个名称空间中都有一个 `Help` 常量，其用法为 `Menu::Help` 和 `Button::Help`。另一个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。

如下的示例表达式：

```
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，为了清楚起见，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，这代表  $e$  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 { 2.71828182845904523536 };
const int HoursPerDay { 24 };
const int SecondsPerHour { 3'600 };
```

#### 注意：

从 C++20 开始，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` 名称空间的 `<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，没有使用\*和&等符号。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。这种思维方式是错误的。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？在现代 C++ 中，处理内存所有权和转让所有权的较好方法是使用第 7 章介绍的智能指针。

### 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以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，一些做法只是品味问题，但是另一些做法会难以进行团队合作。

### 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(整本书都是如此)：

```
void someFunction()
{
    if (condition())
        cout << "condition was true" << endl;
    } else {
        cout << "condition was false" << endl;
    }
}
```

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())
    {
        cout << "condition was true" << endl;
    }
    else
    {
        cout << "condition was false" << endl;
    }
}
```

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())
{
    cout << "condition was true" << endl;
}
else
{
    cout << "condition was false" << endl;
}
}

```

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())
        cout << "condition was true" << endl;
    else
        cout << "condition was false" << endl;
}

```

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，并使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致非常困难，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规定。

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，然后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()`。这是该方法的实现，并带有一些注释：

```
void Database::displayCurrent() const           // The displayCurrent() method
{
    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 个方法。向此代码片段添加适当的 JavaDoc 风格注释。请参阅第 1 章以了解这些方法的具体作用。

```
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 OCurrent();

    // 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." };
}
```



## 第II部分

# 专业的 C++ 软件设计

---

- ▶ 第 4 章 设计专业的 C++ 程序
- ▶ 第 5 章 面向对象设计
- ▶ 第 6 章 设计可重用代码



# 第4章

## 设计专业的 C++ 程序

### 本章内容

- 程序设计的定义
- 程序设计的重要性
- C++程序设计的特点
- 高效 C++程序设计的两个基本主题：抽象和重用
- 不同类型的重用代码
- 代码重用的优缺点
- 选择重用代码库的指导原则
- 开放源代码库
- C++标准库

在编写应用程序代码之前，应该设计程序。使用什么数据结构？编写哪些类？在团队中编程时，规划尤其重要。设想这样的情形：你坐下来编写程序，却不知道还有哪些同事在编写同一程序，这就需要规划！本章将学习如何利用专业的 C++ 方法进行 C++ 设计。

尽管设计很重要，但它可能是受误解最多的、最没有被充分利用的软件工程过程。程序员经常没有清晰地规划好，就一头扎入应用程序：他们一边编写代码，一边设计。这种方法可能导致令人费解的、极度复杂的设计，开发、调试和维护任务也更加困难。

尽管这似乎违反直觉，但在项目开始时投入额外的时间来正确设计实际上可以在项目的整个生命周期中节省时间。

### 4.1 程序设计概述

在启动新程序(或已有程序的新功能)时，第一步是分析需求，包括与利益相关方(stakeholder)进行商讨。分析阶段的重要输出是“功能需求”文档，描述新代码段到底要做什么，但它不解释如何做。需求分析后，也会得到“非功能需求”文档，其中描述最终系统是什么样的，而非应该做什么。非功能需求的例子有：系统必须是安全的、可扩展的，还能满足特定性能标准等。

收集了所有需求后，将可以启动项目的设计阶段。程序设计(或软件设计)是为了满足程序的所有

功能和非功能需求而实现的结构规范。非正式地讲，设计就是规划如何编写程序。通常应该以设计文档的形式写出设计。尽管每个公司或项目都有自己的设计文档格式，但大多数设计文档的常见布局基本类似，包括两个主要部分：

(1) 将程序粗略地分为子系统，包括子系统之间的接口和依赖关系、子系统之间的数据流、每个子系统的输入输出和通用线程模型。

(2) 每个子系统的详情，包括类的细分、类的层次结构、数据结构、算法、具体的线程模型和错误处理的细节。

设计文档通常包括图和表格，以显示子系统交互关系和类层次结构。统一建模语言(UML)是图的行业标准，用于绘制此类图以及后续章节中介绍的图(可参见附录 D 来简单了解 UML 语法)。也就是说，设计文档的精确格式不如设计思考过程重要。

#### 注意：

设计的关键是在编写程序之前进行思考。

通常在编写代码之前，应该尽可能使设计趋于完善。设计应该提供一个方案图，任何理智的程序员都能够遵循它，实现应用程序。当然，一旦开始编写代码，如果遇到以前没有想到的问题，修改原有设计是不可避免的。软件工程过程可灵活地进行这种修改。敏捷软件开发方法 Scrum 是此类迭代过程的一个示例，根据这个方法以循环方式(称为 sprint)开发应用程序。在每个 sprint 中，都可以修改设计，并考虑新需求。第 28 章将详细讲述各种软件工程过程模型。

## 4.2 程序设计的重要性

为尽快开始编程，很容易忽略分析和设计阶段，或者只是草率地进行设计。这一点儿也不像看着代码编译并运行时那种工作取得了进展的感觉。当或多或少地了解如何构建程序时，完成正式的设计或写下功能需求似乎是在浪费时间。此外，编写设计文档并不像编写代码那么有趣。如果打算整天编写文档，就好像不是当程序员！笔者自己也是程序员，我理解在一开始就编写代码的诱惑，有时也会这么做。但是，这种做法极可能出问题，除非项目非常简单。如果在实现之前没有进行设计，能否成功取决于编程经验、对常用设计模式的精通程度，还取决于对 C++、问题域和需求的理解程度。

如果你所在团队的每个成员都处理项目的不同部分，那么必须编写一个所有团队成员都遵循的设计文档。设计文档可以帮助新手快速了解项目的设计。

一些公司有专门的功能分析师来编写功能需求和专门的软件架构师来制定软件设计。在这些公司中，开发人员通常可以只专注于项目的编程方面。在其他公司，开发人员必须自己进行需求收集和设计。有些公司介于这两个极端之间，也许他们的软件架构师负责做出更大的架构决策，而开发人员自己做更小的设计。

为理解程序设计的重要性，请想象要在一小块土地上修建一栋住宅。施工人员来了后，你要求看设计图，“什么设计图？”他回答，“我知道我在做什么，我不需要提前规划每个细节。两层的住宅？没问题——我几个月前刚建了一栋一层的住宅——我就用这个模型开始工作。”

假定你放下心头的疑惑，允许施工人员开始修建。几个月后，你发现管道露在墙外，而不是在里面。你向施工人员询问这一异常现象时，他说，“噢，我忘了在墙体中给管道预留位置了。使用新的石膏板技术让我太激动，我忘了这回事。但是管道在外面也能正常工作，功能才是最重要的。”你开始怀疑他的做法，但是仍然允许他继续修建，而不是做出更好的决定。

当你第一次查看已竣工的建筑时，发现厨房少了一个水池。施工人员道歉说：“当厨房完工三分

之二时，我们意识到没有地方放水池了，我们在隔壁添加了一间独立水池房，而不是从头开始，这个水池也能用，对吧？”

如果将施工人员的借口放到软件行业，听着熟悉吗？就像将管道放到住宅外面一样，你在解决问题时使用过“丑陋”方案吗？例如，你忘了对多个线程共享的队列数据结构加锁。当意识到这个问题时，你决定在用到队列的所有地方手动加锁。你觉得：“虽然这很丑陋，但并不影响运行。”确实如此，但是某个新加入项目的员工可能假定这个数据结构内建有锁，没有确保在访问共享数据时实现互斥，导致竞态条件错误，三个星期之后才找到这个问题。当然，加锁问题只是丑陋解决方法的一个示例。显然，专业的C++程序员永远不会在每个队列访问中手动加锁，而是在队列类中直接加锁，或者让队列类在不使用锁的情况下以线程安全方式运行。

在编写代码之前进行规范的设计，有助于判断如何令所有内容互相适应。住宅的设计图可以显示房间之间的关系和结合方式，以满足住宅的要求，与此类似，程序的设计也显示了程序子系统之间的关系和配合方式，以满足软件的需求。如果没有设计规划，可能会漏掉子系统之间的联系、重用或共享信息，以及失去用最简单的方法完成任务的可能性。如果没有“设计宏图”，可能会在某个实现细节上钻牛角尖，以至于忘记整体结构和目标。此外，设计可提供编写好的文档，供所有项目成员参考。如果使用敏捷Scrum方法这样的迭代流程，则需要确保在流程的每个周期中使设计文档保持最新，只要这样做有价值。敏捷方法的支柱之一是更喜欢“软件开发胜过详细文档”。至少应该维护关于项目较大部分如何协同工作的设计文档，而在我看来，维护有关项目较小方面的设计文档是否对未来有任何价值取决于团队。如果没有，请确保删除此类文档或将该标记为过期。

如果前面的类比还没有让你下决心在编写代码前进行设计，这里有一个直接编写代码，导致优化失败的示例。假定编写一个国际象棋程序，你不想在编程前设计整个程序，而是决定从最简单的部分开始，缓慢过渡到较难的部分。按照面向对象的方法(参见第1章，详见第5章)，你决定用类建立棋子模型。兵是最小的棋子，因此选择从这里开始。在考虑兵的特性和行为之后，编写了一个具有一些属性和方法的类，图4-1显示了UML类图。

在这个设计中，`m_color`特性指定了棋子是黑色还是白色。在到达对方棋盘时执行`promote()`方法。

当然你不会绘制这个类图，而是直接实现这个类。这个类编写得很轻松，然后开始编写下一个非常容易的棋子：象。在考虑它的特性和功能后，编写了一个具有一些属性和方法的类，如图4-2所示。

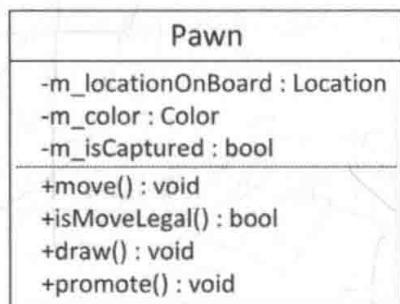


图4-1 UML类图

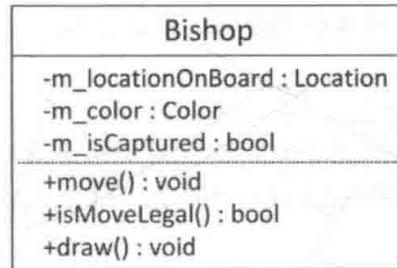


图4-2 棋子象

由于直接编写代码，因此没有生成类图。然而，此时你开始怀疑是不是做错了什么。象和兵有些相似，实际上，它们的属性完全相同，并且有很多共同的行为。尽管象和兵的移动行为有不同的实现方式，但两个棋子都需要移动。如果在编写代码之前进行了设计，就会意识到不同的棋子实际上非常相似，并找到一次性编写通用功能的方法。第5章讲述与此相关的面向对象设计技术。

此外，象棋棋子的某些特征取决于程序的其他子系统。例如，如果不知道棋盘的模型，在棋子类中就无法准确表示棋子的位置。另外也可能这样设计程序，让棋盘以某种方式管理棋子，这样棋子就不需要知道自身的位置。无论是哪种情况，在设计棋盘之前就在棋子类中编写位置代码都会出问题。

此外，如果不给出程序的用户界面，如何编写棋子的绘制方法？是使用图形还是基于文本呢？棋盘应该是什么样子的？问题在于，程序的子系统并不是孤立存在的——子系统彼此有联系。大部分设计工作都用来判断和定义这些联系。

## 4.3 C++设计

在使用 C++ 进行设计时，需要考虑 C++ 语言的一些性质：

- C++ 具有庞大的功能集。它几乎是 C 语言的完整超集，此外还有类、对象、运算符重载、异常、模板和其他功能。由于该语言非常庞大，使设计成为一项令人生畏的任务。
- C++ 是一门面向对象语言。这意味着设计应该包含类层次结构、类接口和对象交互。这种设计类型与传统的 C 和其他过程式语言的设计完全不同。第 5 章重点介绍 C++ 面向对象设计。
- C++ 有许多通用设计的、可重用代码的功能。除了基本的类和继承之外，还可以使用其他语言功能进行高效的设计，如模板和运算符重载。第 6 章将详细讨论可重用代码的设计技术。
- C++ 提供了一个有用的标准库，包含字符串类、I/O 工具、许多常见的数据结构和算法。所有这些都便于 C++ 代码的编写。
- C++ 语言提供了许多设计模式或解决问题的通用方法。

考虑到以上问题，完成 C++ 程序的设计并非易事。笔者曾经花费数天时间在纸上勾画设计想法，然后把它们擦掉，写出更多想法，又擦掉，如此反复。有时这个过程是有效的，几天或几星期后会得到整洁高效的设计。有时这个过程令人沮丧，得不到任何结果，但不是一无所获。如果必须再次实现已经出错的设计，很可能浪费更多的时间。重要的是清楚地认识到是否取得真正意义上的进展。如果发现无法继续，可采用以下方法：

- **寻求帮助。**请教同事、顾问，或者查阅书本、新闻组或 Web 页面。
- **做一会儿别的事情。**稍后回来进行设计。
- **做出决定并继续前进。**即使这并非理想方案，也要做出决定并用来工作。不适当的选择会很快表现出来。然而，它可能变成一个可以接受的方法，因为或许没有清晰的方法能完成想要的设计。如果某个方案是唯一满足要求的可行策略，有时不得不接受这个“丑陋的”方案。无论做了什么决定，都应该用文档记录这个决定，这样自己或其他人在以后就可以知道为什么做这样的决定。这包括记录被拒绝的设计，并记录拒绝的原因。

### 注意：

记住，优秀的设计难能可贵，获取这样的设计需要实践。不要期望一夜之间成长为专家，当你发现掌握 C++ 设计比 C++ 编码更难时，不要惊讶。

## 4.4 C++设计的两个原则

C++ 设计有两个基本的原则：抽象和重用。这些指导方针非常重要，甚至可以认为是本书的主题。这两个原则贯穿全书，贯穿于高效 C++ 程序设计的所有领域。

### 4.4.1 抽象

与现实事物进行类比，将最便于理解“抽象”原则。电视是一种大多数家庭都有的简单科技产品。读者很熟悉其功能：可将其打开或关闭、调换频道、调节音量，还可以添加附属组件，如扬声器、数

字视频录像机和蓝光播放器。然而，你能解释这个黑盒子的工作原理吗？也就是说，知道电视机如何从空中或电缆中接收信号、转换信号并在屏幕上显示吗？大多数人肯定解释不了电视机的工作原理，但可以使用它。这是由于电视机明确地将内部的实现与外部的接口分离开来。我们通过接口与电视机进行交互：开关、频道变换器和音量控制器。我们不知道也不关心电视机的工作原理，我们不关心它是使用了阴极射线管技术还是其他技术在屏幕上生成图像，这无关紧要，因为不会影响接口。

## 1. 抽象的作用

在软件中也有类似的抽象原则。可使用代码而不必了解底层的实现。在此有一个简单示例，程序可调用在<cmath>头文件中声明的 sqrt() 函数，而不需要知道这个函数使用什么算法求平方根。实际上，平方根计算的底层实现可能因库版本而异；但只要接口不变，函数调用就可以照常运行。抽象原则也可以扩展到类。第 1 章已经介绍过，可使用 ostream 类的 cout 对象将数据传输到标准输出：

```
cout << "This call will display this line of text" << endl;
```

在这行代码中，使用 cout 插入运算符(<< )的已经编写好的接口输出了一个字符数组。然而，不需要知道 cout 如何将文本输出到用户屏幕，只需要了解公有接口。cout 的底层实现可随意改动，只要公开的行为和接口保持不变即可。

## 2. 在设计中使用抽象

应该设计函数和类，使自己和其他程序员可以使用它们，而不需要知道(或依赖)底层的实现。为说明暴露在实现之外和隐藏在接口之后设计的不同，再次考虑前面的国际象棋程序。假定使用一个指向 ChessPiece 对象的二维指针数组实现场棋的棋盘。可以这样声明并使用棋盘：

```
ChessPiece* chessBoard[8][8]{ }; // Zero-initialized array.  
...  
chessBoard[0][0] = new Rook{};
```

然而，这种方法没有用到抽象概念。每个使用象棋棋盘的程序员都知道这是一个二维数组。将该实现转换为其他类型(如一维矢量数组，大小为 64)比较难，因为需要改变整个程序中每一处用到棋盘的代码。棋盘的每个使用者也必须恰当地管理内存。在此没有将实现与接口分开。

更好的方法是将象棋棋盘建立为类。这样就可以暴露接口，并隐藏底层的实现细节。下面给出 ChessBoard 类的示例：

```
class ChessBoard  
{  
public:  
    void setPieceAt(size_t x, size_t y, ChessPiece* piece);  
    ChessPiece* getPieceAt(size_t x, size_t y) const;  
    bool isEmpty(size_t x, size_t y) const;  
private:  
    // Private implementation details...  
};
```

注意，该接口并不决定底层实现方式。ChessBoard 可以是一个二维数组，但是接口对此并没有要求。改变实现并不需要改变接口。此外，这个实现还可提供更多功能，如边界检测。

从这个示例可以了解到，抽象是 C++ 程序设计中的重要技术。第 5 章将详细讲述面向对象设计，第 6 章将深入讨论抽象原则，第 8、9、10 章讲述与编写自己的类有关的所有细节。

## 4.4.2 重用

C++设计的第二个基本原则是重用。用现实世界做类比同样有助于理解这个概念。假定你放弃了编程生涯，而选择自己更喜欢的面包师工作。第一天，面包师主管让你烤饼干。为完成任务，你找到了巧克力饼干的配方，混合原料，在饼干盘上把饼干成型，并将盘子放入烤箱。面包房主管对结果感到十分满意。

现在，很明显，你没有自己做一个烤箱来烘烤饼干，也没有亲自制作黄油、磨制面粉、制作巧克力片。你可能觉得这匪夷所思：“这还用做？”如果你真的是一位厨师，当然是这样；但如果你是一位编写烘焙模拟游戏的程序员，又会怎样？在此情况下，你不希望编写程序的全部组件，从巧克力片到烤箱；而是查找可重用的代码以节约时间，或许同事编写了一个烹饪模拟程序，其中有很好的烤箱代码。或许这些代码并不能完成你需要的所有操作，但你可以修改这些代码，并添加必要的功能。

另一件你认为理所当然的事情是，你采用饼干的某个配方而不是自己做一个配方，这也是不言而喻的。然而，在 C++ 编程中并非如此。尽管在 C++ 中不断涌现处理问题的标准方法，但许多程序员仍然在每个设计中无谓地重造这些策略。

使用已有代码的思想并非首次出现。使用 `cout` 输出，就已经在重用代码了。你并没有编写将数据输出到屏幕的代码，而使用已有的 `cout` 实现完成这项任务。

遗憾的是，并非所有程序员都利用已有的代码。设计时应该考虑已有的代码，并在适当时重用它们。

### 1. 编写可重用的代码

重用的设计思想适用于自己编写和使用的代码。应该设计程序，以重用类、算法和数据结构。自己和同事应能在当前项目和今后的项目中重用这些组件。通常，应该避免设计只适用于当前情况的特定代码。

在 C++ 中，模板是一种编写多用途代码的语言技术。考虑编写一个可用于任何类型的二维棋盘游戏(例如国际象棋或西洋跳棋)的泛型 `GameBoard` 类模板，而不像前面那样编写一个存储 `ChessPiece` 的特定 `ChessBoard` 类。只需要修改类的声明，在接口中将棋子当作模板参数 `PieceType` 而不是固定类型。这个模板如下所示，如果在此之前没有见过这种语法，不要着急！第 12 章将深入讲解这一语法。

```
template <typename PieceType>
class GameBoard
{
public:
    void setPieceAt(size_t x, size_t y, PieceType* piece);
    PieceType* getPieceAt(size_t x, size_t y) const;
    bool isEmpty(size_t x, size_t y) const;
private:
    // Private implementation details...
};
```

在接口中完成如上简单修改后，现在有了一个可用于任何二维棋盘游戏的泛型游戏棋盘类。尽管代码的变动很简单，但在设计阶段做这样的决定非常重要，以便能有效且高效地实现代码。

第 6 章将讲述设计可重用代码的更多细节。

### 2. 重用设计

学习 C++ 语言与成为优秀的 C++ 程序员是两码事。如果你坐下来，阅读 C++ 标准，记住每个事实，那么你对 C++ 的了解程度将与其他人差不多。但只有分析代码，并编写自己的程序，积累了一定

经验后，才可能成为优秀的程序员。原因在于，C++语法以原始形式定义了该语言的作用，但并未指定每项功能的使用方式。

如面包师示例所示，重新发明每道菜的配方是可笑的。然而，程序员在设计期间却常犯类似的错误。他们不是使用已有的“配方”或模式，而是在每次设计程序时都重造这些技术。

随着C++语言使用经验的增加，C++程序员自己总结出使用该语言功能的方式。C++社区通常已经构建起利用该语言的标准方式，一些方式是正规的，一些则不正规。本书将指出该语言的可重用模式，称为设计技术或设计模式。另外，第32章和第33章将专门讲解设计技术和模式。你可能已经熟悉其中的一些模式，这些只是平日里司空见惯的解决方案的正式化产物。其他方案描述你在过去遇到的问题的新解决方案。还有一些则以全新方式思考程序的结构。

例如，假定要设计一个国际象棋程序：使用一个ErrorLogger对象将不同组件发生的所有错误都按顺序写入一个日志文件。当试着设计ErrorLogger类时，你意识到只想在一个程序中有一个ErrorLogger实例。还要使程序的多个组件都能使用这个ErrorLogger实例，即所有组件都想要使用同一个ErrorLogger服务。实现此类服务机制的一个标准策略是使用依赖注入(dependency injection)。使用依赖注入时，为每个服务创建一个接口，并将组件需要的接口注入组件。因此，此时良好的设计应当使用“依赖注入”模式。

你必须熟悉这些模式和技术，根据特定设计问题选择正确的解决方案。在C++中，还可以使用更多技术和模式。详细讲述设计模式和技术超出了本书的范围，如果读者想要一本关于更多不同设计模式的参考书籍，可以参阅附录B给出的建议。

## 4.5 重用现有代码

经验丰富的C++程序员绝不会完全从零开始启动一个项目。他们会利用各种资源提供的代码，如标准模板库、开放源代码库、他们公司拥有的专用代码和以前项目的代码。应该在项目中自由地重用代码。为了充分利用这条规则，本节首先解释可以重用的不同类型的代码，然后是重用现有代码和自己编写代码之间的权衡。一旦你决定不自己编写代码而是重用现有代码，本节的最后一部分将讨论一些选择要重用的库的指南。

### 注意：

重用代码并不意味着复制和粘贴现有代码！事实上，它的意思恰恰相反：重用代码而不复制它。

### 4.5.1 关于术语的说明

在分析代码重用的优缺点之前，有必要指出涉及的术语，并将重用代码分类。有3种可以重用的代码：

- 过去编写的代码
- 同事编写的代码
- 当前组织或公司以外的第三方编写的代码

所重用的代码可通过以下几种方式构建：

- **独立的函数或类** 当重用自己或同事的代码时，通常会遇到这种类型。
- **库** 库是用于完成特定任务(例如解析XML)或者针对特定领域(如密码系统)的代码集合。在库中经常可以找到其他许多功能，如线程和同步支持、网络和图像。

- **框架(Framework)** 框架是代码的集合，围绕框架设计程序。例如，微软基础类(Microsoft Foundation Classes, MFC)提供了在 Microsoft Windows 中创建图形用户界面应用程序的框架。框架通常指定了程序的结构。

**注意：**

程序使用库，但会适应框架。库提供了特定功能，而框架是程序设计和结构的基础。

应用程序编程接口(API)是另一个经常出现的术语。API 是库或代码为特定目的提供的接口。例如，程序员经常会提到套接字 API，这指的是套接字联网库的公开接口，而不是库本身。

**注意：**

尽管人们将库以及 API 互换使用，但二者并不是等价的。库指的是“实现”，而 API 指的是“库的公开接口”。

为简洁起见，本章剩余部分用术语“库”表示任何可重用的代码，它实际上可能是库、框架、完整的应用程序，或是同事编写的函数的随机集合。

#### 4.5.2 决定是否重用代码

重用代码的原则在抽象上很容易理解。但当涉及细节时，这一原则就会有些模糊。什么时候适合重用代码？重用哪些代码？要根据具体情况做出决定，并要权衡利弊。当然，重用代码的利弊还是有一般性规律的。

##### 1. 重用代码的优点

重用代码可以给程序员和项目带来极大好处：

- 你可能不知道如何编写所需的代码，或者抽不出时间来编写代码。的确要编写处理格式化输入输出的代码吗？当然不需要，这正是使用标准 C++ I/O 流的原因。
- 重用的应用程序组件不需要设计，从而简化了设计。
- 重用的代码通常不需要调试。一般可以认为库代码是没有 bug 的，因为它们已通过测试，并得到广泛使用。
- 相对于初次编写的代码，库可以处理更多的错误情况。在项目开始时或许会忘记隐藏的错误或边缘情况，以后则要花时间修正这些问题。重用的库代码通常经过广泛的测试，之前已经被许多程序员使用过，因此可认为它能正确处理大多数错误。
- 库通常在具有不同硬件、不同操作系统和操作系统版本、不同显卡等的各种平台上进行过测试，比自己可以用来测试的平台要多得多。有时，库包含使它们在特定平台上工作的变通方法。
- 库通常可以检测用户的错误输入。如果发现对于当前状况无效或不适当的请求，通常会给出正确的错误提示。例如，如果请求查找数据库中不存在的记录，或者在没有打开的数据库中读取记录，库都会采取得当的措施。
- 由某个领域的专家编写的重用代码比自己为这个领域编写的代码安全。例如，如果不是安全领域的专家，就不应该试着编写安全代码。如果程序需要安全代码或加密代码，就应该使用库。这种性质的代码中的许多看似次要的细节可能会危及整个程序的安全性，甚至会影响整个系统。

- 库代码会持续改进。如果重用这些代码，不需要自己动手就可以享受这些改进带来的好处。实际上，如果库的作者恰当地将接口和实现分开，通过升级库版本就可以享受到这些好处，并不需要改变与库的交互方式。良好的更新会修改底层的实现，而不会修改接口。

## 2. 重用代码的缺点

遗憾的是，重用代码也有一些缺点：

- 当只使用自己编写的代码时，能完全理解代码的运行方式。当使用并非由自己编写的库时，在使用之前必须花时间理解接口和正确的用法。在项目开始时，这些额外的时间会拖延初始的设计和编码。
- 当编写自己的代码时，代码的功能正是所需要的。库代码提供的功能与需要的功能未必完全吻合。
- 即使库代码提供的功能正是所需要的，其性能也未必符合要求。一般来说，库代码的性能可能不太好，不太适用于特定场合，甚至所有场合。
- 使用库代码就像打开支持问题的潘多拉魔盒。如果发现库中存在 bug，该怎么办？通常无法获取源代码，因此即使想修正这个问题也没办法。如果已经花费大量的时间学习这个库的接口并使用这个库，可能不想放弃，但很难说服库的开发人员按你的时间安排修正这个 bug。此外，如果使用第三方的库，但库的开发人员停止对这个库的支持，而产品依赖这个库，该怎么办？在决定使用某个无法获取源代码的库之前，应该仔细考虑这个问题。
- 除支持问题外，库还涉及许可协议问题，涉及的问题包括公开源代码、再分发费用(通常称为二进制许可协议费用)、版权归属和开发许可协议。在使用任何库之前都应该仔细检查许可协议问题。例如，某些开源库要求你也公开源代码。
- 重用代码要求可靠的供应商，必须信赖编写代码的人，认为他的工作非常出色。某些人喜欢控制项目的方方面面，包括每一行源代码。
- 库版本的升级可能会引发问题。升级可能引入 bug，让产品出现致命问题。与性能有关的升级在某些情况下可能会优化性能，但是在特定的情况下可能会使性能恶化。
- 使用纯粹的二进制库时，将编译器升级为新版本会导致问题。只有当库供应商提供与你的新编译器版本兼容的二进制库时，你才能升级编译器。

## 3. 综合考虑做出决定

熟悉了重用代码的术语和优缺点后，就可以决定是否重用代码。通常，这个决定是显而易见的。例如，如果想要用 C++ 在 Microsoft Windows 上编写图形用户界面(GUI)，应该使用 MFC(Microsoft Foundation Class)或 Qt 等框架。你可能不知道如何在 Windows 上编写创建 GUI 的底层代码，更重要的是不想浪费时间去学习。在此情况下使用框架可以节省数人年的时间。

然而，有时情况并不明显。例如，如果不熟悉某个库或框架，并且只需要其中某个简单的数据结构，那就不值得花时间去学习整个框架来重用某个只需要花费数天就能编写出来的组件。

总之，这个决定是根据特定的需求做出的选择。通常是在自己编写代码所花的时间和查找库并了解如何使用库来解决问题所使用时间之间的权衡。应该针对具体情况，仔细考虑前面列出的优缺点，并判断哪些因素是最重要的。最后，可随时改变想法，如果正确处理了抽象，这并不需要太多工作量。

### 4.5.3 重用代码的指导原则

当使用库、框架以及同事或自己的代码时，应该记住一些指导方针。

### 1. 理解功能和限制因素

花点时间熟悉代码，对于理解其功能和限制因素而言都很重要。可从文档、公开的接口或 API 开始，理想情况下，这样做足以理解代码的使用方式。然而，如果库未将接口和实现明确分离，可能还要研究源代码。此外，还可与其他使用过这些代码或能解释这些代码的程序员交流。首先应该理解基本功能。如果是库，那么该库可提供哪些行为？如果是框架，代码如何适应这个框架？应该编写哪些类的子类？需要亲自编写哪些代码？还应该根据代码的类型考虑特定的问题。

下面是选择库时应记住的一些要点：

- 对于多线程程序而言，代码安全吗？
- 库是否要求使用它的代码进行特定的编译器设置？如有必要，项目可以接受吗？
- 库依赖于其他哪些库？

此外，对于特定的库，你可能需要做一些更详细的研究：

- 需要什么样的初始化和清理调用？
- 如果从某个类继承，应该调用哪个构造函数？应该重写哪些虚方法？
- 如果某个调用返回内存指针，调用者还是库负责内存的释放？如果库对此负责，什么时候释放内存？强烈建议查看是否可使用智能指针(见第 7 章)来管理由库分配的内存。
- 某个调用的全部返回值(按值或按引用)有哪些？
- 所有可能抛出的异常有哪些？
- 库调用检查哪些错误情况？此时做出了什么假定？如何处理错误？如何提醒客户端程序发生了错误？应该避免使用弹出消息框、将消息传递到 stderr/cerr 或 stdout/cout 以及终止程序的库。

### 2. 理解学习成本

学习成本是开发人员学习如何使用库所花费的时间。这不仅仅是开始使用库时的初始成本，而是随着时间的推移而产生的经常性成本。每当新的团队成员加入项目时，都需要学习如何使用该库。

对于某些库而言，此成本可能很高。因此，如果在知名库中找到所需的功能，我建议使用该库，而不是一些奇特的、鲜为人知的库。例如，如果标准库提供了需要的数据结构或算法，请使用标准库而不是另一个库。

### 3. 理解性能

了解库或其他代码提供的性能保障很重要。即使某个程序对性能不敏感，也应该确保使用的代码在具体的使用中性能不会太糟。

#### 大 O 表示法

程序员经常使用大 O 表示法(Big-O Notation)讨论并记录算法和库的性能。本节阐述算法复杂度分析的一般概念和大 O 表示法，而不涉及不必要的数学知识。如果已经熟悉这些概念，可跳过本节。

大 O 表示法表示相对性能而不是绝对性能。例如，大 O 表示法不会指出某个算法运行需要的时间(如 300 毫秒)，而是指出当输入规模增加时算法如何执行。排序算法需要排序的项数、执行键查找时哈希表中的元素数目、磁盘之间复制文件的大小都是输入规模的例子。

#### 注意：

注意大 O 表示法仅适用于速度依赖于输入的算法，不适用于没有输入或者运行时间随机的算法。实际上，大多数算法的运行时间都取决于输入，因此这个限制并不重要。

更正式的提法：大O表示法将算法的运行时间表示为输入规模(也就是算法的复杂度)的函数。实际上并没有这么复杂，例如，假定某个排序算法对400个元素排序需要2毫秒，对800个元素排序需要4毫秒。由于元素数量增加一倍时，排序时间也增加一倍，因此其性能是输入的线性函数，如图4-3所示。也就是说，可用直线表示输入规模与性能的关系。

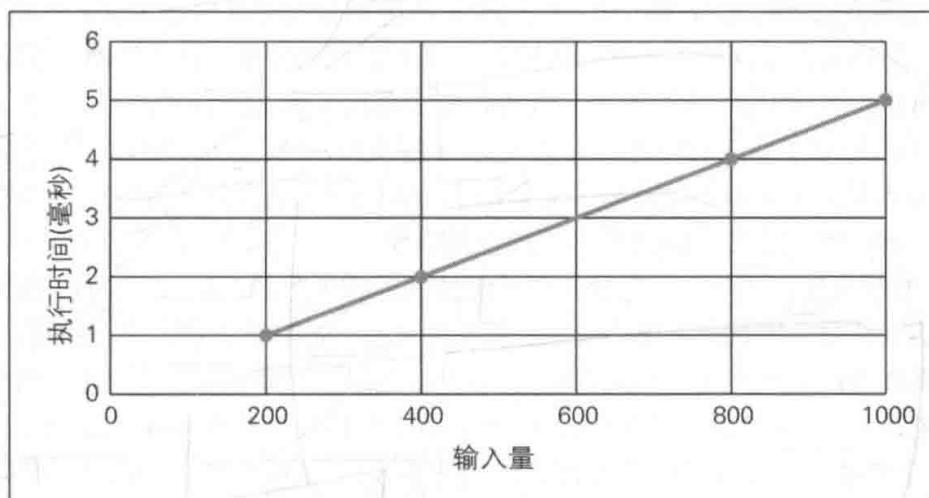


图4-3 输入规模与性能的关系

大O表示法用 $O(n)$ 表示这个排序算法的性能。 $O$ 意味着使用大O表示法， $n$ 表示输入规模。 $O(n)$ 表示排序算法的速度是输入规模的直接线性函数。

当然，并非所有算法的性能与输入规模的关系都是线性的。表4-1总结了常见的函数类型，按照性能从好到差的顺序排列。

表4-1 函数类型的性能排序

算法复杂度	大O表示法	说明	示例算法
常数	$O(1)$	运行时间与输入规模无关	访问数组中的某个元素
对数	$O(\log n)$	运行时间是输入规模以2为底的对数的函数	使用二分法查找有序列表中的元素
线性	$O(n)$	运行时间与输入规模成正比	在未排序列表中查找元素
线性对数	$O(n \log n)$	运行时间是输入规模的对数函数的线性倍数函数	归并排序
二次方	$O(n^2)$	运行时间是输入规模的平方函数	较慢的排序算法，如选择排序法
指数	$O(2^n)$	运行时间是输入规模的指数函数	优化的旅行商问题

用输入规模的函数(而不是绝对数字)表示性能有两个好处：

(1) 这是独立于平台的。在某台计算机上运行一段代码需要200毫秒，在另一台计算机上未必就是这个速度。如果不在同一台计算机上加载完全相同的负荷，很难比较两个不同算法。另一方面，将性能表示为输入规模的函数适用于任何平台。

(2) 用输入规模的函数表示性能时，可用一种方法表示算法所有可能的输入。如果用毫秒表示算法所需的特定时间，那么这个时间只针对某个特定的输入，对另一种输入而言毫无意义。

### 理解性能的几点提示

熟悉了大O表示法，就可以理解大多数性能记录。C++标准库特意用大O表示法描述算法和数

据结构的性能。然而，大  $O$  表示法有时表达不充分，甚至引起误解。当使用大  $O$  表示法表示性能时，要考虑以下问题：

- 当数据量加倍时，算法所需要的时间也加倍，这根本就没有体现需要多长时间！如果某个糟糕的算法具有较好的复杂度，这仍然不符合你的要求。例如，如果算法进行了不必要的磁盘访问，可能不会影响大  $O$  表示法，但性能非常糟糕。
- 按照这一思路，很难比较两个具有相同大  $O$  运行时间的算法。例如，两个不同的排序算法都声称  $O(n \log n)$ ，如果不进行测试，很难说哪个算法实际上更快些。
- 大  $O$  表示法描述了算法的渐进时间复杂度，因为输入规模会无限增大。对于小规模输入，大  $O$  时间很容易引起误解。当输入规模不大时， $O(n^2)$  算法的实际执行性能可能要优于  $O(\log n)$  算法。在做出决定之前应该考虑可能的输入规模。

除了考虑算法的大  $O$  特性外，还需要考虑算法性能的其他方面。应该记住下面的指导方针：

- 应该考虑某段特定库代码的使用频率。有人发现了“90/10”法则：大多数程序 90% 的运行时间都花费在 10% 的代码上。如果打算使用的库代码是那 10% 的常用代码，就应该仔细分析代码的性能。相反，如果是运行时间常可忽略不计的剩余 90% 的代码，就不需要花费太多的时间分析性能，因为这对程序整体性能的提高不会有太大帮助。第 29 章将介绍有助于查找代码中的性能瓶颈的分析器和工具。
- 不要信任文档。一定要运行性能测试，以判断库代码是否提供了可接受的性能。

#### 4. 理解平台限制

在开始使用库代码之前，一定要理解运行库的平台。如果想编写跨平台应用程序，请确保选择的库也是跨平台可移植的。这看上去是显而易见的，但即使是那些号称跨平台的库，在不同的平台上也会有微妙差别。

此外，平台不仅包括不同的操作系统，还包括同一操作系统的不同版本。如果想编写一个在 Solaris 8/9/10 上运行的应用程序，就要确保所使用的任何库都支持以上版本。不能假定操作系统的不同版本会向前兼容或向后兼容。也就是说，某个库能在 Solaris 9 上运行，并不意味着可在 Solaris 10 上运行，反之亦然。

#### 5. 理解许可协议

使用第三方的库常会带来复杂的许可协议问题。为使用第三方供应商提供的库，有时必须支付许可协议费用。还可能有其他的许可协议限制，包括出口限制。此外，开源库有时会要求与其有关的任何代码都公开源代码。本章后面会讨论开源库常用的许多许可协议。

##### 警告：

如果打算分发或销售自己开发的代码，一定要理解所用的任何第三方库的许可协议限制。有疑问时，应该请教知识产权方面的法律专家。

#### 6. 了解支持以及在哪里寻求帮助

使用第三方库还带来了支持问题。在使用某个库之前，一定要理解提交 bug 的过程，并了解修正 bug 所需的时间。如果可能，判断这个库会被支持多长时间，这样就可以相应地制定计划。

有趣的是，即使使用组织内部的库也会带来支持问题。与公司其他部门的同事交流以修正库中 bug 的难度，跟与其他公司的陌生人交流以解决同样问题的难度差不多。实际上可能会更难，因为你并非花了钱的顾客。在使用内部库之前，一定要理解组织内部的政策和组织性问题。

为了重用整个应用程序，支持问题甚至可能变得更加复杂。如果客户遇到捆绑 Web 服务器的问题，他们应该联系你还是 Web 服务器供应商？确保在发布软件之前解决此问题。

开始使用库或框架时可能令人畏惧。幸运的是，可用的支持方法很多。首先参考库自带的文档。如果库被广泛使用，如标准库或 MFC，就应该能找到与此主题相关的优秀书籍。实际上，本书的第 16~25 章都讲述标准库。如果某个特定问题在手册或产品文档中没有提及，可搜索 Web。在选择的搜索引擎中输入问题来寻找讨论这个库的 Web 页面。例如，当查找短语“introduction to C++ Standard Library”时，会找到与 C++ 和标准库有关的数百个站点。此外，许多站点包含关于特定主题的新闻组或论坛，可注册并寻找答案。

#### 警告：

不要盲目相信在 Web 上看到的所有内容。Web 页面没有像出版书籍和文档那样的审查程序，因此可能包含错误。

### 7. 原型

当首次使用某个新库或框架时，最好编写一个快速原型。测试代码是熟悉库功能的最好方法。应该考虑在程序设计之前测试库，这样就可以熟悉库的功能和限制。这种实际检验还可判断库的性能特征。

即使原型应用程序与最终应用程序没有任何相似之处，花费在原型上的时间也不会浪费。不要觉得编写实际应用程序的原型很难，可编写一个虚拟程序来测试想使用的库功能，这样做是为了让自己熟悉库。

#### 警告：

由于时间限制，程序员有时会发现他们的原型逐渐变成了最终产品。如果原型不够成熟，不能作为最终产品的基础，那么切勿使用这种方法。

### 8. 开源库

开放源代码库是一种日益流行的可重用代码类型。开放源代码(open-source)通常意味着任何人都可以查看源代码。关于分发软件时包含源代码，有正式的定义和法规，但最重要的是，任何人(包括你)都能查看开放源代码软件的源代码。注意开放源代码不仅适用于库，实际上最著名的开放源代码产品可能是 Android 操作系统。Linux 操作系统是另一个著名的开放源代码操作系统。Google Chrome 和 Mozilla Firefox 是两个开放源代码的著名 Web 浏览器。

#### 开源运动

遗憾的是，开源社区中的术语有些混乱。首先，这个运动有两个互相竞争的名称(有些人说这是两个独立但相似的运动)。Richard Stallman 和 GNU 项目使用术语“自由软件”(free software)。注意术语“自由”并不意味着最终产品必须是免费的。开发人员可以自由决定价格，术语“自由”指可自由查看源代码、修改源代码并重新分发软件。应该将自由(free)理解为“言论自由”中的自由，而不是“啤酒免费”中的免费。可在 [www.gnu.org](http://www.gnu.org) 上阅读与 Richard Stallman 和 GNU 项目有关的更多内容。

开放源代码促进会(Open Source Initiative)使用术语开放源代码软件(open-source software)描述必须公开源代码的软件。与自由软件一样，开放源代码软件并不要求产品或库免费。但开放源代码软件和自由软件之间的区别是，开放源代码软件不需要提供使用、修改和重新分发的自由。在 [www.opensource.org](http://www.opensource.org) 上可找到关于开放源代码促进会的更多内容。

关于开放源代码项目有多种许可协议供选择。其中之一是 GPL(GNU Public License)，然而使用 GPL 下的库要求产品也公开源代码。此外，开源项目可使用的许可协议还有 Boost Software License、BSD(Berkeley Software Distribution)、MIT 许可协议、Apache 许可协议等，这些许可协议允许在闭源产品中使用开源库。一些许可协议有着不用的版本，例如，BSD 许可协议实际上有 4 种版本。开源项目的另一个选择是使用知识共享 (CC) 许可协议的 6 种风格之一。

某些许可协议要求在最终产品中包含库的许可协议。使用库时，某些许可协议需要署名。最重要的是，所有许可协议都带有微妙之处，如果在闭源项目中使用库，了解这些细节很重要。[opensource.org/licenses](http://opensource.org/licenses) 网站全面概述了已批准的开源许可协议。

由于“开放源代码”比“自由软件”的意义更明确，本书使用“开放源代码”说明可获取源代码的产品和库。这一名称的选择并不意味着开放源代码体系优于自由软件体系：这只是为了便于理解。

### 寻找并使用开源库

抛开术语，可从开放源代码软件获得巨大好处。最主要的好处是功能，在此有众多的面向各种任务的开放源代码 C++ 库：从 XML 解析、跨平台的错误日志到使用人工神经网络进行深度学习以及数据挖掘。

尽管并不要求开源库提供免费分发和免费许可协议，仍可免费得到许多开源库。使用开源库通常能够节约许可协议费用。

最后，常常(但并不总是)可以自由修改开放源代码库，以满足特定的需求。

大多数开放源代码库都可在网上找到。例如，查找 open-source C++ library XML parsing 会得到一个用 C 或 C++ 编写的 XML 库链接列表。还可从以下站点寻找开放源代码资源，包括：

- [www.boost.org](http://www.boost.org)
- [www.gnu.org](http://www.gnu.org)
- [github.com/open-source](https://github.com/open-source)
- [www.sourceforge.net](http://www.sourceforge.net)

### 使用开源库的指导方针

开放源代码库带来的问题比较特殊，需要采用新策略。首先，开源库通常是人们在“业余”时间编写的。任何想要添加补丁、继续开发或修正 bug 的程序员都可访问源代码。作为一名编程世界的好公民，如果从开放源代码库获取了利益，那么应该试着对开放源代码项目做出贡献。如果在公司工作，经理或许会反对这种想法，因为这样做不能让公司直接受益。然而，或许可以用非直接的利益(例如提升公司的知名度)说服经理，并从公司获取对开源运动的支持，从而继续从事这项活动。

其次，由于分布式开发的特性和缺少单独的所有权，开源库通常存在支持问题。如果迫切希望修正库中的某个 bug，自己修正通常会比等待其他人修正效率更高。如果真的修正了 bug，一定要把这个修正放到开放源代码库中。有些许可协议甚至要求你必须这么做。即使没有修正 bug，也一定要报告发现的问题，这样其他程序员就不会因为遇到同样的问题而白白浪费时间。

## 9. C++ 标准库

C++ 程序员使用的最重要的库就是 C++ 标准库。顾名思义，这个库是 C++ 标准的一部分，因此任何符合标准的编译器都应该包含这个库。标准库并不是整体式的：它分为几个完全不同的组件，前面已经用过其中的一些。你甚至可能以为这是语言核心的一部分，第 16~25 章将详细讲述标准库。

### C 标准库

由于 C++ 是 C 的超集，因此整个 C 库仍然有效。其功能包括数学函数，例如 abs()、sqrt() 和 pow()，

以及错误处理辅助程序，例如 `assert()` 和 `errno`。此外，C 标准库的一些工具在 C++ 中仍然有效，例如将字符数组作为字符串操作的 `strlen()`、`strcpy()`，以及 C 风格的 I/O 函数，例如 `printf()` 和 `scanf()`。

#### 注意：

C++ 提供了比 C 更好的字符串以及 I/O 支持。尽管 C 风格的字符串和 I/O 例程在 C++ 中仍然有效，但应该避免使用它们，而使用 C++ 字符串(详见第 2 章)和 I/O 流(详见第 13 章)。

第 1 章解释了 C 的头文件名称与 C++ 不同。应该使用 C++ 的名称而不是 C 库名称，因为 C++ 的名称不容易出现名称冲突。例如，如果你需要在 C++ 中使用 C 头文件 `<stdio.h>` 中的功能，推荐使用 `include <cstdio>` 而不是 `<stdio.h>`。有关 C 库的详情以及标准库参考资料，可参阅附录 B。

#### 判断是否使用标准库

设计标准库时优先考虑的是功能、性能和正交性(orthogonality)。使用标准库可获得巨大好处。可回顾在链表或平衡二叉树实现中跟踪指针错误，或者调试不能正确排序的排序算法，如果能够正确使用标准库，几乎不需要再编写这类代码。另一个好处是大多数 C++ 开发人员知道如何使用标准库提供的功能。因此，与使用可能需要大量学习成本的第三方库相比，在项目中使用标准库时，新团队成员将更快地上手。第 16~25 章提供了有关标准库功能的深入信息。

## 4.6 设计一个国际象棋程序

本节通过一个简单的国际象棋程序系统介绍 C++ 程序的设计方法。为提供完整示例，某些步骤用到了后面几章讲述的概念。为了解设计过程的概况，可现在就阅读这个示例，也可以在学完后续章节后返回来学习。

### 4.6.1 需求

在开始设计前，应该弄清楚对于程序功能和性能的需求。理想情况下，这些需求应该是以需求规范(requirements specification)形式给出的文档。国际象棋程序的需求应该包含下列类型的规范，当然实际的需求规范应该比下面的内容更详细，条目更多：

- 程序支持标准的国际象棋规则。
- 程序支持两个玩家。程序不提供具有人工智能的计算机玩家。
- 程序提供基于文本的界面。
  - 程序以纯文本形式提供棋盘和棋子。
  - 玩家通过输入代表位置的数字在棋盘上移动棋子。

需求可保证设计的程序能按用户的期望运行。

### 4.6.2 设计步骤

应按系统的方法设计程序，从一般到特殊。下面的步骤并不一定适用于所有程序，但提供了通用指导方针。在需要时设计应该包含图示和表格。制作图示的行业标准称为 UML(统一建模语言)。有关 UML 的简述，请参阅附录 D。UML 定义了一组标准图示，可用于说明软件设计(如类图、序列图等)。建议使用 UML，至少也要尽量使用类似 UML 的图示。但不一定要严格遵循 UML 语法，因为图示清晰、易于理解，要比语法正确更重要。

### 1. 将程序分割为子系统

设计的第一步是将程序分割为通用功能子系统，并指明子系统之间的接口和交互关系。此时不需要考虑特定的数据结构和算法，甚至不需要考虑类。只是试着感受程序不同部分和它们之间的交互关系。可将子系统列在一张表格中，从而表示子系统的高层行为和功能、子系统展现给其他子系统的接口和这个子系统使用的其他子系统接口。建议国际象棋程序使用模型-视图-控制(MVC)模式将数据存储和数据显示明确分离，MVC 模式建立了如下理念：许多应用程序经常要处理一组数据，处理这些数据上的一个或多个视图，并操作这些数据。在 MVC 中，这组数据称为模型，视图是模型的一个特定界面，控制器修改模型，以响应某个事件的代码。MVC 的 3 个组件在反馈循环中交互操作；动作由控制器处理，控制器会调整模型，把修改返回到视图中。控制器也可以直接修改视图，例如 UI 元素。图 4-4 表示了这种关系。使用这种范式，不同的组件被清楚地分开，允许修改一个组件而不必修改其他组件。例如，不必接触底层数据模型或逻辑，可以轻松地在基于文本的界面和图形用户界面之间切换，或者在桌面 PC 上运行的界面和作为手机应用程序运行的界面之间切换。

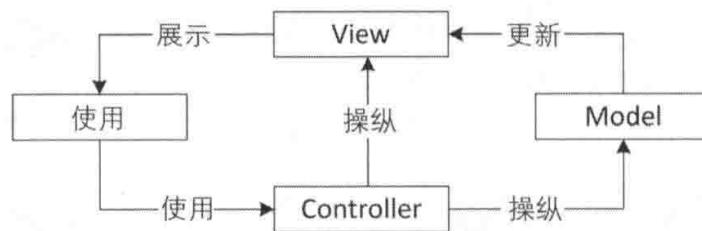


图 4-4 MVC 的 3 个组件的关系

表 4-2 是关于国际象棋游戏子系统的表格。

表 4-2 国际象棋游戏子系统

子系统	实例数	功能	公开的接口	使用的接口
GamePlay	1	开始游戏 控制游戏进度 控制绘图 判断胜方 结束游戏	游戏结束	轮流(Player 提供) 绘图(ChessBoardView 提供)
ChessBoard	1	存储棋子 检测平局和将死	获取棋子 设置棋子	游戏结束(GamePlay 提供)
ChessBoardView	1	绘制相关的棋盘	绘制	绘制(ChessPieceView 提供)
ChessPiece	32	移动自身 检测合法移动	移动 检测移动	获取棋子(ChessBoard 提供) 设置棋子(ChessBoard 提供)
ChessPieceView	32	绘制相关的棋子	绘制	无
Player	2	与用户交互：提醒用户移动，获取用户的移动信息，移动棋子	轮流	获取棋子(ChessBoard 提供) 移动(ChessPiece 提供) 检测移动(ChessPiece 提供)
ErrorLogger	1	将错误信息写入日志文件	记录错误	无

如表 4-2 所示，国际象棋游戏的功能子系统包括：GamePlay、ChessBoard、ChessBoardView、

ChessPiece、ChessPieceView、Player 和 ErrorLogger。然而，这并不是唯一合理的方式。软件设计与编程本身一样，达到同一个目标有多种不同的方法。并不是所有的方法都是等价的：有些方法比另一些方法好。然而，经常有几种同样有效的方法。

划分良好的子系统将程序分割为基本功能单元。例如，Player 就是与 ChessBoard、ChessPiece 和 GamePlay 明显不同的子系统。将 Player 混合在 GamePlay 子系统中没有任何意义，因为在逻辑上它们是独立的子系统。但其他选择未必这么明显。

在 MVC 设计中，ChessBoard 和 ChessPiece 子系统是模型部分。ChessBoardView 和 ChessPieceView 是视图部分，Player 是控制器部分。

由于表格无法形象地表示子系统之间的关系，通常会使用图示表明程序的子系统，在此箭头表示一个子系统对另一子系统的调用。图 4-5 用 UML 通信图可视化了国际象棋游戏的各个子系统。

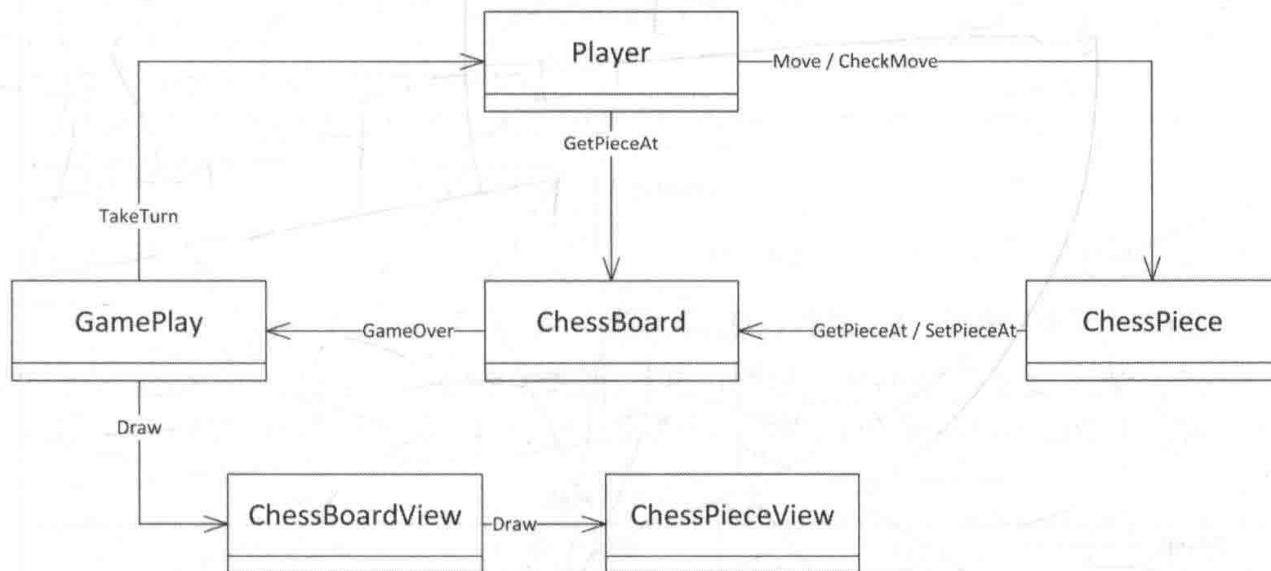


图 4-5 UML 通信图

## 2. 选择线程模型

在设计阶段，考虑如何在要编写的算法中将特定的循环编写为多线程显得太早了。但在这个阶段，可以选择程序中高级线程的数目并指定线程的交互方式。高级线程的示例有 UI 线程、音频播放线程、网络通信线程等。

在多线程设计中，应该尽可能避免共享数据，这样可使程序更简单、更安全。如果无法避免共享数据，应该指定加锁需求。

如果不熟悉多线程程序，或者平台不支持多线程，那么程序应该是单线程的。然而，如果程序有多个不同的任务，每个任务都并行运行，多线程或许是个不错的选择。例如，图形用户界面程序经常让一个线程执行主程序，其他线程等待用户按下按钮或者选择菜单项。多线程程序将在第 27 章讲述。

国际象棋程序只需要一个线程来控制游戏流程。

## 3. 指定每个子系统的类层次结构

在这个步骤中决定程序中要编写的类层次结构。国际象棋程序需要一个类层次结构来代表棋子，这个类层次结构如图 4-6 所示。在这个类层次结构中，ChessPiece 泛型类作为抽象基类。ChessPieceView 类也有类似的类层次结构。

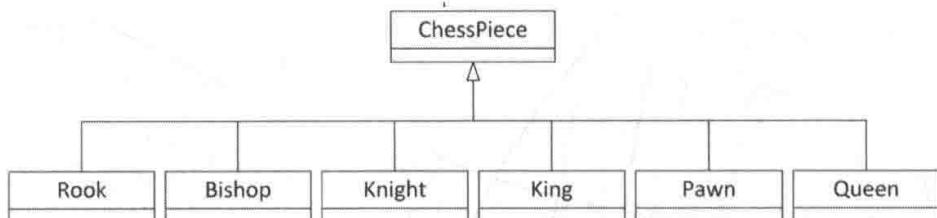


图 4-6 类层次结构代表棋子

另一个类层次结构用于 ChessBoardView 类，以实现游戏的文本界面或用户图形界面。图 4-7 给出了这个类层次结构，可以在控制台以文本方式显示棋盘，也可以用 2D 或 3D 图形显示棋盘。ChessPieceView 类体系中的各个类也需要类似的类层次结构。

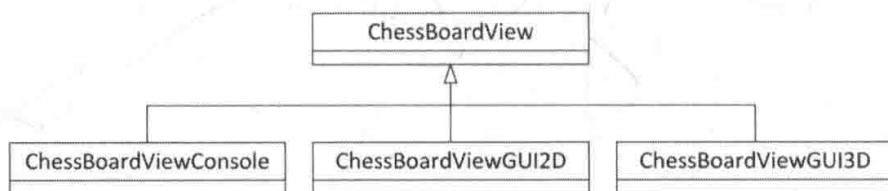


图 4-7 用于 ChessBoardView 类的类层次结构

第 5 章将讲述设计类和类层次结构的细节。

#### 4. 指定每个子系统的类、数据结构、算法和模式

在这个步骤中，需要更大限度地考虑细节，并指定每个子系统的细节，包括指定为每个子系统编写的特定类。可能最后模型中的每个子系统本身都会成为一个类。这些信息也可以用表 4-3 总结。

表 4-3 子系统和类

子系统	类	数据结构	算法	模式
GamePlay	GamePlay 类	GamePlay 对象包含一个 ChessBoard 对象和两个 Player 对象	让每个玩家轮流移动	无
ChessBoard	ChessBoard 类	ChessBoard 对象存储 32 个棋子的二维表示	在每次移动之后检测“胜”或“和”	无
ChessBoardView	抽象超类: ChessBoardView 具体子类: ChessBoardViewConsole、 ChessBoardViewGUI2D ...	存储与如何绘制棋盘有关的信息	绘制棋盘	观察者 (Observer)
ChessPiece	抽象超类: ChessPiece 子类: Rook、Bishop、Knight、 King、Pawn 和 Queen	每个棋子存储它在棋盘上的位置	在棋盘的不同位置查询棋子，判断棋子的移动是否合法	无
ChessPieceView	抽象基类: ChessPieceView 子类: RookView、BishopView ... 具体子类: RookViewConsole、 RookViewGUI2D ...	存储如何绘制棋子的信息	绘制棋子	观察者 (Observer)

(续表)

子系统	类	数据结构	算法	模式
Player	抽象超类: Player 具体子类: PlayerConsole、PlayerGUI2D ...	无	提示用户移动, 检测移动是否合法, 移动棋子	仲裁者(Mediator)
ErrorLogger	ErrorLogger 类	要记录的消息队列	缓存消息, 并将消息写入日志文件	依赖注入

这样的表格已经给出了软件设计中不同类的一些信息, 但并没有清楚地描述它们之间的交互。UML 序列图可用于对此类交互进行建模。图 4-8 显示了这样一个图, 将表 4-3 中的一些类的交互可视化。

图 4-8 中的图表仅显示了一次迭代, 从 GamePlay 到 Player 的单个 TakeTurn 调用, 因此, 它只是一个部分序列图。在 TakeTurn 调用完成后, GamePlay 对象应该要求 ChessBoardView 绘制自己, 而后者又应该要求不同的 ChessPieceView 绘制自己。此外, 应该扩展序列图以可视化棋子如何取走对手的棋子, 并包括对王车易位的支持, 这种移动涉及玩家的国王和玩家的一个车。王车易位是唯一需要同时移动两个棋子的行动。

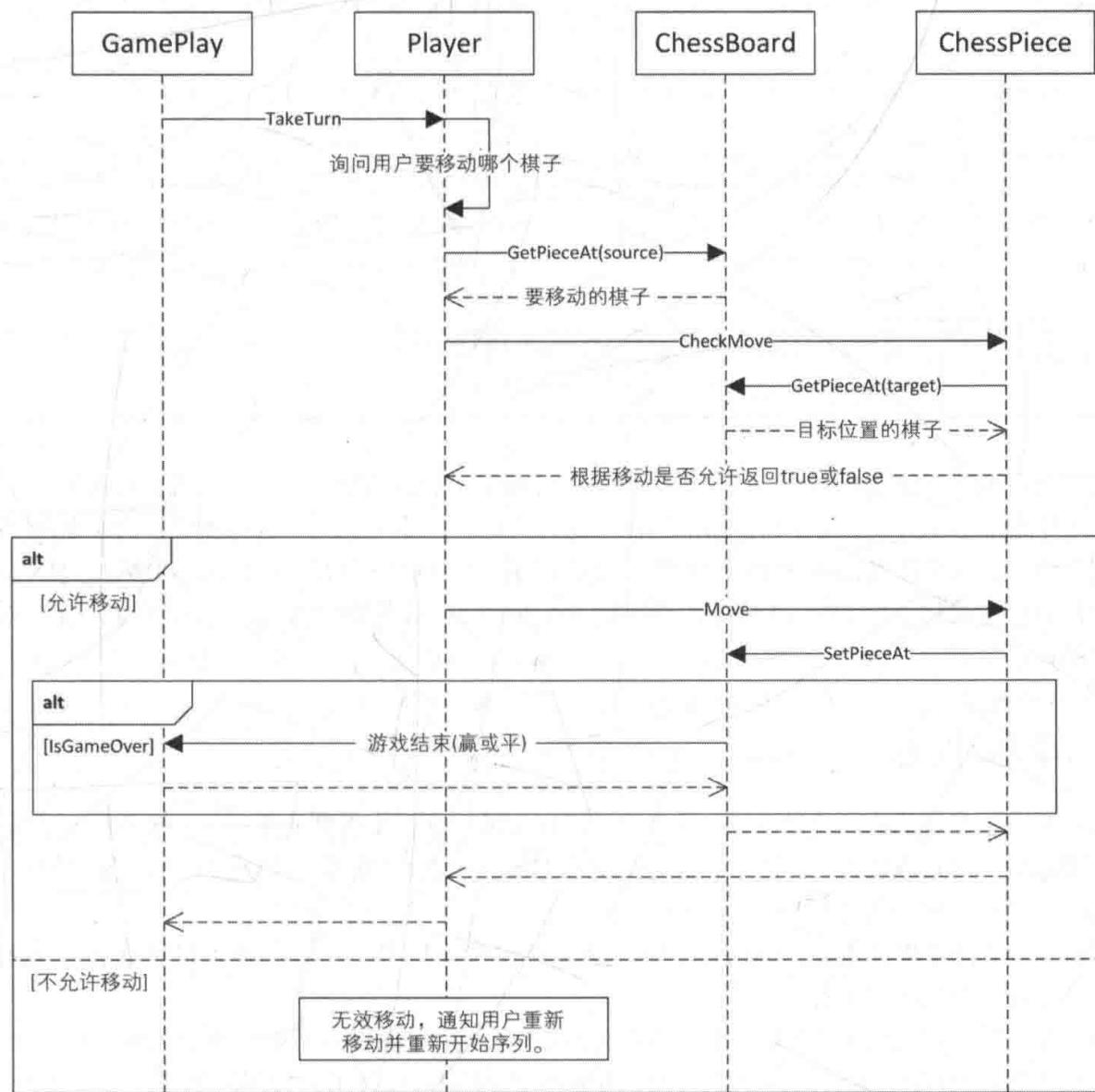


图 4-8 部分序列图

设计文档的这部分内容通常提供每个类的实际接口，但这个示例略去了该层次的细节。

设计类和选择数据结构、算法和模式都很灵活。应该牢牢记住本章前面讲过的抽象和重用法则。对于抽象而言，关键是将接口与实现分开。首先，从用户的观点确定接口，决定想让组件做什么，然后决定如何选择数据结构和算法，让组件做到这一点。对于重用而言，应该熟悉标准数据结构、算法和模式。此外，一定要熟悉 C++ 标准库代码和公司拥有的专用代码。

### 5. 为每个子系统指定错误处理

在这个设计步骤中，描述每个子系统的错误处理。错误处理应该同时包括系统错误(如网络连接失败)和用户错误(如无效输入)，应该确定是否每个子系统都要使用异常。可使用表 4-4 总结这些信息。

表 4-4 错误总结

子系统	处理系统错误	处理用户错误
GamePlay	如果无法为 ChessBoard 或 Player 分配内存，用 ErrorLogger 记录错误，向用户显示一条消息，并按正常步骤关闭程序	不适用(不是直接的用户界面)
ChessBoard	如果无法分配内存，用 ErrorLogger 记录错误并抛出异常	不适用(不是直接的用户界面)
ChessPiece		
ChessBoardView	如果在绘制时出现错误，用 ErrorLogger 记录错误并抛出异常	不适用(不是直接的用户界面)
ChessPieceView		
Player	如果无法分配内存，用 ErrorLogger 记录错误并抛出异常	全面检测用户的移动输入，以确保用户没有离开棋盘；提示用户输入其他信息；在移动棋子之前检测移动的合法性；如果不合法，提示用户重新移动
ErrorLogger	如果无法分配内存，尝试记录错误，通知用户并按正常步骤关闭程序	不适用(不是直接的用户界面)

错误处理的通用法则是处理一切。努力想象所有可能的错误情况，如果忘掉了某种可能性，就可能在程序中形成 bug！不要将任何事情都当作“意外”错误。要考虑到所有可能性：内存分配失败、无效用户输入、磁盘错误和网络错误，以上只是给出了一些示例。然而，正如国际象棋游戏显示的那样，应将内部错误和用户错误区别对待。例如，用户输入无效移动时不应该使程序终止。第 14 章将深入介绍错误处理。

## 4.7 本章小结

本章介绍了专业的 C++ 设计方法。软件设计是任何编程项目中重要的第一步。你还学习了使得设计变得困难的一些 C++ 特性，包括 C++ 关注的面向对象、庞大的功能集和标准库、编写通用代码的功能。这些信息可让程序员更好地处理 C++ 设计。

本章介绍了两个设计主题：抽象和重用。抽象(或将接口与实现分离)的概念贯穿全书，所有的设计工作都应该以此为指导方针。

第二个主题，重用的概念(无论是代码还是设计)在实际项目和本书中经常会出现。C++ 设计应该包含代码的重用(以库或框架的形式)以及思想和设计的重用(以技术和模式的形式)。应该尽可能编写可重用的代码。此外还要记住权衡重用的优缺点和重用代码的特定方针，包括理解功能和限制、性

能、许可协议、支持模式、平台限制、原型和帮助资源。你还学习了性能分析和大O表示法。现在你已经理解了设计的重要性和基本的设计主题，并做好了学习本书第II部分其余章节的准备。第5章将讲述在设计中使用C++面向对象特性的策略。

## 4.8 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，然后再从网站上查看解决方案。

**练习4-1** 在C++中做自己的设计时，最基本的两条设计原则是什么？

**练习4-2** 假设你有如下的Card类。该类只支持一副牌中的普通牌，不需要支持所谓的大小王。

```
class Card
{
public:
    enum class Number { Ace, Two, Three, Four, Five, Six, Seven, Eight,
        Nine, Ten, Jack, Queen, King };
    enum class Figure { Diamond, Heart, Spade, Club };

    Card() {};
    Card(Number number, Figure figure)
        : m_number{ number }, m_figure{ figure } {}
private:
    Number m_number { Number::Ace };
    Figure m_figure { Figure::Diamond };
};
```

如何看待以下使用Card类表示一副纸牌的做法？你能想到哪些改进之处？

```
int main()
{
    Card deck[52];
    // ...
}
```

**练习4-3** 假设你与朋友一起想出了一个为移动设备制作3D游戏的好主意。你有一部Android设备，而你的朋友有一部Apple iPhone，当然你希望游戏能在两种设备上都玩。高层次地解释你将如何处理这两个不同的移动平台，以及如何准备开始游戏开发。

**练习4-4** 给定以下几个大O复杂度： $O(n)$ ,  $O(n^2)$ ,  $O(\log n)$ ,  $O(1)$ 。你能按照递增的复杂度来对它们排序吗？它们的名字是什么？你能想到更差的复杂度吗？



# 第5章

## 面向对象设计

### 本章内容

- 什么是面向对象的程序设计
- 什么是类、对象、属性和行为
- 如何定义不同对象之间的关系

第4章引导你正确认识良好的软件设计，现在可将对象的概念和良好设计的概念组合在一起。在代码中使用对象的程序员与真正掌握面向对象编程的程序员是不同的，后者能更完美地管理对象相互联系的方式和程序的总体设计。

本章首先介绍过程式编程(C 风格)，之后详述面向对象编程(Object Oriented Programming, OOP)。即使有多年使用对象的经验，也仍然应该阅读本章，了解关于对象的新思想。本章讨论对象之间的不同关系，包括创建面向对象程序时可能遇到的陷阱。

思考过程式编程或面向对象编程时，要记住的最重要的一点是，它们只是代表了对程序中发生的事情的不同理解方式。程序员在完全理解什么是对象之前，经常困惑于新的语法和 OOP 术语。本章轻代码，重概念和思想。第 8~10 章将更深入地介绍 C++ 对象语法。

### 5.1 过程化的思考方式

过程语言(例如 C)将代码分割为小块，每个小块(理论上)完成单一的任务。如果在 C 中没有过程，所有代码都会集中在 main() 中。代码将难以阅读，同事会恼火，这还是最轻的。

计算机并不关心代码是位于 main() 中还是被分割成具有描述性名称和注释的小块。过程是一种抽象，它的存在是为了帮助程序员和阅读或维护代码的人。这个概念建立在一个与程序相关的基本问题之上——程序做了什么事情？用自然语言回答这个问题，就是过程化思考。例如，以下的答案为起点设计一个股票选择程序：首先从 Internet 获取股价，然后根据特定的指标对数据排序，之后分析已经排序的数据，最后输出建议购买和出售的列表。当开始编写代码时，可能会将脑海中的模型直接转换为 C 函数：retrieveQuotes()、sortQuotes()、analyzeQuotes() 和 outputRecommendations()。

**注意:**

尽管 C 将过程表示为“函数”，但 C 并非一门函数式语言。术语“函数式(functional)”与“过程式(procedural)”有很大的不同，函数式语言指的是类似于 Lisp 的语言，Lisp 使用完全不同的抽象。

当程序遵循特定的步骤序列时，过程方法运行良好。然而，在现代的大型应用程序中，很少有线性的事件序列，通常用户可在任何时候执行任何命令。此外，过程思想对于数据的表示没有任何说明，在前面的示例中，并没有讨论股价实际上是什么。

如果过程模型听起来像是你处理程序的方法，不要担心。一旦你意识到 OOP 只是一种替代方法，一种更灵活的对软件的思考方法，面向对象编程就会变得十分自然。

## 5.2 面向对象思想

与基于“程序做什么”问题的面向过程方法不同，面向对象方法提出另一个问题：对哪些实际对象进行建模？OOP 的基本观念是不应该将程序分割为若干任务，而是将其分为自然对象的模型。乍看上去这有些抽象，但用类、组件、属性和行为等术语考虑实际对象时，这一思想就会变得更清晰。

### 5.2.1 类

类将对象与其定义区分开来。以橘子为例，一般作为美味水果的长在树上的橘子和某个特定橘子(例如，现在笔者的键盘旁就放着一个往外渗汁的橘子)有所不同。

当回答“什么是橘子”时，就是在谈论“橘子”这种东西的类。所有橘子都是水果，所有橘子都长在树上，所有橘子的颜色都是某种程度的橙色，所有橘子都有特定的味道。类只是封装了用来定义对象类别的信息。

当描述某个特定橘子时，就是在讨论一个对象。所有对象都属于某个特定的类。由于笔者桌子上的对象是一个橘子，因此它属于橘子(Orange)类。因此，它是长在树上的一种水果，它的颜色是中等程度的橙色，并且味道不错。对象是类的一个实例(instance)——它拥有一些特征，从而与同一类型的其他事物区分开来。

上面的股票选择程序是一个更具体的示例。在 OOP 中，“股价”是一个类，因为它定义了报价这个抽象概念。某个特定的报价(例如“当前 Microsoft 股价”)是一个对象，因为它是这个类的特定实例。

具有 C 背景的程序员，可将类和对象类比为类型和变量。实际上，第 1 章提到过，类的语法与 C 的结构体类似。

### 5.2.2 组件

如果考虑一个复杂的真实对象，例如飞机，很容易看到它由许多小组件(component)组成。其中包括机身、控制装置、起落架、引擎和其他很多部件。对于 OOP 而言，将对象分解为更小组件是一项必备能力，就像将复杂任务分解为较小过程是过程式编程的基础一样。

本质上，组件与类相似，但组件更小、更具体。优秀的面向对象程序可能有 Airplane 类，但是，如果要充分描述飞机，这个类将过于庞大。因此，Airplane 类只处理许多更容易管理的小组件。每个组件可能还有更小的组件，例如，起落架是飞机的一个组件，车轮是起落架的一个组件。

### 5.2.3 属性

属性将一个对象与其他对象区分开来。回到橘子(Orange)类，所有橘子都定义为橙色的，并具有

特定口味，这两个特征就是属性。所有橘子都具有相同的属性，但属性的值不同。一个橘子可能“美味可口”，但另一个橘子可能“苦涩难吃”。

也可在类的层次上思考属性。如前所述，所有橘子都是水果，都长在树上。这是水果类的属性，而特定程度的橙色是由特定的水果对象决定的。类属性由所有的类成员共享，而类的所有对象都有对象属性，但具有不同的值。

在股票选择示例中，股价有几个对象属性，包括公司名称、股票代码、当前股价和其他统计数据。

属性用来描述对象的特征，回答“为什么这个对象与众不同”的问题。

#### 5.2.4 行为

行为回答两个问题：“对象做什么”和“能对对象做什么”。在橘子示例中，橘子本身不会做什么，但是我们可对橘子做一些事情。橘子的行为之一是可供食用。与属性类似，可在类或对象层次上思考行为。几乎所有橘子都会以相同的方式被吃掉，但其他行为未必如此，例如被扔到斜坡上向下滚动，圆橘子与扁圆橘子的行为明显不同。

前面的股票选择示例提供了一些更实际的行为。以过程方式思考，该程序的功能之一就是分析股价。以 OOP 的方式思考，股价对象可以自我分析，分析变成股价对象的一个行为。

在面向对象编程中，许多功能性的代码从过程转移到类。通过建立具有某些行为的对象并定义对象的交互方式，OOP 以更丰富的机制将代码和代码操作的数据联系起来。类的行为由类方法实现。

#### 5.2.5 综合考虑

通过这些概念，可回头分析股票选择程序，并以面向对象的方式重新设计这个程序。

前面说过，“股价”类是一个不错的开始。为获取报价表，程序需要股价组的概念，通常称为集合。因此，较好的设计可能使用一个类代表“股价的集合”，这个类由代表单个“股价”的小组件组成。

再来说说属性，这个集合类至少有一个属性——实际接收到的报价表。它可能还有其他附加属性，例如大多数最新检索的确切日期和时间。至于行为，“股价的集合”将从服务器那里获取报价，并提供有序的报价表。这就是“股价检索”和“股价排序”行为。

股价类具有前面讨论的一些属性——名称、代码、当前价格等，此外还具有分析行为。还可能考虑其他行为，如买入和卖出股票。

图示通常有助于呈现组件之间的关系。图 5-1 使用 UML 类图语法(附录 D 介绍 UML 语法)说明一个 StockQuoteCollection 包含零个或多个 StockQuote 对象，StockQuote 对象属于单个 StockQuoteCollection。

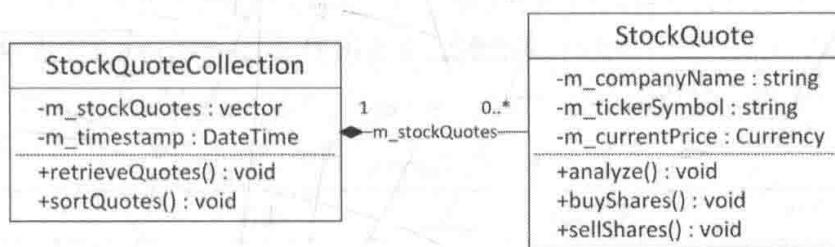


图 5-1 UML 类图语法

图 5-2 显示了 Orange 类可能的 UML 类图。

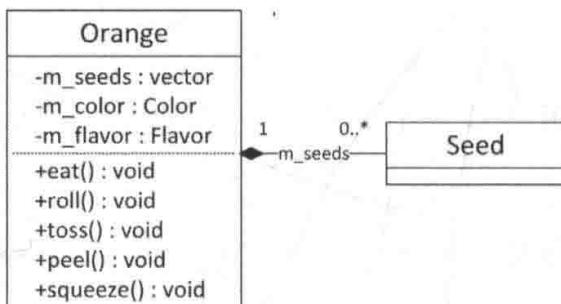


图 5-2 Orange 类可能的 UML 类图

## 5.3 生活在类的世界里

当程序员的思想从面向过程转换到面向对象模式时，对于将属性和行为结合到类，通常会有一种恍然大悟的感觉。某些程序员重新审视项目的设计，并要将某些部分作为类重写。其他程序员可能试着抛开所有代码并重新开始这个项目，将其作为完全的面向对象应用程序。

使用类开发软件主要有两种方法。对于某些人来说，类只是代表数据和功能的良好封装，这些程序员在程序中大量使用类，使代码更容易阅读和维护。采用这种方法的程序员将独立的代码段切除，用类替换它们，就像外科医生给病人植入心脏起搏器一样。这种方法当然没有错，这些人将对象当作在许多情况下有益的工具。程序的某些部分(例如股价)只是“感觉像类”。这些部分可被分离开来，并用实际术语描述。

另一些程序员彻底采用 OOP 范式，将一切都转换为类。在他们的心目中，某些类对应于实际事物，例如橘子或股价，而另一些类封装了更抽象的概念，例如 sorter 或 undo 对象。

理想方法或许介于这两个极端之间。第一个面向对象程序可能实际上只在传统的程式程序中使用几个类；以后可能全力以赴将一切都作为类，从表示 int 的类到表示主应用程序的类。随着时间的推移，一定会找到合理的折中方法。

### 5.3.1 过度使用类

设计一个富于创造性的面向对象系统，将所有细小的事情都转换为类，常常会惹恼团队中的所有其他人。弗洛伊德曾经说过，有时变量就只是变量。下面就解释这句话的含义。

假定设计一款将会畅销的井字棋游戏，打算完全采用 OOP 方法，因此你坐了下来，喝一杯咖啡，在笔记本上勾画出所需的类和对象。在此类游戏中，通常有一个对象监视游戏的进度，并裁定胜方。为了表示游戏棋盘，需要用 Grid 对象跟踪标记和它们的位置。实际上，表示 X 或 O 的 Piece 类是 Grid 的一个组件。

等一下，退回去！这种设计打算用一个类代表 X 或 O。这就是过度使用类的一个例子。用 char 不能代表 X 或 O 吗？另外，用一个枚举类型的二维数组表示 Grid 不是更好吗？Piece 对象只会使代码更复杂。看看表 5-1 建议的 Piece(棋子)类。

表 5-1 Piece(棋子)类

类	相关组件	属性	行为
Piece	无	X 或 O	无

这个表格有点稀疏，这有力地表明此处的内容太细粒度了，并不能成为一个完整的类。

另一方面，深谋远虑的程序员可能认为，尽管当前 Piece 类很小，但使用对象可在将来扩展时不受影响。或许发展下去这会成为一个图形应用程序，用 Piece 类支持绘图行为可能是有用的。其他属性可能是棋子的颜色或者一些判定，用于判断 Piece 是不是最近移动的那枚棋子。

另一种方案是考虑方格(Grid)的状态而不是使用棋子。方格的状态可能是空、X 或 O。为了使设计能够经得起未来的考验，可设计一个抽象基类 State，其能够派生出子类 StateEmpty、StateX 和 StateO。通过这种设计，将来可以向基类或子类添加额外的属性。

显然在此不存在唯一正确的答案，关键是在设计应用程序时应该考虑这些问题。记住类是用来帮助程序员管理代码的，如果使用类只是为使代码“更加面向对象”，就错了。

### 5.3.2 过于通用的类

相对于将不应该确定为对象的事物当作类，过于通用的类可能更糟糕。所有的 OOP 学生都以“橘子”示例开始——这确实是类，没有疑问。在实际编码中，类可以非常抽象。许多 OOP 程序都有一个“应用程序对象”，尽管应用程序并不能以物质的形式表现，但用类表示应用程序仍然是有意义的，因为应用程序本身具有一些属性和行为。

过于通用的类根本无法代表具体的事物。程序员的本意可能是建立一个灵活或可重用的类，但最终得到一个令人迷惑的类。例如，考虑一个组织和显示媒体的程序。这个程序可将照片分类，组织数字音乐唱片和电影收藏，还可作为个人日志。将所有事物都当作 media 对象，并创建一个可容纳所有格式的类，就是一种过分的做法。这个类可能有一个 data 属性，这个属性包含图像、歌曲或日志项的原始位，具体取决于媒体类型，该类还可能有一个 perform 行为，可正确地绘制图像、播放歌曲或编辑日志项。

这个类过于通用的原因在于属性和行为的名称。单词 data 本身几乎没有意义——在此必须使用一个通用词语，因为这个类被过度地扩展到 3 种完全不同的情况。同理，perform 会在 3 种不同的情况下执行差别极大的操作。总之，这个类想做的事情太多了。

然而，当设计一个程序来组织媒体时，你的应用程序中肯定会有个 Media 类。此 Media 类将包含所有类型媒体的通用属性，例如名称、预览、指向相应媒体文件的链接等。但此 Media 类不应包含有关处理特定媒体的详细信息。它不应该包含显示图像、播放歌曲或电影的代码。相反，在设计中应有其他类，例如 Picture 类和 Movie 类。然后，这些特定类包含实际的特定于媒体的功能，例如显示图片或播放电影。显然，这些特定于媒体的类在某种程度上与 Media 类相关，这正是下一节的主题——如何表达类之间关系。

## 5.4 类之间的关系

作为程序员，必然会遇到这样的情况：不同的类具有共同的特征，至少看起来彼此有联系。面向对象的语言提供了许多机制来处理类之间的这种关系。最棘手的问题是理解这些关系的实质。类之间的关系主要有两类——“有一个”(has a)关系和“是一个”(is a)关系。

### 5.4.1 “有一个”关系

“有一个”关系模式是 A 有一个 B，或者 A 包含一个 B。在此类关系中，可认为某个类是另一个类的一部分。前面定义的组件通常代表“有一个”关系，因为组件表示组成其他类的类。

动物园和猴子就是这种关系的一个示例。可以说动物园里有一只猴子，或者说动物园包含一只猴子。在代码中用 Zoo 对象模拟动物园，这个对象有一个 Monkey 组件。

考虑用户界面通常有助于理解类之间的关系。尽管并非所有的 UI 都是(虽然现在大多数都是)以 OOP 方式实现的，屏幕上的视觉元素也能很好地转换为类。UI 关于“有一个”关系的类比就是窗口中包含一个按钮。按钮和窗口是两个明显不同的类，但又明显有某种联系。由于按钮在窗口中，因此说窗口中有一个按钮。

图 5-3 显示了实际的“有一个”关系和用户界面的“有一个”关系。

“有一个”关系有如下两种类型。

- 聚合：通过聚合，当聚合器被销毁时，聚合对象(组件)可以继续存在。例如，假设动物园对象包含一组动物对象。当动物园对象因为破产而被销毁时，动物对象(理想情况下)不会被销毁，他们被转移到另一个动物园。
- 组合：对于组合，如果由其他对象组成的对象被销毁，那么这些其他对象也会被销毁。例如，如果包含按钮的窗口对象被销毁，则这些按钮对象也将被销毁。

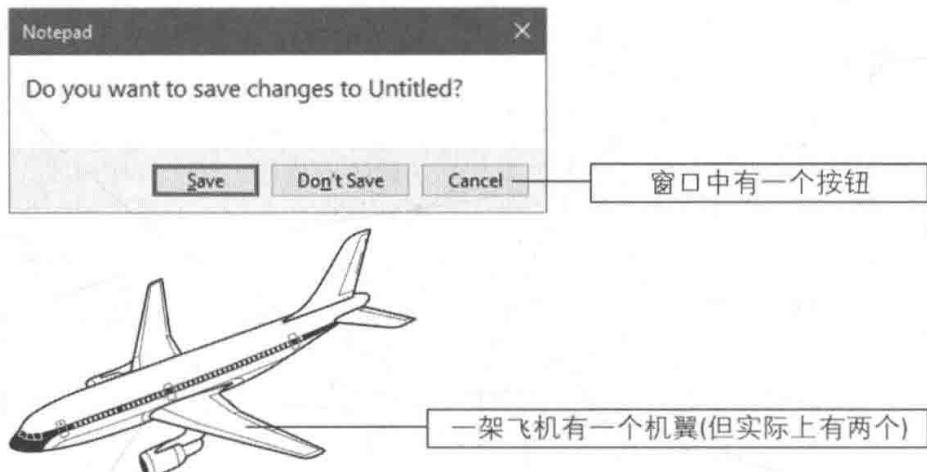


图 5-3 实际的“有一个”关系和用户界面的“有一个”关系

#### 5.4.2 “是一个”关系(继承)

“是一个”关系是面向对象编程中非常基本的概念，因此有许多名称，包括派生(deriving)、子类(subclass)、扩展(extending)和继承(inheriting)。类模拟了现实世界包含具有属性和行为的对象这一事实，继承模拟了这些对象通常以层次方式来组织这一事实。“是一个”说明了这种层次关系。

基本上，继承的模式是：A 是一个 B，或者 A 实际上与 B 非常相似——这可能比较棘手。再次以简单的动物园为例，但假定动物园里除了猴子之外还有其他动物。这句话本身已经建立了关系——猴子是一种动物。同样，长颈鹿也是一种动物，袋鼠是一种动物，企鹅也是一种动物。那又怎么样？意识到猴子、长颈鹿、袋鼠和企鹅具有某种共性时，继承的魔力就开始显现了。这些共性就是动物的一般特征。

对于程序员的启示就是，可定义 Animal 类，用以封装所有动物都有的属性(大小、生活区域、食物等)和行为(走动、进食、睡觉)。特定的动物(例如猴子)成为 Animal 的子类，因为猴子包含动物的所有特征。记住，猴子是动物，并且它还有与众不同的其他特征。图 5-4 显示了动物的继承图示。箭头表明“是一个”关系的方向。

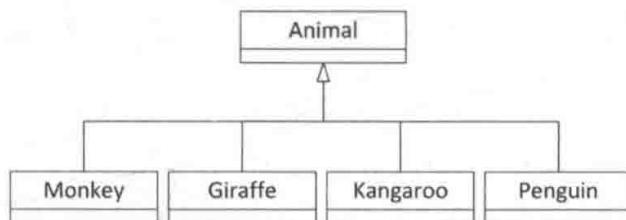


图 5-4 动物的继承图示

就像猴子与长颈鹿是不同类型的动物一样，用户界面中通常也有不同类型的按钮。例如，复选框是一个按钮，按钮只是一个可被单击并执行操作的 UI 元素，`Checkbox` 类通过添加状态(相应的框是否被选中)扩展了 `Button` 类。

当类之间具有“是一个”关系时，目标之一就是将通用功能放入基类(base class)，其他类可扩展基类。如果所有子类都有相似或完全相同的代码，就应该考虑将一些代码或全部代码放入基类。这样，可在什么地方完成所需的改动，将来的子类可“免费”获取这些共享的功能。

## 1. 继承技术

前面的示例非正式地讲述了继承中使用的一些技术。当生成子类时，程序员有多种方法将某个类与其父类(也称为基类或超类)区分开。可使用多种方法生成子类，生成子类实际上就是完成语句 `A is a B that...` 的过程。

### 添加功能

派生类可在基类的基础上添加功能。例如，猴子是一种可以在树间荡秋千的动物。除了具有动物的所有行为以外，`Monkey` 类还有 `swingFromTrees()` 方法，这个行为只存在于 `Monkey` 类中。

### 替换功能

派生类可完全替换或重写父类的行为。例如，大多数动物都步行，因此 `Animal` 类可能拥有模拟步行的 `move` 行为。但袋鼠是一种通过跳跃而不是步行移动的动物，`Animal` 基类的其他属性和行为仍然适用，`Kangaroo` 派生类只需要改变 `move` 行为的运行方式。当然，如果对基类的所有功能都进行替换，就可能意味着采用继承的方式根本就不正确，除非基类是一个抽象基类。抽象基类会强制每个子类实现未在抽象基类中实现的所有方法。无法为抽象基类创建实例，第 10 章将介绍抽象类。

### 添加属性

除了从基类继承属性以外，派生类还可添加新属性。企鹅具有动物的所有属性，此外还有 `beak size`(鸟喙大小)属性。

### 替换属性

与重写方法类似，C++ 提供了重写属性的方法。然而，这么做通常是不合适的，因为这会隐藏基类的属性。也就是说，基类可为具有特定名称的属性指定一个值，而派生类可用同一个名字为另一个属性指定另一个值。有关“隐藏”的内容，详见第 10 章。不要把替换属性的概念与子类具有不同属性值的概念混淆。例如，所有动物都具有表明它们吃什么的 `diet` 属性，猴子吃香蕉，企鹅吃鱼，二者都没有替换 `diet` 属性——只是赋给属性的值不同而已。

## 2. 多态性

多态性(Polymorphism)指遵循一套标准属性和方法的对象可互换使用。类定义就像对象和与之交互的代码之间的契约。根据定义，任意一个 `Monkey` 对象必须支持 `Monkey` 类的属性和行为。

这个概念也可推广到基类。由于所有猴子都是动物，因此所有 Monkey 对象都支持 Animal 类的属性和行为。

多态性是面向对象编程的亮点，因为多态性真正利用了继承所提供的能力。在模拟动物园时，可通过编程遍历动物园中的所有动物，让每个动物都移动一次。由于所有动物都是 Animal 类的成员，因此它们都知道如何移动。某些动物重写了移动行为，但这正是亮点所在——代码只是告诉每个动物移动，而不知道也不关心是哪种动物。所有动物都按自己的方式移动。

### 5.4.3 “有一个”与“是一个”的区别

在现实中，区分对象之间的“有一个”与“是一个”关系相当容易。没人会说橘子有一个水果——橘子是一种水果。在代码中，有时并不会这么明显。

考虑一个代表关联数组的假想类，关联数组是高效地将键映射到值的一种数据结构。例如，保险公司使用 `AssociativeArray` 类将成员 ID 映射到名称，从而给定一个 ID 就可以方便地找到对应的成员名称。成员 ID 是键，成员名称是值。

在关联数组的标准实现中，每个键都有一个值。如果 ID 14534 映射到名称“Kleper, Scott”，就不能再映射到成员名称“Kleper, Marni”。在大多数实现中，如果对一个已经有值的键添加第二个值，第一个值就会消失。换句话说，如果 ID 14534 映射到“Kleper, Scott”，然后又将 ID 14534 分配给“Kleper, Marni”，那么 Scott 将被遗弃。下面的序列调用了两次假想的 `insert()` 方法，并给出了每次调用结束后关联数组的内容。

```
myArray.insert(14534, "Kleper, Scott");
```

键	值
14534	"Kleper, Scott" [字符串]

```
myArray.insert(14534, "Kleper, Marni");
```

键	值
14534	"Kleper, Marni" [字符串]

不难想象类似于关联数组但允许一个键有多个值的数据结构的用处。在保险公司示例中，一个家庭可能有多个名称对应于同一个 ID。由于这种数据结构非常类似于关联数组，因此可用某种方式使用关联数组的功能。关联数组的键只能有一个值，但是这个值可以是任意类型。除字符串外，这个值还可以是一个包含多个值的集合(例如 `vector`)。当向已有 ID 添加新的成员时，可将名称加入集合中。运行方式如下所示：

```
Collection collection; // Make a new collection.
collection.insert("Kleper, Scott"); // Add a new element to the collection.
myArray.insert(14534, collection); // Insert the collection into the array.
```

键	值
14534	{"Kleper, Scott"}[集合]

```
Collection collection { myArray.get(14534) }; // Retrieve the existing collection.
collection.insert("Kleper, Marni"); // Add a new element to the collection.
myArray.insert(14534, collection); // Replace the collection with the updated one.
```

键	值
14534	{"Kleper, Scott", "Kleper, Marni"} [集合]

使用集合而不是字符串有些繁杂，需要大量重复代码。最好在一个单独的类中封装多值功能，可将这个类称为 MultiAssociativeArray。MultiAssociativeArray 类的运行与 AssociativeArray 类似，只是背后将每个值存储为字符串的集合，而不是单个字符串。很明显，MultiAssociativeArray 与 AssociativeArray 有某种联系，因为它仍然使用关联数组存储数据。不明显的是，这是“是一个”关系还是“有一个”关系？

先考虑“是一个”关系。假定 MultiAssociativeArray 是 AssociativeArray 的派生类，它必须重写在表中添加项的行为，从而既可创建集合并添加新元素，又可检索已有集合并添加新元素。此外必须重写检索值的行为。但是有一个复杂的问题：被重写的 get()方法应该返回一个值，而不是一个集合。MultiAssociationArray 应该返回哪个值？一个选项是返回与给定键关联的第一个值。可以添加一个额外的 getAll()方法来检索与键关联的所有值，这似乎是一个非常合理的设计。即使它重写了基类的所有方法，它仍然可以通过使用派生类中的新方法来使用基类的方法。这种方法的 UML 类图如图 5-5 所示。

现在考虑“有一个”关系。MultiAssociativeArray 属于自己的类，但是包含了 AssociativeArray 对象。这个类的接口可能与 AssociativeArray 非常相似，但并不需要相同。在底层，当用户向 MultiAssociativeArray 添加项时，会将这个项封装到一个集合并送入 AssociativeArray 对象。这也很合理，如图 5-6 所示。

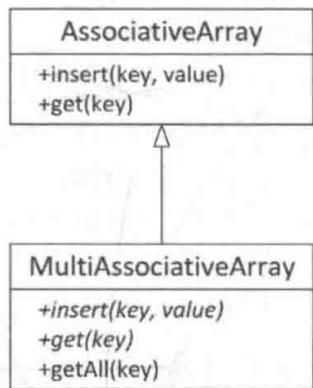


图 5-5 “是一个”关系的 UML 类图

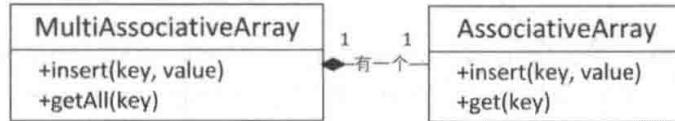


图 5-6 “有一个”关系的 UML 类图

那么，哪个方案是正确的？没有明确的答案，笔者的一个朋友认为这是“有一个”关系，他编写了一个 MultiAssociativeArray 类以供使用。主要原因是允许修改公开的接口，而不必考虑维护关联数组的功能。例如，将图 5-6 中的 get()方法改成 getAll()，以清楚表明将获取 MultiAssociativeArray 中某个特定键的所有值。此外，在“有一个”关系中，不需要担心关联数组的功能会渗透。例如，如果关联数组类提供了获取值的总数的方法，只要 MultiAssociativeArray 不重写这个方法，就可以用这个方法报告集合的总数。

这就是说，MultiAssociativeArray 实际上是一个具有新功能的 AssociativeArray，这一说法能让人信服，所以应该是“是一个”关系。但关键在于有时这两种关系之间的差别很小，需要考虑使用类的方式，还需要考虑所创建的类只是利用了其他类的一些功能，还是在其他类的基础上修改或添加了新功能。

表 5-2 给出了关于 MultiAssociativeArray 的两种方法的支持和反对意见。

表 5-2 支持和反对意见

	是一个	有一个
支持的原因	<ul style="list-style-type: none"> <li>基本上，这是具有不同特征的同一抽象</li> <li>这个类的方法与 AssociativeArray (几乎) 相同</li> </ul>	<ul style="list-style-type: none"> <li>MultiAssociativeArray 可以拥有任何有用的方法，而不需要考虑 AssociativeArray 拥有什么方法</li> <li>可在不改变公开的方法的前提下，不采用 AssociativeArray 的实现方式</li> </ul>
反对的原因	<ul style="list-style-type: none"> <li>根据定义，关联数组的一个键对应一个值，将 MultiAssociativeArray 当作关联数组是错误的</li> <li>MultiAssociativeArray 将关联数组的两个方法全部重写，这有力地说明这种设计是错误的</li> <li>AssociativeArray 未知的或不正确的属性和方法会“渗透”到 MultiAssociativeArray</li> </ul>	<ul style="list-style-type: none"> <li>在某种意义上，MultiAssociativeArray 通过提出新方法导致了重复工作</li> <li>AssociativeArray 的一些其他属性和方法可能是有用的</li> </ul>

反对“是一个”关系的理由在这种情况下非常有力。LSP(Liskov Substitution Principle, 里氏替换原则)可帮助从“是一个”和“有一个”关系中选择。这个原则指出，你应当能在不改变行为的情况下，用派生类替代基类。将这个原则应用于本例，则表明应当是“有一个”关系，因为你无法在以前使用 AssociativeArray 的地方使用 MultiAssociativeArray。否则，行为就会改变。例如，AssociativeArray 的 insert() 方法会删除映射中同一个键的旧值，而 MultiAssociativeArray 不会删除此类值。

本节详细介绍的两种解决方案实际上并不是仅有的两种可能的解决方案。其他选项可以是 AssociateArray 从 MultiAssociateArray 继承，AssociateArray 可以包含 MultiAssociateArray，AssociateArray 和 MultiAssociateArray 都从公共基类继承等。对于某个设计，通常有多种解决方案。

实际上，根据笔者多年的经验，如果确实需要选择，建议采用“有一个”关系，而不是“是一个”关系。

注意，这里使用 AssociateArray 和 MultiAssociateArray 说明了“有一个”和“是一个”关系的不同之处。在代码中，建议使用标准关联数组类，而不是自己写一个。C++ 标准库中提供了 map 类，用来代替 AssociateArray，此外还提供了 multimap 类，用来代替 MultiAssociateArray 类。第 18 章将讨论这两个标准类。

#### 5.4.4 not-a 关系

当考虑类之间的关系时，应该考虑类之间是否真的存在关系。不要把对面向对象设计的热情全部转换为许多不必要的类/子类关系。

当实际事物之间存在明显关系，而代码中没有实际关系时，问题就出现了。OO(面向对象)层次结构需要模拟功能关系，而不是人为制造关系。图 5-7 显示的关系作为概念集或层次结构是有意义的，但在代码中并不能代表有意义的关系。

避免不必要的继承的最好方法是首先给出大概的设计。为每个类和派生类写出计划设置的属性和方法。如果发现某个类没有自己特定的属性或方法，或者某个类的所有属性和方法都被派生类重写，只要这个类不是前面提到的抽象基类，就应该重新考虑设计。

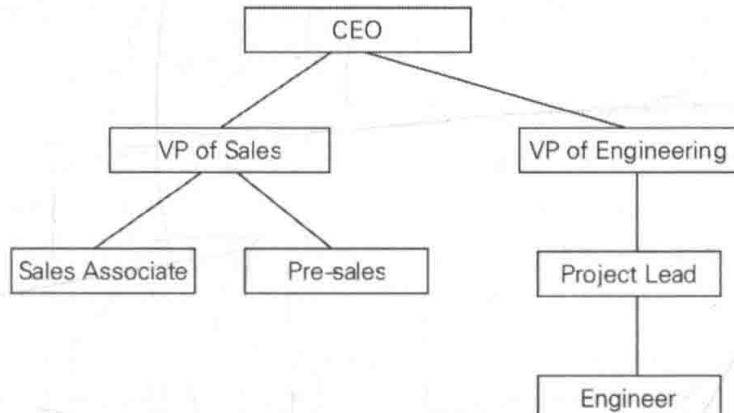
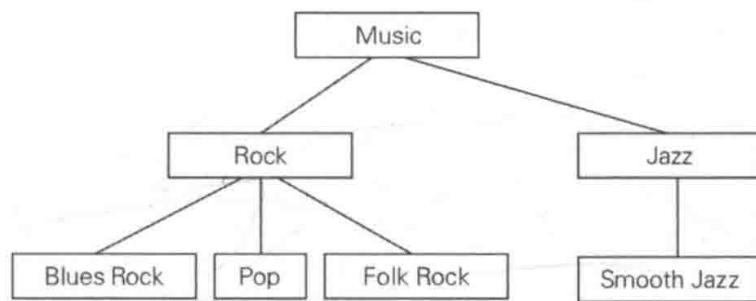


图 5-7 关系作为概念集或层次结构是有意义的

#### 5.4.5 层次结构

正如类 A 可以是类 B 的基类一样，B 也可以是 C 的基类。面向对象层次结构可模拟类似的多层关系。一个具有多种动物的动物园模拟程序，可能会将每种动物作为 Animal 类的子类，如图 5-8 所示。

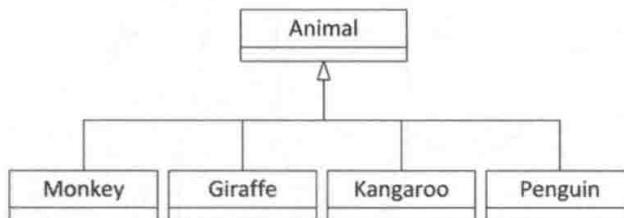


图 5-8 将每种动物作为 Animal 类的子类

当编写每个派生类的代码时，许多代码可能是相似的。此时，应该考虑让它们拥有共同的父类。Lion 和 Panther 的移动方式和食物相同，说明可使用 BigCat 类。还可进一步将 Animal 类细分，以包括 WaterAnimal 和 Marsupial。图 5-9 显示了利用这种共性的更趋层次化的设计。

生物学家看到这个层次结构可能会失望——海豚(Dolphin)和企鹅(Penguin)并不属于同一科。然而，这强调了一个要点——在代码中，需要平衡现实关系和共享功能关系。即使现实中两种事物紧密联系，在代码中也可能没有任何关系，因为它们没有共享功能。可简单地把动物分为哺乳动物和鱼类，但是这会使基类没有任何共同因素。

另一个要点是可用其他方法创建这个层次结构。前面的设计基本上是根据动物的移动方式创建的。如果根据动物常吃的食物或身高创建，层次结构可能会大不相同。总之，关键在于如何使用类，需求决定对象层次结构的设计。

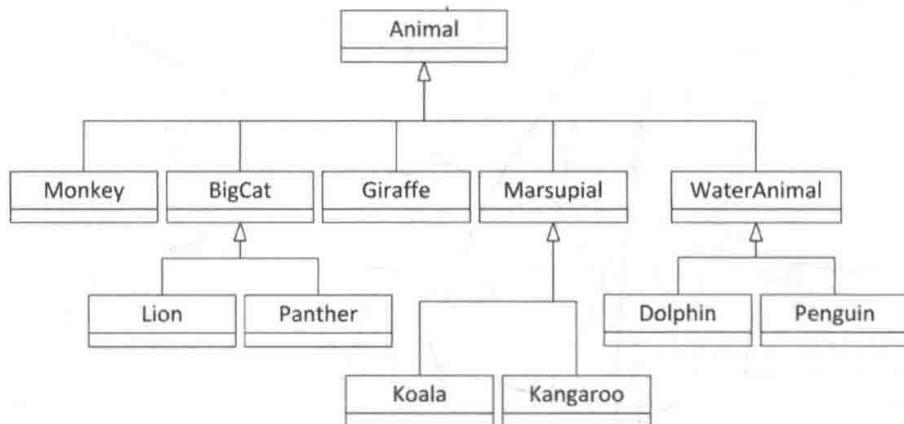


图 5-9 利用这种共性的更趋层次化的设计

优秀的面向对象层次结构能做到以下几点：

- 使类之间存在有意义的功能联系。
- 将共同的功能放入基类，从而支持代码重用。
- 避免子类过多地重写父类的功能，除非父类是一个抽象类。

#### 5.4.6 多重继承

到目前为止，所有示例都只有单一的继承链。换句话说，给定的类最多只有一个直接的父类。事实并非一直如此，在多重继承中，一个类可以有多个基类。

图 5-10 给出了一种多重继承设计。在此仍然有一个基类 Animal，根据大小对这个类进行细分。此外根据食物划分了一个独立的层次类别，根据移动方式又划分了一个层次类别；所有类型的动物都是这三个类的子类。

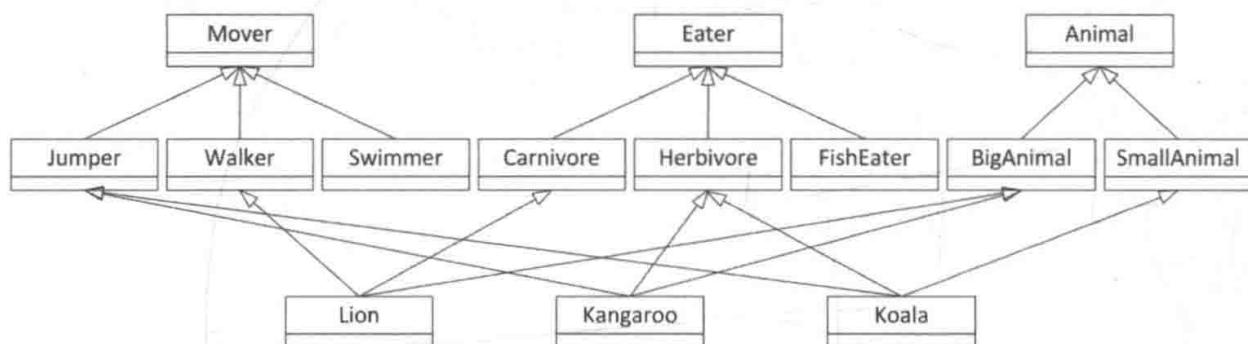


图 5-10 多重继承设计

考虑用户界面环境，假定用户可单击某张图片。这个对象好像既是按钮又是图片，因此其实现同时继承了 Image 类和 Button 类，如图 5-11 所示。

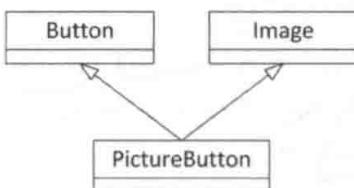


图 5-11 同时继承 Image 类和 Button 类

某些情况下多重继承可能很有用，但必须记住它也有很多缺点。许多程序员不喜欢多重继承，C++ 明确支持这种关系，而 Java 语言根本不予支持，除非通过多个接口来继承(抽象基类)。多重继

承的批评者一般有以下几个原因。

首先，用图形表示多重继承十分复杂。如图 5-10 所示，当存在多重继承和交叉线时，即使简单的类图也会变得非常复杂。类层次结构旨在让程序员更方便地理解代码之间的关系。而在多重继承中，类可有多个彼此没有关系的父类。将这么多类加入对象的代码中，能跟踪发生了什么吗？

其次，多重继承会破坏清晰的层次结构。在动物示例中，使用多重继承方法意味着 Animal 基类的作用降低，因为描述动物的代码现在分成三个独立的层次。尽管图 5-10 中的设计显示了三个清晰的层次，但不难想象它们会变得如何凌乱。例如，如果发现所有的 Jumper 不仅以同样的方式移动，还吃同样的食物，该怎么办？由于层次是独立的，因此无法在不添加其他派生类的情况下加入移动和食物的概念。

最后，多重继承的实现很复杂。如果两个基类以不同方式实现了相同的行为，该怎么办？两个基类本身是同一个基类的子类，可以这样吗？这种可能让实现变得复杂，因为在代码中建立这样错综复杂的关系会给作者和读者带来挑战。

其他语言取消多重继承的原因是，通常可以避免使用多重继承。在设计某个项目时，重新考虑层次结构，通常可避免引入多重继承。

#### 5.4.7 混入类

混入(mixin)类代表类之间的另一种关系。在 C++ 中，混入类的语法类似于多重继承，但语义完全不同。混入类回答“这个类还可以做什么”这个问题，答案经常以“-able”结尾。使用混入类，可向类中添加功能，而不需要保证是完全的“是一个”关系。可将它当作一种分享(share-with)关系。

回到动物园示例，假定想引入某些动物可以“被抚摸”这一概念。也就是说，动物园的游客可以抚摸一些动物，大概率不会被咬伤或伤害。所有可以被抚摸的动物支持“被抚摸”行为。由于可以被抚摸的动物没有其他的共性，且不想破坏已经设计好的层次结构，因此 Pettable 就是很好的混入类。

混入类经常在用户界面中使用。可以说 Image 能够点击(Clickable)，而不需要说 PictureButton 类既是 Image 又是 Button。桌面上的文件夹图标可以是一张可拖动(Draggable)、可点击(Clickable)的图片(Image)。软件开发人员总是喜欢弄一大堆有趣的形容词。

混入类和基类之间的区别更多地取决于对类的看法，而不是代码的区别。因为范围有限，混入类通常比多层次结构容易理解。Pettable 混入类只是在已有类中添加了一个行为，Clickable 混入类或许仅添加了“按下鼠标”和“释放鼠标”行为。此外，混入类很少会有庞大的层次结构，因此不会出现功能的交叉混乱。第 32 章将详细介绍混入类。

### 5.5 本章小结

本章讲述了面向对象程序的设计，没有给出太多代码。本章的概念几乎适用于全部的面向对象语言。有些内容你可能已经知道，有些内容是用新的方法阐述熟悉的概念。或许你会找到解决旧问题的新方法，或者对于你一直在团队中宣讲的概念，本章提供了有利的新论据。即使读者从来没有用过或者只是少量用过对象，现在对设计面向对象程序相关知识的了解也不比很多经验丰富的 C++ 程序员少。

要重点关注对象之间的关系，因为组织良好的对象不仅有助于代码重用并减少混乱，而且因为你可能在团队中工作。以有意义的方式联系在一起的类易于阅读和维护。当设计程序时，可以参阅 5.4 节。

下一章继续讨论设计主题，介绍如何设计可重用代码。

## 5.6 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，然后再从网站上查看解决方案。

**练习 5-1** 假设你想写一个赛车游戏。你需要对某种汽车建模，在本练习假设中只有一种类型的汽车。该车的每个实例都需要多个属性，例如发动机的当前功率输出、当前燃油使用量、轮胎压力、驾驶灯是否打开、挡风玻璃雨刮器是否启动等。游戏应该允许玩家配置不同的引擎、不同的轮胎、定制的驾驶灯和挡风玻璃雨刷等。你会如何对这样一辆车进行建模，为什么？

**练习 5-2** 继续练习 5-1 中的赛车游戏，你当然希望能够支持人工驾驶汽车，但也希望包含对人工智能(AI)驱动汽车的支持。你将如何在游戏中对此进行建模？

**练习 5-3** 假设一个人力资源(HR)应用程序的一部分具有以下 3 个类。

- **Employee:** 关注雇员 ID、薪水、入职日期等
- **Person:** 关注姓名和地址
- **Manager:** 关注团队中有哪些雇员

你认为图 5-12 中的类图怎么样？你能对它进行一些改动吗？该图没有显示不同类的任何属性或行为，因为这是练习 5-4 的主题。

**练习 5-4** 从练习 5-3 的最终类图开始，在类图中添加一些行为和属性。最后，对经理管理员工团队这一事实进行建模。

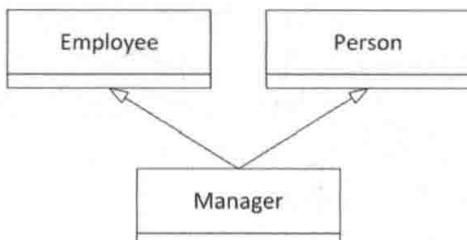


图 5-12 类图

# 第6章

## 设计可重用代码

### 本章内容

- 重用哲学：为什么要设计可重用代码
- 如何设计可重用代码
- 如何使用抽象
- 构建可重用代码的策略
- 设计可用接口的六大策略
- 如何权衡通用性和易用性
- SOLID 原则

正如第4章提到的，在程序中，重用库和其他代码是一项重要的设计策略。然而，这只是重用策略的一半，另一半是设计并编写在程序中可重用的代码。你可能已经发现，设计良好的库和设计不当的库之间存在显著差别。设计良好的库用起来很舒服，而设计糟糕的库可能会促使你厌恶地放弃，并自己编写代码。无论是编写供其他程序员使用的库，还是仅仅设计某个类层次结构，在设计代码时都应该考虑重用。你永远不知道后续项目什么时候会用到相似的功能段。

第4章介绍了重用的设计主题，并阐述了如何通过在设计中整合库和其他代码来应用这个主题，但没有解释如何设计可重用的代码，这是本章的主题。这一内容建立在第5章介绍的面向对象设计原则基础之上。

### 6.1 重用哲学

应该设计自己和其他程序员都可以重用的代码。这条规则不仅适用于专供其他程序员使用的库和框架，还适用于程序中用到的任何类、子系统和组件。你应该牢记以下格言：

- 编写一次，经常使用
- 不惜一切代价避免代码重复
- DRY(Don't Repeat Yourself，不要重写自己写过的代码)

