

Expert C++

Become a proficient programmer by learning coding best practices with C++17 and C++20's latest features



Packt

www.packt.com

Vardan Grigoryan and Shunguang Wu

Expert C++

Become a proficient programmer by learning coding best practices with C++17 and C++20's latest features

作者: Vardan Grigoryan 和 Shuguang Wu

译者; 陈晓伟

本书主旨

- 通过学习函数式编程、模板和网络等高级概念，设计专业的、可维护的应用。
- 应用设计模式和最佳实践来解决实际问题。
- 通过设计并发数据结构和算法提升应用性能。

本书概述

C++ 经过多年的发展，目前的最新标准为 C++20。自 C++11 以来，C++ 语言不断的增强特性集。在新标准中，您将了解到一系列新特性，如概念、模块、范围和协程。这本书将作为学习错综复杂的语言、技术、C++ 工具和 C++ 20 新特性的指南，同时也会帮助你了解，在构建软件时如何应用他们。

本书将从 C++ 的最新特性开始，然后转向高级技术，如多线程、并发性、调试、监视和高性能编程。本书将深入探讨面向对象的编程原理和 C++ 标准模板库，并展示如何创建自定义模板。之后，将学习不同的方法，比如测试驱动开发 (TDD)、行为驱动开发 (BDD) 和领域驱动设计 (DDD)，然后看看构建专业级应用程序所必需的编码最佳实践和设计模式。本书的最后，有关于人工智能和机器学习的 C++ 最新进展的内容。

这本书的末尾，还有实际应用程序开发方面的专业知识，包括设计复杂软件的过程。

将会学到

- 了解内存管理和 C++ 底层编程，编写安全稳定的应用程序。
- 了解 C++20 的新特性，如模块、概念、范围和协程。
- 熟悉调试和测试技术，减少程序中的问题。
- 使用 Qt5 设计带 GUI 的程序。
- 使用多线程和并发性可以使程序运行得更快。
- 使用 C++ 的面向对象的功能开发高端游戏。
- 使用 C++ 探索人工智能和机器学习。

目标读者

这本书是为有经验的 C++ 开发人员准备的，能将他们现有的知识进行升级，并完善在构建专业级应用程序方面的技能。

作者简介

Vardan Grigoryan 是一名高级后端工程师和 C++ 开发者，拥有 9 年以上的开发经验。Vardan 以 C++ 开发人员的身份开始他的职业生涯，然后转到服务器端后端开发领域。在设计可伸缩的后端架构时，总是在耗时敏感的关键部分使用 C++。Vardan 喜欢在更深的层面上处理计算机系统和程序结构，通过对现有解决方案的详细分析和对复杂系统的精心设计，可以实现真正的卓越编程。

Shunguang Wu 是美国约翰霍普金斯大学应用物理实验室高级专业人员，分别在西北大学和美国莱特州立大学获得理论物理和电气工程博士学位。早期职业生涯中，在非线性动力学、统计信号处理和计算机视觉领域发表了大约 50 篇评论期刊论文。与 C++ 的邂逅是在 20 世纪 90 年代末的本科教学中，从那时起，他就一直在学术和工业实验室使用 C++ 设计和开发大量的研究和开发。这些项目都是跨平台项目，主要是 Windows 和 Linux 平台。

书评人简介

Lou Mauget 在密歇根州立大学 (Michigan State University) 主修物理时，使用软件设计了回旋加速器。之后，在 IBM 工作了 34 年，目前是堪萨斯州利伍德的 Keyhole 软件公司的顾问。Lou 会使用 C++、Java、JavaScript、Python 和新语言进行了编程，几乎是个语言通。其目前的关注的领域有，响应式函数编程、容器、Node.js、NoSQL、地理空间系统、移动端，以及任何新的语言或框架。与其他人合著了三本计算机相关的书籍。他编写了两个 IBM DeveloperWorks XML 教程，并与其他人合作为 IBM 编写几个 J2EE 认证测试。并且，他还是 Packt Publishing 等公司的书评人。

Scott Hutchinson 在加州奥克斯纳德领导着一个 C++ 和 F# 开发团队。做了几年 VB/VBA 开发人员之后，他在 2002 年开始使用.NET 框架。2016 年之后，他的大部分开发工作都用 C++ 完成。他是 [F# track on Exercism](#) 项目的导师，并在工作中作为团队教授 F# 函数式编程。他的主要关注函数式编程和机器学习。并且，他在假期时，会常在南加州的山区进行徒步旅行。

本书相关

- github 翻译地址：<https://github.com/xiaoweiChen/Expert-Cpp>
- 英文原版 PDF：<https://zh.lib.us/book/5537006/2e05c8>

目录

感谢我的母亲卡琳，还有我的小公主莱娅，感谢他们的鼓励和支持。

-Vardan Grigoryan

致敬我的妻子文，以及我的儿子贾斯汀和扎卡里。

- Shunguang Wu

这本书将为提供关于 C++17 和 C++20 标准的细节，以及如何编译、链接和执行。还会介绍内存管理是如何工作的，关于内存管理问题的最佳实践，以及相关的类是如何实现的。还有，编译器如何优化代码，以及编译器在支持类继承、虚函数和模板方面的方法。

并告诉读者如何使用内存管理、面向对象编程、并发和设计模式来创建实际的应用。

读者将了解数据结构和算法的内部细节，了解如何衡量和比较它们，并针对问题选择最适合特定的方法。

本书将帮助读者将系统设计和设计模式融入到 C++ 应用中。

另外，还介绍了人工智能，包括使用 C++ 编程语言进行机器学习的基础知识。

最后，读者应该能使用高效的数据结构和算法，设计实际的架构、可扩展的 C++ 应用程序。

目标读者

本书适合于探究语言和程序结构相关细节的 C++ 开发人员，或者尝试深入研究程序的本质来提高自己专业知识体系结构的读者。还有，那些愿意使用 C++17 和 C++20 的新特性（高效数据结构和算法）的开发人员。

章节概要

第 1 章，构建 C++ 应用，包括对 C++ 的介绍，应用程序，以及最新的语言标准。本章还对 C++ 涉及的主题进行了很好的概述，并介绍了代码编译、链接和执行。

第 2 章，C++ 底层编程，重点讨论 C++ 的数据类型、数组、指针、指针寻址和操作，以及条件、循环、函数、函数指针和结构的底层细节。本章还介绍了结构体 (struct)。

第 3 章，面向对象编程，深入研究类和对象的结构，以及编译器如何实现对象生存周期。在本章最后，读者将了解继承函数和虚函数的实现细节，以及 C++ 中面向对象的内部细节。

第 4 章，了解并设计模板，介绍 C++ 模板、模板函数示例、模板类、模板特化和模板元编程。特性和元编程将为 C++ 带来魔法般的效果。

第 5 章，内存管理和智能指针，深入研究内存的相关内容，包括分配和管理的细节，以及使用智能指针来避免内存泄漏。

第 6 章，挖掘 STL 中的数据结构和算法，介绍数据结构以及其 STL 实现。本章还包括数据结构的比较和与对其实现例程的讨论。

第 7 章，函数式编程，着重于函数式编程，这是一种不同的编程范式，允许读者关注代码的“函数式”结构，而不是“物理”结构。掌握函数式编程为开发人员提供了一种新技能，有助于提供更好的问题解决方案。

第 8 章，并发和多线程，如何利用并发性使程序运行得更快。当算法的高效实现遇到性能瓶颈

时，并发就会有用武之地。

第 9 章, 设计并发式数据结构, 重点介绍如何利用数据结构和并发性, 来设计基于锁和无锁的并发数据结构。

第 10 章, 设计实际程序, 通过使用设计模式, 将前面章节中获得的知识整合到设计健壮的实际应用程序中。本章还包括, 通过设计 Amazon 的克隆版来理解和应用领域驱动设计。

第 11 章, 使用设计模式设计策略游戏, 通过使用设计模式和最佳实践, 将前面章节中获得的知识整合到策略游戏中。

第 12 章, 网络和安全, 在 C++ 中的网络编程和如何利用网络编程技能建立一个 dropbox 后端副本。本章还讨论了如何进行最佳实践。

第 13 章, 调试与测试, 着重于调试 C++ 应用程序和最佳实践, 以避免代码中的错误, 应用静态代码分析减少程序中的问题, 引入测试驱动开发和行为驱动开发。本章还讨论了行为驱动开发和 TDD 用例之间的区别。

第 14 章, 使用 Qt 开发图形界面, 介绍 Qt 库及其主要组件。本章还包括对 Qt 跨平台特性的理解进行了介绍, 并通过构建一个简单的桌面客户端继续 dropbox 的例子。

第 15 章, 使用 C++ 进行机器学习, 简要介绍了人工智能的概念和该领域的最新发展。本章还介绍了机器学习的相关知识, 如回归分析和聚类, 以及如何建立一个简单的神经网络。

第 16 章, 实现一个交互式搜索引擎, 应用前面所有章节的知识, 设计一个高效的基于对话框的搜索引擎, 可以通过询问 (和学习) 用户的相应问题来找到正确的文档。

书中程序的应用环境

基本的 C++ 经验包括熟悉内存管理、面向对象编程、基本的数据结构和算法, 要能了解这些就最好了。如果你想要了解复杂程序是如何工作的, 并且渴望理解编程概念的细节和 c++ 应用程序设计的最佳实践, 那么你绝对应该要阅读本书。

本书所需要的软件和硬件	所需操作系统
g++ 编译器	Ubuntu Linux 最好

您还需要在您的计算机上安装 Qt。详情见相关章节。

写这本书的时候, 并不是所有的 C++ 编译器都支持 C++20 的新特性, 可以考虑使用最新版本的编译器来测试本书介绍的新特性。

[下载示例源码](#)

您可以从您的帐户下载本书的示例代码文件 www.packt.com. 如果在别处买到这本书，可以访问 www.packt.com/support，进行注册后，文件会通过电子邮件直接发给你。

你可以通过以下步骤下载代码文件：

1. 在 www.packt.com 登录或注册账号。
2. 选择 **SUPPORT** 页面。
3. 点击 **Code Downloads & Errata**。
4. 在 **Search** 框中输入书名后，根据屏幕上的指示进行操作。

下载文件后，请确保您使用最新版本的解压包：

- Windows 下 WinRAR/7-Zip
- Mac 下 Zipeg/iZip/UnRarX
- Linux 下 7-Zip/PeaZip

下载彩图

我们还提供了一个 PDF 文件，其中有本书中使用的屏幕截图/图表的彩色图像。下载地址：https://static.packt-cdn.com/downloads/9781838552657_ColorImages.pdf

约定惯例

本书中有许多文本约定。

CodeInText: 表示文本中的代码字、数据库表名、文件夹名、文件名、文件扩展名、路径名、虚拟 url、用户输入和 Twitter 句柄。下面是一个例子：“前面的代码声明了两个带有预赋值的 **readonly** 属性。”

代码块样式如下：

```
Range book = 1..4;  
var res = Books[book];  
Console.WriteLine($"Element of array using Range: Books[book] => Books[book]");
```

当我们希望提请您注意代码块的特定部分时，相关的行或项将以粗体显示：

```
private static readonly int num1=5;  
private static readonly int num2=6;
```

任何命令行输入或输出都是这样写的：

dotnet –info

Bold: 指示一个新的术语，一个重要的词，或你在屏幕上看到的词。例如，菜单或对话框中的单词出现在这样的文本中。这里有一个例子：“Select **System info** from **Administration panel**”。



表示警告或重要提示。



表示提示和技巧。

1 C++ 编程基础

本节将学习 C++ 编译和链接的细节，并深入了解面向对象编程 (OOP)、模板和内存管理的细节。

本节包括以下章节：

- 第 1 章，构建 C++ 应用
- 第 2 章，C++ 底层编程
- 第 3 章，面向对象编程
- 第 4 章，了解并设计模板
- 第 5 章，内存管理和智能指针

第 1 章：构建 C++ 应用

不同编程语言的执行模型也不同，最常见的便是解释语言和编译语言。编译器将源代码翻译成机器代码，计算机可以在没有中间系统支持的环境下运行。另外，解释性语言代码需要支持系统、解释器和虚拟环境才能工作。

C++ 是编译语言，所以会比解释型程序运行得更快。但 C++ 程序需要针对每个平台进行编译，但解释型程序可以跨平台运行。

我们将讨论程序构建的细节，从源代码阶段开始——由编译器完成——到可执行文件（编译器的输出）结束。还会去了解，为什么为一个平台构建的程序不能在另一个平台上运行。

本章将讨论以下主题：

- 介绍 C++20。
- C++ 预处理的细节。
- 源代码的（底层）编译。
- 了解连接器及其功能。
- 加载和运行可执行文件的过程。

编译器要求

g++ 编译器需要添加编译选项 `-std=c++2a` 来编译本章的代码。可以从这里获取本章的源码文件：<https://github.com/PacktPublishing/Expert-CPP>

介绍 C++20

C++ 经过多年的发展，目前发展到 C++ 20。自 C++ 11 以来，C++ 标准已经对语言的特性集进行了极大地扩展。现在，让我们来看看 C++ 20 标准中哪些值得关注的特性。

概念 (Concepts):

概念是 C++ 20 的主要特性之一，它为类型提供了一组需求。概念的基本思想是模板参数的编译时进行验证，例如：要指定模板实参必须有默认构造函数，可以使用 `default_constructible` 概念，方法如下：

```
1 template <default_constructible T>
2 void make_T() { return T(); }
```

上面的代码中，我们忽略了 `typename` 关键字，设置一个概念来描述模板函数的形参 `T`。

可以说概念是描述其他类型的类型——可称为元类型。允许在编译时验证模板参数，以及基于类型属性的函数调用。我们将在第 3 章和第 4 章中详细讨论这些概念。

协程 (Coroutines):

协程是能够在执行点停止，并在稍后恢复的特殊函数。协程用以下关键字进行扩展：

1. `co_await` 暂停协程的执行。

2. `co_yield` 暂停协程的执行，同时返回一个值。

3. `co_return` 类似于 `return`，完成协程时返回一个值。举个栗子：

```
1 generator<int> step_by_step(int n = 0) {
2     while (true) {
3         co_yield n++;
4     }
5 }
```

协程与 `promise` 对象相关联，`promise` 可以存储协程的状态并发出警报。我们将在第 8 章中更深入地研究协程。

范围 (Ranges):

范围库提供了一种处理元素范围的新方法。要使用它们，首先要包含 `<ranges>` 头文件。来看一个例子，范围是一个有开始和结束的元素 `vector`，提供了一个 `begin` 迭代器和一个 `end` 哨兵：

```
1 import <vector>
2 int main()
3 {
4     std::vector<int> elements{0, 1, 2, 3, 4, 5, 6};
5 }
```

带有范围适配器 (`|` 操作符) 的范围支持处理一系列元素的功能。看下代码：

```
1 import <vector>
2 import <ranges>
3 int main()
4 {
5     std::vector<int> elements{0, 1, 2, 3, 4, 5, 6};
6     for (int current : elements | ranges::view::filter([](int e) { return
7         e % 2 == 0; }))
8     {
9         std::cout << current << " ";
10    }
11 }
```

前面的代码中，使用 `ranges::view::filter()` 过滤偶数。请注意 `|` 可以在不同的 `vector` 间使用。我们将在第 7 章中讨论范围及其强大的特性。

更多 C++20 的特性

C++ 20 是一个大版本，它包含了许多更加复杂和灵活的特性。概念、范围和协程是本书讨论的众多特性中部分。

最受期待的特性 (之一) 是模块，它提供了声明模块，以及在这些模块中导出类型和值的能力。可以将模块视为带有包含保护头文件，也就是头文件的改进版本。本章将介绍 C++20 中模块的特性。

除了 C++ 20 中添加的一些特性外，我们还将在本书中讨论其他一些特性：

- 超级操作符: `operator<=>()`。现在可以利用 `<=>()` 来控制操作符重载的冗长程度。
- `constexpr` 在这门语言中越来越常见。C++ 20 现在添加了 `consteval` 函数、`constexpr std::vector` 和 `std::string` 等类型。
- 数学常量, 如 `std::number::pi` 和 `std::number::log2e`。
- 线程库的更新, 包括停止令牌和加入线程。
- 概念迭代器。
- 仅移动视图和其他功能。

为了更好地理解一些新特性并深入到该语言的本质, 我们将从以前的版本开始介绍该语言的核心。这将有助于我们找到新特性相对于旧特性的更好用法, 也将有助于支持历史遗留的 C++ 代码。现在让我们从理解 C++ 应用程序的构建开始。

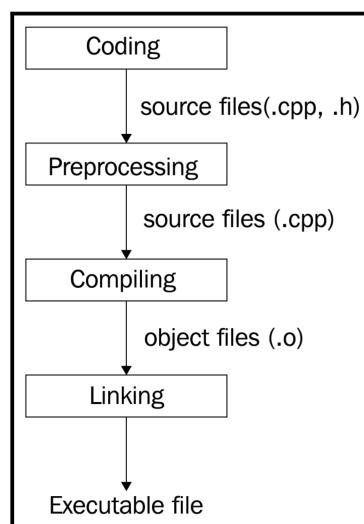
构建并运行程序

可以使用任何文本编辑器来编写代码, 因为代码就是文本。可以在简单的文本编辑器(如 Vim)和高级集成开发环境(IDE)(如 MS Visual Studio)之间自由选择。情书和源代码的唯一区别是后者可能由一种称为编译器的特殊程序解释(虽然情书不能被编译成程序, 但它可能会让你紧张不安)。

为了区分文本文件和源代码, 使用文件扩展名对二者进行区分。C++ 源码文件扩展为.cpp 和.h(可能偶尔也会遇到.cxx 和.hpp)。在讨论细节之前, 请将编译器视为一种将源代码转换为可运行程序(即可执行文件)的工具, 而源代码生成可执行文件的过程称为编译。编译 C++ 程序是生成机器码的一系列复杂任务, 机器码是计算机可以看得懂的语言。

通常, C++ 编译器会解析和分析源码, 然后生成中间码, 对其进行优化, 最后在目标文件中生成机器码。读者们可能见过目标文件了, 它们有各自的扩展名, Linux 中的.o 和 Windows 中的.obj。所创建的目标文件不仅包含计算机可以运行的机器码, 编译通常涉及几个源文件, 编译每个源文件会生成一个目标文件。然后, 这些目标文件通过链接器链接在一起, 形成一个的可执行文件。链接器使用存储在目标文件中的附加信息, 来正确地链接它们(链接将在本章后面讨论)。

下面的图表描述了程序构建各个阶段:



C++ 应用程序构建过程包括三个主要步骤: 预处理、编译和链接。这些步骤使用不同的工具完成, 但是编译器将它们封装在一个工具中, 从而为程序员提供了更直接的接口。

生成的可执行文件保存在计算机的硬盘驱动器上，运行时会复制到主存 RAM 中，复制是由另一个名为加载器的工具完成。加载器是操作系统的一部分，它知道应该从可执行文件的内容中复制什么内容，以及在哪里复制。并且，加载的可执行文件不会从硬盘上删除。

程序的加载和运行由操作系统 (OS) 完成，操作系统管理程序的执行，先执行优先级高的程序，完成后卸载程序等工作。程序的运行副本称为进程，进程是可执行文件的实例。

预处理

预处理器对源文件进行处理，使它们为编译做好准备。预处理器使用预处理器指令，比如 `#define`、`#include` 等等。指令不代表程序语句，但它们是预处理器命令，告诉预处理器如何处理源文件的文本。编译器无法识别这些指令，因此无论何时在代码中使用预处理器指令，预处理器都会在代码开始实际编译之前解析它们。例如，以下代码将在编译器开始编译之前修改：

```
1 #define NUMBER 41
2 int main() {
3     int a = NUMBER + 1;
4     return 0;
5 }
```

使用 `#define` 指令的定义称为宏。经过预处理后，编译器得到转换后的源文件如下：

```
1 int main() {
2     int a = 41 + 1;
3     return 0;
4 }
```

预处理器只处理文本，不关心语言规则或语法。使用预处理器指令，特别是宏定义，如前面的例子中，`#define NUMBER 41` 很容易出错，除非预处理器只是简单地将 `NUMBER` 替换为 `41`，而没有将 `41` 解释为整数。对于预处理器，以下两行都是有效的：

```
1 int b = NUMBER + 1;
2 struct T {};
3 T t = NUMBER; // preprocessed successfully, but compile error
```

预处理后的代码：

```
1 int b = 41 + 1
2 struct T {};
3 T t = 41; // error line
```

编译器开始编译时，发现 `t = 41` 是错误的，因为没有从'int' 到' T' 的转换。

使用语法正确，但有逻辑错误的宏非常危险：

```
1 #define DOUBLE_IT(arg) (arg * arg)
```

预处理器将用 `(arg * arg)` 替换任何出现的 `DOUBLEIT(arg)`，因此下面的代码将输出 16：

```
1 int st = DOUBLE_IT(4);
2 std::cout << st;
```

编译器接收到的代码如下所示：

```
1 int st = (4 * 4);  
2 std::cout << st;
```

当使用复杂表达式作为宏的参数时，会出现问题：

```
1 int bad_result = DOUBLE_IT(4 + 1);  
2 std::cout << bad_result;
```

这段代码期望输出 25，但预处理程序除了文本处理之外什么都不做，所以会像这样替换宏：

```
1 int bad_result = (4 + 1 * 4 + 1);  
2 std::cout << bad_result;
```

输出是 9，而不是期望的 25。

对宏定义进行修正，需要在宏参数周围加上括号：

```
1 #define DOUBLE_IT(arg) ((arg) * (arg))
```

现在预处理后的代码如下：

```
1 int bad_result = ((4 + 1) * (4 + 1));
```

强烈建议在合适的情况下使用 const 声明，而非宏定义。



经验法则：避免使用宏定义。宏易于出错，C++ 提供的构造方式可以不使用宏。

如果使用 constexpr 函数，则会在编译时检查类型并处理，使用上例：

```
1 constexpr int double_it(int arg) { return arg * arg; }  
2 int bad_result = double_it(4 + 1);
```

使用 constexpr 说明符可以在编译时计算函数的返回值（或变量的值）。有数字定义的例子最好使用 const 变量：

```
1 const int NUMBER = 41;
```

头文件

预处理器最常见的用法是 #include 指令，用于在源代码中包含头文件。头文件包含函数、类等定义：

```
1 // file: main.cpp  
2 #include <iostream>  
3 #include "rect.h"  
4 int main() {  
5     Rect r(3.1, 4.05)  
6     std::cout << r.get_area() << std::endl;  
7 }
```

假设头文件 rect.h 的定义如下：

```

1 // file: rect.h
2 struct Rect
3 {
4     private:
5     double side1_;
6     double side2_;
7     public:
8     Rect(double s1, double s2);
9     const double get_area() const;
10 };

```

包含在 rect.cpp 中:

```

1 // file: rect.cpp
2 #include "rect.h"
3 Rect::Rect(double s1, double s2)
4 : side1_(s1), side2_(s2)
5 {}
6 const double Rect::get_area() const {
7     return side1_ * side2_;
8 }

```

预处理器检查 main.cpp 和 rect.cpp 之后，其会将 #include 替换为相应的 iostream 头文件中的内容，并将 rect.h 的内容替换到 main.cpp 和 rect.cpp 中。C++17 引入了 __has_include 预处理器常量表达式。__has_include 如果找到指定名称的文件，则计算结果为 1，否则为 0：

```

1 #if __has_include("custom_iostream.h")
2 #include "custom_iostream.h"
3 #else
4 #include <iostream>
5 #endif

```

声明头文件时，强烈建议使用包含保护 (include-guards) (#ifndef, #define, #endif) 方式，以避免多重声明。同样，这些也是预处理器指令，以避免以下情况：

```

1 // file: square.h
2 #include "rect.h"
3 struct Square : Rect {
4     Square(double s);
5 };

```

在 main.cpp 中同时包含 square.h 和 rect.h 会导致包含 rect.h 两次：

```

1 // file: main.cpp
2 #include <iostream>
3 #include "rect.h"
4 #include "square.h"
5 /*
6     preprocessor replaces the following with the contents of square.h
7 */

```

```
8 // code omitted for brevity
```

预处理后，编译器将接收到如下的 main.cpp:

```
1 // contents of the iostream file omitted for brevity
2 struct Rect {
3     // code omitted for brevity
4 };
5 struct Rect {
6     // code omitted for brevity
7 };
8 struct Square : Rect {
9     // code omitted for brevity
10};
11 int main() {
12     // code omitted for brevity
13}
```

然后，编译器将报出一个错误，因为它遇到了两个 Rect 类型的声明。头文件应该通过以下方式使用包含保护来防止多重包含:

```
1 #ifndef RECT_H
2 #define RECT_H
3 struct Rect { ... }; // code omitted for brevity
4 #endif // RECT_H
```

当预处理器第一次遇到头文件时，RECT_H 没有定义，在 #ifndef 和 #endif 之间的语句都会进行处理，包括 RECT_H 的定义。当预处理器第二次在同一源文件中包含同一头文件时，因为 RECT_H 已经定义，所以会省略其中的内容。

包含保护是控制源文件部分编译的指令的一部分。所有的条件编译指令为 #if、#ifdef、#ifndef、#else、#elif 和 #endif。

条件编译在许多情况下非常有用，可以在调试模式下记录函数调用。在发布程序之前，建议对程序进行调试，并针对逻辑缺陷进行测试。你可能想看看调用某个函数后代码中会发生什么，例如:

```
1 void foo() {
2     log("foo() called");
3     // do some useful job
4 }
5 void start() {
6     log("start() called");
7     foo();
8     // do some useful job
9 }
```

每个函数会调用 log() 函数，其实现如下:

```
1 void log(const std::string& msg) {
2 #if DEBUG
3     std::cout << msg << std::endl;
4 #endif
```

5 }

如果定义了 DEBUG, log() 函数将打印 msg。如果你编译的项目启用了 DEBUG(使用编译器标记, 例如 g++ 中的-D), 那么 log() 函数将打印传递给它的字符串, 否则什么都不做。

C++20 中的模块

模块避免了头文件中恼人的包含保护问题, 现在可以摆脱预处理宏。模块包含有两个相关的关键字, import 和 export。要使用一个模块, 我们需要 import。要声明带有导出属性的模块, 我们使用 export。在列出模块的好处前, 先看一个简单的使用示例。下面的代码声明了一个模块:

```
1 export module test;
2 export int twice(int a) { return a * a; }
```

第一行声明了名为 test 的模块。接下, 声明 twice() 函数, 并将其设置为 export。这意味着可以也有未导出的函数和其他实例, 这些未导出的部分是模块私有的。通过导出, 可以将其设置为模块的公共部分。要使用模块, 可参照下面的代码:

```
1 import test;
2 int main()
3 {
4     twice(21);
5 }
```

模块是 C++ 中一个期待已久的特性, 它在编译和维护方面提供了更好的性能。以下特性使模块比常规头文件的表现得更好:

- 一个模块只导入一次, 类似于自定义语言实现所支持的预编译头文件。这大大减少了编译时间。未导出的元素对导入模块的单元没有影响。
- 模块允许选择哪些单元导出, 哪些不导出, 从而表达代码的逻辑。模块可以绑定到更大的模块中。
- 摆脱前面描述的包含安全之类的工作区。可以以任何顺序导入模块, 不再需要考虑宏的重新定义。

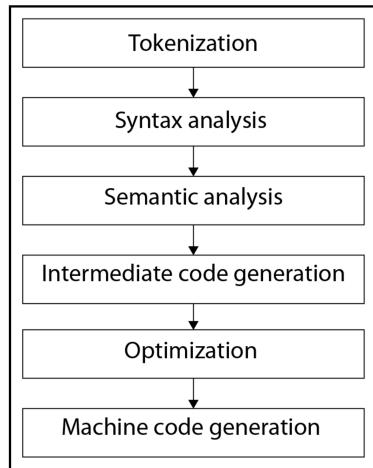
模块可以与头文件一起使用。我们可以在同一个文件中导入和包含头文件, 如下面的例子所示:

```
1 import <iostream>;
2 #include <vector>
3 int main()
4 {
5     std::vector<int> vec{1, 2, 3};
6     for (int elem : vec) std::cout << elem;
7 }
```

创建模块时, 可以自由地导出模块接口文件中的元素, 并将实现移动到其他文件中。逻辑与管理的方式.h 和.cpp 文件相同。

编译阶段

C++ 编译过程包括几个阶段。其中一些阶段用于分析源代码，其他阶段用于生成和优化目标机器代码。下面的图表显示了编译的阶段：



让我们详细地来了解每一个阶段。

符号化

编译器的分析阶段旨在将源代码分割成可符号化的小单元。一个符号可以是一个单词，也可以只是一个操作符，比如 `=`(等号)。符号是源代码中的原子单元，为编译器带来有意义的值，例如：表达式 `int a = 42;` 将被分为符号 `int, a, =, 42, ;`。表达式不用空格分隔，下面的表达式也会分割成相同的标记 (但建议不要忘记操作数之间的空格)：

```
1 int a=42;
```

使用正则表达式的复杂方法，将源码分割成符号的，这个过程称为“词法分析”，或“符号化”(分为符号)。对于编译器来说，使用符号化的输入，是构建用于分析代码语法的内部数据结构的更好方法。

语法分析

当谈到编程语言的编译时，我们通常区分两个术语：语法和语义。语法是代码的结构，定义了符号组合结构的规则，例如：`day nice` 在英语中是一个语法正确的短语，因为它的任何一个标记都不包含错误。另外，语义关注的是代码的实际意义，也就是说 `day nice` 在语义上是不正确的，应该改为 `a nice day`。

语法分析是源码分析的关键部分，即使符号符合一般语法规则，也要对其进行语法和语义分析。以下代码为例：

```
1 int b = a + 0;
```

这对我们来说可能没有意义，因为向变量添加 0 不会改变它的值，但编译器并不寻找逻辑意义——它寻找代码的语法正确性 (缺少分号、缺少闭括号等)。编译的语法分析阶段检查代码的语法正确性。词法分析将代码分成符号，语法分析检查符号语法的正确性。如果我们遗漏了一个分号，上述表达式将产生语法错误：

```
1 int b = a + 0
```

g++ 将会报出一个错误: `expected ';' at end of declaration.`

语义分析

如果前面的表达式是这样的 `b = a + 0;` 时, 编译器将把它分成标记为 `it`、`b`、`=` 和其他。我们知道有些是未知的, 但对编译器来说, 这是没问题的。不过, 这将导致在 g++ 中的编译错误 `unknown type name "it"`。寻找表达式背后的含义, 才是语义分析(解析)的任务。

生成中间码

所有的分析完成后, 编译器会生成中间码, 这是一个阉割的 C++, 主要是由 C 语言构成。一个简单的例子如下:

```
1 class A {  
2     public:  
3         int get_member() { return mem_; }  
4     private:  
5         int mem_;  
6 };
```

对代码进行分析之后, 将生成中间码(这是一个抽象的例子, 意在展示中间代码生成的思想, 编译器可能在实现上有所不同):

```
1 struct A {  
2     int mem_;  
3 };  
4 int A_get_member(A* this) { return this->mem_; }
```

优化

生成中间码有助于编译器在代码中进行优化, 编译器可以优化代码。优化是在多次转换中完成的, 例如下面的代码:

```
1 int a = 41;  
2 int b = a + 1;
```

在编译过程中, 将被优化为:

```
1 int a = 41;  
2 int b = 41 + 1;
```

再次优化为:

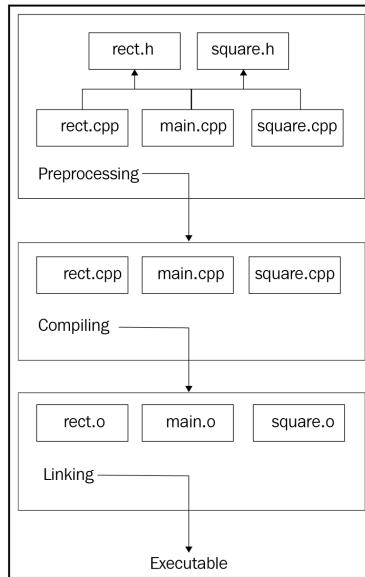
```
1 int a = 41;  
2 int b = 42;
```

一些程序中, 如今的编译器比程序员代码写得更好。

生成机器码

编译器优化是在中间码和生成的机器码中完成的。当我们编译这个项目的时候是什么样子的呢? 之前, 当我们讨论源代码的预处理时, 看了一个包含几个源文件的简单结构, 包括两个头文件,

rect.h 和 square.h，每个头文件都有它的.cpp 文件，以及 main.cpp 包含程序入口点 (main() 函数)。预处理之后，下面的单元是编译器的输入为:main.cpp, rect.cpp 和 square.cpp，如下图所示:



编译器将分别编译。编译单元，也称为源文件，在某种程度上彼此独立。当编译器编译在 Rect 中调用 get_area() 函数的 main.cpp 时，不包含 main.cpp 中的 get_area() 实现。相反，它只能确定该功能是在项目的某个地方实现的。当编译器到达 rect.cpp 时，编译器并不知道 get_area() 函数在哪里使用。

下面是编译器在 main.cpp 通过预处理阶段后得到的结果:

```

1 // contents of the iostream
2 struct Rect {
3     private:
4         double side1_;
5         double side2_;
6     public:
7         Rect(double s1, double s2);
8         const double get_area() const;
9     };
10 struct Square : Rect {
11     Square(double s);
12 };
13 int main() {
14     Rect r(3.1, 4.05);
15     std::cout << r.get_area() << std::endl;
16     return 0;
17 }

```

分析 main.cpp 之后，编译器生成如下中间码 (为了简单地表达编译的思想，省略了很多细节):

```

1 struct Rect {
2     double side1_;
3     double side2_;
4 };

```

```

5 void __Rect_init__(Rect* this, double s1, double s2);
6 double __Rect_get_area__(Rect* this);
7 struct Square {
8     Rect __subobject__;
9 };
10 void __Square_init__(Square* this, double s);
11 int main() {
12     Rect r;
13     __Rect_init__(&r, 3.1, 4.05);
14     printf("%d\n", __Rect_get_area(&r));
15     // we've intentionally replace cout with printf for brevity and
16     // supposing the compiler generates a C intermediate code
17     return 0;
18 }

```

编译器在优化代码时将删除 Square 结构体，及其构造函数（我们将其命名为_Square_init_），因为它从未在源代码中使用过。

此时，编译器只操作 main.cpp，因此看到调用了_Rect_init_ 和_Rect_get_area_ 函数的地方，但实现没有在同一个文件中提供。然而，由于事先提供了声明，所以编译器相信这些函数是在其他编译单元中实现的。基于这种信任和最小信息函数签名（其返回类型、名称和参数）的数量和类型，编译器生成一个对象文件，其中包含 main.cpp 工作代码。而后续的解析工作，是由链接器完成的。

下面的例子中，有生成的目标文件的简化版本，它包含两个部分——代码和信息。代码部分有每个指令的地址（十六进制值）：

```

1 code:
2 0x00 main
3 0x01 Rect r;
4 0x02 __Rect_init__(&r, 3.1, 4.05);
5 0x03 printf("%d\n", __Rect_get_area(&r));
6 information:
7 main: 0x00
8 __Rect_init__: ???
9 printf: ???
10 __Rect_get_area__: ???

```

先来看信息部分。编译器标记了代码部分中使用的所有函数，而这些函数在???? 的同一个编译单元中找不到。这些问号将被链接器在其他单元中找到的实际地址所取代。main.cpp 结束编译后，编译器开始编译 rect.cpp 文件：

```

1 // file: rect.cpp
2 struct Rect {
3     // #include "rect.h" replaced with the contents
4     // of the rect.h file in the preprocessing phase
5     // code omitted for brevity
6 };
7 Rect::Rect(double s1, double s2)
8 : side1_(s1), side2_(s2)
9 {}

```

```
10 const double Rect::get_area() const {
11     return side1_ * side2_;
12 }
```

按照同样的逻辑，编译这个单元会产生以下输出（仍然提供抽象的例子）：

```
1 code:
2     0x00 _Rect_init_
3     0x01 side1_ = s1
4     0x02 side2_ = s2
5     0x03 return
6     0x04 _Rect_get_area_
7     0x05 register = side1_
8     0x06 reg_multiply side2_
9     0x07 return
10 information:
11     _Rect_init_: 0x00
12     _Rect_get_area_: 0x04
```

这个输出中包含了所有函数的地址，因此不需要等待稍后的解析。

平台和对象文件

我们刚才看到的抽象输出，与编译器在编译后产生的实际对象文件结构有些相似。对象文件的结构取决于平台，例如：在 Linux 中，它以 ELF 格式表示（ELF 代表可执行和可链接格式）。平台是程序执行的环境，这里所说的平台是指计算机体系结构（更具体地说，是指令集体系结构）和操作系统的结合。硬件和操作系统是由不同的团队和公司设计和创建的。它们有不同的设计问题解决方案，这导致了平台之间的差异。平台在许多方面存在差异，这些差异也会投射到可执行文件的格式和结构上，例如：Windows 系统中的可执行文件格式是可移植可执行文件（PE），它与 Linux 中的 ELF 格式有不同的结构、编号和序列。

一个目标文件可以分为几个部分。最重要的是代码部分（.text）和数据部分（.data）。.text 部分保存程序指令，.data 部分保存指令使用的数据。数据本身可分割成几个部分，比如初始化的、未初始化的和只读数据。

除了.text 和.data 部分之外，对象文件的一个重要部分是符号表。符号表存储了字符串（符号）到目标文件中的位置的映射。前面的例子中，编译器生成的输出有两个部分，第二部分被标记为 **information:**，它保存了代码中使用的函数名称和相对地址。**information:** 是目标文件实际符号表的抽象版本，符号表包含代码中定义的符号和需要解析的代码中使用的符号。然后，链接器使用这些信息将目标文件链接在一起，形成最终的可执行文件。

连接阶段

编译器为每个编译单元输出一个对象文件。在前面的示例中，我们有三个.cpp 文件，编译器生成了三个目标文件。链接器的任务是将这些目标文件组合成一个单一的目标文件。合并文件会导致相对地址的改变，例如：如果链接器将 rect.o 文件放在 main.o 文件之后 rect.o 的起始地址变成 0x04，而不是以前的值 0x00：

```
1 code:
```

```

2 0x00 main
3 0x01 Rect r;
4 0x02 __Rect_init_(&r, 3.1, 4.05);
5 0x03 printf("%d\n", __Rect_get_area(&r));
6 0x04 __Rect_init_
7 0x05 side1_ = s1
8 0x06 side2_ = s2
9 0x07 return
10 0x08 __Rect_get_area_
11 0x09 register = side1_
12 0x0A reg_multiply side2_
13 0x0B return
14 information (symbol table):
15 main: 0x00
16 __Rect_init_: 0x04
17 printf: ???
18 __Rect_get_area_: 0x08
19 __Rect_init_: 0x04
20 __Rect_get_area_: 0x08

```

链接器相应地更新符号表地址 (例子中的信息部分)。正如前面提到的，每个对象文件都有符号表，它将符号的字符串名称映射到文件中的相对位置 (地址)。链接的下一步是解析目标文件中所有未解析的符号。

现在连接器将 main.o 和 rect.o 合并，因为它们现在位于同一个文件中，就需要知道未解析符号的相对位置。printf 符号将以同样的方式解析，只是这一次是把对象文件与标准库链接起来。在所有的目标文件组合在一起后 (省略了方块的链接)，所有的地址都进行更新，所有的符号都可解析，链接器输出一个最终的目标文件，这个目标文件可以在操作系统中执行。正如本章前面所述，在可执行文件执行前，操作系统会使用加载器将可执行文件的内容加载到内存中。

链接库

库类似于可执行文件，但有一个主要区别：没有 main() 函数，它不能作为常规程序调用。库用于组合多个程序中重用的代码，例如：通过包含 `<iostream>` 头文件将程序与标准库链接起来了。

库可以作为静态库或动态库与可执行文件链接。将它们链接为静态库时，将成为最终可执行文件的一部分。一个动态链接的库也由操作系统加载到内存中，以便为您的程序提供调用其函数的能力。假设我们想求平方根：

```

1 int main() {
2     double result = sqrt(49.0);
3 }

```

C++ 标准库提供了 `sqrt()` 函数，它返回参数的平方根。如果编译前面的示例，它将产生一个错误，提示 `sqrt` 函数没有声明。我们知道，要使用标准库函数，必须包含相应的 `<cmath>` 头文件。但是头文件不包含函数的实现，只是声明了函数 (在 `std` 命名空间中)。我们先包含必要的头文件在源文件中：

```

1 #include <cmath>

```

```
2 int main() {
3     double result = std::sqrt(49.0);
4 }
```

编译器将 sqrt 符号的地址标记为未知的，链接器应该在链接阶段解析它。如果源文件没有链接到标准库实现（包含库函数的目标文件），链接器将无法解析这个问题。

如果链接是静态的，那么链接器生成的最终可执行文件将包含我们的程序和标准库。另一方面，如果链接是动态库，链接器会在运行时标记查找 sqrt 符号。

当我们运行程序时，加载器加载动态链接到程序的库。它也将标准库的内容加载到内存中，然后解析 sqrt() 函数在内存中的实际位置。已经加载到内存中的库，也可以供其他程序使用。

总结

本章中，我们讨论了 C++20 的一些新特性，现在我们准备深入研究这门语言。我们讨论了构建 C++ 应用程序的过程及其编译阶段。这包括分析代码，检测语法错误，生成中间代码以进行优化，最后生成目标文件，该目标文件将与其他生成的目标文件链接在一起，形成最终的可执行文件。

下一章中，我们将学习 C++ 数据类型、数组和指针。还将了解指针是什么，并查看条件语句的底层细节。

问题

1. 编译器和解释器的区别是什么？
2. 列出程序编译阶段。
3. 预处理器做什么？
4. 链接器做了什么？
5. 链接静态库和动态库之间有什么区别？

扩展阅读

更多信息，请参阅高级 C 和 C++ 编译网站：<https://www.amazon.com/Advanced-C-Compiling-Milan-Stevanovic/dp/1430266678/>

LLVM 的信息，<https://www.packtpub.com/application-development/llvm-essentials>

第 2 章：C++ 底层编程

最初，C++ 通常认为是 C 语言的继承者，然而当前的 C++ 已经演变成巨大的，有时可怕，甚至不可驯服的东西。随着语言的更新，则需要更多时间和耐心来了解它。我们将从基本结构开始，比如：数据类型、条件语句和循环语句、指针、结构和函数。并从底层系统程序员的角度来看这些结构，他们对计算机如何执行一条简单的指令感到好奇。深入理解这些基本结构是为更高级的抽象（如面向对象编程），以及为后续学习打下良好基础的必要条件。

本章中，我们将了解以下内容：

- 程序执行的细节及其入口点
- main() 函数的特殊属性
- 函数调用和递归背后的复杂性
- 内存段和寻址基础
- 数据类型以及变量如何驻留在内存中
- 指针和数组
- 条件语句和循环的细节

编译器要求

g++ 编译器需要添加编译选项 `-std=c++2a` 来编译本章的代码。可以从这里获取本章的源码文件：<https://github.com/PacktPublishing/Expert-CPP>

运行程序

第 1 章中，我们了解了编译器在编译源代码后，如何生成一个可执行文件。可执行文件包含机器码，这些机器码可以复制到计算机内存中，由中央处理器（CPU）运行。复制是由操作系统内部的加载器完成。因此，操作系统（OS）将程序的内容复制到内存中，并通过向 CPU 传递条指令开始执行程序。

main()

程序的执行从 main() 函数开始，这是标准中规定的程序的指定入口。一个简单的程序输出 Hello, World! 消息是这样的：

```
1 #include <iostream>
2 int main() {
3     std::cout << "Hello, World!" << std::endl;
4     return 0;
5 }
```

程序中 main() 函数的参数有两个，`argc` 和 `argv`，允许从环境中传递字符串，通常称为命令行参数。

`argc` 和 `argv` 是常规名称，可以用任何你想要的名称替换。`argc` 参数保存了传递给 main() 函数的命令行参数的数量，`argv` 参数保存参数具体信息：

```
1 #include <iostream>
2 int main(int argc, char* argv[]) {
```

```
3 std::cout << "The number of passed arguments is: " << argc << std::endl;
4 std::cout << "Arguments are: " << std::endl;
5 for (int ix = 1; ix < argc; ++ix) {
6     std::cout << argv[ix] << std::endl;
7 }
8 return 0;
9 }
```

例如，我们可以用以下参数运行前面的示例：

```
$ my-program argument1 hello world -some-option
```

输出如下所示：

```
1 The number of passed arguments is: 5
2 Arguments are:
3 argument1
4 hello
5 world
6 --some-option
```

查看参数的数量时，是 5 个。第一个参数是程序名，这就是为什么在例子中跳过它，从第 1 个循环开始。



TIP 很少会看到得到标准化的第三个参数，最常见的名称是 envp。envp 的类型是一个 char 指针数组，它保存着系统的环境变量。

程序可以包含许多函数，但总是从 main() 函数开始执行，至少从程序员的角度来看是这样。我们尝试编译以下代码：

```
1 #include <iostream>
2 void foo() {
3     std::cout << "Risky foo" << std::endl;
4 }
5 // trying to call the foo() outside of the main() function
6 foo();
7 int main() {
8     std::cout << "Calling main" << std::endl;
9     return 0;
10 }
```

g++ 在 foo() 上报出错误： C++ requires a type specifier for all declarations.。调用解析为声明，而不是要执行的指令。对于经验丰富的开发人员来说，尝试在 main() 之前调用函数的方式看起来很愚蠢，所以让尝试另一种方式。如果声明了一个在初始化过程中调用函数呢？下面的例子中，定义了一个 BeforeMain 结构，构造函数打印一条消息，然后在全局作用域中声明 BeforeMain 类型的对象：

```
1 #include <iostream>
2 struct BeforeMain {
```

```
3 BeforeMain() {
4     std::cout << "Constructing BeforeMain" << std::endl;
5 }
6 };
7 BeforeMain b;
8 int main() {
9     std::cout << "Calling main()" << std::endl;
10    return 0;
11 }
```

示例成功编译，程序输出如下信息：

```
1 Constructing BeforeMain
2 Calling main()
```

如果我们向 BeforeMain 添加一个成员函数并尝试调用它，会怎么样？

```
1 struct BeforeMain {
2     // constructor code omitted for brevity
3     void test() {
4         std::cout << "test function" << std::endl;
5     }
6 };
7 BeforeMain b;
8 b.test(); // compiler error
9 int main() {
10    // code omitted for brevity
11 }
```

test() 的调用不会成功。不能在 main() 之前调用函数，但可以声明变量——有默认初始化的对象。实际程序中，会有一些东西需要在 main() 之前进行初始化。事实证明，main() 函数并不是程序的真正起点。程序的实际启动函数准备环境，收集传递给程序的参数，然后调用 main() 函数。因为 C++ 支持全局对象和静态对象，需要在程序开始之前（即 main() 函数被调用之前）进行初始化。在 Linux 世界中，这个函数被称为`_libc_start_main`。编译器使用`_libc_start_main` 调用来生成的代码，该函数在 main() 函数调用之前可能（不）调用其他初始化函数。想象一下前面的代码将被修改成类似下面的样子：

```
1 void __libc_start_main() {
2     BeforeMain b;
3     main();
4 }
5 __libc_start_main(); // call the entry point
```

我们将在接下来的章节中更详细地研究这个入口点。

main() 的特质

我们得出的结论是，main() 实际上不是程序的入口点，尽管标准声明它是起点。编译器特别注意 main()。它的行为就像一个常规的 C++ 函数，但是除了作为第一个调用的函数之外，还有其他特殊的属性。首先，它是唯一可以省略 return 语句的函数：

```
1 int main() {
2     // works fine without a return statement
3 }
```

返回值表示执行状态。通过返回 0，告诉控件 `main()` 已成功结束，因此如结束时没有遇到相应的 `return` 语句，则认为调用成功，效果与 `return 0;` 相同。

`main()` 函数的另一个有趣的特性是，它的返回类型不能自动推断。不允许使用 `auto` 类型说明符。下面是正则函数的工作原理：

```
1 // C++11
2 auto foo() -> int {
3     std::cout << "foo in alternative function syntax" << std::endl;
4     return 0;
5 }
6
7 // C++14
8 auto foo() {
9     std::cout << "In C++14 syntax" << std::endl;
10    return 0;
11 }
```

通过 `auto`，我们告诉编译器自动推断返回类型。在 C++ 11 中，我们还可以将类型名放在箭头后面 (`->`)，尽管第二种语法更短。考虑 `get_ratio()` 函数，它是以整数形式返回：

```
1 auto get_ratio(bool minimum) {
2     if (minimum) {
3         return 12; // deduces return type int
4     }
5     return 18; // fine: get_ratio's return type is already deduced to int
6 }
```



要成功地编译包含 C++11、C++14、C++17 或 C++20 新特性的 C++ 代码，应该使用适当的编译器选项。使用 `g++` 进行编译时，使用 `-std` 标志并指定标准版本。推荐选项为 `-std=c++2a`。

例子编译成功了，但是当我们对 `main()` 函数使用相同的方式时，看看会发生什么：

```
1 auto main() {
2     std::cout << get_ratio(true);
3     return 0;
4 }
```

编译器将产生以下错误：

- `cannot initialize return object of type 'auto' with an rvalue of type 'int'`
- `'main' must return 'int' .`

`main()` 函数发生了一些奇怪的事情。这是因为 `main()` 函数允许省略 `return` 语句，但对于编译器来说，必须存在 `return` 语句才能支持自动返回类型推断。

如果有多个 return 语句，必须推断成相同的类型。假设我们需要更新这个函数的版本，它返回一个整数值（如前面的例子所示）。如果指定了返回类型，则返回一个更精确的浮点值：

```
1 auto get_ratio(bool precise = false) {
2     if (precise) {
3         // returns a float value
4         return 4.114f;
5     }
6     return 4; // returns an int value
7 }
```

上面的代码不能成功编译的原因是，有两个具有不同推断类型的 return 语句。

constexpr

constexpr 声明函数的值可以在编译时求值，也适用于变量。名称本身由 const 和表达式组成。这是一个有用的特性，允许最大限度地优化代码。来看看下面的例子：

```
1 int double_it(int number) {
2     return number * 2;
3 }
4 constexpr int triple_it(int number) {
5     return number * 3;
6 }
7 int main() {
8     int doubled = double_it(42);
9     int tripled = triple_it(42);
10    int test{0};
11    std::cin >> test;
12    int another_triple = triple_it(test);
13 }
```

看看在前面的例子中，编译器是如何修改 main() 函数的。假设编译器不能优化 double_it() 函数（内联函数），main() 函数将采用以下形式：

```
1 int main() {
2     int doubled = double_it(42);
3     int tripled = 126; // 42 * 3
4     int test = 0;
5     std::cin >> test;
6     int another_triple = triple_it(test);
7 }
```

constexpr 不能保证在编译时计算函数值。如果在编译时知道 constexpr 函数的输入，编译器就可以这样做。这就是前面的直接转换为 tripled 变量的计算值 126，并且对 another_triple 变量没有影响的原因。



C++ 20 引入了 consteval 说明符，允许在编译时对函数结果求值。换句话说，consteval 函数

在编译时产生一个常量表达式。说明符使函数成为即时函数，如果函数调用不能导致常量表达式，则会产生错误。注意：main() 函数不能声明为 constexpr。

C++20 还引入了 constinit 说明符。我们使用 constinit 声明一个具有静态或线程存储持续时间的变量。我们将在第 8 章中讨论线程存储持续时间。与 constinit 的区别是，因为 constexpr 要求对象具有静态初始化和常量销毁，所以可以将它用于没有 constexpr 析构函数的对象。而且，constexpr 使对象具有 const 限定条件，而 constinit 则没有。然而，constinit 要求对象具有静态初始化的能力。

递归

main() 的另一个特性是不能递归调用。从操作系统的角度来看，main() 函数是程序的入口点，所以再次调用它将意味着一切重新开，因此是禁止的。例如，print_number() 函数进行递归调用的话，就无法停止了：

```
1 void print_number(int num) {  
2     std::cout << num << std::endl;  
3     print_number(num + 1); // recursive call  
4 }
```

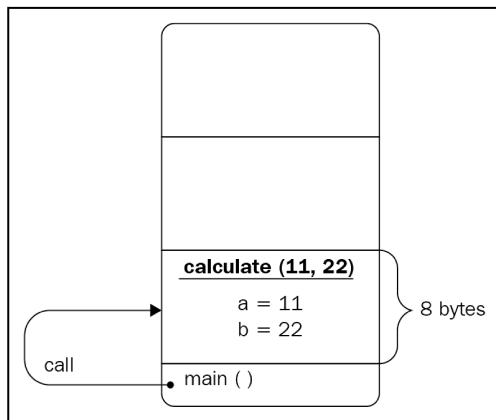
调用 print_number(1) 函数将输出数字 1、2、3 等。这更像是一个无限调用自己的函数，而不是一个正确的递归函数。应该添加更多的属性，使 print_number() 函数成为有用的递归函数。首先，递归函数必须有一个基本情况，即当进一步的函数调用 stop 时，这意味着停止递归。我们可以为 print_number() 函数设置这样的情况，例如：打印 100 以下的所有数字：

```
1 void print_number(int num) {  
2     if (num > 100) return; // base case  
3     std::cout << num << std::endl;  
4     print_number(num + 1); // recursive call  
5 }
```

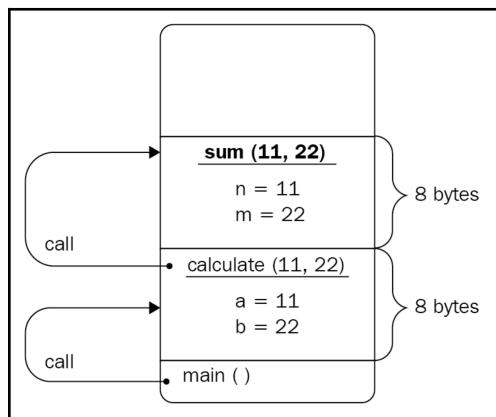
对于递归函数来说还有一个属性：使用基本方法将任务拆分成子任务解决。前面的例子中，我们已经解决了函数的一个任务，也就是打印一个数字。打印了一个数字之后，继续处理下一个任务：打印下一个数字，直到做完。函数调用自身并没有什么神奇之处，可以把它看作是一个函数，调用具有相同实现的另一个函数。真正有趣的是递归函数如何影响整个程序的执行：

```
1 int sum(int n, int m) { return n + m; }  
2 int max(int x, int y) {  
3     int res = x > y ? x : y;  
4     return res;  
5 }  
6 int calculate(int a, int b) {  
7     return sum(a, b) + max(a, b);  
8 }  
9  
10 int main() {  
11     auto result = calculate(11, 22);  
12     std::cout << result; // outputs 55
```

当调用一个函数时，分配内存空间给它的参数和局部变量。程序从 main() 函数开始，在本例中 main() 函数调用 calculate() 函数，并以 11 和 22 为参数。跳转到 calculate() 函数时，main() 函数则处于暂停状态，并等待 calculate() 函数返回后继续执行。calculate() 函数有两个参数 a 和 b，尽管对 sum()、max() 和 calculate() 的形参进行了不同的命名，但可以在所有函数中使用相同的名称。我们假设一个 int 需要 4 个字节的内存，因此要成功执行 calculate() 函数至少需要 8 个字节。分配 8 个字节后，值 11 和 22 需要复制到相应的位置（详见下图）：



calculate() 函数调用 sum() 和 max()，并将参数值传递给它们。相应地，它会等待两个函数按顺序执行，以便形成返回到 main() 的值。sum() 和 max() 函数不会同时调用。首先，调用 sum()，这会导致将变量 a 和 b 的值复制到为 sum() 的参数（命名为 n 和 m）分配的位置，这两个参数总共占用 8 个字节。参考下面的图，以便更好地理解这一点：



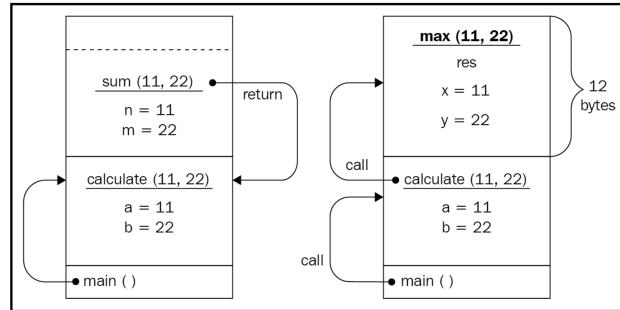
他们的总数经计算后返回。函数完成并返回一个值后，释放内存空间。这意味着变量 n 和 m 不再是可访问的，它们的空间可以重用。



TIP 我们现在不考虑临时变量。稍后会重新讨论这个示例，以展示函数执行的细节，包括临时变量，以及如何尽可能地避免使用临时变量。

sum() 返回一个值后，将调用 max() 函数。遵循相同的逻辑：内存分配给参数 x 和 y，以及 res 变量。我们故意将三元操作符 (?:) 的结果存储在 res 中，以便让 max() 函数为本例分配更多空间。

因此，总共为 max() 函数分配了 12 个字节。此时，main() 函数仍处于等待状态，等待 calculate() 完成，而 calculate() 函数也处于等待状态，等待 max() 函数完成（详情见下图）：



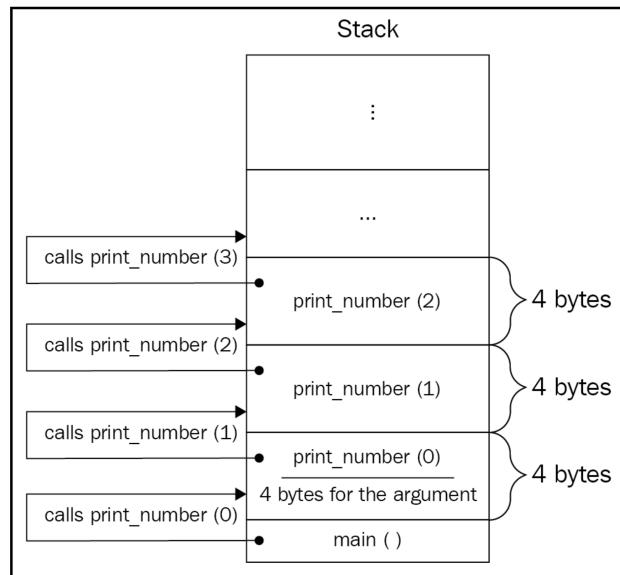
当 max() 完成时，分配给它的内存将被释放，并形成一个返回值。类似地，当 calculate() 返回时，内存被释放，main() 函数的局部变量结果将包含 calculate() 返回的值

然后 main() 函数完成工作，程序退出，操作系统释放分配给该程序的内存，以后可在其他程序重用。为函数分配和释放内存（释放内存）的过程，是使用堆栈的概念来完成的。



TIP 堆栈是一个数据结构，它有自己插入和访问数据的规则。在函数调用的上下文中，堆栈通常用来提供给内存段，该程序根据堆栈数据结构的规则进行自我管理。我们将在本章后面更详细地讨论。

说回到递归，当函数调用自身时，应该为新调用的函数的参数和局部变量（如果有的话）分配内存。该函数再次调用自身，意味着堆栈将继续增长（为新函数提供空间）。我们调用相同的函数没有关系，从堆栈的角度来看，每个新的调用都是对一个完全不同的函数的调用，因此它一边哼着自己最喜欢的小调，一边分配空间。请看下面的图表：



递归函数的第一次调用处于保持状态，并等待同一函数的第二次调用，后者又处于保持状态，并等待第三次调用完成并返回一个值，第三次调用又处于保持状态，依此类推。如果函数中有错误，或者递归基难以达到，堆栈会过度增长，将导致程序崩溃，这称为堆栈溢出。

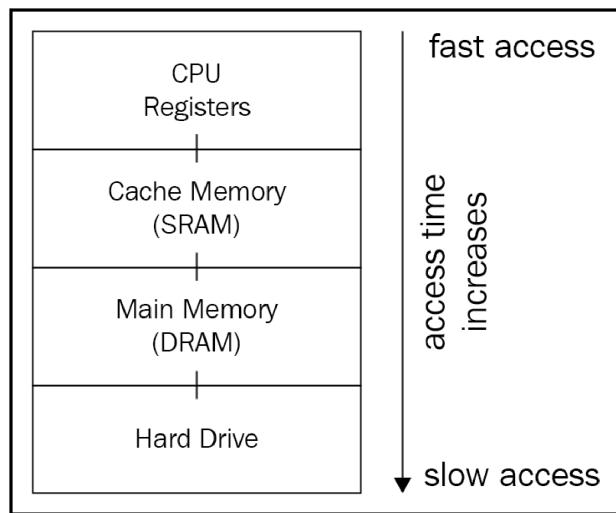


TIP 尽管递归为问题提供了优雅的解决方案，但请避免使用递归，并使用迭代方法（循环）进行替代。在关键任务系统开发指南，如火星探测器导航系统，完全禁止使用递归。

第 1 章中，我们提到了协程。尽管在本书后面会有详细讨论，但需要注意的是主函数不能是协程。

处理数据

当我们提到计算机内存时，我们会默认地想到随机存取存储器 (RAM)，而且 RAM 是 SRAM 或 DRAM 的通用术语，我们默认 RAM 是指 DRAM，除非另有说明。为了清除这些东西，让我们看看下面的图表，它说明了内存的层次结构：



编译一个程序时，编译器将最终的可执行文件存储在硬盘驱动器中。为了运行可执行文件，指令加载到 RAM 中，然后由 CPU 一个个地执行。这使我们得出结论，任何需要执行的指令都应该在 RAM 中。其中，负责运行和监视程序的环境起着主要作用。

我们编写的程序在宿主环境中执行，也就是在操作系统中。OS 不直接将程序的内容（指令和数据，也就是进程）加载到 RAM 中，而是加载到虚拟内存中，这是一种可以方便处理进程和在进程之间共享资源的机制。当我们提到一个进程加载到的内存时，我们指的内存是虚拟内存，它会将进程的内容映射到 RAM 中。



TIP 大多数时候，我们使用术语 RAM、DRAM、虚拟内存和内存，认为虚拟内存是物理内存 (DRAM) 的一种抽象。

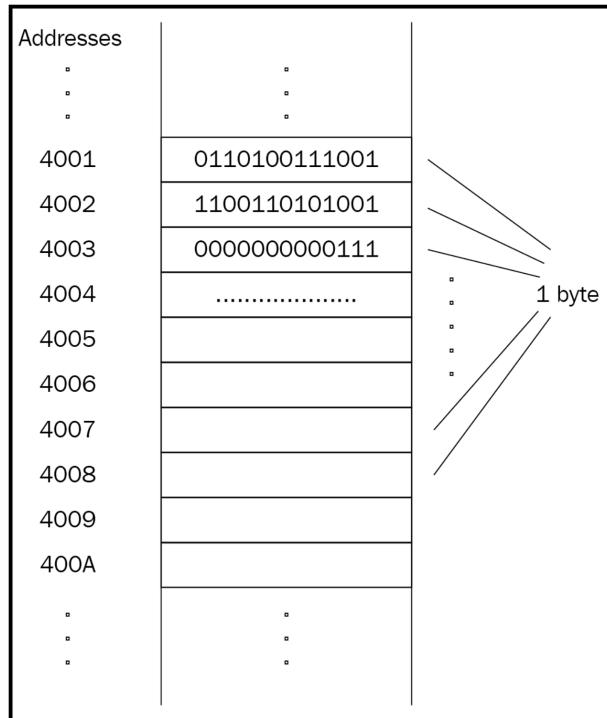
首先介绍内存结构，然后了解内存中的数据类型。

虚拟内存

内存由许多盒子组成，每个盒子都能存储指定数量的数据。考虑到每个单元可以存储代表 8 位的 1 个字节，我们将把这些盒子称为内存单元。每个内存单元都是唯一的，即使它们存储相同的

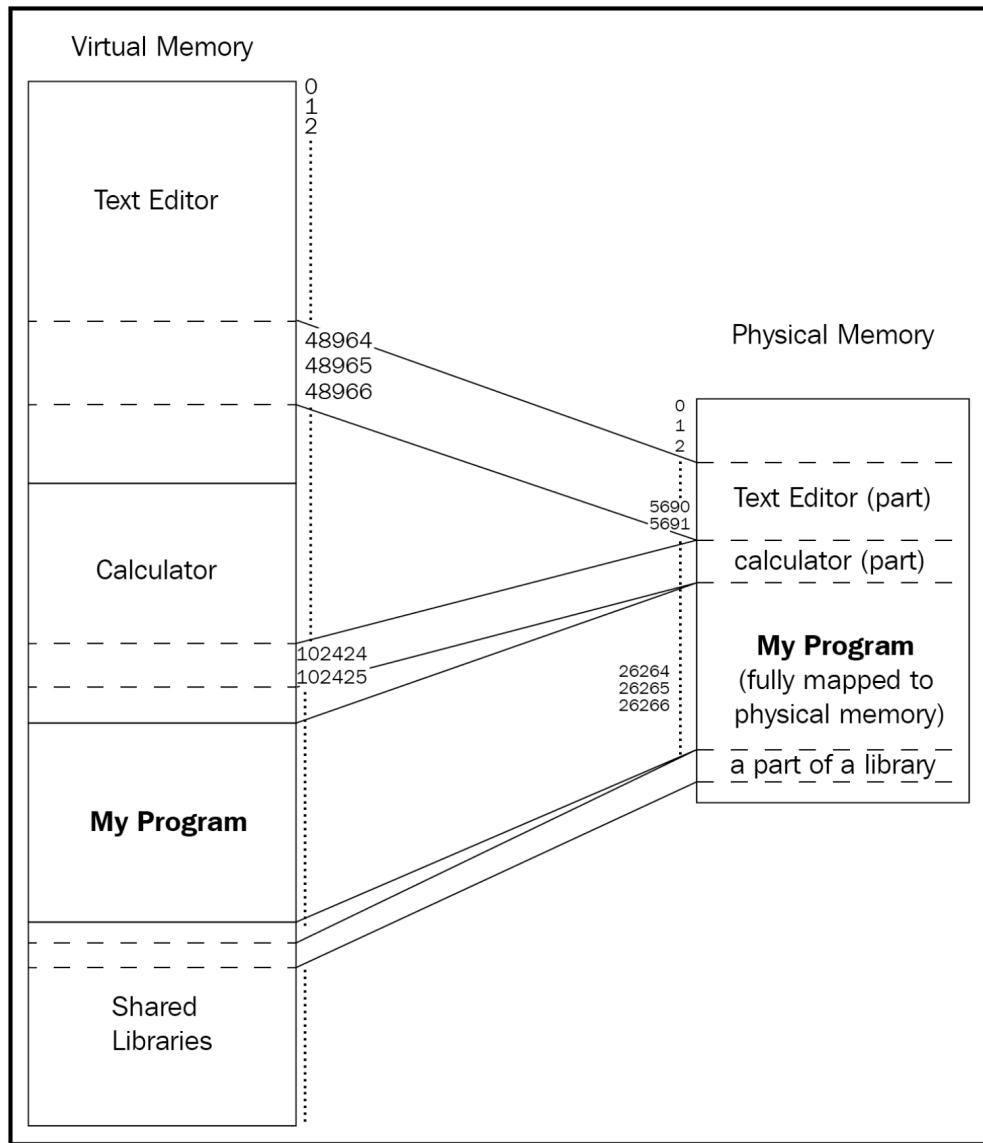
值。这种唯一性是通过对单元寻址来实现的，这样每个单元在内存中都有其唯一的地址。第一个单元格地址为 0，第二个单元格地址为 1，以此类推。

下面的图表是内存的一个摘录，每个单元都有其唯一的地址和存储 1 字节数据的能力：



上图可以用来抽象地表示物理内存和虚拟内存。增加抽象层的目的是便于管理进程，并提供比物理内存更多的功能。例如，操作系统可以执行比物理内存大的程序。以一款电脑游戏为例，它的程序占用了将近 2 GB 的空间，而计算机的物理内存为 512MB。虚拟内存允许操作系统通过从物理内存中卸载旧的部分，并映射新部分来部分地加载程序。

虚拟内存还更好地支持内存中有多个程序，从而支持多个程序的并行（或伪并行）执行。这也提供了共享源码和数据的有效性，比如：动态库。当两个不同的程序需要使用相同的库时，这个库的单一实例可能存在于内存中，并在两个程序不知道对方的情况下使用。看看下面的图表，描述了加载到内存中的三个程序：



上图中有三个正在运行的程序，每个程序都占用虚拟内存中的一些空间。**My Program** 完全包含在物理内存中，而 **Calculator** 和 **Text Editor** 只有部分进行了映射。

地址

如前所述，每个存储单元都有唯一地址，这是每个存储单元唯一性的保证。地址通常用十六进制表示，因为短，转换成二进制比转换成十进制更快。加载到虚拟内存中的程序操作并查看逻辑地址，这些地址也称为虚拟地址，由操作系统提供，在需要时将其转换为物理地址。为了优化转换，CPU 提供了转换备用缓冲区，是内存管理单元 (MMU) 的一部分，转译后备缓冲器缓存虚拟地址到最新的物理地址。因此，有效的地址转换是一个软/硬件任务。我们将在第 5 章中深入研究地址结构和转换的细节。

地址的长度定义了系统可以操作的内存的大小。当遇到与系统位数相关的语句时，它实际上是指地址的长度，即地址长度为 32 位或 64 位。地址越长，内存越大。为了把事情弄清楚，比较一下 8 位长的地址和 32 位长的地址。正如前面约定，每个内存单元能够存储 1 个字节的数据，并且有唯一的地址。如果地址长度为 8 位，则第一个存储单元的地址全部为 0 —— 0000 0000。下一个单

元的地址要大 1，也就是说，它是 0000 0001，以此类推。

可以用 8 位表示的最大值是 1111 1111。那么，有多少内存单元可以用 8 位的地址长度表示呢？这个问题值得更详细地回答。1 位可以表示多少个不同的值？两个！为什么如此？因为 1 位既可以表示 1 也可以表示 0。2 位可以表示多少不同的值？00 是一个值，01 是另一个值，10，最后是 11。所以，总共四个不同的值可以用 2 位表示。让我们做一个表格：

number of bits	number of values	values
1 bit	2	0, 1
2 bit	4	00, 01, 10, 11
3 bit	8	000, 001, 010, 100, ...
4 bit	16	0000, 0001, 0010, ...
.....

我们可以在这里看到一个模式。一个数字的每个位置（每个比特）可以表示两个值，因此可以通过求 2^N 来计算 N 位所代表的不同值的数量。因此，用 8 位表示的不同值的个数是 $2^8 = 256$ 。这意味着一个 8 位系统可以寻址 256 个内存单元。另一方面，32 位系统能够寻址 $2^{32} = 4294967296$ 个内存单元，每个单元存储 1 字节的数据，即存储 $4294967296 * 1\text{byte} = 4\text{GB}$ 的数据。

数据类型

数据类型有什么意义呢？为什么我们不能在 C++ 中使用一些 var 关键字来声明变量，而忘记诸如 short, long, int, char, wchar 等变量呢？C++ 也支持类似的结构，本章之前已经使用过的 auto 关键字，也就是所谓的占位符类型说明符。我们不能（也绝不能）声明一个变量，然后在运行时改变它的类型。下面的代码可能是有效的 JavaScript 代码，但绝对不是有效的 C++ 代码：

```
1 var a = 12;
2 a = "Hello, World!";
3 a = 3.14;
```

假设 C++ 编译器可以编译这段代码。应该为变量 a 分配多少字节的内存？当声明 var a = 12；时，编译器可以将其类型推断为 int，并指定 4 个字节的内存空间，但当变量将其值更改为 Hello, World!，编译器必须重新分配空间，或者创建一个新的名为 a1 的 std::string 类型的隐藏变量。然后，编译器尝试查找代码中，以字符串而不是整数或 double 形式访问变量的每个访问，并用隐藏的 a1 变量。编译器可能会退出并开始思考生命周期的意义。

我们可以在 C++ 中声明类似于前面的代码，如下所示：

```
1 auto a = 12;
2 auto b = "Hello, World!";
3 auto c = 3.14;
```

前两个示例的区别在于，第二个示例声明了三种不同类型的三个不同变量。前面的非 C++ 代码只声明了一个变量，然后将不同类型的值赋给它。在 C++ 中，不能更改变量的类型，但是编译器允许使用自动占位符，并根据赋给该变量的值推断变量的类型。

理解类型是在编译时推导出来的至关重要，而像 JavaScript 这样的语言则允许在运行时推导类型。后者是可能的，因为这类程序运行在虚拟机等环境中，而运行 C++ 程序的环境是操作系统。

C++ 编译器必须生成一个有效的可执行文件，该文件可以复制到内存中，并在没有支持系统的情况下运行。这迫使编译器预先知道变量的实际大小。知道大小对于生成最终的机器码很重要，因为访问变量需要地址和大小，给变量分配内存空间需要它应占用的字节数。

C++ 类型系统将类型分为两大类：

- 基本类型 (int, double, char, void)
- 复合类型 (指针、数组、类)

C++ 甚至支持特殊的类型特征，可以使用 std::is_fundamental 和 std::is_compound，确定类型的类别，例如：

```
1 #include <iostream>
2 #include <type_traits>
3 struct Point {
4     float x;
5     float y;
6 };
7 int main() {
8     std::cout << std::is_fundamental_v<Point> << " "
9     << std::is_fundamental_v<int> << " "
10    << std::is_compound_v<Point> << " "
11    << std::is_compound_v<int> << std::endl;
12 }
```

我们使用了 std::is_fundamental_v 和 std::is_compound_v 定义模板变量，定义如下：

```
1 template <class T>
2 inline constexpr bool is_fundamental_v = is_fundamental<T>::value;
3 template <class T>
4 inline constexpr bool is_compound_v = is_compound<T>::value;
```

程序输出:0 1 1 0。



打印类型类别之前，可以使用 std::boolalpha I/O 操纵符来打印 true 或 false，而不是 1 或 0。

大多数基本类型是算术类型，如 int 或 double，甚至 char 类型也是算术类型。它实际上保存的是一个数字而不是一个字符，例如：

```
1 char ch = 65;
2 std::cout << ch; // prints A
```

一个 char 变量保存 1 个字节的数据，这意味着它可以表示 256 个不同的值（因为 1 个字节是 8 位，8 位可以用 2^8 种方式表示一个数字）。如果使用 1 位作为符号位，例如：允许类型也支持负数，会怎么样？这样我们就剩下 7 位来表示实际值，同理，它允许表示 2^7 个不同的值，128 个（包括 0）不同的正数和相同数量的负数。如果不包含 0，则有符号 char 的范围为 -127 到 +127。这种有符号与无符号的表示形式适用于几乎所有的整型。

例如，遇到一个 int 的大小是 4 个字节，这是 32 位，无符号可表示的数字范围为 0 到 2^{32} 在，有符号可表示的数字范围为 -2^{31} 到 $+2^{31}$ 。

指针

C++ 是一种独特的语言，它提供了对底层细节（如变量地址）的访问。我们可以使用 `&` 操作符获取程序中声明的任何变量的地址，如下所示：

```
1 int answer = 42;  
2 std::cout << &answer;
```

这段代码将输出类似这样的内容：

0x7ffee1bd2adc

注意地址的十六进制表示。虽然这个值是一个整数，但它用来存储指针变量。指针只是一个能够存储地址值并支持 `*` 操作符（解引用）的变量，允许我们找到存储在地址上的实际值。

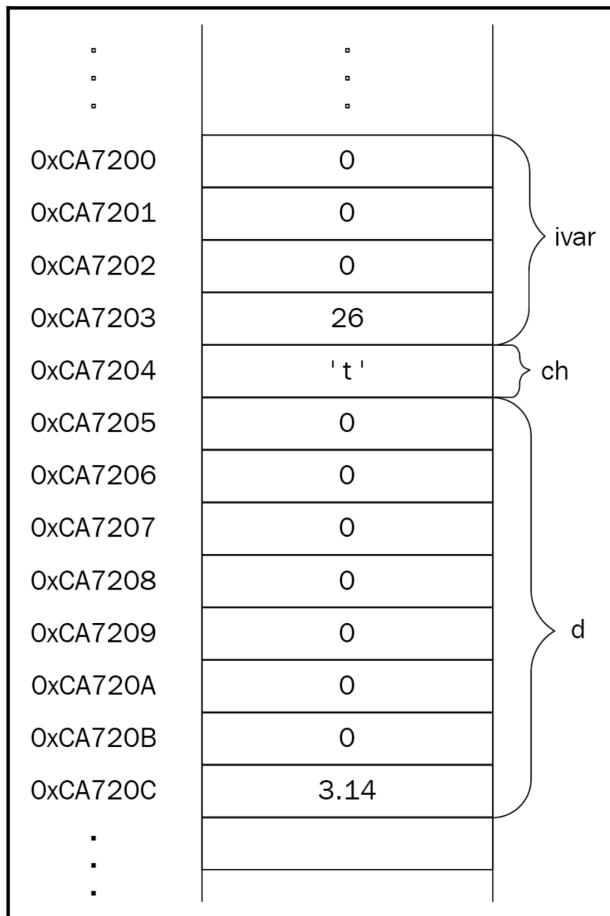
例如，在前面的例子中，为了存储变量 `answer` 的地址，我们可以声明一个指针并给它分配地址：

```
1 int* ptr = &answer;
```

变量 `answer` 声明为 `int`，通常占用 4 个字节的内存空间，每个字节都有自己的唯一地址。我们可以得出 `answer` 变量有四个唯一的地址吗？嗯，有也不是。它确实获得四个不同但相邻的内存字节，但是当对变量使用 `address` 操作符时，它返回第一个字节的地址。让我们看一看声明了几个变量的部分代码，然后说明它们是如何放置在内存中的：

```
1 int ivar = 26;  
2 char ch = 't';  
3 double d = 3.14;
```

数据类型的大小是由实现定义的，尽管 C++ 标准规定了每种类型支持的最小值范围。假设实现为 `int` 提供了 4 个字节，为 `double` 提供了 8 个字节，为 `char` 提供了 1 个字节。前面代码的内存布局应该是这样的：



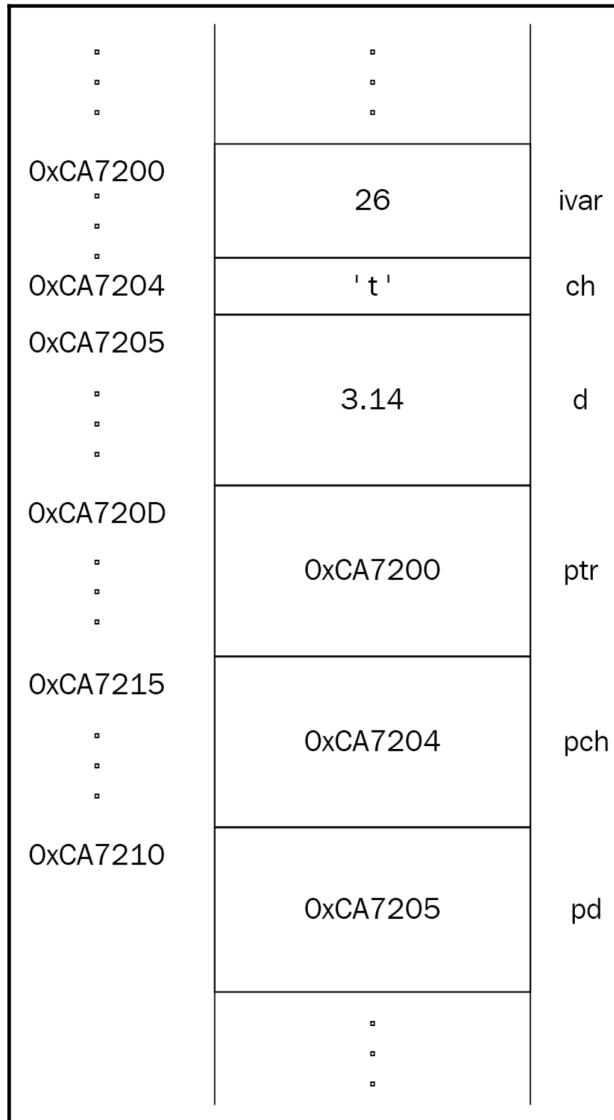
注意内存布局中的 ivar，它存储在四个连续的字节中。

当我们获取一个变量的地址时，无论它存储在单个字节还是多个字节中，我们都获得变量的第一个字节的地址。如果大小不影响寻址操作符背后的逻辑，那么为什么还要声明指针的类型呢？是为了在前面的例子中存储 ivar 的地址，所以将指针声明为 int*:

```

1 int* ptr = &ivar;
2 char* pch = &ch;
3 double* pd = &d;
```

上述代码如下图所示:



事实证明，在使用指针访问变量时，指针的类型是至关重要。C++ 提供了解引用操作符（指针名称前的 * 符号）：

```
1 std :: cout << *ptr; // prints 26
```

它基本上是这样工作的：

1. 读取指针的内容
2. 查找与指针中的地址相等的内存单元的地址
3. 返回存储在该内存单元中的值

问题是，如果指针指向存储在多个内存单元中的数据，该怎么办？这就是指针类型的作用所在。当对指针进行解引用时，它的类型用于确定应该从它所指向的内存单元读取和返回多少字节。

现在知道了指针存储了变量的第一个字节的地址，实际上可以通过向前移动指针来读取变量的任何字节。我们应该记住地址只是一个数字，所以加上或者减去另一个数字会产生另一个地址。如果我一个 char 指针指向一个整型变量会怎样？

```
1 int ivar = 26;
```

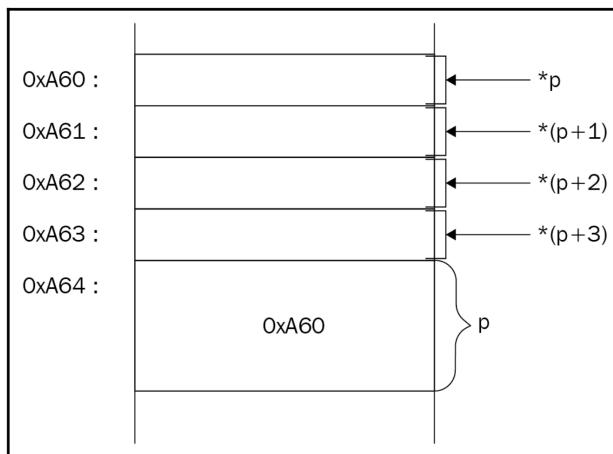
```
2 char* p = (char*)&ivar;
```

当尝试对 p 指针进行解引用时，它将只返回 ivar 的第一个字节。

现在，如果我们想移动到 ivar 的下一个字节，我们给 char 指针加 1:

```
1 // the first byte  
2 *p;  
3 // the second byte  
4 *(p + 1);  
5 // the third byte  
6 *(p + 2);  
7 // dangerous stuff, the previous byte  
8 *(p - 1);
```

观察下面的图表，它清楚地展示了如何访问 ivar:



如果你想读取第一个或最后两个字节，你可以使用 short 类型的指针:

```
1 short* sh = (short*)&ivar;  
2 std::cout << *sh; // print the value in the first two bytes of ivar  
3 std::cout << *(sh + 1); // print the value in the last two bytes of ivar
```



TIP 小心使用指针算术，因为添加或减去一个数字实际上将移动指针，使其与定义的数据类型大小一致。给一个 int 指针加 1 将会给实际的地址加上 `sizeof(int) * 1`。

那么指针的大小呢？如前所述，指针只是一个特殊的变量，它可以存储内存地址，并提供返回位于该地址的数据的解引用操作符。因此，如果指针只是一个变量，也应该存储在内存中。我们可以认为 char 指针的大小小于 int 指针的大小，因为 char 类型的长度小于 int 类型的长度。

关键在于：存储在指针中的数据与指针所指向的数据类型无关。char 指针和 int 指针都存储变量的地址，因此要定义指针的大小，我们应该考虑地址的大小。地址的大小是由系统定义。例如，在 32 位系统中，地址长度为 32 位，而在 64 位系统中，地址长度为 64 位。这使我们得到一个结论：无论指针指向什么类型的数据，指针的大小都是相同的。

```
1 std::cout << sizeof(ptr) << " = " << sizeof(pch) << " = " << sizeof(pd);
```

在 32 位系统中输出 $4 = 4 = 4$, 在 64 位系统中输出 $8 = 8 = 8$ 。

内存段

内存由段组成，程序段在加载期间通过这些内存段分布组成，这些人为划分的内存地址范围使操作系统更容易管理程序。二进制文件也可划分为段，如代码和数据，我们前面提到了代码和数据部分。“节”是链接器所需的二进制文件的分割，用于让链接器正常工作，并将用于加载器的节组合成段。

从运行时的角度讨论二进制文件时，我们指的是“段”。数据段包含程序所需和使用的所有数据，而代码段包含处理相同数据的实际指令。当我们提到数据时，并不是指程序中使用的每一个数据。先来看看个例子：

```
1 #include <iostream>
2 int max(int a, int b) { return a > b ? a : b; }
3 int main() {
4     std::cout << "The maximum of 11 and 22 is: " << max(11, 22);
5 }
```

前面程序的代码段由 `main()` 和 `max()` 函数的指令组成，其中 `main()` 使用 `cout` 对象的操作符 `<<` 打印消息，然后调用 `max()` 函数。哪些数据实际上驻留在数据段中？是否包含 `max()` 函数的 `a` 和 `b` 参数？结果是，数据段中包含的唯一数据是字符串 (`The maximum of 11 and 22 is:`)，以及其他静态、全局或常量。我们没有声明任何全局或静态变量，所以唯一的数据就是上述消息。

有趣的是 11 和 22 的值。这些是文字值，这意味着它们没有地址。因此，它们不在内存中的任何地方。如果它们不在任何地方，那么在程序中位置的唯一就是在代码段中。它们是 `max()` 调用指令的一部分。

`max()` 函数的 `a` 和 `b` 参数呢？这里是虚拟内存中的一个段，它负责存储具有自动存储时间的变量——栈。如前所述，堆栈自动处理局部变量和函数参数的内存空间的分配/释放。当 `max()` 函数被调用时，参数 `a` 和 `b` 将位于堆栈中。通常，如果一个对象具有自动存储时间，那么内存空间将在该封闭块的开始处分配。因此，当函数被调用时，参数推入堆栈：

```
1 int max(int a, int b) {
2     // allocate space for the "a" argument
3     // allocate space for the "b" argument
4     return a > b ? a : b;
5     // deallocate the space for the "b" argument
6     // deallocate the space for the "a" argument
7 }
```

当函数完成时，自动分配的空间将在外围代码块的末尾被释放。



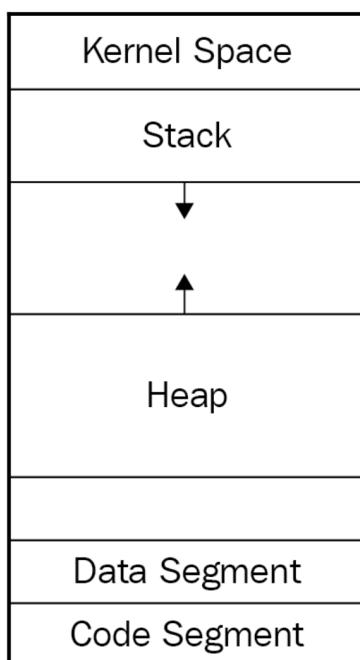
外围的代码块不仅表示函数体，还表示条件语句和循环的块。

参数（或局部变量）从堆栈中弹出，“推入”和“弹出”是栈上下文中使用的术语。通过推入堆栈将数据插入到堆栈中，通过弹出堆栈检索（和删除）数据。你可能遇到过“后进先出”这个术语，这完美地描述了栈的推入和弹出操作。

当程序运行时，操作系统提供了固定的堆栈大小。栈的大小可以增长，如果它增长到没有剩余空间的程度，程序就会因为栈溢出而崩溃。

堆

我们将堆栈描述为具有自动存储时间的变量管理器。“自动”这个词意味着程序员不需要关心实际的内存分配和回收。只有在事先知道数据的大小或数据集合的情况下，才能实现自动存储时间。这样编译器就能知道函数参数和局部变量的数量和类型。但是程序倾向于使用动态数据——大小未知的数据。我们将在第 5 章中详细学习动态内存管理。现在，让我们看看一个简化的内存段图示，看看堆是用来做什么的：



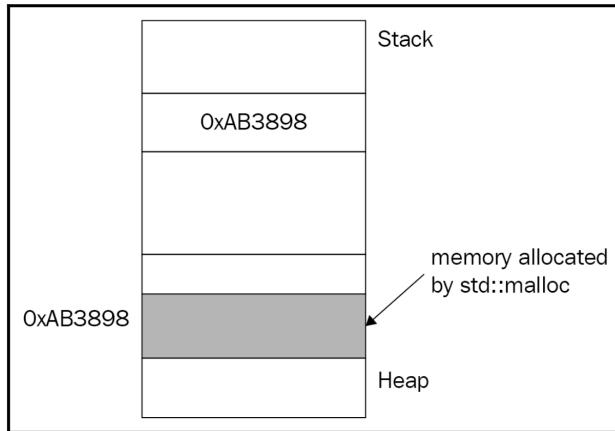
程序使用堆段来请求比以前需要的更多的内存空间，这意味着内存是在程序执行期间动态分配的。无论何时需要，程序都会向操作系统请求新的内存空间。操作系统实际上不知道是整数、用户定义的 Point 类、甚至是用户定义的 Point 数组需要内存。程序通过传递所需要字节的实际大小来请求内存。例如，要为 Point 类型的对象请求一个空间，可以使用 malloc() 函数：

```
1 #include <cstdlib>
2 struct Point {
3     float x;
4     float y;
5 };
6 int main() {
7     std::malloc(sizeof(Point));
8 }
```



malloc() 函数来自 C 语言，为了使用它，需要包含 <cstdlib> 头文件。

`malloc()` 函数分配了一个 `sizeof(Point)` 字节的连续内存空间——假设是 8 个字节。然后，返回该内存的第一个字节的地址，因为这是提供访问空间的唯一方法。问题是，`malloc()` 实际上不知道我们是为 `Point` 对象请求内存空间，还是为 `int` 对象请求内存空间，它只是返回 `void*`。`void*` 存储已分配内存的第一个字节的地址，但它肯定不能通过对指针的解引用来获取实际数据，因为 `void` 没有定义数据的大小。看看下面的图示，它显示了 `malloc` 在堆上分配内存：



要真正使用内存空间，需要强制转换指向所需类型的 `void` 指针：

```
1 void* raw = std::malloc(sizeof(Point));  
2 Point* p = static_cast<Point*>(raw);
```

或者，简单地用强制转换结果声明并初始化指针：

```
1 Point* p = static_cast<Point*>(std::malloc(sizeof(Point)));
```

C++ 通过引入 `new` 操作符解决了这个难题，该操作符自动获取要分配的内存空间大小，并将结果转换为所需类型：

```
1 Point* p = new Point;
```



TIP 动态内存管理是一个手动过程，如果不再需要内存空间，没有类似于堆栈的结构会自动释放内存空间。要正确管理内存资源，当需要释放内存空间时，应该使用 `delete` 操作符。

当访问 `p` 指向的 `Point` 对象的成员时会发生什么？引用 `p` 会返回完整的 `Point` 对象，因此要更改成员 `x` 的值，应该执行以下操作：

```
1 (*p).x = 0.24;
```

或者，更好的是，使用箭头操作符访问：

```
1 p->x = 0.24;
```

我们将在第 3 章中深入研究用户定义类型和结构。

数组

数组是提供连续存储在内存中的数据集合的基本数据结构，许多适配器（如堆栈）都是使用数组实现的。它们的惟一性是数组元素都是同一类型的，这在访问数组元素时起着关键作用。例如，下面的声明创建了一个包含 10 个整数的数组：

```
1 int arr [] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

数组的名称可退化为指向第一个元素的指针。考虑到数组元素具有相同的类型，我们可以通过将指针指向数组的第一个元素来访问数组的任何元素。例如，下面的代码打印数组的第三个元素：

```
1 std :: cout << *(arr + 2);
```

第一个元素也是一样，下面三行代码做了同样的事情：

```
1 std :: cout << *(arr + 0);
2 std :: cout << *arr;
3 std :: cout << arr [0];
```

为了确保 `arr[2]` 和 `*(arr + 2)` 做同样的事情，我们可以这样做：

```
1 std :: cout << *(2 + arr);
```

在 `+` 后面移动 2 不会影响结果，所以下面的代码也是有效的：

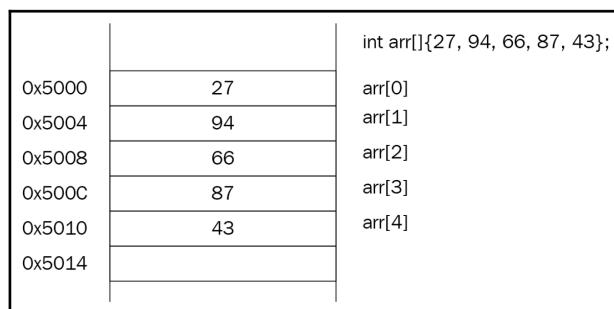
```
1 std :: cout << 2 [arr];
```

然后输出数组的第三个元素。

访问数组元素的时间复杂度是常量，这意味着访问数组的第一个和最后一个元素的时间是相同的。这是因为每次访问数组元素时，都要执行以下操作：

1. 通过添加相应的数值来前进指针
2. 读取放置在指针内存单元的内容

数组的类型指示应该读（或写）多少内存单元。下面的图表说明了访问时的内存排布：



创建动态数组时，这种思想是至关重要的，动态数组是位于堆而不是堆栈中的数组。从堆中分配内存会给出第一个字节的地址，所以访问第一个字节以外的元素的唯一方式是使用指针计算：

```
1 int * arr = new int [10];
2 arr [4] = 2; // the same as *(arr + 4) = 2
```

我们将在第 6 章进一步讨论数组和其他数据结构。

控制流

所有编程语言的最基本概念都是条件语句和循环。

条件

很难想象一个程序不包含条件语句。检查函数的输入参数以确保它们的安全执行几乎是一种习惯。例如，divide() 函数接受两个参数，将其中一个除以另一个，并返回结果。很明显，我们需要确保除数不为零：

```
1 int divide(int a, int b) {
2     if (b == 0) {
3         throw std::invalid_argument("The divisor is zero");
4     }
5     return a / b;
6 }
```

条件语句是编程语言的核心。毕竟，一个程序是行动和决定的集合。例如，下面的代码使用条件语句来查找两个输入参数中的最大值：

