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

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
质 量 反 馈：010-62772015，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

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)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)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章获得所需的大部分知识。

不管属于哪种情况，都应该具有很好的编程基础。应该知道循环、函数和变量。应该知道如何组织一个程序，而且应该熟悉基本技术，例如递归。应该了解一些常见的数据结构(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中。

有很多好的理由让你努力成为一名专业的 C++程序员，而非只是泛泛了解 C++。了解语言的真正工作原理有助于提升代码质量。了解不同的编程方法论和过程可让你更好地和团队协作。探索可重用的库和常用的设计模式可提升日常工作效率，并帮助你避免白费力气的重复工作。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
```

可使用这个在线参考资料，也可下载其离线版本，在没有连接到互联网时使用。

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

本书正文中提到“标准库参考资料”时，就是指上述 C++参考资料。

下面是其他的优质在线资源：

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

C++核心指南是由 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
```

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

```
isocpp.org/faq
```

这是一个频繁被提问的 C++问题的庞大集合。

```
stackoverflow.com
```

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 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 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++ | Genaral，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I部分
专业的 C++简介

第1章 C++和标准库速成

本章内容

- 简要回顾 C++语言和标准库(Standard Library)最重要的部分以及语法
- 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。

C++20

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 #ifndef [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)	int i {-7}; signed int i {-6}; signed i {-5};
(signed) short (int)	短整型整数(通常是 2 字节)	short s {13}; short int s {14}; signed short s {15}; signed short int s {16};
(signed) long (int)	长整型整数(通常是 4 字节)	long l {-7L};
(signed) long long (int)	超长整型整数，范围取决于编译器，但不低于长整型(通常是 8 字节)	long long ll {14LL};
unsigned (int) unsigned short (int) unsigned long (int) unsigned long long (int)	对前面的类型加以限制，使其值≥0	unsigned int i {2U}; unsigned j {5U}; unsigned short s {23U}; unsigned long l {5400UL}; unsigned long long ll {140ULL};
float	浮点型数字	float f {7.2f};
double	双精度数字，精度不低于 float	double d {7.2};
long double	长双精度数字，精度不低于 double	long double d {16.98L};
char unsigned char signed char	单个字符	char ch {'m'};
char8_t(从 C++20 开始) char16_t char32_t	单个 *n* 位 UTF-*n* 编码的 Uincode 字符，*n* 可以是 8，16，32	char8_t c8 {u8'm'}; char16_t c16 {u'm'}; char32_t c32 {U'm'};
wchar_t	单个宽字符，大小取决于编译器	wchar_t w {L'm'};
bool	布尔类型，取值为 true 或 false	bool b {true};

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 可以用如下的方式初始化：

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

注意：

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

1. 数值极限

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

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

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

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

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

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(enumeration-like)[①]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

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

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

C++20

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要的所有背景知识。如果你已经了解 OOP 的理论，下面的内容将帮助你加速了解(或者回顾)基本的 C++对象语法。

1. 定义类

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

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

这个定义首先声明一个类名，在大括号内声明了类的数据成员(属性)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 ariline_ticket 模块的源文件。

```
module airline_ticket;
```

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

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

第二种方法是将初始化任务放在构造函数体中，如下所示。

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

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

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

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

下面是 AirlineTicket 类的析构函数：

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

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

一些 AirlineTicket 类方法的定义如下所示：

```
double AirlineTicket::calculatePriceInDollars()
{
    if (hasEliteSuperRewardsStatus()) {
        // Elite Super Rewards customers fly for free!
        return 0;
    }
    // The cost of the ticket is the number of miles times 0.1.
    // Real airlines probably have a more complicated formula!
    return getNumberOfMiles() * 0.1;
}
string AirlineTicket::getPassengerName() { return m_passengerName; }
void AirlineTicket::setPassengerName(string name) { m_passengerName = name; }
// 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[①]，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

① 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)[①]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 };
```

① GSL 的仅头文件实现可以在 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。

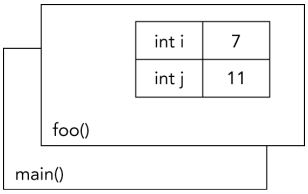

图 1-2 foo()函数中声明了两个整型值

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，可以把它们初始化为空指针 nullptr(详见后面的“空指针常量”小节)。

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

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

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

使用 new 操作符分配内存：

```
myIntegerPointer = new int;
```

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

```
*myIntegerPointer = 8;
```

注意，这并非将 myIntegerPointer 的值设置为 8，在此并没有改变指针，而是改变了指针所指的

内存。如果真要重新设置指针的值，它将指向内存地址 8，这可能是一个随机的无用内存单元，最终会导致程序崩溃。

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。

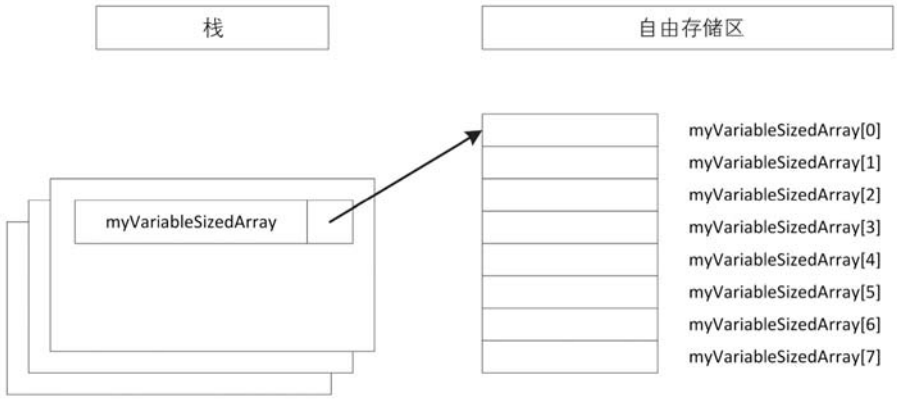

图 1-3　自由存储区

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 { 3.14159265358979238462 };
```

可以将任何变量标记为 const，包括全局变量和类中的数据成员。

const 与指针

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

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

假设你决定对 ip 使用 const。暂时不要考虑这样做的用处，考虑它意味着什么。你是要阻止 ip 变量本身被更改，还是要阻止其指向的值被更改？也就是说，你要阻止第二行还是第三行？

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。

```
const 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
```

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() };
```

刚开始不会觉得 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，你已经为成为专业 C++程序员做好了准备。在开始深入学习本书后面的 C++语言知识时，可查阅本章以回顾需要复习的内容。为了回顾那些被遗忘的概念，可能只需要查看本章的一些示例代码。

你编写的每个程序都必须以这样或那样的方式使用字符串。为此，第 2 章将深入讲解如何在 C++中处理字符串。

1.4　练习

通过做以下习题，可以巩固本章涉及的知识点。所有练习的答案都可扫描封底二维码下载。但是，如果你被某些问题难住，请先重新阅读本章的对应内容，以尝试自己找到答案，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。