这有几个原因：

- 代码不大可能只在一个程序中使用。代码总会以某种方式重用，因此一开始就应该正确设计。

- **重用设计可节约时间和金钱。**如果设计代码时以某种方式妨碍了将来的使用，当遇到相似的功能需求时，你或你的伙伴必然会花时间重复工作。
- **团队中的其他程序员必须能够使用你编写的代码。**你可能并非独自完成某个项目。同事将感谢你提供设计良好、功能集中的库和代码段供他们使用。重用设计也可以称为协作编程。
- **缺乏重用性会导致代码重复，代码重复会导致维护困难。**如果在重复的代码中发现 bug，则必须在所有使用的位置进行修改。如果发现自己在复制和粘贴一段代码，至少要考虑把它们移入一个辅助函数或类。
- **你自己是主要受益人。**经验丰富的程序员永远不会扔掉代码。随着时间的推移，他们会创建一个不断发展的个人工具库。你永远不会事先知道在什么时候会用到类似的功能段。

**警告：**

公司员工设计或编写代码时，通常拥有知识产权的是公司而不是员工本人。当员工终止劳动合同时，保留设计或代码的副本通常是违法的。如果你是为客户工作的个体户，上述规则同样成立。

## 6.2 如何设计可重用代码

可重用代码有两个主要目标：

- 首先，代码必须通用，可用于稍有不同的目的或不同的应用程序领域。涉及特定应用程序细节的组件很难被其他程序重用。
- 其次，可重用代码还应该易于使用，不需要花费大量时间去理解它们的接口或功能。程序员必须能够方便地将这些代码整合到他们的应用程序中。

将库“递交”给客户的方法也很重要。可以源代码形式递交，这样客户只需要将源代码整合到他们的项目中。另一种选择是递交一个静态库，客户将该库链接到他们的应用程序；也可以给 Windows 客户递交一个动态链接库(DLL)，给 Linux 客户递交一个共享对象(.so)。这些递交方式会对编写库的方式施加额外限制。

**注意：**

本章用术语“客户”代表使用接口的程序员。不要将客户与使用程序的“用户”混淆。本章还使用了短语“客户代码”，这代表使用你的接口的那部分代码。

对于设计可重用代码而言，最重要的策略是抽象。

### 6.2.1 使用抽象

抽象的关键是有效地将接口与实现分离。实现是用来完成任务的代码，接口是其他用户使用代码的方式。在 C 语言中，描述所编写库中函数的头文件是一个接口。在面向对象编程中，类的接口是可公开访问的方法和属性的集合。然而，一个好的接口应该只包含公共方法。类的属性永远不应该公开，但是可以通过公有方法公开，这些方法也称为获取器和设置器。

第 4 章介绍了抽象的原则，并给出了一个电视的真实类比，你可以通过它的接口使用它，而不需要了解它内部是如何工作的。同样，在设计代码时，应该将接口与实现明确分开。这种分离使代码更容易使用，主要是因为客户不需要了解内部实现细节就可以使用该功能。

使用抽象对于自己和使用代码的客户都有好处。客户会获得好处，因为他们不需要担心实现细节，他们可利用你提供的功能，而不必理解代码的实际运行方式。你会获得好处，是因为可修改底层的代

码，而不需要改变接口。这样就可升级和修订代码，而不需要客户改变他们的用法。如果使用动态链接库，客户甚至不需要重新生成可执行程序。总之，都获得了好处是因为作为库的编写者，可在接口中明确指定希望的交互方式和支持的功能。第3章讨论了如何编写文档。接口与实现的明确分离可杜绝用户以你不希望的方式使用库，而这些方式可能导致意想不到的行为和bug。

#### 警告：

当设计接口时，不要向客户公开实现细节。

有时为将某个接口返回的信息传递给其他接口，库要求客户代码保存这些信息。这一信息有时称为句柄(handle)，经常用来跟踪某些特定的实例，这些实例调用时的状态需要被记住。如果库的设计需要句柄，不要公开句柄的内部情况。可将句柄放入某个不透明类，程序员不能直接访问这个类的内部数据成员，也不能通过公有的获取器或设置器来访问。不要要求客户代码修改此句柄内的变量。一个不良设计的示例是，一个库为了启用错误日志，要求设置某个结构的特定成员，而这个结构所在的句柄本来应该是不透明的。

#### 注意：

遗憾的是，编写类时，C++对于抽象原则并不友善，其语法要求将 public 接口和非公有(private 或 protected)数据成员以及方法集中到类定义中，从而向客户公开类的一些内部实现细节。第9章将讲述与此有关的处理技巧，以提供清晰的接口。

抽象非常重要，应该贯穿于整个设计。在做出任何决定时，都应该确保满足抽象原则。将自己摆在客户的位置，判断是否需要接口的内部实现知识。这样将很少(甚至永远不)违背这个原则。

当使用抽象设计可重用代码时，你需要注意以下两点：

- 首先，你必须适当地构造代码。要使用怎样的类层次结构？应该使用模板吗？应该如何将代码划分为子系统？
- 其次，必须设计接口，这些接口是访问你提供的功能的库的“入场券”。

这两个主题将在接下来的小节中讨论。

## 6.2.2 构建理想的重用代码

在开始设计时就应该考虑重用，在所有级别上都要考虑，从单个函数、类乃至整个库或框架。在下文中，所有这些不同的级别称为组件。下面的策略有助于恰当地组织代码，注意所有这些策略都关注代码的通用性。设计可重用代码的另一方面是提供易用性，这与接口设计有紧密的联系，本章后面将讨论这些内容。

### 1. 避免组合不相干的概念或者逻辑上独立的概念

当设计组件时，应该关注单个任务或一组任务，即“高内聚”，也称为 SRP (Single Responsibility Principle，单一责任原则)。不要将无关概念组合在一起，例如随机数生成器和 XML 解析器。

即使设计的代码并不是专门用来重用的，也应该记住这一策略。完整的程序本身很少会被重用，但是程序的片段或子系统可直接组合到其他应用程序中，也可以在稍作变动后用于大致相同的环境。因此，设计程序时，应将逻辑独立的功能放到不同的组件中，以便在其他程序中使用。其中的每个组件都应当有明确定义的责任。

这个编程策略模拟了现实中可互换的独立部分的设计原则。例如，可编写一个 Car 类，在其中放入引擎的所有属性和行为。但引擎是独立组件，未与小汽车的其他部分绑定。可将引擎从一辆小汽车

卸下，安装在另一辆小汽车中。合理的设计是添加一个 Engine 类，其中包含与引擎相关的所有功能。此后，Car 实例将包含 Engine 实例。

### 将程序分为逻辑子系统

将子系统设计为可单独重用的分立组件，即“低耦合”。例如，如果设计一款网络游戏，应该将网络和图形用户界面放在独立的子系统中，这样就可以重用一个组件，而不会涉及另一个组件。例如，假定要编写一款单机游戏，就可以重用图形界面子系统，但是不需要网络功能。与此类似，可以设计一个 P2P 文件共享程序，在此情况下可重用网络子系统，但是不需要图形用户界面功能。

每个子系统都应该遵循抽象原则。将每个子系统当作微型库，必须为其提供清晰的、便于使用的接口。即使你是使用这个微型库的唯一程序员，设计良好的接口和从逻辑上分离不同功能的实现也是有益的。

### 用类层次结构分离逻辑概念

除了将程序分为逻辑子系统以外，在类级别上应该避免将无关概念组合在一起。例如，假定要为自动驾驶汽车编写类。你决定首先编写小汽车的基本类，然后在其中直接加入所有自动驾驶逻辑。但是，如果程序中只需要非自动驾驶汽车，该怎么办？此时，与自动驾驶相关的所有逻辑都会失效，而程序必须与本可避开的库(如 vision 库和 LIDAR 库)链接。解决方案是创建一个类层次结构(见第 5 章)，将自动驾驶汽车作为普通汽车的一个派生类。这样，如果不需要自动驾驶功能，就可在程序中使用 Car 基类，从而避免无谓地增加算法成本。图 6-1 显示了这个类层次结构。

如果有两个逻辑概念，如自驾和小汽车，这个策略效果不错。如果有 3 个或更多个逻辑概念，将变得更复杂。例如，假设要提供卡车和小汽车，每种都有自动驾驶和非自动驾驶之分。从逻辑上讲，Car 和 Truck 都是 Vehicle 的特例，因此应当是 Vehicle 类的派生类，如图 6-2 所示。

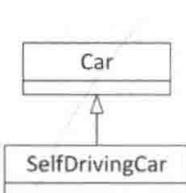


图 6-1 类层次结构

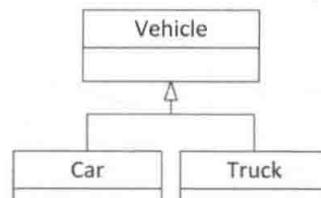


图 6-2 Vehicle 类的派生类

与此类似，可将自动驾驶类作为非自动驾驶类的派生类。无法使用线性层次结构进行分离。一种可能的做法是使自动驾驶功能成为混入类。前一章展示了一种使用多重继承在 C++ 中实现混入类的方法。例如，PictureButton 可以继承自 Image 类和 Clickable 混入类。然而，对于自动驾驶设计，最好使用不同类型的混入类实现，这种实现使用类模板。基本上，SelfDrivable 混入类可以定义如下。这个例子在类模板语法方面有点超前，语法将在第 12 章详细讨论，但这些细节对于后面的讨论并不重要。

```

template <typename T>
class SelfDrivable : public T
{
};
  
```

这个 SelfDrivable 混入类提供了实现自动驾驶功能所需的所有算法。一旦有了这个 SelfDrivable 混入类模板，就可以按如下方式实例化一个用于汽车的模板和一个用于卡车的模板：

```

SelfDrivable<Car> selfDrivingCar;
SelfDrivable<Truck> selfDrivingTruck;
  
```

这两行代码的结果是，编译器将使用 `SelfDriveable` 混入类模板创建一个实例，其中类模板中的所有 `T` 都由 `Car` 替换，因此由 `Car` 派生。另一个实例中，`T` 由 `Truck` 替换，因此由 `Truck` 派生。第 32 章将详细介绍混入类。

这个解决方案需要你编写 4 个不同的类，但为了明确分离功能，这是值得的。

与类级别一样，还应避免在任何设计级别组合不相关的概念，也就是说，要追求高内聚。例如，在方法级别，一个方法不应当执行逻辑上无关的事项，不应当将修改(设置)和查看(获取)混合，等等。

### 用聚合分离逻辑概念

第 5 章讨论的聚合(Aggregation)模拟了“有一个”关系：为完成特定功能，对象会包含其他对象。当不适合使用继承方法时，可以使用聚合分离没有关系的功能或者有关系但独立的功能。

例如，假定要编写一个 `Family` 类来存储家庭成员。显然，树数据结构是存储这些信息的理想结构。不应该把树数据结构的代码整合到 `Family` 类中，而是应该编写一个单独的 `Tree` 类。然后 `Family` 类可以包含并使用 `Tree` 实例。用面向对象的术语来说，`Family` 有一个 `Tree`。通过这种方法，可以在其他程序中方便地重用树数据结构。

### 消除用户界面依赖

如果是一个操作数据的库，就需要将数据操作与用户界面分离开来。这意味着对于这些类型的库，不应该假定哪种类型的用户界面会使用库，因此不应该使用任何标准输入输出流，例如 `cout`、`cerr` 和 `cin`，因为如果在图形用户界面的环境下使用库，这些概念将没有意义。例如，基于 Windows GUI 的应用程序通常不会有任何形式的控制台 I/O。即使库只用于基于 GUI 的程序，也不应该向终端用户弹出任何类型的消息窗口或者其他类型的提示，这是客户代码需要做的事情。这种类型的依赖不仅降低了重用性，还阻止了客户代码正确响应错误，例如在后台将错误处理掉。

第 4 章介绍的 Model-View-Controller(MVC)范式是一种将数据存储和数据显示分开的著名设计模式。在这种范式中，模型可放在库中，客户代码可提供视图和控制器。

## 2. 对泛型数据结构和算法使用模板

C++模板的概念允许以类型或类的形式创建泛型结构。例如，假定为整型数组编写了代码。如果以后要使用 `double` 数组，就需要重写并复制所有代码。模板的概念就是将类型变成一个需要指定的参数，这样就可以创建一个适用于任何类型的代码体。模板允许编写适用于任何类型的数据结构和算法。

最简单的示例是第 1 章介绍的 `std::vector` 类，这个类是 C++ 标准库的一部分。为创建整型的 `vector`，可编写 `std::vector<int>`，为创建 `double` 类型的 `vector`，可编写 `std::vector<double>`。模板编程通常功能强大，但非常复杂。幸运的是，可以创建相对简单的、使用类型作为参数的模板用例。第 12 和 26 章讲述编写自定义模板的技巧，本节讨论模板的一些重要设计特征。

只要有可能，就应该设计泛型(而不是局限于某个特定程序的)数据结构和算法。不要编写只存储 `book` 对象的平衡二叉树结构，要使用泛型，这样就可以存储任何类型的对象。通过这种方法，可将其用于书店、音乐商店、操作系统或需要平衡二叉树的任何地方。这个策略是标准库的基础，标准库提供可用于任何类型的泛型数据结构和算法。

### 模板优于其他泛型程序设计技术的原因

模板不是编写泛型数据结构的唯一机制。在 C 和 C++ 中，可通过存储 `void*` 指针而不是特定类型的指针来编写泛型数据结构。通过将类型转换为 `void*`，客户可用这个结构存储他们想要的任何类型。然而，这种方法的主要问题在于这不是类型安全的：容器无法检测或强制指定所存储元素的类型。可

将任何类型转换为 `void*`，存储在这个结构中。当从这个数据结构中删除指针时，必须将它们转换为对应的类型。由于没有类型检测，结果可能是灾难性的。考虑这样一种情况，某个程序员在一个数据结构中存储了指向 `int` 的指针(通过将其转换为 `void*`的方法)，但另一个程序员认为这是指向 `Process` 对象的指针。第二个程序员愉快地将 `void*`指针转换为 `Process*`指针，并试图将其当作 `Process*`对象使用指针。不必多说，这个程序不会像预期的那样运行。

从 C++17 开始，可以使用 `std::any` 类而不是 `void*`指针来实现不基于模板的泛型数据结构。`any` 类将在第 24 章讨论，但现在知道在 `any` 类的实例中可以存储任意类型的对象就已经足够了。`std::any` 的底层实现中的确在某些情况下使用了 `void*`指针，但它也跟踪存储的类型，因此所有内容都保持类型安全。

另一种方法是为特定的类编写数据结构。通过多态性，这个类的所有子类都可存储在这个结构中。Java 将这种方法发挥到极致，指定所有的类都从 `Object` 类直接或间接派生。早期 Java 版本的容器存储 `Object`，因此可以存储任何类型的对象。然而，这种方法也不是真正类型安全的。从容器中删除某个对象时，必须记得其真实的类型，并向下转换(down-cast)为合适的类型。向下转换意味着转换为类层次结构中更具体的类，即沿着类层次结构“向下”转换。

另一方面，只要使用正确，模板就是类型安全的。模板的每个实例都只存储一种类型，如果试图在同一个模板实例中存储不同的类型，程序将无法编译。此外，模板允许编译器为每个模板实例生成高度优化的代码。与 `void*`和 `std::any` 相比，模板还避免了在自由存储区中的内存分配，从而获得了更好的性能。与 C++ 模板一样，较新的 Java 版本支持类型安全的泛型概念。

### 模板的问题

模板并不是完美的。首先，其语法令人迷惑，对于没有用过模板的人而言更是如此。其次，模板要求相同类型的数据结构，在一个结构中只能存储相同类型的对象。也就是说，如果编写了一个平衡二叉树的类模板，可以创建一个树对象来存储 `Process` 对象，创建另一个树对象来存储 `int`。不能在同一棵树中同时存储 `int` 和 `Process`。这个限制是由模板的类型安全性质决定的。从 C++17 开始，可以采用一种标准方式来绕过这种“相同类型”限制。可编写数据结构来存储 `std::variant` 或 `std::any` 对象。`std::any` 对象可存储任意类型的值，`std::variant` 对象可存储所选类型的值。第 24 章将详细介绍 `any` 和 `variant` 类。

模板的另一个可能的缺点是所谓的代码膨胀：最终二进制代码的大小增加。每个模板实例化的代码比稍慢一点的泛型代码需要更多的大小。然而，如今代码膨胀通常不是什么大问题。

### 模板与继承

程序员有时发现，难以决定使用模板还是继承。在此有一些提示，有助于做出决定。

如果打算为不同的类型提供相同的功能，则使用模板。例如，如果要编写一个适用于任何类型的泛型排序算法，应该使用模板。如果要创建一个可以存储任何类型的容器，应该使用模板。关键的概念在于模板化的类或算法会以相同方式处理所有类型。但是，如有必要，可针对特定的类型对模板进行特化，以区别对待这些类型。模板特化参见第 12 章。

当需要提供相关类型的同行为时，应该使用继承。例如，在图形绘制程序中，使用继承来支持不同的图形，例如圆、正方形、直线等。然后，特定形状派生自一个基类，例如 `Shape` 类。

现在可以把二者结合起来。可以编写一个模板基类，此后从中派生一个模板类。第 12 章将详细讲述模板语法。

### 3. 提供适当的检测和安全措施

编写安全代码有两种截然不同的方法。最优的编程方法可能是适当地混合使用这两种方法。第一种方法是按契约设计，这表示函数或类的文档是契约，详细描述客户代码的作用以及你的函数或类的作用。按契约设计有3个重要方面：前置条件(precondition)、后置条件(postcondition)和不变量(invariant)。前置条件列出为调用函数或方法客户代码必须满足的条件。后置条件列出完成执行后，函数或方法必须满足的条件。最后，不变量列出在函数或方法执行期间必须一直满足的条件。

这种方法常用于标准库。例如，`std::vector` 定义了一个契约，以使用数组标记获取 `vector` 中的某个元素。契约指定，`vector` 不进行边界检查，这是客户代码的责任。也就是说，使用数组标记从 `vector` 获取元素的前置条件是给定索引是有效的。这样可提高客户代码的性能，因为客户代码知道其索引在指定范围内。

第二种方法是以尽可能安全的方式设计函数和类。这个指导方针的主要特征就是在代码中执行错误检测。例如，如果随机数生成器需要一个处于指定范围的种子，不要相信用户一定会正确地传递一个有效的种子。应该检测传递过来的值，如果无效，就拒绝调用。例如，除了按契约设计的数组标记可以获取元素之外，`vector` 还定义了一个 `at()` 方法，用来获取进行边界检查的特定元素。如果用户提供的索引无效，该方法将引发异常。因此，客户代码可以选择是使用不带边界检查的数组标记，还是使用带边界检查的 `at()` 方法。`vector` 还定义了 `at()` 方法，所以客户代码可以选择是使用不带边界检查的数组标记，还是使用带有边界检查的 `at()` 方法。

可以用准备所得税申报的会计师做类比。在美国，雇用一位会计师，给他提供当年所有财务信息，会计师用这些信息填写 IRS 表单。然而，会计师并不是盲目地将这些信息填写在表单上，而是要确认所填的信息有意义。例如，如果你有一栋房子，但忘了说明支付的房产税，会计师将提醒你提供这一信息。同样，如果说支付了\$12 000 的按揭利息，但你的收入只有\$15 000，会计师就会问你是否提供了正确的数字(或者至少会建议你更换负担得起的房子)。

可将会计师当作“程序”，财务信息就是输入，所得税申报表就是输出。然而，会计师的价值不仅是填写表单，选择雇用会计师的另一个原因是它可以提供检测和安全措施。在编程中也与此类似，在实现中应该尽可能提供检测和安全措施。

有些技巧和语言特征有助于编写安全代码，有助于在程序中加入检测和安全措施。为了向客户代码报告错误，可返回错误代码或特定的值，例如 `false`、`nullptr` 或第 1 章介绍的 `std::optional`。此外，还可以抛出异常，以提醒客户代码发生了错误，第 14 章将详细讲述异常。

### 4. 扩展性

设计的类应当具有扩展性，可通过从这些类派生其他类来扩展它们。不过，设计好的类应当不再修改，也就是说，其行为应当是可扩展的，而不必修改其实现。这称为开放/关闭原则(Open/Closed Principle, OCP)。

例如，假设开始实现绘图应用程序。第一个版本只支持绘制正方形，设计中包含两个类：`Square` 和 `Renderer`。前者包含正方形的定义，如边长；后者负责绘制正方形。得到的代码如下：

```
class Square { /* Details not important for this example. */ };
class Renderer
{
public:
    void render(const vector<Square>& squares)
    {
        for (auto& square : squares) { /* Render this square object... */ }
    }
};
```

接下来添加对绘制圆的支持，因此创建 Circle 类：

```
class Circle { /* Details not important for this example. */ };
```

为绘制圆，必须修改 Renderer 类的 render() 方法。你决定将其改为：

```
void Renderer::render(const vector<Square>& squares,
                      const vector<Circle>& circles)
{
    for (auto& square : squares) { /* Render this square object... */ }
    for (auto& circle : circles) { /* Render this circle object... */ }
}
```

此时，你会感觉哪里有些不对，你的感觉是对的！为了扩展这个功能使其支持绘制圆，必须修改现有的 render() 方法的实现，即没有关闭修改。

在这种情况下，你应该使用继承。这个示例对继承语法的使用稍微超前，第 10 章将讨论继承，不过，对于这个示例而言，重点并不在于理解语法细节。此时，只需要知道，以下语法指定 Square 从 Shape 类派生而来：

```
class Square : public Shape {};
```

下面是使用了继承语法的设计：

```
class Shape
{
public:
    virtual void render() = 0;
};

class Square : public Shape
{
public:
    void render() override { /* Render square... */ }
    // Other members not important for this example.
};

class Circle : public Shape
{
public:
    void render() override { /* Render circle... */ }
    // Other members not important for this example.
};

class Renderer
{
public:
    void render(const vector<Shape*>& objects)
    {
        for (auto* object : objects) { object->render(); }
    }
};
```

在这种设计中，如果想要添加对新类型形状的支持，只需要编写一个新类，该类从 Shape 类派生而来，并且实现了 render() 方法。不需要对 Renderer 类做任何修改。因此，可在不修改现有代码的前提下扩展设计，即它对扩展是“开放”的，对修改是“关闭”的。

### 6.2.3 设计有用的接口

除了正确地抽象和构建代码之外，重用设计要求关注与程序员交互的接口。如果实现非常优秀、高效，但接口很糟糕，库就不能算是良好的。

即使不想把程序中的每个组件都应用到多个程序中，也应该让它们具有良好的接口。首先，不知道什么时候会重用其中的一些内容。其次，即使只使用一次，良好的接口也很重要，特别是在团队中编程，并且其他程序员必须使用你设计和编写的代码时。

在 C++ 中，类的属性和方法可以是公有的(public)、受保护的(protected)和私有的(private)。将属性或行为设置为 public 意味着其他代码可以访问它们。protected 意味着其他代码不能访问这个属性或行为，但子类可以访问。private 是最严格的控制，意味着不仅其他代码不能访问这个属性或行为，子类也不能访问。注意，访问修饰符在类级别而非对象级别发挥作用。例如，这意味着类的方法可访问同一个类的其他对象的 private 属性或 private 方法。

设计公开的接口就是选择哪些应该成为 public，应该将设计公开接口看作一个过程。接口的主要目的是使代码易于使用，但是一些接口技术也可以使其遵循通用性的原则。

#### 1. 考虑用户

设计公开接口的第一步是考虑为谁设计。用户是团队中的其他成员吗？这个接口只是个人使用吗？公司外面的程序员会使用接口吗？是某个用户还是国外的承包商？除了判断谁会用到接口以外，还应该注意设计目标。

如果接口供自己使用，那么设计起来会更灵活，为满足自己的需要可加以改变。然而，应该记住团队中的角色会改变，很有可能在某一天，其他人也会用这个接口。

设计供公司内其他程序员使用的接口有一点不同之处。某种意义上，接口变成你与他们之间的契约。例如，如果你实现程序的数据存储组件，其他人依靠这个接口支持某些操作。你应该找出团队中其他成员需要你的类完成的所有工作。他们需要版本控制吗？可以存储什么类型的数据？作为契约，接口应该看成几乎不可改变的。如果在开始编码之前就接口达成一致，而你在开始编码之后改变了接口，就会听到许多抱怨。

如果用户是外部客户，那么理想情况下，目标客户会参与指定接口公开的功能，正如为公司内部用户设计接口一样。你应该同时考虑用户需要的特定功能和他们将来可能需要的功能。接口中使用的术语必须是客户熟悉的，并且编写文档时必须考虑到用户。设计中不应该出现内部的笑话、代号和程序员的俚语。

接口的用户也会影响你在设计上投入的时间。例如，如果正在设计一个只包含几个方法的接口，而这些方法只被少数用户在少数几个地方使用，那么以后修改该接口是可以接受的。但是，如果正在设计一个复杂的接口或一个将被许多用户使用的接口，那么应该在设计上花费更多的时间，并尽最大努力避免在用户开始使用之后再对其进行修改。

#### 2. 考虑目的

编写接口有很多理由。在编写代码前，甚至在决定要公开的功能之前，必须理解接口的目的。

##### 应用程序编程接口

应用程序编程接口(API)是一种外部可见机制，用于扩展产品或者在其他环境中使用其功能。如果说内部的接口是契约，那么 API 更接近于不可更改的法律。当甚至不是公司员工的用户开始使用你的 API 时，他们不希望 API 发生改变，除非加入帮助他们的新功能。在交给用户使用之前，应该关

心 API 的设计，并与用户讨论。

设计 API 时主要考虑易用性和灵活性。由于接口的目标用户并不熟悉产品内部的运行方式，因此学习使用 API 是一个循序渐进的过程。毕竟，公司向用户公开这些 API 的目的是想让用户使用 API。如果使用难度太大，API 就是失败的。灵活性常与此对立，产品可能有许多不同的用途，我们希望用户能使用产品提供的所有功能。然而，如果一个 API 让用户使用产品支持的任何功能，那么它肯定会上过于复杂。

正如一句编程格言所说，“好的 API 使普通的情况变得容易，使高级的/不大可能的情况变得可能”。也就是说，API 应该很容易使用，支持大多数程序员想要做的事情。然而，API 应该允许更高级的用法，因此在罕见的复杂情况和常见的简单情况之间做出折中是可以接受的。本章后面的“设计易用的接口”一节将详细讨论此策略，并为设计提供一些具体提示。

### 工具类或库

通常，程序员的任务是设计某些特定的功能，供应用程序中的其他部分使用，例如一个日志类。在此情况下接口比较容易确定，因为要公开大多数功能或全部功能，理想情况下不应该给出太多与实现有关的内容。通用性是需要考虑的重要问题，由于类或库是通用的，因此在设计中应该考虑设置可能的用例集合。

### 子系统接口

你可能设计程序中两个主要子系统之间的接口，例如访问数据库的机制。在此情况下，将接口与实现分离很重要，原因如下。

最重要的原因之一是可模拟性。在测试场景中，你希望用接口的一个实现替换接口的另一个实现。例如，在为数据库接口编写测试代码时，你可能不希望访问真实的数据库。可以将访问真实数据库的接口实现替换为模拟所有数据库访问的接口实现。

另一个原因是灵活性。即使在测试场景之外，你也可能希望提供某些接口的几种不同实现，这些实现可以互换使用。例如，你可能希望将使用 MySQL 服务器数据库的数据库接口实现替换为使用 SQL 服务器数据库的实现。你甚至可能希望在运行时在不同的实现之间切换。

还有另一个不那么重要的原因，通过先完成接口，其他程序员可以在你的实现完成之前针对接口开始编程。

在处理子系统时，首先考虑它的主要用途。一旦确定了子系统负责的主要任务，就要考虑具体的用途以及应该如何将其呈现给代码的其他部分。试着设身处地为他们着想，不要陷入实施细节的泥潭。

### 组件接口

我们定义的大多数接口可能都比子系统接口或 API 更小。这些将是你在编写其他代码时使用的类。这些情况下，当接口逐渐增大，变得难以控制时，就可能会出现问题。哪怕这些接口是供自己使用的，也要当成不是。与子系统接口类似，此时应该考虑每个类的主要目的，不要公开对这个目的没有贡献的功能。

## 3. 设计易用的接口

接口应该易于使用。这并不意味着接口是微不足道的，而是说接口应该在功能允许的情况下尽量简洁明了。不能要求库的客户为了使用简单的数据结构就去查找数页源代码，或者改动他们的代码，以获得所需的功能。本小节给出了 4 个具体策略，来帮助设计易于使用的接口。

## 采用熟悉的处理方式

开发易用的接口的最佳策略是遵循标准的、熟悉的做法。当人们遇到的接口与他们过去用过的接口类似时，就能更好地理解这个接口，更容易采用这个接口，出错的可能也更低。

例如，假定设计汽车的操纵机构。这有多种可能：一个操纵杆，左移和右移两个按钮，一个滑动水平杠杆，或者一个古老的方向盘。哪个接口最容易使用？采用哪种接口的汽车销量最好？客户熟悉方向盘，因此，这两个问题的答案都是方向盘。即使开发出另一种性能更好、更安全的操纵机构，也需要花费大量的时间去销售产品，还要教会人们如何使用。当需要在遵循标准接口模型和扩展新方向之间做出选择时，通常最好坚持使用人们熟悉的接口。

创新当然很重要，但应该在底层实现中创新，而不是在接口上。例如，消费者对某些车型的创新型全电动发动机感到兴奋。这些车之所以卖得很好，部分原因是使用它们的接口与配备标准汽油发动机的汽车相同。

回到C++，这个策略表明，开发的接口应该遵循C++程序员熟悉的标准。例如，C++程序员希望构造函数初始化对象，析构函数清理对象(详见第8章)。如果需要“重新初始化”现有对象，标准方法是使用一个新构造的对象为其赋值。当设计类时，应该遵循这个标准。如果要求程序员调用initialize()方法初始化对象，调用cleanup()方法清理对象，而不是将这些功能放在构造函数和析构函数中，就会让使用你的类的所有用户感到迷惑。因为这个类与其他C++类的行为不同，程序员需要花费更长的时间学习如何使用这个类，并可能由于忘记调用initialize()或cleanup()方法而出错。

### 注意：

要一直从使用者的角度考虑接口。这些接口有意义吗？这是你想要的接口吗？

C++提供了一种称为运算符重载的语言特性，帮助为对象开发易于使用的接口。运算符重载允许所编写的类使用标准运算符，就像内建的int和double类型一样。例如，可编写一个Fraction类，这个类可以执行加、减和流操作，如下所示：

```
Fraction f1 { 3, 4 };
Fraction f2 { 1, 2 };
Fraction sum { f1 + f2 };
Fraction diff { f1 - f2 };
cout << f1 << " " << f2 << endl;
```

与之对应的方法调用如下所示：

```
Fraction f1 { 3, 4 };
Fraction f2 { 1, 2 };
Fraction sum { f1.add(f2) };
Fraction diff { f1.subtract(f2) };
f1.print(cout);
cout << " ";
f2.print(cout);
cout << endl;
```

可以看出，运算符重载可为类提供容易使用的接口，但也不要滥用运算符重载。虽然可以重载+运算符以实现减法运算，重载-运算符以实现乘法运算，但此类实现违反常理。当然，这并不意味着每个运算符一定要实现与之前完全一样的行为。例如，string类用+运算符连接字符串，直觉上这个接口就是用于连接字符串的。第9和15章将详细讨论运算符重载。

### 不要忽略必需的功能

在设计接口时，应该考虑将来的需求。你会在这个设计上花费数年时间吗？如果是这样，就可能需要使用插件架构，从而留出扩展空间。能够确定人们使用接口的目的与当初设计的目的相同吗？与他们交流，以更好地理解他们的使用情况。否则以后就要重写接口，或者更糟糕的是，以后可能需要不时地添加新功能，使接口变得凌乱不堪。然而要小心！如果将来的用途不清楚，不要设计意图解决一切问题的日志类，因为这样做会不必要地将设计、实现和公有接口复杂化。

该策略分为两部分。首先，接口应该包括用户可能用到的所有行为。这乍看上去好像是显而易见的。回到汽车示例，汽车绝不会有没有供司机查看速度的里程表！与此类似，Fraction 类绝不会有让客户访问分子值和分母值的机制。

然而，其他行为可能就比较模糊。该策略要求预见客户使用代码的所有方法，如果以某种特定的方式思考接口，当客户以不同方式使用时，就可能遗漏他们所需的功能。例如，假定要设计一个游戏棋盘类，可能只想到了典型的游戏，例如国际象棋或西洋跳棋，于是决定棋盘上的一个位置最多放一个棋子。然而，如果后来要编写一个西洋双陆棋游戏(棋盘上允许多个棋子占据一个位置)，该怎么办？由于排除了这种可能性，该棋盘就无法作为西洋双陆棋的棋盘。

显然，预见库所有可能的用法是很困难的，甚至无法做到。不要强迫自己为了将来的潜在用途而设计一个完美接口。只需要对此给予考虑，并尽力去做即可。

这个策略的第二部分是在实现中包含尽可能多的功能。不能要求客户代码指定在实现中已经知道或可以知道的信息(假设以不同的方式设计)。例如，如果库需要一个临时文件，不要让库的用户指定路径。他们不关心库使用什么文件，应该用其他方法决定合适的临时文件路径。

此外，不能要求库的用户完成不必要的任务以合并结果。如果随机数库采用了独立计算随机数低序位和高序位的算法，在传递给用户之前，应该将这些数字合并。

### 提供整洁的接口

为避免在接口中遗漏功能，某些程序员走向另一个极端：包含可以想到的所有功能。使用这个接口的程序员永远不会没有完成任务的方法。遗憾的是，这个接口可能非常混乱，以至于无法实现。

不要在接口中提供多余的功能，保持接口的简洁。乍看上去，这个指导方针与前面避免遗漏必要功能的策略相背。为避免遗漏功能而包含所有想象到的接口尽管是一个策略，但并不是一个健全的策略。应该包含必要的功能，并忽略不必要甚至起反作用的接口。

回顾汽车示例。开车只需要使用几个组件：方向盘、刹车、油门踏板、换挡、后视镜、里程计和仪表板上的一些其他仪表。现在想象一下，如果汽车的仪表板与飞机的驾驶员座舱类似，具有上百个仪表、控制杆、监控器和按钮，这没法用！开汽车比开飞机容易多了，接口也比较简单，不需要关心海拔高度、与控制塔通话或控制飞机中众多的组件(例如机翼、发动机和起落架)。

此外，从发展的观点看，较小的库容易维护。如果试图让每个人都很快乐，就应该留出更多空间来容纳错误，如果实现非常复杂以至于纠缠不清，哪怕一个错误也能导致库失效。

遗憾的是，设计简洁接口的思想看起来很好，实践起来非常难。这个规则基本上是主观的：由你决定什么是必需的，什么不是。当然，当这个判断出错时，客户一定会通知你。

### 提供文档

无论接口多么便于使用，都应该提供使用文档。如果不告诉程序员如何使用，不能期望他们会正确使用库。应该将库或代码看作供其他程序员使用的产品。产品应该带有说明其正确用法的文档。

提供接口文档有两种方法：接口自身内部的注释和外部的文档。应该尽量提供这两种文档。大多数公开的 API 只提供外部文档：许多标准 UNIX 和 Windows 头文件中都缺少注释。在 UNIX 中，文

档形式通常是名为 man pages 的在线手册。在 Windows 中，文档通常附带在集成开发环境中或互联网上。

虽然多数 API 和库都取消了接口本身的注释，但笔者认为，这种形式的文档才是最重要的。绝不应该给出一个只包含代码的“裸”模块或头文件。即使注释与外部文档完全相同，具有友好注释的头文件也比只有代码的模块或头文件看上去舒服，即使最优秀的程序员也希望经常看到书面语言。

第 3 章给出了关于注释要包括什么以及如何写注释的具体技巧，并且提到了基于接口内注释的外部文档自动生成工具。

#### 4. 设计通用接口

接口应该通用，这样就可用于各种任务。如果在原本通用的接口中编写了针对某个程序的代码，就无法将其用作他用。这里给出一些需要记住的指导方针。

##### 提供执行相同功能的多种方法

为让所有“顾客”都满意，有时可提供执行相同功能的多种方法。然而，应该慎用这种方法，因为过多的应用很容易让接口变得混乱不堪。

再次考虑汽车示例。现在大多数新车都提供遥控开锁系统，可按下钥匙扣上的一个按钮开锁。然而，这些车始终提供标准钥匙，当钥匙扣上的电池没电时可用标准钥匙开锁。尽管多了一种方法，但是大多数消费者喜欢拥有两种选择。

有时在程序接口的设计中也是如此。例如，`std::vector` 提供了两种方法来访问特定索引处的元素。可使用 `at()` 方法，该方法执行边界检查；也可使用数组标记方法，该方法不执行边界检查。如果知道索引是有效的，那么使用数组标记方法更合适，这样可省去使用 `at()` 方法时的边界检查开销。

注意，这一策略应该当作接口设计中“整洁”规则的例外。有些情况下这个例外是恰当的，但大多数情况下应该遵循“整洁”规则。

##### 提供定制

为增强接口的灵活性，可提供定制。定制可以很简单，如允许用户打开或关闭错误日志。定制的基本前提是向每个客户提供相同的基本功能，但给予用户稍加调整的能力。

实现此目的的一种方法是通过使用接口来反转依赖关系，也称为依赖倒置原则 (DIP)。依赖注入是这一原则的一种实现。第 4 章“设计专业的 C++ 程序”简要提到了 `ErrorLogger` 服务的一个示例。应该定义一个 `ErrorLogger` 接口，并使用依赖注入将这个接口注入每个想要使用 `ErrorLogger` 服务的组件中。

通过回调和模板参数，可提供更强大的定制能力。例如，可允许客户设置自己的错误处理例程。第 19 章将详细讨论回调。

标准库将定制策略发挥到极致，允许客户为容器指定自己的内存分配器。如果要使用这些特性，就必须编写一个遵循标准库指导方针和符合接口要求的内存分配器对象。标准库中的每个容器都将分配器作为模板参数，第 25 章将详细讲述。

#### 5. 协调通用性和易用性

易用和通用这两个目标有时相互冲突。通常通用性会使接口变得复杂，例如，假定在一个地图程序中需要一个图结构来存储 city。为了通用性，可能会使用模板编写一个适用于任何类型(而不只是 city)的通用图结构。如果在下一个程序中需要编写一个网络模拟器，就可以使用这个图结构存储网络中的路由器。遗憾的是，由于使用了模板，这个接口会有点笨拙，不太容易使用，当潜在客户不熟悉模板时尤其如此。

然而，通用性和易用性之间并不是互斥的。尽管有时增强通用性会降低易用性，但设计既有通用性又有易用性的接口还是有可能的。

为在提供足够功能的同时降低复杂性，可提供两个独立接口。这称为接口隔离原则(Interface Segregation Principle, ISP)。例如，编写的通用网络库可以具有两个独立方向：一个为游戏提供网络接口，另一个为超文本传输协议(HTTP，一种网络浏览协议)提供网络接口。提供多个接口还可使常用功能易于使用，同时仍提供高级功能选项。回到地图程序，可能要为地图的客户提供独立的接口，用不同的语言指定城市的名称。英语是主流语言，因此可作为默认语言。这样大多数客户不需要担心语言的设置，但有需要的用户可以设置语言。

### 6.2.4 设计成功的抽象

经验和重复是良好抽象的基础。只有经历过多年编写代码和使用抽象，才能真正地设计良好的接口。也可通过标准设计模式，重用已有的、设计优秀的抽象代码，利用他人多年编写和使用抽象的经验。当遇到其他抽象时，试着记住什么可行，什么不可行。你发现上周使用的 Windows 文件系统 API 有什么缺陷？如果你编写一个网络包装器，与同事编写的有什么不同？最好的接口往往并不是一次就能得到的，需要反复尝试。把你的设计交给同行，征求他们的反馈。如果公司有代码评审，可以将检查接口规范作为评审代码的开始，此后再开始实现。在开始编码后，也不要惧怕修改抽象，哪怕这样做意味着强制其他程序员进行改动，他们会意识到良好的抽象可让每个人长期受益。

有时与其他程序员交流自己的设计时，应该传播点儿好消息。或许团队的其他成员没有意识到前面设计的问题，或者他们觉得按照你的方法工作量太大。在此类情况下，要准备好保护所做的工作，并适时与他们沟通。

良好的抽象意味着接口只有公有方法而且足够稳定，不会改变。与此对应的技术称为私有实现习语或 pimpl idiom，详见第 9 章。

小心单一类的抽象。如果编写的代码非常深奥，应该考虑用其他类配合主接口。例如，如果公开一个完成数据处理的接口，那么还要考虑编写一个结果对象，从而提供一种简单的方法来查看并说明结果。

始终将属性转换为方法。换句话说，不要让外部代码直接操作类的数据。不要让一些粗心或怀有恶意的程序员把兔子对象的高度设置为负数，为此可对“设置高度”方法进行边界检查。

值得重申的是迭代，因为这非常重要。应该查找并回应设计中的缺陷，在必要时进行修改，并从错误中吸取教训。

### 6.2.5 SOLID 原则

本章和上一章讨论了很多面向对象设计的基本原则。经常使用易记的首字母缩写词 SOLID 来指代它们。表 6-1 汇总了 SOLID 原则。其中的大多数原则都在本章讨论过；对于本章未讨论的原则，则指明相关的章号。

表 6-1 SOLID 原则汇总

S	SRP(Single Responsibility Principle, 单一责任原则) 单个组件应当具有单独、明确定义的责任，不应当与无关的功能组合在一起
O	OCP(Open/Closed Principle, 开放/关闭原则) 一个类对扩展应当是“开放的”(允许派生新类)，但对修改是“关闭的”

(续表)

L	LSP(Liskov Substitution Principle, 里氏替换原则) 对于一个对象而言, 应当能用该对象的子类型替代该对象的实例, 详见 5.4.3 节 用一个示例解释了该原则, <code>AssociativeArray</code> 和 <code>MultiAssociativeArray</code> 之间是“是一个”关系还是“有一个”关系
I	ISP(Interface Segregation Principle, 接口隔离原则) 接口应当简单清楚。不要使用过于宽泛的通用接口, 最好使用多个较小的、明确定义的、责任单一的接口
D	DIP(Dependency Inversion Principle, 依赖倒置原则) 使用接口倒置依赖关系。依赖注入是支持依赖倒置原则的一种方式, 在本章的前面提到过, 在第 33 章还会提到

## 6.3 本章小结

通过阅读本章, 你学习了设计可重用代码的方法, 学习了重用的哲学(总结为“编写一次, 经常使用”), 还知道了可重用代码应该既有通用性, 又有易用性。为设计可重用代码, 需要使用抽象、选择合理的代码结构并设计良好的接口。

本章给出了创建代码的具体提示: 避免组合无关或逻辑独立的概念, 对泛型数据结构和算法使用模板提供适当的检测和安全措施, 注重扩展性。

关于设计接口, 本章给出了 6 个策略: 采用熟悉的处理方式, 不要忽略必需的功能, 提供整洁的接口, 提供文档, 提供执行相同功能的多种方法, 提供定制。还讨论了如何协调经常发生冲突的通用性和易用性要求。

最后介绍 SOLID 原则, SOLID 是一个易记的首字母缩写词, 描述本章和后续章节讨论的最重要设计原则。

本章结束了本书的第 II 部分: 从较高层次讨论设计主题。下一部分将深入探讨软件工程过程的实现阶段和 C++ 编码细节。

## 6.4 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, 然后再从网站上查看解决方案。

**练习 6-1** 使普通情况变得容易并使不太可能的情况成为可能意味着什么?

**练习 6-2** 设计可重用代码的第一策略是什么?

**练习 6-3** 假设你正在编写一个需要处理人员信息的应用程序。应用程序的一部分需要保留一份客户列表, 例如包含最近的订单、会员卡号等数据的列表。应用程序的另一部分需要跟踪公司员工的 ID、职位等。因此, 设计了一个名为 `Person` 的类, 其中包含他们的姓名、电话号码、地址、最近订单列表、会员卡号、薪水、员工 ID、职位(如工程师、高级工程师等)。你觉得这样的类怎么样? 你能够如何改进它?

**练习 6-4** 不回看之前的内容, 解释 SOLID 的含义。



# 第III部分

## C++编码方法

---

- ▶ 第 7 章 内存管理
- ▶ 第 8 章 类和对象
- ▶ 第 9 章 精通类和对象
- ▶ 第 10 章 揭秘继承技术
- ▶ 第 11 章 零碎的工作
- ▶ 第 12 章 利用模板编写泛型代码
- ▶ 第 13 章 C++ I/O 揭秘
- ▶ 第 14 章 错误处理
- ▶ 第 15 章 C++运算符重载
- ▶ 第 16 章 C++标准库概述
- ▶ 第 17 章 理解迭代器与范围库
- ▶ 第 18 章 标准库容器
- ▶ 第 19 章 函数指针，函数对象，lambda 表达式
- ▶ 第 20 章 掌握标准库算法
- ▶ 第 21 章 字符串的本地化与正则表达式
- ▶ 第 22 章 日期和时间工具
- ▶ 第 23 章 随机数工具
- ▶ 第 24 章 其他库工具



# 第 7 章

# 内 存 管 理

## 本章内容

- 使用和管理内存的不同方式
- 数组和指针之间的复杂关系
- 从底层看内存的使用
- 常见内存陷阱
- 智能指针及其用法

从很多方面看，用 C++ 编程就像没有道路的驾驶。当然，你可以去任何想去的地方，但没有任何限制或交通灯来保护你免受伤害。和 C 语言一样，C++ 语言对程序员采取的是不干预的策略。这个语言假设你知道自己在干什么。C++ 允许你采取一些可能会产生问题的做法，因为 C++ 极度灵活，为了性能而牺牲了安全性。

内存的分配和管理是 C++ 编程中特别容易出错的一个领域。为写出高质量的 C++ 程序，专业的 C++ 程序员需要了解内存幕后的工作原理。作为第 III 部分的首章，本章探讨内存管理的来龙去脉，学习动态内存的陷阱，以及避免和消除它们的一些技术。

本章讨论底层内存处理，因为专业的 C++ 程序员将遇到此类代码。但在现代 C++ 中，应尽可能避免底层内存操作。例如，不应使用动态分配内存的 C 风格数组，而应使用标准库容器，例如 vector，它会自动处理所有内存分配操作。不应使用原始指针，而应使用智能指针，例如 unique\_ptr 和 shared\_ptr，它们会自动释放不再需要的底层资源，例如内存。基本上，应尝试避免在代码中调用内存分配例程，例如 new/new[] 和 delete/delete[]。当然，这并不总是可行的，并且在现有的代码中，很可能并非如此，所以专业 C++ 程序员仍需要了解内存在幕后的工作原理。

### 警告：

在现代 C++ 中，应尽可能避免底层内存操作，而使用现代结构，如容器和智能指针。

## 7.1 使用动态内存

内存是计算机的底层组件，但是，即使在 C++ 这样的高级语言中也仍要面对内存的问题。扎实理解 C++ 动态内存的工作原理对于成为一名专业的 C++ 程序员至关重要。

### 7.1.1 如何描绘内存

如果了解对象在内存中的表现形式，对动态内存的理解就会容易得多。在本书中，内存单元表示为一个旁边带有标签的框。该标签表示这个内存对应的变量名。方框内的数据显示内存当前的值。

例如，图 7-1 显示了执行以下代码后的内存状态。这行代码在一个函数内，因此 i 是一个局部变量：

```
int i { 7 };
```

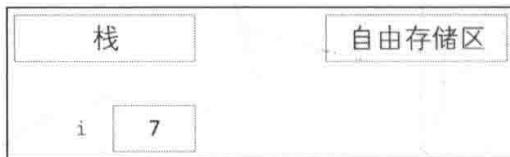


图 7-1 执行代码后的内存状态

i 是在栈上分配的自动变量。当程序流离开作用域(变量在这个作用域中声明)时，会自动释放 i。

使用 new 关键字时，内存分配在自由存储区中。下面的代码在栈上创建了一个变量 ptr，然后在自由存储区上分配内存，ptr 指向这块内存。

```
int* ptr { nullptr };
ptr = new int;
```

也可缩减为一行：

```
int* ptr { new int };
```

图 7-2 显示了执行该代码后内存的状态。注意，变量 ptr 仍在栈上，即使它指向的是自由存储区中的内存。指针只是一个变量，可在栈或自由存储区中，但人们很容易忘记这一点。然而，动态内存总是在自由存储区上分配。

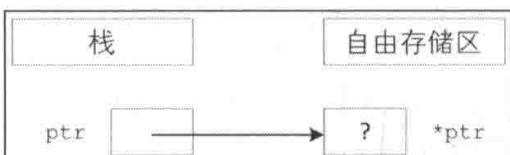


图 7-2 内存状态

#### 警告：

作为经验法则，每次声明一个指针变量时，务必立即用适当的指针或 nullptr 进行初始化！不要令其处于未初始化状态。

下一个例子展示了指针既可在栈中，也可在自由存储区中。

```
int** handle { nullptr };
handle = new int*;
*handle = new int;
```

上面的代码首先声明一个指向整数指针的指针变量 handle。然后，动态分配足够的内存来保存一个指向整数的指针，并将指向这个新内存的指针保存在 handle 中。接下来，将另一块足以保存整数的动态内存的指针保存在 \*handle 的内存位置。图 7-3 展示了这个两级指针，其中一个指针保存在栈中(handle)，另一个指针保存在自由存储区中(\*handle)。

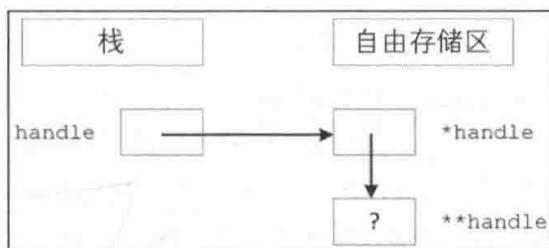


图 7-3 两级指针

### 7.1.2 分配和释放

要为变量创建空间，可使用 `new` 关键字。要释放这个空间给程序中的其他部分使用，可使用 `delete` 关键字。当然，如果 `new` 和 `delete` 这类简单的概念没有一些变化和复杂性，就不是 C++ 语言了。

#### 1. 使用 `new` 和 `delete` 关键字

要分配一块内存，可调用 `new` 关键字，并提供需要空间的变量的类型。`new` 关键字返回指向那个内存的指针，但程序员应将这个指针保存在变量中。如果忽略了 `new` 关键字的返回值，或这个指针变量离开了作用域，那么这块内存就变成了被遗弃的，因为无法再访问这块内存。这也称为内存泄漏。

例如，下面的代码遗弃了一块足以保存 `int` 的内存。图 7-4 显示了代码执行后的内存状态。当自由存储区中有数据块无法从栈中直接或间接访问时，这块内存就被遗弃(或泄漏)了。

```
void leaky()
{
    new int; // BUG! Orphans/leaks memory!
    cout << "I just leaked an int!" << endl;
}
```



图 7-4 代码执行后的内存状态

除非计算机能提供无限制的高速内存，否则就需要告诉编译器，对象关联的内存什么时候可以释放，用作他用。为释放自由存储区中的内存，需要使用 `delete` 关键字，并提供指向那块内存的指针，如下所示。

```
int* ptr { new int };
delete ptr;
ptr = nullptr;
```

#### 警告：

作为经验法则，每写一行通过 `new` 关键字分配内存的代码，并使用原始指针，而不是把指针存储在智能指针中，就应该有一行用 `delete` 关键字释放同一块内存的对应代码。

#### 注意：

建议在释放指针的内存后，将指针重新设置为 `nullptr`。这样就不会在无意中使用一个指向已释放内存的指针。还可以在 `nullptr` 指针上调用 `delete` 关键字，它根本不会做任何事情。

## 2. 关于 malloc() 函数

如果你是一位 C 程序员，你可能想知道 malloc() 函数存在什么问题。在 C 语言中，通过 malloc() 分配给定字节数的内存。大多数情况下，使用 malloc() 简单明了。尽管在 C++ 中仍然存在 malloc()，但应避免使用它。new 关键字相比 malloc() 的主要好处在于，new 关键字不仅分配内存，还构造对象。

例如，考虑下面两行代码，这段代码使用了一个名为 Foo 的假想类：

```
Foo* myFoo { (Foo*)malloc(sizeof(Foo)) };
Foo* myOtherFoo { new Foo() };
```

执行这些代码行后，myFoo 和 myOtherFoo 将指向自由存储区中足以保存 Foo 对象的内存区域。通过这两个指针可访问 Foo 的数据成员和方法。不同之处在于，myFoo 指向的 Foo 对象不是一个正常的对象，因为这个对象从未被构造。malloc() 函数只负责留出一块一定大小的内存。它不知道或关心对象本身。相反，调用 new 关键字不仅会分配正确大小的内存，还会调用相应的构造函数以构建对象。

类似的差异存在于 free() 函数和 delete 运算符之间。使用 free() 时，不会调用对象的析构函数。使用 delete 时，将调用析构函数来恰当地清理对象。

### 警告：

在 C++ 中不应该使用 malloc() 和 free() 函数，只应使用 new 和 delete。

## 3. 当内存分配失败时

很多程序员会假设 new 总是会成功。他们的理由是，如果 new 失败了，则意味着内存量非常低，情况就非常糟糕了。这是一个无法预知的状态，因为不知道程序在这种情况下可能做什么。

默认情况下，如果 new 失败了，会抛出一个异常，例如没有足以满足请求的内存时。如果没有捕获到这个异常，程序会被终止。在许多程序中，这种行为是可以接受的。第 14 章将讲解如何在内存不足的情况下优雅地恢复。

也有不抛出异常的 new 版本。相反，它会返回 nullptr，这类似于 C 语言中 malloc() 的行为。使用这个版本的语法如下所示：

```
int* ptr { new(nothrow) int };
```

这个语法有点奇怪，你需要写 nothrow，就像它是 new 的参数一样（它的确是）。

当然，仍然要面对与抛出异常的版本同样的问题——如果结果是 nullptr，怎么办？编译器不要求检查结果，因此 new 的 nothrow 版本可能导致除了抛出异常的版本遇到的 bug 之外的其他 bug。因此，建议使用标准版本的 new。如果内存不足的恢复对程序非常重要，请参阅第 14 章，该章给出了需要的所有工具。

### 7.1.3 数组

数组将多个同一类型的变量封装在一个通过索引访问的变量中。编程新手很快会熟悉数组的使用，因为很容易想象将值放在编了号的槽中。数组在内存中的表示和这种想法差不多。

#### 1. 基本类型的数组

当程序为数组分配内存时，分配的是连续的内存块，每一块的大小足以容纳数组的单个元素。例如，在栈上分配 5 个 int 型数字的局部数组的声明如下所示：

```
int myArray[5];
```

这种基本类型数组的单个元素未初始化，也就是说，它们包含内存中该位置的任意内容。图 7-5 显示了创建数组后的内存状态。在栈上创建数组时，数组的大小必须是编译时已知的常量值。



图 7-5 创建数组后的内存状态

### 警告：

有些编译器允许栈上存在可变大小的数组。这不是 C++ 的标准功能，所以建议谨慎一些，尽量避免这种用法。

在栈上创建数组时，可以使用初始化列表提供初始元素：

```
int myArray[5] { 1, 2, 3, 4, 5 };
```

如果初始化列表包含的元素数量小于数组的大小，则数组的其余元素将初始化为 0。例如：

```
int myArray[5] { 1, 2 }; // 1, 2, 0, 0, 0
```

这个特性可以用来快速初始化一个所有元素全为 0 的数组：

```
int myArray[5] { 0 }; // 0, 0, 0, 0, 0
```

在这种情况下，0 可以被省略。下面的代码对所有元素进行了零初始化(见第 1 章)：

```
int myArray[5] { }; // 0, 0, 0, 0, 0
```

当使用初始化列表时，编译器可以自动推断元素的数量，不必显式声明数组的大小。例如：

```
int myArray[] { 1, 2, 3, 4, 5 };
```

在自由存储区上声明数组没什么不同，只是需要通过一个指针指向数组的位置。下面的代码为包含 5 个未初始化的 int 值的数组分配内存，并将指向这块内存的指针保存在变量 myArrayPtr 中。

```
int* myArrayPtr { new int[5] };
```

如图 7-6 所示，自由存储区中的数组和栈中的数组类似，只是位置不同而已。myArrayPtr 变量指向数组的第 0 个元素。

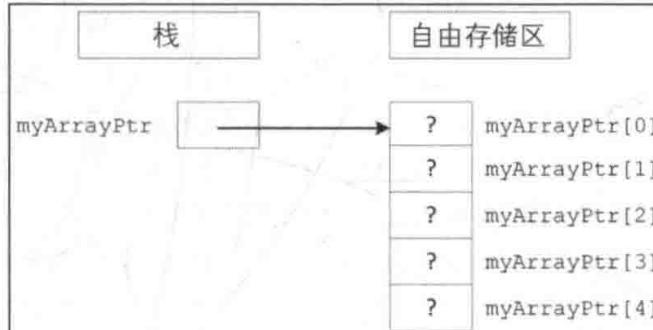


图 7-6 myArrayPtr 变量指向数组的第 0 个元素

与 new 运算符一样，接收 noexcept 参数的 new[] 在分配失败时返回 nullptr，而不是抛出异常。例如：

```
int* myArrayPtr { new(nothrow) int[5] };
```

在自由存储区上动态创建的数组也可以使用初始化列表进行初始化。例如：

```
int* myArrayPtr { new int[] { 1, 2, 3, 4, 5 } };
```

对 new[] 的每次调用都应与 delete[] 调用配对，以清理内存。例如：

```
delete[] myArrayPtr;
myArrayPtr = nullptr;
```

把数组放在自由存储区中的好处在于可在运行时通过动态内存指定数组的大小。例如，下面的代码片段从一个假想的函数 askUserForNumberOfDocuments() 中接收所需的文档数量，并利用这个结果创建 Document 对象的数组。

```
Document* createDocArray()
{
    size_t numDocs { askUserForNumberOfDocuments() };
    Document* docArray { new Document[numDocs] };
    return docArray;
}
```

对 new[] 的每次调用都应与 delete[] 调用配对，所以在本例中，createDocArray() 的调用者必须使用 delete[] 清理返回的内存。另一个问题是 C 风格的数组不知道其大小，因此 createDocArray() 的调用者不知道返回的数组有多少个元素。

在前面的函数中，docArray 是一个动态分配的数组。不要把它和动态数组混为一谈。数组本身不是动态的，因为一旦被分配，数组的大小就不会改变。动态内存允许在运行时指定分配的内存块的大小，但它不会自动调整其大小以容纳数据。

#### 注意：

一些数据结构能够动态调整大小，它们也知道实际大小，例如标准库容器。建议使用这些标准库容器而不是 C 风格的数组，因为这些容器用起来更安全。

在 C++ 中有一个继承自 C 语言的函数 realloc()。不要使用它！在 C 中，realloc() 用于改变数组的大小，采取的方法是分配新大小的新内存块，然后将所有旧数据复制到新位置，再删除旧内存块。在 C++ 中这种做法是极其危险的，因为用户定义的对象不能很好地适应按位复制。

#### 警告：

不要在 C++ 中使用 realloc()。这个函数很危险。

## 2. 对象的数组

对象的数组和简单类型的数组没有区别，除了它们初始化元素的方式不同之外。通过 new[N] 分配 N 个对象的数组时，实际上分配了 N 个连续的内存块，每一块足以容纳单个对象。使用 new[] 时，虽然默认的基本类型数组是未初始化的，但每个对象元素的无参构造函数(=default)会自动调用。这样，通过 new[] 分配对象数组时，会返回一个指向数组的指针，这个数组中的所有对象都被初始化了。

例如，考虑下面的类：

```
class Simple
```

```

{
public:
    Simple() { cout << "Simple constructor called!" << endl; }
    ~Simple() { cout << "Simple destructor called!" << endl; }
};

```

如果要分配包含 4 个 Simple 对象的数组，那么 Simple 构造函数会被调用 4 次。

```
Simple* mySimpleArray { new Simple[4] };
```

这个数组的内存图如图 7-7 所示。从中可以看出，这个数组和基本类型的数组没什么不同。

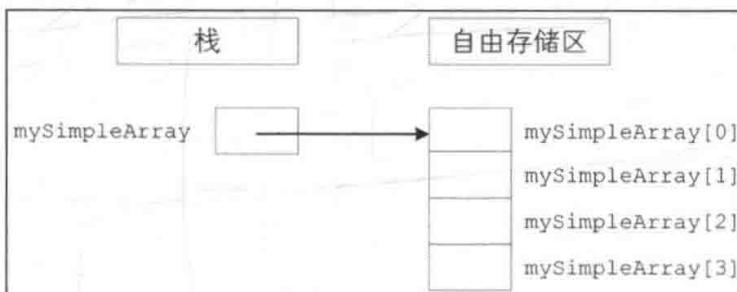


图 7-7 数组的内存图

### 3. 删除数组

如前所述，通过数组版本的 `new(new[])` 分配内存时，必须通过数组版本的 `delete(delete[])` 释放相应的内存。这个版本的 `delete` 会自动析构数组中的对象，并释放这些对象的内存。

```

Simple* mySimpleArray { new Simple[4] };
// Use mySimpleArray...
delete [] mySimpleArray;
mySimpleArray = nullptr;

```

如果不使用数组版本的 `delete`，程序就可能出现异常行为。在一些编译器中，可能只会调用数组中第 1 个元素的析构函数，因为编译器只知道要删除指向一个对象的指针，而数组中的其他所有元素都变成了孤立对象。在其他编译器中，可能出现内存崩溃的情况，因为 `new` 和 `new[]` 可能采用完全不同的内存分配方案。

#### 警告：

总是通过 `delete` 释放通过 `new` 分配的内存，总是使用 `delete[]` 释放通过 `new[]` 分配的内存。

当然，只有在数组元素是对象时才会调用析构函数。如果有一个指针数组，那么还需要逐个释放每个指针指向的对象，就像逐个分配对象一样，如以下代码所示。

```

const size_t size { 4 };
Simple** mySimplePtrArray { new Simple*[size] };

// Allocate an object for each pointer.
for (size_t i { 0 }; i < size; i++) { mySimplePtrArray[i] = new Simple{}; }

// Use mySimplePtrArray...

// Delete each allocated object.
for (size_t i { 0 }; i < size; i++) {
    delete mySimplePtrArray[i];
}

```

```

    mySimplePtrArray[i] = nullptr;
}

// Delete the array itself.
delete [] mySimplePtrArray;
mySimplePtrArray = nullptr;

```

### 注意:

在现代 C++ 中，应避免使用 C 风格的原始指针。因此，不要在 C 风格的数组中保存旧式的普通指针，而应在现代的标准库容器中保存智能指针，例如 `std::vector`。智能指针会在适当时候自动释放与其关联的内存，本章将在后边讨论智能指针。

## 4. 多维数组

多维数组将索引值的表示方式扩展到多索引。例如，井字棋(Tic-Tac-Toe)游戏可能会使用二维数组来表示  $3 \times 3$  的网格。下例在栈上声明了这样一个数组，初始化为零，并通过一些测试代码访问它：

```

char board[3][3] {};
// Test code
board[0][0] = 'X'; // X puts marker in position (0,0).
board[2][1] = 'O'; // O puts marker in position (2,1).

```

二维数组的第一个下标是 x 坐标还是 y 坐标？事实上这不重要，只要按一致的方式使用即可。 $4 \times 7$  的网格可声明为 `char board[4][7]`，也可以声明为 `char board[7][4]`。对于大多数应用程序，最容易想到的是将第一个下标用作 x 轴，将第二个下标用作 y 轴。

### 栈上的多维数组

在内存中，栈中的  $3 \times 3$  的二维数组 `board` 如图 7-8 所示。由于内存中不存在两个数轴(地址只是顺序排列的)，计算机将二维数组以一维数组的方式表示。所不同的是数组的大小和访问时采用的方法。

多维数组的大小是其所有维度的乘积，再乘以这个数组中单个元素的大小。在图 7-8 中，假设一个字符是 1 字节，那么  $3 \times 3$  的棋盘大小为  $3 \times 3 \times 1 = 9$  字节。对于  $4 \times 7$  的字符棋盘，数组大小为  $4 \times 7 \times 1 = 28$  字节。

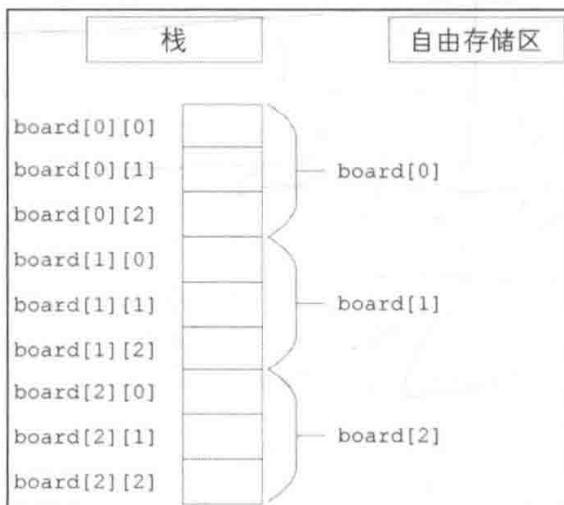


图 7-8  $3 \times 3$  的棋盘大小为 9 字节

要访问多维数组中的值，计算机将每个下标当作多维数组中的另一个子数组。例如，在 $3\times 3$ 的网格中，表达式`board[0]`实际上指图7-9中突出显示的子数组。添加第二个下标，如`board[0][2]`时，计算机通过子数组中的第二个下标访问子数组，从而访问正确的元素，如图7-10所示。

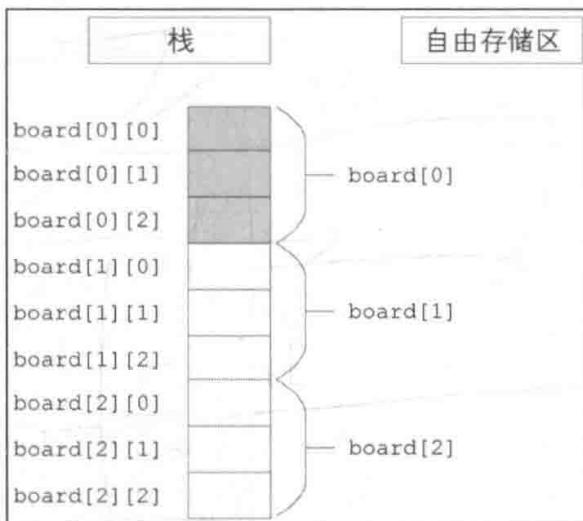


图7-9 子数组

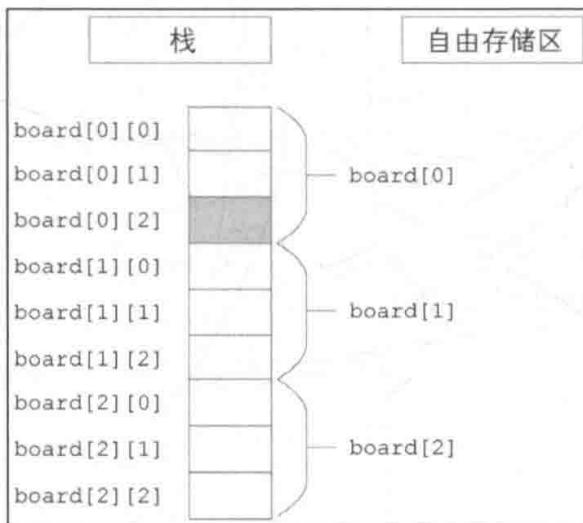


图7-10 添加第二个下标

这些技术可扩展至 $N$ 维数组，不过高于三维的数组很难概念化，因此在日常应用程序中极少使用。

### 自由存储区中的多维数组

如果需要在运行时确定多维数组的维数，可以使用基于自由存储区的数组。正如动态分配的一维数组是通过指针访问一样，动态分配的多维数组也通过指针访问。唯一的区别在于，在二维数组中，需要使用指针的指针，在 $N$ 维数组中，需要使用 $N$ 级指针。下面这种声明并动态分配多维数组的方式初看上去是正确的：

```
char** board { new char[i][j] }; // BUG! Doesn't compile
```

这段代码无法成功编译，因为自由存储区上的数组和栈上的数组的工作方式不一样。多维数组的内存布局是不连续的，可以首先为自由存储区数组的第一个下标分配一个连续的数组。该数组的每个元素实际上是指向另一个数组的指针，另一个数组保存的是第二个下标维度的元素。这种 $2\times 2$ 动态分配的棋盘布局如图7-11所示。

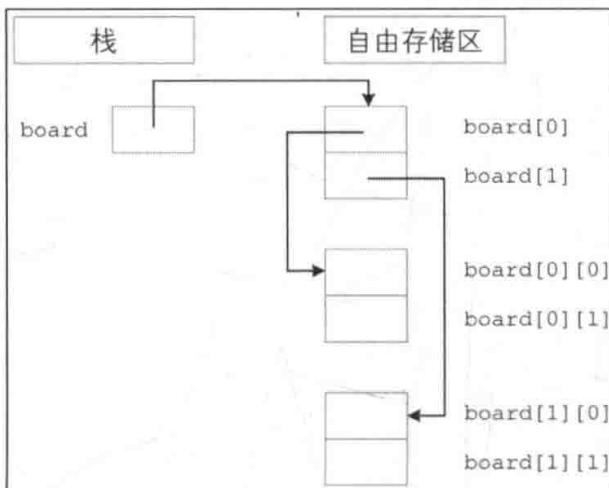


图 7-11 2×2 动态分配的棋盘布局

遗憾的是，编译器并不自动分配子数组的内存。可像分配一维自由存储区数组那样分配第一个维度的数组，但是必须显式地分配每一个子数组。下面的函数正确分配了二维数组的内存：

```
char** allocateCharacterBoard(size_t xDimension, size_t yDimension)
{
    char** myArray { new char*[xDimension] }; // Allocate first dimension
    for (size_t i { 0 }; i < xDimension; i++) {
        myArray[i] = new char[yDimension]; // Allocate ith subarray
    }
    return myArray;
}
```

要释放多维自由存储区数组的内存，数组版本的 `delete[]` 语法也不能自动清理子数组。释放数组的代码应该类似于分配数组的代码，如以下函数所示。

```
void releaseCharacterBoard(char**& myArray, size_t xDimension)
{
    for (size_t i { 0 }; i < xDimension; i++) {
        delete [] myArray[i]; // Delete ith subarray
        myArray[i] = nullptr;
    }
    delete [] myArray; // Delete first dimension
    myArray = nullptr;
}
```

### 注意：

这个分配多维数组的示例并不是最有效的解决方案。它首先为第一维分配内存，然后为每个子数组分配内存。这导致内存块分散在内存中，这将对处理此类数据结构的算法的性能产生影响。如果能够使用连续内存，算法运行速度会快得多。解决方案是分配一个大内存块，其大小等于 `xDimension*yDimension*elementSize`，并使用 `x*yDimension+y` 等公式访问元素。

知道了使用数组的所有细节后，建议尽可能不要使用旧式的 C 风格数组，因为这种数组没有提供任何内存安全性。这里解释它们，是因为可能在遗留代码中遇到它们。在新代码中，应改用 C++ 标准库容器，例如 `std::array` 和 `vector`。例如，用 `vector<T>` 表示一维动态数组，用 `vector<vector<T>>` 表示二维动态数组等。当然，直接使用诸如 `vector<vector<T>>` 的数据结构仍然是繁杂的，构造时尤其如此，这种方式同样存在着“注意”中提到的内存分散的问题。如果应用程序中需要 N 维动态数组，

建议编写辅助类，以方便使用接口。例如，要使用行长相等的二维数据，应当考虑编写(也可以重用)Matrix<T>或Table<T>类模板，该模板对用户隐藏了内存分配与释放、成员获取的细节。有关编写类模板的信息，请参阅第12章。

#### 警告：

不要使用旧式的C风格数组，应改用C++标准库容器，如std::array、vector。

### 7.1.4 使用指针

因为指针很容易被滥用，所以名声不佳。因为指针只是一个内存地址，所以理论上可以手动修改那个地址，甚至像下面这行代码一样做一些很可怕的事情。

```
char* scaryPointer { (char*)7 };
```

这行代码构建了一个指向内存地址7的指针，而这个位置可能是内存中随机的垃圾，或其他应用程序使用的内存。如果开始使用未通过new分配的内存区域，那么最终将损坏与对象关联的内存，或者破坏自由存储区管理相关的内存，使程序无法正常工作。这种故障可体现在几个方面。例如，可表现为无效结果，因为数据已损坏，或因为访问不存在的内存或写入受保护的内存而引发硬件异常。轻则得到错误结果，重则出现严重错误，导致操作系统或C++运行时库终止程序。

#### 1. 大脑中的指针模型

思考指针的方式有两种。更有数学头脑的读者可能会把指针看作地址。这种观点使指针的算术运算更容易理解，本章后面讲解了指针算术。指针不是进入内存的秘密通道，而是表示内存位置的数字。图7-12展示了如何从地址的角度看待 $2\times 2$ 的网格。

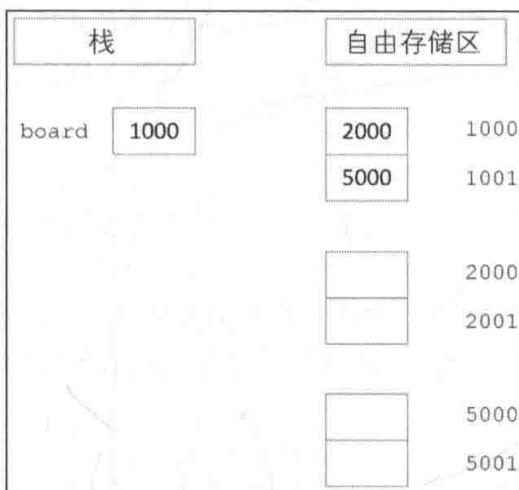


图7-12  $2\times 2$ 的网格

#### 注意：

图7-12中的地址只是用于演示。实际系统中的地址高度依赖于硬件和操作系统。

更熟悉空间表示法的读者应该更喜欢指针的“箭头”意义。指针仅仅是一个间接层，它告诉程序：“看那个地方。”从这个角度看，多层指针只不过是到达数据的路径中的各个步骤。图7-11展示了内存中指针的图形表示。

当通过\*运算符解除对一个指针的引用时，实际上会让程序在内存中更深入一步。从地址的角度看指针时，把解引用想象为跳到与那个指针表示的地址对应的内存。使用图形表示时，每次解除引用

都对应从箭尾到箭头的过程。

当通过&运算符取一个位置的地址时，在内存中添加了一个间接层。从地址的角度看，程序只不过是表示那个位置的地址的数值，这个数值可保存为指针形式。在图形视图中，&运算符创建了一个新箭头，其头部终止于表达式表示的位置，其尾部可以保存为一个指针。

## 2. 指针的类型转换

由于指针是内存地址(或指向某处的箭头)，因此指针的类型比较弱。指向 XML 文档的指针和指向整数的指针的大小相同。编译器允许通过 C 风格的类型转换将任意指针类型方便地转换为其他任意指针类型。

```
Document* documentPtr { getDocument() };
char* myCharPtr { (char*)documentPtr };
```

静态类型转换的安全性更高。编译器将拒绝执行不同数据类型的指针的静态类型转换：

```
Document* documentPtr = getDocument();
char* myCharPtr { static_cast<char*>(documentPtr) }; // BUG! Won't compile
```

如果要转换的两个指针指向的对象有继承关系，那么编译器允许静态类型转换。然而，在继承层次中完成转换的更安全方式是动态类型转换。有关继承以及不同 C++ 风格的类型转换的详细介绍见第 10 章。

## 7.2 数组-指针的对偶性

前面提到，指针和数组之间有一些重叠。在自由存储区上分配的数组通过指向该数组中第一个元素的指针来引用。基于栈的数组通过数组语法([])和普通的变量声明来引用。然而根据下面要学习的内容，数组和指针之间的关系不止如此。指针和数组之间存在复杂的关系。

### 7.2.1 数组就是指针

通过指针不仅能指向基于自由存储区的数组，也可以通过指针语法来访问基于栈的数组的元素。数组的地址就是第 1 个元素(索引 0)的地址。编译器知道，通过变量名使用整个数组时，实际上使用的是第 1 个元素的地址。从这个角度看，指针用起来就像基于自由存储区的数组。下面的代码创建了一个栈上的数组，数组元素初始化为 0，但通过一个指针访问这个数组。

```
int myIntArray[10] {};
int* myIntPtr { myIntArray };
// Access the array through the pointer.
myIntPtr[4] = 5;
```

向函数传递数组时，通过指针使用基于栈的数组的能力非常有用。下面的函数以指针的方式接收一个整数数组。请注意，调用者需要显式地传入数组的大小，因为指针没有包含任何与大小有关的信息。事实上，任何形式的 C++ 数组，不论是不是指针，都没有内含大小信息。这是应使用现代容器(例如标准库提供的容器)的另一个原因。

```
void doubleInts(int* theArray, size_t size)
{
    for (size_t i { 0 }; i < size; i++) { theArray[i] *= 2; }
```

这个函数的调用者可以传入基于栈或自由存储区的数组。在传入基于自由存储区的数组时，指针已经存在了，且按值传入函数。在传入基于栈的数组时，调用者可以传入一个数组变量，编译器会自动把这个数组变量当作指向数组的指针处理，还可以显式地传入第一个元素的地址。这里展示了所有三种形式：

```
size_t arrSize { 4 };
int* freeStoreArray { new int[arrSize]{ 1, 5, 3, 4 } };
doubleInts(freeStoreArray, arrSize);
delete[] freeStoreArray;
freeStoreArray = nullptr;

int stackArray[] { 5, 7, 9, 11 };
arrSize = std::size(stackArray); // Since C++17, requires <array>
//arrSize = sizeof(stackArray) / sizeof(stackArray[0]); // Pre-C++17, see Ch1
doubleInts(stackArray, arrSize);
doubleInts(&stackArray[0], arrSize);
```

数组参数传递的语义和指针参数传递的语义十分相似，因为当把数组传递给函数时，编译器将数组视为指针。函数如果接收数组作为参数，并修改数组中元素的值，实际上修改的是原始数组而不是副本。与指针一样，传递数组实际上模仿的是按引用传递的功能，因为真正传入函数的是原始数组的地址而不是副本。以下 `doubleInts()` 的实现修改了原始数组，即使参数是数组而不是指针，也同样如此：

```
void doubleInts(int theArray[], size_t size)
{
    for (size_t i { 0 }; i < size; i++) { theArray[i] *= 2; }
```

在函数原型中，`theArray` 后面方括号中的数字被忽略了。下面的 3 个版本是等价的：

```
void doubleInts(int* theArray, size_t size);
void doubleInts(int theArray[], size_t size);
void doubleInts(int theArray[2], size_t size);
```

为什么要以这种方式工作？为什么在函数定义中使用数组语法时编译器不复制数组？这样做是为了提高效率——复制数组中的元素需要时间，而且数组可能占用大量的内存。总是传递指针，编译器就不需要包括复制数组的代码。

可“按引用”给函数传递长度已知的基于栈的数组，但其语法并不直观。它不适用于基于自由存储区的数组。例如，下面的 `doubleIntsStack()` 仅接收大小为 4 的基于栈的数组：

```
void doubleIntsStack(int (&theArray)[4]);
```

可以使用函数模板(详见第 12 章)，让编译器自动推断基于栈的数组的大小：

```
template<size_t N>
void doubleIntsStack(int (&theArray)[N])
{
    for (size_t i { 0 }; i < N; i++) { theArray[i] *= 2; }
```

总之，通过数组语法声明的数组可通过指针访问。当把数组传递给函数时，这个数组总是作为指针传递。

## 7.2.2 并非所有指针都是数组

由于编译器允许在需要指针时传入数组，就像前面的 `doubleInts()` 函数那样，你可能会认为指针和数组是相同的，事实上它们有一些微妙但很重要的区别。指针和数组共享许多属性，有时可以互换使用(如前面所示)，但它们是不一样的。

指针本身是没有意义的。它可能指向随机内存、对象或数组。始终可使用指针的数组语法，但这样做并不总是正确的，因为指针并不总是数组。例如，考虑下面的代码：

```
int* ptr { new int };
```

`ptr` 是一个有效的指针，但不是一个数组。可通过数组语法(`ptr[0]`)访问这个指针指向的值，但是这样做的风格很可疑，而且没有什么真正的好处。事实上，对于非数组指针使用数组语法可能导致 bug。`ptr[1]` 处的内存可以是任意内容！

**警告：**

数组可以自动使用指针表示，但并非所有指针都是数组。

## 7.3 底层内存操作

C++相比 C 的一个巨大优势就是不必过分担心内存问题。如果代码使用了对象，只需要确保每个类都妥善管理自己的内存。通过构造和析构，编译器可提示什么时候管理内存。将内存管理隐藏在类中可以极大地改善可用性，标准库类就是一个例子。然而在一些应用程序或旧代码中，可能需要在底层操作内存。不论是为了遗留代码、效率、调试，还是为了满足好奇心，了解一些和底层字节相关的技术总是有帮助的。

### 7.3.1 指针运算

C++编译器通过声明类型的指针允许执行指针运算。如果声明一个指向 `int` 的指针，然后将这个指针递增 1，那么这个指针在内存中向前移动 1 个 `int` 的大小，而不是 1 字节。此类操作对数组最有用，因为数组在内存中包含同构的数据序列。例如，假设在自由存储区中声明一个整数数组：

```
int* myArray { new int[8] };
```

下面的语法给该数组中位置 2 的元素设置值：

```
myArray[2] = 33;
```

使用指针运算可等价地使用下面的语法，这个语法获得 `myArray` 数组中“向前 2 个 `int`”位置的内存地址，然后解除引用来设置值：

```
* (myArray + 2) = 33;
```

作为访问单个元素的替代语法，指针运算似乎没有太大吸引力。其真正的作用在于以下事实：像 `myArray + 2` 这样的表达式仍是一个指向 `int` 的指针，因而可以表示一个更小的整数数组。

来看一个使用宽字符串的示例，宽字符串将在第 21 章讨论，此时不必了解其细节。此处只需要了解宽字符串支持 Unicode 字符来扩大表示范围(如表示日语字符串)。`wchar_t` 类型是字符类型，可容纳此类 Unicode 字符，而且通常比 `char`(1 字节)更大。要告知编译器一个字符串字面量是宽字符串字面量，可加上前缀 L。假设有以下宽字符串：

```
const wchar_t* myString { L"Hello, World" };
```

假设还有一个函数，这个函数接收一个宽字符串，然后返回一个新字符串，新字符串是输入字符串的大写版本(注释：回想第2章的内容，C风格的字符串是以0结尾的。也就是说，它们的最后一个元素是\0。因此，不需要添加一个表示大小的参数来指定输入字符串的长度。这个函数会遍历整个字符串直至遇到\0)：

```
wchar_t* toCaps(const wchar_t* text);
```

将myString传入这个函数，可将myString大写化。不过，如果只想大写化myString的一部分，可以通过指针运算引用这个字符串后面的一部分。下面的代码给指针加7，对宽字符串中的“World”部分调用toCaps()，即使wchar\_t通常超过1字节。

```
toCaps(myString + 7);
```

指针运算的另一个有用应用是减法运算。将一个指针减去另一个同类型的指针，得到的是两个指针之间指针指向的类型的元素个数，而不是两个指针之间字节数的绝对值。

### 7.3.2 自定义内存管理

在99%的情况下(有人可能会说在100%的情况下)，C++中内置的内存分配设施是足够使用的。new和delete在后台完成了所有相关工作：分配正确大小的内存块、管理可用的内存区域链表以及释放内存时将内存块释放回可用内存链表。

资源非常紧张时，或在非常特殊的情况下，例如管理共享内存时，实现自定义的内存管理是一个可行的方案。不必担心——实际没有听起来那样可怕。基本上，自己管理内存通常意味着编写一些分配大块内存，并在需要的情况下发放大块内存中片段的类。

为什么这种方法更好？自行管理内存可能减少开销。当使用new分配内存时，程序还需要预留少量的空间来记录分配了多少内存。这样，当调用delete时，可以释放正确数量的内存。对于大多数对象，这个开销比实际分配的内存小得多，所以差别不大。然而，对于很小的对象或分配了大量对象的程序来说，这个开销的影响可能会很大。

当自行管理内存时，可事先了解每个对象的大小，因此可避免每个对象的开销。对于大量小对象而言，这个差别可能会很大。自定义内存管理需要重载new和delete运算符，这是第15章“C++运算符重载”的主题。

### 7.3.3 垃圾回收

在支持垃圾回收的环境中，程序员几乎不必显式地释放与对象关联的内存。运行时库会在某时刻自动清理没有任何引用的对象。

与C#和Java不一样，在C++语言中没有内建垃圾回收。在现代C++中，使用智能指针管理内存，在遗留代码中，则在对象层次通过new和delete管理内存。诸如shared\_ptr的智能指针(稍后讨论)提供类似于内存垃圾回收后的功能。也就是说，销毁某资源的最后一个shared\_ptr实例时，会同时销毁资源。在C++中实现真正的垃圾回收是可能的，但不容易，而将自己从释放内存的任务中解放出来可能引入新麻烦。

标记(mark)和清扫(sweep)是一种垃圾回收的方法。使用这种方法的垃圾回收器定期检查程序中的每个指针，并将指针引用的内存标记为仍在使用。在每一轮周期结束时，未标记的内存视为没有在使用，因而被释放。在C++中实现这样的算法并不容易，如果做得不对，它可能比使用delete更容

易出错！

人们已经尝试在 C++ 中实现安全简单的垃圾回收机制，但是就算 C++ 中出现了完美的垃圾回收机制，也不一定适用于所有应用程序。垃圾回收存在以下缺点：

- 当垃圾回收器正在运行时，程序可能停止响应。
- 使用垃圾回收器时，析构函数具有不确定性。由于对象在被垃圾回收之前不会销毁，因此对象离开作用域时不会立即执行析构函数。这意味着，由析构函数完成的资源清理操作（如关闭文件、释放锁等）要在将来某个不确定的时刻进行。

编写一个垃圾回收机制是很难的。你无疑会犯错，因为它容易出错，而且很可能会降低运行速度。因此，如果想要在应用程序中使用垃圾回收，强烈建议研究可供重用的现有专用垃圾回收库。

### 7.3.4 对象池

垃圾回收就像买了一堆盘子用来野餐，然后把所有用过的盘子留在院子里，等着什么时候有人把这些盘子捡起来并丢掉。当然，必须有一种更符合生态规律的内存管理方法。

对象池是回收的代名词。购买合理数量的盘子，在使用一个盘子后，就清理它供以后重新使用。使用对象池的理想情况是：在一段时间里，需要使用大量同类型的对象，而且创建每个对象都会有开销。

第 29 章“编写高效的 C++ 代码”将进一步讲解如何使用对象池提高性能。

## 7.4 常见的内存陷阱

使用 new、delete、new[]、delete[] 和底层内存操作来处理动态内存十分容易出错。很难准确指出在哪些情况下会导致内存相关的 bug。每个内存泄漏或错误指针都有微妙的差别。没有解决所有内存问题的灵丹妙药，本节讨论一些常见类型的问题，以及可以检测和解决这些问题的工具。

### 7.4.1 数据缓冲区分配不足以及内存访问越界

分配不足是与 C 风格字符串相关的常见问题，大多数情况下，都是因为程序员没有额外分配尾部的'\0'终止字符。当程序员假设某个固定的最大大小时，也会发生字符串分配不足的情况。基本的内置 C 风格字符串函数不会针对固定的大小操作——而是有多少写多少，如果超出字符串的末尾，就写入未分配的内存。

以下代码演示了分配不足的情况。它从网络连接读取数据，然后将其写入一个 C 风格的字符串。这个过程在一个循环中完成，因为网络连接一次只接收少量的数据。在每个循环中调用 getMoreData() 函数，这个函数返回一个指向动态分配内存的指针。当 getMoreData() 返回 nullptr 时，表示已收到所有数据。strcat() 是一个 C 函数，它把第二个参数的 C 风格字符串连接到第一个参数的 C 风格字符串的尾部。它要求目标缓冲区足够大。

```
char buffer[1024] {0}; // Allocate a whole bunch of memory.
while (true) {
    char* nextChunk = getMoreData();
    if (nextChunk == nullptr) {
        break;
    } else {
        strcat(buffer, nextChunk); // BUG! No guarantees against buffer overrun!
        delete [] nextChunk;
    }
}
```

有3种方法用于解决可能的分配不足问题。按照优先级降序排列，这3种方法为：

(1) 使用C++风格的字符串，它可自动处理与连接字符串相关联的内存。

(2) 不要将缓冲区分配为全局变量或分配在栈上，而是分配在自由存储区上。当剩余空间不足时，分配一个新缓冲区，它大到至少能保存当前内容加上新内存块的内容，将原来缓冲区的内容复制到新缓冲区，将新内容追加到后面，然后删除原来的缓冲区。

(3) 创建另一个版本的getMoreData()，这个版本接收一个最大字符数(包括'\0'字符)，返回的字符数不多于这个值，然后跟踪剩余的空间数以及缓冲区中当前的位置。

数据缓冲区分配不足通常会导致内存访问越界。例如，如果要用数据填充内存缓冲区，当认为缓冲区大于实际大小时，可能会开始在分配的数据缓冲区之外写入数据。内存的重要部分被改写和程序崩溃只是时间问题。想想如果程序中与对象相关的内存突然被改写可能会发生什么，这很糟糕！

在处理由于某种原因丢失了'\0'终止字符的C风格的字符串时，也可能发生内存访问越界。例如，如果一个不正确终止的字符串被传给了下面这个函数，它试图将字符串填满m字符，但实际上可能会继续在字符串后面填充m，改写了字符串边界外的内存。

```
void fillWithM(char* text)
{
    int i { 0 };
    while (text[i] != '\0') {
        text[i] = 'm';
        i++;
    }
}
```

写入数组尾部后面的内存产生的bug称为缓冲区溢出错误。这种bug已经被一些高危的恶意程序使用，例如病毒和蠕虫。狡猾的黑客可利用改写部分内存的能力，将代码注入正在运行的程序中。

#### 警告：

避免使用旧的C风格字符串和数组，它们没有提供任何保护；而要改用像C++ string和vector这样安全的现代结构，它们能够自动管理内存。

### 7.4.2 内存泄漏

C和C++编程中遇到的另一个令人沮丧的问题是找到和修复内存泄漏。程序终于开始工作，看上去能给出正确结果。然后，随着程序的运行，吞掉的内存越来越多。这是因为程序有内存泄漏。

分配了内存，但没有释放，就会发生内存泄漏。起初，这听上去好像是粗心编程的结果，应该很容易避免。毕竟，如果在编写的每个类中，每个new都对应一个delete，那么应该不会出现内存泄漏，对不对？实际上并不总是如此。在下面的代码中，Simple类编写正确，释放了每一处分配的内存。然而，当调用doSomething()函数时，outSimplePtr指针修改为指向另一个Simple对象，但是没有释放原来的Simple对象。为了演示内存泄漏，doSomething()函数故意没有释放旧的对象。一旦失去对象的指针，就几乎不可能删除它了。

```
class Simple
{
public:
    Simple() { m_intPtr = new int{}; }
    ~Simple() { delete m_intPtr; }
    void setValue(int value) { *m_intPtr = value; }
private:
```

```

        int* m_intPtr;
    }

void doSomething(Simple*& outSimplePtr)
{
    outSimplePtr = new Simple{};           // BUG! Doesn't delete the original.
}

int main()
{
    Simple* simplePtr { new Simple{} };   // Allocate a Simple object.
    doSomething(simplePtr);
    delete simplePtr;                   // Only cleans up the second object.
}

```

**警告：**

记住，上述代码仅用于演示目的！在生产环境的代码中，`m_IntPtr` 和 `simplePtr` 都不应该是原始指针，而应该是本章之后将要提到的智能指针。

在上例中，内存泄漏可能来自程序员之间的沟通不畅或糟糕的代码文档。`doSomething()` 的调用者可能没有意识到，该变量是通过引用传递的，因此，没有理由期望该指针会重新赋值。如果你注意到这个参数是一个指针的非 `const` 引用，就可能怀疑会发生奇怪的事情，但是 `doSomething()` 周围并没有说明这个行为的注释。

### 1. 通过 Visual C++ 在 Windows 中查找和修复内存泄漏

内存泄漏很难追查，因为不能轻松地在内存中查看哪些对象在使用，以及最初把对象分配到了内存的哪里。然而有些程序可自动完成这项工作。有很多内存泄漏检测工具，从昂贵的专业软件包到可免费下载的工具。如果使用的是 Microsoft Visual C++(注释：有一个免费的 Microsoft Visual C++ 版本，称为 Community Edition)，其调试库内建了对内存泄漏检测的支持。这个内存泄漏检测功能默认情况下没有启用，除非创建的是 MFC 项目。要在其他项目中启用它，需要在代码开头添加以下 3 行代码：

```
#define _CRTDBG_MAP_ALLOC
#include <cstdlib>
#include <crtdbg.h>
```

这几行应该和以上顺序完全一致。接下来，需要重新定义 `new` 运算符，如下所示。

```
#ifdef _DEBUG
#ifndef DBG_NEW
#define DBG_NEW new ( _NORMAL_BLOCK , __FILE__ , __LINE__ )
#define new DBG_NEW
#endif
#endif // _DEBUG
```

请注意新定义的 `new` 运算符在 “`#ifdef _DEBUG`” 语句中，所以只有在编译调试版的应用程序时，才会使用新的 `new`。这通常就是所需要的。由于性能原因，发行版通常不会执行对内存泄漏的任何检测。

最后，需要在 `main()` 函数的第一行中添加下面这行代码：

```
_CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
```

这行代码告诉 Visual C++ CRT(C 运行时)库，在应用程序退出时，将所有检测到的内存泄漏写入

调试输出控制台。对于前面那个存在内存泄漏的程序，调试控制台应该会包含类似以下的输出：

```
Detected memory leaks!
Dumping objects ->
c:\leaky\leaky.cpp(15) : {147} normal block at 0x014FABF8, 4 bytes long.
Data: < > 00 00 00 00
c:\leaky\leaky.cpp(33) : {146} normal block at 0x014F5048, 4 bytes long.
Data: <Pa > 50 61 20 01
Object dump complete.
```

上述输出清楚地表明在哪个文件的哪一行分配了内存但没有释放。文件名后面括号中的数字就是行号。大括号之间的数字是内存分配的计数器。例如，{147}表示这是程序开始之后进行的第 147 次分配。可使用 VC++ 的 \_CrtSetBreakAlloc() 函数告诉 Visual C++ 调试运行时，进行特定分配时进入调试器。例如，把下面这行代码添加到 main() 函数的开头，使调试器在第 147 次分配时中断。

```
_CrtSetBreakAlloc(147);
```

在这个存在内存泄漏的程序中，有两处泄漏——第一个 Simple 对象没有释放(第 33 行)，这个对象在自由存储区中创建的整数也没有释放(第 15 行)。在 Visual C++ 的调试器输出窗口中，只需要双击某个内存泄漏，就可以自动跳到代码中的那一行。

当然，本节讲解的 Visual C++ 和下一节讲解的 Valgrind 这类程序都不能实际修复内存泄漏——否则还有什么乐趣？通过这些工具提供的信息，可找到实际问题。通常情况下，需要逐步跟踪代码，找到指向某个对象的指针在哪里改写了，而原始对象却没有释放。大多数调试器都提供了“观察点(watch point)”功能，用于在发生这类事件时中断程序的执行。

## 2. 通过 Valgrind 在 Linux 中查找和修复内存泄漏

Valgrind 是一个免费的开源 Linux 工具，这个工具可在代码中精确地定位分配泄漏对象的那行代码。

下面的输出是对前面存在内存泄漏的程序运行 Valgrind 的结果，这个结果精确地指出了分配了内存但未释放的地方。Valgrind 发现两个和之前一样的内存泄漏——第一个 Simple 对象没有释放，这个对象在自由存储区中创建的整数也没有释放。

```
==15606== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==15606== malloc/free: in use at exit: 8 bytes in 2 blocks.
==15606== malloc/free: 4 allocs, 2 frees, 16 bytes allocated.
==15606== For counts of detected errors, rerun with: -v
==15606== searching for pointers to 2 not-freed blocks.
==15606== checked 4455600 bytes.
==15606==

==15606== 4 bytes in 1 blocks are still reachable in loss record 1 of 2
==15606==   at 0x4002978F: __builtin_new (vg_replace_malloc.c:172)
==15606==   by 0x400297E6: operator new(unsigned) (vg_replace_malloc.c:185)
==15606== by 0x804875B: Simple::Simple() (leaky.cpp:4)
==15606==   by 0x8048648: main (leaky.cpp:24)
==15606==

==15606== 4 bytes in 1 blocks are definitely lost in loss record 2 of 2
==15606==   at 0x4002978F: __builtin_new (vg_replace_malloc.c:172)
==15606==   by 0x400297E6: operator new(unsigned) (vg_replace_malloc.c:185)
==15606== by 0x8048633: main (leaky.cpp:20)
==15606==   by 0x4031FA46: __libc_start_main (in /lib/libc-2.3.2.so)
==15606==
```

```

==15606== LEAK SUMMARY:
==15606==   definitely lost: 4 bytes in 1 blocks.
==15606==   possibly lost: 0 bytes in 0 blocks.
==15606==   still reachable: 4 bytes in 1 blocks.
==15606==           suppressed: 0 bytes in 0 blocks.