```
1 int max(int a, int b) {
2     int max;
3     if (a > b) {
4         // the if block
5         max = a;
6     } else {
7         // the else block
8         max = b;
9     }
10    return max;
11 }
```

为了表达 if-else 语句的用法，上面的例子过分简化了。然而，我们最感兴趣的是这样一个条件语句的实现。当遇到 if 语句时，编译器会生成什么？CPU 依次执行指令，指令是只做一件事的简单命令。高级编程语言（如 C++）中，我们可以在一行中使用复杂的表达式，而汇编指令是简单的命令，在一个循环中只能执行一个简单的操作：移动、添加、减去等。

CPU 从代码内存段获取指令，解码找出应该做什么（移动数据，加数字，减数字），然后执行命令。

为了以最快的速度运行，CPU 将操作数和执行结果存储在称为寄存器的存储单元中。你可以把寄存器看作是 CPU 的临时变量，寄存器是位于 CPU 内的物理内存单元，因此比访问 RAM 快得多。为了从汇编语言程序中访问寄存器，我们使用它们的指定名称，如：rax、rbx、rdx 等。CPU 命令操作寄存器而不是 RAM 单元，这就是为什么 CPU 必须将变量的内容从内存复制到寄存器，执行操作并将结果存储在寄存器中，然后将寄存器的值复制回内存单元。

例如，下面的 C++ 表达式只需要一行代码：

```
1 a = b + 2 * c - 1;
```

汇编表示如下（在分号之后添加注释）：

```

1 mov rax, b; copy the contents of "b"
2             ; located in the memory to the register rax
3 mov rbx, c; the same for the "c" to be able to calculate 2 * c
4 mul rbx, 2; multiply the value of the rbx register with
5             ; immediate value 2 (2 * c)
6 add rax, rbx; add rax (b) with rbx (2*c) and store back in the rax
7 sub rax, 1; subtract 1 from rax
8 mov a, rax; copy the contents of rax to the "a" located in the memory

```

条件语句会跳过部分代码。例如，调用 max(11,22) 意味着 if 块将被省略。为了在汇编语言中表达这一点，使用了跳转的思想。我们比较两个值，并根据结果跳转到代码的指定部分。我们对这部分进行标记，以使找到指令集成为可能。例如，要跳过向寄存器 rbx 添加 42，我们可以使用无条件跳转指令 jmp 跳转到标记为 UNANSWERED 的部分，如下所示：

```

1 mov rax, 2
2 mov rbx, 0
3 jmp UNANSWERED
4 add rbx, 42; will be skipped
5 UNANSWERED:
6 add rax, 1
7 ; ...

```

jmp 指令执行无条件跳转，指定的标号处开始执行第一条指令，而不进行任何条件检查。CPU 也提供了条件跳转，max() 函数的函数体将转换为以下的汇编代码（简化），其中 jg 和 jle 命令分别被解释为大于时跳转，小于或等于时跳转（基于使用 cmp 指令进行比较的结果）：

```

1 mov rax, max; copy the "max" into the rax register
2 mov rbx, a
3 mov rdx, b
4 cmp rbx, rdx; compare the values of rbx and rdx (a and b)
5 jg GREATER; jump if rbx is greater than rdx (a > b)
6 jl LESSOREQUAL; jump if rbx is lesser than
7 GREATER:
8 mov rax, rbx; max = a
9 LESSOREQUAL:
10 mov rax, rdx; max = b

```

前面的代码中，标签 GREATER 和 LESSOREQUAL 表示前面实现的 max() 函数的 if 和 else 子句。

switch 语句

switch 语句等条件语句使用的逻辑如下所示：

```

1 switch (age) {
2     case 18:
3         can_drink = false;
4         can_code = true;
5         break;

```

```
6   case 21:  
7     can_drink = true;  
8     can_code = true;  
9     break;  
10    default:  
11      can_drink = false;  
12  }
```

假设 `rax` 代表年龄，`rbx` 代表 `can_drink`，`rdx` 代表 `can_code`。前面的例子将转化为下面的汇编指令（简化以表达基本思想）：

```
1 cmp rax, 18  
2 je CASE_18  
3 cmp rax, 21  
4 je CASE_21  
5 je CASE_DEFAULT  
CASE_18:  
7 mov rbx, 0; cannot drink  
8 mov rdx, 1; can code  
9 jmp BEYOND_SWITCH; break  
CASE_21:  
11 mov rbx, 1  
12 mov rdx, 1  
13 jmp BEYOND_SWITCH  
CASE_DEFAULT:  
15 mov rbx, 0  
BEYOND_SWITCH:  
16 ; ....
```

每个 `break` 语句都转换为跳转到 `BEYOND_SWITCH` 标签，因此，如果我们忘记 `break` 关键字，例如：在 `age` 为 18 的情况下，执行也会跳转到 `case_21`。

让我们找种方法来避免在源代码中使用条件语句，这样既可以使代码更短，也可能更快——函数指针。

用函数指针替换条件语句

前面，我们研究了内存段，其中最重要的是代码段（也称为文本段）。这个段包含程序映像，它是应该执行的程序的指令。指令通常分组成函数，函数提供了名称，允许我们从其他函数中调用它们。函数存储在可执行文件的代码段中。

函数有它自己的地址。可以声明一个接受函数地址的指针，然后在后面调用该函数：

```
1 int get_answer() { return 42; }  
2 int (*fp)() = &get_answer;  
3 // int (*fp)() = get_answer; same as &get_answer
```

函数指针的调用方式与普通函数相同：

```
1 get_answer(); // returns 42  
2 fp(); // returns 42
```

假设我们正在编写一个程序，从输入中获取两个数字和一个字符，并对这些数字执行算术运算。操作由字符指定，可以是 +、-、* 或 /。我们实现了四个函数：add()、subtract()、multiply() 和 divide()，并根据输入的字符的值调用其中一个函数。

我们不需要在一堆 if 语句或 switch 语句中检查字符的值，而是使用哈希表将操作类型映射到指定的函数：

```
1 #include <unordered_map>
2 int add(int a, int b) { return a + b; }
3 int subtract(int a, int b) { return a - b; }
4 int multiply(int a, int b) { return a * b; }
5 int divide(int a, int b) { return (b == 0) ? 0 : a / b; }
6 int main() {
7     std::unordered_map<char, int (*)(int, int)> operations;
8     operations['+'] = &add;
9     operations['-'] = &subtract;
10    operations['*'] = &multiply;
11    operations['/'] = &divide;
12    // read the input
13    char op;
14    int num1, num2;
15    std::cin >> num1 >> num2 >> op;
16    // perform the operation, as follows
17    operations[op](num1, num2);
18 }
```

std::unordered_map 将 char 映射到定义为 (*) (int, int) 的函数指针。也就是说，可以指向任何接受两个整数并返回一个整数的函数。



哈希表由 std::unordered_map 表示，定义在 <unordered_map> 头文件中。

现在我们不需要这样写：

```
1 if (op == '+') {
2     add(num1, num2);
3 } else if (op == '-') {
4     subtract(num1, num2);
5 } else if (op == '*') {
6     ...
7 }
```

我们只需调用由字符映射的函数：

```
1 operations[op](num1, num2);
```



尽管哈希表的使用很漂亮，看起来也更专业，但应该注意一些意想不到的情况，比如：无效的输入。

函数类型

unordered_map 的第二个参数是 int (*)(int, int)，字面意思是指向函数的指针，该函数接受两个整数并返回一个整数。C++ 支持类模板 std::function 作为通用函数包装器，允许存储可调用对象，包括普通函数、lambda 表达式、函数对象等。存储的对象称为 std::function 的目标，如果没有目标，将在调用时抛出 std::bad_function_call 异常。这有助于使哈希表接受任何可调用对象作为第二个参数，并处理异常情况，如前面提到的无效字符输入。

下面的代码段说明了这一点：

```
1 #include <functional>
2 #include <unordered_map>
3 // add, subtract, multiply and divide declarations omitted for brevity
4 int main() {
5     std::unordered_map<char, std::function<int(int, int)>> operations;
6     operations['+'] = &add;
7     // ...
8 }
```

注意 std::function 的参数，它的形式是 int(int, int)，而不是 int (*)(int, int)。使用 std::function 可以帮助我们处理异常情况。例如，operations['x'](num1, num2)；将导致创建一个映射到字符 x 的空 std::function。

并且调用它会抛出异常，所以可以通过正确处理调用来确保代码的安全性：

```
1 // code omitted for brevity
2 std::cin >> num1 >> num2 >> op;
3 try {
4     operations[op](num1, num2);
5 } catch (std::bad_function_call e) {
6     // handle the exception
7     std::cout << "Invalid operation";
8 }
```

最后，我们可以使用 lambda 表达式——在适当位置构造的未命名函数，并能够捕获作用域中的变量。例如，可以将 lambda 表达式插入到哈希表之前创建一个 lambda 表达式，而不是声明上述函数，然后将它们插入到哈希表中：

```
1 std::unordered_map<char, std::function<int(int, int)>> operations;
2 operations['+'] = [] (int a, int b) { return a + b; }
3 operations['-'] = [] (int a, int b) { return a * b; }
4 // ...
5 std::cin >> num1 >> num2 >> op;
6 try {
7     operations[op](num1, num2);
8 } catch (std::bad_functional_call e) {
9     // ...
10 }
```

lambda 表达式将贯穿全书。

循环

循环可以认为是可重复的 if 语句，同样应该转换成 CPU 比较和跳转指令。例如，我们可以使用 while 循环计算从 0 到 10 的数字总和：

```
1 auto num = 0;
2 auto sum = 0;
3 while (num <= 10) {
4     sum += num;
5     ++num;
6 }
```

这将转译成以下的汇编代码（简化）：

```
1 mov rax, 0; the sum
2 mov rcx, 0; the num
3 LOOP:
4 cmp rbx, 10
5 jg END; jump to the END if num is greater than 10
6 add rax, rcx; add to sum
7 inc rcx; increment num
8 jmp LOOP; repeat
9 END:
10 ...
```

C++17 引入了 init 语句，可以在条件语句和循环中使用。在 while 循环外声明的 num 变量现在可以移动到循环中：

```
1 auto sum = 0;
2 while (auto num = 0; num <= 10) {
3     sum += num;
4     ++num;
5 }
```

同样的规则也适用于 if 语句，例如：

```
1 int get_absolute(int num) {
2     if (int neg = -num; neg < 0) {
3         return -neg;
4     }
5     return num;
6 }
```

C++ 11 引入了基于范围的 for 循环，使得语法更加清晰。例如，让我们使用新的 for 循环调用前面定义的所有算术运算：

```
1 for (auto& op: operations) {
2     std::cout << op.second(num1, num2);
3 }
```

迭代 unordered_map 返回一个 pair，其中包含二个成员，第一个是键，第二个是映射到该键的值。C++ 17 更进一步，允许我们编写如下相同的循环：

```
1 for (auto& [op, func]: operations) {  
2     std :: cout << func(num1, num2);  
3 }
```

了解编译器实际生成的过程，是设计和实现高效软件的关键。我们讨论了条件语句和循环的底层细节，它们几乎是每个程序的基础。

总结

这一章中，我们介绍了程序执行的细节。我们讨论了函数和 main() 函数的一些特性。我们了解了递归是如何工作的，以及 main() 函数不能递归调用。

由于 C++ 支持底层编程概念（如通过地址访问内存字节）的少数高级语言之一，我们研究了数据如何驻留在内存中，以及如何在访问数据时结合指针。对于专业 C++ 程序员来说，了解这些细节是必须的。

最后，我们从汇编语言的角度讨论了条件语句和循环的主题。在这一章中，我们介绍了 C++20 的特性。

下一章中，我们将学习更多关于面向对象编程 (OOP) 的知识，包括语言对象模型的内部细节。我们将深入虚函数的细节，并了解如何使用多态性。

问题

1. main() 函数有多少个形参？
2. constexpr 说明符用于什么？
3. 为什么建议使用迭代而不是递归？
4. 栈和堆的区别是什么？
5. 如果 ptr 声明为 int*，它的大小是多少？
6. 为什么将访问数组元素视为常量时间操作？
7. switch 语句中，如果忘记 break 会发生什么？
8. 如何将算术操作示例中的 multiply() 和 divide() 函数实现为 lambda 表达式？

扩展阅读

你可以参考下面的书获得更多关于本章主题的信息：《C++ High Performance》，Viktor Sehr 和 Bjorn Andrist (<https://www.amazon.com/gp/product/1787120953>).

第 3 章：面向对象编程

设计、实现和维护软件项目的难度与复杂性有关。我们可以使用面向过程的方法（即过程编程范式）编写一个简单的计算器，但用相同的方法实现银行账户管理系统就会非常复杂。

C++ 支持面向对象编程 (OOP)，这种范式建立在将实体分解为对象的基础上。想象一下在现实世界中一个简单的场景，当你拿着遥控器换电视频道时。至少有三种不同的物体参与这个动作：遥控器、电视，还有你自己。为了使用编程语言表达现实世界的对象关系，我们不必使用类、继承、抽象、接口、虚函数等等。上面提到的特性和概念将程序设计为过程式会更容易，因为它们允许我们以优雅的方式表达想法，但并不强制。正如 C++ 的创始人 Bjarne Stroustrup 所说：“并不是每个程序都应该面向对象。”为了理解 OOP 范式的高级概念和特性，我们将尝试从了解其背后的信息。本书中，我们将深入研究面向对象程序的设计。理解对象关系的本质，然后使用它们来设计面向对象的程序。

本章中，我们将了解以下内容：

- 了解面向对象编程
- C++ 对象模型
- 类关系，包括继承
- 多态性
- 有用的设计模式

编译器要求

g++ 编译器需要添加编译选项 `-std=c++2a` 来编译本章的代码。可以从这里获取本章的源码文件：<https://github.com/PacktPublishing/Expert-CPP>

理解对象

大多数时候，我们操作的是一组按特定名称分组的数据，从而形成了一种抽象。如 `is_military`、`speed` 和 `seat` 等变量，如果单独使用，就没有多大意义，但把它们放到“宇宙飞船”中，就会改变存储在变量中的数据的意义，所以我们将许多变量打包成一个对象。为此，我们使用抽象，从观察者的角度收集真实世界对象的个体属性。抽象是程序员的关键工具，因为它允许程序员处理更复杂的情况。C 语言引入结构体作为聚合数据的方式，如下所示：

```
1 struct spaceship {
2     bool is_military;
3     int speed;
4     int seats;
5 };
```

对数据进行分组对于面向对象的编程非常有必要，每组数据称为一个对象。

对象的底层细节

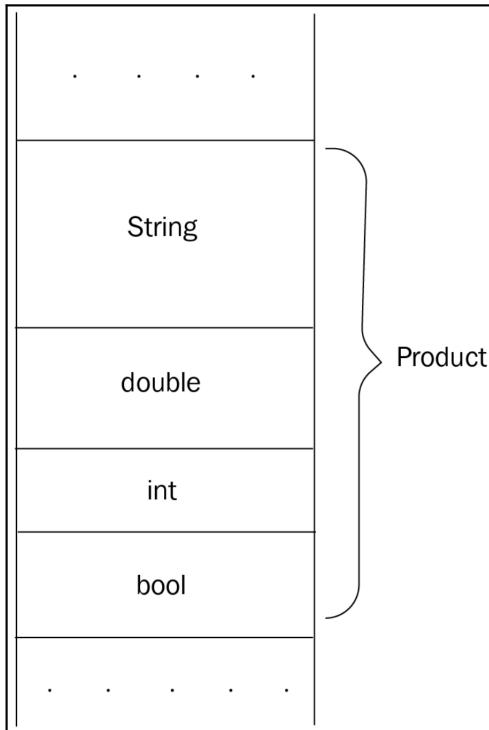
C++ 尽力与 C 语言兼容。C 中的结构体只允许我们聚合数据，而 C++ 则使它们等同于类，允许它们拥有构造函数、虚函数、继承其他结构等。结构体和类之间的唯一区别是默认的可见性修饰符：结构体是 `public`，类是 `private`。类中使用结构通常没有区别，反之亦然。OOP 需要的不仅仅

是数据聚合，为了理解 OOP，让我们看看如果只有提供数据聚合的简单结构而没有其他东西，我们将如何进行 OOP 范式编程。

试试制作一个电子商务市场，如亚马逊或阿里巴巴是产品，我们以以下方式表示：

```
1 struct Product {  
2     std :: string name;  
3     double price;  
4     int rating;  
5     bool available;  
6 };
```

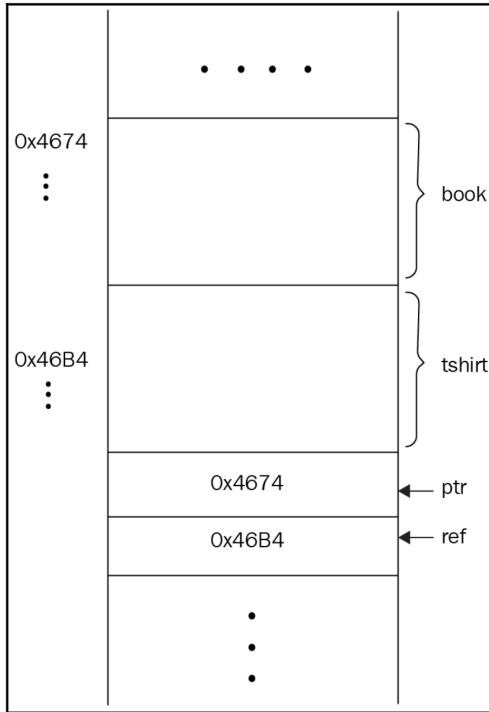
如有必要，我们将向产品中添加更多成员。Product 类型的对象的内存布局如下图所示：



声明 `Product` 对象需要在内存中使用 `sizeof(Product)` 空间，而声明对象的指针或引用需要存储地址所需的空间（通常为 4 或 8 个字节）。参见以下代码块：

```
1 Product book;  
2 Product tshirt;  
3 Product* ptr = &book;  
4 Product& ref = tshirt;
```

上面的代码描述如下：



从 `Product` 对象在内存中占用的空间开始。可以计算 `Product` 对象的大小，并将其成员变量的大小加起来。布尔变量的大小是 1 字节。C++ 标准中没有指定 `double` 类型或 `int` 类型的确切大小。在 64 位机器中，`double` 变量通常需要 8 个字节，而 `int` 变量需要 4 个字节。

`std::string` 的实现没有在标准中指定，所以它的大小取决于标准库的实现。`string` 存储一个指向字符数组的指针，也可以存储已分配字符的数量，可在调用 `size()` 时有效地返回该指针。`string` 的一些实现需要 8、24 或 32 个字节的内存，但在我们的示例中由 24 个字节。`Product` 对象的大小如下：

```
24 (std::string) + 8 (double) + 4 (int) + 1 (bool) = 37 bytes.
```

打印 `Product` 的大小会输出一个不同的值：

```
std::cout << sizeof(Product);
```

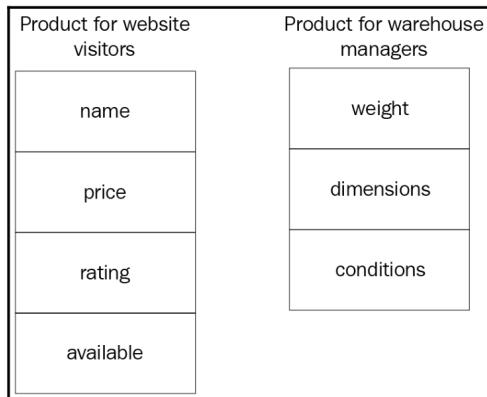
输出的是 40，而不是计算的 37。冗余字节背后的原因是结构体的填充，编译器使用这种技术来优化对对象的各个成员的访问。中央处理器 (CPU) 以固定大小的字读取内存。字的大小由 CPU 定义 (通常是 32 位或 64 位长)。如果从一个字对齐的地址开始，中央处理器则能够访问数据。例如，`Product` 的布尔数据成员需要 1 字节的内存，可以放在 `rating` 之后。编译器会对数据进行对齐，从而实现更快的访问。我们假设单词的大小是 4 个字节，如果变量从一个能被 4 整除的地址开始，CPU 将不需要执行冗余步骤就可以访问这个变量。编译器在前面的结构体中添加额外的字节，使成员地址与边界地址对齐。

对象的高层细节

我们把对象当作代表抽象结果的实体来处理。我们已经提到了观察者，即基于问题域定义对象的开发者，定义它的方式代表了抽象的过程。我们以电子商务市场及其产品为例，两个不同的编程

团队可能对同一产品有不同的看法。实现网站的团队关心对象的属性，这对网站访问者来说是必不可少的买家。我们前面在 Product 结构中显示的属性大多是为网站访问者准备的，如销售价格、产品评级等。实现网站的开发者需要接触问题域，并验证定义 Product 对象所必需的属性。

实现帮助管理仓库中产品的团队关心对象的属性，这些属性在产品放置、质量控制和装运方面必不可少。这个团队实际上不应该关心产品的评级，甚至价格。这个团队主要关心产品的重量、尺寸和条件。下图显示了感兴趣的属性：



开发者在开始项目时应该做的第一件事，是分析问题并收集需求。换句话说，他们应该熟悉问题领域并定义项目需求。分析的过程导致定义对象及其类型，例如：前面讨论的 Product。为了从分析中得到适当的结果，应该在对象中思考，考虑对象的三个主要属性：状态、行为和特性。

状态

每个对象都有一个状态，可能与其他对象的状态不同。我们已经介绍了产品结构，它表示物理（或数字）产品的抽象。Product 对象的所有成员共同表示对象的状态，例如：Product 包含成员 available，这是一个布尔值，如果产品有库存，则为 true。成员变量的值定义了对象的状态。如果给对象成员赋值，它的状态也会改变：

```
1 Product cpp_book; // declaring the object
2 ...
3 // changing the state of the object cpp_book
4 cpp_book.available = true;
5 cpp_book.rating = 5;
```

对象的状态是它所有属性和值的集合。

特性

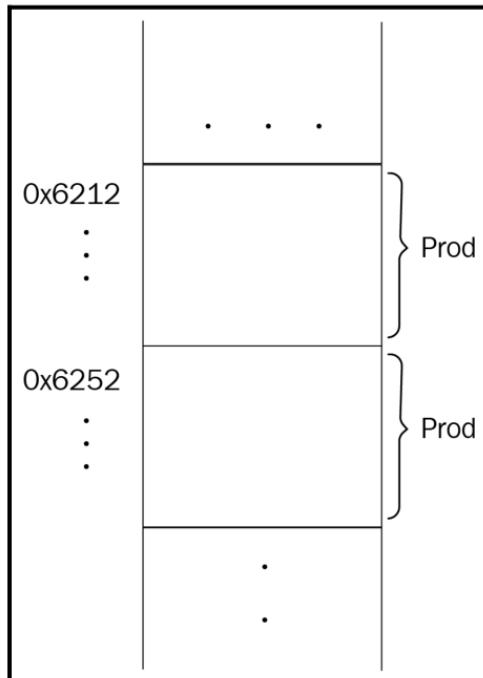
特性是区分一个物体与另一个物体的属性。即使我们试图声明两个物理上不可区分的对象，它们的变量仍然有不同的名称，即不同的标识符：

```
1 Product book1;
2 book1.rating = 4;
3 book1.name = "Book";
4 Product book2;
5 book2.rating = 4;
6 book2.name = "Book";
```

前面示例中的对象具有相同的状态，但不同之处在于我们引用它们的名称，即 book1 和 book2。假设我们能够以某种方式创建具有相同名称的对象，如下面的代码所示：

```
1 Product prod;
2 Product prod; // won't compile, but still "what if?"
```

如果是这样可行，它们会在内存中有不同的地址：



特性是对象的一个基本属性，也是我们不能创建空对象的原因之一，如下所示：

```
1 struct Empty {};
2 int main() {
3     Empty e;
4     std::cout << sizeof(e);
5 }
```

前面的代码不会像预期的那样输出 0。标准中没有指定空对象的大小，编译器开发人员倾向于为这些对象分配 1 个字节，可能也会遇到 4 或 8 个字节。两个或两个以上的 Empty 实例在内存中应该有不同的地址，因此编译器必须确保对象将占用至少 1 个字节的内存。

行为

前面的例子中，我们将 5 和 4 分别赋值给 rating 成员变量，所以很容易因为给对象赋值无效而出错，比如：

```
1 cpp_book.rating = -12;
```

-12 对于产品的评级来说是无效的，否则会让用户感到困惑。我们可以通过提供 setter 函数来控制对象改变的行为：

```
1 void set_rating(Product* p, int r) {
```

```

2   if (r >= 1 && r <= 5) {
3     p->rating = r;
4   }
5   // otherwise ignore
6 }
7 ...
8 set_rating(&cpp_book, -12); // won't change the state

```

对象对的操作和相应源自于其他对象的请求。请求是通过函数调用来执行的，否则称为消息：一个对象将消息传递给另一个对象。前面的示例中，将相应的 set_rating 消息传递给 cpp_book 对象。本例中，我们假设从 main() 调用函数，它实际上根本不表示任何对象。我们可以说它是全局对象，不过 C++ 中没有这样的实体可以操作 main() 函数的对象。

我们从概念上而不是物理上区分对象。面向对象编程的一些概念的物理实现并不是标准化的，因此我们可以将 Product 结构命名为一个类，并声明 cpp_book 是 Product 的一个实例，并且具有一个名为 set_rating() 的成员函数。C++ 实现做了同样的事情：提供了语法上的结构（类、可见性修饰符、继承等等），并将它们转换为简单的结构，使用全局函数，如前面示例中的 set_rating()。现在，让我们深入研究 C++ 对象模型的细节。

模仿类

struct 允许我们对变量进行分组、命名和创建对象。类的思想是在对象中包含相应的操作，对数据和特定数据的操作进行分组。例如，对于 Product 类型的对象，直接调用 setRating() 函数就很自然的，而不是使用全局函数，通过指针接受 Product 对象进行修改。然而，由于在 C 语言中使用结构体，不能使用成员函数。为了模拟使用 C 结构的类，我们必须将使用 Product 对象的函数声明为全局函数，如下面代码所示：

```

1 struct Product {
2   std::string name;
3   double price;
4   int rating;
5   bool available;
6 };
7 void initialize(Product* p) {
8   p->price = 0.0;
9   p->rating = 0;
10  p->available = false;
11 }
12 void set_name(Product* p, const std::string& name) {
13   p->name = name;
14 }
15 std::string get_name(Product* p) {
16   return p->name;
17 }
18 void set_price(Product* p, double price) {
19   if (price < 0 || price > 9999.42) return;
20   p->price = price;
21 }

```

```
22 double get_price(Product* p) {  
23     return p->price;  
24 }  
25 // code omitted for brevity
```

要将结构体作为类使用，我们应该按正确的顺序调用函数。例如，使用正确的初始化默认值对对象进行初始化，所以必须调用 initialize() 函数：

```
1 int main() {  
2     Product cpp_book;  
3     initialize(&cpp_book);  
4     set_name(&cpp_book, "Mastering C++ Programming");  
5     std::cout << "Book title is: " << get_name(&cpp_book);  
6     // ...  
7 }
```

这似乎是可行的，但如果添加了新类型，代码很快就会变得杂乱无章。例如：Warehouse，用于跟踪产品仓库的结构体：

```
1 struct Warehouse {  
2     Product* products;  
3     int capacity;  
4     int size;  
5 };  
6 void initialize_warehouse(Warehouse* w) {  
7     w->capacity = 1000;  
8     w->size = 0;  
9     w->products = new Product[w->capacity];  
10    for (int ix = 0; ix < w->capacity; ++ix) {  
11        initialize(&w->products[ix]); // initialize each Product object  
12    }  
13 }  
14 void set_size(int size) { ... }  
15 // code omitted for brevity
```

第一个明显的问题是函数的命名。我们必须为仓库 initialize_warehouse 的初始化函数命名，以避免与已经声明的产品 initialize() 函数发生冲突。我们可以考虑为产品类型重命名函数，以避免将来可能发生的冲突。接下来是混乱的函数，我们有一堆全局函数，随着我们添加新类型，它们的数量会增加。如果我们添加一些类型层次结构，则更加难以管理。

尽管编译器倾向于将类转换为具有全局函数的结构，但正如前面所示，C++ 和其他高级编程语言解决了这些问题，引入了平滑机制的类，将它们组织到层次结构中。从概念上讲，关键字（类、公共或私有）和机制（继承和多态性）可以方便地让开发人员组织代码。

使用类

处理对象时，类使事情变得容易得多。完成 OOP 中最简单的必要工作：它们将数据与用于操作数据的函数结合起来。我们用类重写 Product 结构体的例子：

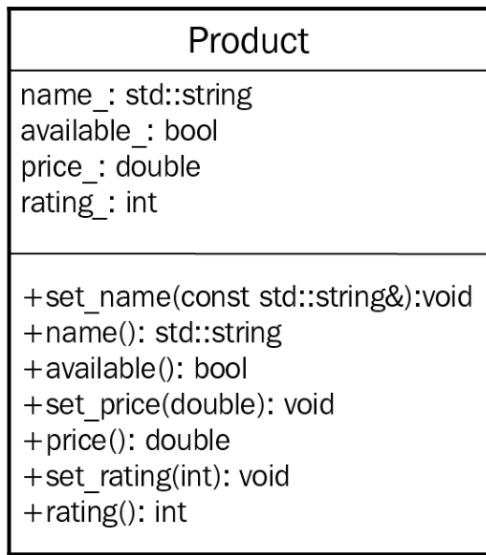
```
1 class Product {
```

```

2 public:
3     Product() = default; // default constructor
4     Product(const Product&); // copy constructor
5     Product(Product&&); // move constructor
6     Product& operator=(const Product&) = default;
7     Product& operator=(Product&&) = default;
8     // destructor is not declared, should be generated by the compiler
9
10    public:
11        void set_name(const std::string&);
12        std::string name() const;
13        void set_availability(bool);
14        bool available() const;
15        // code omitted for brevity
16
17    private:
18        std::string name_;
19        double price_;
20        int rating_;
21        bool available_;
22    };
23
24 std::ostream& operator<<(std::ostream&, const Product&);
25 std::istream& operator>>(std::istream&, Product&);

```

类声明似乎更有组织，尽管它公开的函数比定义类似结构时使用的函数更多。我们应该这样来说明这个类：



前面的图像有些特殊，它有组织的节，函数名之前的符号等。这类图称为统一建模语言 (UML) 类图。UML 是一种标准化阐明类及其关系的过程的方法。第一部分是类的名称 (用粗体表示)，接下来是成员变量部分，然后是成员函数部分。函数名前面的 +(加号) 表示该函数是公共的。成员变量通常是私有的，但如果需要强调这一点，可以使用-(减号)。我们可以通过简单地说明类来省略所有细节，如下面的 UML 图所示：

Product

我们将在本书中使用 UML 图，并根据需要引入新的图类型。在处理初始化、复制、移动、默认和删除函数。在重载操作符之前，先清理一些东西。

从编译器的角度来看类

首先，不管前面的类与前面介绍的结构体相比有多么怪异，编译器都会将其翻译成以下代码（简单起见，稍微修改了下）：

```
1 struct Product {
2     std::string name_;
3     bool available_;
4     double price_;
5     int rating_;
6 };
7
8 // we forced the compiler to generate the default constructor
9 void Product_constructor(Product&);
10 void Product_copy_constructor(Product& this, const Product&);
11 void Product_move_constructor(Product& this, Product&&);
12
13 // default implementation
14 Product& operator=(Product& this, const Product&);
15
16 // default implementation
17 Product& operator=(Product& this, Product&&);
18 void Product_set_name(const std::string&);
19
20 // takes const because the method was declared as const
21 std::string Product_name(const Product& this);
22 void Product_set_availability(Product& this, bool b);
23 bool Product_availability(const Product& this);
24 std::ostream& operator<<(std::ostream&, const Product&);
25 std::istream& operator>>(std::istream&, Product&);
```

基本上，编译器生成的代码与我们前面介绍的代码相同，可以使用一个简单的结构来模拟类行为。尽管编译器在实现 C++ 对象模型的技术和方法上各不相同，但上面的例子是编译器开发人员常用的方法。它平衡了访问对象成员（包括成员函数）的空间和时间效率。

接下来，我们应该考虑编译器如何通过扩充和修改代码来编辑我们的代码。下面的代码声明了全局的 `create_apple()` 函数，该函数创建并返回一个 `Product` 对象，该对象的值特定为 `apple`。还在 `main()` 函数中声明了一个 `book` 对象：

```
1 Product create_apple() {
2     Product apple;
3     apple.set_name("Red apple");
4     apple.set_price("0.2");
5     apple.set_rating(5);
```

```

6     apple.set_available(true);
7     return apple;
8 }
9
10 int main() {
11     Product red_apple = create_apple();
12     Product book;
13     Product* ptr = &book;
14     ptr->set_name("Alice in Wonderland");
15     ptr->set_price(6.80);
16     std::cout << "I'm reading " << book.name()
17     << " and I bought an apple for " << red_apple.price()
18     << std::endl;
19 }
```

我们已经知道，编译器修改类，将其转换为结构体，并将成员函数移动到全局作用域，每个成员函数都将类的引用（或指针）作为其第一个形参。为了在客户端代码中支持这些修改，还应该修改对象的所有访问方式。



TIP 声明或已声明的类对象的代码称为客户端代码。

下面是我们如何假设编译器修改了前面的代码（使用这个词是试图引入编译器抽象，而不是特定的某个编译器）：

```

1 void create_apple(Product& apple) {
2     Product_set_name(apple, "Red apple");
3     Product_set_price(apple, 0.2);
4     Product_set_rating(apple, 5);
5     Product_set_available(apple, true);
6     return;
7 }
8 int main() {
9     Product red_apple;
10    Product_constructor(red_apple);
11    create_apple(red_apple);
12    Product book;
13    Product* ptr;
14    Product_constructor(book);
15    Product_set_name(*ptr, "Alice in Wonderland");
16    Product_set_price(*ptr, 6.80);
17    std::ostream os = operator<<(std::cout, "I'm reading ");
18    os = operator<<(os, Product_name(book));
19    os = operator<<(os, " and I bought an apple for ");
20    os = operator<<(os, Product_price(red_apple));
21    operator<<(os, std::endl);
22    // destructor calls are skipped because the compiler
23    // will remove them as empty functions to optimize the code
24    // Product_destructor(book);
```

```
25 // Product_destructor(red_apple);  
26 }
```

编译器还优化了对 `create_apple()` 函数的调用，以避免临时创建对象。我们将在本章后面讨论编译器生成的临时文件。

初始化和销毁

如前所述，对象的创建需要两个步骤：内存分配和初始化。内存分配是对象声明的结果，C++ 不关心变量的初始化分配内存（不管是自动的还是手动的）。实际的初始化应该由开发者来完成，这就是为什么首先要有一个构造函数。

析构函数遵循同样的逻辑。如果忽略默认构造函数或析构函数的声明，编译器应该隐式生成它们，如果它们为空，编译器也会删除它们（消除对空函数的冗余调用）。如果声明了任何带形参的构造函数，包括复制构造函数，编译器将不会生成默认构造函数。我们可以强制编译器隐式生成默认构造函数：

```
1 class Product {  
2     public:  
3         Product() = default;  
4         // ...  
5     };
```

也可以使用 `delete` 说明符来强制它不生成编译器，如下所示：

```
1 class Product {  
2     public:  
3         Product() = delete;  
4         // ...  
5     };
```

这将禁止默认初始化的对象声明，即 `Product p;` 编译会报错。



析构函数的调用顺序与对象声明的顺序相反，因为自动内存分配是由堆栈管理的，堆栈是遵循后进先出（LIFO）规则的数据结构。

对象在创建时进行初始化，销毁通常发生在对象不再可访问时。当在堆上分配对象时，后者可能比较棘手。看看下面的代码，在不同的作用域和内存段中声明了四个 `Product` 对象：

```
1 static Product global_prod; // #1  
2 Product* foo() {  
3     Product* heap_prod = new Product(); // #4  
4     heap_prod->name = "Sample";  
5     return heap_prod;  
6 }  
7 int main() {  
8     Product stack_prod; // #2  
9     if (true) {  
10         Product tmp; // #3
```

```
11     tmp.rating = 3;
12 }
13 stack_prod.price = 4.2;
14 foo();
15 }
```

global_prod 有一个静态的存储期，并且放置在程序的 global/static 部分，在 main() 调用前初始化。main() 开始时，在堆栈上分配 stack_prod，并在 main() 结束时销毁（函数的右花括号被认为是它的结束）。尽管条件表达式看起来很奇怪，但它是表达作用域的好方法。

tmp 对象也在堆栈上分配，但是它的存储期限制在声明的范围内：当离开 if 块时，将自动销毁。这就是为什么栈上的变量具有自动存储期。最后，调用 foo() 函数时，声明了 stack_prod 指针，该指针指向在堆上分配的 Product 对象的地址。

前面的代码有内存泄漏，当执行到达 foo() 的末尾时，stack_prod 指针（本身有一个自动存储期）将销毁，而在堆上分配的对象不会受到影响。不要混淆指针和它所指向的实际对象：指针只包含对象的值，但不代表对象。



TIP 不要忘记释放在堆上动态分配的内存，可以手动调用 delete 操作符，也可以使用智能指针。智能指针将在第 5 章讨论。

函数结束时，分配给堆栈的参数和局部变量的内存将被释放。当程序结束时，main() 函数结束后，global_prod 将销毁。析构函数将在对象即将销毁时调用。

复制对象

有两种类型的复制：对象的深度复制和浅层复制。语言允许我们使用复制构造函数和赋值操作符，管理对象的复制初始化和赋值。这对于程序员来说是一个必要的特性，这样我们就可以控制复制语义。看看下面的例子：

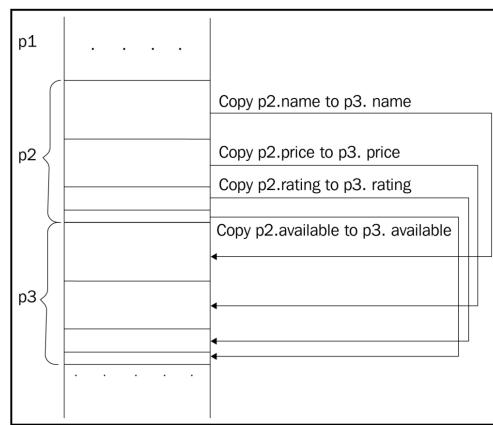
```
1 Product p1;
2 Product p2;
3 p2.set_price(4.2);
4 p1 = p2; // p1 now has the same price
5 Product p3 = p2; // p3 has the same price
```

p1 = p2；是对赋值操作符的调用，而最后一行是对复制构造函数的调用。等号不应该让开发者混淆它是赋值调用还是复制构造函数。每次看到声明后面跟着赋值，就当作复制构造。这适用于新的初始化器语法 (Product p3{p2};)。

编译器将生成以下代码：

```
1 Product p1;
2 Product p2;
3 Product_set_price(p2, 4.2);
4 operator=(p1, p2);
5 Product p3;
6 Product_copy_constructor(p3, p2);
```

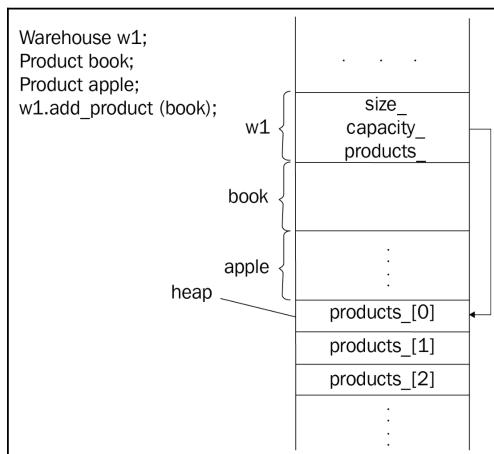
复制构造函数 (和赋值操作符) 的默认实现按成员方式复制对象，如下图所示：



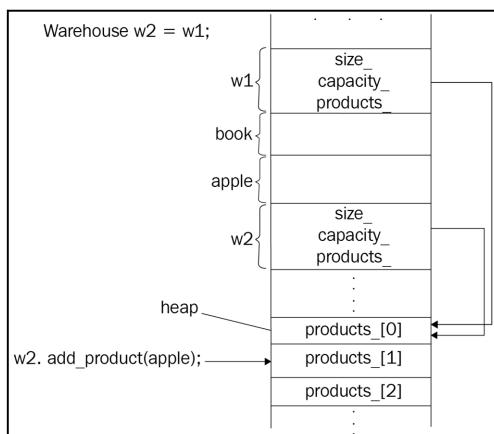
如果成员副本产生无效副本，则需要自定义实现。例如，考虑以下仓库对象的副本：

```
1 class Warehouse {
2 public:
3     Warehouse()
4     : size_{0}, capacity_{1000}, products_{nullptr}
5     {
6         products_ = new Products[capacity_];
7     }
8     ~Warehouse() {
9         delete [] products_;
10    }
11
12 public:
13     void add_product(const Product& p) {
14         if (size_ == capacity_) { /* resize */ }
15         products_[size_++] = p;
16     }
17     // other functions omitted for brevity
18
19 private:
20     int size_;
21     int capacity_;
22     Product* products_;
23 };
24
25 int main() {
26     Warehouse w1;
27     Product book;
28     Product apple;
29     // ... assign values to products (omitted for brevity)
30     w1.add_product(book);
31     Warehouse w2 = w1; // copy
32     w2.add_product(apple);
33     // something somewhere went wrong...
```

前面的代码声明了两个 Warehouse 对象，然后将两个不同的产品添加到仓库中。虽然这个例子有点不自然，但它展示了默认复制实现的危险性。下面的插图向我们展示了代码中出错的地方：



将 w1 分配给 w2 会导致如下结构：



默认实现只是简单地将 w1 的每个成员复制到 w2。复制之后，w1 和 w2 的 products_ 成员都指向堆上的相同位置。当我们向 w2 添加一个新产品时，w1 所指向的数组将受到影响。这是一个逻辑错误，可能会导致程序中未定义的行为。我们需要的是深复制的而不是浅复制，需要实际创建一个包含 w1 数组副本的新产品数组。

复制构造函数和赋值操作符的自定义实现解决了浅复制的问题：

```

1 class Warehouse {
2 public:
3 // ...
4 Warehouse(const Warehouse& rhs) {
5     size_ = rhs.size_;
6     capacity_ = rhs.capacity_;
7     products_ = new Product[capacity_];
8     for (int ix = 0; ix < size_; ++ix) {
9         products_[ix] = rhs.products_[ix];
10    }
    
```

```
11     }
12     // code omitted for brevity
13 }
```

复制构造函数的自定义实现创建了新数组。然后，逐个复制源对象的数组元素，这样就消除了 product_ 指针指向错误内存地址的可能。换句话说，我们通过创建新数组实现了对仓库对象的深复制。

移动对象

代码中到处都是临时对象。大多数情况下，他们都需要让代码按照预期工作。例如，当将两个对象相加时，会创建一个临时对象来保存操作符 + 的返回值：

```
1 Warehouse small;
2 Warehouse mid;
3 // ... some data inserted into the small and mid objects
4 Warehouse large{small + mid}; // operator+(small, mid)
```

让我们来看看 Warehouse 对象的全局操作符 + 的实现：

```
1 // considering declared as friend in the Warehouse class
2 Warehouse operator+(const Warehouse& a, const Warehouse& b) {
3     Warehouse sum; // temporary
4     sum.size_ = a.size_ + b.size_;
5     sum.capacity_ = a.capacity_ + b.capacity_;
6     sum.products_ = new Product[sum.capacity_];
7
8     for (int ix = 0; ix < a.size_; ++ix) { sum.products_[ix] =
9         a.products_[ix]; }
10    for (int ix = 0; ix < b.size_; ++ix) { sum.products_[a.size_ + ix] =
11        b.products_[ix]; }
12
13    return sum;
14 }
```

前面的实现声明了一个临时对象，并在用必要的数据填充它后返回它。前面例子中的调用可以翻译成以下形式：

```
1 Warehouse small;
2 Warehouse mid;
3 // ... some data inserted into the small and mid objects
4 Warehouse tmp{operator+(small, mid)};
5 Warehouse large;
6 Warehouse_copy_constructor(large, tmp);
7 __destroy_temporary(tmp);
```

C++11 中引入的移动语义，允许通过将返回值移动到 Warehouse 对象中，从而跳过临时变量的创建。要做到这一点，我们应该为仓库声明一个移动构造函数，它可以区分临时对象并有效地处理它们：

```
1 class Warehouse {
2 public:
3     Warehouse(); // default constructor
4     Warehouse(const Warehouse&); // copy constructor
5     Warehouse(Warehouse&&); // move constructor
6     // code omitted for brevity
7 }
```

移动构造函数的形参是右值引用 (`&&`).

左值引用

理解为什么引入右值引用之前，让我们先弄清楚左值、引用和左值引用。当一个变量是左值时，它可以寻址，可以指向，并且它有一个作用域存储周期：

```
1 double pi{3.14}; // lvalue
2 int x{42}; // lvalue
3 int y{x}; // lvalue
4 int& ref{x}; // lvalue-reference
```

`ref` 是左值引用，可以视为 `const` 指针的变量同义词：

```
1 int * const ref = &x;
```

除了通过引用修改对象的能力之外，我们还通过引用将重对象传递给函数，以优化和避免冗余的对象副本。例如，仓库的操作符`+`通过引用传入两个对象，这时是复制对象的地址，而不是完整的对象。

左值引用在函数调用方面优化了代码，但是为了优化临时代码，我们应该转向右值引用。

右值引用

不能将左值引用绑定到临时对象。下面的代码就无法进行编译：

```
1 int get_it() {
2     int it{42};
3     return it;
4 }
5 ...
6 int& impossible{get_it()}; // compile error
```

我们需要声明一个右值引用，以便能够绑定到临时对象（包括字面字值）：

```
1 int&& possible{get_it()};
```

右值引用允许我们尽可能地跳过临时对象的生成。例如，将结果作为右值引用的函数，通过消除临时对象的方式运行得更快：

```
1 void do_something(int&& val) {
2     // do something with the val
3 }
4 // the return value of the get_it is moved to do_something rather than
```

```
5 copied  
6 do_something(get_it());
```

想象移动的效果，想象前面的代码将转译成以下代码（只是为了体现移动的完整概念）：

```
1 int val;  
2 void get_it() {  
3     val = 42;  
4 }  
5 void do_something() {  
6     // do something with the val  
7 }  
8 do_something();
```

在引入移动之前，前面的代码看起来是这样的（经过编译器优化）：

```
1 int tmp;  
2 void get_it() {  
3     tmp = 42;  
4 }  
5 void do_something(int val) {  
6     // do something with the val  
7 }  
8 do_something(tmp);
```

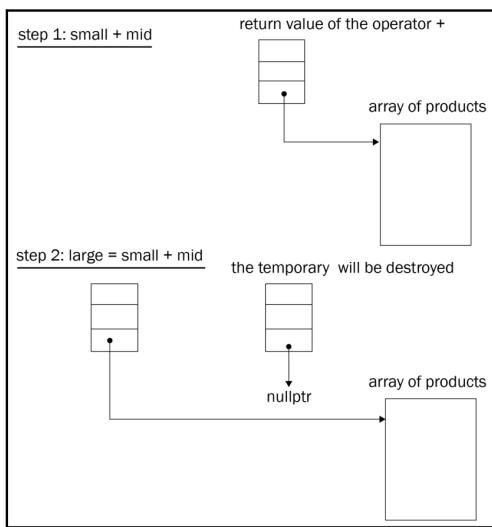
当输入参数表示右值时，移动构造函数和移动操作符`=()`具有复制的效果，而不需要实际执行复制。这就是为什么我们应该在类中实现这些新函数的原因：这样就可以在任何有意义的地方优化代码。移动构造函数可以获取源对象，而不是复制它。代码如下所示：

```
1 class Warehouse {  
2 public:  
3     // constructors omitted for brevity  
4     Warehouse(Warehouse&& src)  
5         : size_{src.size_},  
6         capacity_{src.capacity_},  
7         products_{src.products_}  
8     {  
9         src.size_ = 0;  
10        src.capacity_ = 0;  
11        src.products_ = nullptr;  
12    }  
13};
```

不创建一个`capacity_`大小的新数组，然后复制`products_`数组的每个元素了，这次只是获取了指向该数组的指针。我们知道`src`对象是右值，它很快就会销毁，这意味着析构函数将调用，析构函数将删除已分配的数组。现在，将新创建的`Warehouse`对象指向已分配的数组，这就是为什么不能让析构函数删除源数组的原因。我们将`nullptr`赋值给它，以确保析构函数跳过已分配的对象。因此，下面的代码将因为移动构造函数得到了优化：

```
1 Warehouse large = small + mid;
```

+ 操作符的结果是移动而不是复制。请看下面的图表:



操作符重载的注意事项

C++ 为自定义类型的重载操作符提供了强大的机制，使用 + 操作符计算两个对象的和要比调用成员函数好得多。调用成员函数还需要在调用它之前记住它的名称。它可能是 add, calculatesum, calculate_sum 或其他。操作符重载允许在类设计中采用一致的方法，重载操作符会增加代码中不必要的冗长。下面的代码片段重载了一组比较运算符，以及 Money 类的加减运算符：

```
1 constexpr bool operator<(const Money& a, const Money& b) {
2     return a.value_ < b.value_;
3 }
4 constexpr bool operator==(const Money& a, const Money& b) {
5     return a.value_ == b.value_;
6 }
7 constexpr bool operator<=(const Money& a, const Money& b) {
8     return a.value_ <= b.value_;
9 }
10 constexpr bool operator!=(const Money& a, const Money& b) {
11     return !(a == b);
12 }
13 constexpr bool operator>(const Money& a, const Money& b) {
14     return !(a <= b);
15 }
16 constexpr bool operator>=(const Money& a, const Money& b) {
17     return !(a < b);
18 }
19 constexpr Money operator+(const Money& a, const Money& b) {
20     return Money{a.value_ + b.value_};
21 }
22 constexpr Money operator-(const Money& a, const Money& b) {
23     return Money{a.value_ - b.value_};
24 }
```

前面的大多数函数都直接访问 Money 实例的 value 成员。为了让它起作用，我们应该宣布他们为 Money 的朋友。以下是 Money 的类：

```
1 class Money
2 {
3     public:
4         Money() {}
5         explicit Money(double v) : value_{v} {}
6         // construction/destruction functions omitted for brevity
7     public:
8         friend constexpr bool operator<(const Money&, const Money&);
9         friend constexpr bool operator==(const Money&, const Money&);
10        friend constexpr bool operator<=(const Money&, const Money&);
11        friend constexpr bool operator!=(const Money&, const Money&);
12        friend constexpr bool operator>(const Money&, const Money&);
13        friend constexpr bool operator>=(const Money&, const Money&);
14        friend constexpr bool operator+(const Money&, const Money&);
15        friend constexpr bool operator-(const Money&, const Money&);
16     private:
17         double value_;
18 }
```

这类看起来怪极了。C++20 介绍了 spaceship 操作符，它允许我们跳过比较操作符的定义。`<=>`，也称为三向比较操作符，请求编译器生成关系操作符。对于 Money 类，可以使用默认的`<=>`，如下所示：

```
1 class Money
2 {
3     // code omitted for brevity
4     friend auto operator<=>(const Money&, const Money&) = default;
5 }
```

编译器将生成`==`、`!=`、`<`、`>`、`<=`、`>=`操作符。spaceship 操作符减少了操作符的冗余定义，并为生成的所有操作符提供了一种实现通用方法。在为 spaceship 操作符实现自定义行为时，我们应该注意操作符的返回值类型。它可以是下列之一：

- `std::strong_ordering`
- `std::weak_ordering`
- `std::partial_ordering`
- `std::strong_equality`
- `std::weak_equality`

它们都是在`<compare>`头文件中定义的。编译器根据三向操作符的返回类型生成操作符。

封装和公共接口

封装是面向对象编程中的关键概念，允许对客户端代码隐藏对象的实现细节。以电脑键盘为例，它有字母、数字和符号的键，如果我们按下这些键，就会起作用。键盘的使用简单直观，隐藏了许多只有熟悉电子产品的人才能处理的底层细节。想象一个没有键的键盘，一个有一块没有标记

的针的板，必须猜测要按哪个键才能实现所需的组合键或文本输入。现在，想象一个没有引脚的键盘——必须向相应的套接字发送适当的信号，以获得特定符号的键按事件。用户可能会因为没有标签而感到困惑，他们可能会通过按下或向无效套接字发送信号而错误地使用它。键盘通过封装实现细节解决了这个问题——就像开发者封装对象一样，这样就不会给用户加载冗余的信息，并确保用户不会以错误的方式使用对象。

类中的可见性修饰符通过允许定义任何成员的可访问级别来实现这一目的。`private` 修饰符禁止在客户端代码中使用任何 `private` 成员，可以通过提供相应的成员函数来修改私有成员。`mutator` 函数（许多人都熟悉设置函数）根据类的指定规则测试私有成员之后，修改私有成员的值。下面的代码是一个例子：

```
1 class Warehouse {
2 public:
3     // rather naive implementation
4     void set_size(int sz) {
5         if (sz < 1) throw std::invalid_argument("Invalid size");
6         size_ = sz;
7     }
8     // code omitted for brevity
9
10 private:
11     int size_;
12 };
```

可以通过 `mutator` 函数修改数据成员。实际的数据成员是私有的，这使得客户端代码无法访问它，而类本身提供了公共函数来更新或读取其私有成员的内容。这些函数以及构造函数通常称为类的公共接口。开发者努力使类的公共接口对用户友好。

看一看下面的类，它表示一个二次方程求解器 ($ax^2 + bx + c = 0$)。解决方案之一是找到一个判别使用公式 $D = b^2 - 4ac$ 的值，然后计算基于判别式的值 (D)，下面的类提供了五个函数，可以对于的 a 、 b 和 c 分别进行设置查询判别式，解决方程求解后，返回 x 的值：

```
1 class QuadraticSolver {
2 public:
3     QuadraticSolver() = default;
4     void set_a(double a);
5     void set_b(double b);
6     void set_c(double c);
7     void find_discriminant();
8     double solve(); // solve and return the x
9 private:
10    double a_;
11    double b_;
12    double c_;
13    double discriminant_;
14 };
```

公共接口包括前面提到的四个函数和默认构造函数。为了求解方程 $2x^2 + 5x - 8 = 0$ ，我们应该使用这样的求解器：

```

1 QuadraticSolver solver;
2 solver.set_a(2);
3 solver.set_b(5);
4 solver.set_c(-8);
5 solver.find_discriminant();
6 std::cout << "x is: " << solver.solve() << std::endl;

```

类的公共接口应进行合理设计。前面的例子展示了一个糟糕设计，用户必须知道协议，即调用函数的确切顺序。如果用户没有调用 `find_discriminant()`，则结果将是未定义的或无效的。公共接口强制要求用户学习协议和调用函数以正确的顺序，并设置值 `a`、`b` 和 `c`，然后调用 `find_discriminant()` 函数，最后使用 `solve()` 函数来获得所需的 `x` 的值。好的设计应该提供直观易用的公共界面，可以重写 `QuadraticSolver`，这样它就只有一个函数，接受所有必要的输入值，计算判别式本身，并返回解：

```

1 class QuadtraticSolver {
2 public:
3     QuadraticSolver() = default;
4     double solve(double a, double b, double c);
5 };

```

设计比前面的更直观。下面的代码演示了如何使用二次方程求解器来求解， $2x^2 + 5x - 8 = 0$ ：

```

1 QuadraticSolver solver;
2 std::cout << solver.solve(2, 5, -8) << std::endl;

```

最后要考虑的是二次方程不止一种求解方法，这个是通过求出判别式来完成的，应该将来我们可以向该类添加更多的实现方法。更改函数的名称可以增加公共接口的可读性，并确保类在未来更新时的安全性。我们还应该注意到，前面例子中的 `solve()` 函数接受 `a`、`b` 和 `c` 作为参数，不过不需要将它们存储在类中，因为解决方案是在函数中直接计算的。

显然，为了访问 `solve()` 函数而声明二次方程求解器的对象似乎是一个多余的步骤。该类的最终设计如下所示：

```

1 class QuadraticSolver {
2 public:
3     QuadraticSolver() = delete;
4
5     static double solve_by_discriminant(double a, double b, double c);
6     // other solution methods' implementations can be prefixed by "solve_by_"
7 };

```

我们将 `solve()` 函数重命名为 `solve_by_discriminant()`，这个函数展示了解决方案的底层方法。我们还可以将函数设置为 `static`，这样用户就可以使用它，而无需声明类的实例。然而，我们还需要将默认构造函数标记为 `deleted`，从而再次强制用户可以不声明对象：

```

1 std::cout << QuadraticSolver::solve_by_discriminant(2, 5, -8) << std::endl;

```

现在，客户端代码使用类所的开销就更少了。

C++ 中的结构体

结构体与 C++ 中的类几乎相同。它们具有类的所有特性，类可以继承结构体，反之亦然。类和结构之间的唯一区别是默认可见性。对于结构体，默认的可见性修饰符是 public，例如：当从另一个类继承一个类而不使用修饰符时，它将 private 继承。下面的类 private 继承自 Base：

```
1 class Base
2 {
3     public:
4     void foo() {}
5 };
6 class Derived : Base
7 {
8     // can access foo() while clients of Derived can't
9 };
```

同样的逻辑，下面的结构体是对 Base 类进行 public 继承：

```
1 struct Base
2 {
3     // no need to specify the public section
4     void foo() {}
5 };
6
7 struct Derived: Base
8 {
9     //both Derived and clients of Derived can access foo()
10};
```

这与从结构继承的类有关。例如，如果没有直接指定，派生类将 private 继承 Base：

```
1 struct Base
2 {
3     void foo() {}
4 };
5 // Derived inherits Base privately
6 class Derived: Base
7 {
8     // clients of Derived can't access foo()
9 };
```

C++ 中，结构体和类可以互换，但大多数程序员更喜欢将结构体用于简单的类型。C++ 标准为简单类型提供了更好的定义，并将它们称为聚合。如果一个类符合以下规则，那么它就是一个聚合：

- 没有私有或受保护的非静态数据成员
- 没有用户声明或继承的构造函数
- 没有虚拟、私有或受保护的基类
- 没有虚拟成员函数

当读完这一章后，大部分的规则会更加清晰。下面的结构体是聚合的例子：

```
1 struct Person
2 {
3     std :: string name;
4     int age;
5     std :: string profession;
6 };
```

在深入研究继承和虚函数之前，让我们看看聚合在初始化时带来的好处。我们可以用以下方式初始化 Person 对象：

```
1 Person john{ "John Smith" , 22 , "programmer" };
```

C++20 提供了更奇特的方法来初始化聚合：

```
1 Person mary{ .name = "Mary Moss" , .age{22} , .profession{ "writer" } };
```

注意，我们是如何将成员的初始化混合起来的。

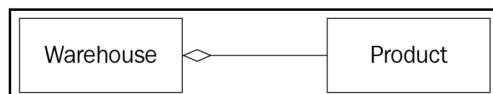
结构化绑定允许我们声明绑定到聚合成员的变量，如下面的代码所示：

```
1 const auto [p_name, p_age, p_profession] = mary;
2 std :: cout << "Profession is: " << p_profession << std :: endl;
```

结构化绑定也适用于数组。

类间关系

对象间通信是面向对象系统的核心。关系是对象之间的逻辑链接。区分或在对象的类之间建立适当关系的方法定义了整个系统设计的性能和质量。考虑 Product 和 Warehouse 类，处于一种称为聚合的关系中，因为 Warehouse 包含 Product，Warehouse 聚合 Product：



就纯 OOP 而言，有几种关系，比如关联、聚合、组合、实例化、泛化等。

聚合和组合

在 Warehouse 类的示例中遇到了聚合。Warehouse 类存储一个 Product 数组。更一般的术语中，可以称为关联，但为了强调包含，我们使用术语聚合或组合。聚合的情况下，可以在没聚合的情况下实例化包含一个或多个其他类实例的类。这意味着可以创建和使用一个 Warehouse 对象，而不必创建包含在 Warehouse 中的 Product 对象。

聚合的另一个例子是 Car 和 Person。Car 可以包含 Person 对象（作为司机或乘客），因为它们彼此关联，但该包含不是强相关。我们可以创建一个没有 Driver 的 Car 对象，如下所示：

```
1 class Person; // forward declaration
2 class Engine { /* code omitted for brevity */ };
3 class Car {
4     public:
5     Car();
```

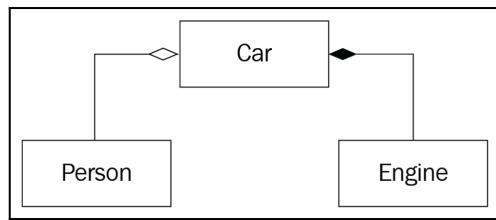
```

6 // ...
7 private:
8 Person* driver_; // aggregation
9 std::vector<Person*> passengers_; // aggregation
10 Engine engine_; // composition
11 // ...
12 };

```

强包容是由组成来表达的。对于 Car 示例，需要 Engine 类的对象来创建完整的 Car 对象。在这个物理表示中，Engine 在创建 Car 时自动创建。

下面是聚合和组合的 UML 表示：



设计类时，我们必须确定它们之间的关系。定义两个类之间的组合的最佳方法是使用 has-a 关系测试。汽车有引擎，任何时候，你不能决定是否应该用组合的形式来表达这种关系时，就问 has-a 问题。聚集和组合有些相似，只是描述了这种联系的强度。对于聚合，恰当的问题是可以是，例如：一辆汽车可以有一个司机 (Person 类型)，是弱关联。

继承

继承是一个编程概念，它允许我们重用类。编程语言提供了不同的继承实现，但一般规则总是成立的：类关系应该回答一个问题。例如，Car 就是 Vehicle，这让我们可以从 Vehicle 继承 Car：

```

1 class Vehicle {
2     public:
3     void move();
4 };
5 class Car : public Vehicle {
6     public:
7     Car();
8     // ...
9 };

```

Car 现在有了从 Vehicle 派生的 move() 成员函数。继承本身表示泛化/特化关系，其中父类 (Vehicle) 是泛化，子类 (Car) 是特化。



父类可以称为基类或超类，而子类可以分别被为派生类或子类 (subclass)。

只有在绝对必要时才应该考虑使用继承。正如我们前面提到的，类应该满足 is-a 关系。考虑 Square 和 Rectangle 类，下面的代码以最简单的形式声明了 Rectangle 类：

```

1 class Rectangle {
2 public:
3     // argument checks omitted for brevity
4     void set_width(int w) { width_ = w; }
5     void set_height(int h) { height_ = h; }
6     int area() const { return width_ * height_; }
7 private:
8     int width_;
9     int height_;
10};

```

正方形是一个矩形，所以我们可以很容易地从矩形继承它：

```

1 class Square : public Rectangle {
2 public:
3     void set_side(int side) {
4         set_width(side);
5         set_height(side);
6     }
7     int area() {
8         area_ = Rectangle::area();
9         return area_;
10    }
11 private:
12     int area_;
13};

```

方形扩展了矩形，添加了一个新的数据成员 `area_`，并用自己的实现覆盖了 `area()` 成员函数。实践中，`area_` 和计算值的方法是冗余的，我们这样做是为了演示一个糟糕的类设计，并在一定程度上让 `Square` 扩展它的父类。很快，我们将得出这样的结论：在本例中，继承是一个糟糕的选择。`Square` 是一个矩形，所以它应该在任何使用矩形的地方作为矩形使用，如下所示：

```

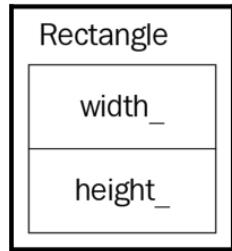
1 void make_big_rectangle(Rectangle& ref) {
2     ref->set_width(870);
3     ref->set_height(940);
4 }
5
6 int main() {
7     Rectangle rect;
8     make_big_rectangle(rect);
9     Square sq;
10    // Square is a Rectangle
11    make_big_rectangle(sq);
12}

```

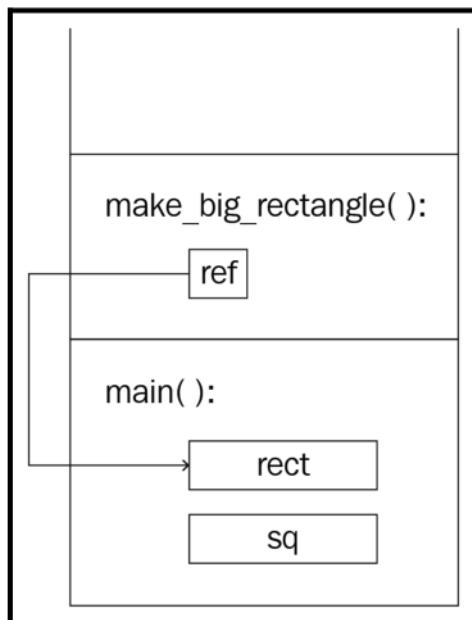
`make_big_rectangle()` 函数接受对矩形的引用，方形继承了它，所以完全可以将一个方形对象发传递给 `make_big_rectangle()` 函数。正方形是一个长方形，成功地用类型的子类型替换的例子称为 Liskov 替换原则。让我们来看看为什么这个替换在实践中是有效的，然后再决定我们是否有设计错误，从矩形继承了方形。

从编译器的角度来看继承

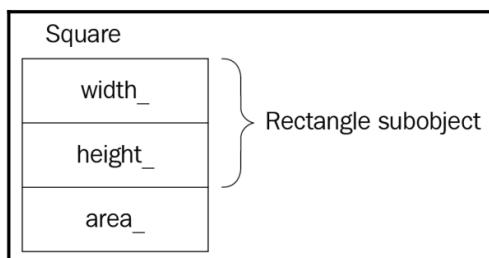
我们可以用以下方式来描绘前面声明的 Rectangle 类:



当在 main() 函数中声明 rect 时, 函数的局部对象所需的空间在堆栈中分配。调用 make_big_rectangle() 函数时遵循相同的逻辑。这里没有局部参数, 相反, 它有一个 `Rectangle&` type 参数, 其行为方式与指针类似: 获取存储内存地址所需的内存空间 (在 32 位和 64 位系统中分别为 4 或 8 个字节)。rect 通过引用传递给 make_big_rectangle(), 这意味着 ref 参数引用 main() 中的局部对象:



下面是 Square 类的一个示例:

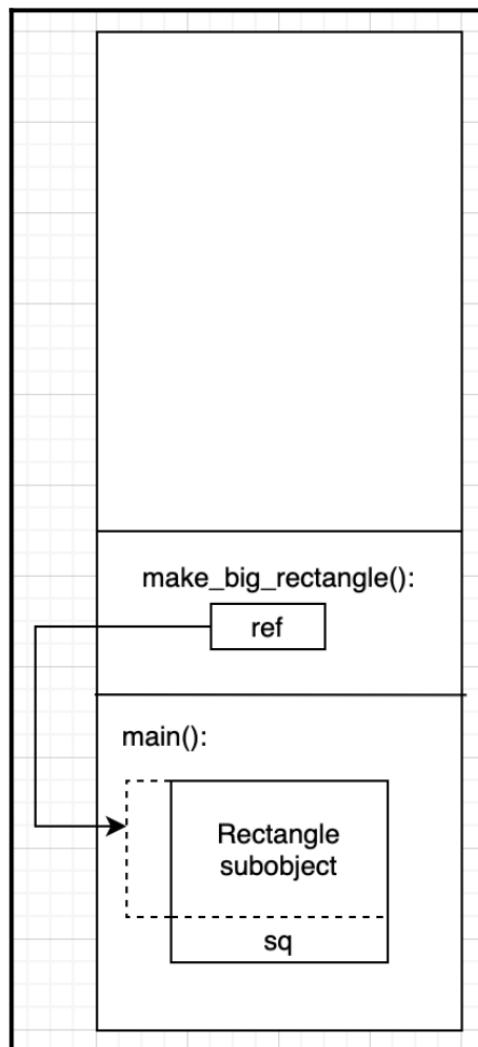


如上图所示, Square 对象包含 Rectangle 的子对象, 部分地代表了一个矩形。在这个特定的示例中, Square 类没有使用新的数据成员扩展 Rectangle。

Square 对象传递给 make_big_rectangle()，后者接受 Rectangle& type 的参数。我们知道访问基础对象时需要指针（引用）的类型，类型定义了应该从指针所指向的起始地址读取多少字节。这种情况下，ref 存储 main() 中声明的本地 rect 对象的起始地址的副本。当 make_big_rectangle() 通过 ref 访问成员函数时，实际上调用以 Rectangle 引用作为第一个形参的全局函数。这个函数转译成如下代码（简单起见，修改了一下）：

```
1 void make_big_rectangle(Rectangle * const ref) {  
2     Rectangle_set_width(*ref, 870);  
3     Rectangle_set_height(*ref, 940);  
4 }
```

解引用 ref 意味着从 ref 所指向的内存位置开始读取 sizeof(Rectangle) 字节。当传递一个 Square 对象给 make_big_rectangle() 时，将 sq(Square 对象) 的起始地址赋值给 ref，因为 Square 对象实际上包含一个 Rectangle 子对象。当 make_big_rectangle() 函数解引用 ref 时，只能访问对象的 sizeof(Rectangle) 字节，而看不到实际 Square 对象的额外字节。下面的图表说明了子对象引用指向的部分：



从 Rectangle 中继承 Square 等同于声明两个结构体，其中一个（子结构体）包含另一个（父结构体）：

```

1 struct Rectangle {
2     int width_;
3     int height_;
4 };
5 void Rectangle_set_width(Rectangle& this, int w) {
6     this.width_ = w;
7 }
8 void Rectangle_set_height(Rectangle& this, int h) {
9     this.height_ = h;
10 }
11 int Rectangle_area(const Rectangle& this) {
12     return this.width_ * this.height_;
13 }
14 struct Square {
15     Rectangle _parent_subobject_;
16     int area_;
17 };
18 void Square_set_side(Square& this, int side) {
19     // Rectangle_set_width(static_cast<Rectangle*>(this), side);
20     Rectangle_set_width(this._parent_subobject_, side);
21     // Rectangle_set_height(static_cast<Rectangle*>(this), side);
22     Rectangle_set_height(this._parent_subobject_, side);
23 }
24 int Square_area(Square& this) {
25     // this.area_ = Rectangle_area(static_cast<Rectangle*>(this));
26     this.area_ = Rectangle_area(this._parent_subobject_);
27     return this.area_;
28 }
```

前面的代码演示了编译器支持继承的方式。看看 `Square_set_side` 和 `Square_area` 函数的注释代码行，表达了编译器如何处理 OOP 代码的全部思想。

组合和继承

C++ 语言为提供了方便且对 OOP 友好的语法，这样就可以表达继承关系，但是编译器处理它的方式类似于组合而不是继承。实际上，在适用的情况下，使用组合而不是继承会更好。`Square` 类及其与矩形的关系是一个糟糕的设计。原因是子类型替换原则，我们以错误的方式使用 `Square`：将其传递给一个函数，该函数将其修改为 `Rectangle` 而不是 `Square`。这告诉我们，`is-a` 关系是不正确的，因为 `Square` 毕竟是 `Rectangle`。它是对 `Rectangle` 的泛化，而不是 `Rectangle`。

使用 `Square` 的用户最好不了解它可以作为 `Rectangle` 使用，否则将向 `Square` 实例发送无效或不受支持的消息。无效消息的例子是调用 `set_width` 或 `set_height` 函数。`Square` 实际上不应该支持两个不同的成员函数来修改它的边长，但它不能隐藏这一点，因为它继承自 `Rectangle`：

```

1 class Square : public Rectangle {
2     // code omitted for brevity
3 };
```

如果我们把修饰语从 public 改为 private 会怎样? C++ 既支持公有继承类型，也支持私有继承类型，还支持受保护的继承。当私有继承一个类时，子类打算使用父类并访问其公共接口。但是，客户端代码并不知道它在处理一个派生类。此外，从父类继承的公共接口对于子类的用户来说变成了私有的。看起来像是将继承转换成组合：

```
1 class Square : private Rectangle {
2     public:
3         void set_side(int side) {
4             // Rectangle's public interface is accessible to the Square
5             set_width(side);
6             set_height(side);
7         }
8         int area() {
9             area_ = Rectangle::area();
10            return area_;
11        }
12    private:
13        int area_;
14 }
```

客户端代码不能访问从 Rectangle 继承的成员：

```
1 Square sq;
2 sq.set_width(14); // compile error, the Square has no such public member
3 make_big_rectangle(sq); // compile error, can't cast Square to Rectangle
```

同样可以通过在 Square 的私有部分声明一个 Rectangle 成员来实现：

```
1 class Square {
2     public:
3         void set_side(int side) {
4             rectangle_.set_width(side);
5             rectangle_.set_height(side);
6         }
7         int area() {
8             area_ = rectangle_.area();
9             return area_;
10        }
11    private:
12        Rectangle rectangle_;
13        int area_;
14 }
```

为了使用继承，应该仔细分析使用场景，并回答 is-a 问题。每当要在组合和继承之间做出选择时，请选择组合。

当私有继承时，可以省略修饰符。类的默认访问修饰符是 private，所以 `class Square: private Rectangle {};` 与 `class Square: Rectangle {};` 相反，struct 的默认修饰符是 public。

保护继承

最后是带 `protected` 修饰符的继承，指定在类主体中使用的类成员的访问级别。受保护的成员对类用户是私有的，但对派生类是公有的。如果修饰符用于指定继承类型，则它的行为类似于派生类用户的私有继承。私有继承向所有派生类用户隐藏基类的公共接口，而受保护继承使派生类的后代可以访问基类。

很难想象需要受保护的继承的场景，但您应该将其视为一种工具，它可能在意想不到的明显设计中有用。让我们假设我们需要设计一个堆栈数据结构适配器。堆栈通常基于向量（一维数组）、链表或队列来实现。



TIP 堆栈符合后进先出规则，即插入到堆栈中的最后一个元素将首先访问。类似地，插入到堆栈中的第一个元素将最后访问。我们将在第 6 章更详细地讨论数据结构和数据结构适配器。

栈本身并不代表数据结构，它位于数据结构之上，并通过限制、修改或扩展其功能来调整其使用。下面是 `Vector` 类的一个简单声明，表示一维整数数组：

```
1 class Vector {  
2     public:  
3         Vector();  
4         Vector(const Vector&);  
5         Vector(Vector&&);  
6         Vector& operator=(const Vector&);  
7         Vector& operator= (Vector&&);  
8         ~Vector();  
9     public:  
10        void push_back(int value);  
11        void insert(int index, int value);  
12        void remove(int index);  
13        int operator[] (int index);  
14        int size() const;  
15        int capacity() const;  
16    private:  
17        int size_;  
18        int capacity_;  
19        int* array_;  
20    };
```

前面的 `Vector` 不是支持随机访问迭代器的 STL 兼容容器，包含动态递增数组的最小值。它可以通过以下方式声明和使用：

```
1 Vector v;  
2 v.push_back(4);  
3 v.push_back(5);  
4 v[1] = 2;
```

`Vector` 类提供了操作符 `[]`，允许随机访问任何项，而堆栈禁止随机访问。栈提供了 `push` 和 `pop` 操作，因此可以分别向其底层数据结构中插入一个值并获取该值：

```
1 class Stack : private Vector {
```

```

2 public:
3 // constructors, assignment operators and the destructor are omitted for brevity
4 void push(int value) {
5     push_back(value);
6 }
7 int pop() {
8     int value{this[size() - 1]};
9     remove(size() - 1);
10    return value;
11 }
12 };

```

栈的使用方式如下:

```

1 Stack s;
2 s.push(5);
3 s.push(6);
4 s.push(3);
5 std::cout << s.pop(); // outputs 3
6 std::cout << s.pop(); // outputs 6
7 s[2] = 42; // compile error, the Stack has no publicly available operator []
8 defined

```

堆栈自适应 Vector，并提供两个成员函数，以便可以访问。私有继承允许使用 Vector 的全部功能，并向堆栈用户隐藏继承信息。如果想要继承堆栈来创建它的高级版本，该怎么办？假设 AdvancedStack 类提供了 min() 函数，该函数在常量时间内返回堆栈中包含的最小值。

私有继承禁止 AdvancedStack，使用 Vector 的公共接口，因此需要一种方法允许 Stack 子类使用基类，但对类用户隐藏基类的存在。保护继承实现了这一目标，如下面的代码所示：

```

1 class Stack : protected Vector {
2     // code omitted for brevity
3 };
4 class AdvancedStack : public Stack {
5     // can use the Vector
6 };

```

通过从 Vector 继承堆栈，允许堆栈的子类使用 Vector 公共接口。但是堆栈和 AdvancedStack 的用户都不能以向量的形式访问。

多态性

多态是面向对象编程中的另一个关键概念，允许子类拥有自己的基类派生函数的实现。假设我们有一个 Musician 类，它有 play() 成员函数：

```

1 class Musician {
2     public:
3     void play() { std::cout << "Play an instrument"; }
4 };

```

现在，声明一个 Guitarist 类，它有 play_guitar() 函数：

```
1 class Guitarist {
2     public:
3     void play_guitar() { std::cout << "Play a guitar"; }
4 };
```

这是使用继承的明显例子，因为 `Guitarist` 是一个 `Musician`。`Guitarist` 自然不会通过添加新函数（如 `play_guitar()`）来扩展 `Musician`。相反，应该提供自己从 `Musician` 派生的 `play()` 函数的实现。为此，我们使用虚函数：

```
1 class Musician {
2     public:
3     virtual void play() { std::cout << "Play an instrument"; }
4 };
5 class Guitarist : public Musician {
6     public:
7     void play() override { std::cout << "Play a guitar"; }
8 };
```

现在，`Guitarist` 类提供了自己的 `play()` 函数实现，客户端代码可以通过使用基类的指针访问它：

```
1 Musician armstrong;
2 Guitarist steve;
3 Musician* m = &armstrong;
4 m->play();
5 m = &steve;
6 m->play();
```

前面的例子展示了多态性的实际应用。虽然虚函数的使用很自然，但除非我们正确地使用它，否则它实际上没有多大意义。首先，`Musician` 的 `play()` 函数不应该有任何实现。原因很简单：音乐家应该能够演奏一种具体的乐器，因为他们不能同时演奏一种以上的乐器。为了避免实现，我们将函数设置为纯虚函数，为其赋值 0：

```
1 class Musician {
2     public:
3     virtual void play() = 0;
4 };
```

当客户端代码试图声明音乐家的实例时，这会导致编译错误，因为不能够创建一个具有未定义函数的对象。`Musician` 的作用只有一个：只能继承。存在的要继承的类称为抽象类。实际上，`Musician` 称为接口，而不是抽象类。抽象类是一种半接口半类的存在，可以拥有两种类型的函数：有实现的函数和没有实现的函数。

回到我们的例子，让我们添加 `Pianist` 类，它也实现了 `Musician` 接口：

```
1 class Pianist : public Musician {
2     public:
3     void play() override { std::cout << "Play a piano"; }
4 };
```

为了表达多态性的全部，假设在某个地方声明了一个函数，它返回一个 Musician 的集合，Guitarist 或 Pianist：

```
1 std::vector<Musician*> get_musicians();
```

从客户端代码的角度来看，很难分析 get_musicians() 函数的返回值并找出对象的实际子类型。可能是吉他手或钢琴家，甚至是一个纯粹的音乐家。关键是客户端不应该真正关心对象的实际类型，因为它知道集合中包含了乐手，而且乐手对象具有 play() 函数。因此，要让它们发挥作用，客户机只需遍历这个集合，并让每个乐师演奏它的乐器（每个对象调用它的实现）：

```
1 auto all_musicians = get_musicians();
2 for (const auto& m: all_musicians) {
3     m->play();
4 }
```

前面的代码表达了多态性的全部功能。现在，让我们了解该如何在语言在底层支持多态。

虚函数

尽管多态性并不局限于虚函数，但我们将它们放在一起讨论。再次强调，更好地理解概念或技术的最好方法是自己实现它。无论是在类中声明虚成员函数，还是在基类中声明虚函数，编译器都会给该类增加一个额外的指针。指针指向一个表，这个表通常称为虚函数表，或者简单地称为虚表。我们也将该指针称为虚表指针。

我们假设我们正在为银行客户账户管理实现一类子系统。假设银行要求我们根据帐户类型实现兑现。例如，储蓄账户允许每年提现一次，而支票账户允许客户随时提现。在不了解任何关于 Account 类（不必要的）细节的情况下，让我们声明一个可以帮助理解虚拟成员函数的最小构造。让我们看看 Account 类的定义：

```
1 class Account
2 {
3     public:
4         virtual void cash_out() // the default implementation for cashing out
5     }
6     virtual ~Account() {}
7     private:
8         double balance_;
9 };
```

编译器将 Account 类转换为一个具有指向虚函数表指针的结构。下面的代码代表伪代码，解释了当我们在类中声明虚函数时会发生什么。和往常一样，请注意提供的是通用的解释，而不是特定于编译器的实现（名称重置整也是通用形式，例如：将 cash_out 更名为 Account_cash_out）：

```
1 struct Account
2 {
3     VTable* __vptr;
4     double balance_;
5 };
6
7 void Account_constructor(Account* this) {
```

```

8   this->_vptr = &Account_VTable;
9 }
10
11 void Account_cash_out(Account* this) {
12   // the default implementation for cashing out
13 }
14
15 void Account_destructor(Account* this) {}

```

仔细看看前面的伪代码。Account 结构体的第一个成员是 _vptr。由于前面声明的 Account 类有两个虚函数，所以可以将虚表想象为一个数组，其中有两个指向虚成员函数的指针。详见以下表示：

```

1 VTable Account_VTable[] = {
2   &Account_cash_out,
3   &Account_destructor
4 };

```

根据之前的假设，看看在对象上调用虚函数时编译器会生成什么代码：

```

1 // consider the get_account() function as already implemented and returning
2 an Account*
3 Account* ptr = get_account();
4 ptr->cash_out();

```

下面是我们可以想象的编译器生成的代码和前面的代码是一样的：

```

1 Account* ptr = get_account();
2 ptr->_vptr[0]();

```

虚函数在层次结构中使用时显示了它们的威力。SavingsAccount 从 Account 类继承如下：

```

1 class SavingsAccount : public Account
2 {
3 public:
4   void cash_out() override {
5     // an implementation specific to SavingsAccount
6   }
7   virtual ~SavingsAccount() {}
8 };

```

当通过指针（或引用）调用 cash_out() 时，虚函数将基于指针所指向的目标对象调用。例如，假设 get_savings_account() 返回一个 SavingsAccount 为 Account*。下面的代码将调用 cash_out() 的 SavingsAccount 实现：

```

1 Account* p = get_savings_account();
2 p->cash_out(); // calls SavingsAccount version of the cash_out

```

下面是编译器为 SavingsClass 生成的内容：

```

1 struct SavingsAccount
2 {

```

```

3     Account __parent_subobject__;
4     VTable* __vptr;
5 };
6 VTable* SavingsAccount_VTable[] = {
7     &SavingsAccount_cash_out,
8     &SavingsAccount_destructor,
9 };
10 void SavingsAccount_constructor(SavingsAccount* this) {
11     this->__vptr = &SavingsAccount_VTable;
12 }
13 void SavingsAccount_cash_out(SavingsAccount* this) {
14     // an implementation specific to SavingsAccount
15 }
16 void SavingsAccount_destructor(SavingsAccount* this) {}

```

我们有两个不同的虚函数表。当创建 Account 类型的对象时，它的__vptr 指向 Account_VTable，而 SavingsAccount 类型的对象的__vptr 指向 SavingsAccount_VTable。让我们来看看下面的代码：

```
1 p->cash_out();
```

前面的代码转译成这样：

```
1 p->__vptr[0]();
```

现在，很明显__vptr[0] 解析为正确的函数，因为它是通过 p 指针读取的。

如果 SavingsAccount 没有覆盖 cash_out() 函数怎么办？在这种情况下，编译器只是将基类实现的地址与 SavingsAccount_VTable 放在同一个槽中，如下所示：

```

1 VTable* SavingsAccount_VTable[] = {
2     // the slot contains the base class version
3     // if the derived class doesn't have an implementation
4     &Account_cash_out,
5     &SavingsAccount_destructor
6 };

```

编译器以不同的方式实现虚拟函数的表示和管理。有些实现甚至使用不同的模型，而不是我们前面介绍的模型。我们引入了一种流行的方法，为了简单起见，我们用通用的方式来表示它。现在，我们将看看在包含动态多态性的代码背后发生了什么。

设计模式

设计模式是程序员最具表现力的工具之一，允许我们以一种优雅且经过良好测试的方式解决设计问题。当努力提供类及其关系的最佳设计时，著名的设计模式可能会帮助您。

设计模式最简单的例子是单例。它为我们提供了一种声明和使用类的一个实例的方法，例如：假设电子商务平台只有一个仓库。为了访问 Warehouse 类，项目可能要求我们在许多源文件中包含并使用它。为了保持同步，我们应该将仓库设置为单例：

```

1 class Warehouse {
2 public:
3     static create_instance() {

```

```

4   if (instance_ == nullptr) {
5     instance_ = new Warehouse();
6   }
7   return instance_;
8 }
9 static remove_instance() {
10   delete instance_;
11   instance_ = nullptr;
12 }
13 private:
14   Warehouse() = default;
15 private:
16   static Warehouse* instance_ = nullptr;
17 };

```

我们声明了一个静态 Warehouse 对象，和两个用于创建和销毁相应实例的静态函数。每当用户试图以旧的方式声明 Warehouse 对象时，private 构造函数都会阻止编译。为了能够使用 Warehouse，客户端代码必须调用 create_instance() 函数：

```

1 Warehouse* w = Warehouse::create_instance();
2 Product book;
3 w->add_product(book);
4 Warehouse::remove_instance();

```

Warehouse 的单例实现并不完整，只是一个引入设计模式的示例。我们将在本书中介绍更多的设计模式。

总结

本章中，我们讨论了面向对象编程的基本概念。讨论了类的底层细节和 C++ 对象模型的编译器实现。知道如何在没有实际类的情况下，设计和实现类对正确使用类有很大帮助。

我们还讨论了对继承的需求，并尝试在可能适用的情况下使用组合而不是继承。C++ 支持三种类型的继承：公有、私有和保护。所有这些类型在特定的类设计中都有它们的应用。最后，我们通过一个例子来理解多态性的使用和强大功能，这个例子极大地增加了客户机代码的便利性。

下一章中，我们将学习更多关于模板和模板元编程的知识，我们将以此为基础深入研究 C++20 的一个新特性，这个特性叫做概念。

问题

1. 对象的三个属性是什么？
2. 移动对象比复制对象有什么优势？
3. C++ 中，结构和类有什么区别？
4. 聚合关系和合成关系有什么区别？
5. 私有继承和受保护的继承有什么区别？
6. 如果在类中定义虚函数，类的大小会受到什么影响？
7. 使用单例设计模式的意义是什么？

扩展阅读

更多信息，请参阅：

- Grady Booch,《面向对象的分析和设计》(<https://www.amazon.com/Object-Oriented-Analysis-Design-Applications-3rd/dp/020189551X/>)
- Stanley Lippman,《C++ 对象的内部模型》(<https://www.amazon.com/Inside- Object-Model-Stanley-Lippman/dp/0201834545/>)

第 4 章：了解并设计模板

模板是 C++ 的特性，这样函数和类可以支持泛型——可以实现独立于特定数据类型的函数或类，例如：客户端可以使用 `max()` 函数来处理不同的数据类型。不需要使用函数重载来实现和维护许多类似的函数，我们只需要实现一个 `max()` 并将数据类型作为参数传递。此外，模板可以与多重继承和操作符重载一起工作，从而在 C++ 中可以创建强大的通用数据结构和算法，如：标准模板库 (STL)。此外，模板还可以应用于编译时计算、编译时和运行时代码优化等。

本章中，我们将学习函数和类模板的语法、它们的实例化和特化。然后，我们将介绍可变参数模板及其应用。接下来，将讨论模板参数和用于实例化它们的相应参数。之后，将学习如何实现一个类型特征，以及如何使用这种信息来优化算法。最后，将介绍一些可以用来提高程序执行速度的技术，包括编译时计算、编译时代码优化和静态多态性。

本章中，我们将了解以下内容：

- 探索函数和类模板
- 了解可变模板
- 理解模板形参和参数
- 什么是特征？
- 模板元编程及其应用

源码位置

本章的代码可以在这本书的 GitHub 中找到：<https://github.com/PacktPublishing/Expert-CPP>。

探索函数和类模板

我们将从介绍函数模板的语法开始，并通过实例化、推断类型和特化开始这一节。然后，转向类模板，看看类似的概念，以及示例。

动机

目前为止，定义了函数或类时，必须提供输入、输出和中间参数。例如，有一个函数来执行两个 `int` 型整数的加法运算。如何扩展它，使它能够处理所有其他基本数据类型，如 `float`、`double`、`char` 等？一种方法是通过手动复制、粘贴和稍微修改每个函数来使用函数重载。另一种方法是定义一个宏来执行加法操作。这两种方法都有各自的副作用。

此外，如果我们修复了一个 bug 或为一种类型添加了一个新特性，而这个更新需要对所有其他重载函数和类执行，这会导致什么情况？除了使用复制-粘贴-替换的方法，有更好的方法来处理这种情况吗？

事实上，这是任何计算机语言都可能面临的问题。这个问题由通用函数编程元语言 (ML, Meta Language) 在 1973 年提出，ML 允许编写仅在使用时的类型集中使用不同的通用函数或类型，从而减少重复代码。后来，受 Ada 语言中生命保护 (CLU, hartered life underwriter) 提供的参数化模块和泛型的启发，C++ 采用了模板概念，它允许函数和类使用泛型类型进行操作。换句话说，它允许函数或类处理不同的数据类型，而不需要重新编写它们。

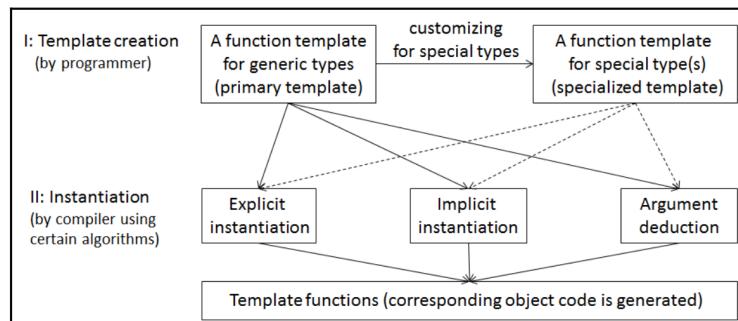
实际上，从抽象的角度来看，C++ 函数或类模板可以作为创建其他类似函数或类的模式。这背后的基本思想是创建一个函数或类模板，而不必指定某些或所有变量的确切类型。我们使用占位符类型定义函数或类模板，称为模板类型形参。一旦有了函数或类模板，就可以使用在其他编译器中实现的算法自动生成函数或类。

C++ 中有三种模板：函数模板、类模板和可变参数模板。接下来我们一个个的来了解。

函数模板

函数模板定义了如何生成一系列函数。这里是指一组行为相似的函数。如下图所示，这包括两个阶段：

- 创建函数模板，书写的规则。
- 模板实例化，从模板中生成函数的规则：



上图的 I 中，讨论了用于为泛型类型创建函数模板的格式，但与特化模板有关，我们也将特化模板称为主模板。II 中，我们介绍从模板生成函数的三种方法。最后，特化和重载小节告诉我们如何为特殊类型定制主模板（通过改变其行为）。

语法

定义函数模板有两种方法，如下面的代码所示：

```

1 template <typename identifier_1 , … , typename identifier_n >
2 function_declaration ;
3
4 template <class identifier_1 ,… , class identifier_n>
5 function_declaration ;
  
```

这里，`identifier_i` ($i=1, \dots, n$) 是类型或形参，而 `function_declaration` 声明了部分函数体。前面两个声明的唯一区别是关键字——一个使用 `class`，另一个使用 `typename`，两者具有相同的含义和行为。因为类型（例如基本类型——`int`、`float`、`double`、`enum`、`struct`、`union` 等）不是类，所以引入了 `typename` 关键字方法以避免混淆。

例如，查找最大值函数模板 `app_max()` 可以这样声明：

```

1 template <class T>
2 T app_max (T a, T b) {
3     return (a>b?a:b); //note: we use ((a)>(b) ? (a):(b)) in macros
4 }                      //it is safe to replace (a) by a, and (b) by b now
  
```

这个函数模板可以用于许多数据类型或类，只要有一个可复制构造的类型，其中表达式是 $a > b$ 。对于用户定义的类，这意味着必须定义大于操作符 ($>$)。

注意，函数模板和模板函数是不同的东西。函数模板是一种编译器用来生成函数的模板，因此编译器不会为它生成任何目标代码。另一方面，模板函数是指函数模板中的一个实例。由于它是一个函数，相应的目标代码由编译器生成。然而，最新的 C++ 标准文档建议避免使用不准确的术语定义模板函数。因此，本书将使用函数模板和成员函数模板。

实例化

因为可能有无限多个类型和类，函数模板的概念不仅节省了源代码的空间，而且使代码更容易阅读和维护。然而，与为应用程序中使用的不同数据类型编写单独的函数或类相比，模板不会产生更小的目标代码。例如，考虑一个使用 float 和 int 版本的 app_max() 的程序：

```
1 cout << app_max<int>(3,5) << endl;
2 cout << app_max<float>(3.0f,5.0f) << endl;
```

编译器将在目标文件中生成两个新函数，如下所示：

```
1 int app_max<int> ( int a, int b ) {
2     return (a>b?a:b);
3 }
4
5 float app_max<float> ( float a, float b ) {
6     return (a>b?a:b);
7 }
```

从函数模板声明创建函数的过程称为模板实例化。这个实例化过程中，编译器确定模板参数，并根据应用程序的需要生成实际的函数代码。实例化通常有三种形式：显式、隐式和推断。我们接下来讨论每一种形式。

显式实例化

很多非常有用的 C++ 函数模板可以在不使用显式实例化的情况下编写和使用，这里对它们进行了解，以便在需要时知道它们的存在。首先，看一下 C++11 之前显式实例化的语法。有两种形式，代码如下所示：

```
1 template return-type
2 function_name < template_argument_list > ( function_parameter-list );
3
4 template return-type
5 function_name ( function_parameter_list );
```

显式实例化定义，强制对特定类型的函数模板进行实例化，而不管将来会调用哪个模板函数。显式实例化可以在函数模板定义之后的任何位置，并且对于源代码中的给定参数列表，它只允许出现一次。

C++11 之后，显式实例化的语法如下所示，关键字 extern 放在关键字 template 之前：

```
1 extern template return-type
```

```
2 function_name < template_argument_list > (function_parameter_list ); \\ (since C
3           ++11)
4
4 extern template return-type
5 function_name ( function_parameter_list ); \\ (since C++11)
```

使用 `extern` 关键字可以防止隐式实例化该函数模板 (参阅下一节)。

前面声明的 `app_max()` 函数模板，可以使用以下代码显式实例化：

```
1 template double app_max<double>(double, double);
2 template int app_max<int>(int, int);
```

也可以使用以下代码显式实例化：

```
1 extern template double app_max<double>(double, double); // (since c++11)
2 extern template int app_max<int>(int, int); // (since c++11)
```

也可以通过模板参数推导的方式完成：

```
1 template double f(double, double);
2 template int f(int, int);
```

最后，也可以这样做：