```

**警告：**

强烈建议尽可能使用 `std::vector`、`array`、`string` 和本章稍后将会提到的智能指针等现代 C++ 工具，以避免内存泄漏。

### 7.4.3 双重释放和无效指针

通过 `delete` 释放某个指针关联的内存时，这个内存就可以由程序的其他部分使用了。然而，无法禁止再次使用这个指针，这个指针成为悬空指针(dangling pointer)。双重释放也是一个问题。如果第二次在同一个指针上执行 `delete` 操作，程序可能会释放重新分配给另一个对象的内存。

双重释放和使用已释放的内存都是很难追查的问题，因为症状可能不会立即显现。如果双重释放在较短的时间内发生，程序可能产生未定义的行为，因为关联的内存可能不会那么快被重用。同样，如果已被释放的对象在被释放后立即被使用，这个对象很有可能仍然完好无缺。

当然，无法保证这种行为一定会发生。一旦释放对象，内存分配器就没有义务保存任何对象。即使程序能正常工作，使用已释放的对象也是极糟糕的编程风格。

在释放指针关联的内存后，将指针设置为 `nullptr`，这样能防止不小心两次释放同一个指针和使用无效的指针。

很多内存泄漏检测程序也会检测双重释放和已释放对象的使用。

## 7.5 智能指针

如前所述，内存管理是 C++ 中常见的错误和 bug 来源。许多这类 bug 都来自动态内存分配和指针的使用。在程序中广泛使用动态内存分配，在对象间传递多个指针时，很容易忘记每个指针都要在合适的时间执行恰好一次 `delete` 操作。出错的后果很严重：当多次释放动态分配的内存时或使用指向已被释放的内存的指针时，可能会导致内存损坏或致命的运行时错误；当忘记释放动态分配的内存时，会导致内存泄漏。

智能指针可帮助管理动态分配的内存，这是避免内存泄漏建议采用的技术。这样，智能指针可保存动态分配的资源，如内存。当智能指针离开作用域或被重置时，会自动释放所占用的资源。智能指针可用于管理在函数作用域内(或作为类的数据成员)动态分配的资源。也可通过函数实参来传递动态分配的资源的所有权。

C++ 提供的一些语言特性使智能指针具有吸引力。首先，可通过模板为任何指针类型编写类型安全的智能指针类(见第 12 章)。其次，可使用运算符重载为智能指针对象提供一个接口，使智能指针对象的使用和普通指针一样(见第 15 章)。确切地讲，可重载\*、-> 和 [] 运算符，使客户代码解除对智能指针对象的引用的方式和解除对普通指针的引用相同。

智能指针有多种类型。最简单的智能指针类型对资源有唯一的所有权。当智能指针离开作用域或被重置时，会自动释放所指向的内存。标准库提供了 `std::unique_ptr`，这是一个具有“唯一所有权”语义的智能指针。

稍微高级一点的智能指针允许共享所有权，也就是说，多个智能指针可以指向同一个资源。当这

样的智能指针离开作用域或被重置时，仅当它是指向该资源的最后一个智能指针时，才应释放指向的资源。标准库提供了 `std::shared_ptr`，它支持共享所有权。

下面将详细讨论标准智能指针 `unique_ptr` 和 `shared_ptr`，使用它们时，需要添加`<memory>`头文件。

#### 注意：

应将 `unique_ptr` 用作默认智能指针。仅当真正需要共享资源时，才使用 `shared_ptr`。

#### 警告：

永远不要将资源分配结果赋值给原始指针。无论使用哪种资源分配方法，都应当立即将资源指针存储在智能指针 `unique_ptr` 或 `shared_ptr` 中，或使用其他 RAII 类。RAII 代表 Resource Acquisition Is Initialization(资源获取即初始化)。RAII 类获取某个资源的所有权，并在适当的时候进行释放。第 32 章将讨论这种设计技术。

### 7.5.1 unique\_ptr

`unique_ptr` 拥有资源的唯一所有权。当 `unique_ptr` 被销毁或重置时，资源将自动释放。`unique_ptr` 的一个优点是，内存和资源总会被释放，即使在执行 `return` 语句或抛出异常时也是如此。例如，当函数有多个返回语句时，这简化了编码，因为不必记住在每个 `return` 语句之前释放资源。

作为经验法则，总是应该将动态分配的有唯一所有者的资源保存在 `unique_ptr` 的实例中。

#### 1. 创建 unique\_ptr

考虑下面的函数，这个函数在自由存储区上分配了一个 `Simple` 对象，但是不释放这个对象，故意产生内存泄漏。

```
void leaky()
{
    Simple* mySimplePtr = new Simple{}; // BUG! Memory is never released!
    mySimplePtr->go();
}
```

有时你可能认为，代码正确地释放了动态分配的内存。遗憾的是，这种想法几乎总是不正确的。看看下面的函数：

```
void couldBeLeaky()
{
    Simple* mySimplePtr = new Simple{};
    mySimplePtr->go();
    delete mySimplePtr;
}
```

上面的函数动态分配一个 `Simple` 对象，使用该对象，然后正确地调用 `delete`。但是，这个例子仍然可能会产生内存泄漏！如果 `go()` 方法抛出一个异常，将永远不会调用 `delete`，导致内存泄漏。

应该使用 `unique_ptr`，它可通过 `std::make_unique()` 辅助函数创建。`unique_ptr` 是一个泛型的智能指针，可以指向任何类型的内存。因此它是一个类模板，`std::make_unique()` 是一个函数模板。两者都需要尖括号`<>`之间的模板参数，指定 `unique_ptr` 需要指向的内存类型。第 12 章将详细讨论模板，但这些细节对于理解如何使用智能指针并不重要。

下面的函数使用了 `unique_ptr`，而不是原始指针。`Simple` 对象不会显式释放，但 `unique_ptr` 实例离开作用域时(在函数的末尾，或者因为抛出了异常)，就会在其析构函数中自动释放 `Simple` 对象：

```

void notLeaky()
{
    auto mySimpleSmartPtr { make_unique<Simple>() };
    mySimpleSmartPtr->go();
}

```

这段代码使用 `make_unique()` 和 `auto` 关键字，所以只需要指定指针的类型，本例中是 `Simple`。如果 `Simple` 构造函数需要参数，就把它们放在 `make_unique()` 调用的圆括号中。

`make_unique()` 使用值初始化，例如，基本类型被初始化为零，对象被默认构造。如果不需要此值初始化，例如在对性能有严格要求的代码中，可以使用 C++20 中引入的 `make_unique_for_overwrite()` 函数，该函数使用默认初始化。对于基本类型，这意味着它们根本没有初始化，包含它们所在内存中的任何内容，而对象仍然是默认构造的。

也可以通过调用构造函数来直接创建一个 `unique_ptr`，如下所示，注意 `Simple` 必须写两次。

```
unique_ptr<Simple> mySimpleSmartPtr { new Simple{} };
```

正如本书前面所讨论的，类模板参数推断(CTAD)通常可用于让编译器根据传递给类模板构造函数的参数推断类模板的模板类型。例如，它允许编写 `vector v{1,2}`，而不是 `vector<int> v{1,2}`。CTAD 不适用于 `unique_ptr`，因此不能忽略模板类型。

在 C++17 之前，必须使用 `make_unique()`，一是因为只能将类型指定一次，二是出于安全考虑！考虑下面对 `foo()` 函数的调用：

```
foo(unique_ptr<Simple> { new Simple{} }, unique_ptr<Bar> { new Bar { data() } });
```

如果 `Simple`、`Bar` 的构造函数或 `data()` 函数抛出异常，具体取决于编译器的优化，很可能导致 `Simple` 或 `Bar` 对象出现内存泄漏。而使用 `make_unique()`，则不会发生内存泄漏。

```
foo(make_unique<Simple>(), make_unique<Bar>(data()))
```

在 C++17 之后，对 `foo()` 的两个调用都是安全的，但仍然建议使用 `make_unique()`，这样代码的可读性更好。

### 注意：

始终使用 `make_unique()` 创建 `unique_ptr`。

## 2. 使用 `unique_ptr`

这个标准智能指针最大的一个亮点是：用户不需要学习大量的新语法，就可以获得巨大好处。像标准指针一样，仍可以使用 \* 或 -> 对智能指针进行解引用。例如，在前面的例子中，使用 -> 运算符调用 `go()` 方法：

```
mySimpleSmartPtr->go();
```

与标准指针一样，也可将其写作：

```
(*mySimpleSmartPtr).go();
```

`get()` 方法可用于直接获得底层指针。这可将指针传递给需要普通指针的函数。例如，假设具有以下函数：

```
void processData(Simple* simple) { /* Use the simple pointer... */ }
```

可采用如下方式进行调用：

```
processData(mySimpleSmartPtr.get());
```

使用 `reset()`, 可释放 `unique_ptr` 的底层指针, 并根据需要将其改成另一个指针。例如:

```
mySimpleSmartPtr.reset();           // Free resource and set to nullptr
mySimpleSmartPtr.reset(new Simple{}); // Free resource and set to a new
                                     // Simple instance
```

可使用 `release()` 断开 `unique_ptr` 与底层指针的连接。`release()` 方法返回资源的底层指针, 然后将智能指针设置为 `nullptr`。实际上, 智能指针失去对资源的所有权, 你需要负责在用完资源时释放资源。例如:

```
Simple* simple { mySimpleSmartPtr.release() }; // Release ownership
// Use the simple pointer...
delete simple;
simple = nullptr;
```

由于 `unique_ptr` 代表唯一拥有权, 因此无法复制它! 但是, 可使用移动语义将一个 `unique_ptr` 移到另一个(详见第 9 章)。`std::move()` 工具函数可以用于显式移动 `unique_ptr` 的所有权, 如下所示。现在不要担心语法, 第 9 章将会详细介绍。

```
class Foo
{
public:
    Foo(unique_ptr<int> data) : m_data { move(data) } {}
private:
    unique_ptr<int> m_data;
};

auto myIntSmartPtr { make_unique<int>(42) };
Foo f { move(myIntSmartPtr) };
```

### 3. `unique_ptr` 和 C 风格数组

`unique_ptr` 适用于存储动态分配的旧式 C 风格数组。下例创建了一个 `unique_ptr` 来保存动态分配的、包含 10 个整数的 C 风格数组:

```
auto myVariableSizedArray { make_unique<int[]>(10) };
```

`myVariableSizedArray` 的类型是 `unique_ptr<int[]>`, 并支持使用数组标记访问其元素。如下所示:

```
myVariableSizedArray[1] = 123;
```

与非数组情况一样, `make_unique()` 对数组的所有元素使用值初始化, 类似于 `std::vector`。对于基本类型, 这意味着初始化为零。自 C++20 以来, 可以使用 `make_unique_for_overwrite()` 函数创建具有默认初始化值的数组, 这意味着未初始化基本类型。但是, 请记住, 应该尽可能避免未初始化的数据, 所以要明智地使用它。

即使可使用 `unique_ptr` 存储动态分配的 C 风格数组, 也建议改用标准库容器, 例如 `std::array` 和 `vector`。

### 4. 自定义 deleter

默认情况下, `unique_ptr` 使用标准的 `new` 和 `delete` 运算符来分配和释放内存。你可以将其改为自己的分配和释放函数。例如:

```

int* my_alloc(int value) { return new int { value }; }
void my_free(int* p) { delete p; }

int main()
{
    unique_ptr<int, decltype(&my_free)> myIntSmartPtr { my_alloc(42), my_free };
}

```

这段代码使用 `my_malloc()` 给整数分配内存，`unique_ptr` 调用 `my_free()` 函数来释放内存。`unique_ptr` 的这项特性很有用，因为还可管理其他类型的资源而不仅是内存。例如，当 `unique_ptr` 离开作用域时，可自动关闭文件或网络套接字以及其他资源。

但是，`unique_ptr` 的自定义 `deleter` 的语法有些费解。需要将自定义 `deleter` 的类型指定为模板类型参数，这是一个接收一个指针参数并且返回 `void` 的函数指针类型。在本例中，`decltype(&my_free)` 用于返回 `my_free()` 的函数指针类型。使用 `shared_ptr` 的自定义 `deleter` 就容易多了。下面讨论 `shared_ptr` 的小节将演示如何使用 `shared_ptr`，在 `shared_ptr` 离开作用域时自动关闭文件。

## 7.5.2 shared\_ptr

有时，多个对象或代码段需要同一指针的副本。由于 `unique_ptr` 无法复制，因此不能用于此类情况。相反，`std::shared_ptr` 是一个可复制的支持共享所有权的智能指针。但是，如果有多个 `shared_ptr` 实例引用同一资源，它们如何知道何时实际释放资源？这可以通过所谓的引用计数来解决，这正是“引用计数的必要性”一节的主题。但首先，让我们看看如何构造和使用 `shared_ptr`。

### 1. 创建并使用 `shared_ptr`

`shared_ptr` 的用法与 `unique_ptr` 类似。要创建 `shared_ptr`，可使用 `make_shared()`，它比直接创建 `shared_ptr` 更高效。例如：

```
auto mySimpleSmartPtr { make_shared<Simple>() };
```

#### 警告：

总是使用 `make_shared()` 创建 `shared_ptr`。

请注意，与 `unique_ptr` 一样，类模板参数推断也不适用于 `shared_ptr`，所以必须指定模板类型。

与 `unique_ptr` 类似，`make_shared()` 使用了值初始化。如果你不需要的话，可以使用 C++20 的 `make_shared_for_overwrite()` 实现默认初始化，与 `make_unique_for_overwrite()` 用法类似。

从 C++17 开始，`shared_ptr` 就可以像 `unique_ptr` 一样可用于存储动态分配的 C 风格数组的指针。另外，从 C++20 开始，你可以使用 `make_shared()` 来做这件事，正如你可以使用 `make_unique()` 一样。但是，尽管这是可能的，仍建议使用标准库容器而非 C 风格数组。

与 `unique_ptr` 一样，`shared_ptr` 也支持 `get()` 和 `reset()` 方法。唯一的区别在于，当调用 `reset()` 时，仅在最后的 `shared_ptr` 销毁或重置时，才释放底层资源。注意，`shared_ptr` 不支持 `release()`。可使用 `use_count()` 方法检索共享同一资源的 `shared_ptr` 实例数量。

与 `unique_ptr` 类似，默认情况下，当存储 C 风格数组时，`shared_ptr` 使用标准的 `new` 和 `delete` 运算符或使用 `new[]` 和 `delete[]` 分配和释放内存，可更改此行为，如下所示：

```
// Implementation of malloc_int() as before.
shared_ptr<int> myIntSmartPtr { malloc_int(42), my_free };
```

可以看到，不必将自定义 `deleter` 的类型指定为模板类型参数，这比 `unique_ptr` 的自定义 `deleter`

更简便。

下面的示例使用 `shared_ptr` 存储文件指针。当 `shared_ptr` 被销毁时(此处为脱离作用域时), 会调用 `close()` 自动关闭文件指针。C++ 中有可以操作文件的面向对象的类(参见第 13 章), 这些类在离开作用域时会自动关闭文件。这个例子使用了旧式 C 语言的 `fopen()` 和 `fclose()` 函数, 只是为了演示 `shared_ptr` 除了管理纯粹的内存之外还可以用于其他目的。例如, 如果必须使用没有 C++ 替代的 C 风格库, 并且具有类似的功能来打开和关闭资源, 则可以使用它。可以使用 `shared_ptr` 包装它们, 如本例所示。

```
void close(FILE* filePtr)
{
    if (filePtr == nullptr) { return; }
    fclose(filePtr);
    cout << "File closed." << endl;
}
int main()
{
    FILE* f { fopen("data.txt", "w") };
    shared_ptr<FILE> filePtr { f, close };
    if (filePtr == nullptr) {
        cerr << "Error opening file." << endl;
    } else {
        cout << "File opened." << endl;
        // Use filePtr
    }
}
```

## 2. 引用计数的必要性

如前所述, 当具有共享所有权的智能指针(如 `shared_ptr`)超出作用域或被重置时, 只有当其是引用资源的最后一个智能指针时, 才应该释放引用的资源。这是如何实现的? 标准库智能指针 `shared_ptr` 使用的一种解决方案是引用计数。

作为一般概念, 引用计数(reference counting)用于跟踪正在使用的某个类的实例或特定对象的个数。引用计数的智能指针跟踪为引用一个真实指针(或某个对象)而建立的智能指针的数目。每次复制这样一个引用计数的智能指针时, 都会创建一个指向同一资源的新实例, 并且引用计数会增加。当此类智能指针实例超出作用域或被重置时, 引用计数将减少。当引用计数降至零时, 资源不再有其他所有者, 因此智能指针将释放资源。

引用计数的智能指针解决了很多内存管理的问题, 例如双重释放。例如, 假设你有如下两个指向同一内存的原始指针。考虑前面引入的 `Simple` 类, 这个类只是打印出创建或销毁一个对象的消息。

```
Simple* mySimple1 { new Simple{} };
Simple* mySimple2 { mySimple1 }; // Make a copy of the pointer.
```

释放两个原始指针会导致双重释放。

```
delete mySimple2;
delete mySimple1;
```

当然, 你(理想情况下)永远不会看到这样的代码, 但当涉及多层函数调用时, 可能会发生这种情况, 一个函数在另一个函数已经释放了内存之后尝试释放内存。

通过使用引用计数的智能指针 `shared_ptr`, 可以避免这种问题。

```
auto smartPtr1 { make_shared<Simple>() };
auto smartPtr2 { smartPtr1 }; // Make a copy of the pointer.
```

这种情况下，只有两个智能指针都离开作用域或被重置时，Simple 实例会被释放恰好一次。

所有这些只有在没有原始指针的情况下才能正常工作！例如，假设你使用 new 分配了一些内存，然后创建了两个指向原始指针的 shared\_ptr 实例：

```
Simple* mySimple = new Simple();
shared_ptr<Simple> smartPtr1 { mySimple };
shared_ptr<Simple> smartPtr2 { mySimple };
```

当被销毁时，两个智能指针都会试图释放同一个对象。取决于你的编译器，这段代码可能会崩溃！如果得到了输出，则输出为：

```
Simple constructor called!
Simple destructor called!
Simple destructor called!
```

糟糕！只调用一次构造函数，却调用两次析构函数？使用 unique\_ptr 也会出现同样的问题。你可能会惊讶连引用计数的 shared\_ptr 类也会以这种方式工作。然而，根据 C++ 标准，这是正确的行为。唯一安全得到多个指向同一内存的 shared\_ptr 实例的方法是创建 shared\_ptr 的副本。

### 3. 强制转换 shared\_ptr

正如某种类型的原始指针可以转换为另一种类型的指针一样，存储某种类型的 shared\_ptr 也可以转换为另一种类型的 shared\_ptr。当然，对什么类型可以强制转换为什么类型有限制，并非所有强制转换都有效。可用于强制转换 shared\_ptr 的函数有 const\_pointer\_cast()、dynamic\_pointer\_cast()、static\_pointer\_cast() 和 reinterpret\_pointer\_cast()。它们的行为和工作方式类似于非智能指针转换函数 const\_cast()、dynamic\_cast()、static\_cast() 和 reinterpret\_cast()，第 10 章将详细讨论这些方法。

### 4. 别名

shared\_ptr 支持所谓的别名。这允许一个 shared\_ptr 与另一个 shared\_ptr 共享一个指针(拥有的指针)，但指向不同的对象(存储的指针)。例如，这可用于使用一个 shared\_ptr 拥有一个对象本身的同时，指向该对象的成员。例如：

```
class Foo
{
public:
    Foo(int value) : m_data { value } {}
    int m_data;
};

auto foo { make_shared<Foo>(42) };
auto aliasing { shared_ptr<int> { foo, &foo->m_data } };
```

仅当两个 shared\_ptr(foo 和 aliasing)都销毁时，才销毁 Foo 对象。

“拥有的指针”用于引用计数，对指针解引用或调用它的 get() 时，将返回“存储的指针”。

#### 警告：

在现代 C++ 代码中，只有在没有涉及所有权时才允许使用原始指针！如果涉及所有权，默认情况下使用 unique\_ptr；如果需要共享所有权，则使用 shared\_ptr。此外，使用 make\_unique() 和 make\_shared() 创建这些智能指针。这样，几乎不需要直接调用 new 运算符，也不需要调用 delete。

### 7.5.3 weak\_ptr

在 C++ 中还有一个类与 `shared_ptr` 有关，那就是 `weak_ptr`。`weak_ptr` 可包含由 `shared_ptr` 管理的资源的引用。`weak_ptr` 不拥有这个资源，所以不能阻止 `shared_ptr` 释放资源。`weak_ptr` 销毁时(例如离开作用域时)不会销毁它指向的资源，然而，它可用于判断资源是否已经被关联的 `shared_ptr` 释放了。`weak_ptr` 的构造函数要求将一个 `shared_ptr` 或另一个 `weak_ptr` 作为参数。为了访问 `weak_ptr` 中保存的指针，需要将 `weak_ptr` 转换为 `shared_ptr`。这有两种方法：

- 使用 `weak_ptr` 实例的 `lock()` 方法，这个方法返回一个 `shared_ptr`。如果同时释放了与 `weak_ptr` 关联的 `shared_ptr`，返回的 `shared_ptr` 是 `nullptr`。
- 创建一个新的 `shared_ptr` 实例，将 `weak_ptr` 作为 `shared_ptr` 构造函数的参数。如果释放了与 `weak_ptr` 关联的 `shared_ptr`，将抛出 `std::bad_weak_ptr` 异常。

下例演示了 `weak_ptr` 的用法：

```
void useResource(weak_ptr<Simple>& weakSimple)
{
    auto resource { weakSimple.lock() };
    if (resource) {
        cout << "Resource still alive." << endl;
    } else {
        cout << "Resource has been freed!" << endl;
    }
}

int main()
{
    auto sharedSimple { make_shared<Simple>() };
    weak_ptr<Simple> weakSimple { sharedSimple };

    // Try to use the weak_ptr.
    useResource(weakSimple);

    // Reset the shared_ptr.
    // Since there is only 1 shared_ptr to the Simple resource, this will
    // free the resource, even though there is still a weak_ptr alive.
    sharedSimple.reset();

    // Try to use the weak_ptr a second time.
    useResource(weakSimple);
}
```

上述代码的输出如下：

```
Simple constructor called!
Resource still alive.
Simple destructor called!
Resource has been freed!
```

从 C++17 开始，`weak_ptr` 可以像 `shared_ptr` 一样支持 C 风格的数组。

### 7.5.4 向函数传递参数

仅当涉及所有权转移或所有权共享时，接受指针作为其参数之一的函数才应接受智能指针。要共享 `shared_ptr` 的所有权，只需要接受按值传递的 `shared_ptr` 作为参数。类似地，要转移 `unique_ptr` 的所

有权，只需要接受按值传递的 unique\_ptr 作为参数。后者需要使用移动语义，第 9 章将详细讨论。

如果既不涉及所有权转移也不涉及所有权共享，那么函数应该简单地使用 non-const 的引用或 const 引用作为参数；或者如果 nullptr 是参数的有效值，则应该使用原始指针作为参数。拥有诸如 const shared\_ptr<T>& 或 const unique\_ptr<T>& 的参数类型没有多大意义。

### 7.5.5 从函数中返回

由于在第 1 章中讨论的(命名的)返回值优化和在第 9 章中讨论的移动语义，标准智能指针 shared\_ptr、unique\_ptr 和 weak\_ptr 可以简单有效地从函数中按值返回。移动语义的详细信息目前并不重要，重要的是，所有这些都意味着从函数返回智能指针是高效的。例如，可以编写以下 create() 函数，并在 main() 中使用它。

```
unique_ptr<Simple> create()
{
    auto ptr { make_unique<Simple>() };
    // Do something with ptr...
    return ptr;
}
int main()
{
    unique_ptr<Simple> mySmartPtr1 { create() };
    auto mySmartPtr2 { create() };
}
```

### 7.5.6 enable\_shared\_from\_this

std::enable\_shared\_from\_this 派生的类允许对象调用方法，以安全地返回指向自己的 shared\_ptr 或 weak\_ptr。如果没有这个基类，返回有效的 shared\_ptr 或 weak\_ptr 的一种方法是将 weak\_ptr 作为成员添加到类中，并返回它的副本或返回由它构造的 shared\_ptr。enable\_shared\_from\_this 类给派生类添加了以下两个方法：

- shared\_from\_this()——返回一个 shared\_ptr，它共享对象的所有权。
- weak\_from\_this()——返回一个 weak\_ptr，它跟踪对象的所有权。

这是一项高级功能，此处不做详述，下面的代码简单演示了它的用法。shared\_from\_this() 和 weak\_from\_this() 都是 public 方法。但是，你可能会觉得 public 接口中的 from\_this 部分令人困惑，因此作为演示，下面的 Foo 类定义了自己的方法 getPointer()。

```
class Foo : public enable_shared_from_this<Foo>
{
public:
    shared_ptr<Foo> getPointer() {
        return shared_from_this();
    }
};

int main()
{
    auto ptr1 { make_shared<Foo>() };
    auto ptr2 { ptr1->getPointer() };
}
```

注意，仅当对象的指针已经存储在 `shared_ptr` 时，才能使用对象上的 `shared_from_this()`。否则，将会抛出 `bad_weak_ptr` 异常。在本例中，在 `main()` 中使用 `make_shared()` 创建一个名为 `ptr1` 的 `shared_ptr`(其中包含 `Foo` 实例)。创建这个 `shared_ptr` 后，将允许它调用 `Foo` 实例上的 `shared_from_this()`。另一方面，调用 `weak_from_this()` 总是允许的，但当对象的指针还未存储在 `shared_ptr` 时，将会返回一个空的 `weak_ptr`。

下面的 `getPointer()` 方法的实现是完全错误的：

```
class Foo
{
public:
    shared_ptr<Foo> getPointer() {
        return shared_ptr<Foo>(this);
    }
};
```

如果像前面那样为 `main()` 使用相同的代码，`Foo` 的该实现将导致双重释放。有两个完全独立的 `shared_ptr` (`ptr1` 和 `ptr2`) 指向同一对象，在离开作用域时，它们都会尝试释放该对象。

### 7.5.7 过时的、移除的 `auto_ptr`

在 C++11 之前，老的标准库包含了一个智能指针的简单实现，称为 `auto_ptr`。遗憾的是，`auto_ptr` 存在一些严重缺点。缺点之一是在标准库容器(例如 `vector`)中使用时，`auto_ptr` 不能正常工作。C++11 和 C++14 已经不赞成使用 `auto_ptr`，C++17 则完全在标准库中移除了 `auto_ptr`。`auto_ptr` 已被 `shared_ptr` 和 `unique_ptr` 取代。这里提到 `auto_ptr` 是为了确保你知道这个智能指针，并且绝不要使用它。

#### 警告：

不要再使用旧的 `auto_ptr` 智能指针，而使用 `unique_ptr` 或 `shared_ptr`！

## 7.6 本章小结

本章介绍了动态内存的方方面面。除了使用内存检查工具和认真编写代码之外，关于避免动态内存相关的问题还有两个关键点。

首先，需要了解在后台指针是如何工作的。在阅读有关大脑中指针的两种不同模型时，你应该对编译器处理内存的方式胸有成竹。

其次，当涉及所有权时，应该避免使用原始指针，避免使用旧式的 C 风格数据结构和函数。而应该使用安全的 C++ 替代品，例如 C++ `string` 类、`vector` 容器和智能指针等可以自动管理其内存的对象。

## 7.7 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，然后再从网站上查看解决方案。

**练习 7-1** 分析以下代码片段。你能列出你在其中发现的问题吗？在本练习中你不需要修复这些问题，那是练习 7-2 的事情。

```

const size_t numberElements { 10 };
int* values { new int[numberElements] };
// Set values to their index value.
for (int index { 0 }; index < numberElements; ++index) {
    values[index] = index;
}
// Set last value to 99.
values[10] = 99;
// Print all values.
for (int index { 0 }; index <= numberElements; ++index) {
    cout << values[index] << " ";
}

```

**练习 7-2** 使用安全的现代 C++ 工具，重写上题中的代码片段。

**练习 7-3** 编写一个基本类来存储具有 x、y 和 z 坐标的三维点。包括接收 x、y 和 z 参数的构造函数。编写一个接收三维点并使用 std::format() 打印其坐标的函数。在 main() 函数中，动态分配类的实例，然后调用函数。

**练习 7-4** 在本章前面，以下函数出现在内存访问越界的部分。你能用安全的 C++ 替代品升级这个函数吗？在 main() 函数中测试你的解决方案。

```

void fillWithM(char* text)
{
    int i { 0 };
    while (text[i] != '\0') {
        text[i] = 'm';
        i++;
    }
}

```

# 第8章

## 类和对象

### 本章内容

- 如何编写具有方法和数据成员的类
- 如何控制方法和数据成员的访问
- 如何在栈和自由存储区中使用对象
- 什么是对象的生命周期
- 如何编写在创建或销毁对象时执行的代码
- 如何编写复制对象或者给对象赋值的代码

作为面向对象语言, C++提供了使用对象和编写对象蓝图的工具, 称为类。当然, 可编写没有类和对象的 C++程序, 但是这样做就没有利用这门语言的最基本、最有用的特性。编写没有类的 C++程序就像去巴黎吃麦当劳一样。为有效地使用类和对象, 必须理解其语法和功能。

第 1 章回顾了类定义的基本语法, 第 5 章介绍了 C++中的面向对象编程方法, 并给出了类与对象的具体设计策略。

本章讲述与类和对象的使用有关的基本概念, 包括编写类定义、定义方法、在栈和自由存储区中使用对象, 以及编写构造函数、默认构造函数、编译器生成的构造函数、构造函数初始化器(称为 ctor-initializer)、拷贝构造函数、初始化列表构造函数、析构函数和赋值运算符。即使已经熟悉了类和对象, 也应该大致了解本章的内容, 因为本章包含了各种细节信息, 其中一些你可能并不熟悉。

### 8.1 电子表格示例介绍

本章和第 9 章将展示一个可运行的、简单的电子表格示例。电子表格是一种二维的“单元格”网格, 每个单元格包含一个数字或字符串。专业的电子表格(例如 Microsoft Excel)提供了执行数学计算的功能, 例如, 对一组单元格的值求和。这里的电子表格示例并不想抢占 Microsoft 的市场, 只是用来说明类和对象。

这个电子表格使用了两个基本类: Spreadsheet 和 SpreadsheetCell。每个 Spreadsheet 对象都包含了若干 SpreadsheetCell 对象。此外, SpreadsheetApplication 类管理 Spreadsheet 集合。本章重点介绍 SpreadsheetCell 类, 第 9 章开发 Spreadsheet 和 SpreadsheetApplication 类。

**注意：**

为循序渐进地讲解概念，本章显示了几个不同版本的 SpreadsheetCell 类。因此，本章关于类的各种尝试并非总是说明编写类的“最佳”方法。特别是早期的示例省略了一些通常会包含但还没有介绍的重要特性。可下载这个类的最终版本，下载方法见本章开头的说明。

## 8.2 编写类

编写类时，需要指定行为或方法(应用于类的对象)，还需要指定属性或数据成员(每个对象都会包含)。

编写类有两个要素：定义类本身和定义类的方法。

### 8.2.1 类定义

下面开始尝试编写一个简单的 SpreadsheetCell 类，定义在 spreadsheet\_cell 模块中，其中每个单元格只能存储一个数字：

```
export module spreadsheet_cell;

export class SpreadsheetCell
{
    public:
        void setValue(double value);
        double getValue() const;
    private:
        double m_value;
};
```

如第 1 章所述，第一行表明了这是一个名为 spreadsheet\_cell 的模块的定义。每个类定义都以关键字 class 和类名开始。如果该类是在模块中定义的，并且该类应该对导入该模块的客户可见，则 class 关键字要加上 export 的前缀。类定义是一个声明，以分号结尾。

如果类定义结束时不使用分号，编译器将给出几个错误，这些错误十分模糊，似乎与缺少分号毫不相干。

类定义通常放在以类命名的文件中。例如，SpreadsheetCell 类定义可放在 SpreadsheetCell.cppm 文件中。有些编译器需要使用特定的扩展名，其他允许使用任何扩展名。

#### 1. 类的成员

类可有许多成员。成员可以是成员函数(方法、构造函数或析构函数)，也可以是成员变量(也称为数据成员)、成员枚举、类型别名和嵌套类等。

下面两行声明了类支持的方法，这有点像函数原型：

```
void setValue(double value);
double getValue() const;
```

第 1 章指出过，最好将不改变对象的成员函数声明为 const。

下面这行声明了类的数据成员，看上去有点像变量的声明。

```
double m_value;
```

类定义了成员函数和数据成员，但它们只应用于类的特定实例，也就是对象。这条规则的唯一例

外是静态成员，参见第9章。类定义概念，对象包含实际的内容。因此，每个对象都会包含自己的 `m_value` 数据成员。成员函数的实现被所有对象共享，类可以包含任意数量的成员函数和数据成员。成员函数和数据成员不能同名。

## 2. 访问控制

类中的每个方法和成员都可用三种访问说明符(access specifiers)之一来说明：`public`、`protected` 或 `private`。`protected` 访问说明符将在第10章提到继承时详细说明。访问说明符将应用于其后声明的所有成员，直到遇到另一个访问说明符。在 `SpreadsheetCell` 类中，`setValue()` 和 `getValue()` 方法是 `public` 访问，而 `m_value` 数据成员是 `private` 访问。

类的默认访问说明符是 `private`：在第一个访问说明符之前声明的所有成员的访问都是 `private` 的。例如，将 `public` 访问说明符移到 `setValue()` 方法声明的下方，`setValue()` 方法就会成为 `private` 访问而不是 `public` 访问。

```
export class SpreadsheetCell
{
    void setValue(double value); // now has private access
public:
    double getValue() const;
private:
    double m_value;
};
```

与类相似，C++中的结构体(struct)也可以拥有方法。实际上，唯一的区别就是结构体的默认访问说明符是 `public`，而类默认是 `private`。(注释：严格来说，第二个区别是，结构体的默认基类访问也是 `public` 的，而类的默认基类访问是 `private` 的，但是基类访问是第10章的主题)

例如，`SpreadsheetCell` 类可以用结构体重写，如下所示：

```
export struct SpreadsheetCell
{
    void setValue(double value);
    double getValue() const;
private:
    double m_value;
};
```

如果只需要一组可供公共访问的数据成员，没有方法或方法数量极少，习惯上用结构体替代类。这是一个简单的用于存储二维点坐标的结构体示例：

```
export struct Point
{
    double x;
    double y;
};
```

## 3. 声明的顺序

可使用任何顺序声明成员和访问控制说明符：C++没有施加任何限制，例如成员函数在数据成员之前，或者 `public` 在 `private` 之前。此外，可重复使用访问说明符。例如，可这样定义 `SpreadsheetCell` 类：

```
export class SpreadsheetCell
{
```

```

public:
    void setValue(double value);
private:
    double m_value;
public:
    double getValue() const;
};

```

当然，为清晰起见，最好将 public、protected 和 private 声明分组，并在这些声明内将成员函数和数据成员分组。

#### 4. 类内成员初始化器

可直接在类定义中初始化成员变量。例如，默认情况下，可在 SpreadsheetCell 类定义中直接将 m\_value 初始化为 0，如下所示。

```

export class SpreadsheetCell
{
    // Remainder of the class definition omitted for brevity
private:
    double m_value { 0 };
};

```

##### 注意：

建议总是初始化类内的数据成员。

### 8.2.2 定义方法

前面 SpreadsheetCell 类的定义足以创建类的对象。然而，如果试图调用 setValue() 或 getValue() 方法，链接器将发出警告，指出方法没有定义。这是因为类定义指明了方法的原型，但是没有定义方法的实现。通常类定义在模块接口文件中，对于方法定义，则有两个选择：既可以在模块接口文件中，也可以在模块实现文件中。

下面是带有类中方法实现的 SpreadsheetCell 类：

```

export module spreadsheet_cell;

export class SpreadsheetCell
{
    public:
        void setValue(double value) { m_value = value; }
        double getValue() const { return m_value; }
    private:
        double m_value { 0 };
};

```

与头文件不同，对于 C++20 模块，将方法定义放入模块接口文件中不会有任何危险。这将在第 11 章中进行更详细的讨论。然而，为了保持模块接口文件整洁且没有任何实现细节，本书通常将方法定义放在模块实现文件中。

模块实现文件的第一行指定实现的方法用于哪个模块。以下是 spreadsheet\_cell 模块中 SpreadsheetCell 类的两个方法的定义：

```
module spreadsheet_cell;
```

```

void SpreadsheetCell::setValue(double value)
{
    m_value = value;
}
double SpreadsheetCell::getValue() const
{
    return m_value;
}

```

注意，每个方法名之前都出现了类名和两个冒号。

```
void SpreadsheetCell::setValue(double value)
```

::称为作用域解析运算符(scope resolution operator)。在此处，这个语法告诉编译器，要定义的 setValue()方法是 SpreadsheetCell 类的一部分。此外还要注意，定义方法时，不要重复使用访问说明符。

## 1. 访问数据成员

类的非静态方法，例如 setValue() 和 getValue()，总是在类的特定对象上执行。在类的方法体中，可以访问对象所属类的所有数据成员。在前面的 setValue() 定义中，无论哪个对象调用这个方法，下面这行代码都会改变 m\_value 变量的值。

```
m_value = value;
```

如果两个不同的对象调用 setValue()，这行代码(对每个对象执行一次)会改变两个不同对象内的变量值。

## 2. 调用其他方法

内部的某个方法可调用其他方法。例如，考虑扩展后的 SpreadsheetCell 类，它允许在单元格中保存文本数据和数字。试图将文本单元格解释为数字时，电子表格会试着将文本转换为数字。如果这个文本不能代表一个有效的值，单元格的值会被忽略。在这个程序中，非数字的字符串会生成值为 0 的单元格。为让 SpreadsheetCell 支持文本数据，将类定义为如下：

```

export module spreadsheet_cell;
import <string>;
import <string_view>;
export class SpreadsheetCell
{
public:
    void setValue(double value);
    double getValue() const;

    void setString(std::string_view value);
    std::string getString() const;
private:
    std::string doubleToString(double value) const;
    double stringtoDouble(std::string_view value) const;
    double m_value { 0 };
};

```

这个类版本只能存储 double 数据。如果客户将数据设置为 string，数据就会转换为 double。如果文本不是有效数字，就将 double 值设置为 0。这个类定义显示了两个设置并获取单元格文本表示的新方法，还有两个新的用于将 double 转换为 string、将 string 转换为 double 的 private 辅助方法。下面是这些方法的实现。

```

module spreadsheet_cell;
import <charconv>;
using namespace std;

void SpreadsheetCell::setValue(double value)
{
    m_value = value;
}

double SpreadsheetCell::getValue() const
{
    return m_value;
}

void SpreadsheetCell::setString(string_view value)
{
    m_value = stringtoDouble(value);
}

string SpreadsheetCell::getString() const
{
    return doubleToString(m_value);
}

string SpreadsheetCell::doubleToString(double value) const
{
    return to_string(value);
}

double SpreadsheetCell::stringtoDouble(string_view value) const
{
    double number { 0 };
    from_chars(value.data(), value.data() + value.size(), number);
    return number;
}

```

`std::to_string()`和`from_chars()`函数在第2章“使用字符串和字符串视图”中介绍过。

注意`doubleToString()`方法的这种实现方式，例如，值6.1会被转换为6.100000。但由于这是一个private辅助方法，因此不必修改任何客户代码即可修改该实现。

### 3. this指针

每个普通的方法调用都会传递一个指向对象的指针，这就是称为“隐藏”参数的this指针。使用这个指针可访问数据成员或者调用方法，也可将其传递给其他方法或函数。有时还用它消除名称的歧义。例如，可使用`value`而不是`m_value`作为`SpreadsheetCell`类的数据成员。在此情况下，`setValue()`如下所示。

```

void SpreadsheetCell::setValue(double value)
{
    value = value; // Confusing!
}

```

加粗显示的代码行令人困惑。是哪个`value`呢？是作为参数传递的`value`，还是对象的成员`value`？

**注意：**

使用某些编译器或某些编译器设置时，前面的歧义行会编译成功，不会有任何警告或错误消息，但得到的结果并不是你所期望的。

为避免名称的歧义，可使用 `this` 指针：

```
void SpreadsheetCell::setValue(double value)
{
    this->value = value;
}
```

然而，如果使用第3章讲述的命名约定，永远不会遇到这类名称冲突问题。

如果对象的某个方法调用了某个函数(或方法)，而这个函数采用指向对象的指针作为参数，就可使用 `this` 指针调用这个函数。例如，假定编写了一个独立的 `printCell()` 函数(不是方法)，如下所示。

```
void printCell(const SpreadsheetCell& cell)
{
    cout << cell.getString() << endl;
}
```

如果想用 `setValue()` 方法调用 `printCell()`，就必须将 `*this` 作为参数，用来将 `setValue()` 操作的 `SpreadsheetCell` 对象的引用传递给 `printCell()`。

```
void SpreadsheetCell::setValue(double value)
{
    this->value = value;
    printCell(*this);
}
```

**注意：**

重载 `<<` 运算符(将在第15章讲述)比编写 `printCell()` 函数更方便，重载 `<<` 后，即可使用下面的代码输出 `SpreadsheetCell`。

```
cout << *this << endl;
```

### 8.2.3 使用对象

前面的 `SpreadsheetCell` 类定义包含1个数据成员、4个公有方法和2个私有方法。然而，类定义实际上并没有创建任何 `SpreadsheetCell`，而只是指定单元格的属性和行为。在某种意义上，类就像建筑蓝图。蓝图指定了房子的样子，但是绘制蓝图并没有构建任何房子，房子必须根据蓝图在后面建造。

与此类似，在C++中通过声明 `SpreadsheetCell` 类型的变量，可根据 `SpreadsheetCell` 类的定义构建一个 `SpreadsheetCell` “对象”。就像施工人员可以根据给定的蓝图建造多个房子一样，程序员可以根据 `SpreadsheetCell` 类创建多个 `SpreadsheetCell` 对象。可采用两种方法创建和使用对象：在栈中或者在自由存储区中。

#### 1. 栈中的对象

下面的代码在栈中创建并使用 `SpreadsheetCell` 对象。

```
SpreadsheetCell myCell, anotherCell;
myCell.setValue(6);
```

```
anotherCell.setString("3.2");
cout << "cell 1: " << myCell.getValue() << endl;
cout << "cell 2: " << anotherCell.getValue() << endl;
```

创建对象类似于声明简单变量，区别在于变量类型是类名。“myCell.setValue(6);”中的.称为“点”运算符，这个运算符允许调用对象的方法。如果对象中有公有数据成员，也可以用点运算符访问。注意不推荐使用公有数据成员。

程序的输出如下：

```
cell 1: 6
cell 2: 3.2
```

## 2. 自由存储区中的对象

还可使用 new 动态分配对象：

```
SpreadsheetCell* myCellp = new SpreadsheetCell { } ;
myCellp->setValue(3.7);
cout << "cell 1: " << myCellp->getValue() <<
    " " << myCellp->getString() << endl;
delete myCellp;
myCellp = nullptr;
```

在自由存储区中创建对象时，通过“箭头”运算符访问其成员。箭头运算符组合了解引用运算符(\*)和成员访问运算符(.)。可用这两个运算符替换箭头，但这么做在形式上很笨拙。

```
SpreadsheetCell* myCellp = new SpreadsheetCell { } ;
(*myCellp).setValue(3.7);
cout << "cell 1: " << (*myCellp).getValue() <<
    " " << (*myCellp).getString() << endl;
delete myCellp;
myCellp = nullptr;
```

就如同必须释放自由存储区中分配的其他内存一样，也必须在对象上调用 delete，释放自由存储区中为对象分配的内存。为了保证安全性，避免发生内存错误，强烈建议使用智能指针。例如：

```
auto myCellp = make_unique<SpreadsheetCell>();
// Equivalent to:
// unique_ptr<SpreadsheetCell> myCellp { new SpreadsheetCell { } };
myCellp->setValue(3.7);
cout << "cell 1: " << myCellp->getValue() <<
    " " << myCellp->getString() << endl;
```

使用智能指针时，不需要手动释放内存，内存会自动释放。

### 警告：

如果用 new 为某个对象分配内存，那么使用完对象后，要用 delete 释放对象，或者使用智能指针自动管理内存。

### 注意：

如果没有使用智能指针，当释放指针所指的对象时，最好将指针重置为 nullptr。这并非强制要求，但这样做可以防止在删除对象后意外使用这个指针，以便于调试。

## 8.3 对象的生命周期

对象的生命周期涉及 3 个活动：创建、销毁和赋值。理解对象什么时候以及如何被创建、销毁、赋值，并且知道如何定制这些行为很重要。

### 8.3.1 创建对象

在声明对象(如果是在栈中)或使用 new、new[] 或智能指针显式分配空间时，就会创建对象。当创建对象时，内嵌的对象也会被创建。例如：

```
import <string>

class MyClass
{
    private:
        std::string m_name;
};

int main()
{
    MyClass obj;
}
```

在 main() 函数中创建 MyClass 对象时，同时创建内嵌的 string 对象。当包含它的对象被销毁时，string 也被销毁。

在声明变量时最好给它们赋初始值。例如：

```
int x { 0 };
```

与此类似，也应该初始化对象。声明并编写一个名为构造函数的方法，可以提供这一功能，在构造函数中可以执行对象的初始化任务。无论什么时候创建对象，都会执行其构造函数。

#### 注意：

C++程序员有时将构造函数称为 ctor。

#### 1. 编写构造函数

从语法上讲，构造函数是与类同名的方法。构造函数没有返回类型，可以有也可以没有参数，没有参数的构造函数称为默认构造函数。可以是无参构造函数，也可以让所有参数都使用默认值。有些情况下，必须提供默认构造函数。如果不提供，就会导致编译器错误，默认构造函数将在稍后讨论。

下面试着在 SpreadsheetCell 类中添加一个构造函数：

```
export class SpreadsheetCell
{
    public:
        SpreadsheetCell(double initialValue);
        // Remainder of the class definition omitted for brevity
};
```

就像必须提供普通方法的实现一样，也必须提供构造函数的实现。

```
SpreadsheetCell::SpreadsheetCell(double initialValue)
{
    setValue(initialValue);
}
```

SpreadsheetCell 构造函数是 SpreadsheetCell 类的一个成员，因此 C++ 在构造函数的名称之前要求正常的 SpreadsheetCell:: 作用域解析。由于构造函数本身的名称也是 SpreadsheetCell，因此代码的 SpreadsheetCell::SpreadsheetCell 看上去有点好笑。这个实现只是简单地调用了 setValue() 方法。

## 2. 使用构造函数

构造函数用来创建对象并初始化其值。在基于栈和自由存储区进行分配时都可以使用构造函数。

### 在栈中使用构造函数

在栈中分配 SpreadsheetCell 对象时，可这样使用构造函数：

```
SpreadsheetCell myCell(5), anotherCell(4);
cout << "cell 1: " << myCell.getValue() << endl;
cout << "cell 2: " << anotherCell.getValue() << endl;
```

或者，可以使用统一初始化语法：

```
SpreadsheetCell myCell { 5 }, anotherCell { 4 };
```

注意，不要显式地调用 SpreadsheetCell 构造函数。例如，不要采用下面的做法：

```
SpreadsheetCell myCell.SpreadsheetCell(5); // WILL NOT COMPILE!
```

同样，之后也不能调用构造函数。下面的代码也是不正确的：

```
SpreadsheetCell myCell;
myCell.SpreadsheetCell(5); // WILL NOT COMPILE!
```

### 在自由存储区中使用构造函数

当动态分配 SpreadsheetCell 对象时，可这样使用构造函数：

```
auto smartCellp = make_unique<SpreadsheetCell>(4);
// ... do something with the cell, no need to delete the smart pointer

// Or with raw pointers, without smart pointers (not recommended)
SpreadsheetCell* myCellp = new SpreadsheetCell { 5 };
// Or
// SpreadsheetCell* myCellp = new SpreadsheetCell(5);
SpreadsheetCell* anotherCellp = nullptr;
anotherCellp = new SpreadsheetCell { 4 };
// ... do something with the cells
delete myCellp; myCellp = nullptr;
delete anotherCellp; anotherCellp = nullptr;
```

注意可以声明一个指向 SpreadsheetCell 对象的指针，而不立即调用构造函数。与此不同的是，栈中的对象在声明时会调用构造函数。

请记住，总是应该初始化指针，可以使用合适的指针，也可以使用 nullptr。

### 3. 提供多个构造函数

在一个类中可提供多个构造函数。所有构造函数的名称相同(类名)，但不同的构造函数具有不同数量的参数或者不同的参数类型。在 C++ 中，如果多个函数具有相同的名称，那么当调用时编译器会选择参数类型匹配的那个函数。这称为重载，第 9 章将详细讨论。

在 SpreadsheetCell 类中，编写两个构造函数是有益的：一个采用 double 初始值，另一个采用 string 初始值。这是新的类定义：

```
export class SpreadsheetCell
{
public:
    SpreadsheetCell(double initialValue);
    SpreadsheetCell(std::string_view initialValue);
    // Remainder of the class definition omitted for brevity
};
```

下面是第二个构造函数的实现：

```
SpreadsheetCell::SpreadsheetCell(string_view initialValue)
{
    setString(initialValue);
}
```

下面是使用两个不同构造函数的代码：

```
SpreadsheetCell aThirdCell { "test" };           // Uses string-arg ctor
SpreadsheetCell aFourthCell { 4.4 };             // Uses double-arg ctor
auto aFifthCellp { make_unique<SpreadsheetCell>("5.5") }; // string-arg ctor
cout << "aThirdCell: " << aThirdCell.getValue() << endl;
cout << "aFourthCell: " << aFourthCell.getValue() << endl;
cout << "aFifthCellp: " << aFifthCellp->getValue() << endl;
```

当具有多个构造函数时，在一个构造函数中执行另一个构造函数的想法很诱人。例如，以下面的方式让 string 构造函数调用 double 构造函数。

```
SpreadsheetCell::SpreadsheetCell(string_view initialValue)
{
    SpreadsheetCell(string.ToDouble(initialValue));
}
```

这看上去是合理的。因为可在类的一个方法中调用另一个方法。这段代码可以编译、链接并运行，但结果并非预期的那样。显式调用 SpreadsheetCell 构造函数实际上新建了一个 SpreadsheetCell 类型的临时未命名对象，而并不是像预期的那样调用构造函数以初始化对象。

然而，C++ 支持委托构造函数(delegating constructors)，允许构造函数初始化器调用同一个类的其他构造函数。这一内容将在本章后面讨论。

### 4. 默认构造函数

默认构造函数没有参数，也称为无参构造函数。

#### 什么时候需要默认构造函数

考虑一下对象数组。创建对象数组需要完成两个任务：为所有对象分配内存连续的空间，为每个对象调用默认构造函数。C++ 没有提供任何语法，使创建数组的代码直接调用不同的构造函数。例如，如果没有定义 SpreadsheetCell 类的默认构造函数，下面的代码将无法编译。

```
SpreadsheetCell cells[3]; // FAILS compilation without default constructor
SpreadsheetCell* myCellp = new SpreadsheetCell[10]; // Also FAILS
```

可使用下面的初始化器绕过这个限制：

```
SpreadsheetCell cells[3] { SpreadsheetCell { 0 }, SpreadsheetCell { 23 },
                           SpreadsheetCell { 41 } };
```

然而，如果想创建某个类的对象数组，最好还是定义类的默认构造函数。如果没有定义自己的构造函数，编译器会自动创建默认构造函数。编译器生成的构造函数在下一节讨论。

### 如何编写默认构造函数

下面是具有默认构造函数的 SpreadsheetCell 类的部分定义：

```
export class SpreadsheetCell
{
public:
    SpreadsheetCell();
    // Remainder of the class definition omitted for brevity
};
```

下面的代码首次实现了默认构造函数：

```
export SpreadsheetCell::SpreadsheetCell()
{
    m_value = 0;
}
```

如果为 `m_value` 使用类内成员初始化方式，则可以省略这个默认构造函数中的一条语句。

```
SpreadsheetCell::SpreadsheetCell()
{
}
```

可在栈中使用默认构造函数，如下所示：

```
SpreadsheetCell myCell;
myCell.setValue(6);
cout << "cell 1: " << myCell.getValue() << endl;
```

前面的代码创建了一个名为 `myCell` 的新 `SpreadsheetCell`，设置并输出其值。与基于栈的对象的其他构造函数不同，调用默认构造函数不需要使用函数调用的语法。根据其他构造函数的语法，用户或许会试着这样调用默认构造函数：

```
SpreadsheetCell myCell(); // WRONG, but will compile.
myCell.setValue(6); // However, this line will not compile.
cout << "cell 1: " << myCell.getValue() << endl;
```

但是，试图调用默认构造函数的行可以编译，但是后面的代码无法编译。问题在于常说的最头疼的解析(most vexing parse)，编译器实际上将第一行当作函数声明，函数名为 `myCell`，没有参数，返回值为 `SpreadsheetCell` 对象。当编译第二行时，编译器认为用户将函数名用作对象！

当然，可以使用统一初始化语法，而不是使用函数调用样式的括号，如下所示。

```
SpreadsheetCell myCell {} // Calls the default constructor.
```

**警告：**

在栈中创建对象时，可以使用花括号作为统一初始化语法，也可以省略任何括号。

对于自由存储区中的对象，可以这样使用默认构造函数：

```
auto smartCellp { make_unique<SpreadsheetCell>() };
// Or with a raw pointer (not recommended)
SpreadsheetCell* myCellp { new SpreadsheetCell {} };
// Or
// SpreadsheetCell* myCellp { new SpreadsheetCell };
// Or
// SpreadsheetCell* myCellp { new SpreadsheetCell() };
// ... use myCellp
delete myCellp; myCellp = nullptr;
```

**编译器生成的默认构造函数**

本章的第一个 `SpreadsheetCell` 类定义如下所示：

```
export class SpreadsheetCell
{
    public:
        void setValue(double value);
        double getValue() const;
    private:
        double m_value;
};
```

这个类定义没有声明任何默认构造函数，但以下代码仍然可以正常运行：

```
SpreadsheetCell myCell;
myCell.setValue(6);
```

下面的定义与前面的定义相同，只是添加了一个显式的构造函数，用一个 `double` 值作为参数。这个定义仍然没有显式声明默认构造函数：

```
export class SpreadsheetCell
{
    public:
        SpreadsheetCell(double initialValue); // No default constructor
        // Remainder of the class definition omitted for brevity
};
```

使用这个定义，下面的代码将无法编译：

```
SpreadsheetCell myCell;
myCell.setValue(6);
```

为什么会这样？原因在于如果没有指定任何构造函数，编译器将自动生成无参构造函数。类所有的对象成员都可以调用编译器生成的默认构造函数，但不会初始化语言的基本类型，例如 `int` 和 `double`。尽管如此，也可用它创建类的对象。然而，如果声明了任何构造函数，编译器就不会再自动生成默认构造函数。

**注意：**

默认构造函数与无参构造函数是一回事。术语“默认构造函数”并不仅仅是说如果没有声明任何构造函数，就会自动生成一个构造函数；而且指如果不需要参数，构造函数即为默认构造函数。

**显式默认的默认构造函数**

在 C++11 前，如果类需要一些接收参数的显式构造函数，还需要一个什么都不做的默认构造函数，就必须显式地编写空的默认构造函数，如前所述。

为了避免手动编写空的默认构造函数，C++现在支持显式默认的默认构造函数(explicitly defaulted default constructor)。可按如下方法编写类的定义，而不需要在实现文件中实现默认构造函数：

```
export class SpreadsheetCell
{
public:
    SpreadsheetCell() = default;
    SpreadsheetCell(double initialValue);
    SpreadsheetCell(std::string_view initialValue);
    // Remainder of the class definition omitted for brevity
};
```

`SpreadsheetCell` 定义了两个定制的构造函数。然而，由于使用了 `default` 关键字，编译器仍然会生成一个标准的由编译器生成的默认构造函数。

注意，既可以将`=default` 直接放在类定义中，也可以放在实现文件中。

**显式删除的默认构造函数**

C++还支持显式删除的默认构造函数(explicitly deleted default constructor)。例如，可定义一个只有静态方法的类(见第 9 章)，这个类没有任何构造函数，也不想让编译器生成默认构造函数。在此情况下可以显式删除默认构造函数：

```
export class MyClass
{
public:
    MyClass() = delete;
};
```

**注意：**

如果类的数据成员具有删除的默认构造函数，则该类的默认构造函数也会自动删除。

**5. 构造函数初始化器**

本章到现在为止，都是在构造函数体内初始化数据成员，例如：

```
SpreadsheetCell::SpreadsheetCell(double initialValue)
{
    setValue(initialValue);
}
```

C++提供了另一种在构造函数中初始化数据成员的方法，称为构造函数初始化器或成员初始化列表，也可称为 `ctor-initializer`。下面的代码使用构造函数初始化器语法重写了 `SpreadsheetCell` 构造函数：

```
SpreadsheetCell::SpreadsheetCell(double initialValue)
    : m_value { initialValue }
{
```

可以看出，构造函数初始化器出现在构造函数参数列表和构造函数体的左大括号之间。这个列表以冒号开始，由逗号分隔。列表中的每个元素都使用函数符号、统一初始化语法、调用基类构造函数（见第10章），或者调用委托构造函数（参见后面的内容）以初始化某个数据成员。

使用构造函数初始化器与在构造函数体内初始化数据成员不同。当C++创建某个对象时，必须在调用构造函数前创建对象的所有数据成员。如果数据成员本身就是对象，那么在创建这些数据成员时，必须为其调用构造函数。在构造函数体内给某个对象赋值时，并没有真正创建这个对象，而只是改变对象的值。构造函数初始化器允许在创建数据成员时赋初值，这样做比在后面赋值效率高。

如果类的数据成员是具有默认构造函数的类的对象，则不必在构造函数初始化器中显式初始化对象。例如，如果有一个`std::string`数据成员，其默认构造函数将字符串初始化为空字符串，那么在构造函数初始化器中将其初始化为“”是多余的。

而如果类的数据成员是没有默认构造函数的类的对象，则必须在构造函数初始化器中显式初始化对象。例如，考虑下面的`SpreadsheetCell`类：

```
export class SpreadsheetCell
{
public:
    SpreadsheetCell(double d);
};
```

这个类只有一个采用`double`值作为参数的显式构造函数，而没有默认构造函数。可在另一个类中将这个类用作数据成员，如下所示：

```
export class SomeClass
{
public:
    SomeClass();
private:
    SpreadsheetCell m_cell;
};
```

`SomeClass`构造函数的实现如下：

```
SomeClass::SomeClass() {}
```

然而，使用这个实现将无法成功编译。编译器不知道如何初始化`SomeClass`类的`m_cell`数据成员，因为这个类没有默认构造函数。

解决方案是在构造函数初始化器中初始化`m_cell`数据成员，如下所示。

```
SomeClass::SomeClass() : m_cell(1.0) {}
```

### 注意：

构造函数初始化器允许在创建数据成员时执行初始化。

某些程序员喜欢在构造函数体内提供初始值（即使这么做效率不高）。然而，某些数据类型必须在构造函数初始化器中或使用类内初始化器进行初始化。表8-1对此进行了总结。

表 8-1 数据类型

数据类型	说明
const 数据成员	const 变量创建后无法对其正确赋值，必须在创建时提供初始值
引用数据成员	如果不指向什么，引用将无法存在，且一旦创建，引用不得改变指向目标
没有默认构造函数的对象数据成员	C++尝试用默认构造函数初始化成员对象。如果不存在默认构造函数，就无法初始化该对象，必须显式调用它的某个构造函数
没有默认构造函数的基类	在第 10 章讲述

关于构造函数初始化器要特别注意，数据成员的初始化顺序如下：按照它们在类定义中出现的顺序，而不是在构造函数初始化器中的顺序。考虑下面的 Foo 类定义。它的构造函数只是存储 double 值，并将该值显示在控制台上。

```
class Foo
{
public:
    Foo(double value);
private:
    double m_value { 0 };
};

Foo::Foo(double value) : m_value { value }
{
    cout << "Foo::m_value = " << m_value << endl;
}
```

假定有另一个类 MyClass，它将 Foo 对象作为自己的一个数据成员。

```
class MyClass
{
public:
    MyClass(double value);
private:
    double m_value { 0 };
    Foo m_foo;
};
```

其构造函数的实现方式如下：

```
MyClass::MyClass(double value) : m_value { value }, m_foo { m_value }
{
    cout << "MyClass::m_value = " << m_value << endl;
}
```

构造函数初始化器首先在 m\_value 中存储给定的值，然后将 m\_value 作为实参调用 Foo 构造函数。可以创建 MyClass 的实例，如下所示：

```
MyClass instance { 1.2 };
```

输出如下所示：

```
Foo::m_value = 1.2
MyClass::m_value = 1.2
```

看上去一切都不错。现在对 MyClass 定义稍加修改，只是调换 m\_value 和 m\_foo 数据成员的顺序，其他保持不变。

```
class MyClass
{
public:
    MyClass(double value);
private:
    Foo m_foo;
    double m_value { 0 };
};
```

现在，程序的输出取决于系统。可能的示例输出如下：

```
Foo::m_value = -9.25596e+61
MyClass::m_value = 1.2
```

这与我们的期望相去甚远。你可能认为，基于构造函数初始化器，先初始化 m\_value，再在调用 Foo 构造函数时使用 m\_value。但 C++ 并不是这么运行的。按照数据成员在类定义中的顺序(而非构造函数初始化器中的顺序)对数据成员进行初始化。因此，这里首先使用未初始化的 m\_value 调用 Foo 构造函数。

注意，如果类定义中的顺序与构造函数初始化器中的顺序不匹配，一些编译器会发出警告。

对于本例，有一个简单的修复方法。不要将 m\_value 传递给 Foo 构造函数，只需要传递 value 参数。

```
MyClass::MyClass(double value) : m_value { value }, m_foo { value }
{
    cout << "MyClass::m_value = " << m_value << endl;
}
```

### 警告：

使用构造函数初始化器初始化数据成员的顺序如下：按照类定义中声明的顺序而不是构造函数初始化器列表中的顺序。

## 6. 拷贝构造函数

C++ 中有一种特殊的构造函数，称为拷贝构造函数(copy constructor)，允许所创建的对象是另一个对象的副本。下面是 SpreadsheetCell 类中拷贝构造函数的声明：

```
export class SpreadsheetCell
{
public:
    SpreadsheetCell(const SpreadsheetCell& src);
    // Remainder of the class definition omitted for brevity
};
```

拷贝构造函数采用源对象的 const 引用作为参数。与其他构造函数类似，它也没有返回值。在拷贝构造函数内部，应该复制源对象的所有数据成员。当然，严格来说，可在拷贝构造函数内完成任何操作，但最好按照预期的行为将新对象初始化为已有对象的副本。下面是 SpreadsheetCell 拷贝构造函数的示例实现，注意构造函数初始化器的用法。

```
SpreadsheetCell::SpreadsheetCell(const SpreadsheetCell& src)
    : m_value { src.m_value }
```

如果没有编写拷贝构造函数，C++会自动生成一个，用源对象中相应数据成员的值初始化新对象的每个数据成员。如果数据成员是对象，初始化意味着调用它们的拷贝构造函数。假定有一组成员变量，名为 m1、m2、...、mn，编译器生成的拷贝构造函数为：

```
classname::classname(const classname& src)
: m1 { src.m1 }, m2 { src.m2 }, ... mn { src.mn } {}
```

因此，多数情况下，不需要亲自编写拷贝构造函数！

### 注意：

此处显示 SpreadsheetCell 拷贝构造函数只是为了演示目的。实际上，这种情况下，由于默认的由编译器生成的构造函数已经足以满足要求，因此可以省去拷贝构造函数。然而在某些情况下，默认拷贝构造函数的功能不足。第 9 章将讲述这些情况。

### 何时调用拷贝构造函数

C++中传递函数参数的默认方式是值传递，这意味着函数或方法接收某个值或对象的副本。因此，无论何时给函数或方法传递一个对象，编译器都会调用新对象的拷贝构造函数进行初始化。例如，假设以下 printString() 函数接收一个按值传递的 string 参数：

```
void printString(string value)
{
    cout << value << endl;
}
```

回顾一下，C++字符串实际上是一个类，而不是内置类型。当调用 setString() 并传递一个 string 参数时，这个 string 参数 inString 会调用拷贝构造函数进行初始化。传递给拷贝构造函数的参数是传递给 printString() 的字符串。在下例中，为初始化 printString() 中的 inString 对象，会调用 string 拷贝构造函数，其参数为 name。

```
string name { "heading one" };
printString(name); // Copies name
```

当 printString() 方法结束时，inString 被销毁，因为它只是 name 的一个副本，所以 name 完好无缺。当然，可通过将参数作为 const 的引用来传递，从而避免拷贝构造函数的开销。

当函数按值返回对象时，也会调用拷贝构造函数，见本章 8.3.5 节的“1.按值返回对象”。

### 显式调用拷贝构造函数

也可显式地使用拷贝构造函数，从而将某个对象作为另一个对象的精确副本。例如，可这样创建 SpreadsheetCell 对象的副本：

```
SpreadsheetCell myCell1 { 4 };
SpreadsheetCell myCell2 { myCell1 }; // myCell2 has the same values as myCell1
```

### 按引用传递对象

向函数或方法传递对象时，为避免复制对象，可让函数或方法采用对象的引用作为参数。按引用传递对象通常比按值传递对象的效率更高，因为只需要复制对象的地址，而不需要复制对象的全部内容。此外，按引用传递可避免对象动态内存分配的问题，这些内容将在第 9 章讲述。

按引用传递某个对象时，使用对象引用的函数或方法可修改原始对象。如果只是为了提高效率才按引用传递，可将对象声明为 const 以排除这种可能。这称为按 const 的引用传递对象，本书中的多个示例一直是这么做的。

**注意：**

为了提高性能，最好按 `const` 引用而不是按值传递对象。在引入移动语义之后，第 9 章稍微修改了这个规则，允许在某些情况下传递对象的值。

注意，`SpreadsheetCell` 类有多个接收 `std::string_view` 参数的方法。如第 2 章所述，`string_view` 基本上就是指针和长度。因此，复制成本很低，通常按值传递。

另外，诸如 `int` 和 `double` 等基本类型应当按值传递。按 `const` 引用传递这些类型什么好处也得不到。

`SpreadsheetCell` 类的 `doubleToString()` 方法总是按值返回字符串，因为该方法的实现创建了一个局部字符串对象，在方法的最后返回给调用者。返回这个字符串的引用是无效的，因为它引用的字符串在函数退出时会被销毁。

**显式默认或显式删除的拷贝构造函数**

正如可以将编译器生成的默认构造函数设置为显式默认或显式删除，同样可以将编译器生成的拷贝构造函数设置为默认或者将其删除。

```
SpreadsheetCell(const SpreadsheetCell& src) = default;
```

或者

```
SpreadsheetCell(const SpreadsheetCell& src) = delete;
```

通过删除拷贝构造函数，对象将无法被复制。这可用于禁止按值传递对象，如第 9 章所述。

**注意：**

如果类的数据成员具有删除的复制构造函数，则该类的复制构造函数也会自动删除。

**7. 初始化列表构造函数**

初始化列表构造函数(initializer-list constructor)将 `std::initializer_list<T>` 作为第一个参数，并且没有任何其他参数(或者其他参数具有默认值)。`std::initializer_list<T>` 类模板定义在`<initializer_list>`中。下面的类演示了这种用法。该类只接收一个 `initializer_list<double>` 参数，且初始化列表中的元素个数应为偶数，否则将抛出异常。异常在第 1 章介绍过。

```
class EvenSequence
{
public:
    EvenSequence(initializer_list<double> args)
    {
        if (args.size() % 2 != 0) {
            throw invalid_argument("initializer_list should "
                "contain even number of elements.");
        }
        m_sequence.reserve(args.size());
        for (const auto& value : args) {
            m_sequence.push_back(value);
        }
    }

    void dump() const
    {
        for (const auto& value : m_sequence) {
            cout << value << ", ";
        }
    }
}
```

```

        cout << endl;
    }
private:
    vector<double> m_sequence;
};

```

在初始化列表构造函数的内部，可使用基于范围的 for 循环来访问初始化列表的元素。使用 size() 方法可获取初始化列表中元素的数目。

EvenSequence 初始化列表构造函数使用基于区间的 for 循环来复制给定 initializer\_list<T> 中的元素。也可以使用 vector 的 assign() 方法。vector 的 assign() 等方法详见第 18 章。为帮助你了解 vector 的功能，下面给出一个使用 assign() 的初始化列表构造函数。

```

EvenSequence(initializer_list<double> args)
{
    if (args.size() % 2 != 0) {
        throw invalid_argument { "initializer_list should "
            "contain even number of elements." };
    }
    m_sequence.assign(args);
}

```

可按以下方式创建 EvenSequence 对象：

```

EvenSequence p1 { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0 };
p1.dump();

```

```

try {
    EvenSequence p2 { 1.0, 2.0, 3.0 };
} catch (const invalid_argument& e) {
    cout << e.what() << endl;
}

```

创建 p2 时会抛出异常，因为初始化列表中元素的数目为奇数。

标准库完全支持初始化列表构造函数。例如，可使用初始化列表初始化 std::vector 容器。

```
vector<string> myVec { "String 1", "String 2", "String 3" };
```

如果不使用初始化列表构造函数，可通过一些 push\_back() 调用来初始化 vector。

```

vector< string> myVec;
myVec.push_back("String 1");
myVec.push_back("String 2");
myVec.push_back("String 3");

```

初始化列表并不限于构造函数，还可以用于普通函数，如第 1 章所述。

## 8. 委托构造函数

委托构造函数(delegating constructor)允许构造函数调用同一个类的其他构造函数。然而，这个调用不能放在构造函数体内，而必须放在构造函数初始化器中，且必须是列表中唯一的成员初始化器。下面给出了一个示例：

```

SpreadsheetCell::SpreadsheetCell(string_view initialValue)
    : SpreadsheetCell { stringToDouble(initialValue) }
{
}

```

当调用这个 `string_view` 构造函数(委托构造函数)时，首先将调用委托给目标构造函数，也就是 `double` 构造函数。当目标构造函数返回时，再执行委托构造函数。

当使用委托构造函数时，要注意避免出现构造函数的递归。例如：

```
class MyClass
{
    MyClass(char c) : MyClass(1.2) { }
    MyClass(double d) : MyClass('m') { }
};
```

第一个构造函数委托第二个构造函数，第二个构造函数又委托第一个构造函数。C++标准没有定义此类代码的行为，这取决于编译器。

## 9. 转换构造函数和显式构造函数

目前，`SpreadsheetCell` 类的构造函数集合如下所示。

```
export class SpreadsheetCell
{
public:
    SpreadsheetCell() = default;
    SpreadsheetCell(double initialValue);
    SpreadsheetCell(std::string_view initialValue);
    SpreadsheetCell(const SpreadsheetCell& src);
    // Remainder omitted for brevity
};
```

单参数 `double` 和 `string_view` 构造函数可用于将 `double` 或 `string_view` 转换为 `SpreadsheetCell`。这种构造函数称为转换构造函数。编译器可以使用这些构造函数执行隐式转换。例如：

```
SpreadsheetCell myCell { 4 };
myCell = 5;
myCell = "6"sv; // A string_view literal (see Chapter 2).
```

这可能并不总是想要的行为。通过将构造函数标记为 `explicit`，可以防止编译器执行此类隐式转换。`explicit` 关键字只在类定义中出现。例如：

```
export class SpreadsheetCell
{
public:
    SpreadsheetCell() = default;
    SpreadsheetCell(double initialValue);
    explicit SpreadsheetCell(std::string_view initialValue);
    SpreadsheetCell(const SpreadsheetCell& src);
    // Remainder omitted for brevity
};
```

改写之后，如下所示的代码将无法编译。

```
myCell = "6"sv; // A string_view literal (see Chapter 2).
```

### 注意：

建议将任何可以使用单个参数调用的构造函数标记为 `explicit`，以避免意外的隐式转换，除非此类转换是有用的。

在 C++11 之前，转换构造函数只能有一个参数，如 `SpreadsheetCell` 示例中所示。自 C++11 以来，

由于支持列表初始化，转换构造函数可以有多个参数。让我们看一个例子，假设有以下类：

```
class MyClass
{
public:
    MyClass(int) { }
    MyClass(int, int) { }
};
```

这个类有两个构造函数，从 C++11 开始，它们都是转换构造函数。以下示例展示了编译器使用这些转换构造函数自动将给定参数 1、{1} 和 {1,2} 转换为 MyClass 的实例：

```
void process(const MyClass& c) { }

int main()
{
    process(1);
    process({ 1 });
    process({ 1, 2 });
}
```

为了避免编译器执行此类隐式转换，两个转换构造函数都可以标记为 explicit。

```
class MyClass
{
public:
    explicit MyClass(int) { }
    explicit MyClass(int, int) { }
};
```

有了这个变更，必须显式地执行这些转换，例如：

```
process(MyClass{ 1 });
process(MyClass{ 1, 2 });
```

C++20 从 C++20 开始，可以将布尔参数传递给 explicit，以将其转换为条件 explicit。语法如下：

```
explicit(true) MyClass(int);
```

当然，仅仅编写 explicit(true) 就等价于 explicit，但它在使用所谓类型萃取的泛型模板代码中更加有用。使用类型萃取，可以查询给定类型的某些属性，例如某个类型是否可转换为另一个类型。类型萃取的结果可以用作 explicit() 的参数。类型萃取允许编写高级泛型代码，在第 26 章“高级模板”中讨论。

### 总结编译器生成的构造函数

编译器为每个类自动生成默认构造函数和拷贝构造函数。然而，编译器自动生成的构造函数取决于你自己定义的构造函数，对应的规则如表 8-2 所示。

表 8-2 对应的规则

如果定义了	那么编译器会生成	然后可以创建一个对象
没有定义构造函数	一个无参构造函数以及一个拷贝构造函数	使用无参构造函数： SpreadsheetCell a; 作为另一个对象的副本： SpreadsheetCell b{a};

(续表)

如果定义了	那么编译器会生成	然后可以创建一个对象
只定义了默认构造函数	一个拷贝构造函数	使用无参构造函数: SpreadsheetCell a; 作为另一个对象的副本: SpreadsheetCell b{a};
只定义了拷贝构造函数	不会生成构造函数	理论上可复制其他对象,但由于缺少拷贝构造函数以外的构造函数,实际上无法创建任何对象
只定义了一个构造函数(不是拷贝构造函数),该构造函数具有一个或多个参数	一个拷贝构造函数	使用带参数的构造函数: SpreadsheetCell a{6}; 作为另一个对象的副本: SpreadsheetCell b{a};
定义了一个默认构造函数以及一个具有单个或多个参数的构造函数(不是拷贝构造函数)	一个拷贝构造函数	使用无参构造函数: SpreadsheetCell a; 使用带参数的构造函数: SpreadsheetCell b{5}; 作为另一个对象的副本: SpreadsheetCell c{a};

注意默认构造函数和拷贝构造函数之间缺乏对称性。只要没有显式定义拷贝构造函数,编译器就会自动生成一个。另一方面,只要定义了任何构造函数,编译器就不会生成默认构造函数。

本章前面提到过,可通过将构造函数定义为显式默认或显式删除来影响自动生成的默认构造函数和默认拷贝构造函数。

#### 注意:

构造函数的最后一种类型是移动构造函数,用于实现移动语义。移动语义可用于在某些情况下提高性能,详见第9章。

### 8.3.2 销毁对象

当销毁对象时,会发生两件事:调用对象的析构函数,释放对象占用的内存。在析构函数中可以执行对象的清理,例如释放动态分配的内存或者关闭文件句柄。如果没有声明析构函数,编译器将自动生成一个,析构函数会逐一销毁成员,然后允许释放对象。类的析构函数是一个方法,其名称为类的名称,前缀为波浪号(~),并且不返回任何内容。下面是一个析构函数示例,它只将某些内容写入标准输出:

```
export class SpreadsheetCell
{
    public:
        ~SpreadsheetCell(); // Destructor.
        // Remainder of the class definition omitted for brevity
};

SpreadsheetCell::~SpreadsheetCell()
```

```
{
    cout << "Destructor called." << endl;
}
```

当栈中的对象超出作用域时，即当前的函数、方法或其他执行的代码块结束，对象会被销毁。换句话说，当代码遇到结束大括号时，这个大括号中所有创建在栈中的对象都会被销毁。下面的程序显示了这一行为：

```
int main()
{
    SpreadsheetCell myCell { 5 };
    if (myCell.getValue() == 5) {
        SpreadsheetCell anotherCell { 6 };
    } // anotherCell is destroyed as this block ends.

    cout << "myCell: " << myCell.getValue() << endl;
} // myCell is destroyed as this block ends.
```

栈中对象的销毁顺序与声明顺序(和构建顺序)相反。例如，在下面的代码片段中，myCell2 在 anotherCell2 之前分配，因此 anotherCell2 在 myCell2 之前销毁(注意在程序中，可以使用左大括号在任意点开始新的代码块)。

```
{
    SpreadsheetCell myCell2 { 4 };
    SpreadsheetCell anotherCell2 { 5 }; // myCell2 constructed before anotherCell2
} // anotherCell2 destroyed before myCell2
```

如果某个对象是其他对象的数据成员，这一顺序也适用。数据成员的初始化顺序是它们在类中声明的顺序。因此，按对象的销毁顺序与创建顺序相反这一规则，数据成员对象的销毁顺序与其在类中声明的顺序相反。

没有智能指针的帮助，在自由存储区中分配的对象不会自动销毁。必须对对象指针使用 `delete`，从而调用析构函数并释放内存。下面的程序显示了这一行为：

```
int main()
{
    SpreadsheetCell* cellPtr1 { new SpreadsheetCell { 5 } };
    SpreadsheetCell* cellPtr2 { new SpreadsheetCell { 6 } };
    cout << "cellPtr1: " << cellPtr1->getValue() << endl;
    delete cellPtr1; // Destroys cellPtr1
    cellPtr1 = nullptr;
} // cellPtr2 is NOT destroyed because delete was not called on it.
```

### 警告：

不要编写与上例类似的程序，因为 `cellPtr2` 没有释放。一定要根据分配内存时使用的是 `new` 还是 `new[]`，相应地使用 `delete` 或 `delete[]` 释放动态分配的内存。更好的方式是使用前面讲过的智能指针。

### 8.3.3 对象赋值

就像可将一个 `int` 变量的值赋给另一个 `int` 变量一样，在 C++ 中也可将一个对象的值赋给另一个对象。例如，下面的代码将 `myCell` 的值赋给 `anotherCell`：

```
SpreadsheetCell myCell { 5 }, anotherCell;
anotherCell = myCell;
```

myCell 似乎被“复制”给了 anotherCell。然而在 C++ 中，“复制”只在初始化对象时发生。如果一个已经具有值的对象被改写，更精确的术语是“赋值”。注意 C++ 提供的复制工具是拷贝构造函数。因为这是一个构造函数，所以只能用在创建对象时，而不能用于对象的赋值。

因此，C++ 为所有的类提供了执行赋值的方法。这个方法称为赋值运算符(assignment operator)，名称是 operator=，因为实际上是为类重载了=运算符。在上例中，调用了 anotherCell 的赋值运算符，参数为 myCell。

#### 注意：

本节所讲的赋值运算符有时也称为拷贝赋值运算符(copy assignment operator)，因为在赋值后，左边和右边的对象都继续存在。第 9 章将讨论移动赋值运算符(move assignment operator)，为提高性能，当赋值结束后右边的对象会被销毁。

如果没有编写自己的赋值运算符，C++ 将自动生成一个，从而允许将对象赋给另一个对象。默认的 C++ 赋值行为几乎与默认的复制行为相同：以递归方式用源对象的每个数据成员并赋值给目标对象。

#### 1. 声明赋值运算符

下面是 SpreadsheetCell 类的赋值运算符：

```
export class SpreadsheetCell
{
    public:
        SpreadsheetCell& operator=(const SpreadsheetCell& rhs);
        // Remainder of the class definition omitted for brevity
};
```

赋值运算符与拷贝构造函数类似，采用了源对象的 const 引用。在此情况下，将源对象称为 rhs，代表等号的“右边”(可为其指定其他任何名称)，调用赋值运算符的对象在等号的左边。

与拷贝构造函数不同的是，赋值运算符返回 SpreadsheetCell 对象的引用。原因是赋值可以连接在一起，如下所示：

```
myCell = anotherCell = aThirdCell;
```

执行这一行时，首先对 anotherCell 调用赋值运算符，aThirdCell 是“右边”的参数。随后对 myCell 调用赋值运算符。然而，此时 anotherCell 并不是参数，右边的值是将 aThirdCell 赋值给 anotherCell 时赋值运算符的返回值。如果赋值运算符不返回结果，myCell 将无法赋值。等号实际上是方法调用的缩写，看看下面包含完整函数语法的代码时，就会发现问题。

```
myCell.operator=(anotherCell.operator=(aThirdCell));
```

现在可以看到，anotherCell 调用的 operator= 必须返回一个值，这个值会传递给 myCell 调用的 operator=。正确的返回值是 anotherCell 本身，这样它就可以赋值给 myCell 的源对象。

#### 警告：

实际上可使赋值运算符返回任意类型，包括 void。然而应该返回被调用对象的引用，因为客户希望这样。

## 2. 定义赋值运算符

赋值运算符的实现与拷贝构造函数类似，但存在一些重要的区别。首先，拷贝构造函数只有在初始化时才调用，此时目标对象还没有有效的值。赋值运算符可以改写对象的当前值。在对象中拥有动态分配的资源(例如内存)之前，可以不考虑这个问题，第9章将详细讨论这个主题。

其次，在C++中允许将对象的值赋给自身。例如，下面的代码可以编译并运行：

```
SpreadsheetCell cell { 4 };
cell = cell; // Self-assignment
```

赋值运算符不应该阻止自我赋值。在SpreadsheetCell类中，这并不重要，因为它的唯一数据成员是基本类型double。但当类具有动态分配的内存或其他资源时，必须将自我赋值考虑在内，第9章将详细讨论。为阻止在此类情况下发生问题，赋值运算符通常在方法开始时检测自我赋值，如果发现自我赋值，则立刻返回。

下面是SpreadsheetCell类的赋值运算符的定义的开始：

```
SpreadsheetCell& SpreadsheetCell::operator=(const SpreadsheetCell& rhs)
{
    if (this == &rhs) {
```

第一行检测自我赋值，但有一个神秘之处。当等号的左边和右边相同时，就是自我赋值。判断两个对象是否相同的方法之一是检查它们在内存中的位置是否相同——更明确地说，是检查指向它们的指针是否相等。回顾一下，调用对象上的任何方法时，都会使用指向这个对象的this指针。因此，this是一个指向左边对象的指针。与此类似，&rhs是一个指向右边对象的指针。如果这两个指针相等，赋值必然是自我赋值。但由于返回类型是SpreadsheetCell&，因此必须返回一个正确的值。所有赋值运算符都返回\*this，自我赋值情况也不例外。

```
    return *this;
}
```

this指针指向执行方法的对象，因此\*this就是对象本身。编译器将返回一个对象的引用，从而与声明的返回值匹配。如果不是自我赋值，就必须对每个成员赋值。

```
m_value = rhs.m_value;
return *this;
}
```

这个方法复制了成员的值，最后返回\*this，前面已经对此做过解释。

### 注意：

此处显示SpreadsheetCell赋值运算符只是为了演示目的。实际上，这种情况下，由于默认的由编译器生成的运算符已经足以满足要求，本可以省去这里的赋值运算符；它只是对所有数据成员进行逐个成员的赋值。然而在某些情况下，默认赋值运算符的功能不足。第9章将讲述这些情况。

## 3. 显式默认或显式删除的赋值运算符

可显式地默认或删除编译器生成的赋值运算符，如下所示。

```
SpreadsheetCell& operator=(const SpreadsheetCell& rhs) = default;
```

或者

```
SpreadsheetCell& operator=(const SpreadsheetCell& rhs) = delete;
```

### 8.3.4 编译器生成的拷贝构造函数和拷贝赋值运算符

在 C++11 中，如果类具有用户声明的拷贝赋值运算符或析构函数，编译器将不会生成拷贝构造函数。如果仍然需要编译器生成的拷贝构造函数，则可以显式指定 default。

```
MyClass(const MyClass& src) = default;
```

同样，在 C++11 中，如果类具有用户声明的拷贝构造函数或析构函数，编译器也不会生成拷贝赋值运算符。如果在此类情况下仍然需要编译器生成的拷贝赋值运算符，可以显式指定 default：

```
MyClass& operator=(const MyClass& rhs) = default;
```

### 8.3.5 复制和赋值的区别

有时很难区分对象何时用拷贝构造函数初始化，何时用赋值运算符赋值。基本上，声明时会使用拷贝构造函数，赋值语句会使用赋值运算符。考虑下面的代码：

```
SpreadsheetCell myCell { 5 };
SpreadsheetCell anotherCell { myCell };
```

anotherCell 由拷贝构造函数创建。现在考虑这行代码：

```
SpreadsheetCell aThirdCell = myCell;
```

aThirdCell 也是由拷贝构造函数创建的，因为这条语句是一个声明。这行代码不会调用 operator=，这只是编写 SpreadsheetCell aThirdCell(myCell); 的另一种语法。不过，考虑以下代码：

```
anotherCell = myCell; // Calls operator= for anotherCell
```

此处，anotherCell 已经构建，因此编译器会调用 operator=。

#### 1. 按值返回对象

当函数或方法返回对象时，有时很难看出究竟执行了什么样的复制和赋值。例如，SpreadsheetCell::getString() 的实现如下所示：

```
string SpreadsheetCell::getString() const
{
    return doubleToString(m_value);
}
```

现在考虑下面的代码：

```
SpreadsheetCell myCell2 { 5 };
string s1;
s1 = myCell2.getString();
```

当 getString() 返回 mString 时，编译器实际上调用 string 拷贝构造函数，创建一个未命名的临时 string 对象。将结果赋给 s1 时，会调用 s1 的赋值运算符，将这个临时字符串作为参数。然后，这个临时的字符串对象被销毁。因此，这行简单的代码调用了拷贝构造函数和赋值运算符（针对两个不同的对象）。然而，编译器可实现（有时需要实现）返回值优化（Return Value Optimization, RVO），利用复制省略（copy elision）在返回值时优化掉成本高昂的拷贝构造函数。

了解上面的内容后，考虑下面的代码：

```
SpreadsheetCell myCell3 { 5 };
string s2 = myCell3.getString();
```

在此情况下，`getString()`返回时创建了一个临时的未命名 `string` 对象。但现在 `s2` 调用的是拷贝构造函数，而不是赋值运算符。

通过移动语义(move semantics)，编译器可使用移动构造函数而不是拷贝构造函数，从 `getString()` 返回该字符串，这样做效率更高。第 9 章将讨论移动语义。

如果忘记了这些事情发生的顺序，或忘记调用了哪个构造函数或运算符，只要在代码中临时添加辅助输出或者用调试器逐步调试代码，就能很容易地找到答案。

## 2. 拷贝构造函数和对象成员

还应注意构造函数中赋值和调用拷贝构造函数的不同之处。如果某个对象包含其他对象，编译器生成的拷贝构造函数会递归调用每个被包含对象的拷贝构造函数。当编写自己的拷贝构造函数时，可使用前面所示的构造函数初始化器提供相同的语义。如果在构造函数初始化器中省略某个数据成员，在执行构造函数体内的代码之前，编译器将对该成员执行默认的初始化(为对象调用默认构造函数)。这样，在执行构造函数体时，所有数据成员都已经初始化。

例如，可这样编写拷贝构造函数：

```
SpreadsheetCell::SpreadsheetCell(const SpreadsheetCell& src)
{
    m_value = src.m_value;
}
```

然而，在拷贝构造函数的函数体内对数据成员赋值时，使用的是赋值运算符而不是拷贝构造函数，因为它们已经初始化，就像前面讲述的那样。

如果编写如下拷贝构造函数，则使用拷贝构造函数初始化 `m_value`。

```
SpreadsheetCell::SpreadsheetCell(const SpreadsheetCell& src)
    : m_value { src.m_value }
{}
```

## 8.4 本章小结

本章讲述了 C++ 为面向对象编程提供的基本工具：类和对象。首先回顾编写类和使用对象的基本语法，包括访问控制。然后讲述对象的生命周期：何时构建、销毁和赋值，这些操作会调用哪些方法。本章包含构造函数的语法细节，包括构造函数初始化器和初始化列表构造函数，此外还介绍了拷贝赋值运算符的概念。本章还明确指出在什么情况下编译器会自动生成什么样的构造函数，并解释了默认构造函数不需要任何参数。

对于某些人来说，本章基本上只是回顾。对于另一些人来说，通过本章可更好地了解 C++ 中的面向对象编程世界。无论如何，现在你已经了解了对象和类，可阅读第 9 章，学习与类和对象有关的更多技巧。

## 8.5 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，然后再从网站上查看解决方案。

**练习 8-1** 实现一个 Person 类，将名字和姓氏存储为数据成员。添加一个接收两个参数(名字和姓氏)的构造函数，提供适当的获取器和设置器。编写一个小的 main() 函数，通过在栈和自由存储区中创建 Person 对象来测试你的实现。

**练习 8-2** 使用练习 8-1 中实现的一组成员函数，以下代码行无法编译：

```
Person phoneBook[3];
```

你能解释原因吗？修改你的 Person 类的实现使其可以成功编译。

**练习 8-3** 将以下方法添加到 Person 类的实现中：拷贝构造函数、赋值运算符和析构函数。在所有这些方法中，实现你认为必要的内容，另外，向控制台输出一行文本，以便跟踪它们的执行时间。修改 main() 函数以测试这些新方法。注意：严格来说，这些新方法对于这个 Person 类不是严格要求的，因为编译器生成的版本已经足够好了，但是这个练习是为了练习编写它们。

**练习 8-4** 从 Person 类中删除拷贝构造函数、赋值运算符和析构函数，因为编译器生成的默认版本正是这个简单类所需要的。接下来，添加一个新的数据成员来存储一个人的姓名首字母，并提供获取器和设置器。添加一个新的构造函数，它接收 3 个参数，一个名字和姓氏，以及一个人的首字母。修改原始的双参数构造函数，以自动生成给定名字和姓氏的首字母缩写，并将实际构造工作委托给新的三参数构造函数。



# 第9章

## 精通类和对象

### 本章内容

- 如何使类成为其他类的友元
- 如何在对象中使用动态内存分配
- “复制和交换”惯用方法
- 右值和右值引用的含义
- 移动语义如何提高性能
- “零规则”的含义
- 可使用的数据成员类型(static、const 和 reference)
- 可实现的方法类型(static、const 和 inline)
- 方法重载的细节
- 如何使用默认参数
- 如何使用嵌套类
- 什么是运算符重载
- 如何将类的接口与实现分离

第8章讨论了类和对象。现在应掌握其微妙之处，以充分利用类和对象。阅读本章，你将学会如何操纵并利用C++语言中最强大的特性，以编写安全、高效、有用的类。

本章的许多概念会出现在C++高级编程，特别是标准库中。让我们从友元的概念开始介绍吧。

### 9.1 友元

C++允许某个类将其他类、其他类的成员函数或非成员函数声明为友元(friend)，友元可以访问类的protected、private数据成员和方法。例如，假设有两个类 Foo 和 Bar。可将 Bar 类指定为 Foo 类的友元，如下所示：

```
class Foo
{
    friend class Bar;
    // ...
};
```

现在，Bar类的所有成员可访问Foo类的private、protected数据成员和方法。

也可将Bar类的一个特定方法作为友元。假设Bar类拥有一个processFoo(const Foo&)方法，下面的语法将使该方法成为Foo类的友元。

```
class Foo
{
    friend void Bar::processFoo(const Foo&);
    // ...
};
```

独立函数也可成为类的友元。例如，假设要编写一个函数，将Foo对象的所有数据打印到控制台。你可能希望将这个函数放在Foo类之外，因为打印并不是Foo类的核心功能，但该函数应当可以访问Foo对象的内部数据成员，以对其进行打印。下面是Foo类定义和printFoo()友元函数：

```
class Foo
{
    friend void printFoo(const Foo&);
    // ...
};
```

类中的friend声明用作函数的原型。不需要在别处编写原型(当然，如果你那样做，也无害处)。下面是函数定义：

```
void printFoo(const Foo& foo)
{
    // Print all data of foo to the console, including
    // private and protected data members.
}
```

你编写的函数与其他函数相似，只是可以用这个函数直接访问Foo类的private和protected数据成员。在函数定义中不需要重复使用friend关键字。

注意，类需要知道其他哪些类、方法或函数希望成为它的友元，类、方法或函数不能将自身声明为其他类的友元并访问这些类的非公有成员。

friend类和方法很容易被滥用，友元可以违反封装的原则，将类的内部暴露给其他类或函数。因此，只有在特定的情况下才应该使用它们，本章将穿插介绍一些用例。

## 9.2 对象中的动态内存分配

有时在程序实际运行前，并不知道需要多少内存。如第7章所述，解决这一问题的方法是根据程序执行的需要动态分配足够大的空间。类也不例外，编写类时，有时并不知道某个对象需要占用多少内存。在此情况下，应该为这个对象动态分配内存。对象中的动态内存分配提出了一些挑战，包括释放内存、处理对象复制和对象赋值。

### 9.2.1 Spreadsheet类

第8章介绍了SpreadsheetCell类，本章继续编写Spreadsheet类。与SpreadsheetCell类相似，Spreadsheet类将在本章逐步改进。因此，不同的尝试并非总是为了说明编写类的最佳方法。最初的Spreadsheet类只是一个SpreadsheetCell类型的二维数组，其中具有设置和获取Spreadsheet中特定位置单元格的方法。尽管大多数电子表格应用程序为了指定单元格，会在一个方向上使用字母，而在另

一个方向上使用数字，但此处的 `Spreadsheet` 类在两个方向上都使用数字。

`Spreadsheet` 类使用 `size_t` 类型，该类型在名为`<cstddef>`的 C 头文件中定义。正如第 1 章所说的，C 头文件不能保证是可导入的，最好包含它们。这些应该包含在所谓的全局模块部分。通过 `Spreadsheet.cppm` 文件的前几行，可完成此操作。

```
module;
#include <cstddef>
```

接下来，定义本模块的名字：

```
export module spreadsheet;
```

`Spreadsheet` 类需要访问 `SpreadsheetCell` 类，因此需要导入 `spreadsheet_cell` 模块。此外，为了保持 `spreadsheet` 模块用户对 `SpreadsheetCell` 类的可见性，将使用以下有趣的语法导入和导出 `spreadsheet_cell` 模块。

```
export import spreadsheet_cell;
```

最终，下面是 `Spreadsheet` 类的第一个定义：

```
export class Spreadsheet
{
    public:
        Spreadsheet(size_t width, size_t height);
        void setCellAt(size_t x, size_t y, const SpreadsheetCell& cell);
        SpreadsheetCell& getCellAt(size_t x, size_t y);
    private:
        bool inRange(size_t value, size_t upper) const;
        size_t m_width { 0 };
        size_t m_height { 0 };
        SpreadsheetCell** m_cells { nullptr };
};
```

### 注意：

`Spreadsheet` 类中，`m_cell` 数组使用的是普通指针。这种做法贯穿整个第 9 章，目的是说明因果关系，并解释如何在类中处理资源(例如动态内存分配)。在产品代码中，应该使用标准 C++ 容器，例如 `std::vector` 可极大地简化 `Spreadsheet` 类的实现，但目前还没有学习如何使用原始指针正确地处理动态内存。在现代 C++ 中，涉及所有权语义时，绝对不应使用原始指针，但可能在旧代码中遇到它，此时就需要知道如何处理它。

注意，`Spreadsheet` 类并没有包含一个 `SpreadsheetCell` 类型的标准二维数组，而是包含一个 `SpreadsheetCell**`。原因是不同 `Spreadsheet` 对象的维度可能不同，因此类的构造函数必须根据客户指定的宽度和高度动态分配二维数组。为动态分配二维数组，可编写如下代码。注意在 C++ 中与 Java 不同，不能只编写 `new SpreadsheetCell[m_width][m_height]`。

```
Spreadsheet::Spreadsheet(size_t width, size_t height)
    : m_width { width }, m_height { height }
{
    m_cells = new SpreadsheetCell*[m_width];
    for (size_t i { 0 }; i < m_width; i++) {
        m_cells[i] = new SpreadsheetCell[m_height];
    }
}
```

名为 s1 的 Spreadsheet 对象分配的内存如图 9-1 所示，宽度为 4，高度为 3。

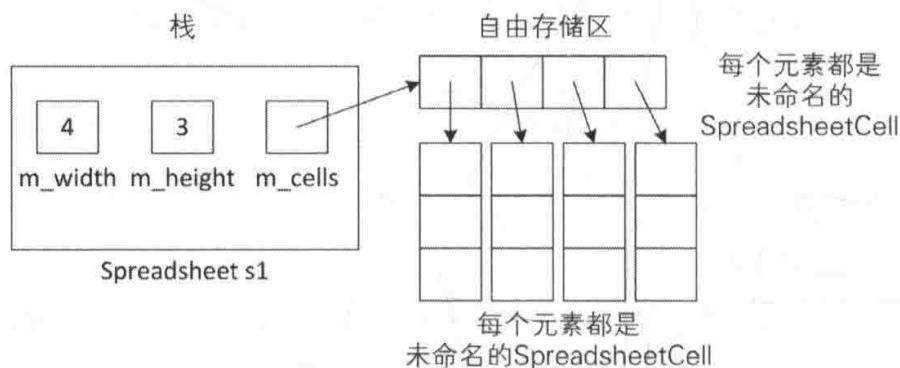


图 9-1 名为 s1 的 Spreadsheet 对象分配的内存

设置和获取方法的实现简洁明了：

```
void Spreadsheet::setCellAt(size_t x, size_t y, const SpreadsheetCell& cell)
{
    if (!inRange(x, m_width)) {
        throw out_of_range { format("{} must be less than {}", x, m_width) };
    }
    if (!inRange(y, m_height)) {
        throw out_of_range { format("{} must be less than {}", y, m_height) };
    }
    m_cells[x][y] = cell;
}

SpreadsheetCell& Spreadsheet::getCellAt(size_t x, size_t y)
{
    if (!inRange(x, m_width)) {
        throw out_of_range { format("{} must be less than {}", x, m_width) };
    }
    if (!inRange(y, m_height)) {
        throw out_of_range { format("{} must be less than {}", y, m_height) };
    }
    return m_cells[x][y];
}
```

注意这两个方法使用辅助方法 `inRange()` 检测电子表格中的 x 和 y 坐标是否有效。试图访问索引范围外的数组元素将导致程序故障。这个示例使用了异常，异常曾在第 1 章提到并将在第 14 章详细讲述。

如果查看 `setCellAt()` 和 `getCellAt()` 方法，会发现明显有一些代码是重复的。第 6 章介绍过，要不惜一切代价避免代码重复。按照这个原则，不再使用辅助方法 `inRange()`，而是为该类定义以下 `verifyCoordinate()` 方法。

```
void verifyCoordinate(size_t x, size_t y) const;
```

实现类检查给定的坐标，如果坐标无效，将抛出异常。

```
void Spreadsheet::verifyCoordinate(size_t x, size_t y) const
{
    if (x >= m_width) {
        throw out_of_range { format("{} must be less than {}", x, m_width) };
    }
    if (y >= m_height) {
```

```

        throw out_of_range { format("{} must be less than {}.", y, m_height) };
    }
}

```

现在，可简化 setCellAt() 和 getCellAt() 方法。

```

void Spreadsheet::setCellAt(size_t x, size_t y, const SpreadsheetCell& cell)
{
    verifyCoordinate(x, y);
    m_cells[x][y] = cell;
}

SpreadsheetCell& Spreadsheet::getCellAt(size_t x, size_t y)
{
    verifyCoordinate(x, y);
    return m_cells[x][y];
}

```

## 9.2.2 使用析构函数释放内存

如果不再需要动态分配的内存，就必须释放它们。如果在对象中动态分配了内存，就在析构函数中释放内存。当对象被销毁时，编译器确保调用析构函数。下面是带有析构函数的 Spreadsheet 类定义。

```

export class Spreadsheet
{
public:
    Spreadsheet(size_t width, size_t height);
    ~Spreadsheet();
    // Code omitted for brevity
};

```

析构函数与类(和构造函数)同名，名称的前面有一个波浪号(~)。析构函数没有参数，并且只能有一个析构函数。析构函数永远不应抛出异常，原因将在第 14 章解释。

下面是 Spreadsheet 类析构函数的实现：

```

Spreadsheet::~Spreadsheet()
{
    for (size_t i { 0 }; i < m_width; i++) {
        delete [] m_cells[i];
    }
    delete [] m_cells;
    m_cells = nullptr;
}

```

析构函数释放在构造函数中分配的内存。当然，并没有规则要求在析构函数中释放内存。在析构函数中可以编写任何代码，但最好让析构函数只负责释放内存或者清理其他资源。

## 9.2.3 处理复制和赋值

回顾第 8 章，如果没有自行编写拷贝构造函数或赋值运算符，C++ 将自动生成。编译器生成的方法递归调用对象数据成员的拷贝构造函数或赋值运算符。然而对于基本类型，如 int、double 和指针，只是提供表层(或按位)复制或赋值：只是将数据成员从源对象直接复制或赋值到目标对象。当在对象

内动态分配内存时，这样做会引发问题。例如，当 s1 被传递给函数 printSpreadsheet() 时，下面的代码复制了 s1 以初始化 s。

```
import spreadsheet;

void printSpreadsheet(Spreadsheet s) { /* Code omitted for brevity. */ }

int main()
{
    Spreadsheet s1 { 4, 3 };
    printSpreadsheet(s1);
}
```

Spreadsheet 包含一个指针变量：m\_cells。Spreadsheet 的浅复制向目标对象提供了一个 m\_cells 指针的副本，但没有复制底层数据。最终结果是 s 和 s1 都有一个指向同一数据的指针，如图 9-2 所示。

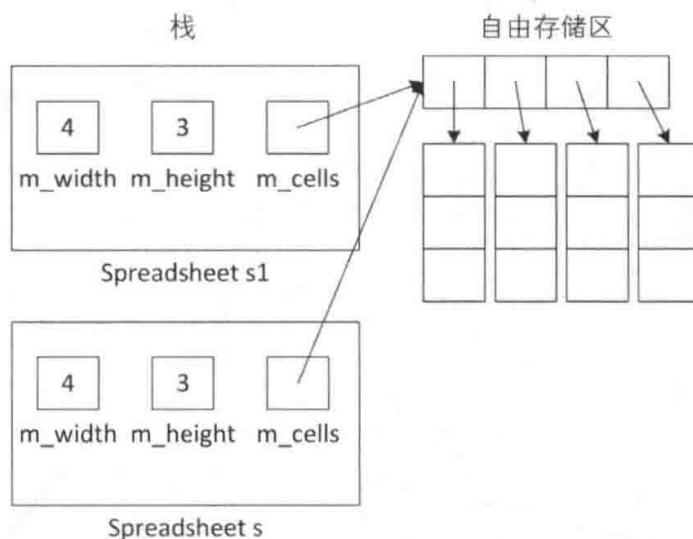


图 9-2 s 和 s1 都有一个指向同一数据的指针

如果 s 修改了 m\_cells 所指的内容，这一改动也会在 s1 中表现出来。更糟糕的是，当函数 printSpreadsheet() 退出时，会调用 s 的析构函数，释放 m\_cells 所指的内存。图 9-3 显示了这一状况。

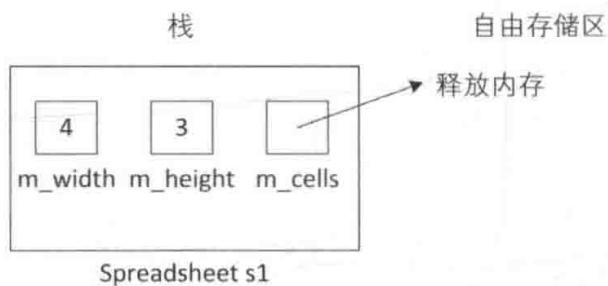


图 9-3 释放 m\_cells 所指的内存

现在 s1 拥有的指针所指的内存不再有效，这称为悬空指针(dangling pointer)。令人难以置信的是，当使用赋值时，情况会变得更糟。假定编写了下面的代码：

```
Spreadsheet s1 { 2, 2 }, s2 { 4, 3 };
s1 = s2;
```

在第一行之后，当构建两个对象时，内存的布局如图 9-4 所示。

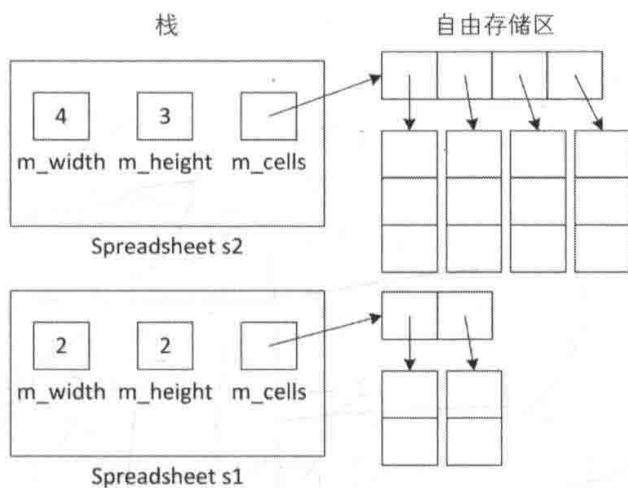


图 9-4 当构建两个对象时，内存的布局

当执行赋值语句后，内存的布局如图 9-5 所示。

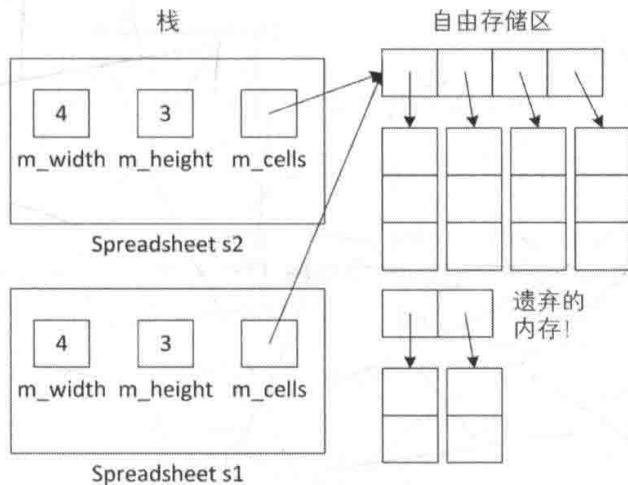


图 9-5 当执行赋值语句后，内存的布局

现在，不仅 `s1` 和 `s2` 中的 `m_cells` 指针指向同一内存，而且 `s1` 前面所指的内存被遗弃，这称为内存泄漏。

现在应该明白，拷贝构造函数和赋值运算符中必须进行深层复制。也就是说，它们不能只复制指针数据成员，必须复制指针所指向的实际数据。

可以看出，依赖 C++ 默认的拷贝构造函数或赋值运算符并不总是正确的。

### 警告：

无论什么时候，在类中动态分配内存后，都应该编写自己的拷贝构造函数和赋值运算符，以提供深层的内存复制。

## 1. Spreadsheet 类的拷贝构造函数

下面是 `Spreadsheet` 类中拷贝构造函数的声明：

```
export class Spreadsheet
{
public:
    Spreadsheet(const Spreadsheet& src);
    // Code omitted for brevity
```

```
};
```

下面是拷贝构造函数的定义：

```
Spreadsheet::Spreadsheet(const Spreadsheet& src)
    : Spreadsheet { src.m_width, src.m_height }
{
    for (size_t i { 0 }; i < m_width; i++) {
        for (size_t j { 0 }; j < m_height; j++) {
            m_cells[i][j] = src.m_cells[i][j];
        }
    }
}
```

注意使用了委托构造函数。把这个拷贝构造函数的初始化器首先委托给非拷贝构造函数，以分配适当的内存量。之后，拷贝构造函数体复制实际值。两个步骤合到一起，对 `m_cells` 动态分配的二维数组进行了深层复制。

在此不需要删除已有的 `m_cells`，因为这是一个拷贝构造函数，在 `this` 对象中尚不存在 `m_cells`。

## 2. Spreadsheet 类的赋值运算符

下面是包含赋值运算符的 `Spreadsheet` 类定义：

```
export class Spreadsheet
{
public:
    Spreadsheet& operator=(const Spreadsheet& rhs);
    // Code omitted for brevity
};
```

下面是一个不成熟的实现：

```
Spreadsheet& Spreadsheet::operator=(const Spreadsheet& rhs)
{
    // Check for self-assignment
    if (this == &rhs) {
        return *this;
    }

    // Free the old memory
    for (size_t i { 0 }; i < m_width; i++) {
        delete[] m_cells[i];
    }
    delete[] m_cells;
    m_cells = nullptr;

    // Allocate new memory
    m_width = rhs.m_width;
    m_height = rhs.m_height;

    m_cells = new SpreadsheetCell*[m_width];
    for (size_t i { 0 }; i < m_width; i++) {
        m_cells[i] = new SpreadsheetCell[m_height];
    }

    // Copy the data
    for (size_t i { 0 }; i < m_width; i++) {
```

```

        for (size_t j { 0 }; j < m_height; j++) {
            m_cells[i][j] = rhs.m_cells[i][j];
        }
    }

    return *this;
}

```

代码首先检查自我赋值，然后释放 this 对象的当前内存，此后分配新内存，最后复制各个元素。这个方法存有不少问题，有不少地方会出错。this 对象可能进入无效状态。

例如，假设成功释放了内存，合理设置了 m\_width 和 m\_height，但分配内存的循环抛出了异常。如果发生这种情况，将不再执行该方法的剩余部分，而是从该方法中退出。此时，Spreadsheet 实例受损，它的 m\_width 和 m\_height 数据成员声明了指定大小，但 m\_cells 数据成员不指向正确数量的内存。根本上说，该代码不能安全地处理异常！

我们需要一种全有或全无的机制：要么全部成功，要么该对象保持不变。为实现这样一个能安全处理异常的赋值运算符，要使用“复制和交换”惯用方法。可以给 Spreadsheet 类添加 swap()方法。但是，建议你提供一个非成员函数的 swap()版本，这样一来，各种标准库算法都可使用它。下面是包含赋值运算符、swap()方法以及非成员函数的 Spreadsheet 类的定义：

```

export class Spreadsheet
{
public:
    Spreadsheet& operator=(const Spreadsheet& rhs);
    void swap(Spreadsheet& other) noexcept;
    // Code omitted for brevity
};

export void swap(Spreadsheet& first, Spreadsheet& second) noexcept;

```

要实现能安全处理异常的“复制和交换”惯用方法，要求 swap()函数永不抛出异常，因此将其标记为 noexcept。

### 注意：

可使用 noexcept 关键字标记函数，指示不会抛出异常。例如：

```
void myNonThrowingFunction() noexcept { /* ... */ }
```

如果 noexcept 函数真的抛出了异常，程序将终止。有关 noexcept 的更多信息，请参阅第 14 章，但知道这些细节对本章的内容并不重要。

交换每个数据成员的 swap()方法的实现使用标准库中提供的<utility>中的 std::swap()工具函数，std::swap()可以高效地交换两个值。

```

void Spreadsheet::swap(Spreadsheet& other) noexcept
{
    std::swap(m_width, other.m_width);
    std::swap(m_height, other.m_height);
    std::swap(m_cells, other.m_cells);
}

```

非成员的 swap()函数只是简单地调用了 swap()方法：

```

void swap(Spreadsheet& first, Spreadsheet& second) noexcept
{
    first.swap(second);
}

```

现在就有了能安全处理异常的 swap() 函数，它可用来实现赋值运算符。

```
Spreadsheet& Spreadsheet::operator=(const Spreadsheet& rhs)
{
    Spreadsheet temp { rhs }; // Do all the work in a temporary instance
    swap(temp); // Commit the work with only non-throwing operations
    return *this;
}
```

该实现使用“复制和交换”惯用方法。首先，先创建一份右边的副本，名为 temp。然后用当前对象与这个副本交换。这个模式是实现赋值运算符的推荐方法，因为它保证强大的异常安全性。这意味着如果发生任何异常，当前的 Spreadsheet 对象保持不变。这通过 3 个阶段来实现：

- 第一阶段创建一个临时副本。这不修改当前 Spreadsheet 对象的状态，因此，如果在这个阶段发生异常，不会出现问题。
- 第二阶段使用 swap() 函数，将创建的临时副本与当前对象交换。swap() 永远不会抛出异常。
- 第三阶段销毁临时对象(由于发生了交换，现在包含原始对象)以清理内存。

如果不使用“复制和交换”惯用方法实现赋值运算符，那么为了提高效率，有时也为了确保正确性，赋值运算符中的第一行代码通常会检查自我赋值。例如：

```
Spreadsheet& Spreadsheet::operator=(const Spreadsheet& rhs)
{
    // Check for self-assignment
    if (this == &rhs) { return *this; }
    // ...
    return *this;
}
```

使用“复制和交换”惯用方法的情况下，就不再需要自我赋值的检查了。

#### 警告：

当实现赋值运算符时，使用“复制和交换”惯用方法，既可以避免代码重复，又能保证强大的异常安全性。

#### 注意：

“复制和交换”惯用方法不仅仅适用于赋值运算符。它可以用于任何需要多个步骤的操作，并且你希望将其转换为全有或全无操作。

### 3. 禁止赋值和按值传递

在类中动态分配内存时，如果只想禁止其他人复制对象或者为对象赋值，只需要显式地将 operator= 和拷贝构造函数标记为 delete。通过这种方法，当其他任何人按值传递对象时、从函数或方法返回对象时，或者为对象赋值时，编译器都会报错。下面的 Spreadsheet 类定义禁止赋值和按值传递：

```
export class Spreadsheet
{
public:
    Spreadsheet(size_t width, size_t height);
    Spreadsheet(const Spreadsheet& src) = delete;
```

```

~Spreadsheet();
Spreadsheet& operator=(const Spreadsheet& rhs) = delete;
// Code omitted for brevity
};

```

不需要提供`=delete`方法的实现，链接器永远不会查看它们，因为编译器不允许代码调用它们。当代码复制`Spreadsheet`对象或者对`Spreadsheet`对象赋值时，编译器将给出如下消息：

```
'Spreadsheet &Spreadsheet::operator =(const Spreadsheet &)': attempting to
reference a deleted function
```

## 9.2.4 使用移动语义处理移动

对象的移动语义(move semantics)需要实现移动构造函数(move constructor)和移动赋值运算符(move assignment operator)。如果源对象是操作结束后会被销毁的临时对象，或显式使用`std::move()`时，编译器就会使用这两个方法。移动将内存和其他资源的所有权从一个对象移动到另一个对象。这两个方法基本上只对成员变量进行浅复制(shallow copy)，然后转换已分配内存和其他资源的所有权，从而阻止悬空指针和内存泄漏。

移动构造函数和移动赋值运算符将数据成员从源对象移动到新对象，然后使源对象处于有效但不确定的状态。通常，源对象的数据成员被重置为空值，但这不是必需的。为了安全起见，不要使用任何已被移走的对象，因为这会触发未定义的行为。`std::unique_ptr` 和 `std::shared_ptr` 是例外情况。标准库明确规定，这些智能指针在移动时必须将其内部指针重置为`nullptr`，这使得从智能指针移动后可以安全地重用这些智能指针。

在实现移动语义前，需要学习右值(rvalue)和右值引用(rvalue reference)。

### 1. 右值引用

在 C++ 中，左值(lvalue)是可获取其地址的一个量，例如一个有名称的变量。由于经常出现在赋值语句的左边，因此将其称作左值。另外，所有不是左值的量都是右值(rvalue)，例如字面量、临时对象或临时值。通常右值位于赋值运算符的右边。例如，考虑下面的语句：

```
int a { 4 * 2 };
```

在这条语句中，`a` 是左值，它具有名称，它的地址为`&a`。右侧表达式`4 * 2` 的结果是右值。它是一个临时值，将在语句执行完毕时销毁。在本例中，将这个临时值的副本存储在变量`a` 中。

右值引用是一个对右值(rvalue)的引用。特别地，这是一个当右值是临时对象或使用`std::move()`显式移动时才适用的概念。右值引用的目的是在涉及右值时提供可选用的特定重载函数。通过右值引用，某些涉及复制大量值的操作可通过简单地复制指向这些值的指针来实现。

函数可将`&&`作为参数说明的一部分(例如`type&& name`)，以指定右值引用参数。通常，临时对象被当作`const type&`，但当函数重载使用了右值引用时，可以解析临时对象，用于该函数重载。下面的示例说明了这一点。代码首先定义了两个`handleMessage()`函数，一个接收左值引用，另一个接收右值引用。

```

void handleMessage(string& message) // lvalue reference parameter
{
    cout << format("handleMessage with lvalue reference: {}", message) << endl;
}

void handleMessage(string&& message) // rvalue reference parameter

```

```
{
    cout << format("handleMessage with rvalue reference: {}", message) << endl;
}
```

可使用命名变量作为实参调用 handleMessage() 函数：

```
string b { "World" };
handleMessage(a + b); // Calls handleMessage(string&& value)
```

由于 a 是一个命名变量，接收左值引用的 handleMessage() 函数被调用。handleMessage() 函数通过其引用参数所执行的任何更改都会更改 a。

还可用表达式作为实参来调用 handleMessage() 函数：

```
string b { "World" };
handleMessage(a + b); // Calls handleMessage(string&& value)
```

此时无法使用接收左值引用作为参数的 handleMessage() 函数，因为表达式 a+b 的结果是临时的，这不是一个左值。在此情况下，会调用右值引用版本。由于参数是一个临时值，handleMessage() 函数调用结束后，会丢失通过引用参数所做的任何更改。

字面量也可作为 handleMessage() 调用的参数，此时同样会调用右值引用版本，因为字面量不能作为左值(但字面量可作为 const 引用形参的对应实参传递)。

```
handleMessage("Hello World"); // Calls handleMessage(string&& value)
```

如果删除接收左值引用的 handleMessage() 函数，使用有名称的变量调用 handleMessage() 函数(例如 handleMessage(b))，会导致编译错误，因为右值引用参数(string&&) 永远不会与左值(b) 绑定。可使用 std::move() 强迫编译器调用 handleMessage() 函数的右值引用版本。move() 做的唯一的事就是将左值转换为右值，也就是说，它不做任何实际的移动。但是，通过返回右值引用，它可以使编译器找到接受右值引用的 handleMessage() 重载，然后执行移动。下面是使用 move() 的示例：

```
handleMessage(std::move(b)); // Calls handleMessage(string&& value)
```

重申一次，有名称的变量是左值。因此，在 handleMessage() 函数中，右值引用参数 message 本身是一个左值，原因是它具有名称！如果希望将这个右值引用参数作为右值传递给另一个函数，则需要使用 std::move()，将左值转换为右值。例如，假设要添加以下函数，使用右值引用参数：

```
void helper(std::string&& message) { }
```

如果按如下方式调用，则无法编译：

```
void handleMessage(std::string&& message) { helper(message); }
```

helper() 函数需要右值引用，而 handleMessage() 函数传递 message，message 具有名称，因此是左值，导致编译错误。正确的方式是使用 std::move()：

```
void handleMessage(std::string&& message) { helper(std::move(message)); }
```

### 警告：

有名称的右值引用，如右值引用参数，本身就是左值，因为它具有名称！

右值引用并不局限于函数的参数。可以声明右值引用类型的变量，并对其进行赋值，尽管这一用法并不常见。考虑下面的代码，在 C++ 中这是不合法的：

```
int& i { 2 }; // Invalid: reference to a constant
```

```
int a { 2 }, b { 3 };
int& j { a + b };      // Invalid: reference to a temporary
```

使用右值引用后，下面的代码完全合法：

```
int&& i { 2 };
int a { 2 }, b { 3 };
int&& j { a + b };
```

前面示例中单独使用右值引用的情况很少见。

### 注意：

如果将临时值赋值给右值引用，则只要右值引用在作用域内，临时值的生命周期就会延长。

## 2. 实现移动语义

移动语义是通过右值引用实现的。为了对类增加移动语义，需要实现移动构造函数和移动赋值运算符。移动构造函数和移动赋值运算符应使用 `noexcept` 限定符标记，这告诉编译器，它们不会抛出任何异常。这对于与标准库兼容非常重要，因为如果实现了移动语义，标准库容器会移动存储的对象，且确保不抛出异常。下面的 `Spreadsheet` 类定义包含一个移动构造函数和一个移动赋值运算符。也引入了两个辅助方法 `cleanup()` 和 `moveFrom()`。前者在析构函数和移动赋值运算符中调用。后者用于把成员变量从源对象移动到目标对象，接着重置源对象。

```
export class Spreadsheet
{
public:
    Spreadsheet(Spreadsheet&& src) noexcept;                                // Move constructor
    Spreadsheet& operator=(Spreadsheet&& rhs) noexcept;                      // Move assign
    // Remaining code omitted for brevity
private:
    void cleanup() noexcept;
    void moveFrom(Spreadsheet& src) noexcept;
    // Remaining code omitted for brevity
};
```

实现代码如下所示：

```
void Spreadsheet::cleanup() noexcept
{
    for (size_t i { 0 }; i < m_width; i++) {
        delete[] m_cells[i];
    }
    delete[] m_cells;
    m_cells = nullptr;
    m_width = m_height = 0;
}

void Spreadsheet::moveFrom(Spreadsheet& src) noexcept
{
    // Shallow copy of data
    m_width = src.m_width;
    m_height = src.m_height;
    m_cells = src.m_cells;

    // Reset the source object, because ownership has been moved!
    src.m_width = 0;
```

```

src.m_height = 0;
src.m_cells = nullptr;
}

// Move constructor
Spreadsheet::Spreadsheet(Spreadsheet&& src) noexcept
{
    moveFrom(src);
}

// Move assignment operator
Spreadsheet& Spreadsheet::operator=(Spreadsheet&& rhs) noexcept
{
    // Check for self-assignment
    if (this == &rhs) {
        return *this;
    }

    // Free the old memory and move ownership
    cleanup();
    moveFrom(rhs);
    return *this;
}

```

移动构造函数和移动赋值运算符都将 `m_cells` 的内存所有权从源对象移动到新对象，它们将源对象的 `m_cells` 指针设置为空指针，将源对象的 `m_width` 和 `m_height` 设置为 0，以防源对象的析构函数释放这块内存，因为新的对象现在拥有了这块内存。

很明显，只有你知道源对象不会再被使用时，移动语义才有用。

注意，此实现在移动赋值运算符中包含一个自我赋值检查。取决于你的类以及将类的一个实例移动到另一个实例的方法，此自我赋值检查可能并不总是必要的。但是，我总是将其包括在内，以确保以下代码在运行时不会导致崩溃。

```
sheet1 = std::move(sheet1);
```

例如，就像普通的构造函数或拷贝赋值运算符一样，可显式将移动构造函数和移动赋值运算符设置为默认或将其实删除，如第 8 章所述。

仅当类没有用户声明的拷贝构造函数、拷贝赋值运算符、移动赋值运算符或析构函数时，编译器才会为类自动生成默认的移动构造函数。仅当类没有用户声明的拷贝构造函数、移动构造函数、拷贝赋值运算符或析构函数时，才会为类生成默认的移动赋值运算符。

### 注意:

当你声明了一个或多个特殊成员函数(析构函数、拷贝构造函数、移动构造函数、拷贝赋值运算符和移动赋值运算符)时，通常需要声明所有这些函数，这称为“5 规则”(Rule of Five)。可以为它们提供显式实现，也可以显式默认(`=default`)或删除(`=delete`)它们。

### 使用 `std::exchange`

定义在`<utility>`中的 `std::exchange()`，可以用一个新的值替换原来的值，并返回原来的值。例如：

```

int a { 11 };
int b { 22 };
cout << format("Before exchange(): a = {}, b = {}", a, b) << endl;
int returnedValue { exchange(a, b) };

```

```
cout << format("After exchange(): a = {}, b = {}", a, b) << endl;
cout << format("exchange() returned: {}", returnedValue) << endl;
```

输出如下：

```
Before exchange(): a = 11, b = 22
After exchange(): a = 22, b = 22
exchange() returned: 11
```

在实现移动赋值运算符时，`exchange()`十分有用。移动赋值运算符需要将数据从源对象移动到目标对象，之后源对象中的数据通常为空。在上一节中，这是按如下方式完成的：

```
void Spreadsheet::moveFrom(Spreadsheet& src) noexcept
{
    // Shallow copy of data
    m_width = src.m_width;
    m_height = src.m_height;
    m_cells = src.m_cells;

    // Reset the source object, because ownership has been moved!
    src.m_width = 0;
    src.m_height = 0;
    src.m_cells = nullptr;
}
```

此方法从源对象复制数据成员 `m_width`、`m_height` 和 `m_cells`，然后将其设置为 0 或 `nullptr`，因为所有权已被移动。使用 `exchange()` 可以更简洁地编写，如下所示。

```
void Spreadsheet::moveFrom(Spreadsheet& src) noexcept
{
    m_width = exchange(src.m_width, 0);
    m_height = exchange(src.m_height, 0);
    m_cells = exchange(src.m_cells, nullptr);
}
```

### 移动对象数据成员

`moveFrom()` 方法对 3 个数据成员直接赋值，因为这些成员都是基本类型。如果对象还将其他对象作为数据成员，则应当使用 `std::move()` 移动这些对象。假设 `Spreadsheet` 类有一个名为 `m_name` 的 `std::string` 数据成员。接着采用以下方式实现 `moveFrom()` 方法：

```
void Spreadsheet::moveFrom(Spreadsheet& src) noexcept
{
    // Move object data members
    m_name = std::move(src.m_name);

    // Move primitives:
    m_width = exchange(src.m_width, 0);
    m_height = exchange(src.m_height, 0);
    m_cells = exchange(src.m_cells, nullptr);
}
```

### 用交换方式实现移动构造函数和移动赋值运算符

前面的移动构造函数和移动赋值运算符的实现都使用了 `moveFrom()` 辅助方法，该辅助方法通过执行浅复制来移动所有数据成员。在此实现中，如果给 `Spreadsheet` 类添加新的数据成员，则必须修

改 swap() 函数和 moveFrom() 方法。如果忘了更新其中的一个，则会引入 bug。为避免此类 bug，可使用 swap() 函数，编写移动构造函数和移动赋值运算符。

首先删除 cleanup() 和 moveFrom() 辅助方法，将 cleanup() 方法中的代码移入析构函数。此后，可按如下方式实现移动构造函数和移动赋值运算符。

```
Spreadsheet::Spreadsheet(Spreadsheet&& src) noexcept
{
    swap(*this, src);
}

Spreadsheet& Spreadsheet::operator=(Spreadsheet&& rhs) noexcept
{
    swap(*this, rhs);
    return *this;
}
```

移动构造函数只是简单地将默认构造的 \*this 与给定的源对象进行交换。同样，移动赋值运算符将 \*this 与给定的 rhs 对象进行交换。

#### 注意：

用 swap() 实现移动构造函数和移动赋值运算符所需的代码更少。当加入新的数据成员时出现 bug 的可能性也更低，因为你只需要升级 swap() 函数，使其包括新的数据成员。

### 3. 测试 Spreadsheet 移动操作

可使用如下代码测试 Spreadsheet 移动构造函数和移动赋值运算符：

```
Spreadsheet createObject()
{
    return Spreadsheet { 3, 2 };
}

int main()
{
    vector<Spreadsheet> vec;
    for (size_t i { 0 }; i < 2; ++i) {
        cout << "Iteration " << i << endl;
        vec.push_back(Spreadsheet { 100, 100 });
        cout << endl;
    }
    Spreadsheet s { 2, 3 };
    s = createObject();

    Spreadsheet s2 { 5, 6 };
    s2 = s;
}
```

第 1 章介绍了 vector。vector 的大小会动态增长以容纳新对象。为此，可分配较大的内存块，然后将对象从旧的 vector 复制到(或移动到)较大的、新的 vector。如果编译器发现了 noexcept 的移动构造函数，就会移动(而非复制)对象。由于移动了对象，因此不需要深度复制，从而提高了效率。

对于 Spreadsheet 类的所有构造函数和赋值运算符添加输出语句后，前面测试程序的输出如下所示。这个输出结果和下面的讨论基于 Microsoft Visual C++ 2019 编译器，代码采用发行版本。C++ 标准未指定 vector 的初始容量和增长策略，因此，在不同的编译器上，输出是不同的。

Iteration 0	
Normal constructor	(1)
Move constructor	(2)
Iteration 1	
Normal constructor	(3)
Move constructor	(4)
Move constructor	(5)
Normal constructor	(6)
Normal constructor	(7)
Move assignment operator	(8)
Normal constructor	(9)
Copy assignment operator	(10)
Normal constructor	(11)
Copy constructor	(12)

在该循环的第一个迭代中，vector 仍然为空。分析循环中的下一行：

```
vec.push_back(Spreadsheet { 100, 100 });
```

这一行调用一个普通构造函数(1)，创建了一个新的 Spreadsheet 对象。vector 将改变自身的大小，为容纳新对象提供空间。随后调用移动构造函数(2)，把创建的 Spreadsheet 对象移动到 vector 中。

循环第二次迭代时，会使用普通构造函数(3)创建第二个 Spreadsheet 对象。此时，vector 只能保存一个元素，因此再次调整大小，为第二个对象提供空间。在调整 vector 大小时，前面添加的元素需要从旧的 vector 移动到新的、较大的 vector，因此会为前面添加的每个元素调用移动构造函数。目前 vector 中只有一个元素，所以移动构造函数(4)会被调用一次。然后，新的 Spreadsheet 对象使用移动构造函数(5)移动到 vector。

接着使用普通构造函数(6)创建了一个 Spreadsheet 对象。createObject()函数使用普通构造函数(7)创建了一个临时 Spreadsheet 对象，这个对象此后由函数返回，然后赋给变量 s。由于 createObject()返回的临时对象在赋值结束后会终止并退出，编译器将调用移动赋值运算符(8)而不是普通的拷贝赋值运算符。使用普通的构造函数(9)创建另一个 Spreadsheet 对象 s2。赋值语句 s2=s 将调用拷贝赋值运算符(10)，因为右边的对象不是临时对象，而是一个有名字的对象。这个拷贝赋值运算符使用了“复制和交换”惯用方法，创建一个临时副本，这将触发对拷贝构造函数的调用，此时，首先委托给普通构造函数(11 和 12)。