```
1 extern template double f(double, double); // (since c++11)
2 extern template int f(int, int); // (since c++11)
```

此外，还有一些用于显式实例化的规则。如果想了解更多，请参考扩展阅读部分的了解更多信息。

隐式实例化

当一个函数调用时，该函数的定义必须存在。如果这个函数没有显式实例化，则会进行隐式实例化。隐式实例化中，模板参数列表需要显式地提供或从上下文推导出来参数类型。下面程序的 A 部分提供了 `app_max()` 隐式实例化的例子：

```
1 //ch4_2_func_template_implicit_inst.cpp
2 #include <iostream>
3 template <class T>
4 T app_max ( T a, T b) { return (a>b?a:b); }
5 using namespace std ;
6 int main(){
7     //Part A: implicit instantiation in an explicit way
8     cout << app_max<int>(5, 8) << endl; //line A
9     cout << app_max<float>(5.0, 8.0) << endl; //line B
10    cout << app_max<int>(5.0, 8) << endl; //Line C
11    cout << app_max<double>(5.0, 8) << endl; //Line D
12    //Part B: implicit instantiation in an argument deduction way
13    cout << app_max(5, 8) << endl; //line E
14    cout << app_max(5.0f, 8.0f) << endl; //line F
15    //Part C: implicit instantiation in a confuse way
16    //cout<<app_max(5, 8.0)<<endl; //line G
```

```
17     return 0;  
18 }
```

行 A、B、C 和 D 的隐式实例化分别为 `int app_max<int>(int,int)`、`float app_max<float>(float, float)`、`int app_max<int>(int,int)` 和 `double app_max<double>(double, double)`。

类型推断

当调用模板函数时，即使不是每个模板实参都指定了，编译器也需要找出具体的实参。大多数情况下，它会从函数参数中推断出缺少的模板参数。例如，在前面函数的 B 部分中，当在 E 行中调用 `app_max(5,8)` 时，编译器将模板实参推断为 `int` 类型 (`int app_max<int>(int,int)`)，因为输入形参 5 和 8 都是整数。类似地，F 行会将类型推断为 `float`，即 `float app_max<float>(float,float)`。

如果在实例化过程中出现混乱怎么办？例如，前一个程序的 G(注释) 行，根据编译器的不同，可以调用 `app_max<double>(double, double)`、`app_max<int>(int, int)`，或者抛出编译错误消息。帮助编译器推断类型的最好方法是通过显式地给出模板实参来调用函数模板。这种情况下，如果调用 `app_max<double>(5,8.0)`，任何混淆都将得到解决。



TIP 从编译器的角度来看，有几种方法可以进行模板参数推断——从函数调用推断、从类型推断、自动类型推断和上下文。然而，从开发者角度来看，永远不要编写花哨的代码来使用函数模板来迷惑其他开发者，比如：前面的例子中的第 G 行。

特化和重载

特化允许为一组给定的模板实参定制模板代码，允许我们为特定的模板参数定义特定的行为。特化仍然是模板，仍然需要实例化来获得真正的代码（由编译器自动完成）。

下面的示例代码中，主函数模板 `T app_max(T a, T b)` 将根据操作符 `a>b` 返回 `a` 或 `b`，可以将其特化为 `T = std::string`，这样我们只比较 `a` 和 `b` 的第 0 个元素，即 `a[0] >b[0]` 就可以了：

```
1 //ch4_3_func_template_specialization.cpp  
2 #include <iostream>  
3 #include <string>  
4 //Part A: define a primary template  
5 template <class T> T app_max (T a, T b) { return (a>b?a:b); }  
6 //Part B: explicit specialization for T=std::string,  
7 template <>  
8 std::string app_max<std::string> (std::string a, std::string b){  
9     return (a[0]>b[0]?a:b);  
10 }  
11 //part C: test function  
12 using namespace std;  
13 void main(){  
14     string a = "abc", b="efg";  
15     cout << app_max(5, 6) << endl; //line A  
16     cout << app_max(a, b) << endl; //line B  
17     //question: what's the output if un-comment lines C and D?
```

```

18 //char *x = "abc", *y="efg"; //Line C
19 //cout << app_max(x, y) << endl; //line D
20 }

```

前面的代码定义了主模板，然后显式特化为 std::string，我们不比较 a 和 b 的值，只关心 a[0] 和 b[0] (app_max() 的行为是特化的)。在测试函数中，行 A 调用 app_max<int>(int,int)，行 B 调用专用版本，因为在推导时没有歧义。如果取消注释 C 和 D 行，则会调用主函数模板 char* app_max<char> (char*, char*)，因为 char* 和 std::string 是不同的数据类型。

本质上，特化与函数重载有些冲突：编译器需要一种算法来解决这种冲突，方法是在模板和重载函数之间找到正确的匹配。选择正确函数的算法包括以下两个步骤：

1. 常规函数和非特化模板之间执行重载解析。
2. 如果选择了非特化模板，请检查是否存在与之更匹配的特化模板。

例如，下面的代码块中，我们声明了主函数 (第 0 行) 和特化函数模板 (第 1-4 行)，以及 f() 的重载函数 (第 5-6 行)：

```

1 template<typename T1, typename T2> void f( T1, T2 ); // line 0
2 template<typename T> void f( T ); // line 1
3 template<typename T> void f( T, T ); // line 2
4 template<typename T> void f( int , T* ); // line 3
5 template<> void f<int>( int ); // line 4
6 void f( int , double ); // line 5
7 void f( int ); // line 6

```

f() 将在下面的代码块中多次调用。根据前面的两步规则，可以在注释中显示选择了哪个函数。我们将在下面解释这样做的原因：

```

1 int i=0;
2 double d=0;
3 float x=0;
4 complex<double> c;
5 f(i); //line A: choose f() defined in line 6
6 f(i,d); //line B: choose f() defined in line 5
7 f<int>(i); //line C: choose f() defined in line 4
8 f(c); //line D: choose f() defined in line 1
9 f(i,i); //line E: choose f() defined in line 2
10 f(i,x); //line F: choose f() defined in line 0
11 f(i, &d); //line G: choose f() defined in line 3

```

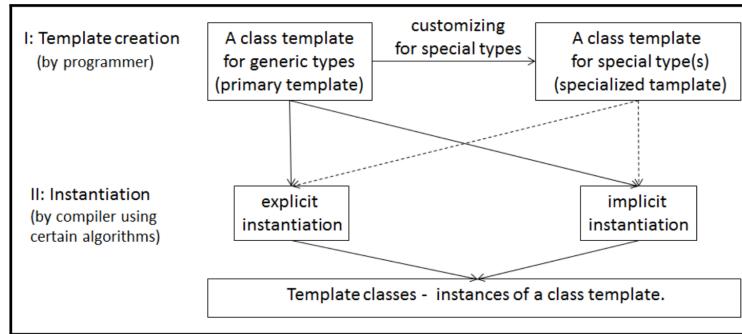
对于直线 A 和直线 B，由于第 5 行和第 6 行定义的 f() 是常规函数，它有最高的优先级，因此 f(i) 和 f(i,d) 将选择它。对于行 C，因为特化模板存在，所以从第 4 行生成的 f() 比从第 1 行创建的 f() 更匹配。对于第 D 行，因为 c 是一个 complex<double> 类型，只有第 1 行中定义的主函数模板匹配它。E 行选择第 2 行创建的 f()，因为这两个输入变量是同一类型的。最后，第 F 行和第 G 行将分别从第 0 行和第 3 行模板中获取函数。

了解了函数模板之后，来看看类型模板。

类型模板

类模板定义了一系列类，通常用于实现容器。例如，C++ 标准库包含许多类模板，如 std::vector、std::map、std::deque 等。OpenCV 中，cv::Mat 是一个非常强大的类模板，可以处理带有内置数据类型如 int8_t、uint8_t、int16_t、uint16_t、int32_t、uint32_t、float、double 等的 1D、2D 和 3D 矩阵或图像。

类似于函数模板，如下图所示，类模板的概念包含一个模板创建语法，它的特化，以及它的隐式和显式实例化：



图的 **I** 中，使用特定的语法格式，可以为泛型类型创建类模板，也称为主模板，并且可以为具有不同成员函数和/或变量的特殊类型定制。**II** 中，当我们有了类模板，编译器就会根据应用程序的需求显式或隐式地将模板实例化。

现在，了解一下创建类模板的语法。

语法

创建类模板的语法如下：

```
1 [export] template <template_parameter_list> class-declaration
```

这里，我们有以下这些方式：

- **template_parameter-list**(请参扩展阅读中的链接) 是一个非空的以逗号分隔的模板形参列表，每个模板形参要么是一个非类型形参，要么是一个类型形参，或是一个模板形参。
- **class-declaration** 用来声明类，这个类包含一个类名和用花括号括起来的类体。通过这样做，声明的类名也变成了模板名。

例如，可以定义一个类模板 V，使其包含各种 1D 数据类型：

```

1 template <class T>
2 class V {
3 public:
4     V( int n = 0 ) : m_nEle(n), m_buf(0) { creatBuf(); }
5     ~V() { deleteBuf(); }
6     V& operator = (const V &rhs) { /* ... */ }
7     V& operator = (const V &rhs) { /* ... */ }
8     T getMax() { /* ... */ }
9 protected:
10    void creatBuf() { /* ... */ }
11    void deleteBuf() { /* ... */ }
12 public:

```

```
13 int m_nEle;
14 T * m_buf;
15 };
```

有了这个模板类，编译器就可以在实例化过程中生成类。由于在函数模板小节中提到的原因，本书中我们将避免使用不精确的术语定义模板类。

实例化

回顾一下上一节定义的类模板 V，假设后面会出现以下声明：

```
1 V<char> cV;
2 V<int> iV(10);
3 V<float> fV(5);
```

然后，编译器将创建 V 类的三个实例，如下所示：

```
1 class V<char>{
2 public:
3     V(int n=0);
4     // ...
5 public:
6     int m_nEle;
7     char *m_buf;
8 };
9 class V<int>{
10 public:
11     V(int n=0);
12     // ...
13 public:
14     int m_nEle;
15     int *m_buf;
16 };
17 class V<float>{
18 public:
19     V(int n = 0);
20     // ...
21 public:
22     int m_nEle;
23     float *m_buf;
24 };
```

与函数模板实例化类似，类模板实例化有两种形式：显式和隐式。

显式实例化

显式实例化的语法如下：

```
1 template class template_name < argument_list >;
2 extern template class template_name < argument_list > // (since C++11)
```

显式实例化定义强制实例化的所引用的类、结构体或联合体。C++11 标准中，模板特化或其成员的隐式实例化将被抑制。与函数模板的显式实例化类似，该显式实例化的位置可以在其模板定义之后的任何位置，并且只允许在整个程序的一个文件中定义。

而且，由于 C++11 隐式实例化将显式的声明 (extern 模板) 绕过。这可以用来减少编译时间。

回到模板类 V，我们可以显式地实例化它：

```
1 template class V<int>;
2 template class V<double>;
```

或者，我们可以这样做 (从 C++11 开始)：

```
1 extern template class V<int>;
2 extern template class V<double>;
```

如果显式实例化一个函数或类模板，但在程序中没有相应的定义，编译器将报出错误消息，如下所示：

```
1 //ch4_4_class_template_explicit.cpp
2 #include <iostream>
3 using namespace std;
4 template <typename T> //line A
5 struct A {
6     A(T init) : val(init) {}
7     virtual T foo();
8     T val;
9 }; //line B
10 //line C
11 template <class T> //T in this line is template parameter
12 T A<T>::foo() { //the 1st T refers to function return type,
13     //the T in <> specifies that this function's template
14     //parameter is also the class template parameter
15     return val;
16 } //line D
17 extern template struct A<int>; //line E
18 #if 0 //line F
19 int A<int>::foo() {
20     return val+1;
21 }
22 #endif //line G
23 int main(void) {
24     A<double> x(5);
25     A<int> y(5);
26     cout<<"fD="<<x.foo()<<, fI="<<y.foo()<< endl;
27     return 0; //output: fD=5,fI=6
28 }
```

前面的代码中，在 A 行和 B 行之间定义了一个类模板，然后从 C 行到 D 行实现了成员函数 foo()。接下来，在 E 行为 int 类型显式实例化。由于在 F 行和 G 行之间的代码块注释掉了（这意味着对于这个显式的 int 类型实例化没有相应的 foo() 定义），我们有看到了链接错误。为了解决这

个问题，我们需要在 F 行用 `#if 1` 替换 `#if 0`。

最后，对于显式实例化声明还有一些限制：

- **静态函数**：可以命名静态类成员，但不能在显式实例化声明中指定静态函数。
- **内联函数**：显式实例化声明中对内联函数没有影响，而内联函数是隐式实例化的。
- **类及其成员**：显式实例化类及其所有成员并不等价。

隐式实例化

引用模板类时，编译器会在模板没有显式实例化或特化的情况下，按需生成代码，这称为隐式实例化：

```
1 class<argument list> object_name; //for non-pointer object
2 class<argument list> *p_object_name; //for pointer object
```

对于非指针对象，实例化模板类并创建对象，只生成该对象使用的成员函数。对于指针对象，除非使用了其成员，否则不会实例化。

以下示例中，我们在 ch4_5_class_template_implicit_insta.h 文件中定义了一个类模板 X：

```
1 //file ch4_5_class_template_implicit_inst.h
2 #ifndef __CH4_5_H__
3 #define __CH4_5_H__
4 #include <iostream>
5 template <class T>
6 class X {
7 public:
8     X() = default;
9     ~X() = default;
10    void f() { std::cout << "X::f()" << std::endl; };
11    void g() { std::cout << "X::g()" << std::endl; };
12};
13#endif
```

然后，它包含在以下四个 cpp 文件中，每个 cpp 文件中都有 main() 函数：

```
1 //file ch4_5_class_template_implicit_inst_A.cpp
2 #include "ch4_5_class_template_implicit_inst.h"
3 void main()
{
    //implicit instantiation generates class X<int>, then create object xi
    X<int> xi ;
    //implicit instantiation generates class X<float>, then create object
    xf
    X<float> xf;
    return 0;
}
```

在 ch4_5_class_template_implicit_inst_A.cpp 中，编译器将隐式实例化 `X<int>` 和 `X<float>` 类，然后创建 `xi` 和 `xf` 对象。但是因为 `X::f()` 和 `X::g()` 没有使用，所以其实是没有实例化的。

现在，来看看 ch4_5_class_template_implicit_inst_B.cpp：

```

1 //file ch4_5_class_template_implicit_inst_B.cpp
2 #include "ch4_5_class_template_implicit_inst.h"
3 void main()
{
    //implicit instantiation generates class X<int>, then create object xi
    X<int> xi;
    xi.f(); //and generates function X<int>::f(), but not X<int>::g()
    //implicit instantiation generates class X<float>, then create object
    //xf and generates function X<float>::g(), but not X<float>::f()
    X<float> xf;
    xf.g();
}

```

译器将隐式实例化 `X<int>` 类, 创建 `xi` 对象, 然后生成 `X<int>::f()` 函数, 但不生成 `X<int>::g()`。类似地, 它将实例化 `X<float>` 类, 创建 `xf` 对象, 并生成 `X<float>::g()` 函数, 但不生成 `X<float>::f()`。

然后, 是 `ch4_5_class_template_implicit_inst_C.cpp`:

```

1 //file ch4_5_class_template_implicit_inst_C.cpp
2 #include "ch4_5_class_template_implicit_inst.h"
3 void main()
{
    //inst. of class X<int> is not required, since p_xi is pointer object
    X<int> *p_xi ;
    //inst. of class X<float> is not required, since p_xf is pointer object
    X<float> *p_xf ;
}

```

因为 `p_xi` 和 `p_xf` 是指针对象, 所以不需要通过编译器实例化它们。

最后是 `ch4_5_class_template_implicit_inst_D.cpp` :

```

1 //file ch4_5_class_template_implicit_inst_D.cpp
2 #include "ch4_5_class_template_implicit_inst.h"
3 void main()
{
    //inst. of class X<int> is not required, since p_xi is pointer object
    X<int> *p_xi;

    //implicit inst. of X<int> and X<int>::f(), but not X<int>::g()
    p_xi = new X<int>();
    p_xi->f();

    //inst. of class X<float> is not required, since p_xf is pointer object
    X<float> *p_xf;
    p_xf = new X<float>(); //implicit inst. of X<float> occurs here
    p_xf->f(); //implicit inst. X<float>::f() occurs here
    p_xf->g(); //implicit inst. of X<float>::g() occurs here

    delete p_xi;
    delete p_xf;
}

```

}

这会隐式实例化 `X<int>` 和 `X<int>::f()`，但不实例化 `X<int>::g()`。`X<float>`，`X<float>::f()` 和 `X<float>::g()` 也会实例化。

特化

与函数特化类似，当特定类型作为模板形参传递时，类模板的显式特化定义了不同的实现。然而，它仍然是一个类模板，需要通过实例化来获得真正的代码。

例如，假设有一个 `struct X` 模板，它可以存储任何数据类型的元素，并且它只有一个名为 `increase()` 的成员函数。但是对于 `char` 类型的数据，需要 `increase()` 有不同的实现，并需要向其添加一个名为 `toUpperCase()` 的新成员函数。因此，我们为该类型声明一个类模板的特化：

1. 声明一个主类模板：

```

1  template <typename T>
2  struct X {
3      X(T init) : m(init) {}
4      T increase() { return ++m; }
5      T m;
6  };
7

```

声明了一个主类模板，它的构造函数初始化 `m` 成员变量，`increase()` 给 `m` 加 1 并返回它的值。

2. 接下来，我们需要对 `char` 类型数据进行特化：

```

1  template <> //Note: no parameters inside <>, it tells compiler
2  // "hi i am a fully specialized template"
3  struct X<char> { //Note: <char> after X, tells compiler
4      // "Hi, this is specialized only for type char"
5      X(char init) : m(init) {}
6      char increase() { return (m<127) ? ++m : (m=-128); }
7      char toUpperCase() {
8          if ((m >= 'a') && (m <= 'z')) m += 'A' - 'a';
9          return m;
10     }
11     char m;
12 };
13

```

创建一个特化的（相对于主类模板）类模板，多带有一个成员函数 `toUpperCase()`，仅用于 `char` 类型数据。

3. 现在，运行一个测试：

```

1  int main() {
2      X<int> x1(5); //line A
3      std::cout << x1.increase() << std::endl;
4      X<char> x2('b'); //line B

```

```
5     std :: cout << x2.toUpperCase() << std :: endl;
6     return 0;
7 }
8 }
```

最后，用一个 main() 函数来测试它。第 A 行，x1 是一个从主模板 X<t> 隐式实例化的对象。因为 x1 的初始值。m 是 5,6 将从 x1.increase() 返回。第 B 行，x2 是从特化模板 X<char> 和 x2 的值实例化的对象。m 在执行时是 ‘b’，在调用 x2.toUpperCase() 之后，‘B’ 将是返回值。



可以在 ch4_6_class_template_specialization.cpp 看到这个示例的完整代码。

总之，类模板的显式特化中的使用方式如下所示：

```
1 template <> class [ struct ] class_name<template argument list> { ... };
```

这里，空的模板形参表 template <> 显式地声明为模板特化，< 目标参数列表 > 是要特化的类型形参。例如，在 ex4_6_class_template_specialize.cpp 中，使用以下方法：

```
1 template <> struct X<char> { ... };
```

这里，X 之后的 <char> 标识了要为其声明模板类特化的类型。

此外，在对模板类进行特化时，必须定义所有成员（甚至主模板中相同的成员），因为在模板特化期间，没有继承主模板。

接下来，我们将研究部分特化。这是显式专门化的一般表达。与只有模板实参列表的显式特化相比，部分特化同时需要模板形参列表和实参列表。对于模板实例化，如果用户的模板实参列表匹配模板实参的子集，编译器将选择部分特化模板。然后，编译器将生成来部分特化模板的类定义。

下面的例子中，对于主类模板 A，可以在实参列表中为 const T 部分特化它。注意，它们都有相同的参数列表，即 <typename T>：

```
1 //primary class template A
2 template <typename T> class A{ /* ... */ };
3
4 //partial specialization for const T
5 template <typename T> class A<const T>{ /* ... */ };
```

下面的示例中，主类模板 B 有两个参数:<typename T1, typename T2>。我们通过 T1=int 部分特化它，保持 T2 不变：

```
1 //primary class template B
2 template <typename T1, typename T2> class B{ /* ... */ };
3
4 //partial specialization for T1 = int
5 template <typename T2> class B<int, T2>{ /* ... */ };
```

最后，在下面的示例中，可以看到部分特化中的模板形参数量不必与原始主模板中出现的形参数量匹配。但是，模板实参的数量（出现在尖括号中的类名之后）必须与主模板中的形参的数量和类型匹配：

```

1 //primary class template C: template one parameter
2 template <typename T> struct C { T type; };
3
4 //specialization: two parameters in parameter list
5 //but still one argument (<T[N]>) in argument list
6 template <typename T, int N> struct C<T[N]>
7 {T type; };

```

同样，模板类的部分特化仍然是模板类。必须分别为其成员函数和数字变量提供定义。

我们总结一下到目前为止所学的内容。下表中，可以看到函数模板和类模板的比较：

	函数模板	模板类	注释
声明	template <class T1, class T2> void f(T1 a, T2 b) ...	template <class T1, class T2> class X ...;	声明中定义了一个函 数/类模板，<class T1, class T2> 称为模板形 参。
显式实例化	template void f<int, int >(int, int); 或 extern template void f <int, int>(int, int); // (since C++11)	template class X<int, float>; 或 extern template class X<int, float>; //(since C++11)	在实例化之后，现在有函 数/类，但它们叫做模板函 数/类。
隐式实例化	{ ... f(3, 4.5); f<char, float>(120, 3.14); }	{ ... X<int, float> obj; X<char, float> *p; }	当函数调用或类对象/指 针声明时，如果没有显式 实例化，则使用隐式实例 化。
特化	template <> void f<int, float>(int a, float b) ...	template <> class X <int, float> ... ;	主模板的完全定制版本 (没有参数列表) 仍然需要 实例化。
部分特化	template <class T> void f<T, T>(T a, T b) ...	template <class T> class X <T, T> ... ;	主模板的部分定制版本 (有一个参数列表) 仍然需 要实例化。

这里需要强调五个概念：

- **声明**: 需要遵循定义函数或模板类的语法。此时，函数或模板类本身就不是类型、函数或其他实体。换句话说，源文件中只有模板定义，而没有可以编译为目标文件的代码。
- **隐式实例化**: 对于任何代码，模板必须实例化。此过程中，必须确定模板参数，以便编译器能够生成实际的函数或类。换句话说，这是按需编译的，在给出带有特定模板参数的实例化之前，不会编译模板函数或类的代码。
- **显式实例化**: 告诉编译器使用给定类型实例化模板，而不管是否使用了这些类型。通常，用于提供库。

- **全特化**: 没有模板参数列表 (完全定制)。模板特化最有用的一点是可以为特定的类型参数创建特定的模板。
- **部分特化**: 与全特化类似，但是只有部分形参 (部分定制) 和部分实参。

了解可变模板

上一节中，学习了如何编写固定数量类型参数的函数或类模板。不过，自 C++11 以来，标准泛型函数和类模板可以接受数量可变的类型参数，这称为可变参数模板，C++ 在进一步读取上下文中的扩展。我们将通过示例，了解可变参数模板的语法和用法。

语法

如果函数或模板类接受零个或多个形参，则可以如下定义：

```

1 //a class template with zero or more type parameters
2 template <typename... Args> class X { ... };
3
4 //a function template with zero or more type parameters
5 template <typename... Args> void foo( function param list ) { ... }

```

在这里，`<typename ... Args>` 声明了一个参数包。注意，这里 `Args` 不是关键字，可以使用任何有效的变量名对其进行替换。前面的模板类/函数可以接受任意数量的 `typename`，作为实例化的参数，如下所示：

```

1 X<> x0; //with 0 template type argument
2 X<int, std::vector<int>> x1; //with 2 template type arguments
3
4 //with 4 template type arguments
5 X<int, std::vector<int>, std::map<std::string, std::vector<int>>> x2;
6
7 //with 2 template type arguments
8 foo<float, double>( function argument list );
9
10 //with 3 template type arguments
11 foo<float, double, std::vector<int>>( function argument list );

```

如果可变参数模板至少需要一个类型形参，可以使用以下定义：

```

1 template <typename A, typename... Rest> class Y { ... };
2
3 template <typename A, typename... Rest>
4 void goo( const int a, const float b ) { .... };

```

类似地，可以使用下面的代码来实例化它们：

```

1 Y<int> y1;
2 Y<int, std::vector<int>, std::map<std::string, std::vector<int>>> y2;
3 goo<int, float>( const int a, const float b );
4 goo<int, float, double, std::vector<int>>( const int a, const float b );

```

代码中，使用 variadic 类模板 Y 的实例化中创建了 y1 和 y2 对象，分别带有一个和三个模板参数。对于可变参数函数 goo 模板，将其实例化为两个模板函数，分别带有两个和三个模板参数。

例子

下面可能是最简单的例子，展示了使用可变参数模板来查找任何输入参数列表的最小值。这个例子使用了递归，直到 my_min(double n) 时退出迭代：

```
1 //ch4_7_variadic_my_min.cpp
2 //Only tested on g++ (Ubuntu/Linaro 7.3.0-27 ubuntu1~18.04)
3 //It may have compile errors for other platforms
4 #include <iostream>
5 #include <math.h>
6 double my_min(double n){
7     return n;
8 }
9 template<typename ... Args>
10 double my_min(double n, Args... args){
11     return fmin(n, my_min(args...));
12 }
13 int main() {
14     double x1 = my_min(2);
15     double x2 = my_min(2, 3);
16     double x3 = my_min(2, 3, 4, 5, 4.7, 5.6, 9.9, 0.1);
17     std::cout << "x1=" << x1 << ", x2=" << x2 << ", x3=" << x3 << std::endl;
18     return 0;
19 }
```

printf() 变参数函数可能是 C 和 C++ 中最有用和最强大的函数。然而，它不是类型安全的。在下面的代码块中，我们将采用类型安全的 printf() 示例来演示可变参数模板的用法。和往常一样，首先定义一个基函数 void printf_vt(const char *s)，以结束递归：

```
1 //ch4_8_variadic_printf.cpp part A: base function - recursive end
2 void printf_vt(const char *s)
3 {
4     while (*s) {
5         if (*s == '%' && *(++s) != '%')
6             throw std::runtime_error("invalid format string: missing arguments");
7         std::cout << *s++;
8     }
9 }
```

然后，它的可变参数模板函数 printf_vt() 中，每当有% 时，该值就会打印出来，其余的则传递给它的递归，直到基函数为止：

```
1 //ch4_8_variadic_printf.cpp part B: recursive function
2 template<typename T, typename ... Rest>
3 void printf_vt(const char *s, T value, Rest... rest)
4 {
5     while (*s) {
```

```

6   if (*s == '%' && *(++s) != '%') {
7     std::cout << value;
8     printf_vt(s, rest...); //called even when *s is 0,
9     return; //but does nothing in that case
10  }
11  std::cout << *s++;
12 }
13 }
```

最后，可以使用下面的代码来测试它，并将其与传统的 printf() 进行比较：

```

1 //ch4_8_variadic_printf.cpp Part C: testing
2 int main() {
3   int x = 10;
4   float y = 3.6;
5   std::string s = std::string("Variadic templates");
6   const char* msg1 = "%s can accept %i parameters (or %s), x=%d, y=%f\n";
7   printf(msg1, s, 100, "more", x, y); //replace 's' by 's.c_str()',
8   //to prevent the output bug
9   const char* msg2 = "% can accept % parameters (or %); x=%,y=%\n";
10  printf_vt(msg2, s, 100, "more", x, y);
11  return 0;
12 }
```

上述代码的输出如下：

```

1 p.] i£jU can accept 100 parameters (or more), x=10, y=3.600000
2 Variadic templates can accept 100 parameters (or more); x=10,y=3.6
```

第一行的开始，可以看到 printf() 中的一些 ASCII 字符，因为对应的变量类型%s 应该是一个指向 char 的指针，但我们给它指定了 std::string 类型。要解决这个问题，需要传递 s.c_str()。然而，有了可变参数版本函数，就没有这个问题了。此外，只需要提供%，就好了——至少对于这个实现来说是这样的。

总之，本节简要介绍了可变参数模板及其应用。可变参数模板有以下优点（自 C++ 11 以来）：

- 模板的轻量级扩展
- 演示了在不使用模板和预处理宏的情况下实现模板库的能力。因此，代码能够理解和调试，同时也节省了编译时间。
- 支持 printf() 可变参数函数的类型安全实现。

接下来，我们将探讨模板形参和参数。

探索模板形参和参数

前两节中学习了函数和模板类以及它们的实例化。在定义模板时，需要给出形参列表。当实例化时，必须提供相应的参数列表。本节中，将进一步研究这两个列表的分类和细节。

模板参数

回想一下下面的语法，它是用来定义模板类/函数的。模板关键字后面有一个 `<>` 符号，必须给出一个或多个模板形参：

```
1 //class template declaration
2 template <parameter-list> class-declaration
3
4 //function template declaration
5 template <parameter-list> function-declaration
```

形参列表中的可以是以下三种类型之一：

- 非类型模板形参：指引用静态实体的编译时常量值，如整数和指针。这些参数通常称为非类型参数。
- 类型模板形参：它引用内置类型名或用户定义的类。
- 模板的模板参数：表示该参数为其他模板。

我们将在下面的小节中更详细地讨论这些问题。

非类型模板参数

非类型模板形参的语法如下：

```
1 //for a non-type template parameter with an optional name
2 type name(optional)
3
4 //for a non-type template parameter with an optional name
5 //and a default value
6 type name(optional)=default
7
8 //For a non-type template parameter pack with an optional name
9 type ... name(optional) (since C++11)
```

这里，类型是以下类型之一——整型、枚举、指向对象或函数的指针、指向对象或函数的左值引用、指向成员对象或成员函数的指针，以及 `std::nullptr_t`(C++11)。此外，可以在模板声明中放入数组和/或函数类型，它们会被数据和/或函数指针自动替换。

下面的示例显示了一个使用非类型模板形参 `int N` 的类模板。`main()` 中实例化并创建了一个对象 `x`，因此 `x.a` 有五个初始值为 1 的元素。将第四个元素的值设置为 10 之后，输出结果：

```
1 //ch4_9_none_type_template_param1.cpp
2 #include <iostream>
3 template<int N>
4 class V {
5     public:
6     V(int init) {
7         for (int i = 0; i < N; ++i) { a[i] = init; }
8     }
9     int a[N];
10 };
11 int main()
12 {
```

```

13 V<5> x(1); //x.a is an array of 5 int, initialized as all 1's
14 x.a[4] = 10;
15 for( auto &e : x.a) {
16     std::cout << e << std::endl;
17 }
18 }
```

下面是使用 const char* 作为非类型模板形参的函数模板示例:

```

1 //ch4_10_none_type_template_param2.cpp
2 #include <iostream>
3 template<const char* msg>
4 void foo() {
5     std::cout << msg << std::endl;
6 }
7 // need to have external linkage
8 extern const char str1[] = "Test 1";
9 constexpr char str2[] = "Test 2";
10 extern const char* str3 = "Test 3";
11 int main()
12 {
13     foo<str1>(); //line 1
14     foo<str2>(); //line 2
15     //foo<str3>(); //line 3
16     const char str4[] = "Test 4";
17     constexpr char str5[] = "Test 5";
18     //foo<str4>(); //line 4
19     //foo<str5>(); //line 5
20     return 0;
21 }
```

main() 中成功地用 str1 和 str2 实例化了 foo(), 它们都是编译时常量值, 并且具有外部链接。如果取消注释第 3-5 行, 编译器将报告错误消息。得到这些编译错误的原因如下:

- Line 3: str3 不是 const 变量, 因此不能更改 str3 所指向的值。str3 的值可以修改。
- Line 4: str4 不是 const char* 类型的有效模板实参, 因为它没有链接。
- Line 5: str5 不是 const char* 类型的有效模板实参, 因为它没有链接。

非类型形参的另一种最常见用法是使用数组的大小。如果你想了解更多, 请访问
<https://stackoverflow.com/questions/33234979>

模板参数类型

type 模板形参的语法如下:

```

1 //A type Template Parameter (TP) with an optional name
2 typename | class name(optional)
3
4 //A type TP with an optional name and a default
5 typename[ class ] name(optional) = default
```

```
6 //A type TP pack with an optional name
7 typename[ class] ... name(optional) (since C++11)
8
```



注意: 这里, **typename** 和 **class** 关键字可以互换。模板声明体中, 类型参数的名称是 **typedef-name**。当模板实例化时, 为类型提供别名。

现在, 让我们来看一些例子:

- 没有默认值的类型模板参数:

```
1 Template<class T> //with name
2   class X { /* ... */ };
3
4 Template<class > //without name
5   class Y { /* ... */ };
6
```

- 默认的类型模板参数:

```
1 Template<class T = void> //with name
2   class X { /* ... */ };
3
4 Template<class = void > //without name
5   class Y { /* ... */ };
6
```

- 类型模板参数包:

```
1 template<typename... Ts> //with name
2   class X { /* ... */ };
3
4 template<typename... > //without name
5   class Y { /* ... */ };
6
```

模板形参包可以接受零个或多个模板形参, 并且只在 C++11 以后有效。

模板的模板参数

模板参数 template 的语法如下:

```
1 //A template template parameter with an optional name
2 template <parameter-list> class name(optional)
3
4 //A template template parameter with an optional name and a default
5 template <parameter-list> class name(optional) = default
6
7 //A template template parameter pack with an optional name
8 template <parameter-list> class ... name(optional) (since C++11)
```



TIP 注意: 模板参数的形参声明中, 只能使用 class 关键字, 不允许使用 typename。在模板声明的主体中, 形参的名称是模板名, 需要实参来实例化。

假设你有一个函数, 作为对象列表的流输出操作符:

```
1 template<typename T>
2     static inline std::ostream &operator << ( std::ostream &out ,
3         std::list<T> const& v )
4     {
5         /*...*/
6     }
```

前面的代码中可以看到, 对于序列容器 (如数组、双端队列和多种映射类型) 是相同的。因此, 使用 template 模板形参的概念, 可以使用操作符 << 来规则所有这些。例子可以在 exch4_tp_c.cpp 中找到:

```
1 //ch4_11_template_template_param.cpp (courtesy: https://stackoverflow.com/
2 questions/213761)
3 #include <iostream>
4 #include <vector>
5 #include <deque>
6 #include <list>
7
8 using namespace std;
9 template<class T, template<class , class ...> class X, class ... Args>
10 std::ostream& operator <<(std::ostream& os, const X<T, Args...>& objs) {
11     os << __PRETTY_FUNCTION__ << ":" << endl;
12     for (auto const& obj : objs)
13         os << obj << ' ';
14     return os;
15 }
16
17 int main() {
18     vector<float> x{ 3.14f, 4.2f, 7.9f, 8.08f };
19     cout << x << endl;
20     list<char> y{ 'E', 'F', 'G', 'H', 'I' };
21     cout << y << endl;
22     deque<int> z{ 10, 11, 303, 404 };
23     cout << z << endl;
24     return 0;
25 }
```

上述程序的输出如下所示:

```
1 class std::basic_ostream<char, struct std::char_traits<char> > &__cdecl
2 operator
3 <<<float , class std::vector , class std::allocator<float >>(class
4 std::basic_ostream
```

```

5 <char , struct std :: char_traits<char> > &,const class std :: vector<float , class
6 std :
7 : allocator<float> > &):
8 3.14 4.2 7.9 8.08
9 class std :: basic_ostream<char , struct std :: char_traits<char> > &__cdecl
10 operator
11 <<<char , class std :: list , class std :: allocator<char>>(class
12 std :: basic_ostream<cha
13 r , struct std :: char_traits<char> > &,const class std :: list <char , class
14 std :: alloca
15 tor<char> > &):
16 E F G H I
17 class std :: basic_ostream<char , struct std :: char_traits<char> > &__cdecl
18 operator
19 <<<int , class std :: deque , class std :: allocator<int>>(class
20 std :: basic_ostream<char
21 , struct std :: char_traits<char> > &,const class std :: deque<int , class
22 std :: allocat
23 or<int> > &):
24 10 11 303 404

```

如预期的那样，每次调用的输出的第一部分是模板函数名，而第二部分则输出每个容器的元素值。

模板参数

实例化一个模板，所有的模板形参必须用它们对应的模板形参替换。实参要么显式提供，要么从初始化式推导（对于类模板），要么从上下文推导（对于函数模板），或有默认值。由于有三种类型的模板形参，我们也将有三个相应的模板形参。它们是模板非类型实参、模板类型实参和模板模板实参。除此之外，我们还将讨论默认的模板实参。

模板非类型参数

回想一下，非类型模板形参引用的是编译时常量值，比如：整数、指针和对静态实体的引用。模板参数列表中提供的非类型模板参数必须与这些值匹配。通常，非类型模板实参用于类初始化或类容器的大小。

虽然，详细的规则为讨论每个类型（整型和算术类型，指针/函数/对象成员，左值引用参数等）的实参数超出了本书的范畴，一般规则模板实参数应该转化为模板参数常数表达式。

让我们来看看下面的例子：

```

1 //part 1: define template with non-type template parameters
2 template<const float* p> struct U {};//float pointer non-type parameter
3 template<const Y& b> struct V {};//L-value non-type parameter
4 template<void (*pf)(int)> struct W {};//function pointer parameter
5
6 //part 2: define other related stuff
7 void g(int ,float); //declare function g()
8 void g(int); //declare an overload function of g()

```

```

9 struct Y { //declare structure Y
10   float m1;
11   static float m2;
12 };
13 float a[10];
14 Y y; //line a: create a object of Y
15
16 //part 3: instantiation template with template non-type arguments
17 U<a> u1; //line b: ok: array to pointer conversion
18 U<&y> u2; //line c: error: address of Y
19 U<&y.m1> u3; //line d: error: address of non-static member
20 U<&y.m2> u4; //line e: ok: address of static member
21 V<y> v; //line f: ok: no conversion needed
22 W<&g> w; //line g: ok: overload resolution selects g(int)

```

前面第 1 部分的代码中，我们定义了三个具有不同非类型模板形参的模板结构。然后，第 2 部分中，声明了两个重载函数和结构体 Y。最后，第 3 部分中，讨论了通过不同的非类型参数来实例化它们的正确方法。

模板类型的参数

与模板的非类型实参相比，模板类型实参（用于类型模板形参）的规则很简单，必须是 typeid。在这里，typeid 是一个标准 C++ 操作符，会在运行时返回类型标识信息。它返回一个 type_info 对象，可以与其他 type_info 对象进行比较。

看看下面的例子：

```

1 //ch4_12_template_type_argument.cpp
2 #include <iostream>
3 #include <typeinfo>
4 using namespace std;
5
6 //part 1: define templates
7 template<class T> class C {};
8 template<class T> void f() { cout << "T" << endl; };
9 template<int i> void f() { cout << i << endl; };
10
11 //part 2: define structures
12 struct A{}; // incomplete type
13 typedef struct {} B; // type alias to an unnamed type
14
15 //part 3: main() to test
16 int main() {
17   cout << "Tid1=" << typeid(A).name() << ";" ;
18   cout << "Tid2=" << typeid(A*).name() << ";" ;
19   cout << "Tid3=" << typeid(B).name() << ";" ;
20   cout << "Tid4=" << typeid(int()).name() << endl;
21
22 C<A> x1; //line A: ok, 'A' names a type
23 C<A*> x2; //line B: ok, 'A*' names a type

```

```

24 C<B> x3; //line C: ok, 'B' names a type
25 f<int ()>(); //line D: ok, since int() is considered as a type,
26     //thus calls type template parameter f()
27 f<5>(); //line E: ok, this calls non-type template parameter f()
28 return 0;
29 }

```

本例(第1部分)中,定义了三个类模板和函数模板:带有类型模板形参的类模板C,两个带有类型模板形参的函数模板,以及一个非类型模板形参。第2部分中,有一个不完整的结构体A和一个未命名的结构体B。最后,第3部分中,我们测试了它们。4个 typeid() 在 Ubuntu 18.04 中的输出如下:

```

1 Tid1=A; Tid2=P1A; Tid3=1B; Tid4=FivE

```

x86 的 MSVC v19.24, 会输出以下内容:

```

1 Tid1=struct A; Tid2=struct A; Tid3=struct B; Tid4=int __cdecl(void)

```

另外,由于A、A*、B和int()都有 typeid, 所以从A行到D行的代码段与模板类型类或函数相链接。只有E行是从非类型模板形参函数模板实例化的, 即f()。

模板的模板参数

对于模板模板形参,其对应的模板实参是类模板的名称或模板别名。寻找与模板参数匹配的模板时,只考虑主类模板。

这里,主模板指的是特化的模板。即使它们的形参列表可能匹配,编译器也不会考虑使用模板形参的部分特化。

下面是一个模板参数的例子:

```

1 //ch4_13_template_template_argument.cpp
2 #include <iostream>
3 #include <typeinfo>
4 using namespace std;
5
6 //primary class template X with template type parameters
7 template<class T, class U>
8 class X {
9     public:
10    T a;
11    U b;
12 };
13
14 //partially specialization of class template X
15 template<class U>
16 class X<int, U> {
17     public:
18     int a; //customized a
19     U b;
20 };

```

```

21 //class template Y with template template parameter
22 template<template<class T, class U> class V>
23 class Y {
24     public:
25     V<int, char> i;
26     V<char, char> j;
27 };
28
29 Y<X> c;
30 int main() {
31     cout << typeid(c.i.a).name() << endl; //int
32     cout << typeid(c.i.b).name() << endl; //char
33     cout << typeid(c.j.a).name() << endl; //char
34     cout << typeid(c.j.b).name() << endl; //char
35
36     return 0;
37 }
```

本例中，定义了一个主类模板 X 及其特化，然后定义了类模板 Y，带有模板形参。接下来，我们用模板参数 X 隐式实例化 Y，并创建一个对象 c。最后，main() 输出四个 typeid() 的名称，结果分别是 int、char、char 和 char。

默认模板参数

C++ 中函数通过传递参数来调用，而参数由函数使用。如果在调用函数时没有传递参数，则使用默认值。与函数形参默认值类似，模板形参也可以有默认实参。定义模板时，可以设置它的默认实参，如下所示：

```

1 //ch4_14_default_template_arguments.cpp //line 0
2 #include <iostream> //line 1
3 #include <typeinfo> //line 2
4 template<class T1, class T2 = int> class X; //line 3
5 template<class T1 = float, class T2> class X; //line 4
6 template<class T1, class T2> class X { //line 5
7     public: //line 6
8     T1 a; //line 7
9     T2 b; //line 8
10 }; //line 9
11 using namespace std;
12 int main() {
13     X<int> x1; //<int,int>
14     X<float> x2; //<float,int>
15     X<> x3; //<float,int>
16     X<double, char> x4; //<double, char>
17     cout << typeid(x1.a).name() << ", " << typeid(x1.b).name() << endl;
18     cout << typeid(x2.a).name() << ", " << typeid(x2.b).name() << endl;
19     cout << typeid(x3.a).name() << ", " << typeid(x3.b).name() << endl;
20     cout << typeid(x4.a).name() << ", " << typeid(x4.b).name() << endl;
21     return 0
22 }
```

为模板形参设置默认实参时，需要遵循以下规则：

- 声明顺序很重要——默认模板实参的声明必须在主模板声明的顶部。例如，在前面的示例中，不能将第 3 行和第 4 行代码移到第 9 行之后。
- 如果一个形参有一个默认实参，那么它之后的所有形参也必须有默认实参。例如，以下代码是不正确的：

```
1 template<class U = char, class V, class W = int> class X { }; //Error
2 template<class V, class U = char, class W = int> class X { }; //OK
3
```

- 不能在同一个作用域中两次给出相同的形参默认实参。例如，如果使用下面的代码，会得到一个错误消息：

```
1 template<class T = int> class Y;
2
3 //compiling error, to fix it, replace "<class T = int>" by "<class T>"
4 template<class T = int> class Y {
5     public: T a;
6 };
7
```

这里，讨论了两个列表:template_parameter_list 和 template_argument_list。它们分别用于函数或类模板的创建和实例化。

还要了解了另外两条重要的规则：

- 定义模板类或函数时，需要给出它的 template_parameter_list:

```
1 template <template_parameter_list>
2 class X { ... }
3
4 template <template_parameter_list>
5 void foo( function_argument_list ) { ... } //assume return type is void
6
```

- 实例化时，必须提供相应的 argument_list:

```
1 class X<template_argument_list> x
2 void foo<template_argument_list>( function_argument_list )
3
```

这两个列表中的形参或实参类型可以分为三类，如下表所示。注意，虽然第一行是模板类，但这些属性也适用于模板函数：

	定义模板时 template <template_parameter_list> class X { ... }	实例化一个模板时 class X<template_argument_list> x
非类型	该参数列表中的实体可以是下列类型之一： <ul style="list-style-type: none">• 整型或枚举• 指向对象或函数的指针• 对象的左值引用或函数的左值引用• 成员指针• C++11 std::nullptr_t	<ul style="list-style-type: none">• 此列表中的非类型参数是表达式，其值可以在编译时确定。• 这些参数必须是常量表达式、具有外部链接的函数或对象的地址，或静态类成员的地址。• 非类型实参通常用于初始化类或指定类成员的大小。
类型	该参数列表中的实体可以是下列类型之一： <ul style="list-style-type: none">• 必须以 typename 或 class 开始• 在模板声明体中，类型的名称 参数是 typedef-name。当模板实例化时，它为提供的类型提供别名。	<ul style="list-style-type: none">• 参数的类型必须有一个 typeid• 不能是局部类型、没有链接的类型、未命名类型或由任何这些类型复合而成的类型。
模板	该参数列表中的实体可以是下列类型之一： <ul style="list-style-type: none">• template <parameter-list> class name• template <parameter-list> class ... name (可选) (C++11)	列表中的模板参数是类模板的名称

下一节中，我们将探索如何用 C++ 实现特征，并使用它们优化算法。

探索特征

泛型编程需要编写在特定需求下，适用于任何数据类型的代码。这是在软件工程行业中交付可重用高质量代码的最有效的方法。然而，在泛型编程中，有时泛型还不够好。当类型之间的差异过于复杂时，高效泛型很难通用化实现。例如，实现 sort 函数模板时，如果知道参数类型是链表而不是数组，那么将实现不同的策略来优化性能。

尽管模板特化是克服这个问题的方法，但并没有以广泛的方式提供与类型相关的信息。类型特征是一种用来收集类型信息的技术。我们可以做出更智能的决策，开发高质量的泛型编程优化算法。

本节中，我们将介绍如何实现类型特征，然后展示如何使用类型信息来优化算法。

类型特征实现

为了理解类型特征，来看看 boost::is_void 和 boost::is_pointer 的实现。

boost::is_void

首先，看一个最简单的 trait 类，is_void 特征是由 boost 创建的，定义了一个用于实现默认行为的通用模板，接受一个空类型，但其他类型都为空。因此，可得到 `is_void::value = false`:

```
1 //primary class template is_void
2 template< typename T >
3 struct is_void{
4     static const bool value = false; //default value=false
5 };
```

然后，我们对 void 类型进行完全特化:

```
1 //"<>" means a full specialization of template class is_void
2 template<>
3 struct is_void< void >{ //fully specialization for void
4     static const bool value = true; //only true for void type
5 };
```

这样，我们就有了一个完整的 traits 类型，可以通过检查以下表达式，来检测给定类型 T 是否为 void:

```
1 is_void<T>::value
```

接下来，了解下如何在 boost::is_pointer 中使用部分特化。

boost::is_pointer

与 boost::void 特征类似，主类模板的定义如下:

```
1 //primary class template is_pointer
2 template< typename T >
3 struct is_pointer{
4     static const bool value = false;
5 };
```

然后，部分特化所有指针类型:

```
1 //"typename T" in "<>" means partial specialization
2 template< typename T >
3 struct is_pointer< T* >{ //<T*> means partial specialization only for type
4     T*
5     static const bool value = true; //set value as true
6 };
```

现在，我们有了一个完整的特征类型，可以通过检查以下表达式来检测给定类型 T 是否指针:

```
1 is_pointer<T>::value
```

由于 boost 类型特征特性已经正式加入到 C++11 标准库中，我们可以在以下示例中展示 std::is_void 和 std::is_pointer 的用法，而不需要包含前面示例中的源代码:

```

1 //ch4_15_traits_boost.cpp
2 #include <iostream>
3 #include <type_traits> //since C++11
4 using namespace std;
5 struct X {};
6 int main()
7 {
8     cout << boolalpha; //set the boolalpha format flag for str stream.
9     cout << is_void<void>::value << endl; //true
10    cout << is_void<int>::value << endl; //false
11    cout << is_pointer<X *>::value << endl; //true
12    cout << is_pointer<X>::value << endl; //false
13    cout << is_pointer<X &>::value << endl; //false
14    cout << is_pointer<int *>::value << endl; //true
15    cout << is_pointer<int **>::value << endl; //true
16    cout << is_pointer<int [10]>::value << endl; //false
17    cout << is_pointer<nullptr_t>::value << endl; //false
18 }

```

代码在开头为字符串流设置了 `boolalpha` 格式标记。通过这样做，所有的 `bool` 值都通过它们的文本表示来提取，文本表示为真或假。然后，使用 `std::cout` 来打印 `is_void<t>::value` 和 `is_pointer<t>::value` 的值。每个值的输出显示在相应行的末尾注释中。

利用特征优化算法

我们将使用一个优化复制的示例，来展示类型特征的用法，而不是通过的抽象方式来讨论。使用标准库中的 `copy` 算法：

```

1 template<typename It1, typename It2>
2 It2 copy(It1 first, It1 last, It2 out);

```

显然，可以为任何类型的迭代器编写 `copy()` 的泛型版本，即这里的 `It1` 和 `It2`。然而，正如 `boost` 库作者所说的那样，某些情况下，`memcpy()` 可以执行复制操作。如果满足以下所有条件，就可以使用 `memcpy()`：

- `It1` 和 `It2` 这两种类型的迭代器都是指针。
- 除了 `const` 和 `volatile` 限定符之外，`It1` 和 `It2` 必须指向同一类型
- 由 `It1` 所指向的类型必须提供一个简单的赋值操作符

赋值操作符意味着该类型要么是标量类型，要么是以下类型之一：

- 该类型没有用户定义的赋值操作符
- 该类型没有数据成员的引用类型
- 赋值操作符必须在所有基类和数据成员对象中定义

标量类型包括算术类型、枚举类型、指针、成员指针或这些类型之一的 `const` 或 `volatile` 版本。

现在，让我们看看原始实现。它包括两个部分——复制器类模板和用户界面函数，即 `copy()`：

```

1 namespace detail{

```

```

2 //1. Declare primary class template with a static function template
3 template <bool b>
4 struct copier {
5     template<typename I1, typename I2>
6         static I2 do_copy(I1 first, I1 last, I2 out);
7 };
8 //2. Implementation of the static function template
9 template <bool b>
10 template<typename I1, typename I2>
11     I2 copier<b>::do_copy(I1 first, I1 last, I2 out) {
12     while(first != last) {
13         *out = *first;
14         ++out;
15         ++first;
16     }
17     return out;
18 };
19 //3. a full specialization of the primary function template
20 template <>
21 struct copier<true> {
22     template<typename I1, typename I2>
23         static I2* do_copy(I1* first, I1* last, I2* out) {
24             memcpy(out, first, (last-first)*sizeof(I2));
25             return out+(last-first);
26         }
27 };
28 } //end namespace detail

```

如注释行中所示，前面的复制器类模板有两个静态函数模板——一个是主函数模板，另一个是完全特化的。泛型类型的模板是逐个元素进行硬拷贝，而全特化对象通过 `memcpy()` 一次复制所有元素：

```

1 //copy() user interface
2 template<typename I1, typename I2>
3 inline I2 copy(I1 first, I1 last, I2 out) {
4     typedef typename boost::remove_cv
5         <typename std::iterator_traits<I1>::value_type>::type v1_t;
6     typedef typename boost::remove_cv
7         <typename std::iterator_traits<I2>::value_type>::type v2_t;
8     enum{ can_opt = boost::is_same<v1_t, v2_t>::value
9         && boost::is_pointer<I1>::value
10        && boost::is_pointer<I2>::value
11        && boost::has_trivial_assign<v1_t>::value
12    };
13 //if can_opt== true, using memcpy() to copy whole block by one
14 //call(optimized); otherwise, using assignment operator to
15 //do item-by-item copy
16     return detail::copier<can_opt>::do_copy(first, last, out);
17 }

```

为了优化复制操作，前面的用户界面函数定义了两个 `remove_cv` 模板对象:`v1_t` 和 `v2_t`，然后计算 `can_opt` 是否为真。之后，调用 `do_copy()` 模板函数。通过使用 boost 库 (`algo_opt_examples.cpp`) 中的测试代码，我们可以看到在使用优化的实现方面有了显著的改进，复制 `char` 或 `int` 类型的数据要快 8 到 3 倍。

最后，让我们总结一下：

- 除了类型之外，特征还提供了其他信息，并通过模板特化实现的
- 按照惯例，特征总是作为结构来实现的。用于实现特征的结构称为特征类
- Bjarne Stroustrup 说，我们应该把特征看作一个小对象，它的主要目的是携带另一个对象或算法，从而确定策略或实现细节的信息
- Scott Meyers 还总结说，我们应该使用特征类来收集信息
- 特征可以帮助我们以一种高效/优化的方式实现通用算法

接下来，我们将探索 C++ 中的模板元编程。

探索模板元编程

一种计算机程序能够把其他程序当作它们的数据来处理的编程技术，称为元编程。程序可以设计成读取、生成、分析或转换其他程序，甚至在运行时修改自己。元编程的一种是编译器，将文本格式程序作为输入语言 (C、Fortran、Java 等)，并以输出生成另一种二进制机器码格式。

C++ 模板元编程 (TMP) 是指在 C++ 中使用模板生成元程序。其有两个条件——必须定义一个模板，以及必须实例化一个已定义的模板。TMP 是图灵完备的，这意味着它有能力计算任何可计算的东西。另外，由于 TMP 中的变量都是不可变的 (变量是常量)，所以使用递归，而不是迭代来处理集合中的元素。

为什么需要 TMP？因为它可以加快我们的程序在执行时间！由于在优化中没有免费的午餐，我们为 TMP 付出的代价不再是编译时和/或更大的二进制代码大小。另外，并不是所有的问题都可以用 TMP 解决，只有当我们计算的是编译时不变的才会起作用，例如：找出所有小于一个常数整数的所有数、一个常数整数的阶乘、展开一个常数数量的循环或迭代等。

从实用的角度来看，模板元编程能够解决以下三类问题：编译时计算、编译时优化，以及静态多态性。下面的小节中，我们将提供来自每个类别的示例，以演示元编程的工作原理。

编译时计算

通常，如果任务的输入和输出在编译时是已知的，可以使用模板元编程在编译期间执行计算，从而节省运行时开销和内存占用。这在 CPU 利用率高的实时项目中非常有用。

我们看看阶乘函数，它计算 $n!$ 。这是所有小于等于 n 的正整数与 0 的乘积 $= 1$ 的定义。由于递归的概念，可以使用一个简单的函数来实现它，如下所示：

```
1 //ch4_17_factorial_recursion.cpp
2 #include <iostream>
3 uint32_t f1(const uint32_t n) {
4     return (n<=1) ? 1 : n * f1(n - 1);
5 }
6 constexpr uint32_t f2(const uint32_t n) {
7     return (n<=1) ? 1 : n * f2(n - 1);
```

```

8 }
9 int main() {
10    uint32_t a1 = f1(10); //run-time computation
11    uint32_t a2 = f2(10); //run-time computation
12    const uint32_t a3 = f2(10); //compile-time computation
13    std::cout << "a1=" << a1 << ", a2=" << a2 << std::endl;
14 }
```

f1() 在运行时执行计算，f2() 可以在运行时或编译时执行，这取决于它的使用。

类似地，通过使用带有非类型形参、特化和递归概念的模板，这个问题的模板元编程版本如下：

```

1 //ch4_18_factorial_metaprogramming.cpp
2 #include <iostream>
3 //define a primary template with non-type parameters
4 template <uint32_t n>
5 struct fact {
6     const static uint32_t value = n * fact<n - 1>::value;
7     //use next line if your compiler does not support declare and initialize
8     //a constant static int type member inside the class declaration
9     //enum { value = n * fact<n - 1>::value };
10 };
11
12 //fully specialized template for n as 0
13 template <>
14 struct fact<0> {
15     const static uint32_t value = 1;
16     //enum { value = 1 };
17 };
18 using namespace std;
19 int main() {
20     cout << "fact<0>=" << fact<0>::value << endl; //fact<0>=1
21     cout << "fact<10>=" << fact<10>::value << endl; //fact<10>=3628800
22     //Lab: uncomment the following two lines, build and run
23     // this program, what are you expecting?
24     //uint32_t m=5;
25     //std::cout << fact<m>::value << std::endl;
26 }
```

创建了一个带有非类型形参的类模板，并且像其他 const 表达式一样，const static uint32_t 或枚举常量的值在编译时计算。这个编译时求值约束意味着只有 const 变量才有意义。另外，因为我们只使用类，所以静态对象是有意义的。

当编译器看到模板的新参数时，会创建该模板的新实例。例如，当编译器看到 fact<10>::value 并试图创建一个实参为 10 的 fact 实例时，结果是 fact<9> 也必须创建。对于 fact<9>，它需要 fact<8>，以此类推。最后，编译器使用 fact<0>::value(即 1)，并且在编译期间递归终止。这个过程可以在下面的代码块中看到：

```

1 fact <10>::value = 10* fact <9>::value;
2 fact <10>::value = 10* 9 * fact <8>::value;
3 fact <10>::value = 10* 9 * 8 * fact <7>::value;
```

```

4 ...
5 fact <10>::value = 10* 9 * 8 *7*6*5*4*3*2*fact <1>::value;
6 fact <10>::value = 10* 9 * 8 *7*6*5*4*3*2*1*fact <0>::value;
7 ...
8 fact <10>::value = 10* 9 * 8 *7*6*5*4*3*2*1*1;

```

注意，为了能够以这种方式使用模板，必须在模板实参列表中提供一个常量实参。这就是为什么如果取消对最后两行代码的注释，编译器就会提示：

```

1 fact:template parameter n: m: a variable
2 with non-static storage duration cannot be used as a non-type argument

```

最后，我们通过简单比较 conexpr 函数 (CF) 和 TMP 来结束本小节：

- 计算时间:**CF 在编译时或运行时执行，这取决于它的使用，但 TMP 只在编译时执行。
- 参数列表:**CF 只能接受值，但是 TMP 可以同时接受值和类型参数。
- 控制结构:**CF 可以使用递归、条件和循环，但是 TMP 只使用递归。

编译时优化

虽然前面的示例可以在编译时计算常数整数的阶乘，但可以使用运行时循环展开两个 n 维向量的点积（其中 n 在编译时是已知的），n 维向量的好处是可循环展开的。

以传统的点积函数模板为例，可以采用以下方式实现：

```

1 //ch4_19_loop_unrolling_traditional.cpp
2 #include <iostream>
3 using namespace std;
4
5 template<typename T>
6 T dotp( int n, const T* a, const T* b)
7 {
8     T ret = 0;
9     for ( int i = 0; i < n; ++i) {
10         ret += a[ i ] * b[ i ];
11     }
12     return ret;
13 }
14
15 int main()
16 {
17     float a[5] = { 1, 2, 3, 4, 5 };
18     float b[5] = { 6, 7, 8, 9, 10 };
19     cout<<"dot_product(5,a,b)=" << dotp<float>(5, a, b) << '\n'; //130
20     cout<<"dot_product(5,a,a)=" << dotp<float>(5, a, a) << '\n'; //55
21 }

```

循环展开意味着，如果我们可以优化 dotp() 函数内部的 for 循环为 $[0]*b[0] + a[1]*b[1] + a[2]*b[2] + a[3]*b[3] + a[4]*b[4]$ ，将节省更多的运行时计算。这正是元编程在下面的代码块中所做的：

```

1 //ch4_20_loop_unroolling_metaprogramming.cpp
2 #include <iostream>
3
4 //primary template declaration
5 template <int N, typename T>
6 class dotp {
7     public:
8         static T result(T* a, T* b) {
9             return (*a) * (*b) + dotp<N - 1, T>::result(a + 1, b + 1);
10        }
11    };
12
13 //partial specialization for end condition
14 template <typename T>
15 class dotp<1, T> {
16     public:
17         static T result(T* a, T* b) {
18             return (*a) * (*b);
19         }
20    };
21
22 int main()
23 {
24     float a[5] = { 1, 2, 3, 4, 5 };
25     float b[5] = { 6, 7, 8, 9, 10 };
26     std::cout << "dot_product(5,a,b) = "
27     << dotp<5, float>::result( a, b ) << '\n'; //130
28     std::cout << "dot_product(5,a,a) = "
29     << dotp<5,float>::result( a, a ) << '\n'; //55
30 }

```

与阶乘元编程示例类似，在 `dotp<5, float>::result(a, b)` 中，实例化进程递归地执行以下计算：

```

1 dotp<5, float>::result( a, b )
2 = *a * *b + dotp<4,float>::result(a+1,b+1)
3 = *a * *b + *(a+1) * *(b+1) + dotp<3,float>::result(a+2,b+2)
4 = *a * *b + *(a+1) * *(b+1) + *(a+2) * *(b+2)
5     + dotp<2,float>::result(a+3,b+3)
6 = *a * *b + *(a+1) * *(b+1) + *(a+2) * *(b+2) + *(a+3) * *(b+3)
7     + dotp<1,float>::result(a+4,b+4)
8 = *a * *b + *(a+1) * *(b+1) + *(a+2) * *(b+2) + *(a+3) * *(b+3)
9     + *(a+4) * *(b+4)

```

当 N 是 5，就会递归地调用 `dotp< N, float>::results()` 模板函数四次，直到 `dotp<1, float>::results()`。代码块的最后两行，是由 `dotp<5, float>::result(a, b)` 计算的最终表达式。

静态多态性

多态意味着多个函数具有相同的名称。动态多态性允许用户决定在运行时执行实际的函数方法，而静态多态性意味着，实际的函数在编译时是已知的。默认情况下，C++ 在编译时通过检查函数类型和/或实参数量来匹配函数调用和正确的函数定义。这个过程称为静态绑定或重载。但是，通过使用虚函数，编译器也在运行时进行动态绑定或覆盖。

例如，下面的代码中，基类 B 和派生类 D 中都定义了一个虚函数 alg()。我们使用派生对象指针 p 作为基类的实例指针时，p->alg() 函数调用将调用派生类中定义的派生 alg():

```
1 //ch4_21_polymorphism_traditional.cpp
2 #include <iostream>
3
4 class B{
5     public:
6     B() = default;
7     virtual void alg() {
8         std::cout << "alg() in B";
9     }
10};
11
12 class D : public B{
13     public:
14     D() = default;
15     virtual void alg(){
16         std::cout << "alg() in D";
17     }
18};
19
20 int main()
21{
22     //derived object pointer p as an instance pointer of the base class
23     B *p = new D();
24     p->alg(); //outputs "alg() in D"
25     delete p;
26     return 0;
27}
```

但多态性行为是不变的，并且可以在编译时确定的情况下，使用奇怪重复的模板模式 (CRTP) 来实现静态多态性，模仿静态多态性并在编译时解析绑定。这样，程序就不用在运行时检查虚拟查找表了。下面的代码以静态多态性的方式实现了前面的例子：

```
1 //ch4_22_polymorphism_metaprogramming.cpp
2 #include <iostream>
3
4 template <class D> struct B {
5     void ui() {
6         static_cast<D*>(this)->alg();
7     }
8};
```

```

9
10 struct D : B<D> {
11     void alg() {
12         cout << "D::alg()" << endl;
13     }
14 };
15
16 int main() {
17     B<D> b;
18     b.ui();
19     return 0;
20 }
```

总之，模板元编程的一般思想是让编译器在编译时做一些计算。通过这种方式，可以在一定程度上解析运行时开销。之所以能在编译期间计算一些东西，是因为有些东西在运行前是常量。

C++TMP 是在编译时执行计算任务的一种非常强大的方法。第一种方法必须非常小心地处理编译错误，因为模板树是展开的。从实用的角度来看，boost 元编程库 (MPL) 是一个很好的入门参考，以通用的方式为算法、序列和元函数提供了编译时 TMP 框架。此外，C++17 中新的 std::variant 和 std::visit 特性，可以用于没有相关类型共享继承接口的静态多态性。

总结

本章中，讨论了 C++ 中泛型编程相关的主题。从回顾 C 宏和函数重载开始，介绍了 C++ 模板开发的动机。然后，给出了带有固定数量参数的类和函数模板的语法，以及特化和实例化。自 C++11 以来，可变参数模板被标准泛型函数模板和类模板所接受。在此基础上，进一步将模板形参和实参分为三类：非类型模板形参/实参、类型模板形参/实参和模板模板形参/实参。

还了解了特征和模板元编程。作为模板特化的副产品，特征类可以为提供更多关于类型的信息。在类型信息的帮助下，最终让实现泛型算法的优化成为可能。类和/或函数模板的另一个应用是在编译时通过递归计算一些常量任务，称为模板元编程。有能力执行编译时计算和/或优化，以及避免在运行时查找虚拟表。

现在，对模板有了深刻的理解，应该能够在应用程序中创建自己的函数和类模板，并实践使用特征来优化算法，并使用模板元编程来执行编译时计算，以获得额外的优化。

下一章中，我们将了解与内存和管理相关的主题，例如：内存访问的概念、内存分配和回收技术，以及垃圾收集基础知识。这是 C++ 的特性，因此每个 C++ 开发者都必须理解它。

问题

1. 宏的副作用是什么？
2. 什么是类/函数模板？什么是模板类/函数？
3. 什么是模板参数列表？什么是模板参数列表？有了类模板，就可以显式或隐式地实例化它。在什么样的场景中需要显式实例化？
4. 多态性在 C++ 中意味着什么？函数重载和函数重写之间的区别是什么？
5. 什么是类型特征？我们如何实现类型特征？

6. 在 ch4_5_class_template_implicit_inst_B.cpp 文件中，我们讨论过隐式实例化生成 $X<\text{int}>$ 类，然后创建 xi 对象并生成 $X<\text{int}>::\text{f}()$ 函数，而不是 $X<\text{int}>::\text{g}()$ 。如何验证 $X<\text{int}>::\text{g}()$ 没有生成？
7. 使用模板元编程，求解 $f(x, n) = x^n$ 的问题，其中 n 是 7。const 和 x 是变量。
8. 将 ch4_17_loop_unrolling_metaprogramming.cpp 扩展为 8。 $n = 10 \ 100 \ 10^3 \ 10^4 \ 10^6 \dots$ 直到系统内存限制为止。比较编译时间、对象文件大小和 CPU 运行时间。

扩展阅读

作为本章的引用，请查看以下资料，以了解本章相关的更多内容：

- Milner, R., Morris, L., Newey, M. (1975). A Logic for Computable Functions with Reflexive and Polymorphic Types. Proceedings of the Conference on Proving and Improving Programs. [https://www.research.ed.ac.uk/portal/en/publications/a-logic-for-computable-functions-with-reflexive-and-polymorphic-types\(9a69331e-b562-4061-8882-2a89a3c473bb\).html](https://www.research.ed.ac.uk/portal/en/publications/a-logic-for-computable-functions-with-reflexive-and-polymorphic-types(9a69331e-b562-4061-8882-2a89a3c473bb).html)
- Curtis, Dorothy (2009-11-06). CLU home page. Programming Methodology Group, Computer Science and Artificial Intelligence Laboratory. Massachusetts Institute of Technology. <http://www.pmg.csail.mit.edu/CLU.html>
- Technical Corrigendum for Ada 2012, published by ISO. Ada Resource Association. 2016-01-29. <https://www.adaic.org/2016/01/technical-corrigendum-for-ada-2012-published-by-iso/>
- B. Stroustrup, C++. <https://dl.acm.org/doi/10.5555/1074100.1074189>
- S. Meyers, Effective C++ 55 Specific Ways to Improve Your Programs and Designs (3rd Edition), Chapter 7. <https://www.oreilly.com/library/view/effective-c-55/0321334876/>
- D. Gregor and J. Järvi (February 2008). Variadic Templates for C++0x. Journal of Object Technology. pp. 31–51 http://www.jot.fm/issues/issue_2008_02/article2.pdf
- <https://www.boost.org/> for type traits, unit testing etc.
- https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_72/rzarg/templates.htm for generic templates discussions.
- <https://stackoverflow.com/questions/546669/c-code-analysis-tool> for code analysis tools.
- <https://en.cppreference.com> for template explicit instantiations.
- <http://www.cplusplus.com> for library references and usage examples.
- <http://www.drdobbs.com/cpp/c-type-trait/184404270> for type-trait.
- <https://accu.org/index.php/journals/424> for template metaprogramming.
- https://en.wikipedia.org/wiki/Template_metaprogramming for template metaprogramming.
- K. Czarnecki, U. W. Eisenecker, Generative Programming: Methods, Tools, and Applications, Chapter 10.
- N. Josuttis; D. Gregor and D. Vandevoorde, C++ Templates: The Complete Guide (2nd Edition), Addison-Wesley Professional 2017.

第 5 章：内存管理和智能指针

C++ 中内存管理是要付出代价的。开发者经常抱怨 C++ 需要手动进行内存管理。像 C# 和 Java 这样的语言可以自动内存管理，但程序运行得要比 C++ 程序慢，而手动内存管理通常容易出错且不安全。正如前几章了解到的，程序是数据和指令的集合。几乎每个程序都在某种程度上使用计算机内存，没有一个程序不需要内存分配。

内存分配和回收从最简单的函数调用开始。调用函数通常意味着向它传递参数，函数需要空间来存储这些参数。当代码中声明对象时，会进行自动分配，它们的生存期取决于作用域。只要超出了作用域，就会自动回收它们。大多数编程语言都为动态内存提供了类似的自动回收功能。动态分配内存——与自动分配相反——是开发者根据需要请求新内存的代码部分。例如，当客户数量增加时，将用于存储客户对新内存空间的请求列表到程序中。为了区分不同类型的内存管理，不管是自动的还是手动，开发者使用的都是内存分段。一个程序使用几个内存段、堆栈、堆、只读段等进行操作，尽管它们都具有相同的结构，并且都是虚拟内存的一部分。

大多数语言都提供了访问动态内存的方法，而不关心回收策略，将困难的工作留给运行时环境，C++ 开发者必须处理内存管理的底层细节。时至今日，C++ 还是没有提供高级的内存管理功能。因此，对内存结构及其管理的深刻理解是每个 C++ 开发者必须的。这一章中阐明内存背后的奥秘，以及正确的内存管理技术。

本章中，我们将了解以下内容：

- 什么是内存，C++ 中如何访问？
- 详细的内存分配
- 内存管理技术和习惯用法
- 垃圾收集的基础

编译器要求

g++ 编译器需要添加编译选项 `-std=c++2a` 来编译本章的代码。可以从这里获取本章的源码文件：<https://github.com/PacktPublishing/Expert-CPP>

理解计算机内存

底层上理解，存储器是一种存储位的状态的设备。假设我们正在发明一种可以存储单个比特信息的设备，发明很久以前就已经发明了的东西是没有意义的。但它的神奇之处在于，现在的开发者拥有稳定的多功能环境，提供了大量的库、框架和工具来创建程序，甚至不需要理解它们。声明一个变量或分配一个动态内存变得非常容易，如下面的代码片段所示：

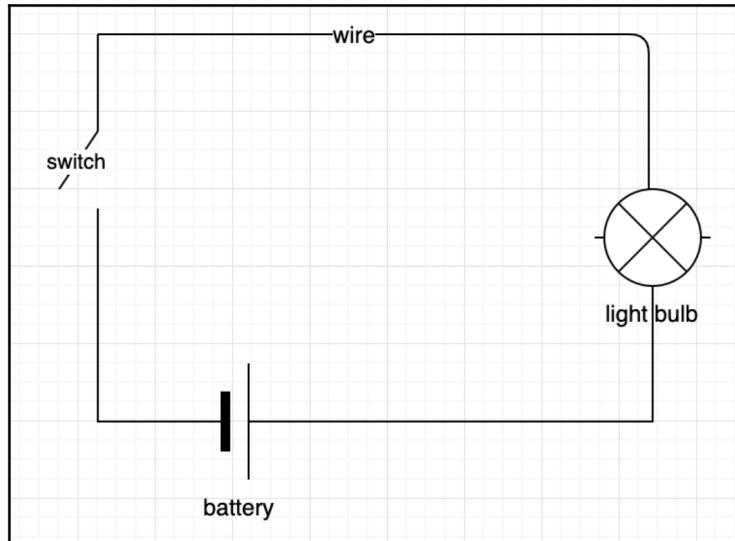
```
1 int var;  
2 double* pd = new double(4.2);
```

很难描述设备如何存储这些变量。为了解释这个过程，让我们试着设计一个可以存储信息的设备。

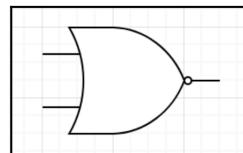
内存存储的设计

我们使用电路、继电器和逻辑门来设计能够进行存储的简单设备。本节的目的是了解内存底层的结构。

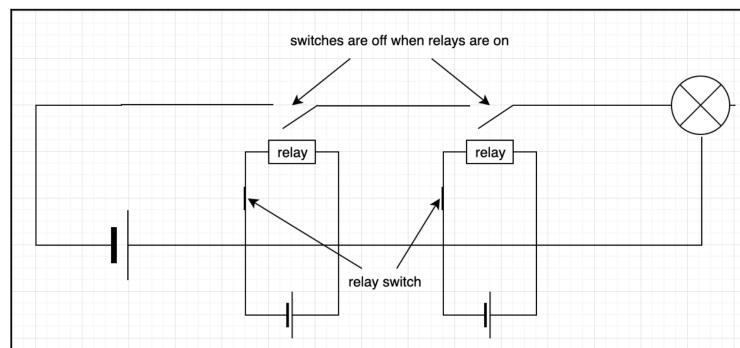
这是一个简单的电路图，你们可能在物理课上见过：



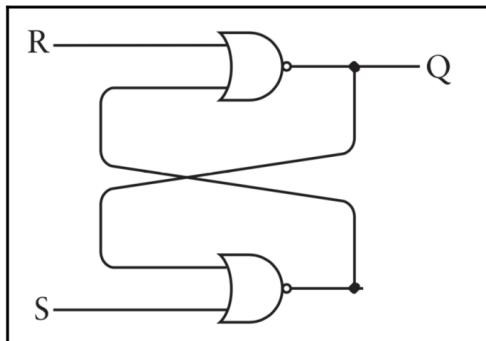
它由一根连接电池和灯泡的电线组成。这根电线上有一个开关，控制灯泡的状态。当开关闭合时，灯泡是亮的，否则是关的。我们将在这个电路中加入两个逻辑非元件，NOR 是 Not OR 的缩写。它通常是这样表示的：



它有两个输入（连接元件的导线），每一个都代表一个电子信号。我们说，如果两个输入都是 0，那么输出（从元素出来的导线）是 1，这就是为什么我们称它为 Not。如果有输入为 1，那么 OR 元素的输出是 1。前面的 NOR 元素简单地使用两个继电器构造。继电器是一种使用电磁铁来闭合和打开触点的开关。请看下图：



当两个继电器的开关都合上时（即继电器正在工作，并拉下电路的开关），灯泡就灭了。当我们把开关移动到两个继电器的开位时，灯泡就亮了。上图是描述 NOR 门的一种方法。现在，我们可以使用电线、灯泡、电池和继电器创建逻辑元素。现在让我们来看看两个 NOR 元素的奇怪组合会导致一个有趣的发现：



上图是 R-S 触发器的经典表示。R 代表 reset, S 代表 set。按照上述方案构建的设备只能存储 1 位。输出 Q 是读取设备内容的连接线。如果我们将触发器设置为存储该位，则输出为 1。仔细检查这个图，想象一个个地传递信号给它的输入端，或者同时将信号传递给两个输入端，并看到 Q 处的输出。当输入 S 为 1 时，Q 变为 1。当 R = 1 Q = 0。通过这种方式，我们可以设置或重置位。只要给设备供电，它就会储存数据。

想象一下，有很多早期设计的设备连接在一起，就可以存储不止很多信息。通过这种方式，我们可以构造复杂的存储设备，来存储字节甚至千字节的数据。

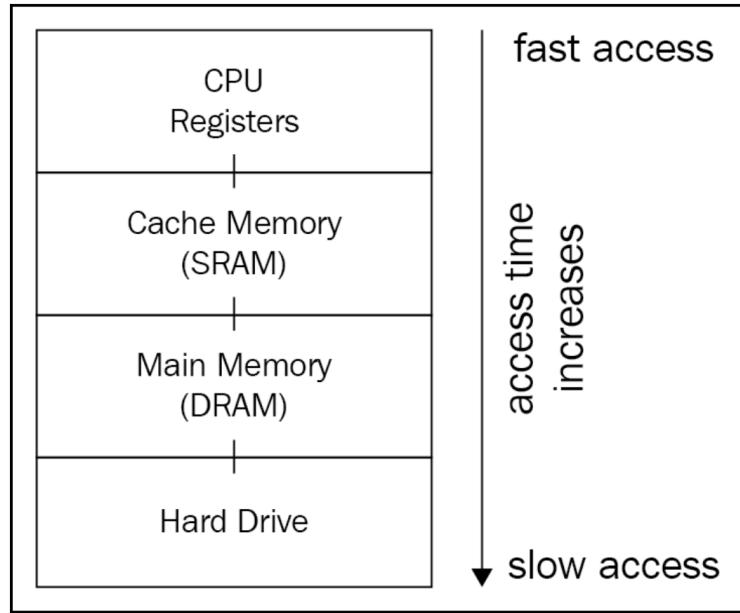
前面的设备与晶体管发明之前的计算机中使用的设备相似。晶体管是一种存储量小得多的设备。现代设备不使用继电器，而是结合了数百万个晶体管来存储和操作数据。中央处理单元 (CPU) 寄存器是一种利用晶体管来存储特定数量位元的设备。通常，一个通用寄存器最多存储 64 位数据。不仅能使用寄存器来存储所有的程序和数据，现代计算机的内存结构要复杂得多。现在，让我们从更的角度来研究计算机内存的层次结构。

从更高层次的角度理解计算机内存

编写专业程序时，了解计算机内存和数据存储的细节至关重要。当开发者提到内存时，大多数指的是虚拟内存。虚拟内存是操作系统 (OS) 支持的一种抽象内存，它控制并为进程提供内存空间。每个进程都有它的地址空间表示为几个段的集合。第 2 章中讨论了内存段的种类，以及给定程序如何使用它们。从开发者的角度来看，访问内存空间主要限于对象声明和使用，无论是堆栈、堆还是静态内存上声明对象，都访问相同的内存抽象——虚拟内存，直接使用物理内存比较困难。作为开发者至少应该知道有哪些内存存储单元，以及如何利用这些知识来编写更好的代码。

本节中，讨论了内存层次物理结构，我们称它为层次结构，因为在较低层次的每个内存单元提供更快的访问，但空间小。每一个连续的更高级别的内存提供了更多的空间，但访问速度更慢。讨论物理内存层次结构，有助于我们设计更好的代码。

了解内存各个层次上的工作原理，有助于开发者更好地组织数据操作。下面的图表说明了内存的层次结构：



寄存器是放置在中央处理器中的可访问存储器单元。寄存器的数量有限，所以不能把所有的程序数据都保存在里面。另一方面，动态 RAM (DRAM) 能够为程序存储数据。由于 DRAM 的物理结构和与 CPU 之间的距离，需要更长的时间来访问数据。CPU 通过数据总线访问 DRAM，数据总线是一组在 CPU 和 DRAM 之间传输数据的线。为了通知 DRAM 控制器它将读或写数据，CPU 使用控制总线。我们将 DRAM 称为主存，来看看内存层次结构。

寄存器

寄存器保存固定数量的数据。CPU 字大小通常由寄存器的最大长度定义，例如：8 个字节或 4 个字节。C++ 程序不能直接访问寄存器。



C++ 支持使用 `asm` 声明嵌入汇编代码，例如，`asm("mov edx, 4")`。属于特定于平台的代码，所以不建议使用。

在旧版本中，可以在声明变量时使用 `register` 关键字：

```
1 register int num = 14;
```

修饰符指定编译器将变量存储在寄存器中。这样，它给开发者一种代码被优化的假象。



TIP 编译器是将高级 C++ 代码转换为机器代码的工具。翻译过程中，代码需要进行多次转换，包括代码优化。当开发者使用技巧使编译器优化部分代码时，编译器会把它们当作建议而不是命令。

例如，在循环中访问一个变量，如果该变量放在寄存器中比放在 DRAM 中要快。例如，下面的循环访问对象一百万次：

```
1 auto number{42};
2 for (int ix = 0; ix < 10000000; ++ix) {
```

```
3 int res{number + ix};  
4 // do something with res  
5 }
```

这个数字有一个自动的存储时间（与 `auto` 关键字无关），并放置在堆栈上。堆栈是虚拟内存中的一段，而虚拟内存是物理 DRAM 的抽象。寄存器中访问对象要比在 DRAM 中快得多，假设从 DRAM 中读取 `number` 的值比从寄存器中读取要慢 5 倍。使用 `register` 关键字优化前面循环的结果是显而易见的，如下所示：

```
1 register auto number{42};  
2 // the loop omitted for code brevity
```

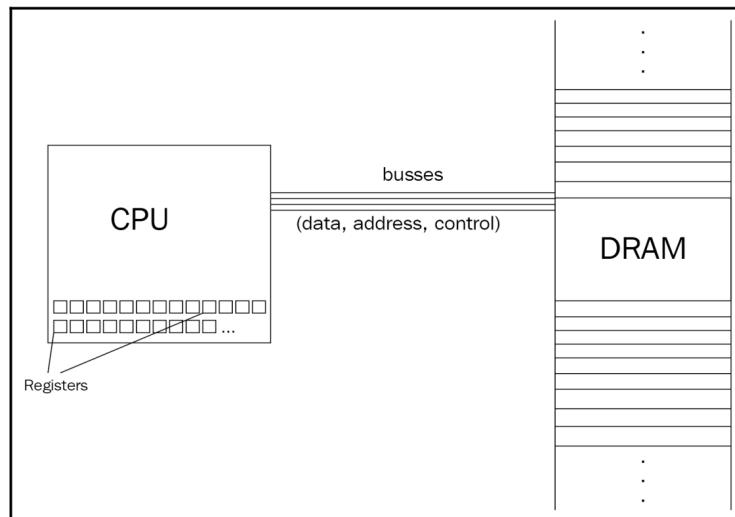
然而，现在的编译器可以进行更好的优化，因此随着时间的推移，对修饰符的需求已经消失，现在它已经是弃用的语言特性了。更好的优化方法是完全去掉 `number` 对象。

例如，下面的代码代表了编译优化版本，它使用实际的值，而不是通过驻留在 DRAM 中的变量来访问它：

```
1 for (int ix = 0; ix < 1000000; ++ix) {  
2     int res{42 + ix};  
3     // do something with res  
4 }
```

虽然前面的例子很简单，但是应该考虑编译过程中发生的编译器优化。

寄存器可以提高我们对程序执行细节的理解，CPU 执行的所有操作都通过寄存器进行，包括 CPU 解码和执行的指令，都是使用特定的寄存器访问，通常称指定执行指令的寄存器为指令指针。当运行程序时，CPU 访问它的指令，解码并执行。从主存读取数据和向内存写入数据，通过从寄存器复制数据和向寄存器复制数据来完成。通常，当 CPU 对数据执行操作时，通用寄存器用来保存数据。下面的图表描述了 CPU 的抽象视图，以及通过总线与主存储器的交互：



CPU 和 DRAM 之间的通信通过各种总线进行。第 2 章中用 C++ 进行底层编程时，我们讨论了 C++ 程序的底层表示——应该会看一下，以便更好地理解下面的例子。

现在，让我们看看实际的寄存器。下面的 C++ 代码声明了两个变量，并将它们的和存储在第三个变量中：

```
1 int a{40}, b{2};  
2 int c{a + b};
```

为了执行 sum 指令，CPU 将变量 a 和 b 的值移动到寄存器中。在计算和之后，将结果移动到另一个寄存器。程序的汇编伪代码表示形式类似于下面的代码：

```
1 mov eax, a  
2 mov ebx, b  
3 add eax, ebx
```

编译器并不是必须生成将每个变量映射到寄存器的代码——因为寄存器的数量是有限的。只需要记住，将经常访问的变量保持在足够小的大小，以便能够将其放入其中一个寄存器。对于更大的对象，缓存、内存会起到帮助作用。

缓存

缓存的概念在编程和计算机系统中很常见。加载在浏览器中的图像可以缓存，以避免将来用户再次访问该网站时向 Web 服务器请求下载它。缓存使程序运行得更快，这个概念可以以多种形式使用，包括单个功能。例如，下面的递归函数计算阶乘：

```
1 long factorial(long n) {  
2     if (n <= 1) { return 1; }  
3     return n * factorial(n - 1);  
4 }
```

这个函数不记得它以前计算的值，所以下面的调用会导致 5 次和 6 次递归调用：

```
1 factorial(5); // calls factorial(4), which calls factorial(3), and so on  
2 factorial(6); // calls factorial(5), which calls factorial(4), and so on
```

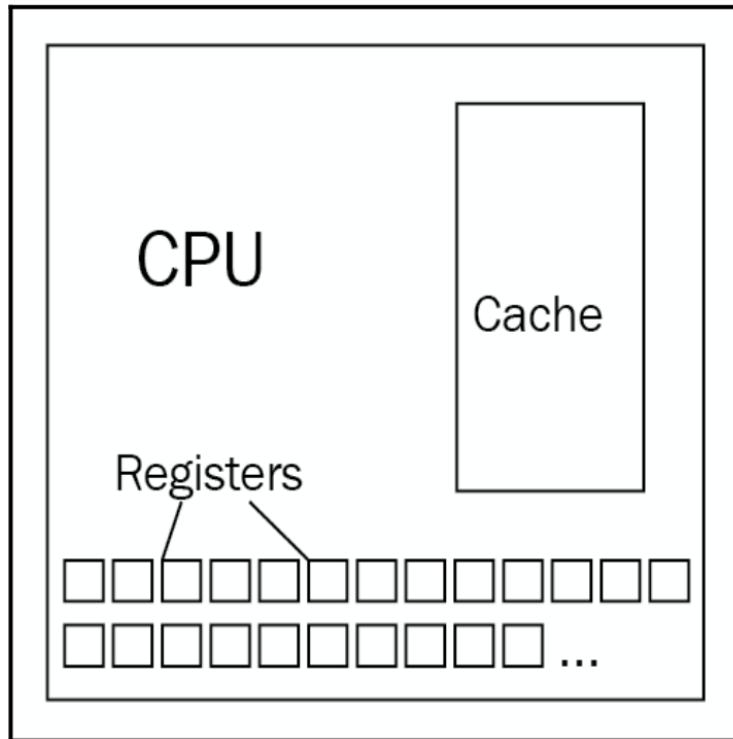
可以将已经计算好的值存储在一个全局可访问的变量中，如下所示：

```
1 std::unordered_map<long, long> cache;  
2  
3 long factorial(long n) {  
4     if (n <= 1) return 1;  
5     if (cache.contains(n)) return cache[n];  
6     cache[n] = n * factorial(n - 1);  
7     return cache[n];  
8 }
```

这些修改优化了对该函数的进一步调用：

```
1 factorial(4);  
2 // the next line calls factorial(4), stores the result in cache[5], which  
3 then calls factorial(3)  
4 // and stores the result in cache[4] and so on  
5 factorial(5);  
6 factorial(6); // calls the factorial(5) which returns already calculated  
7 value in cache[5]
```

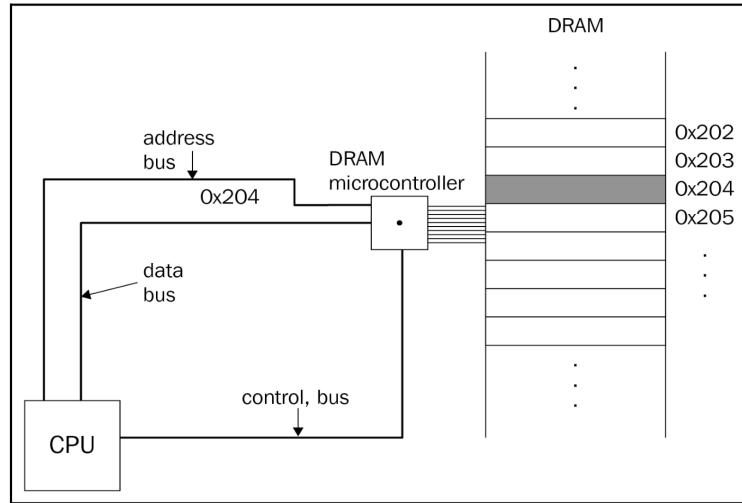
缓存的概念使阶乘函数运行得更快，与此相同的是，名为 cache 的实际内存设备被放置在 CPU 中。该设备存储最近访问的数据，以便进一步更快地访问该数据。下面的图表描述了 CPU 内部的寄存器和缓存内存：



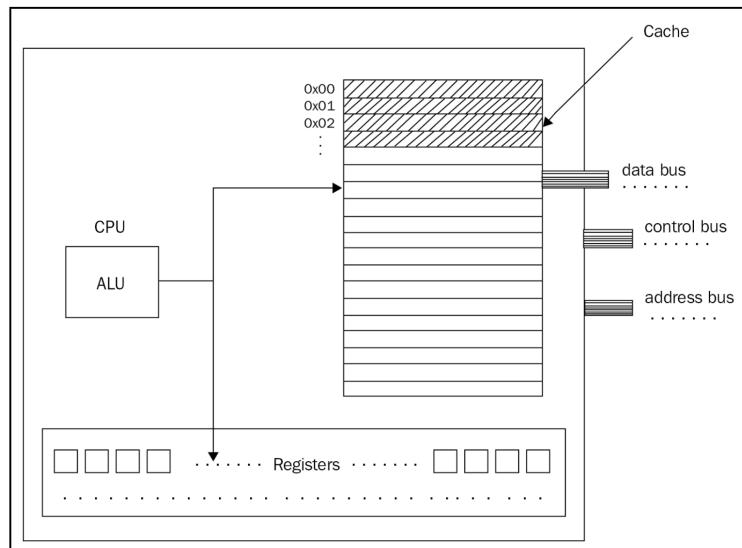
缓存大小通常在 2KB 到 64KB 之间（很少是 128KB）。虽然对于像 Photoshop 这样的应用程序来说不够大，因为图像数据的大小可能比缓存本身大得多，但在很多情况下，确实有帮助。例如，假设我们在一个向量中存储了 1000 多个数字：

```
1 std :: vector<int> vec;
2 vec.push_back(1);
3 ...
4 vec.push_back(9999);
5 for (auto it: vec) {
6     std :: cout << it;
7 }
8 // 1
9 // 2
10 // 3
11 // ...
12 // 9999
```

假设要打印该项，CPU 将其从内存复制到 rax 寄存器，然后调用操作符 <<，该操作符将 rax 的值打印到屏幕上。每次迭代循环中，CPU 将 vector 的下一项复制到 rax 寄存器中，并调用该函数打印其值。每个拷贝操作都要求 CPU 在地址总线上放置项目的地址，并将控制总线设置为读模式。DRAM 微控制器通过地址总线接收到的地址访问数据，并将其值复制到数据总线，从而将数据发送给 CPU。CPU 将值定向到 rax 寄存器，然后执行指令打印它的值。下面的图表显示了 CPU 和 DRAM 之间的这种交互：



为了优化循环，CPU 保持了数据本地化的思想，将整个向量复制到缓存中，并从缓存中访问向量，省略了对 DRAM 不必要的请求。下面的图表中，你可以看到通过数据总线从 DRAM 接收到的数据，然后存储在高速缓存存储器中：



位于 CPU 中的高速缓存称为一级 (L1) 高速缓存，它的容量最小，位于 CPU 内部。许多架构都有 2 级 (L2) 缓存，它位于 CPU 之外 (虽然比主存更近)，并且以与 DRAM 相同的方式访问。L2 高速缓存和 DRAM 之间的区别是物理结构和数据访问模式。L2 高速缓存代表静态 RAM(SRAM)，它比 DRAM 快，但价格更高。



一些运行时环境在实现垃圾收集时，会利用缓存的思想。它们根据对象的生存期将对象划分为不同的类别，其中生存期最小的对象（如在代码的局部范围内分配的对象）放置在缓存中，以便更快地访问和释放。

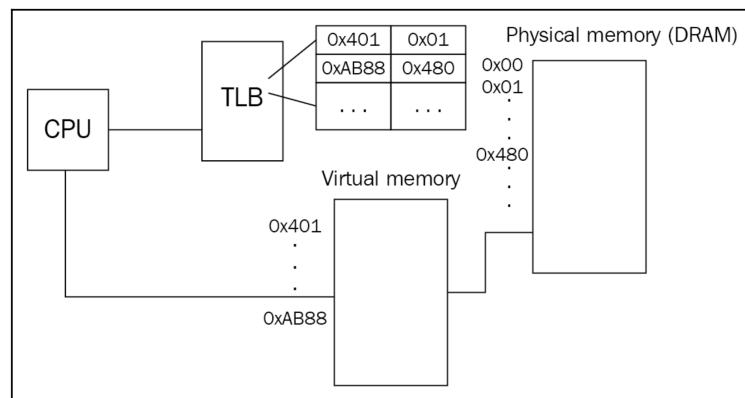
新的缓存内存级别充当较低级别的缓存。例如，L2 缓存作为 L1 缓存的缓存内存。当 CPU 遇到缓存缺失时，它请求 L2 缓存，以此类推。

主存

DRAM 的物理结构迫使它以刷新电荷以保持数据稳定，而 SRAM 不需要像 DRAM 那样刷新。我们称 DRAM 为主存主要是因为程序被装入其中，操作系统维护虚拟内存可以映射到 DRAM 中。所有实际的工作首先通过主存进行。

主内存表示可寻址的数据字节序列。每个字节都有唯一地址，并使用该地址访问。前面提到过 CPU 如何在地址总线上放置数据的地址，从而让 DRAM 控制器获取请求的数据并通过数据总线发送它。

操作系统引入虚拟内存作为物理内存的抽象，将虚拟内存的内容映射到物理内存，这涉及到 CPU 的译后备用缓冲区 (TLB)。TLB 是另一种形式的缓存内存：它存储虚拟内存到物理内存的最近转换，因此缓存它以备将来的请求。如下图所示，CPU 与 TLB 协调，以便将虚拟地址转换为正确地物理地址：



尽管内存管理很复杂，但操作系统为我们提供了一个足够简单的抽象来管理程序所需的内存。我们可以使用堆栈自动分配它，也可以在堆上动态分配。自动内存分配实际上不涉及很多问题和困难，我们只是声明对象，然后将它们放在堆栈上，然后在执行离开作用域时自动删除。动态内存的情况下（不要与前面提到的硬件 DRAM 混淆），分配和回收都应该手动完成，这可能会导致导致内存泄漏。

固定存储器

当我们关闭计算机时，主存储器的内容将被删除（因为电荷不再刷新）。为了在断电的情况下永久存储数据，计算机配备了硬盘驱动器 (HDD) 或固态驱动器 (SSD)。从开发者的角度来看，永久存储是用来存储程序所需的数据。为了运行一个程序，它应该加载到主存储器——从硬盘复制到 DRAM。操作系统使用加载程序处理它，并在内存中创建一个程序映像，称为进程。当程序完成或用户关闭它时，操作系统将进程的地址范围标记为可自由使用。

我们假设在学习 C++ 的时候使用一个文本编辑器来写笔记。输入到编辑器中的文本驻留在主存中，除非我们将其保存在硬盘上。注意这一点很重要，因为大多数程序都会跟踪用户最近的活动，并且允许用户修改程序设置。为了保持这些设置，即使在程序重新启动后，用户修改他们的方式，程序将他们作为一个单独文件存储在硬盘上。程序下次运行时，首先从硬盘读取相应的设置文件，并更新自己以应用最近的设置。

通常，永久存储比主存有更大的容量，这使得可以使用硬盘驱动器作为虚拟内存的备份。操作系统可以维持虚拟内存并伪造其大小，使其比物理 DRAM 更大。例如，通过启动几个重量级应用程序，DRAM 的最大容量可能会很快耗尽。然而，操作系统仍然可以通过备份硬盘上的额外空间来维持更大的虚拟内存。当用户在应用程序之间切换时，操作系统将超出的虚拟内存字节复制到硬盘，并将当前运行的应用程序映射到物理内存。

这使得程序和操作系统运行得更慢，但允许我们在不考虑主存有限大小的情况下保持它们的打开状态。现在让我们更深入地研究一下 C++ 中的内存管理。

内存管理的基础知识

大多数情况下，当开发者忘记重新分配内存空间时，内存管理中出现的问题就会发生。这将导致内存泄漏。内存泄漏在程序中都是普遍存在的问题。当程序为其数据请求一个新的内存空间时，操作系统将提供的空间标记为 busy，该程序的其他指令或任何其他程序都不能请求那么多内存空间。理想情况下，当程序的部分使用完内存空间后，必须通知操作系统删除 busy 标签，以使该空间可供其他程序使用。有些语言提供了对动态分配内存的自动控制，这样就能开发者专注于程序的逻辑，而不是关注内存资源的回收。然而，C++ 假设开发者是负责的（通常不是这样），动态分配的内存管理是开发者的职责。这就是为什么该语言同时提供了 new 和 delete 操作符来处理内存空间，其中 new 操作符分配内存空间，delete 操作符释放内存空间。换句话说，处理动态分配内存的理想代码如下所示：