如果 Spreadsheet 类未实现移动语义，对移动构造函数和移动赋值运算符的所有调用将被替换为对拷贝构造函数和拷贝赋值运算符的调用。在前面的示例中，循环中的 Spreadsheet 对象拥有 10 000( $100 \times 100$ )个元素。Spreadsheet 移动构造函数和移动赋值运算符的实现不需要任何内存分配，而拷贝构造函数和拷贝赋值运算符各需要 101 次分配。因此，某些情况下使用移动语义可以大幅提高性能。

#### 4. 使用移动语义实现交换函数

考虑交换两个对象的 swap() 函数模板，这是另一个使用移动语义提高性能的示例。下面的 swapCopy() 实现没有使用移动语义：

```
template <typename T>
void swapCopy(T& a, T& b)
{
    T temp { a };
```

```

    a = b;
    b = temp;
}

```

这一实现首先将 `a` 复制到 `temp`, 其次将 `b` 复制到 `a`, 最后将 `temp` 复制到 `b`。如果类型 `T` 的复制开销很大, 这个交换实现将严重影响性能。使用移动语义, `swap()` 函数可避免所有复制。

```

template <typename T>
void swapMove(T& a, T& b)
{
    T temp { std::move(a) };
    a = std::move(b);
    b = std::move(temp);
}

```

这正是标准库的 `std::swap()` 的实现方式。

## 5. 在返回语句中使用 `std::move()`

对于 `return object;` 形式的语句, 如果 `object` 是局部变量、函数的参数或临时值, 则它们被视为右值表达式, 并触发返回值优化(RVO)。此外, 若 `object` 是一个局部变量, 则会启动命名返回值优化(NRVO)。RVO 和 NRVO 都是复制省略的形式, 使得从函数返回对象非常有效。使用复制省略, 编译器可以避免复制和移动函数返回的对象。这导致了所谓的零拷贝值传递语义。

现在, 使用 `std::move()` 返回对象时会发生什么? 不管是写 `return object;`, 还是 `return std::move(object);`, 在这两种情况下, 编译器都将其视为右值表达式。

但是, 通过使用 `std::move()`, 编译器无法再应用 RVO 或 NRVO, 因为这只适用于形式为 `return object;` 的语句。由于 RVO 和 NRVO 不再适用, 编译器的下一个选择是在对象支持的情况下使用移动语义。如果不支持, 则使用复制语义, 这会对性能产生很大影响! 因此, 请记住以下规则。

### 注意:

当从函数中返回一个局部变量或参数时, 只要写 `return object;` 就可以了, 不要使用 `std::move()`。

请记住, (N)RVO 仅适用于局部变量或函数参数。因此, 返回对象的数据成员不会触发(N)RVO。此外, 请注意以下表达式:

```
return condition ? object1 : object2;
```

这不是 `return object;` 的形式, 所以编译器不会应用(N)RVO, 而是使用拷贝构造函数返回 `object1` 或 `object2`。你可以重写返回语句, 使编译器可以使用(N)RVO:

```

if (condition) {
    return object1;
} else {
    return object2;
}

```

如果确实想使用条件运算符, 可以编写以下语句。但请记住, 这不会触发(N)RVO 并强制使用移动或复制语义:

```
return condition ? std::move(object1) : std::move(object2);
```

## 6. 向函数传递参数的最佳方法

到目前为止，建议对非基本类型的函数参数使用 `const` 引用参数，以避免对传递给函数的实参进行不必要的昂贵复制。但是，如果混合使用右值，情况会略有变化。假设一个函数复制了作为其参数之一传递的实参(这种情况经常在类方法中出现)。下面是一个简单的例子：

```
class DataHolder
{
public:
    void setData(const std::vector<int>& data) { m_data = data; }
private:
    std::vector<int> m_data;
};
```

`setData()`方法生成一份传入数据的副本。既然你对右值和右值引用都很熟悉，那么你可能需要添加一个重载来优化 `setData()`方法，以避免右值情况下的任何复制。例如：

```
class DataHolder
{
public:
    void setData(const std::vector<int>& data) { m_data = data; }
    void setData(std::vector<int>&& data) { m_data = std::move(data); }
private:
    std::vector<int> m_data;
};
```

当以临时值调用 `setData()`时，不会产生任何复制，数据会被移动。

以下代码段中的代码触发对 `setData()`的 `const` 引用重载版本的调用，从而生成数据的副本。

```
DataHolder wrapper;
std::vector myData { 11, 22, 33 };
wrapper.setData(myData);
```

另一方面，下面的代码段使用临时变量调用 `setData()`，这会触发对 `setData()`的右值引用重载版本的调用。随后将移动数据，而不是复制数据。

```
wrapper.setData({ 22, 33, 44 });
```

但是，这种为左值和右值优化 `setData()`的方法需要实现两个重载。幸运的是，对于单个方法来说有一种更好的方法。是的，值传递！到目前为止，建议始终使用 `const` 引用参数来传递对象，以避免任何不必要的复制，但现在我们建议使用值传递。让我们澄清一下，对于不被复制的参数，通过 `const` 引用传递仍然是应使用的方法，值传递建议仅适用于函数无论如何都要复制的参数。在这种情况下，通过使用值传递语义，代码对于左值和右值都是最优的。如果传入一个左值，它只复制一次，就像 `const` 引用参数一样。如果传入一个右值，则不会进行复制，就像右值引用参数一样。让我们看一些代码：

```
class DataHolder
{
public:
    void setData(std::vector<int> data) { m_data = std::move(data); }
private:
    std::vector<int> m_data;
};
```

如果将左值传递给 `setData()`，则会将其复制到 `data` 参数中，然后移动到 `m_data`。如果将右值传

递交给 `setData()`，则会将其移动到 `data` 参数中，然后再次移动到 `m_data` 中。

#### 注意：

对于函数本身将复制的参数，更倾向于值传递，但仅当该参数属于支持移动语义的类型时。否则，请使用 `const` 引用参数。

### 9.2.5 零规则

前面介绍过 5 规则(rule of five)。前面的讨论一直在解释如何编写以下 5 个特殊的成员函数：析构函数、拷贝构造函数、移动构造函数、拷贝赋值运算符和移动赋值运算符。但在现代 C++ 中，你需要接受零规则(rule of zero)。

“零规则”指出，在设计类时，应当使其不需要上述 5 个特殊成员函数。如何做到这一点？基本上，应当避免拥有任何旧式的、动态分配的内存。而改用现代结构，如标准库容器。例如，在 `Spreadsheet` 类中，用 `vector<vector<SpreadsheetCell>>` 替代 `SpreadsheetCell**` 数据成员。`vector` 自动处理内存，因此不需要上述 5 个特殊成员函数。

#### 警告：

在现代 C++ 中，要应用零规则！

5 规则应限于自定义资源获取即初始化(Resource Acquisition Is Initialization, RAII)类。RAII 类获取资源的所有权，并在正确的时间处理其释放。例如，`vector` 和 `unique_ptr` 使用了这种设计技术，并将在第 32 章进一步讨论。

## 9.3 与方法有关的更多内容

C++ 为方法提供了许多选择，本节详细介绍这些棘手的细节。

### 9.3.1 static 方法

与数据成员类似，方法有时会应用于全部类对象而不是单个对象。可以编写 `static` 方法和数据成员。以第 8 章的 `SpreadsheetCell` 类为例，这个类有两个辅助方法：`string.ToDouble()` 和 `double.ToString()`。这两个方法没有访问特定对象的信息，因此可以是 `static` 的。下面的类定义将这些方法设置为 `static`：

```
export class SpreadsheetCell
{
    // Omitted for brevity
private:
    static std::string doubleToString(double value);
    static double stringToDouble(std::string_view value);
    // Omitted for brevity
};
```

这两个方法的实现与前面的实现相同，在方法定义前不需要重复 `static` 关键字。然而，注意 `static` 方法不属于特定对象，因此没有 `this` 指针。当用某个特定对象调用 `static` 方法时，`static` 方法不会访问这个对象的非 `static` 数据成员。实际上，`static` 方法就像普通函数，唯一区别在于它可以访问类的 `private` 和 `protected` 的 `static` 成员。如果同一类型的其他对象对于 `static` 方法可见(例如传递了对象的指针或引用作为参数)，那么 `static` 方法也可访问其他对象的 `private` 和 `protected` 的非 `static` 数据成员。

类中的任何方法都可像调用普通函数那样调用静态方法，因此 SpreadsheetCell 类中所有方法的实现都没有改变。

如果要在类的外面调用静态方法，需要用类名和作用域解析运算符来限定方法的名称(就像静态数据成员那样)，静态方法的访问控制与普通方法一样。例如，类 Foo 中有一个名为 bar() 的 public 的 static 方法，此时，可在任意位置这样调用这 bar()：

```
Foo::bar();
```

### 9.3.2 const 方法

const 对象的值不能改变。如果使用 const 对象、const 对象的引用和指向 const 对象的指针，编译器将不允许调用对象的任何方法，除非这些方法保证不改变任何数据成员。为了保证方法不改变数据成员，可以用 const 关键字标记方法本身。在 SpreadsheetCell 类的开发过程中，第 8 章已经完成了这项工作。为了使你回想起来，下面的 SpreadsheetCell 类包含了用 const 标记的不改变任何数据成员的方法。

```
export class SpreadsheetCell
{
    public:
        double getValue() const;
        std::string getString() const;
        // Omitted for brevity
};
```

const 说明符是方法原型的一部分，必须放在方法的定义中。

```
double SpreadsheetCell::getValue() const
{
    return m_value;
}

std::string SpreadsheetCell::getString() const
{
    return doubleToString(m_value);
}
```

将方法标记为 const，就是与客户代码立下了契约，承诺不会在方法内改变对象内部的值。如果将实际上修改了数据成员的方法声明为 const，编译器将会报错。const 的工作原理是将方法内用到的数据成员都标记为 const 引用，因此如果试图修改数据成员，编译器会报错。

不能将 static 方法声明为 const，因为这是多余的。静态方法没有类的实例，因此不可能改变内部的值。

非 const 对象可调用 const 方法和非 const 方法。然而，const 对象只能调用 const 方法，下面是一些示例：

```
SpreadsheetCell myCell { 5 };
cout << myCell.getValue() << endl;           // OK
myCell.setString("6");                         // OK

const SpreadsheetCell& myCellConstRef { myCell };
cout << myCellConstRef.getValue() << endl;     // OK
myCellConstRef.setString("6");                 // Compilation Error!
```

应该养成习惯，将不修改对象的所有方法声明为 `const`，这样就可在程序中使用 `const` 对象的引用。注意 `const` 对象也会被销毁，它们的析构函数也会被调用，因此不应该将析构函数标记为 `const`。

### mutable 数据成员

有时编写的方法“逻辑上”是 `const` 方法，但是碰巧改变了对象的数据成员。这个改动对于用户可见的数据没有任何影响，但严格来说确实做了改动，因此编译器不允许将这个方法声明为 `const`。例如，假定电子表格应用程序要获取数据的读取频率。完成这个任务的笨拙办法是在 `SpreadsheetCell` 类中加入一个计数器，计算 `getValue()` 和 `getString()` 调用的次数。遗憾的是，这样做使编译器认为这些方法是非 `const` 的，这并非你的本意。解决方法是将计数器变量设置为 `mutable`，告诉编译器在 `const()` 方法中允许改变这个值。下面是新的 `SpreadsheetCell` 类定义：

```
export class SpreadsheetCell
{
    // Omitted for brevity
private:
    double m_value { 0 };
    mutable size_t m_numAccesses { 0 };
};
```

下面是 `getValue()` 和 `getString()` 的定义：

```
double SpreadsheetCell::getValue() const
{
    m_numAccesses++;
    return m_value;
}

std::string SpreadsheetCell::getString() const
{
    m_numAccesses++;
    return doubleToString(m_value);
}
```

### 9.3.3 方法重载

在类中可编写多个构造函数，所有这些构造函数的名称都相同。这些构造函数只是参数数量或类型不同。在 C++ 中，可对任何方法或函数做同样的事情。具体来讲，可重载函数或方法，具体做法是将函数或方法的名称用于多个函数，但是参数的类型或数目不同。例如在 `SpreadsheetCell` 类中，可将 `setString()` 和 `setValue()` 全部重命名为 `set()`。类定义如下所示：

```
export class SpreadsheetCell
{
public:
    void set(double value);
    void set(std::string_view value);
    // Omitted for brevity
};
```

`set()` 方法的实现保持不变。当编写调用 `set()` 方法的代码时，编译器根据传递的参数判断调用哪个实例：如果传递了 `string_view`，则编译器调用 `string_view` 实例；如果传递了 `double`，则编译器调用 `double` 实例。这称为重载解析（overload resolution）。

对 `getValue()` 和 `getString()` 执行同样的操作：将它们重命名为 `get()`。然而，这样的代码将无法编译。

C++不允许仅根据方法的返回类型重载方法名称，因为在许多情况下，编译器不可能判断调用哪个方法实例。例如，如果任何地方都没有使用方法的返回值，编译器将无从判断要使用哪个方法实例。

### 1. 基于 const 的重载

还要注意，可根据 `const` 重载方法。也就是说，可以编写两个名称相同、参数也相同的方法，其中一个 `const`，另一个不是。如果是 `const` 对象，就调用 `const` 方法；如果是非 `const` 对象，就调用非 `const` 方法。

通常情况下，`const` 版本和非 `const` 版本的实现是一样的。为避免代码重复，可使用 Scott Meyer 的 `const_cast()` 模式。例如，`Spreadsheet` 类有一个 `getCellAt()` 方法，该方法返回 `SpreadsheetCell` 的非 `const` 引用。可添加 `const` 重载版本，它返回 `SpreadsheetCell` 的 `const` 引用。

```
export class Spreadsheet
{
public:
    SpreadsheetCell& getCellAt(size_t x, size_t y);
    const SpreadsheetCell& getCellAt(size_t x, size_t y) const;
    // Code omitted for brevity.
};
```

对于 Scott Meyer 的 `const_cast()` 模式，你可像往常一样实现 `const` 版本，此后通过适当转换，传递对 `const` 版本的调用，以实现非 `const` 版本。如下所示：

```
const SpreadsheetCell& Spreadsheet::getCellAt(size_t x, size_t y) const
{
    verifyCoordinate(x, y);
    return m_cells[x][y];
}

SpreadsheetCell& Spreadsheet::getCellAt(size_t x, size_t y)
{
    return const_cast<SpreadsheetCell&>(as_const(*this).getCellAt(x, y));
}
```

基本上，你使用 `std::as_const()`（在 `<utility>` 中定义）将 `*this` 转换为 `const Spreadsheet&`，调用 `getCellAt()` 的 `const` 版本，它返回 `const SpreadsheetCell&`，然后使用 `const_cast()`，将其转换为非 `const` 的 `SpreadsheetCell&`。

有了这两个重载的 `getCellAt()`，现在可在 `const` 和非 `const` 的 `Spreadsheet` 对象上调用 `getCellAt()`。

```
Spreadsheet sheet1 { 5, 6 };
SpreadsheetCell& cell1 { sheet1.getCellAt(1, 1) };

const Spreadsheet sheet2 { 5, 6 };
const SpreadsheetCell& cell2 { sheet2.getCellAt(1, 1) };
```

这里，`getCellAt()` 的 `const` 版本做的事情不多，因此使用 `const_cast()` 模式的优势不明显。但可以想一下，如果 `getCellAt()` 的 `const` 版本能做更多工作，那么通过将非 `const` 版本传递给 `const` 版本，可省去很多代码。

### 2. 显式删除重载

重载方法可被显式删除，可以用这种方法禁止调用具有特定参数的成员函数。例如，

SpreadsheetCell 类有一个方法 setValue(double)，可按如下方式调用：

```
SpreadsheetCell cell;
cell.setValue(1.23);
cell.setValue(123);
```

在第三行，编译器将整型值(123)转换为 double，然后调用 setValue(double)。如果出于某些原因，你不希望以整型调用 setValue()，可以显式删除 setValue()的整型重载版本。

```
export class SpreadsheetCell
{
    public:
        void setValue(double value);
        void setValue(int) = delete;
};
```

通过这一改动，以整型为参数调用 setValue()时，编译器会给出错误提示。

### 3. 引用限定方法

可以对类的非临时和临时实例调用普通类方法。假设有以下类：

```
class TextHolder
{
public:
    TextHolder(string text) : m_text { move(text) } {}
    const string& getText() const { return m_text; }
private:
    string m_text;
};
```

毫无疑问，可以在 TextHolder 的非临时实例上调用 getText()方法。例如：

```
TextHolder textHolder { "Hello world!" };
cout << textHolder.getText() << endl;
```

然而，getText()也可被临时实例调用。

```
cout << TextHolder{ "Hello world!" }.getText() << endl;
cout << move(textHolder).getText() << endl;
```

可以显式指定能够调用某个方法的实例类型，无论是临时实例还是非临时实例。这是通过向方法添加一个所谓的引用限定符(ref-qualifier)来实现的。如果只应在非临时实例上调用方法，则在方法头之后添加一个&限定符。类似地，如果只应在临时实例上调用方法，则要添加一个&&限定符。

下面修改的 TextHolder 类通过将对返回 m\_text 的 const 引用来实现&限定的 getText()。另一方面，&&限定的 getText()返回对 m\_text 的右值引用，以便可以将 m\_text 移出 TextHolder。例如，如果你希望从临时 TextHolder 实例检索文本，则这可能更有效。

```
class TextHolder
{
public:
    TextHolder(string text) : m_text { move(text) } {}
    const string& getText() const & { return m_text; }
    string&& getText() && { return move(m_text); }
private:
    string m_text;
};
```

假设你有以下调用：

```
TextHolder textHolder { "Hello world!" };
cout << textHolder.getText() << endl;
cout << TextHolder{ "Hello world!" }.getText() << endl;
cout << move(textHolder).getText() << endl;
```

第一个对 `getText()` 的调用使用了`&`限定符重载，第二个和第三个使用了`&&`限定符重载。

### 9.3.4 内联方法

C++提供了这样一种能力：可以建议函数或方法的调用不在生成的代码中实现，就像调用独立的代码块那样。相反，编译器应将方法体直接插入调用方法的位置。这个过程称为内联(inline)，具有这一行为的方法称为内联方法。

可在方法或函数定义的名称之前使用 `inline` 关键字，将某个方法或函数指定为内联的。例如，要让 `SpreadsheetCell` 类的访问方法成为内联的，可以这样定义：

```
inline double SpreadsheetCell::getValue() const
{
    m_numAccesses++;
    return m_value;
}

inline std::string SpreadsheetCell::getString() const
{
    m_numAccesses++;
    return doubleToString(m_value);
}
```

这提示编译器，用实际的方法体替换对 `getValue()` 和 `getString()` 的调用，而不是生成代码进行函数调用。注意，`inline` 关键字只是提示编译器。如果编译器认为这会降低性能，就会忽略该关键字。

需要注意，在所有调用了内联函数或内联方法的源文件中，内联方法或内联函数的定义必须有效。考虑这个问题：如果没有看到函数的定义，编译器如何替换函数体？因此，如果编写了内联方法，应该将方法定义与其所在的类的定义放在同一文件中。

#### 注意：

高级 C++ 编译器不要求把内联方法的定义和类定义放在同一个文件中。例如，Microsoft Visual C++ 支持链接时代码生成(Link-Time Code Generation, LTCG)，会自动将较小的函数内联，哪怕这些函数没有声明为内联的或者没有在头文件中定义，也同样如此。GCC 和 Clang 具有类似的功能。

在 C++20 模块之外，如果方法的定义直接放在类定义中，则该方法会隐式标记为 `inline`，即使不使用 `inline` 关键字。对于从 C++20 中的模块导出的类，情况不再如此。如果希望这些方法是内联的，则需要使用 `inline` 关键字标记它们。例如：

```
export class SpreadsheetCell
{
public:
    inline double getValue() const { m_numAccesses++; return m_value; }

    inline std::string getString() const
    {
        m_numAccesses++;
        return doubleToString(m_value);
    }
}
```

```

    }
    // Omitted for brevity
};

```

**注意：**

如果使用调试器单步调试内联函数的调用，某些高级的 C++ 调试器会跳到内联函数实际的源代码处，就会造成函数调用的假象，但实际上代码是内联的。

许多 C++ 程序员了解 `inline` 方法的语法并使用这种语法，但不理解把方法标记为内联的结果。把方法或函数标记为 `inline`，仅提示编译器要内联函数或方法。编译器只会内联最简单的方法和函数，如果将编译器不想内联的方法定义为内联方法，编译器会自动忽略这个指令。现代编译器在内联方法或函数之前，会考虑代码膨胀等指标，因此不会内联任何没有效益的方法。

### 9.3.5 默认参数

C++ 中，默认参数(default arguments)与方法重载类似。在原型中可为函数或方法的参数指定默认值。如果用户指定了这些参数，默认值会被忽略；如果用户忽略了这些参数，将会使用默认值。但是存在一个限制：只能从最右边的参数开始提供连续的默认参数列表，否则编译器将无法用默认参数匹配缺失的参数。默认参数可用于函数、方法和构造函数。例如，可在 `Spreadsheet` 构造函数中设置宽度和高度的默认值。

```

export class Spreadsheet
{
    public:
        Spreadsheet(size_t width = 100, size_t height = 100);
        // Omitted for brevity
};

```

`Spreadsheet` 构造函数的实现不变。注意只在方法声明中指定了默认参数，在定义中没有这么做。现在可以用 0、1 或 2 个参数调用 `Spreadsheet` 构造函数，尽管只有一个非拷贝构造函数。

```

Spreadsheet s1;
Spreadsheet s2 { 5 };
Spreadsheet s3 { 5, 6 }

```

所有参数都有默认值的构造函数等同于默认构造函数。也就是说，可构建类的对象而不指定任何参数。如果试图同时声明默认构造函数，以及具有多个参数并且所有参数都有默认值的构造函数，编译器会报错。因为如果不指定任何参数，编译器不知道该调用哪个构造函数。

注意，任何默认参数能做到的事情，都可以用方法重载做到。可编写 3 个不同的构造函数，每个都具有不同数量的参数。然而，默认参数允许在一个构造函数中使用 3 个不同数量的参数。应该使用最得心应手的机制。

## 9.4 不同的数据成员类型

C++ 为数据成员提供了多种选择。除了在类中简单地声明数据成员外，还可创建 `static` 数据成员(类的所有对象共享)、`const` 数据成员、引用数据成员、`const` 引用数据成员和其他成员。本节解释这些不同类型的数据成员。

### 9.4.1 静态数据成员

有时让类的所有对象都包含某个变量的副本是没必要的。数据成员可能只对类有意义，而每个对象都拥有其副本是不合适的。例如，每个电子表格或许需要一个唯一的数字 ID，这需要一个从 0 开始的计数器，每个对象都可以从这个计数器得到自身的 ID。电子表格的计数器确实属于 `Spreadsheet` 类，但没必要使每个 `Spreadsheet` 对象都包含这个计数器的副本，因为必须让所有的计数器都保持同步。C++用 `static` 数据成员解决了这个问题。`static` 数据成员属于类但不是对象的数据成员，可将 `static` 数据成员当作类的全局变量。下面是 `Spreadsheet` 类的定义，其中包含了新的 `static` 数据成员。

```
export class Spreadsheet
{
    // Omitted for brevity
private:
    static size_t ms_counter;
};
```

不仅要在类定义中列出 `static` 类成员，还需要在源文件中为其分配内存，通常是定义类方法的那个源文件。在此还可初始化 `static` 成员，但注意与普通的变量和数据成员不同，默认情况下它们会初始化为 0。`static` 指针会初始化为 `nullptr`。下面是为 `ms_counter` 分配空间并初始化为 0 的代码：

```
size_t Spreadsheet::ms_counter;
```

静态数据成员默认情况下初始化为 0，但如果需要，可将它们显式地初始化为 0，如下所示。

```
size_t Spreadsheet::ms_counter { 0 };
```

这行代码在函数或方法外部，与声明全局变量非常类似，只是使用作用域解析运算符 `Spreadsheet::` 指出这是 `Spreadsheet` 类的一部分。

#### 1. 内联变量

从 C++17 开始，可将静态数据成员声明为 `inline`。这样做的好处是不必在源文件中为它们分配空间。下面是一个示例：

```
export class Spreadsheet
{
    // Omitted for brevity
private:
    static inline size_t ms_counter { 0 };
};
```

注意其中的 `inline` 关键字。有了这个类定义，可从源文件中删除下面的代码行。

```
size_t Spreadsheet::ms_counter;
```

#### 2. 在类方法内访问静态数据成员

在类方法内部，可以像使用普通数据成员那样使用静态数据成员。例如，为 `Spreadsheet` 类创建一个 `m_id` 成员，并在 `Spreadsheet` 构造函数中用 `ms_counter` 成员初始化它。下面是包含了 `m_id` 成员的 `Spreadsheet` 类定义。

```
export class Spreadsheet
{
public:
```

```

    // Omitted for brevity
    size_t getId() const;
private:
    // Omitted for brevity
    static inline size_t ms_counter { 0 };
    size_t m_id { 0 };
};

```

下面是 Spreadsheet 构造函数的实现，在此赋予初始 ID。

```

Spreadsheet::Spreadsheet(size_t width, size_t height)
: m_id { ms_counter++ }, m_width { width }, m_height { height }
{
    // Omitted for brevity
}

```

可以看出，构造函数可访问 ms\_counter，就像这是一个普通成员。在拷贝构造函数中，也要指定新的 ID。由于 Spreadsheet 拷贝构造函数委托给非拷贝构造函数(会自动创建新的 ID)，因此这可以自动进行处理。

对于本例，可以假设一旦给某个对象指定 ID，就不应该再改变。所以，在拷贝赋值运算符中不应该复制 ID。因此，建议把 m\_id 设置为 const 数据成员。

```

export class Spreadsheet
{
private:
    // Omitted for brevity
    const size_t m_id { 0 };
};

```

由于 const 数据成员一经创建就无法更改，例如，无法在构造函数体内初始化它们。此类数据成员必须直接在类定义内部或构造函数初始化器中初始化，这也意味着不能在赋值运算符中为此类数据成员赋值。这对于 m\_id 来说不是问题，因为一旦电子表格有了 ID，它就永远不会改变。但是，取决于实际情况，如果这使类变得不可赋值，则通常会显式删除赋值运算符。

### 3. 在方法外访问静态数据成员

访问控制限定符适用于 static 数据成员：ms\_counter 是 private 的，因此不能在类方法之外访问。如果 ms\_counter 是 public 的，就可在类方法外访问，具体方法是用::作用域解析运算符指出这个变量是 Spreadsheet 类的一部分。

```
int c { Spreadsheet::ms_counter };
```

然而，建议不要使用 public 数据成员(下一节讨论的 const static 数据成员属于例外)。应该提供 public 的 get/set 方法来授予访问权限。如果要访问 static 数据成员，可以实现 static 的 get/set 方法。

#### 9.4.2 const static 数据成员

类中的数据成员可声明为 const，意味着在创建并初始化后，数据成员的值不能再改变。如果某个常量只适用于类，应该使用 static const(或 const static)数据成员，也称为类常量，而不是全局常量。可在类定义中定义和初始化整型和枚举类型的 static const 数据成员，而不需要将其指定为内联变量。例如，你可能想指定电子表格的最大高度和宽度。如果用户想要创建的电子表格的高度或宽度大于最大值，就改用最大值。可将最大高度和宽度设置为 Spreadsheet 类的 static const 成员：

```

export class Spreadsheet
{
    public:
        // Omitted for brevity
        static const size_t MaxHeight { 100 };
        static const size_t MaxWidth { 100 };
};

```

可在构造函数中使用这些新常量，如下面的代码片段所示。

```

Spreadsheet::Spreadsheet(size_t width, size_t height)
    : m_id { ms_counter++ }
    , m_width { min(width, MaxWidth) } // std::min() requires <algorithm>
    , m_height { min(height, MaxHeight) }
{
    // Omitted for brevity
}

```

#### 注意：

当高度或宽度超出最大值时，除了自动使用最大高度或宽度外，也可以抛出异常。然而，在构造函数中抛出异常时，不会调用析构函数，因此需要谨慎处理。第 14 章将对此进行详细解释。

这些常量也可用作构造函数参数的默认值。记住，只能为一组连续的参数(从最右面的参数开始)指定默认值。例如：

```

export class Spreadsheet
{
    public:
        Spreadsheet(size_t width = MaxWidth, size_t height = MaxHeight);
        // Omitted for brevity
};

```

### 9.4.3 引用数据成员

Spreadsheet 和 SpreadsheetCell 很好，但这两个类本身并不能组成非常有用的应用程序。为了用代码控制整个电子表格程序，可将这两个类一起放入 SpreadsheetApplication 类。假设将来需要在每一个 Spreadsheet 类中存储一个应用程序对象的引用。SpreadsheetApplication 类的实现在此并不重要，所以下面的代码简单地将其定义为空类。Spreadsheet 类被修改了，以容纳一个名为 m\_theApp 的新的引用数据类型。

```

export class SpreadsheetApplication { };

export class Spreadsheet
{
    public:
        Spreadsheet(size_t width, size_t height,
                    SpreadsheetApplication& theApp);
        // Code omitted for brevity.
    private:
        // Code omitted for brevity.
        SpreadsheetApplication& m_theApp;
};

```

这个定义将一个 SpreadsheetApplication 引用作为数据成员添加进来。在此情况下建议使用引用而不是指针，因为 Spreadsheet 总会指向一个 SpreadsheetApplication，而指针则无法保证这一点。

请注意，存储对应用程序的引用，仅是为了演示把引用作为数据成员的用法。不建议以这种方式把 Spreadsheet 和 SpreadsheetApplication 类组合在一起，而应改用 MVC(模型-视图-控制器)范式(见第4章)。

在构造函数中，每个 Spreadsheet 都得到了一个应用程序引用。如果不指向某些事物，引用将无法存在，因此在构造函数的初始化器中必须给 m\_theApp 指定一个值。

```
Spreadsheet::Spreadsheet(size_t width, size_t height,
    SpreadsheetApplication& theApp)
: m_id { ms_counter++ }
, m_width { std::min(width, MaxWidth) }
, m_height { std::min(height, MaxHeight) }
, m_theApp { theApp }
{
    // Code omitted for brevity.
}
```

在拷贝构造函数中也必须初始化这个引用成员。由于 Spreadsheet 拷贝构造函数委托给非拷贝构造函数(初始化引用成员)，因此这将自动处理。

记住，在初始化一个引用后，不能改变它指向的对象，因此无法在赋值运算符中对引用赋值。这意味着根据使用情形，可能无法为具有引用数据成员的类提供赋值运算符。如果属于这种情况，通常将赋值运算符标记为删除。

最后，引用数据成员也可被标记为 const。例如，你可能决定让 Spreadsheet 只包含应用程序对象的 const 引用，只需要在类定义中将 m\_theApp 声明为 const 引用。

```
export class Spreadsheet
{
public:
    Spreadsheet(size_t width, size_t height,
        const SpreadsheetApplication& theApp);
    // Code omitted for brevity.
private:
    // Code omitted for brevity.
    const SpreadsheetApplication& m_theApp;
};
```

## 9.5 嵌套类

类定义不仅可包含成员函数和数据成员，还可编写嵌套类和嵌套结构体、声明类型别名或者创建枚举类型。类中声明的一切内容都具有类作用域。如果声明的内容是 public 的，那么可在类外使用 ClassName::作用域解析语法访问。

可在类的定义中提供另一个类定义。例如，假定 SpreadsheetCell 类实际上是 Spreadsheet 类的一部分，因此不妨将 SpreadsheetCell 重命名为 Cell。可将二者定义为：

```
export class Spreadsheet
{
public:
    class Cell
    {
public:
    Cell() = default;
```

```

        Cell(double initialValue);
        // Omitted for brevity
    };

    Spreadsheet(size_t width, size_t height,
               const SpreadsheetApplication& theApp);
    // Remainder of Spreadsheet declarations omitted for brevity
};

```

现在 Cell 类定义位于 Spreadsheet 类内部，因此在 Spreadsheet 类外引用 Cell 必须用 Spreadsheet:: 作用域限定名称，即使在方法定义时也是如此。例如，Cell 的 double 构造函数应如下所示：

```

Spreadsheet::Cell::Cell(double initialValue)
    : m_value { initialValue }
{
}

```

甚至在 Spreadsheet 类中方法的返回类型(不是参数)也必须使用这一语法：

```

Spreadsheet::Cell& Spreadsheet::getCellAt(size_t x, size_t y)
{
    verifyCoordinate(x, y);
    return m_cells[x][y];
}

```

如果在 Spreadsheet 类中直接完整定义嵌套的 Cell 类，将使 Spreadsheet 类的定义略显臃肿。为缓解这一点，只需要在 Spreadsheet 中为 Cell 添加前置声明，然后独立地定义 Cell 类，如下所示。

```

export class Spreadsheet
{
public:
    class Cell;

    Spreadsheet(size_t width, size_t height,
               const SpreadsheetApplication& theApp);
    // Remainder of Spreadsheet declarations omitted for brevity
};

class Spreadsheet::Cell
{
public:
    Cell() = default;
    Cell(double initialValue);
    // Omitted for brevity
};

```

普通的访问控制也适用于嵌套类定义。如果声明了一个 private 或 protected 嵌套类，这个类只能在外围类(outer class，即包含它的类)中使用。嵌套的类有权访问外围类中的所有 private 或 protected 成员，而外围类却只能访问嵌套类中的 public 成员。

## 9.6 类内的枚举类型

枚举类型也可以作为类的数据成员。例如，可在 SpreadsheetCell 类中支持单元格颜色，如下所示。

```

export class SpreadsheetCell
{

```

```

public:
    // Omitted for brevity
    enum class Color { Red = 1, Green, Blue, Yellow };
    void setColor(Color color);
    Color getColor() const;
private:
    // Omitted for brevity
    Color m_color { Color::Red };
};

```

setColor()和getColor()方法的实现简单明了：

```

void SpreadsheetCell::setColor(Color color) { m_color = color; }
SpreadsheetCell::Color SpreadsheetCell::getColor() const { return m_color; }

```

可通过下面的方式使用这些新方法：

```

SpreadsheetCell myCell { 5 };
myCell.setColor(SpreadsheetCell::Color::Blue);
auto color { myCell.getColor() };

```

## 9.7 运算符重载

经常需要在对象上执行操作，例如相加、比较、将对象输入文件或者从文件读取。对电子表格而言，只有能执行算术运算(例如将整行单元格相加)才算真正有用。

### 9.7.1 示例：为 SpreadsheetCell 实现加法

在真正的面向对象风格中，SpreadsheetCell 对象应该能与其他 SpreadsheetCell 对象相加。将一个单元格与另一个单元格相加，结果放在第 3 个单元格中，不会改变原始单元格。将 SpreadsheetCell 对象相加的意义是单元格值的相加。

#### 1. 首次尝试：add 方法

可像下面这样声明并定义 SpreadsheetCell 类的 add()方法：

```

export class SpreadsheetCell
{
    public:
        SpreadsheetCell add(const SpreadsheetCell& cell) const;
        // Omitted for brevity
};

```

这个方法将两个单元格相加，返回第三个新的单元格，其值为前两个单元格的和。将这个方法声明为 const，并把 const SpreadsheetCell 的引用作为参数，原因是 add()不改变任意一个原始单元格。下面是这个方法的实现：

```

SpreadsheetCell SpreadsheetCell::add(const SpreadsheetCell& cell) const
{
    return SpreadsheetCell { getValue() + cell.getValue() };
}

```

可以这样使用 add()方法：

```

SpreadsheetCell myCell { 4 }, anotherCell { 5 };

```

```
SpreadsheetCell aThirdCell { myCell.add(anotherCell) };
auto aFourthCell { aThirdCell.add(anotherCell) };
```

这样做可行，但是有一点笨拙。还可以做得更好。

## 2. 第二次尝试：将 operator+ 作为方法重载

用加号相加两个单元格会比较方便，就像相加两个 int 或 double 值那样，如下所示。

```
SpreadsheetCell myCell { 4 }, anotherCell { 5 };
SpreadsheetCell aThirdCell { myCell + anotherCell };
auto aFourthCell { aThirdCell + anotherCell };
```

C++ 允许编写自己的加号版本，以正确地处理类，称为加运算符。为此可编写一个名为 operator+ 的方法，如下所示。

```
export class SpreadsheetCell
{
    public:
        SpreadsheetCell operator+(const SpreadsheetCell& cell) const;
        // Omitted for brevity
};
```

### 注意：

在 operator 和加号之间可以使用空格。例如，可用 operator+ 替代 operator+。本书采用没有空格的格式。

该方法的实现与 add() 方法的实现一样：

```
SpreadsheetCell SpreadsheetCell::operator+(const SpreadsheetCell& cell) const
{
    return SpreadsheetCell { getValue() + cell.getValue() };
}
```

现在可使用加号将两个单元格相加，就像前面做的那样。

这种语法需要花点工夫去适应。不要过于担心这个奇怪的方法名称 operator+——这只是一个名称，就像 foo 或 add 一样。理解此处实际发生的事情有助于理解其余的语法。当 C++ 编译器分析一个程序，遇到运算符（例如，+、-、= 或 <>）时，就会试着查找名为 operator+、operator-、operator= 或 operator<<，且具有适当参数的函数或方法。例如，当编译器看到下面这行时，就会试着查找 SpreadsheetCell 类中名为 operator+ 并将另一个 SpreadsheetCell 对象作为参数的方法，或者查找用两个 SpreadsheetCell 对象作为参数、名为 operator+ 的全局函数。

```
SpreadsheetCell aThirdCell { myCell + anotherCell };
```

如果 SpreadsheetCell 类包含 operator+ 方法，上述代码就会转换为：

```
SpreadsheetCell aThirdCell { myCell.operator+(anotherCell) };
```

注意，用作 operator+ 参数的对象类型并不一定要与编写 operator+ 的类相同。可为 SpreadsheetCell 编写 operator+，将 Spreadsheet 与 SpreadsheetCell 相加。对于这个程序而言这样做没有意义，但是编译器允许这样做。

此外还要注意，可任意指定 operator+ 的返回值类型。但你应该遵循最不使人诧异的原则，也就是说，operator+ 的返回值通常应该是用户期望的。

### 隐式转换

令人惊讶的是，一旦编写前面所示的 `operator+`，不仅可将两个单元格相加，还可将单元格与 `string_view`、`double` 或 `int` 值相加。

```
SpreadsheetCell myCell { 4 }, aThirdCell;
string str { "hello" };
aThirdCell = myCell + string_view{ str };
aThirdCell = myCell + 5.6;
aThirdCell = myCell + 4;
```

上面的代码之所以可运行，是因为编译器会试着查找合适的 `operator+`，而不是只查找指定类型的那个 `operator+`。为找到 `operator+`，编译器还试图查找合适的类型转换。`SpreadsheetCell` 类拥有转换构造函数(见第 8 章)，可以将 `double` 或 `string_view` 转换为 `SpreadsheetCell`。在上例中，当编译器看到 `SpreadsheetCell` 试图与 `double` 值相加时，发现了用 `double` 值作为参数的 `SpreadsheetCell` 构造函数，就会构建一个临时的 `SpreadsheetCell` 对象，传递给 `operator+`。与此类似，当编译器看到试图将 `SpreadsheetCell` 与 `string_view` 相加的代码时，会调用把 `string_view` 作为参数的 `SpreadsheetCell` 构造函数，创建一个临时 `SpreadsheetCell` 对象，传递给 `operator+`。

由于必须创建临时对象，隐式使用构造函数的效率不高。为避免与 `double` 值相加时隐式地使用构造函数，可编写第二个 `operator+`，如下所示。

```
SpreadsheetCell SpreadsheetCell::operator+(double rhs) const
{
    return SpreadsheetCell { getValue() + rhs };
}
```

### 3. 第三次尝试：全局 `operator+`

隐式转换允许使用 `operator+` 方法将 `SpreadsheetCell` 对象与 `int` 和 `double` 值相加。然而，这个运算符不具有互换性，如下所示。

```
aThirdCell = myCell + 5.6;           // Works fine.
aThirdCell = myCell + 4;             // Works fine.
aThirdCell = 5.6 + myCell;          // FAILS TO COMPILE!
aThirdCell = 4 + myCell;            // FAILS TO COMPILE!
```

当 `SpreadsheetCell` 对象在运算符的左边时，隐式转换正常运行，但在右边时无法运行。加法是可交换的，因此这里存在错误。问题在于必须在 `SpreadsheetCell` 对象上调用 `operator+` 方法，对象必须在 `operator+` 的左边。这是 C++ 语言定义的方式，因此使用 `operator+` 方法无法让上面的代码运行。

然而，如果用不局限于某个特定对象的全局 `operator+` 函数替换类内的 `operator+` 方法，上面的代码就可以运行，函数如下所示。

```
SpreadsheetCell operator+(const SpreadsheetCell& lhs,
                           const SpreadsheetCell& rhs)
{
    return SpreadsheetCell { lhs.getValue() + rhs.getValue() };
}
```

需要在模块接口文件中声明运算符并将其导出：

```
export class SpreadsheetCell { /* Omitted for brevity */ };

export SpreadsheetCell operator+(const SpreadsheetCell& lhs,
```

```
const SpreadsheetCell& rhs);
```

这样，下面的 4 个加法运算都可按预期运行：

```
aThirdCell = myCell + 5.6; // Works fine.  
aThirdCell = myCell + 4; // Works fine.  
aThirdCell = 5.6 + myCell; // Works fine.  
aThirdCell = 4 + myCell; // Works fine.
```

那么，如果编写以下代码，会发生什么情况呢？

```
aThirdCell = 4.5 + 5.5;
```

这段代码可编译并运行，但并没有调用前面编写的 `operator+`。这段代码将普通的 `double` 型数值 4.5 和 5.5 相加，得到了下面所示的中间语句。

```
aThirdCell = 10;
```

为了让赋值操作继续，运算符右边应该是 `SpreadsheetCell` 对象。编译器找到非 `explicit` 的由用户定义的用 `double` 值作为参数的构造函数，然后用这个构造函数隐式地将 `double` 值转换为一个临时 `SpreadsheetCell` 对象，最后调用赋值运算符。

#### 注意：

在 C++ 中，不能更改运算符的优先级。例如，`*` 和 `/` 始终在 `+` 和 `-` 之前计算。对于用户定义的运算符，唯一能做的只是在确定运算的优先级后指定其实现。C++ 也不允许发明新的运算符号，不允许更改运算符的实参数数。

### 9.7.2 重载算术运算符

现在，你理解了如何编写 `operator+`，剩余的基本算术运算符就变得简单了。下面是 `+`、`-`、`*` 和 `/` 的声明（必须用 `+`、`-`、`*` 和 `/` 替代 `<op>`，最终得到 4 个函数）。还可重载 `%`，但这对于存储在 `SpreadsheetCell` 中的 `double` 值而言没有意义。

```
export class SpreadsheetCell { /* Omitted for brevity */ };

export SpreadsheetCell operator<op>(const SpreadsheetCell& lhs,
    const SpreadsheetCell& rhs);
```

`operator-` 和 `operator*` 的实现与 `operator+` 十分类似，因此这里未显示。对于 `operator/` 而言，唯一棘手之处是记着检查除数是否为 0。如果检测到除数为 0，该实现将抛出异常。

```
SpreadsheetCell operator/(const SpreadsheetCell& lhs,
    const SpreadsheetCell& rhs)
{
    if (rhs.getValue() == 0) {
        throw invalid_argument { "Divide by zero." };
    }
    return SpreadsheetCell { lhs.getValue() / rhs.getValue() };
}
```

C++ 并没有真正要求在 `operator*` 中实现乘法，在 `operator/` 中实现除法。可在 `operator/` 中实现乘法，在 `operator+` 中实现除法，以此类推。然而这样做会让人非常迷惑，也没理由这么做。在实现中应该尽量使用常用的运算符含义。

### 重载简写算术运算符

除基本算术运算符外，C++还提供了简写运算符，例如`+=`和`-=`。你或许认为编写类的`operator+`时也就提供了`operator+=`。不是这么简单，必须显式地重载简写算术运算符(Arithmetic Shorthand Operators)。这些运算符与基本算术运算符不同，它们会改变运算符左边的对象，而不是创建一个新对象。此外还有一个微妙差别，它们生成的结果是对被修改对象的引用，这一点与赋值运算符类似。

简写算术运算符的左边总要有一个对象，因此应该将其作为方法而不是全局函数。下面是`SpreadsheetCell`类的声明：

```
export class SpreadsheetCell
{
public:
    SpreadsheetCell& operator+=(const SpreadsheetCell& rhs);
    SpreadsheetCell& operator-=(const SpreadsheetCell& rhs);
    SpreadsheetCell& operator*=(const SpreadsheetCell& rhs);
    SpreadsheetCell& operator/=(const SpreadsheetCell& rhs);
    // Omitted for brevity
};
```

下面是`operator+=`的实现，其他的与此类似。

```
SpreadsheetCell& SpreadsheetCell::operator+=(const SpreadsheetCell& rhs)
{
    set(getValue() + rhs.getValue());
    return *this;
}
```

简写算术运算符是对基本算术运算符和赋值运算符的结合。根据上面的定义，可编写如下代码：

```
SpreadsheetCell myCell { 4 }, aThirdCell { 2 };
aThirdCell -= myCell;
aThirdCell += 5.4;
```

然而不能编写这样的代码(这是好事一桩！)：

```
5.4 += aThirdCell;
```

如果既有某个运算符的普通版本，又有简写版本，建议你基于简写版本实现普通版本，以避免代码重复。例如：

```
SpreadsheetCell operator+(const SpreadsheetCell& lhs, const SpreadsheetCell& rhs)
{
    auto result { lhs };      // Local copy
    result += rhs;           // Forward to +=()
    return result;
}
```

### 9.7.3 重载比较运算符

比较运算符(例如`>`、`<`、`<=`、`>=`、`==`和`!=`)是另一组对类有用的运算符。C++20 标准为这些操作符带来了很多变化，并添加了三向比较运算符，也称为宇宙飞船运算符，`<=>`，第 1 章曾介绍过。为了让你更了解 C++20 提供的功能，让我们首先看看在 C++20 之前必须做什么，或者说，只要编译器还不支持三向比较运算符，你还需要做什么。

与基本算术运算符一样，6 个 C++20 之前的比较运算符应该是全局函数，这样就可以在运算符的

左侧和右侧参数上使用隐式转换。比较运算符都返回 `bool`，当然，也可以更改返回类型，但不建议这样做。

下面是比较运算符的声明；必须用`==`、`<`、`>`、`!=`、`<=`和`>=`替换`<op>`，得到 6 个函数。

```
export class SpreadsheetCell { /* Omitted for brevity */ };

export bool operator<op>(const SpreadsheetCell& lhs, const SpreadsheetCell& rhs);
```

下面是 `operator==` 的定义，其他的与此类似：

```
bool operator==(const SpreadsheetCell& lhs, const SpreadsheetCell& rhs)
{
    return (lhs.getValue() == rhs.getValue());
}
```

### 注意：

前面重载的比较运算符使用了 `double` 值。大多数时候，最好不要对浮点数执行相等或不相等测试。应该使用  $\epsilon$  测试(epsilon test)，但这一内容超出了本书的讨论范围。

当类中的数据成员较多时，比较每个数据成员可能比较痛苦。然而，当实现了`=`和`<`之后，可以根据这两个运算符编写其他比较运算符。例如，下面的 `operator>=` 定义使用了 `operator<`。

```
bool operator>=(const SpreadsheetCell& lhs, const SpreadsheetCell& rhs)
{
    return !(lhs < rhs);
}
```

可使用这些运算符将某个 `SpreadsheetCell` 与其他 `SpreadsheetCell` 进行比较，也可与 `double` 和 `int` 值进行比较。

```
if (myCell > aThirdCell || myCell < 10) {
    cout << myCell.getValue() << endl;
}
```

如你所见，需要编写 6 个单独的函数来支持 6 个比较运算符，这只是为了相互比较两个 `SpreadsheetCell`。使用当前实现的 6 个比较函数，可以将 `SpreadsheetCell` 与 `double` 进行比较，因为 `double` 参数隐式转换为 `SpreadsheetCell`。如前所述，这种隐式转换可能效率低下，因为必须创建临时对象。与前面的 `operator+`一样，可以通过显式实现与 double 进行比较的函数来避免这种情况。对于每个操作符<op>，将需要以下 3 个重载。`

```
bool operator<op>(const SpreadsheetCell& lhs, const SpreadsheetCell& rhs);
bool operator<op>(double lhs, const SpreadsheetCell& rhs);
bool operator<op>(const SpreadsheetCell& lhs, double rhs);
```

如果你需要支持所有的比较运算符，需要写很多重复的代码！

 现在让我们切换一下，看看 C++20 带来了什么。C++20 简化了向类中添加对比较运算符的支持。首先，对于 C++20，实际上建议将 `operator==` 实现为类的成员函数，而不是全局函数。还要注意，添加`[[nodiscard]]` 属性是一个好主意，这样操作符的结果就不能被忽略。例如：

```
[[nodiscard]] bool operator==(const SpreadsheetCell& rhs) const;
```

在 C++20 中，单个的 operator== 重载就可以使下面的比较生效。

```
if (myCell == 10) { cout << "myCell == 10\n"; }
if (10 == myCell) { cout << "10 == myCell\n"; }
```

诸如 10==myCell 的表达式由 C++20 编译器重写为 myCell==10，可以为其调用 operator== 成员函数。此外，通过实现 operator==，C++20 会自动添加对!=的支持。

接下来，要实现对全套比较运算符的支持，在 C++20 中，只需要实现一个额外的重载运算符，operator<=。一旦类有运算符==和<=的重载，C++20 就会自动为所有 6 个比较运算符提供支持！对于 SpreadsheetCell 类，运算符<=如下所示。

```
[[nodiscard]] std::partial_ordering operator<=(const SpreadsheetCell& rhs) const;
```

### 注意：

C++20 编译器不会根据<=重写==或!=。这样做是为了避免性能问题，因为 operator==的显式实现通常比使用<=更高效。

存储在 SpreadsheetCell 中的值是 double 类型的。请记住，第 1 章提到过，浮点类型只有偏序，这就是重载返回 std::partial\_ordering 的原因。实现非常简单：

```
partial_ordering SpreadsheetCell::operator<=(const SpreadsheetCell& rhs) const
{
    return getValue() <= rhs.getValue();
}
```

通过实现 operator<=，C++20 可以自动提供对>、<、<=和>=的支持，通过将使用这些运算符的表达式重写为使用<=的。例如，诸如 myCell<aThirdCell 的表达式会自动重写为与 std::is\_lt(myCell<=aThirdCell)等价的内容，其中 is\_lt()是一个命名的比较函数，见第 1 章。

所以，只需要实现 operator== 和 operator<=，SpreadsheetCell 类就可以支持所有的比较运算符。

```
if (myCell < aThirdCell) { cout << "myCell < aThirdCell\n"; }
if (aThirdCell < myCell) { cout << "aThirdCell < myCell\n"; }

if (myCell <= aThirdCell) { cout << "myCell <= aThirdCell\n"; }
if (aThirdCell <= myCell) { cout << "aThirdCell <= myCell\n"; }

if (myCell > aThirdCell) { cout << "myCell > aThirdCell\n"; }
if (aThirdCell > myCell) { cout << "aThirdCell > myCell\n"; }

if (myCell >= aThirdCell) { cout << "myCell >= aThirdCell\n"; }
if (aThirdCell >= myCell) { cout << "aThirdCell >= myCell\n"; }

if (myCell == aThirdCell) { cout << "myCell == aThirdCell\n"; }
if (aThirdCell == myCell) { cout << "aThirdCell == myCell\n"; }

if (myCell != aThirdCell) { cout << "myCell != aThirdCell\n"; }
if (aThirdCell != myCell) { cout << "aThirdCell != myCell\n"; }
```

由于 SpreadsheetCell 类支持从 double 到 SpreadsheetCell 的隐式转换，因此也支持以下比较：

```
if (myCell < 10) { cout << "myCell < 10\n"; }
if (10 < myCell) { cout << "10 < myCell\n"; }
if (10 != myCell) { cout << "10 != myCell\n"; }
```

与比较两个 SpreadsheetCell 对象一样，编译器根据运算符—和 $\leq\geq$ 重写此类表达式，并可选地交换参数的顺序。例如，`10<myCell`首先被重写为与 `is_lt(10<=myCell)`等价的内容，这将不起作用，因为只有一个作为成员的 $\leq\geq$ 重载，这意味着左侧参数必须是 SpreadsheetCell。注意到这一点后，编译器尝试将表达式重写为与 `is_gt(myCell<=10)`等价的内容，结果可以正常运行。

与前面一样，如果希望避免隐式转换对性能的轻微影响，可以为 `double` 提供特定的重载。对于现在，有了 C++20，甚至没有太多的工作。只需要提供以下两个额外的重载运算符作为方法：

```
[[nodiscard]] bool operator==(double rhs) const;
[[nodiscard]] std::partial_ordering operator<=(double rhs) const;
```

它们的实现如下：

```
bool SpreadsheetCell::operator==(double rhs) const
{
    return getValue() == rhs;
}
std::partial_ordering SpreadsheetCell::operator<=(double rhs) const
{
    return getValue() <= rhs;
}
```

### 编译器生成的比较运算符

注意 `SpreadsheetCell` 的 `operator==` 和 `<=` 的实现，它们只是简单比较所有数据成员。在这种情况下，可以进一步减少需要编写的代码行数，因为 C++20 可以为我们编写这些代码。例如，拷贝构造函数可以显式设置为默认，`operator==` 和 `<=` 也可以默认，在这种情况下，编译器会为你编写它们，并通过依次比较每个数据成员来实现它们。此外，如果只是显式地使用默认 `operator<=`，编译器也会自动包含一个默认 `operator==`。因此，对于 `SpreadsheetCell` 类，如果没有显式的 `double` 版本的 `operator==` 和 `<=`，只需要编写以下单行代码，即可添加对所有 6 个比较运算符的完全支持，以比较两个 `SpreadsheetCell`。

```
[[nodiscard]] std::partial_ordering operator<=
    const SpreadsheetCell&) const = default;
```

此外，还可以使用 `auto` 作为 `operator<=` 的返回类型，在这种情况下，编译器将根据数据成员的 `<=` 运算符的返回类型推断返回类型。如果类的数据成员不支持 `operator<=`，则返回类型推断将不起作用，需要显式指定返回类型为 `strong_ordering`、`partial_ordering` 或 `weak_ordering`。为了使编译器能够编写默认的 `<=` 运算符，类的所有数据成员需要要么支持 `operator<=`，在这种情况下返回类型可以是 `auto`；要么支持 `operator<` 和 `==`，在这种情况下返回类型不能是 `auto`。由于 `SpreadsheetCell` 只有一个 `double` 数据成员，编译器将返回类型推断为 `partial_ordering`。

```
[[nodiscard]] auto operator<=(const SpreadsheetCell&) const = default;
```

本节开始时，曾提到这个显式默认的 `operator<=` 适用于没有显式的 `double` 版本的 `operator==` 和 `<=` 的 `SpreadsheetCell`。如果确实添加了这些显式 `double` 版本，则添加的是用户声明的 `operator==(double)`。因此，编译器将不再自动生成 `operator==(const SpreadsheetCell&)`，因此必须将其显式默认，如下所示。

```
export class SpreadsheetCell
{
public:
```

```

// Omitted for brevity
[[nodiscard]] auto operator<=>(const SpreadsheetCell&) const = default;
[[nodiscard]] bool operator==(const SpreadsheetCell&) const = default;

[[nodiscard]] bool operator==(double rhs) const;
[[nodiscard]] std::partial_ordering operator<=>(double rhs) const;
// Omitted for brevity
};

```

如果可以显式地将 `operator<=>` 设置为默认，我建议这样做，而不是自己实现它。通过让编译器为你编写，它将与新添加或修改的数据成员保持同步。如果自己实现运算符，则无论何时添加数据成员或更改现有数据成员，都需要记住更新 `operator<=>` 的实现。如果编译器没有自动生成 `operator==`，则以上内容对其也适用。

只有当 `operator==` 和 `<=>` 使用定义操作符的类类型的 `const` 引用作为参数时，才可能显式将 `operator==` 和 `<=>` 设置为默认。例如，以下操作不起作用：

```
[[nodiscard]] auto operator<=>(double) const = default; // Does not work!
```

#### 注意：

在 C++20 中为了添加对所有 6 个比较运算符的支持：

- 如果一个默认的 `operator<=>` 适用于你的类，那么只需要一行代码来显式地将默认运算符 `<=>` 作为一个方法。在某些情况下，你可能还需要显式地将 `operator==` 设置为默认。
- 否则，需要作为方法重载 `operator==` 和 `<=>` 的实现。  
不需要手动实现其他的比较运算符。

### 9.7.4 创建具有运算符重载的类型

许多人觉得运算符重载的语法深奥难懂，至少刚看上去是这样的。让事情变得简单好像是一句反话，对于编写类的人来说这并不简单，但是对于使用类的人来说确实简单。关键在于使新类尽量类似于内建类型(例如 `int` 和 `double`)。在执行两个对象的加法时，使用`+`比记住应该调用 `add()` 还是 `sum()` 更简单。

#### 注意：

提供运算符重载，将其作为向类的客户提供的服务。

应该重载哪些运算符？答案是“几乎全部运算符都可以重载——甚至是从来没有听说过的那些”。实际上，你刚刚触及了表面：在 8.3 节“对象的生命周期”中看到的赋值运算符、基本算术运算符、简写算术运算符和比较运算符。重载流插入和提取操作符也很有用。此外，还有一些棘手但有趣的事情，可以通过运算符重载来完成，而这些事情在一开始可能是预料不到的。标准库广泛使用了运算符重载。第 15 章“重载 C++ 运算符”解释了如何以及何时重载其余的运算符。第 16~24 章介绍了标准库。

## 9.8 创建稳定的接口

理解了在 C++ 中编写类的所有语法后，回顾第 5 章和第 6 章的设计原则会对此有所帮助。在 C++ 中，类是主要的抽象单元，应将抽象原则应用到类，尽可能分离接口和实现。确切地讲，应该将所

有数据成员设置为 `private`, 并提供相应的获取器和设置器方法。这就是 `SpreadsheetCell` 类的实现方式: 将 `m_value` 设置为 `private`, 用 `public` 的 `set()` 设置这个值, 用 `getValue()` 和 `getString()` 获取这些值。

## 使用接口类和实现类

即使提前进行估算并采用最佳设计原则, C++语言本质上对抽象原则也不友好。其语法要求将 `public` 接口和 `private`(或 `protected`)数据成员及方法放在一个类定义中, 从而将类的某些内部实现细节向客户公开。这种做法的缺点在于, 如果不得不在类中加入新的非 `public` 方法或数据成员, 所有的客户代码都必须重新编译, 对于较大项目而言这是负担。

有个好消息: 可创建清晰的接口, 并隐藏所有实现细节, 从而得到稳定的接口。还有个坏消息: 这样做有点繁杂。基本原则是为想编写的每个类都定义两个类: 接口类和实现类。实现类与不采用这种方法编写的类相同, 接口类给出了与实现类一样的 `public` 方法, 但只有一个数据成员: 指向实现类对象的一个指针。这称为 `pimpl idiom`(*private implementation idiom*, 私有实现惯用方法)或 `bridge` 模式, 接口类方法的实现只是调用实现类对象的等价方法。这样做的结果是无论实现如何改变, 都不会影响 `public` 接口类, 从而降低了重新编译的可能。当有且只有实现改变时, 使用接口类的客户不需要重新编译。注意只有在单个数据成员是实现类的指针时, 这个方法才有效。如果它是按值传递的数据成员, 在实现类的定义改变时, 客户代码必须重新编译。

为将这种方法应用到 `Spreadsheet` 类, 需要定义如下 `public` 接口类 `Spreadsheet`。重要的部分以高亮表示。

```
module;
#include <cstddef>

export module spreadsheet;

export import spreadsheet_cell;
import <memory>;

export class SpreadsheetApplication { };

export class Spreadsheet
{
    public:
        Spreadsheet(const SpreadsheetApplication& theApp,
                    size_t width = MaxWidth, size_t height = MaxHeight);
        Spreadsheet(const Spreadsheet& src);
        Spreadsheet(Spreadsheet&&) noexcept;
        ~Spreadsheet();

        Spreadsheet& operator=(const Spreadsheet& rhs);
        Spreadsheet& operator=(Spreadsheet&&) noexcept;

        void setCellAt(size_t x, size_t y, const SpreadsheetCell& cell);
        SpreadsheetCell& getCellAt(size_t x, size_t y);

        size_t getId() const;

        static const size_t MaxHeight { 100 };
        static const size_t MaxWidth { 100 };

        void swap(Spreadsheet& other) noexcept;
```

```

private:
    class Impl;
    std::unique_ptr<Impl> m_Impl;
};

export void swap(Spreadsheet& first, Spreadsheet& second) noexcept;

```

实现类 `Impl` 是一个 `private` 嵌套类，因为只有 `Spreadsheet` 需要用到这个实现类。`Spreadsheet` 现在只包含一个数据成员：指向 `Impl` 实例的指针。`public` 方法与原来的 `Spreadsheet` 相同。

嵌套的 `Spreadsheet::Impl` 类定义在 `spreadsheet` 模块实现文件中。`Impl` 类不应被导出，因为它应该对用户隐藏起来。`Spreadsheet.cpp` 模块实现文件的开头如下所示：

```

module;
#include <cstddef>

module spreadsheet;

import <utility>;
import <stdexcept>;
import <format>;
import <algorithm>;

using namespace std;

// Spreadsheet::Impl class definition.
class Spreadsheet::Impl { /* Omitted for brevity */ };

// Spreadsheet::Impl method definitions.
Spreadsheet::Impl::Impl(const SpreadsheetApplication& theApp,
    size_t width, size_t height)
: m_id { ms_counter++ }
, m_width { min(width, Spreadsheet::MaxWidth) }
, m_height { min(height, Spreadsheet::MaxHeight) }
, m_theApp { theApp }
{
    m_cells = new SpreadsheetCell*[m_width];
    for (size_t i{ 0 }; i < m_width; i++) {
        m_cells[i] = new SpreadsheetCell[m_height];
    }
}
// Other method definitions omitted for brevity.

```

`Impl` 类具有与 `Spreadsheet` 类几乎相同的接口。对于方法实现，需要记住 `Impl` 是一个嵌套类，因此，需要将作用域指定为 `Spreadsheet::Impl`。因此，对于构造函数，它变成了 `Spreadsheet::Impl::Impl(...)`。

现在，`Spreadsheet` 类有一个指向实现类的 `unique_ptr`，`Spreadsheet` 类需要一个用户声明的析构函数。我们不需要对这个析构函数进行任何处理，可在文件中设置=`default`，如下所示。

```
Spreadsheet::~Spreadsheet() = default;
```

事实上，它必须在实现文件中而不是直接在类定义中默认。原因是 `Impl` 类仅在 `Spreadsheet` 类定义中提前声明，也就是说，编译器知道某个地方将有一个 `Spreadsheet::Impl` 类，但此时它还不知道定义。因此，不能在类定义中设置默认析构函数，因为这样编译器将尝试使用尚未定义的 `Impl` 类的析构函数。在这种情况下，将其他方法设置为默认时也是如此，例如移动构造函数和移动赋值运算符。

`Spreadsheet` 方法(例如 `setCellAt()` 和 `getCellAt()`) 的实现只是将请求传递给底层的 `Impl` 对象:

```
void Spreadsheet::setCellAt(size_t x, size_t y, const SpreadsheetCell& cell)
{
    m_impl->setCellAt(x, y, cell);
}

SpreadsheetCell& Spreadsheet::getCellAt(size_t x, size_t y)
{
    return m_impl->getCellAt(x, y);
}
```

`Spreadsheet` 的构造函数必须创建一个新的 `Impl` 实例来完成这个任务。

```
Spreadsheet::Spreadsheet(const SpreadsheetApplication& theApp,
    size_t width, size_t height)
{
    m_impl = make_unique<Impl>(theApp, width, height);
}

Spreadsheet::Spreadsheet(const Spreadsheet& src)
{
    m_impl = make_unique<Impl>(*src.m_impl);
}
```

拷贝构造函数看上去有点奇怪, 因为需要从源 `Spreadsheet` 复制底层的 `Impl`。由于拷贝构造函数采用一个指向 `Impl` 的引用而不是指针, 因此为了获取对象本身, 必须对 `m_Impl` 指针解除引用, 以得到它的对象。

`Spreadsheet` 赋值运算符必须采用类似方式传递到底层的 `Impl` 的赋值:

```
Spreadsheet& Spreadsheet::operator=(const Spreadsheet& rhs)
{
    *m_impl = *rhs.m_impl;
    return *this;
}
```

赋值运算符的第一行看上去有点奇怪。`Spreadsheet` 赋值运算符需要传递对 `Impl` 赋值运算符的调用, 而这个运算符只有在复制直接对象时才会执行。通过对 `m_Impl` 指针解除引用, 会强制使用直接对象赋值, 从而调用 `Impl` 的赋值运算符。

`swap()` 函数用于交换单独的数据成员:

```
void Spreadsheet::swap(Spreadsheet& other) noexcept
{
    std::swap(m_impl, other.m_impl);
}
```

真正将接口和实现分离的技术功能强大。尽管开始时有点笨拙, 但是一旦适应这种技术, 就会觉得这么做很自然。然而, 在多数工作环境中这并不是常规做法, 因此这么做会遇到来自同事的一些阻力。支持这种方法最有力的论据不是将接口分离的美感, 而是类的实现改变后大幅缩短构建时间。一个类不使用 `pimpl idiom` 时, 对实现类的更改将导致漫长的构建过程。例如, 给类定义增加数据成员时, 将触发使用这个类的其他所有源文件的重新构建。而使用 `pimpl idiom`, 可以修改实现类的定义, 只要公共接口类保持不变, 就不会触发长时间的构建过程。

**注意：**

使用稳定接口类，可缩短构建时间。

为将实现与接口分离，另一种方法是使用抽象接口以及实现该接口的实现类，抽象接口是只有纯虚方法(pure virtual method)的接口。第 10 章将讨论抽象接口。

## 9.9 本章小结

利用本章和第 8 章介绍的工具，可编写稳定的、设计良好的类，并且有效地使用对象。

对象的动态内存分配遇到了新挑战：需要实现析构函数、拷贝构造函数、拷贝赋值运算符、移动构造函数和移动赋值运算符，它们能够合理地复制、移动和释放内存。为阻止赋值和按值传递，可显式地删除拷贝构造函数和赋值运算符。可以使用“复制和交换”惯用方法实现拷贝赋值运算符，你还学习了零规则。

本章介绍了不同类型的数据成员，包括 static 数据成员、const 数据成员、const 引用数据成员和 mutable 成员。还讨论了 static、inline 和 const 方法，以及方法重载和默认参数。本章随后讲解了嵌套类的定义、友元类、友元函数和友元方法。

本章接下来讲述了运算符重载，介绍如何重载算术运算符和比较运算符，这些重载的运算符都可作为全局友元函数，也可作为类方法。你还学习了如何在自己的类中添加对 C++20 的三向比较运算符的支持。

最后，学习了如何分离接口类和实现类，从而将抽象发挥到极致。

熟悉了面向对象编程语言后，就可以考虑使用继承，第 10 章将讲述这一内容。

## 9.10 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，然后再从网站上查看解决方案。

**练习 9-1** 以练习 8-3 中 Person 类的实现为例，对其进行调整，以你所能想到的最佳方式传递字符串。此外，向其添加移动构造函数和移动赋值运算符。在这两种方法中，向控制台写入消息，以便跟踪它们何时被调用。为了实现移动方法以及改进其他方法的实现从而避免代码重复，添加所需的任何方法。修改 main() 以测试你的方法。

**练习 9-2** 基于练习 8-4 中的 Person 类，添加对所有 6 个比较运算符的支持，以比较两个 Person 对象。尝试用最少的代码实现此支持。通过在 main() 中执行各种比较来测试你的实现。Person 是如何排序的？是基于名字，还是基于姓氏，还是基于名字和姓氏的组合？

**练习 9-3** 在 C++20 之前，添加对所有 6 个比较运算符的支持需要更多的代码行。从练习 9-2 中的 Person 类开始，删除 operator<=>，并添加必要的代码，以实现所有比较操作符来比较两个 Person 对象，而不使用<=>。执行与练习 9-2 相同的测试集。

**练习 9-4** 在本练习中，你将练习编写稳定的接口。将练习 8-4 中的 Person 类拆分为一个稳定的公共接口类和一个单独的实现类。

# 第 10 章

## 揭秘继承技术

### 本章内容

- 如何通过继承扩展类
- 如何使用继承重用代码
- 基类和派生类如何交互
- 如何使用继承实现多态性
- 如何使用多重继承
- 如何处理继承中的罕见问题
- 如何将一种类型强制转换为另一种类型

如果没有继承，类只是具有一些相关行为的数据结构。这只是对过程语言的一大改进，而继承则开辟了完全不同的新天地。通过继承，可在已有类的基础上创建新类。这样，类就成为可重用和可扩展的组件。本章将介绍各种利用继承功能的方法。通过学习继承的语法，并最大限度地利用一些继承的复杂技术。

本章中多态性相关的部分大量借鉴了第 8 章“熟悉类和对象”和第 9 章“精通类与对象”中讲述的电子表格示例。本章还涉及第 5 章介绍的面向对象的方法论，如果没有阅读这一章，不熟悉继承的理论，应该在学习本章内容之前回顾第 5 章的内容。

### 10.1 使用继承构建类

第 5 章提到，“是一个”关系是实际对象在继承层次中的存在模式。在编程时，如果要基于某个类编写另一个类，或者对某个类进行少量修改，都会涉及这个模式。完成这一目标的方法之一是将代码从一个类复制出来，然后粘贴到另一个类中。之后修改新类中相关部分的代码，就可以创建一个与原始类稍有不同的新类。但是这种方法会让 OOP 程序员感到不快，原因如下：

- 修订原始类的 bug 不会影响新类，因为两个类包含完全独立的代码。
- 编译器不知道这两个类之间的关系，因此不具备多态性(请参阅第 5 章)——这两个类不是同一事物的不同变种。

- 这种方法没有建立真正的“是一个”关系。新类与原始类非常相似的原因是因为共享了代码，而不是因为它们是同一类对象。
- 原始代码可能无法获得。其可能存在于预编译的二进制格式的文件中，因此不可能复制和粘贴这些代码。

不要惊讶，C++为定义真正的“是一个”关系提供了内建支持。C++中“是一个”关系的特征将在下面介绍。

### 10.1.1 扩展类

当使用C++编写类定义时，可以告诉编译器，该类继承、派生或扩展了一个已有的类。通过这种方式，该类将自动包含原始类的数据成员和方法；原始类称为父类(parent class)、基类或超类(superclass)。扩展已有类可以使该类(现在称为派生类或子类)只描述与父类不同的那部分内容。

在C++中，为扩展一个类，可在定义类时指定要扩展的类。为说明继承的语法，此处使用了名为Base和Derived的两个类。不要担心——后面还有许多更有趣的示例。首先考虑Base类的定义：

```
class Base
{
public:
    void someMethod() {}
protected:
    int m_protectedInt { 0 };
private:
    int m_privateInt { 0 };
};
```

如果要构建一个从Base类继承的新类Derived，应该使用下面的语法：

```
class Derived : public Base
{
public:
    void someOtherMethod() {}
};
```

Derived本身就是一个完整的类，这个类只是刚好共享了Base类的特性而已。现在不用担心public关键字——本章后面将解释其含义。图10-1显示了Derived类与Base类之间的简单关系。可以像声明其他对象那样声明Derived类型的对象，甚至可以定义Derived的派生类作为第三个类，从而形成类的链条，如图10-2所示。

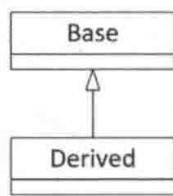


图10-1 Derived类与Base类之间的简单关系

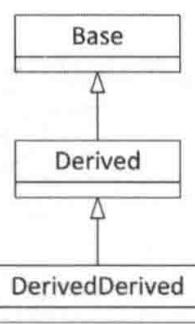
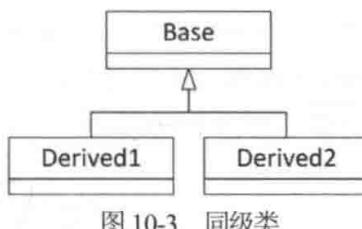


图10-2 形成类的链条

Derived不一定是Base唯一的派生类。其他类也可以是Base的派生类，这些类是Derived的同级类(sibling)，如图10-3所示。



### 1. 客户对继承的看法

对于客户或代码的其他部分而言，Derived 类型的对象仍然是 Base 对象，因为 Derived 类继承自 Base 类。这意味着 Base 类中所有 public 方法和数据成员，以及 Derived 类中所有 public 方法和数据成员都是可用的。

在调用某个方法时，使用派生类的代码不需要知道是继承链中的哪个类定义了这个方法。例如，下面的代码调用了 Derived 对象中的两个方法，而其中一个方法是在 Base 类中定义的。

```

Derived myDerived;
myDerived.someMethod();
myDerived.someOtherMethod();

```

要知道继承的运行方式是单向的，这一点很重要。Derived 类与 Base 类具有明确的关系，但是 Base 类并不知道 Derived 类的任何信息。这意味着 Base 类型的对象不支持 Derived 类的 public 方法和数据成员，因为 Base 类不是 Derived 类。

下面的代码将无法编译，因为 Base 类不包含名为 someOtherMethod() 的 public 方法。

```

Base myBase;
myBase.someOtherMethod(); // Error! Base doesn't have a someOtherMethod().

```

#### 注意：

从其他代码的观点看，一个对象既属于定义它的类，又属于所有的基类。

指向某个对象的指针或引用可以指向声明类的对象，也可以指向其任意派生类的对象。本章后面将详细介绍这一复杂的主题。此时需要理解的概念是，指向 Base 对象的指针可以指向 Derived 对象，对于引用也是如此。客户仍然只能访问 Base 类的方法和数据成员，但是通过这种机制，任何操作 Base 对象的代码都可以操作 Derived 对象。

例如，下面的代码可以正常编译并运行，尽管看上去好像类型并不匹配。

```
Base* base { new Derived{} }; // Create Derived, store it in Base pointer.
```

然而，不能通过 Base 类型的指针调用 Derived 类的方法。下面的代码无法运行：

```
base->someOtherMethod();
```

编译器会报错，因为尽管对象是 Derived 类型，并且定义了 someOtherMethod() 方法，但编译器只是将它看成 Base 类型，而 Base 类中没有定义 someOtherMethod() 方法。

### 2. 从派生类的角度分析继承

对于派生类自身而言，其编写方式或行为并没有改变。仍然可以在派生类中定义方法和数据成员，就像这是一个普通的类。前面 Derived 类的定义中声明了一个名为 someOtherMethod() 的方法。

因此 Derived 类增加了一个额外的方法，从而扩展了 Base 类。

派生类可访问基类中声明的 public、protected 方法和数据成员，就好像这些方法和数据成员是派

生类自己的，因为从技术上讲，它们属于派生类。例如，Derived 类中 someOtherMethod()的实现可以使用在 Base 类中声明的数据成员 m\_protectedInt。下面的代码显示了这一实现，访问基类的数据成员或方法与访问派生类中的数据成员和方法并无不同之处。

```
void Derived::someOtherMethod()
{
    cout << "I can access base class data member m_protectedInt." << endl;
    cout << "Its value is " << m_protectedInt << endl;
}
```

如果类将数据成员或方法声明为 protected，派生类就可以访问它们；如果声明为 private，派生类就不能访问。下面的 someOtherMethod()实现将无法编译，因为派生类试图访问基类的 private 数据成员。

```
void Derived::someOtherMethod()
{
    cout << "I can access base class data member m_protectedInt." << endl;
    cout << "Its value is " << m_protectedInt << endl;
cout << "The value of m_privateInt is " << m_privateInt << endl; // Error!
}
```

`private` 访问说明符可以控制派生类与基类的交互方式。建议将所有数据成员都默认声明为 `private`，如果希望任何代码都可以访问这些数据成员，可以提供 `public` 的 getter 和 setter。如果仅希望派生类访问它们，就可以提供 `protected` 的 getter 和 setter。把数据成员默认设置为 `private` 的原因是，这会提供最高级别的封装，这意味着可以改变数据的表示方式，而保持 `public` 或 `protected` 的接口不变。另外，不直接访问数据成员，也可以在 `public` 或 `protected` 的 setter 中方便地添加对输入数据的检查。方法也应默认设置为 `private`，只有需要公开的方法才设置为 `public`，只有派生类需要访问的方法才设置为 `protected`。

#### 注意：

从派生类的观点看，基类的所有 `public`、`protected` 数据成员和方法都是可用的。

表 10-1 总结了所有 3 个访问说明符的含义。

表 10-1 访问说明符的含义

访问说明符	含义	使用时机
public	任何代码都可以调用对象的 <code>public</code> 的成员函数或访问该对象的 <code>public</code> 的数据成员	希望客户使用的行为(方法)。 <code>private</code> 和 <code>protected</code> 的成员的访问方法(getter 和 setter)
protected	类的任何成员函数都可以调用 <code>protected</code> 的成员函数和访问 <code>protected</code> 的数据成员。 派生类的成员函数可以访问基类的 <code>protected</code> 的成员	不希望客户使用的“帮助”方法
private	只有类的成员函数才能调用 <code>private</code> 的成员函数和访问 <code>private</code> 的数据成员。 派生类中的成员函数不能访问基类的 <code>private</code> 的成员	默认情况下，所有内容都应该是 <code>private</code> ，特别是数据成员。 如果只允许派生类访问它们，则可以提供受保护的 getter 和 setter 方法；如果希望客户访问它们，则可以提供 <code>public</code> 的 getter 和 setter 方法

### 3. 禁用继承

C++ 允许将类标记为 final，这意味着继承这个类会导致编译错误。将类标记为 final 的方法是直接在类名的后面使用 final 关键字。比如，如果一个类试图从下面的 Foo 类继承，这会导致编译错误。

```
class Foo final { };
```

## 10.1.2 重写方法

从某个类继承的主要原因是为了添加或替换功能。Derived 类定义在父类的基础上添加了功能，提供了额外的 someOtherMethod() 方法。另一个方法 someMethod() 从 Base 类继承而来，这个方法在派生类中的行为与在基类中的相同。在许多情况下，可能需要替换或重写某个方法来修改类的行为。

### 1. 关键字 virtual

在 C++ 中，重写(override)方法有一点别扭，因为必须使用关键字 virtual。只有基类中声明为 virtual 的方法才能被派生类正确地重写。关键字 virtual 出现在方法声明的开头，下面显示了 Base 类的修改版本。

```
class Base
{
public:
    virtual void someMethod() {}
protected:
    int m_protectedInt { 0 };
private:
    int m_privateInt { 0 };
};
```

派生类也是如此。如果想进一步在派生类中重写它们，相应的方法也应该标记为 virtual。

```
class Derived : public Base
{
public:
    virtual void someOtherMethod();
};
```

### 2. 重写方法的语法

为了重写某个方法，需要在派生类的定义中重新声明这个方法，就像在基类中声明的那样，但是需要添加关键字 override，并从派生类中删除关键字 virtual。

比如，Base 类包含了一个 someMethod() 方法，someMethod() 方法的定义如下：

```
void Base::someMethod()
{
    cout << "This is Base's version of someMethod()." << endl;
}
```

注意，在方法定义中，不需要在方法名前重复使用关键字 virtual。

如果希望在 Derived 类中提供 someMethod() 的新定义，首先应该在 Derived 类的定义中添加这个方法，示例如下。注意，缺失的关键字 virtual 和新增的关键字 override：

```
class Derived : public Base
{
public:
```

```

void someMethod() override; // Overrides Base's someMethod()
virtual void someOtherMethod();
};

```

someMethod()方法的新定义与 Derived 类的其他方法一并给出：

```

void Derived::someMethod()
{
    cout << "This is Derived's version of someMethod()." << endl;
}

```

如果需要的话，可以在重写方法的前面添加关键字 virtual，但这样做是多余的。示例如下：

```

class Derived : public Base
{
public:
    virtual void someMethod() override; // Overrides Base's someMethod()
};

```

一旦将方法或析构函数标记为 virtual，它们在所有派生类中就一直是 virtual，即使在派生类中删除了 virtual 关键字，也同样如此。

### 3. 客户对重写方法的看法

经过前面的改动后，其他代码仍可用先前的方法调用 someMethod()。和之前一样，可以用 Base 或 Derived 类的对象调用这个方法。然而，现在 someMethod() 的行为将根据对象所属类的不同而变化。

比如，下面的代码与先前一样可以运行，调用 Base 版本的 someMethod()。

```

Base myBase;
myBase.someMethod(); // Calls Base's version of someMethod().

```

代码的输出为：

```
This is Base's version of someMethod().
```

如果声明一个 Derived 类对象，将自动调用派生类版本的 someMethod()：

```

Derived myDerived;
myDerived.someMethod(); // Calls Derived's version of someMethod()

```

代码的输出为：

```
This is Derived's version of someMethod().
```

Derived 类对象的其他方面维持不变。从 Base 类继承的其他方法仍然保持 Base 类提供的定义，除非在 Derived 类中显式地重写这些方法。

如前所述，指针或引用可指向某个类或其派生类的对象。对象本身“知道”自己所属的类，因此只要这个方法声明为 virtual，就会自动调用对应的方法。比如，如果一个对 Base 对象的引用实际引用的是 Derived 对象，调用 someMethod() 实际上会调用派生类版本，示例如下。如果在基类中省略了 virtual 关键字，重写功能将无法正确运行。

```

Derived myDerived;
Base& ref { myDerived };
ref.someMethod(); // Calls Derived's version of someMethod()

```

记住，即使基类的引用或指针知道这实际上是一个派生类，也无法访问没有在基类中定义的，定义在派生类中的方法或成员。下面的代码无法编译，因为 Base 引用了在基类中没有定义的

someOtherMethod()方法。

```
Derived myDerived;
Base& ref { myDerived };
myDerived.someOtherMethod(); // This is fine.
ref.someOtherMethod(); // Error
```

非指针或非引用对象无法正确处理派生类的特征信息。可将 Derived 对象转换为 Base 对象，或将 Derived 对象赋值给 Base 对象，因为 Derived 对象也是 Base 对象。然而，此时这个对象将丢失派生类的所有信息。

```
Derived myDerived;
Base assignedObject { myDerived }; // Assigns a Derived to a Base.
assignedObject.someMethod(); // Calls Base's version of someMethod()
```

为记住这个看上去有点奇怪的行为，可考虑对象在内存中的状态。将 Base 对象当作占据了一块内存的盒子。Derived 对象是稍微大一点的盒子，因为它拥有 Base 对象的一切，还添加了一点内容。对于指向 Derived 对象的引用或指针，这个盒子并没有变——只是可以用新的方法访问它。然而，如果将 Derived 对象转换为 Base 对象，就会为了适应较小的盒子而扔掉 Derived 类中全部的“独有特征”。

#### 注意：

当基类的指针或引用指向派生类对象时，派生类保留其数据成员和重写的方法。但是通过类型转换将派生类对象转换为基类对象时，就会丢失其独有特征。重写方法和派生类数据的丢失称为截断(slicing)。

#### 4. override 关键字

override 关键字的使用是可选的，但强烈推荐使用。如果没有 override 关键字，可能会偶然创建一个新的虚方法，而不是重写基类的方法。考虑下面的 Base 和 Derived 类，其中 Derived 类正确重写了 someMethod()，但没有使用 override 关键字。

```
class Base
{
public:
    virtual void someMethod(double d);
};

class Derived : public Base
{
public:
    virtual void someMethod(double d);
};
```

可通过引用调用 someMethod()，如下所示。

```
Derived myDerived;
Base& ref { myDerived };
ref.someMethod(1.1); // Calls Derived's version of someMethod()
```

上述代码能正确地调用 Derived 类重写的 someMethod()。现在假定重写 someMethod() 时，使用整数(而不是双精度数)作为参数，如下所示。

```
class Derived : public Base
{
```

```

public:
    virtual void someMethod(int i);
};

```

这段代码没有重写 Base 类的 someMethod(), 而是创建了一个新的虚方法。如果试图像下面的代码这样通过引用调用 someMethod(), 将会调用 Base 类的 someMethod()而不是 Derived 类中定义的那个方法。

```

Derived myDerived;
Base& ref { myDerived };
ref.someMethod(1.1); // Calls Base's version of someMethod()

```

如果只修改了 Base 类, 但却忘记更新所有派生类, 就会发生这类问题。比如, 或许 Base 类的第一个版本中有一个以整数作为参数的 someMethod()方法。然后在 Derived 派生类中重写了 someMethod()方法, 仍然以整数作为参数。后来发现 Base 类中的 someMethod()方法需要一个双精度数而不是整数, 因此更新了 Base 类中的 someMethod()。极有可能的是, 此时你可能忘记更新派生类中的 someMethod(), 让它们接收双精度数而不是整数。由于忘记了这一点, 实际上就是创建了一个新的虚方法, 而不是正确地重写基类的这个方法。

可以使用 override 关键字避免这种情况, 如下所示。

```

class Derived : public Base
{
public:
    void someMethod(int i) override;
};

```

Derived 类的定义将导致编译错误, 因为 Derived 类中的 override 关键字表明, 将重写 Base 类的 someMethod()方法, 但 Base 类中的 someMethod()方法只接收双精度数, 而不接收整数。

重命名基类中的某个方法, 但忘记重命名派生类中的重写方法时, 就会出现上述“不小心创建了新方法, 而不是正确重写方法”的问题。

#### 注意:

要想重写基类方法, 始终在方法上使用 override 关键字。

### 5. virtual 的真相

如果方法不是 virtual, 也可以试着重写这个方法, 但是这样做会导致微妙的错误。

#### 隐藏而不是重写

下面的代码显示了一个基类和一个派生类, 每个类都有一个方法。派生类试图重写基类的方法, 但是在基类中没有将这个方法声明为 virtual。

```

class Base
{
public:
    void go() { cout << "go() called on Base" << endl; }
};

class Derived : public Base
{
public:
    void go() { cout << "go() called on Derived" << endl; }
};

```

试着用 Derived 对象调用 go() 方法好像没有问题。

```
Derived myDerived;
myDerived.go();
```

正如预期的那样，这个调用的结果是“go() called on Derived”。然而，由于这个方法不是 virtual，因此实际上没有被重写。相反，Derived 类创建了一个新的方法，名称也是 go()，这个方法与 Base 类的 go() 方法完全没有关系。为证实这一点，只需要用 Base 指针或引用调用这个方法。

```
Derived myDerived;
Base& ref { myDerived };
ref.go();
```

你可能希望输出是“go() called on Derived”，但实际上，输出是“go() called on Base”。这是因为 ref 变量是一个 Base 类型的引用，因为此处省略了 virtual 关键字。当调用 go() 方法时，只是执行了 Base 类中的 go() 方法。由于不是虚方法，因此不需要考虑派生类是否重写了这个方法。

### 警告：

试图重写非虚方法将会“隐藏”基类定义的方法，并且重写的这个方法只能在派生类环境中使用。

### 如何实现 virtual

为理解如何避免隐藏方法，需要了解 virtual 关键字的真正作用。C++ 在编译类时，会创建一个包含类中所有方法的二进制对象。在非虚情况下，将控制交给正确方法的代码是硬编码，此时会根据编译期的类型调用对应的方法。这称为静态绑定(static binding)，也称为早绑定(early binding)。

如果方法声明为 virtual，会使用名为虚表(vtable)的特定内存区域来调用正确的实现。每个具有一个或多个虚方法的类都有一张虚表，这种类的每个对象都包含指向虚表的指针，这个虚表包含指向虚方法实现的指针。通过这种方法，当使用某个对象调用方法时，指针也进入虚表，然后根据实际的对象类型在运行期执行正确版本的方法。这称为动态绑定(dynamic binding)或晚绑定(late binding)。

为更好地理解虚表是如何实现方法的重写的，考虑下面的 Base 和 Derived 类。

```
class Base
{
public:
    virtual void func1();
    virtual void func2();
    void nonVirtualFunc();
};

class Derived : public Base
{
public:
    void func2() override;
    void nonVirtualFunc();
};
```

对于这个示例，考虑下面的两个实例：

```
Base myBase;
Derived myDerived;
```

图 10-4 显示了这两个实例虚表的高级视图。myBase 对象包含了指向虚表的一个指针，虚表有两项，一项是 func1()，另一项是 func2()。这两项指向 Base::func1() 和 Base::func2() 的实现。

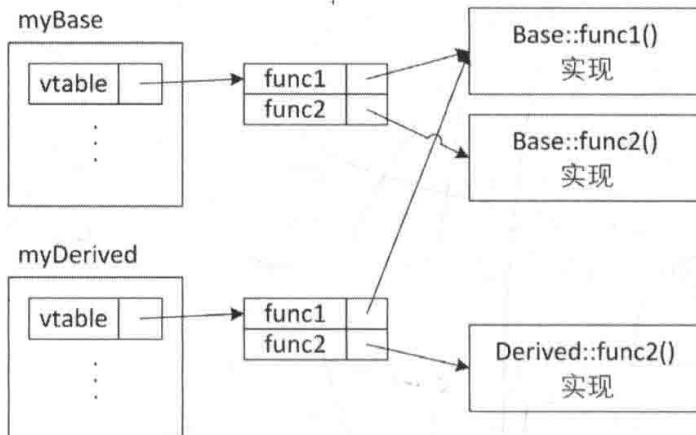


图 10-4 两个实例虚表的高级视图

`myDerived` 也包含指向虚表的一个指针，这个虚表也包含两项，一项是 `func1()`，另一项是 `func2()`。`myDerived` 虚表的 `func1()` 项指向 `Base::func1()`，因为 `Derived` 类没有重写 `func1()`；但是 `myDerived` 虚表的 `func2()` 项指向 `Derived::func2()`。

注意两个虚表都不包含用于 `nonVirtualFunc()` 方法的项，因为该方法没有设置为 `virtual`。

### 使用 `virtual` 的理由

在一些语言中(比如 Java)，所有的方法都默认声明为 `virtual`，因此，这些方法可以被正确的重写。但 C++ 并不是这么做的。有关无所不在地使用 `virtual` 的争论，以及首先创建该关键字的原因，都与虚表的开销有关。要调用虚方法，程序需要执行一项附加操作，即对指向要执行的适当代码的指针解除引用。在多数情况下，这样做会轻微地影响性能，但是 C++ 的设计者认为，最好让程序员决定是否有必要影响性能。如果方法永远不会重写，就没必要将其声明为 `virtual`，从而影响性能。然而对于当今的 CPU 而言，对性能的影响可以用十亿分之一秒来度量，将来的 CPU 会使时间进一步缩短。在多数应用程序中，很难察觉到使用虚方法和不使用虚方法所带来的性能上的差别。

但在某些情况下，性能开销确实不小，需要避免。比如，假设 `Point` 类有一个虚方法。如果另一个数据结构存储着数百万个甚至数十亿个 `Point` 对象，在每个 `Point` 对象上调用虚方法将带来极大的开销。此时，最好避免在 `Point` 类中使用虚方法。

`virtual` 对于每个对象的内存使用也有轻微影响。除了方法的实现之外，每个对象还需要一个指向虚表的指针，这个指针会占用一点空间。绝大多数情况下，这都不是问题。但有时并非如此。再看一下 `Point` 类以及存储数十亿个 `Point` 对象的容器。此时，附带的内存开销将会很大。

### 析构函数的需求

析构函数应该都声明为 `virtual`。原因是，如果析构函数未声明为 `virtual`，很容易在销毁对象时不释放内存。唯一允许不把析构函数声明为 `virtual` 的例外情况是，类被标记为 `final`。

例如，派生类使用的内存由构造函数动态分配，在析构函数中释放。如果不调用析构函数，这块内存将无法释放。类似地，如果派生类具有一些成员，这些成员在类的实例销毁时自动删除，如 `std::unique_ptr`，那么如果从未调用析构函数，将不会删除这些成员。

如下面的代码所示，如果析构函数不是 `virtual`，很容易欺骗编译器忽略析构函数的调用。

```
class Base
{
public:
    Base() {}
    ~Base() {}
};
```

```

class Derived : public Base
{
public:
    Derived()
    {
        m_string = new char[30];
        cout << "m_string allocated" << endl;
    }

    ~Derived()
    {
        delete[] m_string;
        cout << "m_string deallocated" << endl;
    }

private:
    char* m_string;
};

int main()
{
    Base* ptr { new Derived() }; // m_string is allocated here.
    delete ptr; // ~Base is called, but not ~Derived because the destructor
                 // is not virtual!
}

```

从输出可以看到，从未调用 Derived 对象的析构函数：

```
m_string allocated
```

实际上，在上面的代码中，`delete` 调用的行为未在标准中定义。在这样的不明确情形中，C++ 编译器会随意做事，但大多数编译器只是调用基类中的析构函数，而非派生类中的析构函数。

#### 注意：

如果在析构函数中什么都不做，只想把它设置为 `virtual`，可显式地设置 “`= default`”，例如：

```

class Base
{
public:
    virtual ~Base() = default;
};

```

注意从 C++11 开始，如果类具有用户声明的析构函数，就不会生成复制构造函数和拷贝赋值运算符。基本上，一旦有了用户声明的析构函数，C++ 的 5 法则<sup>1</sup>就会开始执行。在此类情况下，如果仍然需要编译器生成的复制构造函数、拷贝赋值运算符、移动构造函数和移动赋值运算符，可将它们显式设置为默认(`= default`)。为保持简洁，本章的这个示例没有这么做。

#### 警告：

除非有特别原因，或者类被标记为 `final`，否则强烈建议将所有方法(包括析构函数，构造函数除外)声明为 `virtual`。构造函数不需要，也无法声明为 `virtual`，因为在创建对象时，总会明确地指定类。

<sup>1</sup> 译者注：由于拷贝控制操作是由 3 个特殊的成员函数(复制构造函数、复制赋值运算符和析构函数)完成的，因此称此为“C++ 三法则”。在较新的 C++11 标准中，为了支持移动语义，又增加了移动构造函数和移动赋值运算符，这样共有 5 个特殊的成员函数，所以又称为“C++ 五法则”。

也就是说，“三法则”是针对较旧的 C++8、9 标准说的，“五法则”是针对较新的 C++11 标准说的。

## 6. 禁用重写

除了将实体类标记为 final, C++还允许将方法标记为 final。这意味着无法在派生类中重写这个方法。比如，在派生类中试图重写 someMethod()方法将导致编译错误。

```
class Base
{
public:
    virtual ~Base() = default;
    virtual void someMethod();
};

class Derived : public Base
{
public:
    void someMethod() override final;
};

class DerivedDerived : public Derived
{
public:
    void someMethod() override; // Compilation error.
};
```

## 10.2 使用继承重用代码

熟悉了继承的基本语法后，下面解释为什么继承是 C++语言中的重要特性。继承是利用已有代码来避免重复造轮子，本节给出了使用继承重用代码的实际程序。

### 10.2.1 WeatherPrediction 类

假定要编写一个简单的天气预报程序，同时给出华氏温度和摄氏温度。天气预报可能超出了作为程序员的研究领域，因此可以使用一个第三方的类库，这个类库根据当前温度和火星与木星之间的距离(这荒谬吗？不，是有点道理的)预测天气。为保护天气预报算法的知识产权，将第三方的包作为已编译的库分布，但还是可以看到类的定义。WeatherPrediction 类的定义如下：

```
// Predicts the weather using proven new-age techniques given the current
// temperature and the distance from Jupiter to Mars. If these values are
// not provided, a guess is still given but it's only 99% accurate.
export class WeatherPrediction
{
public:
    // Virtual destructor
    virtual ~WeatherPrediction();
    // Sets the current temperature in Fahrenheit
    virtual void setCurrentTempFahrenheit(int temp);
    // Sets the current distance between Jupiter and Mars
    virtual void setPositionOfJupiter(int distanceFromMars);
    // Gets the prediction for tomorrow's temperature
    virtual int getTomorrowTempFahrenheit() const;
    // Gets the probability of rain tomorrow. 1 means
    // definite rain. 0 means no chance of rain.
    virtual double getChanceOfRain() const;
    // Displays the result to the user in this format:
    // Result: x.xx chance. Temp. xx
```

```

    virtual void showResult() const;
    // Returns a string representation of the temperature
    virtual std::string getTemperature() const;
private:
    int m_currentTempFahrenheit { 0 };
    int m_distanceFromMars { 0 };
};

```

注意这个类将所有方法标记为 `virtual`，因为这个类假定这些方法可能在派生类中重写。

这个类解决了大部分问题。然而与多数情况一样，它与实际的需求并不完全吻合。首先，所有的温度都以华氏温度给出，程序还需要处理摄氏温度。其次，`showResult()`方法显示结果的方式可能并不是想要的。

## 10.2.2 在派生类中添加功能

第 5 章讲述继承时，首先描述的技巧就是添加功能。基本上该程序需要一个类似于 `WeatherPrediction` 的类，但还需要添加一些附加功能。使用继承重用代码听起来是个好主意。首先定义一个新类 `MyWeatherPrediction`，这个类从 `WeatherPrediction` 类继承。

```

import weather_prediction;

export class MyWeatherPrediction : public WeatherPrediction
{
};

```

前面的类定义可以成功编译。`MyWeatherPrediction` 类已经可以替代 `WeatherPrediction` 类。这个类可提供相同的功能，但没有新功能。开始修改时，要在类中添加摄氏温度的信息。这里有点小问题，因为不知道这个类的内部在做什么。如果所有的内部计算都使用华氏温度，如何添加对摄氏温度的支持呢？方法之一是采用派生类作为用户(可以使用两种温度)和基类(只理解华氏温度)之间的中间接口。

支持摄氏温度的第一步是添加新方法，允许客户用摄氏温度(而不是华氏温度)设置当前的温度，从而获取明天以摄氏温度(而不是华氏温度)表示的天气预报。还需要包含在摄氏温度和华氏温度之间转换的私有辅助方法。这些方法可以是静态方法，因为它们对于类的所有实例都相同。

```

export class MyWeatherPrediction : public WeatherPrediction
{
public:
    virtual void setCurrentTempCelsius(int temp);
    virtual int getTomorrowTempCelsius() const;
private:
    static int convertCelsiusToFahrenheit(int celsius);
    static int convertFahrenheitToCelsius(int fahrenheit);
};

```

新方法遵循与父类相同的命名约定。记住，从其他代码的角度看，`MyWeatherPrediction` 对象具有 `MyWeatherPrediction` 和 `WeatherPrediction` 类定义的所有功能。采用父类的命名约定可以提供前后一致的接口。

我们把摄氏温度/华氏温度转换方法的实现作为练习留给读者——这是一种乐趣。另外两个方法更有趣。为了用摄氏温度设置当前温度，首先需要转换温度，其次将其以父类可以理解的单位传递给父类。

```
void MyWeatherPrediction::setCurrentTempCelsius(int temp)
{
    int fahrenheitTemp { convertCelsiusToFahrenheit(temp) };
    setCurrentTempFahrenheit(fahrenheitTemp);
}
```

可以看出，执行温度转换后，这个方法调用了基类中的已有功能。同样，getTomorrowTempCelsius()的实现使用了父类的已有功能，获取华氏温度，但是在返回结果之前将其转换为摄氏温度。

```
int MyWeatherPrediction::getTomorrowTempCelsius() const
{
    int fahrenheitTemp { getTomorrowTempFahrenheit() };
    return convertFahrenheitToCelsius(fahrenheitTemp);
}
```

这两个新方法都有效地重用了父类，因为它们以某种方式“封装”了类已有的功能，并提供了使用这些功能的新接口。

还可添加与父类已有功能无关的全新功能。比如，可以添加一个方法，从 Internet 获取其他天气预报，或添加一个方法，根据天气预报给出建议的活动。

### 10.2.3 在派生类中替换功能

与派生类相关的另一个主要技巧是替换已有的功能。WeatherPrediction 类中的 showResult()方法急需修改。MyWeatherPrediction 类可以重写这个方法，以替换原始实现中的行为。

新的 MyWeatherPrediction 类定义如下所示：

```
export class MyWeatherPrediction : public WeatherPrediction
{
public:
    virtual void setCurrentTempCelsius(int temp);
    virtual int getTomorrowTempCelsius() const;
    void showResult() const override;
private:
    static int convertCelsiusToFahrenheit(int celsius);
    static int convertFahrenheitToCelsius(int fahrenheit);
};
```

下面给出了一个对用户友好的新实现：

```
void MyWeatherPrediction::showResult() const
{
    cout << format("Tomorrow's temperature will be {} degrees Celsius ({} degrees Fahrenheit)",
        getTomorrowTempCelsius(), getTomorrowTempFahrenheit()) << endl;
    cout << format("Chance of rain is {}%", getChanceOfRain() * 100) << endl;
    if (getChanceOfRain() > 0.5) { cout << "Bring an umbrella!" << endl; }
}
```

对于使用这个类的客户而言，就像旧版本的 showResult()不曾存在一样。只要对象是一个 MyWeatherPrediction 对象，就会调用新版本的方法。这些改动的结果是，MyWeatherPrediction 表现得像一个新类，具有适应更具体目标的新功能。另外，由于利用了基类的已有功能，因此这个类不需要太多代码。

## 10.3 利用父类

编写派生类时，需要知道父类和派生类之间的交互方式。创建顺序、构造函数链和类型转换都是潜在的 bug 来源。

### 10.3.1 父类构造函数

对象并不是突然建立起来的，创建对象时必须同时创建父类和包含于其中的对象。C++ 定义了如下的创建顺序：

- (1) 如果某个类具有基类，则执行基类的默认构造函数。除非在 ctor-initializer 中调用了基类构造函数，此时调用这个构造函数而不是默认构造函数。
- (2) 类的非静态数据成员按照声明的顺序创建。
- (3) 执行该类的构造函数。

可递归使用这些规则。如果类有祖父类，祖父类就在父类之前初始化，以此类推。下面的代码显示了创建顺序。代码正确执行时输出结果为 123。

```
class Something
{
public:
    Something() { cout << "2"; }
};

class Base
{
public:
    Base() { cout << "1"; }
};

class Derived : public Base
{
public:
    Derived() { cout << "3"; }
private:
    Something m_dataMember;
};

int main()
{
    Derived myDerived;
}
```

创建 myDerived 对象时，首先调用 Base 类的构造函数，输出字符串"1"。随后，初始化 m\_dataMember，调用 Something 类的构造函数，输出字符串"2"。最后，调用 Derived 类的构造函数，输出字符串"3"。

注意，Base 类的构造函数是自动调用的。C++ 将自动调用父类的默认构造函数(如果存在的话)。如果父类的默认构造函数不存在，或者存在默认构造函数但希望使用其他构造函数，可在构造函数初始化器(constructor initializer)中像初始化数据成员那样链接(chain)构造函数。比如，下面的代码显示了没有默认构造函数的 Base 版本。相关版本的 Derived 必须显式地告诉编译器如何调用 Base 构造函数，否则代码将无法编译。

```

class Base
{
public:
    Base(int i){}
};

class Derived : public Base
{
public:
    Derived() : Base(7) /* Other Derived's other initialization ... */
};

```

在前面的代码中，Derived 构造函数向 Base 构造函数传递了固定值(7)。当然，如果 Derived 构造函数也可以传递一个变量：

```
Derived::Derived(int i) : Base(i) {}
```

从派生类向基类传递构造函数的参数很正常，毫无问题，但是无法传递数据成员。如果这么做，代码可以编译，但是记住在调用基类构造函数之后才会初始化数据成员。如果将数据成员作为参数传递给父类构造函数，数据成员不会初始化。

#### 警告：

虚方法的行为在构造函数中是不同的，如果派生类重写了基类中的虚方法，从基类构造函数中调用虚方法，就会调用虚方法的基类实现而不是派生类中的重写版本。

### 10.3.2 父类的析构函数

由于析构函数没有参数，因此始终可自动调用父类的析构函数。析构函数的调用顺序刚好与构造函数相反：

- (1) 调用类的析构函数。
- (2) 销毁类的数据成员，销毁顺序与创建的顺序相反。
- (3) 如果有父类，调用父类的析构函数。

也可递归使用这些规则。链的最底层成员总是第一个被销毁。下面的代码是在前面的示例中加入了析构函数。所有析构函数都声明为 virtual，这一点非常重要！代码执行时将输出"123321"。

```

class Something
{
public:
    Something() { cout << "2"; }
    virtual ~Something() { cout << "2"; }
};

class Base
{
public:
    Base() { cout << "1"; }
    virtual ~Base() { cout << "1"; }
};

class Derived : public Base
{
public:
    Derived() { cout << "3"; }
    virtual ~Derived() { cout << "3"; }
};

```

```

    virtual ~Derived() { cout << "3"; }
private:
    Something m_dataMember;
};

```

即使前面的析构函数没有声明为 `virtual`，代码也可以继续运行。然而，如果代码使用 `delete` 删除一个实际指向派生类的基类指针，析构函数调用链将被破坏。比如，如果从前面的所有析构函数中移除 `virtual` 关键字，当使用指向 `Base` 对象的指针访问 `Derived` 对象，并 `delete` 对象时，就会出问题。

```

Base* ptr { new Derived{} };
delete ptr;

```

代码的输出很短，是"1231"。当删除 `ptr` 变量时，只调用了 `Base` 类的析构函数，因为析构函数没有声明为 `virtual`。结果是不会调用 `Derived` 类的析构函数，也不会调用 `Derived` 类中数据成员的析构函数！

从技术角度看，将 `Base` 类的析构函数声明为 `virtual`，可修正上面的问题。派生类将自动“虚化”。然而，建议显式地将所有析构函数声明为 `virtual`，这样就再也不必担心这个问题了。

#### 警告：

将所有析构函数声明为 `virtual`！编译器生成的默认析构函数不是 `virtual`，因此应该定义自己(或显式设置为默认)的虚析构函数，至少在父类中应该这么做。

#### 警告：

与构造函数一样，在析构函数中调用虚方法时，虚方法的行为将有所不同。如果派生类重写了基类中的虚方法，在基类的析构函数中调用该方法，会执行该方法的基类实现，而不是派生类的重写版本。

### 10.3.3 使用父类方法

在派生类中重写方法时，将有效地替换原始方法。然而，方法的父类版本仍然存在，仍然可以使用这些方法。比如，某个重写方法可能除了完成父类实现完成的任务之外，还会完成一些其他任务。考虑 `WeatherPrediction` 类中的 `getTemperature()` 方法，这个方法返回当前温度的字符串表示：

```

export class WeatherPrediction
{
public:
    virtual std::string getTemperature() const;
    // Omitted for brevity
};

```

在 `MyWeatherPrediction` 类中，可按如下方式重写这个方法：

```

export class MyWeatherPrediction : public WeatherPrediction
{
public:
    std::string getTemperature() const override;
    // Omitted for brevity
};

```

假定派生类要先调用基类的 `getTemperature()` 方法，然后将°F 添加到 `string`。为此，编写如下代码：

```
string MyWeatherPrediction::getTemperature() const
{
    // Note: \u00B0 is the ISO/IEC 10646 representation of the degree symbol.
    return getTemperature() + "\u00B0F"; // BUG
}
```

然而，上述代码无法运行。根据 C++ 的名称解析规则，首先解析的是局部作用域，然后是类作用域，根据这个顺序，函数中调用的是 `MyWeatherPrediction::getTemperature()`。其结果是无限递归，直到耗尽堆栈空间(某些编译器在编译期，会发现这种错误并报错)。

为让代码运行，需要使用作用域解析运算符，如下所示：

```
string MyWeatherPrediction::getTemperature() const
{
    // Note: \u00B0 is the ISO/IEC 10646 representation of the degree symbol.
    return WeatherPrediction::getTemperature() + "\u00B0F";
}
```

### 注意：

Microsoft Visual C++ 支持非标准 `_super` 关键字(两条下画线)。这个关键字允许编写如下代码：

```
return _super::getTemperature() + "\u00B0F";
```

在 C++ 中，调用当前方法的父类版本是一种常见操作。如果存在派生类链，每个派生类都可能想执行基类中已经定义的操作，同时添加自己的附加功能。

另一个示例是书本类型的类层次结构。图 10-5 显示了这个层次结构。

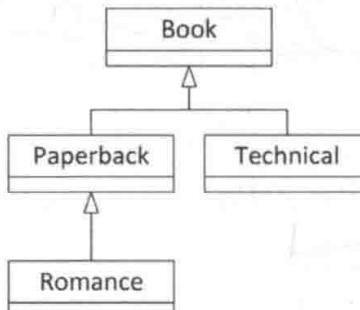


图 10-5 书本类型的层次结构

由于层次结构底层的类更具体地指出了书本的类型，获取书本描述信息的方法实际上需要考虑层次结构中的所有层次。为此，可连续调用父类方法，下面的代码演示了这一模式：

```
class Book
{
public:
    virtual ~Book() = default;
    virtual string getDescription() const { return "Book"; }
    virtual int getHeight() const { return 120; }
};

class Paperback : public Book
{
public:
    string getDescription() const override {
        return "Paperback " + Book::getDescription();
    }
};
```

```

    }

};

class Romance : public Paperback
{
public:
    string getDescription() const override {
        return "Romance " + Paperback::getDescription();
    }
    int getHeight() const override { return Paperback::getHeight() / 2; }
};

class Technical : public Book
{
public:
    string getDescription() const override {
        return "Technical " + Book::getDescription();
    }
};

int main()
{
    Romance novel;
    Book book;
    cout << novel.getDescription() << endl;      // Outputs "Romance Paperback Book"
    cout << book.getDescription() << endl;          // Outputs "Book"
    cout << novel.getHeight() << endl;             // Outputs "60"
    cout << book.getHeight() << endl;               // Outputs "120"
}

```

Book 基类有两个虚方法：getDescription()和getHeight()。所有派生类都重写了getDescription()，只有Romance类通过调用父类(Paperback)的getHeight()，然后将结果除以2，由此重写了getHeight()。Paperback类没有重写getHeight()，因此C++会沿着类层次结构向上寻找实现了getHeight()的类。在本例中，Paperback::getHeight()将解析为Book::getHeight()。

#### 10.3.4 向上转型和向下转型

如前所述，对象可转换为父类对象，或者赋值给父类。示例如下：

```
Base myBase { myDerived }; // Slicing!
```

这种情况下会导致截断，因为赋值结果是Base类的对象，而Base类的对象缺少Derived类中定义的附加功能。然而，如果用派生类对基类的指针或引用赋值，则不会产生截断：

```
Base& myBase { myDerived }; // No slicing!
```

这是通过基类使用派生类的正确途径，也称为向上转型(upcasting)。这也是让方法和函数使用类的引用而不是直接使用类对象的原因。通过使用引用，派生类在传递时没有发生截断。

##### 警告：

当向上转型时，使用基类指针或引用以避免截断。

将基类转换为其派生类也称为向下转型(downcasting)，专业的C++程序员通常不赞成这种转换，因为无法保证对象实际上属于派生类，也因为向下转型是不好的设计。比如，考虑下面的代码：

```

void presumptuous(Base* base)
{
    Derived* myDerived { static_cast<Derived*>(base) };
    // Proceed to access Derived methods on myDerived.
}

```

如果 `presumptuous()` 的作者还编写了调用 `presumptuous()` 的代码，那么可能一切正常，因为作者知道这个函数需要 `Derived*` 类型的参数。然而，如果其他程序员调用 `presumptuous()`，他们可能传递 `Base*`。编译期的检测无法强制参数类型，因此函数盲目地假定参数 `base` 实际上是一个指向 `Derived` 对象的指针。

向下转型有时是必需的，在可控环境中可充分利用这种转换。然而，如果打算进行向下转型，应该使用 `dynamic_cast()`，以使用对象内建的类型信息，拒绝没有意义的类型转换。这种内建信息通常驻留在虚表中，这意味着 `dynamic_cast()` 只能用于具有虚表的对象，即至少有一个虚编号的对象。如果针对某个指针的 `dynamic_cast()` 失败，这个指针的值就是 `nullptr`，而不是指向某个无意义的数据。如果针对对象引用的 `dynamic_cast()` 失败，将抛出 `std::bad_cast` 异常。本章的最后一节将会详细讨论类型转换。

前面的示例可以这样编写：

```

void lessPresumptuous(Base* base)
{
    Derived* myDerived { dynamic_cast<Derived*>(base) };
    if (myDerived != nullptr) {
        // Proceed to access Derived methods on myDerived.
    }
}

```

请记住，向下转型通常是设计不良的标志。为避免使用向下转型，应当反思，并修改设计。比如，`lessPresumptuous()` 函数实际上只能用于 `Derived` 类的对象，因此不应当接收 `Base` 类的指针，而应接收 `Derived` 类的指针。这样就不需要进行向下转型了。如果函数用于从 `Base` 类继承的不同派生类，则应考虑使用多态性的解决方案，如下所述。

#### 警告：

仅在必要的情况下才使用向下转型，一定要使用 `dynamic_cast()`。

## 10.4 继承与多态性

在理解了派生类与父类的关系后，就可以用最有力的方式使用继承——多态性(polymorphism)。第 5 章说过，多态性可以互换地使用具有共同父类的对象，并用对象替换父类对象。

### 10.4.1 回到电子表格

第 8 章和第 9 章使用电子表格程序作为示例来说明面向对象设计。`SpreadsheetCell` 代表一个数据元素。到目前为止，这个元素始终存储的是单个 `double` 值。下面给出了简化的 `SpreadsheetCell` 类定义。注意单元格里的数据类型可以是 `double` 或 `string_view`，但通常总是 `double`。示例中单元格的当前值总以字符串的形式返回。

```

class SpreadsheetCell
{

```

```

public:
    virtual void set(double value);
    virtual void set(std::string_view value);
    virtual std::string getString() const;
private:
    static std::string doubleToString(double value);
    static double stringToDouble(std::string_view value);
    double m_value;
};

}

```

在实际的电子表格应用程序中，单元格可以存储不同的数据类型，有时单元格是双精度值，有时是文本。如果单元格需要其他类型，例如公式单元格或日期单元格，该怎么办？

#### 10.4.2 设计多态性的电子表格单元格

`SpreadsheetCell` 类急需改变层次结构。一种合理方法是让 `SpreadsheetCell` 只包含字符串，从而限制其范围，在此过程中或许将其重命名为 `StringSpreadsheetCell`。为处理双精度值，可使用第二个类 `DoubleSpreadsheetCell`，这个类从 `StringSpreadsheetCell` 继承，并以自己的方式提供功能。图 10-6 演示了这一设计，这一方法想通过继承重用代码，因为 `DoubleSpreadsheetCell` 是 `StringSpreadsheetCell` 的唯一派生类，并利用了 `StringSpreadsheetCell` 内建的一些功能。

如果实现了图 10-6 所示的设计，就会发现派生类将重写基类的大多数(但不是全部)功能。因为双精度值与字符串的处理方式几乎完全不同，这个关系似乎与最初的理解差别很大。当然，包含字符串的单元格与包含双精度值的单元格存在明显的关系。图 10-6 使用的模型在某种意义上暗示 `DoubleSpreadsheetCell` “是一个” `StringSpreadsheetCell`。在此有一种更好的设计，让这两个类的地位等同，并有共同的父类 `SpreadsheetCell`，如图 10-7 所示。

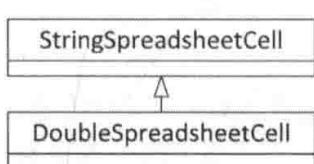


图 10-6 派生类重写基类的功能

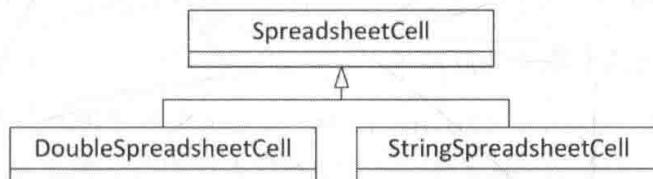


图 10-7 使两个类的地位等同

图 10-7 所示的设计显示了让 `SpreadsheetCell` 层次结构具有多态性的方法。由于 `DoubleSpreadsheetCell` 和 `StringSpreadsheetCell` 都从同一个父类 `SpreadsheetCell` 继承，从其他代码的角度看，它们是可以互换的。实际上这意味着：

- 两个派生类都支持由基类定义的同一接口(方法集)。
- 使用 `SpreadsheetCell` 对象的代码可调用接口中的任何方法，而不需要知道这个单元格是 `StringSpreadsheetCell` 还是 `DoubleSpreadsheetCell`。
- 由于虚方法的特殊能力，会根据对象所属的类调用接口中每个方法的正确实例。
- 其他数据结构(比如第 9 章讲述的 `Spreadsheet` 类)可通过引用父类类型，包含一组多类型的单元格。

#### 10.4.3 `SpreadsheetCell` 基类

由于所有电子表格单元格都是 `SpreadsheetCell` 基类的派生类，因此最好先编写这个类。当设计基类时，应该考虑派生类之间的关系。根据这些信息，可提取共有特性并将其放到父类中。比如，字符串单元格和双精度值单元格的共同点在于都包含单个数据块。由于数据来自用户，并最终显示给用户。

可以把这个值设置为字符串，并作为字符串来获取。这些行为就是用来组成基类的共享功能。

### 1. 初次尝试

`SpreadsheetCell` 基类负责定义所有派生类支持的行为。在本例中，所有单元格都需要将值设置为字符串。此外，所有单元格都需要将当前值返回为字符串。基类定义中声明了这些方法，以及显式设置为默认的虚析构函数，但没有数据成员。`SpreadsheetCell` 定义在 `spreadsheet_cell` 模块中。

```
export module spreadsheet_cell;
import <string>;
import <string_view>;

export class SpreadsheetCell
{
public:
    virtual ~SpreadsheetCell() = default;
    virtual void set(std::string_view value);
    virtual std::string getString() const;
};
```

当开始编写这个类的.cpp 文件时，很快就会遇到问题。由于电子表格单元格的基类既不包含 `double` 也不包含 `string` 类型的数据成员，如何实现这个类呢？更宽泛地讲，如何定义这样一个父类，这个父类声明了派生类支持的行为，但是并不定义这些行为的实现。

可能的方法之一是为这些行为实现“什么都不做”的功能。比如，调用 `SpreadsheetCell` 基类的 `set()` 方法将没有任何效果，因为基类没有任何成员需要设置。然而这种方法仍然存在问题。理想情况下，基类不应该有实例。调用 `set()` 方法应该总是有效，因为总是会基于 `DoubleSpreadsheetCell` 或 `StringSpreadsheetCell` 调用这个方法。好的解决方案应该强制执行这一限制。

### 2. 纯虚方法和抽象基类

纯虚方法(pure virtual methods)在类定义中显式说明该方法不需要定义。如果将某个方法设置为纯虚方法，就是告诉编译器当前类中不存在这个方法的定义。具有至少一个纯虚方法的类称为抽象类，因为这个类不能被实例化。编译器会强制接受这个事实：如果某个类包含一个或多个纯虚方法，就无法构建这种类型的对象。

采用专门的语法指定纯虚方法：方法声明后紧接着=0。不需要编写任何代码。

```
export class SpreadsheetCell
{
public:
    virtual ~SpreadsheetCell() = default;
    virtual void set(std::string_view inString) = 0;
    virtual std::string getString() const = 0;
};
```

现在基类成了抽象类，无法创建 `SpreadsheetCell` 对象，下面的代码将无法编译，并给出诸如“‘`SpreadsheetCell`’:cannot instantiate abstract class”的错误。

```
SpreadsheetCell cell; // Error! Attempts creating abstract class instance
```

然而，一旦实现了 `StringSpreadsheetCell` 类，下面的代码就可成功编译，原因在于实例化了抽象基类的派生类。

```
unique_ptr<SpreadsheetCell> cell { new StringSpreadsheetCell{} };
```

**注意：**

抽象类提供了一种禁止其他代码直接实例化对象的方法，而它的派生类可以实例化对象。

注意，不需要实现 SpreadsheetCell 类。所有方法都是纯虚方法，析构函数显式地设置为默认。

#### 10.4.4 独立的派生类

编写 StringSpreadsheetCell 和 DoubleSpreadsheetCell 类只需要实现父类中定义的功能。由于想让客户实现并使用字符串单元格和双精度值单元格，因此单元格不可以是抽象的——必须实现从父类继承的所有纯虚方法。如果派生类没有实现从父类继承的所有纯虚方法，那么派生类也是抽象的，客户就不能实例化派生类的对象。

##### 1. StringSpreadsheetCell 类的定义

StringSpreadsheetCell 类定义在名为 string\_spreadsheet\_cell 的自身模块中。

编写 StringSpreadsheetCell 类定义的第一步是从 SpreadsheetCell 类继承。为此，string\_spreadsheet\_cell 模块需要被导入。

第二步是重写继承的纯虚方法，此次不将其设置为 0。

最后一步是为字符串单元格添加一个私有数据成员 m\_value，在其中存储实际单元格数据。这个数据成员是本书第 1 章中介绍的 std::optional。通过使用 optional 类型，可以确认是否没有设置单元格的值或单元格的值设为空字符串。

```
export module string_spreadsheet_cell;
import spreadsheet_cell;
import <string>;
import <string_view>;
import <optional>

export class StringSpreadsheetCell : public SpreadsheetCell
{
public:
    void set(std::string_view value) override;
    std::string getString() const override;
private:
    std::optional<std::string> m_value;
};
```

##### 2. StringSpreadsheetCell 的实现

set()方法十分简单，因为内部表示已经是一个字符串。getString()方法必须考虑到 m\_value 的类型是 optional，可能不具有值。如果 m\_value 不具有值，getString()将返回一个默认字符串，本例中是空字符串。可以使用 std::optional 的 value\_or()方法对此进行简化。使用 m\_value.value\_or("")，如果 m\_value 包含实际的值，将返回相应的值，否则将返回空值。

```
void set(std::string_view value) override { m_value = value; }
std::string getString() const override { return m_value.value_or(""); }
```

##### 3. DoubleSpreadsheetCell 类的定义和实现

双精度版本遵循类似的模式，但具有不同的逻辑。除了基类的 set()方法以 string\_view 作为参数之外，还提供了新的 set()方法以允许用户使用双精度值设置其值。两个新的 private static 方法用于字符

串和双精度值的互相转换，反之亦然。与 StringSpreadsheetCell 相同，这个类也有一个 m\_value 数据成员，此时这个成员的类型是 optional<double>。

```
export module double_spreadsheet_cell;
import spreadsheet_cell;
import <string>;
import <string_view>;
import <optional>

export class DoubleSpreadsheetCell : public SpreadsheetCell
{
public:
    virtual void set(double value);
    void set(std::string_view value) override;
    std::string getString() const override;
private:
    static std::string doubleToString(double value);
    static double stringToDouble(std::string_view value);
    std::optional<double> m_value;
};
```

以双精度值作为参数的 set() 方法简单明了。string\_view 版本使用 private static 方法 stringToDouble()。getString() 方法返回存储的双精度值作为字符串；如果未存储任何值，则返回一个空字符串。它使用 std::optional 的 has\_value() 方法来查询 optional 是否具有实际值。如果具有值，则使用 value() 方法获取。

```
virtual void set(double value) { m_value = value; }
void set(string_view value) override { m_value = stringToDouble(value); }

std::string getString() const override
{
    return (m_value.has_value() ? doubleToString(m_value.value()) : "");}
```

可以看到，在层次结构中实现电子表格单元格的主要优点是代码更加简单。每个对象都以自我为中心，只执行各自的功能。

注意在此省略了 doubleToString() 和 stringToDouble() 方法的实现，因为与第 8 章的实现相同。

#### 10.4.5 利用多态性

现在 SpreadsheetCell 层次结构具有多态性，客户代码可利用多态性提供的种种好处。下面的测试程序显示了这些功能。

为演示多态性，测试程序声明了一个具有 3 个 SpreadsheetCell 指针的矢量，记住由于 SpreadsheetCell 是一个抽象类，因此不能创建这种类型的对象。然而，仍然可以使用 SpreadsheetCell 的指针或引用，因为它实际上指向的是其中一个派生类。由于是父类型 SpreadsheetCell 的矢量，因此可以任意存储两个派生类。这意味着矢量元素可以是 StringSpreadsheetCell 或 DoubleSpreadsheetCell。

```
vector<unique_ptr<SpreadsheetCell>> cellArray;
```

矢量的前两个元素指向新建的 StringSpreadsheetCell，第三个元素指向一个新的 DoubleSpreadsheetCell。

```
cellArray.push_back(make_unique<StringSpreadsheetCell>());
cellArray.push_back(make_unique<StringSpreadsheetCell>());
```

```
cellArray.push_back(make_unique<DoubleSpreadsheetCell>());
```

现在矢量包含了多类型数据，基类声明的任何方法都可以应用到矢量中的对象。代码只是使用了 SpreadsheetCell 指针——编译器在编译期不知道对象的实际类型是什么。然而，由于这两个类是 SpreadsheetCell 的派生类，因此必须支持 SpreadsheetCell 的方法。

```
cellArray[0]->set("hello");
cellArray[1]->set("10");
cellArray[2]->set("18");
```

当调用 `getString()` 方法时，每个对象都会正确地返回值的字符串表示。重要的(某种意义上令人惊讶的)是，不同的对象以不同的方式完成这一任务。`StringSpreadsheetCell` 返回它存储的值，或返回空字符串。如果包含值，`DoubleSpreadsheetCell` 首先执行转换；否则返回一个空字符串。程序员不需要知道对象如何做到这一点——只需要知道因为对象是一个 `SpreadsheetCell`，所以可以执行此行为。

```
cout << format("Vector: [{}, {}, {}]", cellArray[0]->getString(),
               cellArray[1]->getString(),
               cellArray[2]->getString()) << endl;
```

#### 10.4.6 考虑将来

`SpreadsheetCell` 层次结构的新实现从面向对象设计的观点来看当然是一个进步。但是，对于实际的电子表格程序来说，这个类层次结构还不够充分，主要有以下几个原因。

首先，即使不考虑改进设计，现在仍然缺少一个功能：将某个单元格类型转换为其他类型。由于将单元格分为两类，单元格对象的结合变得更松散。为提供将 `DoubleSpreadsheetCell` 转换为 `StringSpreadsheetCell` 的功能，应添加一个转换构造函数(或类型构造函数)，这个构造函数类似于复制构造函数，但参数不是对同类对象的引用，而是对同级类对象的引用。另外注意，现在必须声明一个默认构造函数，可将其显式设置为默认，因为一旦自行声明任何构造函数，编译器将停止生成。

```
export class StringSpreadsheetCell : public SpreadsheetCell
{
public:
    StringSpreadsheetCell() = default;
    StringSpreadsheetCell(const DoubleSpreadsheetCell& cell)
        : m_value { cell.getString() }
    {}
    // Omitted for brevity
};
```

通过转换构造函数，可以很方便地用 `DoubleSpreadsheetCell` 创建 `StringSpreadsheetCell`。然而不要将其与指针或引用的类型转换混淆。类型转换无法将一个指针或引用转换为同级的另一个指针或引用，除非按照第 15 章讲述的方法重载类型转换运算符。

#### 警告：

在层次结构中，总可以向上转型，有时也可以向下转型。通过改变类型转换运算符的行为，或者使用 `reinterpret_cast<>`(不推荐采用这些方法)，就可以在层次结构中进行类型转换。

其次，如何为单元格实现运算符重载是一个很有趣的问题，在此有几种可能的解决方案。其中一种方案是：针对每个单元格组合，实现每个运算符的重载版本。由于只有两个派生类，因此这样做并不难。可编写一个 `operator+` 函数，将两个双精度单元格相加，将两个字符串单元格相加，将双精度单

元格与字符串单元格相加。另一种方案是给出一种通用表示，前面的实现已将字符串作为标准化的通用类型表示。通过这种通用表示，一个 `operator+` 函数就可以处理所有情况。假定两个单元格相加的结果始终是字符串单元格，那么一个在 `string_spreadsheet_cell` 模块中的可能的实现如下所示。

```
export StringSpreadsheetCell operator+(const StringSpreadsheetCell& lhs,
                                         const StringSpreadsheetCell& rhs)
{
    StringSpreadsheetCell newCell;
    newCell.set(lhs.getString() + rhs.getString());
    return newCell;
}
```

只要编译器可将特定的单元格转换为 `StringSpreadsheetCell`，这个运算符就可以运行。考虑前面的示例，`StringSpreadsheetCell` 构造函数采用 `DoubleSpreadsheetCell` 作为参数，如果这是 `operator+` 运行的唯一方法，那么编译器将自动执行转换。这意味着下面的代码可以运行，尽管 `operator+` 被显式地用于 `StringSpreadsheetCell`。

```
DoubleSpreadsheetCell myDbl;
myDbl.set(8.4);
StringSpreadsheetCell result { myDbl + myDbl };
```

当然，相加的结果实际上并不是将数字相加，而是将双精度单元格转换为字符串单元格，然后将字符串相加，结果是一个值为 8.4000008.400000 的 `StringSpreadsheetCell`。

如果对多态性还不确定，可运行这个示例的代码并获取答案。如果只是为了体验这个类的各种特性，前面示例中的 `main()` 函数是一个很好的起点。

## 10.5 多重继承

第 5 章已经讲过，多重继承通常被认为是面向对象编程中一种复杂且不必要的部分。请判断多重继承是否有用，本节将阐述 C++ 中多重继承的机制。

### 10.5.1 从多个类继承

从语法角度看，定义具有多个父类的类很简单。为此，只需要在声明类名时分别列出基类。

```
class Baz : public Foo, public Bar { /* Etc. */};
```

由于列出了多个父类，`Baz` 对象具有如下特性：

- `Baz` 对象支持 `Foo` 和 `Bar` 类的 `public` 方法，并且包含这两个类的数据成员。
- `Baz` 类的方法有权访问 `Foo` 和 `Bar` 类的 `protected` 数据成员和方法。
- `Baz` 对象可以向上转型为 `Foo` 或 `Bar` 对象。
- 创建新的 `Baz` 对象将自动调用 `Foo` 和 `Bar` 类的默认构造函数，并按照类定义中列出的类顺序进行。
- 删除 `Baz` 对象将自动调用 `Foo` 和 `Bar` 类的析构函数，调用顺序与类在类定义中的顺序正好相反。

下例显示了一个 `DogBird` 类，它有两个父类——`Dog` 类和 `Bird` 类，如图 10-8 所示。这是一个荒谬的示例，但是不应该认为多重继承本身是荒谬的，请自行判断。

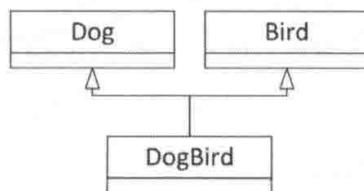


图 10-8 DogBird 有两个父类

```

class Dog
{
public:
    virtual void bark() { cout << "Woof!" << endl; }
};

class Bird
{
public:
    virtual void chirp() { cout << "Chirp!" << endl; }
};

class DogBird : public Dog, public Bird
{
};

```

使用具有多个父类的类对象与使用具有单个父类的类对象没什么不同。实际上，客户代码甚至不需要知道这个类有两个父类。需要关心的只是这个类支持的属性和行为。在此情况下，DogBird 对象支持 Dog 和 Bird 类的所有 public 方法。

```

DogBird myConfusedAnimal;
myConfusedAnimal.bark();
myConfusedAnimal.chirp();

```

程序的输出如下：

```

Woof!
Chirp!

```

## 10.5.2 名称冲突和歧义基类

多重继承崩溃的场景并不难想象，下面的示例显示了一些必须要考虑的边缘情况。

### 1. 名称歧义

如果 Dog 类和 Bird 类都有一个 eat()方法，会发生什么？由于 Dog 类和 Bird 类毫不相干，eat()方法的一个版本无法重写另一个版本——在派生类 DogBird 中这两个方法都存在。

只要客户代码不调用 eat()方法，就不会出现问题。尽管有两个版本的 eat()方法，但 DogBird 类仍然可以正确编译。然而，如果客户代码试图调用 DogBird 类的 eat()方法，编译器将报错，指出对 eat()方法的调用有歧义。编译器不知道该调用哪个版本。下面的代码存在歧义错误：

```

class Dog
{
public:
    virtual void bark() { cout << "Woof!" << endl; }
    virtual void eat() { cout << "The dog ate." << endl; }
};

class Bird
{
public:
    virtual void chirp() { cout << "Chirp!" << endl; }
    virtual void eat() { cout << "The bird ate." << endl; }
};

```

```

class DogBird : public Dog, public Bird
{
};

int main()
{
    DogBird myConfusedAnimal;
    myConfusedAnimal.eat(); // Error! Ambiguous call to method eat()
}

```

为了消除歧义，可使用 `dynamic_cast` 显式地将对象向上转型(本质上是向编译器隐藏多余的方法版本)，也可以使用歧义消除语法。下面的代码显示了调用 `eat()` 方法的 `Dog` 版本的两种方案：

```

dynamic_cast<Dog&>(myConfusedAnimal).eat(); // Calls Dog::eat()
myConfusedAnimal.Dog::eat(); // Calls Dog::eat()

```

使用与访问父类方法相同的语法(`::`运算符)，派生类的方法本身可以显式地为同名的不同方法消除歧义。比如，`DogBird` 类可以定义自己的 `eat()` 方法，从而消除其他代码中的歧义错误。在方法内部，可以判断调用哪个父类版本：

```

class DogBird : public Dog, public Bird
{
public:
    void eat() override
    {
        Dog::eat(); // Explicitly call Dog's version of eat()
    }
};

```

另一种防止歧义错误的方式是使用 `using` 语句显式指定，在 `DogBird` 类中应继承哪个版本的 `eat()` 方法，如下面的 `DogBird` 类定义所示。

```

class DogBird : public Dog, public Bird
{
public:
    using Dog::eat; // Explicitly inherit Dog's version of eat()
};

```

## 2. 歧义基类

另一种引起歧义的情况是从同一个类继承两次。例如，如果出于某种原因 `Bird` 类从 `Dog` 类继承，`DogBird` 类的代码将无法编译，因为 `Dog` 变成了歧义基类。

```

class Dog {};
class Bird : public Dog {};
class DogBird : public Bird, public Dog {} // Error!

```

多数歧义基类的情况或者是由人为的“what-if”示例(如前面的示例)引起的，或者是由于类层次结构的混乱引起的。图 10-9 显示了前面示例中的类图，并指出了歧义。

数据成员也可以引起歧义。如果 `Dog` 和 `Bird` 类具有同名的数据成员，当客户代码试图访问这个成员时，就会发生歧义错误。

多个父类本身也可能有共同的父类。例如，`Bird` 和 `Dog` 类可能都是 `Animal` 类的派生类，如图 10-10 所示。

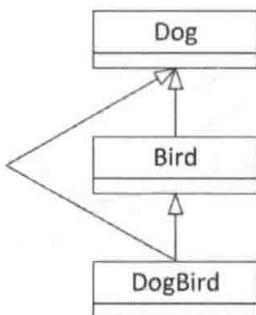


图 10-9 类图

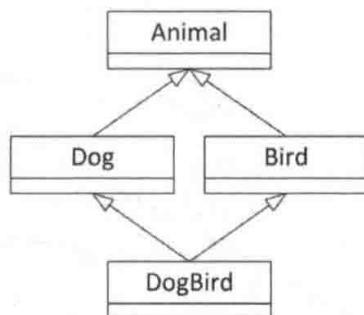


图 10-10 Dog 和 Bird 类都是 Animal 类的派生类

C++ 允许这种类型的类层次结构，尽管仍然存在着名称歧义。例如，如果 Animal 类有一个公有方法 sleep()，DogBird 对象无法调用这个方法，因为编译器不知道该调用 Dog 类继承的版本还是 Bird 类继承的版本。

使用“菱形”类层次结构的最佳方法是将最顶部的类设置为抽象类，将所有方法都设置为纯虚方法。由于类只声明方法而不提供定义，在基类中没有方法可以调用，因此在这个层次上就没有歧义。

下例实现了菱形类层次结构，其中有一个每个派生类都必须定义的纯虚方法 eat()。DogBird 类仍然必须显式说明使用哪个父类的 eat() 方法，但是 Dog 和 Bird 类引起歧义的原因是它们具有相同的方法，而不是因为从同一个类继承。

```

class Animal
{
public:
    virtual void eat() = 0;
};

class Dog : public Animal
{
public:
    virtual void bark() { cout << "Woof!" << endl; }
    void eat() override { cout << "The dog ate." << endl; }
};

class Bird : public Animal
{
public:
    virtual void chirp() { cout << "Chirp!" << endl; }
    void eat() override { cout << "The bird ate." << endl; }
};

class DogBird : public Dog, public Bird
{
public:
    using Dog::eat;
};

```

虚基类是处理菱形类层次结构中顶部类的更好方法，将在本章最后讲述。

### 3. 多重继承的用途

为什么程序员要在代码中使用多重继承？多重继承最直接的用例就是定义一个既“是一个”事物，又“是一个”其他事物的类对象。第 5 章已经说过，遵循这个模式的实际对象很难恰当地转换为代码。

多重继承最简单有力的用途就是实现混入(mix-in)类。混入类的介绍请参阅第 5 章，更多详情请

参阅第 32 章“整合设计技术和框架”。

使用多重继承的另一个原因是给基于组件的类建模。第 5 章给出了飞机模拟示例，Airplane 类有引擎、机身、控制系统和其他组件。尽管 Airplane 类的典型实现是将这些组件当作独立的数据成员，但也可以使用多重继承。飞机类可从引擎、机身、控制系统继承，从而有效地获得这些组件的行为和属性。建议不要使用这种类型的代码，这将“有一个”关系与继承混淆了，而继承用于“是一个”关系。推荐的解决方案是让 Airplane 类包含 Engine、Fuselage 和 Controls 类型的数据成员。

## 10.6 有趣而晦涩的继承问题

扩展类引发了多种问题。类的哪些特征可以改变，哪些不能改变？什么是非公共继承？什么是虚基类？下面将回答这些问题。

### 10.6.1 修改重写方法的返回类型

重写某个方法的主要原因是为了修改方法的实现。然而，有时是为了修改方法的其他特征，比如返回类型。

根据经验，重写方法要使用与基类一致的方法声明(或方法原型)。实现可以改变，但原型保持不变。

然而事实未必总是如此，在 C++ 中，如果原始的返回类型是某个类的指针或引用，重写的方法可将返回类型改为派生类的指针或引用。这种类型称为协变返回类型(covariant return types)。如果基类和派生类处于平行层次结构(parallel hierarchy)中，使用这个特性可以带来便利。平行层次结构是指，一个类层次结构与另一个类层次结构没有相交，但是存在联系。

例如，考虑樱桃果园的模拟程序。可使用两个类层次结构模拟不同但明显相关的实际对象。第一个是 Cherry 类层次结构，Cherry 基类有一个名为 BingCherry 的派生类。与此类似，另一个类层次结构的基类为 CherryTree，派生类为 BingCherryTree。图 10-11 显示了这两个类层次结构。



图 10-11 两个类层次结构

现在假定 CherryTree 类有一个虚方法 pick()，这个虚方法从樱桃树上获取一个樱桃：

```
Cherry* CherryTree::pick() { return new Cherry{}; }
```

#### 注意：

为了演示如何更改返回类型，本例未返回智能指针，而是返回普通指针。本节末尾将解释其中的原因。当然，调用者应当在智能指针(而非普通指针)中立即存储结果。

在 BingCherryTree 派生类中，要重写这个方法。或许 BingCherry 在摘下来时需要擦拭(请允许我们这么说)。由于 BingCherry 也是 Cherry，在下例中，方法的原型保持不变，而方法却被重写。BingCherry 指针被自动转换为 Cherry 指针。注意，这个实现使用 unique\_ptr 确保当 polish() 抛出异常时，没有泄漏内存。

```
Cherry* BingCherryTree::pick()
```

```

{
    auto theCherry { make_unique<BingCherry>() };
    theCherry->polish();
    return theCherry.release();
}

```

上面的实现非常好。然而，由于 BingCherryTree 类始终返回 BingCherry 对象，因此可通过修改返回类型，向这个类的潜在用户指明这一点，如下所示。

```

BingCherry* BingCherryTree::pick()
{
    auto theCherry { make_unique<BingCherry>() };
    theCherry->polish();
    return theCherry.release();
}

```

下面的示例显示了 BingCherryTree::pick()方法的用法。printType()是基类 Cherry 的虚方法，由 BingCherryTree 派生类重写，用于输出 cherry 的类型。

```

BingCherryTree theTree;
unique_ptr<Cherry> theCherry { theTree.pick() };
theCherry->printType();

```

为判断能否修改重写方法的返回类型，可以考虑已有代码是否能够继续运行，这称为里氏替换原则(Liskov Substitution Principle, LSP)。在上例中，修改返回类型没有问题，因为假定 pick()方法总是返回 Cherry\* 的代码仍然可以成功编译并正常运行。由于 BingCherry 也是 Cherry，因此根据 CherryTree 版本的 pick()返回值调用的任何方法，仍然可以基于 BingCherryTree 版本的 pick()返回值进行调用。

不能将返回类型修改为完全不相关的类型，例如 void\*。下面的代码无法编译：

```
void* BingCherryTree::pick() { /*...*/ } // Error!
```

这段代码会导致编译错误，如下所示：

```
'BingCherryTree::pick': overriding virtual function return type differs and is not covariant
from 'CherryTree::pick'.
```

如前所述，这个示例正用普通指针替代智能指针。将 unique\_ptr 用作返回类型时，这不能用于本例。假设 CherryTree::pick()方法返回 unique\_ptr<Cherry>，如下所示：

```
unique_ptr<Cherry> CherryTree::pick() { return make_unique<Cherry>(); }
```

此时，无法将 BingCherryTree::pick()方法的返回类型改成 unique\_ptr<BingCherry>。下面的代码无法编译：

```
unique_ptr<BingCherry> BingCherry::pick() override { /*...*/ }
```

原因在于 unique\_ptr 是类模板。创建了 unique\_ptr 类模板的两个实例 unique\_ptr<Cherry> 和 unique\_ptr<BingCherry>。这两个实例是完全不同的类型，完全无关。无法更改重写方法的返回类型来返回完全不同的类型。

## 10.6.2 派生类中添加虚基类方法的重载

可以向派生类中添加虚基类方法的新重载。也就是说，可以使用新原型在派生类中添加虚方法的重载，但继续继承基类版本。该技术使用 using 声明将方法的基类定义显式包含在派生类中。示

例如：

```
class Base
{
public:
    virtual void someMethod();
};

class Derived : public Base
{
public:
    using Base::someMethod;           // Explicitly inherit the Base version
    virtual void someMethod(int i);   // Add a new overload of someMethod().
    virtual void someOtherMethod();
};
```

#### 注意：

对于派生类和基类同名的方法，具有不同参数列表的情况很少见。

### 10.6.3 继承的构造函数

上一节提到，可在派生类中使用 `using` 关键字显式地包含基类中定义的方法。这适用于普通类方法，也适用于构造函数，允许在派生类中继承基类的构造函数。考虑以下 `Base` 和 `Derived` 类定义：

```
class Base
{
public:
    virtual ~Base() = default;
    Base() = default;
    Base(std::string_view str);
};

class Derived : public Base
{
public:
    Derived(int i);
};
```

只能用提供的 `Base` 构造函数构建 `Base` 对象，要么是默认构造函数，要么是包含 `string_view` 参数的构造函数。另外，只能用 `Derived` 构造函数创建 `Derived` 对象，这个构造函数需要一个整数作为参数。不能使用 `Base` 类中接收 `string_view` 的构造函数构建 `Derived` 对象。例如：

```
Base base { "Hello" };           // OK, calls string_view Base ctor.
Derived derived1 { 1 };          // OK, calls integer Derived ctor.
Derived derived2 { "Hello" };    // Error, Derived does not inherit string_view ctor.
Derived derived3;               // Error, Derived does not have a default ctor.
```

如果喜欢使用基于 `string_view` 的 `Base` 构造函数构建 `Derived` 对象，可在 `Derived` 类中显式地继承 `Base` 构造函数，如下所示。

```
class Derived : public Base
{
public:
    using Base::Base;
    Derived(int i);
};
```

`using` 语句从父类继承除默认构造函数外的其他所有构造函数，现在可通过下面两种方法构建 `Derived` 对象。

```
Derived derived1 { 1 };           // OK, calls integer Derived ctor.
Derived derived2 { "Hello" };     // OK, calls inherited string_view Base ctor.
Derived derived3;                // OK, calls inherited default Base ctor.
```

当然，`Derived` 类可以从所有 `Base` 类中继承构造函数，而不定义任何构造函数。示例如下：

```
class Base
{
public:
    virtual ~Base() = default;
    Base() = default;
    Base(std::string_view str);
    Base(float f);
};

class Derived : public Base
{
public:
    using Base::Base;
};
```

可以如下创建 `Derived` 类的实例：

```
Derived derived1("Hello");      // OK, calls inherited string_view Base ctor.
Derived derived2(1.23f);        // OK, calls inherited float Base ctor.
Derived derived3;              // OK, calls inherited default Base ctor.
```

### 1. 隐藏继承的构造函数

定义的构造函数可与从 `Base` 类继承的构造函数有相同的参数列表。与所有的重写一样，此时 `Derived` 类的构造函数的优先级高于继承的构造函数。在下例中，`Derived` 类使用 `using` 关键字，继承了 `Base` 类中除默认构造函数外的其他所有构造函数。然而，由于 `Derived` 类定义了一个使用浮点数作为参数的构造函数，从 `Base` 类继承的使用浮点数作为参数的构造函数被重写。

```
class Base
{
public:
    virtual ~Base() = default;
    Base() = default;
    Base(std::string_view str);
    Base(float f);
};

class Derived : public Base
{
public:
    using Base::Base;
    Derived(float f);    // Hides inherited float Base ctor.
};
```

根据这个定义，可用下面的代码创建 `Derived` 对象：

```
Derived derived1 { "Hello" };    // OK, calls inherited string_view Base ctor.
```

```
Derived derived2 { 1.23f };           // OK; calls float Derived ctor.
Derived derived3;                     // OK, calls inherited Base ctor.
```

使用 `using` 子句从基类继承构造函数有一些限制。

- 当从基类继承构造函数时，会继承除默认构造函数外的其他全部构造函数，不能只是继承基类构造函数的一个子集。
- 当继承构造函数时，无论 `using` 声明位于哪个访问规范下，都将使用与基类中相同的访问规范来继承。

## 2. 继承的构造函数和多重继承

第二个限制与多重继承有关。如果一个基类的某个构造函数与另一个基类的构造函数具有相同的参数列表，就不可能从基类继承构造函数，因为那样会导致歧义。为解决这个问题，`Derived` 类必须显式地定义冲突的构造函数。例如，下面的 `Derived` 类试图继承 `Base1` 和 `Base2` 基类的所有构造函数，这会产生编译错误，因为使用浮点数作为参数的构造函数存在歧义。

```
class Base1
{
public:
    virtual ~Base1() = default;
    Base1() = default;
    Base1(float f);
};

class Base2
{
public:
    virtual ~Base2() = default;
    Base2() = default;
    Base2(std::string_view str);
    Base2(float f);
};

class Derived : public Base1, public Base2
{
public:
    using Base1::Base1;
    using Base2::Base2;
    Derived(char c);
};

int main()
{
    Derived d { 1.2f }; // Error, ambiguity.
}
```

`Derived` 类定义中的第一条 `using` 语句继承了 `Base1` 类的构造函数。这意味着 `Derived` 类具有如下构造函数：

```
Derived(float f); // Inherited from Base1
```

`Derived` 类定义中的第二条 `using` 语句试图继承 `Base2` 类的全部构造函数。然而，这会导致编译错误，因为这意味着 `Derived` 类拥有第二个 `Derived(float f)` 构造函数。为解决这个问题，可在 `Derived` 类中显式声明冲突的构造函数，如下所示。

```

class Derived : public Base1, public Base2
{
public:
    using Base1::Base1;
    using Base2::Base2;
    Derived(char c);
    Derived(float f);
};

```

现在，`Derived` 类显式地声明了一个采用浮点数作为参数的构造函数，从而解决了歧义问题。如果愿意，在`Derived` 类中显式声明的使用浮点数作为参数的构造函数仍然可以在`ctor-initializer` 中调用`Base1` 和 `Base2` 构造函数，如下所示。

```
Derived::Derived(float f) : Base1{f}, Base2{f} {}
```

### 3. 数据成员的初始化

当使用继承的构造函数时，要确保所有成员变量都正确地初始化。比如，考虑下面 `Base` 和 `Derived` 类的新定义。这个示例没有正确地初始化 `m_int` 数据成员，在任何情况下这都是一个严重错误。

```

class Base
{
public:
    virtual ~Base() = default;
    Base(std::string_view str) : m_str{str} {}
private:
    std::string m_str;
};

class Derived : public Base
{
public:
    using Base::Base;
    Derived(int i) : Base{""}, m_int{i} {}
private:
    int m_int;
};

```

可采用如下方法创建一个 `Derived` 对象：

```
Derived s1{2};
```

这条语句将调用 `Derived(int i)` 构造函数，这个构造函数将初始化 `Derived` 类的 `m_int` 数据成员，并调用 `Base` 构造函数，用空字符串初始化 `m_str` 数据成员。

由于 `Derived` 类继承了 `Base` 构造函数，还可按下面的方式创建一个 `Derived` 对象：

```
Derived s2{"Hello World"};
```

这条语句调用从 `Base` 类继承的构造函数。然而，从 `Base` 类继承的构造函数只初始化了 `Base` 类的 `m_str` 成员变量，没有初始化 `Derived` 类的 `m_int` 成员变量，`m_int` 处于未初始化状态。通常不建议这么做。

解决方法是使用类内成员初始化器，第 8 章已讨论过这个特性。以下代码使用类内成员初始化器将 `m_int` 初始化为 0。`Derived(int i)` 构造函数仍可修改这一初始化行为，将 `m_int` 初始化为参数 `i` 的值。

```

class Derived : public Base
{
public:
    using Base::Base;
    Derived(int i) : Base( "" ), m_int( i ) {}
private:
    int m_int { 0 };
};

```

## 10.6.4 重写方法时的特殊情况

当重写方法时，需要注意几种特殊情况。本节将列出可能遇到的一些情况。

### 1. 静态基类方法

在 C++ 中，不能重写静态方法。对于多数情况而言，知道这一点就足够了。然而，在此需要了解一些推论。

首先，方法不可能既是静态的又是虚的。出于这个原因，试图重写一个静态方法并不能得到预期的结果。如果派生类中存在的静态方法与基类中的静态方法同名，实际上这是两个独立的方法。

下面的代码显示了两个类，这两个类都有一个名为 beStatic() 的静态方法。这两个方法毫无关系。

```

class BaseStatic
{
public:
    static void beStatic() {
        cout << "BaseStatic being static." << endl;
    }
};

class DerivedStatic : public BaseStatic
{
public:
    static void beStatic() {
        cout << "DerivedStatic keepin' it static." << endl;
    }
};

```

由于静态方法属于类，因此调用两个类的同名方法时，将调用各自的方法。

```
BaseStatic::beStatic();
DerivedStatic::beStatic();
```

输出为：

```
BaseStatic being static.
DerivedStatic keepin' it static.
```

用类名访问这些方法时一切都很正常。当涉及对象时，这一行为就不是那么明显。在 C++ 中，可以使用对象调用静态方法，但由于方法是静态的，因此没有 this 指针，也无法访问对象本身，使用对象调用静态方法，等价于使用 classname::method() 调用静态方法。回到前面的示例，可以编写如下代码，但是结果令人惊讶。

```

DerivedStatic myDerivedStatic;
BaseStatic& ref { myDerivedStatic };
myDerivedStatic.beStatic();
ref.beStatic();

```

对 `beStatic()` 的第一次调用显然调用了 `DerivedStatic` 版本，因为调用它的对象被显式地声明为 `DerivedStatic` 对象。第二次调用的运行方式可能并非预期的那样。这个对象是一个 `BaseStatic` 引用，但指向的是一个 `DerivedStatic` 对象。在此情况下，会调用 `BaseStatic` 版本的 `beStatic()`。原因是当调用静态方法时，C++ 不关心对象实际上是什么，只关心编译期的类型。在此情况下，该类型为指向 `BaseStatic` 对象的引用。

前面示例的输出如下：

```
DerivedStatic keepin' it static.  
BaseStatic being static.
```

### 注意：

静态方法属于定义它的类，而不属于特定的对象。当类中的方法调用静态方法时，所调用的版本是通过正常的名称解析来决定的。当使用对象调用时，对象实际上并不涉及调用，只是用来判断编译期的类型。

## 2. 重载基类方法

当指定名称和一组参数以重写某个方法时，编译器隐式地隐藏基类中同名方法的所有其他实例。想法为：如果重写了给定名称的某个方法，可能是想重写所有的同名方法，只是忘记这么做了，因此应该作为错误处理。这是有意义的，可以这么考虑——为什么要修改方法的某些版本而不修改其他版本呢？考虑下面的 `Derived` 类，它重写了一个方法，而没有重写相关的同级重载方法。

```
class Base  
{  
public:  
    virtual ~Base() = default;  
    virtual void overload() { cout << "Base's overload()" << endl; }  
    virtual void overload(int i) {  
        cout << "Base's overload(int i)" << endl; }  
};  
  
class Derived : public Base  
{  
public:  
    void overload() override {  
        cout << "Derived's overload()" << endl; }  
};
```

如果试图用 `Derived` 对象调用以 `int` 值作为参数的 `overload()` 版本，代码将无法编译，因为没有显式地重写这个方法。

```
Derived myDerived;  
myDerived.overload(2); // Error! No matching method for overload(int).
```

然而，使用 `Derived` 对象访问该版本的方法是可行的。只需要使用指向 `Base` 对象的指针或引用：

```
Derived myDerived;  
Base& ref { myDerived };  
ref.overload(7);
```

在 C++ 中，隐藏未实现的重载方法只是表象。显式声明为子类型实例的对象无法使用这些方法，但可将其转换为基类类型，以使用这些方法。

如果只想改变一个方法，可以使用 `using` 关键字避免重载该方法的所有版本。在下面的代码中，`Derived` 类定义中使用了从 `Base` 类继承的一个 `overload()` 版本，并显式地重写了另一个版本。

```
class Derived : public Base
{
public:
    using Base::overload;
    void overload() override {
        cout << "Derived's overload()" << endl;
    }
};
```

`using` 子句存在一定风险。假定在 `Base` 类中添加了第三个 `overload()` 方法，本来应该在 `Derived` 类中重写这个方法。但由于使用了 `using` 子句，在派生类中没有重写这个方法不会被当作错误，`Derived` 类显式地说明“接收父类其他所有的重载方法。”

#### 警告：

为了避免歧义 bug，应该重写所有版本的重载方法。

### 3. 基类的 `private` 方法

重写 `private` 方法当然没有问题。记住方法的访问说明符会判断谁可以调用这些方法。派生类无法调用父类的 `private` 方法，并不意味着无法重写这个方法。实际上，在 C++ 中，重写 `private` 方法是一种常见模式。这种模式允许派生类定义自身的“独特性”，在基类中会引用这种独特性。注意 Java 和 C# 允许重写 `public` 和 `protected` 方法，但不能重写 `private` 方法。

例如，下面的类是汽车模拟程序的一部分，根据汽油消耗量和剩余的燃料计算汽车可以行驶的里程。`getMilesLeft()` 是所谓的模板方法。通常，模板方法都不是虚方法。模板方法通过在基类中定义一些算法的骨架，并调用虚方法来完成相关信息的查询。在子类中，可以重写这些虚方法，并修改相关算法而不会真正修改基类中已定义的算法。

```
export class MilesEstimator
{
public:
    virtual ~MilesEstimator() = default;
    int getMilesLeft() const { return getMilesPerGallon() * getGallonsLeft(); }
    virtual void setGallonsLeft(int gallons) { m_gallonsLeft = gallons; }
    virtual int getGallonsLeft() const { return m_gallonsLeft; }
private:
    int m_gallonsLeft { 0 };
    virtual int getMilesPerGallon() const { return 20; }
};
```

`getMilesLeft()` 方法根据两个方法的返回结果执行计算。下面的代码使用 `MilesEstimator` 计算两加仑汽油可以行驶的里程：

```
MilesEstimator myMilesEstimator;
myMilesEstimator.setGallonsLeft(2);
cout << format("Normal estimator can go {} more miles.",
    myMilesEstimator.getMilesLeft()) << endl;
```

代码的输出如下：

```
Normal estimator can go 40 more miles.
```

为让这个模拟程序更有趣，可引入不同类型的车辆，或许是效率更高的汽车。现有的 MilesEstimator 假定所有的汽车燃烧一加仑的汽油可以跑 20 公里，这个值是从一个单独的方法返回的，因此派生类可以重写这个方法。下面就是这样一个派生类：

```
export class EfficientCarMilesEstimator : public MilesEstimator
{
    private:
        int getMilesPerGallon() const override { return 35; }
};
```

通过重写这个 `private` 方法，新类完全修改了没有更改的现有 `public` 方法的行为。基类中的 `getMilesLeft()` 方法将自动调用 `private getMilesPerGallon()` 方法的重写版本。下面是一个使用新类的示例：

```
EfficientCarMilesEstimator myEstimator;
myEstimator.setGallonsLeft(2);
cout << format("Efficient estimator can go {} more miles.",
    myEstimator.getMilesLeft()) << endl;
```

此时的输出表明了重写的功能：

```
Efficient estimator can go 70 more miles.
```

#### 注意：

重写 `private` 或 `protected` 方法可在不做重大改动的情况下改变类的某些特性。

#### 4. 基类方法具有默认参数

派生类与基类可以具有不同的默认参数，但使用的参数取决于声明的变量类型，而不是底层的对象。下面是一个简单的派生类示例，派生类在重写的方法中提供了不同的默认参数。

```
class Base
{
public:
    virtual ~Base() = default;
    virtual void go(int i = 2) {
        cout << "Base's go with i=" << i << endl;
    }
};

class Derived : public Base
{
public:
    void go(int i = 7) override {
        cout << "Derived's go with i=" << i << endl;
    }
};
```

如果调用 `Derived` 对象的 `go()`，将执行 `Derived` 版本的 `go()`，默认参数为 7。如果调用 `Base` 对象的 `go()`，将执行 `Base` 版本的 `go()`，默认参数为 2。然而(有些怪异)，如果使用实际指向 `Derived` 对象的 `Base` 指针或 `Base` 引用调用 `go()`，将调用 `Derived` 版本的 `go()`，但实际使用 `Base` 版本的默认参数 2。下面的示例显示了这种行为：

```
Base myBase;
Derived myDerived;
Base& myBaseReferenceToDerived { myDerived };
```

```
myBase.go();
myDerived.go();
myBaseReferenceToDerived.go();
```

代码的输出如下所示：

```
Base's go with i=2
Derived's go with i=7
Derived's go with i=2
```

产生这种行为的原因是C++根据表达式的编译期类型(而非运行期类型)绑定默认参数。在C++中，默认参数不会被“继承”。如果上面的 Derived 类没有像父类那样提供默认参数，就用新的非零参数版本重载 go() 方法。

#### 注意：

当重写具有默认参数的方法时，也应该提供默认参数，这个参数的值应该与基类版本相同。建议使用命名的常量作为默认值，这样可在派生类中使用同一个命名的常量。

## 5. 基类方法具有不同的访问级别

可以采用两种方法修改方法的访问级别——可以加强限制，也可放宽限制。在C++中，这两种方法的意义都不大，但是这么做也是有合理原因的。

为加强某个方法(或数据成员)的限制，有两种方法。一种方法是修改整个基类的访问说明符，本章后面将讲述这种方法。另一种方法是在派生类中重新定义访问限制，下面的 Shy 类演示了这种方法：

```
class Gregarious
{
public:
    virtual void talk() {
        cout << "Gregarious says hi!" << endl;
    };

class Shy : public Gregarious
{
protected:
    void talk() override {
        cout << "Shy reluctantly says hello." << endl;
    };
}
```

Shy 类中 protected 版本的 talk() 方法适当地重写 Gregarious::talk() 方法。任何客户代码试图使用 Shy 对象调用 talk() 都会导致编译错误。

```
Shy myShy;
myShy.talk(); // Error! Attempt to access protected method.
```

然而，这个方法并不是完全受保护的。可使用 Gregarious 引用或指针访问 protected 方法。

```
Shy myShy;
Gregarious& ref { myShy };
ref.talk();
```

上面代码的输出如下：

```
Shy reluctantly says hello.
```

这说明在派生类中将方法设置为 protected 实际上是重写了这个方法(因为可以正确地调用这个方

法的派生类版本), 此外还证明如果基类将方法设置为 public, 就无法完整地强制访问 protected 方法。

**注意:**

无法(也没有很好的理由)限制访问基类的 public 方法。

**注意:**

上例重新定义了派生类中的方法, 因为它希望显示另一条消息。如果不希望修改实现, 只想改变方法的访问级别, 首选方法是在具有所需访问级别的派生类定义中添加 using 语句。

在派生类中放宽访问限制就比较容易(也更有意义)。最简单的方法是提供一个 public 方法来调用基类的 protected 方法, 如下所示。

```
class Secret
{
protected:
    virtual void dontTell() { cout << "I'll never tell." << endl; }

class Blabber : public Secret
{
public:
    virtual void tell() { dontTell(); }
};
```

调用 Blabber 对象的 public 方法 tell() 的客户代码可有效地访问 Secret 类的 protected 方法。当然, 这并未真正改变 dontTell() 的访问级别, 只是提供了访问这个方法的公共方式。

也可在 Blabber 派生类中显式地重写 dontTell(), 并将这个方法设置为 public。这样做比降低访问级别更有意义, 因为当使用基类指针或引用时, 可以清楚地表明发生的事情。例如, 假定 Blabber 类实际上将 dontTell() 方法设置为 public:

```
class Blabber : public Secret
{
public:
    void dontTell() override { cout << "I'll tell all!" << endl; }
```

现在调用 Blabber 对象的 dontTell() 方法:

```
myBlabber.dontTell(); // Outputs "I'll tell all!"
```

如果不想更改重写方法的实现, 只想更改访问级别, 可使用 using 子句。例如:

```
class Blabber : public Secret
{
public:
    using Secret::dontTell;
};
```

这也允许调用 Blabber 对象的 dontTell() 方法, 但这一次, 输出将会是 “I'll never tell”。

```
myBlabber.dontTell(); // Outputs "I'll never tell."
```

然而, 在上述情况下, 基类中的 protected 方法仍然是受保护的, 因为使用 Secret 指针或引用调用 Secret 类的 dontTell() 方法将无法通过编译。

```

Blabber myBlabber;
Secret& ref { myBlabber };
Secret* ptr { &myBlabber };
ref.dontTell(); // Error! Attempt to access protected method.
ptr->dontTell(); // Error! Attempt to access protected method.

```

**注意：**

修改方法访问级别的唯一真正有用的方式是对 `protected` 方法提供较宽松的访问限制。

### 10.6.5 派生类中的复制构造函数和赋值运算符

第9章讲过，在类中使用动态内存分配时，提供复制构造函数和赋值运算符是良好的编程习惯。当定义派生类时，必须注意复制构造函数和 `operator=`。

如果派生类没有任何需要使用非默认复制构造函数或 `operator=` 的特殊数据(通常是指针)，无论基类是否有这类数据，都不需要它们。如果派生类省略了复制构造函数或 `operator=`，派生类中指定的数据成员就使用默认的复制构造函数或 `operator=`，基类中的数据成员使用基类的复制构造函数或 `operator=`。

另外，如果在派生类中指定了复制构造函数，就需要显式地链接到父类的复制构造函数，下面的代码演示了这一内容。如果不这么做，将使用默认构造函数(不是复制构造函数！)初始化对象的父类部分。

```

class Base
{
public:
    virtual ~Base() = default;
    Base() = default;
    Base(const Base& src) { }
};

class Derived : public Base
{
public:
    Derived() = default;
    Derived(const Derived& src) : Base { src } {}
};

```

与此类似，如果派生类重写了 `operator=`，则几乎总是需要调用父类版本的 `operator=`。唯一的例外是因为某些奇怪的原因，在赋值时只想给对象的一部分赋值。下面的代码显示了如何在派生类中调用父类的赋值运算符。

```

Derived& Derived::operator=(const Derived& rhs)
{
    if (&rhs == this) {
        return *this;
    }
    Base::operator=(rhs); // Calls parent's operator=.
    // Do necessary assignments for derived class.
    return *this;
}

```

**警告：**

如果派生类不指定自己的复制构造函数或 operator=，基类的功能将继续运行。否则，就需要显式引用基类版本。

**注意：**

如果在继承层次结构中需要复制功能，专业 C++ 开发人员惯用的做法是实现多态 clone() 方法，因为不能完全依靠标准复制构造函数和复制赋值运算符来满足需要。第 12 章“使用模板编写泛型代码”将讨论多态的 clone() 方法。

### 10.6.6 运行期类型工具

相对于其他面向对象语言，C++ 以编译期为主。如前所述，重写方法是可行的，这是由于方法和实现之间的间隔，而不是由于对象有关于自身所属类的内建信息。

然而在 C++ 中，有些特性提供了对象的运行期视角。这些特性通常归属于一个名为运行期类型信息(Run Time Type Information, RTTI)的特性集。RTTI 提供了许多有用的特性，用于判断对象所属的类。其中一个特性是本章前面说过的 dynamic\_cast()，可以在 OO 层次结构中进行安全的类型转换。本章前面讨论过这一点。如果使用类上的 dynamic\_cast()，但没有虚表，即没有虚方法，将导致编译错误。

RTTI 的第二个特性是 typeid 运算符，这个运算符可在运行期查询对象，从而判别对象的类型。大多数情况下，不应该使用 typeid，因为最好用虚方法处理基于对象类型运行的代码。下面的代码使用了 typeid，根据对象的类型输出消息。

```
import <typeinfo>

class Animal { public: virtual ~Animal() = default; };
class Dog : public Animal {};
class Bird : public Animal {};

void speak(const Animal& animal)
{
    if (typeid(animal) == typeid(Dog)) {
        cout << "Woof!" << endl;
    } else if (typeid(animal) == typeid(Bird)) {
        cout << "Chirp!" << endl;
    }
}
```

一旦看到这样的代码，就应该立即考虑用虚方法重新实现该功能。在此情况下，更好的实现是在 Animal 类中声明一个 speak() 虚方法。Dog 类会重写这个方法，输出“Woof! ”；Bird 类也会重写这个方法，输出“Chirp! ”。这种方式更适合面向对象程序，会将与对象有关的功能给予这些对象。

**警告：**

类至少有一个虚方法(即类具有一张虚函数表)， typeid 运算符才能正常运行。另外， typeid 运算符也会从实参中去除引用和 const 限定符。

typeid 运算符的主要价值之一在于日志记录和调试。下面的代码将 typeid 用于日志记录。logObject() 函数将“可记录”对象作为参数。设计是这样的：任何可记录的对象都从 Loggable 类中继承，都支持 getLogMessage() 方法。

```

class Loggable
{
public:
    virtual ~Loggable() = default;
    virtual std::string getLogMessage() const = 0;
};

class Foo : public Loggable
{
public:
    std::string getLogMessage() const override { return "Hello logger."; }
};

void logObject(const Loggable& loggableObject)
{
    cout << typeid(loggableObject).name() << ": ";
    cout << loggableObject.getLogMessage() << endl;
}

```

`logObject()`函数首先将对象所属类的名称写到输出流，随后是日志信息。这样以后阅读日志时，就可以看出文件每一行涉及的对象。使用 `Foo` 实例调用 `logObject()` 函数时，Microsoft Visual C++ 2017 生成的输出如下所示。

```
class Foo: Hello logger.
```

可以看到，`typeid` 运算符返回的名称是 `class Foo`。但这个名称因编译器而异。比如，若用 GCC 编译相同的代码，输出如下所示：

```
3Foo: Hello logger.
```

#### 注意：

如果不是为了日志记录或调试而使用 `typeid`，应该考虑用虚方法替代 `typeid`。

### 10.6.7 非 `public` 继承

在前面的所有示例中，总是用 `public` 关键字列出父类。父类是否可以是 `private` 或 `protected`？实际上可以这样做，尽管二者并不像 `public` 那样普遍。如果没有为父类指定任何访问说明符，就说明是类的 `private` 继承、结构的 `public` 继承。

将父类的关系声明为 `protected`，意味着在派生类中，基类的所有 `public` 方法和数据成员都成为受保护的。与此类似，指定 `private` 继承意味着基类的所有 `public`、`protected` 方法和数据成员在派生类中都成为私有的。

使用这种方法统一降低父类的访问级别有许多原因，但多数原因都是层次结构的设计缺陷。有些程序员滥用这一语言特性，经常与多重继承一起实现类的“组件”。不是让 `Airplane` 类包含引擎数据成员和机身数据成员，而将 `Airplane` 类作为 `protected` 引擎和 `protected` 机身。这样，对于客户代码来说，`Airplane` 对象看上去并不像引擎或机身（因为一切都是受保护的），但在内部可以使用引擎和机身的功能。

#### 注意：

非 `public` 继承很少见，建议慎用这一特性，因为多数程序员并不熟悉它。

### 10.6.8 虚基类

本章前面学习了歧义父类，当多个基类拥有公共父类时，就会发生这种情况，如图 10-12 所示。建议的解决方案是让共享的父类本身没有任何自有功能。这样就永远无法调用这个类的方法，因此也就不存在歧义。

如果希望被共享的父类拥有自己的功能，C++ 提供了另一种机制来解决这个问题。如果被共享的基类是一个虚基类(virtual base class)，就不存在歧义。以下代码在 Animal 基类中添加了 sleep() 方法，并修改了 Dog 和 Bird 类，从而将 Animal 作为虚基类继承。如果不使用 virtual 关键字，用 DogBird 对象调用 sleep() 会产生歧义，从而导致编译错误。因为 DogBird 对象有 Animal 类的两个子对象，一个来自 Dog 类，另一个来自 Bird 类。然而，如果 Animal 被作为虚基类，DogBird 对象就只有 Animal 类的一个子对象，因此调用 sleep() 也就不存在歧义。

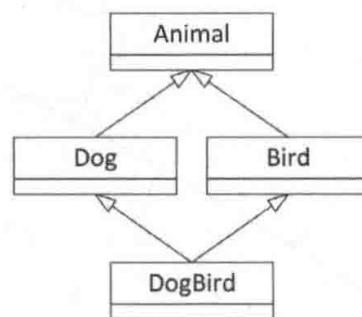


图 10-12 歧义父类

```

class Animal
{
public:
    virtual void eat() = 0;
    virtual void sleep() { cout << "zzzz...." << endl; }
};

class Dog : public virtual Animal
{
public:
    virtual void bark() { cout << "Woof!" << endl; }
    void eat() override { cout << "The dog ate." << endl; }
};

class Bird : public virtual Animal
{
public:
    virtual void chirp() { cout << "Chirp!" << endl; }
    void eat() override { cout << "The bird ate." << endl; }
};

class DogBird : public Dog, public Bird
{
public:
    void eat() override { Dog::eat(); }
};

int main()
{
    DogBird myConfusedAnimal;
    myConfusedAnimal.sleep(); // Not ambiguous because of virtual base class
}
  
```

#### 注意：

虚基类是在类层次结构中避免歧义的好办法。唯一的缺点是许多 C++ 程序员不熟悉这个概念。

## 10.7 类型转换

第1章介绍了C++中的基本类型，而第8~10章介绍了如何使用类编写自定义的类型。本节将探讨如何将一种类型转换为另一种类型。

C++提供了5种特定的强制类型转换：`const_cast()`、`static_cast()`、`reinterpret_cast()`、`dynamic_cast()`和`std::bit_cast()`(从C++20开始)。`const_cast()`在第1章进行了讨论。第1章还介绍了`static_cast()`，用于在某些基本类型之间进行转换，但在继承的上下文中，还有更多内容要介绍。鉴于已经能够熟练地编写类并了解类的继承，现在仔细研究这些强制类型转换。

注意，旧的C风格的强制类型转换(如`(int)myFloat`)仍可以在C++中使用，并且仍在现有代码库中广泛使用。C风格的强制类型转换涵盖了所有4种C++强制类型转换，因此它们更容易出错，因为要实现的目标并不总是很明显，并且最终可能会带来意想不到的结果。强烈建议仅在新代码中使用C++风格的强制转换，因为它们更安全，并且在代码中在语法上更加突出。

### 10.7.1 static\_cast()

可以使用`static_cast()`执行语言直接支持的显式转换。比如，如果编写一个算术表达式，需要将`int`转换为`double`以避免整数除法，则可以使用`static_cast()`。在本示例中，只对`i`使用`static_cast()`就足够了，因为这会使两个操作数中的一个变成双精度，从而确保C++执行浮点数除法。

```
int i { 3 };
int j { 4 };
double result { static_cast<double>(i) / j };
```

还可以使用`static_cast()`执行显式转换，这是用户定义的构造函数或转换例程允许的。比如，如果类A有一个接受类B对象的构造函数，则可以使用`static_cast()`将B对象转换为A对象。但是，在大多数情况下，需要这种行为时，编译器会自动执行转换。

`static_cast()`的另一个用途是在继承层次结构中执行向下强制转换，示例如下：

```
class Base
{
public:
    virtual ~Base() = default;
};

class Derived : public Base
{
public:
    virtual ~Derived() = default;
};

int main()
{
    Base* b { nullptr };
    Derived* d { new Derived{} };
    b = d; // Don't need a cast to go up the inheritance hierarchy.
    d = static_cast<Derived*>(b); // Need a cast to go down the hierarchy.

    Base base;
    Derived derived;
    Base& br { derived };
}
```

```
Derived& dr { static_cast<Derived&>(br) };
};
```

这些强制类型转换既可以使用指针也可以使用引用。它们不使用对象。

注意，使用 `static_cast()` 的强制类型转换不会执行运行期类型检查。它们允许将任何 `Base` 指针转换为 `Derived` 指针，或是将 `Base` 引用转换为 `Derived` 引用。即使在运行期，`Base` 实际上不是 `Derived` 也是如此。比如，编译并执行以下代码，但使用指针 `d` 可能会导致潜在的灾难性故障，包括在对象边界之外进行内存覆盖。

```
Base* b { new Base{} };
Derived* d { static_cast<Derived*>(b) };
```

要在运行期类型检查时安全地执行转换，请使用 `dynamic_cast()`，本章稍后将对此进行说明。

`static_cast()` 不是万能的。不能将一种类型的指针 `static_cast()` 转换为另一种不相关类型的指针；如果没有可用的转换构造函数，则无法将一种类型的对象直接 `static_cast()` 转换为另一种类型的对象；不能将常量类型的 `static_cast()` 转换为非常量类型；不能将指针 `static_cast()` 转换为整数。基本上，根据 C++ 的类型规则，不能做任何没有意义的事情。

### 10.7.2 reinterpret\_cast()

与 `static_cast()` 相比，`reinterpret_cast()` 的功能更强大，同时安全性也更低。可以使用它执行 C++ 类型规则中在技术上不允许的某些强制转换，但在某些情况下，这对程序员来说可能是有意义的。比如，即使类型不相关，也可以将一种类型的引用强制转换对另一种类型的引用。同样，可以将指针类型强制转换为任何其他指针类型，即使它们与继承层次结构无关。这通常用于将指针强制转换为 `void*` 类型的指针。这可以隐式完成，因此不需要显式强制转换。但是，将 `void*` 强制转换回正确类型的指针需要 `reinterpret_cast()`。`void*` 类型的指针只是指向内存中的某个位置。没有类型信息与 `void *` 类型的指针相关联。示例如下：

```
class X {};
class Y {};

int main()
{
    X x;
    Y y;
    X* xp { &x };
    Y* yp { &y };
    // Need reinterpret_cast for pointer conversion from unrelated classes
    // static_cast doesn't work.
    xp = reinterpret_cast<X*>(yp);
    // No cast required for conversion from pointer to void*
    void* p { xp };
    // Need reinterpret_cast for pointer conversion from void*
    xp = reinterpret_cast<X*>(p);
    // Need reinterpret_cast for reference conversion from unrelated classes
    // static_cast doesn't work.
    X& xr { x };
    Y& yr { reinterpret_cast<Y&>(x) };
}
```

`reinterpret_cast()` 并非万能；它对可以转换为什么有很多的限制。这些限制在本书不会进一步讨论，

因为建议谨慎使用这些类型的强制类型转换。通常，应谨慎使用 `reinterpret_cast()`，因为它允许在不执行任何类型检查的情况下进行转换。

#### 警告：

还可以使用 `reinterpret_cast()` 将指针转换为整数类型并返回。但是，只能将指针转换为足够容纳它的整数类型。比如，如果使用 `reinterpret_cast()` 将 64 位指针强制转换为 32 位整数，会导致编译错误。



### 10.7.3 std::bit\_cast()

C++ 20 引入了 `std::bit_cast()`，它定义在 `<bit>` 中。这是标准库中唯一的强制类型转换；其他强制转换是 C++ 语言本身的一部分。`bit_cast()` 与 `reinterpret_cast()` 类似，但它会创建一个指定目标类型的新对象，并按位从源对象复制到此新对象。它有效地将源对象的位解释为目标对象的位。`bit_cast()` 要求源对象和目标对象的大小相同，并且两者都是可复制的。示例如下：

```
float asFloat { 1.23f };
auto asUint { bit_cast<unsigned int>(asFloat) };
if (bit_cast<float>(asUint) == asFloat) { cout << "Roundtrip success." << endl; }
```

#### 注意：

普通可复制类型是，组成对象的底层字节可以复制到一个数组中(比如 `char`)。如果数组的数据随后被复制回对象，则该对象将保持其原始值。

`bit_cast()` 的一个用例是普通可复制类型的二进制 I/O。比如，可以将此类型的各个字节写入文件。当文件读回到内存时，可以使用 `bit_cast()` 正确地解释从文件中读取的字节。

### 10.7.4 dynamic\_cast()

`dynamic_cast()` 提供了对继承层次结构中的强制转换的运行期检查。可以使用它强制转换指针或引用。`dynamic_cast()` 在运行期检查底层对象的运行期类型信息。如果强制转换无效，`dynamic_cast()` 将返回空指针(对于指针)，或者抛出 `std::bad_cast` 异常(对于引用)。

比如，假设具有以下类层次结构：

```
class Base
{
public:
    virtual ~Base() = default;
};

class Derived : public Base
{
public:
    virtual ~Derived() = default;
};
```

下面的示例演示了 `dynamic_cast()` 的正确使用：

```
Base* b;
Derived* d { new Derived{} };
b = d;
d = dynamic_cast<Derived*>(b);
```

下面对于引用的 `dynamic_cast()` 将抛出异常：

```
Base base;
Derived derived;
Base& br { base };
try {
    Derived& dr { dynamic_cast<Derived&>(br) };
} catch (const bad_cast&) {
    cout << "Bad cast!" << endl;
}
```

注意，可以使用 `static_cast()` 或 `reinterpret_cast()` 在继承层次结构上执行相同的强制转换。与 `dynamic_cast()` 的区别在于：`dynamic_cast()` 执行运行期(动态)类型检查，而 `static_cast()` 和 `reinterpret_cast()` 则执行强制转换，即使转换的类型是错误的。

请记住，运行期的类型信息存储在对象的 vtable 中。因此，要使用 `dynamic_cast()`，类中必须至少具有一个虚方法。如果类没有 vtable，尝试使用 `dynamic_cast()` 会导致编译错误。比如，Microsoft Visual C++ 会出现以下的错误：

```
error C2683: 'dynamic_cast' : 'MyClass' is not a polymorphic type.
```

### 10.7.5 类型转换小结

表 10-2 总结了在不同情况下应使用的强制类型转换。

表 10-2 强制类型转换

场景	类型转换
移除 <code>const</code> 属性	<code>const_cast()</code>
语言支持的显式强制转换(比如，将 <code>int</code> 转换为 <code>double</code> ，将 <code>int</code> 转换为 <code>bool</code> )	<code>static_cast()</code>
用户定义的构造函数或转换函数支持的显式强制转换	<code>static_cast()</code>
一个类的对象转换为另一个(无关的)类的对象	<code>bit_cast()</code>
在同一继承层次结构中，一个类的指针转换为另一个类的指针	建议 <code>dynamic_cast()</code> ，或 <code>static_cast()</code>
在同一继承层次结构中，一个类的引用转换为另一个类的引用	建议 <code>dynamic_cast()</code> ，或 <code>static_cast()</code>
指向一种类型的指针转换为指向其他不相关类型的指针	<code>reinterpret_cast()</code>
一种类型的引用转换为其他不相关的类型的引用	<code>reinterpret_cast()</code>
指向函数的指针转换为指向函数的指针	<code>reinterpret_cast()</code>

## 10.8 本章小结

本章讲述了许多与继承有关的细节。学习了许多与继承有关的应用，包括代码重用和多态性。了解了许多不恰当的继承用法，其中包括设计不当的多重继承。在本章还看到了许多需要特别注意的情况。

继承是一个功能强大的语言特性，需要花费许多时间去熟悉。在学习了本章的示例，并亲自体验之后，希望在面向对象设计中，继承会成为其中可供选择的工具。

## 10.9 练习

通过完成以下习题，可以巩固本章讨论的知识点。所有练习的答案都可扫描封底二维码下载。然而，如果你困在一个练习中，在网站上查看答案之前，请先重新阅读本章的内容，试着自己找到答案。

**练习 10-1** 从练习 9-2 中获取 Person 类，并添加一个名为 Employee 的派生类。可以省略练习 9-2 中的运算符 $\langle=\rangle$ 的重载。向 Employee 类添加一个数据成员，即 employee ID。提供适当的构造函数。再从 Employee 类派生另外两个类，称为 Manager 类和 Director 类。

将所有的类(包括 Person 类)放置在名为 HR 的名称空间中。注意，可以按以下方式导出名称空间中的所有内容。

```
export namespace HR { /* ... */ }
```

**练习 10-2** 继续练习 10-1 中的解决方案，向 Person 类添加 `toString()` 方法，返回人员的字符串表示。在 Employee、Manager 和 Director 类中重写此方法，通过将其部分工作委托给父类，以此构建完整的字符串表示。

**练习 10-3** 从练习 10-2 开始，练习“人”层次结构中类的多态行为。定义一个向量用来存储员工、经理和主管的混合，并用一些测试数据填充它。最后，使用一个基于范围 `for` 的循环，对向量中的所有元素调用 `toString()`。

**练习 10-4** 在真实的公司中，员工可以晋升为经理或主管。在练习 10-3 的类层次结构中，是否可以添加对此功能的支持？

# 第 11 章

## 零碎的工作

### 本章内容

- 什么是模块，以及如何编写和使用模块
- 头文件的细节
- static 关键字的不同用法
- C 风格的变长参数列表和预处理器宏

本章讨论了 C++ 语言的一些特性，这些特性不适合在其他章节中讨论。本章首先详细讨论如何使用模块编写可重用组件，并将其与旧式头文件进行了对比。还讨论了 static 和 extern 关键字的不同用法。本章的最后一部分讨论了一些 C 风格的实用程序：变长参数列表和预处理器宏。

C++20

### 11.1 模块

模块，在第 1 章中介绍过。在前面的章节中，已经编写并使用了简单的模块。然而，关于模块还有很多事情要讲。在 C++ 20 中引入模块之前，头文件(本章稍后将讨论)是用来为可重用代码段提供接口的。但是头文件确实有很多问题，比如避免多次包含同一个头文件，以及确保头文件以正确的顺序包含。此外，只需要简单的#include(比如#include <iostream>)，就会新增上万行编译器必须处理的代码。如果多个源文件都包含#include <iostream>，那么所有的这些编译单元就会变得更大。这还仅仅只是包含了一个头文件。想象一下，如果需要包含<iostream>、<vector>、<format>等的情形。

模块解决了所有这些问题，甚至更多。模块导入的顺序并不重要。模块只需要编译一次，而不是像头文件那样需要反复编译；因此，可以大大缩短编译时间。模块中的某些修改(比如在模块接口文件中修改一个导出函数的实现)，不会触发该模块的用户重新编译(在本章后面将详细讨论)。模块不受任何外部定义的宏的影响，并且在模块内部定义的任何宏对模块外部的任何代码都不可见。

具有这些优点，如果编译器支持模块，那么新编写的代码应该使用模块。

#### 注意：

如何编译模块取决于编译器。请查阅编译器相关文档，了解如何使用特定编译器编译模块。

**注意：**

在撰写本书时，还没有完全支持模块的编译器。在一些编译器中有实验性的支持，但仍然不完整。本书在所有地方都使用模块。本书尽了最大努力确保一旦编译器完全支持模块，所有示例代码都可以编译，但由于无法编译和测试所有示例，因此可能会出现一些错误。当使用支持模块的编译器时，如果遇到任何代码示例的问题，请仔细检查本书在 [www.wiley.com/go/proc++5e](http://www.wiley.com/go/proc++5e) 上的勘误表，确认是否为已知问题。如果编译器还不支持模块，则可以将模块化代码转换为非模块化代码，如下所述。

如果想用一个尚不支持模块的编译器来编译本书中的代码示例，可以按如下方式对代码进行非模块化。

- 重命名.cppm 的模块接口文件为.h 的头文件。
- 在每个.h 头文件的顶部添加#pragma once。
- 移除 export module xyz 的声明。
- 使用#include 替换 module xyz 的声明，以包含相应的头文件。
- 使用适当的#include 指令替换 import 和 export import 的声明。
- 移除任何 export 关键字。
- 移除所有出现的 module;，它表示一个全局模块片段的开始。
- 如果函数定义或变量定义出现在.h 的头文件中，则在它前面添加 inline 关键字。

### 11.1.1 模块接口文件

模块接口文件，是为模块提供的功能定义的接口。模块接口文件通常以.cppm 作为扩展名。模块接口以声明开头，声明该文件正在定义一个具有特定名称的模块。这称为模块声明。模块的名称可以是任何有效的 C++ 标识符。名称中可以包含点(.)，但不能以点(.)开头或结尾，也不能在一行中包含多个点(.)。有效名称的示例有 datamodel、mycompany.datamodel、mycompany.datamodel.core、datamodel\_core 等。

**注意：**

目前，模块接口文件还没有标准化的扩展名。然而，大多数编译器都支持.cppm(C++模块)扩展，所以本书使用的就是这个扩展。请查阅编译器的相关文档，了解需要使用哪个扩展。

模块需要显式地声明要导出的内容，即当用户代码导入模块时，哪些内容应该是可见的。使用 export 关键字从模块中导出实体(比如类、函数、常量、其他模块等)。任何没有从模块中导出的内容，只在该模块中可见。所有导出实体的集合称为模块接口。

下面是一个名为 Person.cppm 的模块接口文件的示例，它定义了 person 模块，并导出了 person 类。注意，它导入了<string>提供的功能。

```
export module person; // Module declaration

import <string>; // Import declaration

export class Person // Export declaration
{
    public:
        Person(std::string firstName, std::string lastName)
            : m(firstName { std::move(firstName) })
            , m(lastName { std::move(lastName) }) {}
        const std::string& getFirstName() const { return m.firstName; }
```

```

    const std::string& getLastName() const { return m_lastName; }
private:
    std::string m(firstName);
    std::string m.lastName;
};

```

可以通过以下方式导入 Person 模块，来使用 Person 类(test.cpp):

```

import person;      // Import declaration for person module
import <iostream>;
import <string>;   // For operator<< for std::string

using namespace std;

int main()
{
    Person person { "Kole", "Webb" };
    cout << person.getLastName() << ", " << person.getFirstName() << endl;
}

```

所有的 C++头文件(比如<iostream>、<vector>、<string>等)，都是所谓的可导入头文件，可以通过 import 声明导入。正如第 1 章介绍的，C++中可用的 C 的头文件不能保证是可导入的。为了安全起见，在 C 的头文件中使用#include 替换 import 声明。这样的#include 指令应该放在所谓的全局模块片段中，它必须出现在任何命名模块的声明之前，以匿名模块的声明开始。全局模块片段只能包含预处理指令，比如#include。这样的全局模块片段和注释是唯一允许出现在命名模块声明之前的内容。比如，如果需要使用 C 的头文件<cstddef>中的功能，可以如下所示：

```

module;                      // Start of the global module fragment
#include <cstddef>          // Include legacy header files

export module person; // Named module declaration

import <string>;

export class Person { /* ... */ };

```

在标准术语中，从已命名模块声明到文件末尾的所有内容称为模块权限。

几乎所有内容都可以从模块导出，只要它有一个名称。比如类定义、函数原型、类枚举类型、using 声明和指令、名称空间等。如果使用 export 关键字显式导出名称空间，则该名称空间中的所有内容也将自动导出。比如，下面的代码片段导出整个 DataModel 名称空间；因此，没有必要显式地导出单个类和类型别名：

```

export module datamodel;
import <vector>;
export namespace DataModel
{
    class Person { /* ... */ };
    class Address { /* ... */ };
    using Persons = std::vector<Person>;
}

```

还可以使用导出块(export block)导出整个声明块。示例如下：

```

export
{

```

```

namespace DataModel
{
    class Person { /* ... */ };
    class Address { /* ... */ };
    using Persons = std::vector<Person>;
}

```

### 11.1.2 模块实现文件

一个模块可以被拆分为一个模块接口文件和一个或多个模块实现文件。模块实现文件通常以.cpp 作为扩展名。可以自由决定将哪些实现移到模块实现文件中，以及将哪些实现留在模块接口文件中。一种选择是将所有函数和方法实现移到模块实现文件中，只在模块接口文件中保留函数原型、类定义等。另一种选择是将小函数和方法的实现留在接口文件中，而将其他函数和方法的实现移到实现文件中。这里有很大的灵活性。

模块实现文件中同样包含一个已命名模块的声明，用于指定实现所针对的模块，但没有 export 关键字。比如，可以将前面的 person 模块拆分为接口和实现文件，如下所示。下面是模块接口文件：

```

export module person; // Module declaration

import <string>

export class Person
{
    public:
        Person(std::string firstName, std::string lastName);
        const std::string& getFirstName() const;
        const std::string& getLastName() const;
    private:
        std::string m(firstName);
        std::string m(lastName);
};

```

实现现在位于模块实现文件 Person.cpp 中：

```

module person; // Module declaration, but without the export keyword

using namespace std;

Person::Person(string firstName, string lastName)
    : m(firstName) { move(firstName) }, m(lastName) { move(lastName) }

const string& Person::getFirstName() const { return m(firstName); }
const string& Person::getLastName() const { return m(lastName); }

```

注意，实现文件没有 person 模块的导入声明。module person 的声明隐式包含 import person 的声明。还要注意的是，实现文件中没有<string>的导入声明，即使它在方法的实现中使用了 std::string。正是由于隐式的 import person，并且因为这个实现文件是同一个 person 模块的一部分，所以它隐式地从模块接口文件中继承了<string>的导入声明。与之相反，向 test.cpp 文件添加 import person 的声明并不会隐式地继承<string>的导入声明，因为 test.cpp 不是 person 模块的一部分。关于这一点还有很多展

开的，这是即将到来的“可见性与可访问性”章节的主题。

**注意：**

模块接口和模块实现文件中的所有导入声明都必须位于文件的顶部，在已命名模块声明之后，但需要在任何其他声明之前。

类似于模块接口文件，如果需要在模块实现文件中，使用`#include`指令导入已有的头文件，则应该将它们放在全局模块片段中，并使用与模块接口文件相同的语法。

**警告：**

模块实现文件不能导出任何内容；只有模块接口文件可以。

### 11.1.3 从实现中分离接口

当使用头文件而不是模块时，强烈建议只把声明放在头文件(.h)中，并将所有实现移到源文件(.cpp)中。原因之一是可以缩短编译时间。如果将实现放在头文件中，任何的代码变更，哪怕只是修改注释，都需要重新编译包含该头文件的所有其他源文件。对于某些头文件，这可能会影响整个代码库，导致整个程序完全重新编译。将实现放到源文件中，在不涉及头文件的情况下，对这些实现进行更改，这样只需要重新编译单个源文件。

模块的工作方式是不同的。模块接口(module interface)仅由类定义、函数原型等组成，虽然这些实现在模块接口文件中，但模块接口中不包含任何函数或方法的实现。因此，只要变更不涉及模块接口的部分，变更模块接口文件中的函数/方法的实现是不需要重新编译该模块的，比如函数头(函数名、参数列表和返回类型)。但有两个例外，使用`inline`关键字标记的函数或方法和模板的定义。对于这两者，编译器需要在编译使用它们的用户代码时知道其完整实现。因此，内联函数/方法或模板定义的任何变更都可能引发用户代码的重新编译。

**注意：**

当头文件中的类定义中包含方法的实现，那么这些方法是隐式的内联，尽管没有使用`inline`关键字标识它们。对于在模块接口文件中的类定义里的方法实现来说，却又不一样了。如果要变成内联方法，就需要使用`inline`关键字显式标识它们。

尽管从技术上讲，不再需要将接口从实现中分离，但在某些情况下，我仍建议这么做。其主要目的是创建整洁和易读的接口。只要不会模糊接口和不会增加用户快速掌握公共接口提供的内容的难度，函数的实现就可以保留在接口中。例如，如果一个模块有一个规模很大的公共接口，那么最好不要使用模糊的接口实现，这样可以更好地了解所提供的内容。尽管如此，规模小的`getter`和`setter`函数仍然可以保留在接口中，因为不会影响接口的可读性。

将接口从实现中分离可以通过几种方式实现。一种是将模块拆分为接口文件和实现文件，如上一节所讨论的。另一种是在单个模块的接口文件中拆分接口和实现。例如，下面是定义在单个模块接口文件(`person.cppm`)中的`Person`类，但该类的实现是从接口中分离的。

```
export module person;
import <string>;
// Class definition
export class Person
{
    public:
```

```

        Person(std::string firstName, std::string lastName);
        const std::string& getFirstName() const;
        const std::string& getLastName() const;
    private:
        std::string m(firstName);
        std::string m(lastName);
    };
// Implementations
Person::Person(std::string firstName, std::string lastName)
    : m(firstName) { std::move(firstName) }, m(lastName) { std::move(lastName) } {}
const std::string& Person::getFirstName() const { return m(firstName); }
const std::string& Person::getLastName() const { return m(lastName); }

```

### 11.1.4 可见性和可访问性

正如前面介绍的，当在不属于 person 模块的另一个源文件(比如 test.cpp 文件)中导入 person 模块时，不会隐式地从 person 模块的接口文件中继承<string>的导入声明。因此，如果在 test.cpp 中没有显式导入<string>，那么 std::string 是不可见的，这意味着以下高亮的代码将无法正常编译。

```

import person;
int main()
{
    std::string str;
    Person person { "Kole", "Webb" };
    const std::string& lastName { person.getLastName() };
}

```

尽管如此，即使在 test.cpp 中没有显式导入<string>，下面的代码也可以正常工作。

```

const auto& lastName { person.getLastName() };
auto length { lastName.length() };

```

为什么会这样呢？在 C++ 中，实体的可见性(visibility)和可访问性(reachability)是有区别的。通过导入 person 模块，<string>中的功能变得可访问，但并不可见；可访问类的成员函数自动变得可见。所有这些都意味着可以使用<string>中的某些功能，比如使用自动类型推断(auto type deduction)和调用 length() 等方法，将 getLastName() 的结果存储在变量中。

为了使 std::string 名称在 test.cpp 中可见，需要显式导入<string>。当想要使用(比如下面代码中的运算符<<时，这样的显式导入是必要的。这是因为运算符<<不是 std::string 的方法，而是只有通过导入<string>才可见的非成员函数。

```

cout << person.getLastName() << endl;

```

### 11.1.5 子模块

C++ 标准中并没有描述子模块(submodule)；但是，标准允许在模块名中使用符号点(.)，这就可以在任何想要的层次结构中构建模块。比如，之前的章节给出的名称空间 DataModel 示例如下：

```

export module datamodel;
import <vector>;
export namespace DataModel
{
    class Person { /* ... */ };
    class Address { /* ... */ };
}

```

```

    using Persons = std::vector<Person>;
}

```

类 Person 和类 Address 都位于名称空间 DataModel 和模块 datamodel 中。这可以通过定义两个子模块来重新进行构造：datamodel.person 和 datamodel.address。子模块 datamodel.person 的模块接口文件如下：

```

export module datamodel.person; // datamodel.person submodule
export namespace DataModel { class Person { /* ... */ }; }

```

下面是子模块 datamodel.address 的模块接口文件：

```

export module datamodel.address; // datamodel.address submodule
export namespace DataModel { class Address { /* ... */ }; }

```

最后，模块 datamodel 的定义如下。它导入并立即导出两个子模块。

```

export module datamodel; // datamodel module
export import datamodel.person; // Import and export person submodule
export import datamodel.address; // Import and export address submodule
import <vector>;
export namespace DataModel { using Persons = std::vector<Person>; }

```

当然，子模块中类的方法实现也可以放入模块实现文件中。比如，假设类 Address 有一个默认构造函数，它只打印一条语句到标准输出流中；这个实现可以放在文件 datamodel.address.cpp 中：

```

module datamodel.address; // datamodel.address submodule
import <iostream>;
using namespace std;
DataModel::Address::Address() { cout << "Address::Address()" << endl; }

```

使用子模块构造代码的好处是：用户可以一次导入他们想要使用的模块特定部分或全部内容。例如，如果用户只对类 Address 的使用感兴趣，那么下面的 import 声明就足够了。

```
import datamodel.address;
```

另一方面，如果客户端代码需要访问模块 datamodel 中的所有内容，那么下面的导入声明是最简单的。

```
import datamodel;
```

### 11.1.6 模块划分

另一种构建模块的方法是将它们拆分到单独的分区中。子模块和分区之间的区别是：子模块的构造对模块的用户来说是可见的，允许用户有选择地只导入想要使用的子模块。另一方面，分区用于在内部构造模块。分区不对模块的用户公开。在模块接口分区文件(module interface partition file)中声明的所有分区，最终必须由主模块接口文件导出。一个模块通常只有一个这样的主模块接口文件，那就是包含导出模块名称声明的接口文件。

模块分区名创建时使用冒号将模块名和分区名分隔开。分区名可以是任何合法的标识符。例如，前一节中的 DataModel 模块可以使用分区而不是子模块来重新构造。下面是在模块接口分区文件 datamodel.person.cppm 的 person 分区：

```

export module datamodel:person; // datamodel:person partition
export namespace DataModel { class Person { /* ... */ }; }

```

下面是包含默认构造函数的 address 分区：

```
export module datamodel:address; // datamodel:address partition
export namespace DataModel
{
    class Address
    {
        public:
            Address();
            /* ... */
    };
}
```

但是，将实现文件与分区结合使用时需要注意：只有一个具有特定分区名称的文件。因此，以下声明开头的实现文件的格式是不正确的。

```
module datamodel:address;
```

相反，可以将 address 分区的实现放在 datamodel 模块的实现文件中，示例如下：

```
module datamodel; // Not datamodel:address!
import <iostream>;
using namespace std;
DataModel::Address::Address() { cout << "Address::Address()" << endl; }
```

#### 警告：

在初始化引用之后无法改变引用所指的变量，而只能改变该变量的值。

多个文件不能具有相同的分区名称。因此，具有多个相同分区名的模块接口分区文件是非法的，并且对于模块接口分区文件中声明的实现，也不能出现在具有相同分区名的实现文件中。相反，只需要将这些实现放在模块的实现文件中即可。

需要记住的重点是，在编写分区结构中的模块时，每个模块接口的分区最终都必须由主模块接口文件直接或间接导出。要导入分区，只需要指定(以冒号作为前缀)分区名称，比如 import:person。而 import datamodel:person 这样的则是非法的。记住，分区不会对模块的用户公开。分区只能在内部构造模块。因此，用户不能导入特定的分区；只能导入整个模块。分区只能在模块内部导入，因此在冒号之前指定模块名称是多余的(也是非法的)。下面是 datamodel 模块的主模块接口文件：

```
export module datamodel; // datamodel module (primary module interface file)
export import :person; // Import and export person partition
export import :address; // Import and export address partition
import <vector>;
export namespace DataModel { using Persons = std::vector<Person>; }
```

这个分区结构的 datamodel 模块可以按以下方式使用：

```
import datamodel;
int main() { DataModel::Address a; }
```

#### 注意：

分区用于在内部构造模块。分区在模块外部不可见。因此，模块的用户无法导入特定的分区；只能导入整个模块。

前面已经解释过模块名称声明中隐式包括导入名称声明，但分区不是这样的。

比如 `datamodel:person` 分区没有隐式导入 `datamodel` 的声明。在本例中，甚至不允许在 `datamodel:person` 接口分区文件中显式导入 `datamodel`。因为这样做将导致循环依赖：`datamodel` 接口文件中包含 `import:person` 的声明，而 `datamodel:person` 接口分区文件中包含 `import datamodel` 声明。

要打破这种循环依赖关系，可以将 `datamodel:person` 分区需要的功能从 `datamodel` 接口文件中转移至另一个分区，随后通过 `datamodel:person` 接口分区文件和 `datamodel` 接口文件导入该分区。

## 实现分区

分区不需要在模块的接口分区文件中声明，但可以在模块的实现分区文件(扩展名为.cpp 的常规源码文件)中声明。这种情况下，它是实现分区，有时也称为内部分区。主模块的接口文件不会导出这样的分区。

例如，假设具有以下 `math` 主模块的接口文件(`math.cppm`)：

```
export module math; // math module declaration
export namespace Math
{
    double superLog(double z, double b);
    double lerchZeta(double lambda, double alpha, double s);
}
```

进一步假设，`math` 函数的实现需要一些辅助函数，这些辅助函数不能被模块导出。实现分区是放置此类辅助函数的理想场所。下面在名为 `math_helpers.cpp` 的文件中定义了这类实现分区：

```
module math:details; // math:details implementation partition
double someHelperFunction(double a) { return /* ... */; }
```

其他的 `math` 模块实现文件可以通过导入此实现分区来访问这些辅助函数。比如，`math` 模块的实现文件(`math.cpp`)可能如下所示：

```
module math;
import :details;
double Math::superLog(double z, double b) { return /* ... */; }
double Math::lerchZeta(double lambda, double alpha, double s) { return /* ... */; }
```

当然，只有当多个其他源文件都使用这些辅助函数的情况下，将实现分区与辅助函数一起使用才会有效果。

### 11.1.7 头文件单元

导入模块时，可以使用导入声明，示例如下：

```
import person;
```

如果有遗留代码，比如定义 `Person` 类的 `person.h` 头文件，则可以通过将其转换为适当的模块(`person.cppm`)进行模块化，并导入声明使其对用户代码可用。但是，有时不能模块化这类头文件。也许对于还不支持模块的编译器，`Person` 类应该保持可用。或者 `person.h` 头文件是不能修改的第三方库的一部分。在这种情况下，可以如下所示，直接导入头文件：

```
import "person.h"; // Can also include a relative or absolute path.
```

有了这样的声明，`person.h` 头文件中的所有内容都将隐式导出。此外，头文件中定义的宏对于用户代码可见，但实际模块中却并非如此。

相比#include 头文件，导入头文件可以提高构建效率，因为 person.h 头文件将隐式转换为模块，因此只需要编译一次，而不是在源文件中#include 头文件时每次都进行编译。因此，它可以作为支持所谓的预编译头文件的标准化方式，而不是使用依赖于编译器的预编译头文件的支持。

对于每个命名头文件的导入声明，编译器必须创建一个模块，该模块具有类似于头文件定义的导出接口。这称为头文件单元。此过程取决于编译器，因此请参阅编译器的文档，了解如何使用头文件单元。

## 11.2 头文件

在介绍 C++20 的模块之前，头文件被用作一种为子系统或代码段提供接口的机制。第 1 章简要介绍了头文件。基本上，头文件最常见的用法是用于声明在其他地方定义的函数。函数声明告诉编译器如何调用函数，声明形参的数量和类型，以及函数的返回类型。定义则包含该函数的实际代码。在 C++ 中，声明通常放入头文件中，扩展名为.h；而定义通常放入源文件中，扩展名为.cpp。本书在任何地方都将使用 C++ 20 的模块，但本节将会简要讨论使用头文件时一些复杂的情况，比如避免重复定义和循环依赖，因为在系统遗留代码中可能会遇到这些问题。

### 11.2.1 重复定义

假设 A.h 包含 Logger.h 头文件，Logger.h 头文件中定义了 Logger 类；B.h 也包含 Logger.h。如果具有一个名为 App.cpp 的源文件，其中同时包含 A.h 和 B.h，将会得到 Logger 类的重复定义，因为 Logger.h 头文件包含在 A.h 和 B.h 中。

这种重复定义的问题可以通过一种称为包含保护(也称为头文件保护)的机制来避免。下面的代码段显示了具有包含保护的 Logger.h 头文件。在每个头文件的开头，#ifndef 指令检查是否有某个标识符没有被定义。如果标识符已定义，那么编译器将会跳转至匹配的#endif(通常将其放置在文件的末尾)。如果没有定义标识符，将会继续定义该标识符，以便跳过同一文件的后续包含。

```
#ifndef LOGGER_H
#define LOGGER_H

class Logger { /* ... */ };

#endif // LOGGER_H
```

另外，现在几乎所有的编译器都支持#pragma once 指令，它取代了 include 保护指令。在头文件的开头放置#pragma once，可以确保它只会被导入一次，从而避免了由于多次包含头而导致的重复定义。示例如下：

```
#pragma once

class Logger { /* ... */ };
```

### 11.2.2 循环依赖

避免头文件问题的另一个工具是前向声明(forward declaration)。如果需要引用一个类，但不能包含它的头文件(比如，它严重依赖于正在编写的类)，则可以(通过前向声明)告诉编译器该类存在，而不需要通过#include 提供正式的定义。当然，不能在代码中实际使用该类，因为编译器对它还一无所

知，只知道已命名的类将在所有内容链接在一起后存在于内存中。但仍然可以在代码中，通过指针和引用来引用前向声明的类。还可以声明按值返回的此类前向声明的类，或者声明一个函数，该函数的参数按值传递，类型为前向声明的类。当然，对于定义了函数的代码和调用该函数的任何代码，都需要包含正确定义了前向声明的类的头文件。

比如，假设 `Logger` 类使用另一个名为 `Preferences` 的类来跟踪用户的设置。`Preferences` 类可能反过来使用 `Logger` 类，因此会具有循环依赖关系，而且不能通过包含保护来解决。在这种情况下，需要使用前向声明。在下面的代码中，`Logger.h` 头文件使用了 `Preferences` 类的前向声明，因此可以在不包括其头文件的情况下，引用 `Preferences` 类。

```
#pragma once

#include <string_view>

class Preferences; // forward declaration

class Logger
{
public:
    void setPreferences(const Preferences& prefs);
    void logError(std::string_view error);
};
```

建议尽可能在头文件中使用前向声明，而不是包含其他头文件。这么做可以减少编译和重新编译的时间，因为它打破了当前头文件对其他头文件的依赖。当然，实现文件还是需要包含正确的头文件，即已前向声明的类型头文件；否则，它将无法通过编译。

### 11.2.3 查询头文件是否存在

若要查询某个头文件是否存在，请使用 `_has_include("filename")` 或 `_has_include(<filename>)` 预处理器常量表达式。如果头文件已经存在，则返回值为 1，否则为 0。例如，在 C++17 完全批准使用 `<optional>` 头文件之前，某些编译器已经在 `<experimental/optional>` 中具有初步版本。可以使用 `_has_include()` 检查这两个头文件中哪一个在系统中可用：

```
#if __has_include(<optional>)
    #include <optional>
#elif __has_include(<experimental/optional>)
    #include <experimental/optional>
#endif
```



## 11.3 核心语言特性的特性测试宏

C++20 新增了特性测试宏(feature test macros)，用于检测编译器支持哪些核心语言特性。所有这些宏都以 `_cpp_` 或 `_has_cpp_` 开头。示例如下：

- `_cpp_range_based_for`
- `_cpp_binary_literals`
- `_cpp_char8_t`
- `_cpp_generic_lambdas`
- `_cpp_consteval`

- \_\_cpp\_coroutines
- \_\_has\_cpp\_attribute(fallthrough)
- ...

这些宏的值是一个数字，表示添加或更新特定特性的月份和年份。日期格式为 YYYYMM。比如，`__has_cpp_attribute(nodiscard)`的值可以是 201603，即 2016 年 3 月，这是[[nodiscard]]属性首次引入的日期。或者可以是 201907，即 2019 年 7 月，这是根据属性指定的理由(比如[[nodiscard(" Reason")]])，更新日期。

在第 16 章“C++标准库概述”中，对于标准库特性，也有类似的特性测试宏。

#### 注意：

除非正在编写非常通用的跨平台和跨编译器库，否则几乎不需要这些特性测试宏。在第 34 章“开发跨平台和跨语言应用程序”中，将会讨论跨平台开发。

## 11.4 STATIC 关键字

在 C++ 中，`static` 关键字有多种用法，这些用法之间好像没有关系。“重载”关键字的部分原因是为了避免在语言中引入新的关键字。

### 11.4.1 静态数据成员和方法

可以声明类的静态数据成员和方法。与非静态数据成员不同，静态数据成员不是对象的一部分。相反，这个数据成员只有一个副本，这个副本存在于类的任何对象之外。

静态方法与此类似，位于类层次(而不是对象层次)。静态方法不会在某个特定对象的上下文中执行；因此，它没有隐式的 `this` 指针。但这也意味着静态方法不能标记为 `const`。

在第 9 章“掌握类和对象”中，提供了静态数据成员和静态方法的示例。

### 11.4.2 静态链接

在介绍用于链接的 `static` 关键字之前，首先需要理解 C++ 中的链接概念。C++ 的每个源文件都是单独编译的，编译得到的目标文件会彼此链接。C++ 源文件中的每个名称(包括函数和全局变量)都具有外部或内部的链接。外部链接意味着这个名称在其他源文件中也有效。内部链接(也称为静态链接)意味着在其他源文件中无效。默认情况下，函数和全局变量都具有外部链接。但是，可以在声明前面加上 `static` 关键字来指定内部(或静态)链接。比如，假设有两个源文件：FirstFile.cpp 和 AnotherFile.cpp。下面的代码是 FirstFile.cpp：

```
void f();  
  
int main()  
{  
    f();  
}
```

注意，这个文件提供了函数 `f()` 的原型，但没有显式给出定义。下面的代码是 AnotherFile.cpp：

```
import <iostream>;  
  
void f();
```

```
void f()
{
    std::cout << "f\n";
}
```

这个文件提供函数 f() 的原型和定义。注意，在两个不同的文件中为同一个函数编写原型是合法的。如果将原型放在头文件中，并且每个源文件中都用 #include 包含这个头文件，那么预处理器就会自动在每个源文件中给出函数原型。在下面的示例中，不使用头文件。过去使用头文件的原因是，它是更易于维护(并保持同步)原型的一个副本，但是从 C++20 开始，使用模块时建议不要使用头文件。

这些源文件中都可以成功编译，并且程序链接也没有问题：因为 f() 具有外部链接，main() 可以从其他文件调用它。

但是，假设对 AnotherFile.cpp 中的函数 f() 的原型应用了 static 关键字。注意，不需要在函数 f() 的定义前重复 static 关键字。只要它位于函数名称的第一个实例之前，就不需要重复它。

```
import <iostream>

static void f();

void f()
{
    std::cout << "f\n";
}
```

现在，每个源文件都可以正确编译，但是链接时却失败了，这是因为函数 f() 具有内部(静态)链接，从而使得 FirstFile.cpp 中无法使用这个函数。如果在源文件中定义了静态方法但是没有使用它，有些编译器会给出警告(指出这些方法不应该是静态的，因为其他文件可能会用到它们)。

将 static 用于内部链接的另一种方式是使用匿名的名称空间。将变量或函数封装到一个没有名字的名称空间中，而不是使用 static 关键字，示例如下：

```
import <iostream>

namespace
{
    void f();

    void f()
    {
        std::cout << "f\n";
    }
}
```

在同一源文件中，可在声明匿名名称空间之后的任何位置访问名称空间中的项，但不能在其他源文件中访问。这一语义与 static 关键字相同。

### 警告：

要获得内部链接，建议使用匿名的名称空间，而不要使用 static 关键字。

### extern 关键字

extern 关键字好像是 static 关键字的反义词，将它后面的名称指定为外部链接，在某些情况下可以使用这种方法。例如，默认情况下，const 和 typedef 具有内部链接，可以使用 extern 使其变为外部

链接。但是，`extern` 有一点复杂。当指定某个名称为 `extern` 时，编译器会将这条语句当作声明，而不是定义。对于变量，这意味着编译器不会为其分配空间。必须为这个变量提供单独的，而不是使用 `extern` 关键字的定义行。例如，下面是 `AnotherFile.cpp` 中的内容：

```
extern int x;
int x { 3 };
```

或者，可以在 `extern` 语句中初始化 `x`，然后将其作为声明和定义。

```
extern int x { 3 };
```

在这种情况下，`extern` 不是非常有用，因为默认情况下，`x` 具有外部链接。当另一个源文件 `FirstFile.cpp` 使用 `x` 时，才会真正用到 `extern`：

```
import <iostream>

extern int x;

int main()
{
    std::cout << x << std::endl;
}
```

本例中 `FirstFile.cpp` 使用了 `extern` 声明，因此可以使用 `x`。编译器需要知道 `x` 的声明之后，才能在 `main()` 中使用它。然而，如果在声明 `x` 时没有使用 `extern` 关键字，编译器会认为这是一个定义，并为 `x` 分配空间，从而导致链接失败(因为有两个全局作用域的 `x` 变量)。使用 `extern` 关键字，就可以在多个源文件中全局访问这个变量。

#### 警告：

不建议使用全局变量。全局变量令人困惑且容易出错，尤其是在大型程序中。请明智地使用它们！

### 11.4.3 函数中的静态变量

在 C++ 中，`static` 关键字的最终目的是创建离开和进入作用域时都可保留值的局部变量。函数中的静态变量就像只能在函数内部访问的全局变量。静态变量最常见的用法是“记住”某个函数是否执行了特定的初始化操作。比如，下面的代码就使用了这一技术：

```
void performTask()
{
    static bool initialized { false };
    if (!initialized) {
        cout << "initializing" << endl;
        // Perform initialization.
        initialized = true;
    }
    // Perform the desired task.
}
```

但是，静态变量容易让人迷惑，在构建代码时通常有更好的方法，以避免使用静态变量。在这种情况下，可以编写一个类，用构造函数执行所需的初始化操作。

**注意：**

避免使用单独的静态变量，为了维持状态可以改用对象。

但有时，它们十分有用。一个示例是实现 Meyer 的单例设计模式，详情请参阅第 33 章“应用设计模式”。

**注意：**

`performTask()`的实现不是线程安全的；它包含一个竞态条件。在多线程环境中，需要使用原子或其他机制来同步多个线程。关于多线程的详细讨论，请参阅第 27 章“使用 C++ 进行多线程编程”。

#### 11.4.4 非局部变量的初始化顺序

在结束静态数据成员和全局变量的主题之前，请考虑这些变量的初始化顺序。在 `main()` 开始之前，将初始化程序中的所有全局变量和静态类数据成员。对于给定源文件中的变量，将按照它们在源文件中出现的顺序进行初始化。例如，在下面的示例中，`Demo::x` 一定会在 `y` 之前初始化。

```
class Demo
{
public:
    static int x;
};

int Demo::x { 3 };
int y { 4 };
```

然而，C++ 没有提供规范，用以说明在不同源文件中初始化非局部变量的顺序。如果在某个源文件中有一个全局变量 `x`，在另一个源文件中有一个全局变量 `y`，无法知道哪个全局变量将首先初始化。通常，不需要关注这一规范的缺失，但如果某个全局变量或静态变量依赖于另一个变量，就可能引发问题。对象的初始化意味着调用构造函数，全局对象的构造函数可能会访问另一个全局对象，并假定另一个全局对象已经构建。如果这两个全局对象在不同的源文件中声明，就不能指望一个全局对象在另一个全局对象之前构建，也无法控制它们的初始化顺序。不同编译器可能有不同的初始化顺序，即使同一编译器的不同版本也可能如此，甚至在项目中添加另一个源文件也可能影响初始化顺序。

**警告：**

在不同源文件中，非局部变量的初始化顺序是不确定的。

#### 11.4.5 非局部变量的销毁顺序

非局部变量的销毁顺序与其初始化顺序相反。对于不同源文件中的非局部变量，初始化的顺序是不确定的，这意味着其销毁的顺序也是不确定的。

### 11.5 C 的实用工具

一些晦涩的 C 特性在 C++ 中也可以用。本节研究其中的两个特性：变长参数列表(variable-length argument lists)和预处理器宏(preprocessor macros)。

### 11.5.1 变长参数列表

本节将介绍旧的 C 风格的变长参数列表。需要知道它们是如何工作的，因为可能会在遗留代码中看到它们。然而，在新的代码中，应该通过变参模板使用类型安全的变长参数列表，变参模板将在第 26 章“高级模板”中进行介绍。对于<cstdio>中的 C 函数 printf()，可以使用任意数量的参数来调用它。

```
printf("int %d\n", 5);
printf("String %s and int %d\n", "hello", 5);
printf("Many ints: %d, %d, %d, %d, %d\n", 1, 2, 3, 4, 5);
```

C/C++ 提供了语法和一些实用宏，用于编写具有可变数量的函数。这些函数通常看起来很像 printf()。尽管并非经常需要，但偶尔需要这个特性。例如，假设想编写一个快速调试的函数，如果设置了调试标记，这个函数向 stderr 输出字符串，但如果设置没有调试标记，就什么都不做。与 printf() 一样，这个函数应该能接收任意数目和任意类型的参数并输出字符串。这个函数的简单实现如下所示：

```
#include <cstdio>
#include <cstdarg>

bool debug { false };

void debugOut(const char* str, ...)
{
    va_list ap;
    if (debug) {
        va_start(ap, str);
        vfprintf(stderr, str, ap);
        va_end(ap);
    }
}
```

首先，注意 debugOut() 函数的原型包含一个具有类型和名称的参数 str，之后是省略号(...)，这代表任意数目和类型的参数。如果要访问这些参数，必须使用<cstdarg>中定义的宏。声明一个 va\_list 类型的变量，并调用 va\_start() 对其进行初始化。va\_start() 的第二个参数必须是参数列表中最右边的已命名变量。所有具有变长参数列表的函数都至少需要一个已命名的参数。debugOut() 函数只是将列表传递给 vfprintf()(<cstdio>中的标准函数)。vfprintf() 的调用返回时，debugOut() 调用 va\_end() 来终止对变长参数列表的访问。在调用 va\_start() 之后，必须总是调用 va\_end()，以确保函数结束后，堆栈处于稳定的状态。

函数 debugOut() 的用法如下：

```
debug = true;
debugOut("int %d\n", 5);
debugOut("String %s and int %d\n", "hello", 5);
debugOut("Many ints: %d, %d, %d, %d, %d\n", 1, 2, 3, 4, 5);
```

#### 1. 访问参数

如果想要访问实际参数，那么可以使用 va\_arg(); 它接收 va\_list 作为第一个参数，以及需要解析的参数的类型作为第二个参数。但是，如果不提供显示的方法，就无法知道参数列表的结尾是什么。例如，可以将第一个参数作为参数个数的计数。或者，当参数是一组指针时，可以要求最后一个指针是 nullptr。方法很多，但对于程序员来说，所有方法都很麻烦。

下面的示例演示了这种技术，其中调用者在第一个已命名参数中指定了所提供参数的数目。函数接收任意数目的 int 参数，并将其输出。

```
void printInts(size_t num, ...)
{
    va_list ap;
    va_start(ap, num);
    for (size_t i { 0 }; i < num; ++i) {
        int temp { va_arg(ap, int) };
        cout << temp << " ";
    }
    va_end(ap);
    cout << endl;
}
```

可以按以下方式调用 printInts()。注意，第一个参数将指定后面整数的数目。

```
printInts(5, 5, 4, 3, 2, 1);
```

## 2. 为什么不应该使用 C 风格的变长参数列表

访问 C 风格的变长参数列表并不十分安全。从 printInts() 函数可以看出，这种方法存在以下风险：

- 不知道参数的数目。在 printInts() 中，必须信任调用者传递了与第一个参数指定的数目相等的参数。在 debugOut() 中，必须相信调用者在字符数组之后传递的参数数目与字符数组中的格式代码一致。
- 不知道参数的类型。va\_arg() 接收一种类型，用来解释当前的值。然而，可让 va\_arg() 将这个值解释为任意类型，无法验证正确的类型。

### 警告：

避免使用 C 风格的变长参数列表。使用第 1 章介绍的初始化列表时，最好传入 std::array 或者值矢量；对于类型安全的变长参数列表，可以使用变参模板，这一主题将在第 22 章讲述。

## 11.5.2 预处理器宏

可使用 C++ 预处理器编写宏，这与函数有点相似。下面是一个示例：

```
#define SQUARE(x) ((x) * (x)) // No semicolon after the macro definition!

int main()
{
    cout << SQUARE(5) << endl;
    return 0;
}
```

宏是 C 遗留下来的特性，非常类似于内联函数，但不执行类型检测。在调用宏时，预处理器会自动使用扩展式替换。预处理器并不会真正地应用函数调用语义，这一行为可能导致无法预测的结果。例如，如果用 2+3 而不是 5 调用 SQUARE 宏，考虑一下会发生什么，如下所示。

```
cout << SQUARE(2 + 3) << endl;
```

SQUARE 宏的计算结果应为 25，结果确实如此。然而，如果在宏定义中省略部分圆括号，这个宏看上去是这样的：

```
#define SQUARE(x) (x * x)
```

现在，调用 `SQUARE(2 + 3)` 的结果是 11，而不是 25！注意宏只是自动扩展，而不考虑函数调用语义。这意味着在宏中，`x` 被替换为 `2+3`，扩展式是：

```
cout << (2 + 3 * 2 + 3) << endl;
```

按照正确的操作顺序，这行代码首先执行乘法，然后是加法，结果是 11 而不是 25。

宏还会影响性能，假定按如下方式调用 `SQUARE` 宏：

```
cout << SQUARE(veryExpensiveFunctionCallToComputeNumber()) << endl;
```

预处理器把它替换为：

```
cout << ((veryExpensiveFunctionCallToComputeNumber()) *  
         (veryExpensiveFunctionCallToComputeNumber())) << endl;
```

现在，这个开销很大的函数调用了两次。这是避免使用宏的另一个原因。

宏还会导致调试问题，因为编写的代码并非编译器看到的代码或者调试工具中显示的代码（因为预处理器的查找和替换功能）。为此，应该全部用内联函数替代宏。在这里详细讲述宏，只是因为相当多的 C++ 代码使用了宏，为了阅读并维护这些代码，应该理解宏。

#### 注意：

大多数编译器可将经过预处理器处理的源代码输出到某个文件。使用这个文件可以观察预处理器是如何处理文件的。例如，使用 Microsoft VC++ 时需要使用 /P 选项，使用 GCC 时，可以使用 -E 选项。

## 11.6 本章小结

本章从编写和使用 C++20 模块的细节开始，然后是使用旧式头文件的一些棘手的方面。还介绍了 `static` 和 `extern` 关键字的不同用法。

本章最后讨论了如何编写 C 风格的变长参数列表以及如何编写预处理器宏。理解这两个主题都很重要，因为可能会在遗留代码中看到它们。但是，在任何新编写的代码中都应避免使用它们。

下一章将开始讨论如何编写通用代码的模板。

## 11.7 练习

通过完成以下习题，可以巩固本章讨论的知识点。所有练习的答案都可扫描封底二维码下载。然而，如果你困在一个练习中，在网站上查看答案之前，请先重新阅读本章的内容，试着自己找到答案。

**练习 11-1** 在 `Simulator` 名称空间中编写一个名为 `Simulator` 的单文件模块，其中包含两个类 `CarSimulator` 和 `BikeSimulator`。课程的内容对于这些练习并不重要。只需要提供一个将消息输出到标准输出的默认构造函数。在 `main()` 函数中测试编写的代码。

**练习 11-2** 练习 11-1 中的解决方案将模块分成几个文件：一个没有任何实现的主模块接口文件和两个模块实现文件，一个用于 `CarSimulator` 类，一个用于 `BikeSimulator` 类。

**练习 11-3** 将练习 11-2 中的解决方案转换为使用一个主模块接口文件和两个模块接口分区文件，一个用于包含 `CarSimulator` 类的模拟器：car 分区；另一个用于包含 `BikeSimulator` 类的模拟器：bike 分区。

**练习 11-4** 对于练习 11-3 中的解决方案，添加一个名为 `internals` 的实现分区，该分区在 `Simulator` 名称空间中包含一个名为 `convertMilesToKm(double miles)` 的辅助方法。一英里等于 1.6 公里。在 `CarSimulator` 类和 `BikeSimulator` 类中都添加一个名为 `setOdometer(double miles)` 的方法，这个方法使用辅助方法将给定的英里转换为公里，然后将其输出到标准输出中。在 `main()` 函数中确认 `setOdometer()` 对这两个类都有效。还要确认 `main()` 函数不能调用 `convertMilesToKm()`。



# 第 12 章

## 利用模板编写泛型代码

### 本章内容

- 如何编写类模板
- 编译器处理模板的方式
- 组织模板源代码
- 使用非类型模板参数
- 编写单独类方法模板
- 为特定类型自定义类模板
- 模板和继承的结合
- 如何编写别名模板
- 如何编写函数模板
- 将函数模板设置为类模板的友元
- 使用缩写函数模板语法
- 使用变量模板
- 使用概念指定模板参数的要求

C++不仅支持面向对象编程，还支持泛型编程(generic programming)。根据第6章“设计可重用代码”的讨论，泛型编程的目的是编写可重用的代码。在C++中，泛型编程的基本工具是模板。尽管从严格意义上说，模板并不是一个面向对象的特性，但模板可与面向对象编程结合使用，从而产生强大的作用。很多程序员认为模板是C++中难度最高的一部分，因此尽量避免使用模板。但是，作为一名专业的C++程序员，应该对模板有所了解。

本章列出了满足第6章讨论的一般性设计原则所需的编码细节，第22章“高级模板”将讨论一些高级模板特性。

### 12.1 模板概述

面向过程编程范型中主要的编程单元是过程或函数。主要使用的是函数，因为函数可用于编写不依赖特定值的算法，从而可重用很多不同的值。例如，C++中的sqrt()函数计算调用者指定的值的平

方根。只能计算一个数字(例如 4)的平方根的函数没有什么实际用途。`sqrt()`函数是基于参数编写的，参数实际上是一个占位符，可表示调用者传入的任何数值。用计算机科学家的话说，就是用函数参数化值。

面向对象编程范式中加入了对象的概念，对象将相关的数据和行为组织起来，但它并没有改变函数和方法参数化值的方式。

模板将参数化的概念推进了一步，不仅允许参数化值，还允许参数化类型。C++中的类型不仅包括基本类型，例如 `int` 和 `double`，还包括用户定义的类，例如 `SpreadsheetCell` 和 `CherryTree`。使用模板，不仅可编写不依赖特定值的代码，还能编写不依赖那些值类型的代码。例如，不需要为保存 `int`、`Car` 和 `SpreadsheetCell` 而编写不同的堆栈类，但可以编写一个堆栈的类定义，这个类定义可用于任何类型。