```
1 T* p = new T(); // allocate memory space
2 p->do_something(); // use the space to do something useful
3 delete p; // deallocate memory space
```

忘记调用 delete 操作符会使分配的内存空间永远处于 busy 状态。现在想象一个始终在用户计算机上打开的 Web 浏览器。随着时间的推移，内存泄漏可能会导致内存不足，用户迟早需要重新启动程序，或者重启操作系统。

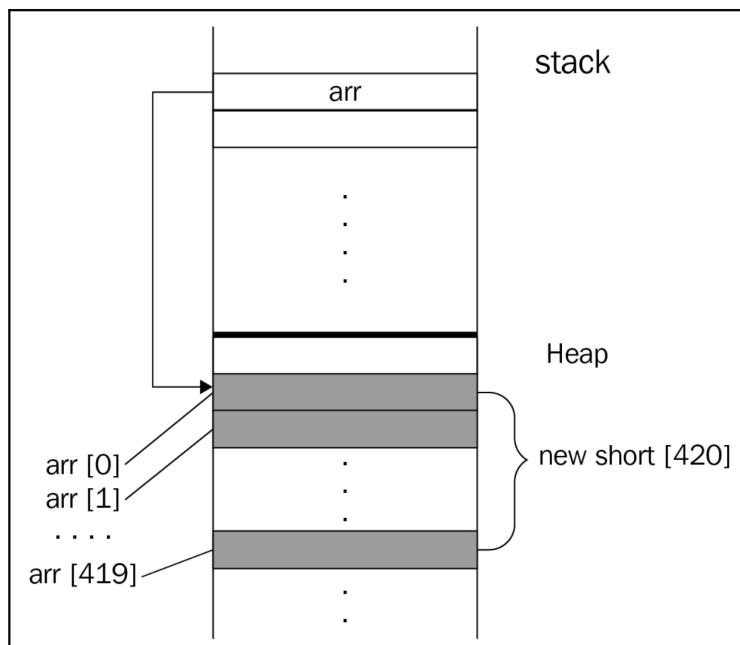
这个问题适用于使用的任何资源，无论是文件还是忘记关闭的套接字（更多关于套接字的内容，请参阅第 12 章）。为了解决这个问题，C++ 开发者使用了资源获取即初始化（RAII）的习惯用法，在获得资源时初始化，适当时释放资源。让我们看看它的实际应用。

内存管理的例子

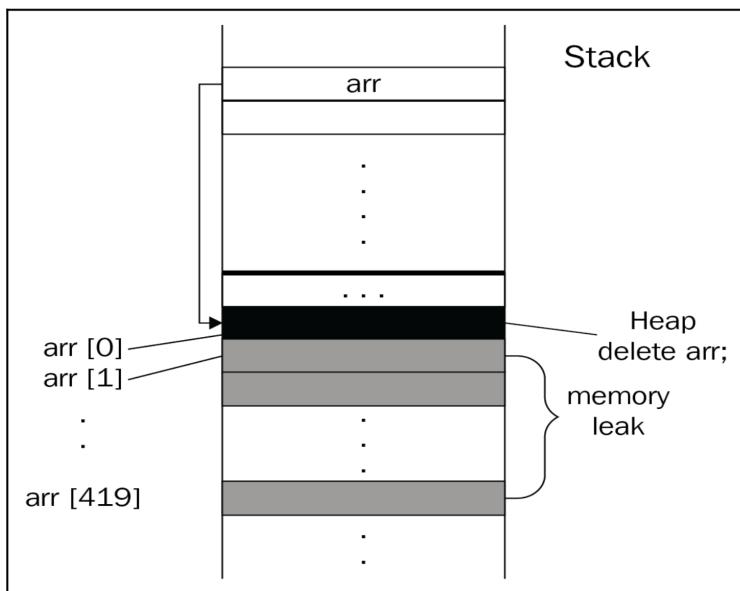
考虑以下函数，动态分配一个包含 420 个 short 的数组，从用户输入中读取它们的值，按升序打印它们，然后释放数组：

```
1 void print_sorted() {
2     short* arr{new short[420]};
3     for (int ix = 0; ix < 420; ++ix) {
4         std::cin >> arr[ix];
5     }
6     std::sort(arr, arr + 420);
7     for (int ix = 0; ix < 420; ++ix) {
8         std::cout << arr[ix];
9     }
10    delete arr; // very bad!
```

前面的代码中，我们已经犯了一个错误，使用了错误的 `delete` 操作符来释放内存。要释放数组，必须使用 `delete[]` 操作符，否则代码会导致内存泄漏。下面是我们如何演示数组的分配：



假设我们使用 `delete` 而不是 `delete[]` 来释放空间。它将把 `arr` 作为一个短指针来处理，因此将删除从 `arr` 指针中包含的地址开始的前两个字节，如下图所示：



所以现在我们从 420 个中移除第一个，并让 419 个 short 保持原样。当我们在堆上需要新的空间时，包含 419 个不可接触对象的那部分将不会再重用。尽管 `new` 和 `delete` 操作符族是由实现定义的，但我们不应该奢望以这种方式避免内存泄漏。

让我们修改前面的代码，以正确释放为数组分配的内存，并确保消除了输入负数的可能：

```

1 void print_sorted() {
2     short* arr{new short[420]};
3     for (int ix = 0; ix < 420; ++ix) {
4         std::cin >> arr[ix];
5         if (arr[ix] < 0) return;
6     }
7     std::sort(arr, arr + 420);
8     // print the sorted array, code omitted for brevity
9     delete[] arr;
10}

```

前面的修改是可能的内存泄漏的另一个例子，不过为了简单起见，我们编写了难看的代码。只要用户输入一个负数，函数就返回。这就给我们留下了 420 个孤儿 short。然而，对分配的内存的唯一访问是 arr 指针，它是在堆栈上声明的，因此当函数返回时，它将自动删除（指针变量，而不是指向它的内存空间）。为了消除内存泄漏的可能性，只需在函数退出之前使用 delete[] 操作符：

```

1 void print_sorted() {
2     short* arr{new short[420]};
3     for (int ix = 0; ix < 420; ++ix) {
4         std::cin >> arr[ix];
5         if (arr[ix] < 0) {
6             delete[] arr;
7             return;
8         }
9     }
10    // sort and print the sorted array, code omitted for brevity
11    delete[] arr;
12}

```

代码变得有些难看，但它修复了内存泄漏。如果进一步修改该函数，并使用第三方标准库函数对数组进行排序，会怎么样呢？

```

1 import <strange_sort.h>;
2 void print_sorted() {
3     short* arr{new short[420]};
4     for (...) { /* code omitted for brevity */ }
5     strange_sort::sort(arr, arr + 420);
6     // print the sorted array, code omitted for brevity
7     delete[] arr;
8 }

```

结果是，当数组项的值超过 420 时，strange_sort::sort 会抛出异常（这就是为什么它是一个奇怪的排序）。如果异常未被捕获，将使用冒泡排序的函数，除非它在某个地方被捕获或程序崩溃。未捕获的异常导致堆栈展开，这将销毁 arr 变量（指针）导致自动，因此我们面临另一种内存泄漏的可能性。为了解决这个问题，我们可以将 strange_sort::sort 封装在一个 try-catch 块中：

```

1 try {
2     strange_sort::sort(arr, arr + 420);
3 } catch (ex) { delete[] arr; }

```

C++ 开发者经常寻找处理内存泄漏的方法，例如：RAII 习惯用法和智能指针。我们将在下一节中讨论这些问题。

使用智能指针

许多语言支持自动垃圾收集，例如：运行时环境会跟踪为对象获取的内存。引用该对象超出作用域后，它将释放内存空间。例如，考虑以下情况：

```
1 // a code sample of the language (not-C++) supporting automated garbage
2 collection
3 void foo(int age) {
4     Person p = new Person("John", 35);
5     if (age <= 0) { return; }
6     if (age > 18) {
7         p.setAge(18);
8     }
9     // do something useful with the "p"
10 }
11 // no need to deallocate memory manually
```

前面的代码中，`p` 引用（通常，垃圾收集语言中的引用类似于 C++ 中的指针）指向由 `new` 操作符返回的内存位置。自动垃圾收集器管理由 `new` 操作符创建的对象的生存期，还跟踪对该对象的引用。只要对象上没有引用，垃圾收集器就会释放它的空间，通过 C++ 中的 RAII 习语可以实现类似的功能。让我们看看它的实际应用。

利用 RAII

如前所述，RAII 习惯用法建议在资源初始化时获取资源。看看下面的类：

```
1 template <typename T>
2 class ArrayManager {
3 public:
4     ArrayManager(T* arr) : arr_{arr} {}
5     ~ArrayManager() { delete[] arr_; }
6     T& operator[](int ix) { return arr_[ix]; }
7     T* raw() { return arr_; }
8 };
```

`print_sorted` 函数可以使用 `ArrayManager` 来正确释放已分配的数组：

```
1 void print_sorted() {
2     ArrayManager<short> arr{new short[420]};
3     for (int ix = 0; ix < 420; ++ix) {
4         std::cin >> arr[ix];
5     }
6     strange_sort::sort(arr.raw(), arr.raw() + 420);
7     for (int ix = 0; ix < 420; ++ix) {
8         std::cout << arr[ix];
9     }
}
```

10 }

建议使用标准容器，如 std::vector，而不是 ArrayManager，尽管它是 RAII 应用程序的一个很好的例子：初始化时获取资源。我们创建了 ArrayManager 的实例，并使用内存资源对它进行了初始化。从这一点上，我们可以忘记它的释放，因为实际的释放发生在 ArrayManager 的析构函数中。当我们在堆栈上声明 ArrayManager 实例时，当函数返回或发生未捕获的异常时，它将自动销毁，析构函数将被调用。

这个场景中，首选使用标准容器，因此让我们为单个指针实现 RAII 用法。下面的代码为 Product 实例动态分配内存：

```
1 Product* apple{new Product};
2 apple->set_name("Red apple");
3 apple->set_price(0.42);
4 apple->set_available(true);
5 // use the apple
6 // don't forget to release the resource
7 delete apple;
```

如果我们把 RAII 用法应用到前面的代码中，它会在适当的代码执行点释放资源：

```
1 ResourceManager<Product> res{new Product};
2 res->set_name("Red apple");
3 res->set_price(0.42);
4 res->set_available(true);
5 // use the res the way we use a Product
6 // no need to delete the res, it will automatically delete when gets out of
7 the scope
```

ResourceManager 类也应该重载操作符 * 和->，因为它必须表现得像一个指针，以便正确地获取和管理一个指针：

```
1 template <typename T>
2 class ResourceManager {
3     public:
4     ResourceManager(T* ptr) : ptr_{ptr} {}
5     ~ResourceManager() { delete ptr_; }
6     T& operator*() { return *ptr_; }
7     T* operator->() { return ptr_; }
8 };
```

ResourceManager 类体现了 C++ 中智能指针的思想。C++11 引入了几种类型的智能指针，因为它们围绕着资源并管理它的自动重新分配。当对象被设置为 destroy 时，将调用对象的析构函数。也就是说，我们通过具有自动存储时间的对象动态分配空间进行操作。当处理程序对象超出作用域时，它的析构函数执行必要的操作来释放底层资源。

然而，智能指针可能带来额外的问题。上一段讨论的简单智能指针有几个最终会出现的问题。例如，我们没有注意 ResourceManager 的复制：

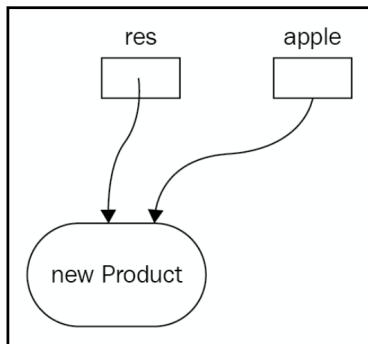
```
1 void print_name(ResourceManager<Product> apple) {
2     std::cout << apple->name();
```

```

3 }
4 ResourceManager<Product> res{new Product};
5 res->set_name("Red apple");
6 print_name(res);
7 res->set_price(0.42);
8 // ...

```

前面的代码导致未定义的行为。问题如下图所示：



`res` 和 `apple` 都拥有相同的资源。每当其中一个超出作用域 (`apple`) 时，底层资源就会释放，这使得另一个 `ResourceManager` 实例有一个悬空指针。当其他 `ResourceManager` 实例超出作用域时，将尝试删除该指针两次。通常，开发者知道他们在特定情况下需要哪种智能指针。这就是为什么 C++ 提供了几种类型的智能指针，我们将进一步讨论这些类型。要在程序中使用它们，需要导入 `<memory>` 头文件。

`std::unique_ptr`

与我们前面实现的 `ResourceManager` 实例类似，`std::unique_ptr` 表示智能指针。例如，要使用这个智能指针管理 `Product` 对象：

```

1 std::unique_ptr<Product> res{new Product};
2 res->set_name("Red apple");
3 // res will delete its acquired resource when goes out of scope

```

请注意我们如何访问 `Product` 成员函数 `set_name`。我们将 `res` 对象视为具有类型指针的对象。

`unique_ptr` 之所以命名为 `unique`，因为提供了严格所有权的语义——它有义务销毁获得的对象，而且 `unique_ptr` 不能复制，它没有复制构造函数或赋值操作符。这就是为什么它是所有权是严格的。当然，这并不意味着不能移动 `unique_ptr`。这种情况下，我们可以将所有权传递给另一个实例。

智能指针的要求之一是轻量级。虽然 `unique_ptr` 是一个包含多个成员函数的完整类，但它不会污染其他数据成员，只是一个封装了指向已分配对象的原始指针的容器。可以通过调用 `unique_ptr` 的 `release()` 成员函数来访问这个原始指针，如下所示：

```

1 Product* p = res.release();
2 // now we should delete p manually to deallocate memory

```

注意 `release()` 函数不调用 `delete` 操作符，只会归还所有权。调用 `release()` 函数后，`unique_ptr` 将不再拥有该资源。要重用已经拥有资源的 `unique_ptr`，应该使用 `reset()` 成员函数。为基础指针

调用 delete 操作符，并重置智能指针，以供进一步使用。另一方面，如果想在不释放所有权的情况下获得基础对象，应该调用 get():

```
1 std :: unique_ptr<Product> up{new Product()};
2 Product* p = res.get();
3 // now p also points to the object managed by up
```

在以下场景中不能使用 unique_ptr 类，因为它不能被复制:

```
1 // Don't do this
2 void print_name( std :: unique_ptr<Product> apple) {
3     std :: cout << apple->name();
4 }
5 std :: unique_ptr<Product> res{new Product};
6 res->set_name("Red apple");
7 print_name(res); // bad code
8 res->set_price(0.42);
9 // ...
```

让我们继续 C++ 中的下一个智能指针，它解决了将 unique_ptr 传递给函数的问题。

std::shared_ptr 和 std::weak_ptr

我们需要提供共享所有权的智能指针，C++11 中引入的 std::shared_ptr。实现具有共享所有权的智能指针比较困难，需要注意资源的正确重新分配。例如，当前面代码块中的 print_name() 函数完成它的工作时，它的参数和局部对象将销毁。销毁智能指针会导致它所拥有的资源被适当地重新分配。智能指针如何知道该资源是否仍为另一个智能指针所拥有？一种解决方案是保存对资源的引用计数。shared_ptr 类做了同样的事情：它保留指向基础对象的指针的数量，并在使用计数变为 0 时删除它。因此，多个共享指针可以拥有同一个对象。

现在，我们刚的例子应该这样改写：

```
1 void print_name( std :: shared_ptr<Product> apple) {
2     std :: cout << apple->name();
3 }
4 std :: shared_ptr<Product> res{new Product};
5 res->set_name("Red apple");
6 print_name(res);
7 res->set_price(0.42);
8 // ...
```

调用 print_name() 函数后，共享指针的使用计数增加 1。当函数完成它的工作，但托管对象不会释放时，计数将减少 1。这是因为 res 对象还没有超出作用域。让我们稍微修改一下示例，打印对共享对象的引用计数：

```
1 void print_name( std :: shared_ptr<Product> apple) {
2     std :: cout << apple.use_count() << " eyes on the " << apple->name();
3 }
4 std :: shared_ptr<Product> res{new Product};
5 res->set_name("Red apple");
6 std :: cout << res.use_count() << std :: endl;
```

```
7 print_name(res);
8 std::cout << res.use_count() << std::endl;
9 res->set_price(0.42);
10 // ...
```

上述代码将会在屏幕上显示如下信息:

```
1
2 2 eyes on the Red apple
3 1
```

当最后一个 shared_ptr 超出作用域时，它也会销毁基础对象。然而，共享指针之间共享对象时应该小心。下面的代码显示了共享所有权的问题:

```
1 std::shared_ptr<Product> ptr1{new Product()};
2 Product* temp = ptr1.get();
3 if (true) {
4     std::shared_ptr<Product> ptr2{temp};
5     ptr2->set_name("Apple of truth");
6 }
7 ptr1->set_name("Peach"); // danger!
```

ptr1 和 ptr2 都指向同一个对象，但是它们不知道对方的存在。因此，当我们通过 ptr2 修改 Product 对象时，将影响 ptr1。当 ptr2 超出作用域（在 if 语句之后）时，将销毁基础对象，该对象仍然由 ptr1 拥有。之所以会出现这种情况，是因为我们通过将原始的 temp 指针传递给 ptr2，使其拥有对象。ptr1 无法追踪。

只能使用复制构造函数或 std::shared_ptr 的赋值操作符共享所有权。这样，如果该对象正在被另一个 shared_ptr 实例使用，就可以避免删除该对象。共享指针使用控制块实现共享所有权。每个共享指针包含两个指针，一个指向它管理的对象，另一个指向控制块。控制块表示动态分配的空间，其中包含资源的使用计数。它还包含其他几个对 shared_ptr 至关重要的东西，例如：资源的分配器和删除器。我们将在下一节中介绍分配器。删除器通常是常规的删除操作符。

控制块还包含弱引用的数量。这样做是因为所拥有的资源也可能指向一个弱指针。std::weak_ptr 是 std::shared_ptr 的表示，它引用由 shared_ptr 实例管理的对象，但不拥有该对象。weak_ptr 是一种访问和使用 shared_ptr 拥有的资源，而不拥有它的方法。但是，有一种方法可以使用 lock() 成员函数将 weak_ptr 实例转换为 shared_ptr。

unique_ptr 和 shared_ptr 都可以用于管理动态分配的数组。模板参数必须正确指定:

```
1 std::shared_ptr<int[]> sh_arr{new int[42]};
2 sh_arr[11] = 44;
```

要访问基础数组的元素，可以使用共享指针的 [] 操作符。另外，可以在动态多态性中使用智能指针。假设我们有如下的类层次结构:

```
1 struct Base
2 {
3     virtual void test() { std::cout << "Base::test()" << std::endl; }
4 };
5
```

```

6 struct Derived : Base
7 {
8     void test() override { std::cout << "Derived::test()" << std::endl; }
9 };

```

以下代码按照预期工作，并将 Derived::test() 输出到屏幕：

```

1 std::unique_ptr<Base> ptr = std::make_unique_default_init<Derived>();
2 ptr->test();

```

尽管智能指针的使用可能会破坏指针的美感，但建议使用智能指针可以避免内存泄漏。值得注意的是，用智能指针替换所有指针，无论是 unique_ptr 指针还是 shared_ptr 指针，都不能解决所有的内存泄漏问题。他们也有自己的缺点。考虑一种平衡的方法，或者更好的方法，在将智能指针应用于问题之前，彻底地了解问题和智能指针本身的细节。

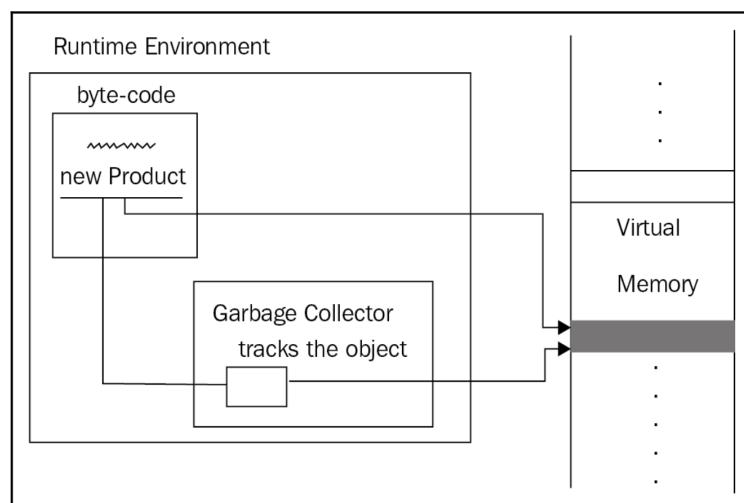
C++ 程序中管理内存是有代价的，我们讨论的最重要的事情是以适当地方式重新分配内存空间。语言不支持自动内存回收，但值得一提的是垃圾收集器。然而，要拥有一个完整的垃圾收集器，我们需要语言级的支持，但 C++ 没有提供任何这些。让我们试着实现一个属于 C++ 的垃圾收集器。

垃圾收集

垃圾收集器是一个单独的模块，通常合并在可解释语言的运行时环境中。C# 和 Java 都有垃圾收集器，这使开发者的工作变得容易得多。垃圾收集器跟踪代码中所有的对象分配，并在它们不再使用时释放它们。称为垃圾收集器，因为它在使用内存资源后删除内存资源：收集开发者留下的垃圾。

据说 C++ 程序员不会留下垃圾，这就是为什么语言不支持垃圾收集器的原因。尽管开发者倾向于为语言进行辩护，说它没有垃圾收集器，因为它是一种快速的语言，但事实是有垃圾收集器也不影响速度。

像 C# 这样的语言，将程序编译成中间字节码表示，然后由运行时环境解释和执行。垃圾收集器是环境的一部分，并积极跟踪所有对象分配。它是一种复杂的动物，它会尽力在合理的时间内管理内存。下图描述了一个典型的运行时环境，在垃圾收集器的监督下分配内存：



C++ 中使用智能指针，也可以手动调用 `delete` 操作符来释放内存空间。智能指针只获取对象，并在对象超出作用域时删除对象。关键的是，即使智能指针引入了一些半自动行为，仍然像开发者没有忘记在代码的指定点释放资源一样工作。垃圾收集器会自动执行，并且通常使用单独的执行线程。它尽量不减慢实际程序的执行速度。

一些垃圾收集实现技术包括根据对象的生存期对其进行分类，分类使垃圾收集器访问对象，并在对象不再使用时释放内存空间。为了使这个进程更快，应该访问生命周期较短的对象比访问生命周期较长的对象更频繁。以下面的代码为例：

```
1 struct Garbage {
2     char ch;
3     int i;
4 };
5
6 void foo() {
7     Garbage* g1 = new Garbage();
8     if (true) {
9         Garbage* g2 = new Garbage();
10    }
11 }
12
13 int main() {
14     static Garbage* g3 = new Garbage();
15 }
```

如果 C++ 有一个垃圾收集器，那么对象 `g1`、`g2` 和 `g3` 将在程序执行的不同时间段删除。如果垃圾收集器根据生命周期进行分类，那么 `g2` 的生命周期将是最短的，并且应该首先访问它以释放它。

真正在 C++ 中实现垃圾收集器，应该让它成为程序的一部分。垃圾收集器首先应该负责分配内存来跟踪和删除它：

```
1 class GarbageCollector {
2 public:
3     template <typename T>
4     static T* allocate() {
5         T* ptr{new T()};
6         objects_[ptr] = true;
7         return ptr;
8     }
9     static void deallocate(T* p) {
10        if (objects_[p]) {
11            objects_[p] = false;
12            delete p;
13        }
14    }
15 private:
16     std::unordered_map<T*, bool> objects_;
17 };
```

前面的类跟踪通过静态 `allocate()` 函数分配的对象。如果对象正在使用，则通过 `deallocate()` 函数删除该对象。下面是如何使用垃圾收集器：

```
1 int* ptr = GarbageCollector::allocate<int>();
2 *ptr = 42;
3 GarbageCollector::deallocate(ptr);
```

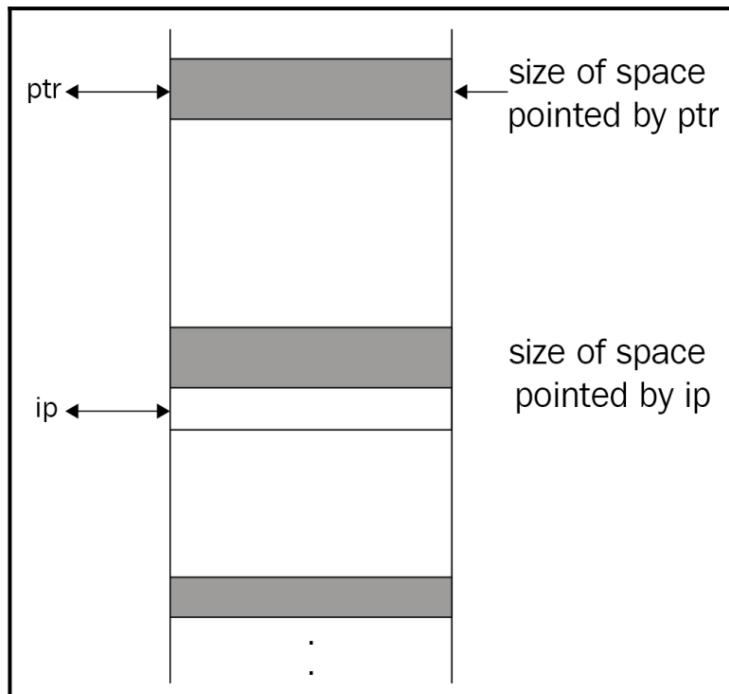
这个类使内存管理比智能指针更难一些。基本上，在 C++ 中不需要实现垃圾收集器，因为智能指针可以处理几乎所有与自动内存回收有关的场景。

但是，让我们来看看垃圾收集器正确地释放指针所指向空间的技巧。前面最简单的实现中，我们跟踪了提供给用户的所有指针。每个指针都指向堆上的某个空间，这些空间应该在程序执行的某个点释放。GarbageCollector 中将使用标准的 `delete` 操作符。问题是，它怎么知道应该释放多少字节？看看下面的例子：

```
1 Student* ptr = new Student;
2 int* ip = new int{42};
3 // do something with ptr and ip
4 delete ptr;
5 delete ip;
```

假设一个 `Student` 实例占用 40 个字节的内存，一个整数占用 4 个字节。我们应该以某种方式将该信息传递给 `delete` 操作符。在前面的代码中，我们删除了 `ptr` 和 `ip`，它们都指向不同大小的内存空间。那么它如何知道在 `ptr` 的情况下 40 个字节应该标记为空闲，而在 `ip` 的情况下 4 个字节应该被标记为空闲？这个问题有不止一个解决方案，所以让我们看看其中一个。

无论何时分配内存，`new` 操作符都会将分配的内存大小置于实际内存空间之前，如下图所示：



然后，该信息由 `delete` 操作符使用，该操作符通过读取放置在内存空间之前的相应字节，来获取内存空间的大小。C++ 最关心的问题是管理数据集合的内存。STL 容器，如 `std::vector` 和

`std::list`, 有不同的处理内存的模型。默认情况下, 容器有一个指定的内存分配器, 用于处理容器元素的内存分配和回收。

使用分配器

分配器背后的思想是为容器内存管理提供控制。简单地说, 分配器是 C++ 容器的高级垃圾收集器。尽管在容器内存管理的范围内讨论分配器, 但肯定可以将这个想法扩展到通用的垃圾收集器。本节的开始部分, 实现了一个设计糟糕的垃圾收集器。检查分配器时, 会发现设计糟糕的 `garbagcollector` 类和 C++ 中的默认分配器之间有很多相似之处。在 `<memory>` 中定义的默认分配器有两个基本函数——`allocate()` 和 `deallocate()`。`allocate()` 函数的定义如下:

```
1 [[nodiscard]] constexpr T* allocate(std::size_t num);
```

`allocate()` 函数为类型为 `T` 的 `num` 对象获取空间。请注意 `[[nodiscard]]` 属性——它意味着返回值不应该被调用者丢弃。否则, 编译器将报出警告消息。

使用分配器, 为 5 个整数获取空间:

```
1 import <memory>;
2 int main()
3 {
4     std::allocator<int> IntAlloc;
5     int* ptr = IntAlloc.allocate(5);
6     // construct an integer at the second position
7     std::allocator_traits<IntAlloc>::construct(IntAlloc, ptr + 1, 42);
8     IntAlloc.deallocate(ptr, 5); // deallocate all
9 }
```

注意, 我们是如何使用 `std::allocator_traits` 在已分配的空间中构造对象的。

`deallocate()` 函数的定义如下:

```
1 constexpr void deallocate(T* p, std::size_t n)
```

前面的代码片段中, 我们通过传递 `allocate()` 函数返回的指针来使用 `deallocate()` 函数。

可能不会在项目中直接使用分配器, 但是每当需要内存管理的自定义行为时, 使用现有的或引入新的分配器可能会有帮助。STL 容器使用分配器主要是因为在结构和行为上的不同, 这导致需要对内存分配和回收有专门的行为。我们将在下一章更详细地讨论 STL 容器。

总结

C# 等语言中提供了垃圾收集器。它们与用户程序并行工作, 并试图在程序看起来有效时在程序结束后进行清理。不能在 C++ 中做同样的事情, 我们所能实现的就是在程序中直接实现垃圾收集器, 提供一种半自动的方式来释放使用过的内存资源。自 C++11 以来, 语言中引入的智能指针, 可以适当地涵盖这种机制。

内存管理是每个计算机程序的关键组成部分之一。程序应该能够在其执行期间动态地请求内存。优秀的开发者理解内存管理的内部细节, 这有助于他们设计和实现性能更高的应用程序。虽然手动内存管理是一种优势, 在大型应用程序中, 往往会变得很痛苦。本章中, 我们已经学习了如何

使用智能指针来避免错误和处理内存回收。有了这些基本的了解，就能增强设计避免内存泄漏的程序的信心。

下一章中，我们将学习 STL，关注数据结构和算法，并深入研究它们的 STL 实现。除了比较数据结构和算法之外，还将介绍 C++20 中一个值得注意的新特性：概念。

问题

1. 解释计算机内存。
2. 什么是虚拟内存？
3. 哪些是用于内存分配和回收的操作符？
4. `delete` 和 `delete[]` 有什么区别？
5. 什么是垃圾收集器？为什么 C++ 不支持垃圾收集？

扩展阅读

更多信息，请参阅以下链接：

- What every programmer should know about memory, by Ulrich Drepper, at <https://people.freebsd.org/~ulrichd/articles/cpumemory.pdf>
- Code: The hidden language of computer hardware and software, by Charles Petzold, at [https://www.amazon.com/Code-Language-Computer-Hardware- Software/dp/0735611319/](https://www.amazon.com/Code-Language-Computer-Hardware-Software/dp/0735611319/)

2 设计健壮和高效的应用

本节将集中讨论数据结构、算法和并发工具数据处理的效率，并且还将介绍基本的设计模式和最佳实践。

本节包括以下章节：

- 第 6 章，挖掘 STL 中的数据结构和算法
- 第 7 章，函数式编程
- 第 8 章，并发和多线程
- 第 9 章，设计并发式数据结构
- 第 10 章，设计实际程序
- 第 11 章，使用设计模式设计策略游戏
- 第 12 章，网络和安全
- 第 13 章，调试与测试
- 第 14 章，使用 Qt 开发图形界面

第 6 章：挖掘 STL 中的数据结构和算法

掌握数据结构对开发者来说很有必要，大多数时候存储数据的方式决定了应用程序的效率。以电子邮件客户端为例，你可以设计一个电子邮件客户端，显示最近 10 封电子邮件，用户在使用你的应用两年内会收到数十万封电子邮件。当用户需要搜索电子邮件时，数据结构就会发挥重要作用。你储存成千上万封电子邮件的方式，排序和搜索它们的方法（算法）则非常重要。

开发者在项目中努力寻找日常问题的最佳解决方案，使用经过验证的数据结构和算法可以极大地改善工作效率。好的程序最重要的特点是速度，我们可以通过设计新的算法，或使用现有的算法来得到良好的运行速度。

C++20 引入了元类型的概念——描述其他类型的类型，这个强大特性使数据架构更加完整。

C++ 标准模板库 (STL) 中包含了大量的数据结构和算法。我们将探索利用 STL 容器来使用数据结构有效地组织数据的方法，然后将深入研究 STL 提供的算法实现。理解和使用 STL 容器中的概念至关重要，因为 C++20 通过引入迭代器概念，对迭代器进行了重大改进。

本章中，我们将了解以下内容：

- 数据结构
- STL 容器
- 概念和迭代器
- 主流算法
- 探索树和图

编译器要求

g++ 编译器需要添加编译选项 `-std=c++2a` 来编译本章的代码。可以从这里获取本章的源码文件：<https://github.com/PacktPublishing/Expert-CPP>

数据结构

开发者可能很熟悉使用数组来存储和排序数据，也会在项目中大量使用数据结构，而不是数组。了解和应用正确的数据结构可能在程序性能中扮演着重要的角色。为了选择正确的数据结构，就需要更好地了解它们。一个明显的问题可能会出现：我们是否需要研究数据结构的集合——向量、链表、哈希表、图、树等。为了回答这个问题，让我们想象一个场景，在这个场景中，合适的数据结构的必要性将很明显地显现出来。

之前介绍中，我们提到了设计电子邮件客户端，大致的了解一下它的设计和实现。

电子邮件客户端是列出不同发件人邮件的应用。我们可以在台式电脑或智能手机上安装它，或者使用浏览器版本。电子邮件客户端应用的主要任务包括发送和接收电子邮件。现在假设我们正在设计简单的电子邮件客户机，假设我们使用一些库来封装发送和接收电子邮件的工作，这让我们更专注于设计专门用于存储和检索电子邮件的机制。客户端用户能够查看留在收件箱的邮件列表，还应该考虑用户可能想要对电子邮件执行的操作。他们可以一个个的删除，或者一次性删除很多，也可以随机选择任何一封邮件，回复发件人或转发给其他人。

我们将在第 10 章中讨论软件设计过程和最佳实践。现在，让我们构建一个简单的 Email 对象，如下所示：

```
1 struct Email
2 {
3     std::string subject;
4     std::string body;
5     std::string from;
6     std::chrono::time_point<std::chrono::system_clock> datetime;
7 };
```

第一件困扰我们的事情是将一组电子邮件存储容易访问的结构中，数组听起来可能不错。假设我们将所有收到的电子邮件存储在一个数组中，如下面的代码块所示：

```
1 // let's suppose a million emails is the max for anyone
2 const int MAX_EMAILS = 1'000'000;
3 Email inbox[MAX_EMAILS];
```

我们可以以任何形式存储 10 封电子邮件——这不会影响应用的性能。然而，随着时间的推移，电子邮件的数量会增长。对于新收到的电子邮件，将带有相应字段的 Email 对象推送到 inbox 数组中，数组的最后一个元素表示最近收到的电子邮件。因此，要显示最近 10 封电子邮件的列表，需要读取并返回数组的最后 10 个元素。

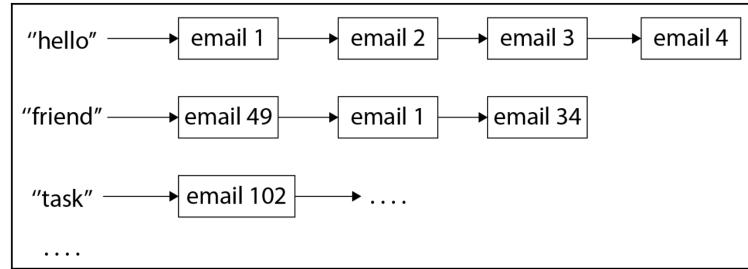
当试图操作存储在 inbox 数组中的数千封电子邮件时，问题就出现了。如果想在所有的电子邮件中搜索“friend”这个词，就必须扫描数组中的所有电子邮件，并在单独的数组中收集包含单词 friend 的电子邮件：

```
1 std::vector<Email> search(const std::string& word) {
2     std::vector<Email> search_results;
3     for (all-million-emails) {
4         if (inbox[i].subject.contains(word)) {
5             search_results.push_back(inbox[i]);
6         }
7     }
8     return search_results;
9 }
```

对于小型集合来说，使用数组存储数据绰绰有余。处理更大数据集的应用中，这种情况会变得复杂。使用特定数据结构的目的是使应用运行得更流畅。前面的示例展示了一个简单的问题：搜索电子邮件列表以匹配特定的值。在电子邮件中进行查找时，需要在合理的时间范围内。

如果假设电子邮件的主题可能包含最多 10 个单词，那么搜索电子邮件主题中的特定单词需要将该单词与主题中的所有单词进行比较。最坏的情况下，没有任何匹配。强调最坏的情况，是因为只有在这种情况下的查找，才需要检查主题中的每个单词。对成千上万的电子邮件执行同样的操作，会让用户等的不耐烦。

就应用程序效率而言，为特定问题选择正确的数据结构至关重要，例如：假设使用哈希表将单词映射到 Email 对象，每个单词都将映射到包含该单词的电子邮件对象列表。这种方法将提高搜索操作的效率，如下图所示：



`search()` 函数只返回哈希表键所指向的列表:

```

1 std :: vector<Email> search( const std :: string& word ) {
2     return table[word];
3 }
```

这种方法只需要处理每个收到的电子邮件，将其拆分为单词并更新哈希表。

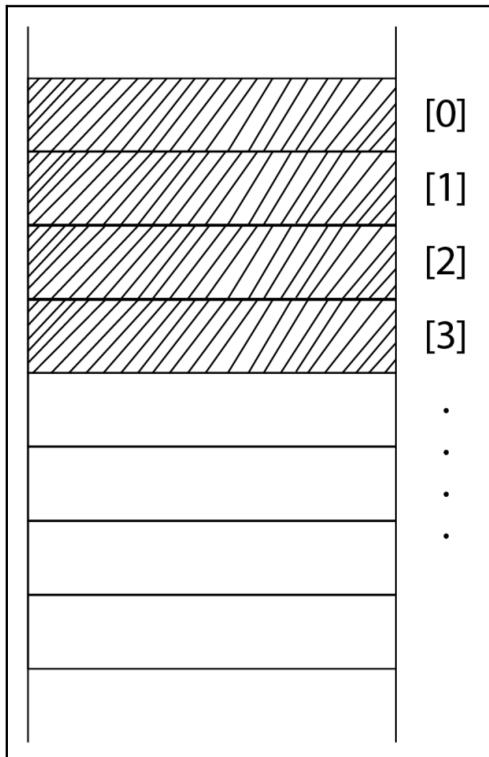


TIP 简单起见，直接使用 Email 对象，而不是引用。请注意，最好将指向电子邮件的指针存储在 vector 中。

现在，让我们看看不同的数据结构及其应用。

连续的数据结构

开发人员使用的最常见数据结构是动态增长的一维数组——vector。STL 提供了一个相同名称的容器: `std::vector`。`vector` 的关键思想是，包含按顺序放置在内存中的相同类型的项。例如，由 4 个字节整数组成的向量，将具有如下的内存布局。每个框代表一个 4 字节的空间。向量的下标在下图的右侧:



vector 的物理结构允许实时访问它的任何元素。



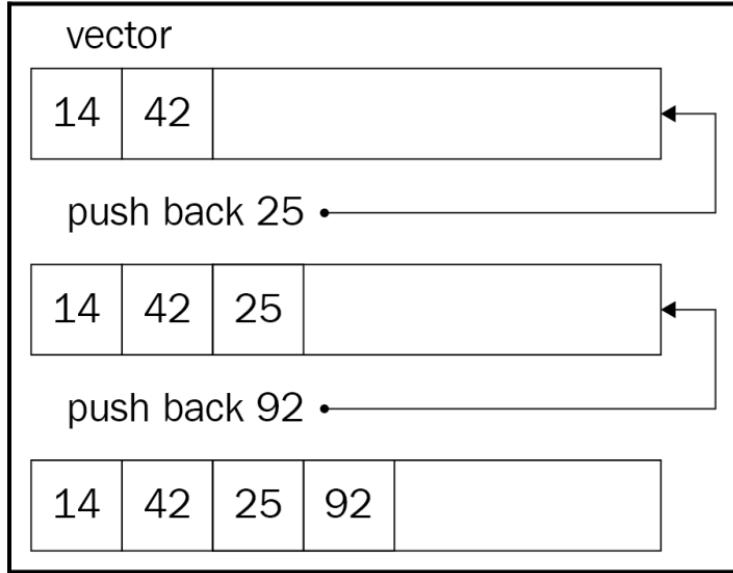
我们应该区分容器和操作，以便在问题中正确地应用。为此，我们根据容器中元素的数量定义操作时间复杂度。例如，vector 的元素访问可定义为常量时间操作，这意味着无论 vector 的长度如何，访问 vector 中的项都需要相同数量的指令。

访问 vector 的第一个元素和访问向量的第 100 个元素需要相同的工作量，这就称之为常数时间操作，也称为 O(1) 操作。

虽然 vector 中的元素访问速度很快，但添加新元素有点棘手。当在 vector 的末尾插入一个新元素时，还应该考虑 vector 的容量。当没有更多的空间分配给 vector 时，它应该动态地增加大小。看看下面的 Vector 类及其 push_back() 函数：

```
1 template <typename T>
2 class Vector
3 {
4 public:
5     Vector() : buffer_{nullptr}, capacity_{2}, size_{0}
6     {
7         buffer_ = new T[capacity_]; // initializing an empty array
8     }
9     ~Vector() { delete [] buffer_; }
10    // code omitted for brevity
11 public:
12     void push_back(const T& item)
13     {
14         if (size_ == capacity_) {
15             // resize
16         }
17         buffer_[size_++] = item;
18     }
19     // code omitted for brevity
20 };
```

深入研究 push_back() 函数的实现之前，让我们先看看下图：



我们应该分配一个全新的数组，将旧数组的所有元素复制到新数组中，然后将新插入的元素添加到新数组末尾的下一个空闲槽中。如下面的代码片段所示：

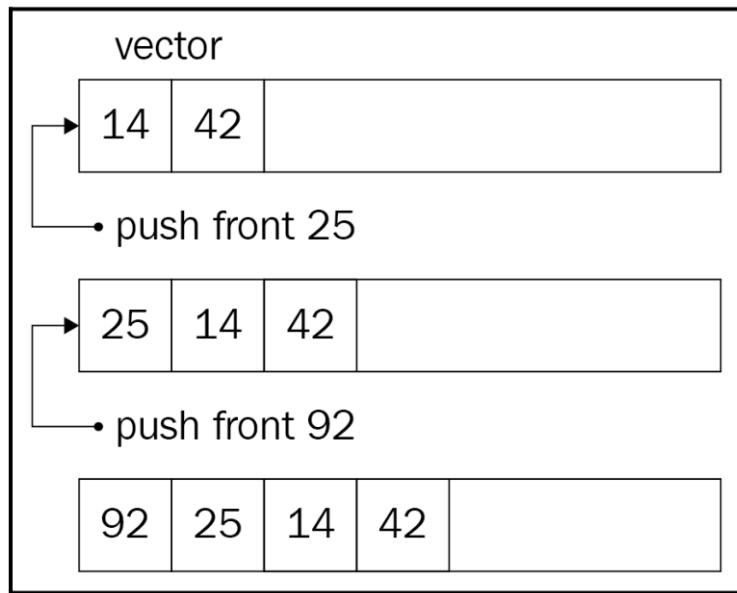
```

1 template <typename T>
2 class Vector
3 {
4     public:
5     // code omitted for brevity
6     void push_back(const T& item)
7     {
8         if (size_ == capacity_) {
9             capacity_ *= 2; // increase the capacity of the vector twice
10            T* temp_buffer = new T[capacity_];
11            // copy elements of the old into the new
12            for (int ix = 0; ix < size_; ++ix) {
13                temp_buffer[ix] = buffer_[ix];
14            }
15            delete [] buffer_; // free the old array
16            buffer_ = temp_buffer; // point the buffer_ to the new array
17        }
18        buffer_[size_++] = item;
19    }
20    // code omitted for brevity
21};

```

大小调整因子可以用不同的方式选择——我们将其设置为 2，这将使 vector 在满载时，空间增长两倍。因此，我们坚持认为，大多数情况下，在向量的末尾插入一个新项需要常数时间。它只是在空闲槽添加项，并增加其私有 size_ 变量。有时，添加新元素需要分配一个新的、更大的 vector，并需要将旧的 vector 复制到新的 vector 中。对于这样的情况，操作需要平摊常数时间才能完成。

但当我们在 vector 前面加一个元素时就不能这么说了，所有其他元素都应该向右移动一个槽，以便为新元素腾出一个槽，如下图所示：



下面是我们如何在 Vector 类中实现它:

```

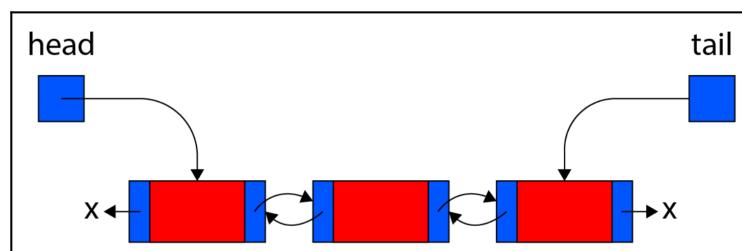
1 // code omitted for brevity
2 void push_front(const T& item)
3 {
4     if (size_ == capacity_) {
5         // resizing code omitted for brevity
6     }
7     // shifting all the elements to the right
8     for (int ix = size_ - 1; ix > 0; --ix) {
9         buffer_[ix] = buffer_[ix - 1];
10    }
11    // adding item at the front
12    buffer_[0] = item;
13    size_++;
14 }
```

在只需要在容器前端插入新元素的情况下，选择 vector 不是一个好的选择，这时需要考虑一下其他容器。

节点式数据结构

节点式的数据结构不需要连续的内存块。基于节点的数据结构为其元素分配节点，没有任何顺序——在内存中随机分布。我们将每个项目表示为一个节点，然后链接到其他节点的节点。

最流行的、入门级的节点式的数据结构是链表。下面的图表可视化显示了双链表的结构:



链表与向量有很大的不同。它的操作更快，但缺乏向量的紧凑性。

为了保持简短，我们在列表的前面实现元素插入，将每个节点保持为一个结构体：

```
1 template <typename T>
2 struct node
3 {
4     node(const T& item) : item{item}, next{nullptr}, prev{nullptr} {}
5     T item;
6     node<T>* next;
7     node<T>* prev;
8 };
```

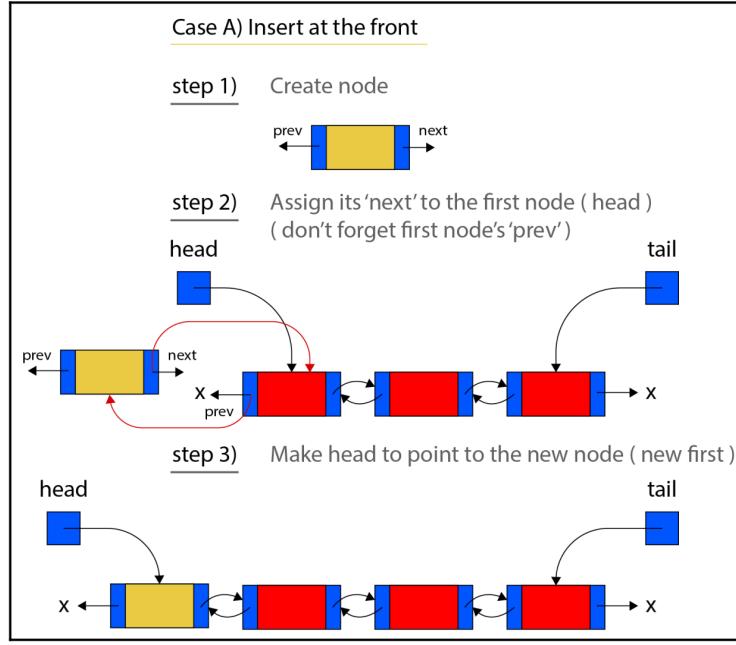
请注意 `next` 成员——它指向同一个结构体，这种方式允许将节点链接在一起，如前面的示例所示。

为了实现一个链表，需要一个指向它的第一个节点的指针，通常称为链表的头节点。在列表的前面插入一个元素很简单：

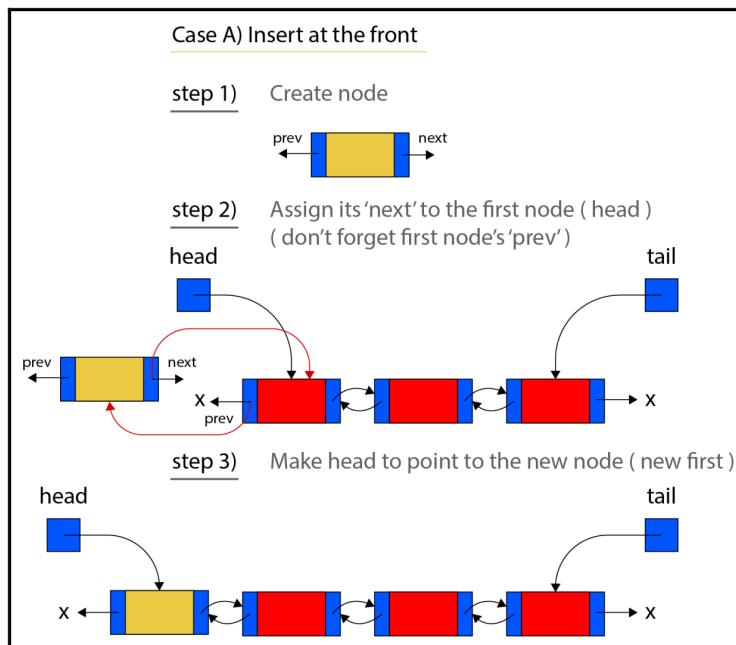
```
1 template <typename T>
2 class LinkedList
3 {
4     // code omitted for brevity
5 public:
6     void push_front(const T& item)
7     {
8         node<T>* new_node = new node<T>{item};
9         if (head_ != nullptr) {
10             new_node->next = head_->next;
11             if (head_->next != nullptr) {
12                 head_->next->prev = new_node;
13             }
14         }
15         new_node->next = head_;
16         head_ = new_node;
17     }
18 private:
19     node<T>* head_;
20 };
```

向列表中插入元素时，需要考虑以下三种情况：

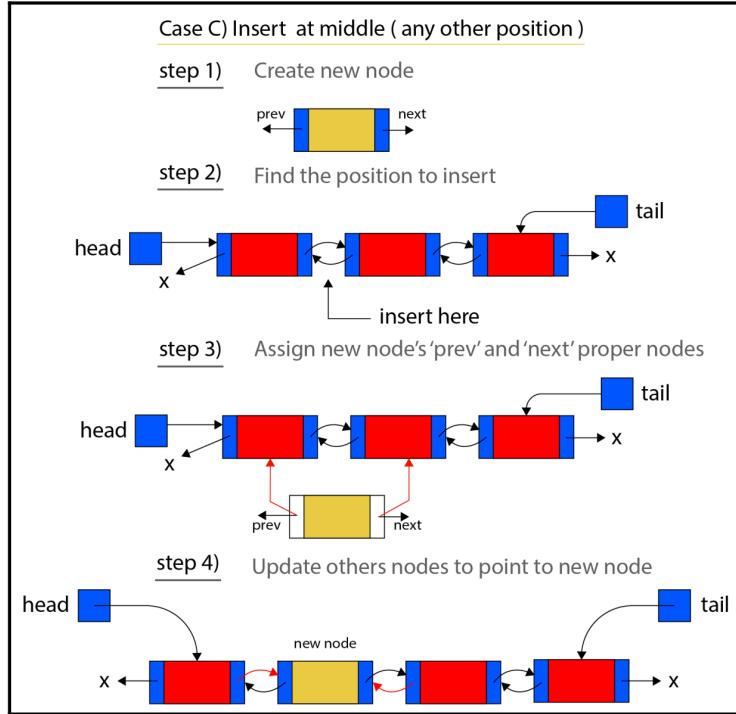
- 如前所述，在列表前面插入元素需要执行以下步骤：



- 在列表末尾插入一个元素，如下图所示:



- 最后，在列表中间插入一个元素的操作如下:



向 vector 对象中插入元素与向 list 对象中插入元素明显不同。如何在 vector 和 list 中选择？应该专注行为和速度。例如，从 vector 中读取任何元素需要常量时间。我们可以在一个向量中存储 100 万封电子邮件，并且不需要任何额外的努力就可以检索位置为 834,000 的电子邮件。对于链表，操作是线性的。因此，如果需要存储主要用于读取而不是写入的数据集合，那么使用 vector 是更合理的选择。

在列表的任何位置插入元素需要常量时间的操作，而 vector 将努力在随机位置插入元素。因此，当需要一个可以集中添加/删除数据的对象集合时，更好的选择是一个链表。

我们还应该考虑缓存内存。向量具有良好的数据局部性，读取 vector 对象的第一个元素需要将前 N 个元素复制到缓存中。进一步读取 vector 元素会更快。对于链表，我们就不能这么说了。为了找出原因，让我们继续比较 vector 和链表的内存布局。

容器的内存

如前面的章节所示，对象会在进程的内存段上占用一定的内存空间。大多数时候，我们感兴趣的是堆栈或堆内存。自动对象占用堆栈空间，下面两个声明都在栈中：

```

1 struct Email
2 {
3     // code omitted for brevity
4 };
5 int main() {
6     Email obj;
7     Email* ptr;
8 }
```

虽然 ptr 表示指向 Email 对象的指针，但也在堆栈上占用空间。可以指向在堆上分配的内存位置，但指针本身（存储内存位置地址的变量）驻留在堆栈上。进一步使用向量和列表之前，理解和

记住这一点非常重要。

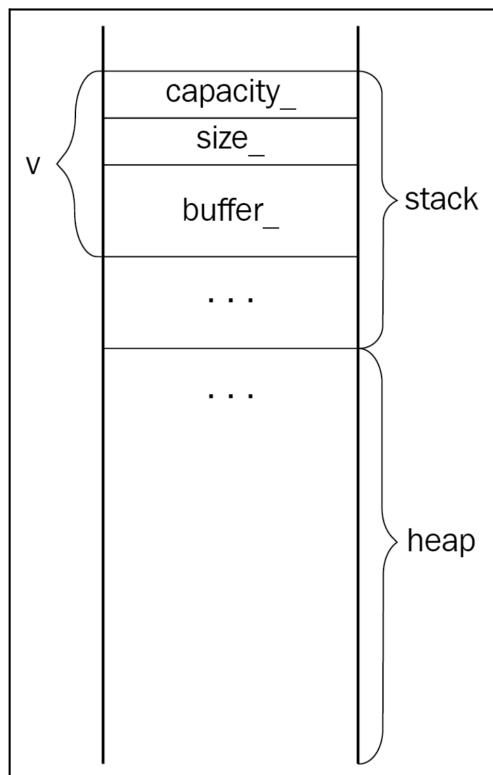
如本章前面所示，实现 vector 涉及到封装指向内部缓冲区的指针，该缓冲区代表指定类型的元素数组。声明 Vector 对象时，需要一定数量的堆栈内存来存储其成员数据。Vector 类有以下三个成员：

```
1 template <typename T>
2 class Vector
3 {
4 public:
5     // code omitted for brevity
6 private:
7     int capacity_;
8     int size_;
9     T* buffer_;
10};
```

假设一个整数占用 4 个字节，一个指针占用 8 个字节，下面的 Vector 对象声明将占用至少 16 个字节的堆栈内存：

```
1 int main()
2 {
3     Vector<int> v;
4 }
```

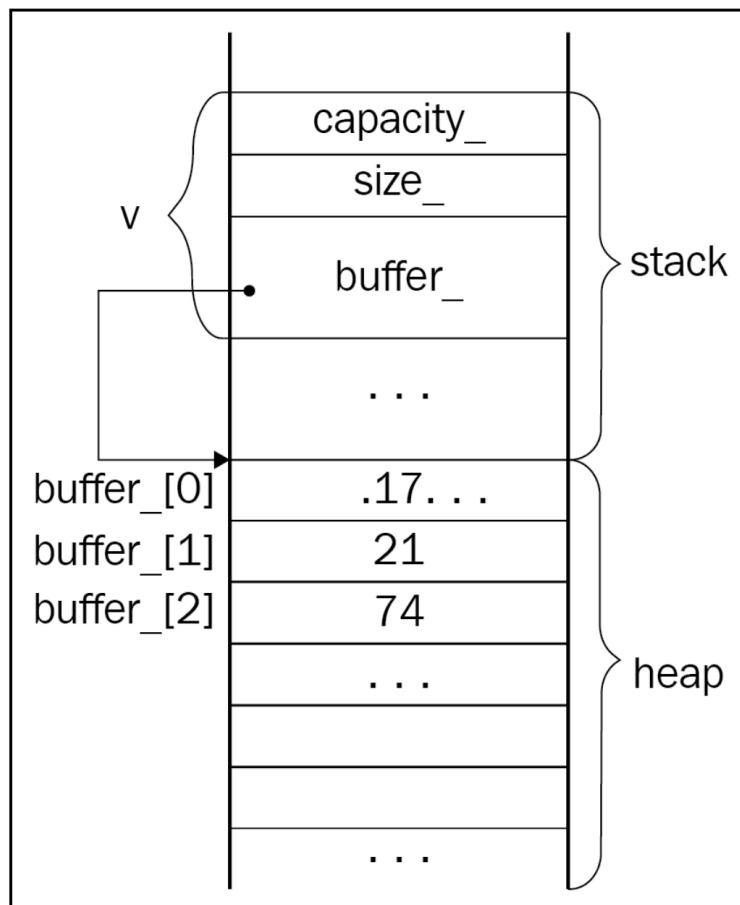
下面是对前面代码的内存布局的描述：



插入元素之后，堆栈上的 vector 对象的大小将保持不变，堆来保存现场。buffer_ 数组指向一个使用 new[] 操作符分配的内存位置。例如下面的代码：

```
1 // we continue the code from previous listing
2 v.push_back(17);
3 v.push_back(21);
4 v.push_back(74);
```

每一个推入 vector 的新元素都将占用堆中的空间，如下图所示：

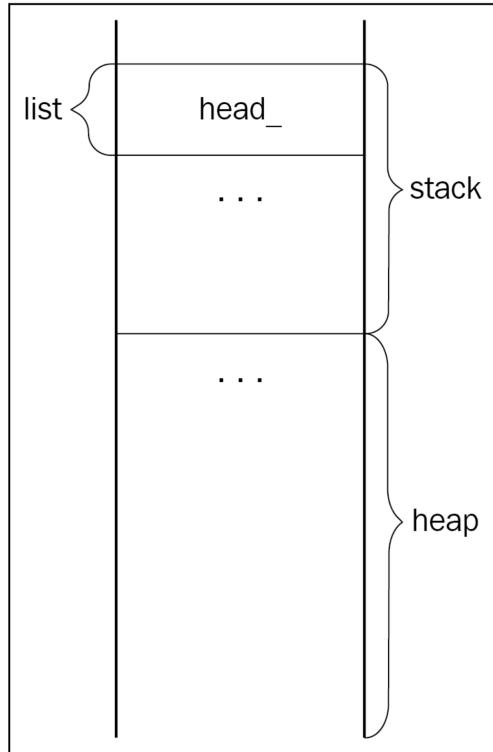


每个新插入的元素都位于 `buffer_` 数组的最后。这就是为什么我们可以说 `vector` 是一个对缓存友好的容器。

声明一个链表对象也会在堆栈上，为其数据成员占用内存空间。如果讨论只存储 `head_` 指针的简单实现，那么下面的 `list` 对象声明将占用至少 8 个字节的内存（仅针对 `head_` 指针）：

```
1 int main()
2 {
3     LinkedList<int> list;
4 }
```

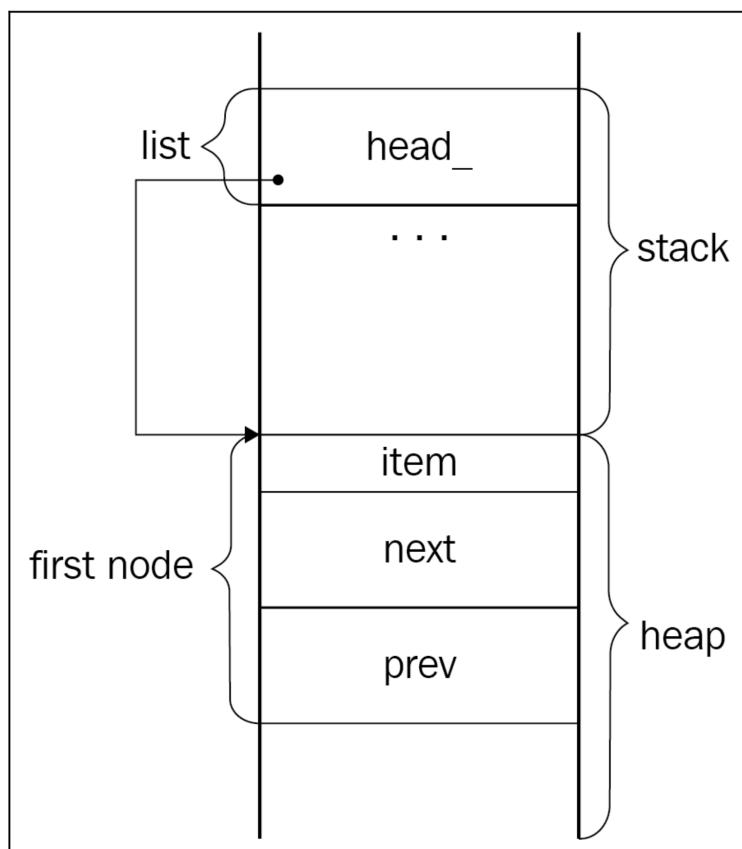
下面的图描述了上述代码的内存布局：



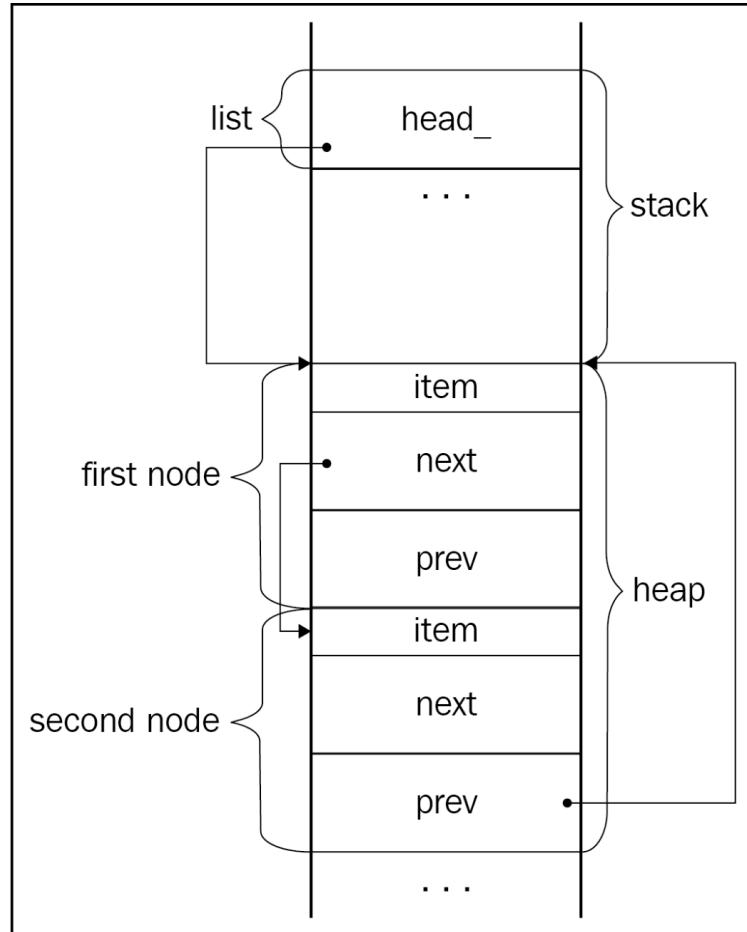
插入新元素会在堆上创建一个 node 类型的对象。看看下面这行:

```
1 list.push_back(19);
```

下面是插入新元素后内存图的变化:



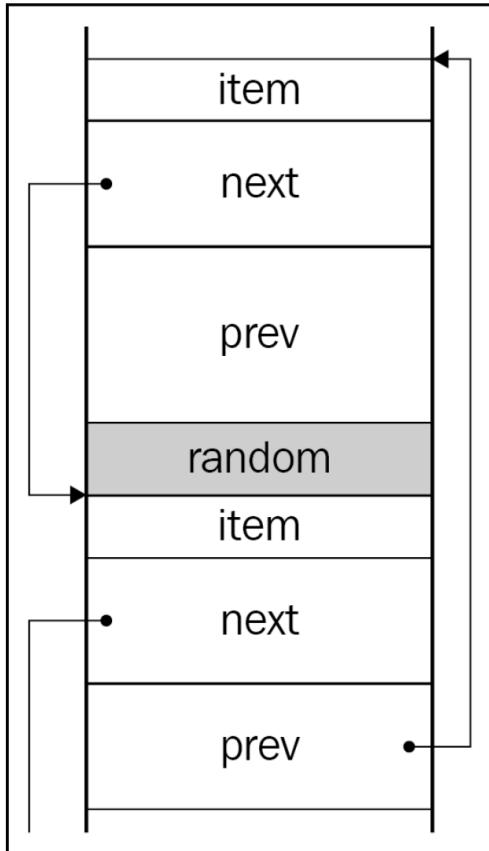
注意，节点的所有数据成员都驻留在堆上，item 存储我们要插入的值。当我们插入另一个元素时，将再次创建一个新节点。这次，第一个节点的下一个指针将指向新插入的元素。新插入节点的 prev 指针将指向列表的前一个节点。下图描述了插入第二个元素后链表的内存布局：



当向列表中插入元素的过程中在堆上分配一些随机对象时，会发生有趣的事情。例如，下面的代码将一个节点插入到列表中，然后为一个整数（与列表无关）分配空间。最后，再次向列表中插入一个元素：

```
1 int main()
2 {
3     LinkedList<int> list;
4     list.push_back(19);
5     int* random = new int(129);
6     list.push_back(22);
7 }
```

这种中间随机对象声明破坏了列表元素的顺序，如下图所示：



上面的图提示我们，由于列表的结构和元素的分配，它不是一个缓存友好的容器。



请注意 将每个新节点合并到代码中所产生的内存开销，我们为一个元素额外使用了 16 个字节（考虑到指针需要 8 个字节的内存）。因此，在最佳内存使用的比拼中，列表输给了 `vector`。

我们可以通过在列表中引入一个预先分配的缓冲区来解决这个问题。然后，每个新节点的创建都将通过重新实现的 `new` 操作符传递。更明智的做法是选择更适合问题的数据结构。

实际的应用程序开发中，开发者很少自己实现向量或链表，通常会使用经过测试的稳定库版本。C++ 同时提供了 `vector` 和链表的标准容器。此外，为单链表和双链表提供了独立的两个容器。

STL 容器

STL 是一个强大的算法和容器集合。虽然理解和实现数据结构对程序员来说是一项很好的技能，但不必在项目中每次都去实现它们。标准库库提供商负责为我们实现稳定的、经过测试的数据结构和算法。通过了解数据结构和算法的细节，在解决问题时更好地选择 STL 容器和算法。

前面讨论的 `vector` 和链表在 STL 中实现为 `std::vector<T>` 和 `std::list<T>`，其中 `T` 是容器中每个元素的类型。除了类型之外，容器还接受分配器作为第二个默认模板参数。例如，`std::vector` 的声明如下：

```

1 template <typename T, typename Allocator = std::allocator<T>>
2 class vector;

```

正如前一章所介绍的，分配器处理容器元素的分配/回收。`std::allocator` 是 STL 中所有标准容器的默认分配器。一个更复杂的基于内存资源行为不同的分配器是 `std::pmr::polymorphic_allocator`。STL 提供了 `std::pmr::vector` 作为使用多态分配器的别名模板，定义如下：

```
1 namespace pmr {
2     template <typename T>
3     using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;
4 }
```

现在让我们来仔细的了解下 `std::vector` 和 `std::list`。

使用 `std::vector` 和 `std::list`

`std::vector` 定义在 `<vector>` 头文件中。最简单的用法示例：

```
1 #include <vector>
2
3 int main()
4 {
5     std::vector<int> vec;
6     vec.push_back(4);
7     vec.push_back(2);
8     for (const auto& elem : vec) {
9         std::cout << elem;
10    }
11 }
```

`vector` 是动态增长的，应该考虑增长因素。声明 `vector` 时，有默认的容量，然后在插入元素时增长。每当元素的数量超过 `vector` 的容量时，`vector` 就将容量增加一个给定的因子（通常是将容量翻倍）的量。如果我们知道 `vector` 中需要的元素的大致数量，可以通过使用 `reserve()` 方法首先为 `vector` 分配容量来优化它的使用。例如，下面的代码保留了 10,000 个元素的容量：

```
1 std::vector<int> vec;
2 vec.reserve(10000);
```

它强制 `vector` 为 10,000 个元素分配空间，从而避免在元素插入期间调整大小（除非达到 10,000 个元素的阈值）。

另一方面，如果遇到容量远远大于 `vector` 中实际元素数量的情况，则可以收缩 `vector` 以释放未使用的内存。我们需要调用 `shrink_to_fit()` 函数，如下面的示例所示：

```
1 vec.shrink_to_fit();
```

访问 `vector` 元素的方法与访问普通数组的方法相同，使用操作符 `[]`。但是，`std::vector` 提供了两个访问其元素的选项。其中一种方法认为是安全的，通过 `at()` 函数完成，如下所示：

```
1 std::cout << vec.at(2);
2 // is the same as
3 std::cout << vec[2];
4 // which is the same as
5 std::cout << vec.data()[2];
```

at() 和操作符 [] 的区别在于, at() 访问指定元素时进行边界检查, 下面一行抛出了一个 std::out_of_range 异常:

```
1 try {
2     vec.at(999999);
3 } catch (std::out_of_range& e) { }
```

我们几乎以同样的方式使用 std::list。这些列表大多具有类似的公共接口。本章后面, 我们将讨论允许从特定容器中抽象的迭代器, 这样就可以用 vector 替换 list, 而不会有太多损失。在此之前, 让我们看看 list 和 vector 的公共接口的区别。

除了这两个容器都支持的标准函数集, 如 size()、resize()、empty()、clear()、erase() 等, 列表还有 push_front() 函数, 用于在列表的前插入一个元素。因为 std::list 表示一个双向链表。如下代码所示, std::list 也支持 push_back():

```
1 std::list<double> lst;
2 lst.push_back(4.2);
3 lst.push_front(3.14);
4 // the list contains: "3.14 -> 4.2"
```

列表的添加操作, 在许多情况下会派上用场。例如, 要合并两个排序列表, 我们使用 merge() 方法。它接受另一个列表作为参数, 并将其所有元素移动到当前列表中。作为参数传递给 merge() 方法的列表在操作之后变为空。



TIP STL 还提供了一个单链表, 用 std::forward_list 表示。要使用它, 你应该包含 <forward_list> 头文件。由于单链表节点只有一个指针, 因此它比双链表更省内存。

splice() 方法类似于 merge(), 不同的是它移动作为参数提供的一部分, 通过移动将内部指针重新指向合适的列表节点。merge() 和 splice() 都是如此。

当我们使用容器存储和操作复杂对象时, 复制元素的代价在程序的性能中扮演着重要的角色。考虑下面一个三维点的结构体:

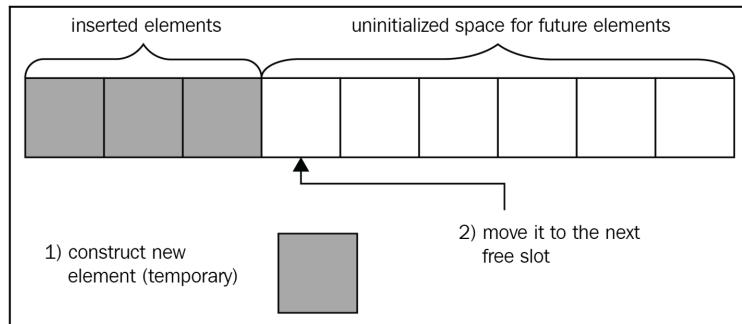
```
1 struct Point
2 {
3     float x;
4     float y;
5     float z;
6     Point(float px, float py, float pz)
7         : x(px), y(py), z(pz)
8     {}
9     Point(Point&& p)
10    : x(p.x), y(p.y), z(p.z)
11    {}
12};
```

现在, 将一个 Point 对象插入到 vector 中:

```
1 std::vector<Point> points;
```

```
2 points.push_back(Point(1.1, 2.2, 3.3));
```

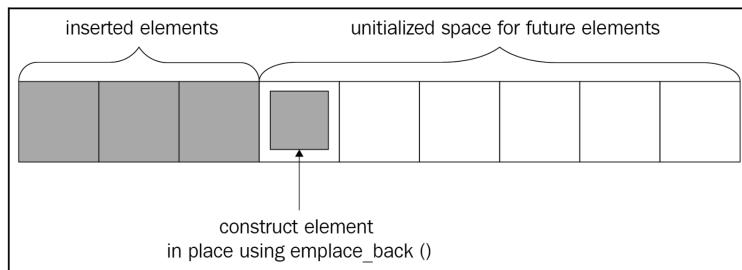
构造一个临时对象，然后移动到向量对应的槽中。我们可以直观地表示为：



vector 会提前占用更多的空间，从而尽可能长地延迟调整操作。当插入新元素时，vector 会将其复制到下一个可用的槽位（如果槽位已满，则会重新分配更多空间）。我们可以使用未初始化的空间在适当位置创建新元素。为此，vector 提供了 emplace_back() 函数。

```
1 points.emplace_back(1.1, 2.2, 3.3);
```

请注意这里直接传递给函数的参数。下图描述了 emplace_back() 的用法：



emplace_back() 通过 std::allocator_traits::construct() 构造元素。后者通常使用 new 操作符在已经分配的未初始化空间中构造元素。

std::list 还提供了一个 emplace_front() 方法。这两个函数都返回对插入元素的引用。唯一的要求是元素的类型必须是 EmplaceConstructible。对于 vector，类型也应该是 MoveInsertable。

使用容器适配器

作为开发者可能遇到过将堆栈和队列描述为数据结构（或者用 C++ 来表示容器）的情况。从技术上讲，它们不是数据结构，而是数据结构适配器。STL 中，std::stack 和 std::queue 通过提供访问容器的特殊接口来适配容器。堆栈这个词几乎无处不在，到目前为止，我们已经使用它来描述具有自动存储时间对象的内存段。由于分配/回收策略，这段内存称为堆栈。

我们说对象在每次声明时都会入栈，在销毁时出栈。对象的弹出顺序与它们推入顺序相反。这就是将内存段调用堆栈的原因，同样的后进先出（LIFO）方法也适用于堆栈适配器。std::stack 提供的关键功能如下：

```
1 void push(const value_type& value);
2 void push(value_type&& value);
```

`push()` 函数有效地调用了底层容器的 `push_back()`。通常，堆栈是使用 `vector` 来实现的。在介绍保护继承时，已经在第 3 章中讨论过这样的场景。`stack` 有两个模板形参，其中一个容器。选择什么并不重要，但它必须有 `push_back()` 成员函数。`std::stack` 和 `std::queue` 的默认容器是 `std::deque`。

`std::deque` 允许在开头和结尾快速插入，是一个类似于 `std::vector` 的索引顺序容器。名称 `deque` 表示双端队列。

让我们看看堆栈的行为：

```
1 #include <stack>
2
3 int main()
4 {
5     std::stack<int> st;
6     st.push(1); // stack contains: 1
7     st.push(2); // stack contains: 2 1
8     st.push(3); // stack contains: 3 2 1
9 }
```

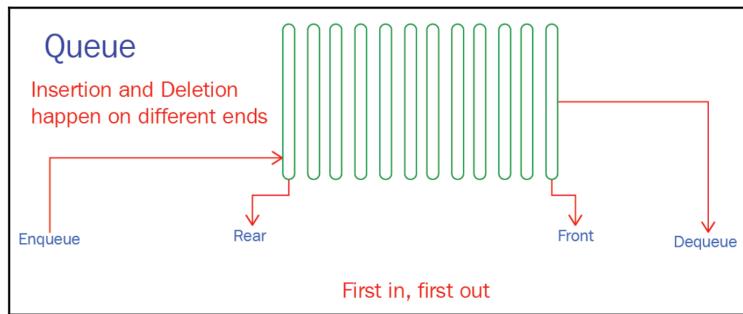
替代 `push()` 函数更好的方法是 `emplace()`。因此，调用底层容器的 `emplace_back()`，在适当的位置构造元素。

为了取出元素，我们调用 `pop()` 函数。它不接受任何参数，也不返回任何东西，它只是从堆栈中删除顶部元素。要访问堆栈的顶部元素，需要调用 `top()` 函数。修改前面的例子，在弹出堆栈元素之前打印所有的堆栈元素：

```
1 #include <stack>
2 int main()
3 {
4     std::stack<int> st;
5     st.push(1);
6     st.push(2);
7     st.push(3);
8     std::cout << st.top(); // prints 3
9     st.pop();
10    std::cout << st.top(); // prints 2
11    st.pop();
12    std::cout << st.top(); // prints 1
13    st.pop();
14    std::cout << st.top(); // crashes application
15 }
```

函数的作用：返回对顶部元素的引用，调用底层容器的 `back()` 函数。请注意在空堆栈上调用的最后一个 `top()` 函数。我们建议在对空堆栈调用 `top()` 之前，使用 `size()` 检查堆栈的大小。

`queue` 是另一个适配器，与栈的行为略有不同。队列背后的逻辑是，它首先返回第一个插入的元素：先进先出（FIFO）原则。



队列中插入和检索操作的正式名称是 enqueue 和 dequeue。queue 保持数据一致性的方法，并提供 push() 和 pop() 函数。要访问队列的第一个和最后一个元素，应该使用 front() 和 back()，两者都返回对元素的引用。下面是一个简单的用法示例：

```

1 #include <queue>
2 int main()
3 {
4     std::queue<char> q;
5     q.push('a');
6     q.push('b');
7     q.push('c');
8     std::cout << q.front(); // prints 'a'
9     std::cout << q.back(); // prints 'c'
10    q.pop();
11    std::cout << q.front(); // prints 'b'
12 }
```

要正确使用各种容器和适配器时，了解它们是很有必要的。在为各种问题选择合适的容器方面，并没有什么灵丹妙药。许多编译器使用堆栈来解析代码表达式，例如：很容易使用栈，验证下面表达式中的圆括号：

```

1 int r = (a + b) + (((x * y) - (a / b)) / 4);
```

试着练习一下。编写一个小程序，使用堆栈验证前面的表达式。

另一个容器适配器是 std::priority_queue。优先队列通常采用均衡的、基于节点的数据结构，比如：大堆或小堆。我们将在本章的末尾研究树和图，以及了解优先队列是如何工作的。

迭代容器

不可迭代的容器就像不能驾驶的汽车。毕竟，容器是项的集合，循环遍历容器元素的一种常用方法是使用普通的 for 循环：

```

1 std::vector<int> vec{1, 2, 3, 4, 5};
2 for (int ix = 0; ix < vec.size(); ++ix) {
3     std::cout << vec[ix];
4 }
```

容器为元素访问提供了一组不同的操作。例如，vector 提供操作符 []，而 list 没有。std::list 有 front() 和 back() 方法，它们分别返回第一个和最后一个元素。如前所述，std::vector 还提供了 at() 和运算符 []。

这意味着不能使用前面的循环来迭代列表元素。但可以使用基于范围的 for 循环遍历列表（和 vector），如下所示：

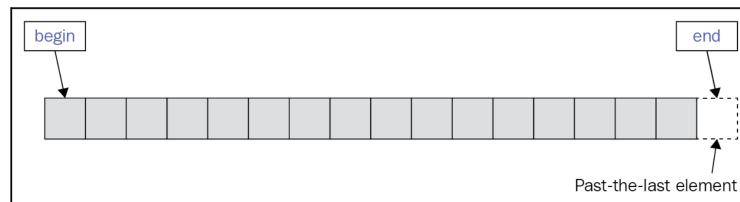
```
1 std::list<double> lst {1.1, 2.2, 3.3, 4.2};  
2 for (auto& elem : lst) {  
3     std::cout << elem;  
4 }
```

看起来可能令人困惑，这个技巧隐藏在基于范围的实现中。它使用 std::begin() 函数检索指向容器第一个元素的迭代器。

迭代器是指向容器元素的对象，可以根据容器的物理结构推进到下一个元素。下面的代码声明了一个 vector 迭代器，并使用指向 vector 开头的迭代器对其进行初始化：

```
1 std::vector<int> vec{1, 2, 3, 4};  
2 std::vector<int>::iterator it{vec.begin()};
```

容器提供了两个成员函数 begin() 和 end()，分别返回指向容器开头和结尾的迭代器。下面的图表显示了如何处理容器的开头和结尾：



前面使用基于范围的 for 遍历列表元素的代码，可以认为类似于以下代码：

```
1 auto it_begin = std::begin(lst);  
2 auto it_end = std::end(lst);  
3 for ( ; it_begin != it_end; ++it_begin) {  
4     std::cout << *it_begin;  
5 }
```

请注意前面代码中使用的 * 操作符，该操作符通过迭代器访问底层元素。我们认为迭代器是指向容器元素的指针。



std::begin() 和 std::end() 函数通常分别调用容器的 begin() 和 end() 方法，也适用于常规数组。

容器迭代器清楚地知道如何使用容器元素，例如：向前移动 vector 迭代器将其移动到数组的下一个槽位，而向前移动 list 迭代器则使用相应的指针将其移动到下一个节点，如下面的代码所示：

```
1 std::vector<int> vec;  
2 vec.push_back(4);  
3 vec.push_back(2);  
4 std::vector<int>::iterator it = vec.begin();  
5 std::cout << *it; // 4  
6 it++;
```

```
7 std::cout << *it; // 2
8
9 std::list<int> lst;
10 lst.push_back(4);
11 lst.push_back(2);
12 std::list<int>::iterator lit = lst.begin();
13 std::cout << *lit; // 4
14 lit++;
15 std::cout << *lit; // 2
```

每个容器都有自己的迭代器实现，这就是为什么 list 迭代器和 vector 迭代器有相同的接口，但行为不同。迭代器的行为由它的类别定义，例如：vector 的迭代器是随机访问迭代器，可以使用该迭代器随机访问任何元素。下面的代码通过 vector 的迭代器向 vector 的第 4 个元素添加 3，如下所示：

```
1 auto it = vec.begin();
2 std::cout << *(it + 3);
```

STL 中有六种类型的迭代器：

- 输入
- 输出（与输入相同，但支持写访问）
- 向前
- 双向
- 随机访问
- 连续性

输入迭代器提供读访问（通过调用 * 操作符），并允许使用前缀和后缀自增操作符转发迭代器位置。输入迭代器不支持多次传递，只能使用迭代器在容器上迭代一次。另一方面，前向迭代器支持多次传递，支持多次遍历可以通过迭代器多次读取元素的值。

输出迭代器不提供对元素的访问，但它允许给元素赋新值。具有多重传递特性的输入迭代器和输出迭代器的组合构成了前向迭代器。然而，前向迭代器只支持自增操作，而双向迭代器支持将迭代器移动到任何位置。它们都支持递减操作，例如：std::list 支持双向迭代器。

最后，随机访问迭代器允许通过在迭代器上加/减一个数字来跳转元素。迭代器将跳转到算术操作指定的位置。vector 提供了随机访问迭代器。

每个类别都定义了一组可应用于迭代器的操作，例如：可以使用输入迭代器读取元素的值，并通过递增迭代器前进到下一个元素。另一方面，随机访问迭代器允许迭代器对任意值递增或递减，读取和写入元素的值等。

本节描述的所有特性的组合都属于连续迭代器类别，它也期望容器是连续的。这意味着容器元素需要紧挨着另一个元素。连续容器的经典例子是 std::array。

像 distance() 这样的函数使用迭代器的信息来实现最快的执行结果。例如，两个双向迭代器之间的 distance() 函数的执行时间是线性的，而用于随机访问迭代器相同函数的执行时间是常量。

下面的伪代码演示了一个示例实现：

```
1 template <typename Iter>
2 std::size_type distance(Iter first, Iter second) {
```

```

3   if (Iter is a random_access_iterator) {
4     return second - first;
5   }
6   std::size_type count = 0;
7   for ( ; first != last; ++count, first++) {}
8   return count;
9 }
```

尽管前面示例中显示的伪代码工作良好，但我们应该考虑在运行时检查迭代器的类别不是一个选项。它是在编译时定义的，因此需要使用模板特化来为随机访问迭代器生成 `distance()` 函数。更好的解决方案是使用 `std::is_same` 类型特征，定义在 `<type_traits>` 中：

```

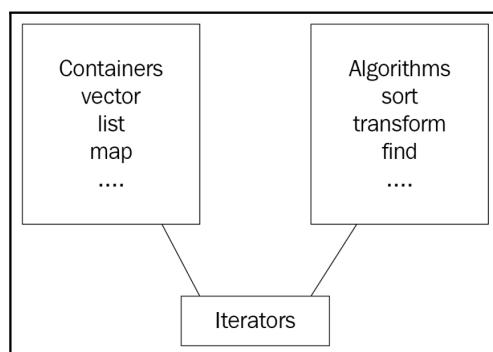
1 #include <iterator>
2 #include <type_traits>
3
4 template <typename Iter>
5 typename std::iterator_traits<Iter>::difference_type distance(Iter first,
6 Iter last)
7 {
8   using category = std::iterator_traits<Iter>::iterator_category;
9   if constexpr (std::is_same_v<category, std::random_access_iterator_tag>)
10  {
11    return last - first;
12  }
13  typename std::iterator_traits<Iter>::difference_type count;
14  for ( ; first != last; ++count, first++) {}
15  return count;
16 }
```

`std::is_same_v` 是 `std::is_same` 的助手模板，定义如下：

```

1 template <class T, class U>
2 inline constexpr bool is_same_v = is_same<T, U>::value;
```

迭代器最重要的特性是，减少了容器和算法之间的耦合：



STL 基于这三个概念：容器、算法和迭代器。虽然 `vector`、`list` 或任何其他容器不同，但它们都有相同的目的：存储数据。

另一方面，算法是处理数据的函数，大部分时间都在收集数据。算法定义通常表示一种通用的方式，指定应该采取的步骤来处理容器元素。例如，排序算法按升序或降序对容器元素进行排序。

vector 是连续容器，而 list 是基于节点的容器。对它们进行排序需要对特定容器的物理结构有更深入的了解。要正确地对向量进行排序，应该为它实现一个单独的排序函数。

迭代器将这种实现的多样性带到通用级别。它们为库设计人员提供了实现一个排序函数的能力，该函数将只处理迭代器，从容器类型中抽象出来。STL 中使用 sort() 算法（在 <algorithm> 中定义）处理迭代器，我们可以用同一个函数对向量和列表进行排序：

```
1 #include <algorithm>
2 #include <vector>
3 #include <list>
4 ...
5 std::vector<int> vec;
6 // insert elements into the vector
7 std::list<int> lst;
8 // insert elements into the list
9
10 std::sort(vec.begin(), vec.end());
11 std::sort(lst.begin(), lst.end());
```

本节中描述的迭代器现在认为是之前版本遗留特性，C++20 引入了一种新的基于概念的迭代器。

概念和迭代器

C++20 的主要特性之一是引入概念。除了概念之外，C++20 还提供了基于概念的新迭代器。虽然本章讨论的迭代器现在认为是遗留特性，但已经用它们编写了许多代码。这就是我们在继续讨论新的迭代器概念之前，首先介绍它们的原因。现在，让我们来看看，什么是概念，以及如何使用。

理解概念

抽象在计算机编程中是必不可少，在第 3 章介绍了类，作为一种将数据和操作表示为抽象实体的方法。在第 4 章中，我们深入探讨了模板，并了解了如何通过对各种聚合类型重用模板，从而使类变得更加灵活。模板不仅提供特定类型的抽象，还可以去除实体类型和聚合类型之间的耦合。以 std::vector 为例，它提供了通用接口来存储和操作对象集合。我们声明三个不同的 vector，包含三种不同类型的对象，如下所示：

```
1 std::vector<int> ivec;
2 std::vector<Person> persons;
3 std::vector<std::vector<double>> float_matrix;
```

如果不是模板，我们必须对前面的代码做如下操作：

```
1 std::int_vector ivec;
2 std::custom_vector persons; // supposing the custom_vector stores void*
3 std::double_vector_vector float_matrix;
```

尽管这段的代码无法可接受，但应该认可模板是泛型编程的基础这一事实。概念为泛型编程引入了更多的灵活性。现在可以对模板参数设置限制，检查约束，并在编译时发现不一致的行为。模板类声明如下：

```
1 template <typename T>
2 class Wallet
3 {
4     // the body of the class using the T type
5 };
```

请注意前面代码块中的 `typename` 关键字。概念允许用描述模板形参的类型，来替换模板形参。假设我们想让 `Wallet` 使用可以添加的类型，应该是可添加的。下面是如何使用概念来帮助代码实现的：

```
1 template <addable T>
2 class Wallet
3 {
4     // the body of the class using addable T's
5 };
```

现在可以通过可添加的类型来创建 `Wallet` 的实例。当类型不满足约束时，编译器将抛出错误。下面的代码片段声明了两个 `Wallet` 对象：

```
1 class Book
2 {
3     // doesn't have an operator+
4     // the body is omitted for brevity
5 };
6 constexpr bool operator+(const Money& a, const Money& b) {
7     return Money{a.value_ + b.value_};
8 }
9
10 class Money
11 {
12     friend constexpr bool operator+(const Money&, const Money&);
13     // code omitted for brevity
14 private:
15     double value_;
16 };
17
18 Wallet<Money> w; // works fine
19 Wallet<Book> g; // compile error
```

`Book` 类没有加法操作符，因此由于模板形参类型的限制，`g` 的构造将失败。

概念的声明是使用 `concept` 关键字完成的，其形式如下：

```
1 template <parameter-list>
2 concept name-of-the-concept = constraint-expression;
```

概念也使用模板声明，我们可以将它们称为描述类型的类型。概念在很大程度上依赖于约束。约束是为模板参数指定要求的一种方式，而概念是一组约束。下面是我们如何实现前面的可添加概念：

```
1 template <typename T>
2 concept addable = requires (T obj) { obj + obj; }
```

标准概念定义在 `<concepts>` 头文件中。

我们还可以通过要求新概念支持其他概念来将几个概念组合成一个概念。使用 `&&` 操作符，让看看迭代器是如何利用概念的，并举例说明结合了其他概念的可递增迭代器概念。

C++20 中使用迭代器

介绍了概念之后，看下迭代器是怎么充分利用概念的。迭代器及其类别现在认为是旧版本遗留的，因为从 C++20 开始，我们使用了概念迭代器，比如 `readable`(通过应用 `*` 操作符指定类型可读) 和 `writable`(指定可将值写入迭代器引用的对象)。让我们看看如何在 `<iterator>` 头文件中定义 `incrementable`:

```
1 template <typename T>
2 concept incrementable = std::regular<T> && std::weakly_incrementable<T>
3     && requires (T t) { {t++} -> std::same_as<T>; };
```

因此，可递增的概念要求类型为 `std::regular`。这意味着它在默认情况下应该是可构造的，并具有复制构造函数和 `==` 操作符。除此之外，`incrementable` 概念要求类型为 `weakly_incrementable`，这意味着该类型支持自增前和自增后操作符，但不要求类型相等可比。这就是为什么可增量联接 `std::regular` 要求类型有可比性。最后，添加要求约束指向这样一个事实，即在递增之后类型不应该改变，它应该与之前的类型相同。`std::same_as` 表示为一个概念(定义在 `<concepts>` 中)，但在以前的版本中，我们使用 `<type_traits>` 中定义的 `std::is_same`。它们做的事情相同，但是 C++17 版本——`std::is same v`——带有后缀。

因此，我们现在不再引用迭代器类别，而是引用迭代器概念。除了我们前面介绍的概念外，还需要考虑以下概念：

- `input_iterator` 指定该类型允许读取其引用的值，并且前后可递增。
- `output_iterator` 指定该类型的值可以写入，并且该类型既可递增递增。
- `input_or_output_iterator` 不必要的长名称，指定类型是可递增的，可以解引用。
- `forward_iterator` 指定类型为 `input_iterator`，另外还支持相等比较和多传递。
- `bidirectional_iterator` 指定该类型支持 `forward_iterator`，另外还支持向后移动。
- `random_access_iterator` 指定类型为双向迭代器，支持在常量时间的随机访问和下标索引。
- `continuous_iterator` 指定类型为 `random_access_iterator`，指向内存中连续的元素。

它们几乎重复我们前面讨论过的迭代器，现在可以在声明模板形参时使用它们，以便编译器处理其余的工作。

主流算法

如前所述，算法是接受输入、处理输入并返回输出的函数。通常，STL 中的算法属于一个处理数据集合的函数。数据集合以容器形式呈现，如 std::vector、std::list 等。

开发者的日常工作中，选择有效的算法是常规任务。例如，使用二分搜索算法搜索排序向量，将比使用顺序搜索高效得多。为了比较算法的效率，考虑到算法的速度与输入数据大小有关，意味着我们不应该通过将两种算法应用到有 10 个或 100 个元素的容器上来进行实际的比较。

当应用到足够大的容器（拥有 100 万条甚至 10 亿条记录）时，算法的实际差异就会显现出来。衡量一个算法的效率也被称为验证其复杂性，可能遇到过 $O(n)$ 种算法或者 $O(\log n)$ 种算法。 $O()$ 函数（发音为 big-oh）定义了算法的复杂性。

让我们来看看这些搜索算法，并比较它们的复杂性。

搜索

容器中搜索元素是一项常见的任务。让我们实现 vector 元素的顺序搜索：

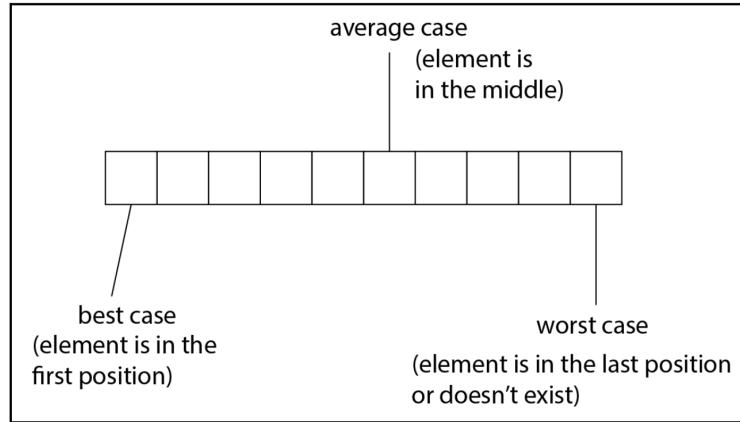
```
1 template <typename T>
2 int search(const std::vector<T>& vec, const T& item)
3 {
4     for (int ix = 0; ix < vec.size(); ++ix) {
5         if (vec[ix] == item) {
6             return ix;
7         }
8     }
9     return -1; // not found
10 }
```

这是一个简单的算法，遍历 vector 并返回元素所在的索引，该索引等于作为搜索键传递的值。之所以称其为顺序搜索，是因为它按顺序扫描 vector 元素。它的复杂度是线性的： $O(n)$ 。为了测量它，我们应该以某种方式定义算法找到结果所需要的次数。假设向量包含 n 个元素，下面的代码在搜索函数的每一行上包含一个关于其操作的注释：

```
1 template <typename T>
2 int search(const std::vector<T>& vec, const T& item)
3 {
4     for (int ix = 0; // 1 copy
5          ix < vec.size(); // n + 1 comparisons
6          ++ix) // n + 1 increments
7     {
8         if (vec[ix] == item) { // n comparisons
9             return ix; // 1 copy
10        }
11    }
12    return -1; // 1 copy
13 }
```

我们有三种复制操作， $n + 1$ 和 n （即 $2n + 1$ ）比较，以及 $n + 1$ 递增操作。如果所需的元素位于 vector 的第一个位置，该怎么办？在这种情况下，只比较 vector 的第一个元素，然后从函数返回。

然而，这并不意味着我们的算法非常高效，只需要一步就能完成任务。为了衡量算法的复杂性，我们应该考虑最坏的情况：所需的元素要么不存在于 vector 中，或查找的元素位于 vector 的最后一个位置。下面的图表显示了我们将要查找的元素的三种场景：



我们应该考虑最坏的情况，因为它也涵盖了所有其他情况。如果在最坏的情况下定义一个算法的复杂度，我们可以肯定其他情况不会比这更慢。

要找出一个算法的复杂度，我们应该找到操作的数量和输入的大小之间的联系。本例中，输入的大小就是容器的长度。我们用 A 表示拷贝，用 C 表示比较，用 I 表示递增操作，这样我们就有 $3A + (2n + 1)C + (n + 1)I$ 个操作。算法的复杂度定义如下：

$$O(3A + (2n + 1)C + (n + 1)I)$$

这可以通过以下方式简化：

- $O(3A + (2n + 1)C + (n + 1)I) =$
- $O(3A + 2nC + C + nI + I) =$
- $O(n(2C + I) + (3A + C + I)) =$
- $O(n(2C + I))$

最后， $O()$ 的属性要去掉常系数和较小的成员，因为实际算法的复杂性只与输入的大小 n 有关，所以得到最终的复杂性等于 $O(n)$ 。换句话说，顺序搜索算法的时间复杂度是线性的。

如前所述，STL 的本质是通过迭代器连接容器和算法。这就是为什么顺序搜索实现不兼容 STL：因为它对输入参数有严格的限制。为了使它泛型，我们应该考虑使用迭代器来实现它。要覆盖更广泛的容器类型，请使用前向迭代器。下面的代码使用 Iter 类型的操作符，假设它是前向迭代器：

```

1 template <typename Iter, typename T>
2 int search(Iter first, Iter last, const T& elem)
3 {
4     for (std::size_t count = 0; first != last; first++, ++count) {
5         if (*first == elem) return count;
6     }
7     return -1;
8 }
9 ...
10 std::vector<int> vec{4, 5, 6, 7, 8};
11 std::list<double> lst{1.1, 2.2, 3.3, 4.4};
12 std::cout << search(vec.begin(), vec.end(), 5);

```

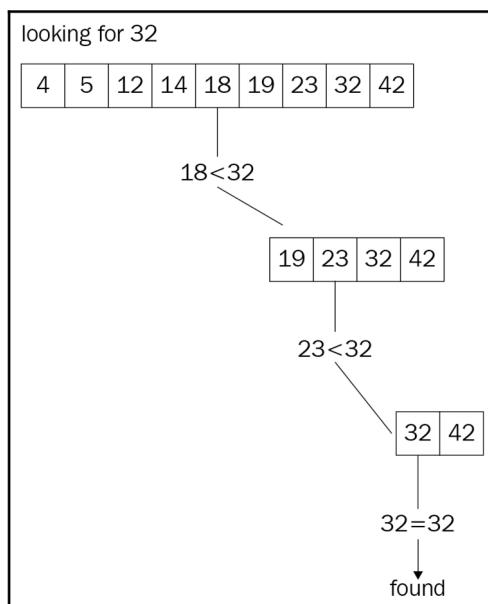
```
13 std::cout << search(lst.begin(), lst.end(), 5.5);
```

实际上，任何类型的迭代器都可以传递给 `search()` 函数。通过对迭代器本身应用操作，确保使用前向迭代器。我们只使用前向迭代器支持的自增（向前移动）、读（`*` 操作符）和严格比较（`==` 和 `!=`）。

二分搜索

二分搜索算法很容易解释。首先，寻找向量的中间元素，并与搜索键进行比较，如果它等，那么算法就完成了：它返回索引。否则，如果搜索键小于中间的元素，它将继续到向量的左边。如果搜索键大于中间元素，则算法继续搜索右边的子向量。

为了使二分搜索对一个向量正确工作，它应该排序。二分查找的本质是将搜索键与向量元素进行比较，然后继续查找左边或右边的子向量，与向量的中间元素相比，每个子向量都包含一个更小或更大的元素。看看下面的图表，它描述了二进制搜索算法的运行：



二分搜索算法有一个优雅的递归实现（尽管使用迭代实现更好）：

```
1 template <typename T>
2 std::size_t binsearch(const std::vector<T>& vec, const T& item, int start,
3 int end)
4 {
5     if (start > end) return -1;
6     int mid = start + (end - start) / 2;
7     if (vec[mid] == item) {
8         return mid; // found
9     }
10    if (vec[mid] > item) {
11        return binsearch(vec, item, start, mid - 1);
12    }
13    return binsearch(vec, item, mid + 1, end);
14 }
```

注意中间元素的计算。而不是 $(\text{start} + \text{end}) / 2$;，我们使用 $\text{start} + (\text{end} - \text{start}) / 2$; 技术只是为了避免二分搜索实现中的 bug(假设我们没有留下其他 bug)。关键是，对于较大的 start 和 end 值，它们的和 $(\text{start} + \text{end})$ 将产生整数溢出，这将使程序在某个点崩溃。

现在来算一下二分搜索的复杂度。执行的每一步中，源数组都减半，所以在下一步中处理它的大小一半，最坏的情况是对向量进行除法，直到剩下一个元素或一个元素都没有。为了求出算法中的步数，我们应该求出与向量大小有关的除法次数。如果向量有 10 个元素，则除以它，得到一个包含 5 个元素的子向量。再除以一次，就得到了两个元素的子向量，最后再除以一次，就得到了一个元素。所以，对于 10 个元素的向量，划分的次数是 3。对于有 n 个元素的向量，划分的次数是 $\log(n)$ ，因为在每一步上，n 变成 $n/2$ ，然后变成 $n/4$ ，以此类推。二分搜索的复杂度是 $O(\log n)$ (即对数)。

STL 算法定义在 `<algorithm>` 头文件中，二分搜索的实现驻留在那里。如果元素在容器中存在，则 STL 实现返回 true。看看它的原型：

```
1 template <typename Iter, typename T>
2 bool binary_search(Iter start, Iter end, const T& elem);
```

STL 算法不直接与容器一起工作，而是与迭代器一起。这允许我们从特定的容器进行抽象，并对所有支持前向迭代器的容器使用 `binary_search()`。下面的例子调用 `binary_search()` 函数来处理 `vector` 和 `list`：

```
1 #include <vector>
2 #include <list>
3 #include <algorithm>
4 ...
5 std::vector<int> vec{1, 2, 3, 4, 5};
6 std::list<int> lst{1, 2, 3, 4};
7 binary_search(vec.begin(), vec.end(), 8);
8 binary_search(lst.begin(), lst.end(), 3);
```

`binary_search()` 检查迭代器的类别，在随机访问迭代器的情况下，它使用二分搜索算法的全部功能（否则，它将退回到顺序搜索）。

排序

二分搜索算法只适用于已排序的容器。排序对于计算机程序员来说是一项古老的任务，现在很少编写自己的排序算法实现。有的开发者可能多次使用过 `std::sort()`，而根本不了解它的实现。基本上，排序算法接受一个集合作为输入，并返回一个新的排序过的集合（按照算法用户定义的顺序）。

在许多排序算法中，最流行的（甚至是最快的）是快速排序。任何排序算法的基本思想都是找到更小（或更大）的元素，并与更大（或更小）的元素交换它们，直到整个集合排序完毕。例如，选择排序逻辑上将集合分为两部分，排序和未排序，排序的子数组最初是空的，如下所示：

sorted	unsorted										
	7 11 -8 3 42 74 66 26 97 -18 0 3										

该算法开始在未排序子数组中寻找最小的元素，并通过与未排序子数组的第一个元素交换将其放入已排序子数组中。每一步之后，已排序子数组的长度增加 1，而未排序子数组的长度减少如下：

sorted	unsorted										
-18	11 -8 3 42 74 66 26 97 7 0 3										

这个过程将继续，直到未排序的子数组变为空。

STL 提供了 `std::sort()` 函数，接受两个随机访问迭代器：

```

1 #include <vector>
2 #include <algorithm>
3 ...
4 std::vector<int> vec{4, 7, -1, 2, 0, 5};
5 std::sort(vec.begin(), vec.end());
6 // -1, 0, 2, 4, 5, 7

```

`sort` 函数不能应用于 `std::list`，因为它不支持随机访问迭代器，而 `list` 应该调用 `sort()` 成员函数。尽管这与 STL 拥有通用函数的想法相矛盾，但这样做是为了提高效率。

`sort()` 函数有第三个形参：一个比较函数，可用于比较容器元素。假设将 `Product` 对象存储在 `vector` 中：

```

1 struct Product
2 {
3     int price;
4     bool available;
5     std::string title;
6 };
7
8 std::vector<Product> products;
9 products.push_back({5, false, "Product 1"});
10 products.push_back({12, true, "Product 2"});

```

要对容器进行正确的排序，容器中的元素必须支持小于操作符。我们应该为自定义类型定义相应的运算符。但是，如果为自定义类型创建单独的比较器函数，则可以省略运算符定义，如下面的代码块所示：

```

1 class ProductComparator
2 {

```

```
3 public:
4     bool operator()(const Product& a, const Product& b) {
5         return a.price > b.price;
6     }
7 };
```

将 ProductComparator 传递给 std::sort() 函数，允许它比较 vector 元素，而无需深入了解元素的类型细节，如下所示：

```
1 std::sort(products.begin(), products.end(), ProductComparator{});
```

虽然这是一种很好的技术，但使用 lambda 函数会更优雅，lambda 函数是一种匿名函数，非常适合前一种情况。下面是我们使用 lambda 对 sort 重新实现：

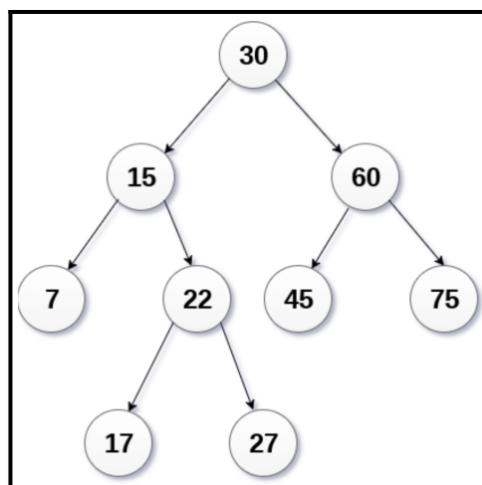
```
1 std::sort(products.begin(), products.end(),
2 [](const Product& a, const Product& b) { return a.price > b.price; })
```

上面的代码允许省略 ProductComparator 的声明。

搜索树和图

二分搜索算法和排序算法结合在一起，就产生了使用一个容器来默认保存条目排序的想法，其中一个容器 std::set 就是基于平衡树的。讨论平衡树本身之前，我们先看看二叉搜索树，它是快速查找的最佳选择。

二叉搜索树的思想是一个节点左子树的值小于该节点的值。相比之下，一个节点的右子树的值大于该节点的值。这里有一个二叉搜索树的例子：



正如图表中看到的，值为 15 的元素驻留在左边的子树中，因为它小于 30(根元素)。另一方面，值为 60 的元素驻留在右边的子树中，因为它比根元素大。同样的逻辑也适用于其余的树元素。

二叉树节点表示为一个结构体，其中包含项目和指向每个子节点的两个指针。下面是一个表示树节点的示例代码：

```
1 template <typename T>
2 struct tree_node {
3 }
```

```
4   T item;
5   tree_node<T>* left;
6   tree_node<T>* right;
7 };
```

一个完全平衡的二叉搜索树中，搜索、插入或删除一个元素需要 $O(\log n)$ 。STL 没有为树提供单独的容器，但是它有类似的基于树实现的容器。例如，`std::set` 容器基于一棵按排序顺序存储元素的平衡树：

```
1 #include <set>
2 ...
3 std::set<int> s{1, 5, 2, 4, 4, 4, 3};
4 // s has {1, 2, 3, 4, 5}
```

`std::map` 也是基于平衡树的，但它提供了一个容器，将一个键映射到某个值，如下所示：

```
1 #include <map>
2 ...
3 std::map<int, std::string> numbers;
4 numbers[3] = "three";
5 numbers[4] = "four";
6 ...
```

如上述代码所示，`map` 将数字映射为字符串。因此，当我们告诉 `map` 将值 3 存储为键值，将字符串“three”存储为值时，它会在其内部树中添加一个新节点，键值为 3，值为“three”。

`set` 和 `map` 操作是对数的，这使得它在大多数情况下是一个非常高效的数据结构。然而，接下来会出现一个更高效的数据结构。

哈希表

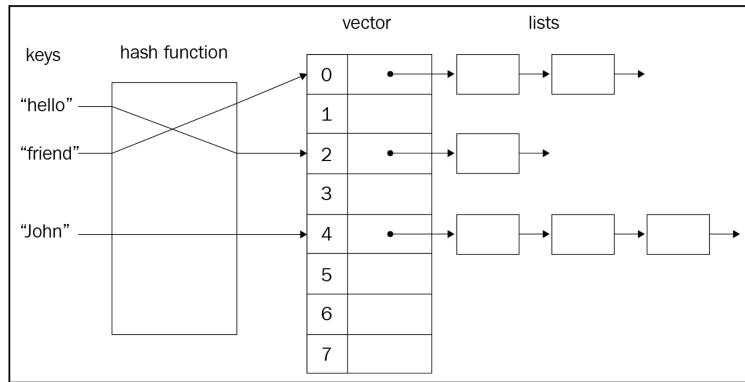
哈希表是最快的数据结构，基于矢量索引的简单思想。想象一个包含指向列表指针的大向量：

```
1 std::vector<std::list<T>> hash_table;
```

访问 `vector` 元素需要常量时间。这是向量的主要优势。哈希表允许我们使用任何类型作为容器的键。哈希表的基本思想是使用经过良好管理的哈希函数，该函数将为输入键生成唯一的索引。例如，当使用字符串作为哈希表的键时，哈希表使用函数生成哈希值作为基础向量的索引值：

```
1 template <typename T>
2 int hash(const T& key)
3 {
4     // generate and return and efficient
5     // hash value from key based on the key's type
6 }
7 template <typename T, typename U>
8 void insert_into_hashtable(const T& key, const U& value)
9 {
10    int index = hash(key);
11    hash_table[index].push_back(value); // insert into the list
12 }
```

下面是如何使用哈希表:



访问哈希表需要常量时间，因为它基于 vector 操作。虽然可能有不同的键会导致相同的散列值，但是这些冲突可以通过使用一个值列表作为 vector 元素来解决（如上图所示）。

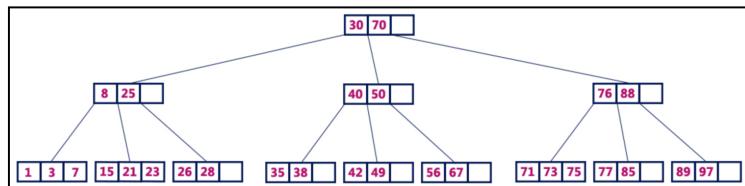
STL 支持一个名为 std::unordered_map 的哈希表:

```
1 #include <unordered_map>
2 ...
3 std::unordered_map<std::string, std::string> hashtable;
4 hashtable["key1"] = "value 1";
5 hashtable["key2"] = "value 2";
6 ...
```

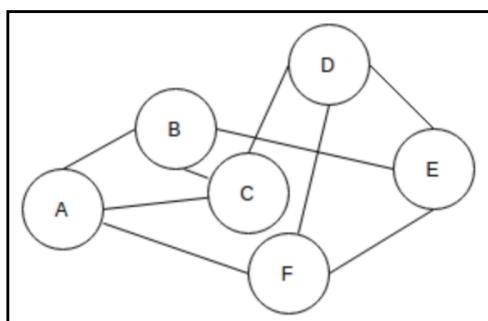
要为提供的键生成散列值，函数 std::unordered_map 使用 <functional> 头文件中定义的 std::hash() 函数。可以为哈希函数指定自定义实现。std::unordered_map 的第三个模板参数是散列函数，默认为 std::hash。

图

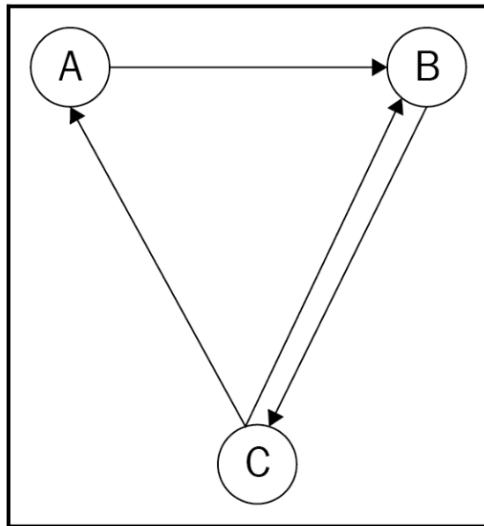
二叉搜索树的平衡特性基于多种搜索索引实现。例如，数据库系统使用一种称为 B-树的平衡树来索引表。B-树不是二叉树，但遵循相同的平衡逻辑，如下图所示：



另一方面，图表示节点间的连接:

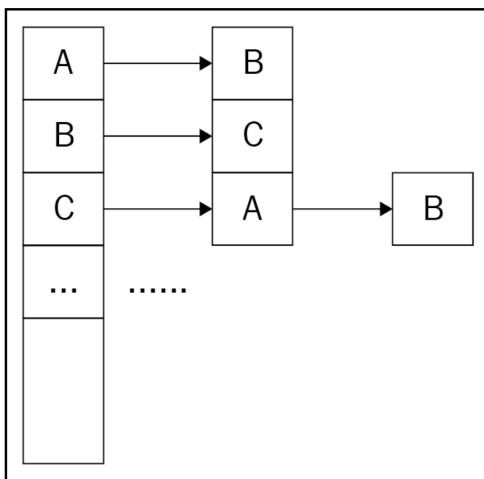


假设正在建立一个最终会在市场上击败 Facebook 的社交网络。社交网络中的用户可以互相关注，可以用图表来表示。例如，如果 A 遵循 B, B 遵循 C，并且 C 同时遵循 B 和 A，那么我们可以将这些关系表示为如下图：



一个节点称为图中的一个顶点。两个节点之间的连接称为边。实际上并没有一个固定的图表示，所以我们应该从几个图中选择一个。想想我们的社交网络——我们如何表示用户 A 关注用户 B 的信息？

最好的选择之一是使用哈希表，可以将用户与他们关注的所有用户进行映射：



图实现变成了一个混合容器：

```
1 #include <list>
2 #include <unordered_map>
3 template <typename T>
4 class Graph
5 {
6 public:
7     Graph();
8     ~Graph();
```

```

9 // copy, move constructors and assignment operators omitted for brevity
10
11 public:
12     void insert_edge(const T& source, const T& target);
13     void remove_edge(const T& source, const T& target);
14
15     bool connected(const T& source, const T& target);
16
17 private:
18     std::unordered_map<T, std::list<T>> hashtable_;
19 };

```

为了创建一个与 STL 兼容的容器，为图添加一个迭代器。虽然迭代一个图不是个好主意，但是添加一个迭代器也不是个坏主意。

字符串

字符串类似于 vector：它们存储字符，具有迭代器，还是容器。然而，字符串则有些不同。下面的图表将字符串“hello, C++”描述为一个以特殊 \0 字符结尾的字符数组：

h	e	l	l	o	,		C	+	+	\0
0	1	2	3	4	5	6	7	8	9	10

特殊的 \0 字符（也称为空字符）用作字符串终止符。编译器一个接一个地读取字符，直到遇到空字符。

string 对象的实现方式与本章开头 vector 对象的实现方式相同：

```

1 class my_string
2 {
3 public:
4     my_string();
5     // code omitted for brevity
6 public:
7     void insert(char ch);
8     // code omitted for brevity
9 private:
10    char* buffer_;
11    int size_;
12    int capacity_;
13 };

```

C++ 有强大的 std::string 类，它提供了一组可以使用的函数。除了 std::string 的成员函数之外，<algorithm> 中定义的算法也适用于字符串。

总结

数据结构和算法是开发高效软件的关键。通过理解和利用本章讨论的数据结构，你将掌握 C++20 的全部功能，并且使你的程序运行得更快。有很强的问题解决能力的开发者在市场上更

受欢迎，这不是什么秘密。解决问题的技巧首先要通过对基本算法和数据结构的深刻理解来获得。正如在本章中看到的，在搜索任务中使用二分搜索算法使代码运行得比顺序搜索算法快得多。高效的软件可以节省时间并提供更好的用户体验，这最终会使你的软件成为现有软件的杰出替代品。

本章中，我们讨论了基本的数据结构及其区别，通过问题分析学会了使用它们。例如，由于 list 元素访问操作的复杂性，在需要随机查找的问题中应用 list 是不合适的。在这种情况下，使用动态增长的向量更合适，因为它的元素访问时间是常量。相反，在需要在容器前端进行快速插入的问题中，使用 vector 的成本比使用 list 的成本更高。

本章还介绍了它们效率的度量算法和方法。我们比较了几个问题，以应用更好的算法来更有效地解决它们。

下一章中，我们将讨论 C++ 中的函数式编程。学习了 STL 的要点之后，我们现在要在容器上应用函数式编程技术了。

问题

1. 描述将元素插入动态增长向量的过程。
2. 在链表的前面插入元素和在 vector 的前面插入元素有什么区别？
3. 实现一个混合数据结构，将其元素存储在 vector 和 list 中。对于每个操作，选择具有最快实现该操作的底层数据结构。
4. 如果我们按递增顺序插入 100 个元素，那么二叉搜索树会是什么样子呢？
5. 选择排序和插入排序算法有什么区别？
6. 实现本章中描述的排序算法，称为计数排序。

扩展阅读

有关更多信息，请参阅以下参考资料：

- Programming Pearls by Jon Bentley, available from <https://www.amazon.com/Programming-Pearls-2nd-Jon-Bentley/dp/0201657880/>
- Data Abstraction and Problem Solving Using C++: Walls and Mirrors by Frank Carrano, and Timothy Henry, available from <https://www.amazon.com/Data-Abstraction-Problem-Solving-Mirrors/dp/0134463978/>
- Introduction to Algorithms by Cormen, Leiserson, Rivest, and Stein, available from <https://www.amazon.com/Introduction-Algorithms-3rd-MIT-Press/dp/0262033844/>
- C++ Data Structures and Algorithms by Wisnu Anggoro, available from <https://www.packtpub.com/app-development/c-data-structures-and-algorithms>

第 7 章：函数式编程

面向对象编程 (OOP) 为我们提供了一种思考方式，可以用类及关系来描述现实世界。函数式编程是另一种完全不同的编程范式，它允许我们专注于函数结构，而不是代码的物理结构。学习和使用函数式编程很有用。首先，它是一种新的范式，可使你以非常不同的方式思考。解决问题需要灵活的思维，执着于单一范式的人往往会为任何问题提供类似的解决方案，而最优雅的解决方案需要更好的方法。掌握函数式编程为开发人员提供了一种新技能，可以帮助他们提供更好的问题解决方案。其次，使用函数式编程减少了软件中的错误数量。函数式编程采用独特方法的最大原因之一是：将程序分解为函数，每个函数都不修改数据的状态。

本章中，我们将讨论函数式编程的基本模块以及范围。C++20 中引入的 range 为我们提供了一种组合算法的方法，这样就可以处理数据集合。组合算法以便我们可以将它们按顺序应用到数据集合中，这是函数式编程的核心。

本章中，我们将了解以下内容：

- 函数式编程简介
- range 库的介绍
- 纯函数
- 高阶函数
- 深入地研究递归
- C++ 中（函数式）的元编程

编译器要求

g++ 编译器需要添加编译选项 `-std=c++2a` 来编译本章的代码。可以从这里获取本章的源码文件：<https://github.com/PacktPublishing/Expert-CPP>

函数式编程

正如我们前面提到的，函数式编程是一种编程范式。构建程序时，可以将范式视为一种思考方式。C++ 是一种多范式语言，可以使用它在过程范式中开发程序，即一个接一个地执行语句。第 3 章中，我们讨论了面向对象方法，涉及到将一个复杂的系统分解为相互通信的对象。另一方面，函数式编程鼓励我们将系统分解为函数，而不是对象。表达式受一些输入，并一个产生输出，输出这可以用作另一个函数的输入。乍一看，似乎很简单，但是函数式编程包含了一些最初感觉很难掌握的规则。然而，当你做到这一点时，大脑就会开启一种新的思维方式——功能性思维方式。

为了更清楚地说明这一点，我们从一个示例开始演示函数式编程的原理。假设得到了一个整数列表，需要计算其中的偶数个数。唯一的问题是我们应该分别计算所有向量中的偶数，并生成一个新向量，其中包含每个输入向量的计算结果。

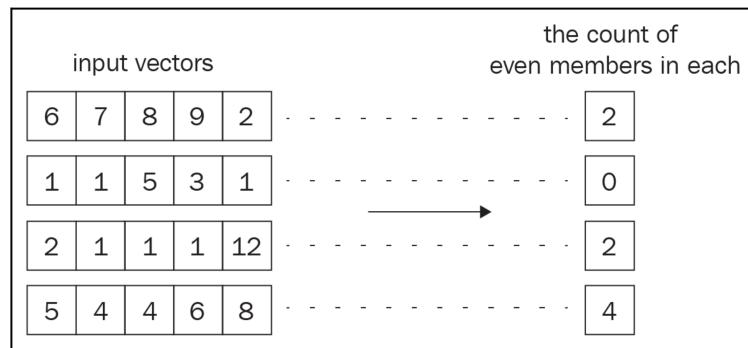
输入是一个矩阵，即向量中的向量。在 C++ 中，最简单的表达方式是使用以下类型：

```
1 std :: vector<std :: vector<int>>
```

我们可以通过使用类型别名来进一步简化前面的代码，如下所示：

```
1 using IntMatrix = std :: vector<std :: vector<int>>;
```

下面是对这个问题的说明。我们有一串包含整数的向量，因此得到一个包含偶数计数的 vector:



看看下面的函数。它以一个由整数向量组成的 vector(也称为矩阵) 作为它的参数。该函数计算偶数的数目:

```
1 std::vector<int> count_all_evens(const IntMatrix& numbers)
2 {
3     std::vector<int> even_numbers_count;
4     for (const auto& number_line: numbers) {
5         int even{0};
6         for (const auto& number: number_line) {
7             if (number % 2 == 0) {
8                 ++even;
9             }
10        }
11        even_numbers_count.push_back(even);
12    }
13    return even_numbers_count;
14 }
```

前面的函数保留一个 vector 来存储每个向量的偶数计数，输入为二维 vector。对于检索到的每个 vector，都将对其进行循环，并在每次遇到 vector 中的偶数时在计数器上增加 1。完成每个向量的循环后，最终结果推入包含数字列表的 vector 中。我们现在继续，并将其分解为更小的函数。首先，我们将负责计算偶数的代码部分移动到单独的函数中。

我们把它命名为 count_evens，如下所示:

```
1 int count_evens(const std::vector<int>& number_line) {
2     return std::count_if(number_line.begin(),
3                          number_line.end(),
4                          [](int num){return num % 2 == 0;});
5 }
```

注意如何使用 count_if() 算法。它接受两个迭代器，并将它们分别放在容器的开头和结尾。它还接受第三个参数，一元谓词，集合的每个元素都会调用该谓词。我们传递一个 lambda 作为一元谓词，也可以使用任何其他可调用实体，例如函数指针、std::function 等。

现在有了独立的计数函数，我们可以在 count_all_evens() 函数中调用它。下面是 count_all_evens() 在 C++ 中函数式编程的实现:

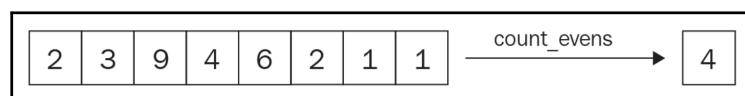
```
1 std::vector<int> count_all_evens(const std::vector<std::vector<int>>&
2 numbers) {
```

```

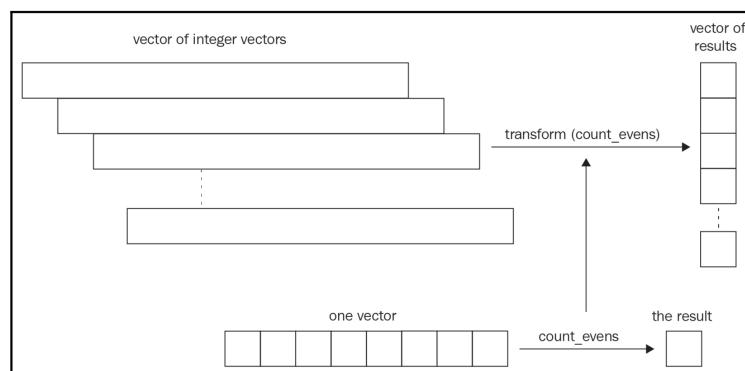
3   return numbers | std::ranges::views::transform(count_evens);
4 }
```

深入研究前面的代码之前，我们需要达成一致意见——这里并不是 | 操作符的什么怪异用法，而是为了简化代码。将其与我们在本节开始时介绍的代码版本进行比较，两者的作用一样，但第二种——功能更简洁。另外，注意这个函数不保留或改变任何状态。这在函数式编程中至关重要，因为函数必须是纯函数。它接受一个参数，然后在不修改它的情况下处理它，并返回一个新值（通常基于输入）。函数式编程的第一个挑战是将一个任务分解为更小的、易于组合的独立函数。

尽管是从命令式解决方案开始使用函数式解决方案，但在利用函数式编程范式时，这不是正确方式。应该改变思考问题的方式和处理问题的方式，而不是首先编写命令式代码，并修改它以获得函数式版本。读者应该经历从功能上思考的过程。所有的偶数的问题，我们使用一个 vector 解决问题。如果能找到解决单个 vector 问题的方法，就能解决所有 vector 问题。count_evens() 函数接受一个 vector 并产生单个值，如下图所示：



解决了 vector 的问题之后，我们应该通过将解应用到所有的向量，来继续解决最初的问题。transform() 函数本质上完成了我们所需要的工作：它接受一个可以应用于单个值的函数，并对其进行转换，以处理一个集合。下面的图像说明了如何使用它来实现一个函数 (count_all_evens)，该函数可以处理来自一次只处理一个项的函数 (count_evens) 的项的集合：



函数式编程的核心是，将较大的问题分解为较小的独立任务。每个函数都专门用于完成一个足够简单的任务，而没有意识到原来的问题。然后将函数组合在一起，从原始的初始输入生成一组已处理的结果。

count_all_evens() 函数的最终版本利用了 range。让我们来看看 range 是什么，以及如何使用它。在后面的例子中，我们还会用到它。

使用 range

range 与 view 绑定，我们将在本节中对它们进行研究。它们为我们提供了一种组合和处理对象集合的通用方法。我们经常使用迭代器来遍历容器并处理其元素，迭代器允许我们对算法和容器进行解耦。

例如，我们对 vector 应用了 count_if()，但是 count_if() 不知道它会应用于哪种容器。看看以下 count_if() 的声明：

```
1 template <typename InputIterator, typename UnaryPredicate>
2 constexpr typename iterator_traits<InputIterator>::difference_type
3 count_if(InputIterator first, InputIterator last, UnaryPredicate p);
```

除了 C++ 特有的详细声明之外，count_if() 不接受容器作为参数。相反，它使用迭代器操作——特别是输入迭代器。



输入迭代器支持使用 ++ 操作符进行迭代，并支持使用 * 操作符访问每个元素。还可以使用 == 和 != 关系来比较输入迭代器。

算法遍历容器而不知道容器的确切类型。可以对任何有开始和结束的实体使用 count_if()，如下所示：

```
1 #include <array>
2 #include <iostream>
3 #include <algorithm>
4 int main()
{
    std::array<int, 4> arr{1, 2, 3, 4};
    auto res = std::count_if(arr.cbegin(), arr.cend(),
        [] (int x){ return x == 3; });
    std::cout << "There are " << res << " number of elements equal to 3";
}
```

通常，我们对一个集合应用一个算法，并将该算法的结果存储为另一个集合，之后我们以同样的方式将该集合应用于更多算法。使用 std::transform() 将结果放入另一个容器中。例如，下面的代码定义了一个乘积向量：

```
1 // consider the Product is already declared and has a "name", "price", and
2 "weight"
3 // also consider the get_products() is defined
4 // and returns a vector of Product instances
5 using ProductList = std::vector<std::shared_ptr<Product>>;
6 ProductList vec{get_products()};
```

假设项目是由不同的程序员团队开发的，他们选择保留产品的名称为任意数字，例如：1 代表苹果，2 代表桃子，以此类推。这意味着 vec 将包含 Product 实例，每个实例在其名称字段中都有一个数字字符（而名称的类型是 std::string——这就是为什么我们将数字作为字符而不是其整数值）。现在，我们的任务是将产品名称从数字转换为完整字符串（apple、peach 等）。我们可以使用 std::transform：

```
1 ProductList full_named_products; // type alias has been defined above
2 using ProductPtr = std::shared_ptr<Product>;
3 std::transform(vec.cbegin(), vec.cend(),
4     std::back_inserter(full_named_products),
```

```
5 []( ProductPtr p){ /* modify the name and return */ } );
```

运行上述代码后，full_named_products 将包含具有产品名的 Product。为了过滤掉所有的 apple 并将它们复制到 apples 向量中，我们需要使用 std::copy_if:

```
1 ProductList apples;
2 std::copy_if(full_named_products.cbegin(), full_named_products.cend(),
3   std::back_inserter(apples),
4   [] (ProductPtr p){ return p->name() == "apple"; } );
```

代码示例的最大缺点是在引入 range 之前，缺乏良好的组合。range 为我们提供了一种处理容器元素和组合算法的优雅方式。

简单地说，range 是一个可遍历的实体，一个 range 有一个 begin() 和一个 end()，这与目前为止使用的容器非常相似。这些条件下，每个 STL 容器都可以视为一个 range。STL 算法重新定义为接受范围作为直接参数，它们允许将一个结果从一个算法直接传递给另一个算法，而不是将中间结果存储在局部变量中。例如，可以在前面在 begin() 和 end() 中使用的 std::transform，如果使用 range，则具有以下形式（以下代码是伪代码）。通过使用 range，可以按照以下方式重写前面的例子：

```
1 ProductList apples = filter(
2   transform(vec, [] (ProductPtr p){/* normalize the name */}),
3   [] (ProductPtr p){return p->name() == "apple";}
4 );
```

不要忘记包含 <ranges> 头文件。transform 将返回一个包含名称的 Product 指针的 range，也就是将数值替换为字符串值。然后，filter 函数将获取结果并返回名称为 apple 的产品 range。



注意，我们通过省略简化了这些代码示例在 filter 和 transform 函数前面的 std::ranges::views。相应地，使用它们作为 std::ranges::views::filter 和 std::ranges::views::transform。

最后，我们在本章开头的示例中使用了重载操作符 |，它允许我们在一起调用管道范围。这样，我们可以组合算法来产生最终结果，如下所示：

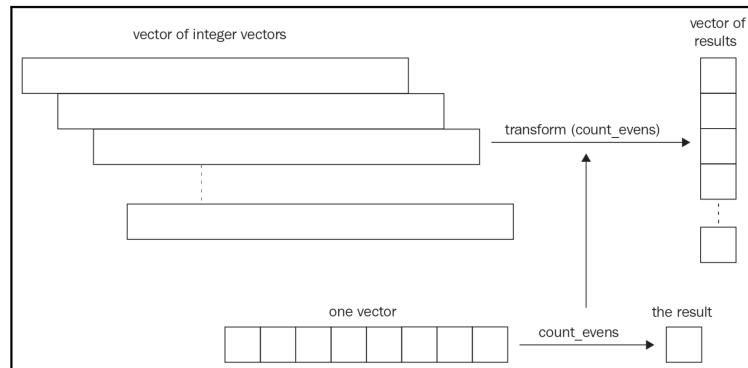
```
1 ProductList apples = vec | transform([] (ProductPtr p){/* normalize the name
2   */})
| filter([] (ProductPtr p){return p->name() ==
3   "apple";});
```

我们使用管道代替嵌套的函数调用。这一开始可能会令人困惑，因为我们使用 | 操作符作为位或。当你看到它被应用到一个集合时，它指的是管道范围。



| 操作符的灵感来自于 Unix shell 管道操作符。在 Unix 中，我们可以通过管道将多个进程的结果连接在一起，例如：ls -l | grep cpp | less 将在 ls 命令的结果中找到 cpp，并使用 less 程序一次显示一个屏幕的最终结果。

range 是集合之上的抽象，这并不意味着它是一个集合。这就是为什么前面的例子没有任何开销——它只是在一个函数之间传递一个 range，这个 range 只是提供一个集合的开始和结束。此外，还允许我们访问底层的集合元素。



该函数 (`transform` 或 `filter`) 返回一个范围结构，而不是一个集合。range 的 `begin()` 迭代器将指向源集合中满足谓词的元素。range 的迭代器是一个代理对象：它与常规迭代器的不同之处在于，指向一个满足给定谓词的元素。我们有时将它们称为智能迭代器，因为每次向前移动它（例如，通过递增），都会找到集合中满足谓词的下一个元素。更有趣的是，迭代器的“智能”取决于我们应用于集合的函数类型。例如，`filter()` 函数返回一个范围，该 range 的自增操作符具有智能迭代器。这主要是因为过滤器的结果可能比原始集合包含更少的元素。另一方面，`Transform` 并不返回元素数量较少的结果——它只是转换元素。这意味着 `transform` 返回的 range 对于自增/自减操作具有相同的功能，但元素访问将有所不同。对于每次访问，range 的智能迭代器将从原始集合返回转换后的元素。换句话说，只是为迭代器实现了 * 操作符，参考下面的代码片段：

```
1 auto operator*()
2 {
3     return predicate(*current_position);
4 }
```

这样，我们就创建了集合的新视图，同样适用于 `filter` 和其他函数。更有趣的是，区间视图利用了惰性求值。对于我们前面的示例，即使我们有两个 range 转换，结果也是通过在一次传递中求值产生的。

在 `transform` 和 `filter` 的示例中，每个函数都定义了一个视图，但不修改或求值。当我们把结果赋值给结果集合时，通过访问每个元素来从视图构造 `vector`。这个 `vector` 才是求值的地方。

就这么简单——range 为我们提供了带有惰性求值的函数组合。现在，我们简要地介绍了函数式编程中使用的工具集。接下来，让我们来看看这个范例的优点。

为什么使用函数式编程？

首先，函数式编程简化了代码。与命令式对应的代码相比，该代码要短得多。它提供了简单但极富表现力的工具。代码越少，bug 就越少。

函数不会改变任何东西，这使得并行化更加容易。这是并发程序的主要关注点之一，因为并发任务需要在它们之间共享可变数据。大多数情况下，必须使用互斥对象等显式地同步线程。函数式编程将我们从显式的同步中解放出来，我们可以在多个线程上运行代码而无需进行调整。第 8 章中，我们将详细讨论数据竞争。

函数式编程认为所有的功能都是纯粹的，是不会改变程序状态的函数。它们只是接受输入，以用户定义的方式对其进行转换，然后提供输出。纯函数对于相同的输入生成相同的结果，而不受调用次数的影响。当我们谈到函数式编程时，在默认情况下，我们应该考虑所有纯函数。

下面的函数接受 double 作为输入，并返回 square:

```
1 double square(double num) { return num * num; }
```



TIP 有些编译器，如 GCC，提供帮助编译器优化代码的属性。例如，[[gnu::pure]] 属性告诉编译器这个函数可以被认为是一个纯函数。这将使编译器确信函数不访问任何全局变量，并且函数的结果完全依赖于它的输入。

许多情况下，常规函数可以带来更快的解决方案。然而，为了适应这种模式，应该强迫自己从功能的角度思考。下面的程序声明了一个 vector，并计算其元素的平方根：

```
1 void calc_square_roots(std::vector<double>& vec)
2 {
3     for (auto& elem : vec) {
4         elem = std::sqrt(elem);
5     }
6 }
7
8 int main()
9 {
10    std::vector<double> vec{1.1, 2.2, 4.3, 5.6, 2.4};
11    calc_square_roots(vec);
12 }
```

这里，我们通过引用传递向量。如果我们在函数中改变它，我们就改变了原始集合。这显然不是一个纯函数因为它改变了输入向量。而函数式编程将在新的 vector 中返回计算后的值，而不影响输入：

```
1 std::vector<double> pure_calc_square_roots(const std::vector<double>& vec)
2 {
3     std::vector<double> new_vector;
4     for (const auto& elem : vec) {
5         new_vector.push_back(std::sqrt(elem));
6     }
7     return new_vector;
8 }
```

功能性思维的一个更好的例子是，解决一个较小的问题并将其应用到集合中。本例中，较小的问题是计算单个数字的平方根，这个数字已经实现为 std::sqrt。

将它应用到集合是通过 std::ranges::views::transform 完成的，如下所示：

```
1 #include <ranges>
2 #include <vector>
3
```

```
4 int main()
5 {
6     std::vector<double> vec{1.1, 2.2, 4.3, 5.6, 2.4};
7     auto result = vec | std::ranges::views::transform(std::sqrt);
8 }
```

我们使用 range，可以避免存储中间对象。前面的例子中，我们直接对向量应用了 transform。transform 返回一个 view，但不是由源向量的已转换元素组成的完整集合，而是在构造结果向量时生成元素的实际转换副本。另外，注意 std::sqrt 是一个纯函数。

我们在本章开头解决的例子，为函数式编程提供了必要的视角。为了更好地理解这个范式，我们应该熟悉它的原理。下一节中，我们将深入研究函数式编程的原理，以便更好地了解如何，以及何时使用。

函数式编程原理

尽管函数范式已经很老了（它诞生于 20 世纪 50 年代），但它并没有席卷编程界。目前占主导地位的大多数范式包括命令式语言和面向对象语言。正如我们在本书和其他许多书中多次提到的，C++ 是一种多范式语言。这就是学习 C++ 的好处：我们可以调整它以适应几乎所有的环境。掌握范式并不是一件容易的事。必须感受它并应用它，直到你最终开始从范例的角度思考。之后，将在几秒钟内看到常规任务的解决方案。

如果还记得第一次学习面向对象编程是什么时候，那么可能还记得在能够释放 OOP 的真正潜力之前，那些有些难懂的原则，函数式编程也是如此。本节中，我们将讨论函数式编程的基本概念，这些概念将成为进一步开发的基础。可以应用（或者已经这样做了）其中的一些概念，而无需实际使用功能范例。然而，请努力理解并应用下面的每一条原则。

纯函数

如果一个函数对其他变量的状态没有影响，那么它就是纯函数。与非纯函数相比，纯函数的性能较差，因为它们避免了由于状态修改而在代码中出现的大多数 bug。程序使用数据，因此使用状态修改功能，为最终用户带来一些预期的结果。

在面向对象编程中，我们将程序分解为对象，每一个对象都有一系列特殊的特性。OOP 中对象的基本特性之一是它的状态。通过向对象发送消息（换句话说，调用其方法）来修改对象的状态，在 OOP 中是至关重要的。通常，成员函数调用会导致对象状态的修改。函数式编程中，我们将代码组织成纯函数的集合，每个函数都有自己的目的，并且独立于其他函数。

我们看一个简单的例子，只是为了使这个概念更清楚。假设我们在一个程序中处理 User 对象，每个 User 对象都包含与 User 相关的信息。User 类在下面的代码块中声明为一个结构体：

```
1 struct User
2 {
3     int age;
4     string name;
5     string phone_number;
6     string email;
7 };
```

需要每年更新用户的年龄，假设我们有一个每年为每个 User 对象调用一次的函数。下面的函数接受一个 User 对象作为输入，并将其年龄增加 1：

```
1 void update_age(User& u)
2 {
3     u.age = u.age + 1;
4 }
```

update_age() 函数通过引用接受输入，并更新原始对象。但在函数式编程中却不是这样。下面的纯函数不是通过引用来获取原始对象，并改变其值，而是返回一个具有相同属性的 User 对象，只是更新了 age 属性：

```
1 User pure_update_age(const User& u) // cannot modify the input argument
2 {
3     User tmp{u};
4     tmp.age = tmp.age + 1;
5     return tmp;
6 }
```

尽管与 update_age() 相比，看起来效率较低，但这种方法的优点是使操作变得非常清晰（这在调试代码时非常有用），可以保证 pure_update_age() 不会修改原始对象。我们可以修改前面的代码，使它按值接受对象。这样，我们将跳过 tmp 对象的创建，因为实参本身表示为一个副本：

```
1 User pure_update_age(User u) // u is the copy of the passed object
2 {
3     u.age = u.age + 1;
4     return u;
5 }
```

如果用相同的参数多次调用纯函数，则每次都必须返回相同的结果。下面的代码演示了 pure_update_age() 函数在给定相同的输入时，返回相同的值：

```
1 User john{.age{21}, .name{"John"}};
2
3 auto updated{pure_update_age(john)};
4 std::cout << updated.age; // prints 22
5
6 updated = pure_update_age(john);
7 std::cout << updated.age; // prints 22
```

对于函数来说，每次为相同的输入数据调用它时，都可以以相同的方式运行。这意味着可以通过将应用程序分解成更小的函数，来设计应用程序的逻辑，每个函数都有明确的目的。然而，就额外的临时对象而言，纯函数有开销。常规的设计包括有一个集中存储程序状态的存储，由纯函数间接更新。每次使用纯函数后，该函数将修改后的对象作为新对象返回，必要时可以存储该对象。可以将其视为调整代码，以忽略传递的对象。

高阶函数

函数式编程中，函数是一等对象。我们应该把函数当作对象，而不是一组指令。这对我们有什么

么区别？此时，要将一个函数视为对象，体现的是将它传递给其他函数的能力。接受其他函数作为参数的函数称为高阶函数。

C++ 开发者将一个函数传递给另一个函数并不少见。下面是一些经典的方法：

```
1 typedef void (*PF)(int);
2 void foo(int arg)
3 {
4     // do something with arg
5 }
6
7 int bar(int arg, PF f)
8 {
9     f(arg);
10    return arg;
11 }
12
13 bar(42, foo);
```

代码中，声明了一个指向函数的指针。PF 表示函数的类型，接受一个整型参数，不返回任何值。示例是将指针函数作为参数，传递给其他函数的一种方法。我们把函数看作为一个对象，这需要我们对对象更深的理解。

前面的章节中，我们将对象定义为具有状态的东西。如果将函数视为对象，我们也应该能够在需要时以某种方式改变其状态。对于函数指针，情况并非如此。下面是传递一个函数给另一个函数的更好方法：

```
1 class Function
2 {
3     public:
4     void modify_state(int a) {
5         state_ = a;
6     }
7     int get_state() {
8         return state_;
9     }
10    void operator()() {
11        // do something that a function would do
12    }
13     private:
14     int state_;
15 };
16 void foo(Function f)
17 {
18     f();
19     // some other useful code
20 }
```

声明了一个具有重载函数操作符的类。每当重载类的操作符时，就使其可调用。因此，具有重载函数操作符的类对象可以视为函数（也称为函子）。这在某种程度上更像是一个技巧，因为我们没

有把函数变成对象，而是把对象变成了可调用的函数。然而，这让我们能够实现我们所寻找的：一个具有状态的函数。下面的客户端代码演示了函数对象有一个状态：

```
1 void foo(Function f)
2 {
3     f();
4     f.modify_state(11);
5     cout << f.get_state(); // get the state
6     f(); // call the "function"
7 }
```

通过这样做，我们可以跟踪函数被调用了多少次。下面是一个跟踪调用数量的简单示例：

```
1 class Function
2 {
3     public:
4         void operator()() {
5             // some useful stuff
6             ++called_;
7         }
8     private:
9         int called_ = 0;
10    };

```

最后，在<functional>头文件中定义的std::function演示了另一种定义高阶函数的方法：

```
1 #include <functional>
2
3 void print_it(int a) {
4     cout << a;
5 }
6 std::function<void(int)> function_object = print_it;
```

当调用function_object时（使用（）），它将调用委托给print_it函数。function封装了函数，并允许将其作为对象使用（并将其传递给其他函数）。

例子中以其他函数作为参数的函数，都是高阶函数的例子。返回函数的函数也称为高阶函数。总而言之，高阶函数是接受或返回另一个或多个函数的函数。看看下面的例子：

```
1 #include <functional>
2 #include <iostream>
3
4 std::function<int(int, int)> get_multiplier()
5 {
6     return [] (int a, int b) { return a * b; };
7 }
8
9 int main()
10 {
11     auto multiply = get_multiplier();
12     std::cout << multiply(3, 5) << std::endl; // outputs 15
13 }
```

`get_multiplier()` 返回一个用 `std::function` 包装的 lambda 函数。然后调用，就像调用一个常规函数一样。`get_multiplier()` 函数是一个高阶函数。我们可以使用高阶函数实现套用。函数式编程中，套用就是将一个函数带几个参数放入几个函数中，每个函数带一个参数，例如：将 `multiply(3,5)` 变为 `multiply(3)(5)`。我们可以这样做：

```
1 std :: function<int (int )> multiply (int a )
2 {
3     return [a]( int b) { return a * b; };
4 }
5
6 int main()
7 {
8     std :: cout << multiply (3)(5) << std :: endl ;
9 }
```

`multiply()` 只接受一个参数，返回的函数也只有一个参数。请注意 lambda 捕获：它捕获 `a` 的值，以便在它的体中将其乘以 `b`。



Curry 指的是逻辑学家 Haskell Curry。Haskell、Brook 和 Curry 编程语言也以他的名字命名。

套用的一个最有用的特性是，可以将抽象函数组合在一起。我们可以创建 `multiply()` 的特化版本，并将其传递给其他函数，或者在任何合适的地方使用。这可以在下面的代码中看到：

```
1 auto multiplyBy22 = multiply (22) ;
2 auto fiveTimes = multiply (5) ;
3
4 std :: cout << multiplyBy22(10); // outputs 220
5 std :: cout << fiveTimes(4); // outputs 20
```

使用 STL 时，一定使用了高阶函数。许多 STL 算法使用谓词来过滤或处理对象集合，例如：`std::find_if` 函数查找满足传递的谓词对象的元素，如下面的示例所示：

```
1 std :: vector<int> elems{1, 2, 3, 4, 5, 6};
2 std :: find _if (elems . begin () , elems . end () , [] ( int el) { return el % 3 == 0;});
```

`std::find_if` 将 lambda 作为其谓词，并对 `vector` 中的所有元素使用它。满足条件的元素将作为请求的元素返回。

另一个高阶函数的例子是 `std::transform`，我们在本章开始时介绍过（不要与 `ranges::view::transform` 混淆）。让我们用它把一个字符串转换成大写字母：

```
1 std :: string str = "lowercase" ;
2 std :: transform (str . begin () , str . end () , str . begin () ,
3     [] ( unsigned char c) { return std :: toupper (c); });
4 std :: cout << str ; // "LOWERCASE"
```

第三个形参是容器的开头，也是 `std::transform` 函数插入其当前结果的地方。

折叠表达式

折叠 (或简化) 是将一组值组合在一起以生成较少结果的过程。大多数时候，我们谈论的是单一的结果。折叠抽象了递归的结构上迭代的过程。例如，就元素访问而言，list 或 vector 具有递归性质。虽然 vector 的递归性质具有争议，但我们将其视为递归，因为它允许我们通过重复递增索引来访问其元素。为了处理这样的结构，我们通常会跟踪每一步的结果，然后处理下一个项目，以便稍后与前一个结果结合。根据我们处理集合元素的方向，折叠称为左折叠或右折叠。

例如，std::accumulate 函数 (另一个高阶函数的例子) 是折叠功能的完美例子，因为它组合了集合中的值。看看下面这个简单的例子：

```
1 std :: vector<double> elems {1.1, 2.2, 3.3, 4.4, 5.5};  
2 auto sum = std :: accumulate(elems.begin(), elems.end(), 0);
```

函数的最后一个参数是累加器。这个初始值应该用作集合第一个元素的前一个值。上面的代码计算 vector 元素的和。这是 std::accumulate 函数的默认行为。正如我们前面提到的，它是一个高阶函数，这意味着函数可以作为它的参数传递。然后对每个元素调用这个函数，以产生所需的结果，例如：找到前面声明的 elems 向量的乘积：

```
1 auto product = std :: accumulate(elems.begin(), elems.end(), 1, [](int prev, int cur)  
{ return prev * cur; });
```

需要一个有两个参数的函数，第一个参数是计算的前一个值，第二个参数是当前值。操作的结果将是下一个步骤的前一个值。代码可以用 STL 中的操作，以简洁的方式重写：

```
1 auto product = std :: accumulate(elems.begin(), elems.end(), 1, std :: multiplies<int>())  
 );
```

std::accumulate 函数的更好替代是 std::reduce 函数。reduce() 类似于 accumulate()，不同的是它不保持操作的顺序，它不必按顺序处理集合元素。可以将执行策略传递给 std::reduce 函数，并更改其行为，比如：并行处理元素。下面是如何使用并行执行策略，将 reduce 函数应用于前面例子中的 elems：

```
1 std :: reduce(std :: execution :: par, elems.begin(), elems.end(), 1, std :: multiplies<int>());
```

虽然 std::reduce 看起来比 std::accumulate 快，但是在使用非交换二元操作时应该小心。

折叠和递归密不可分。递归函数也通过将问题分解成更小的任务，逐个解决来解决整个问题。

深入地研究递归

第 2 章中讨论了递归函数的主要特性。来看看以下递归计算一个数字的阶乘的简单例子：

```
1 int factorial(int n)  
2 {  
3     if (n <= 1) return 1;  
4     return n * factorial(n - 1);  
5 }
```

与迭代函数相比，递归函数提供了优雅的解决方案。但是，应该谨慎地决定是否使用递归。递归函数最常见的问题之一是堆栈溢出。

头递归

头递归是我们熟悉的常规递归方式。前面的例子中，factorial 函数的行为就是一个头递归函数，它在处理当前步骤的结果之前进行递归调用。看一下阶乘函数的下面一行：

```
1 ...
2 return n * factorial(n - 1);
3 ...
```

为了找到并返回乘积的结果，调用 factorial 函数，并带有一个简化的参数，即 (n-1)。这意味着乘积 (* 操作符) 在某种程度上处于暂停状态，正在等待它的第二个参数通过阶乘 (n-1) 返回。堆栈的增长与递归调用函数的次数一致，让我们试着比较一下递归阶乘的实现和下面的迭代方法：

```
1 int factorial(int n)
2 {
3     int result = 1;
4     for (int ix = n; ix > 1; --ix) {
5         result *= ix;
6     }
7     return result;
8 }
```

这里的主要区别是，在每一步都将 product 的结果存储在同一个变量（名为 result）中。让我们尝试分解阶乘函数的递归实现。

很明显，每个函数调用都会占用堆栈上指定的空间。每个步骤的每个结果都应该存储在堆栈的某个地方。虽然我们知道它应该是同一个变量，但递归函数并不关心，它会为变量分配空间。常规递归函数要求我们找到一种解决方案，即知道每个递归调用的结果应该存储在相同的地方。

尾递归

尾递归是解决递归函数中存在多个不必要的变量问题的方法。尾递归函数的基本思想是在递归调用之前进行实际处理。下面是我们如何将阶乘函数转换为尾递归函数：

```
1 int tail_factorial(int n, int result)
2 {
3     if (n <= 1) return result;
4     return tail_factorial(n - 1, n * result);
5 }
```

注意函数的新参数。仔细阅读前面的代码，我们可以了解正在发生的尾递归的基本概念：处理在递归调用之前完成在 tail_factorial 函数再次调用之前，计算当前的结果 ($n * result$) 并传递给它。

虽然这种想法看起来并不吸引人，但如果编译器支持尾部调用优化 (TCO)，它就会非常高效。TCO 基本上涉及到阶乘函数（尾部）的第二个参数可以在每次递归调用时存储在相同的位置。这允许堆栈保持相同的大小，独立于递归调用的数量。

说到编译器优化，我们不能忽略模板元编程。我们在这里与编译器优化一起提到它，可以将元编程视为对程序的最大优化。编译时进行计算总是比在运行时进行计算更好。

C++ 中（函数式）的元编程

元编程可以看作是另一种编程范式，是一种完全不同的编码方法，因为不是在处理常规的编程过程。所谓常规过程，我们指的是程序在其生命周期中所经历的三个阶段：编码、编译和运行。很明显，一个程序在执行时做了它应该做的事情，可执行文件由编译器通过编译和链接生成。另一方面，元编程是代码编译期间执行代码的地方。如果你是第一次面对它，这听起来可能很神奇。如果程序还不存在，我们怎么执行代码？回顾第 4 章中关于模板的内容，我们知道编译器处理模板需要多次传递。第一次传递中，编译器定义模板类或函数中使用的必要类型和形参。接下来，编译器开始用我们熟悉的方式编译，它生成一些代码，这些代码将被链接器链接，以生成最终的可执行文件。

由于元编程是在代码编译过程中发生，我们应该知道使用了哪些语言概念和结构。任何可以在编译时计算的东西都可以用作元编程构造，比如：模板。

下面是用 C++ 进行元编程的经典例子：

```
1 template <int N>
2 struct MetaFactorial
3 {
4     enum {
5         value = N * MetaFactorial<N - 1>::value
6     };
7 };
8
9 template <>
10 struct MetaFactorial<0>
11 {
12     enum {
13         value = 1
14     };
15 };
16
17 int main() {
18     std::cout << MetaFactorial<5>::value; // outputs 120
19     std::cout << MetaFactorial<6>::value; // outputs 720
20 }
```

我们在上一节用不到 5 行代码编写了阶乘，为什么要为阶乘写这么多代码呢？原因在于它的效率。虽然编译代码需要花费更多的时间，但与普通的阶乘函数（递归或迭代实现）相比，是高效的。这种效率背后的原因是，实际的阶乘计算是在编译时进行的，当可执行文件运行时，结果已经可以使用了。我们只是在运行程序时使用了计算值，运行时不进行计算。如果是第一次看到这段代码，下面的解释会让您爱上元编程。

让我们详细分解和分析前面的代码。首先，元阶乘模板是用带有 value 属性的单个 enum 声明的。选择此 enum 仅仅是因为它的属性是在编译时计算的。因此，当我们访问 MetaFactorial 的 value 属性时，它已经在编译时计算（评估）了。看看枚举的实际值。它从相同的元阶乘类中建立递归依赖：

```
1 template <int N>
2 struct MetaFactorial
3 {
4     enum {
```

```
5     value = N * MetaFactorial<N - 1>::value
6 }
7 }
```

聪明的你可能已经注意到了这个技巧。MetaFactorial<N - 1> 和 MetaFactorial<N> 不是同一个结构。尽管具有相同的名称，但是具有不同类型或值的每个模板都将作为单独的新类型生成。所以，我们调用类似这样的东西：

```
1 std :: cout << MetaFactorial<3>::value ;
```

编译器会为每个值生成三个不同的结构体（下面是一些表示我们应该如何描绘编译器工作的伪代码）：

```
1 struct MetaFactorial<3>
2 {
3     enum {
4         value = 3 * MetaFactorial<2>::value
5     };
6 };
7
8 struct MetaFactorial<2>
9 {
10    enum {
11        value = 2 * MetaFactorial<1>::value ;
12    };
13 };
14
15 struct MetaFactorial<1>
16 {
17    enum {
18        value = 1 * MetaFactorial<0>::value ;
19    };
20 };
```

下一段代码中，编译器将生成的每个结构体的值替换为各自的数值，如下面的伪代码所示：

```
1 struct MetaFactorial<3>
2 {
3     enum {
4         value = 3 * 2
5     };
6 };
7
8 struct MetaFactorial<2>
9 {
10    enum {
11        value = 2 * 1
12    };
13 };
14
```

```
15 struct MetaFactorial<1>
16 {
17     enum {
18         value = 1 * 1
19     };
20 };
```

然后，编译器删除未使用的生成的结构，只留下 `MetaFactorial<3>`，如同 `MetaFactorial<3>::value` 一样。这也可以进行优化。这样做，我们可以得到如下结果：

```
1 std::cout << 6;
```

将这一行与前面的行进行比较：

```
1 std::cout << MetaFactorial<3>::value;
```

这就是元编程的美妙之处——在编译时完成，不会留下任何痕迹，就像忍者一样。与常规解决方案相比，编译需要更长的时间，但程序的执行速度是最快的。我们建议读者们尝试实现其他代价昂贵的计算的元版本，例如：计算第 n 个斐波那契数。这并不像为运行时编写代码那么简单，但是现在读者们已经对它的强大功能有了一定的了解。

总结

本章中，我们有了一个新视角。作为一种多范式语言，C++ 可以用作函数式编程语言。

我们学习了函数式编程的主要原理，如：纯函数、高阶函数和折叠表达式。纯函数不会改变状态的函数。纯函数的优点是，避免由于状态突变而引入的 bug。

高阶函数是接受或返回其他函数的函数。与函数式编程不同，C++ 开发者在处理 STL 时使用高阶函数。

纯函数，以及高阶函数，允许我们将整个应用程序分解成一个大的函数流水线。这个流水线中的每个函数负责接收数据并返回原始数据的新修改版本（不改变原始状态）。当这些功能组合在一起时，提供了一个协调良好的任务组。

下一章中，我们将深入研究多线程编程，并讨论 C++ 中的线程库组件。

问题

1. 列出 range 的优点。
2. 已知哪些函数是纯函数？
3. 从函数式编程的角度来看，纯虚函数和纯函数有什么区别？
4. 什么是折叠表达式？
5. 尾递归比头递归有什么好处？

扩展阅读

有关本章内容的更多信息，请查看以下链接：

- Learning C++ Functional Programming by Wisnu Anggoro: <https://www.packtpub.com/application-development/learning-c-functional-programming>

- Functional Programming in C++: How to Improve Your C++ Programs Using Functional Techniques by Ivan Cukic: <https://www.amazon.com/Functional-Programming-programs-functional-techniques/dp/1617293814/>

第 8 章：并发和多线程

并发编程可以让程序更高效。C++11 之前，C++ 都没有对并发性或多线程进行内置支持。现在它支持并发编程、线程、线程同步对象，以及将在本章讨论的其他功能。

为支持线程而更新语言之前，开发者必须使用第三方库。最流行的多线程解决方案之一是 POSIX(可移植操作系统接口) 线程。C++11 开始引入了线程支持，它使该 C++ 更加健壮，适用于更广泛的软件开发领域。理解线程对于 C++ 开发者来说很是关键，这样可以压缩程序的每个部分，使其运行得更快。线程向我们展示了一种完全不同的方法，通过并发运行函数来提高程序的速度。学习多线程是每个 C++ 程序员都要做的事情。有很多程序是无法避免使用多线程的，比如：网络应用程序、游戏和 GUI 应用程序。本章将介绍 C++ 中的并发和多线程的基础知识，以及并发代码设计的最佳实践。

本章中，我们将了解以下内容：

- 理解并发性和多线程
- 处理线程
- 管理线程和共享数据
- 设计并发代码
- 使用线程池避免线程创建的开销
- 了解 C++20 中的协程

编译器要求

g++ 编译器需要添加编译选项 `-std=c++2a` 来编译本章的代码。可以从这里获取本章的源码文件：<https://github.com/PacktPublishing/Expert-CPP>

理解并发和多线程

运行程序的最简单形式是由中央处理器 (CPU) 一个个地执行程序指令。程序的其中一个部分包含程序的指令，每条指令都会加载到 CPU 寄存器中，以便 CPU 解码并执行它。实际上，你使用什么编程方式来开发并不重要，结果总是相同的——形成包含机器码的可执行文件。

我们提到过像 Java 和 C# 这样的编程语言使用需要环境支持。但是，如果缺失了中间的环境支持（通常是虚拟机），那么执行的指令应该与特定 CPU 熟悉的形式和格式。很明显，CPU 运行语句的顺序在任何情况下都不会混合。例如，我们确定和可以继续为，以便下面的程序将分别输出 4、“hello” 和 5：

```
1 int a{4};  
2 std::cout << a << std::endl;  
3 int b{a};  
4 ++b;  
5 std::cout << "hello" << std::endl;  
6 b--;  
7 std::cout << (b + 1) << std::endl;
```

在将 a 变量的值输出到屏幕之前将初始化。同样，可以保证 “hello” 字符串将在 b 的值递减之前被打印出来，并且 (b + 1) 和将在将结果打印到屏幕之前被计算出来。每条指令的执行都可能涉

及从内存中读取数据或向内存中写入数据。

正如第 5 章所介绍，内存层次结构已经足够复杂，使得对程序执行的理解有点困难。例如，`int ba;` 前面例子中的行假设 `a` 的值从内存中加载到 CPU 的寄存器中，然后该寄存器将写入 `b` 的内存位置。这里的关键字是位置，为我们带来了一些特殊的解释。更具体地说，我们讨论的是内存位置。并发支持取决于语言的内存模型，即一组对内存并发访问的保证。虽然字节是最小的可寻址内存单元，但中央处理器处理数据中的 word 是 CPU 读写内存的最小单位。例如，我们认为以下两个声明是独立的变量：

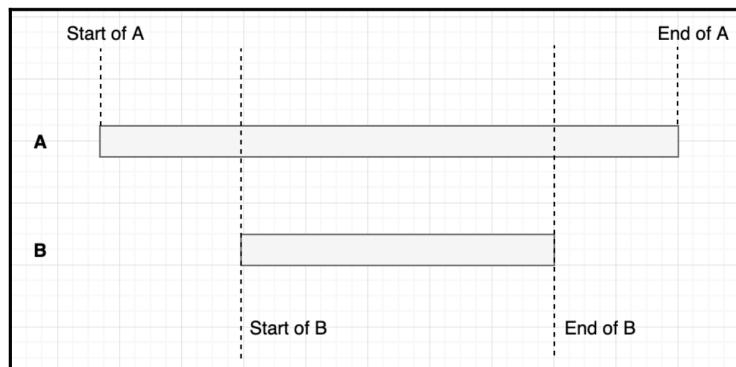
```
1 char one;
2 char two;
```

如果在同一个 word 中分配这些变量（考虑 word 的大小大于一个字符的大小），读写任何一个变量都涉及到读取包含这两个变量的 word。对变量的并发访问可能会导致意想不到的行为，这就是需要内存模型要解决的问题。C++ 内存模型保证两个线程可以访问和更新不同的内存位置，而不会相互干扰。内存位置是一个标量类型，标量类型是算术类型、指针、枚举或 `nullptr_t`。长度非零的相邻位域最大序列也是一个内存位置。例子如下：

```
1 struct S
2 {
3     char a; // location #1
4     int b: 5; // location #2
5     unsigned c: 11;
6     unsigned :0; // :0 separates bit fields
7     unsigned d: 8; // location #3
8     struct {
9         int ee: 8;
10    } e; // location #4
11};
```

前面的例子中，两个线程访问同一结构体的不同内存位置不会相互干扰。那么，当谈到并发或多线程时，我们应该考虑什么呢？

并发通常与多线程混淆，它们在本质上是相似的，但是不同的概念。方便起见，可以把并发想象成两个运行时间交错的操作。如果操作 A 和操作 B 的开始时间和结束时间在任意点交错，则操作 A 和操作 B 并发运行，如下图所示：



两个任务同时运行时，它们不必同时运行。想象一下这样的情景：你一边看电视一边上网。虽然这样做不是一个好习惯，但当有一个你不能错过的最喜欢的电视节目正在直播的同时，你的朋友

让你做一些关于蜜蜂的研究。你不能同时专注于两项任务，在任何固定的时刻，你的注意力要么被你正在看的节目，要么被你在网上读到的关于蜜蜂的有趣事实所吸引。你的注意力会时不时地从表演转移到蜜蜂身上。

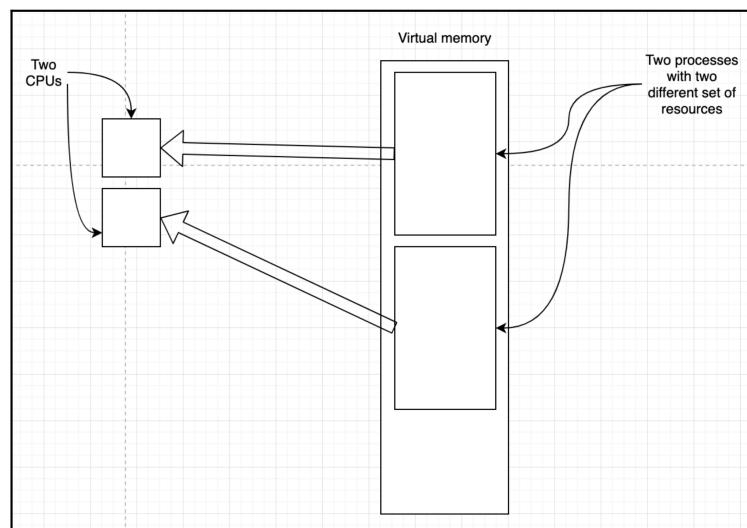
就并发性而言，你将同时执行两个任务。你的大脑给了节目一个时间部分：你看电视节目，然后切换到蜜蜂的文章，读几个句子，然后切换回节目。这是并发运行任务的一个简单示例。仅仅因为它们的开始时间和结束时间交错，并不意味着它们同时运行。另一方面，你在做前面提到的任何任务时都要呼吸。呼吸发生在背景中，你的大脑不会把你的注意力从节目或文章转移到你的肺部来吸气或呼气。一边看节目一边呼吸是并行运行任务的一个例子。这两个例子都向我们展示了并发性的本质。

当您在计算机上运行多个应用程序时会发生什么呢？它们是并行运行的吗？可以肯定的是，它们并发运行，但实际的并行度取决于计算机的硬件。如我们在前面的章节中所知道的，CPU 的主要工作是一个接一个地运行应用程序的指令。单个 CPU 如何同时处理两个应用程序的运行？为了理解这一点，我们先来了解进程。

进程

进程是运行在内存中程序的映像。当我们启动程序时，操作系统从硬盘读取程序内容，将其复制到内存中，并将 CPU 指向程序的启动指令。进程有私有的虚拟地址空间、堆栈和堆。两个进程不会相互干扰，这是操作系统提供的保障。如果开发者的工作的目标是进程间通信 (IPC)，那工作将会非常困难。本书中，我们不讨论底层的硬件特性，但应该对运行程序时发生的事情有大致的了解。这实际上取决于底层硬件——更确切地说，是 CPU 的类型和结构。CPU 的数量、CPU 内核的数量、缓存内存的级别，以及 CPU 或其内核之间的共享缓存内存——所有这些都会影响操作系统运行和执行程序的方式。

计算机系统中 CPU 的数量定义了真正并行运行的进程的数量。如下图所示：

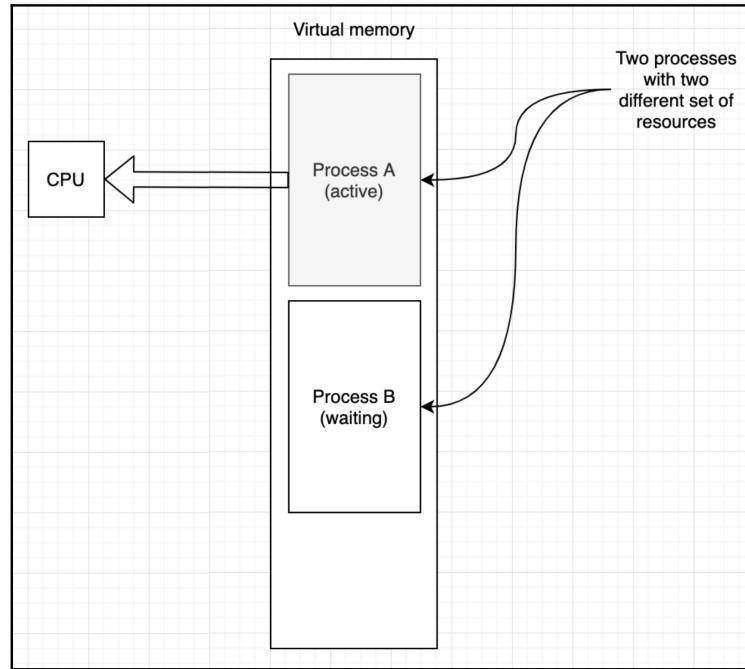


当我们谈到多处理器时，也就是可以多个进程并发运行的环境，同样棘手的部分也来了。如果这些进程实际上同时运行，那么它们在并行运行。所以，并发不是并行，而并行肯定是并发。

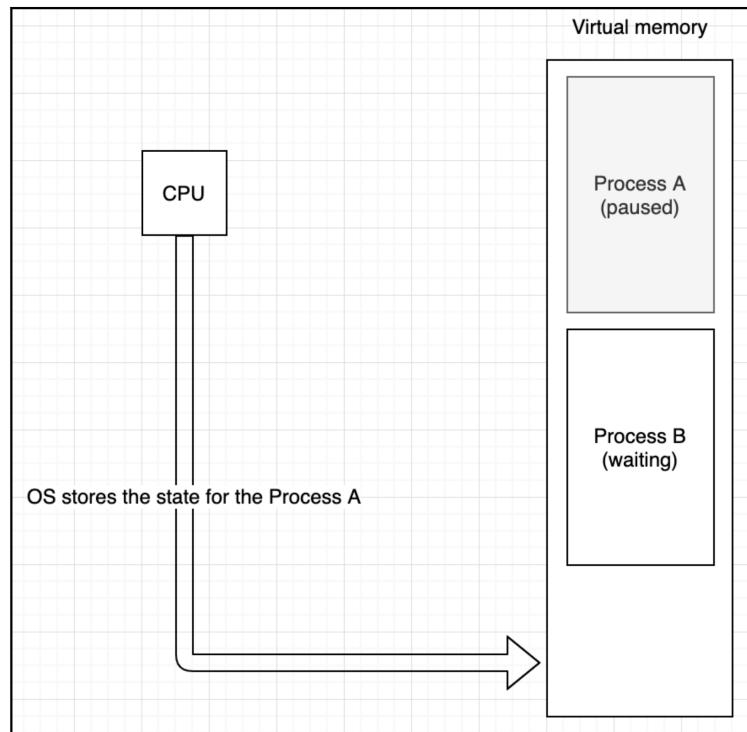
如果系统只有一个 CPU，则进程是并发运行，而不是并行运行。操作系统通过一种称为上下文切换的机制来管理这一点，上下文切换意味着暂时冻结进程的工作，复制进程当前使用的所有寄

存器值，并存储进程的所有活动资源和数据。当一个进程停止时，另一个进程拥有运行的权限。在为第二个进程提供了指定的时间后，操作系统开始对上下文进行切换，也会复制进程使用的所有资源。然后，启动前一个进程。在启动该进程之前，操作系统将资源和值复制回第一个进程所使用的相应槽位，然后再恢复该进程的执行。

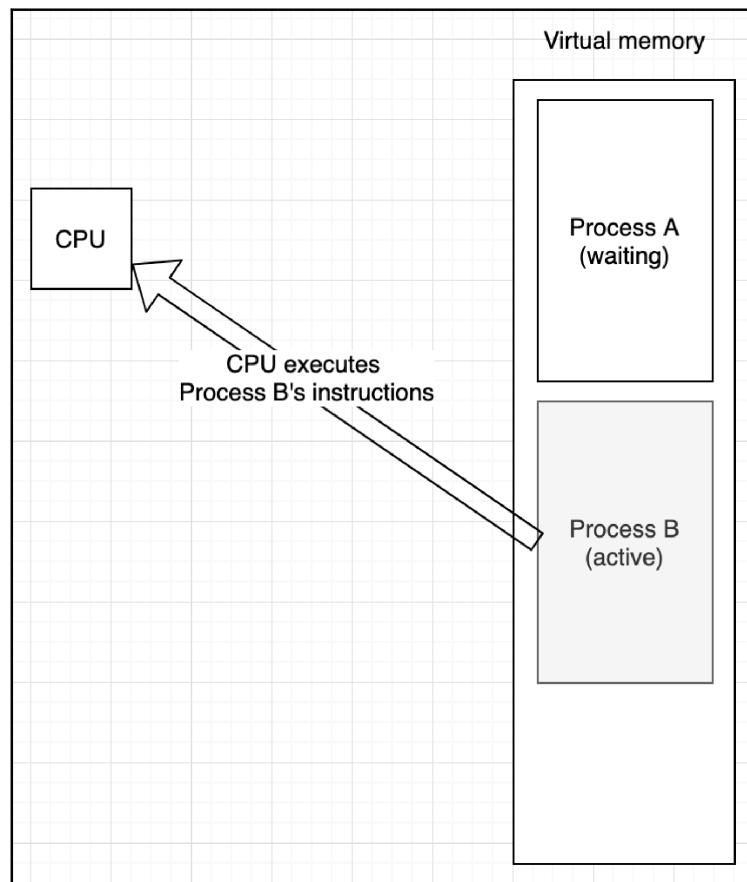
有趣的是，进程进行得如此之快，以至于用户实际上无法注意到操作系统中运行的程序实际上并不是同时运行的。下图描述了单个 CPU 运行的两个进程。当其中一个进程处于活动状态时，CPU 会按顺序执行指令，并将中间数据存储在寄存器中（也应该考虑缓存内存）。另一个进程正在等待操作系统提供它的运行时间：



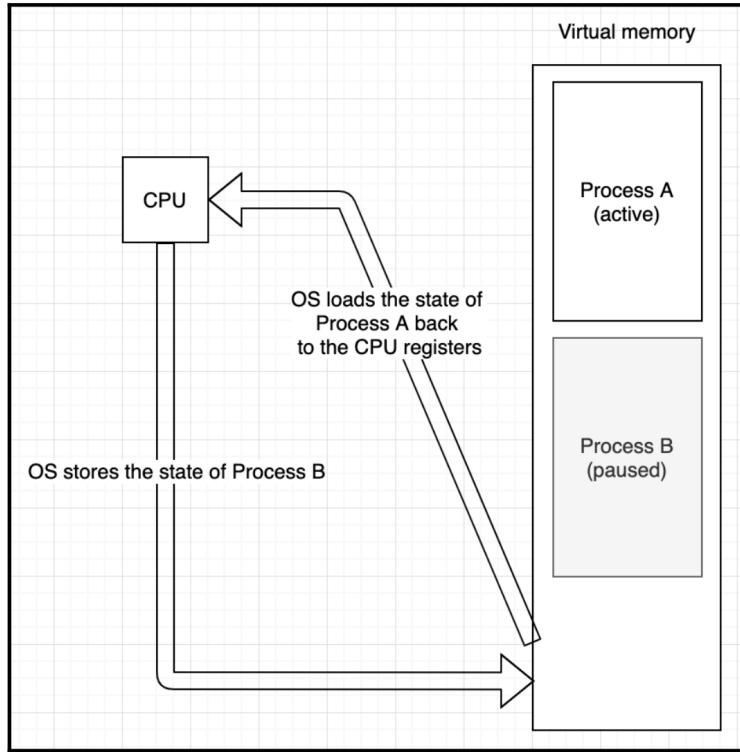
对操作系统来说，运行多个进程是一项复杂的工作。它管理进程的状态，定义哪个进程应该比其他进程占用更多的 CPU 时间等。在操作系统切换到另一个进程之前，每个进程都有一个固定的运行时间。这个时间对于一个过程来说可能更长，而对于另一个过程来说可能更短。操作系统为优先级高的进程提供更多的时间，例如：系统进程优先级高于用户进程。另一个例子，监视网络运行状况的后台任务具有比计算器应用程序更高的优先级。当提供的时间片结束时，OS 发起一个上下文切换，即它存储进程 A 的状态，以便以后继续执行：



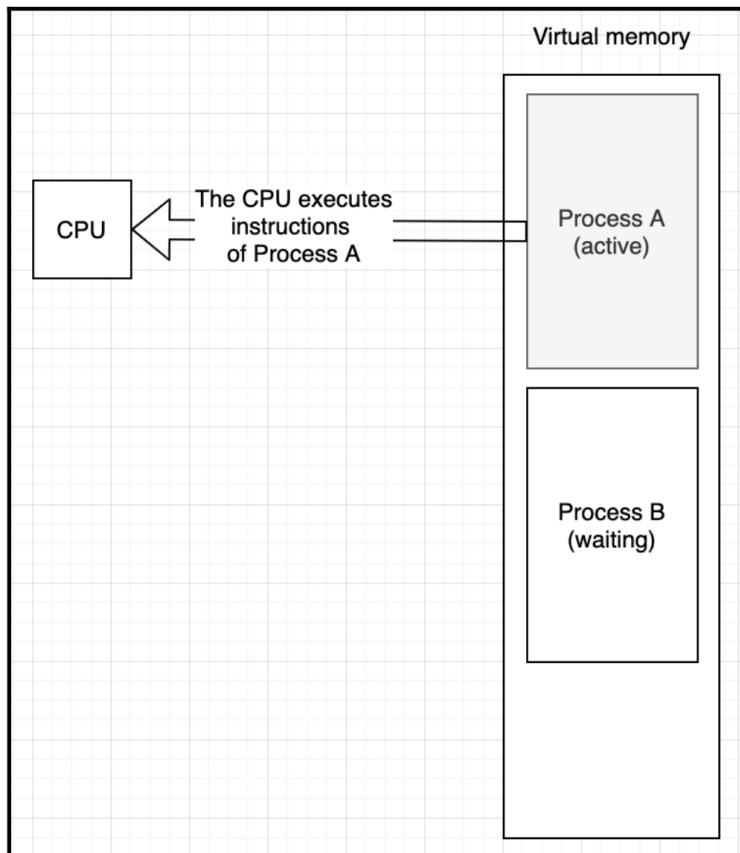
存储状态后，如下图所示，切换到下一个进程执行：



显然，如果进程 B 之前正在运行，那么它的状态应该加载回 CPU。同样的，当进程 B 的时间片（或时间量）打开时，操作系统存储进程 B 的状态并将进程 A 的状态加载回 CPU（被操作系统暂停之前的状态）：



过程没有任何共同之处——至少他们是这样认为的。每个正在运行的进程行为像单独的一样，它拥有操作系统所能提供的所有资源。其实，操作系统会设法让进程不知道彼此，因此每个进程是独立的。最后，返回进程 A 的状态后，CPU 继续执行它的指令，就像什么都没发生一样：



进程 B 被冻结，直到有一个新的时间片可供它运行。

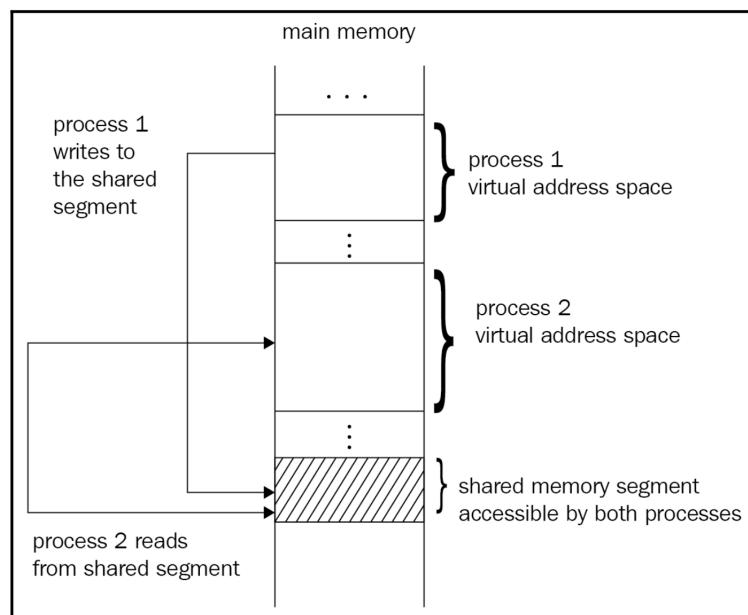
单个 CPU 运行多个进程类似于老师检查学生的试卷。教师一次只能检查一份试卷，不过他们可以通过逐一检查每个考试的答案来引入一些并发性。首先，他们检查一个学生的第一道题的答案，然后切换到第二个学生的第一道题，然后切换回第一个学生的第二道题，以此类推。当老师从一张试卷切换到另一张试卷时，他们会在结束的地方记下问题的编号。这样，当他们回到同一张试卷时，他们就知道从哪里开始。

同样，操作系统在暂停一个进程以恢复另一个进程之前，记录下它的执行点。第二个进程可以（而且很可能会）使用暂停进程使用的寄存器集。这将迫使操作系统将第一个进程的注册值存储在某个地方，以便稍后恢复。当操作系统暂停第二个进程以恢复第一个进程时，会将已经保存的寄存器值加载回相应的寄存器中。恢复的进程不会注意到任何差异，并且会像从未暂停过一样继续工作。

前面两段中所描述的一切都与单 CPU 系统有关。在多 CPU 系统中，系统中的每个 CPU 都有自己的一组寄存器。而且，每个 CPU 都可以独立于其他 CPU 执行程序指令，这就允许并行运行进程而不需要暂停和恢复它们。本例中，有几个助手的教师类似于有三个 CPU 的系统。每人可检查一份试卷，所以他们在任何时间都在检查三份不同的试卷。

进程中的挑战

当进程需要以某种方式相互联系时，就会出现麻烦。假设一个进程应该计算一些东西，并将值传递给一个完全不同的进程。有几种方法可以实现 IPC——其中一种是使用进程之间共享的内存段。下面的图表描述了两个访问共享内存段的进程：



一个进程将计算结果存储到内存中的共享段中，另一个进程从该段中读取计算结果。前面的例子中，老师和他们的助手在共享的论文中分享他们的检查结果。另一方面，线程共享进程的地址空间，因为它们运行在进程的上下文中。当进程是程序时，线程是函数而不是程序。也就是说，一个进程必须至少有一个线程，我们称之为执行线程。线程是在系统中运行的程序指令的容器，而进程封装线程并为其提供资源。我们最感兴趣的是线程及其编排机制，现在让我们来会会他们。

线程

线程是操作系统调度程序可以调度范围内的一段代码，尽管进程是正在运行的程序的镜像，但与利用多线程的项目相比，管理多进程项目比 IPC 要困难得多。程序处理数据，通常是数据的收集、访问、处理和更新数据是由对象方法，或组合在一起以实现最终结果的自由函数的函数完成的。在大多数项目中，我们处理成千上万的函数和对象。每个函数都代表一串指令，这些指令以一个名称命名，其他函数可以调用。多线程旨在同时运行函数以获得更好的性能。

例如，计算三个不同向量的和并输出，程序会调用计算第一个向量和第二个向量和最后一个向量的和的函数。这一切都是串行发生的。如果处理单个向量需要一定的时间，那么程序将以 $3A$ 的时间运行。下面的代码演示了这个例子：