尽管模板是一个令人惊叹的语言特性，但由于 C++ 模板的语法令人费解，很多程序员会选择忽略或避免使用模板。不过，每个程序员至少需要知道如何使用模板，因为程序库(例如 C++ 标准库)中广泛使用了模板。

这一章讲解 C++ 中的模板支持，重点讲述模板在标准库中的使用。在学习过程中，可以学会一些有用的细微特性，除了标准库之外，还可以在程序中使用这些特性。

## 12.2 类模板

类模板定义了一个类，其中，将一些变量的类型、方法的返回类型和/或方法的参数类型指定为参数。类模板主要用于容器，或用于保存对象的数据结构。本节使用一个 `Grid` 容器作为示例。为了让这些例子长度合理，且足够简单地演示特定的知识点，本章不同的小节会向 `Grid` 容器添加一些特性，这些特性在接下来的几个章节中将不会被用到。

### 12.2.1 编写类模板

假设想要一个通用的棋盘类，可将其用作象棋棋盘、跳棋棋盘、井字游戏棋盘或其他任何二维的棋盘。为让这个棋盘通用，这个棋盘应该能保存象棋棋子、跳棋棋子、井字游戏棋子或其他任何游戏类型的棋子。

#### 1. 编写不使用模板的代码

如果不使用模板，编写通用棋盘最好的方法是采用多态技术，保存通用的 `GamePiece` 对象。然后，可让每种游戏的棋子继承 `GamePiece` 类。例如，在象棋游戏中，`ChessPiece` 可以是 `GamePiece` 的派生类。通过多态技术，既能保存 `GamePiece` 的 `GameBoard`，也能保存 `ChessPiece`。`GameBoard` 可以复制，所以 `GameBoard` 需要能复制 `GamePiece`。这个实现利用了多态技术，因此一种解决方案是给 `GamePiece` 基类添加纯虚方法 `clone()`，它的派生类必须实现 `clone()`，并返回一个具体的 `GamePiece` 的副本。`GamePiece` 基类如下：

```
export class GamePiece
{
public:
    virtual ~GamePiece() = default;
    virtual std::unique_ptr<GamePiece> clone() const = 0;
};
```

GamePiece 是一个抽象基类。ChessPiece 等具体类派生于它，并实现了 clone()方法。

```
class ChessPiece : public GamePiece
{
public:
    std::unique_ptr<GamePiece> clone() const override
    {
        // Call the copy constructor to copy this instance
        return std::make_unique<ChessPiece>(*this);
    }
};
```

GameBoard 的实现使用 unique\_ptr 矢量组中的矢量来存储 GamePiece。

```
export class GameBoard
{
public:
    explicit GameBoard(size_t width = DefaultWidth,
                        size_t height = DefaultHeight);
    GameBoard(const GameBoard& src); // copy constructor
    virtual ~GameBoard() = default; // virtual defaulted destructor
    GameBoard& operator=(const GameBoard& rhs); // assignment operator

    // Explicitly default a move constructor and assignment operator.
    GameBoard(GameBoard&& src) = default;
    GameBoard& operator=(GameBoard&& src) = default;

    std::unique_ptr<GamePiece>& at(size_t x, size_t y);
    const std::unique_ptr<GamePiece>& at(size_t x, size_t y) const;

    size_t getHeight() const { return m_height; }
    size_t getWidth() const { return m_width; }

    static const size_t DefaultWidth { 10 };
    static const size_t DefaultHeight { 10 };

    void swap(GameBoard& other) noexcept;

private:
    void verifyCoordinate(size_t x, size_t y) const;

    std::vector<std::vector<std::unique_ptr<GamePiece>>> m_cells;
    size_t m_width { 0 }, m_height { 0 };
};

export void swap(GameBoard& first, GameBoard& second) noexcept;
```

在这个实现中，at()返回指定位置的棋子的引用，而不是返回棋子的副本。GameBoard 作为一个二维数组的抽象，它应给出实际对象的索引，而不是给出对象的副本，以提供数组访问语义。客户代码不应存储这个引用供将来使用，因为它可能是无效的；相反，应该在使用返回的引用之前调用 at()。这遵循了标准库中 std::vector 类的设计原理。

### 注意：

GameBoard 类的这个实现提供了 at() 的两个版本，一个版本返回引用，另一个版本返回 const 引用。

下面是方法的定义。注意，这个实现为赋值运算符使用了“复制和交换”惯用语法，还使用 Scott Meyer 的 `const_cast()` 模式来避免代码重复，第 9 章“精通类与对象”将讨论这些主题。

```
GameBoard::GameBoard(size_t width, size_t height)
    : m_width { width }, m_height { height }
{
    m_cells.resize(m_width);
    for (auto& column : m_cells) {
        column.resize(m_height);
    }
}

GameBoard::GameBoard(const GameBoard& src)
    : GameBoard { src.m_width, src.m_height }
{
    // The ctor-initializer of this constructor delegates first to the
    // non-copy constructor to allocate the proper amount of memory.

    // The next step is to copy the data.
    for (size_t i { 0 }; i < m_width; i++) {
        for (size_t j { 0 }; j < m_height; j++) {
            if (src.m_cells[i][j])
                m_cells[i][j] = src.m_cells[i][j]->clone();
        }
    }
}

void GameBoard::verifyCoordinate(size_t x, size_t y) const
{
    if (x >= m_width) {
        throw out_of_range { format("{} must be less than {}.", x, m_width) };
    }
    if (y >= m_height) {
        throw out_of_range { format("{} must be less than {}.", y, m_height) };
    }
}

void GameBoard::swap(GameBoard& other) noexcept
{
    std::swap(m_width, other.m_width);
    std::swap(m_height, other.m_height);
    std::swap(m_cellst, other.m_cellst);
}

void GameBoard::swap(GameBoard& first, GameBoard& second) noexcept
{
    first.swap(second);
}

GameBoard& GameBoard::operator=(const GameBoard& rhs)
{
    // Copy-and-swap idiom
    GameBoard temp { rhs }; // Do all the work in a temporary instance
    swap(temp);           // Commit the work with only non-throwing operations
    return *this;
}
```

```

}

const unique_ptr<GamePiece>& GameBoard::at(size_t x, size_t y) const
{
    verifyCoordinate(x, y);
    return m_cells[x][y];
}

unique_ptr<GamePiece>& GameBoard::at(size_t x, size_t y)
{
    return const_cast<unique_ptr<GamePiece>&>(as_const(*this).at(x, y));
}

```

这个 GameBoard 类可以很好地完成任务。

```

GameBoard chessboard { 8, 8 };
auto pawn = std::make_unique<ChessPiece>();
chessBoard.at(0, 0) = std::move(pawn);
chessBoard.at(0, 1) = std::make_unique<ChessPiece>();
chessBoard.at(0, 1) = nullptr;

```

## 2. 模板 Grid 类

前面定义的 GameBoard 类很好，但不够完善。第一个问题是无法使用 GameBoard 按值存储元素，它总是存储指针。另一个更重要的问题与类型安全相关。GameBoard 中的每个网格都存储 unique\_ptr<GamePiece>。即使存储 ChessPiece，当使用 at() 请求某个网格时，也会得到 unique\_ptr<GamePiece>。这意味着，只有将检索到的 GamePiece 向下转型为 ChessPiece，才能使用 ChessPiece 的特定功能。GameBoard 的另一个缺点是不能用来存储基本类型，如 int 或 double，因为存储在网格中的类型必须派生自类 GamePiece。

因此，最好编写一个通用的 Grid 类，该类可用于存储 ChessPiece、SpreadsheetCell、int 和 double 等。在 C++ 中，可通过编写类模板来实现这一点，编写类模板可避免编写需要指定一种或多种类型的类。客户通过指定要使用的类型对模板进行实例化。这称为泛型编程，其最大的优点是类型安全。类及其方法中使用的类型是具体的类型，而不是多态方案中的抽象基类类型。例如，假设不仅有派生类 ChessPiece，还有派生类 TicTacToePiece。

```

class TicTacToePiece : public GamePiece
{
public:
    std::unique_ptr<GamePiece> clone() const override
    {
        // Call the copy constructor to copy this instance
        return std::make_unique<TicTacToePiece>(*this);
    }
};

```

使用前面介绍的多态解决方案，显然可以在同一棋盘中存储象棋棋子和井字游戏棋子。

```

GameBoard chessboard { 8, 8 };
chessBoard.at(0, 0) = std::make_unique<ChessPiece>();
chessBoard.at(0, 1) = std::make_unique<TicTacToePiece>();

```

最大的问题在于，在一定程度上，只有记住网格中存储的内容，才能在调用 at() 时正确地向下转型。

## Grid 类定义

为理解类模板，最好首先看一下类模板的语法。下例展示了如何修改 GameBoard 类，得到模板化的 Grid 类。代码之后会对所有语法进行解释。注意，类名从 GameBoard 变为 Grid。这个 Grid 类还应该可以用于基本类型，如 int 和 double。因此，这里选用不带多态性的值语义来实现这个解决方案，而 GameBoard 实现中使用了多态指针语义。m\_cells 容器中存储的是真实的对象，而不是指针。与指针语义相比，使用值语义的缺点在于不能拥有真正的空网格，也就是说，网格始终要包含一些值。而使用指针语义，在空网格中可以存储 nullptr。幸运的是，第 1 章介绍的 std::optional(在<optional>中定义)可弥补这一点。它允许使用值语义，并且同时允许表示空网格。

```
export template <typename T>
class Grid
{
public:
    explicit Grid(size_t width = DefaultWidth,
                  size_t height = DefaultHeight);
    virtual ~Grid() = default;

    // Explicitly default a copy constructor and assignment operator.
    Grid(const Grid& src) = default;
    Grid& operator=(const Grid& rhs) = default;

    // Explicitly default a move constructor and assignment operator.
    Grid(Grid&& src) = default;
    Grid& operator=(Grid&& rhs) = default;

    std::optional<T>& at(size_t x, size_t y);
    const std::optional<T>& at(size_t x, size_t y) const;

    size_t getHeight() const { return m_height; }
    size_t getWidth() const { return m_width; }

    static const size_t DefaultWidth { 10 };
    static const size_t DefaultHeight { 10 };

private:
    void verifyCoordinate(size_t x, size_t y) const;

    std::vector<std::vector<std::optional<T>>> m_cells;
    size_t m_width { 0 }, m_height { 0 };
};
```

现在已展示了完整的类定义，下面逐行分析这些代码：

```
export template <typename T>
```

第一行表示，下面的类定义是基于一种类型(T)的模板，该类型从模块中导出。template 和 typename 都是 C++ 中的关键字。如前所述，模板“参数化”类型的方式与函数“参数化”值的方式相同。就像在函数中通过参数名表示调用者要传入的参数一样，在模板中使用模板类型参数名称(例如 T)表示调用者传入的作为模板类型实参的类型。名称 T 没有什么特别之处，可以使用任何名称。按照惯例，只使用一种类型时，将这种类型称为 T，但这只是一项历史约定，就像把索引数组的整数命名为 i 或 j 一样。这个模板说明符应用于整个语句，在这里是整个类定义。

**注意：**

基于历史原因，指定模板类型参数时，可用关键字 `class` 替代 `typename`。因此，很多书籍和现有的程序使用了这样的语法：`template <class T>`。不过，在这个上下文中使用 `class` 这个词会产生一些误解，因为这个词暗示这种类型必须是一个类，而实际上并不要求这样。这种类型可以是类、`struct`、`union`、语言的基本类型（例如 `int` 或 `double` 等）。

在前面的 `GameBoard` 类中，`m_cells` 数据成员是指针的矢量组中的矢量，这需要特定的复制代码，因此需要复制构造函数和赋值运算符。在 `Grid` 类中，`m_cells` 是可选值的矢量组中的矢量，所以编译器生成的复制构造函数和赋值运算符可以运行得很好。但如第 8 章“属性类和对象”所述，一旦有了用户声明的析构函数，建议不要使用编译器隐式生成复制构造函数或赋值运算符，因此 `Grid` 类模板将其显式设置为 `default`，并且将移动构造函数和赋值运算符显式设置为 `default`。下面将复制赋值运算符显式设置为 `default`：

```
Grid<T>& operator=(const Grid& rhs) = default;
```

从中可以看出，`rhs` 参数的类型不再是 `const GameBoard&` 了，而是 `const Grid&`。在类定义中，编译器根据需要将 `Grid` 解释为 `Grid<T>`，但可以根据需要，显式地使用 `Grid<T>`。

```
Grid<T>& operator=(const Grid<T>& rhs) = default;
```

但在类定义之外，必须使用 `Grid<T>`。在编写类模板时，以前的类名(`Grid`)现在实际上是模板名称。讨论实际的 `Grid` 类或类型时，需要使用模板 ID(如：`Grid<T>`)，它是 `Grid` 类模板对某种类型实例化的结果，例如 `int`、`SpreadsheetCell` 或 `ChessPiece`。

`m_cells` 不再存储指针，而是存储可选值。`at()`方法现在返回 `optional<T>&`，而不是返回 `unique_ptr`。也就是说，`optional` 中可以包含 `T` 类型的值或者为空。

```
std::optional<T>& at(size_t x, size_t y);
const std::optional<T>& at(size_t x, size_t y) const;
```

**Grid 类的方法定义**

`template <typename T>` 访问说明符必须在 `Grid` 模板的每一个方法定义的前面。构造函数如下所示：

```
template <typename T>
Grid<T>::Grid(size_t width, size_t height)
: m_width{width}, m_height{height}
{
    m_cells.resize(m_width);
    for (auto& column : m_cells) {
        column.resize(m_height);
    }
}
```

**注意：**

类模板中方法的定义需要对所有使用该模板的客户代码可见。这对放置方法定义的位置有一些限制。通常，它们只是与类模板定义本身放在相同的文件中。本章稍后将讨论绕过这个限制的一些方法。

注意：之前的类名是`::Grid<T>`，而不是 `Grid`。必须在所有的方法和静态数据成员定义中将 `Grid<T>` 指定为类名。构造函数的函数体类似于 `GameBoard` 构造函数。

其他方法定义也类似于 GameBoard 类中对应的方法定义，只是适当地改变了模板和 Grid<T>的语法。

```
template <typename T>
void Grid<T>::verifyCoordinate(size_t x, size_t y) const
{
    if (x >= m_width) {
        throw std::out_of_range {
            std::format("{} must be less than {}.", x, m_width) };
    }
    if (y >= m_height) {
        throw std::out_of_range {
            std::format("{} must be less than {}.", y, m_height) };
    }
}

template <typename T>
const std::optional<T>& Grid<T>::at(size_t x, size_t y) const
{
    verifyCoordinate(x, y);
    return m_cells[x][y];
}

template <typename T>
std::optional<T>& Grid<T>::at(size_t x, size_t y)
{
    return const_cast<std::optional<T>&>(std::as_const(*this).at(x, y));
}
```

### 注意：

如果类模板方法的实现需要特定模板类型参数(例如 T)的默认值，可使用 T{} 语法。如果 T 是类类型，T{} 调用对象的默认构造函数，或者如果 T 是基本类型，则生成 0。这称为“初始化为 0”语法。最好为类型尚不确定的变量提供合理的默认值。

### 3. 使用 Grid 模板

创建网格对象时，不能单独使用 Grid 作为类型；必须指定这个网格保存的元素类型。为某种类型创建一个模板类对象的过程称为模板的实例化。下面举一个示例：

```
Grid<int> myIntGrid; // declares a grid that stores ints,
                     // using default arguments for the constructor
Grid<double> myDoubleGrid { 11, 11 }; // declares an 11x11 Grid of doubles

myIntGrid.at(0, 0) = 10;
int x { myIntGrid.at(0, 0).value_or(0) };

Grid<int> grid2 { myIntGrid }; // Copy constructor
Grid<int> anotherIntGrid;
anotherIntGrid = grid2; // Assignment operator
```

注意 myIntGrid、grid2 和 anotherIntGrid 的类型为 Grid<int>。不能将 SpreadsheetCell 或 ChessPiece 保存在这些网格中，否则编译器会生成错误消息。

另外，这里使用了 value\_or()。at()方法返回 std::optional 引用。Optional 可以包含值，也可以不包

含值。如果 optional 包含值, value\_or()方法返回这个值; 否则返回给 value\_or()提供的实参。

类型规范非常重要, 下面两行代码都无法编译:

```
Grid test;      // WILL NOT COMPILE
Grid<> test;  // WILL NOT COMPILE
```

编译器对第一行代码会给出如下错误: “使用类模板要求提供模板参数列表。”编译器对第二行代码会给出如下错误: “模板参数太少。”

如果要声明一个接收 Grid 对象的函数或方法, 必须在 Grid 类型中指定保存在网格中的元素类型。

```
void processIntGrid(Grid<int>& grid) { /* Body omitted for brevity */ }
```

另外, 可使用将在本章后面介绍的函数模板, 基于网格中的元素类型编写函数模板。

#### 注意:

为避免每次都编写完整的 Grid 类型名称, 例如 Grid<int>, 可通过类型别名指定一个更简单的名称。

```
using IntGrid = Grid<int>;
```

现在可编写以下代码:

```
void processIntGrid(IntGrid& grid) { }
```

Grid 类模板能保存的数据类型不只是 int。例如, 可实例化一个保存 SpreadsheetCell 的网格:

```
Grid<SpreadsheetCell> mySpreadsheet;
SpreadsheetCell myCell { 1.234 };
mySpreadsheet.at(3, 4) = myCell;
```

还可保存指针类型:

```
Grid<const char*> myStringGrid;
myStringGrid.at(2, 2) = "hello";
```

指定的类型甚至可以是另一个模板类型:

```
Grid<vector<int>> gridOfVectors;
vector<int> myVector { 1, 2, 3, 4 };
gridOfVectors.at(5, 6) = myVector;
```

还可在自由存储区(堆)上动态分配 Grid 模板实例:

```
auto myGridOnHeap { make_unique<Grid<int>>(2, 2) }; // 2x2 Grid on the heap
myGridOnHeap->at(0, 0) = 10;
int x { myGridOnHeap->at(0, 0).value_or(0) };
```

## 12.2.2 编译器处理模板的原理

为理解模板的复杂性, 必须学习编译器处理模板代码的原理。编译器遇到模板方法定义时, 会进行语法检查, 但是并不编译模板。编译器无法编译模板定义, 因为它不知道要使用什么类型。如果不知道 x 和 y 的类型, 那么编译器就无法为 x = y 这样的语句生成代码。

编译器遇到一个实例化的模板时, 例如 Grid<int>, 就会将模板类定义中的每一个 T 替换为 int, 从而生成 Grid 模板的 int 版本代码。当编译器遇到这个模板的另一个实例时, 例如 Grid<SpreadsheetCell>, 就会为 SpreadsheetCell 生成另一个版本的 Grid 类。编译器生成代码的方式就

好像语言不支持模板时程序员编写代码的方式：为每种元素类型编写一个不同的类。这里没有什么神奇之处，模板只是自动完成了一个令人厌烦的过程。如果在程序中没有将类模板实例化为任何类型，就不会编译类方法的定义。

这个实例化的过程也解释了为什么需要在定义中的多个地方使用 `Grid<T>` 语法。当编译器为某种特定类型实例化模板时，例如 `int`，就将 `T` 替换为 `int`，变成 `Grid<int>` 类型。

### 1. 选择性实例化

类模板的隐式实例化如下所示：

```
Grid<int> myIntGrid;
```

编译器总是为类模板的所有虚方法生成代码。但对于非虚方法，编译器只会为那些实际为某种类型调用的非虚方法生成代码。例如，给定前面定义的 `Grid` 类模板，假设在 `main()` 中编写这段代码(而且只有这段代码)：

```
Grid<int> myIntGrid;
myIntGrid.at(0, 0) = 10;
```

编译器只会为 `int` 版本的 `Grid` 类生成无参构造函数、析构函数和非常量 `at()` 方法的代码。它不会为其他方法生成代码，例如复制构造函数、赋值运算符或 `getHeight()`。编译器的以上行为称为选择性实例化(selective instantiation)。

存在这样的危险：在一些类模板方法中存在编译错误，而这些错误会被忽略。类模板中未使用的方法可能包含语法错误，因为这些错误不会被编译。这使得很难检测所有代码中的语法错误。通过显式的模板实例化，可以强制编译器为所有的方法生成代码，包括虚方法和非虚方法。示例如下：

```
template class Grid<int>;
```

#### 注意：

显式的模板实例化有助于发现错误，因为它会强制编译所有的类模板方法，甚至在未使用的时候也是如此。

当使用显式的模板实例化时，不要只尝试使用基本类型(如 `int`)来实例化类模板，而应该尝试使用更复杂的类型(如 `string`)来实例化类模板。

### 2. 模板对类型的要求

在编写与类型无关的代码时，必须对这些类型有一些假设。例如，在 `Grid` 类模板中，假设元素类型(用 `T` 表示)是可析构的、可复制/移动构造的、可复制/移动赋值的。

如果在程序中试图用一种不支持模板使用的所有操作的类型对模板进行实例化，那么这段代码无法编译，而且错误消息几乎总是晦涩难懂。然而，就算要使用的类型不支持所有模板代码所需的操作，也仍然可以利用选择性实例化使用某些方法，而避免使用另一些方法。

`C++20` 引入了概念(concepts)，允许编写编译器可以解释和验证的模板参数需求。如果传递给实例化模板的模板实参不满足这些要求，编译器会生成更多可读的错误。相关概念将在本章后面讨论。

#### 12.2.3 将模板代码分布到多个文件中

对于类模板，编译器必须可以从使用它们的任何源文件中获取类模板定义和方法定义。有几种机制可以实现这一点。

## 1. 将方法的定义与模板的定义放在相同文件中

可以将方法的定义直接放在定义类模板的相同模块接口文件中。当使用模板的另一个源文件中导入此模块时，编译器将能够访问所需的所有代码。这个机制可以用于之前的 Grid 实现。

## 2. 将方法的定义放在单独的文件中

或者，可以将类模板方法的定义放在单独的模块接口分区文件中。然后，还需要将类模板的定义放在它自己的分区中。例如，Grid 类模板的基本模块接口文件看起来像下面这样：

```
export import grid;

export import :definition;
export import :implementation;
```

这里导入和导出两个模块分区：definition 和 implementation。类模板定义在 definition 分区中：

```
export module grid:definition;

import <vector>;
import <optional>;

export template <tynname T> class Grid { ... };
```

如果方法的实现在 implementation 分区中，那么 implementation 分区也需要导入 definition 分区，因为它需要 Grid 类模板的定义：

```
export module grid:implementation;

import :definition;
import <vector>;
...

export template <tynname T>
Grid<T>::Grid(size_t width, size_t height) : m_width { width }, m_height { height }
{ ... };
```

### 12.2.4 模板参数

在 Grid 示例中，Grid 模板包含一个模板参数：保存在网格中的元素的类型。在编写这个类模板时，需要在尖括号中指定参数列表，示例如下：

```
template <tynname T>
```

这个参数列表类似于函数或方法中的参数列表。与函数或方法一样，可使用任意多个模板参数来编写类。此外，这些参数可以不是类型，并且可以具有默认值。

#### 1. 非类型的模板参数

非类型的模板参数是“普通”参数，例如 int 和指针：函数和方法中你十分熟悉的那种参数。然而，非类型的模板参数只能是整数类型(char、int、long 等)、枚举类型、指针、引用、std::nullptr\_t、auto、auto& 和 auto\*。此外，从 C++20 开始，可允许浮点型(甚至类类型)的非类型的模板参数。然而，后者有很多限制，本文不再进一步讨论。

在 Grid 类模板中，可通过非类型模板参数指定网格的高度和宽度，而不是在构造函数中指定它们。在模板列表中指定非类型参数而不是在构造函数中指定的主要好处是：在编译代码之前就已经知

道这些参数的值了。前面提到，编译器为模板化的方法生成代码的方式是通过在编译之前替换模板参数。因此，在这个实现中，可使用普通的二维数组而不是动态分配大小的矢量组中的矢量。下面是突出显示的新的类定义：

```
export template <typename T, size_t WIDTH, size_t HEIGHT>
class Grid
{
public:
    Grid() = default;
    virtual ~Grid() = default;

    // Explicitly default a copy constructor and assignment operator.
    Grid(const Grid& src) = default;
    Grid& operator=(const Grid& rhs) = default;

    std::optional<T>& at(size_t x, size_t y);
    const std::optional<T>& at(size_t x, size_t y) const;

    size_t getHeight() const { return HEIGHT; }
    size_t getWidth() const { return WIDTH; }

private:
    void verifyCoordinate(size_t x, size_t y) const;

    std::optional<T> m_cells[WIDTH][HEIGHT];
};
```

这个类没有显式地将移动构造函数和移动赋值运算符设置为默认，原因是 C 风格的数组不支持移动语义。可以将实现对移动语义的支持作为练习。

注意，模板参数列表需要 3 个参数：网格中保存的对象类型以及网格的宽度和高度。宽度和高度用于创建保存对象的二维数组。下面是类方法定义：

```
template <typename T, size_t WIDTH, size_t HEIGHT>
void Grid<T, WIDTH, HEIGHT>::verifyCoordinate(size_t x, size_t y) const
{
    if (x >= WIDTH) {
        throw std::out_of_range {
            std::format("{} must be less than {}.", x, WIDTH) };
    }
    if (y >= HEIGHT) {
        throw std::out_of_range {
            std::format("{} must be less than {}.", y, HEIGHT) };
    }
}

template <typename T, size_t WIDTH, size_t HEIGHT>
const std::optional<T>& Grid<T, WIDTH, HEIGHT>::at(size_t x, size_t y) const
{
    verifyCoordinate(x, y);
    return m_cells[x][y];
}

template <typename T, size_t WIDTH, size_t HEIGHT>
std::optional<T>& Grid<T, WIDTH, HEIGHT>::at(size_t x, size_t y)
{
    return const_cast<std::optional<T>&>(std::as_const(*this).at(x, y));
}
```

注意之前所有指定 `Grid<T>` 的地方，现在都必须指定 `Grid<T, WIDTH, HEIGHT>` 来表示这 3 个模板参数。

可通过以下方式实例化这个模板：

```
Grid<int, 10, 10> myGrid;
Grid<int, 10, 10> anotherGrid;
myGrid.at(2, 3) = 42;
anotherGrid = myGrid;
cout << anotherGrid.at(2, 3).value_or(0);
```

这段代码看上去很棒。但遗憾的是，实际中的限制比想象中的要多。首先，不能通过非常量的整数指定高度或宽度。下面的代码无法编译：

```
size_t height { 10 };
Grid<int, 10, height> testGrid; // DOES NOT COMPILE
```

然而，如果把 `height` 声明为 `const`，这段代码可以编译通过。

```
const size_t height { 10 };
Grid<int, 10, height> testGrid; // Compiles and works
```

带有正确返回类型的 `constexpr` 函数也可以编译。例如，如果有一个返回 `size_t` 的 `constexpr` 函数，那么可以使用它初始化 `height` 模板参数。

```
constexpr size_t getHeight() { return 10; }
...
Grid<double, 2, getHeight()> myDoubleGrid;
```

另一个限制可能更明显。既然宽度和高度都是模板参数，那么它们也是每种网格类型的一部分。这意味着 `Grid<int, 10, 10>` 和 `Grid<int, 10, 11>` 是两种不同类型。不能将一种类型的对象赋给另一种类型的对象，而且一种类型的变量不能传递给接收另一种类型的变量的函数或方法。

### 注意：

非类型模板参数是实例化的对象的类型规范中的一部分。

## 2. 类型参数的默认值

如果继续采用将高度和宽度作为模板参数的方式，就可能需要为高度和宽度（它们是非类型模板参数）提供默认值，就像之前 `Grid<T>` 类的构造函数一样。C++ 允许使用类似的语法向模板参数提供默认值。在这里也可以给 `T` 类型参数提供默认值。下面是类定义：

```
export template <typename T = int, size_t WIDTH = 10, size_t HEIGHT = 10>
class Grid
{
    // Remainder is identical to the previous version
};
```

不需要在方法定义的模板规范中指定 `T`、`WIDTH` 和 `HEIGHT` 的默认值。例如，下面是 `at()` 方法的实现：

```
template <typename T, size_t WIDTH, size_t HEIGHT>
const std::optional<T>& Grid<T, WIDTH, HEIGHT>::at(size_t x, size_t y) const
{
    verifyCoordinate(x, y);
    return m_cells[x][y];
}
```

现在，实例化 Grid 时，可以不指定模板参数，只指定元素类型，或者指定元素类型和宽度，或者指定元素类型、宽度和高度。

```
Grid<> myIntGrid;
Grid<int> myGrid;
Grid<int, 5> anotherGrid;
Grid<int, 5, 5> aFourthGrid;
```

注意，如果未指定任何类模板参数，那么仍需要指定一组空尖括号。例如，以下代码无法编译！

```
Grid myIntGrid;
```

模板参数列表中默认参数的规则与函数或方法是一样的。可以从右向左提供参数的默认值。

### 3. 类模板的实参推导

通过类模板实参推导，编译器可以自动从传递给类模板构造函数的实参推导出模板参数。

例如，标准库有一个名为 std::pair 的类模板，定义在<utility>中，在前面第 1 章中介绍过。一个 pair 存储两种不同类型的两个值，必须将其指定为模板参数。示例如下：

```
pair<int, double> pair1 { 1, 2.3 };
```

为避免编写模板参数，可以使用一个辅助的函数模板 std::make\_pair()。本章后面将详细讨论函数模板。函数模板始终支持基于传递给函数模板的实参自动推导模板参数。因此，make\_pair()能根据传递给它的值自动推导模板类型参数。例如，编译器为以下调用推导出 pair<int, double>：

```
auto pair2 { make_pair(1, 2.3) };
```

使用类模板实参推导(CTAD)，这样的辅助函数模板就不再需要了。现在，编译器可以根据传递给构造函数的实参自动推导模板类型参数。对于 pair 类模板，只需要编写以下代码：

```
pair pair3 { 1, 2.3 }; // pairs has type pair<int, double>
```

当然，推导的前提是类模板的所有模板参数要么有默认值，要么用作构造函数中的参数。

注意，CTAD 需要一个初始化式才能工作。以下是非法的：

```
pair pair4;
```

许多标准库类都支持 CTAD，例如，向量、数组等。

#### 注意：

std::unique\_ptr 和 shared\_ptr 会禁用类型推导。给它们的构造函数传递 T\*，这意味着，编译器必须选择推导<T>还是<T[]>，这是一个可能出错的危险选择。因此只需要记住，对于 unique\_ptr 和 shared\_ptr，需要继续使用 make\_unique() 和 make\_shared()。

#### 用户定义的推导原则

可以编写自己的推导原则(即，用户定义的推导原则)来帮助编译器。它允许编写如何推导模板参数的规则。这是一个高级主题，这里不对其进行详细讨论，但会举一个例子来演示其功能。

假设具有以下 SpreadsheetCell 类模板：

```
template<typename T>
class SpreadsheetCell
{
public:
    SpreadsheetCell(const T& t) : m_content(t) {}
```

```

    const T& getContent() const { return m_content; }
private:
    T m_content;
};

```

通过自动推导模板参数，可使用 std::string 类型创建 SpreadsheetCell：

```

string myString { "Hello World!" };
SpreadsheetCell cell { myString };

```

但是，如果给 SpreadsheetCell 构造函数传递 const char\*，那么会将类型 T 推导为 const char\*，这不是需要的结果。可以创建以下用户定义的推导原则，在将 const char\* 作为实参传递给构造函数时，将 T 推导为 std::string。

```
SpreadsheetCell(const char*) -> SpreadsheetCell<std::string>;
```

在与 SpreadsheetCell 类相同的名称空间中，在类定义之外定义该原则。

通用语法如下。explicit 关键字是可选的，它的行为与构造函数的 explicit 相同。通常，这些推导规则也是模板。

```
explicit TemplateName(Parameters) -> DducedTemplate;
```

## 12.2.5 方法模板

C++ 允许模板化类中的单个方法。这些方法称为方法模板(method template)，它们可以在类模板中，也可以在非模板化的类中。在编写方法模板时，实际上是在为很多不同的类型编写很多不同版本的方法。在类模板中，方法模板对赋值运算符和复制构造函数非常有用。

### 警告：

不能用方法模板编写虚方法和析构函数。

考虑最早只有一个模板参数(即，元素类型)的 Grid 模板。可实例化很多不同类型的网格，例如 int 网格和 double 网格。

```
Grid<int> myIntGrid;
Grid<double> myDoubleGrid;
```

然而，Grid<int> 和 Grid<double> 是两种不同的类型。如果编写的函数接收类型为 Grid<double> 的对象，就不能传入 Grid<int>。即使 int 网格中的元素可以复制到 double 网格中(因为 int 可以强制转换为 double)，也不能将类型为 Grid<int> 的对象赋给类型为 Grid<double> 的对象，也不能从 Grid<int> 构造 Grid<double>。下面两行代码都无法编译：

```
myDoubleGrid = myIntGrid; // DOES NOT COMPILE
Grid<double> newDoubleGrid { myIntGrid }; // DOES NOT COMPILE
```

问题在于 Grid 模板的复制构造函数和赋值运算符如下所示：

```
Grid(const Grid& src);
Grid& operator=(const Grid& rhs);
```

它们等同于：

```
Grid(const Grid<T>& src);
Grid<T>& operator=(const Grid<T>& rhs);
```

Grid 复制构造函数和 operator= 都会接收 const Grid<T> 的引用作为参数。当实例化 Grid<double>, 并试图调用复制构造函数和 operator= 时, 编译器通过这些原型生成方法:

```
Grid(const Grid<double>& src);
Grid<double>& operator=(const Grid<double>& rhs);
```

注意, 在生成的 Grid<double> 类中, 构造函数或 operator= 都不会接收 Grid<int> 作为参数。

幸运的是, 在 Grid 类中添加模板化的复制构造函数和赋值运算符, 可生成一种将网格类型转换为另一种网格类型的方法, 从而修复这个疏漏。下面是新的 Grid 类定义:

```
export template <typename T>
class Grid
{
public:
    template <typename E>
    Grid(const Grid<E>& src);

    template <typename E>
    Grid<T>& operator=(const Grid<E>& rhs);

    void swap(Grid& other) noexcept;

    // Omitted for brevity
};
```

不能删除原始复制构造函数和复制赋值运算符。如果 E 等于 T, 那么编译器将不会调用这些新的模板化复制构造函数和模板化复制赋值运算符。

首先检查新的模板化的复制构造函数:

```
template <typename E>
Grid(const Grid<E>& src);
```

可看到另一个具有不同类型名称 E(即, element 的简写)的模板声明。这个类在类型 T 上被模板化, 这个新的复制构造函数又在另一个不同的类型 E 上被模板化。通过这种双重模板化可将一种类型的网格复制到另一种类型的网格。下面是新的复制构造函数的定义:

```
template <typename T>
template <typename E>
Grid<T>::Grid(const Grid<E>& src)
    : Grid { src.getWidth(), src.getHeight() }
{
    // The ctor-initializer of this constructor delegates first to the
    // non-copy constructor to allocate the proper amount of memory.

    // The next step is to copy the data.
    for (size_t i { 0 }; i < m_width; i++) {
        for (size_t j { 0 }; j < m_height; j++) {
            m_cells[i][j] = src.at(i, j);
        }
    }
}
```

可以看出，必须将声明类模板的那一行(带有 T 参数)放在成员模板的那一行声明(带有 E 参数)的前面。不能像下面这样合并两者：

```
template <typename T, typename E> // Wrong for nested template constructor!
Grid<T>::Grid(const Grid<E>& src)
```

除了构造函数定义之前的额外模板参数行之外，注意必须通过公共的访问方法 `getWidth()`、`getHeight()` 和 `at()` 访问 `src` 中的元素。这是因为复制目标对象的类型为 `Grid<T>`，而复制来源对象的类型为 `Grid<E>`。这两者不是同一类型，因此必须使用公共方法。

`swap()` 方法的实现如下所示：

```
template <typename T>
void Grid<T>::swap(Grid& other) noexcept
{
    std::swap(m_width, other.m_width);
    std::swap(m_height, other.m_height);
    std::swap(m_cells, other.m_cells);
}
```

模板化的赋值运算符接收 `const Grid<E>&` 作为参数，但返回 `Grid<T>&`。

```
template <typename T>
template <typename E>
Grid<T>& Grid<T>::operator=(const Grid<E>& rhs)
{
    // Copy-and-swap idiom
    Grid<T> temp { rhs }; // Do all the work in a temporary instance.
    swap(temp); // Commit the work with only non-throwing operations.
    return *this;
}
```

这个赋值运算符的实现使用第 9 章介绍的“复制和交换”惯用语法。`swap()` 方法只能交换同类网格，但这是可行的，因为模板化的赋值运算符首先使用模板化的复制构造函数，将给定的 `Grid<E>` 转换为 `Grid<T>`(称为 `temp`)。此后，它使用 `swap()` 方法，将 `this`(也是 `Grid<T>` 类型) 替换临时的 `Grid<T>`。

### 带有非类型参数的方法模板

在之前用于 `HEIGHT` 和 `WIDTH` 整数模板参数的例子中，一个主要问题是高度和宽度成为类型的一部分。因为存在这个限制，所以不能将一个拥有某种高度和宽度的网格赋值给另一个拥有不同高度和宽度的网格。不一定要把源对象完美地复制到目标对象，可从源数组中只复制那些能够放在目标数组中的元素；如果源数组在任何一个维度上都比目标数组小，那么可以用默认值填充目标数组。有了赋值运算符和复制构造函数的方法模板后，完全可实现这个操作，从而允许对不同大小的网格进行赋值和复制。下面是类定义：

```
export template <typename T, size_t WIDTH = 10, size_t HEIGHT = 10>
class Grid
{
public:
    Grid() = default;
    virtual ~Grid() = default;

    // Explicitly default a copy constructor and assignment operator.
    Grid(const Grid& src) = default;
```

```

Grid& operator=(const Grid& rhs) = default;

template <typename E, size_t WIDTH2, size_t HEIGHT2>
Grid(const Grid<E, WIDTH2, HEIGHT2>& src);

template <typename E, size_t WIDTH2, size_t HEIGHT2>
Grid& operator=(const Grid<E, WIDTH2, HEIGHT2>& rhs);

void swap(Grid& other) noexcept;

std::optional<T>& at(size_t x, size_t y);
const std::optional<T>& at(size_t x, size_t y) const;

size_t getHeight() const { return HEIGHT; }
size_t getWidth() const { return WIDTH; }

private:
    void verifyCoordinate(size_t x, size_t y) const;

    std::optional<T> m_cells[WIDTH][HEIGHT];
};

```

这个新定义包含复制构造函数和赋值运算符的方法模板，还包含辅助方法 swap()。注意，将非模板化的复制构造函数和赋值运算符显式设置为默认(原因在于用户声明的析构函数)。这些方法只是将 m\_cells 从源对象复制或赋值到目标对象，语义和两个一样大小的网格的语义完全一致。

下面是模板化的复制构造函数：

```

template <typename T, size_t WIDTH, size_t HEIGHT>
template <typename E, size_t WIDTH2, size_t HEIGHT2>
Grid<T, WIDTH, HEIGHT>::Grid(const Grid<E, WIDTH2, HEIGHT2>& src)
{
    for (size_t i { 0 }; i < WIDTH; i++) {
        for (size_t j { 0 }; j < HEIGHT; j++) {
            if (i < WIDTH2 && j < HEIGHT2) {
                m_cells[i][j] = src.at(i, j);
            } else {
                m_cells[i][j].reset();
            }
        }
    }
}

```

注意，该复制构造函数只从 src 在 x 维度和 y 维度上分别复制 WIDTH 和 HEIGHT 个元素，即使 src 比 WIDTH 和 HEIGHT 指定的大小要大，也是如此。如果 src 在任何一个维度上都比这个指定值小，那么可以使用 reset()方法重置附加点中的 std::optional 对象。

下面是 swap() 和 operator= 的实现：

```

template <typename T, size_t WIDTH, size_t HEIGHT>
void Grid<T, WIDTH, HEIGHT>::swap(Grid& other) noexcept
{
    std::swap(m_cells, other.m_cells);
}

template <typename T, size_t WIDTH, size_t HEIGHT>
template <typename E, size_t WIDTH2, size_t HEIGHT2>

```

```

Grid<T, WIDTH, HEIGHT>& Grid<T, WIDTH, HEIGHT>::operator=(  

    const Grid<E, WIDTH2, HEIGHT2>& rhs)  

{  

    // Copy-and-swap idiom  

    Grid<T, WIDTH, HEIGHT> temp { rhs }; // Do all the work in a temp instance.  

    swap(temp); // Commit the work with only non-throwing operations.  

    return *this;  

}

```

## 12.2.6 类模板的特化

对于特定类型，可以给类模板提供不同的实现。例如，Grid的行为对const char\*(C风格的字符串)来说没有意义。Grid<const char\*>将在vector<vector<optional<const char\*>>>中存储其元素。复制构造函数和赋值运算符执行这种const char\*指针类型的浅层复制(shallow copy)。对于const char\*来说，对字符串进行深层复制(deep copy)才有意义。最简单的解决方法是专门给const char\*编写另一个实现，把C风格的字符串转换为C++字符串，并且将字符串存储在vector<vector<optional<string\*>>>中。

模板的另一个实现称为模板特化(template specialization)。模板特化的语法也有点奇怪。编写一个类模板的特化时，必须指明这是一个模板，以及正在为哪种特定的类型编写这个模板。下面是为const char\*特化的Grid的语法。对于这个实现，原始的Grid类模板被移动到一个名为main的模块接口分区中，并且特化在一个名为string的模块接口分区中。

```

export module grid:string;
// When the template specialization is used, the original template must be
// visible too.
import :main;

export template <>
class Grid<const char*>
{
public:
    explicit Grid(size_t width = DefaultWidth,
                  size_t height = DefaultHeight);
    virtual ~Grid() = default;

    // Explicitly default a copy constructor and assignment operator.
    Grid(const Grid& src) = default;
    Grid& operator=(const Grid& rhs) = default;

    // Explicitly default a move constructor and assignment operator.
    Grid(Grid&& src) = default;
    Grid& operator=(Grid&& rhs) = default;

    std::optional<std::string>& at(size_t x, size_t y);
    const std::optional<std::string>& at(size_t x, size_t y) const;

    size_t getHeight() const { return m_height; }
    size_t getWidth() const { return m_width; }

    static const size_t DefaultWidth { 10 };
    static const size_t DefaultHeight { 10 };

private:
    void verifyCoordinate(size_t x, size_t y) const;
}

```

```

    std::vector<std::vector<std::optional<std::string>>> m_cells;
    size_t m_width { 0 }, m_height { 0 };
};

}

```

注意，在这个特化中不要指定任何类型变量(例如 T)，而是直接处理 `const char*` 和 `string`。现在有个明显的问题是：为什么这个类仍然是一个模板。也就是说，下面这种语法有什么意义？

```

template <>
class Grid<const char*>

```

上述语法告诉编译器，这个类是 `Grid` 类的 `const char*` 特化版本。假设没有使用这种语法的经验，那么尝试编写下面这样的代码：

```
class Grid
```

编译器不允许这样做，因为已经有一个名为 `Grid` 的类(原始的类模板)。只能通过特化重用这个名称。特化的主要好处就是可以对用户隐藏。当用户创建 `int` 类型或 `SpreadsheetCell` 类型的 `Grid` 时，编译器从原始的 `Grid` 模板生成代码。当用户创建 `const char*` 类型的 `Grid` 时，编译器会使用 `const char*` 的特化版本。这些全部在后台自动完成。

基本模块接口文件简单地导入和导出两个模块的接口分区：

```

export module grid;

export import :main;
export import :string;

```

特化的测试方法如下：

```

Grid<int> myIntGrid; // Uses original Grid template
Grid<const char*> stringGrid1 { 2, 2 }; // Uses const char* specialization

const char* dummy { "dummy" };
stringGrid1.at(0, 0) = "hello";
stringGrid1.at(0, 1) = dummy;
stringGrid1.at(1, 0) = dummy;
stringGrid1.at(1, 1) = "there";

Grid<const char*> stringGrid2 { stringGrid1 };

```

特化一个模板时，并没有“继承”任何代码；特化和派生不同。必须重新编写类的整个实现。不要求提供具有相同的名称或行为的方法。例如，`Grid` 的 `const char*` 特化仅实现 `at()` 方法，返回 `optional<string>`，而非 `optional<const char*>`。事实上，可以编写一个和原来的类无关的、完全不同的类。当然，这样做会滥用模板特化的功能，如果没有正当理由，不应该这样做。下面是 `const char*` 特化版本的方法的实现。与模板定义不同，不必在每个方法定义之前重复 `template <>` 语法：

```

Grid<const char*>::Grid(size_t width, size_t height)
    : m_width { width }, m_height { height }
{
    m_cells.resize(m_width);
    for (auto& column : m_cells) {
        column.resize(m_height);
    }
}

```

```

void Grid<const char*>::verifyCoordinate(size_t x, size_t y) const
{
    if (x >= m_width) {
        throw std::out_of_range {
            std::format("{} must be less than {}.", x, m_width) };
    }
    if (y >= m_height) {
        throw std::out_of_range {
            std::format("{} must be less than {}.", y, m_height) };
    }
}

const std::optional<std::string>& Grid<const char*>::at(
    size_t x, size_t y) const
{
    verifyCoordinate(x, y);
    return m_cells[x][y];
}

std::optional<std::string>& Grid<const char*>::at(size_t x, size_t y)
{
    return const_cast<std::optional<std::string>&>(
        std::as_const(*this).at(x, y));
}

```

本节讨论了如何使用类模板特化。通过这项特性，当模板类型被特定类型替换时，可以为模板编写特殊的实现。第 26 章“高级模板”将继续讨论特化，讨论的是一项称为部分特化(partial specialization)的更高级特性。

### 12.2.7 从类模板派生

可从类模板中派生。如果一个派生类从模板本身继承，那么这个派生类也必须是模板。此外，还可从类模板派生某个特定实例，这种情况下，这个派生类不需要是模板。下面针对前一种情况举一个例子，假设 Grid 类没有提供足够的棋盘功能。确切地讲，要给棋盘添加 move() 方法，允许棋盘上的棋子从一个位置移动到另一个位置。下面是这个 GameBoard 模板的类定义：

```

import grid;

export template <typename T>
class GameBoard : public Grid<T>
{
public:
    explicit GameBoard(size_t width = Grid<T>::DefaultWidth,
                       size_t height = Grid<T>::DefaultHeight);
    void move(size_t xSrc, size_t ySrc, size_t xDest, size_t yDest);
};

```

这个 GameBoard 模板派生自 Grid 模板，因此继承了 Grid 模板的所有功能。不需要重写 at()、getHeight() 以及其他任何方法。也不需要添加复制构造函数、operator= 或析构函数，因为在 GameBoard 中没有任何动态分配的内存。

继承的语法和普通继承一样，区别在于基类是 Grid<T>，而不是 Grid。设计这种语法的原因是 GameBoard 模板并没有真正地派生自通用的 Grid 模板。相反，GameBoard 模板对特定类型的每个实例化都派生自 Grid 对那种类型的实例化。例如，如果使用 ChessPiece 类型实例化 GameBoard，那么

编译器也会生成 `Grid<ChessPiece>` 的代码。“`: public Grid<T>`” 语法表明，这个类继承了 `Grid` 实例化对类型参数 `T` 有意义的所有内容。注意，虽然某些编译器并不强制它，但 C++ 名称查找的规则要求使用 `this` 指针或 `Grid<T>::` 来指涉基类模板中的数据成员和方法。因此，使用 `Grid::DefaultWidth` 替换 `DefaultWidth`。

下面是构造函数和 `move()` 方法的实现：

```
template <typename T>
GameBoard<T>::GameBoard(size_t width, size_t height)
    : Grid<T> { width, height }
{ }

template <typename T>
void GameBoard<T>::move(size_t xSrc, size_t ySrc, size_t xDest, size_t yDest)
{
    Grid<T>::at(xDest, yDest) = std::move(Grid<T>::at(xSrc, ySrc));
    Grid<T>::at(xSrc, ySrc).reset(); // Reset source cell
    // Or:
    // this->at(xDest, yDest) = std::move(this->at(xSrc, ySrc));
    // this->at(xSrc, ySrc).reset();
}
```

可使用如下 `GameBoard` 模板：

```
GameBoard<ChessPiece> chessboard { 8, 8 };
ChessPiece pawn;
chessBoard.at(0, 0) = pawn;
chessBoard.move(0, 0, 0, 1);
```

#### 注意：

当然，如果你想重载来自 `Grid` 的方法，则必须在 `Grid` 类模板中将它们标记为虚方法。

### 12.2.8 继承还是特化

有些程序员感觉模板继承和模板特化之间的区别很难理解。表 12-1 总结了两者的区别。

表 12-1 模板继承和模板特化之间的区别

	继承	特化
是否重用代码	是：派生类包含基类的所有成员和方法	否：必须在特化中重写需要的所有代码
是否重用名称	否：派生类名必须和基类名不同	是：特化的名称必须和原始名称一致
是否支持多态	是：派生类的对象可以代替基类的对象	否：模板对一种类型的每个实例化都是不同类型

#### 注意：

通过继承扩展实现和使用多态。通过特化自定义特定类型的实现。

### 12.2.9 模板别名

第 1 章介绍了类型别名和 `typedef` 的概念。通过 `typedef` 可给特定类型赋予另一个名称。例如可以编写以下类型别名，给类型 `int` 指定另一个名称。

```
using MyInt = int;
```

类似地，可使用类型别名给模板化的类赋予另一个名称。假定有如下类模板：

```
template<typename T1, typename T2>
class MyTemplateClass { /* ... */ };
```

可以定义如下类型别名，给定两个类模板的类型参数。

```
using OtherName = MyTemplateClass<int, double>;
```

也可以用 `typedef` 替代类型别名。

还可以仅指定一些类型，其他类型则保持为模板类型参数，这称为别名模板(alias template)。例如：

```
template<typename T1>
using OtherName = MyTemplateClass<T1, double>;
```

这无法用 `typedef` 完成。

## 12.3 函数模板

还可以为独立函数编写模板。例如，可以编写一个通用函数，该函数在数组中查找一个值并返回这个值的索引：

```
static const size_t NOT_FOUND = static_cast<size_t>(-1);

template <typename T>
size_t Find(const T& value, const T* arr, size_t size)
{
    for (size_t i = 0; i < size; i++) {
        if (arr[i] == value) {
            return i; // Found it; return the index
        }
    }
    return NOT_FOUND; // Failed to find it; return NOT_FOUND
}
```

### 注意：

当然，如果找不到元素，可以不返回一些 sentinel 值(例如 `NOT_FOUND`)，可重写代码，返回 `std::optional<size_t>` 而非 `size_t`。这将会是使用 `std::optional` 的有趣练习。

这个 `Find()` 函数可用于任何类型的数组。例如，可通过这个函数在 `int` 数组中查找一个 `int` 值的索引，还可在 `SpreadsheetCell` 数组中查找一个 `SpreadsheetCell` 值的索引。

可通过两种方式调用这个函数：一种是通过尖括号显式地指定类型；另一种是忽略类型，让编译器根据实参自动推导类型。下面列举一些例子：

```
int myInt = 3, intArray[] = {1, 2, 3, 4};
const size_t sizeIntArray = std::size(intArray);

size_t res;
res = Find(myInt, intArray, sizeIntArray);           // calls Find<int> by deduction
res = Find<int>(myInt, intArray, sizeIntArray); // calls Find<int> explicitly
if (res != NOT_FOUND)
    cout << res << endl;
else
```

```

cout << "Not found" << endl;

double myDouble = 5.6, doubleArray[] = {1.2, 3.4, 5.7, 7.5};
const size_t sizeDoubleArray = std::size(doubleArray);

// calls Find<double> by deduction
res = Find(myDouble, doubleArray, sizeDoubleArray);
// calls Find<double> explicitly
res = Find<double>(myDouble, doubleArray, sizeDoubleArray);
if (res != NOT_FOUND)
    cout << res << endl;
else
    cout << "Not found" << endl;

//res = Find(myInt, doubleArray, sizeDoubleArray); // DOES NOT COMPILE!
// Arguments are different types
// calls Find<double> explicitly, even with myInt
res = Find<double>(myInt, doubleArray, sizeDoubleArray);

SpreadsheetCell cell1(10), cellArray[] =
    {SpreadsheetCell(4), SpreadsheetCell(10)};
const size_t sizeCellArray = std::size(cellArray);

res = Find(cell1, cellArray, sizeCellArray);
res = Find<SpreadsheetCell>(cell1, cellArray, sizeCellArray);

```

前面 Find() 函数的实现需要把数组的大小作为一个参数。有时编译器知道数组的确切大小，例如，基于堆栈的数组。用这种数组调用 Find() 函数，就不需要传递数组的大小。为此，可添加如下函数模板。该实现仅把调用传递给前面的 Find() 函数模板。这也说明函数模板可接收非类型的参数，与类模板一样。

```

template <typename T, size_t N>
size_t Find(const T& value, const T(&arr) [N])
{
    return Find(value, arr, N);
}

```

这个版本的 Find() 函数语法有些特殊，但其用法相当直接，如下所示。

```

int myInt { 3 }, intArray[] {1, 2, 3, 4};
size_t res { Find(myInt, intArray) };

```

与类模板方法定义一样，函数模板定义（不仅是原型）必须对所有使用它们的源文件可用。因此，如果多个源文件使用函数模板，或使用本章前面讨论的显式实例化，就应把其定义放在头文件中。函数模板的模板参数可以具有默认值，这与类模板一样。

### 注意：

C++ 标准库提供了功能比这里的 Find() 函数模板更强大的模板化函数 std::find()。详情请参阅第 18 章。

### 12.3.1 函数模板的重载

理论上，C++ 语言允许编写函数模板特化，就像编写类模板特化一样。但是，很少会想要这样做，由于这样的函数模板特化不参与重载解析，因此可能会出现意外的行为。

相反，可以使用非模板函数来重载函数模板。例如，假如想为 `const char*` 类型的 C 风格的字符串编写一个 `Find()` 重载，以便与 `strcmp()`（参见第 2 章“处理字符串和字符串视图”）进行比较，而不是使用 `operator==`，因为 `=` 只会比较指针（地址），而不是实际的字符串。下面是一个这样的重载：

```
size_t Find(const char* value, const char** arr, size_t size)
{
    for (size_t i { 0 }; i < size; i++) {
        if (strcmp(arr[i], value) == 0) {
            return i; // Found it; return the index
        }
    }
    return NOT_FOUND; // Failed to find it; return NOT_FOUND
}
```

这个函数重载的使用示例如下：

```
const char* word { "two" };
const char* words[] { "one", "two", "three", "four" };
const size_t sizeWords { std::size(words) };
size_t res { Find(word, words, sizeWords); }; // Calls non-template function!
```

如果确实像下面这样显式地指定模板类型参数，那么函数模板将被 `T=const char*` 调用，而不是 `const char*` 的重载。

```
res = Find<const char*>(word, words, sizeWords);
```

### 12.3.2 类模板的友元函数模板

如果需要在类模板中重载运算符，函数模板会非常有用。例如，可以通过重载 `Grid` 类模板的加法运算符(`operator+`)，把两个网格加在一起。这样得到的网格大小与两个操作数中较小的网格相同。只有在两个网格都包含实际值时，才会将相应的网格相加。假设希望将 `operator+` 设置为独立的函数模板。它的定义应该放在 `Grid.cppm` 模块接口文件中，如下所示。这个实现使用`<algorithm>`中定义的 `std::min()` 返回两个给定实参中的最小值：

```
export template <typename T>
Grid<T> operator+(const Grid<T>& lhs, const Grid<T>& rhs)
{
    size_t minWidth { std::min(lhs.getWidth(), rhs.getWidth()) };
    size_t minHeight { std::min(lhs.getHeight(), rhs.getHeight()) };

    Grid<T> result { minWidth, minHeight };
    for (size_t y { 0 }; y < minHeight; ++y) {
        for (size_t x { 0 }; x < minWidth; ++x) {
            const auto& leftElement { lhs.m_cells[x][y] };
            const auto& rightElement { rhs.m_cells[x][y] };
            if (leftElement.has_value() && rightElement.has_value()) {
                result.at(x, y) = leftElement.value() + rightElement.value();
            }
        }
    }
    return result;
}
```

要查询 `std::optional` 是否包含实际值，可使用 `has_value()` 方法，同时使用 `value()` 方法检索这个值。

这个函数模板可用于任何网格，只要网格中的元素支持相加运算符即可。这个实现的唯一问题是 operator+ 访问了 Grid 类的私有成员 m\_cells。显然，解决方法是使用公有的 at() 方法，下面看看如何把函数模板作为类模板的友元。对于这个示例，可将该运算符作为 Grid 类的友元。然而，Grid 类和 operator+ 都是模板。实际上需要以下效果：operator+ 对每一种特定类型 T 的实例化都是 Grid 模板对这种类型实例化的友元。语法如下所示：

```
// Forward declare Grid template.
export template <typename T> class Grid;

// Prototype for templatized operator+.
export template<typename T>
Grid<T> operator+(const Grid<T>& lhs, const Grid<T>& rhs);

export template <typename T>
class Grid
{
public:
    friend Grid<T> operator+<T>(const Grid& lhs, const Grid& rhs);
    // Omitted for brevity
};


```

友元声明比较棘手：上述语法表明，对于这个模板对类型 T 的实例，operator+ 对类型 T 的实例是这个模板实例的友元。换句话说，类实例和函数实例之间存在一对一的友元映射关系。特别要注意 operator+ 中显式的模板规范<T>(operator+ 后面的空格是可选的)。这一语法告诉编译器，operator+ 本身也是模板。

### 12.3.3 对模板参数推导的更多介绍

编译器根据传递给函数模板的实参来推导模板参数的类型；而对于无法推导的模板参数，则需要显式指定。例如，如下 add() 函数模板需要 3 个模板参数：返回值的类型以及两个操作数的类型。

```
template<typename RetType, typename T1, typename T2>
RetType add(const T1& t1, const T2& t2) { return t1 + t2; }
```

调用这个函数模板时，可指定如下所有 3 个参数。

```
auto result { add<long long, int, int>(1, 2) };
```

但由于模板参数 T1 和 T2 是函数的参数，编译器可以推导这两个参数，因此调用 add() 时可仅指定返回值的类型。

```
auto result { add<long long>(1, 2) };
```

当然，仅在要推导的参数位于参数列表的最后时，这才可行。假设以如下方式定义函数模板：

```
template<typename T1, typename RetType, typename T2>
RetType add(const T1& t1, const T2& t2) { return t1 + t2; }
```

必须指定 RetType，因为编译器无法推导该类型。但由于 RetType 是第 2 个参数，因此必须显式指定 T1。

```
auto result { add<int, long long>(1, 2) };
```

也可提供返回类型模板参数的默认值，这样调用 add() 时可不指定任何类型。

```
template<typename RetType = long long, typename T1, typename T2>
RetType add(const T1& t1, const T2& t2) { return t1 + t2; }

...
auto result { add(1, 2) };
```

### 12.3.4 函数模板的返回类型

继续分析 add() 函数模板的示例，使编译器推导返回值的类型岂不更好？确实是好，但返回类型取决于模板类型参数，如何才能做到这一点？例如，考虑如下模板函数：

```
template<typename T1, typename T2>
RetType add(const T1& t1, const T2& t2) { return t1 + t2; }
```

在这个示例中，RetType 应当是表达式  $t1+t2$  的类型，但由于不知道 T1 和 T2 是什么类型，因此并不知道这一点。

如第 1 章所述，从 C++14 开始，可要求编译器自动推导函数的返回类型。因此，只需要编写如下 add() 函数模板：

```
template<typename T1, typename T2>
auto add(const T1& t1, const T2& t2) { return t1 + t2; }
```

但是，使用 auto 推导表达式类型时去掉了引用和 const 限定符；但是 decltype 没有去除这些限定符。这种剥离对于 add() 函数模板来说是合适的，因为 operator+ 通常会返回一个新对象，但这种剥离对于某些其他函数模板可能并不理想，所以需要考虑如何避免这种剥离。但是，在继续函数模板示例之前，首先使用一个非模板示例，来看看 auto 和 decltype 之间的区别。假设具有以下功能：

```
const std::string message { "Test" };

const std::string& getString() { return message; }
```

可以调用 getString()，并且将结果存储在 auto 类型的变量中，如下所示。

```
auto s1 { getString() };
```

由于 auto 会去掉引用和 const 限定符，因此 s1 的类型是 string，并制作一个副本。如果需要一个 const 引用，可将其显式地设置为引用，并标记为 const，如下所示。

```
const auto& s2 { getString() };
```

另一个解决方案是使用 decltype，decltype 不会去掉引用和 const 限定符。

```
decltype(getString()) s3 { getString() };
```

这里，s3 的类型是 const string&，但存在代码冗余，因为需要指定 getString() 两次。如果 getString() 是更复杂的表达式，这就会很麻烦。

为解决这个问题，可以使用 decltype(auto)：

```
decltype(auto) s4 { getString() };
```

s4 的类型也是 const string&。

了解到这些后，可使用 decltype(auto) 编写 add() 函数，以避免去掉任何 const 和引用限定符。

```
template<typename T1, typename T2>
decltype(auto) add(const T1& t1, const T2& t2) { return t1 + t2; }
```

在 C++14 之前，不支持推导函数的返回类型和 decltype(auto)。C++11 引入的 decltype(expression) 解决了这个问题。例如，或许会编写如下代码：

```
template<typename T1, typename T2>
decltype(t1+t2) add(const T1& t1, const T2& t2) { return t1 + t2; }
```

但这是错误的。在原型行的开头使用了 t1 和 t2，但这些尚且不知。在语义分析器到达参数列表的末尾时，才能知道 t1 和 t2。

通常使用替换函数语法(alternative function syntax)解决这个问题。注意，在这种语法中，auto 被用于原型行的开头，而实际返回类型是在参数列表之后指定的(后置返回类型)；因此，在解析时参数的名称(以及参数的类型，因此也包括类型 t1+t2)是已知的。

```
template<typename T1, typename T2>
auto add(const T1& t1, const T2& t2) -> decltype(t1+t2)
{
    return t1 + t2;
}
```

#### 注意：

现在，C++ 支持自动返回类型推导和 decltype(auto)，建议使用其中的一种机制，而不要使用替换函数语法。

C++20

## 12.4 简化函数模板的语法

C++20 引入了一个简化函数模板的语法。再回顾前一节中的 add() 函数模板。下面是推荐的版本：

```
template<typename T1, typename T2>
decltype(auto) add(const T1& t1, const T2& t2) { return t1 + t2; }
```

在这里，指定一个简单的函数模板是一种相当冗长的语法。使用简化函数模板的语法，可以更优雅地编写如下代码：

```
decltype(auto) add(const auto& t1, const auto& t2) { return t1 + t2; }
```

使用这种语法，不再有 template<> 规范来指定模板参数。相反，以前的实现使用 T1 和 T2 作为函数参数的类型，现在将它们指定为 auto。这种简化的语法只是语法糖；编译器会自动将这个简化的实现转换为更长的原始代码。基本上，每个被指定为 auto 的函数参数都成为模板类型参数。

但必须牢记两个注意事项。首先，每个指定为 auto 的参数都会成为不同的模板类型参数。假设有一个这样的函数模板：

```
template<typename T>
decltype(auto) add(const T& t1, const T& t2) { return t1 + t2; }
```

这个版本只有一个模板类型参数，函数的 2 个参数 t1 和 t2 都是 const T& 类型。对于这样的函数模板，不能使用简化的语法，因为这将转换为具有 2 个不同模板类型参数的函数模板。

第 2 个问题是，不能在函数模板的实现中显式地使用导出的类型，因为这些自动导出的类型没有

名称。如果需要这样做，那么可以继续使用普通的函数模板语法，或者使用 `decltype()` 确定其类型。

## 12.5 变量模板

除了类模板、类方法模板和函数模板外，C++还支持变量模板。语法如下：

```
template <typename T>
constexpr T pi { T { 3.141592653589793238462643383279502884 } };
```

这是 `pi` 值的可变模板。为了在某种类型中获得 `pi` 值，可以使用如下语法：

```
float piFloat { pi<float> };
auto piLongDouble { pi<long double> };
```

这样总会得到在所请求的类型中可表示的 `pi` 近似值。与其他类型的模板一样，变量模板也可以特化。

**注意：**

C++20 引入了`<numbers>`，它定义了一组常用的数学常量，包括 `pi: std::numbers::pi`。



## 12.6 概念

C++20 引入了概念(concept)，命名需求用来约束类模板和函数模板的模板类型和非类型参数。这些是作为谓词编写的，在编译期计算这些谓词，以验证传递给模板的模板参数。概念的主要目标是使与模板相关的编译器错误更具有可读性。每个人都遇到过这样的情况：当为类或函数模板提供了错误的参数时，编译器会抛出数百行的错误信息。但要找出那些编译器错误的根本原因并不总是那么容易。

概念允许编译器在不满足某些类型约束时输出可读的错误消息。因此，为了得到有意义的语义错误，建议编写代码时使用概念来建模语义需求。避免只验证没有任何语义意义的语法方面的概念，例如只检查类型是否支持 `operator+` 的概念。这样的概念将只检查语法，而不是语义。`std::字符串` 支持 `operator+`，但很明显，它与整数的 `operator+` 具有完全不同的含义。另一方面，诸如可排序和可交换的概念是使用概念对某些语义进行建模的好例子。

**注意：**

编写概念时，请确保它们是在为语义建模，而不仅仅是语法建模。

接下来介绍编写概念的语法。

### 12.6.1 语法

概念定义的泛型语法如下：

```
template <parameter-list>
concept concept-name = constraints-expression;
```

这里从熟悉的模板`<>`说明符开始，但与类模板和函数模板不同，概念从不会被实例化。接下来，使用一个新的关键字 `concept`，然后是概念的名称。名称可以是任意的。`constraints-expression` 可以是任意的常量表达式(即，可以在编译期计算的任何表达式)。约束表达式必须要产生一个布尔值；但约束永远不会在运行期计算。下一节将详细讨论约束表达式。

概念表达式的语法如下：

```
concept-name<argument-list>
```

概念表达式的计算结果为真或假。如果它的计算结果为真，那么表示使用给定的模板实参为概念建模。

## 12.6.2 约束表达式

计算结果为布尔值的常量表达式可以直接用作概念定义的约束。但它的结果必须精确计算为一个布尔值，并且没有任何类型转换。下面是一个示例：

```
template <typename T>
concept C = sizeof(T) == 4;
```

随着概念的引入，还引入了一种新的常量表达式类型，称为 `require` 表达式，接下来对此进行解释。

### require 表达式

`require` 表达式具有如下的语法：

```
requires (parameter-list) { requirements; }
```

`parameter-list` 为可选参数。每个 `requirement` 必须以分号作为结尾。

有 4 种类型的 `requirement`：简单 `requirement`、类型 `requirement`、复合 `requirement` 和嵌套 `requirement`，所有这些都将在接下来的部分中讨论。

#### 简单 `requirement`

一个简单的 `requirement` 是一个任意的表达式语句，而不是以 `requires` 开头。不允许使用变量声明、循环、条件语句等。并且这个表达式语句永远不会被计算；编译器也只是用于验证它是否已通过编译。

例如，下面的概念定义指定某种类型 `T` 必须是可递增的；也就是说，类型 `T` 必须支持后缀和前缀`++`运算符：

```
template <typename T>
concept Incrementable = requires(T x) { x++; ++x; };
```

`require` 表达式的参数列表用于引入位于 `require` 表达式主体中的命名变量。并且 `require` 表达式的主体不能有常规变量的声明。

#### 类型 `requirement`

类型 `requirement` 用于验证特定类型是否有效。例如，下面的概念要求特定类型 `T` 有 `value_type` 成员：

```
template <typename T>
concept C = requires { typename T::value_type; };
```

类型需求可以用来验证某个模板是否可以使用给定的类型进行实例化。下面是一个示例：

```
template <typename T>
concept C = requires { typename SomeTemplate<T>; };
```

## 复合 requirement

复合 requirement 可以用于验证某些东西不会抛出任何异常和/或验证某个方法是否返回某个类型。语法如下：

```
{ expression } noexcept -> type-constraint;
```

noexcept 和-> type-constraint 都是可选的。例如，下面的概念验证给定类型是否具有标记为 noexcept 的 swap() 方法：

```
template <typename T>
concept C = requires (T x, T y) {
    { x.swap(y) } noexcept;
};
```

type-constraint 可以是任何所谓的类型约束。类型约束(type constraint)只是一个概念的名称，它包含 0 个或多个模板类型参数。箭头左边表达式的类型自动作为类型约束的第一个模板类型参数进行传递。因此，类型约束的实参总是比对应概念定义的模板类型参数的数目少一个。例如，具有单一模板类型的概念定义的类型约束不需要任何模板实参；可以指定空方括号<>，或者省略它们。这听起来可能有点棘手，但通过一个示例就可以清楚地说明这一点。以下概念验证了给定类型具有一个名为 size() 的方法，该方法返回的类型可转换为 size\_t 的类型。

```
template <typename T>
concept C = requires (const T x) {
    { x.size() } -> convertible_to<size_t>;
};
```

std::convertible\_to<From, To> 是标准库在<concepts> 中预定义的概念，它有两个模板类型参数。箭头左边的表达式的类型自动作为第一个模板类型参数传递给 convertible\_to 的类型约束。因此，在这种情况下，只需要指定 To 模板类型实参(本例中为 size\_t)。

一个 require 表达式可以有多个参数，并且可以由一系列需求组成。例如，下面的概念要求类型 T 的实例是可比较的。

```
template <typename T>
concept Comparable = requires(const T a, const T b) {
    { a == b } -> convertible_to<bool>;
    { a < b } -> convertible_to<bool>;
    // ... similar for the other comparison operators ...
};
```

## 嵌套 requirement

require 表达式可以有嵌套的需求。例如，这里有一个概念，要求类型的大小为 4 个字节，并且该类型支持前缀和后缀的自增和自减操作。

```
template <typename T>
concept C = requires (T t) {
    requires sizeof(t) == 4;
    ++t; --t; t++; t--;
};
```

## 组合概念表达式

现有的概念表达式可以使用合取(&&)和析取(||)进行组合。例如，假设有一个递减的概念，类似

于递增的概念；下面的示例演示了一个概念，它要求一个类型既可以是自增，也可以是自减的。

```
template <typename T>
concept IncrementableAndDecrementable = Incrementable<T> && Decrementable<T>;
```

### 12.6.3 预定义的标准概念

标准库定义了一系列预定义的概念，分为若干类别。下面的列表给出了每个类别的一些示例概念，它们都定义在`<concepts>`和`std`名称空间中。

- 核心语言概念：`same_as`、`derived_from`、`convertible_to`、`integral`、`floating_point`、`copy_constructible`等
- 比较概念：`equality_comparable`、`totally_ordered`等
- 对象概念：`movable`、`copyable`等
- 可调用的概念：`invocable`、`predicate`等

此外，`<iterator>`定义了与迭代器相关的概念，如`random_access_iterator`、`forward_iterator`等。它还定义了算法需求，比如可合并、可排序、可置换等。C++20 范围程序库还提供了许多标准概念。第 17 章“理解迭代器和范围程序库”详细讨论了迭代器和范围程序库，而第 20 章更深入地讨论了标准库提供的算法。请参阅最喜欢的标准库参考资料，以获得可用标准概念的完整列表。

如果需要这些标准概念中的任何一个，那么可以直接使用它们，而不必实现自己的。例如，下面的概念要求类型 `T` 派生自类 `Foo`。

```
template <typename T>
concept IsDerivedFromFoo = derived_from<T, Foo>;
```

下面的概念要求 `T` 类型可以转换为 `bool` 类型：

```
template <typename T>
concept IsConvertibleToBool = convertible_to<T, bool>;
```

接下来的部分将提供更具体的示例。

当然，这些标准概念也可以组合成更具体的概念。例如，下面的概念要求类型 `T` 既是默认的也是可复制的。

```
template <typename T>
concept DefaultAndCopyConstructible =
default_initializable<T> && copy_constructible<T>;
```

#### 注意：

编写完整且正确的概念并不总是那么容易的。如果可能，尝试使用可用的标准概念或它们的组合来约束类型。

### 12.6.4 类型约束的 auto

类型约束可用于约束用自动类型推导定义的变量，在使用函数返回类型推导时约束其返回类型，约束在简化函数模板和泛型 lambda 表达式中的参数，等等。

例如，下面的代码编译得很好，因为类型被推导为 `int`，它模拟了 `Incrementable` 概念。

```
Incrementable auto value1 { 1 };
```

但是，下面的操作会导致编译错误。该类型被推导为 std::string(注意使用 s 标准的用户定义的字面量)，并且 string 不建模 Incrementable。

```
Incrementable auto value { "abc"s };
```

## 12.6.5 类型约束和函数模板

在函数模板中使用类型约束有几种不同的语法方式。第1种是简单地使用熟悉的 template<>语法，但不是使用 typename(或 class)，而是使用类型约束。示例如下：

```
template <convertible_to<bool> T>
void handle(const T& t);

template <Incrementable T>
void process(const T& t);
```

使用整型参数调用 process()可以按预期工作。例如，用 std::string 调用它会导致一个错误，编译器会抱怨不满足约束。例如，Clang 编译器会产生以下错误。乍一看，它可能仍然有点冗长，但实际上具有惊人的可读性。

```
<source>:17:2: error: no matching function for call to 'process'
    process(str);
    ^~~~~~
<source>:9:6: note: candidate template ignored: constraints not satisfied [with T =
std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>]
void process(const T& t)
^

<source>:8:11: note: because 'std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>' does not satisfy 'Incrementable'
template <Incrementable T>
^

<source>:6:42: note: because 'x++' would be invalid: cannot increment value of type
'std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>'
concept Incrementable = requires(T x) { x++; ++x; };
^
```

另一种语法是使用 require 子句，示例如下。

```
template <typename T> requires constant_expression
void process(const T& t);
```

constant\_expression 可以是任何产生布尔类型的常量表达式。例如，常量表达式可以是一个概念表达式：

```
template <typename T> requires Incrementable<T>
void process(const T& t);
```

或者一个预定义的标准概念：

```
template <typename T> requires convertible_to<T, bool>
void process(const T& t);
```

或者一个 require 表达式(注意 2 个 require 关键字)：

```
template <typename T> requires requires(T x) { x++; ++x; }
void process(const T& t);
```

或者任何产生布尔值的常量表达式：

```
template <typename T> requires (sizeof(T) == 4)
void process(const T& t);
```

或者是合取和析取的组合：

```
template <typename T> requires Incrementable<T> && Decrementable<T>
void process(const T& t);
```

或者类型萃取(详情请参阅第 26 章)：

```
template <typename T> requires is_arithmetic_v<T>
void process(const T& t);
```

也可以在函数头之后指定 require 子句，即所谓的后置 require 子句。

```
template <typename T>
void process(const T& t) requires Incrementable<T>;
```

使用类型约束的一种优雅的方式是将本章前面讨论过的简化函数模板的语法和类型约束结合起来，从而产生以下漂亮而紧凑的语法。请注意，即使没有模板 $\diamond$ 说明符，也不要被愚弄了：process() 仍然是一个函数模板。

```
void process(const Incrementable auto& t);
```

### 注意：

随着类型约束的引入，函数模板和类模板中不受约束的模板类型参数应该成为过去。对于每个模板类型，都不可避免地需要满足实现中与该类型直接相关的某些约束。因此，应该对其施加类型约束，以便编译器在编译时对它进行验证。

### 约束包含

可以使用不同的类型约束重载函数模板。编译器总是使用具有最具体约束的模板；更具体的约束包含/暗示较少的约束。下面是一个示例：

```
template <typename T> requires integral<T>
void process(const T& t) { cout << "integral<T>" << endl; }

template <typename T> requires (integral<T> && sizeof(T) == 4)
void process(const T& t) { cout << "integral<T> && sizeof(T) == 4" << endl; }
```

假设对 process() 有以下调用：

```
process(int { 1 });
process(short { 2 });
```

在一个典型的系统中，其中 int 有 32 位，short 有 16 位，输出如下所示：

```
integral<T> && sizeof(T) == 4
integral<T>
```

编译器首先通过规范化约束表达式来解析任何包容。在约束表达式的规范化过程中，所有概念表达式都会被递归地扩展它们的定义，直到结果是一个由常量布尔表达式的合取和析取组成的常量表达式。如果编译器可以证明一个规范化的约束表达式包含另一个约束表达式，那么它就包含另一个约束表达式。只考虑使用合取和析取来证明任何包容，而不是否定。

这种包容推断只在语法层面上完成，而不是语义层面。例如，`sizeof(T)>4` 在语义上比 `sizeof(T)>=4` 更具体，但在语法上前者并不会包含后者。

但是，需要注意的是，类型萃取（比如前面使用的 `std::is_arithmetic` 特征）在规范化期间不会被扩展。因此，如果有一个预定义的概念和一个类型萃取可用，那么应该使用这个概念而不是这个萃取。例如，使用 `std::integral` 概念来代替 `std::is_integral` 类型萃取。

## 12.6.6 类型约束和类模板

到目前为止，所有类型约束的示例都使用函数模板。然而，类型约束也可以与类模板一起使用，并使用类似的语法。举个例子，回顾一下本章前面提到的 `GameBoard` 类模板。下面是它的新定义，要求它的模板类型参数是 `GamePiece` 的派生类。

```
template <std::derived_from<GamePiece> T>
class GameBoard : public Grid<T>
{
public:
    explicit GameBoard(size_t width = Grid<T>::DefaultWidth,
                       size_t height = Grid<T>::DefaultHeight);
    void move(size_t xSrc, size_t ySrc, size_t xDest, size_t yDest);
};
```

相关方法的实现也需要更新。示例如下：

```
template <std::derived_from<GamePiece> T>
void GameBoard<T>::move(size_t xSrc, size_t ySrc, size_t xDest, size_t
yDest) { ... }
```

或者，也可以使用 `requires` 子句，示例如下。

```
template <typename T> requires std::derived_from<T, GamePiece>
class GameBoard : public Grid<T> { ... };
```

## 12.6.7 类型约束和类方法

也可以对类模板的特定方法添加额外的约束。例如，可以进一步限制 `GameBoard` 类模板的 `move()` 方法，要求类型 `T` 是可移动的。

```
template <std::derived_from<GamePiece> T>
class GameBoard : public Grid<T>
{
public:
    explicit GameBoard(size_t width = Grid<T>::DefaultWidth,
                       size_t height = Grid<T>::DefaultHeight);
    void move(size_t xSrc, size_t ySrc, size_t xDest, size_t yDest)
        requires std::movable<T>;
};
```

这样的 `require` 子句也需要在方法定义上重复：

```
template <std::derived_from<GamePiece> T>
void GameBoard<T>::move(size_t xSrc, size_t ySrc, size_t xDest, size_t yDest)
    requires std::movable<T>
{ ... }
```

请记住，基于本章前面讨论的选择性实例化，仍然可以使用非移动类型的 GameBoard 类模板，只要不调用它的 move() 方法。

### 12.6.8 类型约束和模板特化

如本章前面所述，可以为类模板编写特化，为函数模板编写重载，从而为特定类型编写不同的实现。也可以为满足特定约束的类型的集合编写特化。

这里回顾一下本章前面提到的 Find() 函数模板。刷新一下记忆：

```
template <typename T>
size_t Find(const T& value, const T* arr, size_t size)
{
    for (size_t i { 0 }; i < size; i++) {
        if (arr[i] == value) {
            return i; // Found it; return the index.
        }
    }
    return NOT_FOUND; // Failed to find it; return NOT_FOUND.
}
```

该实现使用`==`运算符比较值。通常不建议使用`==`比较浮点类型是否相等，而是使用所谓的 epsilon 检测。下面针对浮点类型的 Find() 特化使用了在 AreEqual() 辅助函数中实现的 epsilon 检测，而不是 operator`==`。

```
template <std::floating_point T>
size_t Find(const T& value, const T* arr, size_t size)
{
    for (size_t i { 0 }; i < size; i++) {
        if (AreEqual(arr[i], value)) {
            return i; // Found it; return the index.
        }
    }
    return NOT_FOUND; // Failed to find it; return NOT_FOUND.
}
```

AreEqual() 的定义如下，同样也使用类型约束。关于 epsilon 检测逻辑背后的数学的讨论超出了这本书的范围，并且对于本次讨论并不重要。

```
template <std::floating_point T>
bool AreEqual(T x, T y, int precision = 2)
{
    // Scale the machine epsilon to the magnitude of the given values and multiply
    // by the required precision.
    return fabs(x - y) <= numeric_limits<T>::epsilon() * fabs(x + y) * precision
        || fabs(x - y) < numeric_limits<T>::min(); // The result is subnormal.
}
```

## 12.7 本章小结

本章开始讨论如何使用模板进行泛型编程。内容包括：关于如何编写模板的语法，以及模板真正有用的示例。它解释了如何编写类模板、如何使用模板参数，以及如何将类的方法模板化。并进一步讨论了如何使用类模板特化来编写模板的特殊实现，其中模板参数被特定实参替换。

还学习了变量模板、函数模板和优雅的新 C++20 简化函数模板的语法。本章最后解释了 C++20 的另一个特性：概念，它允许你对模板参数进行约束。

第 26 章将继续讨论模板以及一些更高级的特性，如类模板部分特化、变参模板和元编程。

## 12.8 练习

通过完成以下习题，可以巩固本章讨论的知识点。所有练习的答案都可扫描封底二维码下载。然而，如果你困在一个练习中，在网站上查看答案之前，请先重新阅读本章的内容，试着自己找到答案。

**练习 12-1** 编写一个 KeyValuePair 类模板，其中包含两个模板类型参数：键和值。该类应该有两个私有数据成员来存储一个键和一个值。提供一个接收键和值的构造函数，并添加适当的 getter 和 setter。通过在 main() 函数中创建几个实例来测试新编写的类，并尝试类模板参数推断。

**练习 12-2** 练习 12-1 中的 KeyValuePair 类模板支持其键和值模板类型参数的所有类型的数据类型。例如，下面的示例使用 std::string 作为键和值的类型来实例化类模板：

```
KeyValuePair<std::string, std::string> kv { "John Doe", "New York" };
```

然而，使用 const char\* 作为类型将会导致 const char\* 类型的数据成员，但这不是期望的结果。

为 const char\* 键和值编写类模板特化，将给定字符串转换为 std::string。

**练习 12-3** 从练习 12-1 中获取解决方案，进行适当的更改，只允许整数类型作为键的类型，并且只允许浮点类型作为值的类型。

**练习 12-4** 编写一个函数模板 concat()，它带有两个模板类型参数和两个函数参数 t1 和 t2。该函数首先将 t1 和 t2 转换为一个字符串，然后返回这两个字符串的连接。对于这个练习，只关注数字的输入，比如支持 std::to\_string() 的整数和浮点数。创建并使用正确的概念，以确保函数模板的用户不会试图将其用于不支持的类型。尝试在不使用 template 关键字的情况下编写函数模板。

**练习 12-5** 练习 12-4 中的 concat() 函数模板只支持数值类型。在本练习中，请修改 12-4 中的解决方案，使其在 2 个参数都是字符串、1 个参数是数值类型，而另 1 个参数是字符串类型时都能工作。

**练习 12-6** 使用本章前面的原始 Find() 函数模板，并在类型 T 上添加一个适当的约束。



# 第 13 章

## C++ I/O 揭秘

### 本章内容

- 流的含义
- 如何使用流输入输出数据
- 标准库中提供的标准流
- 如何使用文件系统支持程序库

程序的基本任务是接收输入和生成输出。不能生成任何类型输出的程序不会太有用。所有语言都提供了某种 I/O 机制，这种机制既有可能内建在语言中，也有可能提供操作系统特定的 API。优秀的 I/O 系统应该兼具灵活性和易用性。灵活的 I/O 系统支持通过不同设备进行输入输出，例如文件和用户控制台，还支持读写不同类型的数据。I/O 很容易出错，因为来自用户的数据可能是不正确的，或者底层的文件系统或其他数据源有可能无法访问。因此，优秀的 I/O 系统还应当能处理错误情形。

如果已经熟悉了 C 语言，就肯定用过 `printf()` 和 `scanf()`。作为 I/O 机制，`printf()` 和 `scanf()` 确实很灵活。通过转义代码和变量占位符(类似于 `std::format()`)的格式说明符和占位符，这些函数可定制为读取特定格式的数据，或输出格式化代码允许的任何值，此类值仅局限于整数/字符值、浮点值和字符串。然而，`printf()` 和 `scanf()` 在优秀 I/O 系统的其他指标方面表现落后。这些函数不能很好地处理错误，处理自定义数据类型不够灵活，也不是类型安全的，在 C++ 这样的面向对象语言中，它们根本不是面向对象的！

C++ 通过一种称为流(stream)的机制提供了更精良的输入输出方法。流是一种灵活且面向对象的 I/O 方法。本章将会介绍如何将流用于数据的输入输出，并学习如何通过流机制从不同的来源读取数据，以及向不同的目的地写出数据，例如用户控制台、文件甚至字符串。本章将讲解最常用的 I/O 特性。

本章的最后一部分讨论了 C++ 标准库提供的文件系统支持库。这个程序库允许处理路径、目录和文件，它很好地补充了流提供的 I/O 机制。

### 13.1 使用流

需要花一些工夫才能习惯流的隐喻。初看上去，流似乎比传统的 C 风格 I/O(例如 `printf()`)要复

杂。事实上，流初看上去更复杂的原因是相比于 `printf()`，流背后的隐喻更深刻。不过不必担心，看过一些示例后，就再也不会想用旧式的 I/O 了。

### 13.1.1 流的含义

第 1 章将 `cout` 流比喻为数据的洗衣滑槽。把一些变量丢到流中，这些变量就会被写到用户屏幕上，即控制台。更一般地，所有的流都可以看成数据滑槽。流的方向不同，关联的来源和目的地也不同。例如，你已经熟悉的 `cout` 流是一个输出流，因此它的方向是“流出”。这个流将数据写入控制台，因此它关联的目的地是“控制台”。`cout` 中的 `c` 并不代表所期望的“控制台”，而是代表“字符”，因为它是基于字符的流。还有一个称为 `cin` 的标准流，它接收来自用户的输入。这个流的方向为“流入”，关联的来源为“控制台”。和 `cout` 一样，`cin` 中的 `c` 也是代表“字符”。`cout` 和 `cin` 都是 C++ 在 std 名称空间中预定义的流实例。表 13-1 简要描述了定义在 `<iostream>` 中的所有预定义的流。

缓冲流和未缓冲流之间的区别是缓冲流不会立即将数据发送到目的地。相反，它会缓冲(即，收集)传入的数据，然后以块的形式分块发送。另一方面，非缓冲流会立即将数据发送到目的地。缓冲通常是为了提高性能，因为某些目的地(如文件)在一次写入更大的块时性能更好。请注意，总是可以通过使用 `flush()` 方法刷新缓冲流的缓冲区，强制缓冲流将当前缓冲的所有数据发送到目的地。缓冲和刷新将在本章后面更详细地讨论。

表 13-1 预定义的流

流	说明
<code>cin</code>	输入流，从“输入控制台”读取数据
<code>cout</code>	缓冲的输出流，向“输出控制台”写入数据
<code>cerr</code>	非缓冲的输出流，向“错误控制台”写入数据，“错误控制台”通常等同于“输出控制台”
<code>clog</code>	<code>cerr</code> 的缓冲版本

缓冲的流和非缓冲的流的区别在于，前者不是立即将数据发送到目的地，而是缓冲输入的数据，然后以块方式发送；而非缓冲的流则立即将数据发送到目的地。缓冲的目的通常是提高性能，对于某些目的地(如文件)而言，一次性写入较大的块时速度更快。注意，始终可使用 `flush()` 方法刷新缓冲区，强制要求缓冲的流将其当前所有的缓冲数据发送到目的地。

这些流还存在宽字符版本，以字符 `w` 开头：`wcin`、`wcout`、`wcerr` 和 `wclog`。宽字符可以用于字符数多于英语的语言，例如中文。第 21 章“字符串的本地化与正则表达式”将讨论宽字符。

#### 注意：

所有输入流都有一个关联的来源，所有输出流都有一个关联的目的地。

有关流的另一个要点是：流不仅包含普通数据，还包含称为当前位置(current position)的特殊数据。当前位置指的是流将要进行下一次读写操作的位置。

#### 注意：

图形用户界面应用程序通常没有控制台。换言之，如果向 `cout` 写入一些数据，用户无法看到。如果编写的是程序库，那么绝对不要假定存在 `cout`、`cin`、`cerr` 或 `clog`，因为不可能知道库会应用到控制台应用程序还是 GUI 应用程序。

### 13.1.2 流的来源和目的地

流这个概念可应用于任何接收数据或生成数据的对象。因此可编写基于流的网络类，还可编写MIDI设备的流式访问类。在C++中，流可使用3个公共的来源和目的地：控制台、文件和字符串。

前面的章节已经介绍了很多用户(或控制台)流的例子。控制台输入流允许程序在运行时从用户那里获得输入，使程序具有交互性。控制台输出流向用户提供反馈和输出结果。

文件流，顾名思义，它从文件系统中读取数据并向文件系统写入数据。文件输入流适用于读取配置数据、读取保存的文件，也适用于批处理基于文件的数据等任务。文件输出流适用于保存状态数据和提供输出等任务。如果熟悉C风格的输入和输出，那么会发现文件流包含C语言输出函数fprintf()、fwrite()和fputs()的功能，以及输入函数fscanf()、fread()和fgets()的功能。这些C风格的函数并不推荐在C++中使用，因此本书不作进一步的讨论。

字符串流是将流隐喻应用于字符串类型的例子。使用字符串流时，可像处理其他任何流一样处理字符数据。就字符串流的大部分功能而言，只不过是为string类的很多方法能够完成的功能提供了便利的语法。然而，使用流式语法为优化提供了机会，而且比直接使用string类方便得多。字符串流包含sprintf()、sprintf\_s()和sscanf()的功能，以及很多C语言字符串格式化函数的功能，这些都不是本书讨论的范畴。

本节接下来主要讲解控制台流(cin和cout)。本章后面还会列举文件流和字符串流的例子。其他类型的流，例如打印输出和网络I/O等往往与平台相关，因此这些流也不是本书讨论的范畴。

### 13.1.3 流式输出

第1章介绍了流式输出，在本书中，几乎每一章都使用了流式输出。本节首先简单回顾一些基本概念，然后介绍一些更高级的内容。

#### 1. 输出的基本概念

输出流定义在<iostream>头文件中。大部分程序员都会在程序中包含<iostream>头文件，这个头文件又包含输入流和输出流的头文件。<iostream>头文件还声明了所有预定义的流实例：cout、cin、cerr、clog以及对应的宽字符版本。

使用输出流的最简单方法是使用<<运算符。通过<<可输出C++的基本类型，如int、指针、double和字符。此外，C++的string类也兼容<<，C风格的字符串也能正确输出。下面列举一些使用<<的示例：

```
int i { 7 };
cout << i << endl;

char ch { 'a' };
cout << ch << endl;

string myString { "Hello World." };
cout << myString << endl;
```

输出如下所示：

```
7
a
Hello World.
```

`cout` 流是写入控制台的内建流，控制台也称为标准输出(standard output)。可将`<<`的使用串联起来，从而输出多个数据段。这是由于`<<`运算符返回一个流的引用，因此可以立即对同一个流再次应用`<<`运算符。示例如下：

```
int j { 11 };
cout << "The value of j is " << j << "!" << endl;
```

输出如下所示：

```
The value of j is 11!
```

C++流可正确解析C风格的转义字符，例如包含\n的字符串，也可使用`std::endl`开始一个新行。`\n`和`endl`的区别是：`\n`仅开始一个新行，而`endl`还会刷新缓冲区。使用`endl`时要小心，因为过多的缓冲区刷新会降低性能。接下来的示例使用`endl`输出，通过一行代码输出多行文本：

```
cout << "Line 1" << endl << "Line 2" << endl << "Line 3" << endl;
```

输出如下所示：

```
Line 1
Line 2
Line 3
```

### 警告：

`endl`会刷新目标缓冲区，因此在性能关键的代码中要明智地使用它，比如紧密的循环中。

## 2. 输出流的方法

毫无疑问，`<<`运算符是输出流最有用的部分。然而，你还需要了解一些额外功能。如果看一下`<ostream>`头文件，就会发现重载`<<`运算符定义的很多代码行(支持输出各种不同的数据类型)，还可看到一些有用的公有方法。

### put()和 write()

`put()`和`write()`是原始的输出方法。这两个方法接收的不是定义了输出行为的对象或变量，`put()`接收单个字符，`write()`接收一个字符数组。传给这些方法的数据按照原本的形式输出，没有做任何特殊的格式化和处理操作。例如，下面的代码段接收一个C风格的字符串，并将它输出到控制台，这个函数没有使用`<<`运算符。

```
const char* test { "hello there\n" };
cout.write(test, strlen(test));
```

下面的代码段通过`put()`方法，将C风格字符串的给定索引输出到控制台：

```
cout.put('a');
```

### flush()

当向输出流写入数据时，流不一定会将数据立即写入目的地。大部分输出流都会进行缓冲，也就是积累数据，而不是立即将得到的数据写出去。缓冲的目的通常是提高性能，对于某些目的地(如文件)而言，与逐字符写入相比，一次性写入较大的块时速度更快。在以下任意一种条件下，流将刷新(或写出)积累的数据：

- 遇到`endl`操作算子时。
- 流离开作用域被析构时。

- 流缓冲区满时。
- 显式地要求流刷新缓冲区时。
- 要求从对应的输入流输入数据时(即，要求从 cin 输入时，cout 会刷新)。在有关文件流的章节中，将学习如何建立这种连接。

显式要求流刷新缓冲区的方法是调用流的 flush()方法，如下所示：

```
cout << "abc";
cout.flush(); // abc is written to the console.
cout << "def";
cout << endl; // def is written to the console.
```

#### 注意：

不是所有的输出流都会缓存。例如，cerr 流就不会缓存其输出。

### 3. 处理输出错误

输出错误可能会在多种情况下出现。例如，你有可能试图打开一个不存在的文件；有可能因为磁盘错误导致写入操作失败，例如磁盘已满。到目前为止，前面使用流的代码都没有考虑这些可能性，主要是为了使代码简洁。然而，处理任何可能发生的错误是非常重要的。

当一个流处于正常的可用状态时，称这个流是“好的”。调用流的 good()方法可以判断这个流当前是否处于正常状态。

```
if (cout.good()) {
    cout << "All good" << endl;
}
```

通过 good()方法可方便地获得流的基本验证信息，但不能提供流不可用的原因。还有一个 bad()方法提供了稍多信息。如果 bad()方法返回 true，那么意味着发生了致命错误(相对于非致命错误，例如到达文件结尾 eof())。另一个方法 fail()在最近一次操作失败时返回 true，但没有说明下一次操作是否也会失败。例如，在对输出流调用 flush()后，可以通过调用 fail()来确保刷新成功。

```
cout.flush();
if (cout.fail()) {
    cerr << "Unable to flush to standard out" << endl;
}
```

流具有可转换为 bool 类型的转换运算符。转换运算符与调用!fail()时返回的结果相同。因此，可将前面的代码段重写为：

```
cout.flush();
if (!cout) {
    cerr << "Unable to flush to standard out" << endl;
}
```

有一点需要指出：遇到文件结束标记时，good()和 fail()都会返回 false。关系如下：good() == (!fail() && !eof())。

还可要求流在发生故障时抛出异常。然后编写一个 catch 处理程序来捕捉 ios\_base::failure 异常，然后对这个异常调用 what()方法，来获得错误的描述信息；调用 code()方法获得错误代码。但是，是否能获得有用信息取决于所使用的标准库实现。

```
cout.exceptions(ios::failbit | ios::badbit | ios::eofbit);
```

```

try {
    cout << "Hello World." << endl;
} catch (const ios_base::failure& ex) {
    cerr << "Caught exception: " << ex.what()
    << ", error code = " << ex.code() << endl;
}

```

可以通过 `clear()` 方法重置流的错误状态:

```
cout.clear();
```

控制台输出流的错误检查不如文件输入输出流的错误检查频繁。这里讨论的方法也适用于其他类型的流，后面讨论每一种类型时都会回顾这些方法。

#### 4. 输出操作算子

流的一项独特特性是，放入数据滑槽的内容并非仅限于数据。C++流还能识别操作算子(**manipulator**)，操作算子是能修改流行为的对象，而不是(或额外提供)流能够操作的数据。

`endl` 就是一个操作算子。`endl` 操作算子封装了数据和行为。它要求流输出一个行结束序列，并且刷新缓冲区。下面列出了其他有用的操作算子，大部分定义在`<iostream>`和`<iomanip>`标准头文件中。列表后的例子展示了如何使用这些操作算子：

- `boolalpha` 和 `noboolalpha`: 要求流将布尔值输出为 `true` 和 `false`(`boolalpha`)或 `1` 和 `0`(`noboolalpha`)。默认行为是 `noboolalpha`。
- `hex`、`oct` 和 `dec`: 分别以十六进制、八进制和十进制输出数字。
- `setprecision`: 设置输出小数时的小数位数。这是一个参数化的操作算子(也就是说，这个操作算子接收一个参数)。
- `setw`: 设置输出数值数据的字段宽度。这是一个参数化的操作算子。
- `fill`: 将一个字符设置为流的新的填充字符。填充字符根据 `setw` 设置的宽度填充输出。这是一个参数化的操作算子。
- `showpoint` 和 `noshowpoint`: 对于不带小数部分的浮点数，强制流总是显示或不显示小数点。
- `put_money`: 一个参数化的操作算子，向流写入一个格式化的货币值。
- `put_time`: 一个参数化的操作算子，向流写入一个格式化的时间值。
- `quoted`: 一个参数化的操作算子，把给定的字符串封装在引号中，并转义嵌入的引号。

上述操作算子对后续输出到流中的内容有效，直到重置操作算子为止，但 `setw` 仅对下一个输出有效。下面的示例通过这些操作算子自定义输出：

```

// Boolean values
bool myBool { true };
cout << "This is the default: " << myBool << endl;
cout << "This should be true: " << boolalpha << myBool << endl;
cout << "This should be 1: " << noboolalpha << myBool << endl;

// Simulate "%6d" with streams
int i { 123 };
printf("This should be ' 123': %6d\n", i);
cout << "This should be ' 123': " << setw(6) << i << endl;

// Simulate "%06d" with streams
printf("This should be '000123': %06d\n", i);
cout << "This should be '000123': " << setfill('0') << setw(6) << i << endl;

```

```

// Fill with *
cout << "This should be '***123': " << setfill('*') << setw(6) << i << endl;
// Reset fill character
cout << setfill(' ');

// Floating point values
double dbl { 1.452 };
double dbl2 {.5};
cout << "This should be ' 5': " << setw(2) << noshowpoint << dbl2 << endl;
cout << "This should be @@1.452: " << setw(7) << setfill('@') << dbl << endl;
// Reset fill character
cout << setfill(' ');

// Instructs cout to start formatting numbers according to your location.
// Chapter 19 explains the details of the imbue call and the locale object.
cout.imbue(locale { "" });

// Format numbers according to your location
cout << "This is 1234567 formatted according to your location: " << 1234567
    << endl;

// Monetary value. What exactly a monetary value means depends on your
// location. For example, in the USA, a monetary value of 120000 means 120000
// dollar cents, which is 1200.00 dollars.
cout << "This should be a monetary value of 120000, "
    << "formatted according to your location: "
    << put_money("120000") << endl;

// Date and time
time_t t_t { time(nullptr) }; // Get current system time
tm* t { localtime(&t_t) }; // Convert to local time
cout << "This should be the current date and time "
    << "formatted according to your location: "
    << put_time(t, "%c") << endl;

// Quoted string
cout << "This should be: \"Quoted string with \\"embedded quotes\\\".\\"": "
    << quoted("Quoted string with \"embedded quotes\".") << endl;

```

### 注意：

这个示例在 `localtime()` 调用中可能会输出与安全相关的错误或警告。在 Microsoft Visual Studio 中，可以使用安全版本 `localtime_s()`；在 Linux 中，可以使用 `localtime_r()`。

如果不关心操作算子的概念，通常也能应付过去。流通过 `precision()` 这类方法提供了大部分相同的功能。以如下代码为例：

```
cout << "This should be '1.2346': " << setprecision(5) << 1.23456789 << endl;
```

这行代码可转换为方法调用。该方法的优点是，它们返回前面的值以便恢复：

```
cout.precision(5);
cout << "This should be '1.2346': " << 1.23456789 << endl;
```

流方法和操作算子的详细信息请参阅 Standard Library Reference。

### 13.1.4 流式输入

输入流为结构化数据和非结构化数据的读入提供了简单方法。本节以 `cin` 为例讨论输入技术，`cin` 即控制台输入流。

#### 1. 输入的基本概念

通过输入流，可采用两种简单方法来读取数据。第一种方法类似于 `<<` 运算符，`<<` 向输出流输出数据。读入数据对应的运算符是 `>>`。通过 `>>` 从输入流读入数据时，代码提供的变量保存接收的值。例如，以下程序从用户那里读入一个单词，并保存在一个字符串中。然后将这个字符串输出到控制台：

```
string userInput;
cin >> userInput;
cout << "User input was " << userInput << endl;
```

默认情况下，`>>` 运算符根据空白字符对输入值进行标志化。例如，如果用户运行以上程序，并键入 "hello there" 作为输入，那么只有第一个空白字符(在这个例子中为空格符)之前的字符才会存储在 `userInput` 变量中。输出如下所示：

```
User input was hello
```

在输入中包含空白字符的一种方法是使用 `get()`，本章后面将会讨论这个方法。

#### 注意：

C++ 中的空白字符包括空格(' ')、换行('\n')、换行('\r')、水平制表符('\t')和垂直制表符('\v')。

`>>` 运算符可用于不同的变量类型，就像 `<<` 运算符一样。例如，要读取一个整数，代码仅在变量类型上区别：

```
int userInput;
cin >> userInput;
cout << "User input was " << userInput << endl;
```

通过输入流可以读入多个值，并且可根据需要混合和匹配类型。例如，下面这个函数摘自一个餐馆预订系统，它要求用户输入姓氏和聚会就餐的人数。

```
void getReservationData()
{
    string guestName;
    int partySize;
    cout << "Name and number of guests: ";
    cin >> guestName >> partySize;
    cout << "Thank you, " << guestName << "." << endl;
    if (partySize > 10) {
        cout << "An extra gratuity will apply." << endl;
    }
}
```

注意，`>>` 运算符会根据空白字符进行符号化，因此 `getReservationData()` 函数不允许输入带有空白字符的姓名。一种解决方法是使用本章后面讲解的 `unget()` 方法。注意，尽管这里首次使用 `cout` 时，没有通过 `endl` 或 `flush()` 显式地刷新缓冲区，但仍可将文本写入控制台，因为使用 `cin` 会立即刷新 `cout` 缓冲区；`cin` 和 `cout` 通过这种方式连接在一起。

**注意：**

如果分不清<<和>>的作用，只要联想箭头的方向指向它们的目的地即可。在输出流中，<<指向流本身，因为数据被发送至流。在输入流中，>>指向变量，因为数据被保存。

## 2. 处理输入错误

输入流提供了一些方法用于检测异常情形。大部分和输入流有关的错误条件都发生在无数据可读时。例如，可能到达流尾(称为文件末尾 `eof`，即使不是文件流)。查询输入流状态的最常见方法是在条件语句中访问输入流。例如，只要 `cin` 保持在“良好”状态，下面的循环就继续进行。这种模式利用了这样一个事实：只有当流没有处于任何错误状态时，在条件上下文中计算输入流时才会返回 `true`。遇到错误会导致流的计算结果为 `false`。实现这种行为所需的转换操作的底层细节将在第 15 章“C++ 运算符重载”中进行解释。

```
while (cin) { ... }
```

同时可以输入数据：

```
while (cin >> ch) { ... }
```

还可在输入流上调用 `good()`、`bad()` 和 `fail()` 方法，就像输出流那样。还有一个 `eof()` 方法，如果流到达尾部，就返回 `true`。与输出流类似，遇到文件结束标记时，`good()` 和 `fail()` 都会返回 `false`。关系如下：`good() == (!fail() && !eof())`。

还应该养成读取数据后就检查流状态的习惯，这样可从异常输入中恢复。

下面的程序展示了从流中读取数据并处理错误的常用模式。这个程序从标准输入中读取数字，到达文件末尾时显示这些数字的总和。注意在命令行环境中，需要用户键入一个特殊的字符来表示文件结束。在 UNIX 和 Linux 中，这个特殊字符是 Control+D；在 Windows 中，这个特殊字符是 Control+Z。具体的字符与操作系统相关，因此需要了解操作系统要求的字符。

```
cout << "Enter numbers on separate lines to add.\n"
      << "Use Control+D to follow by Enter to finish (Control+Z in Windows).\n";
int sum = 0;

if (!cin.good()) {
    cerr << "Standard input is in a bad state!" << endl;
    return 1;
}

while (!cin.fail()) {
    int number;
    cin >> number;
    if (cin.good()) {
        sum += number;
    } else if (cin.eof()) {
        break; // Reached end of file
    } else if (cin.fail()) {
        // Failure!
        cin.clear(); // Clear the failure state.
        string badToken;
        cin >> badToken; // Consume the bad input.
        cerr << "WARNING: Bad input encountered: " << badToken << endl;
    }
}
cout << "The sum is " << sum << endl;
```

下面是这个程序的一些示例输出。当按下 Control+Z 时，输出中的^Z 字符出现：

```
Enter numbers on separate lines to add.
Use Control+D followed by Enter to finish (Control+Z in Windows).
1
2
test
WARNING: Bad input encountered: test
3
^Z
The sum is 6
```

### 3. 输入方法

与输出流一样，输入流也提供了一些方法，它们可获得相比普通>>运算符更底层的访问功能。

#### get()

get()方法允许从流中读入原始输入数据。get()的最简单版本返回流中的下一个字符，其他版本一次读入多个字符。get()常用于避免>>运算符的自动标志化。例如，下面这个函数从输入流中读入一个由多个单词构成的名字，一直读到流尾。

```
string readName(istream& stream)
{
    string name;
    while (stream) { // Or: while (!stream.fail()) {
        int next = stream.get();
        if (!stream || next == std::char_traits<char>::eof())
            break;
        name += static_cast<char>(next); // Append character.
    }
    return name;
}
```

在这个 readName() 函数中，有一些有趣的发现：

- 这个函数的参数是一个对 istream 的非 const 引用，而不是一个 const 引用。从流中读入数据的方法会改变实际的流(主要改变当前位置)，因为它们都不是 const 方法。因此，不能对 const 引用调用这些方法。
- get() 的返回值保存在 int 变量而不是 char 变量中，由于 get() 会返回一些特殊的非字符值，例如 std::char\_traits<char>::eof()(文件结束)，因此使用 int。

readName() 有一点奇怪，因为可采用两种方式跳出循环。一种方式是流进入“不好的”状态，另一种方式是到达流尾。从流中读入数据的更常用方法是使用另一个版本的 get()，这个版本接收一个字符的引用，并返回一个流的引用。只有当输入流没有处于任何错误状态时，在条件环境中对一个输入流求值时才会得到 true。同一个函数的下面这个版本稍微简洁一些：

```
string readName(istream& stream)
{
    string name;
    char next;
    while (stream.get(next)) {
        name += next;
    }
    return name;
}
```

### unget()

对于大多数场合来说，理解输入流的正确方式是将输入流理解为单方向的滑槽。数据被丢入滑槽，然后进入变量。`unget()`方法打破了这个模型，允许将数据塞回滑槽。

调用`unget()`会导致流回退一个位置，将读入的前一个字符放回流中。调用`fail()`方法可查看`unget()`是否成功。例如，如果当前位置就是流的起始位置，那么`unget()`会失败。

本章前面出现的`getReservationData()`函数不允许输入带有空白字符的名字。下面的代码使用了`unget()`，允许名字中出现空白字符。将这段代码逐字符读入，并检查字符是否为数字。如果字符不是数字，就将字符添加到`guestName`。如果字符是数字，就通过`unget()`将这个字符放回到流中，循环停止，然后通过`>>`运算符输入一个整数`partySize`。`noskipws`输入操作算子告知流不要跳过空白字符，就像读取其他任何字符一样读取空白字符。

```
void getReservationData()
{
    string guestName;
    int partySize { 0 };
    // Read characters until we find a digit
    char ch;
    cin >> noskipws;
    while (cin >> ch) {
        if (isdigit(ch)) {
            cin.unget();
            if (cin.fail())
                cout << "unget() failed" << endl;
            break;
        }
        guestName += ch;
    }
    // Read partysize, if the stream is not in error state
    if (cin)
        cin >> partySize;

    if (!cin) {
        cerr << "Error getting party size." << endl;
        return;
    }

    cout << format("Thank you '{}', party of {}", guestName, partySize) << endl;
    if (partySize > 10) {
        cout << "An extra gratuity will apply." << endl;
    }
}
```

### putback()

`putback()`和`unget()`一样，允许在输入流中反向移动一个字符。区别在于`putback()`方法将放回流中的字符接收为参数，示例如下：

```
char c;
cin >> c;
cout << format("Retrieved {} before putback('e').", c) << endl;

cin.putback('e'); // 'e' will be the next character read off the stream.
```

```
cin >> c;
cout << format("Retrieved {} after putback('e').", c) << endl;
```

结果输入如下：

```
w
Retrieved w before putback('e').
Retrieved e after putback('e').
```

### peek()

通过 peek() 方法可预览调用 get() 后返回的下一个值。再次以滑槽为例，可想象为查看一下滑槽，但是不把值取出来。

peek() 非常适合于在读取前需要预先查看一个值的场合。例如下面的代码实现了 getReservationData() 函数，这个函数允许名字中出现空白字符，但使用的是 peek() 而不是 unget()。

```
void getReservationData()
{
    string guestName;
    int partySize { 0 };
    // Read characters until we find a digit
    char ch;
    cin >> noskipws;
    while (true) {
        // 'peek' at next character
        ch { static_cast<char>(cin.peek()) };
        if (!cin)
            break;
        if (isdigit(ch)) {
            // next character will be a digit, so stop the loop
            break;
        }
        // next character will be a non-digit, so read it
        cin >> ch;
        if (!cin)
            break;
        guestName += ch;
    }
    // Read partysize, if the stream is not in error state
    if (cin)
        cin >> partySize;
    if (!cin) {
        cerr << "Error getting party size." << endl;
        return;
    }

    cout << format("Thank you '{}', party of {}", guestName, partySize) << endl;
    if (partySize > 10) {
        cout << "An extra gratuity will apply." << endl;
    }
}
```

### getline()

从输入流中获得一行数据是一种非常常见的需求，有一个方法能完成这个任务。getline() 方法用一行数据填充字符缓冲区，数据量最多至指定大小。指定的大小中包括 \0 字符。因此，下面的代码最

多从 cin 中读取 kBufferSize-1 个字符，或者读到行尾为止。

```
char buffer[kBufferSize] { 0 };
cin.getline(buffer, kBufferSize);
```

调用 `getline()` 时，从输入流中读取一行，读到行尾为止。不过，行尾字符不会出现在字符串中。注意，行尾序列和平台相关。例如，行尾序列可以是 `\r\n`、`\n` 或 `\n\r`。

有个版本的 `get()` 函数执行的操作和 `getline()` 一样，区别在于 `get()` 把换行序列留在输入流中。

还有一个用于 C++ 字符串的 `std::getline()` 函数。这个函数定义在 `<string>` 头文件和 std 名称空间中。它接收一个流引用、一个字符串引用。使用这个版本的 `getline()` 函数的优点是不需要指定缓冲区的大小。

```
string myString;
getline(cin, myString);
```

`getline()` 方法和 `std::getline()` 函数都接收一个可选的分隔符作为最后一个参数。默认分隔符是 `\n`。通过更改此分隔符，可以使用这些函数读取多行文本，直到达到给定的分隔符。例如，下面的代码读取多行文本，直到读取一个 @ 字符。

```
cout << "Enter multiple lines of text. "
     << "Use an @ character to signal the end of the text.\n> ";
string myString;
getline(cin, myString, '@');
cout << format("Read text: \"{}\"", myString) << endl;
```

下面是可能的输出结果：

```
Enter multiple lines of text. Use an @ character to signal the end of the text.
> This is some
text on multiple
lines.@
Read text: "This is some
text on multiple
lines."
```

#### 4. 输入操作算子

下面列出了内建的输入操作算子，它们可发送到输入流中，以自定义数据读入的方式。

- `boolalpha` 和 `noboolalpha`: 如果使用了 `boolalpha`，字符串 `false` 会被解析为布尔值 `false`；其他任何字符串都会被解析为布尔值 `true`。如果设置了 `noboolalpha`，`0` 会被解析为 `false`，其他任何值都被解析为 `true`。默认为 `noboolalpha`。
- `dec`、`hex` 和 `oct`: 分别读取以 10 为基数的十进制、十六进制或八进制记数法的数字。例如，以 10 为基数的十进制数 207 用十六进制表示为 `cf`，用八进制表示是 317。
- `skipws` 和 `noskipws`: 告诉输入流在标记化时跳过空白字符，或者读入空白字符作为标记。默认为 `skipws`。
- `ws`: 一个简便的操作算子，表示跳过流中当前位置的一串空白字符。
- `get_money`: 一个参数化的操作算子，从流中读入一个格式化的货币值。
- `get_time`: 一个参数化的操作算子，从流中读入一个格式化的时间值。
- `quoted`: 一个参数化的操作算子，读取封装在引号中的字符串，并转义嵌入的引号。

输入支持本地化。例如，下面的代码将为 `cin` 启用本地化。第 19 章将讨论本地化：

```
cin.imbue(locale { "" });
int i;
cin >> i;
```