```
1 void process_vector(const std::vector<int>& vec)
2 {
3     // calculate the sum and print it
4 }
5 int main()
6 {
7     std::vector<int> vec1{1, 2, 3, 4, 5};
8     std::vector<int> vec2{6, 7, 8, 9, 10};
9     std::vector<int> vec3{11, 12, 13, 14, 15};
10    process_vector(vec1); // takes A amount of time
11    process_vector(vec2); // takes A amount of time
12    process_vector(vec3); // takes A amount of time
13 }
```

如果有一种方法可以同时为三个不同的向量运行同一个函数，那么在前面的例子中，整个程序只需要花费一个函数的时间。执行线程，或者是线程，是并发运行任务的方式。所谓任务，通常指的是函数，可以参考 `std::packaged_task`。同样，不应该将并发与并行混淆。当我们谈到并发运行的线程时，应该考虑前面讨论的进程上下文切换。这同样适用于线程。



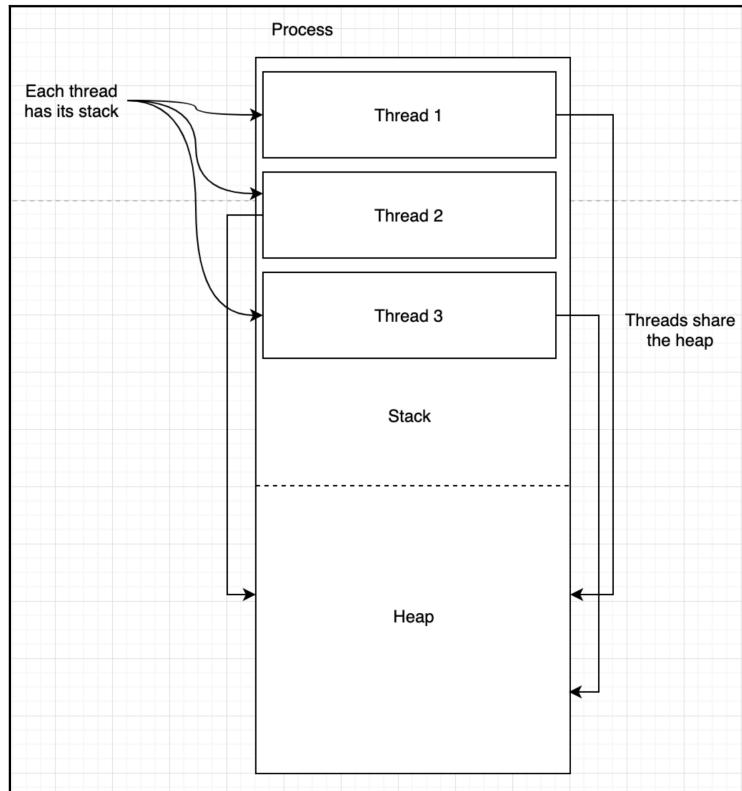
`std::packaged_task` 类似于 `std::function`。它包装了一个可调用对象——函数、lambda、函数对象或绑定表达式，与 `std::packaged_task` 的区别在于可以异步调用。

每个进程都有一个执行线程，有时称为主线程。一个进程可以有多个线程，这就是我们称之为多线程的时候。线程以与进程几乎相同的方式运行。它们也有上下文切换。

线程彼此分开运行，它们共享进程的大部分资源，因为所有线程都属于进程。进程占用硬件和软件资源，如 CPU 寄存器和内存段，包括自己的堆栈和堆。当一个进程不与其他进程共享其堆栈或堆时，线程必须使用该进程可用的相同资源。线程生命周期中发生的所有事情都发生在进程生命周期中。

然而，线程并不共享堆栈，每个线程都有自己的栈。这种隔离背后的原因是，线程只是一个函数，函数本身能够访问堆栈，以管理其参数和局部变量的生命周期。当我们以两个（或多个）单独运行的线程运行同一个函数时，运行时应该以某种方式处理它们的边界。尽管很容易出错，但可以将变量从一个线程传递到另一个线程（通过值或引用）。假设我们启动了三个线程，为前面示例中的三个向量运行 `process_vector()` 函数。想象一下，启动一个线程意味着以某种方式复制底层函数

(变量，而不是指令)，并将其与任何其他线程分开运行。这个场景中，相同的函数将复制为三个不同的镜像，每个镜像都将独立于其他映像运行，因此每个镜像都应该有自己的堆栈。另一方面，堆是线程之间共享的。所以，我们可以得出如下结论：

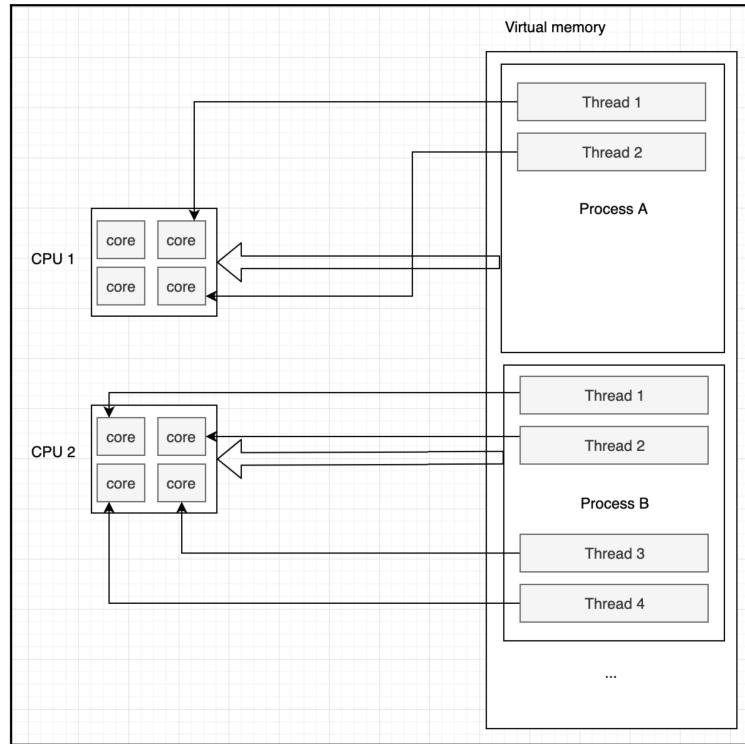


在进程的情况下，并发运行的线程不一定是并行运行的。每个线程获得一小部分 CPU 时间来运行，并且从一个线程切换到另一个线程也有开销。每个暂停线程的状态应该存储在某个地方，以便以后恢复时恢复。CPU 的内部结构决定了线程是否能够真正并行运行，CPU 的物理核数定义了真正可以并行运行的线程数。

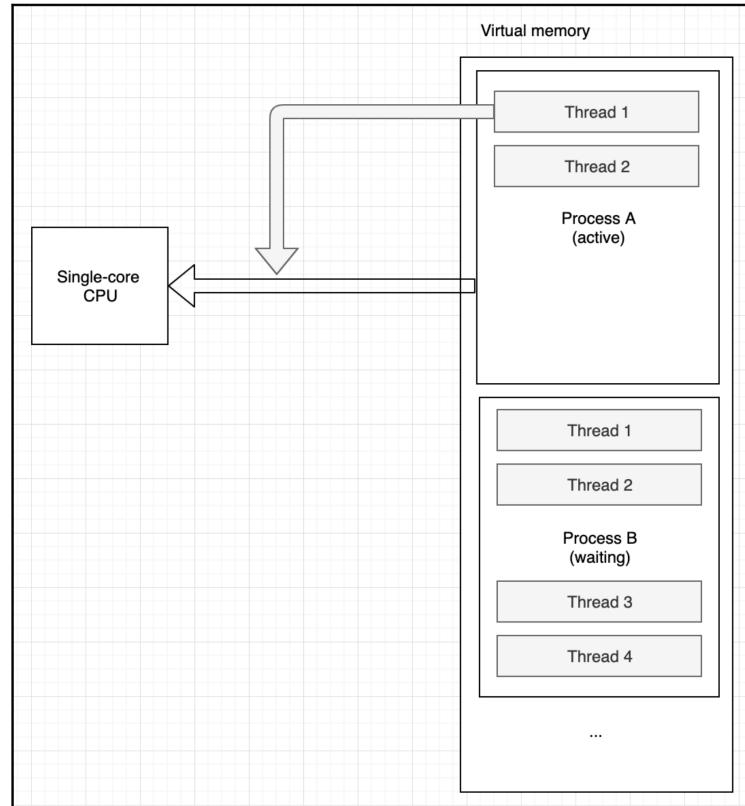


TIP C++ 线程库提供了 `hardware_concurrency()` 函数来查看可以并发运行的线程数。开发者可以在设计并发代码时参考这个数字。

下图描述了两个 CPU，每个 CPU 有四个核。每个核可以独立运行一个线程：



两个进程不仅并行运行，而且它们的线程也使用 CPU 内核并行运行。如果我们有几个线程，但只有一个单核 CPU，情况会如何变化？这几乎与我们前面介绍的过程相同。看看下面的图表——它描述了 CPU 在某个时间片中是如何执行线程 1 的：



当前活动的进程 A 有两个并发运行的线程。每个指定的时间点，只有一个线程被执行。当线程 1 的时间片就绪时，将执行线程 2。与我们讨论的进程模型不同的是，线程共享进程的资源，如

果我们不关心并发代码设计问题，会导致不自然的行为。让我们继续深入研究 C++ 的多线程，并找出使用多线程时会出现什么问题。

使用线程

当启动 C++ 程序时，即 main() 函数开始执行时，可以创建并启动将与主线程并发运行的新线程。要在 C++ 中启动线程，应该声明一个 thread 对象，并将想要并发运行的函数传递给主线程。下面的代码演示了使用 <thread> 中定义的 std::thread 声明和启动一个线程：

```
1 #include <thread>
2 #include <iostream>
3
4 void foo() { std::cout << "Testing a thread in C++" << std::endl; }
5
6 int main()
7 {
8     std::thread test_thread{foo};
9 }
```

我们可以创建一个更好的示例来展示两个线程是如何并发工作的。假设我们在一个循环中同时打印数字，看看哪个线程打印什么：

```
1 #include <thread>
2 #include <iostream>
3
4 void print_numbers_in_background()
5 {
6     auto ix{0};
7     // Attention: an infinite loop!
8     while (true) {
9         std::cout << "Background: " << ix++ << std::endl;
10    }
11 }
12
13 int main()
14 {
15     std::thread background{print_numbers_in_background};
16     auto jx{0};
17     while (jx < 1000000) {
18         std::cout << "Main: " << jx++ << std::endl;
19     }
20 }
```

前面的例子将同时输出 Main: 和 Background: 两个前缀。输出的片段如下：

```
1 ...
2 Main: 90
3 Main: 91
4 Background: 149
5 Background: 150
```

```
6 Background: 151
7 Background: 152
8 Background: 153
9 Background:
10 Main: 92
11 Main: 93
12 ...
```

每当主线程完成它的工作（在屏幕上打印一百万次），程序就希望在不等待后台线程完成的情况下完成（会导致程序终止）。让我们看看应该如何修改前面的示例。

等待线程

线程提供了 `join()` 函数，如果想等待它完成的话。下面是前一个等待后台线程的例子的修改版本：

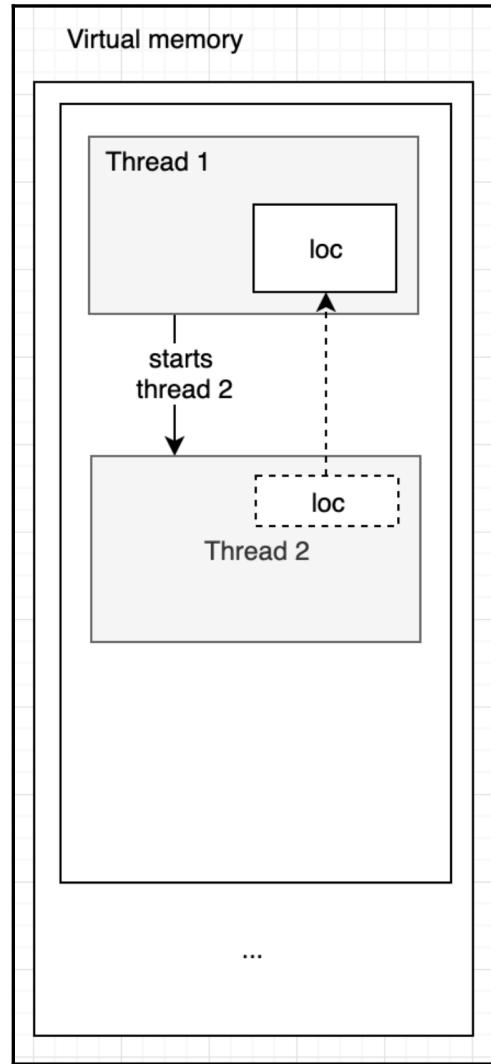
```
1 #include <thread>
2 #include <iostream>
3
4 void print_numbers_in_background()
5 {
6     // code omitted for brevity
7 }
8
9 int main()
10 {
11     std::thread background{print_numbers_in_background};
12     // the while loop omitted for brevity
13     background.join();
14 }
```

`thread` 函数作为独立的实体运行，独立于其他线程——甚至是启动它的线程。它不会等待它刚刚启动的线程，这就是为什么您应该显式地告诉调用函数等待它结束。有必要通知调用线程（主线程）正在等待线程在它自己之前结束。

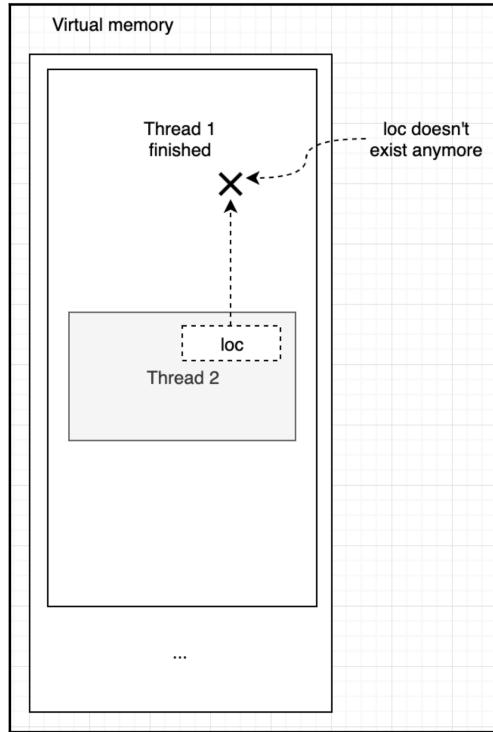
与 `join()` 函数相反的函数是 `detach()` 函数。`detach()` 函数表示调用者对等待线程结束不感兴趣。这种情况下，线程可以有一个独立的生命。如图所示：

```
1 std::thread t{foo};
2 t.detach();
```

尽管分离线程看起来很自然，但是在很多情况下我们都需要等待线程完成。例如，我们可以将 `local` 变量传递给调用者变量，并将其传递给正在运行的线程。这种情况下，我们不能让调用者分离线程，因为调用者可能在线程开始之前完成它的工作。为了清楚起见，我们来说明一下。线程 1 声明了 `loc` 变量并将其传递给线程 2，线程 2 是从线程 1 开始的：



如果线程 1 没有汇入 loc，那么将 loc 的地址传递给线程 2 是容易出错的。如果线程 1 在线程 2 之前完成了它的执行，那么通过它的地址访问 loc 会导致一个未定义的行为：



现在已经没有这样的对象了，所以我们对这个程序最好的期望就是崩溃。它将导致意想不到的行为，因为正在运行的线程将不再能够访问调用者的局部变量。所以，应该显式汇入或分离一个线程。

我们可以将任何可调用对象传递给 `std::thread`。下面的例子演示了如何将 lambda 表达式传递给线程：

```

1 #include <thread>
2
3 int main() {
4     std::thread t1[] {
5         std::cout << "A lambda passed to the thread";
6     };
7     t1.join();
8 }
```

此外，我们可以使用可调用对象作为线程参数。看看下面的代码，自定义了 `TestTask` 类的函数操作符：

```

1 #include <thread>
2
3 class TestTask
4 {
5 public:
6     TestTask() = default;
7     void operator()() {
8         state_++;
9     }
10 private:
```

```

11 int state_ = 0;
12 };
13
14 int main() {
15     std::thread t{TestTask()};
16     t.join();
17 }

```

函数(带有覆盖 operator() 函数的 TestTask 类)的优点是它能够存储状态信息。回到线程, 让我们继续讨论新标准添加的新特性, 它允许以更好的方式汇入线程。

使用 std::jthread

C++20 引入了一个可汇入的线程——std::jthread。它提供了与所提供的相同的接口 std::thread, 因此我们可以在代码中用 jthread 替换所有线程。它实际上封装了 std::thread, 所以基本上它委托给封装的线程。

如果编译器的版本不支持 std::jthread, 可以使用 RAII(获取资源即初始化)的习惯用法, 这完全适用于线程。看看下面的代码:

```

1 class thread_raii
2 {
3 public:
4     explicit thread_raii(std::thread& t)
5     : thread_(std::move(t))
6     {}
7     thread_raii() {
8         thread_.join();
9     }
10 private:
11     std::thread thread_;
12 };
13
14 void foo() {
15     std::cout << "Testing thread join";
16 }
17
18 int main() {
19     std::thread t{foo};
20     thread_raii r{t};
21     // will automatically join the thread
22 }

```

但是, 前面的代码缺少额外的检查, 因为传递给 RAII 类的线程可能已经分离了。为了查看是否可以汇入线程, 我们使用 joinable() 函数。下面是重写 thread_raii 类的方法:

```

1 class thread_raii
2 {
3 public:
4     explicit thread_raii(std::thread& t)

```

```

5   : thread_(std::move(t))
6   {}
7   ~thread_raii()
8   {
9     if (thread_.joinable()) {
10       thread_.join();
11     }
12   }
13 private:
14   std::thread thread_;
15 };

```

调用 `join()` 函数之前，析构函数首先测试线程是否可汇入。但是，我们更喜欢使用 `std::jthread`，而不是处理习惯用法，也不关心线程在汇入之前是否已经汇入。下面是我们如何使用前面声明的 `TestTask` 函数来实现这一点：

```

1 std::jthread jt{TestTask()};

```

就是这样——不需要调用 `jt.join()`，也不需要通过合并 `jthread` 来立即使用新的合作中断续特性。`jthread` 是协作可中断的，因为它提供了 `request_stop()` 函数，该函数的作用正如其名——请求线程停止。尽管该请求的实现由实现定义，但这是一种不需要永远等待线程的好方法。回想一下在无限循环中线程打印数字的示例。我们修改了主线程来等待它，而这将导致永远等待它。下面是我们如何使用 `std::jthread` 的 `request_stop()` 成员函数来修改线程：

```

1 int main()
2 {
3   std::jthread background{print_numbers_in_background};
4   auto jx{0};
5   while (jx < 1000000) {
6     std::cout << "Main: " << jx << std::endl;
7   }
8   // The main thread is about to finish, so we request the background thread to stop
9   background.request_stop();
10 }

```

`print_numbers_in_background()` 函数现在接收请求，并可以相应地进行操作。现在，让我们看看如何向 `thread` 函数传递参数。

向线程函数传递参数

`std::thread` 构造函数接受参数并将它们转发给底层线程函数。例如，为了将参数 4 和 2 传递给 `foo()` 函数，我们将参数传递给 `std::thread` 构造函数：

```

1 void foo(int one, int two) {
2   // do something
3 }
4
5 std::thread t{foo, 4, 2};

```

参数 4 和 2 将作为第一个和第二个参数传递给 foo() 函数。

下面的例子演示了通过引用传递参数:

```
1 class big_object {};
2
3 void make_changes(big_object&);
4
5 void error_prone()
6 {
7     big_object b;
8     std::thread t{make_changes, b};
9     // do something else
10 }
```

要理解为什么将函数命名为 error_prone，我们应该知道 thread 构造函数复制传递给它的值，然后将它们传递给带有右值引用的 thread 函数，这样做是为了只使用移动类型。因此，它将尝试使用右值调用 make_changes() 函数，这将导致编译失败（不能将右值传递给需要非常量引用的函数）。我们需要将其作为 std::ref 引用的参数包装起来。

```
1 std::thread t{make_changes, std::ref(b)};
```

前面的代码强调了参数应该通过引用传递。使用线程需要更加注意，因为有很多方法会在程序中触发异常或未定义的行为。让我们看看如何管理线程以生成更安全的多线程应用程序。

管理线程和共享数据

如果线程的数量超过了硬件支持的并行运行线程的数量，那么线程的执行包括暂停和恢复线程。除此之外，创建线程也有开销。处理项目中有多个线程的建议实践之一是使用线程池。

线程池的概念基于缓存的概念。我们在某个容器中创建并保存线程，以供以后使用，容器称为池。例如，下面的向量表示一个简单的线程池:

```
1 #include <thread>
2 #include <vector>
3
4 std::vector<std::thread> pool;
```

每当我们需要一个新线程时，我们不用声明相应的 std::thread 对象，而是使用一个已经在池中创建的对象。处理完线程后，可以将其推回到 vector 中，以便在必要时在以后使用。这在使用 10 个或更多线程时节省了一些时间。一个例子是 web 服务器。

web 服务器是一个程序，它等待传入的客户端连接，并为每个客户端创建一个独立的连接，以独立于其他客户端进行处理。典型的 web 服务器通常同时处理数千个客户机。每次用某个客户机发起一个新连接时，web 服务器都会创建一个新线程并处理客户机请求。下面的伪代码演示了一个 web 服务器的传入连接管理的简单实现:

```
1 void process_incoming_connections() {
2     if (new connection from client) {
3         t = create_thread(); // potential overhead
4         t.handle_requests(client);
```

```

5     }
6 }
7 while (true) {
8     process_incoming_connections();
9 }
```

使用线程池时，上述代码会避免在每次处理客户机请求时创建线程。创建新线程需要操作系统进行额外且相当耗时的工作。为了节省时间，我们使用了一种机制，它省略了为每个请求创建新线程。为了使池更好用，我们用队列替换它的容器。当我们请求线程时，线程池将返回一个空闲线程，当我们完成一个线程时，我们将它推回线程池。线程池的简单设计如下所示：

```

1 #include <queue>
2 #include <thread>
3 class ThreadPool
4 {
5 public:
6     ThreadPool(int number_of_threads = 1000) {
7         for (int ix = 0; ix < number_of_threads; ++ix) {
8             pool_.push(std::thread());
9         }
10    }
11    std::thread get_free_thread() {
12        if (pool_.empty()) {
13            throw std::exception("no available thread");
14        }
15        auto t = pool_.front();
16        pool_.pop();
17        return t;
18    }
19    void push_thread(std::thread t) {
20        pool_.push(t);
21    }
22 private:
23     std::queue<std::thread> pool_;
24 };
```

构造函数创建线程并将其推入队列。下面的伪代码中，我们用 ThreadPool 代替了直接创建处理客户端请求的线程，这是我们之前看到的：

```

1 ThreadPool pool;
2 void process_incoming_connections() {
3     if (new connection from client) {
4         auto t = pool.get_free_thread();
5         t.handle_request(client);
6     }
7 }
8
9 while (true) {
10     process_incoming_connections();
```

```
11 }
```

假设 handle_request() 函数在完成后将线程推回池，那么池就作为线程的集中式存储区。尽管前面的代码片段不能用于生产，但它表达了在密集型应用程序中使用线程池的基本思想。

共享数据

竞争条件是使用多线程的开发者所头痛的，并且应该避免的。假设有两个函数同时处理相同的数据，如下所示：

```
1 int global = 0;
2 void inc() {
3     global = global + 1;
4 }
5 ...
6 std::thread t1{inc};
7 std::thread t2{inc};
```

潜在的竞争条件正在发生，因为线程 t1 和 t2 用多个步骤修改同一个变量。单个线程安全的步骤中，执行的任何操作都称为原子操作。本例中，即使使用自增操作符，递增变量的值也不是原子操作。

使用互斥锁来保护共享数据

为了保护共享数据，可使用互斥对象。互斥对象是控制线程运行的对象。想象一下，线程在一个接一个地处理数据。当一个线程锁定一个互斥锁时，另一个线程会等待直到数据处理完毕，然后解锁互斥锁。然后，另一个线程锁定互斥锁并开始处理数据。下面的代码演示了如何使用互斥锁解决竞争条件的问题：

```
1 #include <mutex>
2 ...
3 std::mutex locker;
4 void inc() {
5     locker.lock();
6     global = global + 1;
7     locker.unlock();
8 }
9 ...
10 std::thread t1{inc};
11 std::thread t2{inc};
```

当 t1 开始执行 inc() 时，锁定了互斥锁，这就避免了任何其他线程访问全局变量，除非原始线程没有为下一个线程解锁。

C++17 引入了锁保护，它允许保护互斥对象，以避免忘记解锁它：

```
1 std::mutex locker;
2 void inc() {
3     std::lock_guard g(locker);
4     global = global + 1;
```

避免死锁

新的问题会出现在互斥对象上，比如死锁。死锁是多线程代码的一种情况，即两个或多个线程锁定一个互斥锁，并等待另一个线程解锁另一个互斥锁。

避免死锁的常见建议是总是以相同的顺序锁定两个或多个互斥锁。C++ 提供了 std::lock() 函数，它具有相同的目的。

下面的代码演示了 swap 函数，它接受两个类型为 X 的参数。我们假设 X 有一个成员 mt，它是一个互斥对象。swap 函数的实现首先锁定左侧对象的互斥量，然后锁定右侧对象的互斥量：

```

1 void swap(X& left, X& right)
2 {
3     std::lock(left.mt, right.mt);
4     std::lock_guard<std::mutex> lock1(left.mt, std::adopt_lock);
5     std::lock_guard<std::mutex> lock2(right.mt, std::adopt_lock);
6     // do the actual swapping
7 }
```

一般而言，要避免死锁，先避免嵌套锁。也就是说，如果已经持有一个锁，就不要再获取它。如果不是这样，那么以固定的顺序获取锁可以避免死锁。

设计并发代码

当引入并发后，项目的复杂性会急剧上升。与并发代码相比，处理顺序执行的同步代码要容易得多。许多系统通过引入事件驱动的开发概念（如事件循环）来避免使用多线程。使用事件循环的目的是为异步编程引入一种可管理的方法。为了进一步推广这一概念，可以想象任何提供图形用户界面（GUI）的应用程序。每当用户单击任何 GUI 组件时，比如按钮、在字段类型，或者是移动鼠标，应用程序就会接收到所谓的关于用户操作的事件。无论它是 button_press、button_release、mouse_move 还是任何其他事件，都向应用程序表示一条信息，以便正确地做出反应。一种流行的方法是合并一个事件循环，来对用户交互期间发生的事件进行排队。

当应用程序忙于当前任务时，由用户操作产生的事件将排队等待将来的某个时间处理。处理过程包括调用附加到每个事件的处理函数，它们按放入队列的顺序被调用。

在项目中引入多线程会带来额外的复杂性。应该注意竞争条件和正确的线程处理，甚至可以使用线程池来重用线程对象。在顺序执行的代码中，只关心代码本身即可。使用多线程，更需要关心的是执行相同代码的方式。例如，单例这样的简单设计模式在多线程环境中的行为就不同。单例的经典实现如下所示：

```

1 class MySingleton
2 {
3 public:
4     static MySingleton* get_instance() {
5         if (instance_ == nullptr) {
6             instance_ = new MySingleton();
7         }
8         return instance_;
9 }
```

```

9     }
10    // code omitted for brevity
11
12    private:
13        static inline MySingleton* instance_ = nullptr;
14    };

```

下面的代码启动两个线程，都使用了 MySingleton 类：

```

1 void create_something_unique()
2 {
3     MySingleton* inst = MySingleton::get_instance();
4     // do something useful
5 }
6 void create_something_useful()
7 {
8     MySingleton* anotherInst = MySingleton::get_instance();
9     // do something unique
10 }
11 std::thread t1{create_something_unique};
12 std::thread t2{create_something_useful};
13 t1.join();
14 t2.join();
15 // some other code

```

线程 t1 和 t2 调用 MySingleton 类的 get_instance() 静态成员函数。t1 和 t2 通过了对空实例的检查，并执行 new 操作。显然，这里有一个竞争条件。资源（在本例中是类实例）应该受到保护，避免出现这种情况。这是一个使用互斥的解决方案：

```

1 class MySingleton
2 {
3     public:
4         static MySingleton* get_instance() {
5             std::lock_guard lg{mutex_};
6             if (instance_ == nullptr) {
7                 instance_ = new MySingleton();
8             }
9             return instance_;
10        }
11        // code omitted for brevity
12
13     private:
14         static std::mutex mutex_;
15         static MySingleton* instance_;
}

```

使用互斥锁可以解决这个问题，但会使函数工作得更慢，因为每次线程请求一个实例时，互斥锁就会被锁定（这涉及到 OS 内核的额外操作）。正确的解决方案是使用双重检查锁定模式。它的基本思想是：

1. 在检查完 instance_ 之后锁定互斥锁。

- 在互斥锁锁定后再次检查 `instance_`，因为另一个线程可能已经通过了第一次检查，并等待互斥锁解锁。

```

1 static MySingleton* get_instance() {
2     if (instance_ == nullptr) {
3         std::lock_guard lg{mutex_};
4         if (instance_ == nullptr) {
5             instance_ = new MySingleton();
6         }
7     }
8     return instance_;
9 }
```

多个线程可能通过第一次检查，其中一个将锁定互斥锁，所以只有一个线程到进行 `new` 操作。但是，在对互斥锁解锁之后，通过第一个检查的线程将尝试锁定互斥锁并创建实例。第二种检查是为了防止这种情况发生。前面的代码允许我们减少同步代码的性能开销。我们在这里提供的方法是为并发代码设计做准备的方法之一。

并发代码设计在很大程度上是基于语言本身的能力。C++ 最早的版本中，它没有内置的多线程支持。现在，它有了一个可靠的线程库，新的 C++20 标准为我们提供了更强大的工具，比如协程。

介绍协程

讨论 GUI 应用程序时，我们讨论了一个异步代码执行的示例。GUI 组件通过触发相应的事件来响应用户的操作，将这些事件推入事件队列。然后通过调用附加的处理程序函数逐个处理这个队列。所描述的流程在循环中发生，这就是为什么我们通常把这个概念称为事件循环。

异步系统在 I/O 操作中非常有用，因为任何输入或输出操作都会在 I/O 调用时阻塞执行。例如，下面的伪代码从一个目录中读取一个文件，然后在屏幕上打印一条欢迎消息：

```

1 auto f = read_file("filename");
2 cout << "Welcome to the app!";
3 process_file_contents(f);
```

附加到同步执行模式，我们知道消息欢迎到应用程序！只有在 `read_file()` 函数执行完毕后才会输出。

`process_file_contents()` 将在 `cout` 完成后调用。处理异步代码时，我们了解的代码行为开始变得不可预测。下面的修改版本使用 `read_file_async()` 函数异步读取文件内容：

```

1 auto p = read_file_async("filename");
2 cout << "Welcome to the app!";
3 process_file_contents(p); // we shouldn't be able to do this
```

考虑到 `read_file_async()` 是一个异步函数，消息欢迎 `Welcome to the app!` 将会比文件内容更快地打印出来。异步执行的本质允许我们调用在后台执行的函数，这为我们提供了非阻塞的输入/输出。

但是，在处理函数返回值的方式上有一个细微的变化。如果我们处理一个异步函数，它的返回值认为是一个称为 promise 或 promise 对象。这是系统在异步功能完成时通知我们的方式。promise

对象有三种状态:

- Pending
- Rejected
- Fulfilled

如果函数完成并且结果准备好被处理，那么 promise 对象就已经实现。发生错误的情况下，promise 对象将处于 rejected 状态。如果 promise 没有拒绝，也没有执行，那么就处于 pending 状态。C++20 引入协程作，可作为对异步函数的补充。

协程将代码的后台执行移动到下一个级别，它们允许一个函数在必要时被暂停和恢复。想象一个函数读取文件内容并在中间停止，将执行上下文传递给另一个函数，然后继续读取文件直到结束。因此，在深入研究之前，请将协程视为如下函数：

- 开始
- 暂停
- 重新开始
- 完成

要使函数成为协程，可以使用关键字 `co_await`、`co_yield` 或 `co_return` 之一。`co_await` 是一个告诉代码等待异步执行代码的构造。这意味着函数可以在此时挂起，并在结果就绪时继续执行。例如，下面的代码使用套接字从网络请求一张图：

```
1 task<void> process_image()
2 {
3     image i = co_await request_image("url");
4     // do something useful with the image
5 }
```

由于网络请求操作也被认为是一个输入/输出操作，它可能会阻塞代码的执行。为了防止阻塞，我们使用异步调用。前面的例子中，使用 `co_await` 的行是可以暂停函数执行的地方。简单地说，当执行到达有 `co_await` 的那一行时，会发生以下情况：

1. 暂时退出函数（直到没有准备好的数据）。
2. 它从调用 `process_image()` 之前的位置继续执行。
3. 在离开时的位置继续执行 `process_image()`。

为了实现这一点，协程 (`process_image()` 函数是一个协程) 的处理方式与 C++ 中的常规函数不同。协程的一个特性是无堆栈。我们知道函数不能没有堆栈。在这里，函数甚至在执行指令之前就推入它的参数和局部变量。另一方面，协程不是将任何东西压入堆栈，而是将它们的状态保存在堆中，并在恢复时恢复它。



也有堆的协程——堆叠协程，也称为纤维，有单独的堆栈。

协程与调用者连接，调用 `sprocess_image()` 的函数将执行转移到协程，而协程的暂停将执行转移回调用者。如前所述，堆用于存储协程的状态，但特定于函数的实际数据（参数和局部变量）存

储在调用者的堆栈中。协程与存储在调用方函数堆栈中的对象相关联。显然，协程的寿命和对象的一样长。

协程可能会给人一种添加到语言中的冗余复杂性的错误印象，但它们的用例对于改进使用异步 I/O 代码（如前面的例子）或惰性计算的应用程序非常有用。当我们发明新的模式或引入复杂的项目来处理时，例如：惰性计算，就可以通过使用 C++ 中的协程来改善我们的体验。请注意，异步 I/O 或延迟计算只是协程应用程序的两个例子，相关例子还有很多。

总结

本章中，我们讨论了并发性的概念，并展示了并行性之间的区别。我们了解了进程和线程之间的区别，后者是我们感兴趣的。多线程允许我们更有效地管理程序，尽管也带来了额外的复杂性。为了处理数据竞争，我们使用了互斥锁等同步原语。互斥锁是一种锁定线程数据的方法，以避免多个线程同时访问相同的数据，而产生无效行为。

我们还讨论了将输入/输出操作视为阻塞的思想，而异步函数是使其不阻塞的方法之一。协程作为代码异步执行的一部分是在 C++20 中引入的。

我们学习了如何创建和启动线程。更重要的是，我们学习了如何在线程之间管理数据。下一章中，我们将深入研究并发环境中使用的数据结构。

问题

1. 什么是并发？
2. 并发和并行的区别是什么？
3. 什么是进程？
4. 进程和线程有什么区别？
5. 编写代码启动线程。
6. 如何使单例模式是线程安全的？
7. 重写 MySingleton 类，使用 std::shared_ptr 返回的实例。
8. 什么是协程？co_await 用在哪里？

扩展阅读

- Anthony Williams, C++ Concurrency in Action, <https://www.amazon.com/C-Concurrency-Action-Anthony-Williams/dp/1617294691/>

第 9 章：设计并发式数据结构

前一章中，讨论了 C++ 中的并发和多线程的基础知识。并发代码设计中最大的挑战是正确处理数据竞争。线程同步和调度也不是一个容易掌握的知识点。我们可以在任何怀疑有数据竞争的地方使用同步原语，比如：互斥锁，但这并不是最佳方式。

设计并发代码的更好方法是不惜一切代价不使用锁。这不仅会提高应用程序的性能，而且会使它比以前更安全。说起来容易做起来难——无锁编程是本章的一个具有挑战性的主题。特别是，我们将进一步深入设计无锁算法和数据结构的基础知识。这是许多优秀开发者不断研究的一个课题。我们将接触无锁编程的基础知识，这将使您了解如何以一种有效的方式构造代码。阅读完本章后，读者能够更好地理解数据竞争的问题，并获得设计并发算法和数据结构所需的基本知识。这些可能对构建容错系统的设计也有帮助。

本章中，我们将了解以下内容：

- 理解数据竞争和基于锁的解决方案
- 在 C++ 中使用原子操作
- 设计无锁的数据结构

编译器要求

g++ 编译器需要添加编译选项 `-std=c++2a` 来编译本章的代码。可以从这里获取本章的源码文件：<https://github.com/PacktPublishing/Expert-CPP>

近距离观察数据竞争

如前所述，数据竞争是开发者需要不惜一切代价试图避免的情况。前一章中，我们讨论了死锁以及避免它的方法。我们在前一章中使用的最后一个例子，是创建线程安全的单例模式。假设使用一个类来创建数据库连接（一个经典的例子）。

下面是数据库连接模式的一个简单实现。每次访问数据库时，保持一个单独的连接并不是一个好做法。所以，我们重用现有的连接从程序不同的部分查询数据库：

```
1 namespace Db {
2     class ConnectionManager
3     {
4     public:
5         static std::shared_ptr<ConnectionManager> get_instance()
6         {
7             if (instance_ == nullptr) {
8                 instance_.reset(new ConnectionManager());
9             }
10            return instance_;
11        }
12        // Database connection related code omitted
13    private:
14        static std::shared_ptr<ConnectionManager> instance_{nullptr};
15    };
16 }
```

让我们更详细地讨论这个示例。前一章中，我们加入了锁来保护 `get_instance()` 函数不受数据竞争的影响。让我们详细说明一下这样做的原因。为了简化这个例子，下面是我们感兴趣部分的四行伪码：

```
1 get_instance()
2   if (_instance == nullptr)
3     instance_.reset(new)
4   return instance_;
```

现在，假设我们运行一个访问 `get_instance()` 函数的线程。我们将其命名为 Thread A，它执行的第一行是条件语句，如下所示：

```
1 get_instance()
2   if (_instance == nullptr) <-- Thread A
3     instance_.reset(new)
4   return instance_;
```

它将逐行执行指令。更让我们感兴趣的是第二个线程（标记为 Thread B），它开始执行与 Thread A 并发的函数。函数的并发执行过程中可能会出现以下情况：

```
1 get_instance()
2   if (_instance == nullptr) <-- Thread B (checking)
3     instance_.reset(new) <-- Thread A (already checked)
4   return instance_;
```

Thread B 在比较 `instance_` 和 `nullptr` 时为真。Thread A 传递了相同的检查并将 `instance_` 设置为一个新对象。虽然从 Thread A 的角度来看，一切看起来都很好，但它只是传递了条件检查，重置实例，并将转到下一行返回 `instance_`。然而，Thread B 在 `instance_` 的值改变之前比较了它。因此，Thread B 也会继续设置 `instance_` 的值：

```
1 get_instance()
2   if (_instance == nullptr)
3     instance_.reset(new) <-- Thread B (already checked)
4   return instance_;<-- Thread A (returns)
```

前面的问题是，Thread B 在 `instance_` 设置好之后，又重新设置了它。它由几个指令组成，每个指令都由一个线程按顺序执行。为了使两个线程不相互干扰，操作不应该包含一个以上的指令。

我们关心数据竞争的原因是前面代码块中的间隙，行与行之间的间隙允许线程相互干扰。因为这个解决方案可能不是正确的，所以使用同步原语（比如互斥锁）的解决方案时，应该想象所有的空隙。下面的修改使用了互斥锁和双重检查锁的模式：

```
1 static std::shared_ptr<ConnectionManager> get_instance()
2 {
3   if (instance_ == nullptr) {
4     // mutex_ is declared in the private section
5     std::lock_guard lg{mutex_};
6     if (instance_ == nullptr) { // double-checking
7       instance_.reset(new ConnectionManager());
8     }
9 }
```

```
10     return instance_;
11 }
```

下面是两个线程试图访问 `instance_` 对象时会发生的事情:

```
1 get_instance()
2     if (instance_ == nullptr)    <-- Thread B
3         lock mutex            <-- Thread A (locks the mutex)
4     if (instance_ == nullptr)
5         instance_.reset(new)
6         unlock mutex
7     return instance_
```

现在，即使两个线程都通过了第一次检查，其中一个线程也会锁定互斥锁。当一个线程试图锁定互斥锁时，另一个线程将重置实例。为了确保它没有被设置，我们使用了第二个检查(这就是为什么它被称为双重检查):

```
1 get_instance()
2     if (instance_ == nullptr)
3         lock mutex            <-- Thread B (tries to lock, waits)
4     if (instance_ == nullptr)    <-- Thread A (double check)
5         instance_.reset(new)
6         unlock mutex
7     return instance_
```

当 Thread A 完成了 `instance_` 的设置后，就会解锁互斥锁，这样 Thread B 就可以继续锁定并重置 `instance_`:

```
1 get_instance()
2     if (instance_ == nullptr)
3         lock mutex            <-- Thread B (finally locks the mutex)
4     if (instance_ == nullptr)    <-- Thread B (check is not passed)
5         instance_.reset(new)
6         unlock mutex          <-- Thread A (unlocked the mutex)
7     return instance_         <-- Thread A (returns)
```

作为经验法则，应该始终在代码的行与行之间寻找线索。两个语句之间总是有一个间隙，这个间隙会使两个或多个线程相互干扰。下一节将详细讨论数字递增的示例。

添加同步机制

几乎每一本涉及线程同步的书，都会使用了一个增加数字作为数据竞争示例。本书也不例外，示例如下:

```
1 #include <thread>
2
3 int counter = 0;
4
5 void foo()
6 {
7     counter++;
```

```

8 }
9
10 int main()
11 {
12     std::jthread A{foo};
13     std::jthread B{foo};
14     std::jthread C{[] { foo(); } };
15     std::jthread D{
16         []{
17             for (int ix = 0; ix < 10; ++ix) { foo(); }
18         }
19     };
20 }
```

我们添加了两个线程，使示例更加复杂。前面的代码只是使用四个不同的线程递增计数器变量。乍一看，在任何时间点，只有一个线程增量计数器。然而，正如我们在前一节中提到的，我们应该注意并寻找代码中的漏洞。`foo()` 函数似乎缺少一个。自增操作符的行为方式如下（伪代码）：

```

1 auto res = counter;
2 counter = counter + 1;
3 return res;
```

现在，我们发现了本不应该存在的问题。在任何时候，只有一个线程执行前面三条指令中的一条。也就是说，可能出现如下情况：

```

1 auto res = counter; <— thread A
2 counter = counter + 1; <— thread B
3 return res; <— thread C
```

例如：thread B 可能修改 `counter` 的值，而 thread A 读取了修改之前的值。这意味着 thread A 将在 thread B 已经完成该操作时，给 `counter` 分配一个新的增量值。非常混乱，我们的大脑迟早会因试图理解操作的顺序而爆炸。作为一个经典示例，我们将使用线程锁的机制来解决它。这里有一个解决方案：

```

1 #include <thread>
2 #include <mutex>
3
4 int counter = 0;
5 std::mutex m;
6
7 void foo()
8 {
9     std::lock_guard g{m};
10    counter++;
11 }
12
13 int main()
14 {
15     // code omitted for brevity
16 }
```

无论哪个线程先到达 lock_guard，都会首先锁定 mutex，如下所示：

```
1 lock mutex; <— thread A, B, D wait for the locked mutex
2 auto res = counter; <— thread C has locked the mutex
3 counter = counter + 1;
4 unlock mutex; <— A, B, D are blocked until C reaches here
5 return res;
```

使用锁的问题是性能。理论上，我们使用线程来加速程序的执行，更确切地说是数据处理。在大数据集合的情况下，使用多线程可能会大大提高程序的性能。但是，在多线程环境中，我们首先要注意并发访问，因为使用多个线程访问集合可能会导致数据的损坏。让我们来看看线程安全的堆栈实现。

实现一个线程安全的栈

回顾第 6 章中的堆栈数据结构适配器，我们将使用锁来实现线程安全的堆栈版本。栈有两个基本操作，push 和 pop。它们都修改容器的状态。堆栈本身不是容器，它是一种适配器，其包装了一个容器，并提供了一个适合访问的接口。合并线程安全，我们把 std::stack 封装在一个新类中。除了构造函数和销毁函数外，std::stack 还提供了以下函数：

- top()：访问堆栈的顶部元素
- empty()：如果堆栈为空，则返回 true
- size()：返回堆栈的当前大小
- push()：在堆栈（顶部）中插入一个新项
- emplace()：在堆栈的顶部构造一个元素
- pop()：删除堆栈的顶部元素
- swap()：将内容与另一个栈交换

我们将保持它的简单性，并将重点放在线程安全上。这里主要关注的是修改底层数据结构的函数。我们感兴趣的是 push() 和 pop() 函数。如果多个线程相互干扰，这些函数可能会破坏数据结构。因此，下面的声明是表示线程安全堆栈的类：

```
1 template <typename T>
2 class safe_stack
3 {
4 public:
5     safe_stack();
6     safe_stack(const safe_stack& other);
7     void push(T value); // we will std::move it instead of copy-referencing
8     void pop();
9     T& top();
10    bool empty() const;
11 private:
12    std::stack<T> wrappee_;
13    mutable std::mutex mutex_;
14 }
```

注意，我们将 mutex_ 声明为可变的，因为我们将它锁定在 const 函数 empty() 中。与删除 empty() 的常量相比，这可以说是一个更好的设计选择。然而，对数据成员使用“可变”表示我们做出了糟糕的设计选择。无论如何，safe_stack 的客户端代码不会太关心实现的内部细节，甚至不知道堆栈使用互斥锁来同步并发访问。

现在让我们看看它成员函数的实现和一个简短的描述。让我们从复制构造函数开始：

```
1 safe_stack::safe_stack(const safe_stack& other)
2 {
3     std::lock_guard<std::mutex> lock(other.mutex_);
4     wrappee_ = other.wrappee_;
5 }
```

注意，我们锁定了另一个堆栈的互斥锁。尽管看起来不公平，但我们需要确保在复制另一个栈的基础数据时，数据不会被修改。

接下来，让我们看看 push() 函数的实现。我们锁定互斥锁并将数据推入底层堆栈：

```
1 void safe_stack::push(T value)
2 {
3     std::lock_guard<std::mutex> lock(mutex_);
4     // note how we std::move the value
5     wrappee_.push(std::move(value));
6 }
```

几乎所有的函数都以相同的方式合并线程同步：锁定互斥锁、执行任务和解锁互斥锁。这确保了在任何时候只有一个线程在访问数据。也就是说，为了保护数据不受竞争条件的影响，我们必须确保函数不变量没有被破坏。



TIP 如果你不喜欢输入很长的 C++ 类型名，比如：std::lock_guard<std::mutex>，可以使用 using 关键字为类型创建简短的别名，例如，使用 locker = std::guard<std::mutex>；。

现在，转到 pop() 函数，我们可以修改类声明，使 pop() 直接返回堆栈顶部的值。我们这样做的主要原因是不希望访问栈顶（使用引用），然后从另一个线程中弹出数据。因此，我们将修改 pop() 函数，使其成为一个共享对象，然后返回堆栈元素：

```
1 std::shared_ptr<T> pop()
2 {
3     std::lock_guard<std::mutex> lock(mutex_);
4     if (wrappee_.empty()) {
5         throw std::exception("The stack is empty");
6     }
7     std::shared_ptr<T>
8     top_element{std::make_shared<T>(std::move(wrappee_.top()))};
9     wrappee_.pop();
10    return top_element;
11 }
```

注意，safe_stack 类的声明也应该根据 pop() 函数的改变而改变。此外，我们不再需要 top()。

设计无锁的数据结构

如果有一个线程可以保证进程进展，那么就可称其是一个无锁函数。与基于锁的函数（其中一个线程可以阻塞另一个线程，都可能在取得进展之前等待某些条件）相比，无锁状态确保有一个线程取得进展。使用数据同步原语的算法和数据结构正在阻塞，一个线程被挂起，直到另一个线程执行一个操作。这意味着在块移除（通常是为互斥锁解锁）之前，线程不能继续运行。我们的关注点在于不使用阻塞函数的数据结构和算法。我们称其中一些为无锁算法，尽管还应该区分非阻塞算法和数据结构的类型。

使用原子类型

前面，我们介绍了作为数据争用的原因的源代码行之间的间隙。当有一个包含不止一条指令的操作时，大脑就会提醒你可能出现的问题。然而，无论你如何努力使操作独立和单一，在大多数情况下，如果不将操作分解为涉及多个指令的步骤，将无法实现。C++ 通过提供原子类型来解决这个问题。

首先，为什么使用原子这个词。一般来说，我们理解原子是指不能分解成更小部分的东西。也就是说，原子操作是一种不能做一半的操作：要么做了，要么没做。原子操作的例子可能是对整数的简单赋值：

```
1 num = 37;
```

如果两个线程访问这行代码，它们都不会遇到半途而废的代码。换句话说，任务之间没有间隙。当然，如果 num 表示带有用户定义的赋值操作符的复杂对象，同样的语句可能会有很多间隙。



原子操作是一种不可分割的操作。

另一方面，非原子操作可以视为完成了一半，例子是我们前面讨论的自增操作。在 C++ 中，所有关于原子类型的操作也是原子类型。这意味着我们可以通过使用原子类型来避免行之间的间隙。在使用原子之前，我们可以通过使用互斥来创建原子操作。例如，我们可以认为下面的函数是原子的：

```
1 void foo ()  
2 {  
3     mutex.lock ();  
4     int a{41};  
5     int b{a + 1};  
6     mutex.unlock ();  
7 }
```

真正的原子操作和上面假原子操作之间的区别是，原子操作不需要锁。这实际上是一个很大的区别，因为互斥等同步机制合并了开销和性能损失。更准确地说，原子类型利用低级机制来确保指令的独立性和原子性。标准原子类型定义在 `<atomic>` 头文件中，标准原子类型也可以使用内部锁定。为了确保它们不使用内部锁定，标准库中的所有原子类型都 `is_lock_free()` 函数。



唯一没有 `is_lock_free()` 成员函数的原子类型是 `std::atomic_flag`。此类型上的操作必须是无锁的。它是一个布尔标志，大多数时候它用作实现其他无锁类型的基值。

也就是说，如果直接使用原子指令对 `obj` 进行操作，`obj.is_lock_free()` 将返回 `true`。如果返回 `false`，则表示使用了内部锁定。如果原子类型对于所有支持的硬件都是无锁的，则静态 `constexpr` 函数 `is_always_lock_free()` 返回 `true`。由于该函数是 `constexpr`，允许我们定义该类型在编译时是否无锁。这是一个很大的进步，会以一种良好的方式影响代码的结构和执行。例如，`std::atomic<int>::is_always_lock_free` 返回 `true`，因为 `std::atomic<int>` 很可能总是无锁的。



TIP 在希腊语中，`a` 的意思是不，`tomo` 的意思是切。原子这个词来自希腊语 `atomos`，翻译过来就是不可切割的。也就是说，我们认为原子是不可分割的最小单位。我们使用原子类型和操作来避免指令之间的间隙。

我们对原子类型使用特化，例如：`std::atomic<long>;`。可以参考下表来获得更方便的原子类型名称。表的左列包含原子类型，右列包含它的特化：

原子类型	特化版
<code>atomic_bool</code>	<code>std::atomic<bool></code>
<code>atomic_char</code>	<code>std::atomic<char></code>
<code>atomic_schar</code>	<code>std::atomic<signed char></code>
<code>atomic_uchar</code>	<code>std::atomic<unsigned char></code>
<code>atomic_int</code>	<code>std::atomic<int></code>
<code>atomic_uint</code>	<code>std::atomic<unsigned></code>
<code>atomic_short</code>	<code>std::atomic<short></code>
<code>atomic_ushort</code>	<code>std::atomic<unsigned short></code>
<code>atomic_long</code>	<code>std::atomic<long></code>
<code>atomic_ulong</code>	<code>std::atomic<unsigned long></code>
<code>atomic_llong</code>	<code>std::atomic<long long></code>
<code>atomic_ullong</code>	<code>std::atomic<unsigned long long></code>
<code>atomic_char16_t</code>	<code>std::atomic<char16_t></code>
<code>atomic_char32_t</code>	<code>std::atomic<char32_t></code>
<code>atomic_wchar_t</code>	<code>std::atomic<wchar_t></code>

上表表示基本原子类型。正则类型和原子类型之间的根本区别是，我们可以对它们进行操作。现在让我们更详细地讨论原子操作。

操作原子的类型

回想一下我们在前一节中讨论的间隙。原子类型的目标是消除指令之间的间隙，或者提供一些操作，这些操作负责将几个指令包装成一条指令组合在一起。以下是对原子类型的操作：

- load()
- store()
- exchange()
- compare_exchange_weak()
- compare_exchange_strong()
- wait()
- notify_one()
- notify_all()

load() 操作自动加载并返回原子变量的值。store() 用提供的非原子参数替换原子变量的值。

load() 和 store() 都类似于对非原子变量的读取和分配操作。每当我们访问一个对象的值时，我们执行一个 read 指令。例如，下面的代码打印了 double 变量的内容：

```
1 double d{4.2}; // "store" 4.2 into "d"
2 std::cout << d; // "read" the contents of "d"
```

原子类型的情况下，类似的读操作转换为：

```
1 atomic_int m;
2 m.store(42); // atomically "store" the value
3 std::cout << m.load(); // atomically "read" the contents
```

虽然前面的代码没有任何意义，但包含了这个示例来表示处理原子类型时的区别。访问原子变量应该通过原子操作来完成。下面的代码表示 load()、store() 和 exchange() 函数的定义：

```
1 T load(std::memory_order order = std::memory_order_seq_cst) const noexcept;
2 void store(T value, std::memory_order order =
3     std::memory_order_seq_cst) noexcept;
4 T exchange(T value, std::memory_order order =
5     std::memory_order_seq_cst) noexcept;
```

有一个类型为 std::memory_order 的额外参数。exchange() 函数由 store() 和 load() 函数组成，以原子的方式用提供的参数替换值，并原子地获取前一个值。

compare_exchange_weak() 和 compare_exchange_strong() 函数的原理类似。下面是它们的定义：

```
1 bool compare_exchange_weak(T& expected_value, T target_value,
2     std::memory_order order =
3     std::memory_order_seq_cst) noexcept;
4 bool compare_exchange_strong(T& expected_value, T target_value,
5     std::memory_order order =
6     std::memory_order_seq_cst) noexcept;
```

比较第一个参数 (expected_value) 与原子变量，如果它们相等，则用第二个参数 (target_value) 替换该变量。否则，它们会自动地将值加载到第一个参数中（这就是为什么它是通过引用传递的）。弱交换和强交换之间的区别是，compare_exchange_weak() 允许错误地失败（称为假失败），也就是说，即使 expected_value 等于基值，函数也认为它们不相等。这样做是因为在某些平台上，可以提升性能。

`wait()`、`notify_one()` 和 `notify_all()` 函数从 C++20 开始添加。`wait()` 函数阻塞线程，直到修改原子对象的值。它接受一个参数来与原子对象的值进行比较。如果两个值相等，则阻塞线程。要手动解除线程阻塞，可以调用 `notify_one()` 或 `notify_all()`。它们之间的区别是，`notify_one()` 至少解除一个被阻塞的操作的阻塞，而 `notify_all()` 解除所有这些操作的阻塞。

现在，让我们讨论一下在前面声明的原子类型成员函数时遇到的内存序。`memory_order` 定义了原子操作前后内存访问的顺序。当多个线程同时读写变量时，一个线程读取变量的顺序与另一个线程存储变量的顺序不同。原子操作的默认顺序是连续一致的顺序——这就是 `std::memory_order_seq_cst` 的作用。有几种类型的内存序，包括 `memory_order_relax`、`memory_order_consume`、`memory_order_acquire`、`memory_order_release`、`memory_order_acq_rel` 和 `memory_order_seq_cst`。下一节中，我们将设计一个使用默认内存顺序的原子类型的无锁堆栈。

设计一个无锁的栈

在设计堆栈时，要记住的关键事情是确保推送的值可以安全地从另一个线程返回。同样重要的是确保只有一个线程返回值。

前面几节中，我们实现了一个包装 `std::stack` 的基于锁的堆栈。我们知道堆栈不是一个真正的数据结构，而是一个适配器。通常，在实现堆栈时，我们选择向量或链表作为其底层数据结构。让我们看一个基于链表的无锁堆栈示例。将新元素推入堆栈涉及到创建一个新的列表节点，将它的 `next` 指针设置为当前的头节点，然后将头节点设置为指向新插入的节点。



如果你对“头”或“`next` 指针”这两个术语感到困惑，请回顾第 6 章，我们在其中详细讨论了链表。

单线程上下文中，所描述的步骤是正确的。但是，如果有多个线程在修改堆栈，我们就应该担心了。让我们找出 `push()` 操作的缺陷。当一个新元素被推入堆栈时，下面是三个主要步骤：

1. `node* new_elem = new node(data);`
2. `new_elem->next = head_;`
3. `head_ = new_elem;`

第 1 步中，我们将新节点插入到底层链表中。第 2 步描述我们将它插入到列表的前面——这就是为什么新节点的 `next` 指针指向 `head_`。最后，由于 `head_` 指针代表列表的起点，我们应该将其值重置为指向新添加的节点，就像第 3 步中那样。

节点类型是我们在堆栈中用来表示列表节点的内部结构。下面是它的定义：

```
1 template <typename T>
2 class lock_free_stack
3 {
4     private:
5         struct node {
6             T data;
7             node* next;
8             node(const T& d) : data(d) {}
9         };
}
```

```
10 node* head_;  
11 // the rest of the body is omitted for brevity  
12 };
```

我们建议您做的第一件事是在当前代码中寻找问题——不是在前面的代码中，而是在我们描述的将新元素放入堆栈的步骤中。假设两个线程同时在添加节点。第 2 步中的一个线程将新元素的 next 指针设置为指向 head_。另一个线程使 head_ 指针指向另一个新元素。很明显，这可能会导致数据损坏。对于一个线程来说，步骤 2 和步骤 3 有相同的 head_ 是至关重要的。为了解决步骤 2 和步骤 3 之间的竞争条件，我们应该使用原子比较/交换操作来保证在之前读取 head_ 的值时 head_ 没有修改。由于需要原子地访问 head 指针，下面是如何修改 lock_free_stack 类中的 head_ 成员：

```
1 template <typename T>  
2 class lock_free_stack  
3 {  
4     private:  
5         // code omitted for brevity  
6         std::atomic<node*> head_;  
7         // code omitted for brevity  
8 };
```

下面是我们如何实现原子 head 指针的无锁 push():

```
1 void push(const T& data)  
2 {  
3     node* new_elem = new node(data);  
4     new_elem->next = head_.load();  
5     while (!head_.compare_exchange_weak(new_elem->next, new_elem));  
6 }
```

我们使用 compare_exchange_weak() 来确保 head_ 指针与我们在 new_elem->next 中存储的值相同。如果是，则将其设置为 new_elem。当 compare_exchange_weak() 执行成功，就可以确定节点已经成功插入到列表中。

看看我们如何使用原子操作访问节点。类型为 `T - std::atomic<T*>` -的指针的原子形式提供了相同的接口。除此之外，`std::atomic<T*>` 提供了指向算术操作 `fetch_add()` 和 `fetch_sub()` 的指针。它们对存储的地址进行原子加法和减法运算。这里有一个例子：

```
1 struct some_struct {};  
2 any arr[10];  
3 std::atomic<some_struct*> ap(arr);  
4 some_struct* old = ap.fetch_add(2);  
5 // now old is equal to arr  
6 // ap.load() is equal to &arr[2]
```

我们故意将该指针命名为 old，因为 `fetch_add()` 将该数字添加到指针的地址中，并返回旧值。这就是为什么 old 和 arr 指向同一个地址。

下一节中，我们将介绍有关原子类型的更多操作。现在，让我们回到我们的无锁堆栈。要 pop() 一个元素，即移除一个节点，我们需要读取 head_ 并将其设置为 head_ 的 next 元素，如下所示：

```

1 void pop(T& popped_element)
2 {
3     node* old_head = head_;
4     popped_element = old_head->data;
5     head_ = head_->next;
6     delete old_head;
7 }

```

现在，假设有几个线程并发地执行它。如果两个线程从栈中移除项，读取了相同的 head_ 值，该怎么办？这个和其他一些竞态条件产生了下面的实现：

```

1 void pop(T& popped_element)
2 {
3     node* old_head = head_.load();
4     while (!head_.compare_exchange_weak(old_head, old_head->next));
5     popped_element = old_head->data;
6 }

```

我们在前面的代码中应用了与 push() 函数几乎相同的逻辑。前面的代码并不完美，应该得到加强。我们建议您努力修改它，以消除内存泄漏。

我们已经看到，无锁实现严重依赖于原子类型和操作。我们在上一节中讨论的操作并不是终点。现在让我们来发现更多的原子操作。

原子类型的更多操作方式

在上一节中，我们在指向用户定义类型的指针上使用了 std::atomic<>。也就是说，我们为 list 节点声明了以下结构：

```

1 // the node struct is internal to
2 // the lock_free_stack class defined above
3 struct node
4 {
5     T data;
6     node* next;
7 };

```

节点结构是一个用户定义的类型。在上一节中我们实例化了 std::atomic<node*>，但是我们可以用同样的方式，为几乎任何用户定义的类型实例化 std::atomic<>，即 std::atomic<T>。但是，应该注意到 std::atomic<T> 的接口仅限于以下函数：

- load()
- store()
- exchange()
- compare_exchange_weak()
- compare_exchange_strong()
- wait()
- notify_one()

- `notify_all()`

现在，让我们根据底层类型的细节，来查看原子类型上可用的操作的完整列表。

`std::atomic<T>` 实例化为整型（例如整数或指针），具有以下操作以及我们前面列出的操作：

- `fetch_add()`
- `fetch_sub()`
- `fetch_or()`
- `fetch_and()`
- `fetch_xor()`

此外，除了自增（`++`）和自减（`-`）之外，还可以使用以下操作符：`+=`、`-=`、`|=`、`&=` 和 `^=`。

最后，一个特殊的原子类型叫做 `atomic_flag`，它有两个可用的操作：

- `clear()`
- `test_and_set()`

对 `std::atomic_flag` 使用原子操作时，需要使用 `clear()` 函数清除它，而 `test_and_set()` 将值更改为 `true`，并返回前一个值。

总结

应该考虑一下使用原子操作的 `std::atomic_flag`。本章的测试中，我们介绍了一个相当简单的堆栈例子，当然还有更复杂的例子需要研究和遵循。当我们讨论设计并发栈时，我们考虑了两个版本，其中一个表示无锁堆栈。与基于锁的解决方案相比，无锁的数据结构和算法是开发者的最终目标，因为它们提供了避免数据竞争的机制，甚至不需要同步资源。

还介绍了可以在项目中使用的原子类型和操作，以确保指令是不可分割的。如果一条指令是原子的，就不需要担心它的同步。我们强烈建议读者继续研究这个主题，并构建更健壮和复杂的无锁数据结构。下一章中，我们将看到如何设计实际使用的应用程序。

问题

1. 为什么要在多线程的单例实现中检查两次实例？
2. 在实现基于锁的堆栈的复制构造函数时，我们锁定了另一个堆栈的互斥锁。为什么？
3. 什么是原子类型和原子操作？
4. 为什么我们对原子类型使用 `load()` 和 `store()`？
5. `std::atomic<T*>` 支持哪些操作？

扩展阅读

- Concurrent Patterns and Best Practices by Atul Khot, at <https://www.packtpub.com/application-development/concurrent-patterns-and-best-practices>
- Mastering C++ Multithreading by Maya Posch, at <https://www.packtpub.com/application-development/mastering-c-multithreading>

第 10 章：设计实际程序

实际项目中使用编程语言，是学习语言的一个步骤。有时，本书中的简单例子可能采用不同的方法，或者在实际程序中面临许多困难。理论需要与实践相结合。与学习语法、解决书本上的问题或理解书本上一些简单的例子不同，当创建实际的应用程序时，我们会面临一系列不同的挑战，有时书籍缺乏理论来支持实际问题。

我们将尝试梳理 C++ 实际编程的知识，这将帮助你更好地处理实际的应用程序。复杂的项目需要大量的思考和设计。有时候，因为开发者在开发之初做出了糟糕的设计选择，所以需要从头开始重写项目。本章描述了软件的设计过程。读者将了解为项目构建架构的步骤。

本章中，我们将了解以下内容：

- 了解项目开发周期
- 设计模式及应用
- 领域驱动设计
- 设计一个 Amazon 的克隆版本作为一个真实项目

编译器要求

g++ 编译器需要添加编译选项 `-std=c++2a` 来编译本章的代码。可以从这里获取本章的源码文件：<https://github.com/PacktPublishing/Expert-CPP>

项目开发周期

当处理一个问题时，应该仔细考虑需求分析的过程。项目开发中最大的错误之一是，在开始编码时没有对问题本身进行彻底的分析。

想象一下这样一个场景，你的任务是创建一个计算器，允许用户对数字进行算术计算的简单工具。假设你奇迹般地按时完成了项目并发布了程序。现在，用户开始使用计算器，他们迟早会发现他们的计算结果不会超过整数的最大值。当他们抱怨这个问题时，就可以准备用可靠的编码来辩解，比如：它是因为在计算中使用了 `int` 数据类型。这对于你和你的程序员伙伴来说是完全可以理解的，但是最终用户就是不能接受你的观点。他们想要一个能够对大数字求和的工具，否则，他们根本不会使用你的程序。开始开发下一个版本的计算器时，将使用长数或甚至自定义实现的大数。当你自豪的将第二版程序交付给等待已久的用户时，用户可能会抱怨没有找到数字的对数或指数的功能。这就令人生畏了，因为可能会有越来越多的特性请求和越来越多的抱怨。

虽然这个例子有点简单，但它完全涵盖了现实世界中经常发生的事情。即使开发者实现了程序的所有特性，并且正在考虑休假，用户也会开始抱怨程序中的错误。结果表明，有几种情况下，计算器的行为异常，没有给出或给出了错误的结果。迟早，您会意识到，在将程序发布给大众之前，需要的是适当的测试。

我们将涉及在实际项目中工作时应该考虑的主题。当开始一个新项目时，应该考虑以下步骤：

1. 需求收集和分析
2. 创建规范
3. 设计和测试计划
4. 编码

5. 测试和稳定性

6. 发布和维护

前面的步骤并不是针对每个项目都固定的，应该认为是每个软件开发团队为了实现成功的产品发布而应该完成的最低要求。现实中，大部分的步骤都被省略了，因为在 IT 领域的每个人都缺乏一件东西——时间。但是，强烈建议遵循上述步骤，因为从长远来看，它会节省更多的时间。

需求收集和分析

这是创建稳定产品的最关键的一步。开发者不能按时完成任务，或在代码中留下大量 bug 的一个最普遍的原因是缺乏对项目的完整理解。

领域知识是如此重要，不应该被忽略。在任何情况下，你可能会很幸运地开发与你非常了解的东西相关的项目。然而，你应该考虑到不是每个人都像你一样幸运。

假设您正在进行一个项目，该项目将自动分析和报告某个公司的股票交易信息。现在想象一下，你对股票和股票交易一无所知。你不知道熊市或牛市，交易的局限性等等。你将如何成功地完成这个项目？

即使您了解股票市场和交易，您也可能不知道您的下一个大型项目域。如果你的任务是设计和执行（有或没有团队）一个控制城市气象站的项目，那该怎么办？当你开始这个项目时，你首先要做些什么？

你肯定应该从需求收集和分析开始。这只是一个涉及到与客户沟通和问很多关于项目的问题的过程。如果你在一家产品公司工作，而不是与任何客户打交道，那么项目经理就应该视为客户。即使这个项目是你的想法，你是独自工作，你也应该把自己当成客户，尽管这听起来可能有些荒谬，但你应该问自己很多问题（关于这个项目）。

假设我们将征服电子商务，并希望发布一种产品，最终将在自己的业务领域击败市场大鳄。受欢迎和成功的电子商务市场有亚马逊、eBay、阿里巴巴等。我们应该把这个问题表述为，编写自己的 Amazon 版本。我们应该如何收集这个项目的需求？

首先，我们应该列出所有我们应该实现的功能，然后我们将优先级排序。例如，对于 Amazon 的项目，我们可能会得到以下特性列表：

- 创建产品
- 展示产品
- 购买产品
- 修改产品的信息
- 删除产品
- 通过名称、价格范围和重量搜索产品
- 每隔一段时间通过电子邮件告知用户产品的可用性

应该尽可能详细地描述功能，这将为开发人员解决问题。例如，创建产品应该由项目管理员或任何用户完成。如果用户可以创建产品，则应该有限制。用户可能会错误地在我们的系统中创建数百个产品，以增加他们唯一产品的可见性。

与客户沟通的过程中，应陈述、讨论和最终确定细节。如果项目中只有你一人，并且你是项目的客户，那么交流就是你自己对项目需求进行思考的过程。

当完成获取需求时，我们建议对每个特性进行优先级排序，并将它们分为以下类别之一：

- 刚性要求
- 应该拥有
- 最好具有

进一步思考并对上述特性进行分类后，我们可以得出以下列表：

- 创建产品 [刚性要求]
- 展示产品 [刚性要求]
- 购买产品 [刚性要求]
- 修改产品的信息 [应该拥有]
- 删除产品 [刚性要求]
- 通过名称搜索产品 [刚性要求]
- 通过价格范围搜索产品 [应该拥有]
- 通过重量搜索产品 [最好具有]
- 每隔一段时间通过电子邮件告知用户产品的可用性 [最好具有]

分类会让你大致知道从哪里开始。开发者是贪婪的人，他们想为自己的产品实现所有可能的功能，这肯定会导致失败。你应该先从最基本的功能开始——这就是为什么会有一些很好的功能。

创建规范

并不是每个人都喜欢创建规范说明。大多数开发者讨厌这个步骤，因为它不是编码，而是编写文档。

收集项目需求之后，应该创建一个文档，其中包括描述项目的每个细节。该规范有许多名称和类型。可以称为项目需求文档 (PRD)、功能规范、开发规范等。认真的开发者和认真的团队根据需求分析生成一个 PRD，下一步是创建功能规范和开发规范等。我们将所有的文档组合在一个名为规范创建的步骤中。

你和你的团队可以决定是否需要前面提到的任何子文档，用可视化的产品表示比用文本文档更好。无论文档采用何种形式，它都应该仔细地表示在需求收集步骤中取得的成果。为了对此有一个基本的理解，让我们尝试着记录一些之前收集到的特性 (我们把这个项目称为 `the platform`)

- 创建一个产品。拥有管理员权限的平台用户可以创建产品。
- 平台必须允许创建具有定义权限的用户。此时，应该有两种类型的用户，即普通用户和管理员用户。
- 任何使用该平台的用户都必须能够看到可用产品的列表。
- 产品应该有图片、价格、名称、重量和描述。
- 要购买一个产品，用户提供他们的卡的详细信息来兑现和产品发货的详细信息。
- 每个注册用户应提供送货地址、信用卡详细信息和电子邮件帐户。

列表可能很长，实际上也应该很长，因为列表越长，开发人员对项目的理解就越多。

设计和测试计划

尽管我们坚持认为需求收集步骤是软件开发中最重要的步骤，但是设计和测试计划也是同样重要的步骤。尽管励志格言坚持认为没有什么是不可能的，但开发者们确信至少有一件事是不可能的，

那就是不先设计就成功地完成一个项目。

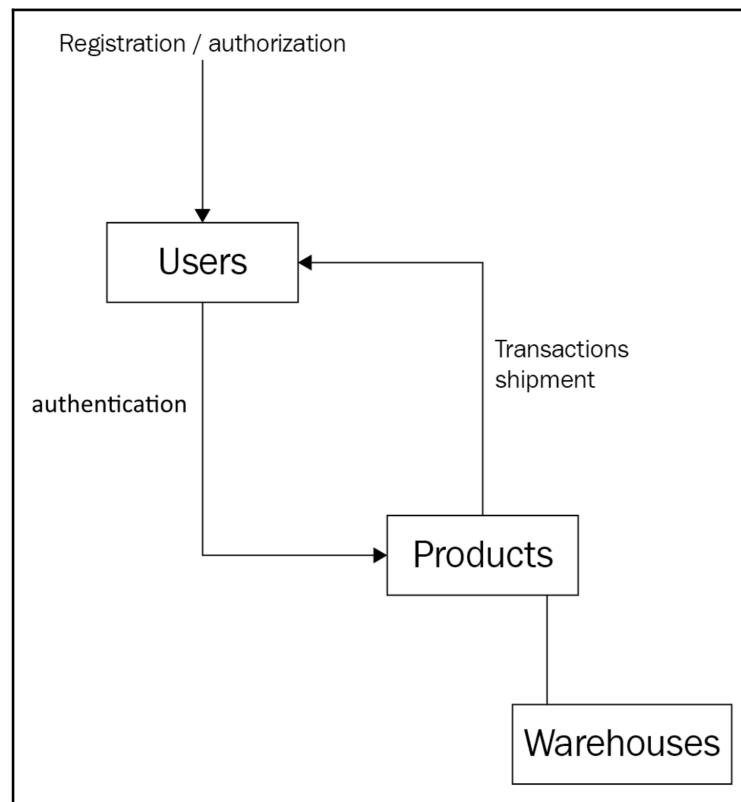
设计的过程是最有趣的步骤，它迫使我们思考、描绘、再思考、清除一切，然后重新开始。项目的许多特性是在设计时发现的。设计一个项目，应该从头开始。首先，列出以某种方式与项目相关的所有实体和过程。对于 Amazon 克隆的例子，我们可以列出以下过程：

- 用户
- 注册和授权
- 产品
- 交易
- 仓库 (包含产品)
- 装运

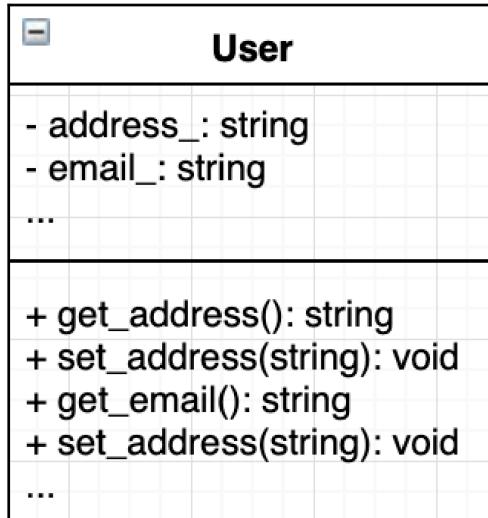
这是一个高层次的设计——贯穿最终设计的起点。本章中，我们将主要集中在项目的设计上。

需求分解

列出关键步骤和流程之后，我们将它们分解为更详细的步骤，这些步骤稍后将转换为类。最好是画出项目的设计草图，只需绘制包含实体名称的矩形，如果以某种方式连接在一起或是同一过程的一部分，则用箭头连接它们，这是一个更好的理解项目必要的一步。例如，请看下面的图表：



把实体和过程分解成类，它们之间的相互交流就是一门艺术，需要耐心和一致性。例如，我们尝试添加用户实体的详细信息。如创建规范步骤中所述，注册用户应该提供送货地址、电子邮件地址和信用卡详细信息。让我们画一个表示用户的类图：



有一个有趣的问题：我们如何处理复杂类型？例如，用户的投递地址是复杂类型。它不可能只是字符串，因为我们可能需要根据用户的送货地址进行排序，以实现最佳的发货顺序。如果用户的送货地址与包含购买产品的仓库地址在不同的国家，那么货运公司可能会让我们（或用户）花费一大笔钱。这是一个很好的场景，它引入了一个新的问题，并更新了我们对项目的理解。当用户订购的产品分配到离用户很远的仓库时，我们应该处理这种情况。如果我们有很多仓库，我们应该选择一个距离用户最近的。这些问题不能马上得到具体的回答，但这可以保证项目的高质量。否则，这些问题就会在编码过程中出现，我们困在这些问题上的时间，就要比我们想象的要长。

现在，如何将用户地址存储在 User 类中？一个简单的 std::string 就可以了，如下面的例子所示：

```

1 class User
2 {
3     public:
4         // code omitted for brevity
5     private:
6         std :: string address_;
7         // code omitted for brevity
8 }

```

就其组件而言，地址是一个复杂的对象。地址可以包含国家名称、国家代码、城市名称和街道名称，甚至还可以包含纬度和经度。如果您需要找到离用户最近的仓库，那么越详细越好。制作更多类型，让开发者的设计更直观，这完全没有问题。例如，下面的结构体可能很适合描述用户的地址：

```

1 struct Address
2 {
3     std :: string country;
4     std :: string city;
5     std :: string street;
6     float latitude {};
7     float longitude {};
8 }

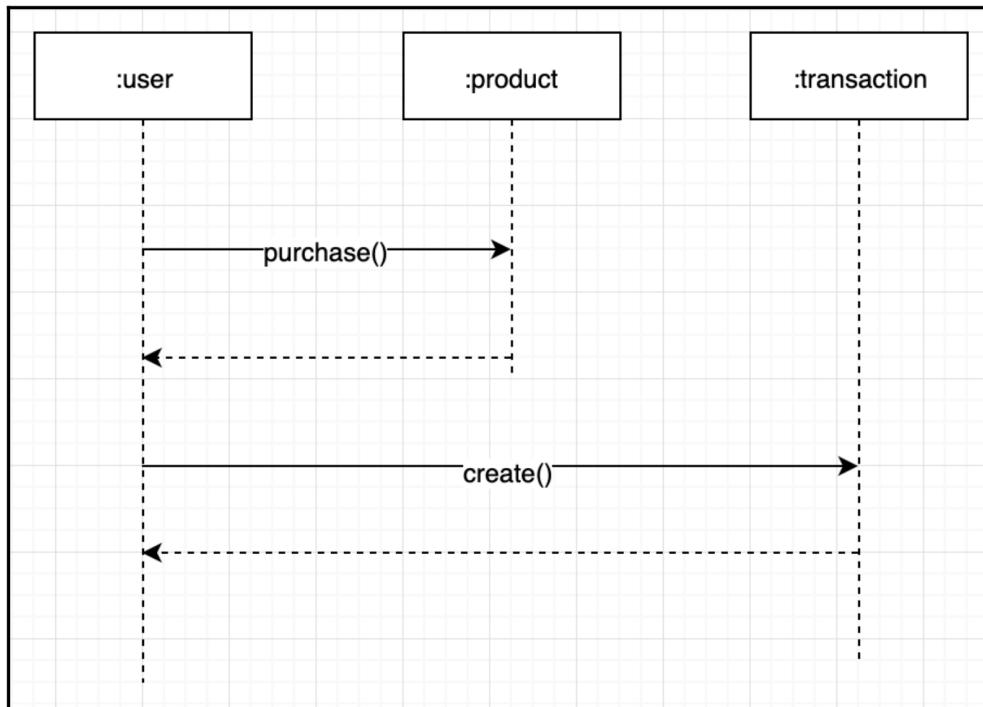
```

现在，存储用户地址变得更加简单：

```
1 class User
2 {
3     // code omitted for brevity
4     Address address_;
5 };
```

设计项目的过程，可能需要有几个步骤重申项目需求。用前面的步骤澄清设计步骤之后，我们可以继续将项目分解为更小的组件，最好也都创建交互图。

如下所示的交互图，将描述用户购买产品时所做的交易等操作：



测试计划也可以认为是设计的一部分，包括如何测试最终应用程序的计划，例如：在此之前的步骤包括一个地址的概念，地址可以包含国家、城市等。正确的测试应该包括，检查是否可以在用户地址中成功设置国家。虽然测试计划通常不认为是开发者的任务，但对项目来说，它是一种很好的实践。在设计项目时，适当的测试计划将产生更多有用的信息。大多数输入数据处理和安全检查都在测试计划中发现，例如：在进行需求分析或编写功能规范时，对用户名或电子邮件地址设置严格的限制可能并不适用。测试计划会关注这样的场景，并迫使开发人员进行数据检查。然而，大多数开发人员并没有耐心做这些，而是直接进入项目开发的下一个步骤——编码。

编码

如前所述，编码并不是项目开发的唯一。编码之前，应该通过规范中的需求来仔细设计项目。在完成前面的项目开发步骤后，编码将变得容易和高效。

一些团队实践测试驱动开发 (TDD)，这是产生更稳定项目的好方法。TDD 的主要概念是在项目实现之前编写测试。这是开发者定义项目需求和回答开发过程中出现问题的好方法。

假设我们正在为 User 类实现 setter。User 对象包含电子邮件字段，所以应该也有一个 set_email() 方法，如下所示：

```
1 class User
2 {
3     public:
4         // code omitted for brevity
5         void set_email(const std::string&);
6     private:
7         // code omitted for brevity
8         std::string email_;
9 };
```

TDD 方法建议在实现 set_email() 之前，为 set_email() 方法编写一个测试函数。测试函数如下所示：

```
1 void test_set_email()
2 {
3     std::string valid_email = "valid@email.com";
4     std::string invalid_email = "112%$";
5     User u;
6     u.set_email(valid_email);
7     u.set_email(invalid_email);
8 }
```

前面的代码中，我们声明了两个字符串变量，其中一个包含无效的电子邮件地址值。在运行测试函数之前，我们就知道在无效数据输入的情况下，set_email() 方法应该以某种方式做出反应。常见的方法是抛出一个提示无效输入的异常。可以忽略 set_email 实现中的无效输入，并返回一个布尔值，指示操作的成功。错误处理应该在项目中保持一致，并得到所有团队成员的同意。考虑一下抛出异常，测试函数在处理无效值时应该期待异常。

上面的代码应该重写，如下所示：

```
1 void test_set_email()
2 {
3     std::string valid_email = "valid@email.com";
4     std::string invalid_email = "112%$";
5
6     User u;
7     u.set_email(valid_email);
8     if (u.get_email() == valid_email) {
9         std::cout << "Success: valid email has been set successfully" <<
10         std::endl;
11     } else {
12         std::cout << "Fail: valid email has not been set" << std::endl;
13     }
14
15     try {
16         u.set_email(invalid_email);
17         std::cerr << "Fail: invalid email has not been rejected" << std::endl;
18     } catch (std::exception& e) {
19         std::cout << "Success: invalid email rejected" << std::endl;
20     }
21 }
```

```
20 }  
21 }
```

测试功能似乎已经完成。无论何时运行测试函数，都会输出 `set_email()` 的当前状态。即使我们还没有实现 `set_email()` 函数，相应的测试函数也是实现细节的一大步。现在有了函数实现，所以了解了如何对有效和无效的数据输入作出反应。我们可以添加更多类型的数据，以确保 `set_email()` 方法在实现完成时进行彻底的测试。例如，可以用空字符串和长字符串来测试它。

下面是 `set_email()` 方法的初始实现：

```
1 #include <regex>  
2 #include <stdexcept>  
3  
4 void User::set_email(const std::string& email)  
5 {  
6     if (!std::regex_match(email,  
7         std::regex("(\\w+)(\\.|_)?(\\w*)@((\\w+)(\\.((\\w+))+))")) {  
8         throw std::invalid_argument("Invalid email");  
9     }  
10  
11     this->email_ = email;  
12 }
```

初始实现之后，我们应该再次运行我们的测试函数，以确保实现与定义的测试用例一致。



为项目编写测试被认为是一种良好的编码实践。有不同类型的测试，比如：单元测试、回归测试、冒烟测试等。开发人员应该支持他们项目的单元测试覆盖率。

编码过程是项目开发生命周期中最混乱的步骤之一。很难估计实现一个类或它的方法需要多长时间，因为大多数问题和困难都出现在编码过程中。本章开头所描述的项目开发生命周期的前面步骤，往往涵盖了这些问题中的大部分，并简化了编码过程。

测试和稳定性

项目完成后，应该对其进行适当的测试。通常，软件开发公司有质量保证 (QA) 工程师，他们会一丝不苟地测试项目。

测试阶段验证的问题会转换成分配给开发者的相应任务，问题可能会影响项目的发布。

开发者的基本任务不是立即修复问题，而是找到问题的根本原因。为了简单起见，让我们看一下 `generate_username()` 函数，它使用随机数结合电子邮件来生成用户名：

```
1 std::string generate_username(const std::string& email)  
2 {  
3     int num = get_random_number();  
4     std::string local_part = email.substr(0, email.find('@'));  
5     return local_part + std::to_string(num);  
6 }
```

`generate_username()` 函数调用 `get_random_number()`，将返回的值与电子邮件地址的私有部分结合起来。私有部分是电子邮件地址中 @ 符号之前的部分。

QA 工程师报告说，邮件中附在私有部分的数字一直是一样的。例如，对于电子邮件 `john@gmail.com`，生成的用户名是 `john42`，对于 `amanda@yahoo.com`，生成的用户名是 `amanda42`。因此，下一次使用电子邮件 `amanda@hotmail.com` 的用户试图在系统中注册时，生成的用户名 `amanda42` 与已经存在的用户名冲突。测试人员完全不知道项目的实现细节，所以他们会在用户名生成功能中报告这个问题。虽然您可能已经猜到真正的问题隐藏在 `get_random_number()` 函数中，但总是存在这样的情况，问题修复了，但没找到根本原因。修复问题的错误方法可能会改变 `generate_username()` 函数的实现。`generate_random_number()` 函数也可能在其他函数中使用，这反过来会使所有调用 `get_random_number()` 的函数得到错误结果。尽管例子很简单，但深入思考并找到问题背后的真正原因才是至关重要的。

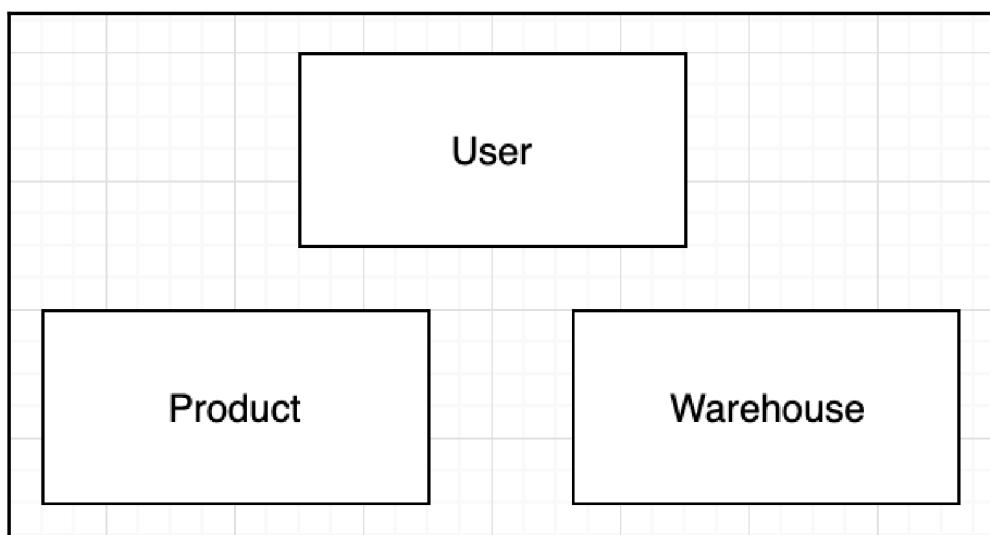
发布和维护

通过修复所有关键和主要问题，项目稳定之后，就可以发布了。有时候，公司发布的软件打着测试版的标签，这样就有了一个借口，以防用户发现软件漏洞百出。需要注意的是，很少有软件能够完美地工作。在发布之后，会出现更多的问题。因此，当开发人员致力于修复和发布更新时，就会进入维护阶段。

开发者有时开玩笑说，发布和维护是从未实现的步骤。然而，如果花了足够的时间来设计项目，那么发布第一个版本就不会花太多时间。正如我们在前一节已经介绍过的，设计从收集需求开始。之后，我们花时间定义规范，分解它们，分解成更小的组件，编码，测试，最后发布它。作为开发人员，我们对设计和编码阶段更感兴趣。如前所述，一个好的设计选择对进一步的项目开发有很大的影响。现在让我们从整体上仔细看看设计过程。

深入设计过程

如前所述，在设计电子商务平台时，项目设计从列出一般实体开始，如用户、产品和仓库：



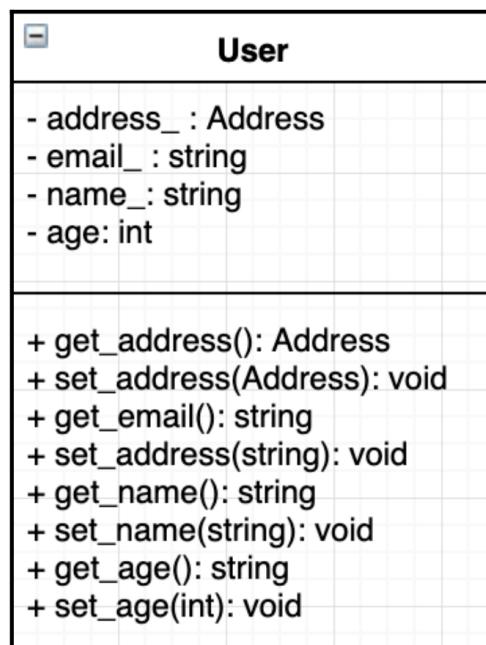
然后我们将每个实体分解成更小的组件。为了让事情更清楚，把每个实体看作一个单独的类。当把一个实体看作一个类时，它在分解方面更有意义。例如，我们将 `User` 实体表示为一个类：

```

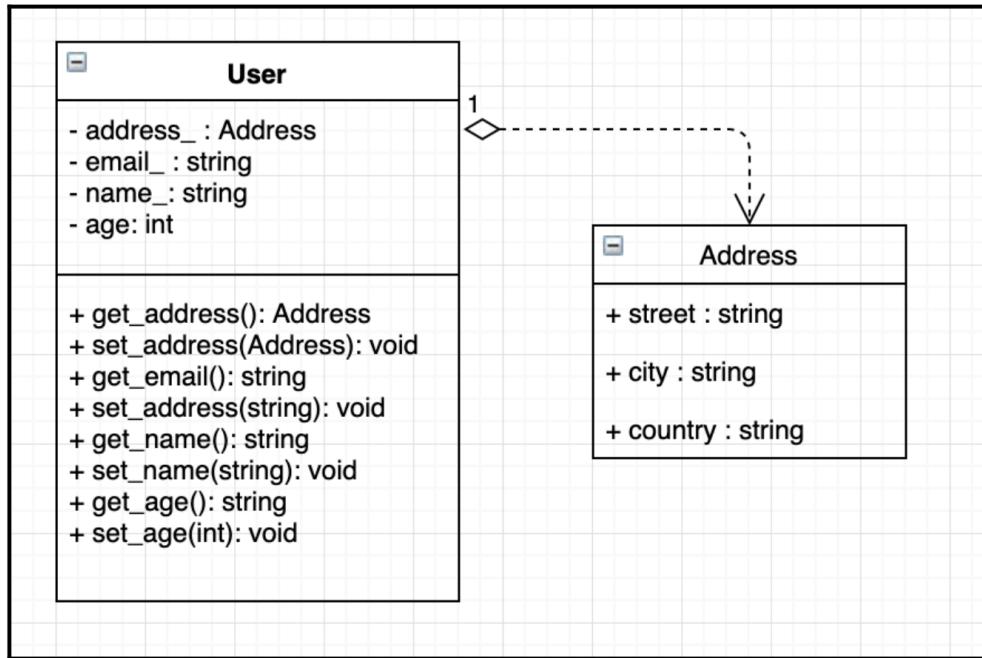
1 class User
2 {
3 public:
4 // constructors and assignment operators are omitted for code brevity
5 void set_name(const std::string& name);
6 std::string get_name() const;
7 void set_email(const std::string& );
8 std::string get_email() const;
9 // more setters and getters are omitted for code brevity
10
11 private:
12 std::string name_;
13 std::string email_;
14 Address address_;
15 int age;
16 };

```

User 类的类图如下:



然而，正如我们已经讨论过的，User 类的 address 字段可以表示为一个单独的类型 (class 或 struct，这还不是很重要)。无论是数据聚合还是复杂类型，类图都需要进行以下更改：



这些实体之间的关系将在设计过程中变得清晰。例如，地址本身不是一个实体，它是用户的一部分。也就是说，如果用户对象没有实例化，它就不能有实例。然而，正如我们可能希望指向可重用代码，地址类型也可以用于仓库对象。也就是说，用户和地址之间的关系是简单的聚合，而不是组合。

在讨论支付选项时，我们可以对 User 类型提出更多要求。该平台的用户应该能够插入为产品付费的选项。在决定如何在 User 类中表示支付选项之前，应该找出这些选项到底是什么。简单起见，假设支付选项包含信用卡号、持卡人姓名、到期日期和卡的安全码。这听起来像是另一种数据聚合，所以让我们将所有这些都收集到一个结构中，如下所示：

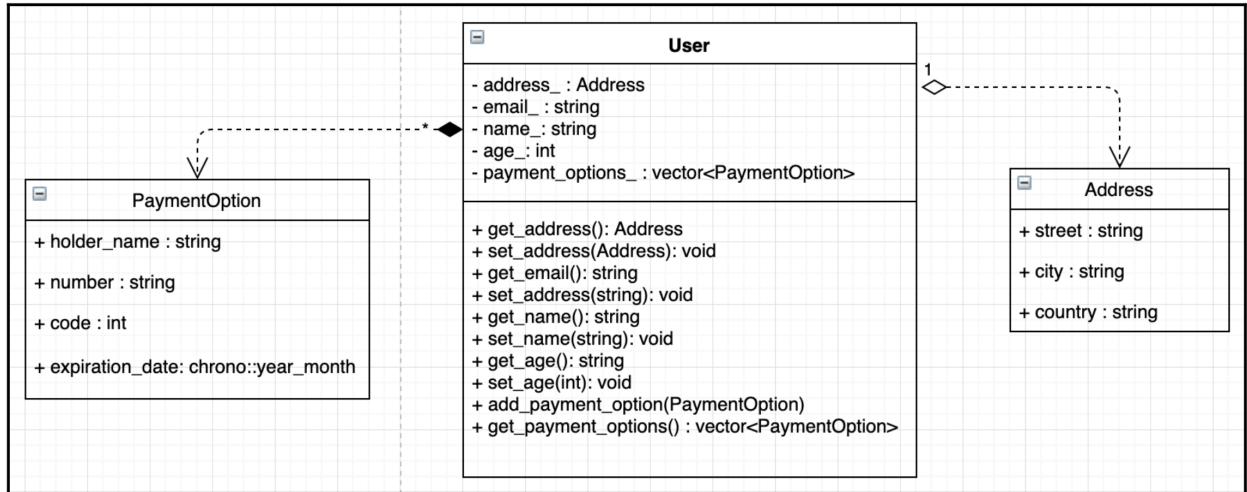
```

1 struct PaymentOption
2 {
3     std :: string number;
4     std :: string holder_name;
5     std :: chrono :: year_month expiration_date;
6     int code;
7 };

```

注意 std::chrono::year_month 在前面的结构体中，表示特定年份的特定月份 (C++20)。大多数支付卡只携带卡过期的月份和年份，所以这个 std::chrono::year_month 功能对于 PaymentOption 是完美的。

因此，在设计用户类的过程中，我们提出了一个新的类型——PaymentOption。一个用户可以有多个支付选项，因此 User 和 PaymentOption 之间的关系是一对多。现在让我们用这个新的聚合来更新我们的用户类图 (这个例子中我们使用的是复合)：



User 和 PaymentOption 之间的依赖关系如下代码所示:

```

1 class User
2 {
3     public:
4         // code omitted for brevity
5         void add_payment_option(const PaymentOption& po) {
6             payment_options_.push_back(po);
7         }
8         std::vector<PaymentOption> get_payment_options() const {
9             return payment_options_;
10        }
11    private:
12        // code omitted for brevity
13        std::vector<PaymentOption> payment_options_;
14    };

```

即使用户可能设置了多个支付选项，我们也应该将其中一个标记为主要支付选项。这很棘手，因为我们可以将所有选项存储在 vector 中，但现在我们必须将其中一个选项作为主选项。

我们可以使用 pair 或 tuple(如果比较花哨的话)，将 vector 中的选项映射为布尔值，以指示它是否是主值。下面的代码描述了前面介绍的 User 类中 tuple 的用法:

```

1 class User
2 {
3     public:
4         // code omitted for brevity
5         void add_payment_option(const PaymentOption& po, bool is_primary) {
6             payment_options_.push_back(std::make_tuple(po, is_primary));
7         }
8
9         std::vector<std::tuple<PaymentOption, bool>> get_payment_options()
10        const {
11            return payment_options_;
12        }
13    private:

```

```
14 // code omitted for brevity
15 std::vector<std::tuple<PaymentOption, boolean>> payment_options_;
16 };
```

我们可以通过以下方式，利用类型别名来简化代码：

```
1 class User
2 {
3 public:
4     // code omitted for brevity
5     using PaymentOptionList = std::vector<std::tuple<PaymentOption, boolean>>;
6
7     // add_payment_option is omitted for brevity
8     PaymentOptionList get_payment_options() const {
9         return payment_options_;
10    }
11
12
13 private:
14     // code omitted for brevity
15     PaymentOptionList payment_options_;
16 };
```

下面是 User 类如何检索用户的主要支付选项：

```
1 User john = get_current_user(); // consider the function is implemented and works
2 auto payment_options = john.get_payment_options();
3 for (const auto& option : payment_options) {
4     auto [po, is_primary] = option;
5     if (is_primary) {
6         // use the po payment option
7     }
8 }
```

我们在访问 for 循环中的元组项时使用了结构化绑定。然而，在学习了关于数据结构和算法的章节之后，现在意识到搜索主要支付选项是一个线性操作。每次我们需要检索主要支付选项时都要遍历 vector，这可能是一种糟糕的做法。



TIP 您可以更改底层的数据结构，以提高运行速度。元素的访问是常量时间复杂度时，速度会变得更快。在这个场景中，我们应该将一个布尔值映射为支付选项。对于除一个选项外的所有选项，布尔值都是相同的值。它将导致哈希冲突，这种冲突将通过将映射到相同散列表值的值链接一起来处理。使用哈希表的唯一好处是对主要支付选项的固定时间访问。

最后，我们给出了在类中单独存储主要支付选项的最简单的解决方案。以下，是我们应该如何重写用户类中的支付选项处理部分：

```
1 class User
2 {
```

```

3 public:
4 // code omitted for brevity
5 using PaymentOptionList = std::vector<PaymentOption>;
6 PaymentOption get_primary_payment_option() const {
7     return primary_payment_option_;
8 }
9
10 PaymentOptionList get_payment_options() const {
11     return payment_options_;
12 }
13
14 void add_payment_option(const PaymentOption& po, bool is_primary) {
15     if (is_primary) {
16         // moving current primary option to non-primaries
17         add_payment_option(primary_payment_option_, false);
18         primary_payment_option_ = po;
19         return;
20     }
21     payment_options_.push_back(po);
22 }
23
24 private:
25 // code omitted for brevity
26 PaymentOption primary_payment_option_;
27 PaymentOptionList payment_options_;
28 };

```

目前为止，我们经历了定义存储支付选项的方法的过程，只是为了展示编码之前的设计过程。虽然我们已经为单一情况下的支付选项创造了许多版本，但都不是最终版本。在支付选项向量中总是存在处理重复值的情况。当向用户添加一个支付选项作为主要选项，然后再添加另一个选项作为主要选项时，前一个选项将进入非主要列表。如果我们改变主意，再次将旧的支付选项添加为主要选项，其不会从非主要选项列表中删除。

所以，总有机会深入思考，避免潜在的问题。设计和编码是相辅相成的，也别忘了 TDD。大多数情况下，在编写代码之前编写测试将帮助您发现许多问题。

使用 SOLID 原则

有很多原则和设计方法可以在项目设计中使用。保持设计简单是最好的，然而，有一些原则在几乎所有项目中都是有用的。例如，SOLID 由 5 个原则组成，其中的全部或部分原则可能对设计有用。

SOLID 代表以下原则：

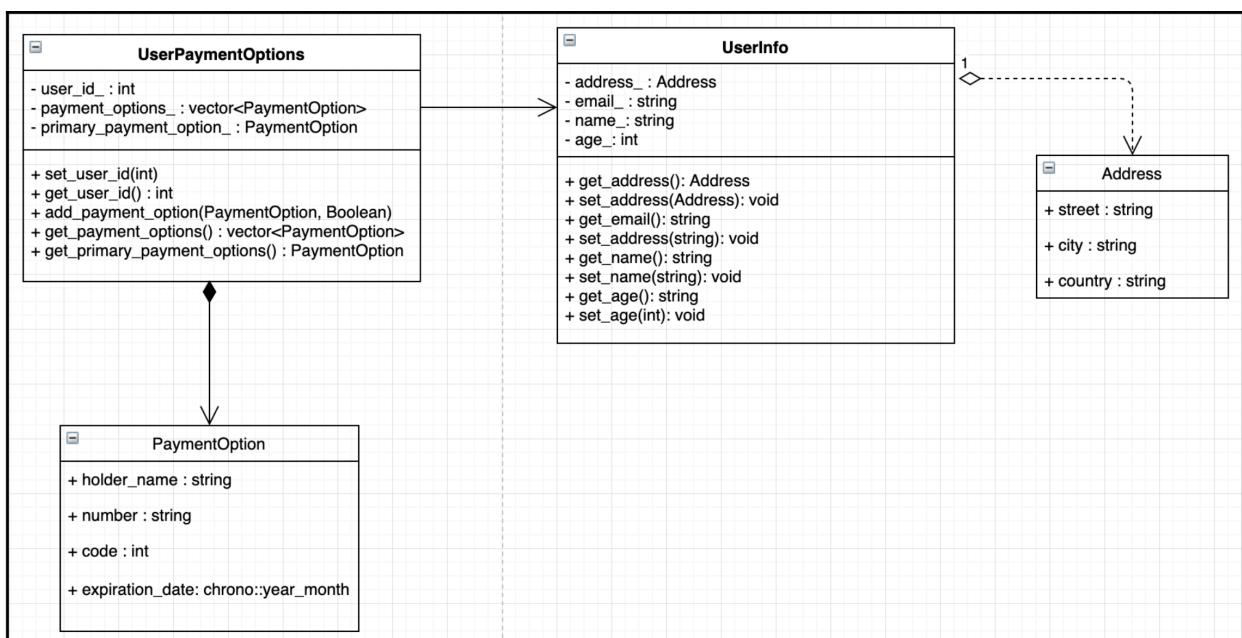
- 单一责任原则
- 开闭原则
- 里氏替换原则
- 接口分离原则
- 依赖倒置原则

让我们结合例子来讨论下每个原则。

单一责任原则

单一责任原则强调简单，即一个对象一项任务。尽量减少对象功能关系的复杂性。让每个对象都有一个职责，不过把一个复杂的对象分解成更小更简单的组件并不总是那么容易。单一责任是一个上下文绑定的概念，不是关于在一个类中只有一个方法，而是关于让类或模块负责一件事。例如，我们前面设计的 User 类有一个职责：存储用户信息。然而，我们在 User 类中添加了支付选项，并强制它具有添加和删除支付选项的方法。我们还引入了一个主要的支付选项，它在用户方法中包含了额外的逻辑。

第一个建议将 User 类分解为两个单独的类。每个类都有一个单独的职责。下面的类图描述了这个想法：

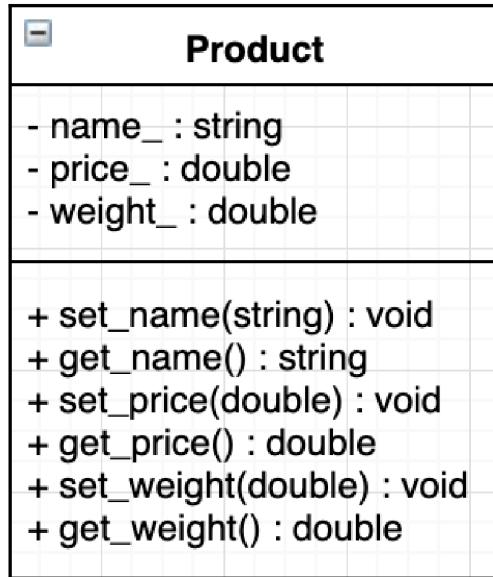


其中一个将只存储用户的基本信息，下一个将存储用户的支付选项。我们将它们命名为 UserInfo 和 UserPaymentOptions。有些人可能会喜欢这个新设计，但我们会坚持旧的设计。这是为什么呢？虽然 User 类包含用户信息和支付选项，但后者也表示一条信息。我们设置和获取支付选项的方式与设置和获取用户电子邮件的方式相同。因此，我们保持 User 类不变，因为它满足了单一职责原则。当我们在 User 类中添加支付功能时，这将打破原始设计。在该场景中，User 类将存储用户信息并进行支付事务。就单一责任原则而言，这是不可接受的，因此，我们不会这么做。

单一责任原则也与职能有关。add_payment_option() 方法有两个职责。如果函数的第二个（默认）参数为 true，将添加一个新的主要支付选项。否则，会将新的支付选项添加到非主要选项列表中。最好有一个单独的方法来添加一个主要的支付选项。这样，每个方法都有一个单独的职责。

开闭原则

开闭原则是指类应该对扩展开放，但对修改关闭。这意味着当需要新的功能时，最好扩展基本功能，而不是修改。以设计的电子商务应用程序的 Product 类为例，下面是 Product 类的一个简单图：



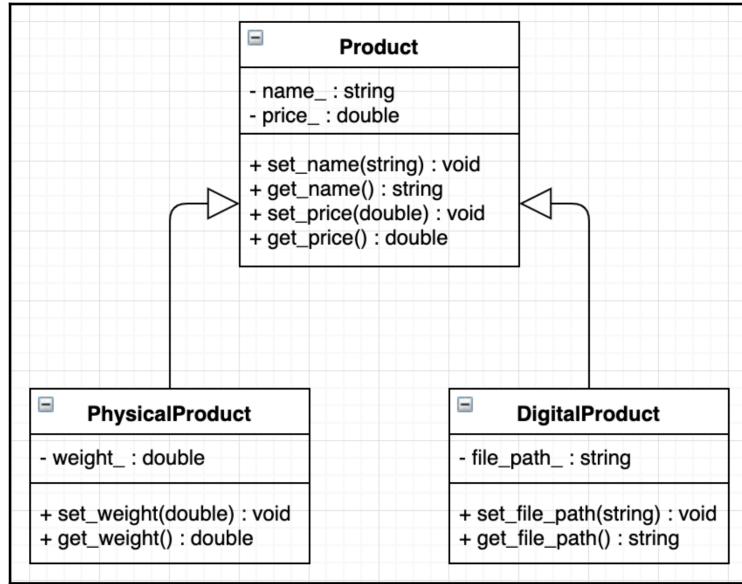
每个 Product 对象有三个属性: 名称、价格和重量。想象一下, 在设计了产品类和整个电子商务平台之后, 来自客户的新需求。他们现在想买数字产品, 如: 电子书、电影和录音。除了产品的重量之外, 一切都很好。现在可能有两种产品——有形的和数字的——我们应该重新思考产品使用的逻辑。我们可以在产品中加入一个新功能, 如下代码所示:

```

1 class Product
2 {
3     public:
4         // code omitted for brevity
5         bool is_digital() const {
6             return weight_ == 0.0;
7         }
8         // code omitted for brevity
9     };

```

显然, 我们修改了类, 违背了开闭原则。这个原则是说这个类应该关闭以进行修改。它应该开放扩展。我们可以通过重新设计 Product 类并使其成为所有产品的抽象基类来实现这一点。接下来, 我们创建另外两个继承 Product 基类的类:PhysicalProduct 和 DigitalProduct。下面的类图描述了新的设计:



正如您在前面的图表中看到的，我们从 Product 类中删除了 weight_ 属性。现在我们又多了两个类，PhysicalProduct 有一个 weight_property，而 DigitalProduct 没有。相反，其有一个 file_path_ 属性。这种方法满足开闭原则，因为现在所有的类都可以扩展。我们使用继承来扩展类，下面的原则与之密切相关。

里氏替换原则

里氏替换原则是以正确的方式继承。简单地说，如果有函数接受某种类型的参数，那么这个函数应该接受派生类型的参数。



里氏替代原理是以图灵奖得主、计算机科学博士 Barbara Liskov 的名字命名的。

当你理解了继承和里氏替换原理，你就很难忘记它。让我们继续开发 Product 类并添加一个新方法，该方法根据货币类型返回产品的价格。我们可以将价格存储在相同的货币单位中，并提供一个函数将价格转换为指定的货币。下面是该方法的简单实现：

```

1 enum class Currency { USD, EUR, GBP }; // the list goes further
2
3 class Product
4 {
5     public:
6         // code omitted for brevity
7         double convert_price(Currency c) {
8             // convert to proper value
9         }
10
11        // code omitted for brevity
12    };

```

过了一段时间，该公司决定纳入所有数字产品的终身折扣。现在，所有的数码产品都有 12%

的优惠。我们在 DigitalProduct 类中添加了一个单独的函数，该函数通过应用折扣返回转换后的价格。下面是它在 DigitalProduct 中的样子：

```
1 class DigitalProduct : public Product
2 {
3     public:
4         // code omitted for brevity
5         double convert_price_with_discount(Currency c) {
6             // convert by applying a 12% discount
7         }
8 };
```

设计上的问题显而易见。在 DigitalProduct 实例上调用 convert_price() 不起作用。更糟糕的是，客户机代码不能调用它。它应该调用 convert_price_with_discount()，因为所有数字产品都必须以 12% 的折扣销售。这种设计与里氏替换原理相矛盾。

我们应该记住多态的美妙之处，而不是破坏类的层次结构。更好的版本应该是这样的：

```
1 class Product
2 {
3     public:
4         // code omitted for brevity
5         virtual double convert_price(Currency c) {
6             // default implementation
7         }
8
9         // code omitted for brevity
10 };
11
12 class DigitalProduct : public Product
13 {
14     public:
15         // code omitted for brevity
16         double convert_price(Currency c) override {
17             // implementation applying a 12% discount
18         }
19
20         // code omitted for brevity
21 };
```

可以看到，我们不再需要 convert_price_with_discount() 函数。里氏替换原则也成立。然而，我们应该再检查设计中的缺陷。让我们通过在基类中合并私有虚方法来计算折扣来改进它。以下更新版本的 Product 类包含一个名为 calculate_discount() 的私有虚成员函数：

```
1 class Product
2 {
3     public:
4         // code omitted for brevity
5         virtual double convert_price(Currency c) {
6             auto final_price = apply_discount();
7             // convert the final_price based on the currency
8 };
```

```
8     }
9
10    private:
11        virtual double apply_discount() {
12            return getPrice(); // no discount by default
13        }
14
15        // code omitted for brevity
16    };
```

convert_price() 函数调用私有 apply_discount() 函数，该函数按原样返回价格。在派生类中重写 apply_discount() 函数，如下面的 DigitalProduct 实现所示：

```
1 class DigitalProduct : public Product
2 {
3     public:
4         // code omitted for brevity
5     private:
6         double apply_discount() override {
7             return getPrice() * 0.12;
8         }
9         // code omitted for brevity
10    };
```

我们不能在类之外调用私有函数，但可以在派生类中重写它。前面的代码展示了重写私有虚函数的优点。我们修改实现，保持接口不变。如果派生类不需要为折扣计算提供自定义功能，则不需要重写。另一方面，DigitalProduct 在转换之前需要在价格上申请 12% 的折扣。不需要修改基类的公共接口。



应该考虑重新思考 Product 类的设计。因为总是返回最新的有效价格，所以在 getPrice() 中直接调用 apply_discount() 似乎更好。

设计过程是创造性的。由于出现了意想不到的新需求，重写所有代码并不少见。我们使用原则和方法来最小化实现新功能后可能出现的破坏性更改。SOLID 的下一个原则是使您的设计更加灵活的最佳实践。

接口分离原则

接口分离原则建议将复杂的接口划分为更简单的接口，这种分离允许类避免实现它们不使用的接口。

在电子商务应用程序中，我们应该实现产品交付、替换和过期功能。产品的运输将产品项目移动到其买家（现阶段我们不关心装运细节）。更换产品考虑更换损坏或丢失的产品后，运输给买家。最后，让产品过期意味着扔掉那些在过期日期前没有销售的产品。

我们可以自由地实现前面介绍的 Product 类中的所有功能。然而，我们会偶然发现一些类型的产品，例如：不能运输（例如，出售房子很少涉及运输给买家）。可能会有不可替代的产品。例如，

一幅原画是不可能替代的，即使它丢失或损坏。最后，数字产品永远不会过期（）大多数情况下是这样）。

我们不应该强迫客户端代码实现它不需要的行为，指的是实现类的行为。下面的例子是一个不好的做法，与接口隔离原则相矛盾：

```
1 class IShippableReplaceableExpirable
2 {
3     public:
4         virtual void ship() = 0;
5         virtual void replace() = 0;
6         virtual void expire() = 0;
7 }
```

现在，Product 类实现了前面所示的接口，必须为所有的方法提供一个实现。接口隔离原则提出以下模型：

```
1 class IShippable
2 {
3     public:
4         virtual void ship() = 0;
5     };
6 class IReplaceable
7 {
8     public:
9         virtual void replace() = 0;
10    };
11 class IExpirable
12 {
13     public:
14         virtual void expire() = 0;
15 }
```

现在，Product 类跳过实现任何接口。它的派生类从特定类型派生（实现）。下面的示例声明了几种产品类，每种产品类都支持前面介绍的有限数量的行为。请注意，为了代码简洁，我们省略了类的主体：

```
1 class PhysicalProduct : public Product {};
2 // The book does not expire
3 class Book : public PhysicalProduct, public IShippable, public IReplaceable
4 {
5 };
6
7 // A house is not shipped, not replaced, but it can expire
8 // if the landlord decided to put it on sell till a specified date
9 class House : public PhysicalProduct, public IExpirable
10 {
11 };
12
13 class DigitalProduct : public Product {};
```

```
15 // An audio book is not shippable and it cannot expire.  
16 // But we implement IReplaceable in case we send a wrong file to the user.  
17 class AudioBook : public DigitalProduct, public IReplaceable  
18 {  
19 };
```

如果您想打包一个下载的文件，请考虑为 AudioBook 实现 IShippable。

依赖倒置原则

最后，依赖倒置表明对象不应该强耦合，允许轻松地切换到另一个依赖项。例如，当用户购买一个产品时，我们发送一个关于购买的收据。从技术上讲，发送收据有几种方式，即通过邮件打印发送，通过电子邮件发送，或在平台的用户账户页面上显示收据。对于后者，我们通过电子邮件或应用程序向用户发送通知，告诉他们收据已经可以查看了。看看下面打印收据的接口：

```
1 class IReceiptSender  
2 {  
3     public:  
4         virtual void send_receipt() = 0;  
5 };
```

让我们假设我们已经实现了 Product 类中的 purchase() 方法。完成后，我们发送收据。下面的代码发送了收据：

```
1 class Product  
2 {  
3     public:  
4         // code omitted for brevity  
5         void purchase(IReceiptSender* receipt_sender) {  
6             // purchase logic omitted  
7             // we send the receipt passing purchase information  
8             receipt_sender->send(/* purchase-information */);  
9         }  
10    };
```

我们可以根据需要添加尽可能多的收据打印选项来扩展应用程序。下面的类实现了 IReceiptSender 接口：

```
1 class MailReceiptSender : public IReceiptSender  
2 {  
3     public:  
4         // code omitted for brevity  
5         void send_receipt() override { /* ... */ }  
6 };
```

另外两个类——EmailReceiptSender 和 InAppReceiptSender——都实现了 IReceiptSender。因此，要使用特定的收据，我们只需通过 purchase() 方法将依赖注入到 Product 类当中，如下所示：

```
1 IReceiptSender* rs = new EmailReceiptSender();  
2 // consider the get_purchasable_product() is implemented somewhere in the code
```

```
3 auto product = get_purchasable_product();
4 product.purchase(rs);
```

我们可以更进一步，在 User 类中实现一个方法，返回具体用户想要的收据发送选项。这将使类更加解耦。

前面讨论的所有 SOLID 原则是一个自然的方式组合类。并不是必须要坚持这些原则，但它将会改进你的设计。

使用域驱动的设计

域是程序的主区域。我们正在讨论和设计一个以电子商务为主要概念，领域为电子商务平台。我们建议您在项目中考虑领域驱动设计。然而，这种方法也并不是程序设计的灵丹妙药。

为了方便设计项目，有以下三层架构：

- 业务描述
- 业务逻辑
- 业务数据

三层架构适用于客户端-服务器软件，就像我们在本章设计的那样。表述层向用户提供与产品、购买和运输相关的信息。它通过将结果输出到客户端来与其他层进行通信。它是客户端可以直接访问的层，例如：web 浏览器。

业务逻辑关心应用程序的功能。例如，用户浏览表示层提供的产品，并决定购买其中之一。处理请求是业务层的任务。在领域驱动的设计中，我们倾向于结合领域的实体，其属性可用来解决应用程序的复杂性。我们将用户作为 User 类的实例处理，将产品作为 Product 类的实例处理，以此类推。业务逻辑将购买产品的用户解释为创建 Order 对象的用户对象，而 Order 对象又与 Product 对象相关。然后，将 Order 对象绑定到与购买该产品相关的事务对象。购买的相应结果通过表述层表示。

最后，数据层处理数据的存储和检索。从用户身份验证到产品购买，每个步骤都是从系统数据库检索或记录在系统数据库。

将应用程序划分为几个层可以处理复杂性，编写具有单一职责的对象要好得多。领域驱动设计将实体与没有概念标识的对象区分开来，后者称为对象。例如，用户不区分每个事务，他们只关心事务所代表的信息。另一方面，User 对象具有 User 类（实体）形式的标识。

使用其他对象（或不使用）的对象上允许的操作，称为命名服务。服务是一种不绑定到特定对象的操作，例如：通过 set_name() 方法设置用户名是不应该视为服务的操作。另一方面，用户购买产品是一种由服务封装的操作。

最后，领域驱动的设计集成了存储库和工厂模式。存储库模式负责检索和存储域对象的方法，工厂模式创建域对象。使用这些模式允许在需要的时候交换替代的实现。现在，让我们来看看电子商务平台中设计模式的威力。

利用设计模式

设计模式是软件设计中的体系结构解决方案。需要注意的是，设计模式既不是方法，也不是算法。它们是一种体系结构构造，提供了一种组织类及其关系的方法，以便在代码可维护性方面获得更好的结果。即使以前没有使用过一个设计模式，在软件设计中许多问题往往会重复出现。例如，

为现有库创建更好的接口是一种称为 Facade 模式。设计模式有名称，以便开发者在对话或文档中使用。您应该很自然地与使用 Facade 模式、工厂模式等的其他开发者沟通。

我们前面已经提到过，域驱动的设计合并了存储库和工厂模式。现在让我们来看看它们是什么，以及它们在我们的设计工作中有什么用处。

库模式

正如 Martin Fowler 所描述的那样，存储库模式“使用一个用于访问域对象的类集成接口在域层和数据映射层间充当中介”。

该模式为数据操作提供了简单的方法，不需要直接使用数据库驱动程序。添加、更新、删除或选择适合应用程序域。

其中一种方法是创建提供必要函数的通用存储库类。简单接口如下：

```
1 class Entity; // new base class
2
3 template <typename T, typename = std::enable_if_t<std::is_base_of_v<Entity,
4 T>>>
5 class Repository
6 {
7     public:
8     T get_by_id(int);
9     void insert(const T&);
10    void update(const T&);
11    void remove(const T&);
12    std::vector<T> get_all(std::function<bool(T)> condition);
13};
```

我们引入了一个名为 Entity 的新类。Repository 类处理实体，并确保每个实体都符合实体的相同接口，将 std::enable_if 和 std::is_base_of_v 应用到模板参数中。



std::is_base_of_v 是 std::is_base_of<>::value 的简短表示。

同样，std::enable_if_t 替换了 std::enable_if<>::type。

Entity 类就像下面这样简单：

```
1 class Entity
2 {
3     public:
4     int get_id() const;
5     void set_id(int);
6     private:
7     int id_;
8};
```

每个业务对象都是一个实体。因此，前面讨论的类应该更新为从 Entity 继承。例如，User 类采用以下形式：

```

1 class User : public Entity
2 {
3     // code omitted for brevity
4 };

```

因此，我们可以用以下方式来使用存储库：

```

1 Repository<User> user_repo;
2 User fetched_user = user_repo.get_by_id(111);

```

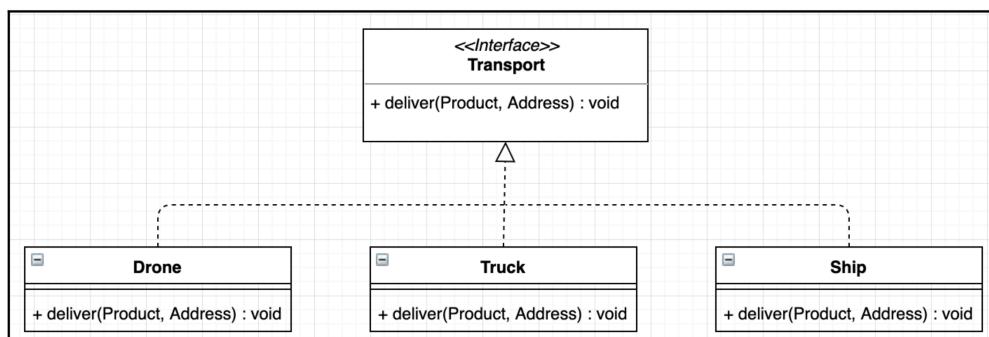
前面的存储库模式只是对这个主题的简单介绍，但可以使它更加强大。尽管使用 Facade 模式的重点不是访问数据库，但仍然可以用这个示例来解释它。Facade 模式包装了一个或多个复杂的类，为客户端提供了预定义接口来处理底层功能。

工厂模式

当开发者谈论工厂模式时，他们可能会混淆工厂方法和抽象工厂模式，这两种创建模式都提供了各种创建机制。让我们讨论一下工厂的方法。它提供了一个在基类中创建对象的接口，并允许派生类修改将要创建的对象。

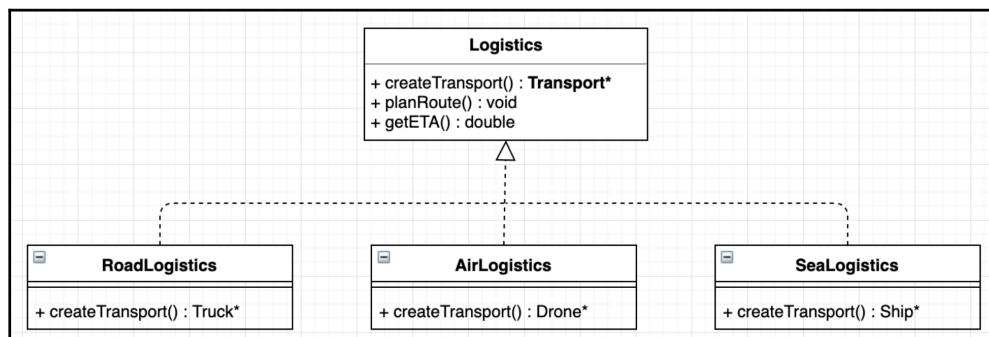
现在是处理物流的时候了，工厂方法会帮助我们。当开发一个提供产品发货的电子商务平台时，应该考虑到并非所有用户都居住在仓库所在的同一区域。所以，将产品从仓库运送给买家，你应该选择合适的运输方式，自行车、无人机、卡车等。设计一个灵活的物流管理系统是研究的重点。

不同的运输方式需要不同的实现。然而，它们都符合一个接口。以下是传输接口及其派生的特定传输实现类图：



上图中的每个具体类都提供了交付的具体实现。

假设我们设计了以下物流基类，负责与物流相关的行动，包括选择合适的运输方式，如下图所示：



前面应用的工厂方法允许增加新的运输类型和新的物流方法。请注意 `createTransport()` 方法，它返回一个指向 `Transport` 的指针。派生类覆盖该方法，每个派生类都返回 `Transport` 的子类，因此提供了特定的传输模式，否则，我们不能在重写基类方法时返回不同的类型。

物流中的 `createTransport()` 如下所示：

```
1 class Logistics
2 {
3     public:
4     Transport* getLogistics() = 0;
5     // other functions are omitted for brevity
6 }
```

`Transport` 类表示无人机、卡车和船舶的基类。这意味着我们可以为它们创建一个实例，并使用一个 `Transport` 指针来引用它们，如下所示：

```
1 Transport* ship_transport = new Ship();
```

这是工厂模式的基础，因为 `RoadLogistics`，例如：复写 `getLogistics()`：

```
1 class RoadLogistics : public Logistics
2 {
3     public:
4     Truck* getLogistics() override {
5         return new Truck();
6     }
7 }
```

注意函数的返回类型，它是 `Truck` 而不是 `Transport`。它能起作用是因为卡车继承了 `Transport`。另外，还可以看到对象创建是如何与对象本身解耦的。创建新对象通过工厂完成，这与前面讨论的 SOLID 原则保持一致。

乍一看，利用设计模式将额外的复杂性合并到设计中，这可能令人困惑。然而，在实践设计模式时，应该具有更好的设计，因为允许项目的灵活性和可扩展性。

总结

软件开发需要细致的规划和设计。本章中，我们了解到项目开发包括以下步骤：

- 需求收集和分析：了解项目的领域，讨论并确定应该实现的特性。
- 创建规则说明：记录需求和项目功能。
- 设计和测试计划：指从较大的实体开始设计项目，将每个实体分解为与项目中的其他类相关的单独的类。这一步还包括计划如何测试项目。
- 编码：此步骤涉及编写代码，以实现前面步骤中指定的项目。
- 测试和稳定性：根据预先计划的用例和场景检查项目，以发现问题并修复它们。
- 发布和维护：项目发布和维护是最后一步。

项目设计对开发者来说是一项复杂的任务。他们应该提前考虑，因为部分特性是在开发过程中引入的。

为了使设计更灵活和健壮，我们讨论了更好架构的原则和模式。我们已经了解了设计一个复杂软件项目的过程。

避免糟糕设计决策的最佳方法之一是，遵循已经设计好的模式和实践。应该考虑在未来的项目中，使用可靠的原则和经过验证的设计模式。

下一章中，我们将设计一款策略游戏。将熟悉的设计模式进行应用，并看到它们如何出现在游戏开发中。

问题

1. TDD 的好处是什么？
2. UML 中交互图的目的是什么？
3. 合成和聚合有什么区别？
4. 如何描述里氏替换原则？
5. 让我们假设给您一个类 Animal 和一个类 Monkey。后者描述的是一种特殊的在树上跳的动物。从动物类继承猴子类是否违背开闭原则？
6. 在本章的 Product 类及其子类上应用工厂模式。

扩展阅读

更多资料请参阅：

- Object-Oriented Analysis and Design with Applications by Grady Booch, <https://www.amazon.com/Object-Oriented-Analysis-Design-Applications-3rd/dp/020189551X/>
- Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma et al, <https://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/>
- Code Complete: A Practical Handbook of Software Construction by Steve McConnell, <https://www.amazon.com/Code-Complete-Practical-Handbook-Construction/dp/0735619670/>
- Domain-Driven Design: Tackling Complexity in the Heart of Software by Eric Evans, <https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/>

第 11 章：使用设计模式设计策略游戏

游戏开发是软件工程中最有趣的话题之一。C++ 因其高效而广泛应用于游戏开发中，但由于 C++ 没有 GUI 组件，所以目前只能在后端使用。本章中，我们将学习如何在后端设计策略游戏。我们将把前面章节中学到的所有东西都结合起来，包括设计模式和多线程。

我们要设计的游戏是一款名为《读者与扰乱者》的策略游戏。玩家创造了能够建造图书馆和其他建筑的单位，也就是所谓的“读者”，以及守卫这些建筑不受敌人攻击的士兵。

本章中，我们将了解以下内容：

- 游戏制作入门
- 深入研究游戏设计过程
- 使用设计模式
- 设计游戏循环

编译器要求

g++ 编译器需要添加编译选项 `-std=c++2a` 来编译本章的代码。可以从这里获取本章的源码文件：<https://github.com/PacktPublishing/Expert-CPP>

游戏制作入门

我们将设计一款策略游戏的后端，玩家可以创建单位（工人，士兵），建造建筑，并与敌人战斗。设计一款游戏，无论是战略游戏还是第一人称射击游戏，有一些基本元素是相同的，比如：游戏物理元素，能够让玩家觉得游戏更真实，更有沉浸感。

游戏设计中有些组件会在所有游戏中重复出现，如：碰撞检测机制、音频系统、图像渲染等。设计游戏时，我们既可以区分引擎和游戏，也可以开发一个紧密结合的应用程序，将引擎和游戏作为单独的成果。单独设计的游戏引擎，需要允许为进一步开发进行扩展，甚至用于其他游戏。毕竟，游戏都有相同的机制和流程。他们的不同之处在于情节主线。

设计游戏引擎时，应该仔细规划使用该引擎的游戏类型。独立于游戏类型之外，3D 射击游戏和策略游戏也有区别。策略游戏中，玩家在一个大的游戏场上，有策略地部署单位。游戏世界是以自上而下的视角呈现的。

了解《读者与扰乱者》游戏

这款游戏很简单：玩家拥有有限的资源。这些资源可以用来为游戏角色创造建筑。我们将角色单位命名为读者和士兵。读者是建造图书馆和其他建筑的人物。每个构建的库最多可以容纳 10 个读者。如果玩家将 10 个读者移进图书馆，在一段特定时间后，图书馆将产生 1 个教授。教授是一个强大的单位，可以摧毁三个敌人。教授可以为士兵创造更好的武器。

游戏一开始有一座已经建好的房子，两个士兵和三个读者。一所房子每 5 分钟生产一个新的读者。读者可以建造新的房子，这样就能产生更多的读者。他们也可以建造兵营来生产士兵。

玩家的目标是建立 5 个库，每个库至少有一个教授。玩家必须在游戏中保护他/她的建筑和读者不受敌人攻击。敌人称为干扰者，因为他们的目标是干扰读者在图书馆里学习。

战略游戏组件

我们的策略游戏将包含基本组件——读者和士兵（称为单位）、建筑和地图。游戏地图包含游戏中每个对象的坐标。我们将讨论一个更轻版本的游戏地图。现在，让我们利用设计技能分解游戏本身。

游戏由以下角色单位组成：

- 读者
- 士兵
- 教授

它还包括以下建筑物：

- 图书馆
- 房子
- 兵营

现在，让我们讨论游戏中每个组件的属性。游戏角色有以下属性：

- 生命值（整数，每次敌人攻击后会减少）
- 攻击力（整数，可以对敌人单位造成的伤害）
- 角色（读者，士兵，教授）

生命值应该有一个基于单位类型的初始值。例如，读者的初始生命值是 10，而士兵的初始生命值是 12。当在游戏中，所有的单位都可能受到敌人单位的攻击。每次攻击描述为生命点的减少。我们将减少的生命点数是基于攻击者的攻击力。例如，士兵的攻击力设置为 3，这意味着士兵的每一次攻击都会让生命值减少 3 点。当受攻击的单位生命值小于等于 0 时，单位将死亡。

建筑也是如此。建筑物有一个建造周期，一个建筑物也有生命值，任何敌人对建筑造成的伤害都会降低这些生命值。以下是建筑物业的完整列表：

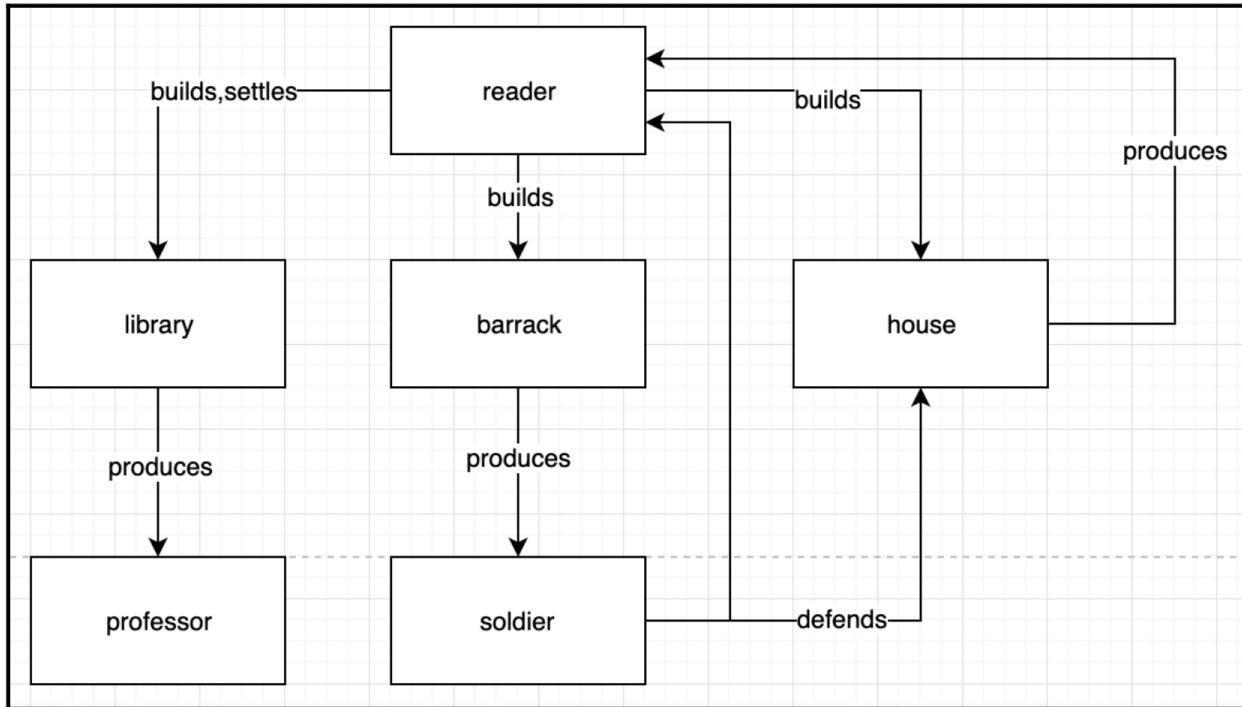
- 生命值
- 建筑类型
- 建造时间
- 生产时间

生产单位持续时间是指，生产一个新角色单位所需要的时间。例如，一个兵营每 3 分钟产生一个士兵，一个房子每 5 分钟产生一个读者，一个图书馆在 10 个读者进入图书馆时立即产生一个教授。

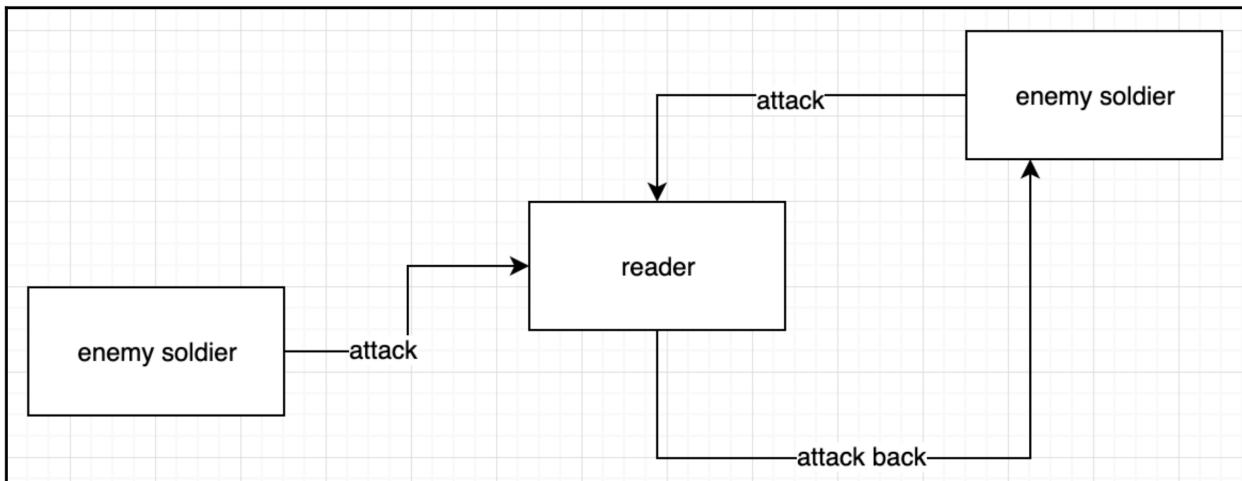
现在定义了游戏组件，让我们来讨论它们之间的交互作用。

组件之间的相互作用

游戏设计的下一个重要内容是角色之间的互动。我们已经提到，读者可以建造建筑物。游戏中，这一过程应该得到重视，因为每种类型的建筑都有其建造时间。因此，如果读者正忙于构建过程，我们应该计算时间，以确保构建将在指定的时间后完成。然而，为了让游戏变得更好，我们应该考虑到不止一个读者可以参与到构建过程中。这应该会使建造楼房的速度加快。例如，如果一个兵营是一个读者需要 5 分钟建造的，那么两个读者就应该在 2.5 分钟完成，以此类推。这是游戏中复杂互动的一个例子，可以用下图来描述：

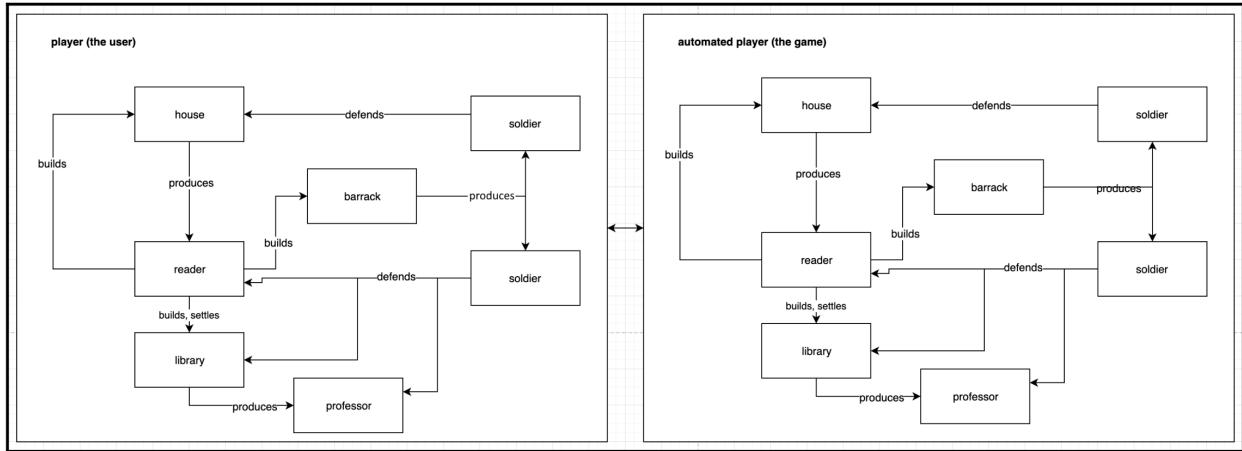


接下来是处理攻击。当一个单位遭受攻击时，我们应该减少单位的生命值。单位可以进行反攻（保护自己）。当有多个攻击者或防御者时，我们应该对每次攻击都进行处理。还定义单位每次命中的持续时间，一个单位不应该快速攻击另一个单位。为了让事情更自然，我们会在每次点击之间引入 1 秒或 2 秒的暂停。下图描述了一个简单的攻击交互：



更多的互动发生在游戏中。游戏中有两组，其中一组由玩家控制。另一个是由电脑控制。这意味着作为游戏设计师必须定义敌人。游戏将自动创建读者，并分配给他们创建图书馆、兵营和房屋的任务。每个士兵都应该承担保卫建筑物和读者（人民）的责任。有时，士兵们应该聚在一起，执行攻击任务。

我们将设计一个平台，让玩家创建一个帝国，而游戏也应该创造敌人来完善游戏。玩家将经常面临敌人的攻击，敌人将通过建造更多建筑和生产更多单位而进化。总的来说，我们可以用下图来描述交互：



我们将在设计游戏时参考上述图表。

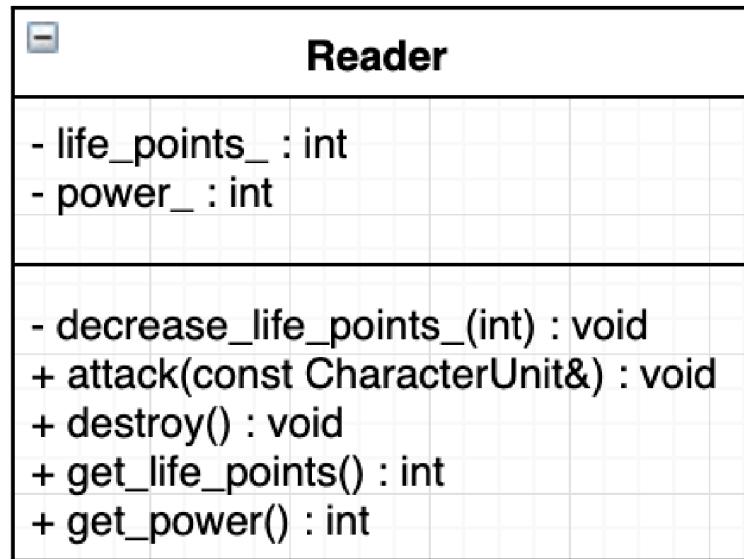
设计游戏

虽然游戏并不是一款典型的软件，但它的设计却与常规应用设计并无太大区别。我们将从主要实体开始，并将它们进一步分解为类的关系。

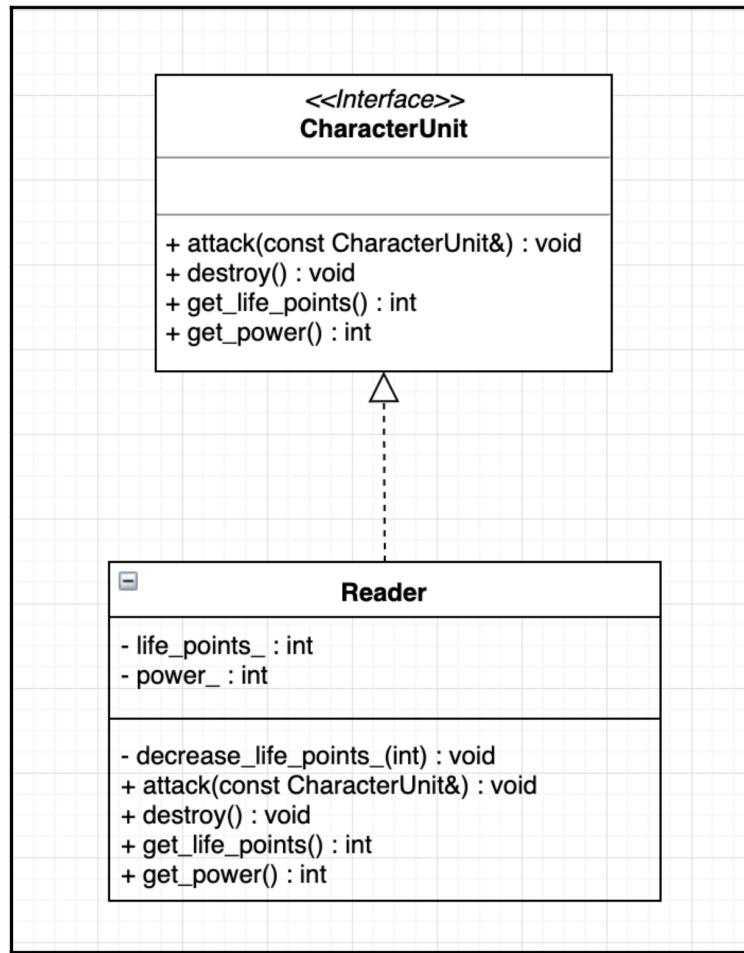
在前一节中，我们讨论了游戏组件及其交互作用。我们根据项目开发生命周期进行了需求分析和收集。现在，我们开始设计游戏。

设计角色

下面的类图代表了一个读者：



当我们浏览其他角色单位时，我们将为每个角色单位想出一个基类。每个特定的单元将从这个基类继承，并添加特定的属性（如果有的话）。以下是角色单位的完整类图：



请注意基类——它是一个接口，而不是一个常规类。它定义了在派生类中实现的纯虚函数。以下是 `CharacterUnit` 接口在代码中的声明：

```

1 class CharacterUnit
2 {
3     public:
4         virtual void attack( const CharacterUnit&) = 0;
5         virtual void destroy() = 0;
6         virtual int get_power() const = 0;
7         virtual int get_life_points() const = 0;
8 }

```

`attack()` 会降低角色的生命值，`destroy()` 则会摧毁角色。摧毁不仅意味着将角色从场景中移除，还意味着停止该单位正在进行的所有互动（如建造建筑、自卫等等）。

派生类提供了 `CharacterUnit` 接口类的纯虚函数的实现。让我们来看看 `Reader` 字符单元的代码：

```

1 class Reader : public CharacterUnit
2 {
3     public:
4         Reader();
5         Reader( const Reader&) = delete;
6         Reader& operator=( const Reader&) = delete;

```

```

7
8 public:
9     void attack(const CharacterUnit& attacker) override {
10        decrease_life_points_by_(attacker.get_power());
11    }
12
13    void destroy() override {
14        // we will leave this empty for now
15    }
16
17    int get_life_points() const override {
18        return life_points_;
19    }
20
21    int get_power() const override {
22        return power_;
23    }
24
25 private:
26    void decrease_life_points_(int num) {
27        life_points_ -= num;
28        if (life_points_ <= 0) {
29            destroy();
30        }
31    }
32
33 private:
34     int life_points_;
35     int power_;
36 };

```

现在，我们可以通过以下任何一种方式声明 Reader:

```

1 Reader reader;
2 Reader* pr = new Reader();
3 CharacterUnit* cu = new Reader();

```

主要通过基接口类来引用角色单位。



TIP 请注意复制构造函数和赋值操作符。我们故意把它们标记为删除，因为我们不想通过复制其他单位来创建单位。我们将为该行为使用原型模式。

我们应该为不同类型的单位做同样的事情的场景中，拥有 CharacterUnit 接口非常重要，例如：我们必须计算两名士兵、一名读者和一名教授对一座建筑造成的全部损害。我们不必保留三个不同的引用来引用三种不同类型的单元，而是将它们都称为 CharacterUnits。方法如下：

```

1 int calculate_damage(const std::vector<CharacterUnit*>& units)
2 {

```

```

3   return std::reduce(units.begin(), units.end(), 0,
4   [](CharacterUnit& u1, CharacterUnit& u2) {
5     return u1.get_power() + u2.get_power();
6   }
7 );
8 }
```

calculate_damage() 函数对单元类型进行抽象，它不关心读者或士兵。它只调用 CharacterUnit 接口的 get_power() 方法，该方法保证特定对象的实现。

我们将更新角色单位类，让我们继续为建筑设计类。

设计建筑

建筑类型与角色单位的接口相似。例如，可以定义建筑的类：

```

1 class House
2 {
3 public:
4   House();
5   // copying will be covered by a Prototype
6   House(const House&) = delete;
7   House& operator=(const House&) = delete;
8
9 public:
10  void attack(const CharacterUnit&);
11  void destroy();
12  void build(const CharacterUnit&);
13  // ...
14
15 private:
16  int life_points_;
17  int capacity_;
18  std::chrono::duration<int> construction_duration_;
19 };
```

我们使用 std::chrono::duration 来统计房屋建造的时长，它在 <chrono> 头文件中定义为一个刻度数和一个刻度周期，其中刻度周期是从一个刻度到下一个刻度的秒数。

House 类需要更多的细节，需要为所有的建筑提供一个基接口（甚至是一个抽象类）。本章中描述的建筑都有相同的行为。建筑接口如下：

```

1 class IBuilding
2 {
3 public:
4   virtual void attack(const CharacterUnit&) = 0;
5   virtual void destroy() = 0;
6   virtual void build(CharacterUnit*) = 0;
7   virtual int get_life_points() const = 0;
8 };
```

注意建筑物前面的 I 前缀。许多开发人员建议为接口类使用前缀或后缀，以提高可读性。例如，Building 可能被命名为 IBuilding 或 BuildingInterface。我们将对前面描述的 CharacterUnit 使用相同的命名技术。

House、Barrack 和 Library 类实现了 IBuilding 接口，并且必须提供纯虚方法的实现。例如，Barrack 类如下所示：

```
1 class Barrack : public IBuilding
2 {
3     public:
4         void attack(const ICharacterUnit& attacker) override {
5             decrease_life_points_(attacker.get_power());
6         }
7
8         void destroy() override {
9             // we will leave this empty for now
10        }
11
12        void build(ICharacterUnit* builder) override {
13            // construction of the building
14        }
15
16        int get_life_points() const override {
17            return life_points_;
18        }
19
20     private:
21         int life_points_;
22         int capacity_;
23         std::chrono::duration<int> construction_duration_;
24 };
```

让我们更详细地讨论构建时长的实现，std::chrono::duration 可以告诉我们构建花费的时间。另外，类的最终设计可能在本章的行文过程中发生变化。现在，让我们看看如何让游戏的组件彼此互动。

设计游戏控制器

为角色单位和建筑设计类型只是设计游戏的第一步。游戏中最重要的组件间的互动。我们需要仔细分析和设计案例，比如：两个或两个以上的读者建造一座建筑。我们已经介绍了建筑的建造时间，但我们没有考虑到一个建筑可能由多个读者建造（可以建造建筑的角色单位）。

两个读者建一座大楼的速度应该比一个读者建的快两倍。如果另一个读者加入了，我们应该重新计算持续时间。然而，我们应该限制能建造同一建筑的读者数量。

如果读者被敌人攻击，这会打断读者建造，他们会进行自卫。当读者停止在建筑上工作时，我们应该重新计算施工时间。当角色受到攻击时，它应该反击来保护自己。每次命中都会减少角色的生命值。一个角色可能同时受到多个敌人角色的攻击。这将更快地降低他们的生命值。

建筑有一个计时器，它会周期性地产生角色。设计最重要的是游戏动态，也就是循环。在每个

特定的时间框架，游戏中都会发生一些事情。这可能是敌人士兵靠近，角色单位建造东西，或其他任何东西。一个操作的执行与另一个不相关的操作的完成没有严格的联系。这意味着建筑的建造与角色的创造同时发生。与大多数应用程序不同，游戏应该保持移动，即使用户不提供任何输入。如果玩家未能执行某个行动，游戏也不会停止。角色单位可能会等待一个命令，但是建筑会继续工作——产生新的角色。同样地，敌人玩家（自动玩家）也会为胜利不停的奋斗。

并发行为

游戏中的许多行动是同时发生的。就像我们之前所讨论的，建筑的建造不应该因为未参与建造的单位或受到敌人的攻击而停止。这意味着我们应该为游戏中的许多对象设计并发行为。

C++ 中实现并发的最佳方法是使用线程。我们可以重新设计单位和建筑，以便它们在基类中有可重写的方法，该方法将在单独的线程中执行。让我们重新设计 IBuilding，使它成为一个抽象类，它有一个额外的 run() 虚函数：

```
1 class Building
2 {
3     public:
4         virtual void attack(const ICharacterUnit&) = 0;
5         virtual void destroy() = 0;
6         virtual void build(ICharacterUnit*) = 0;
7         virtual int get_life_points() const = 0;
8
9     public:
10        void run() {
11            std::jthread{Building::background_action_, this};
12        }
13
14    private:
15        virtual void background_action_() {
16            // no or default implementation in the base class
17        }
18    };
}
```

注意 background_action_() 函数，它是私有的，但却是虚的，我们可以在派生类中重写它。run() 函数不是虚函数，它在线程中运行私有实现。派生类可能会提供 background_action_() 的实现。当一个单元来建造建筑时，就会调用 build() 虚函数。build() 函数将计算构造时间的工作委托给 run() 函数。

游戏事件循环

解决这个问题最简单的方法是定义一个事件循环。事件循环如下所示：

```
1 while (true)
2 {
3     processUserActions();
4     updateGame();
5 }
```

即使用户（玩家）没有采取任何行动，游戏仍然通过调用 `updateGame()` 函数继续运行。请注意，前面的代码只是对事件循环的介绍，它会无限循环并在每次迭代中处理和更新游戏。

每次循环迭代都会推进游戏状态。如果用户操作处理需要很长时间，它可能会阻塞循环，游戏会暂停一会儿。我们通常以每秒帧数 (FPS) 来衡量游戏速度。值越高，游戏就越流畅。

我们需要设计在游戏过程中持续运行的游戏循环。重要的是要将其设计成用户操作处理不会阻塞循环的方式。

游戏循环负责处理游戏中发生的一切，包括 AI。关于 AI，指的是之前讨论过的电脑玩家。除此之外，游戏循环还会处理角色的行动，并相应地更新游戏状态。

在深入研究游戏循环设计之前，让我们先了解一些能够帮助我们完成这一复杂任务的设计模式。毕竟，游戏循环是另一种设计模式！

使用设计模式

使用面向对象编程 (OOP) 范式设计游戏是很香的，游戏代表的是物体之间紧密互动的组合。在我们的策略游戏中，我们的建筑由单位建造。单位防御敌人单位等。这种内部交流导致了复杂性的增长。随着项目的发展和更多特性的增加，维护将变得更加困难。很明显，设计是建筑项目中最重要的部分之一。合并设计模式将大大改善设计过程和项目支持。

让我们来看看一些在游戏开发中有用的设计模式。我们将从经典模式开始，然后讨论更多特定于游戏的模式。

命令模式

开发人员将设计模式分为创建的、结构的和行为的三种类别。命令模式是一种行为设计模式。行为设计模式主要关注在对象之间的通信中提供灵活性。在此上下文中，命令模式将操作封装在一个对象中，该对象包含必要的信息以及操作本身。这样，命令模式的行为就像一个智能函数。在 C++ 中实现它的最简单方法是重载类的函数操作 ()，如下所示：

```
1 class Command
2 {
3     public:
4     void operator()() { std::cout << "I'm a smart function!"; }
5 };
```

带有重载函数操作符 () 的类有时称为仿函数。以上代码与下面的常规函数声明相同：

```
1 void myFunction() { std::cout << "I'm not so smart!" }
```

调用常规函数和 Command 类的对象看起来类似，如下所示：

```
1 myFunction();
2 Command myCommand;
3 myCommand();
```

当我们需要为函数使用状态时，这两者之间的区别很明显。为了存储常规函数的状态，我们使用静态变量。为了在对象中存储状态，我们使用对象本身。下面是如何跟踪重载函数操作符的调用次数：

```
1 class Command
2 {
3     public:
4         Command() : called_(0) {}
5
6     void operator()() {
7         ++called_;
8         std::cout << "I'm a smart function." << std::endl;
9         std::cout << "I've been called" << called_ << " times." << std::endl;
10    }
11    private:
12        int called_;
13 };
```

调用的数量对于命令类的每个实例是唯一的。下面的代码声明了两个 Command 实例，并分别调用它们两次和三次：

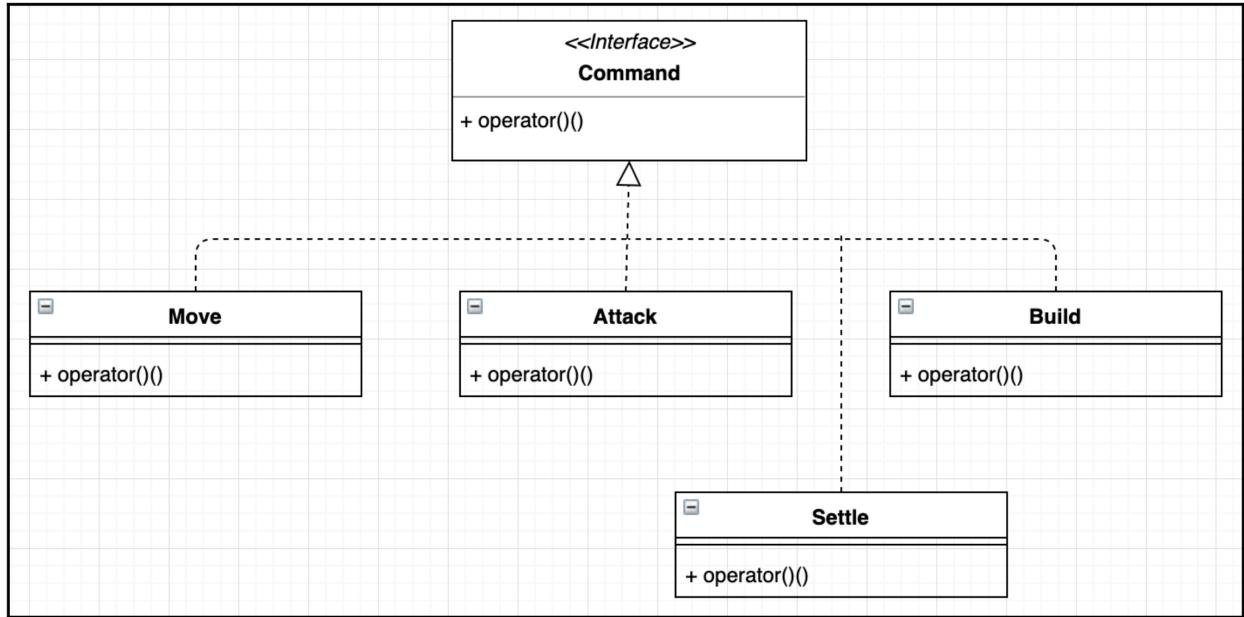
```
1 Command c1;
2 Command c2;
3 c1();
4 c1();
5 c2();
6 c2();
7 c2();
8 // at this point, c1.called_ equals 2, c2.called_ equals 3
```

现在，让我们尝试将这个模式应用到我们的策略游戏中。游戏的最终版本有一个图形界面，允许用户使用各种按钮和鼠标点击来控制游戏。例如，要让一个角色单位建造一座房子，而不是一个兵营，我们应该在游戏面板上选择相应的图标。让我们想象一个带有游戏地图和一些控制游戏动态的按钮的游戏面板。

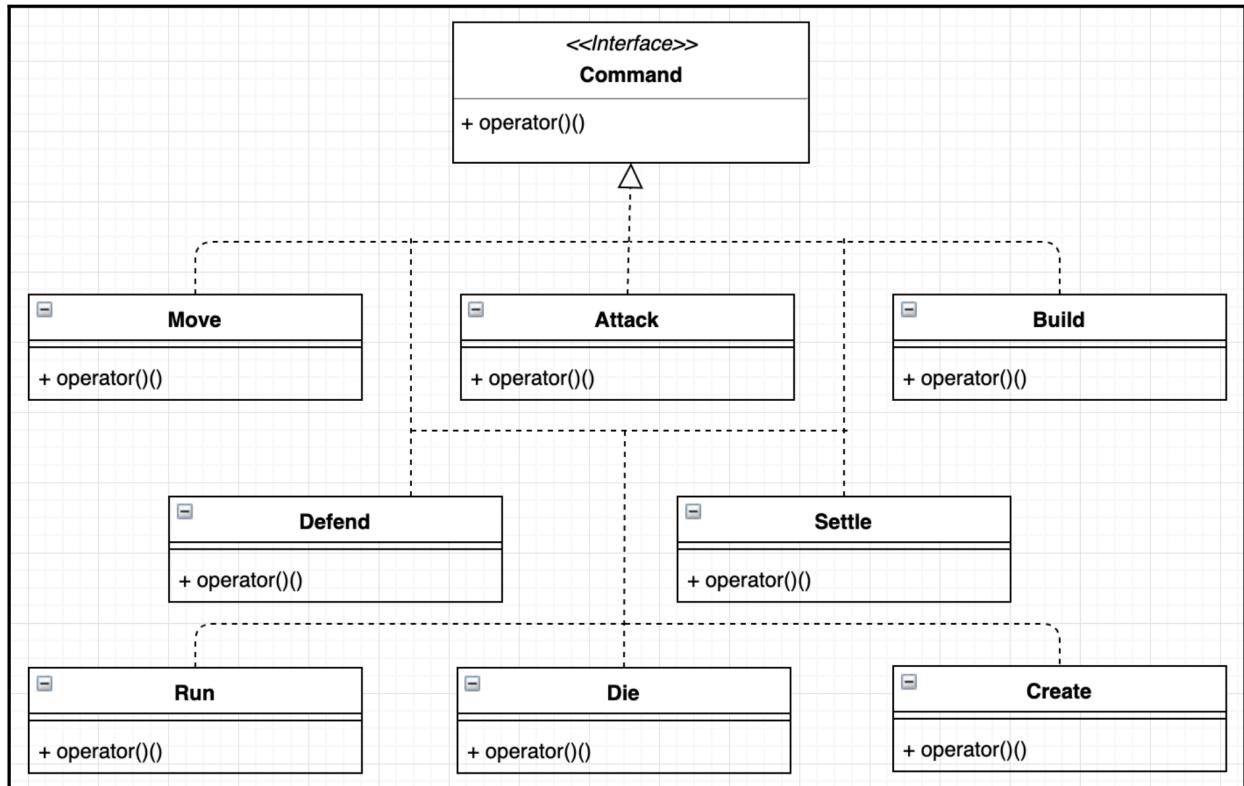
游戏向玩家提供以下命令：

- 将角色单位从 A 点移动到 B 点
- 攻击敌人
- 建造建筑
- 攻击建筑

游戏命令的设计如下：



每个类都封装了操作逻辑。客户端代码与处理操作无关。它使用命令指针进行操作，每个命令指针都指向具体的命令（如前面的图像所示）。注意，我们只描述了玩家将执行的命令。游戏本身使用命令在模块之间进行通信，自动命令的例子包括 Run、Defend、Die 和 Create。下面是一个更广泛的图表，展示了游戏中的命令：



上述命令执行游戏玩法中出现的任何事件。为了监听这些事件，我们应该考虑使用观察者模式。

观察者模式

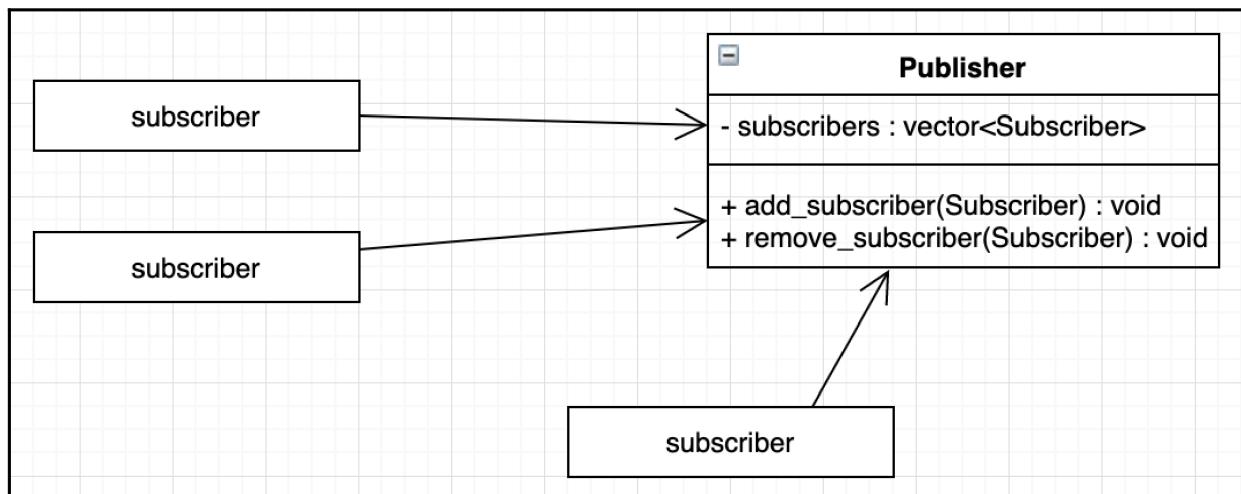
观察者模式是一种体系结构机制，允许我们关注对象状态更改，我们观察物体的变化。观察者模式也是一种行为设计模式。

大多数策略游戏都包含资源的概念。可能是石材、金币、木材等，例如：在建造一座建筑时，玩家需要花费 20 个木材，40 个石材和 10 个金币。最终，玩家将耗尽资源并需要收集这些资源。这个玩家创造了更多的角色单位，并让他们收集资源——几乎就像在现实生活一样。

现在，我们假设游戏中也有类似的资源收集。当玩家让单位收集资源时，他们应该在每次收集到固定数量的资源时通知我们。玩家是资源收集事件的关注者。

建筑也是如此。建筑物产生一个角色-玩家会收到通知。一个角色单位完成建筑建造-玩家会收到通知。我们更新玩家仪表板以保持玩家的游戏状态更新，玩家在玩游戏时可以了解拥有多少资源、多少单位和多少建筑。

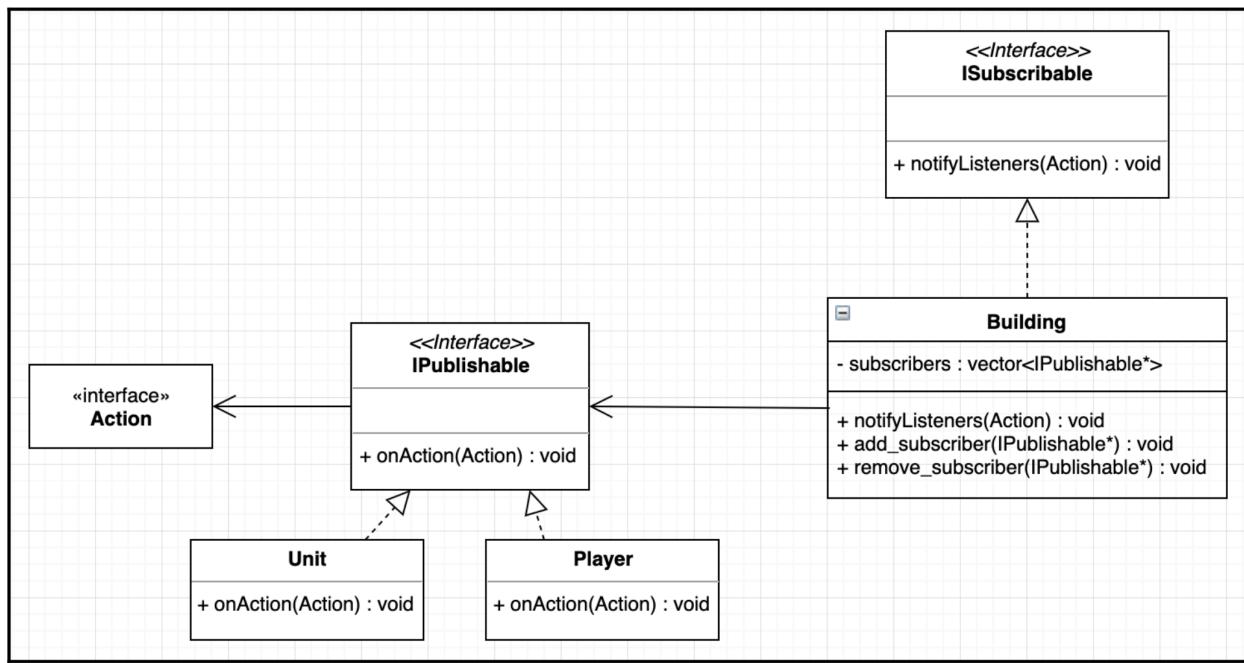
观察者涉及到实现一个类，该类存储其关注者并在事件上调用指定的函数。它由两个实体组成：关注者和执行者。如下图所示，用户数量不限于 1 个：



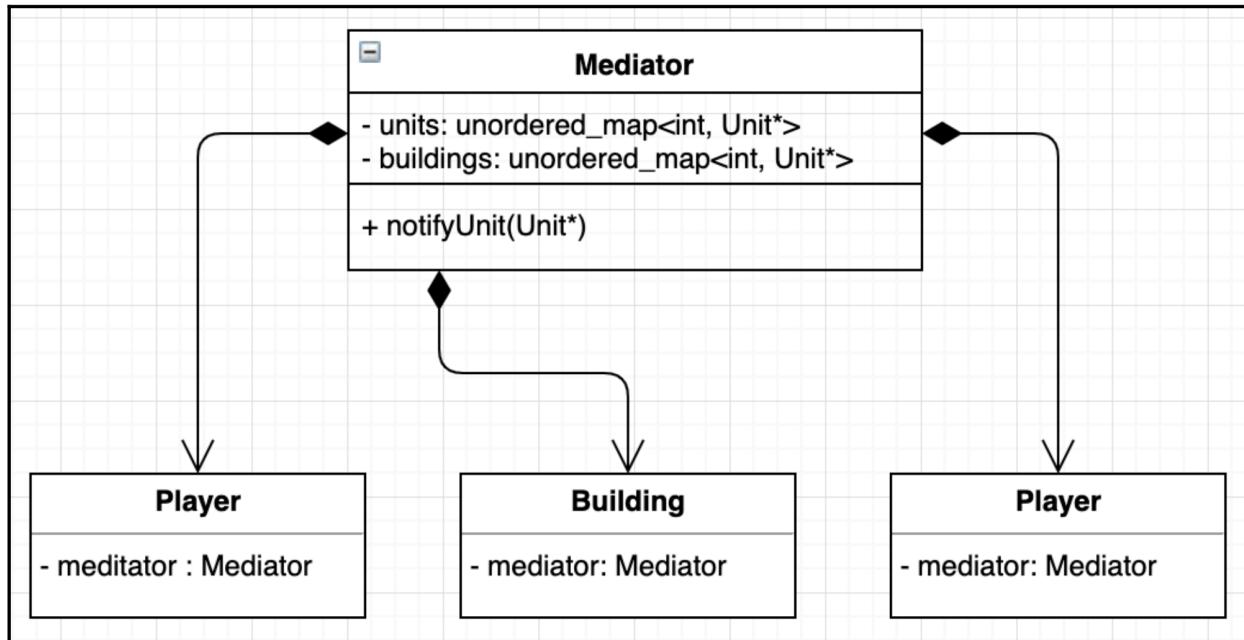
例如，当一个角色单位去建造一座建筑时，它会不断的去建造，除非被阻止。造成这种情况的原因有很多：

- 玩家决定取消建造建筑的过程。
- 角色单位必须防御敌人的攻击，并暂停建造过程。
- 建筑已经完成，所以角色单位停止了工作。

玩家还希望在建筑完成时得到通知，因为他们可能计划在完成建筑后让角色单位执行其他任务。我们可以设计构建流程，使其在事件完成时通知其侦听器（关注者）。下面的类图还涉及到一个操作接口。将其视为命令模式的实现：



针对观察者开发类会让我们意识到，游戏中几乎所有的实体都是关注者、执行者或两者兼有。如果您遇到类似的场景，您可以考虑使用中介——另一种行为模式。对象通过中介对象相互通信。触发事件的对象可以让中介知道该事件。然后，中介将消息传递给“订阅”到对象状态的任何相关对象。下图是中介集成的简化版本：



每个对象都包含一个中介，用于将更改通知关注者。中介对象通常包含所有相互通信的对象。对于事件，每个对象通过中介通知相关方，例如：当构建构建完成时，会触发中介，中介会通知所有关注方。要接收这些通知，每个对象都应该事先关注中介。

享元模式

享元是一种结构设计模式。结构模式负责将对象和类组装成更大、更灵活的结构。享元允许我们通过共享对象的公共部分来缓存对象。

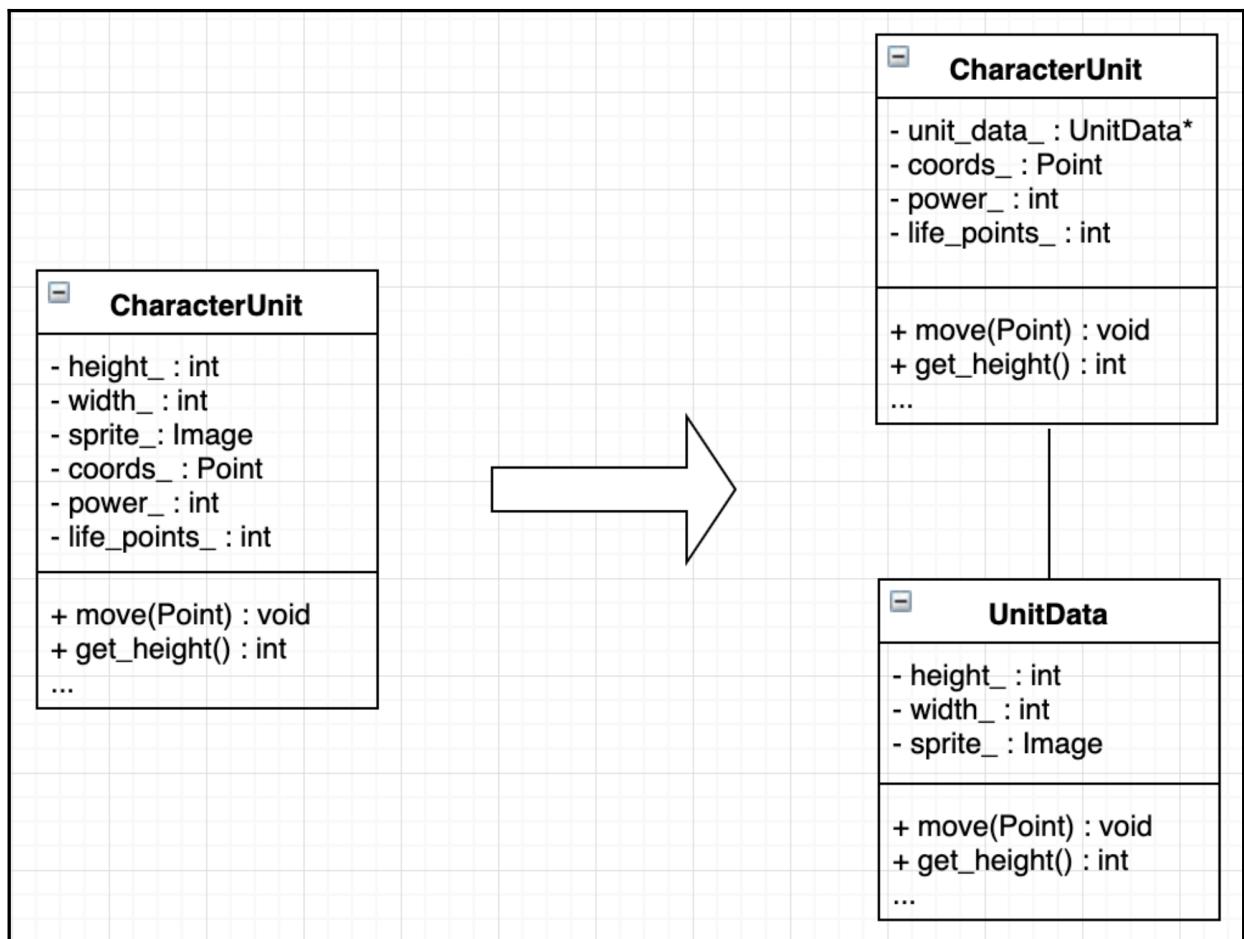
在策略游戏中，我们要处理许多呈现在屏幕上的对象。游戏过程中，对象的数量会增加。玩家玩游戏的时间越长，创造的角色单位和建筑就越多（敌人也是如此）。游戏中的每个单位代表一个包含数据的独立对象。字符单元至少占用 16 个字节的内存（对于它的两个整数数据成员和虚拟表指针）。

当我们为了在屏幕上呈现单位而添加额外字段时，情况就变得更糟了。例如，高度、宽度和精灵（代表渲染单元的图像）。除了角色单位之外，游戏还应该添加一些辅助道具，例如：树、石头等装饰性道具，以提升用户体验。某种程度上，我们在屏幕上渲染了大量对象，每个对象几乎代表相同的对象，但它们的状态有微小的差异。享元模式在这里起到了很大的作用。对于角色单位，它的高度、宽度和精灵在所有单位中存储了几乎相同的数据。

享元模式建议将一个重物体分解成两个：

- 一个不可变的对象，它为相同类型的每个对象包含相同的数据
- 一个可变对象，将自己与其他对象区分开来

例如，一个移动的角色单位有它自己的高度、长度和精灵，所有这些都在所有的角色单位中重复。因此，我们可以将这些属性表示为单个不可变对象，并为所有对象的属性提供相同的值。然而，角色单位在屏幕上的位置可能与其他单位不同，当玩家命令该单位移动到其他地方或开始建造建筑时，该单位的位置会不断变化，直到终点。在每一步中，单元都应该在屏幕上重新绘制。通过这样做，我们得到了如下设计：



左边是修改前的 CharacterUnit，而右边表示使用享元模式进行的最近的修改。游戏现在可以处理一堆 CharacterUnit 对象，而每一个都将存储一些 UnitData 对象的引用。这样，我们节省了很多内存。我们将每个单元唯一的值存储在 CharacterUnit 对象中。这些价值观会随着时间而改变。尺寸和精灵是恒定的，所以我们可以保持一个具有这些值的对象。这种不可变的数据称为内在状态，而对象的可变部分 (CharacterUnit) 称为外在状态。

我们有意地将数据成员移动到 CharacterUnit，从而将其从接口重新设计为抽象类。正如我们在第 3 章中所讨论的，抽象类几乎与可能包含实现的接口相同。move() 方法是所有类型单元的默认实现的。因为所有的单元都有共同的属性，派生类只提供必要的行为就好，比如：生命点和能量。

优化内存使用之后，我们应该处理复制对象。这款游戏涉及大量创造新对象。每个建筑都产生一个特定的角色单位，角色单位建造建筑，游戏世界本身呈现装饰元素（树木、岩石等）。现在，让我们尝试通过合并克隆功能来改进 CharacterUnit。在本章的前面，我们故意删除了复制构造函数和赋值操作符。现在，是时候提供一种从现有对象创建新对象的机制了。

原型模式

该模式允许我们独立于对象类型创建对象的副本。下面的代码代表了关于我们最近修改的 CharacterUnit 类的最终版本。我们还将添加新的 clone() 成员函数，以合并原型模式：

```

1 class CharacterUnit
2 {
3     public:

```

```

4 CharacterUnit() {}
5 CharacterUnit& operator=(const CharacterUnit&) = delete;
6 virtual ~Character() {}
7 virtual CharacterUnit* clone() = 0;
8
9 public:
10 void move(const Point& to) {
11     // the graphics-specific implementation
12 }
13 virtual void attack(const CharacterUnit&) = 0;
14 virtual void destroy() = 0;
15 int get_power() const { return power_; }
16 int get_life_points() const { return life_points_; }
17
18 private:
19 CharacterUnit(const CharacterUnit& other) {
20     life_points_ = other.life_points_;
21     power_ = other.power_;
22 }
23
24 private:
25 int life_points_;
26 int power_;
27 };

```

我们删除了赋值操作符，并将复制构造函数移动到 private 部分。派生类覆盖 clone() 成员函数，如下所示：

```

1 class Reader : public CharacterUnit
2 {
3 public:
4     Reader* clone() override {
5         return new Reader(*this);
6     }
7
8     // code omitted for brevity
9 };

```

原型模式将复制委托给对象。公共接口允许我们将客户端代码与对象的类解耦。现在，我们可以在不知道它是读者或士兵的情况下，复制一个角色单位。请看下面的例子：

```

1 // The unit can have any of the CharacterUnit derived types
2 CharacterUnit* new_unit = unit->clone();

```

当需要将对象转换为特定类型时，要强制将工作动态转换为良好。

在本节中，我们讨论了许多有用的设计模式。如果你不熟悉这些模式，这似乎有点难以应付，正确使用它们可以让我们设计灵活和可维护的项目。最后让我们回到我们之前介绍的游戏循环。

设计游戏循环

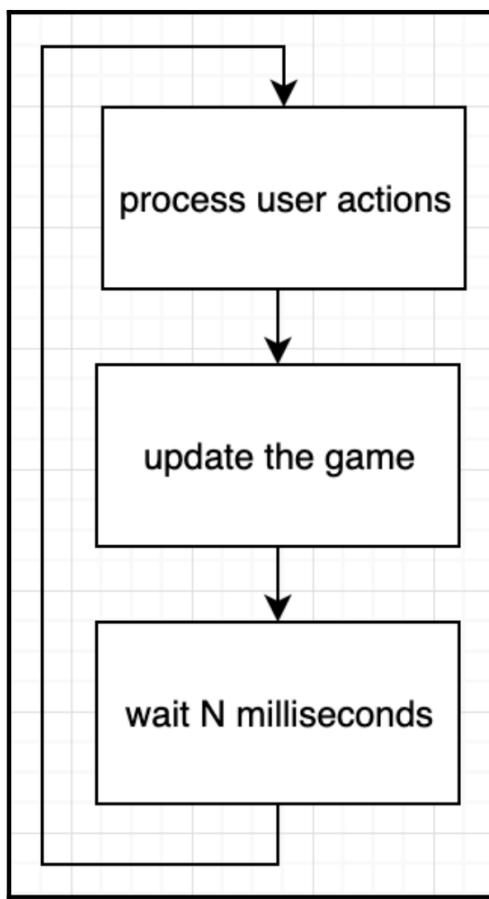
策略游戏的玩法变化最大。在每一个时间点上，许多动作同时发生。读者完成了他们的建筑，兵营生士兵，士兵攻击敌人，玩家命令单位移动、建造、攻击或奔跑等。游戏循环处理一切。通常，游戏引擎提供设计良好的游戏循环。

游戏循环在我们玩游戏时运行。正如我们已经提到的，循环处理玩家的动作，更新游戏状态，并呈现游戏（让玩家可以看到状态变化）。它在每次迭代中都这样做。循环还应该控制游戏玩法的速率，也就是 FPS。例如，如果你设计一款以 60 帧每秒运行的游戏，这意味着每帧大约需要 16 毫秒。

以下代码是在本章前面的简单游戏循环中使用的：