如果系统被本地化为 U.S. English，那么输入 1,000 会被解析为 1000，输入 1.000 会被解析为 1。如果系统被本地化为 Dutch Belgium，那么输入 1.000 会被解析为 1000，而输入 1,000 会被解析为 1。在这两种情形中，如果输入不带数字分隔符的 1000，会得到值 1000。

### 13.1.5 对象的输入输出

即使不是基本类型，也可以通过<<运算符输出 C++字符串。在 C++中，对象可描述其输入输出方式。为此，需要重载<<和>>运算符，以理解新的类型或类。

为什么要重载这些运算符？如果熟悉 C 语言中的 printf()函数，就会知道 printf()在这方面并不灵活。尽管 printf()知道多种数据类型，但是无法让其知道更多的知识。例如，考虑下面这个简单的类：

```
class Muffin
{
public:
    virtual ~Muffin() = default;

    const_string& getDescription() const { return m_description; }
    void setDescription(string_description)
    {
        m_description = description;
    }

    int getSize() const { return m_size; }
    void setSize(int size) { m_size = size; }

    bool hasChocolateChips() const { return m_hasChocolateChips; }
    void setHasChocolateChips(bool hasChips)
    {
        m_hasChocolateChips = hasChips;
    }

private:
    string m_description;
    int m_size { 0 };
    bool m_hasChocolateChips { false };
};
```

为了使用 printf()输出 Muffin 类的对象，最好将其指定为实参，可能需要使用%om 作为占位符。

```
printf("Muffin: %m\n", myMuffin); // BUG! printf doesn't understand Muffin.
```

但是，printf()函数完全不了解 Muffin 类型，因此无法输出 Muffin 类型的对象。更糟糕的是，由于 printf()函数的声明方式，这样的代码会导致运行时错误而不是编译时错误(不过优秀的编译器会给出警告消息)。

如果想要使用 printf()，那么最好在 Muffin 类中添加新的 output()方法：

```
class Muffin
{
public:
    void output() const
{
```

```

        printf("%s, Size is %d, %s\n", getDescription().data(), getSize(),
               (hasChocolateChips() ? "has chips" : "no chips"));
    }
    // Omitted for brevity
};

```

不过，使用这种机制显得非常笨拙。如果要在另一行文本的中间输出 Muffin，那么需要将这一行分解为两个调用，并在两个调用之间插入一个 Muffin::output() 调用，如下所示。

```

printf("The muffin is ");
myMuffin.output();
printf(" -- yummy!\n");

```

通过重载<<运算符，使得输出 Muffin 就像输出字符串一样简单——只要将其作为<<的实参即可。第 15 章将讲解运算符<<和>>的重载。

### 13.1.6 自定义的操作算子

标准库提供了许多内置的流操作算子，但如果需要，可以编写自定义的操作算子。这涉及使用 ios\_base 公开的功能，例如 xalloc()、iword()、pword() 和 register\_callback()。由于很少需要自定义操作算子，因此，本文不再进一步讨论这个主题。如果感兴趣，请参阅最喜欢的标准库参考资料。

## 13.2 字符串流

可通过字符串流将流语义用于字符串。通过这种方式，可得到一个内存中的流(in memory stream)来表示文本数据。例如，在 GUI 应用程序中，可能需要用流构建文本数据，但不是将文本输出到控制台或文件中，而是把结果显示在 GUI 元素中，例如消息框和编辑框。另一个例子是，要将一个字符串流作为参数传给不同函数，同时维护当前的读取位置，这样每个函数都可以处理流的下一部分。字符串流也非常适合于解析文本，因为流内建了标记化的功能。

std::ostringstream 类用于将数据写入字符串，std::istringstream 类用于从字符串读出数据。ostringstream 中的 o 代表输出，而 istringstream 中的 i 代表输入。这两个类都定义在<sstream>头文件中。由于 ostringstream 和 istringstream 把同样的行为分别继承为 ostream 和 istream，因此这两个类的使用也非常类似。

下面的程序从用户那里请求单词，然后输出到一个 ostringstream 中，通过制表符将单词分开。在程序的最后，整个流通过 str() 方法转换为字符串对象，并写入控制台。输入标记“done”，可停止标记的输入，按下 Control+D(UNIX) 或 Control+Z(Windows) 可关闭输入流。

```

cout << "Enter tokens. "
     << "Control+D (Unix) or Control+Z (Windows) followed by Enter to end."
     << endl;
ostringstream outStream;
while (cin) {
    string nextToken;
    cout << "Next token: ";
    cin >> nextToken;
    if (!cin || nextToken == "done")
        break;
    outStream << nextToken << "\t";
}
cout << "The end result is: " << outStream.str();

```

从字符串流中读入数据非常类似。下面的函数创建一个 `Muffin` 对象，并填充字符串输入流中的数据(参见此前的例子)。流数据的格式固定，因此这个函数可轻松地将数据值转换为对 `Muffin` 类的设置方法的调用。

```
Muffin createMuffin(istringstream& stream)
{
    Muffin muffin;
    // Assume data is properly formatted:
    // Description size chips

    string description;
    int size;
    bool hasChips;

    // Read all three values. Note that chips is represented
    // by the strings "true" and "false"
    stream >> description >> size >> boolalpha >> hasChips;
    if (stream) { // Reading was successful.
        muffin.setSize(size);
        muffin.setDescription(description);
        muffin.setHasChocolateChips(hasChips);
    }
    return muffin;
}
```

#### 注意：

将对象转换为“扁平”类型(例如字符串类型)的过程通常称为编组(marshall)。将对象保存至磁盘或通过网络发送时，编组操作非常有用。

相对于标准 C++ 字符串，字符串流的主要优点是除了数据之外，这个对象还知道从哪里进行下一次读或写操作，这个位置也称为当前位置。

与字符串相比，字符串流的另一个优势是支持操作算子和本地化，格式化功能更加强大。

最后，如果需要通过连接几个较小的字符串来构建一个字符串，那么与直接连接字符串对象相比，使用字符串流会更高效。

## 13.3 文件流

文件本身非常符合流的抽象，因为读写文件时，除数据外，还涉及读写的位置。在 C++ 中，`std::ofstream` 和 `std::ifstream` 类提供了文件的输入输出功能。这两个类在 `<fstream>` 头文件中定义。

在处理文件系统时，错误情形的检测和处理非常重要。比如，当前处理的文件可能在一个刚下线的网络存储中，或者可能写入已满磁盘上的一个文件，以及也许试图打开一个用户没有访问权限的文件。可以通过前面描述的标准错误处理机制检测错误情形。

输出文件流和其他输出流的唯一主要区别在于：文件流的构造函数可以接收文件名以及打开文件的模式作为参数。默认模式是写文件(`ios_base::out`)，这种模式从文件开头写文件，改写任何已有的数据。给文件流构造函数的第 2 个参数指定常量 `ios_base::app`，还可按追加模式打开输出文件流。表 13-2 列出了可供使用的不同常量。

表 13-2 可供使用的常量

常量	说明
ios_base::app	打开文件，在每一次写操作之前，移到文件末尾
ios_base::ate	打开文件，打开之后立即移到文件末尾
ios_base::binary	以二进制模式执行输入输出操作(相对于文本模式)
ios_base::in	打开文件，从开头开始读取
ios_base::out	打开文件，从开头开始写入，覆盖已有的数据
ios_base::trunc	打开文件，并删除(截断)任何已有数据

注意，可组合模式。例如，如果要打开文件用于输出(以二进制模式)，同时截断现有数据，可采用如下方式指定打开模式。

```
ios_base::out | ios_base::binary | ios_base::trunc
```

ifstream 自动包含 ios\_base::in 模式，ofstream 自动包含 ios\_base::out 模式，即使不显式地将 in 或 out 指定为模式，也同样如此。

下面的程序打开文件 test.txt，并写入程序的实参。ifstream 和 ofstream 析构函数会自动关闭底层文件，因此不需要显式调用 close()：

```
int main(int argc, char* argv[])
{
    ofstream outFile { "test.txt", ios_base::trunc };
    if (!outFile.good()) {
        cerr << "Error while opening output file!" << endl;
        return -1;
    }
    outFile << "There were " << argc << " arguments to this program." << endl;
    outFile << "They are: " << endl;
    for (int i { 0 }; i < argc; i++) {
        outFile << argv[i] << endl;
    }
}
```

### 13.3.1 文本模式与二进制模式

默认情况下，文件流在文本模式中打开。如果指定 ios\_base::binary 标志，将在二进制模式中打开文件。

在二进制模式中，要求把流处理的字节写入文件。读取时，将完全按文件中的形式返回字节。

在文本模式中，会执行一些隐式转换，写入文件或从文件中读取的每一行都以\n 结束。但是，行结束符在文件中的编码方式与操作系统相关。例如，在 Windows 上，行结束符是\r\n，而不是单个字符\n。因此，如果文件以文本模式打开，那么写入的行以\n 结尾，并且在写入文件前，底层实现会自动将\n 转换为\r\n。同样，从文件读取行时，从文件读取的\r\n 会自动转换回\n。

### 13.3.2 通过 seek()和 tell()在文件中转移

所有的输入流和输出流都有 seekx()和 tellx()方法。seekx()方法允许在输入流或输出流中移动到任意位置。seek()有好几种形式。对于输入流，这个方法实际上称为 seekg()(g 表示 get)；对于输出流，

这个方法实际上称为 seekp(p 表示 put)。为什么同时存在 seekg() 和 seekp() 方法，而不是 seek() 方法？原因是有的流既可以输入又可以输出，例如文件流。在这种情况下，流需要记住读位置和独立的写位置。这也称为双向 I/O(bidirectional I/O)，具体细节将在本章后面讨论。

seekg() 和 seekp() 有两个重载版本。其中一个重载版本接收一个实参：绝对位置。这个重载版本将定位到这个绝对位置。另一个重载版本接收一个偏移量和一个位置，这个重载版本将定位到距离给定位置一定偏移量的位置。位置的类型为 std::streampos，偏移量的类型为 std::streamoff，这两种类型都以字节计数。预定义的 3 个位置如表 13-3 所示。

表 13-3 预定义的 3 个位置

位置	说明
ios_base::beg	表示流的开头
ios_base::end	表示流的结尾
ios_base::cur	表示流的当前位置

例如，要定位到输出流中的一个绝对位置，可使用接收一个参数的 seekp() 版本，如下所示，这个示例通过 ios\_base::beg 常量定位到流的开头。

```
outStream.seekp(ios_base::beg);
```

在输入流中，定位方法完全一样，只不过用的是 seekg() 方法。

```
inStream.seekg(ios_base::beg);
```

接收两个实参的版本可定位到流中的相对位置。第 1 个实参表示要移动的位置数，第 2 个实参表示起始点。要相对文件的起始位置移动，使用 ios\_base::beg 常量。要相对文件的末尾位置移动，使用 ios\_base::end 常量。要相对文件的当前位置移动，使用 ios\_base::cur 常量。例如，下面这行代码从流的起始位置移动到第 2 个字节。注意，整数被隐式地转换为 streampos 和 streamoff 类型：

```
outStream.seekp(2, ios_base::beg);
```

下例转移到输入流中的倒数第 3 个字节：

```
inStream.seekg(-3, ios_base::end);
```

可以通过 tellx() 方法查询流的当前位置，这个方法返回一个表示当前位置的 streampos 值。利用这个结果，可在执行 seekx() 之前记住当前标记的位置，还可查询是否在某个特定位置。和 seekx() 一样，输入流和输出流也有不同版本的 tellx()。输入流使用的是 tellg()，而输出流使用的是 tellp()。

下面的代码检查输入流的当前位置，并判断是否在起始位置。

```
std::streampos curPos { inStream.tellg() };
if (ios_base::beg == curPos) {
    cout << "We're at the beginning." << endl;
}
```

下面是一个整合了所有内容的示例程序。这个程序写入 test.out 文件，并执行下面的测试：

- (1) 将字符串 54321 输出至文件。
- (2) 验证标记在流中的位置 5。
- (3) 转移到输出流的位置 2。
- (4) 在位置 2 输出 0，并关闭输出流。
- (5) 在文件 test.out 上打开输入流。

(6) 将第一个标记以整数的形式读入。

(7) 确认这个值是否为 54021。

```

ofstream fout { "test.out" };
if (!fout) {
    cerr << "Error opening test.out for writing" << endl;
    return 1;
}

// 1. Output the string "54321".
fout << "54321";

// 2. Verify that the marker is at position 5.
streampos curPos { fout.tellp() };
if (curPos == 5) {
    cout << "Test passed: Currently at position 5" << endl;
} else {
    cout << "Test failed: Not at position 5" << endl;
}

// 3. Move to position 2 in the output stream.
fout.seekp(2, ios_base::beg);

// 4. Output a 0 in position 2 and close the output stream.
fout << 0;
fout.close();

// 5. Open an input stream on test.out.
ifstream fin { "test.out" };
if (!fin) {
    cerr << "Error opening test.out for reading" << endl;
    return 1;
}

// 6. Read the first token as an integer.
int testVal;
fin >> testVal;
if (!fin) {
    cerr << "Error reading from file" << endl;
    return 1;
}

// 7. Confirm that the value is 54021.
const int expected { 54021 };
if (testVal == expected) {
    cout << format("Test passed: Value is {}", expected) << endl;
} else {
    cout << format("Test failed: Value is not {} (it was {})" ,
        expected, testVal) << endl;
}

```

### 13.3.3 将流链接在一起

任何输入流和输出流之间都可以建立链接，从而实现“访问时刷新(flush-on-access)”的行为。换句话说，当从输入流请求数据时，链接的输出流会自动刷新。这种行为可用于所有流，但对于可能互

相依赖的文件流来说特别有用。

通过 tie()方法完成流的链接。要将输出流链接至输入流，对输入流调用 tie()方法，并传入输出流的地址。要解除链接，需要传入 nullptr。

下面的程序将一个文件的输入流链接至一个完全不同的文件的输出流。也可链接至同一个文件的输出流，但是双向 I/O(详见稍后的描述)可能是实现同时读写同一个文件的更优雅方式。

```
ifstream inFile { "input.txt" }; // Note: input.txt must exist.
ofstream outFile { "output.txt" };
// Set up a link between inFile and outFile.
inFile.tie(&outFile);
// Output some text to outFile. Normally, this would
// not flush because std::endl is not sent.
outFile << "Hello there!";
// outFile has NOT been flushed.
// Read some text from inFile. This will trigger flush()
// on outFile.
string nextToken;
inFile >> nextToken;
// outFile HAS been flushed.
```

flush()方法在 ostream 基类上定义，因此可将一个输出流链接至另一个输出流。示例如下：

```
outFile.tie(&anotherOutputFile);
```

这种关系意味着：每次写入一个文件时，发送给另一个文件的缓存数据会被刷新。可通过这种机制保持两个相关文件的同步。

这种流链接的一个例子是 cout 和 cin 之间的链接。每当从 cin 输入数据时，都会自动刷新 cout。cerr 和 cout 之间也存在链接，这意味着到 cerr 的任何输出都会导致刷新 cout，而 clog 未链接到 cout。这些流的宽字符版本也具有类似的链接。

## 13.4 双向 I/O

目前，本章把输入流和输出流当作独立但又关联的类来讨论。事实上，有一种流可同时执行输入和输出：bidirectional stream。

双向流是 iostream 的子类，而 iostream 是 istream 和 ostream 的子类，因此这是一个多重继承示例。显然，双向流支持>>和<<运算符，还支持输入流和输出流的方法。

fstream 类提供了双向文件流。fstream 特别适用于需要替换文件中数据的应用程序，因为可通过读取文件找到正确的位置，然后立即切换为写入文件。例如，假设程序保存了 ID 号和电话号码之间的映射表。它可能使用以下格式的数据文件：

```
123 408-555-0394
124 415-555-3422
263 585-555-3490
100 650-555-3434
```

一种合理方案是当这个程序打开文件时读取整个数据文件，然后在程序结束时，将所有的变化重新写入这个文件。然而，如果数据集庞大，可能无法把所有数据都保存在内存中。如果使用 iostream，则不需要这样做。可轻松扫描文件，找到记录，然后以追加模式打开输出文件，从而添加新的记录。如果要修改已有记录，可使用双向流，例如在下面的函数中，可替换指定 ID 的电话号码：

```

bool changeNumberForID(string_view filename, int id, string_view newNumber)
{
    fstream ioData {filename.data()};
    if (!ioData) {
        cerr << "Error while opening file " << filename << endl;
        return false;
    }

    // Loop until the end of file
    while (ioData) {
        int idRead;
        // Read the next ID.
        ioData >> idRead;
        if (!ioData)
            break;

        // Check to see if the current record is the one being changed.
        if (idRead == id) {
            // Seek the write position to the current read position.
            ioData.seekp(ioData.tellg());
            // Output a space, then the new number.
            ioData << " " << newNumber;
            break;
        }

        // Read the current number to advance the stream.
        String number;
        ioData >> number;
    }
    return true;
}

```

当然，只有在数据大小固定时，这种方法才能正常工作。当以上程序从读取切换到写入时，输出数据会改写文件中的其他数据。为保持文件的格式，并避免写入下一条记录，数据大小必须相同。还可通过 `stringstream` 类双向访问字符串流。

#### 注意：

双向流用不同的指针保存读位置和写位置。在读取和写入之间切换时，需要定位到正确的位置。

## 13.5 文件系统支持库

C++标准库包含一个文件系统支持库，定义在`<filesystem>`头文件中，并且位于 `std::filesystem` 名称空间中。它允许编写可移植的代码来处理文件系统。可以使用这个程序库来查询某个内容是目录还是文件、遍历目录的内容、操作路径，以及检索有关文件的信息，如文件的大小、扩展名、创建时间等。程序库的最重要的两个部分(路径和目录条目)将在下一节介绍。

### 13.5.1 路径

程序库的基本组成部分是路径。路径可以是绝对的，也可以是相对的，并且可以选择包含文件名。例如，下面的代码定义了几个路径。注意第2章介绍的原始字符串文字的使用，以避免转义反斜杠：

```
path p1 { R"(D:\Foo\Bar)" };
```

```
path p2 { "D:/Foo/Bar" };
path p3 { "D:/Foo/Bar/MyFile.txt" };
path p4 { R"(..\SomeFolder)" };
path p5 { "/usr/lib/X11" };
```

通过调用 `c_str()` 或 `native()` 或将路径插入流，可以将路径转换为代码所运行的系统的本机格式。示例如下：

```
path p1 { R"(D:\Foo\Bar)" };
path p2 { "D:/Foo/Bar" };
cout << p1 << endl;
cout << p2 << endl;
```

在支持正反斜杠的 Windows 上，输出如下：

```
"D:\\\\Foo\\\\Bar"
"D:/Foo/Bar"
```

可以使用 `append()` 方法或 `operator+=` 将组件附加到路径中。平台相关的路径分隔符会自动插入。示例如下：

```
path p { "D:\\\\Foo" };
p.append("Bar");
p /= "Bar";
cout << p << endl;
```

Windows 上的输出是 "D:\\\\Foo\\\\Bar\\\\Bar"。

可以使用 `concat()` 或 `operator+=` 将字符串连接到现有路径。这不会插入任何路径分隔符！示例如下：

```
path p { "D:\\\\Foo" };
p.concat("Bar");
p += "Bar";
cout << p << endl;
```

现在在 Windows 上的输出是 "D:\\\\FooBarBar"。

### 警告：

`append()` 和 `operator+=` 会自动插入一个平台相关的路径分隔符，而 `concat()` 和 `operator+=` 却不会。

基于范围的 `for` 循环可用于遍历路径的不同组件。示例如下：

```
path p { R"(C:\\Foo\\Bar)" };
for (const auto& component : p) {
    cout << component << endl;
}
```

Windows 上的输出如下：

```
"C:"
"\\"
"Foo"
"Bar"
```

`path` 接口支持 `remove_filename()`、`replace_filename()`、`replace_extension()`、`root_name()`、`parent_path()`、`extension()`、`stem()`、`filename()`、`has_extension()`、`is_absolute()`、`is_relative()` 等操作。下面的代码片段演

示了其中的一些：

```
path p { R"(C:\Foo\Bar\file.txt)" };
cout << p.root_name() << endl;
cout << p.filename() << endl;
cout << p.stem() << endl;
cout << p.extension() << endl;
```

这段代码在 Windows 上产生以下结果：

```
"C:"
"file.txt"
"file"
".txt"
```

有关所有可用功能的完整列表，请参阅最喜欢的标准库参考。

### 13.5.2 目录条目

路径仅表示文件系统中的目录或文件。路径可能会指向不存在的目录或文件。如果要查询文件系统上的实际目录或文件，需要从路径构造一个 `directory_entry`。`directory_entry` 接口支持 `exists()`、`is_directory()`、`is_regular_file()`、`file_size()`、`last_write_time()` 等操作。

下面的示例从路径构造了一个 `directory_entry` 来查询文件的大小：

```
path myPath { "c:/windows/win.ini" };
directory_entry dirEntry { myPath };
if (dirEntry.exists() && dirEntry.is_regular_file()) {
    cout << "File size: " << dirEntry.file_size() << endl;
}
```

### 13.5.3 辅助函数

还有一个完整的辅助函数集合。例如，可以使用 `copy()` 复制文件或目录；`create_directory()` 在文件系统上创建一个新目录；`exists()` 用来查询给定目录或文件是否存在；`file_size()` 获取一个文件的大小；`last_write_time()` 获取文件最后修改的时间；`remove()` 用来删除一个文件；`temp_directory_path()` 获取适合存储临时文件的目录；`space()` 用于查询文件系统上的可用空间等。有关完整列表，请参阅标准库参考。

下面的示例打印了文件系统的容量和剩余空间的大小：

```
space_info s { space("c:\\") };
cout << "Capacity: " << s.capacity << endl;
cout << "Free: " << s.free << endl;
```

在下面的目录遍历部分中，可以找到关于这些辅助函数的更多示例。

### 13.5.4 目录遍历

如果希望递归地遍历给定目录中的所有文件和子目录，可以使用 `recursive_directory_iterator`。如果要开始迭代过程，那么需要一个指向第一个 `directory_entry` 的迭代器。如果要知道何时停止迭代，那么需要一个结束迭代器。如果要创建起始迭代器，那么需要构造一个 `recursive_directory_iterator`，并将要遍历目录的路径作为参数传递。如果要构造结束迭代器，那么需要默认构造一个 `recursive_directory_iterator`。如果要访问迭代器所引用的 `directory_entry`，请使用解引用操作符\*。遍历

集合中的所有元素只需要使用`++`运算符对迭代器递增，直到到达结束迭代器即可完成。注意，结束迭代器不再是集合的一部分，因此不再引用有效的`directory_entry`，并且不能被解引用。

```
void printDirectoryStructure(const path& p)
{
    if (!exists(p)) {
        return;
    }

    recursive_directory_iterator begin { p };
    recursive_directory_iterator end { };
    for (auto iter { begin }; iter != end; ++iter) {
        const string spacer(iter.depth() * 2, ' ');

        auto& entry { *iter }; // Dereference iter to access directory_entry.

        if (is_regular_file(entry)) {
            cout << format("{}File: {} ({} bytes)",
                           spacer, entry.path().string(), file_size(entry)) << endl;
        } else if (is_directory(entry)) {
            cout << format("{}Dir: {}", spacer, entry.path().string()) << endl;
        }
    }
}
```

这个函数可以调用如下：

```
path p { R"(D:\Foo\Bar)" };
printDirectoryStructure(p);
```

还可以使用`directory_iterator`迭代目录的内容，并实现递归。下面的示例与前面的示例做了相同的事情，但是使用了`directory_iterator`，而不是`recursive_directory_iterator`。

```
void printDirectoryStructure(const path& p, size_t level = 0)
{
    if (!exists(p)) {
        return;
    }

    const string spacer(level * 2, ' ');

    if (is_regular_file(p)) {
        cout << format("{}File: {} ({} bytes)",
                       spacer, p.string(), file_size(p)) << endl;
    } else if (is_directory(p)) {
        cout << format("{}Dir: {}", spacer, p.string()) << endl;
        for (auto& entry : directory_iterator { p }) {
            printDirectoryStructure(entry, level + 1);
        }
    }
}
```

## 13.6 本章小结

流为输入输出提供了一种灵活且面向对象的方式。本章中最重要的内容是流的概念，这个概念甚

至比流的使用还重要。一些操作系统可能有自己的文件访问和 I/O 工具，但掌握流和类似流的库的工作方式，是使用任何类型现代 I/O 系统的关键。

本章最后介绍了文件系统支持库，可以使用它以平台无关的方式处理文件和目录。

## 13.7 练习

通过完成以下习题，可以巩固本章讨论的知识点。所有练习的答案都可扫描封底二维码下载。然而，如果你困在一个练习中，在网站上查看答案之前，请先重新阅读本章的内容，试着自己找到答案。

**练习 13-1** 请回顾一下在前面章节的练习中开发的 Person 类。从练习 9-2 中获取实现，并添加一个 output()方法，该方法将一个人的详细信息写入标准输出控制台。

**练习 13-2** 上一个练习中的 output()方法总是将一个人的详细信息写入标准输出控制台。请更改 output()方法，使其具有一个输出流作为参数，并将一个人的详细信息写入该流。通过将一个人写入标准输出控制台、一个字符串流和一个文件，在 main()中测试新的实现。请注意，如何通过使用流的单一方法将一个人输出到各种不同的目标(输出控制台、字符串流、文件等)。

**练习 13-3** 开发一个名为 Database 的类，在 std::vector 中存储 Persons(来自练习 13-2)。提供一个 add()方法来将一个人添加到数据库中，还提供一个 save()方法，接收一个文件的名称，它将数据库中的所有人保存到该文件中，并且文件中的任何现有内容都将被删除。添加一个 load()方法，接收一个文件的名称，并且数据库从中加载所有人的信息。提供 clear()方法，以便从数据库中删除所有人员。最后，添加一个 outputAll()方法，对数据库中的所有人员调用 output()。即使一个人的名字或姓氏中有空格，也要确保该实现可以工作。

**练习 13-4** 练习 13-3 中的数据库将所有人存储在一个文件中。为了练习文件系统支持库，将其更改为将每个人存储在其自己的文件中。修改 save()和 load()方法以接收一个目录作为参数，文件应该存储在这个目录中或从该目录加载。save()方法将数据库中的每个人保存到其自己的文件中，以该人的姓名首字母作为名称，并使用 .person 作为扩展名。如果这个文件已经存在，那么将会被覆盖。load()方法遍历给定目录中的所有 .person 文件，并加载所有这些文件。



# 第14章

# 错误处理

## 本章内容

- 如何处理 C++ 中的错误，异常有哪些优缺点
- 异常的语法
- 异常类层次结构和多态性
- 堆栈的释放和清理
- 常见的错误处理情况

C++ 程序不可避免地会遇到错误。例如，程序可能无法打开某个文件，网络连接可能断开，或者用户可能输入不正确的值。C++ 语言提供了一个名为“异常”的特性，用来处理这些不正常的但能预料的情况。

为简单起见，本书大部分代码到目前为止都忽略了出错的情况。这一章讲述如何在一开始就将错误处理整合到程序中，以改正这种简化状况。本章重点介绍 C++ 异常(包括语法的细节)，并讲述如何有效地利用异常创建设计良好的错误处理程序。

## 14.1 错误与异常

程序不是孤立存在的；它们都依赖于外部工具，例如操作系统界面、网络和文件系统、外部代码(如第三方库)和用户输入。所有这些领域都可能出现这样的状况：需要响应所遇到的错误。这些潜在问题就是异常情况(exceptional situations)，这是一个常见术语。即使编写的代码比较完美，也会遇到错误和异常。因此，任何程序开发者都必须包含错误处理功能。某些语言(例如 C)没有包含太多用于错误处理的特定语言工具，使用这种语言的程序员通常依赖于函数的返回值和其他专门方法。其他语言(例如 Java)强迫使用名为“异常”的语言特性作为错误处理机制。C++ 介于这两个极端之间，提供了对异常的语言支持，但不要求使用异常。然而，在 C++ 中无法完全忽略异常，因为一些基本工具(例如内存分配例程)会用到它们。

### 14.1.1 异常的含义

异常是这样一种机制：一段代码提醒另一段代码存在“异常”情况或错误情况，所采用的路径与正常的代码路径不同。遇到错误的代码抛出异常，处理异常的代码捕获异常。异常不遵循你所熟悉的

逐步执行的规则，当某段代码抛出异常时，程序控制立刻停止逐步执行，并转向异常处理程序(exception handler)，异常处理程序可在任何地方，可位于同一函数中的下一行，也可在堆栈中相隔好几个函数调用。如果用体育运动做类比，将抛出异常的代码当作棒球的外场手将棒球抛回内场，离球最近的内场手(最近的异常处理程序)会捕获棒球。图 14-1 显示了假想堆栈中的 3 个函数调用。函数 A()具有异常处理程序，A()调用函数 B()，B()调用函数 C()，C()抛出异常。

图 14-2 显示了捕获异常的处理程序。C()和 B()的堆栈帧被删除，只留下 A()。

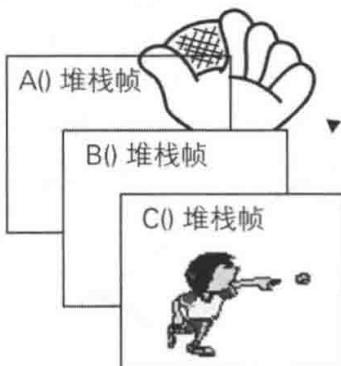


图 14-1 假想堆栈中的 3 个函数调用



图 14-2 捕获异常的处理程序

大多数现代编程语言，例如 C# 和 Java，都支持异常，C++ 也全面支持异常。使用 C 的一些程序员可能没有见过异常，但是，一旦习惯使用异常，就离不开它们了。

### 14.1.2 C++ 中异常的优点

如前所述，程序运行时错误是不可避免的。尽管如此，多数 C 和 C++ 程序中的错误处理比较混乱，不能普遍适用。事实上，C 错误处理标准使用函数返回的整数代码和 `errno` 宏表示错误，许多 C++ 程序也采用了这种方法。每个线程都有自己的 `errno` 值。`errno` 用作线程局部整数变量(thread-local integer variable)，被调用函数使用这个变量将发生的错误告诉调用函数。

遗憾的是，整数返回代码和 `errno` 的使用并不一致。有些函数可能用返回值 0 表示成功，用 -1 表示错误。如果函数返回 -1，还会将 `errno` 设置为某个错误代码。另一些函数用返回值 0 表示成功，用非 0 值表示错误，实际的返回值表示错误代码，这些函数没有使用 `errno`。还有一些函数将返回值 0 当作错误而不是成功，这大概是因为在 C 和 C++ 中，0 的求值结果为 `false`。

这些不一致性可能会引起问题，因为程序员在遇到新函数时，会假定它的返回代码与其他类似函数相同，这一假定并非总是正确的。在 Solaris 9 中，有两个不同的同步对象库：POSIX 版本和 Solaris 版本。在 POSIX 版本中初始化信号量的函数是 `sem_init()`，而在 Solaris 版本中初始化信号量的函数是 `sema_init()`。不仅如此，这两个函数处理错误代码的方式也不同！`sem_init()` 返回 -1 并根据错误设置 `errno`，而 `sema_init()` 直接将错误代码作为正整数返回，并未设置 `errno`。

另一个问题是，C++ 中函数的返回类型只能有一种。因此，如果需要返回一个错误和一个值，就必须寻找其他机制。解决方案之一是返回 `std::pair` 或 `std::tuple`，这两个对象可用来存储两种或多种类型。在第 1 章中已经介绍过 `pair` 类，将在第 16 章中介绍 `tuple` 类。另一个选择是定义自己的结构或类，使其包含多个值，然后让函数返回结构或类的实例。还有一个选择是使用引用参数返回值或错误，或将错误代码作为返回类型的一个可能值，例如 `nullptr` 指针。所有这些情况都要求调用者负责显式地检测函数返回的所有错误，如果函数没有处理错误，就应该将错误提交给调用者。遗憾的是，这样做经常导致遗失与错误有关的重要细节。

C 程序员可能很熟悉 `setjmp()`/`longjmp()` 机制，这一机制在 C++ 中无法正确使用，因为会绕开堆栈中的作用域析构函数。应尽力避免使用这一机制，在 C 程序中也是如此；因此本书不解释这一机制。

的使用细节。

异常提供了方便、一致、安全的错误处理机制。相对于 C 和 C++ 中的专门方法，异常具有许多优点。

- 将返回代码作为报告错误的机制时，调用者可能会忽略返回的代码，不进行局部处理或不将错误代码向上提交。第 1 章介绍的 [[nodiscard]] 特性提供了可行的解决方案，以防止返回代码被忽略，但这并非周全的方案。异常不能被忽略：如果没有捕获异常，程序会终止。
- 返回的整数代码通常不会包含足够的信息。使用异常时，可将任何信息从发现错误的代码传递到处理错误的代码。除错误信息外，异常还可用来传递其他信息，尽管许多程序员（包括我自己）认为这样做是滥用异常机制。
- 异常处理可跳过调用堆栈的层次。也就是说，某个函数可处理沿着堆栈进行数次函数调用后发生的错误，而中间函数不需要有错误处理程序。返回代码要求堆栈中每一层调用的函数都必须在前一层之后显式地执行清理，并且要显式地传播错误码。

在某些编译器中（现在这种编译器越来越少），异常处理让所有具有异常处理程序的函数都多了一点儿开销。在现代编译器中，不抛出异常时几乎没有这个开销，实际抛出异常时这一开销也非常小。这并不是坏事，因为抛出异常应是例外情况。

在 C++ 中，并不强制异常处理。在 Java 中是强制的，没有给出可能抛出异常列表的函数不允许抛出任何异常。在 C++ 中恰好相反，函数可抛出它想要抛出的任何异常，除非指定不会抛出任何异常（使用 noexcept 关键字）。

### 14.1.3 建议

异常是一种有用的错误处理机制，异常提供的结构和错误处理形式的优点大于缺点。因此，本章剩余的部分将重点讨论异常。此外，许多流行的库（例如标准库和 Boost）都使用了异常，因此应该准备好处理这些异常。

## 14.2 异常机制

在文件的输入输出中经常发生异常情况。下面的函数打开一个文件，从这个文件中读取整数列表，然后将整数存储在 std::vector 数据结构中。其中缺少错误处理代码：

```
vector<int> readIntegerFile(string_view filename)
{
    ifstream inputStream {filename.data()};
    // Read the integers one-by-one and add them to a vector.
    vector<int> integers;
    int temp;
    while (inputStream >> temp) {
        integers.push_back(temp);
    }
    return integers;
}
```

下面的代码行从 ifstream 持续地读取值，一直到文件的结尾或发生错误为止。

```
while (inputStream >> temp) {
```

如果 >> 运算符遇到错误，就会设置 ifstream 对象的错误位。在此情况下，bool() 转换运算符将返

回 false, while 循环将终止。有关流的内容已在第 13 章“揭秘 C++ I/O”详细讨论。

可这样使用 readIntegerFile():

```
const string filename { "IntegerFile.txt" };
vector<int> myInts { readIntegerFile(filename) };
for (const auto& element : myInts) {
    cout << element << " ";
}
cout << endl;
```

本节的其余内容将说明如何使用异常进行错误处理，但首先需要深入理解如何抛出和捕获异常。

### 14.2.1 抛出和捕获异常

为了使用异常，要在程序中包括两部分：处理异常的 try/catch 结构和抛出异常的 throw 语句。二者都必须以某种形式出现，以进行异常处理。然而在许多情况下，throw 在一些库的深处(包括 C++ 运行时)发生，程序员无法看到这一点，但仍然不得不用 try/catch 结构处理抛出的异常。

try/catch 结构如下所示：

```
try {
    // ... code which may result in an exception being thrown
} catch (exception-type1 exception-name) {
    // ... code which responds to the exception of type 1
} catch (exception-type2 exception-name) {
    // ... code which responds to the exception of type 2
}
// ... remaining code
```

导致抛出异常的代码可能直接包含 throw 语句，也可能调用一个函数，这个函数可能直接抛出异常，也可能经过多层调用后抛出，每层调用一个抛出异常的函数。

如果没有抛出异常，catch 块中的代码不会执行，其后“剩余的代码”将在 try 块最后执行的语句之后执行。

如果抛出了异常，throw 语句之后或者在抛出异常的函数后的代码不会执行，根据抛出的异常的类型，控制会立刻转移到对应的 catch 块。

如果 catch 块没有执行控制转移(例如从函数中返回，抛出新的异常或者重新抛出异常)，那么会执行 catch 块最后语句之后的“剩余代码”。

演示异常处理的最简单示例是避免除 0。这个示例抛出一个 std::invalid\_argument 类型的异常，它定义在<stdexcept>中。

```
double SafeDivide(double num, double den)
{
    if (den == 0)
        throw invalid_argument { "Divide by zero" };
    return num / den;
}

int main()
{
    try {
        cout << SafeDivide(5, 2) << endl;
        cout << SafeDivide(10, 0) << endl;
    }
```

```

        cout << SafeDivide(3, 3) << endl;
    } catch (const invalid_argument& e) {
        cout << "Caught exception: " << e.what() << endl;
    }
    return 0;
}

```

输出如下所示：

```

2.5
Caught exception: Divide by zero

```

`throw` 是 C++ 中的关键字，这是抛出异常的唯一方法。在前面的代码片段中，抛出了一个 `invalid_argument` 的新对象。这是 C++ 标准库提供的标准异常。标准库中的所有异常构成了一个层次结构，详见本章后面的内容。该层次结构中的每个类都支持 `what()` 方法，该方法返回一个描述异常的 `const char*` 字符串。该字符串在异常的构造函数中提供。

#### 注意：

虽然 `what()` 的返回类型是 `const char*`，但如果使用 UTF-8 编码，异常可支持 Unicode 字符串。有关 Unicode 字符串的详情，可参阅第 21 章“字符串的本地化和正则表达式”。

回到 `readIntegerFile()` 函数，最容易发生的问题就是打开文件失败。这正是需要抛出异常的情况，代码抛出一个 `std::exception` 类型的异常，这种异常类型定义在 `<exception>` 中。如果文件打开失败：

```

vector<int> readIntegerFile(string_view filename)
{
    ifstream inputStream {filename.data()};
    if (inputStream.fail()) {
        // We failed to open the file: throw an exception.
        throw exception {};
    }

    // Read the integers one-by-one and add them to a vector.
    vector<int> integers;
    int temp;
    while (inputStream >> temp) {
        integers.push_back(temp);
    }
    return integers;
}

```

#### 注意：

始终在代码文档中记录函数可能抛出的异常，因为函数的用户需要了解可能抛出哪些异常，从而加以适当处理。

如果函数打开文件失败并执行了 `throw exception();` 语句，那么函数的其余部分将被跳过，把控制转交给最近的异常处理程序。

如果还编写了处理异常的代码，这种情况下抛出异常效果最好。异常处理是这样一种方法：“尝试”执行一块代码，并用另一块代码响应可能发生的任何错误。在下面的 `main()` 函数中，`catch` 语句响应任何被 `try` 块抛出的 `exception` 类型异常，并输出错误消息。如果 `try` 块结束时没有抛出异常，`catch` 块将被忽略。可将 `try/catch` 块当作 `if` 语句。如果在 `try` 块中抛出异常，就会执行 `catch` 块，否则忽略 `catch` 块。

```

int main()
{
    const string filename { "IntegerFile.txt" };
    vector<int> myInts;
    try {
        myInts = readIntegerFile(filename);
    } catch (const exception& e) {
        cerr << "Unable to open file " << filename << endl;
        return 1;
    }
    for (const auto& element : myInts) {
        cout << element << " ";
    }
    cout << endl;
}

```

**注意：**

尽管默认情况下，流不会抛出异常，但是针对错误情况，仍然可以调用 exceptions() 方法通知流抛出异常。但是，大多数编译器都会在抛出的流异常中给出无用信息。对于这些编译器，最好直接处理流状态而不是使用异常。本书不使用流异常。

### 14.2.2 异常类型

可抛出任何类型的异常。可以抛出一个 std::exception 类型的对象，但异常未必是对象。也可以抛出一个简单的 int 值，如下所示：

```

vector<int> readIntegerFile(string_view filename)
{
    ifstream inputStream { filename.data() };
    if (inputStream.fail()) {
        // We failed to open the file: throw an exception
        throw 5;
    }
    // Omitted for brevity
}

```

此后必须修改 catch 语句：

```

try {
    myInts = readIntegerFile(filename);
} catch (int e) {
    cerr << format("Unable to open file {} (Error Code {})", filename, e) << endl;
    return 1;
}

```

另外，也可抛出一个 C 风格的 const char\* 字符串。这项技术有时有用，因为字符串可包含与异常相关的信息。

```

vector<int> readIntegerFile(string_view filename)
{
    ifstream inputStream { filename.data() };
    if (inputStream.fail()) {
        // We failed to open the file: throw an exception.
        throw "Unable to open file";
}

```

```

    // Omitted for brevity
}

当捕获 const char* 异常时，可输出结果：

try {
    myInts = readIntegerFile(filename);
} catch (const char* e) {
    cerr << e << endl;
    return 1;
}

```

尽管前面有这样的示例，但通常应将对象作为异常抛出，原因有以下两点：

- 对象的类名可传递信息。
- 对象可存储信息，包括描述异常的字符串。

C++标准库在类层次结构中定义了许多预定义的异常类，也可编写自己的异常类，并将它们放入标准层次结构中。本章后面将就此详细讨论。

### 14.2.3 按 const 引用捕获异常对象

在前面的示例中，`readIntegerFile()`抛出一个 `exception` 类型的对象。`catch` 处理如下所示：

```

} catch (const exception& e) {

```

然而，在此并没有要求按 `const` 引用捕获对象。可按值捕获对象，如下所示：

```

} catch (exception e) {

```

此外，也可按非 `const` 引用捕获对象：

```

} catch (exception& e) {

```

另外，如 `const char*` 示例所示，只要指向异常的指针被抛出，就可以捕获它。

#### 注意：

建议按 `const` 引用捕获异常，这可避免按值捕获异常时可能出现的对象截断(可参阅第 10 章“揭秘继承技术” )。

### 14.2.4 抛出并捕获多个异常

打开文件失败并不是 `readIntegerFile()` 遇到的唯一问题。如果格式不正确，读取文件中的数据也会导致错误。下面是 `readIntegerFile()` 的一个实现，如果无法打开文件，或者无法正确读取数据，就会抛出异常。这里使用从 `exception` 派生的 `runtime_error`，它允许你在构造函数中指定描述字符串。我们在`<stdexcept>`中定义 `runtime_error` 异常。

```

vector<int> readIntegerFile(string_view filename)
{
    ifstream inputStream {filename.data()};
    if (inputStream.fail()) {
        // We failed to open the file: throw an exception
        throw runtime_error("Unable to open the file.");
    }

    // Read the integers one-by-one and add them to a vector
    vector<int> integers;

```

```

int temp;
while (inputStream >> temp) {
    integers.push_back(temp);
}

if (!inputStream.eof()) {
    // We did not reach the end-of-file.
    // This means that some error occurred while reading the file.
    // Throw an exception.
    throw runtime_error("Error reading the file.");
}

return integers;
}

```

`main()`中的代码不需要改变，因为已可捕获 `exception` 类型的异常，`runtime_error` 派生于 `exception`。然而，现在可在两种不同情况下抛出该异常，可以通过 `what()` 方法获得被捕获异常适当的描述。

```

try {
    myInts = readIntegerFile(filename);
} catch (const exception& e) {
    cerr << e.what() << endl;
    return 1;
}

```

另外，也可让 `readIntegerFile()` 抛出两种不同类型的异常。下面是 `readIntegerFile()` 的实现，如果不能打开文件，就抛出 `invalid_argument` 类对象；如果无法读取整数，就抛出 `runtime_error` 类对象。`invalid_argument` 和 `runtime_error` 都是定义在`<stdexcept>`中的类，这是 C++ 标准库的一部分。

```

vector<int> readIntegerFile(string_view fileName)
{
    ifstream inputStream {fileName.data()};
    if (inputStream.fail()) {
        // We failed to open the file: throw an exception
        throw invalid_argument { "Unable to open the file." };
    }

    // Read the integers one-by-one and add them to a vector
    vector<int> integers;
    int temp;
    while (inputStream >> temp) {
        integers.push_back(temp);
    }

    if (!inputStream.eof()) {
        // We did not reach the end-of-file.
        // This means that some error occurred while reading the file.
        // Throw an exception.
        throw runtime_error("Error reading the file.");
    }

    return integers;
}

```

`invalid_argument` 和 `runtime_error` 类没有公有的默认构造函数，只有以字符串作为参数的构造函数。

现在 main() 可用两个 catch 语句捕获 invalid\_argument 和 runtime\_error 异常：

```
try {
    myInts = readIntegerFile(fileName);
} catch (const invalid_argument& e) {
    cerr << e.what() << endl;
    return 1;
} catch (const runtime_error& e) {
    cerr << e.what() << endl;
    return 2;
}
```

如果异常在 try 块内部抛出，编译器将使用恰当的 catch 处理程序与异常类型匹配。因此，如果 readIntegerFile() 无法打开文件并抛出 invalid\_argument 异常，第一个 catch 语句将捕获这个异常。如果 readIntegerFile() 无法正确读取文件并抛出 runtime\_error 异常，第二个 catch 语句将捕获这个异常。

### 1. 匹配和 const

对于想要捕获的异常类型而言，增加 const 属性不会影响匹配的目的。也就是说，这一行可以与 runtime\_error 类型的任何异常匹配。

```
} catch (const runtime_error& e) {
```

下面这行也可与 runtime\_error 类型的任何异常匹配：

```
} catch (runtime_error& e) {
```

### 2. 匹配所有异常

可用特定语法编写与所有异常匹配的 catch 语句，如下所示：

```
try {
    myInts = readIntegerFile(fileName);
} catch (...) {
    cerr << "Error reading or opening file " << fileName << endl;
    return 1;
}
```

三个点并非排版错误，而是与所有异常类型匹配的通配符。当调用缺乏文档的代码时，可以用这一技术确保捕获所有可能的异常。然而，如果有被抛出的一组异常的完整信息，这种技术并不理想，因为它将所有异常都同等对待。更好的做法是显式地匹配异常类型，并采取恰当的针对性操作。

与所有异常匹配的 catch 块可以用作默认的 catch 处理程序。当异常抛出时，会按在代码中的显示顺序查找 catch 处理程序。下例用 catch 处理程序显式地处理 invalid\_argument 和 runtime\_error 异常，并用默认的 catch 处理程序处理其他所有异常。

```
try {
    // Code that can throw exceptions
} catch (const invalid_argument& e) {
    // Handle invalid_argument exception
} catch (const runtime_error& e) {
    // Handle runtime_error exception
} catch (...) {
    // Handle all other exceptions
}
```

### 14.2.5 未捕获的异常

如果程序抛出的异常没有捕获，程序将终止。可对 main() 函数使用 try/catch 结构，以捕获所有未经处理的异常，如下所示。

```
try {
    main(argc, argv);
} catch (...) {
    // issue error message and terminate program
}
// Normal termination code
```

然而，这一行为通常并非我们希望的。异常的作用在于给程序一个机会，以处理和修正不希望看到的或不曾预期的情况。

#### 警告：

捕获并处理程序中可能抛出的所有异常。

如果存在未捕获的异常，程序行为也可能发生变化。当程序遇到未捕获的异常时，会调用内建的 terminate() 函数，这个函数调用<cstdlib>中的 abort() 来终止程序。可调用 set\_terminate() 函数设置自己的 terminate\_handler()，这个函数采用指向回调函数(既没有参数，也没有返回值)的指针作为参数。terminate()、set\_terminate() 和 terminate\_handler() 都在<exception>中声明。下面的代码高度概括了其运行原理：

```
try {
    main(argc, argv);
} catch (...) {
    if (terminate_handler != nullptr) {
        terminate_handler();
    } else {
        terminate();
    }
}
// Normal termination code
```

不要为这一特性激动，因为回调函数必须终止程序(使用 abort() 函数或者 \_Exit() 函数)。错误是无法忽略的。abort() 函数和 \_Exit() 函数定义在<cstdlib>中，会在不清理资源的情况下终止程序。例如对象的析构函数不会被调用。\_Exit() 函数接收一个返回给操作系统的整型参数，用于确定进程如何退出。值为 0 或者 EXIT\_SUCCESS 表示程序没有任何错误的退出，否则程序异常终止。abort() 函数不接收任何参数。还有个 exit() 函数，它也接收返回给操作系统的参数，并调用析构函数来清理资源，但不建议在 terminate\_handler 中调用 exit() 函数。

在退出之前可以使用 terminate\_handler 打印有用的消息。下例中，main() 函数没有捕获 readIntegerFile() 抛出的异常，而将 terminate\_handler() 设置为自定义回调。这个回调打印错误消息并调用 \_Exit() 终止进程。

```
[[noreturn]] void myTerminate()
{
    cout << "Uncaught exception!" << endl;
    _Exit(1);
}
```

```

int main()
{
    set_terminate(myTerminate);

    const string filename { "IntegerFile.txt" };
    vector<int> myInts { readIntegerFile(filename) };

    for (const auto& element : myInts) {
        cout << element << " ";
    }
    cout << endl;
}

```

当设置新的 `terminate_handler()` 时, `set_terminate()` 会返回旧的 `terminate_handler()`。`terminate_handler()` 被应用于整个程序, 因此当需要新 `terminate_handler()` 的代码结束后, 最好保存旧的 `terminate_handler()`。上面的示例中, 整个程序都需要新的 `terminate_handler()`, 因此不需要重新保存。

尽管有必要了解 `set_terminate()`, 但这并不是一种非常有效的处理异常的方法。建议分别捕获并处理每个异常, 以提供更精确的错误处理。

#### 注意:

在专门编写的软件中, 通常会设置 `terminate_handler()`, 在进程结束前创建崩溃转储。崩溃转储通常包含未捕获的异常被抛出时的调用栈和局部变量等信息。此后将崩溃转储上传给调试器, 进而确定未捕获的异常是什么, 起因是什么。但是, 崩溃转储的编写方式与平台相关, 因此本书不予进一步讨论。

### 14.2.6 noexcept 说明符

默认情况下函数可抛出任何异常。但可使用 `noexcept` 关键字标记函数, 指出它不抛出任何异常。例如, 为下面的函数标记了 `noexcept`, 即不允许它抛出任何异常:

```
void printValues(const vector<int>& values) noexcept;
```

#### 注意:

带有 `noexcept` 标记的函数不应抛出任何异常。

如果一个函数带有 `noexcept` 标记, 却以某种方式抛出了异常, C++ 将调用 `terminate()` 来终止应用程序。

在派生类中重写虚方法时, 可将重写的虚方法标记为 `noexcept`(即使基类中的版本不是 `noexcept`)。反过来不行。

### 14.2.7 noexcept(expression) 说明符

当且仅当给定的表达式返回 `true` 时, `noexcept(expression)` 说明符才会将函数标记为 `noexcept`。换句话说, `noexcept` 等同于 `noexcept(true)`, 而 `noexcept(false)` 则相反。也就是说, 标记为 `noexcept(false)` 的方法可以抛出任何异常。

### 14.2.8 noexcept(expression) 运算符

如果给定的表达式标记为 `noexcept`, 那么 `noexcept(expression)` 运算符会返回 `true`, 要么使用

`noexcept` 说明符，要么使用 `noexcept(expression)` 说明符。这会在编译时评估。

看下例：

```
void f1() noexcept {}
void f2() noexcept(false) {}
void f3() noexcept(noexcept(f1())) {}
void f4() noexcept(noexcept(f2())) {}
int main()
{
    std::cout << noexcept(f1())
        << noexcept(f2())
        << noexcept(f3())
        << noexcept(f4());
}
```

代码片段的输出为 1010。

- `noexcept(f1())` 值为 true：因为 `f1()` 被 `noexcept` 说明符显式地标记。
- `noexcept(f2())` 值为 false：因为 `f2()` 被 `noexcept(expression)` 说明符标记。
- `noexcept(f3())` 值为 true：因为只在 `f1()` 被标记为 `noexcept` 时，`f3()` 才被标记为 `noexcept`。
- `noexcept(f4())` 值为 false：因为只在 `f2()` 被标记为 `noexcept` 时，`f4()` 才被标记为 `noexcept`，而 `f2()` 没有被标记为 `noexecpt`。

### 14.2.9 抛出列表

C++ 的旧版本允许指定函数或方法可抛出的异常，这种规范称为抛出列表(throw list)或异常规范(exception specification)。

**警告：**

自 C++11 之后，已不赞成使用异常规范；自 C++17 之后，已不再支持异常规范。但 `noexcept` 和 `throw()` 除外，`throw()` 与 `noexcept` 等效。C++20 则完全不支持 `throw()`。

由于 C++17 已正式取消对异常规范的支持，本书将不再讨论它们。

## 14.3 异常与多态性

如前所述，可抛出任何类型的异常。然而，类是最有用的异常类型。实际上异常类通常具有层次结构，因此在捕获异常时可使用多态性。

### 14.3.1 标准异常层次结构

前面介绍了 C++ 标准异常层次结构中的一些异常：`exception`、`runtime_error` 和 `invalid_argument`。图 14-3 显示了完整的层次结构。为完整起见，图 14-3 中显示了所有标准异常，包括标准库抛出的标准异常；后续章节将介绍这些异常。

C++ 标准库中抛出的所有异常都是这个层次结构中类的对象。这个层次结构中的每个类都支持 `what()` 方法，这个方法返回一个描述异常的 `const char*` 字符串。可在错误信息中使用这个字符串。

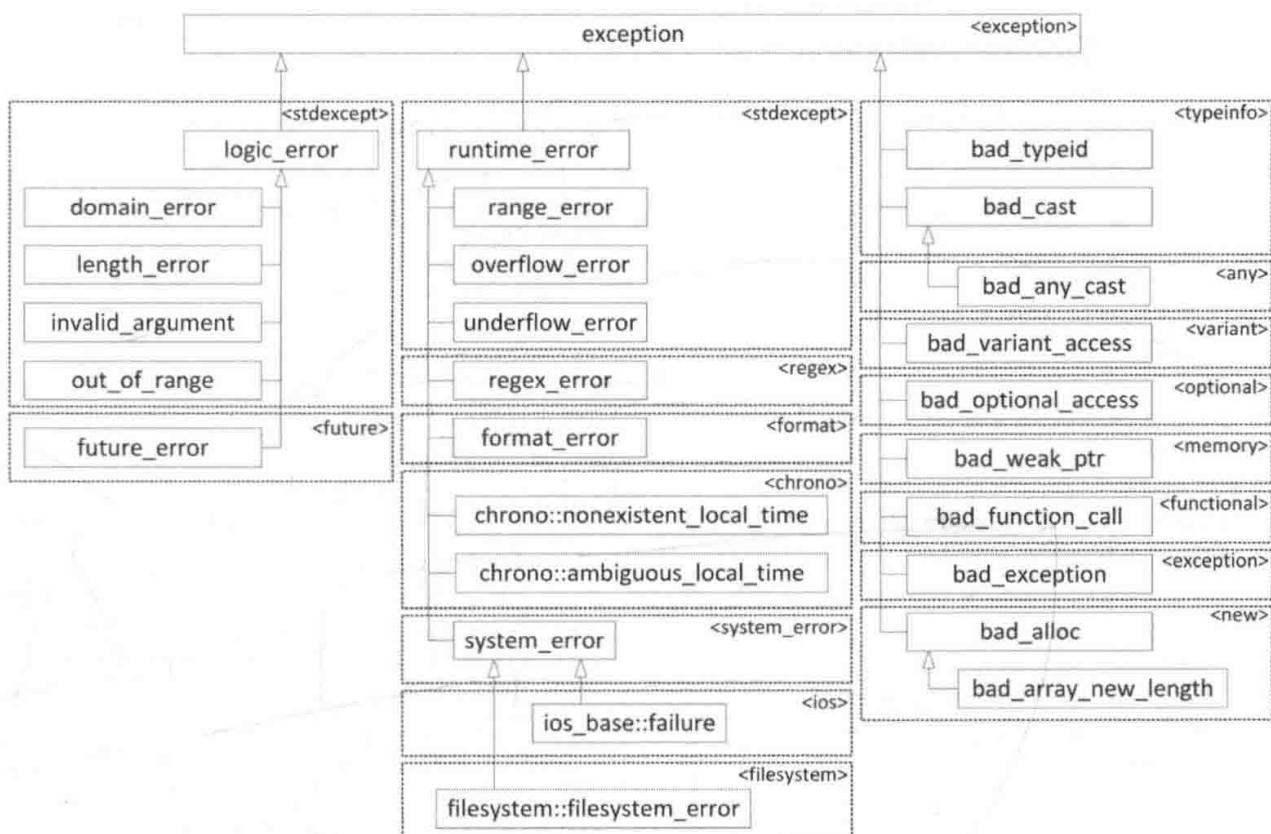


图 14-3 完整的异常层次结构

有些异常类要求在构造函数中设置 what() 返回的字符串。这就是必须在 runtime\_error 和 invalid\_argument 构造函数中指定字符串的原因。本章的示例都是这么做的。下面是 readIntegerFile() 的另一个版本，它也在错误消息中包含文件名。

```

vector<int> readIntegerFile(string_view filename)
{
    ifstream inputStream {filename.data()};
    if (inputStream.fail()) {
        // We failed to open the file: throw an exception
        const string error {format("Unable to open file {}.", filename.data())};
        throw invalid_argument {error};
    }

    // Read the integers one-by-one and add them to a vector
    vector<int> integers;
    int temp;
    while (inputStream >> temp) {
        integers.push_back(temp);
    }

    if (!inputStream.eof()) {
        // We did not reach the end-of-file.
        // This means that some error occurred while reading the file.
        // Throw an exception.
        const string error {format("Unable to open file {}.", filename.data())};
        throw runtime_error {error};
    }

    return integers;
}
  
```

### 14.3.2 在类层次结构中捕获异常

异常层次结构的一个特性是可利用多态性捕获异常。例如，如果观察下面这两条 catch 语句，就可以发现这两条语句除了处理的异常类不同之外没有区别。

```
try {
    myInts = readIntegerFile(filename);
} catch (const invalid_argument& e) {
    cerr << e.what() << endl;
    return 1;
} catch (const runtime_error& e) {
    cerr << e.what() << endl;
    return 1;
}
```

`invalid_argument` 和 `runtime_error` 都是 `exception` 的派生类，因此可使用 `exception` 类的一条 catch 语句替换这两条 catch 语句。

```
try {
    myInts = readIntegerFile(filename);
} catch (const exception& e) {
    cerr << e.what() << endl;
    return 1;
}
```

`exception` 引用的 catch 语句可与 `exception` 的任何派生类匹配，包括 `invalid_argument` 和 `runtime_error`。注意捕获的异常在异常层次结构中的层次越高，错误处理就越不具有针对性。通常应该尽可能有针对性地捕获异常。

#### 警告：

当利用多态性捕获异常时，一定要按引用捕获。如果按值捕获异常，就可能发生截断，在此情况下将丢失对象的信息。关于截断请参阅第 10 章。

当使用了多条 catch 子句时，会按在代码中出现的顺序匹配 catch 子句，第一条匹配的 catch 子句将被执行。如果某条 catch 子句比后面的 catch 子句范围更广，那么这条 catch 子句首先被匹配，而后面限制更多的 catch 子句根本不会被执行。因此，catch 子句应按限制最多到限制最少的顺序出现。例如，假定要显式捕获 `readIntegerFile()` 的 `invalid_argument`，就应该让一般的异常与其他类型的异常匹配。正确做法如下所示：

```
try {
    myInts = readIntegerFile(filename);
} catch (const invalid_argument& e) { // List the derived class first.
    // Take some special action for invalid filenames.
} catch (const exception& e) { // Now list exception
    cerr << e.what() << endl;
    return 1;
}
```

第一条 catch 子句捕获 `invalid_argument` 异常，第二条 catch 子句捕获任何类型的其他异常。然而，如果将 catch 子句的顺序弄反，就不会得到同样的结果。

```
try {
    myInts = readIntegerFile(filename);
} catch (const exception& e) { // BUG: catching base class first!
```

```

        cerr << e.what() << endl;
        return 1;
    } catch (const invalid_argument& e) {
        // Take some special action for invalid filenames.
    }
}

```

使用这一顺序，第一条 catch 子句会捕获任何派生类类型的异常，第二条 catch 子句永远无法执行。某些编译器在此情况下会给出警告，但是不应该指望编译器。

### 14.3.3 编写自己的异常类

编写自己的异常类有两个好处：

- (1) C++ 标准库中的异常数目有限，可在程序中为特定错误创建更有意义的类名，而不是使用具有通用名称的异常类，例如 `runtime_error`。
- (2) 可在异常中加入自己的信息，而标准层次结构中的大多数异常只允许设置错误字符串，例如可能想在异常中传递不同信息。

建议自己编写的异常类从标准的 `exception` 类直接或间接继承。如果项目中的所有人都遵循这条规则，就可确定程序中的所有异常都是 `exception` 类的派生类(假定不会使用第三方库破坏这条规则)。这一指导方针更便于用多态性处理异常。

例如，在 `readIntegerFile()` 中，`invalid_argument` 和 `runtime_error` 不能很好地捕获文件打开和读取错误。可为文件错误定义自己的错误层次结构，从泛型类 `FileError` 开始。

```

class FileError : public exception
{
public:
    FileError(string filename) : m_filename { move(filename) } {}
    const char* what() const noexcept override { return m_message.c_str(); }
    virtual const string& getFilename() const noexcept { return m_filename; }
protected:
    virtual void setMessage(string message) { m_message = move(message); }
private:
    string m_filename;
    string m_message;
};

```

作为一名优秀的程序员，可将 `FileError` 作为标准异常层次结构的一部分。将其作为 `exception` 的子类是恰当的。编写 `exception` 的派生类时，需要重写 `what()` 方法，这个方法的原型已经出现过，其返回值为一个在对象销毁之前一直有效的 `const char*` 字符串。在 `FileError` 中，这个字符串来自 `m_message` 数据成员。`FileError` 的派生类可使用受保护的 `setMessage()` 方法设置消息。泛型类 `FileError` 还包含文件名以及文件名的公共访问器。

`readIntegerFile()` 中的第一种异常情况是无法打开文件。因此，下面编写 `FileError` 的派生类 `FileOpenError`：

```

class FileOpenError : public FileError
{
public:
    FileOpenError(string filename) : FileError { move(filename) }
    {
        setMessage(format("Unable to open {}.", getFilename()));
    }
};

```

FileOpenError 异常修改 m\_message 字符串，令其表示文件打开错误。

readIntegerFile()中的第二种异常情况是无法正确读取文件。对于这一异常，或许应该包含文件中发生错误的行号，以及 what()返回的错误信息字符串中的文件名。下面是 FileError 的派生类 FileReadError：

```
class FileReadError : public FileError
{
public:
    FileReadError(string filename, size_t lineNumber)
        : FileError { move(filename) }, mLineNumber { lineNumber }
    {
        setMessage(format("Error reading {}, line {}.",,
                           getFilename(), lineNumber));
    }

    virtual size_t getLineNumber() const noexcept { return mLineNumber; }

private:
    size_t mLineNumber { 0 };
};
```

当然，为正确地设置行号，需要修改 readIntegerFile() 函数，以跟踪读取的行号，而不只是直接读取整数。下面是使用了新异常的 readIntegerFile() 函数：

```
vector<int> readIntegerFile(string_view filename)
{
    ifstream inputStream { filename.data() };
    if (inputStream.fail()) {
        // We failed to open the file: throw an exception
        throw FileOpenError { filename };
    }

    vector<int> integers;
    size_t lineNumber { 0 };
    while (!inputStream.eof()) {
        // Read one line from the file
        string line;
        getline(inputStream, line);
        ++lineNumber;

        // Create a string stream out of the line
        istringstream lineStream { line };

        // Read the integers one-by-one and add them to a vector
        int temp;
        while (lineStream >> temp) {
            integers.push_back(temp);
        }

        if (!lineStream.eof()) {
            // We did not reach the end of the string stream.
            // This means that some error occurred while reading this line.
            // Throw an exception.
            throw FileReadError { filename, lineNumber };
        }
    }
}
```

```

    return integers;
}

```

现在，调用 `readIntegerFile()` 的代码可使用多态性捕获 `FileError` 类型的异常，如下所示。

```

try {
    myInts = readIntegerFile(filename);
} catch (const FileError& e) {
    cerr << e.what() << endl;
    return 1;
}

```

在编写将其对象用作异常的类时，有一个诀窍。当某段代码抛出一个异常时，使用移动构造函数或复制构造函数，移动或复制被抛出的值或对象。因此，如果编写的类的对象将作为异常抛出，对象必须复制和/或移动。这意味着如果动态分配了内存，就必须编写析构函数、复制构造函数、复制赋值运算符和/或移动构造函数与移动赋值运算符，参见第 9 章“精通类和对象”。

**警告：**

作为异常抛出的对象至少要复制或移动一次。

异常可能被复制多次，但只有按值(而不是按引用)捕获异常才会如此。

**注意：**

按引用(最好是 `const` 引用)捕获异常对象可避免不必要的复制。



#### 14.3.4 源码位置

在 C++20 之前，可以使用以下预处理宏获取源代码中的位置信息，如表 14-1 所示。

表 14-1 预处理宏

宏	描述
<code>_FILE_</code>	用当前源代码文件名替换
<code>_LINE_</code>	用当前源代码的行号替换

此外，每个函数都有一个局部定义的静态字符数组 `_func_`，它包含函数的名字。

C++20 在 `<source_location>` 中，以 `std::source_location` 类的形式，为 `_func_` 和这些 C 风格的预处理器宏引入了一个适当的面向对象方式的替代品。如表 14-2 所示，`source_location` 的实例有下面这些公有访问器(公有方法)。

表 14-2 访问器

访问器	描述
<code>file_name()</code>	包含当前源代码文件名
<code>function_name()</code>	如果当前位置在函数中，则包含当前函数名
<code>line()</code>	包含当前源代码中的当前行号
<code>column()</code>	包含当前源代码中的当前列号

使用静态方法 `current()` 可以在方法被调用的源代码位置上创建 `source_location` 实例。

### 14.3.5 日志记录的源码位置

`source_location` 类对于日志记录很有用。以前，日志记录通常会涉及写一些 C 风格的宏，用来自动收集当前文件名、函数名和行号，以便将它们包括到日志输出中。现在使用 `source_location`，可以编写一个纯 C++ 的函数来记录日志并自动收集所需要的位置数据。如下，一个很好的技巧是定义一个 `logMessage` 函数。这次，代码以行号作为前缀，以便更好地理解代码。

```

8. void logMessage(string_view message,
9.   const source_location& location = source_location::current())
10. {
11.   cout << format("{}({}): {}:{}: {}", location.file_name(),
12.                 location.line(), location.function_name(), message) << endl;
13. }
14.
15. void foo()
16. {
17.   logMessage("Starting execution of foo().");
18. }
19.
20. int main()
21. {
22.   foo();
23. }
```

`logMessage` 的第二个参数是一个 `source_location`，静态方法 `current()` 的结果作为默认参数值。这里的技巧是，对 `current()` 的调用不会发生在第 9 行，实际是在 `logMessage()` 被调用的位置，也就是第 17 行，这恰恰也是感兴趣的位置。

当执行这个程序时，输出如下所示：

```
./01Logging.cpp(17): foo: Starting exceution of foo().
```

第 17 行确实对应于调用 `logMessage()` 的那一行。

### 14.3.6 异常的源码位置

`source_location` 另一个有趣的例子是在自己的异常类中存储抛出异常的位置，如下所示：

```

class MyException : public exception
{
public:
    MyException(string message,
                source_location location = source_location::current())
        : m_message { move(message) }
        : m_location { move(location) }
    {}

    const char* what() const noexcept override { return m_message.c_str(); }
    virtual const source_location& where() const noexcept { return m_location; }

private:
    string m_message;
    source_location m_location;
};
```

```

void doSomething()
{
    throw MyException { "Throwing MyException." };
}

int main()
{
    try {
        doSomething();
    }
    catch (const MyException& e) {
        const auto& location { e.where() };
        cerr << format("Caught: '{}' at line {} in {}.", e.what(), location.line(), location.function_name()) << endl;
    }
}

```

输出如下：

```
Caught: 'Throwing MyException.' at line 30 in doSomething.
```

### 14.3.7 嵌套异常

当处理第一个异常时，可能触发第二个异常，从而要求抛出第二个异常。遗憾的是，当抛出第二个异常时，正在处理的第一个异常的所有信息都会丢失。C++用嵌套异常(nested exception)提供了解决这一问题的方案，嵌套异常允许将捕获的异常嵌套到新的异常环境。如果调用第三方库中抛出某类异常(A)的函数，但代码中需要另一类异常(B)，这也十分有用。此时，从库捕获所有异常，并将它们嵌入类型B的异常中。

使用`std::throw_with_nested()`抛出一个异常时，这个异常中嵌套着另一个异常。第二个异常的`catch`处理程序可使用`dynamic_cast()`访问代表第一个异常的`nested_exception`。下例演示了嵌套异常的用法。这个示例定义了一个从`exception`类派生的`MyException`类，其构造函数接收一个字符串。

```

class MyException : public std::exception
{
public:
    MyException(string message) : m_message { move(message) } {}
    virtual const char* what() const noexcept override { return m_message.c_str(); }
private:
    string m_message;
};

```

下面的`doSomething()`函数抛出一个`runtime_error`异常，这个异常立即被`catch`处理程序捕获。`catch`处理程序编写了一条消息，然后使用`throw_with_nested()`函数抛出第二个异常，第一个异常嵌套在其中。注意嵌套异常是自动实现的：

```

void doSomething()
{
    try {
        throw runtime_error { "Throwing a runtime_error exception" };
    } catch (const runtime_error& e) {
        cout << format("{} caught a runtime_error", __func__) << endl;
        cout << format("{} throwing MyException", __func__) << endl;
    }
}

```

```

        throw_with_nested(
            MyException { "MyException with nested runtime_error" });
    }
}

```

`throw_with_nested()`工作方式是抛出一个未命名的编译器生成的新类型，这个类型是由`nested_exception`和例子中的`MyException`派生而来。因此它是C++中有用的多重继承的另一个示例。`nested_exception`基类的另一个默认构造函数通过调用`std::current_exception()`自动捕获正在处理的异常，并将其存储在`std::exception_ptr`中。`exception_ptr`是一种类似指针的类型，可以存储空指针或用`current_exception()`抛出和捕获的异常对象的指针。`exception_ptr`的实例可以通过不同的线程传递给函数(通常是通过值传递)。

最后，下面的代码片段演示了如何处理具有嵌套异常的异常。这段代码调用了`doSomething()`函数，还有一个处理`MyException`类型的`catch`处理程序。当捕获到这类异常时，会编写一条消息，然后使用`dynamic_cast()`访问嵌套的异常。如果内部没有嵌套异常，结果为空指针。如果有嵌套异常，会调用`nested_exception`的`rethrow_nested()`方法。这样会再次抛出嵌套异常，这一异常可在另一个`try/catch`块中捕获。

```

try {
    doSomething();
} catch (const MyException& e) {
    cout << format("{} caught MyException: {}", __func__, e.what()) << endl;

    const auto* nested = dynamic_cast<const nested_exception*>(&e);
    if (nested) {
        try {
            nested->rethrow_nested();
        } catch (const runtime_error& e) {
            // Handle nested exception
            cout << format(" Nested exception: {}", e.what()) << endl;
        }
    }
}

```

输出如下所示：

```

doSomething caught a runtime_error
doSomething throwing MyException
main caught MyException: MyException with nested runtime_error
    Nested exception: Throwing a runtime_error exception

```

代码使用`dynamic_cast()`检测嵌套异常。如果想要检测嵌套异常，就不得不经常执行`dynamic_cast()`，因此标准库提供了一个名为`std::rethrow_if_nested()`的辅助函数，其用法如下所示。

```

try {
    doSomething();
} catch (const MyException& e) {
    cout << format("{} caught MyException: {}", __func__, e.what()) << endl;
    try {
        rethrow_if_nested(e);
    } catch (const runtime_error& e) {
        // Handle nested exception
        cout << format(" Nested exception: {}", e.what()) << endl;
    }
}

```

`throw_with_nested()`、`nested_exception`、`rethrow_if_nested()`、`current_exception()`和`exception_ptr`都定义在`<exception>`中。

## 14.4 重新抛出异常

可使用`throw`关键字重新抛出当前异常，如下例所示：

```
void g() { throw invalid_argument { "Some exception" }; }

void f()
{
    try {
        g();
    } catch (const invalid_argument& e) {
        cout << "caught in f: " << e.what() << endl;
        throw; // rethrow
    }
}

int main()
{
    try {
        f();
    } catch (const invalid_argument& e) {
        cout << "caught in main: " << e.what() << endl;
    }
}
```

该例生成以下输出：

```
caught in f: Some exception
caught in main: Some exception
```

你或许认为，可使用`throw e;`重新抛出异常，但事实并非如此，因为那样会截断异常对象。例如，假如修改`f()`以捕获`std::exception`异常，修改`main()`以捕获`exception`和`invalid_argument`异常：

```
void g() { throw invalid_argument { "Some exception" }; }

void f()
{
    try {
        g();
    } catch (const exception& e) {
        cout << "caught in f: " << e.what() << endl;
        throw; // rethrow
    }
}

int main()
{
    try {
        f();
    } catch (const invalid_argument& e) {
        cout << "invalid_argument caught in main: " << e.what() << endl;
    } catch (const exception& e) {
```

```

    cout << "exception caught in main: " << e.what() << endl;
}
}

```

记住, invalid\_argument 从 exception 派生而来。上述代码的输出与我们的预期相符:

```

caught in f: Some exception
invalid_argument caught in main: Some exception

```

现在尝试用以下代码替换 f() 中的 throw e; 语句, 输出如下所示:

```

caught in f: Some exception
exception caught in main: Some exception

```

main() 似乎在捕获 exception 对象而非 invalid\_argument 对象。这是由于 throw e; 语句会执行截断操作, 将 invalid\_argument 缩减为 exception。

#### 警告:

始终使用 throw; 重新抛出异常。永远不要试图用 throw e; 重新抛出 e!

## 14.5 堆栈的释放与清理

当某段代码抛出一个异常时, 会在堆栈中寻找 catch 处理程序。catch 处理程序可以是在堆栈上执行的 0 个或多个函数调用。当发现一个 catch 处理程序时, 堆栈会释放所有中间堆栈帧, 直接跳到定义 catch 处理程序的堆栈层。堆栈释放(stack unwinding)意味着调用所有具有局部作用域的名称的析构函数, 并忽略在当前执行点之前的每个函数中的所有代码。

然而当释放堆栈时, 并不释放指针变量, 也不会执行其他清理工作。这一行为会引发问题, 下面的代码演示了这一点:

```

void funcOne();
void funcTwo();

int main()
{
    try {
        funcOne();
    } catch (const exception& e) {
        cerr << "Exception caught!" << endl;
        return 1;
    }
}

void funcOne()
{
    string str1;
    string* str2 = new string {};
    funcTwo();
    delete str2;
}

void funcTwo()
{
    ifstream fileStream;
    fileStream.open("filename");
}

```

```

    throw exception();
    fileStream.close();
}

```

当 funcTwo() 抛出一个异常时，最近的异常处理程序在 main() 中。控制立刻从 funcTwo() 的这一行：

```
throw exception();
```

跳转到 main() 的这一行：

```
cerr << "Exception caught!" << endl;
```

在 funcTwo() 中，控制依然在抛出异常的那一行，因此后面的行永远没机会执行。

```
fileStream.close();
```

然而幸运的是，因为 fileStream 是堆栈中的局部变量，所以会调用 ifstream 析构函数。ifstream 析构函数会自动关闭文件，因此在此不会泄漏资源。如果动态分配了 fileStream，那么这个指针不会销毁，文件也不会关闭。

在 funcOne() 中，控制在 funcTwo() 的调用中，因此后面的行永远没机会执行。

```
delete str2;
```

在此情况下，确实会发生内存泄漏。堆栈释放不会自动调用 str2 上的 delete，然而会正确地销毁 str1，因为 str1 是堆栈中的局部变量。堆栈释放会正确地销毁所有局部变量。

### 警告：

粗心的异常处理会导致内存和资源的泄漏。

不应将旧的 C 分配模式(即便使用了 new，看起来也像 C++ 模式)混入类似于异常的现代编程方法中，这就是原因之一。在 C++ 中，应该用基于堆栈的内存分配或者下面将要讨论的技术处理这种情况。

### 14.5.1 使用智能指针

如果基于堆栈的内存分配不可用，就应使用智能指针。在处理异常时，智能指针可使编写的代码自动防止内存或资源的泄漏。无论什么时候销毁智能指针对象，都会释放底层资源。下面使用智能指针 unique\_ptr(在<memory>中定义，如第 7 章“内存管理”章节所述)重写了前面的 funcOne() 函数：

```

void funcOne()
{
    string str1;
    auto str2 { make_unique<string>("hello") };
    funcTwo();
}

```

当从 funcOne() 返回或抛出异常时，将自动删除 str2 指针。

当然，只在确有必要时，才进行动态分配。例如，在前面的 funcOne() 函数中，没必要使 str2 成为动态分配的字符串。它应当只是一个基于堆栈的字符串变量。这里只将其作为一个仿造的例子，演示抛出异常的结果。

### 注意：

使用智能指针或其他 RAII 对象时，永远不必考虑释放底层的资源：RAII 对象的析构函数会自动完成这一操作，无论是正常退出函数，还是抛出异常退出函数，都是如此。这项技术会在第 32 章“设计技术与框架的组合”介绍。

### 14.5.2 捕获、清理并重新抛出

避免内存和资源泄漏的另一种技术是针对每个函数，捕获可能抛出的所有异常，执行必要的清理，并重新抛出异常，供堆栈中更高层的函数处理。下面是使用这一技术修改后的 funcOne() 函数：

```
void funcOne()
{
    string str1;
    string* str2 { new string() };
    try {
        funcTwo();
    } catch (...) {
        delete str2;
        throw; // Rethrow the exception.
    }
    delete str2;
}
```

这个函数用异常处理程序封装对 funcTwo() 函数的调用，处理程序执行清理（在 str2 上调用 delete）并重新抛出异常。关键字 throw 本身会重新抛出最近捕获的任何异常。注意 catch 语句使用... 语法捕获所有异常。

这一方法运行良好，但有点繁杂容易出错。需要特别注意，现在有两行完全相同的代码在 str2 上调用 delete：一行用来处理异常，另一行在函数正常退出的情况下执行。

#### 警告：

智能指针或其他 RAII 类是比捕获、清理和重新抛出技术更好的解决方案。

## 14.6 常见的错误处理问题

是否在程序中使用异常取决于程序员及其同事。然而无论是否使用异常，我们都强烈建议为程序提供正式的错误处理计划。如果使用异常，通常会比较容易地想出统一的错误处理计划，但不使用异常，也可能得到这样的计划。良好设计的最重要特征是所有程序模块的错误处理都是一致的。参与项目的所有程序员都应该理解并遵循这条错误处理规则。

本章讨论使用异常时最常见的错误处理问题，但不使用异常的程序也会涉及这些问题。

### 14.6.1 内存分配错误

尽管本书到目前为止的所有示例都忽略了这种可能，但内存分配确实可能失败。在当前的 64 位平台上，这几乎从不发生，但在移动或传统系统中，内存分配可能失败。在此类系统中，必须考虑内存分配失败的情形。C++ 提供了处理内存错误的多种不同方式。

如果无法分配内存，new 和 new[] 的默认行为是抛出 bad\_alloc 类型的异常，这种异常类型在 <new> 中定义。代码应该捕获并正确地处理这些异常。

不可能把对 new 和 new[] 的调用都放在 try/catch 块中，但至少在分配大块内存时应这么做。下例演示了如何捕获内存分配异常：

```
int* ptr { nullptr };
size_t integerCount { numeric_limits<size_t>::max() };
cout << format("Trying to allocate memory for {} integers.", integerCount) << endl;
```

```

try {
    ptr = new int[integerCount];
} catch (const bad_alloc& e) {
    auto location { source_location::current() };
    cerr << format("{}({}): Unable to allocate memory: {}", location.file_name(), location.line(), e.what()) << endl;
    // Handle memory allocation failure.
    return;
}
// Proceed with function that assumes memory has been allocated.

```

注意上面的代码使用了 `source_location` 将文件名和当前行号包含在错误信息中，从而使调试更容易。

当然，可用一个 `try/catch` 块在程序的高层批量处理许多可能的新错误，如果这样做对程序有效的话。

另一个考虑是记录错误时可能尝试分配内存。如果 `new` 执行失败，可能没有记录错误消息的足够内存。

### 1. 不抛出异常的 `new`

如果不喜欢异常，可回到旧的 C 模式；在这种模式下，如果无法分配内存，内存分配例程将返回一个空指针。`C++` 提供了 `new` 和 `new[]` 的 `nothrow` 版本，如果内存分配失败，将返回 `nullptr`，而不是抛出异常。使用语法 `new(nothrow)` 而不是 `new` 可做到这一点，如下所示。

```

int* ptr { new(nothrow) int[integerCount] };
if (ptr == nullptr) {
    auto location { source_location::current() };
    cerr << format("{}({}): Unable to allocate memory!", location.file_name(), location.line()) << endl;
    // Handle memory allocation failure.
    return;
}
// Proceed with function that assumes memory has been allocated.

```

#### 注意：

不建议使用不抛出异常的 `new`，而建议使用带有异常的默认的 `new`。分配失败时抛出的异常不能被忽略，而使用不抛出异常的 `new` 则很容易忘记对 `nullptr` 的检查。

### 2. 定制内存分配失败的行为

`C++` 允许指定 `new handler` 回调函数。默认情况下不存在 `new handler`，因此 `new` 和 `new[]` 只是抛出 `bad_alloc` 异常。然而如果存在 `new handler`，当内存分配失败时，内存分配例程会调用 `new handler` 而不是抛出异常。如果 `new handler` 返回，内存分配例程试着再次分配内存；如果失败，会再次调用 `new handler`。这个循环变成无限循环，除非 `new handler` 用下面的 3 个选项之一改变这种情况。下面列出这些选项，并给出了注释。

- **提供更多的可用内存** 提供空间的技巧之一是在程序启动时分配一大块内存，然后在 `new handler` 中释放这块内存。一个实际示例是，当遇到内存分配错误时，需要保存用户状态，这样就不会有工作丢失。关键在于，在程序启动时，分配一块足以完整保存文档的内存。当触发 `new handler` 时，可释放这块内存、保存文档、重启应用程序并重新加载保存的文档。

- **抛出异常** C++标准指出，如果 new handler 抛出异常，那么必须是 bad\_alloc 异常或者派生于 bad\_alloc 的异常。
- **编写和抛出 document\_recovery\_alloc 异常**，这种异常从 bad\_alloc 继承而来。可在应用程序的某个地方捕获这种异常，然后触发文档保存操作，并重启应用程序。
- **编写和抛出派生于 bad\_alloc 的 please\_terminate\_me 异常**。在顶层函数中，例如 main()，可捕获这种异常，并通过从顶层函数返回来对其进行处理。建议不要调用 exit()，而应从顶层函数返回以终止程序。
- **设置不同的 new handler**。从理论上讲，可使用一系列 new handler，每个都试图分配内存，并在失败时设置一个不同的 new handler。然而，这种情形通常过于复杂，并不实用。

如果在 new handler 中没有这么做，任何内存分配失败都会导致无限循环。

如果有一些内存分配会失败，但又不想调用 new handler，那么在调用 new 之前，只需要临时将新的 new handler 重新设置为默认值 nullptr。

调用在<new>中声明的 set\_new\_handler()，从而设置 new handler。下面是一个记录错误消息并抛出异常的 new handler 示例：

```
class please_terminate_me : public bad_alloc { };

void myNewHandler()
{
    cerr << "Unable to allocate memory." << endl;
    throw please_terminate_me();
}
```

new handler 不能有参数，也不能返回值。如前面列表中所述，new handler 抛出 please\_terminate\_me 异常。可采用以下方式设置 new handler：

```
int main()
{
    try {
        // Set the new new_handler and save the old one.
        new_handler oldHandler { set_new_handler(myNewHandler) };

        // Generate allocation error
        size_t numInts { numeric_limits<size_t>::max() };
        int* ptr { new int[numInts] };

        // Reset the old new_handler
        set_new_handler(oldHandler);
    } catch (const please_terminate_me&) {
        auto location { source_location::current() };
        cerr << format("{}({}): Terminating program.", location.file_name(), location.line()) << endl;
        return 1;
    }
}
```

new\_handler 是函数指针类型的类型别名，set\_new\_handler()会将其作为参数。

## 14.6.2 构造函数中的错误

在异常出现之前，C++程序员经常受到错误处理和构造函数的困扰。如果构造函数没有正确地创

建对象，会发生什么？构造函数没有返回值，因此在异常出现之前，标准的错误处理机制无法运行。如果不使用异常，最多可在对象中设置一个标记，指明对象没有正确构建。可提供一个类似于 `checkConstructionStatus()` 的方法，返回这个标志的值，并期望用户记得在构建对象之后调用这个方法。

异常提供了更好的解决方案。虽然无法在构造函数中返回值，但是可以抛出异常。通过异常可很方便地告诉客户是否成功创建了对象。然而在此有一个重要问题：如果异常离开构造函数，对象的析构函数将无法调用。因此在异常离开构造函数前，必须在构造函数中仔细清理所有资源，并释放分配的所有内存。其他函数也会遇到这个问题，但在构造函数中更微妙，因为我们习惯于让析构函数处理内存分配和释放资源。

本节以 `Matrix` 类作为示例，这个类的构造函数可正确处理异常。注意这个示例使用裸指针 `m_matrix` 来演示问题。在生产级代码中，应避免使用裸指针，而使用标准库容器！`Matrix` 类模板的定义如下所示：

```
export template <typename T>
class Matrix
{
public:
    Matrix(size_t width, size_t height);
    virtual ~Matrix();
private:
    void cleanup();

    size_t m_width { 0 };
    size_t m_height { 0 };
    T** m_matrix { nullptr };
};
```

`Matrix` 类的实现如下所示。注意对 `new` 的第一个调用并没有用 `try/catch` 块保护。第一个 `new` 抛出异常也没有关系，因为构造函数此时还没有分配任何需要释放的内存。如果后面的 `new` 抛出异常，构造函数必须清理所有已经分配的内存。然而，由于不知道 `T` 构造函数本身会抛出什么异常，因此通过“...”捕获所有异常，并将捕获的异常嵌套在 `bad_alloc` 异常中。注意，使用 {} 语法，通过首次调用 `new` 分配的数组执行零初始化，即每个元素都是 `nullptr`。这简化了 `cleanup()` 方法，因为它允许它在 `nullptr` 上调用 `delete`。

```
template <typename T>
Matrix<T>::Matrix(size_t width, size_t height)
{
    m_matrix = new T*[width] {};
    // Array is zero-initialized!

    // Don't initialize the mWidth and mHeight members in the ctor-
    // initializer. These should only be initialized when the above
    // mMatrix allocation succeeds!
    m_width = width;
    m_height = height;

    try {
        for (size_t i { 0 }; i < width; ++i) {
            m_matrix[i] = new T[height];
        }
    } catch (...) {
        std::cerr << "Exception caught in constructor, cleaning up..." 
        << std::endl;
        cleanup();
    }
}
```

```

    // Nest any caught exception inside a bad_alloc exception.
    std::throw_with_nested(std::bad_alloc());
}

template <typename T>
Matrix<T>::~Matrix()
{
    cleanup();
}

template <typename T>
void Matrix<T>::cleanup()
{
    for (size_t i { 0 }; i < m_width; ++i)
        delete[] m_matrix[i];
    delete[] m_matrix;
    m_matrix = nullptr;
    m_width = m_height = 0;
}

```

**警告:**

记住, 如果异常离开了构造函数, 将永远不会调用对象的析构函数!

可采用如下方式测试 Matrix 类模板:

```

class Element
{
    // Kept to a bare minimum, but in practice, this Element class
    // could throw exceptions in its constructor.
private:
    int m_value;
};

int main()
{
    Matrix<Element> m { 10, 10 };
}

```

如果使用了继承, 会发生什么情况? 那样的话, 基类的构造函数在派生类的构造函数之前运行, 如果派生类的构造函数抛出一个异常, 那么 C++会运行任何构建完整的基类的析构函数。

**注意:**

C++保证会运行任何构建完整的“子对象”的析构函数。因此, 任何没有发生异常的构造函数所对应的析构函数都会运行。

### 14.6.3 构造函数的function-try-blocks

本章到目前为止讨论的异常机制可完美处理函数内的异常。然而, 如果在构造函数的 ctor-initializer 中抛出异常, 该如何处理呢? 本节介绍能捕获这类异常的 function-try-blocks。function-try-blocks 用于普通函数和构造函数。本节重点介绍 function-try-blocks 如何用于构造函数。大多数 C++程序员, 甚至包括经验丰富的 C++程序员, 都不了解这一特性的存在, 尽管这一特性已经问

世很多年了。

下面的伪代码显示了构造函数的 function-try-blocks 的基本语法:

```
MyClass::MyClass()
try
    : <ctor-initializer>
{
    /* ... constructor body ... */
}
catch (const exception& e)
{
    /* ... */
}
```

可以看出, try 关键字应该刚好在 ctor-initializer 之前。catch 语句应该在构造函数的右花括号之后, 实际上是将 catch 语句放在构造函数体的外部。当使用构造函数的 function-try-blocks 时, 要记住如下限制和指导方针:

- catch 语句将捕获任何异常, 无论是构造函数体还是 ctor-initializer 直接或间接抛出的异常, 都是如此。
- catch 语句必须重新抛出当前异常或抛出一个新异常。如果 catch 语句没有这么做, 运行时将自动重新抛出当前异常。
- catch 语句可访问传递给构造函数的参数。
- 当 catch 语句捕获 function-try-blocks 内的异常时, 构造函数已构建的所有对象都会在执行 catch 语句之前销毁。
- 在 catch 语句中, 不应访问对象成员变量, 因为它们在执行 catch 语句前就销毁了(参见上一条)。但是, 如果对象包含非类数据成员, 例如裸指针, 并且它们在抛出异常之前初始化, 就可以访问它们。如果有这样的裸资源, 就必须在 catch 语句中释放它们。
- 对于 function-try-blocks 中的 catch 语句而言, 其中包含的函数不能使用 return 关键字返回值。构造函数与此无关, 因为构造函数没有返回值。

由于有以上限制, 构造函数的 function-try-blocks 只在少数情况下有用:

- 将 ctor-initializer 抛出的异常转换为其他异常。
- 将消息记录到日志文件。
- 释放在抛出异常之前就在 ctor-initializer 中分配了内存的裸资源。

下例演示 function-try-blocks 的用法。下面的代码定义了一个 SubObject 类, 这个类只有一个构造函数, 这个构造函数抛出一个 runtime\_error 类型的异常。

```
class SubObject
{
public:
    SubObject(int i) { throw std::runtime_error("Exception by SubObject ctor"); }
};
```

接下来, MyClass 类有一个 int\*类型的成员变量以及一个 SubObject 类型的成员变量:

```
class MyClass
{
public:
    MyClass();
private:
```

```

    int* m_data { nullptr };
    SubObject m_subObject;
};

}

```

SubObject 类没有默认构造函数。这意味着需要在 MyClass 类的 ctor-initializer 中初始化 m\_subObject。MyClass 类的构造函数将使用 function-try-block 捕获 ctor-initializer 中抛出的异常。

```

MyClass::MyClass()
try
: m_data { new int[42]{ 1, 2, 3 } }, m_subObject { 42 }
{
    /* ... constructor body ... */
}
catch (const exception& e)
{
    // Cleanup memory.
    delete[] m_data;
    m_data = nullptr;
    cout << format("function-try-block caught: '{}'", e.what()) << endl;
}

```

记住，构造函数的 function-try-block 中的 catch 语句必须重新抛出当前异常，或者抛出新异常。前面的 catch 语句没有抛出任何异常，因此 C++ 运行时将自动重新抛出当前异常。下面的简单函数使用了 MyClass 类：

```

int main()
{
    try {
        MyClass m;
    } catch (const std::exception& e) {
        cout << format("main() caught: '{}'", e.what()) << endl;
    }
}

```

输出如下所示：

```

function-try-block caught: 'Exception by SubObject ctor'
main() caught: 'Exception by SubObject ctor'

```

注意，该例中的代码很容易出错，而且不推荐使用。更合适的方案是为 m\_data 成员使用智能指针(如 std::unique\_ptr)以删除 function-try-block。

### 警告：

避免使用 function-try-block！

通常，仅将裸资源作为数据成员时，才有必要使用 function-try-block。可使用诸如 std::unique\_ptr 的 RAII 类来避免使用裸资源。第 32 章将讨论 RAII 类。

function-try-block 并不局限于构造函数，也可用于普通函数。然而，对于普通函数而言，没理由使用 function-try-block，因为可方便地将 function-try-block 转换为函数体内部的简单 try/catch 块。与构造函数相比，对普通函数使用 function-try-block 的明显不同在于，catch 语句不需要重新抛出当前异常或新异常，C++ 运行时也不会自动重新抛出异常。

#### 14.6.4 析构函数中的错误

必须在析构函数内部处理析构函数引起的所有错误。不应该让析构函数抛出任何异常，原因如下：

(1) 析构函数会被隐式标记为 `noexcept`，除非添加了 `noexcept(false)` 标记，或者类具有子对象，而子对象的析构函数是 `noexcept(false)`。如果带 `noexcept` 标记的析构函数抛出一个异常，C++运行时会调用 `std::terminate()` 来终止应用程序。本书不详细讨论 `noexcept(expression)` 说明符。只需要知道 `noexcept` 等同于 `noexcept(true)`，而 `noexcept(false)` 与 `noexcept(true)` 相反；也就是说，标记了 `noexcept(false)` 的方法可抛出任何需要的异常。

(2) 在堆栈释放过程中，在处理另一个异常时，析构函数可以运行。如果在堆栈释放期间从析构函数抛出一个异常，C++运行时会调用 `std::terminate()` 来终止应用程序。C++确实为勇敢而好奇的人提供了一种功能，用于判断析构函数是因为正常的函数退出或调用了 `delete` 而执行，还是因为堆栈释放而执行。`<exception>` 中声明了一个函数 `uncaught_exception()`，该函数可返回未捕获异常的数量；所谓未捕获异常，是指已经抛出但尚未到达匹配 `catch` 的异常。如果 `uncaught_exceptions()` 的结果大于 0，则说明正在执行堆栈释放。但这个函数用起来十分复杂，应避免使用。注意在 C++17 之前，该函数名为 `uncaught_exception()`，即 `exception` 之后不带 `s`，如果正在执行堆栈释放，该函数返回布尔值 `true`。这个单一版本在 C++17 中被废弃，并在 C++20 中被移除。

(3) 客户应该采取什么措施？客户不会显式调用析构函数：客户调用 `delete`, `delete` 调用析构函数。如果在析构函数中抛出一个异常，让客户怎么办？客户无法在此使用对象调用 `delete`，也不能显式地调用析构函数。客户无法采取合理行动，因此这里没理由让代码处理异常。

(4) 析构函数是释放对象使用的内存和资源的一个机会。如果因为异常而提前退出这个函数，就会浪费这个机会，将永远无法回头释放内存或资源。

**警告：**

小心不要让任何异常逃离开析构函数。

### 14.7 本章小结

本章讲述了 C++ 程序中与错误处理相关的主题，并强调在设计和编写程序时必须有错误处理方案。通过阅读本章，可详细了解 C++ 异常语法和行为。本章还讲述了错误处理扮演着重要角色的一些领域，包括 I/O 流、内存分配、构造函数和析构函数。

### 14.8 练习

通过完成下面的习题，可以巩固本章所讨论的知识点。所有习题的答案都可扫描封底二维码下载。然而如果你受困于某个习题，可以在看网站上的答案之前，先重新阅读本章的部分内容，尝试着自己找到答案。

**练习 14-1** 在不编译和执行的情况下，找到并纠正下面代码中的错误。

```
// Throws a logic_error exception if the number of elements
// in the given dataset is not even.
void verifyDataSet(const vector<int>& data)
{
    if (data.size() % 2 != 0)
        ,
}
```

```

        throw logic_error { "Number of data points must be even." };
    }

    // Throws a logic_error exception if the number of elements
    // in the given dataset is not even.
    // Throws a domain_error exception if one of the datapoints is negative.
    void processData(const vector<int>& data)
    {
        // Verify the size of the given dataset.
        try {
            verifyDataSize(data);
        } catch (const logic_error& caughtException) {
            // Write message on standard error output.
            cerr << "Invalid number of data points in dataset. Aborting." << endl;
            // And rethrow the exception.
            throw caughtException;
        }
        // Verify for negative datapoints.
        for (auto& value : data) {
            if (value < 0)
                throw domain_error { "Negative datapoints not allowed." };
        }
        // Process data ...
    }

    int main()
    {
        try {
            vector data { 1, 2, 3, -5, 6, 9 };
            processData(data);
        } catch (const logic_error& caughtException) {
            cerr << "logic_error: " << caughtException.what() << endl;
        } catch (const domain_error& caughtException) {
            cerr << "domain_error: " << caughtException.what() << endl;
        }
    }
}

```

**练习 14-2** 拿到第 13 章中双向 I/O 例子中的代码(可在源代码文件中的 c13\_code\19\_Bidirectional 目录中找到)。示例中实现了 changeNumberForID() 函数。修改代码, 以便在合适的地方使用异常。一旦代码使用了异常, 是否可以对 changeNumberForID() 函数的头文件进行更改?

**练习 14-3** 将使用异常的正确错误处理添加到练习 13-3 的个人数据库解决方案中。

**练习 14-4** 请看第 9 章中 SpreadSheet 的示例代码(可在源代码文件中的 c09\_code\07\_SpreadsheetMoveSemanticsWithSwap 目录中找到), 该示例使用 swap() 支持移动语义。在代码中添加适当的错误处理, 包括内存分配失败的处理。为类添加最大宽高, 并添加恰当的验证检查。编写自己的异常类 InvalidCoordinate, 存储无效坐标和有效坐标的范围, 在 verifyCoordinate() 方法中使用。在 main() 方法中写两个测试程序来测试各种错误条件。

# 第 15 章

## C++ 运算符重载

### 本章内容

- 运算符重载的含义
- 运算符重载的合理性
- 运算符重载的限制、注意事项和选择
- 可以重载、不可以重载和不应该重载的运算符小结
- 如何重载一元加运算符、一元减运算符、递增和递减运算符
- 如何重载 I/O 流运算符(operator<< 和 operator>>)运算符
- 如何重载下标(数组索引)运算符
- 如何重载函数调用运算符
- 如何重载解除引用运算符(\*和->)
- 如何编写转换运算符
- 如何重载内存分配和释放运算符
- 如何定义自己的用户定义的字面量运算符
- 可用的标准用户定义的字面量运算符

C++ 允许为自己的类重定义运算符(如+、-和=)的含义。由于很多面向对象的语言没有提供这种能力,因此你可能会低估这种特性在 C++ 中的作用。然而,能让自己的类具有内建类型(例如 int 和 double)的类似行为是有益的,甚至可编写看上去类似于数组、函数或指针的类。

第 5 章“面向对象设计”和第 6 章“设计可重用代码”分别介绍了面向对象的设计和运算符重载。第 8 章“熟悉类和对象”和第 9 章“精通类和对象”介绍了对象和基本运算符重载的语法细节。本章讲解第 9 章没有涉及的有关运算符重载的内容。

### 15.1 运算符重载概述

根据第 1 章的描述,C++ 中的运算符是一些类似于+、<、\*和<<的符号。这些运算符可应用于内建类型,例如 int 和 double,从而实现算术操作、逻辑操作和其他操作。还有->和\*运算符可对指针进行解除引用操作。C++ 中运算符的概念十分广泛,甚至包含[](数组索引)、()函数调用)、类型转换以

及内存分配和内存释放例程。可通过运算符重载来改变语言运算符对自定义类的行为。然而，使用这项功能时需要符合规则，满足限制条件，还需要做出选择。

### 15.1.1 重载运算符的原因

在学习重载运算符前，首先需要了解为什么需要重载运算符。不同的运算符有不同的理由，但是基本指导原则是为了让自定义类的行为和内建类型一样。自定义类的行为越接近内建类型，就越便于这些类的客户使用。例如，如果要编写一个表示分数的类，最好定义+、-、\*和/运算符在应用于这个类的对象时的意义。

重载运算符的另一个原因是获得对程序行为更大的控制权。例如，可对自定义类重载内存分配和内存释放例程，以精确控制每个新对象的内存分配和内存回收。

需要强调的一点是，运算符重载未必能给类的开发者带来便利；主要用途是给类的客户带来便利。

### 15.1.2 运算符重载的限制

下面列出了重载运算符时不能做的事情：

- 不能添加新的运算符。只能重定义语言中已经存在的运算符的意义。表 15-1 列出了所有可重载的运算符。
- 有少数运算符不能重载，例如.\*(对象成员访问运算符)、::(作用域解析运算符)和?:(条件运算符)以及其他几个运算符。表 15-1 列出了所有可重载的运算符。不能重载的运算符通常不需要重载的，因此这些限制应该不会令人感到受限。
- arity 描述了运算符关联的参数或操作数的数量。只能修改函数调用、new 和 delete 运算符的 arity。其他运算符的 arity 不能修改。一元运算符，例如++，只能用于一个操作数。二元运算符，例如/，只能用于两个操作数。这条限制会带来麻烦的主要情形是[](数组索引)的重载，详见本章后面的讨论。
- 不能修改运算符的优先级和结合性。优先级用于决定哪些操作符需要在其他操作符之前执行，而结合性可以是从左到右或从右到左，并指定具有相同优先级的运算符的执行顺序。同样，这条限制对于大多数程序来说不是问题，因为改变求值顺序并不会带来什么好处。
- 不能对内建类型重定义运算符。运算符必须是类中的一个方法，或者全局重载运算符函数至少有一个参数必须是一种用户定义的类型(例如一个类)。这意味着不允许做一些荒唐的事情，例如将 int 类型的+运算符重定义为减法(尽管自定义的类可以这么做)。这条限制有一个例外，那就是内存分配和内存释放运算符；可替换程序中所有内存分配使用的全局运算符。

有些运算符已经有两种不同的含义。例如，-运算符可用作二元运算符，例如  $x = y - z$ ，还可用作一元运算符，如  $x = -y$ 。\*运算符可用作乘法操作，也可以用于解除指针的引用。根据上下文的不同，<<可以是插入运算符，也可以是左移运算符。可以重载具有双重意义的运算符。

### 15.1.3 运算符重载的选择

重载运算符时，需要编写名为 operatorX 的函数或方法，X 是表示这个运算符的符号，可以在 operator 和 X 之间添加空白字符。例如，第 9 章声明了 SpreadsheetCell 对象的 operator+ 运算符，如下所示：

```
SpreadsheetCell operator+(const SpreadsheetCell& lhs, const SpreadsheetCell& rhs);
```

下面描述了编写每个重载运算符函数或方法时需要做出的选择。

### 1. 方法还是全局函数

首先，要决定运算符应该实现为类的方法还是全局函数(通常是类的友元)。如何做出选择？你需要理解这两个选择之间的区别。当运算符是类的方法时，运算符表达式的左侧必须是这个类的对象。当编写全局函数时，运算符表达式的左侧可以是不同类型的对象。

有 3 种不同类型的运算符，如下所示。

- **必须为方法的运算符：**C++ 语言要求一些运算符必须是类中的方法，因为这些运算符在类的外部没有意义。例如，operator= 和类绑定得非常紧密，不能出现在其他地方。表 15-1 列出了所有必须为方法的运算符。大部分运算符都没有施加这种要求。
- **必须为全局函数的运算符：**如果允许运算符左侧的变量是除了自定义的类之外的任何类型，那么必须将这个运算符定义为全局函数。确切地讲，这条规则适用于 operator<< 和 operator>>，这两个运算符的左侧是 ostream 对象，而不是自定义类的对象。此外，可交换的运算符(例如二元的 + 和 -)允许运算符左侧的变量不是自定义类的对象。第 9 章曾提及这个问题。
- **既可为方法又可为全局函数的运算符：**有关编写方法重载运算符更好还是编写全局函数重载运算符更好的问题在 C++ 社区中有一些争议。不过建议遵循如下规则：把所有运算符都定义为方法，除非根据以上描述必须定义为全局函数。这条规则的一个主要优点是，方法可以是虚方法，但全局函数不能是虚函数。因此，如果准备在继承树中编写重载的运算符，应尽可能将这些运算符定义为方法。

将重载的运算符定义为方法时，如果这个运算符不修改对象，应将整个方法标记为 const。这样，就可对 const 对象调用这个方法。

将重载的运算符定义为全局函数时，将其放在同样包含编写运算符的类的名称空间中。

### 2. 选择参数类型

参数类型的选择有一些限制，因为如前所述，大多数运算符不能修改参数数量。例如，operator/ 在作为全局函数的情况下必须总是接收两个参数；在作为类方法的情况下必须总是接收一个参数。如果不遵循这条规则，编译器会生成错误。从这个角度看，运算符函数和普通函数有区别，普通函数可使用任意数量的参数重载。此外，尽管可编写接收任何类型参数的运算符，但可选范围通常都受这个运算符所在的类的限制。例如，如果要为类 T 实现加法操作，就不能编写接收两个字符串的 operator+。真正需要选择的地方在于判断是按值还是按引用接收参数，以及是否需要把参数标记为 const。

按值传递还是按引用传递的决策很简单：除非函数需要传入对象的一份拷贝，否则应按引用接收每一个非基本类型的参数，详见第 9 章。

const 决策也很简单：除非要真正修改参数，否则将每个参数都设置为 const。表 15-1 列出了所有运算符的示例原型，并根据需要将参数标记为 const 和引用。

### 3. 选择返回类型

C++ 不根据返回类型来解析重载。因此，在编写重载运算符时，可指定任意返回类型。然而，可做某件事并不意味着应该做这件事。这种灵活性可能导致令人迷惑的代码，例如比较运算符返回指针，算术运算符返回 bool 类型。不应该编写这样的代码。实际上，在编写重载运算符时，应该让运算符返回的类型和运算符对内建类型操作时返回的类型一样。如果编写比较运算符，那么应该返回 bool 类型。如果编写算术运算符，那么应该返回表示运算结果的对象。有时，返回类型初看上去并不明显。例如根据第 8 章的描述，operator= 应该返回一个对调用这个运算符的对象的引用，以支持嵌套的赋值。其他运算符也有类似的不容易理解的返回类型，所有这些内容都在表 15-1 中列出。

引用和 `const` 决策也适用于返回类型。不过对返回值来说，这些决策要更困难一些。值还是引用的一般原则是：如果可以，就返回一个引用，否则返回一个值。如何判断何时能返回引用？这个决策只能应用于返回对象的运算符：对于返回布尔值的比较运算符、没有返回类型的转换运算符和函数调用运算符(可能返回所需的任何类型)来说，这个决策没有意义。如果运算符构造了一个新对象，那么必须按值返回这个新对象。如果不构造新对象，可返回对调用这个运算符的对象的引用，或返回其中一个参数的引用。表 15-1 给出了示例。

可作为左值(赋值表达式左侧的部分)修改的返回值必须是非 `const`。否则，这个值应该是 `const`。大部分很容易想到的运算符都要求返回左值，包括所有赋值运算符(`operator=`、`operator+=`和 `operator-=`等)。

#### 4. 选择行为

在重载的运算符中，可提供任意需要的实现。例如，可编写一个启动 Scrabble 拼字游戏的 `operator+`。然而根据第 6 章的描述，通常情况下，应将实现约束为客户期待的行为。编写 `operator+` 时，使这个运算符能执行加法或其他类似加法的操作，例如字符串串联。

这一章讲解应该如何实现重载的运算符。在特殊情况下，可以不采用这些建议，但一般情况下都应该遵循这些标准模式。

### 15.1.4 不应重载的运算符

有些运算符即使允许重载，也不应该重载。具体来说，取地址运算符(`operator&`)的重载一般没什么特别的用途，如果重载会导致混乱，因为这样做会以可能异常的方式修改基础语言的行为(获得变量的地址)。整个标准库大量使用了运算符重载，但从没有重载取地址运算符。

此外，还要避免重载二元布尔运算符 `operator&&` 和 `operator||`，因为这样会使 C++ 的短路求值规则失效。

最后，不要重载逗号运算符。没错，C++ 中确实有一个逗号运算符，也称为序列运算符(*sequencing operator*)，用于分隔一条语句中的两个表达式，确保求值顺序从左至右。下面的代码片段演示了逗号运算符：

```
int x { 1 };
cout << (++x, 2 * x); // Sets x to 2, and prints 4.
```

几乎没有什么正当理由需要重载这个运算符。

### 15.1.5 可重载运算符小结

表 15-1 列出了所有可重载的运算符，标明了运算符应该是类的方法还是全局函数，总结了什么时候应该(或不应该)重载，并提供了示例原型，展示了正确的返回值。某些不能重载的运算符，例如 `.运算符`、`.*运算符`、`::运算符` 和 `?::运算符`，不在此列表内。

如果需要编写重载运算符，表 15-1 是很好的参考资源。你肯定会忘了应该使用哪种返回类型，以及应该使用函数还是方法。

在表 15-1 中，T 表示要编写重载运算符的类名，E 是一种不同的类型。注意给出的示例原型并不全面，给定的运算符常常可能有 T 和 E 的其他组合。

表 15-1 可重载的运算符

运算符	名称或类别	方法还是全局函数	何时重载	示例原型
operator+	二元算术运算符	建议使用全局函数	类需要提供这些操作时	T operator+(const T&, const T&); T operator+(const T&, const E&);
operator-				
operator*				
operator/				
operator%				
operator~	一元算术运算符和按位运算符	建议使用方法	类需要提供这些操作时	T operator~() const;
operator++	前缀递增运算符和前缀递减运算符	建议使用方法	重载了 += 和 -= 时(使用算术参数, 如 int 和 long)	T& operator++();
operator--				
operator++	后缀递增运算符和后缀递减运算符	建议使用方法	重载了 += 和 -= 时(使用算术参数, 如 int 和 long)	T operator++(int);
operator--				
operator=	赋值运算符	必须使用方法	在类中动态分配了内存或资源, 或者成员是引用时	T& operator=(const T&);
operator+=	算术运算符赋值的简写	建议使用方法	重载了二元算术运算符, 并且类没有设计为不可变时	T& operator+=(const T&); T& operator+=(const E&);
operator-=				
operator*/				
operator/=				
operator%=/				
operator<<	二元按位运算符	建议使用全局函数	需要提供这些操作时	T operator<<(const T&, const T&); T operator<<(const T&, const E&);
operator>>				
operator&				
operator				
operator^				
operator<<=	按位运算符赋值的简写	建议使用方法	重载了二元按位运算符, 并且类没有设计为不可变时	T& operator<<=(const T&); T& operator<<=(const E&);
operator>>=				
operator&=				
operator =				
operator^=				
operator<=>	三元比较运算符	建议使用方法	需要为类提供比较操作时	auto operator<=>(const T&) const; partial_ordering operator<=>(const E&) const;
operator==	二元相等运算符	C++20: 建议使用方法 C++20 前: 建议使用全局函数	需要为类提供比较操作时	bool operator==(const T&) const; bool operator==(const E&) const; bool operator==(const T&, const T&); bool operator==(const T&, const E&);

(续表)

运算符	名称或类别	方法还是全局函数	何时重载	示例原型
operator!=	二元不等运算符	C++20: 建议使用方法 C++20 前: 建议使用全局函数	C++20: 当提供了==运算符且不需要编译器自动提供!=运算符时 C++20 前: 需要为类提供比较操作时	bool operator!= (const T&) const; bool operator!= (const E&) const; bool operator!= (const T&, const T&); bool operator!= (const T&, const E&);
operator<	二元比较运算符	建议使用全局函数	需要提供这些操作时	bool operator<(const T&, const T&); bool operator<(const T&, const E&);
operator>				
operator<=				
operator>=				
operator<<	I/O 流运算符(插入操作和提取操作)	建议使用全局函数	需要提供这些操作时	ostream& operator<<(ostream&, const T&); istream& operator>>(istream&, T&);
operator!	布尔非运算符	建议使用方法	很少重载; 应该改用 bool 或 void* 类型转换	bool operator!() const;
operator&&	二元布尔运算符	建议使用全局函数	很少重载, 否则会失去短路能力。更好的做法是重载&和  , 因为它们从不出现短路	bool operator&&(const T&, const T&);
operator				
operator[]	下标(数组索引)运算符	必须使用方法	需要支持下标访问时	E& operator[](size_t); const E& operator[](size_t) const;
operator()	函数调用运算符	必须使用方法	需要让对象的行为和函数指针一致时; 或者是多维数组访问, 因为[]只能有一个索引	返回类型和参数可以多种多样, 参见本章的示例
operator type()	转换(或强制类型转换)运算符(每种类型有不同的运算符)	必须使用方法	需要将自己编写的类转换为其他类型时	operator type() const;
operator ""_x	用户定义字面量运算符	必须使用全局函数	需要支持用户定义的字面量时	T operator""_i(long double d);
operator new	内存分配例程	建议使用方法	需要控制类的内存分配时(很少见)	void* operator new(size_t size); void* operator new[](size_t size);
operator new[]				

(续表)

运算符	名称或类别	方法还是全局函数	何时重载	示例原型
operator delete	内存释放例程	建议使用方法	重载了内存分配例程时 (很少见)	void operator delete(void* ptr) noexcept; void operator delete[](void* ptr) noexcept;
operator delete[]				
operator*	解除引用运算符	对于 operator*, 建议使用方法; 对于 operator->, 必须使用方法	适用于智能指针	E& operator*() const; E* operator->() const;
operator&	取地址运算符	不可用	永远不要	不可用
operator->*	解除引用指针-成员	不可用	永远不要	不可用
operator,	逗号运算符	不可用	永远不要	不可用

### 15.1.6 右值引用

第 9 章讨论了移动语义和右值引用。通过定义移动赋值运算符演示了这些概念，当源对象是一个将会在赋值操作后销毁的临时对象，或是一个使用了 `std::move()` 显式移动的对象时，编译器会使用移动赋值运算符。表 15-1 列出的普通赋值运算符的原型如下所示：

```
T& operator=(const T&);
```

移动赋值运算符的原型几乎一致，但使用了右值引用。这个运算符会修改参数，因此不能传递 `const` 参数。第 9 章详细讨论了这些内容：

```
T& operator=(T&&);
```

表 15-1 没有包含右值引用语义的示例原型。然而，对于大部分运算符来说，编写一个使用普通左值引用的版本以及一个使用右值引用的版本都是有意义的，但是否真正有意义取决于类的实现细节。`operator=` 是第 9 章的一个例子。另一个例子是通过 `operator+=` 避免不必要的内存分配。例如标准库中的 `std::string` 类利用右值引用实现了 `operator+=`，如下所示(已简化)：

```
string operator+(string&& lhs, string&& rhs);
```

这个运算符的实现会重用其中一个参数的内存，因为这些参数是以右值引用传递的。也就是说，这两个参数表示的都是 `operator+=` 完成之后销毁的临时对象。上述 `operator+=` 的实现具有以下效果(具体取决于两个操作数的大小和容量)：

```
return move(lhs.append(rhs));
```

或

```
return move(rhs.insert(0, lhs));
```

事实上，`string` 定义了几个 `operator+=` 重载，接收两个 `string` 作为参数，带有不同的左值和右值引用的组合，如下(已经简化)：

```
string operator+(const string& lhs, const string& rhs); // No memory reuse.  
string operator+(string&& lhs, const string& rhs); // Can reuse memory of lhs.  
string operator+(const string& lhs, string&& rhs); // Can reuse memory of rhs.  
string operator+(string&& lhs, string&& rhs); // Can reuse memory of lhs or rhs.
```

重用其中一个右值引用参数的内存的实现方式和第 9 章中讲解的移动赋值运算符一致。

### 15.1.7 优先级和结合性

在包含多个运算符的语句中，运算符的优先级用于决定哪个运算需要在其他运算之前执行。例如，\*和/运算总是在+和-运算前执行。

结合性可以是从左到右或从右到左，可指定具有相同优先级的运算符的执行顺序。

表 15-2 列出了所有可用的 C++ 运算符，包括不能重载的运算符，并列出了它们的优先级和结合性。低优先级的运算在较高优先级的运算之前执行。在表 15-2 中，T 表示类型，x、y、z 表示对象。

表 15-2 可用的 C++ 运算符

优先级	运算符	结合性
1	::	从左到右
2	x++ x-- x() x[] T()	从左到右
	T{} . ->	
3	+x -x +x -x ! ~	从右到左
	*x &x (T)	
	sizeof co_await new delete	
	new[] delete[]	
4	* ->*	从左到右
5	x*y x/y x%y	从左到右
6	x+y x-y	从左到右
7	<< >>	从左到右
8	<=>	从左到右
9	< <= > >=	从左到右
10	= !=	从左到右
11	&	从左到右
12	^	从左到右
13		从左到右
14	&&	从左到右
15		从左到右
16	x?y:z throw co_yield	从右到左
	= += -= *= /= %= <=> &=	
	^=  =	
17	,	从左到右

### 15.1.8 关系运算符

在 std::rel\_ops 名称空间的<utility>中，有下面关系运算符的函数模板定义：

```
template<class T> bool operator!=(const T& a, const T& b); // Needs operator==
template<class T> bool operator>(const T& a, const T& b); // Needs operator<
template<class T> bool operator<=(const T& a, const T& b); // Needs operator<
template<class T> bool operator>=(const T& a, const T& b); // Needs operator<
```

这些函数模板根据==和<运算符给任意类定义了运算符!=、>、<=和>=。如果在类中实现operator==和operator<，就会通过这些模板自动获得其他关系运算符。

然而这项技术存在大量问题。第一个问题是，这些运算符可能是为关系运算中使用的所有类创建的，而不仅仅是自己的类。

另一个问题是诸如 std::greater<T> 的实用工具模板不能用于这些自动生成的关系运算符，实用工具模板见第 19 章的讨论。

还有一个问题是隐式转换不可行。

最后，有了 C++20 的三元比较运算符和 C++20 已经弃用了的 std::rel\_ops 名称空间，就再也没有理由继续使用它了。通过一行代码显式地默认使用三元比较运算符<=>，类自动得到所有 6 个比较运算符的支持。详见第 9 章。

#### 警告：

不要使用 std::rel\_ops，相反只需要显式地为类声明默认或实现<=>运算符。

## 15.2 重载算术运算符

第 9 章讲解了如何编写二元算术运算符和简写的算术赋值运算符，但没有讲解如何重载其他算术运算符。

### 15.2.1 重载一元负号和一元正号运算符

C++有几个一元算术运算符。一元负号和一元正号运算符是其中的两个。下面列出一些使用整数的运算符示例：

```
int i, j { 4 };
i = -j;      // Unary minus
i = +i;      // Unary plus
j = +(-i);   // Apply unary plus to the result of applying unary minus to i.
j = -(-i);   // Apply unary minus to the result of applying unary minus to i.
```

一元负号运算符对其操作数取反，而一元正号运算符直接返回操作数。注意，可以对一元正号或一元负号运算符生成的结果应用一元正号或一元负号运算符。这些运算符不改变调用它们的对象，所以应把它们标记为 const。

下例将一元 operator- 运算符重载为 SpreadsheetCell 类的成员函数。一元正号运算符通常执行恒等运算，因此这个类没有重载这个运算符。

```
SpreadsheetCell SpreadsheetCell::operator-() const
{
    return SpreadsheetCell { -getValue() };
}
```

operator- 没有修改操作数，因此这个方法必须构造一个新的带有相反值的 SpreadsheetCell 对象，并返回这个对象的副本。因此，这个运算符不能返回引用。可按以下方式使用这个运算符：

```
SpreadsheetCell c1 { 4 };
SpreadsheetCell c3 { -c1 };
```

### 15.2.2 重载递增和递减运算符

可采用 4 种方法给变量加 1：

```
i = i + 1;
i += 1;
++i;
i++;
```

后两种称为递增运算符。第一种形式是前缀递增，这个操作将变量增加 1，然后返回增加后的newValue，供表达式的其他部分使用。第二种形式是后缀递增，它也给变量加 1，但返回旧值(非自增的值)，供表达式的其他部分使用。递减运算符的功能类似。

`operator++` 和 `operator--` 的双重意义(前缀和后缀)给重载带来了问题。例如，编写重载的 `operator++` 时，怎样表示重载的是前缀版本还是后缀版本？C++ 引入了一种方法来区分：前缀版本的 `operator++` 和 `operator--` 不接收参数，而后缀版本的接收一个不使用的 int 类型参数。

如果要为 `SpreadsheetCell` 类重载这些运算符，原型如下所示：

```
SpreadsheetCell& operator++();      // Prefix
SpreadsheetCell operator++(int);    // Postfix
SpreadsheetCell& operator--();      // Prefix
SpreadsheetCell operator--(int);    // Postfix
```

前缀形式的结果值和操作数的最终值一致，因此前缀递增和前缀递减返回被调用对象的引用。而后缀版本的递增操作和递减操作返回的结果值和操作数的最终值不同，因此不能返回引用。

下面是 `operator++` 运算符的实现：

```
SpreadsheetCell& SpreadsheetCell::operator++()
{
    set(getValue() + 1);
    return *this;
}

SpreadsheetCell SpreadsheetCell::operator++(int)
{
    auto oldCell(*this); // Save current value
    ++(*this);          // Increment using prefix ++
    return oldCell;     // Return the old value
}
```

`operator--` 的实现与此几乎相同。现在可随意递增和递减 `SpreadsheetCell` 对象了：

```
SpreadsheetCell c1 { 4 };
SpreadsheetCell c2 { 4 };
c1++;
++c2;
```

递增和递减还能应用于指针。当编写的类是智能指针或迭代器时，可重载 `operator++` 和 `operator--`，以提供指针的递增和递减操作。

## 15.3 重载按位运算符和二元逻辑运算符

按位运算符和算术运算符类似，简写的按位赋值运算符也和简写的算术赋值运算符类似。不过由于这些运算符明显少见得多，因此这里不提供示例了。表 15-1 展示了示例原型，如有必要，应该可

以很容易地实现这些运算符。

逻辑运算符要困难一些。建议不要重载`&&`和`||`。这些运算符并不应用于单个类型，而是整合布尔表达式的结果。此外，重载这些运算符时，会失去短路求值，原因是在将运算符左侧和右侧的值绑定至重载的`&&`和`||`运算符之前，必须对运算符的左侧和右侧进行求值。因此，一般对特定类型重载这些运算符都没有意义。

## 15.4 重载插入运算符和提取运算符

在 C++ 中，不仅算术操作需要使用运算符，从流中读写数据都可使用运算符。例如，向 `cout` 写入整数和字符串时使用插入运算符`<<`：

```
int number { 10 };
cout << "The number is " << number << endl;
```

从流中读取数据时，使用提取运算符`>>`：

```
int number;
string str;
cin >> number >> str;
```

还可为自定义的类编写合适的插入和提取运算符，从而可按以下方式进行读写：

```
SpreadsheetCell myCell, anotherCell, aThirdCell;
cin >> myCell >> anotherCell >> aThirdCell;
cout << myCell << " " << anotherCell << " " << aThirdCell << endl;
```

在编写插入和提取运算符前，需要决定如何将自定义的类向流输出，以及如何从流中提取自定义的类。在这个例子中，`SpreadsheetCell` 将读取和写入 `double` 值。

插入和提取运算符左侧的对象是 `istream` 或 `ostream`(例如 `cin` 和 `cout`)，而不是 `SpreadsheetCell` 对象。由于不能向 `istream` 类或 `ostream` 类添加方法，因此应将插入和提取运算符写为 `SpreadsheetCell` 类的全局函数。这些函数在 `SpreadsheetCell` 类中的声明如下所示：

```
export std::ostream& operator<<(std::ostream& ostr, const SpreadsheetCell& cell);
export std::istream& operator>>(std::istream& istr, SpreadsheetCell& cell);
```

将插入运算符的第一个参数设置为 `ostream` 的引用，这个运算符就能应用于文件输出流、字符串输出流、`cout`、`cerr` 和 `clog` 等。详情参阅第 13 章“解密 C++ I/O”。与此类似，将提取运算符的参数设置为 `istream` 的引用，这个运算符就能应用于文件输入流、字符串输入流和 `cin`。

`operator<<` 和 `operator>>` 的第二个参数是对要写入或读取的 `SpreadsheetCell` 对象的引用。插入运算符不会修改写入的 `SpreadsheetCell` 对象，因此这个引用可以是 `const` 引用。然而提取运算符会修改 `SpreadsheetCell` 对象，因此要求这个参数为非 `const` 引用。

这两个运算符返回的都是第一个参数传入的流的引用，所以这两个运算符的调用可以嵌套。记住，运算符的语法实际上是显式调用全局 `operator>>` 函数或 `operator<<` 函数的简写形式。例如下面这行代码：

```
cin >> myCell >> anotherCell >> aThirdCell;
```

实际上是由下代码行的简写形式：

```
operator>>(operator>>(operator>>(cin, myCell), anotherCell), aThirdCell);
```

从中可以看出，第一次调用 `operator>>` 的返回值被用作下一次调用的输入值。因此必须返回流的引用，结果才能用于下一次嵌套的调用，否则嵌套调用无法编译。

下面是 `SpreadsheetCell` 类的 `operator<<` 和 `operator>>` 运算符的实现：

```
ostream& operator<<(ostream& ostr, const SpreadsheetCell& cell)
{
    ostr << cell.getValue();
    return ostr;
}

istream& operator>>(istream& istr, SpreadsheetCell& cell)
{
    double value;
    istr >> value;
    cell.set(value);
    return istr;
}
```

## 15.5 重载下标运算符

现在暂时假设你从未听说过标准库中的 `vector` 或 `array` 类模板，因此决定自行实现一个动态分配的数组类。这个类允许设置和获取指定索引位置的元素，并会自动完成所有的内存分配操作。一个动态分配的数组类的定义可能是这样的：

```
export template <typename T>
class Array
{
public:
    // Creates an array with a default size that will grow as needed.
    Array();
    virtual ~Array();

    // Disallow assignment and pass-by-value.
    Array& operator=(const Array& rhs) = delete;
    Array(const Array& src) = delete;

    // Move constructor and move assignment operator.
    Array(Array&& src) noexcept;
    Array& operator=(Array&& rhs) noexcept;

    // Returns the value at index x. Throws an exception of type
    // out_of_range if index x does not exist in the array.
    const T& getElementAt(size_t x) const;

    // Sets the value at index x. If index x is out of range,
    // allocates more space to make it in range.
    void setElementAt(size_t x, const T& value);

    // Returns the number of elements in the array.
    size_t getSize() const noexcept;
private:
    static const size_t AllocSize { 4 };
    void resize(size_t newSize);
    T* m_elements { nullptr };
```

```
    size_t m_size { 0 };
};
```

这个接口支持设置和访问元素。它为随机访问提供了保证：客户可创建数组，并设置元素1、100和1000，而不必考虑内存管理问题。

下面是这些方法的实现：

```
template <typename T> Array<T>::Array()
{
    m_size = kAllocSize;
    m_elements = new T[m_size] {}; // Elements are zero-initialized!
}

template <typename T> Array<T>::~Array()
{
    delete [] m_elements;
    m_elements = nullptr;
}

template <typename T> Array<T>::Array(Array&& src) noexcept
: m_elements { std::exchange(src.m_elements, nullptr) }
, m_size { std::exchange(src.m_size, 0) }
{
}

template <typename T> Array<T>& Array<T>::operator=(Array<T>&& rhs) noexcept
{
    if (this == &rhs) { return *this; }
    delete[] m_elements;
    m_elements = std::exchange(rhs.m_elements, nullptr);
    m_size = std::exchange(rhs.m_size, 0);
    return *this;
}

template <typename T> void Array<T>::resize(size_t newSize)
{
    // Create new bigger array with zero-initialized elements.
    auto newArray { std::make_unique<T[]>(newSize) };

    // The new size is always bigger than the old size (mSize)
    for (size_t i { 0 }; i < m_size; i++) {
        // Copy the elements from the old array to the new one
        newArray[i] = m_elements[i];
    }

    // Delete the old array, and set the new array
    delete[] m_elements;
    M_size = newSize;
    M_elements = newArray.release();
}

template <typename T> const T& Array<T>::getElementAt(size_t x) const
{
    if (x >= m_size) { throw std::out_of_range { "" }; }
    return m_elements[x];
}
```

```

template <typename T> void Array<T>::setElementAt(size_t x, const T& val)
{
    if (x >= m_size) {
        // Allocate kAllocSize past the element the client wants
        resize(x + kAllocSize);
    }
    m_elements[x] = val;
}

template <typename T> size_t Array<T>::getSize() const noexcept
{
    return m_size;
}

```

注意 `resize()` 方法的异常安全实现。首先，它创建一个适当大小的新数组，将其存储在 `unique_ptr` 中。然后，将所有元素从旧数组复制到新数组。如果在复制值时任何地方出错，`unique_ptr` 会自动清理新分配的内存。最后，在成功分配新数组和复制所有元素后，即未抛出异常，我们才删除旧的 `m_elements` 数组，并为其指定新数组。最后一行必须使用 `release()` 来释放 `unique_ptr` 的新数组的所有权，否则，在调用 `unique_ptr` 的析构函数时，将销毁这个数组。

下面是使用这个类的简短示例：

```

Array<int> myArray;
for (size_t i { 0 }; i < 10; i++) {
    myArray.setElementAt(i, 100);
}
for (size_t i { 0 }; i < 10; i++) {
    cout << myArray.getElementAt(i) << " ";
}

```

从中可以看出，永远都不需要告诉数组需要多少空间。数组会分配保存给定元素所需要的足够空间。

#### 注意：

这不是高效的内存实现方式。如果创建一个数组且只给索引为 4000 的元素赋值，那么它将为 4000 个元素分配内存，除了索引为 4000 的元素，其他元素都初始化为 0。

然而，总是使用 `setElementAt()` 和 `getElementAt()` 方法并不方便。此外，这两种方法的实现方式对于多维数组的处理会很麻烦。

这里应该使用重载的下标运算符。通过以下方式给类添加 `operator[]`：

```

template <typename T> T& Array<T>::operator[](size_t x)
{
    if (x >= m_size) {
        // Allocate AllocSize past the element the client wants.
        resize(x + AllocSize);
    }
    return m_elements[x];
}

```

有了这层变化，就可以使用传统的数组索引表示法，如下：

```

Array<int> myArray;
for (size_t i { 0 }; i < 10; i++) {

```

```

    myArray[i] = 100;
}
for (size_t i { 0 }; i < 10; i++) {
    cout << myArray[i] << " ";
}

```

`operator[]`既可以用于设置元素，也可以用于获取元素，因为它返回的是对位置 `x` 的元素的引用。此引用可用于赋值给该元素。当 `operator[]` 在赋值语句左侧使用时，赋值实际改变了 `m_elements` 数组中位置 `x` 的值。

有了 `operator[]` 的支持，多维数组的处理现在也很容易，看下例：

```

Array<Array<int>> a;
a[2][4] = 24;
cout << a[2][4] << endl;

```

### 15.5.1 通过 `operator[]` 提供只读访问

尽管有时 `operator[]` 返回可作为左值的元素会很方便，但并非总是需要这种行为。最好还能返回 `const` 引用，提供对数组中元素的只读访问。理想情况下，可提供两个 `operator[]`：一个返回非 `const` 引用，另一个返回 `const` 引用。为此，编写下面这样的代码：

```

T& operator[](size_t x);
const T& operator[](size_t x) const;

```

记住，不能仅基于返回类型重载方法或运算符，因此第二个重载返回 `const` 引用并被标记为 `const`。

下面是 `const operator[]` 的实现：如果索引超出了范围，这个运算符不会分配新空间，而是抛出异常。如果只是读取元素值，那么分配新的空间就没有意义了：

```

template <typename T> const T& Array<T>::operator[](size_t x) const
{
    if (x >= m_size) { throw std::out_of_range{ "" }; }
    return m_elements[x];
}

```

下面的代码演示了这两种形式的 `operator[]`：

```

void printArray(const Array<int>& arr)
{
    for (size_t i { 0 }; i < arr.getSize(); i++) {
        cout << arr[i] << " "; // Calls the const operator[] because arr is
                               // a const object.
    }
    cout << endl;
}

int main()
{
    Array<int> myArray;
    for (size_t i { 0 }; i < 10; i++) {
        myArray[i] = 100; // Calls the non-const operator[] because
                          // myArray is a non-const object.
    }
    printArray(myArray);
}

```

注意，仅因为 arr 是 const，所以 printArray() 中调用的是 const operator[]。如果 arr 不是 const，那么调用的是非 const 的 operator[]，尽管事实上并没有修改结果。

为 const 对象调用 const operator[]，因此无法增加数组大小。当给定索引越界时，当前实现抛出异常。另一种做法是返回而非抛出零初始化元素。代码如下：

```
template <typename T> const T& Array<T>::operator[](size_t x) const
{
    if (x >= m_size) {
        static T nullValue { T() };
        return nullValue;
    }
    return m_elements[x];
}
```

使用零初始化语法 T{} 初始化静态变量 nullValue。可根据具体情况自行选用抛出版本或返回 null 值的版本。

#### 注意：

零初始化，是指使用默认构造函数构造对象，将基本的整数类型(如 char 和 int 等)初始化为 0，将基本的浮点类型初始化为 0.0，将指针类型初始化为 nullptr。

### 15.5.2 非整数数组索引

这是通过提供某种类型的键，对集合进行“索引”的范型的自然延伸；vector(或更广义的任何线性数组)是一种特例，其中的“键”只是数组中的位置。将 operator[] 的参数看成提供的两个域之间的映射：键域到值域的映射。因此，可编写一个将任意类型作为索引的 operator[]。这种类型未必是整数类型。标准库的关联容器就是这么做的，例如第 18 章“标准库容器”讨论的 std::map。

例如，可创建一个关联数组，在其中使用字符串而不是整数作为键。这种类的 operator[] 将接收 string 或者更好的 string\_view 作为参数。这样的类实现是本章节最后的练习题。

#### 注意：

不能重载下标运算符以便接收多个参数。如果要提供接收多个索引的下标访问功能，可使用 15.6 节介绍的函数调用运算符。

## 15.6 重载函数调用运算符

C++ 允许重载函数调用运算符，写作 operator()。如果在自定义的类中编写了一个 operator()，那么这个类的对象就可以当成函数指针使用。包含函数调用运算符的类对象称为函数对象，或简称为仿函数(functor)。只能将这个运算符重载为类中的非静态方法。下例是一个简单的类，它带有一个重载的 operator() 以及一个具有相同行为的类方法。

```
class FunctionObject
{
public:
    int operator() (int param); // Function call operator
    int doSquare(int param);   // Normal method
};

// Implementation of overloaded function call operator
```

```

int FunctionObject::operator() (int param)
{
    return doSquare(param);
}

// Implementation of normal method
int FunctionObject::doSquare(int param)
{
    return param * param;
}

```

下面是使用函数调用运算符的代码示例，注意和类的普通方法调用进行比较：

```

int x { 3 }, xSquared, xSquaredAgain;
FunctionObject square;
xSquared = square(x);           // Call the function call operator
xSquaredAgain = square.doSquare(x); // Call the normal method

```

一开始，函数调用运算符可能看上去有点怪。为什么要为类编写一个特殊方法，使这个类的对象看上去像函数指针？为什么不直接编写一个函数或标准的类方法？

相比标准的对象方法，函数对象的好处很简单：这些对象有时可以伪装成函数指针，可将这些函数对象当成回调函数传给其他函数。第 19 章将详细讨论这些内容。

相比全局函数，函数对象的好处较为复杂。有两个主要好处：

- 对象可在对其函数调用运算符的重复调用之间，在数据成员中保存信息。例如，函数对象可用于记录每次通过函数调用运算符调用采集到的数字连续总和。
- 可通过设置数据成员来自定义函数对象的行为。例如，可编写一个函数对象，比较函数调用运算符的参数和数据成员的值。这个数据成员是可配置的，因此这个对象可自定义为执行任何比较操作。

当然，通过全局变量或静态变量都可实现上述任何好处。然而，函数对象提供了一种更简洁的方式，此外，应该避免使用全局变量或静态变量，这可能会在多线程程序中产生问题。第 20 章“掌握标准库算法”通过标准库展示了函数对象的真正好处。

通过遵循一般的方法重载规则，可为类编写任意数量的 operator()。例如，可向 FunctionObject 类添加一个带 std::string\_view 参数的 operator()。

```

int operator() (int param);
void operator() (std::string_view str);

```

函数调用运算符还可用于提供多维数组的下标。只要编写一个行为类似于 operator[]，但接收多个索引的 operator() 即可。这项技术的唯一问题是需要使用()而不是[]进行索引，例如 myArray(3, 4) = 6。

## 15.7 重载解除引用运算符

可重载 3 个解除引用运算符：\*、-> 和 ->\*。目前暂不考虑->\*(之后详述)，只考虑\*和->的原始意义。`*`解除对指针的引用，允许直接访问这个指针指向的值，`->`是使用`*`解除引用之后再执行`.成员选择操作`的简写。以下代码验证了这两者的一致性：

```

SpreadsheetCell* cell { new SpreadsheetCell };
(*cell).set(5); // Dereference plus member selection
cell->set(5); // Shorthand arrow dereference and member selection together

```

为了使类的对象行为和指针一致，可以在类中重载解除引用运算符。这种功能的主要用途是实现智能指针，参见第7章“内存管理”。还能用于标准库广泛使用的迭代器，参见第17章“理解迭代器和范围库”。本章通过一个简单的智能指针类模板，讲解重载相关运算符的基本机制。

### 警告：

C++有两个标准的智能指针：`std::shared_ptr` 和 `std::unique_ptr`。应该使用这些标准的智能指针类而不是自己编写。这里列举的例子只是为了演示如何编写解除引用运算符。

下面是这个智能指针类模板的定义，其中尚未填入解除引用运算符：

```
export template <typename T> class Pointer
{
public:
    Pointer(T* ptr) : m_ptr { ptr } {}
    virtual ~Pointer() {
        delete m_ptr;
        m_ptr = nullptr;
    }

    // Prevent assignment and pass by value.
    Pointer(const Pointer& src) = delete;
    Pointer<T>& operator=(const Pointer& rhs) = delete;

    // Dereferencing operators will go here.
private:
    T* m_ptr { nullptr };
};
```

这个智能指针就像看上去那么简单。它只是保存了一个普通指针，在这个智能指针被销毁时，将删除它指向的存储空间。这个实现同样十分简单：构造函数接收一个真正的指针（普通指针），将该指针保存为类中仅有的一组数据成员。析构函数释放这个指针引用的存储空间。

可采用以下方式使用这个智能指针类模板：

```
Pointer<int> smartInt { new int };
*smartInt = 5; // Dereference the smart pointer.
cout << *smartInt << endl;

Pointer<SpreadsheetCell> smartCell { new SpreadsheetCell };
smartCell->set(5); // Dereference and member select the set method.
cout << smartCell->getValue() << endl;
```

从这个例子可看出，必须提供 `operator*` 和 `operator->` 的实现。

### 警告：

一般情况下，不要只实现 `operator*` 和 `operator->` 运算符中的一个。几乎总是应该同时实现这两个运算符。如果未同时提供这两个运算符的实现，类的用户可能会感到困惑。

## 15.7.1 实现 `operator*`

当解除对指针的引用时，经常希望能访问这个指针指向的内存。如果那块内存包含一种简单类型，例如 `int`，那么应该可直接修改这个值。如果内存中包含更复杂的类型，例如对象，那么应该能通过

运算符访问对象的数据成员或方法。

为提供这些语义，operator\*应该返回一个引用。在 Pointer 类中，执行下面的操作：

```
export template <typename T> class Pointer
{
public:
    // Omitted for brevity
    T& operator*() { return *m_ptr; }
    const T& operator*() const { return *m_ptr; }
    // Omitted for brevity
};
```

从这个例子可看出，operator\*返回的是底层普通指针指向的对象或变量的引用。与重载下标运算符一样，同时提供方法的 const 版本和非 const 版本也很有用，这两个版本分别返回 const 引用和非 const 引用。

### 15.7.2 实现 operator->

箭头运算符稍微复杂一些。应用箭头运算符的结果应该是对象的成员或方法。然而，要实现这一点，应该能够实现 operator\* 和 operator.; 而 C++ 有充足的理由不允许重载 operator.: 不可能编写单个原型来捕捉任何可能选择的成员或方法。因此，C++ 将 operator-> 当成一种特例。例如下面这行代码：

```
smartCell->set(5);
```

C++ 将这行代码解释为：

```
(smartCell.operator->())->set(5);
```

从中可看出，C++ 给重载的 operator-> 返回的任何结果应用了另一个 operator->。因此，必须返回一个指向对象的指针，如下所示：

```
export template <typename T> class Pointer
{
public:
    // Omitted for brevity
    T* operator->() { return m_ptr; }
    const T* operator->() const { return m_ptr; }
    // Omitted for brevity
};
```

operator\* 和 operator-> 是不对称的，这可能有点令人费解，但见过几次之后就会习惯了。

### 15.7.3 operator.\* 和 operator->\* 的含义

在 C++ 中，获得类的数据成员和方法的地址，以获得指向这些数据成员和方法的指针是完全合法的。然而，不能在没有对象的情况下访问非静态数据成员或调用非静态方法。类的数据成员和方法的重点在于它们依附于对象。因此，通过指针调用方法或访问数据成员时，必须在对象的上下文中解除对指针的引用。operator.\* 和 operator->\* 的语法细节会在第 19 章才介绍，因为它依赖如何定义函数指针的知识点。

C++ 不允许重载 operator.\* (就像不允许重载 operator.; 一样)，但可以重载 operator->\*。然而这个运算符的重载非常复杂，大部分 C++ 程序员甚至不知道可通过指针访问方法和数据成员，因此不值得这么麻烦进行重载。例如，标准库中的 shared\_ptr 智能指针就没有重载 operator->\*。

## 15.8 编写转换运算符

回到 SpreadsheetCell 示例，考虑下面两行代码：

```
SpreadsheetCell cell { 1.23 };
double d1 { cell }; // DOES NOT COMPILE!
```

SpreadsheetCell 包含 double 表达方式，因此将 SpreadsheetCell 赋值给 double 变量看上去是符合逻辑的。但不能这么做。编译器会表示不知道如何将 SpreadsheetCell 转换为 double 类型。你可能会通过下述方式迫使编译器进行这种转换：

```
double d1 { (double)cell }; // STILL DOES NOT COMPILE!
```

首先，上述代码依然无法编译，因为编译器还是不知道如何将 SpreadsheetCell 转换为 double 类型。从这行代码中编译器已知你想让编译器做这种转换，所以编译器如果知道如何转换，就会进行转换。其次，一般情况下，最好不要在程序中添加这种无理由的类型转换。

如果想允许这类赋值，就必须告诉编译器具体如何执行。确切地讲，可编写一个将 SpreadsheetCell 转换为 double 类型的转换运算符。原型如下所示：

```
operator double() const;
```

函数名为 operator double。它没有返回类型，因为返回类型是通过运算符的名称确定的：double。这个函数是 const，因为这个函数不会修改被调用的对象。实现如下所示：

```
SpreadsheetCell::operator double() const
{
    return getValue();
}
```

这就完成了从 SpreadsheetCell 到 double 类型的转换运算符的编写。现在编译器可接受下面这行代码，并且在运行时执行正确的操作。

```
SpreadsheetCell cell { 1.23 };
double d1 { cell }; // Works as expected
```

可用同样的语法编写任何类型的转换运算符。例如，下面是从 SpreadsheetCell 到 std::string 的转换运算符：

```
SpreadsheetCell::operator std::string() const
{
    return doubleToString(getValue());
}
```

现在可将 SpreadsheetCell 转换为 string，但由于 string 提供了构造函数，下面的方法不起作用：

```
string str { cell };
```

相反，可以使用普通的赋值语法来替代统一的初始化，或者使用下面的显式转换 static\_cast()：

```
string str = cell;
string str2 { static_cast<string>(cell) };
```

### 15.8.1 auto 运算符

可以指定 auto 并让编译器推导类型而不是显式指定转换运算符返回的类型，例如 SpreadsheetCell

的 double 转换运算符可以这样写：

```
operator auto() const { return getValue(); }
```

有个警告，具有 auto 返回类型推导的方法实现必须对类的用户可见。因此，本例将实现直接放在类的定义中。

记得第 1 章提到过 auto 会移除引用和 const 限定符。因此如果 operator auto 返回类型 T 的引用，则推导出的类型将通过值方式返回 T，从而生成一份拷贝。如果需要，可以显式地添加引用和 const 限定符，例子如下：

```
operator const auto&() const { /* ... */ }
```

## 15.8.2 使用显式转换运算符解决多义性问题

为 SpreadsheetCell 对象编写 double 转换运算符时会引入多义性问题。例如下面这行加粗代码：

```
SpreadsheetCell cell { 6.6 };  
double d2 { cell + 3.3 }; // DOES NOT COMPILE IF YOU DEFINE operator double()
```

现在这行代码无法成功编译。在编写 operator double() 前，这行代码可编译，那么现在出了什么问题？问题在于，编译器不知道应该通过 operator double() 将 cell 对象转换为 double 类型，再执行 double 加法，还是通过 double 构造函数将 3.3 转换为 SpreadsheetCell，再执行 SpreadsheetCell 加法。在编写 operator double() 前，编译器只有一个选择：通过 double 构造函数将 3.3 转换为 SpreadsheetCell，再执行 SpreadsheetCell 加法。然而，现在编译器可执行两种操作。编译器不想做出让人不喜欢的决定，因此拒绝做出任何决定。

在 C++11 之前，通常解决这个难题的方法是将构造函数标记为 explicit，以避免使用这个构造函数进行自动转换（见第 8 章）。然而，我们不想把这个构造函数标记为 explicit，因为通常希望进行从 double 到 SpreadsheetCell 的自动类型转换。自 C++11 以后，可将 double 类型转换运算符标记为 explicit 以解决这个问题：

```
explicit operator double() const;
```

如此修改后，下面这行代码会正常编译：

```
double d1 { cell + 3.3 }; // 9.9
```

## 15.8.3 用于布尔表达式的转换

有时，能将对象用在布尔表达式中会非常有用。例如，程序员经常在条件语句中这样使用指针：

```
if (ptr != nullptr) { /* Perform some dereferencing action. */ }
```

有时程序员会编写这样的简写条件：

```
if (ptr) { /* Perform some dereferencing action. */ }
```

有时还能看到这样的代码：

```
if (!ptr) { /* Do something. */ }
```

目前，上述任何表达式都不能和此前定义的 Pointer 智能指针类模板一起编译。然而，可给类添加一个转换运算符，将它转换为指针类型。然后，对这种类型和 nullptr 所做的比较操作，以及单个对

象在 if 语句中的形式都会触发这个对象向指针类型的转换。转换运算符常用的指针类型为 void\*，因为这种指针类型除了在布尔表达式中测试外，不能执行其他操作。

```
operator void*() const { return m_ptr; }
```

现在下面的代码可成功编译，并能完成预期的任务：

```
void process(Pointer<SpreadsheetCell>& p)
{
    if (p != nullptr) { cout << "not nullptr" << endl; }
    if (p != NULL) { cout << "not NULL" << endl; }
    if (p) { cout << "not nullptr" << endl; }
    if (!p) { cout << "nullptr" << endl; }
}

int main()
{
    Pointer<SpreadsheetCell> smartCell { nullptr };
    process(smartCell);
    cout << endl;

    Pointer<SpreadsheetCell> anotherSmartCell { new SpreadsheetCell { 5.0 } };
    process(anotherSmartCell);
}
```

输出如下所示：

```
nullptr

not nullptr
not NULL
not nullptr
```

另一种方法是重载 operator bool() 而非 operator void\*()。毕竟是在布尔表达式中使用对象，为什么不直接转换为 bool 类型呢？

```
operator bool() const { return m_ptr != nullptr; }
```

下面的比较仍可以运行：

```
if (p != NULL) { cout << "not NULL" << endl; }
if (p) { cout << "not nullptr" << endl; }
if (!p) { cout << "nullptr" << endl; }
```

然而，使用 operator bool() 时，下面和 nullptr 的比较会导致编译错误：

```
if (p != nullptr) { cout << "not nullptr" << endl; } // Error
```

因为 nullptr 有自己的类型 nullptr\_t，这种类型没有自动转换为整数 0 (false)。编译器找不到接收 Pointer 对象和 nullptr\_t 对象的 operator!=。可将这样的 operator!= 实现为 Pointer 类的友元：

```
export template <typename T>
class Pointer
{
public:
    // Omitted for brevity
    template <typename T>
    friend bool operator!=(const Pointer<T>& lhs, std::nullptr_t rhs);
    // Omitted for brevity
```

```

};

export template <typename T>
bool operator!=(const Pointer<T>& lhs, std::nullptr_t rhs)
{
    return lhs.m_ptr != rhs;
}

```

然而，实现这个 `operator!=` 后，下面的比较会无法工作，因为编译器不知道该用哪个 `operator!=`：

```
if (p != NULL) { cout << "not NULL" << endl; }
```

通过这个例子，可能得出以下结论：`operator bool()` 从技术上看只适用于不表示指针的对象，以及转换为指针类型并没有意义的对象。遗憾的是，添加转换至 `bool` 类型的转换运算符会产生其他一些无法预知的后果。当条件允许时，C++ 会使用“类型提升”规则将 `bool` 类型自动转换为 `int` 类型。因此，采用 `operator bool()` 时，下面的代码可编译运行：

```
Pointer<SpreadsheetCell> anotherSmartCell { new SpreadsheetCell { 5.0 } };
int i { anotherSmartCell }; // Converts Pointer to bool to int.
```

这通常并不是期望或需要的行为。为阻止此类赋值，通常会显式删除到 `int`、`long` 和 `long long` 等类型的转换运算符。但这显得十分凌乱。因此，很多程序员更偏爱使用 `operator void*()` 而不是 `operator bool()`。

从中可以看出，重载运算符时需要考虑设计因素。哪些运算符需要重载的决策会直接影响到客户对类的使用方式。

## 15.9 重载内存分配和内存释放运算符

C++ 允许重定义程序中内存的分配和释放方式。既可在全局层次也可在类层次进行这种自定义。这种功能在可能产生内存碎片的情况下最有用，当分配和释放大量小对象时会产生内存碎片。例如，每次需要内存时，不使用默认的 C++ 内存分配，而是编写一个内存池分配器，以重用固定大小的内存块。本节详细讲解内存分配和内存释放例程，以及如何定制化它们。有了这些工具，就可以根据需求编写自己的分配器。

### 警告：

除非十分了解内存分配的策略，否则尝试重载内存分配例程通常都不值得。不要仅因为看上去不错，就重载它们。只有在真正需要且掌握足够的知识后才这么做。

### 15.9.1 `new` 和 `delete` 的工作原理

C++ 最复杂的地方之一就是 `new` 和 `delete` 的细节。考虑下面这行代码：

```
SpreadsheetCell* cell { new SpreadsheetCell() };
```

`new SpreadsheetCell()` 这部分称为 `new` 表达式。它完成了两件事情。首先，通过调用 `operator new` 为 `SpreadsheetCell` 对象分配了空间。然后，为这个对象调用构造函数。只有这个构造函数完成才返回指针。

`delete` 的工作方式与此类似。考虑下面这行代码：

```
delete cell;
```

这一行称为 `delete` 表达式。它首先调用 `cell` 的析构函数，然后调用 `operator delete` 释放内存。

可重载 `operator new` 和 `operator delete` 来控制内存的分配和释放，但不能重载 `new` 表达式和 `delete` 表达式。因此，可自定义实际的内存分配和释放，但不能自定义构造函数和析构函数的调用。

### 1. `new` 表达式和 `operator new`

有 6 种不同形式的 `new` 表达式，每种形式都有对应的 `operator new`。前面的章节已经展示了 4 种 `new` 表达式：`new`、`new[]`、`new(nothrow)` 和 `new(nothrow)[]`。下面列出了`<new>` 中对应的 4 种 `operator new` 的重载：

```
void* operator new(size_t size);
void* operator new[](size_t size);
void* operator new(size_t size, const std::nothrow_t&) noexcept;
void* operator new[](size_t size, const std::nothrow_t&) noexcept;
```

有两种特殊的 `new` 表达式，它们不进行内存分配，而在已经分配的内存片段上调用构造函数。这种操作称为 `placement new` 运算符(包括单对象和数组形式)。它们在已有的内存中构造对象，如下所示：

```
void* ptr { allocateMemorySomehow() };
SpreadsheetCell* cell { new (ptr) SpreadsheetCell() };
```

对应的两个 `operator new` 重载如下所示，然而，C++ 标准禁止重载它们：

```
void* operator new(size_t size, void* p) noexcept;
void* operator new[](size_t size, void* p) noexcept;
```

这个特性有点偏门，但知道这项特性的存在非常重要。如果需要实现内存池，以便在不释放内存的情况下重用内存，这项特性就非常方便。这样可以构造和销毁对象的实例而不需要为每个新实例分配内存。第 29 章“编写高效的 C++ 代码”有内存池实现的示例。

### 2. `delete` 表达式和 `operator delete`

只可调用两种不同形式的 `delete` 表达式：`delete` 和 `delete[]`，没有 `nothrow` 和 `placement` 形式。然而，`operator delete` 有 6 种重载。为什么有这种不对称性？`nothrow` 和 `placement` 形式只有在构造函数抛出异常时才会使用。这种情况下，匹配调用构造函数之前分配内存时使用的 `operator new` 的 `operator delete` 会被调用。然而，如果正常地删除指针，`delete` 会调用 `operator delete` 或 `operator delete[]`(绝不会调用 `nothrow` 或 `placement` 形式)。在实际上，这并没有关系：C++ 标准指出，从 `delete`(例如 `delete` 调用的析构函数)抛出异常的行为是未定义的。也就是说，`delete` 永远都不应该抛出异常。因此 `nothrow` 版本的 `operator delete` 是多余的；而 `placement` 版本的 `delete` 应该是一个空操作，因为在 `placement new` 中并没有分配内存，因此也不需要释放内存。

下面是对于 `operator new` 重载的 6 个 `operator delete` 重载的原型：

```
void operator delete(void* ptr) noexcept;
void operator delete[](void* ptr) noexcept;
void operator delete(void* ptr, const std::nothrow_t&) noexcept;
void operator delete[](void* ptr, const std::nothrow_t&) noexcept;
void operator delete(void* p, void*) noexcept;
void operator delete[](void* p, void*) noexcept;
```

### 15.9.2 重载 operator new 和 operator delete

如有必要，可替换全局的 operator new 和 operator delete 例程。这些函数会被程序中的每个 new 表达式和 delete 表达式调用，除非在类中有更特别的版本。然而，引用 Bjarne Stroustrup 的一句话：“用……替换全局的 operator new 和 operator delete 是需要胆量的。”我们不建议替换。

#### 警告：

如果没有听取我们的建议并决定替换全局的 operator new，一定要注意在这个运算符的代码中不要对 new 进行任何调用，否则会产生无限循环。例如，不能通过 cout 向控制台写入消息。

更有用的技术是重载特定类的 operator new 和 operator delete。仅当分配或释放特定类的对象时，才会调用这些重载的运算符。下面这个类重载了 4 个非 placement 形式的 operator new 和 operator delete：

```
export class MemoryDemo
{
public:
    virtual ~MemoryDemo() = default;

    void* operator new(size_t size);
    void operator delete(void* ptr) noexcept;

    void* operator new[](size_t size);
    void operator delete[](void* ptr) noexcept;

    void* operator new(size_t size, const std::nothrow_t&) noexcept;
    void operator delete(void* ptr, const std::nothrow_t&) noexcept;

    void* operator new[](size_t size, const std::nothrow_t&) noexcept;
    void operator delete[](void* ptr, const std::nothrow_t&) noexcept;
};
```

下面是这些运算符的简单实现，它们简单地将消息写到标准输出中，并将参数传递给这些运算符全局版本的调用。注意 noexcept 实际上是一个 noexcept\_t 类型的变量：

```
void* MemoryDemo::operator new(size_t size)
{
    cout << "operator new" << endl;
    return ::operator new(size);
}

void MemoryDemo::operator delete(void* ptr) noexcept
{
    cout << "operator delete" << endl;
    ::operator delete(ptr);
}

void* MemoryDemo::operator new[](size_t size)
{
    cout << "operator new[]" << endl;
    return ::operator new[](size);
}

void MemoryDemo::operator delete[](void* ptr) noexcept
{
    cout << "operator delete[]" << endl;
    ::operator delete[](ptr);
}
```

```

}

void* MemoryDemo::operator new(size_t size, const noexcept_t&) noexcept
{
    cout << "operator new noexcept" << endl;
    return ::operator new(size, noexcept);
}

void MemoryDemo::operator delete(void* ptr, const noexcept_t&) noexcept
{
    cout << "operator delete noexcept" << endl;
    ::operator delete(ptr, noexcept);
}

void* MemoryDemo::operator new[](size_t size, const noexcept_t&) noexcept
{
    cout << "operator new[] noexcept" << endl;
    return ::operator new[](size, noexcept);
}

void MemoryDemo::operator delete[](void* ptr, const noexcept_t&) noexcept
{
    cout << "operator delete[] noexcept" << endl;
    ::operator delete[](ptr, noexcept);
}

```

下面的代码以不同方式分配和释放这个类的对象：

```

MemoryDemo* mem { new MemoryDemo{} };

delete mem;
mem = new MemoryDemo[10];
delete [] mem;
mem = new (nothrow) MemoryDemo{};
delete mem;
mem = new (nothrow) MemoryDemo[10];
delete [] mem;

```

下面是运行这个程序后得到的输出：

```

operator new
operator delete
operator new[]
operator delete[]
operator new noexcept
operator delete
operator new[] noexcept
operator delete[]

```

operator new 和 operator delete 的这些实现非常简单，但作用不大。它们旨在呈现语法形式，以便在实现真正版本时参考。

### 警告：

当重载 operator new 时，要重载对应形式的 operator delete。否则，内存会根据指定的方式分配，但是根据内建的语义释放，这两者可能不兼容。

重载所有不同形式的 operator new 看上去有点过分。但一般情况下最好这么做，从而避免内存分配的不一致。如果不想提供任何实现，可使用=delete 显式地删除函数，以避免别人使用。

**警告：**

重载所有形式的 operator new，或显式删除不想使用的形式，以免内存分配中出现不一致的情形。

### 15.9.3 显式地删除/默认化 operator new 和 operator delete

第8章展示了如何删除或默认化构造函数或赋值运算符。显式地删除或默认化并不局限于构造函数和赋值运算符。例如，下面的类删除了 operator new 和 new[]。也就是说，这个类不能通过 new 或 new[] 动态创建：

```
class MyClass
{
public:
    void* operator new(size_t size) = delete;
    void* operator new[](size_t size) = delete;
};
```

按以下方式使用这个类会生成编译错误：

```
MyClass* p1 { new MyClass };
MyClass* pArray { new MyClass[2] };
```

### 15.9.4 重载带有额外参数的 operator new 和 operator delete

除了重载标准形式的 operator new 外，还可编写带额外参数的版本。这些额外参数可用于向内存分配例程传递各种标志或计数器。例如，一些运行时库在调试模式中使用这种形式，在分配对象的内存时提供文件名和行号，这样在发生内存泄漏时，便可识别出发生问题的分配内存的那行代码。

下面是 MemoryDemo 类中带有额外整数参数的 operator new 和 operator delete 原型：

```
void* operator new(size_t size, int extra);
void operator delete(void* ptr, int extra) noexcept;
```

实现如下所示：

```
void* MemoryDemo::operator new(size_t size, int extra)
{
    cout << "operator new with extra int: " << extra << endl;
    return ::operator new(size);
}
void MemoryDemo::operator delete(void* ptr, int extra) noexcept
{
    cout << "operator delete with extra int: " << extra << endl;
    return ::operator delete(ptr);
}
```

编写带有额外参数的重载 operator new 时，编译器会自动允许编写对应的 new 表达式。new 的额外参数以函数调用的语法传递(和 nothrow new 一样)。因此，可编写这样的代码：

```
MemoryDemo* memp { new(5) MemoryDemo() };
delete memp;
```

输出如下所示：

```
operator new with extra int: 5
operator delete
```

定义带有额外参数的 operator new 时，还应该定义带有额外参数的对应 operator delete。然而不能自己调用这个带有额外参数的 operator delete，只有在使用了带有额外参数的 operator new 且对象的构造函数抛出异常时，才调用这个 operator delete。

### 15.9.5 重载带有内存大小参数的 operator delete

另一种形式的 operator delete 提供了要释放的内存大小和指针。只需要声明带有额外大小参数的 operator delete 原型。

#### 警告：

如果类声明了两个一样版本的 operator delete，只不过一个接收大小参数，另一个不接收，那么不接收大小参数的版本总是会被调用。如果需要使用带有大小参数的版本，请只编写那个版本。

可独立地将任何版本的 operator delete 替换为接收大小参数的 operator delete 版本。下面是 MemoryDemo 类的定义，将其中的第一个 operator delete 改为接收要释放的内存大小作为参数：

```
export class MemoryDemo
{
public:
    // Omitted for brevity
    void* operator new(size_t size);
    void operator delete(void* ptr, size_t size) noexcept;
    // Omitted for brevity
};
```

这个 operator delete 的实现调用没有大小参数的全局 operator delete，因为并不存在接收这个大小参数的全局 operator delete。

```
void MemoryDemo::operator delete(void* ptr, size_t size) noexcept
{
    cout << "operator delete with size " << size << endl;
    ::operator delete(ptr);
}
```

只有在需要为自定义类编写复杂的内存分配和释放方案时，才使用这个功能。

### 15.9.6 重载用户定义的字面量运算符

C++有大量的可在代码中使用的标准字面量，示例如下。

- 'a': 字符
- "character array": 0 终止的字符数组，C 风格的字符串
- 3.14f: 单精度浮点数值
- 0xabc: 十六进制数值

然而，C++也允许定义自己的字面量。用户定义的字面量应该以下划线开头，下划线后的第一个字符必须是小写字母，例如\_i、\_s、\_km、\_miles 等。

用户定义的字面量通过编写字面量运算符来实现。字面量运算符可以 raw 模式和 cooked 模式工作。在 raw 模式下，字面量运算符接收字符序列，在 cooked 模式下，字面量运算符接收特定的解释类型。以 C++字面量 123 为例，raw 模式字面量运算符当作'1', '2', '3'的字符序列来接收，cooked 模式字面量运算符当作整数 123 来接收。0x23 字面量被 raw 运算符接收为字符'0', 'x', '2', '3'，而 cooked 运算符接收为整数 35。像 3.14 这样的字面量被 raw 运算符接收为字符'3', '.', '1', '4'，而 cooked 运算

符接收的是浮点数 3.14。

### 15.9.7 cooked 模式字面量运算符

cooked 模式的字面量运算符应该具有以下任意一种：

- 处理数值：unsigned long long、long double、char、wchar\_t、char8\_t、char16\_t 或 char32\_t 的类型参数。
- 处理字符串：两个参数，第一个是字符数组，第二个是字符数组的长度，例如(const char\* str, size\_t len)。

下例为用户定义的字面量运算符\_i 实现了 cooked 字面量运算符，来定义复数字面量：

```
complex<long double> operator""_i(long double d)
{
    Return complex<long double> { 0, d };
}
```

\_i 字面量运算符可以像下面这样使用：

```
complex<long double> c1 { 9.634_i };
auto c2 { 1.23_i }; // c2 has as type complex<long double>
```

下例为用户定义的字面量运算符\_s 实现了 cooked 字面量运算符，来定义 std::string 字面量：

```
string operator""_s(const char* str, size_t len)
{
    return string(str, len);
}
```

字面量运算符可以像下面这样使用：

```
string str1 { "Hello World"_s };
auto str2 { "Hello World"_s }; // str2 has as type string
```

如果没有\_s 字面量运算符，auto 类型推导会推导出 const char\*：

```
auto str3 { "Hello World" }; // str3 has as type const char*
```

### 15.9.8 raw 模式字面量运算符

raw 模式的字面量运算符需要一个 const char\*类型的参数，即以 0 为终止符的 C 风格的字符串参数，下例讲字面量运算符定义为 raw 模式的字面量运算符：

```
Complex<long double> operator""_i(const char* p)
{
    // Implementation omitted; it requires parsing the C-style
    // string and converting it to a complex number.
}
```

使用 raw 模式字面量运算符的方式与 cooked 版本相同。

#### 注意：

raw 模式字面量运算符只能处理非字符串字面量。例如，1.23\_i 可以用 raw 模式的字面量运算符来实现，但是"1.23"\_i 却不能。后者需要带有两个参数的 cooked 字面量运算符：以零结束的字符串和它的长度。

### 15.9.9 标准用户定义的字面量

C++定义了表 15-3 所示的标准用户定义的字面量，注意这些标准用户定义的字面量不以下画线开头。

表 15-3 标准用户定义的字面量

字面量	创建.....类型的实例	例子	需要的名称空间
s	string	auto myString { "Hello"s};	string_literals
sv	string_view	auto myStringView { "Hello"sv};	string_view_literals
h, min, s, ms, us, ns	chrono::duration 在第 22 章讨论	auto myDuration { 42min };	chrono_literals
y, d	chrono::year 和 chrono::day 在第 22 章讨论	auto thisYear { 2020y };	chrono_literals
i, il, if	complex<T>, T 分别等于 double、long double 和 float 类型	auto myComplexNumber { 1.3i };	complex_literals

从技术上讲，这些都是在 std::literals 的子名称空间中定义，例如 std::literals::string\_literals。但是 string\_literals 和 literals 都是内联名称空间，它们会自动使其内容在父名称空间中可用。因此如果想使用 s 字符串字面量，可以使用以下任何一个 using 指令：

```
using namespace std;
using namespace std::literals;
using namespace std::string_literals;
using namespace std::literals::string_literals;
```

## 15.10 本章小结

本章总结了运算符重载的合理性，并对不同类别的运算符提供了重载示例和讲解。希望通过这一章，读者能意识到运算符重载的威力。贯穿本书，运算符重载都用于提供抽象和易用的类接口。

接下来该研究 C++ 标准库了。下一章首先概述 C++ 标准库提供的功能，之后的各章将深入探讨 C++ 标准库的特殊功能。

## 15.11 练习

通过完成下面的习题，可以巩固本章所讨论的知识点。所有习题的答案都可扫描封底二维码下载。然而如果你受困于某个习题，可以在看网站上的答案之前，先重新阅读本章的部分内容，尝试着自己找到答案。

**练习 15-1 实现 AssociativeArray 类**，类应该在 vector 中存储大量元素，每个元素由键和值组成。键总是字符串，而值的类型可以使用模板类型参数指定。提供重载的下标运算符，以便可以根据元素的键检索元素，并在 main() 函数中测试实现。注意，这个习题只是为了练习使用非整数下标实现下标运算符。在实践中，应该只使用标准库提供的 std::map 类模板，第 18 章将讨论这样的关联数组。

**练习 15-2** 从练习 13-2 中获取 Person 类的实现，向其添加插入和提取运算符的实现，确保提取运算符可以读回插入运算符所写的内容。

**练习 15-3** 将字符串转换运算符添加到练习 15-2 的答案中，该运算符简单地返回一个根据人的名字和姓氏构造的字符串。

**练习 15-4** 基于练习 15-3 的答案，添加用户定义的字面量运算符 `_p`，根据字符串字面量构造 Person。它应该在姓氏中支持空格，但在名字中不支持。例如 "Peter Van Weert" `_p` 应该生成一个名为 Peter 姓为 Van Weert 的 Person 对象。



# 第 16 章

## C++ 标准库概述

### 本章内容

- 贯穿标准库的编码原则
- 标准库提供的功能概述

作为一名 C++ 程序员，使用的最重要的库就是 C++ 标准库。顾名思义，这个库是 C++ 标准的一部分，因此任何符合标准的编译器都应该带有这个库。标准库并不是一体性库，而是包含一些完全不同的组件，有些组件可能已经正在使用，甚至可能会认为那些部分就是核心语言的一部分。所有标准库类和函数都在 std 名称空间或其子名称空间中声明。

C++ 标准库的核心是泛型容器和泛型算法。该库中的这个子集通常称为标准模板库(Standard Template Library, STL)，因为它最初基于第三方库“标准模板库”，该库大量使用了模板。但 STL 并非由 C++ 标准本身定义的术语，因此本书不会使用这个术语。标准库的威力在于提供了泛型容器和泛型算法，使大部分算法可用于大部分容器，无论容器中保存的数据类型是什么。性能是标准库中非常重要的一部分。标准库的目标是要让标准库容器和算法与手工编写的代码速度相当甚至更快。

C++ 标准库也包含了 C11 标准的大多数 C 头文件，但使用了新名称。例如，可以通过包含 <cstdio>(全部在 std 名称空间中)，可以访问 C 中<stdio.h>头文件的功能。前者将所有内容都放在全局名称空间中，而后者将所有内容都放在 std 名称空间中。但是，从技术上讲，前者也可以将内容放入 std 名称空间中，而后者则可以将内容都放入全局名称空间中。C11 头文件<stdatomic.h><stdnoreturn.h><threads.h>，以及对应的<c...>不包含在 C++ 标准中。另外，C++17 已经弃用，并且 C++ 20 已经删除了下面的 C 头文件。

- <ccomplex>和<ctgmath>：使用<complex>和/或<cmath>替换它们的使用。
- <ciso646><cstdalign>和<cstdbool>：这些头文件在 C++ 中是没有作用的，因为它们是空的或这些定义作为 C++ 关键字的宏。

C 头文件不能保证是可导入的。可以使用#include 代替 import 来访问它们定义的功能。

### 注意：

优先使用 C++ 功能，尽量不要使用 C 头文件中包含的功能。

渴望成为语言专家的 C++ 程序员应当熟悉标准模板库。如果在自己的程序中整合使用标准库容器和算法，而不是编写和调试自己的容器和算法，那么节省的时间是不可估量的。下面开始深入介绍标准库。

这是介绍标准库的第 1 章，对可用功能进行了概述。此后的几章将更深入地讲解标准库的几个方面，包括容器、迭代器、泛型算法、预定义的函数对象类、正则表达式、文件系统支持和随机数生成等。另外，第 25 章“自定义和扩展标准库”专门介绍自定义和扩展标准库。

尽管这一章和此后几章中的内容深度都不浅，但标准库实在太大，本书不可能全部覆盖。不过应该可以通过本章和随后几章了解标准库，但要记住，这几章并没有囊括标准库中各种类提供的所有方法和数据成员，也没有展示所有算法的原型。附录 C 概述了标准库中的所有头文件。有关所有可用功能的完整列表，请参阅最喜欢的标准库参考。

## 16.1 编码原则

标准库大量使用了 C++ 的模板功能(templates)和运算符重载功能(operator overloading)。

### 16.1.1 使用模板

模板用于实现泛型编程(generic programming)。通过模板，才能编写适用于所有类型对象的代码，模板甚至可用于编写代码时未知的对象。编写模板代码的程序员负责指定定义这些对象的类的需求，例如，这些类要有用于比较的运算符，或者要有复制构造函数，或者要满足其他任何必要条件，然后要确保编写的代码只使用必需的功能。创建对象的程序员负责提供模板编写者要求的那些运算符和方法。

遗憾的是，很多程序员认为模板是 C++ 中最难的部分，因此试图避免使用模板。不过，即使永远也不会编写自己的模板，也需要了解模板的语法和功能，以使用标准库。模板详情请参阅第 12 章“利用模板编写泛型代码”。如果跳过该章，也不熟悉模板，建议先阅读第 12 章，然后再继续学习标准库。

### 16.1.2 使用运算符重载

C++ 标准库大量使用了运算符重载。第 9 章的 9.7 节专门介绍了运算符重载。在阅读这一章和后续各章时一定要首先阅读那一节的内容。此外，第 15 章讲解了运算符重载的更多细节。

运算符重载是 C++ 标准库广泛使用的另一个特性。第 9 章“精通类与对象”有一整节专门讨论运算符重载。在继续处理本章和阅读后续章节之前，确保已经阅读了那一部分，并理解了它。此外，第 15 章“C++ 运算符重载”提供了有关运算符重载主题的细节，但这些细节对于理解接下来的章节来说不是必需的。

## 16.2 C++ 标准库概述

本节从设计角度介绍标准库中的几个组件，学习有哪些可供使用的工具，但不会学习编码细节。这些细节将在其他章节中介绍。

### 16.2.1 字符串

C++ 在<string>头文件中提供了内建的 string 类。尽管仍可以使用 C 风格的字符数组字符串，但 C++ 的 string 类几乎在各个方面都比字符数组好。string 类处理内存管理；提供一些边界检查、赋值语

义以及比较操作；还支持一些操作，例如串联、子字符串提取以及子字符串或字符的替换。

#### 注意：

从技术角度看，`std::string` 是对 `std::basic_string` 类模板进行 `char` 实例化的类型别名。但是，不需要关注这些细节；只要像非模板类那样使用 `string` 类即可。

标准库还提供在`<string_view>`中定义的 `string_view` 类。这是各类字符串表示的只读视图，可用于简单替换 `const string&`，而且不会带来开销。它从不复制字符串！

C++ 支持 Unicode 和本地化。Unicode 允许编写使用不同语言(如阿拉伯语、汉语、日语等)的程序。在`<locale>`头文件中定义的 Locale 允许根据某个国家或地区的规则格式化数据，例如数字和日期等数据。

C++20 包括一个强大的类型安全的字符串格式化库，可以通过定义在`<format>`头文件中的 `std::format()` 进行访问。这个库是可扩展的，允许添加对自定义类型的支持。

提示一下，第 2 章“使用 `string` 和 `string_view`”详细介绍了 `string` 和 `string_view` 类以及字符串格式化库的所有细节。而第 21 章“字符串的本地化与正则表达式”将讨论 Unicode 和本地化。

### 16.2.2 正则表达式

`<regex>` 头文件提供了正则表达式。正则表达式简化了文本处理中常用的模式匹配任务。通过模式匹配可在字符串中搜索特定的模式，还能酌情将搜索到的模式替换为新模式。第 21 章将讨论正则表达式。

### 16.2.3 I/O 流

C++ 引入了一种新的使用流的输入输出模型。C++ 库提供了能在文件、控制台/键盘和字符串中读写内建类型的例程。C++ 提供的工具还可编写读写自定义对象的例程。大多数的 I/O 功能在如下几个头文件中定义：`<fstream>`、`<iomanip>`、`<ios>`、`<iosfwd>`、`<iostream>`、`<istream>`、`<ostream>`、`<sstream>`、`<streambuf>` 和 `<strstream>`。第 1 章“C++ 和标准库速成”概述了基本 I/O 流。第 13 章“C++ I/O 揭秘”详细讨论了流的细节。

### 16.2.4 智能指针

编写健壮程序时需要面对的一个问题就是要知道何时删除对象。有几种可能发生的故障。第 1 个问题是根本没有删除对象(没有释放存储)。这称为内存泄漏(memory leaks)，发生这种问题时对象越积越多，占用了空间，并且这些空间不再被使用。另一个问题是一段代码删除了存储，而另一段代码仍然引用了这个存储，导致指向那个存储的指针不再可用或已重新分配作他用。这称为悬挂指针(dangling pointer)。还有一个问题是段代码释放了一块存储，而另一段代码试图释放同一块存储。这称为双重释放(double deletion)。所有这些问题都会导致程序发生某种故障。有些故障很容易检测出来，而有些故障会导致程序产生错误的结果。这些错误大多很难发现和修复。

C++ 用智能指针 `unique_ptr`、`shared_ptr` 和 `weak_ptr` 解决了这些问题。`shared_ptr` 和 `weak_ptr` 是线程安全的，它们都在`<memory>`头文件中定义。这些智能指针将在第 7 章“内存管理”中详细讨论。

### 16.2.5 异常

C++ 语言支持异常，函数和方法能通过异常将不同类型的错误向上传递至调用的函数或方法。C++

标准库提供了异常的类层次结构，在程序中可使用这些类，也可通过继承方式创建自己的异常类型。大多数异常支持在下面几个头文件中定义：`<exception>`、`<stdexcept>`和`<system_error>`。第 14 章“错误处理”中详细讨论了异常和标准异常类。

### 16.2.6 数学工具

C++标准库提供了一些数学工具类和函数。

有一组完整且常见的数学函数可供使用，如 `abs()`、`remainder()`、`fma()`、`exp()`、`log()`、`pow()`、`sqrt()`、`sin()`、`atan2()`、`sinh()`、`erf()`、`tgamma()`、`ceil()` 和 `floor()` 等。`C++17` 增加了大量的特殊数学函数，以处理勒让德多项式、 $\beta$  函数、椭圆积分、贝塞尔函数、柱函数和诺伊曼函数等。这些特殊的函数具有既定的名称和符号，常用于数学分析、泛函分析、几何、物理和其他应用。`C++20` 添加了一个 `lerp()` 函数来计算线性插值或外插：`lerp(a,b,t)` 用于计算  $a + t(b - a)$ 。线性插值计算给定数据点之间的某个值，而外插计算低于或高于最小或最大数据点的值。所有这些函数都在`<cmath>`头文件中定义。

`<numeric>` 定义了 `gcd()` 和 `lcm()`，它们分别计算两种整数类型的最大公约数和最小公倍数。`C++20` 添加了 `midpoint()` 来计算两个值(整数、浮点数或指针)的中点。

标准库在`<complex>`头文件中提供了一个复数类，名为 `complex`，这个类提供了对包含实部和虚部的复数的操作抽象。

编译期有理数运算库在`<ratio>`头文件中提供了 `ratio` 类模板。这个 `ratio` 类模板可精确表示任何由分子和分母定义的有限有理数。这个程序库将在第 22 章“日期和时间工具”中进行讨论。

标准库还在`<valarray>`头文件中包含一个 `valarray` 类，这个类和 `vector` 类相似，但对高性能数值应用做了特别优化。这个库提供了一些表示矢量切片概念的相关类。通过这些构件，可构建执行矩阵数学运算的类。没有内建的矩阵类，但是像 Boost 这样的第三方库提供了矩阵的支持。本书不会详细讨论 `valarray` 类。

 `C++20` 包含了一组数学常量，它们都定义在`<numbers>`头文件中的 `std::numbers` 名称空间中。表 16-1 显示了一些可用的常量：

表 16-1 可用的常量

常量	描述	近似值
<code>pi</code>	<code>pi</code> ( $\pi$ ) 的值	3.141593
<code>inv_pi</code>	$\pi$ 的倒数	0.318310
<code>sqrt2</code>	2 的平方根	1.414214
<code>e</code>	欧拉数 e	2.718282
<code>phi</code>	黄金比例	1.618034

 `C++20` 添加了如表 16-2 所示的函数来处理位，它们都定义在`<bit>`头文件中。所有这些函数都需要无符号整型作为第一个参数：

表 16-2 函数

函数	描述
<code>has_single_bit()</code>	如果给定值只包含一位，即是 2 的整数次幂，则返回 <code>true</code>
<code>bit_ceil()</code>	返回大于或等于给定值的 2 的最小整数次幂
<code>bit_floor()</code>	返回小于或等于给定值的 2 的最大整数次幂
<code>bit_width()</code>	返回存储给定值所需的位数

(续表)

函数	描述
rotl()	将给定值的位分别在给定数量的位置上向左或向右旋转
rotr()	
countl_zero	从左边开始, 即从最高位开始, 分别返回给定值中连续的 0 位或 1 位的个数
countl_one	
countr_zero	从右边开始, 即从最低有效位开始, 返回给定值中从右开始的连续 0 位或 1 位的个数
countr_one	
popcount()	返回给定值中 1 位的个数

示例如下:

```

cout << popcount(0b10101010u) << endl; // 4

uint8_t value { 0b11101011u };
cout << countl_one(value) << endl; // 3
cout << countr_one(value) << endl; // 2

value = 0b10001000u;
cout << format("{:08b}", rotl(value, 2)) << endl; // 00100010

value = 0b00001011u;
cout << format("bit_ceil({0:08b} = {0}) = {1:08b} = {1}", value, bit_ceil(value)) << endl; // bit_ceil(00001011 = 11) = 00010000 = 16
cout << format("bit_floor({0:08b} = {0}) = {1:08b} = {1}", value, bit_floor(value)) << endl; // bit_floor(00001011 = 11) = 00001000 = 8

```

+20 C++20 还添加了以下比较函数: std::cmp\_equal()、cmp\_not\_equal()、cmp\_less()、cmp\_less\_equal()、cmp\_greater() 和 cmp\_greater\_equal(), 所有这些函数都在<utility>中定义。它们可以对两个整数进行适当的比较, 并且可以安全地用于有符号数和无符号数的混合比较。

例如, 下面的代码比较有符号数 -1 和无符号数 0u。输出的结果是 1 (= true), 因为 -1 首先会被转换为一个无符号整数, 因此 -1 将变成一个大数字, 例如 4 294 967 295, 它肯定大于 0:

```
cout << (-1 > 0u) << endl; // 1 (= true)
```

使用 cmp\_greater() 获得正确的输出:

```
cout << cmp_greater(-1, 0u) << endl; // 0 (= false)
```

## 16.2.7 时间和日期工具

C++ 在<chrono>头文件中包含了 chrono 库。这个程序库简化了与时间相关的操作, 例如特定时间间隔的定时操作或定时相关的操作。C++20 添加了对时区的支持, 包括在不同时区之间转换时间的功能, 并添加了日历的概念来处理日期。<ctime> 头文件提供了许多 C 风格的时间和日期实用程序。

第 22 章将详细讨论时间和日期工具。

## 16.2.8 随机数

C++ 很久以前就支持使用 srand() 和 rand() 函数生成随机数, 但这些函数只提供非常初级的随机数。例如, 无法修改生成的随机数的分布。

自 C++11 以后，在标准库中添加了一个完善的随机数库，这个新库在`<random>`头文件中定义，带有随机数引擎、随机数引擎适配器以及随机数分布。通过这些组件可以生成更适合特定问题域的随机数，如正态分布、负指数分布等。

第 23 章“随机数工具”将详细讨论这个新库。

### 16.2.9 初始化列表

初始化列表在`<initializer_list>`头文件中定义，它们便于编写参数数目可变的函数，相关讨论请参阅第 1 章。示例如下：

```
int makeSum(initializer_list<int> values)
{
    int total { 0 };
    for (int value : values) { total += value; }
    return total;
}
int a { makeSum({ 1, 2, 3 }) };
int b { makeSum({ 10, 20, 30, 40, 50, 60 }) };
```

### 16.2.10 Pair 和 Tuple

`<utility>`头文件定义了 pair 类模板，用于存储两种不同类型的元素。这称为存储异构元素。本章后面讨论的所有标准库容器都只能存储同构元素。也就是说，容器中的所有元素都必须具有相同的类型。pair 允许在一个对象中保存类型毫不相关的元素。Pair 类模板在第 1 章中介绍过。

`<tuple>`中定义的 tuple 是 pair 的一种泛化，它是固定大小的序列，元组的元素可以是异构的，tuple 实例化的元素数目和类型在编译期是固定不变的。关于 tuple 的讨论请参阅第 20 章。

### 16.2.11 词汇类型

C++支持下面所谓的词汇类型(vocabulary type)：

- optional，在`<optional>`中定义，要么存储指定类型的值，要么什么都不存储。如果允许值是可选的，那么它可用于类数据成员、函数的参数和函数的返回类型。Optional 在第 1 章中介绍过。
- variant，在`<variant>`中定义，可存储单个值(属于一组给定类型中的一种类型)。相关讨论，请参阅第 24 章“其他库工具”。
- any，在`<any>`中定义，可包含单个值，值可以是任何类型。相关讨论，请参阅第 24 章。

### 16.2.12 函数对象

实现函数调用运算符的类称为函数对象(function object)。函数对象可用作某些标准库算法的谓词。`<functional>`头文件定义了一些预定义的函数对象，并支持根据已有的函数对象创建新的函数对象。

函数对象请参阅第 19 章“函数指针、函数对象和 lambda 表达式”。

### 16.2.13 文件系统

文件系统支持库的所有内容都在`<filesystem>`头文件中定义，位于 `std::filesystem` 名称空间。它允许编写可用于文件系统的可移植代码。可以使用它确定是目录还是文件，迭代目录内容，操纵路径，

以及检索有关文件的信息(如文件大小、扩展名和创建时间等)。第 13 章将讨论文件系统支持库。

### 16.2.14 多线程

所有主流 CPU 经销商都销售多核处理器，它们用于从服务器到用户计算机等所有设备，甚至还用于智能手机。如果希望软件利用所有这些核，就需要编写多线程代码。标准库提供了几个基本的构件块来编写这种代码。单个线程可以用`<thread>`头文件中的 `thread` 类创建。C++20 将 `jthread` 添加到程序库中，这是一个可以被取消的线程，当它被销毁时，它会自动执行所谓的联结(join)操作。

在多线程代码中，需要考虑如下问题：几个线程不能同时读写同一份数据，因为那将会引发所谓的数据竞争(data race)。为避免这种情形发生，可使用`<atomic>`头文件中定义的原子性，它提供了对数据的线程安全的原子访问。`<condition_variable>`和`<mutex>`提供了其他线程同步机制。

C++20 增加了对其他同步原语的支持：信号量(`<semaphore>`)、锁存(`<latch>`)和屏障(`<barrier>`)。

如果只需要计算某个数据(可能在不同线程上)，得到结果，并具有相应的异常处理，可使用`<future>`头文件中定义的 `async` 和 `future`，这些比直接使用 `thread` 类更容易。

更多有关编写多线程代码的内容，请参阅第 27 章“C++ 多线程编程”。

### 16.2.15 类型萃取

类型萃取在`<type_traits>`头文件中定义，提供了编译期间的类型信息。编写高级模板时可使用它，关于类型萃取的讨论，请参阅第 26 章“高级模板”。

### 16.2.16 标准整数类型

`<cstdint>`头文件定义了大量标准整数类型，如 `int8_t` 和 `int64_t` 等，还包含多个宏(指定这些类型的最小值和最大值)。在第 34 章“开发跨平台和跨语言应用程序”中编写跨平台代码时，将讨论这些整数类型。

### 16.2.17 标准库特性测试宏

从 C++20 开始，所谓的特性测试宏可以用于标准库特性。它们用于验证标准库实现是否支持某个特性。所有这些宏都以`_cpp_lib_`开头。下面是一些示例：

- `_cpp_lib_concepts`
- `_cpp_lib_ranges`
- `_cpp_lib_scoped_lock`
- `_cpp_lib_atomic_float`
- `_cpp_lib_constexpr_vector`
- `_cpp_lib_constexpr_tuple`
- `_cpp_lib_filesystem`
- `_cpp_lib_three_way_comparison`
- ...

这种宏的值是一个数字，表示添加或更新特定特性的月份和年份。日期格式为 YYYYMM。例如，`_cpp_lib_filesystem` 的值为 201703，即 2017 年 3 月。

#### 注意：

很少需要这些特性测试宏，除非正在编写非常通用的跨平台和交叉编译器库。



### 16.2.18 <version>

<version>可用于查询正在使用的 C++ 标准库的相关实现信息。<version>头文件提供的具体内容取决于库实现。下面的内容可能会被公开：

- 版本号
- 发布日期
- 版权声明

此外，<version>头文件还公开了上一节讨论的所有标准库特性测试宏。



### 16.2.19 源位置

C++20 添加了一个名为 std::source\_location 的类，定义在<source\_location>头文件中。它可以用来查询有关源代码的信息，例如文件名、函数名、行号和列号，并可用于替换旧的 C 风格宏 \_\_FILE\_\_ 和 \_\_LINE\_\_。示例中的用例是在记录消息或抛出异常时提供源代码信息。第 14 章给出了这两种用例的示例。

### 16.2.20 容器

标准库提供了常用数据结构(例如链表和队列)的实现。当使用 C++ 时，不需要自己编写这类数据结构。数据结构的实现使用了一个称为容器的概念，容器中保存的信息称为元素，保存信息的方式能够正确地实现数据结构(链表和队列等)。不同数据结构有不同的插入、删除和访问行为，性能特性也不同。重要的是要熟悉可用的数据结构，这样才能在面对特定任务时选择最合适的数据结构。

标准库中的所有容器都是类模板，因此可通过这些容器保存任意类型的数据，从内建的 int 和 double 等类型到自定义的类。每个容器实例都只能保存一种类型的对象，也就是说，这些容器都是同构集合(homogeneous collection)。如果需要大小可变的异构集合(heterogeneous collection)，可将每个元素包装在 std::any 实例中，并将这些实例存储在容器中。另外，可在容器中存储 std::variant 实例。如果所需的不同类型的数量受限，并且是在编译期已知，那么可以使用 variant。any 和 variant 都是在 C++17 中引入的，详情请参阅第 24 章。如果需要 C++17 之前的异构集合，可创建具有多个派生类的类，每个派生类可包装所需类型的对象。

#### 注意：

C++ 标准库容器是同构的：在每个容器中只允许一种类型的元素。

注意，C++ 标准定义了每个容器和算法的接口(interface)，但没有定义实现。因此，不同的供应商可以自由提供不同的实现。然而，作为接口的一部分，标准还定义了性能需求，实现必须满足性能需求。

下面概述了标准库中可用的几个容器。

#### 1. vector

<vector>头文件定义了 vector，vector 保存了元素序列，提供对这些元素的随机访问。可将 vector 想象为一个元素的数组，当插入元素时，这个数组会动态增长，还提供了一些边界检查功能。与数组一样，vector 中的元素保存在连续内存中。

**注意：**

C++ 中的 `vector` 相当于动态数组：这个数组会随着所保存元素数目的变化自动增长或收缩。

`vector` 能够在 `vector` 尾部快速地插入和删除元素(摊还常量时间，amortized constant time)。摊还常量时间指的是大部分插入操作都是在常量时间内完成的( $O(1)$ )，第 4 章解释了大  $O$  表示法)。然而，有时 `vector` 需要增长大小以容纳新元素，此时的复杂度为  $O(N)$ 。这个结果的平均复杂度为  $O(1)$ ，或称为摊还常量时间。第 18 章“标准库容器”将讨论相关细节。`vector` 其他部位的插入和删除操作比较慢(线性时间)，因为这种操作必须将所有元素向上或向下挪动一个位置，为新元素腾出空间，或填充删除元素后留下的空间。与数组一样，`vector` 提供对任意元素的快速访问(常量时间)。

虽然在 `vector` 中插入和删除元素需要上下移动其他元素，但应将 `vector` 用作默认容器！即使在中部插入和删除元素，`vector` 也比链表等更快。原因在于，`vector` 在内存中连续存储，而链表却分散在内存中。计算机可极快地处理连续数据，这样，`vector` 操作的速度更快。仅当性能分析器的结果显示链表比 `vector` 更快时，才应当使用链表。

**注意：**

应将 `vector` 用作默认容器。

对于在 `vector` 中保存布尔值有一个模板特化 `vector<bool>`。这个模板特化特别对布尔元素进行空间分配的优化，但是，标准并未规定 `vector<bool>` 的实现应该如何优化空间。`vector<bool>` 的特化和本章后面要讨论的 `bitset` 之间的区别在于，`bitset` 容器的大小是固定的，而 `vector<bool>` 的大小可以根据需要自动增大或缩小。

**2. list**

标准库 `list` 是一种双向链表数据结构，在`<list>`头文件中定义。与数组和 `vector` 一样，`list` 保存了元素的序列。然而，与数组或 `vector` 的不同之处在于，`list` 中的元素不一定保存在连续内存中。相反，`list` 中的每个元素都指定了如何在 `list` 中找到前一个和后一个元素(通常是通过指针)，所以得名双向链表。

`list` 的性能特征和 `vector` 完全相反。`list` 提供较慢的元素查找和访问(线性时间)，而找到相应的位置之后，元素的插入和删除却很快(常量时间)。然而，`vector` 通常比 `list` 更快。使用性能分析器(相关讨论请参阅第 29 章)可以确认这一点。

**3. forward\_list**

`<forward_list>` 头文件中定义的 `forward_list` 是一种单向链表，而 `list` 容器是双向链表。`forward_list` 只支持前向迭代，需要的内存比 `list` 少。与 `list` 类似，一旦找到相关位置，`forward_list` 允许在任何位置执行快速插入和删除操作(常量时间)；与 `list` 一样，不能快速地随机访问元素。

**4. deque**

`deque` 是双向队列(double-ended queue)的简称。`deque` 在`<deque>`头文件中定义，它能实现元素的快速访问(常量时间)。在序列的两端还实现了快速插入和删除(常量时间)，但在序列中间插入和删除的速度较慢(线性时间)。`deque` 中的元素在内存中的存储不连续，速度可能比 `vector` 慢。

如果需要在序列两端快速插入或删除元素，还要求快速访问所有元素，那么应该使用 `deque` 而不是 `vector`。然而，很多编程问题并不满足这个要求，因此在大部分情况下，建议使用 `vector`。

## 5. array

<array>头文件定义了 `array`, 这是标准 C 风格数组的替代品。有时可事先知道容器中元素的确切数量, 因此不需要 `vector` 或 `list` 提供的灵活性, `vector` 和 `list` 能动态增长以容纳新元素。`array` 特别适用于大小固定的集合, 而且没有 `vector` 的开销;`array` 实际上是对 C 风格数组的简单包装。使用 `array`(而不是标准的 C 风格数组)有几点好处: `array` 总能知道自己的大小; 不会自动转换为指针类型, 从而避免了某些类型的 bug。`array` 没有提供插入和删除操作, 但大小固定。大小固定的优点是, 允许 `array` 在栈上分配内存, 而不总是像 `vector` 那样需要堆访问权限。与 `vector` 一样, 对元素的访问速度极快(常量时间)。

### 注意:

`vector`、`list`、`forward_list`、`deque` 和 `array` 容器都称为顺序容器(sequential container), 因为它们保存的是元素的序列。



## 6. span

C++20 中引入了 `span`, 并定义在<span>头文件中, `span` 可以用于表示连续数据序列的视图。它可以是只读视图, 也可以是对底层元素具有读/写访问权限的视图。`span` 允许编写单个函数, 该函数可以处理来自 `vector`、数组、C 风格数组等的数据。第 18 章更详细地讨论了 `span`。

### 注意:

例如, 在编写一个接收参数类型为 `const vector<T>&` 的函数时, 考虑替换为接收 `span<const T>` 类型的参数, 这样该函数就可以处理来自 `vector`、`array`、C 风格数组等的数据序列的视图和子视图。

## 7. queue

`queue` 的含义是一队人或一列对象。`queue` 容器在<queue>头文件中定义, 提供标准的先入先出(First In First Out, FIFO)语义。在使用 `queue` 容器时, 从一端插入元素, 从另一端取出元素。插入元素(摊还常量时间)和删除元素(常量时间)的操作都很快。

当需要建模真实世界中的“先入先出”语义时, 应该使用 `queue` 结构。以银行为例, 随着顾客到达银行, 顾客在队伍中等待。当柜员可提供服务时, 柜员首先服务队伍中的下一位顾客, 因此这是一种“先来先服务”的行为。在 `queue` 中保存 `Customer` 对象可实现对银行的模拟。随着顾客到达银行, 将顾客添加到 `queue` 的尾部。当柜员服务顾客时, 首先从队列头部的顾客开始服务。

## 8. priority\_queue

`priority_queue` 也在<queue>头文件中定义, 提供与 `queue` 相同的功能, 但其中的每个元素都有优先级。元素按优先顺序从队列中移除。在优先级相同的情况下, 删除元素的顺序没有定义。对 `priority_queue` 的插入和删除一般比简单的队列插入和删除要慢, 因为只有对元素重排序, 才能支持优先级。

通过 `priority_queue` 可建模“带有意外的队列”。例如, 在前面的银行模拟中, 假设带有企业账号的顾客比普通顾客的优先级高。很多银行都通过两个独立队列来实现这个行为: 一个队列用于企业账号顾客, 另一个队列用于其他所有顾客。企业账号队列中的任何顾客都优先于其他队列中的顾客。不过, 银行也提供单队列的行为, 在这种行为中, 企业顾客可转移到队列中所有非企业顾客前面的位置。在程序中, 可使用 `priority_queue`, 其中顾客具有两种优先级中的一种: 企业顾客和普通顾客。所有的企业顾客在所有普通顾客之前得到服务。

## 9. stack

`<stack>` 头文件定义了 `stack`，它提供标准的先入后出(First-In Last-Out, FILO)语义，这种语义也称为后入先出(Last-In First-Out, LIFO)语义。和 `queue` 一样，在容器中插入和删除元素。然而，在堆栈中，最新插入的元素第一个被移除。向堆栈中添加对象元素时，这个元素下面的所有对象都被遮住了。

`stack` 容器实现了元素的快速插入和删除(常量时间)。如果需要使用 FILO 语义，则应该使用 `stack` 结构。例如，错误处理工具可将错误保存在堆栈中，这样管理员可直接读到最新的一条错误。按 FILO 的顺序处理错误通常更有用，因为更新的错误可能会消除较早的错误。

### 注意：

从技术角度看，`queue`、`priority_queue` 和 `stack` 容器都是容器适配器(adapter)。它们只是构建在某种标准顺序容器(`vector`、`list` 或 `deque`)上的简单接口。

## 10. set 和 multiset

`set` 类模板在`<set>`头文件中定义。顾名思义，标准库中的 `set` 保存的是元素的集合，和数学概念中的集合比较类似：每个元素都是唯一的，在集合中每个元素最多只有一个实例。标准库中的 `set` 和数学中集合的概念有一点区别：在标准库中，元素按照一定的顺序保存。排序的原因是当客户枚举元素时，元素能够以这种类型的 `operator<` 运算符或用户自定义的比较器得到的顺序出现。`set` 提供对数时间的插入、删除和查找操作。这意味着插入和删除操作比 `vector` 快，但比 `list` 慢。查找操作比 `list` 块，但比 `vector` 慢。与前面所讲的一样，在实际运用中，可使用性能分析器确定哪种容器更快。

当需要保证元素顺序，使插入/删除操作数目和查找操作数目接近，并且尽可能优化这两种操作的性能时，应该使用 `set`。例如，一家繁忙书店的库存跟踪程序可使用 `set` 保存图书。库存的图书列表必须在有图书进货或售出时更新，因此插入和删除操作应该快速。客户还需要能查找某本书，因此这个程序还应该提供快速查找功能。

不能修改集合中的元素，因为这会使得元素的顺序无效。如果需要更改一个元素，请首先删除该元素，然后再插入一个具有新值的新元素。

### 注意：

如果需要保持顺序，并且要求插入、删除和查找操作的性能接近，那么应当优先使用 `set` 而不是 `vector` 或 `list`。如果严禁出现重复元素，也应当使用 `set`。

注意，`set` 不允许重复元素。也就是说，`set` 中的每个元素都必须唯一。如果想要存储重复元素，那么必须使用`<set>`头文件中定义的 `multiset`。

## 11. map 和 multimap

`<map>` 头文件定义了 `map` 类模板，这是一个关联数组(associative array)。可将其用作数组，其中的索引可以是任意类型，如 `string`。`map` 保存的是键/值对。`map` 按顺序保存元素，排序的依据是键值而非对象值。`map` 还提供了 `operator[]`，而 `set` 并未提供 `operator[]`。在其他大多数方面，`map` 和 `set` 都是一致的。如果需要关联键和值，就应该使用 `map`。例如，在前面的书店例子中，可能需要将图书保存在 `map` 中，其中键是图书的 ISBN 号，而值是 `Book` 对象，`Book` 对象包含那本书的详细信息。

`multimap` 也在`<map>`头文件中定义，它和 `map` 的关系等同于 `multiset` 和 `set` 的关系。确切地讲，`multimap` 是允许重复键的 `map`。

**注意：**

`set` 和 `map` 容器都是关联容器，因为它们关联了键和值。将 `set` 称为关联容器可能会难以理解，因为在 `set` 中，键本身就是值。这些容器会对元素进行排序，因此将这些容器称为排序或有序关联容器。

**12. 无序关联容器/哈希表**

标准库支持哈希表(hash table)，哈希表也称为无序关联容器(unordered associative container)。有 4 个无序关联容器：

- `unordered_map`/`unordered_multimap`
- `unordered_set`/`unordered_multiset`

前两个容器在`<unordered_map>`中定义，后两个容器在`<unordered_set>`中定义。更贴切的名字应该是 `hash_map` 和 `hash_set` 等。遗憾的是，在 C++11 之前，哈希表并不属于 C++ 标准库的一部分，因此导致很多第三方库在实现哈希表时都使用 `hash` 作为前缀，例如 `hash_map`。因此，C++ 标准委员会决定使用 `unordered` 而不是 `hash` 作为前缀，以避免名称冲突。

这些无序关联容器在行为上和对应的有序关联容器是类似的。`unordered_map` 和标准的 `map` 类似，只不过标准的 `map` 会对元素排序，而 `unordered_map` 不会对元素排序。

这些无序关联容器的插入、删除和查找操作能以平均常量时间完成。最坏情况是线性时间。在无序的容器中查找元素的速度比普通 `map` 或 `set` 中的查找速度快得多，在容器中元素数量特别大的情况下尤其如此。

第 18 章将介绍这些无序关联容器的工作原理，还将介绍哈希表名称的来历。

**13. bitset**

C 和 C++ 程序员经常将一组标志位保存在单个 `int` 或 `long` 中，每个位对应一个标志。程序员通过按位运算符`&`、`|`、`^`、`~`、`<<`和`>>`设置和访问这些位。C++ 标准库提供了 `bitset` 类，这个类抽象了这些位操作，因此再也不需要使用这些位运算符了。

`<bitset>` 头文件定义了 `bitset` 容器，但是这个容器不是常规意义的容器，因为这个容器没有实现某种特定的可插入或删除元素的数据结构；`bitset` 有固定大小。可将 `bitset` 想象为可以读写的布尔值序列。然而，和 C 语言中常规的布尔值读写方法不同，`bitset` 不局限于 `int` 或其他基本数据类型的大小。因此，能操作 40 位的 `bitset`，也能操作 213 位的 `bitset`。`bitset` 的实现会使用实现  $N$  个位所需的足够存储空间，通过 `bitset<N>` 声明 `bitset` 时指定  $N$ 。

**14. 标准库容器小结**

表 16-3 总结了标准库提供的容器。表中使用第 4 章介绍的大  $O$  表示法来表示含  $N$  个元素的容器的性能特性。N/A 表示这个容器的语义中不存在这个操作。

表 16-3 标准库提供的容器

容器名	容器类型	插入性能	删除性能	查找性能
vector	顺序	尾端摊还性能 $O(1)$ ，在其他位置为 $O(N)$	尾端摊还性能 $O(1)$ ，在其他位置为 $O(N)$	$O(1)$

(续表)

容器名	容器类型	插入性能	删除性能	查找性能
list	顺序	在开头和结尾, 以及处于要插入元素的位置时为 $O(1)$	在开头和结尾, 以及处于要删除元素的位置时为 $O(1)$	访问第一个和最后一个元素时为 $O(1)$ , 其他情况下为 $O(N)$
<b>适用情形:</b> 极少使用。除非性能分析器确认在具体使用情形中 list 更快, 否则使用 vector				
forward_list	顺序	在开头, 以及处于要插入元素的位置时为 $O(1)$	在开头, 以及处于要删除元素的位置时为 $O(1)$	访问第一个元素时为 $O(1)$ , 其他情况下为 $O(N)$
<b>适用情形:</b> 极少使用。除非性能分析器确认在具体使用情形中 forward_list 更快, 否则使用 vector				
deque	顺序	在开头和结尾为 $O(1)$ , 其他情况下为 $O(N)$	在开头和结尾为 $O(1)$ , 其他情况下为 $O(N)$	$O(1)$
<b>适用情形:</b> 通常较少使用, 而是改用 vector				
array	顺序	N/A	N/A	$O(1)$
<b>适用情形:</b> 需要使用固定大小的数组来替换标准 C 风格数组时				
queue	容器适配器	取决于底层容器, 底层容器是 list 和 deque 时为 $O(1)$	取决于底层容器, 底层容器是 list 和 deque 时为 $O(1)$	N/A
<b>适用情形:</b> 需要使用 FIFO 结构时				
priority_queue	容器适配器	取决于底层容器, 底层容器是 vector 时为摊还性能 $O(\log(N))$ , 底层容器是 deque 时为 $O(\log(N))$	取决于底层容器, 底层容器是 vector 和 deque 时为 $O(\log(N))$	N/A
<b>适用情形:</b> 需要使用带优先级的 queue 时				
stack	容器适配器	取决于底层容器, 底层容器是 list 和 deque 时为 $O(1)$ , 底层容器是 vector 时为摊还性能 $O(1)$	取决于底层容器, 底层容器是 list、vector 和 deque 时为 $O(1)$	N/A
<b>适用情形:</b> 需要使用 FILO/LIFO 结构时				
set	排序关联	$O(\log(N))$	$O(\log(N))$	$O(\log(N))$
multiset	<b>适用情形:</b> 需要对元素排序的集合, 并且查找、插入和删除性能等同时。需要不含重复元素的集合时使用 set			
map	排序关联	$O(\log(N))$	$O(\log(N))$	$O(\log(N))$
multimap	<b>适用情形:</b> 需要有序集合将值关联到键(即关联数组), 并且查找、插入和删除性能等同时			
unordered_map	无序关联/哈希表	平均情况 $O(1)$ , 最坏情况 $O(N)$	平均情况 $O(1)$ , 最坏情况 $O(N)$	平均情况 $O(1)$ , 最坏情况 $O(N)$
unordered_multimap	<b>适用情形:</b> 需要关联值和键, 查找、插入和删除时间相同, 但不要求对元素排序时。性能可能比普通的 map 要好, 但具体取决于元素本身			
unordered_set	无序关联/哈希表	平均情况 $O(1)$ , 最坏情况 $O(N)$	平均情况 $O(1)$ , 最坏情况 $O(N)$	平均情况 $O(1)$ , 最坏情况 $O(N)$
unordered_multiset	<b>适用情形:</b> 要求元素集合的查找、插入和删除时间相同, 但不要求对元素排序时。性能可能会比普通的 set 要好, 但这取决于元素本身			
bitset	特殊	N/A	N/A	$O(1)$
	<b>适用情形:</b> 需要标志集合时			

注意，从技术角度看字符串也是容器。因此，下面要描述的一些算法也能用于字符串。

#### 注意：

`vector` 应是默认使用的容器！实际上，`vector` 中的插入和删除常常快于 `list` 或 `forward_list`。这是因为现代 CPU 上内存和缓存的工作方式，而 `list` 或 `forward_list` 需要先移动到要插入或删除元素的位置。`list` 或 `forward_list` 的内存可能是碎片化的，所以迭代慢于 `vector`。

### 16.2.21 算法

除容器外，标准库还提供了很多泛型算法的实现。算法指的是执行某项任务时采取的策略，例如排序任务和搜索任务。这些算法也是用函数模板实现的，因此可用于大部分不同类型的容器。注意，算法一般不属于容器的一部分。标准库采取了一种分离数据(容器)和功能(算法)的方式。尽管这种方式看上去违背了面向对象编程的思想，但为了在标准库中支持泛型编程，有必要这么做。正交性(orthogonality)指导原则使算法和容器分离开，(几乎)所有算法都可以用于(几乎)所有容器。

#### 注意：

尽管算法和容器在理论上是无关的，但有些容器以类方法的形式提供了某些算法，因为泛型算法在这些特定类型的容器上表现不佳。例如，`set` 提供了自己的 `find()` 算法，这个算法比泛型的 `find()` 算法要快。如果提供的话，应该使用容器特定的方法形式的算法，因为通常情况下这些算法更高效或更适合当前容器。

注意，泛型算法并不直接对容器进行操作。泛型算法要么使用迭代器(iterator)，要么使用范围，这两者都将在第 17 章“理解迭代器和范围库”进行讨论。

本节概述了标准库中可用的算法，但没有给出所有细节。第 20 章“掌握标准库算法”通过列举一些编码示例来演示如何选择算法。要了解所有可用算法的准确原型，请参阅标准库参考资源。

标准库中大约有 100 种算法，下面将这些算法分为不同的类别。除非特别说明，否则这些算法都在`<algorithm>`中定义。注意当将以下算法用于元素“序列”时，这个序列是通过迭代器向算法呈现的。

#### 注意：

在查看算法列表时，要记住标准库在设计时考虑了一般性情况，这些一般性情况可能永远也用不到，但是一旦需要，就会非常重要。也许不需要每个算法，也不用担心那些比较模糊的参数，因为那些参数是为了满足可能发生的一般性情况。重要的是了解哪些算法是可用的，以备不时之需。

#### 1. 非修改顺序算法

非修改类的算法查找元素的序列，返回一些有关元素的信息。因为是非修改类的算法，所以这些算法不会改变序列中元素的值或顺序。这类算法包含 3 种算法。表 16-4 列出了对不同非修改类算法的概述。有了这些算法后，几乎不再需要编写 `for` 循环以迭代值序列了。

#### 搜索算法

如表 16-4 所示的算法不需要元素是有序的。 $N$  是搜索的序列的大小， $M$  是要查找的模式的大小。

表 16-4 搜索算法

算法名称	算法概要	复杂度
adjacent_find()	查找有两个连续元素相等或匹配谓词的第一个实例	$O(N)$
find()	查找第一个匹配值或使谓词返回 true 的元素	$O(N)$
find_if()		
find_first_of()	与 find() 类似，只是同时搜索多个元素中的某个元素	$O(N*M)$
find_if_not()	查找第一个使谓词返回 false 的元素	$O(N)$
find_end()	在序列中查找最后一个匹配另一个序列的子序列，这个子序列的元素和谓词指定的一致	$O(M*(N-M))$
search()	在序列中查找第一个匹配另一个序列的子序列，这个子序列的元素和谓词指定的一致*	$O(N*M)*$
search_n()	查找前 $n$ 个等于某个给定值或根据某个谓词和那个值相关的连续元素	$O(N)$

\*从 C++17 开始，search() 接收一个可选的附加参数，以指定要使用的搜索算法(default\_searcher、boyer\_moore\_searcher 或 boyer\_moore\_horspool\_searcher)。使用 boyer\_moore 搜索算法，最坏情况下，模式未找到时的复杂度是  $O(N+M)$ ，模式找到时的复杂度为  $O(N*M)$ 。

### 比较算法

标准库提供了表 16-5 中列出的比较算法。这些算法都不要求排序源序列。所有算法的最差复杂度都为线性复杂度。

表 16-5 比较算法

算法名称	算法概要
equal()	检查相应元素是否相等或匹配谓词，以判断两个序列是否相等
mismatch()	返回每个序列中第一个出现的和其他序列中同一位置元素不匹配的元素
lexicographical_compare()	比较两个序列，判断这两个序列的“词典顺序”。将第一个序列中的每一个元素和第二个序列中对应的元素进行比较。如果一个元素小于另一个元素，那么这个序列按照词典顺序在前面。如果两个元素相等，则按顺序比较下一个元素
lexicographical_compare_three_way()	使用 C++20 的三路比较来比较两个序列，判断这两个序列的“字典序顺序”，并返回一个比较类别类型(strong_ordering、weak_ordering 或 partial_ordering)。

### 计数算法

计数算法如表 16-6 所示。

表 16-6 计数算法

算法名称	算法概要
all_of()	如果谓词对序列中的所有元素都返回 true，或者序列为空，则返回 true；否则返回 false
any_of()	如果谓词对序列中的至少一个元素返回 true，则返回 true；否则返回 false
none_of()	如果谓词对序列中的所有元素都返回 false，或者序列为空，则返回 true；否则返回 false
count()	计算匹配某个值或使谓词返回 true 的元素个数
count_if()	

## 2. 修改序列算法

修改算法会修改序列中的一些元素或所有元素。有些修改算法在原位置修改元素，因此原始序列会发生变化。另一些修改算法将结果复制到另一个不同的序列，所以原始序列没有变化。所有这些修改算法的最坏复杂度都为线性复杂度。表 16-7 汇总了这些修改算法。

表 16-7 修改序列算法

算法名称	算法概要
copy()	将一个序列的元素复制到另一个序列
copy_backward()	
copy_if()	将一个序列中谓词返回 true 的元素复制到另一个序列
copy_n()	从一个序列中复制 $n$ 个元素到另一个序列
fill()	将一个序列中的所有元素设置为一个新值
fill_n()	将一个序列中的前 $n$ 个元素设置为一个新值
generate()	调用指定函数，为一个序列中的每个元素生成一个新值
generate_n()	调用指定函数，为一个序列中的前 $n$ 个元素生成一个新值
move()	将一个序列的元素移到另一个序列。这两个算法使用了高效移动语义(请参阅第 9 章)
move_backward()	
remove()	删除所有匹配给定值或使谓词返回 true 的元素，就地删除或将结果复制到另一个不同的序列
remove_if()	
remove_copy()	
remove_copy_if()	
replace()	将匹配给定值或导致谓词返回 true 的所有元素替换为新元素，在原位置替换或将结果复制到新序列
replace_if()	
replace_copy()	
replace_copy_if()	
reverse()	反转序列中元素的顺序，在原位置操作或将结果复制到另一个不同的序列
reverse_copy()	
rotate()	交换序列的前半部分和后半部分，在原位置操作或将结果复制到另一个不同的序列。两个要交换的子序列不一定要一样大
rotate_copy()	
sample()	从序列中选择 $n$ 个随机元素
shift_left()	将序列中的元素按向左或向右移动给定数量的位置。元素被移动到它们的新位置。位于序列两端的 $s$ 元素将被删除。shift_left()返回到新序列末尾的迭代器；shift_right()返回一个指向新序列开头的迭代器
shift_right()	
shuffle()	随机重排元素，打乱元素的顺序。可以指定用于打乱元素顺序的随机数生成器的属性。
random_shuffle()	random_shuffle()在 C++14 之后已经不赞成使用，在 C++17 中被删除
transform()	对序列中的每个元素调用一元函数，或对两个队列中的对应元素调用二元函数。这属于原位 置转换 如果源序列和目标序列相同，那么就在原位置进行转换
unique()	从序列中删除连续出现的重复元素，在原位置删除或将结果复制到另一个不同的序列
unique_copy()	

### 3. 操作算法

操作算法在单独的元素序列上执行函数。C++ 标准库提供了两种操作算法，如表 16-8 所示。它们的复杂度都是线性复杂度，不要求对原始序列进行排序。

表 16-8 操作算法

算法名称	算法概要
for_each()	对序列中的每个元素执行函数。使用首尾迭代器指定该序列
for_each_n()	与 for_each() 类似，但仅处理序列中的前 $n$ 个元素。用开始迭代器以及元素个数( $n$ )指定该序列。

### 4. 交换算法

C++ 标准库提供如表 16-9 所示的交换算法。

表 16-9 交换算法

算法名称	算法概要
iter_swap()	交换两个元素或两个元素的序列
swap_ranges()	

### 5. 分区算法

如果谓词返回 true 的所有元素都在谓词返回 false 的所有元素之前，则序列将根据某个谓词对序列进行分区。序列中不满足谓词的第一个元素称为分区点(partition point)。C++ 标准库提供如表 16-10 所示的分区算法。

表 16-10 分区算法

算法名称	算法概要	复杂度
is_partitioned()	如果谓词返回 true 的所有元素都在谓词返回 false 的所有元素的前面，那么就返回 true	线性复杂度
partition()	对序列进行排序，使谓词返回 true 的所有元素在谓词返回 false 的所有元素之前，不会保留之前元素在每个分区中的顺序	线性复杂度
stable_partition()	对序列进行排序，使谓词返回 true 的所有元素在谓词返回 false 的所有元素之前，同时保留之前元素在每个分区中的顺序	线性对数复杂度
partition_copy()	将一个序列中的元素复制到两个不同的序列中。目标序列的选择依据是谓词返回的结果，即 true 和 false	线性复杂度
partition_point()	返回一个迭代器，使谓词对所有在这个迭代器之前的元素都返回 true，对所有在这个迭代器之后的元素都返回 false	对数复杂度

### 6. 排序算法

C++ 标准库提供了一些不同的排序算法，不同的排序算法有不同的性能保证，如表 16-11 所示。

表 16-11 排序算法

算法名称	算法概要	复杂度
is_sorted()	检查一个序列是否已经排序，或检查哪个子序列已经排序	线性复杂度
is_sorted_until()		
nth_element()	重定位序列中的第 $n$ 个元素，使第 $n$ 个位置的元素就是排好序之后第 $n$ 个位置的元素。该算法会重新安排所有元素，使第 $n$ 个元素前面的所有元素都小于新的第 $n$ 个元素，使第 $n$ 个元素之后的所有元素都大于新的第 $n$ 个元素	线性复杂度
partial_sort()	只排序序列中的一部分元素：只有前 $n$ 个元素(由迭代器指定)排序，其余元素不排序。在原位置排序，或者复制到新的序列	线性对数复杂度
partial_sort_copy()		
stable_sort()	在原位置排序，保留重复元素的顺序或者不保留	线性对数复杂度
sort()		

## 7. 二叉树搜索算法

二叉树搜索算法通常用于已排序的序列。从技术角度看，它们仅要求至少根据要搜索的元素进行分区。这可使用 std::partition() 来完成。排好的序列也满足这个要求。所有这些算法都具有对数复杂度，如表 16-12 所示。

表 16-12 二叉树搜索算法

算法名称	算法概要
lower_bound()	查找序列中不小于(即大于或等于)给定值的第一个元素
upper_bound()	查找序列中大于给定值的第一个元素
equal_range()	返回 pair，其中包含 lower_bound() 和 upper_bound() 的结果
binary_search()	如果在序列中找到给定值，则返回 true；否则返回 false

## 8. 集合算法

集合算法是特殊的修改算法，对序列执行集合操作，如表 16-13 所示。这些算法最适合操作 set 容器的序列，但也能操作大部分容器的排序后序列。

表 16-13 集合算法

算法名称	算法概要	复杂度
inplace_merge()	在原位置将两个排好的序列合并	线性对数复杂度
merge()	合并两个排好的序列，将两个序列复制到一个新的序列	线性复杂度
includes()	确定一个序列中的每个元素是否都在另一个序列中	线性复杂度
set_union()	在两个排好的序列上执行特定的集合操作，将结果复制到第三个排好的序列中	线性复杂度
set_intersection()		
set_difference()		
set_symmetric_difference()		

## 9. 堆算法

堆(heap)是一种标准的数据结构，数组或序列中的元素在其中以半排序的方式排序，因此能够快

速找到“顶部”元素。例如，堆数据结构通常用于实现 priority\_queue。通过使用 6 种算法可以对序列进行堆排序，如表 16-14 所示。

表 16-14 堆算法

算法名称	算法概要	复杂度
is_heap()	检查某个范围内的元素是不是堆	线性复杂度
is_heap_until()	在给定范围的元素堆中查找最大的子范围	线性复杂度
make_heap()	从某个范围的元素中创建堆	线性复杂度
push_heap()	在堆中添加或删除元素	对数复杂度
pop_heap()		
sort_heap()	把堆转换到升序排列的元素范围内	线性对数复杂度

## 10. 最小/最大算法

表 16-15 中的算法提供了寻找最小和最大元素，或者限制值。

表 16-15 最小/最大算法

算法名称	算法概要
clamp()	确保一个值(v)在给定的最小值(lo)和最大值(hi)之间。如果 v < lo，则返回对 lo 的引用；如果 v > hi，则返回对 hi 的引用；否则返回对 v 的引用
min()	返回两个或多个值中的最小值或最大值
max()	
minmax()	以 pair 方式返回两个或多个值中的最小值和最大值
min_element()	返回序列中的最小或最大元素
max_element()	
minmax_element()	找到序列中的最小和最大元素，把结果返回为 pair

## 11. 数值处理算法

<numeric>头文件提供了下述数值处理算法。这些算法都不要求排序原始序列。所有算法的复杂度都为线性复杂度，如表 16-16 所示。

表 16-16 数值处理算法

算法名称	算法概要
iota()	用连续递增的值(以给定值开头)填充序列
adjacent_difference()	生成一个新的序列，其中每一个元素都是原始序列中相邻元素对的后一个与前一个之差(或其他二元操作)
partial_sum()	生成一个新的序列，这个序列中的每个元素是对应元素和原始序列中之前的所有元素的和(或其他二元操作)
exclusive_scan()	类似于 partial_sum()。如果给定的求和操作具有关联性，则 inclusive 扫描与 partial 扫描相同。但是，inclusive_scan()以不确定的顺序求和，而 partial_sum()从左到右求和，因此对于非关联求和操作，前者的结果不是确定的。exclusive_scan()算法也以不确定顺序求和。
inclusive_scan()	对于 inclusive_scan()，第 i 个元素包含在第 i 个和值中，与 partial_sum()相同。对于 exclusive_scan()，第 i 个元素未包含在第 i 个和值中

(续表)

算法名称	算法概要
transform_exclusive_scan()	对序列中的每个元素应用转换，然后执行 exclusive/inclusive 扫描
transform_inclusive_scan()	
accumulate()	“累加”一个序列中所有元素的值。默认行为是计算元素的和，但调用者可以提供不同的二元函数
inner_product()	与 accumulate()类似，但对两个序列操作。对序列中的并行元素调用二元函数(默认做乘法)，通过另一个二元函数(默认做加法)累加结果值。如果序列表示数学矢量，那么这个算法计算矢量的点积
reduce()	与 accumulate()类似，但支持并行执行。reduce()的计算顺序是不确定的，而 accumulate()是从左到右计算。这意味着如果给定的二元操作是非关联的或不可交换的，则前者的行为是不确定的
transform_reduce()	对序列中的每个元素应用转换，然后执行 reduce()

## 12. 置换算法

序列的置换包含相同的元素，但顺序变了。表 16-17 列出了用于置换的算法。

表 16-17 置换算法

算法名称	算法概要	复杂度
is_permutation()	如果某个范围内的元素是另一个范围内元素的转换，就返回 true	二次复杂度
next_permutation()	修改序列，将序列转换为下一个或前一个排列。如果从正确排序的序列开始，则连续调用其中一个或另一个可以获得元素的所有可能的排列。如果没有更多排列，则返回 false	线性复杂度
prev_permutation()		

## 13. 选择算法

一下子出现这么多种不同功能的算法可能让人难以接受。一开始可能还很难知道如何应用这些算法。不过，既然已经了解有哪些选择，就应该能更好地处理程序设计了。后续章节将详细讲解如何在代码中使用这些算法。



### 16.2.22 范围库

C++20 引入了范围(ranges)库，这使得处理元素序列变得更加简单和优雅。前面几节讨论的大多数算法都有对应的范围算法，而不是迭代器算法。范围提供了更好、更容易阅读的语法，并消除了不匹配的开始/结束迭代器的可能性。此外，范围适配器允许你惰性地转换和过滤底层序列，并提供范围工厂来构建范围。

范围库在<ranges>头文件中定义，并位于 std::ranges 名称空间中。第 17 章将讨论范围库。

### 16.2.23 标准库中还缺什么

尽管标准库非常强大，但并不完美。下面列出了标准库缺乏的内容和不支持的功能：

- 在通过多线程同时访问容器时，标准库不能保证任何线程安全。

- 标准库没有提供任何泛型的树结构或图结构。尽管 map 和 set 通常都实现为平衡二叉树，但标准库没有在接口中公开该实现。如果在任务中需要树结构或图结构，例如编写解析器，就需要自己实现或寻找其他库的实现。

记住，标准库是可扩展的。可以编写适用于现有算法和容器的容器及算法。因此，如果标准库没有提供需要的内容，可考虑编写兼容标准库的代码。第 25 章将介绍通过定义算法和自定义容器来自定义和扩展标准库的相关主题。

## 16.3 本章小结

这一章概述了 C++ 标准库，标准库是要在代码中使用的最重要的库。标准库包含了 C 库，还包含了其他很多工具，用于字符串、I/O、错误处理和其他任务。标准库还包含泛型容器和泛型算法。后续章节会更详细地讲解标准库。

## 16.4 练习

通过完成以下习题，可以巩固本章讨论的知识点。所有习题的答案都可扫描封底二维码下载。然而，如果你困在一个练习中，在网站上查看答案之前，请先重新阅读本章的内容，试着自己找到答案。

练习 16-1 C++ 标准库提供了一整套容器供选择。应该选择哪一种容器，为什么？

练习 16-2 map 和 unordered\_map 之间有什么区别？

练习 16-3 C++ 标准库提供的所谓词汇类型有哪些，它们的区别是什么？

练习 16-4 自助餐厅通常有一个装有盘子的弹簧装置。作为客户，从顶部拿走一个盘子。当盘子被清洗干净并准备再次使用时，它们被放置在装置中任何剩余的盘子的顶部。如何在 C++ 中为这样的系统建模？

练习 16-5 什么是分区？

“随着新程序员转向(或返回)C++以应对代码的高要求，而C++本身也在随着C++20成为近十年来最大的C++版本而发展，因此人们非常需要最新的指导。编写C++20代码常常感觉像是在用一种全新的语言编写，所以很高兴看到像Marc这样一位知名且经验丰富的C++培训师带来这本书，帮助大家以全新的视角掌握C++这门焕然一新的语言。”

— 赫伯·萨特

### ★★★★★ C++初学者的卓越教材

——Ettienne Hugo

作为一名在微软工作的专业C++程序员，我强烈向任何想学习C++的人推荐本书。我甚至向我的同事推荐本书，因为大家总需要学习和复习不经常使用的C++部分——本书深入研究了这些内容，真正达到了“高级编程”的水准。

不像大多数其他书停留在基本的语法/概念，本书仅在第1章介绍了基本的语法和概念，然后就推进到成为一名专业C++程序员所需要的高阶内容方面。此外，还向读者介绍了单元测试、编写可移植/高效的代码、软件架构和设计模式这些软件工程实践——正是这些内容确定了专业程序员与中级程序员的不同，即确保你不会写出糟糕的软件设计。最后，如果你考虑申请C++编程工作，本书甚至涵盖了技术面试。

我应该指出，没有一本书可以成为C++领域详尽的参考，但本书肯定会带你在这条路上走得很远，对你不断地进行帮助或提升。

### ★★★★★ C++高级编程的优秀指南

——Lee G

我从20世纪90年代就开始编写C++代码。《C++20高级编程(第5版)》是我读过的第3个版本的C++高级编程。本书介绍的是现代C++20，帮助读者写出更好的现代C++代码，作者Marc是C++社区的专家。你可以在cppcon之类的会议上找到Marc的教学课程，这些课程被录制下来，并在youtube上发布。

使得本书从初学者书籍中脱颖而出的是，阅读本书就像与专家对话，这使它的阅读价值非常高。读完第1章，你就能理解现代C++的力量。如果你曾使用JavaScript、Java或早期版本的C++做过编程，那么你就完全准备好了编写内存安全和高效的现代C++程序。书中还通过很多章节介绍设计、标准库用法和代码示例来巩固你的理解。

感谢Marc为这本阅读愉悦的C++教程及时更新了新版！(注：除了Marc的书和视频，我并不认识他本人)

本书在线资源



扫描下载

清华社官方微信号



扫我有惊喜

ISBN 978-7-302-60213-2



9 787302 602132 >

定价：228.00元(全二册)

WILEY