```
1 while (true)
2 {
3     processUserActions();
4     updateGame();
5 }
```

如果没有需要处理的用户操作，上述代码将快速运行。它在速度较快的机器上运行得更快（目标是 16 毫秒每帧）。这可能需要我们在处理动作和更新游戏状态后等待一段时间，如下图所示：



每次更新都将游戏时间提前一个固定的时间，这需要在现实世界中处理一个固定的时间。另一方面，如果一个帧的处理时间超过了指定的毫秒，游戏就会变慢。

如上图所示，游戏中发生的所有事情都包含在游戏的更新部分中。大多数时候，更新可能需要一次执行多个操作。此外，我们必须为游戏后台发生的一些操作保留计时器，这主要取决于游戏的细节。例如，建造一座建筑可以表现为两种状态：初始状态和最终状态。

在平面设计上，这两种状态应该代表两种不同的图像。第一张图片包含了建筑的一些基本部分，可能还包括一些围绕它的岩石，就好像它正在准备被建造一样。下一个图像代表了最终建成的建筑。当角色单位开始建造建筑时，我们便会向玩家呈现第一张图像（注：即围绕着一些岩石的基座）。当建筑完成后，我们用包含最终建筑的图像替换第一张图像。为了使这个过程更自然（更真实），我们人为地延长了时间。这意味着我们在图像的两个状态之间保持一个持续 30 秒或更多的计时器。

我们用最少的细节描述了最简单的情况。如果我们需要让游戏变得更加详细，例如：通过渲染建筑在建造过程中的每个变化，我们就应该在代表建筑每个步骤的图像之间保留大量计时器。在更新游戏后，我们等待 N 毫秒。等待更多毫秒会让游戏流程更接近现实生活。如果更新时间太长，导致玩家体验滞后怎么办？在这种情况下，我们需要优化游戏，使其符合用户的最佳体验。现在，假设更新游戏需要执行数百次操作，玩家实现了一个繁荣的帝国，现在正在建造许多建筑物，并且用许多士兵攻击敌人。

一个角色单位的每个行动，如从一个点移动到另一个点，攻击一个敌人单位，建造一座建筑等，都会在屏幕上及时呈现。现在，如果我们一次在屏幕上渲染数百个单位的状态会怎样？这就是我们使用多线程方法的地方。每个行动都涉及独立修改对象（注：对象是游戏中的任何单位，包括静态建筑）的状态。

总结

设计游戏是一项复杂的任务。我们可以将游戏开发视为一个独立的编程领域。游戏有不同的类型，其中之一就是策略游戏。策略游戏设计包括设计单位和建筑等游戏组件。通常情况下，策略游戏包括收集资源、建立帝国和与敌人战斗。游戏玩法包括游戏组件之间的动态交流，如角色单位建造建筑和收集资源，士兵防御敌人的土地等。

为了恰当地设计一款策略游戏，我们结合了 OOP 设计技巧和设计模式。设计模式在设计整个游戏及其组件的交互过程中扮演着重要角色。在本章中，我们讨论了命令模式，它将动作封装在对象中；观察者模式，用于关注对象事件；以及中介模式，该模式用于将观察者推进到组件之间复杂交互的级别。

游戏最重要的部分是它的循环。游戏循环控制渲染、游戏状态的及时更新和其他子系统。设计它涉及到使用事件队列和计时器。现代游戏使用网络，允许多个玩家通过互联网一起玩游戏。

在下一章中，我们将介绍 C++ 的网络编程，这样你就能将网络整合到你的游戏中。

问题

1. 重写一个私有虚拟函数的目的是什么？
2. 描述命令设计模式。
3. 享元模式如何节省内存使用？
4. 观察者模式和中介模式之间有什么区别？
5. 为什么我们要将游戏循环设计成一个无限循环？

扩展阅读

- Game Development Patterns and Best Practices: Better games, less hassle by John P.Doran, Matt Casanova: <https://www.amazon.com/Game-Development-Patterns-Best-Practices/dp/1787127834>

第 12 章：网络和安全

网络编程越来越受欢迎。大多数计算机都连接到互联网上，现在越来越多的应用程序依赖于此。从需要 Internet 连接的简单程序更新，到依赖稳定 Internet 连接的应用程序，网络编程正成为应用程序开发的必要部分。支持网络编程估计要推迟到以后的标准了，很可能要等到 C++23。

但我们可以提前通过处理网络应用程序做好准备。我们还将讨论网络的标准扩展，并看看在 C++ 中支持网络会是什么样子。本章将集中讨论网络的主要原理和驱动设备间通信的协议。作为一个开发者，设计网络应用程序是一个强大的技能。

开发人员经常面临的主要问题之一是应用程序的安全性。无论是与正在处理的输入数据相关，还是与使用经过验证的模式和实践进行编码相关，应用程序的安全性必须是第一位的。这对于网络应用程序尤为重要。本章中，我们还将深入探讨 C++ 中进行安全编程的技术和最佳实践。

本章中，我们将了解以下内容：

- 计算机网络概论
- C++ 编写套接字和套接字编程
- 设计网络应用
- 理解 C++ 中的安全问题
- 在项目开发中利用安全编程技术

编译器要求

g++ 编译器需要添加编译选项 `-std=c++2a` 来编译本章的代码。可以从这里获取本章的源码文件：<https://github.com/PacktPublishing/Expert-CPP>

用 C++ 进行网络编程

两台计算机通过网络相互作用。计算机通过一种称为网络适配器或网络接口控制器的特殊硬件组件连接到 Internet。安装在计算机上的操作系统提供与网络适配器一起工作的驱动程序，要支持网络通信，计算机必须有一个安装了支持网络堆栈的操作系统的网络适配器。所谓堆栈，我们指的是数据从一台计算机传输到另一台计算机时所经过的修改层。例如，在浏览器上打开一个网站会呈现通过网络收集到的数据。该数据以 0 和 1 的序列接收，然后转换为 Web 浏览器更容易理解的形式。分层在网络中是必不可少的，网络通信由符合我们将在这里讨论的 OSI 模型的几个层组成。网络接口控制器是支持 OSI(Open System Interconnection) 模型的物理链路层和数据链路层的硬件部件。

OSI 模型旨在对各种设备之间的通信功能进行标准化。设备在结构和组织上有所不同。这涉及到硬件和软件。例如，使用英特尔 CPU、运行 Android 操作系统的智能手机与运行 macOS Catalina 的 MacBook 电脑是不同的。不同的不是上述产品背后的名称和公司，而是硬件和软件的结构和组织。为了消除网络通信中的差异，提出了一套标准化协议和内部通信功能作为 OSI 模型。我们前面提到的层次如下：

- 应用层
- 表示层
- 会话层

- 传输层
- 网络层
- 数据链路层
- 物理层

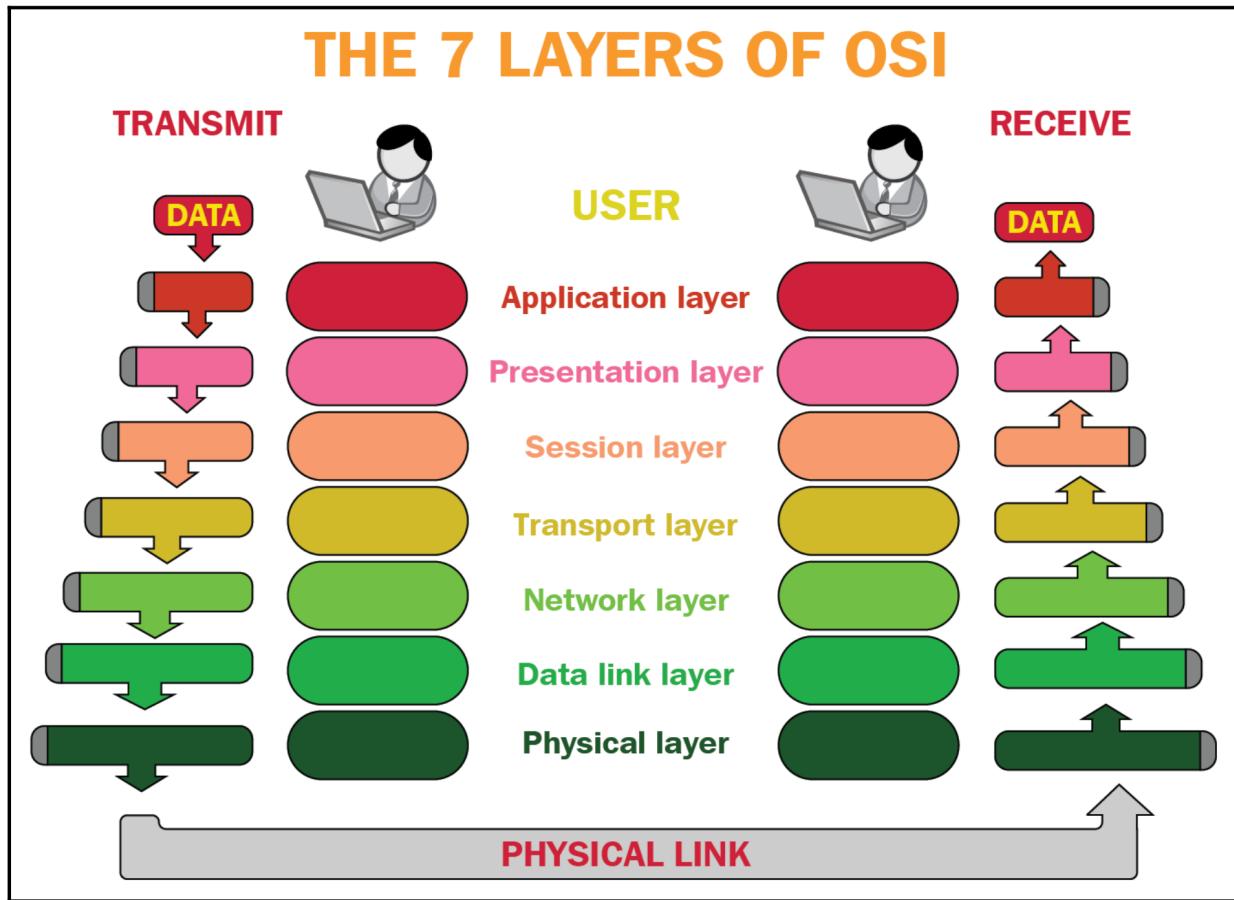
可以简化成四个层：

- 应用层：处理特定应用程序的细节。
- 传输层：提供两台主机之间的数据传输。
- 网络层：处理数据包在网络上的传输。
- 数据链路层：这包括操作系统中的设备驱动程序，以及计算机内部的网络适配器。

链路（或数据链路）层包括操作系统中的设备驱动程序，以及计算机中的网络适配器。

为了理解这些层，我们假设正在使用一个用于消息传递的桌面应用程序，如 Skype 或 Telegram。当你输入一条信息并点击发送按钮时，信息就会通过网络到达目的地。这个场景中，假设正在向计算机上安装了相同应用程序的朋友发送一条文本消息。从高级的角度来看，这似乎很简单，但这个过程是复杂的，即使是最简单的消息在到达目的地之前也要经历大量转换。首先，当您点击发送按钮时，文本消息将转换为二进制形式。网络适配器使用二进制文件进行操作。它的基本功能是通过媒体发送和接收二进制数据。除了通过网络发送的实际数据外，网络适配器还应该知道数据的目的地址。目标地址是附加到用户数据的许多属性之一。所谓用户数据，指的是输入并发送给朋友的文本。目的地址是你朋友计算机的唯一地址，输入的文本与目的地地址和其他发送到目标所必需的信息打包在一起，您朋友的计算机（包括网络适配器、操作系统和消息传递应用程序）接收并解包数据，包中包含的文本然后由消息传递应用程序呈现在屏幕上。

本章开头提到的每一层 OSI 都将其特定的头添加到通过网络发送的数据中。下图描述了来自应用层的数据在移动到目的地之前，是如何与报头堆叠的：

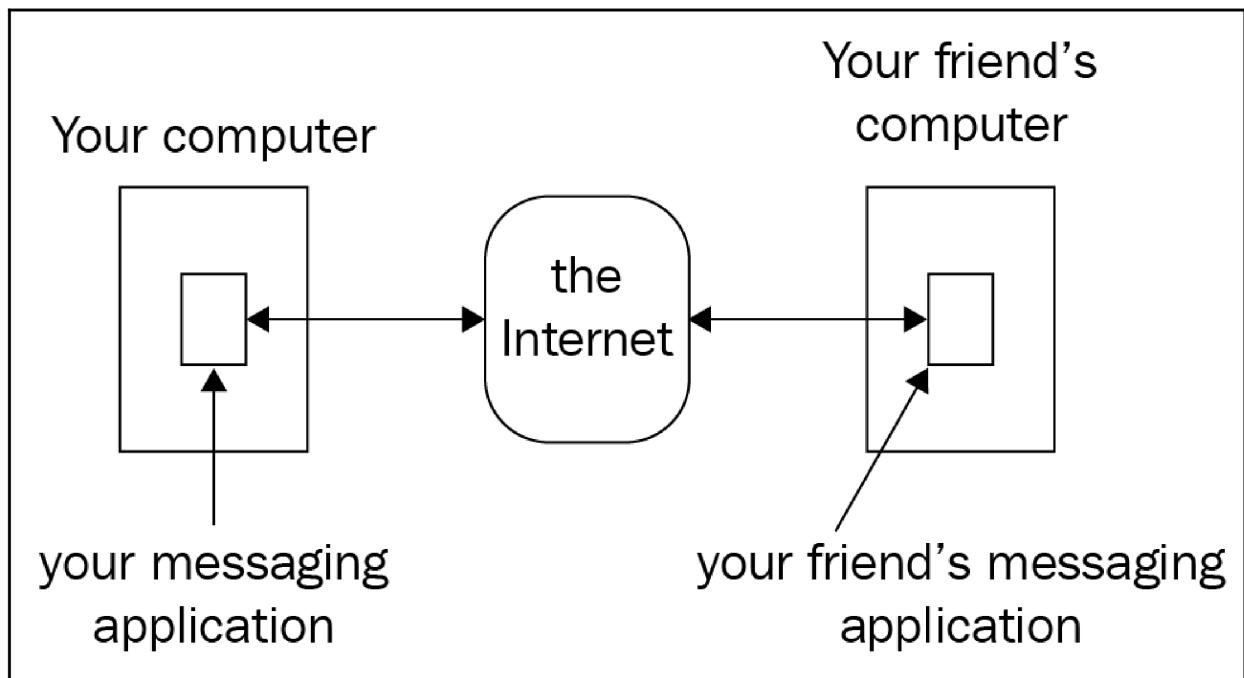


上图中的第一行（应用层）。数据是您在消息传递应用程序中输入的文本，以便将其发送给您的朋友。每一层数据都会向下传递，直到到物理层，数据打包与报头特定于 OSI 模型的每一层相对应。另一边的计算机接收和检索打包的数据。在每一层中，它删除特定于该层的头，并将包的其余部分移到下一层。最后，数据到达朋友的消息应用程序。

作为开发者，我们主要关心的是编写能够通过网络发送和接收数据的应用程序，而不需要深入了解层的细节。然而，我们需要对层如何使用头文件在更高层次上增加数据的理解很少。让我们了解网络应用程序在实践中是如何工作的。

底层的网络应用程序

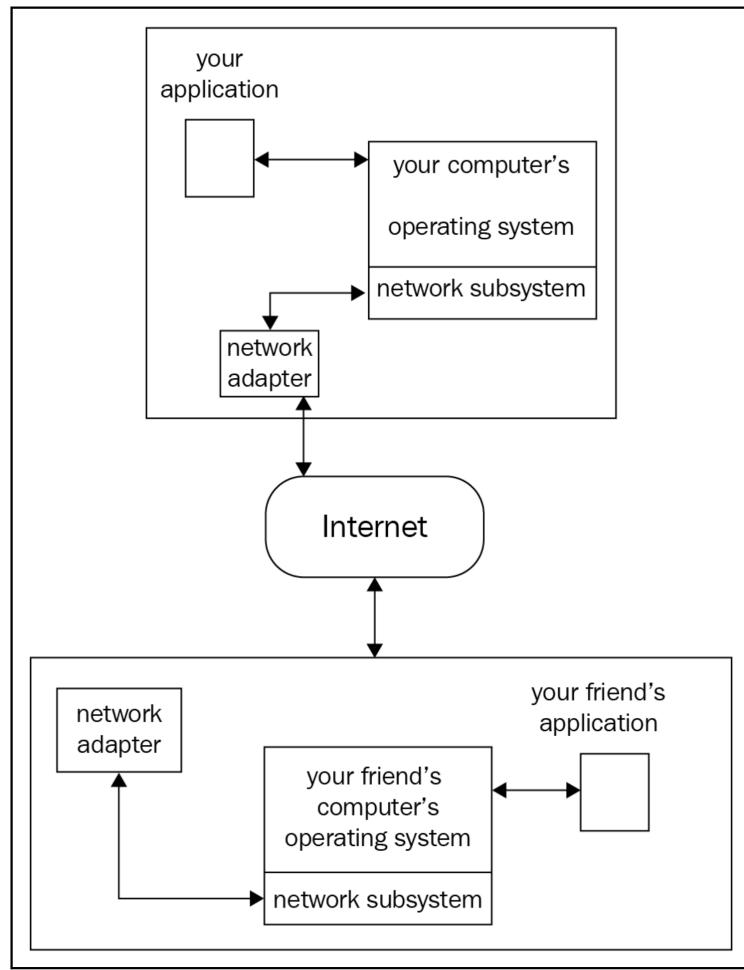
安装在设备上的网络应用程序通过网络与安装在其他设备上的其他应用程序进行通信。本章中，我们将讨论通过 Internet 协同工作的应用程序。下面的图表可以看到这种交流的上层概述：



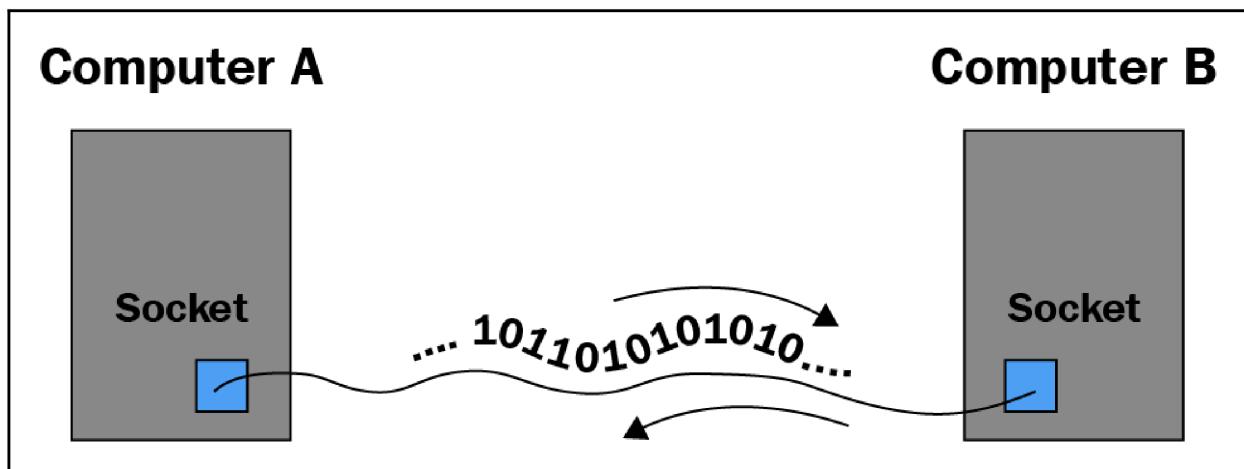
通信的最低层是物理层，它通过媒体传输数据位。在这种情况下，一种媒介就是网线（也可以考虑 Wi-Fi 通信）。用户应用程序从较底层的网络通信中抽象出来。操作系统提供了程序员所需要的一切。操作系统实现了网络通信的底层细节，比如传输控制协议/Internet 协议 (TCP/IP) 套件。

当应用程序需要访问网络时，无论是局域网还是 Internet，都会请求操作系统提供一个接入点。操作系统设法通过使用一个网络适配器和特定的软件与硬件对话来提供一个网络网关。

更详细的说明如下：



操作系统提供了与其网络子系统一起工作的 API。开发者应该关心的主要抽象是套接字。我们可以将套接字视为通过网络适配器发送内容的文件。套接字是通过网络连接两台计算机的接入点，如下图所示：



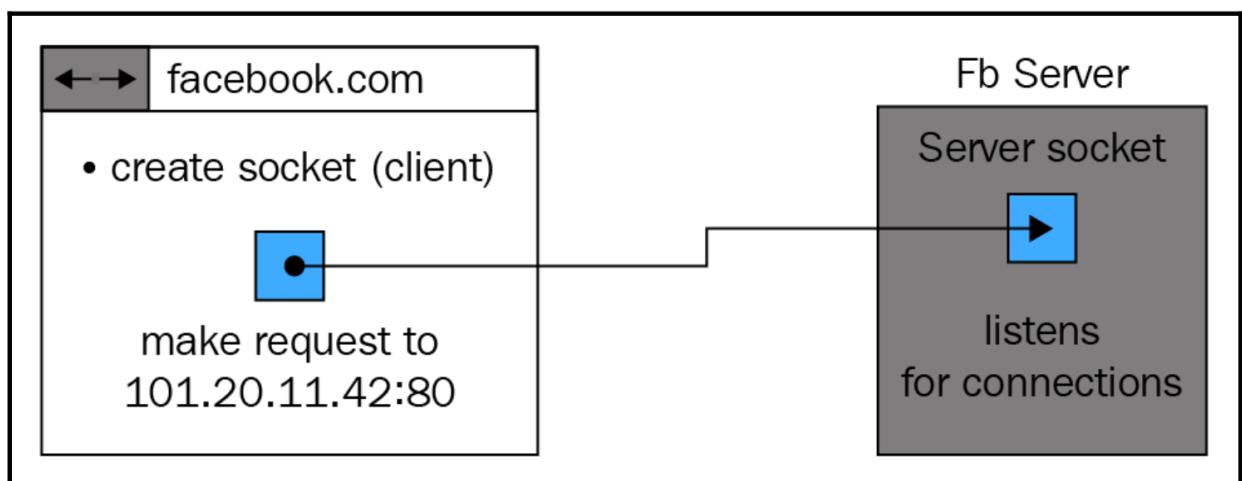
从开发者的角度来看，套接字是一种结构，它允许我们在应用程序中通过网络实现数据。套接字是发送或接收数据的连接点，应用程序通过套接字接收数据。操作系统根据请求为应用程序提供套接字，一个应用程序可以有多个套接字。客户机-服务器体系结构中的客户机应用程序通常使用单个套接字进行操作。现在，让我们详细研究一下套接字编程。

使用套接字编写网络应用

如前所述，套接字是对网络通信的抽象。我们将它们视为常规文件——所有写入套接字的内容都由操作系统通过网络发送到目的地。所有通过网络接收到的东西都被写进套接字——同样是由操作系统写的。通过这种方式，操作系统为网络应用程序提供了双向通信。

我们假设我们在网络上运行两个不同的应用程序。例如，打开一个 Web 浏览器，并使用一个消息传递应用程序（如 Skype）与朋友聊天。Web 浏览器表示客户机-服务器网络体系结构中的客户机应用程序。本例中，服务器是响应请求数据的计算机。例如，在 Web 浏览器的地址栏中输入一个地址，然后在屏幕上看到生成的 Web 页面。每当我们访问一个网站时，Web 浏览器都会向操作系统请求一个套接字。编码方面，Web 浏览器使用操作系统提供的 API 创建套接字。我们可以用一个更具体的前缀来描述这个套接字：客户端套接字。为了让服务器处理客户端请求，运行 Web 服务器的计算机必须侦听传入的连接，这时服务器应用创建了一个用于侦听连接的服务器套接字。

只要客户端和服务器之间建立了连接，就可以进行数据通信。下图描述了一个 Web 浏览器对 facebook.com 的请求：



请注意上图中的数字组。这被称为 Internet 协议 (IP) 地址。IP 地址是我们需要的位置，以便将数据传输到设备。有数十亿的设备连接到互联网上。为了在它们之间做出区分，每个设备都公开一个表示其地址的唯一数值。连接是使用 IP 协议建立的，我们称它为 IP 地址。一个 IP 地址由四组 1 字节长度的数字组成。它的点分十进制表示形式为 X.X.X.X，其中 X 是 1 字节数。每个位置的取值范围为 0 到 255。更具体地说，这是一个版本 4 的 IP 地址。现代系统使用 IPv6 地址，它是数字和字母的组合，提供更广泛的可用地址。

创建套接字时，我们将本地计算机的 IP 地址分配给，我们将套接字绑定到地址。当使用套接字向网络中的另一个设备发送数据时，我们应该设置它的目的地址。目的地址被该设备上的另一个套接字持有。为了在两个设备之间创建连接，我们使用两个套接字。一个合理的问题可能会出现——如果有几个应用程序在设备上运行怎么办？如果我们运行几个应用程序，每个应用程序都为自己创建了一个套接字，那会怎么样？哪一个应该接收传入的数据？

要回答这些问题，请仔细看看上面的图表。应该在 IP 地址的末尾看到冒号后的数字，这称为端口号。端口号是一个由操作系统分配给套接字的 2 字节长度的数字。由于 2 字节的长度限制，操作系统不能分配超过 65,536 个唯一的端口号给套接字，不能有超过 65,536 个同时运行的进程或线

程通过网络进行通信（但是，有一些方法可以重用套接字）。除此之外，还有一些端口号保留给特定的应用程序。这些端口称为知名端口，范围为 0 ~ 1023。它们是预留给特权服务的。例如，HTTP 服务器的端口号为 80。这并不意味着它不能使用其他端口。

我们学习如何在 C++ 中创建套接字。我们将设计一个封装可移植操作系统接口（POSIX）套接字的包装器类，也称为 Berkeley 或 BSD 套接字，其有一组用于套接字编程的标准函数。C++ 的网络编程扩展将是对该语言的巨大补充，草案中包含了网络接口的信息。我们将在本章后面讨论这一点。在此之前，让我们尝试为现有的底层库创建我们自己的网络包装器。当我们使用 POSIX 套接字时，我们依赖于操作系统的 API。操作系统提供了一个 API，该 API 表示用于创建套接字、发送和接收数据等的函数和对象。

POSIX 将套接字表示为文件描述符。我们几乎把它当作一个常规文件来使用。文件描述符遵循 UNIX 哲学，为数据输入/输出提供公共接口。下面的代码会使用 `socket()` 函数（定义在 `<sys/socket.h>` 头文件中）：

```
1 int s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

`socket()` 函数的声明如下：

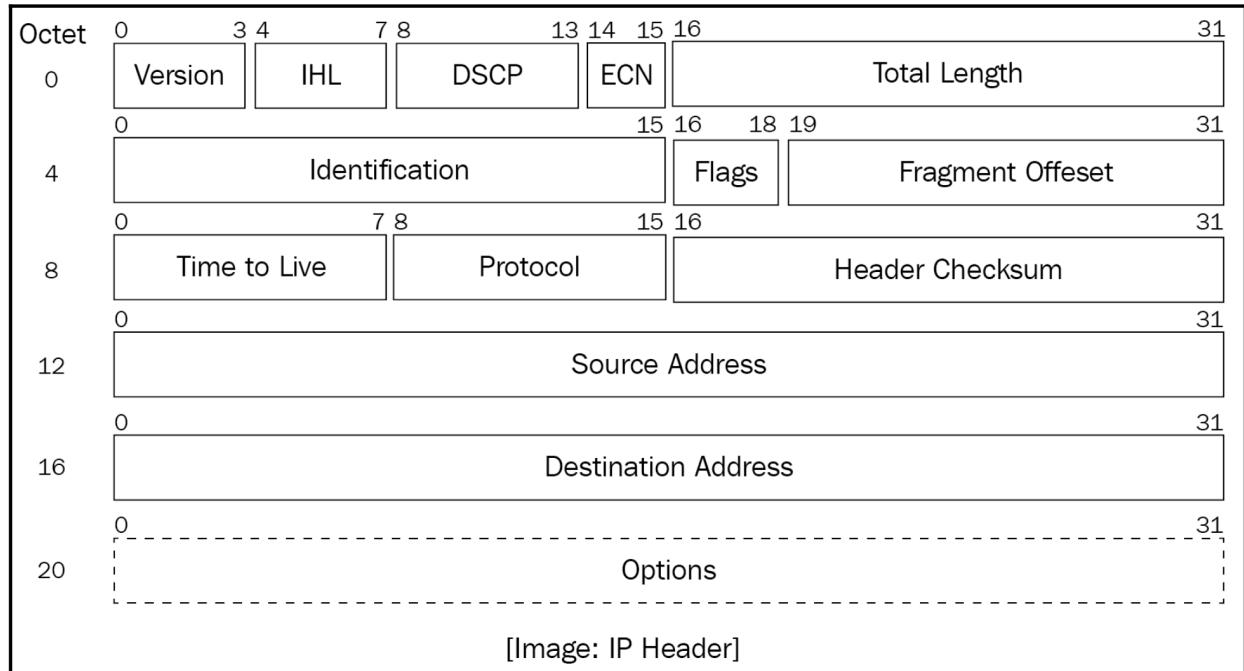
```
1 int socket(int domain, int type, int protocol);
```

因此，`AF_INET`、`SOCK_STREAM` 和 `IPPROTO_TCP` 都是数值。`domain` 参数指定套接字的协议族。我们使用 `AF_INET` 来指定 IPv4 协议。对于 IPv6，我们使用 `AF_INET6`。第二个参数指定套接字的类型，即它是面向流的套接字还是数据报的套接字。对于每个特定类型，应该相应地指定最后一个参数。在前面的示例中，我们使用 `IPPROTO_TCP` 指定了 `SOCK_STREAM`。传输控制协议（TCP）代表了一个可靠的面向流的协议，这就是我们将类型参数设置为 `SOCK_STREAM` 的原因。在实现套接字应用之前，让我们了解更多关于网络协议的信息。

网络协议

网络协议是定义应用程序之间相互通信的规则和数据格式的集合。例如，一个 Web 浏览器和一个 Web 服务器通过超文本传输协议（HTTP）进行通信。HTTP 更像是一组规则，而不是一个传输协议。传输协议是每一个网络通信的基础。传输协议的一个例子是 TCP。我们提到了 TCP/IP 套件，我们指的是基于 IP 的 TCP 实现。我们可以把网络协议（IP）看作是网络通信的核心。

它提供主机到主机的路由和寻址。我们通过互联网发送或接收的所有东西都打包成一个 IP 包。下面是 IPv4 数据包的样子：



IP 报头为 20 字节，它结合了将包从源地址发送到目的地址所需的标志和选项。在 IP 协议的域中，我们通常称一个包为数据报。每一层都有其特定的数据包术语。更谨慎的专家讨论将 TCP 段封装到 IP 数据报中。把它们叫做包是完全可以的。

更高层次上的每个协议都在通过网络发送和接收的数据上附加元信息，例如：TCP 数据封装在 IP 数据报中。除了这个元信息之外，协议还定义了底层的规则和操作，这些规则和操作应该被执行以完成两个或更多设备之间的数据传输。



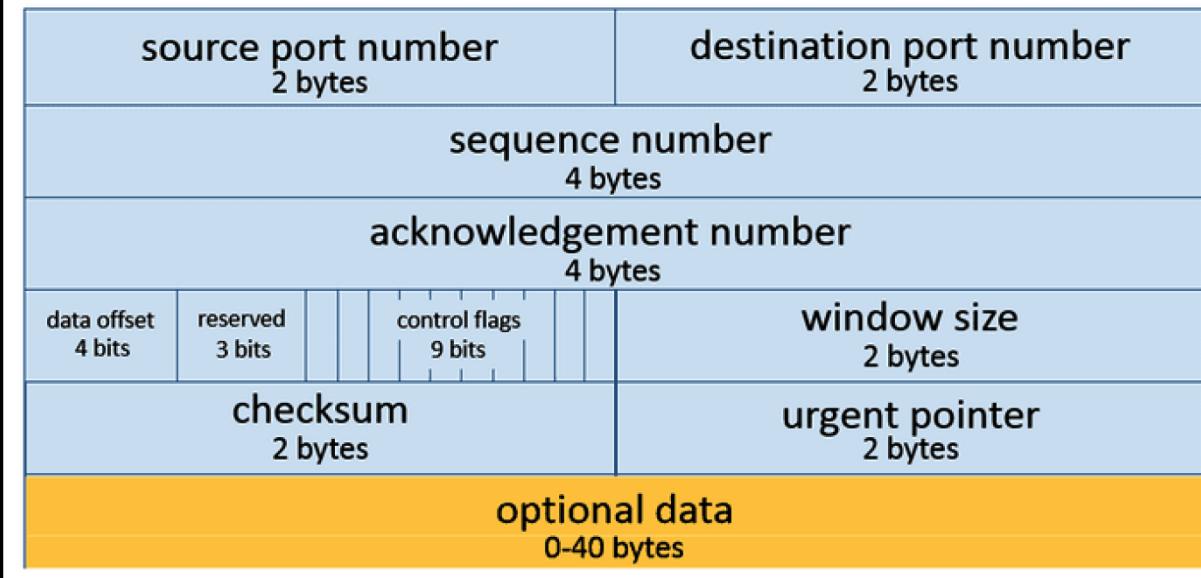
可以在称为请求注释 (Request for Comments, RFC) 的特定文档中找到更详细的信息。例如，RFC 791 描述了 Internet 协议，而 RFC 793 描述了传输控制协议。

许多流行的应用程序——文件传输、电子邮件、Web 等等——使用 TCP 作为它们的主要传输协议。例如，HTTP 协议定义了从客户机到服务器传输的消息的格式，反之亦然。实际的传输是使用传输协议进行的——本例中是 TCP。然而，HTTP 标准并没有将 TCP 限制为唯一的传输协议。

下图说明了在将 TCP 报头传递到底层之前，添加了 TCP 报头：

Transmission Control Protocol (TCP) Header

20-60 bytes



注意源端口号和目的端口号。这些是区分操作系统中运行的进程的唯一标识符。另外，看一下序列号和确认号。他们是 TCP 特定的，并用于传输可靠性。

实际应用中，TCP 主要有以下特点：

- 重新传输丢失的数据
- 顺序发送
- 数据完整性
- 避免拥塞

IP(Internet Protocol) 不可靠，它不关心丢失的数据包，这就是为什么 TCP 处理丢失数据包的重传。它用唯一的标识符标记每个包，该标识符应被传输的另一方确认。如果发送方没有收到包的确认码 (ACK)，协议将重新发送包 (有限的次数)。以正确的顺序接收数据包也是至关重要的。TCP 重新排序接收到的数据包，以表示正确排序的信息。这就是为什么，当我们在网上听音乐的时候，我们不会在开头听歌曲的结尾。

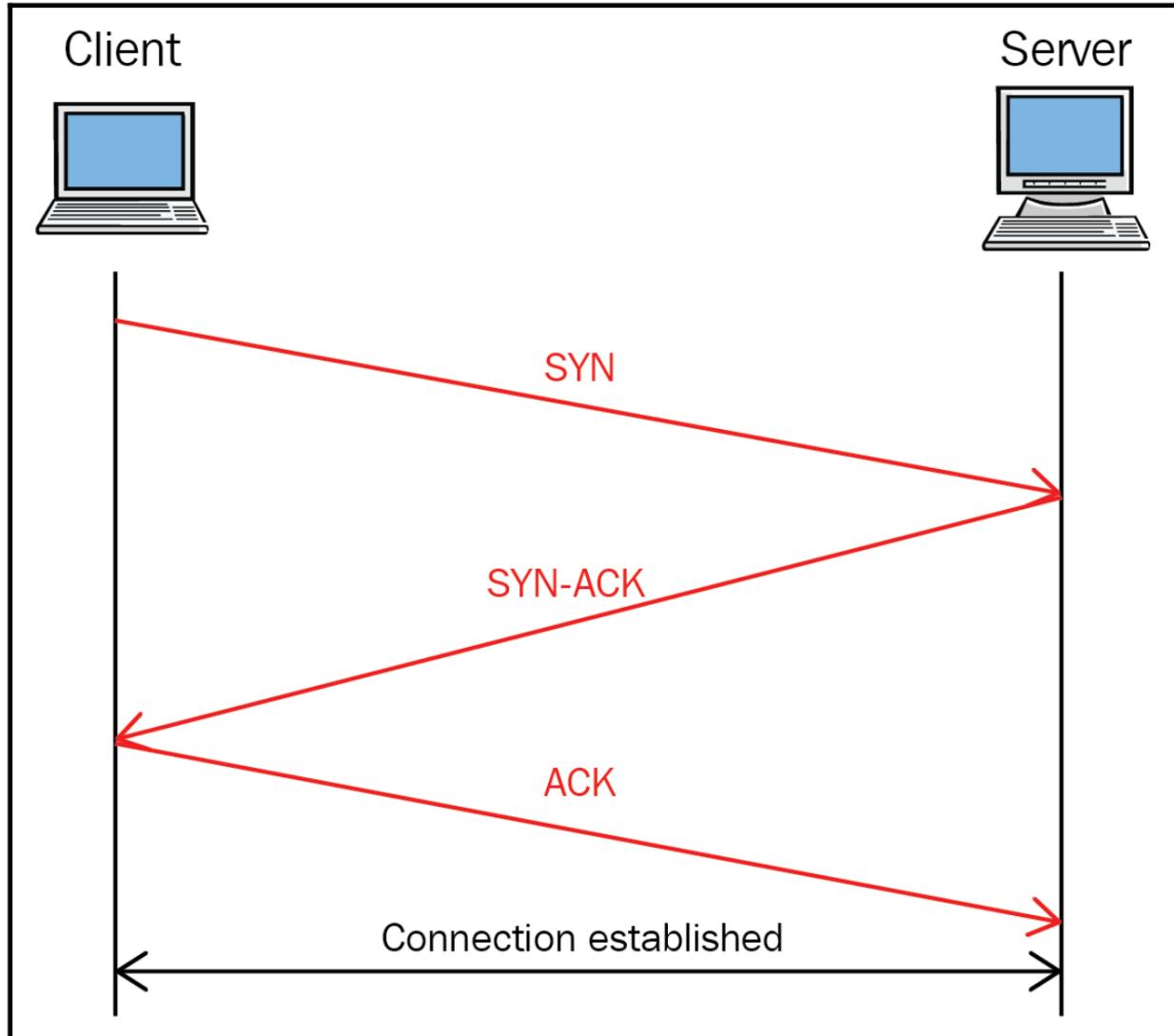
数据包的重传可能会导致另一个称为网络拥塞的问题。当节点发送数据包的速度不够快时就会发生这种情况。数据包会被卡住一段时间，不必要的重传增加了它们的数量。TCP 的各种实现都采用了避免拥塞的算法。

它维护一个阻塞窗口——一个决定可以发送出去的数据量的因素。TCP 使用慢启动机制，在初始化连接后缓慢增加拥塞窗口。尽管在相应的请求注释 (Request for Comments, RFC) 中描述了该协议，但在操作系统中有很多不同的实现机制。

另一个是用户数据报协议 (UDP)。与 TCP 的主要区别是 TCP 是可靠的。在丢失网络包的情况下，它重新发送相同的包，直到它到达指定的目的地。由于它的可靠性，通过 TCP 传输数据被认为比使用 UDP 需要更长的时间。UDP 不能保证我们可以正确地传递数据包而不会有损失。相反，开发人员应该注意重新发送、检查和验证数据传输。需要快速通信的应用程序往往依赖 UDP。例如，视频电话应用程序或在线游戏使用 UDP，因为它的即时性。即使在传输过程中丢失了几个

包，也不会影响用户体验。在玩游戏或与朋友视频聊天时出现小故障比等待下一帧游戏或视频好得多。

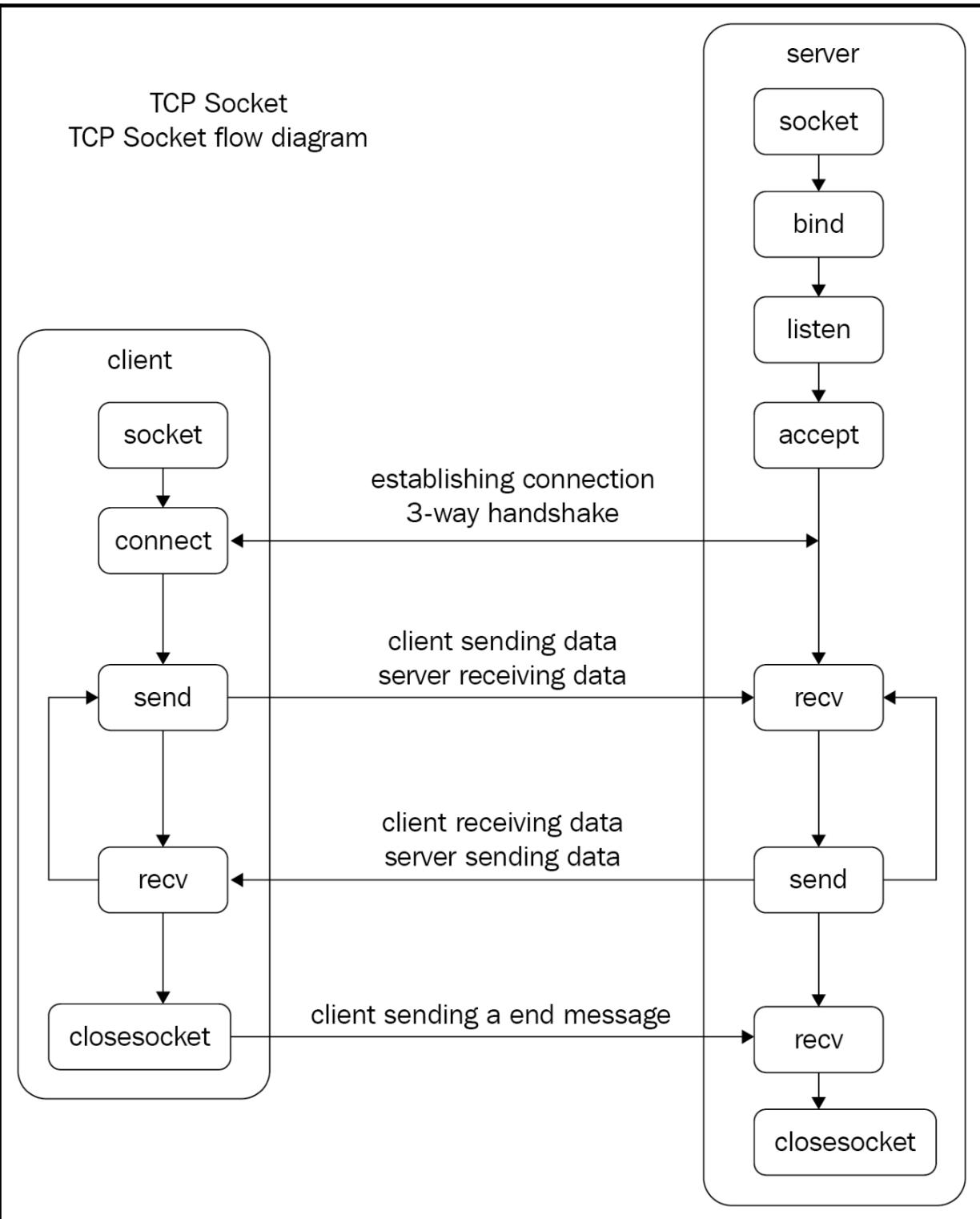
TCP 比 UDP 慢的一个主要原因是 TCP 的连接发起过程中有更多的步骤。下图展示了 TCP 连接建立的过程，也称为三次握手：



客户端在向服务器发送 SYN 包时会选择一个随机数。服务器将这个随机数加 1，然后选择另一个随机数，然后用一个 SYN-ACK 报文进行应答。客户机将从服务器接收到的两个数字都加 1，并通过向服务器发送最后一个 ACK 来完成握手。三次握手成功完成后，客户端和服务器就可以相互传输数据包了。这个连接建立过程适用于每一个 TCP 连接。对网络应用程序的开发人员隐藏了握手的细节。我们创建套接字并开始侦听传入的连接。

请注意这两种端点之间的区别。其中一个是客户。在实现网络应用程序时，我们应该明确区分客户端和服务器，因为它们有不同的实现。这也与套接字类型有关。当创建一个服务器套接字时，我们让它监听传入的连接，而客户端不监听——发出请求。下图描述了客户端和服务器的某些函数及其调用顺序：

TCP Socket TCP Socket flow diagram



在代码中创建套接字时，我们指定套接字的协议和类型。当我们想要在两个端点之间建立可靠的连接时，我们选择 TCP。有趣的是，我们可以使用 TCP 这样的传输协议来构建我们自己的协议。假设我们定义了要发送和接收的特殊文档格式，以便有效地处理通信。例如，每个文档都应该以单词 pack 开头。HTTP 以同样的方式工作。它使用 TCP 进行传输，并在其上定义一种通信格式。在 UDP 的情况下，我们还应该设计并实现一个通信的可靠性策略。上面的图显示了 TCP 如何在两个端点之间建立连接。客户机向服务器发送 SYN 请求。服务器用 SYN-ACK 响应进行应答，

让客户机知道可以继续握手。最后，客户机用一个 ACK 应答服务器，声明连接正式建立。他们可以随时交流。



同步 (SYN) 和 ACK 是协议定义的术语，在网络编程中很常见。

UDP 不以这种方式工作。它将数据发送到目的地，而不必担心已建立的连接。如果你使用 UDP，但需要一些可靠性，你应该自己实现它，例如：通过检查数据的一部分是否到达目的地，可以用一个自定义的 ACK 包等待目的地的应答检查它。大多数面向可靠性的实现可能重复已经存在的协议，比如 TCP。然而，在许多情况下，您并不需要它们，例如：不需要避免拥塞，因为不需要发送相同的数据包两次。

在前一章我们设计了一款策略游戏。假设这是一款在线游戏，你正在与一个真正的对手对战，而不是一个自动的敌人玩家一起游戏。游戏的每一帧都是基于网络接收到的数据进行渲染的。如果我们努力确保数据传输的可靠性，增加数据的完整性，并确保没有数据包丢失，用户体验可能会因为玩家的不同步而受到损害，这个场景很适合使用 UDP。我们可以在不使用重传策略的情况下实现数据传输，从而保证游戏的速度。当然，使用 UDP 并不会缺少可靠性。在相同的场景中，我们可能需要确保数据包被玩家成功接收。例如，当玩家投降时，我们应该确保对手收到消息。因此，我们可以根据数据包优先级有条件的可靠性。UDP 在网络应用中提供了非常高灵活性和速度。

让我们看一下 TCP 服务器应用程序的实现。

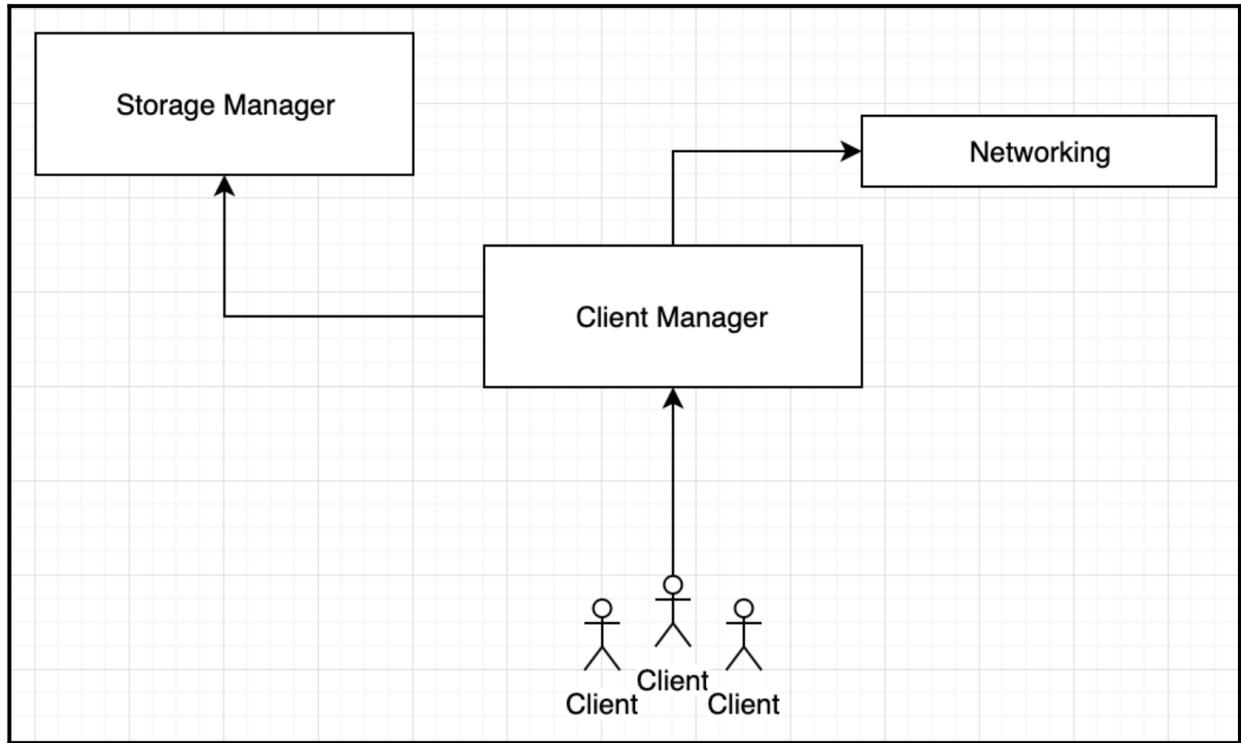
设计一个网络应用

设计带有需要网络连接的子系统的应用程序的方法，与完全与网络相关的应用程序不同。后者的一个例子可能是用于文件存储和同步的客户机-服务器应用程序（如 Dropbox）。它由服务器和客户端组成，其中客户端作为桌面或移动应用程序安装，也可以用作文件资源管理器。Dropbox 控制的系统中文件的每次更新都将立即与服务器同步。这样，文件就会一直保存在云端，只要有网络连接，你就可以在任何地方访问它们。

我们将为文件存储和操作设计一个类似的简化服务器应用程序。服务器的主要任务如下：

- 从客户端应用程序接收文件
- 将文件存储在指定的位置
- 客户端请求时发送文件

参考第 10 章，我们可以进一步了解应用程序的顶层设计：



上图中的每个矩形表示一个类或与特定任务相关的类的集合。例如，存储管理器处理与存储和检索文件相关的一切。在这一点上，是否使用诸如文件、位置、数据库等类我们并不太担心。

客户端管理器是一个或一组类，表示处理与客户端身份验证或授权（通过客户端，我们指的是客户端应用程序）、与客户端保持稳定连接、从客户端接收文件、向客户端发送文件等等相关的一切。

在本章中，我们特别强调了网络是一个值得关注的实体。所有与网络连接相关的事情，以及与客户机之间的数据传输，都是通过网络来处理的。现在，让我们看看可以使用哪些功能来设计 Networking 类（为了方便起见，我们将其称为 Network Manager）。

使用 POSIX 套接字

如前所述，socket()、bind() 和 accept() 等函数是大多数 Unix 系统默认支持的库函数，包含在 <sys/socket.h> 头文件中。除此之外，我们还需要其他几个头文件。让我们实现经典的 TCP 服务器示例，并将其封装在文件传输应用程序服务器的网络模块中。

正如我们前面提到的，服务器端开发在套接字类型及其行为方面不同于客户端开发。尽管双方都使用套接字操作，但服务器端套接字一直在监听传入的连接，而客户端套接字则发起与服务器的连接。为了让服务器套接字等待连接，我们创建了一个套接字，并将其绑定到客户机将尝试连接到的服务器 IP 地址和端口号。下面的 C 代码表示 TCP 服务器套接字的创建和绑定：

```

1 int s = socket(AF_INET, SOCK_STREAM, 0);
2
3 struct sockaddr_in server;
4 server.sin_family = AF_INET;
5 server.sin_port = htons(port);
6 server.sin_addr.s_addr = INADDR_ANY;
7

```

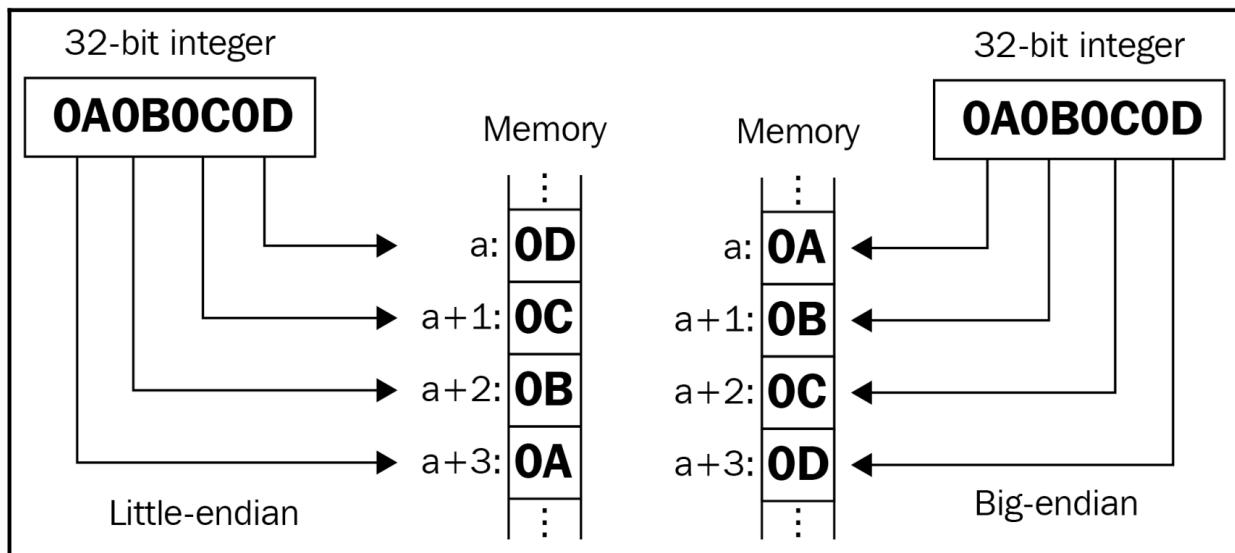
```
8 bind(s, (struct sockaddr*)&server, sizeof(server));
```

第一个调用创建一个套接字。第三个参数设置为 0，表示将根据套接字的类型选择默认协议。类型作为第二个参数 SOCK_STREAM 传递，它使协议值在默认情况下等于 IPPROTO_TCP。函数的作用是：将套接字与指定的 IP 地址和端口号绑定。我们在 sockaddr_in 结构中指定了它们，该结构将网络地址相关的细节组合在了其中。



尽管我们在前面的代码中跳过了这一点，但应该考虑检查对 socket() 和 bind() 函数（以及 POSIX 套接字中的其他函数）的调用，防止出现错误。在出现错误的情况下，它们都返回 -1。

另外，请注意 htons() 函数。它负责将其参数转换为网络字节顺序，这个问题隐藏在计算机的设计方式中。一些机器（例如：英特尔处理器）使用小端字节顺序，而另一些则使用大端顺序。小端顺序将最低有效字节放在前面。大端顺序将最有效字节放在前面。下图显示了两者之间的区别：



网络字节顺序是独立于特定机器架构的一种约定。函数的作用是：将提供的端口号从主机字节顺序（小端或大端）转换为网络字节顺序（独立于机器）。

就这样，套接字就准备好了。现在，我们应该指定它为传入连接做好准备。为此，我们使用 listen() 函数：

```
1 listen(s, 5);
```

顾名思义，它监听传入的连接。传递给 listen() 函数的第二个参数指定服务器在丢弃新的传入请求之前将排队的连接数。在前面的代码中，我们指定 5 为最大值。在高负载环境中，我们将增加这个数字。最大数量由 <sys/socket.h> 头文件中定义的 SOMAXCONN 常量指定。

选择待办数量（listen() 函数的第二个参数）的选择基于以下因素：

- 如果连接请求的速率在短时间内很高，则待办数应该有一个较大的值。
- 服务器处理传入连接的持续时间。时间越短，待办数就越小。

当连接初始化发生时，我们可以放弃它，也可以接受它并继续处理连接。这就是为什么我们在下面的代码片段中使用了 accept() 函数的原因：

```
1 struct sockaddr_in client;
2 int addrlen;
3 int new_socket = accept(s, (struct sockaddr_in*)&client, &addrlen);
4 // use the new_socket
```

在上述代码中需要考虑的两件事如下:

- 首先，接受的套接字连接信息写入客户机的 sockaddr_in 结构体中。我们可以从这个结构中收集关于客户机的一切必要信息。
- 接下来，注意 accept() 函数的返回值。它是一个新的套接字，用于处理来自特定客户端的请求。下一个对 accept() 函数的调用将返回另一个值，该值将代表另一个具有单独连接的客户端。我们应该正确地处理这个问题，因为 accept() 调用正在阻塞，并等待新的连接请求。我们将修改前面的代码，以便它接受在单独的线程中处理的多个连接。

前面代码中带有注释的最后一行声明，可以使用 new_socket 接收或向客户机发送数据。让我们看看如何实现这一点，然后开始设计我们的 Networking 类。要读取套接字接收到的数据，我们需要使用 recv() 函数，如下所示:

```
1 char buffer[BUFFER_MAX_SIZE]; // define BUFFER_MAX_SIZE based on the
2 specifics of the server
3 recv(new_socket, buffer, sizeof(buffer), 0);
4 // now the buffer contains received data
```

recv() 函数接受一个 char* 缓冲区，以便将数据写入其中。在 sizeof(buffer) 时停止写入。函数的最后一个参数是我们可以为读取设置的附加标志，应该考虑多次调用该函数来读取大于 BUFFER_MAX_SIZE 的数据。

最后，为了通过套接字发送数据，调用 send() 函数，如下所示:

```
1 char msg[] = "From server with love";
2 send(new_socket, msg, sizeof(msg), 0);
```

至此，我们已经介绍了实现服务器应用程序所需的几乎所有函数。现在，让我们将它们封装在一个 C++ 类中，并结合多线程，就可以并发地处理客户机的请求了。

实现 POSIX 套接字包装类

让我们设计并实现一个类，它将作为基于网络的应用程序的起点。类的主接口如下所示:

```
1 class Networking
2 {
3 public:
4     void start_server();
5
6 public:
7     std::shared_ptr<Networking> get_instance();
8     void remove_instance();
9
10 private:
11     Networking();
```

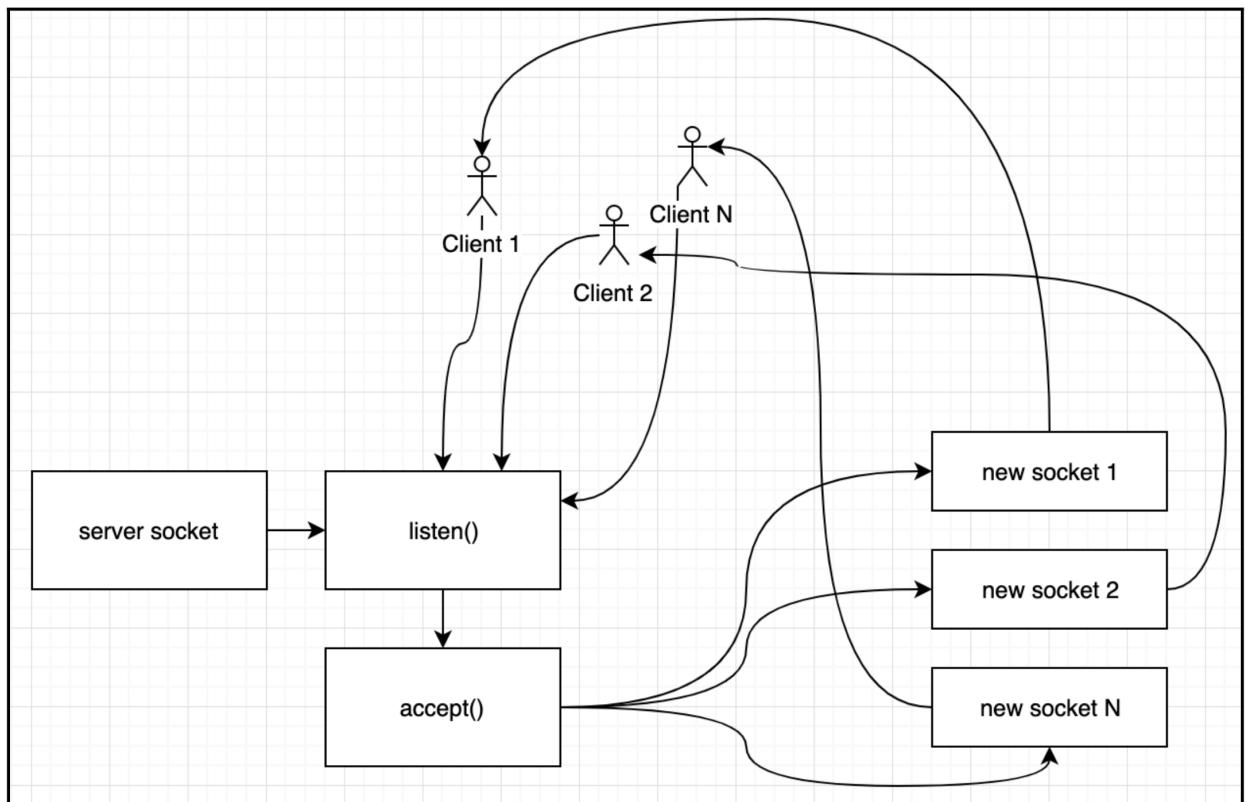
```

12 ~Networking();
13
14 private:
15     int socket__;
16     sockaddr_in server__;
17     std::vector<sockaddr_in> clients__;
18
19 private:
20     static std::shared_ptr<Networking> instance__ = nullptr;
21     static int MAX_QUEUED_CONNECTIONS = 1;
22 };

```

Networking 类很自然地成为单例，因为想要单个实例来侦听传入的连接。拥有多个对象也很重要，每个对象代理与客户机的单独连接。让我们逐步完善类设计。我们看到，每个新的客户端套接字是在服务器套接字侦听，并接受连接请求之后创建的。

之后，我们可以通过新的客户端套接字发送或接收数据。服务器的运行方式与下图类似：



也就是说，在接受每个传入连接之后，我们将为连接提供单独的套接字，并将它们存储在网络类的 clients 中。因此，可以在并发函数中编写创建服务器套接字、侦听和接受新连接的主要逻辑。startserver() 函数作为服务器侦听传入连接的起点。下面的代码块说明了这一点：

```

1 void Networking::start_server()
2 {
3     socket__ = socket(AF_INET, SOCK_STREAM, 0);
4     // the following check is the only one in this code snippet
5     // we skipped checking results of other functions for brevity,

```

```

6 // you shouldn't omit them in your code
7 if (socket_ < 0) {
8     throw std::exception("Cannot create a socket");
9 }
10 struct sockaddr_in server;
11 server.sin_family = AF_INET;
12 server.sin_port = htons(port);
13 server.sin_addr.s_addr = INADDR_ANY;
14
15 bind(s, (struct sockaddr*)&server, sizeof(server));
16 listen(s, MAX queued CONNECTIONS);
17 // the accept() should be here
18 }

```

现在，我们已经在应该接受传入连接的地方停止了（请参阅前面代码片段中的注释）。这里我们有两个选择（实际上不止两个选择，但我们只讨论其中的两个）。我们可以直接在 `start_server()` 函数中调用 `accept()`，也可以实现一个单独的函数，`Networking` 类用户可以随时调用这个函数。



对于项目中的每个错误情况都有特定的异常类，这不是一个坏习惯。在考虑自定义异常时，可能会重写上述代码。你们可以把它作为作业来做。

其中一个选项是 `start_server()` 函数中的 `accept()` 函数，它将每个新连接推入 `clients_`，如下所示：

```

1 void Networking::start_server()
2 {
3     // code omitted for brevity (see in the previous snippet)
4     while (true) {
5         sockaddr_in client;
6         int addrlen;
7         int new_socket = accept(socket_, (sockaddr_in*)&client, &addrlen);
8         clients_.push_back(client);
9     }
10 }

```

我们用了无限循环。这听起来可能很糟糕，但是只要服务器在运行，就必须接受新的连接。然而，我们都应该知道无限循环会阻塞代码的执行，它永远不会离开 `start_server()` 函数。我们介绍的网络应用程序是至少包含三个组件的项目：客户端管理器、存储管理器和我们正在设计的 `Networking` 类。

一个组件的执行不得对其他组件产生不良影响，我们可以使用线程让一些组件在后台运行。运行在线程上下文中的 `start_server()` 函数是一个很好的解决方案，但现在应该关注在第 8 章中讨论的同步问题。

此外，还要注意前面循环的不完整性。在接受连接后，它将客户端数据推入 `clients_`。我们应该考虑使用另一种结构，因为我们还需要存储套接字描述符和客户机。我们可以使用 `std::unordered_map` 将套接字描述符映射到客户端连接信息，使用简单的 `std::pair` 或 `std::tuple` 也可以。

让我们更进一步，创建一个表示客户端连接的自定义对象，如下所示：

```
1 class Client
2 {
3     public:
4         // public accessors
5
6     private:
7         int socket_;
8         sockaddr_in connection_info_;
9 };
```

我们修改 Networking 类，使其存储一个客户端对象的 vector：

```
1 std::vector<Client> clients_;
```

现在，我们可以改变设计方法，让客户端对象负责发送和接收数据：

```
1 class Client
2 {
3     public:
4         void send(const std::string& data) {
5             // wraps the call to POSIX send()
6         }
7
8         std::string receive() {
9             // wraps the call to POSIX recv()
10        }
11
12        // code omitted for brevity
13 };
```

我们可以将一个 std::thread 对象附加到 Client 类，以便每个对象在单独的线程中处理数据传输。但应该注意不要让系统饥饿。传入连接的数量可能会急剧增加，服务器应用程序将会卡住。我们将在下一节讨论安全问题时讨论这个场景。建议利用线程池，既可以帮助我们重用线程，又可以控制程序中运行的线程数量。

类的最终设计取决于我们接收和发送给客户机的数据类型。至少有两种不同的方法。其中一个是连接到客户端，接收必要的数据，然后关闭连接。第二种方法是实现客户机和服务器通信的协议。尽管它听起来很复杂，但协议可能很简单。

它还具有可扩展性，并使应用程序更加健壮，因为随着项目的发展，可以支持更多的特性。下一节中，当我们讨论如何保护网络服务器应用程序时，我们将回到设计用于对客户机请求进行身份验证的协议。

使用 C++ 代码

与许多语言相比，C++ 在安全编码方面有点难掌握。有大量的指导方针提供了关于如何，以及如何避免 C++ 程序中的安全风险的建议。在第 1 章中讨论的一个问题是使用预处理宏。我们使用的例子有以下宏：

```
1 #define DOUBLE_IT(arg) (arg * arg)
```

不恰当地使用宏会导致逻辑错误，而且很难发现。下面的代码中，开发者期望将 16 打印到屏幕上：

```
1 int res = DOUBLE_IT(3 + 1);
2 std::cout << res << std::endl;
```

结果是 7。这里的问题是 arg 参数周围缺少圆括，前面的宏应该重写如下：

```
1 #define DOUBLE_IT(arg) ((arg) * (arg))
```

虽然这个例子很典型，但是我们强烈建议避免使用宏。C++ 提供了大量可以在编译时处理的构造，如 `constexpr`、`consteval` 和 `constinit`——即使语句有 `constexpr` 替代。如果需要在代码中进行编译时处理，可以使用它们。当然，还有模块，这是对语言的补充。你应该在使用 `#include` 的任何地方使用模块，去掉头文件的包含保护：

```
1 module my_module;
2 export int test;
3
4 // instead of
5
6 #ifndef MY_HEADER_H
7 #define MY_HEADER_H
8 int test
9#endif
```

它不仅更安全，而且更高效，因为模块只处理一次（我们可以将它们视为预编译的头文件）。

尽管我们不希望开发者对安全问题变得多疑，但应该在所有地方都小心。通过学习这门语言的怪癖和古怪之处，你可以避免大多数这些问题。另外，一个好的做法是使用最新的特性来替换或修复以前版本的缺点。例如，考虑以下 `create_array()` 函数：

```
1 // Don't return pointers or references to local variables
2 double* create_array()
3 {
4     double arr[10] = {0.0};
5     return arr;
6 }
```

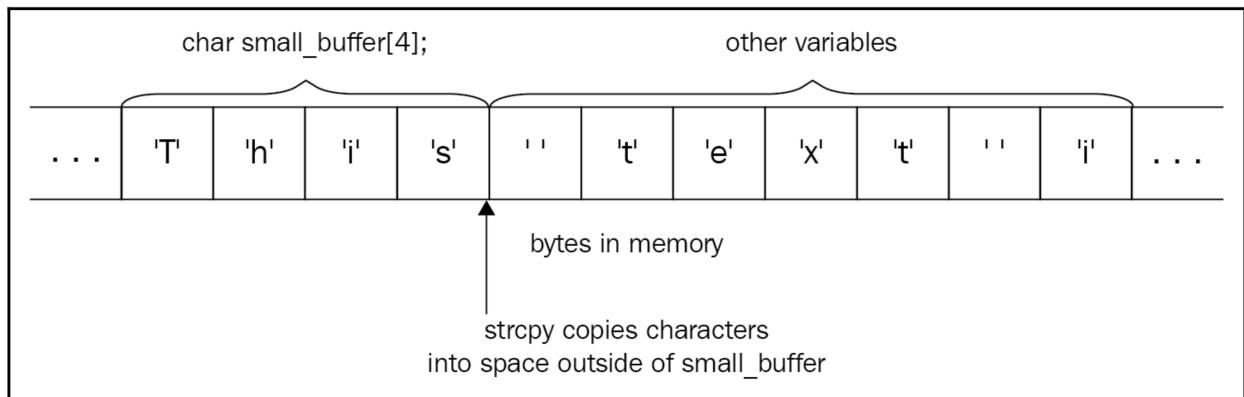
`create_array()` 函数的调用会留下一个指向不存在的数组的指针，因为 `arr` 具有自动存储时间。如果需要，我们可以用一个更好的替代方案替换前面的代码：

```
1 #include <array>
2
3 std::array<double> create_array()
4 {
5     std::array<double> arr;
6     return arr;
7 }
```

字符串可视为字符数组，也是许多缓冲区溢出问题的原因。主要的问题是将数据写入字符串缓冲区时忽略其大小。这方面，`std::string` 类比 C 字符串更安全。但是，当需要支持历史遗留代码时，在使用 `strcpy()` 这样的函数时应该小心，如下所示：

```
1 #include <cstdio>
2 #include <cstring>
3
4 int main()
{
5     char small_buffer[4];
6     const char* long_text = "This text is long enough to overflow small buffers!";
7     strcpy(small_buffer, long_text);
8 }
9 }
```

考虑到这一点，`small_buffer` 的末尾应该有一个空结束符，它将只处理 `long_text` 字符串的前三个字符。然而，在调用 `strcpy()` 之后会发生以下情况：



实现网络应用程序时，应该更加小心。大多数来自客户端连接的数据都应该得到正确的处理，缓冲区溢出的情况并不少见。让我们学习如何使网络应用程序更加安全。

通信网络应用

在本书的前一节中，我们设计了一个网络应用程序，它使用套接字连接接收客户端数据。除了大多数入侵系统的病毒来自外部世界这一事实之外，网络应用程序也有这种自然趋势，使计算机对 Internet 上的各种威胁敞开大门。首先，无论何时运行网络应用程序，系统中都存在一个开放的端口。知道应用程序正在监听的确切端口的人，可以通过伪造协议数据进行入侵。我们主要讨论的网络应用程序是在服务器端。但也有一些主题也适用于客户端应用。

应该做的第一件事是合并客户端授权和身份验证，这两个术语很容易混淆。注意不要交替使用，它们是不同的：

- 身份验证是验证客户端访问的过程。这意味着不是每个传入的连接请求都能立即得到服务。在与客户端之间传输数据之前，服务器应用程序必须确保客户端是已知的。与我们通过输入电子邮件和密码访问社交网络平台的方式相同，客户端的身份验证确定了客户端是否有权访问系统。
- 授权定义了客户端在系统中可以做什么。它是一组提供给特定客户端的权限。例如，我们在上一节中讨论的客户机应用程序能够向系统上传文件。你可能会想要加入付费订阅，并为付

费客户提供更广泛的功能，例如：允许他们创建文件夹来组织文件。因此，当客户端请求创建文件夹时，我们可能希望授权请求，以确定客户端是否有权这样做。

当客户端应用程序发起与服务器的连接时，服务器获得的只是连接详细信息 (IP 地址、端口号)。为了让服务器知道谁是客户机应用程序 (实际用户) 的后台，客户机应用程序将发送用户的凭据。通常，这个过程包括向用户发送唯一标识符 (如用户名或电子邮件地址) 和密码，以访问系统。然后，服务器根据其数据库检查这些凭证，并验证是否应该授予对客户端的访问权。客户端和服务器之间的这种通信形式可能是简单的文本传输或格式化的对象传输。

例如，由服务器定义的协议可能要求客户端以以下形式发送一个 JavaScript 对象符号 (JSON) 文档：

```
1 {
2   "email": "myemail@example.org",
3   "password": "notSoSIMPLEp4s8"
4 }
```

来自服务器的响应允许客户机进一步进行或更新其 UI，以让用户知道操作的结果。在登录时使用 Web 或网络应用程序时，可能会遇到几种情况。例如，输入错误的密码可能会导致服务器返回无效的用户名或密码错误。

除了这第一个必要的步骤之外，验证来自客户机应用程序的每个数据片段是明智的。如果检查电子邮件字段的大小，则可以很容易地避免缓冲区溢出，例如：当客户端应用程序故意破坏系统时，可能会发送一个 JSON 对象，该对象的字段具有非常大的值。这是服务器的责任，防止安全缺陷需要从数据验证开始。

另一种形式的安全攻击是每秒从单个或多个客户端发出太多请求，例如：一个客户端应用程序在一秒钟内发出数百个身份验证请求，这会导致服务器集中处理这些请求，并在试图为所有请求提供服务时浪费资源。最好检查客户机请求的速率，例如：将请求限制为每秒一个。

这些形式的攻击 (有意或无意的) 称为拒绝服务 (DOS) 攻击。DOS 攻击的更高级版本采取了从多个客户机向服务器发出大量请求的形式，这种形式称为分布式 DOS(DDOS) 攻击。一种简单的方法是，通过每秒发出多个请求来将试图使系统崩溃的 IP 地址列入黑名单。作为一名网络应用程序的开发者，在开发应用程序时，应该考虑到这里描述的所有问题以及本书范围之外的许多其他问题。

总结

本章中，我们介绍了用 C++ 设计网络应用程序。C++23 标准可能会支持。

我们首先介绍了网络的基础知识。完全理解网络需要花费大量时间，但在实现与网络相关的应用程序之前，有几个基本概念是每个开发者都必须了解的。这些基本概念包括 OSI 模型中的分层和不同类型的传输协议，如：TCP 和 UDP。对于开发者来说，理解 TCP 和 UDP 之间的差异都是必要的。正如我们所了解的，TCP 在套接字之间建立可靠的连接，而套接字是程序员在开发网络应用程序时遇到的下一个问题。它们是应用程序的两个实例的连接点。当我们需要通过网络发送或接收数据时，我们应该定义一个套接字，并像处理常规文件一样处理它。

我们在应用程序开发中使用的所有抽象和概念都由操作系统处理，最终由网络适配器处理。这是一种能够通过网络媒体发送数据的设备。从媒体接收数据并不能保证安全。网络适配器接收来自

媒体的任何东西。为了确保正确地处理传入的数据，我们还应该注意应用的安全性。本章的最后一节是关于编写安全代码和验证输入，以确保不会对程序造成损害。程序安全是确保高质量程序的一个重要步骤。当然，确保程序无误的最佳方法是彻底测试它们。第 10 章中我们讨论了软件开发步骤，并解释了在编码阶段完成后对程序进行测试的重要性。测试之后，很可能会发现许多 Bug。其中一些错误很难重现或修复，这就是调试的作用所在。

下一章就来介绍一下正确的测试和如何调试程序。

问题

1. 列出 OSI 模型的所有七个层。
2. 端口号有什么意义？
3. 为什么要在网络应用程序中使用套接字？
4. 描述在服务器端执行的操作序列，以便使用 TCP 套接字接收数据。
5. TCP 和 UDP 之间的区别是什么？
6. 为什么不应该在代码中使用宏定义？
7. 实现服务器应用程序时，如何区分不同的客户机应用程序？

扩展阅读

- TCP/IP Illustrated, Volume 1: The Protocols, by R. Stevens: <https://www.amazon.com/TCP-Illustrated-Protocols-Addison-Wesley-Professional/dp/0321336313/> .
- Networking Fundamentals, by Gordon Davies: <https://www.packtpub.com/cloud-networking/networking-fundamentals>

第 13 章：调试与测试

调试和测试在软件开发过程的管道中扮演着极其重要的角色。测试可以帮助我们发现问题，而调试可以修复问题。然而，如果在实现阶段遵循一定的规则，许多潜在的缺陷是可以避免的。此外，由于测试过程非常耗时，如果能够在人工测试之前使用某些工具自动分析软件，那就很好了。此外，我们应该在什么时候、如何以及对什么软件进行测试也很重要。

本章中，我们将了解以下内容：

- 了解问题的根本原因
- 调试 C++ 程序
- 了解静态和动态分析
- 理解 C++ 中的安全问题
- 探索单元测试、TDD 和 BDD

这一章中，我们将学习如何分析一个软件缺陷，如何使用 GNU 调试器 (GDB) 工具来调试程序，以及如何使用工具来自动分析软件。我们还将学习单元测试、测试驱动开发 (TDD) 和行为驱动开发 (BDD) 的概念，以及如何在软件工程开发过程中实践使用它们。

代码位置

可以从这里获取本章的源码文件：<https://github.com/PacktPublishing/Expert-CPP>

了解问题的根本原因

在医学上，一个好的医生需要理解治疗症状和治愈疾病之间的区别。例如，给一个手臂骨折的病人止痛药只会消除症状，手术可能是帮助骨骼逐渐愈合的正确方法。

根本原因分析 (RCA, Root Cause Analysis) 是一个系统的过程，用于识别问题的根本原因。在相关适当工具的帮助下，尝试使用一组特定的步骤来确定问题的主要原因。通过这样做，我们可以确定以下内容：

- 发生了什么？
- 怎么发生的？
- 为什么会这样？
- 应该采取什么措施来防止或减少这种情况的发生，使其不再发生？

RCA 假设某个操作会触发另一个操作，依此类推。通过追溯行动链的起点，我们可以发现问题的根源，以及是如何发展成我们的症状的。这正是我们应该遵循的修复或减少软件缺陷的过程。在接下来的小节中，我们将学习基本的 RCA 步骤，如何应用 RCA 过程来检测软件缺陷，以及 C++ 开发人员应该遵循哪些规则来防止此类缺陷在软件中发生。

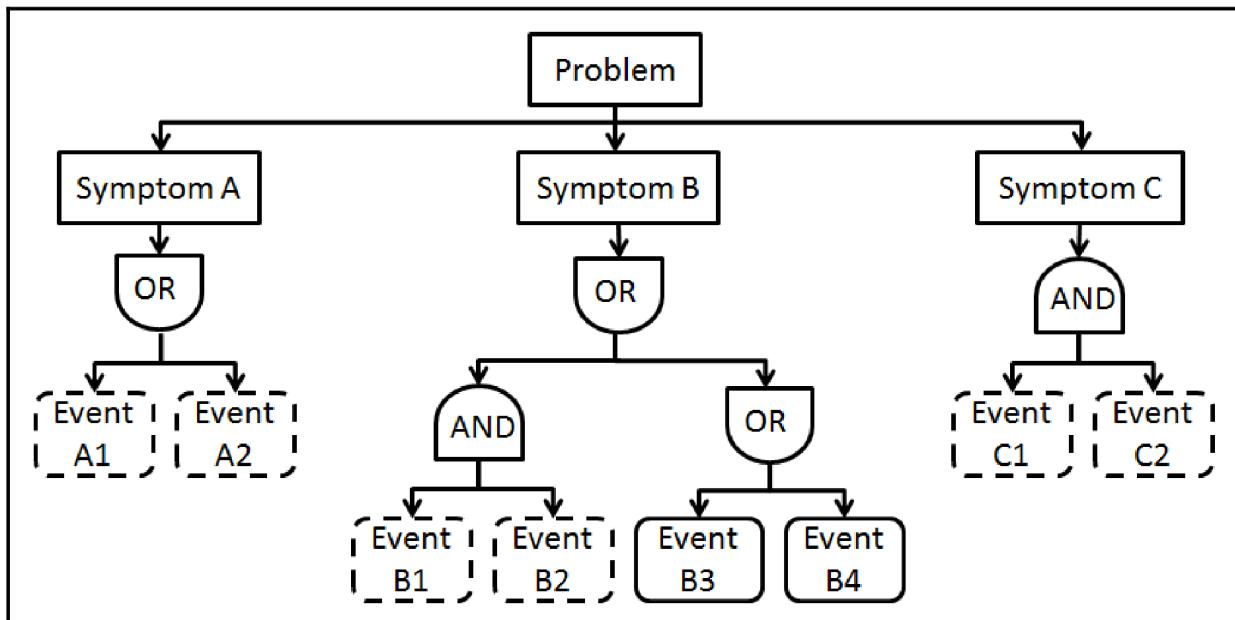
RCA 概述

通常，RCA 进程包含以下五个步骤：

1. 确定问题：在这个阶段，我们可以找到以下问题的答案：发生了什么？问题是什么？问题是在什么环境或条件下发生？
2. 收集数据：为了制作原因图，我们需要收集足够的数据。这个步骤可能要花费大量的时间。

3. 制作因果图: 因素图提供了一个可视化的结构, 我们可以用它来组织和分析收集到的数据。因果图只不过是一个带有逻辑测试的序列图, 它解释了导问题发生的事件。这一制图过程应推动数据收集过程, 直到审查人员满意为止。
4. 查询根本原因: 通过检查因果图, 我们可以制作一个称为根本原因图来识别问题根源的原因。
5. 建议和解决方案: 确定了根本原因, 以下问题的答案可以帮助我们找到解决方案: 可以做什么来防止问题再次发生? 如何实施解决方案? 谁将对此负责? 实现解决方案的成本或风险是什么?

RCA 树图是软件工程行业中最常用的因素图之一。下面是它的一个结构示例:



假设有 A、B、C 三种问题。现象 A 可以由事件 A1 或事件 A2 引起, 现象 B 可以由事件 B1 和事件 B2 或事件 B3 和事件 B4 引起, 现象 C 可以由事件 C1 和事件 C2 引起。数据收集后, 我们发现症状 A 和 C 从未出现, B1 和 B2 没有发生, 那我们可以确定这个问题发生的原因只能是 B3 和 B4。

如果软件存在缺陷, 我们应该对其应用 RCA, 并调查问题的根本原因, 而不是在故障点上修复它。然后, 问题的根本原因可以追溯到需求、设计、实现、验证和/或测试计划和输入数据。当从根本上发现并解决问题时, 可以提高软件的质量, 从而大大降低维护费用。

我们刚刚学会了如何找到问题的根本原因, 但请记住, 最好的防御是进攻。所以, 与其分析和解决问题, 不如我们阻止它的发生?

防患于未然——是一种良好的编码习惯

从成本的角度来看, IBM 的研究表明, 假设需求和设计的总体成本是 1X, 然后和编码实现过程需要 5X, 单元测试和集成测试需要大约 10X, 综合客户测试成本将 15X, 修复缺陷的成本在产品发布会上占据了大约 30X! 因此, 最小化代码缺陷是降低生产成本最有效的方法之一。

尽管找到软件缺陷根本原因的通用方法非常重要, 但如果我们在实现阶段防止一些缺陷就更好了。要做到这一点, 我们需要有良好的编码习惯, 这意味着必须遵守某些规则。这些规则可以分为低级别和高级别。低级规则可能包括以下这些:

- 未初始化变量
- 整数除法
- 错误地使用 = 而不是 ==
- 可能会将有符号变量赋值给无符号变量
- switch 语句中缺少 break
- 复合表达式或函数调用中的错误

当涉及到高级规则时，则为：

- 接口
- 资源管理
- 内存管理
- 并发性

B. Stroustrup 和 H. Sutter 在他们的在线文档，C++ 核心指南 (0.8 版) 中建议遵循这些规则，其中强调了静态类型安全和资源安全。还强调了范围检查的可能性，以避免对空指针、悬空指针进行解引用，以及系统地使用异常。如果开发人员遵循这些规则，那么代码将是静态安全的，不会有任何资源泄漏。此外，它不仅可以捕获更多的编程逻辑错误，而且还可以运行得更快。

由于页面的限制，我们将在本小节中只看几个例子。如果你想看更多的例子，请访问 <https://isocpp.github.io/CppCoreGuidelines>

未初始化变量的问题

未初始化的变量是程序员最常犯的错误之一。当声明一个变量时，将为它分配一定数量的连续内存。如果它没有初始化，它仍然有一些值，但没有确定的方法来预测它。因此，当我们执行程序时，会出现不可预知的行为：

```

1 //ch13_rca_uninit_variable.cpp
2 #include <iostream>
3 int main()
4 {
5     int32_t x;
6     // ... //do something else but not assign value to x
7     if (x>0) {
8         std::cout << "do A, x=" << x << std::endl;
9     }
10    else {
11        std::cout << "do B, x=" << x << std::endl;
12    }
13    return 0;
14 }
```

代码中，当声明 x 时，操作系统将分配 4 个字节的未使用内存给它，这意味着 x 的值是驻留在该内存中的任何值。每次运行这个程序时，x 的地址和值都可能不同。此外，一些编译器，如 Visual Studio，会在 Debug 版本中将 x 的值初始化为 0，但在 Release 版本中保持它不初始化。在这种情况下，我们在 Debug 和 Release 版本中有完全不同的输出。

复合表达式中的副作用

当运算符、表达式、语句或函数完成计算后，它可能会被延长或持续存在于其复合函数中。这种持续的存在有一些副作用，可能会导致一些不确定的行为。让我们看看下面的代码来理解这一点：

```
1 //ch13_rca_compound.cpp
2 #include <iostream>
3 int f(int x, int y)
4 {
5     return x*y;
6 }
7
8 int main()
9 {
10    int x = 3;
11    std::cout << f(++x, x) << std::endl; //bad, f(4,4) or f(4,3)?
12 }
```

由于操作数求值顺序的未定义行为，前面代码的结果可能是 16 或 12。

混合有符号和无符号的问题

通常情况下，二元操作符 +, -, *, /, %, <, <=, >, >=, ==, !=, &&, ||, !, &, |, <<, >>, , ^=, +=, -=, *=, /=, 和 %= 要求两边操作数都是同一类型的。如果这两个操作数的类型不同，其中一个将隐式转换为与另一个相同的类型。[ISO/IEC 9999:2011]6.3.1.1 中给出了三种 C 标准转换规则：

- 当混合相同级别的类型时，有符号类型将被提升为无符号类型。
- 当混合不同级别的类型时，如果低级别的一方的所有值都可以由高级别的一方表示，那么低级别的一方将提升为高级别的类型。
- 如果在前面的情况下，低级别类型的所有值都不能用高级别类型表示，那么将使用高级别类型的无符号版本。

现在，让我们来看看传统的有符号整数减无符号整数的问题：

```
1 //ch13_rca_mix_sign_unsigned.cpp
2 #include <iostream>
3 using namespace std;
4 int main()
5 {
6     int32_t x = 10;
7     uint32_t y = 20;
8     uint32_t z = x - y; //z=(uint32_t)x - y
9     cout << z << endl; //z=4294967286.
10 }
```

上面的例子中，有符号的 int 会自动转换成 uint，结果是 uint32_t z = -10。另一方面，因为 10 不能表示为无符号整数值，它的十六进制值 0xFFFFFFF6 在 2 的补码机器上被解释为 UINT_MAX - 9(即 4294967286)。

顺序问题

下面的示例是关于构造函数中类成员的初始化顺序的。由于初始化顺序是类成员在类定义中出现的顺序，因此最好将每个成员的声明分成不同的行：

```
1 //ch13_rea_order_of_evaluation.cpp
2 #include <iostream>
3 using namespace std;
4
5 class A {
6 public:
7     A(int x) : v2(v1), v1(x) {
8     };
9     void print() {
10         cout << "v1=" << v1 << ",v2=" << v2 << endl;
11    };
12 protected:
13     //bad: the order of the class member is confusing, better
14     //separate it into two lines for non-ambiguity order declare
15     int v1, v2;
16 };
17
18 class B {
19 public:
20     //good: since the initialization order is: v1 -> v2,
21     //after this we have: v1==x, v2==x.
22     B(int x) : v1(x), v2(v1) {};
23
24     //wrong: since the initialization order is: v1 -> v2,
25     //after this we have: v1==uninitialized, v2==x.
26     B(float x) : v2(x), v1(v2) {};
27     void print() {
28         cout << "v1=" << v1 << ", v2=" << v2 << endl;
29    };
30
31 protected:
32     int v1; //good, here the declaration order is clear
33     int v2;
34 };
35 int main()
36 {
37     A a(10);
38     B b1(10), b2(3.0f);
39     a.print(); //v1=10,v2=10,v3=10 for both debug and release
40     b1.print(); //v1=10, v2=10 for both debug and release
41     b2.print(); //v1=-858993460,v2=3 for debug; v1=0,v2=3 for release.
42 }
```

A 类中，虽然声明顺序是 v1->v2，将它们放在一行会使其他开发人员感到困惑。在 B 类的第一个构造函数中，v1 将初始化为 x，然后 v2 将初始化为 v1，因为它的声明顺序是 v1->v2。然而，

在第二个构造函数中，v1 将首先初始化为 v2(此时 v2 还没有初始化!)，然后 v2 将由 x 初始化。这将导致在 Debug 版本和 Release 版本中 v1 的输出值不同。

编译时检查与运行时检查

下面的例子展示了运行时检查(整数类型变量的位数)可以转化到编译时检查：

```
1 //check # of bits for int
2 //courtesy: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines
3 int nBits = 0; // don't: avoidable code
4 for (int i = 1; i; i <= 1){
5     ++nBits;
6 }
7 if (nBits < 32){
8     cerr << "int too small\n";
9 }
```

由于 int 既可以是 16 位也可以是 32 位，这取决于操作系统，因此本例无法实现。我们应该用 int32_t，或者用以下代码替换它：

```
1 static_assert(sizeof(int) >= 4); //compile-time check
```

另一个例子是将最大 n 个整数读入一维数组：

```
1 void read_into(int* p, int n); // a function to read max n integers into *p
2 ...
3 int v[10];
4 read_into(v, 100); //bad, off the end, but the compiler cannot catch this error.
```

这个问题可以通过使用 span<int> 来修复：

```
1 void read_into(span<int> buf); // read into a range of integers
2 ...
3 int v[10];
4 read_into(v); //better, the compiler will figure out the number of elements
```

这里的一般规则是尽可能在编译时进行分析，而不是推迟到运行时。

避免内存泄漏

内存泄漏意味着不释放已分配的动态内存。在 C 中，我们使用 malloc() 和/或 calloc() 来分配内存，使用 free() 来释放内存。在 C++ 中，new 操作符和 delete 或 delete[] 操作符用于动态地管理内存。尽管在智能指针和资源获取初始化(RAII)的帮助下可以降低内存泄漏的风险，但如果我们要构建高质量的代码，仍然需要遵循一些规则。

首先，最简单的内存管理方法是不手动为代码分配内存。例如，当可以写 T x；时，就不要写成 T* x = new T(); 或 shared_ptr<t> x(new T());。

接下来，不要使用自己的代码管理内存，如下所示：

```
1 void f_bad(){
2     T* p = new T();
3     ... //do something with p
```

```
4 delete p ; //leak if throw or return before reaching this line
5 }
```

尝试使用 RAII，如下所示：

```
1 void f_better()
2 {
3     std::auto_ptr<T> p(new T()) ; //other smart pointers is ok also
4     ... //do something with p
5     //will not leak regardless whether this point is reached or not
6 }
```

然后，使用 unique_ptr 替换 shared_ptr，除非需要共享所有权，如下所示：

```
1 void f_bad()
2 {
3     shared_ptr<Base> b = make_shared<Derived>();
4     ...
5 } //b will be destroyed at here
```

因为 b 是在本地使用的，而没有复制它，所以它的 refcount 将始终为 1。这意味着我们可以使用 unique_ptr 来替换它：

```
1 void f_better()
2 {
3     unique_ptr<Base> b = make_unique<Derived>();
4     ... //use b locally
5 } //b will be destroyed at here
```

最后，即使您确实需要自己动态地管理内存，如果有一个标准容器库可用，也不要手动分配内存。

本节中，我们学习了如何使用 RCA 定位问题，以及如何通过编写最佳实践来预防问题。接下来，我们将学习如何使用调试器工具来逐行控制程序的执行，并在运行时检查变量和表达式的值。

调试 C++ 程序

调试是发现和解决程序问题或缺陷的过程。这可能包括交互调试、数据/控制流分析以及单元和集成测试。本节中，我们只关注交互式调试，这是使用断点逐行执行源代码的过程，同时显示所使用的变量的值及其对应的内存地址。

调试 C/C++ 程序的工具

根据开发环境，C++ 社区中有许多可用的工具。下面的列表显示了不同平台上受欢迎的应用程序。

- Linux/Unix:
 - GDB: 免费的开源命令行接口 (CLI) 调试器。
 - Eclipse: 免费的开源集成开发环境 (IDE)。它不仅支持调试，还支持编译、分析和智能编辑。

- Valgrind: 开源动态分析工具，可以用于调试内存泄漏和线程 bug。
 - Affinic: 为 GDB、LLDB 和 LLVM 调试器构建的商业图形用户界面 (GUI) 工具。
 - DDD: 用于 GDB、DBX、JDB、XDB 和 Python 的开放源码数据显示调试器，它以图形的形式显示数据结构。
 - GDB 的 Emacs 模式: 开源 GUI 工具，在使用 GDB 调试时，使用 GNU Emacs 查看和编辑源代码。
 - KDevelop: 免费的开源 IDE 和调试器工具，用于编程语言，如 C/ C++、Objective-C 等。
 - Nemiver: 在 GNOME 桌面环境中工作良好的开源工具。
 - SlickEdit: 调试多线程和多处理器代码的好工具。
- Windows:
 - Visual Studio: 带有 GUI 的商业工具，社区版本是免费的。
 - GDB: 在 Cygwin 或 MinGW 的帮助下，这可以在 Windows 上运行。
 - Eclipse: 它的 C++ 开发工具 (CDT) 可以通过工具链中的 MinGW GCC 编译器安装在 Windows 上。
 - macOS:
 - LLDB: Xcode 在 macOS 上的默认调试器，并支持 C/ C++ 和 Objective-C 在桌面和 iOS 设备及其模拟器。
 - GDB: 命令行调试器，也用于 macOS 和 iOS 系统。
 - Eclipse: 使用 GCC 的免费 IDE 适用于 macOS。

由于 GDB 可以在所有平台上运行，我们将在下面的小节中向您展示如何使用 GDB。

GDB 概述

GDB 代表 GNU 调试器，它允许开发人员在另一个程序执行时查看它内部发生了什么，或者在另一个程序崩溃时查看它在做什么。GDB 可以做以下四件事情：

- 启动一个程序并指定任何可能影响其行为的内容。
- 使程序在给定条件下停止运行。
- 检查当程序停止时发生了什么。
- 运行程序时更改变量的值。这意味着我们可以来纠正 bug 的影响，或者继续查找另一个 bug。

注意，这里涉及两个程序：一个是 GDB，而另一个是要调试的程序。由于这两个程序既可以在同一台机器上运行，也可以在不同的机器上运行，所以我们可以有以下三种类型的调试：

- 本机调试：两个程序在同一台机器上运行。
- 远程调试：GDB 运行在主机上，而经过调试的程序运行在远程机器上。
- 模拟器调试：GDB 运行在主机上，而经过调试的程序运行在模拟器上。

基于编写本书时的最新版本 (GDB v8.3), GDB 支持的语言包括 C、C++、Objective-C、Ada、Assembly、D、Fortran、Go、OpenCL、Modula-2、Pascal 和 Rust。

因为 GDB 是调试行业中最先进的工具，而且很复杂，有很多功能，所以在本节中不可能了解它的所有特性。相反，我们将通过示例来研究最有用的特性。

使用 GDB 的例子

在练习这些例子之前，我们需要运行以下代码来检查 gdb 是否已经安装在我们的系统上：

```
1 ~wus1/chapter-13$ gdb --help
```

如果显示以下类型的信息，我们继续：

```
1 This is the GNU debugger. Usage:  
2 gdb [ options ] [ executable-file [ core-file or process-id ]]  
3 gdb [ options ] --args executable-file [ inferior-arguments ... ]  
4  
5 Selection of debuggee and its files:  
6 --args Arguments after executable-file are passed to inferior  
7 --core=COREFILE Analyze the core dump COREFILE.  
8 --exec=EXECFILE Use EXECFILE as the executable.  
9 ...
```

否则，我们需要安装它。让我们来看看如何在不同的操作系统上安装它：

- Debian-based Linux:

```
1 ~wus1/chapter-13$ sudo apt-get install build-essential  
2
```

- Redhat-based Linux:

```
1 ~wus1/chapter-13$ sudo yum install build-essential  
2
```

- macOS:

```
1 ~wus1/chapter-13$ brew install gdb  
2
```

Windows 用户可以通过 MinGW 分发版安装 GDB。macOS 将需要任务门配置。

然后，再次键入 `gdb -help` 检查是否成功安装。

设置断点和检查变量值

下面的例子中，我们将学习如何设置断点、继续、步进或步进入函数、打印变量值以及如何在 gdb 中使用帮助。源码如下：

```
1 //ch13_gdb_1.cpp  
2 #include <iostream>  
3 float multiple(float x, float y);  
4 int main()
```

```

5 {
6     float x = 10, y = 20;
7     float z = multiple(x, y);
8     printf("x=%f, y=%f, x*y = %f\n", x, y, z);
9     return 0;
10 }
11
12 float multiple(float x, float y)
13 {
14     float ret = x + y; //bug, should be: ret = x * y;
15     return ret;
16 }
```

正如我们在第 3 章中提到的，让我们在调试模式下构建这个程序，如下所示：

```
1 ~wus1/chapter-13$ g++ -g ch13_gdb_1.cpp -o ch13_gdb_1.out
```

注意，对于 `g++`, `-g` 选项意味着调试信息将包含在输出二进制文件中。如果我们运行这个程序，它将显示如下输出：

```
1 x=10.000000, y=20.000000, x*y = 30.000000
```

现在，使用 `gdb` 来查看 bug 在哪里。为此，我们需要执行以下命令行：

```
1 ~wus1/chapter-13$ gdb ch13_gdb_1.out
```

我们将看到以下输出：

```

1 GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
2 Copyright (C) 2018 Free Software Foundation, Inc.
3 License GPLv3+: GNU GPL version 3 or later
4 <http://gnu.org/licenses/gpl.html>
5 This is free software: you are free to change and redistribute it.
6 There is NO WARRANTY, to the extent permitted by law. Type "show copying"
7 and "show warranty" for details.
8 This GDB was configured as "aarch64-linux-gnu".
9 Type "show configuration" for configuration details.
10 For bug reporting instructions, please see:
11 <http://www.gnu.org/software/gdb/bugs/>.
12 Find the GDB manual and other documentation resources online at:
13 <http://www.gnu.org/software/gdb/documentation/>.
14 For help, type "help".
15 Type "apropos word" to search for commands related to "word"...
16 Reading symbols from a.out...done.
17 (gdb)
```

现在，让我们详细看看各种命令：

- `break` 和 `run`: 如果我们输入 `b main` 或 `break main` 并按 Enter，一个断点将被插入到 `main` 函数中。然后，我们可以输入 `run` 或 `r` 来开始调试程序。以下信息将显示在终端窗口中。在这里，我们可以看到我们的第一个断点在源代码的第 6 行，并且为了等待新命令，调试过的程序已暂停：

```

1 (gdb) b main
2 Breakpoint 1 at 0x8ac: file ch13_gdb_1.cpp, line 6.
3 (gdb) r
4 Starting program: /home/nvidia/wus1/Chapter-13/a.out
5 [Thread debugging using libthread_db enabled]
6 Using host libthread_db library "/lib/aarch64-linux-
7 gnu/libthread_db.so.1".
8
9 Breakpoint 1, main () at ch13_gdb_1.cpp:6
10 6 float x = 10, y = 20;
11

```

- next, print 和 quit:n 或 next 命令将转到代码的下一行。如果该行调用了一个子程序，它不会进入子程序，而会跳过调用并将其视为单个行。如果要显示变量的值，可以使用 p 或 print 命令，后面跟着变量名。最后，如果我们想要退出 gdb，可以使用 q 或 quit 命令。以下是运行这些操作后，终端窗口的输出：

```

1 (gdb) n
2 7 float z = multiple(x, y);
3 (gdb) p z
4 $1 = 0
5 (gdb) n
6 8 printf("x=%f, y=%f, x*y = %f\n", x, y, z);
7 (gdb) p z
8 $2 = 30
9 (gdb) q
10 A debugging session is active.
11 Inferior 1 [process 29187] will be killed.
12 Quit anyway? (y or n) y
13 ~/wus1/Chapter-13$
14

```

- step：现在，让我们学习如何进入 multiple() 函数并发现 bug。为此，我们需要使用 b、r 和 n 命令重新开始，首先到达第 7 行。然后，我们可以使用 s 或 step 命令进入 multiple() 函数。接下来，我们使用 n 命令到达第 14 行和 p 来打印 ret 变量的值，它是 30。**ahha the bug is at line 14!**：有一个拼写错误，即 x+y，而不是 x*y。下面的代码块是这些命令对应的输出：

```

1 ~/wus1/Chapter-13$gdb ch13_gdb_1.out
2 ...
3 (gdb) b main
4 Breakpoint 1 at 0x8ac: file ch13_gdb_1.cpp, line 6.
5 (gdb) r
6 The program being debugged has been started already.
7 Start it from the beginning? (y or n) y
8 Starting program: /home/nvidia/wus1/Chapter-13/a.out
9 [Thread debugging using libthread_db enabled]
10 Using host libthread_db library "/lib/aarch64-linux-
11 gnu/libthread_db.so.1".

```

```
12 Breakpoint 1, main () at ch13_gdb_1.cpp:6
13   6 float x = 10, y = 20;
14 (gdb) n
15   7 float z = multiple(x, y);
16 (gdb) s
17 multiple (x=10, y=20) at ch13_gdb_1.cpp:14
18 14 float s = x + y;
19 (gdb) n
20 15 return s;
21 (gdb) p s
22 $1 = 30
23
```

- help : 最后，让我们学习帮助命令来结束这个小示例。当启动 gdb 时，我们可以使用 help 或 h 命令在其命令输入行中获取特定命令的使用信息。例如，下面的终端窗口总结了我们到目前为止所学的内容：

```
1 (gdb) h b
2 Set breakpoint at specified location.
3 break [PROBE_MODIFIER] [LOCATION] [thread THREADNUM] [ if
4 CONDITION]
5 PROBE_MODIFIER shall be present if the command is to be placed in
6 a
7 probe point. Accepted values are ‘-probe’ (for a generic,
8 automatically
9 guessed probe type), ‘-probe-stap’ (for a SystemTap probe) or
10 ‘-probe-dtrace’ (for a DTrace probe).
11 LOCATION may be a linespec, address, or explicit location as
12 described
13 below.
14 ....
15 (gdb) h r
16 Start debugged program.
17 You may specify arguments to give it.
18 Args may include "*", or "[...]" ; they are expanded using the
19 shell that will start the program (specified by the "$SHELL"
20 environment
21 variable). Input and output redirection with ">", "<", or ">>"
22 are also allowed.
23 (gdb) h s
24 Step program until it reaches a different source line.
25 Usage: step [N]
26 Argument N means step N times (or till program stops for another
27 reason).
28 (gdb) h n
29 Step program, proceeding through subroutine calls.
30 Usage: next [N]
31 Unlike "step", if the current source line calls a subroutine,
32 this command does not enter the subroutine, but instead steps over
```

```
1      the call , in effect treating it as a single source line.
2      (gdb) h p
3      Print value of expression EXP.
4      Variables accessible are those of the lexical environment of the
5      selected
6      stack frame, plus all those whose scope is global or an entire
7      file .
8      (gdb) h h
9      Print list of commands.
10     (gdb) h help
11     Print list of commands.
12     (gdb) help h
13     Print list of commands.
14     (gdb) help help
15     Print list of commands.
16
17
```

现在, 我们已经了解了一些可以用来调试程序的基本命令。这些命令是 break、run、next、print、quit、step 和 help。我们将在下一小节中学习函数和条件断点、观察点以及 continue 和 finish 命令。

函数断点、条件断点、watchpoint 以及 continue 和 finish 命令

本例中, 我们将学习如何设置函数断点、条件断点和使用 continue 命令。然后, 我们将学习如何在不需要一步一步执行所有代码行的情况下完成一个函数调用。源代码如下:

```
1 //ch13_gdb_2.cpp
2 #include <iostream>
3
4 float dotproduct( const float *x, const float *y, const int n);
5 int main()
6 {
7     float sxx,sxy ;
8     float x [] = {1,2,3,4,5};
9     float y [] = {0,1,1,1,1};
10
11    sxx = dotproduct( x, x, 5);
12    sxy = dotproduct( x, y, 5);
13    printf( "dot(x,x) = %f\n", sxx );
14    printf( "dot(x,y) = %f\n", sxy );
15    return 0;
16 }
17
18 float dotproduct( const float *x, const float *y, const int n )
19 {
20     const float *p = x;
21     const float *q = x; //bug: replace x by y
22     float s = 0;
23     for( int i=0; i<n; ++i, ++p, ++q){
```

```
24     s += (*p) * (*q);  
25 }  
26 return s;  
27 }
```

同样，在构建并运行 ch13_gdb_2.cpp 之后，我们得到如下输出：

```
1 ~/wus1/Chapter-13$ g++ -g ch13_gdb_2.cpp -o ch13_gdb_2.out  
2 ~/wus1/Chapter-13$ ./ch13_gdb_2.out  
3 dot(x,x) = 55.000000  
4 dot(x,y) = 55.000000
```

既然点乘 (x,x) 和点乘 (x,y) 得到的结果是一样的，这里肯定有问题。现在，让我们通过学习如何在 dot() 函数中设置断点来调试它：

- 函数断点：要在函数的开头设置断点，可以使用 b function_name 命令。我们可以在输入过程中使用制表符联想补全。例如，我们输入以下内容：

```
1 (gdb) b dot<Press TAB Key>  
2
```

如果我们这样做，下面的命令行将自动弹出：

```
1 (gdb) b dotproduct(float const*, float const*, int)  
2
```

如果是类的成员函数，则应该包含类名，如下所示：

```
1 (gdb) b MyClass::foo(<Press TAB key>  
2
```

- 条件断点：有几种设置条件断点的方法：

```
1 (gdb) b f.cpp:26 if s==0 //set a breakpoint in f.cpp, line 26 if  
2 s==0  
3 (gdb) b f.cpp:20 if ((int)strcmp(y, "hello")) == 0  
4
```

- 列出和删除断点：我们设置了一些断点，就可以列出或删除它们，如下所示：

```
1 (gdb) i b  
2 (gdb) delete breakpoints 1  
3 (gdb) delete breakpoints 2-5  
4
```

- 使断点无条件：由于每个断点都有一个数字，我们可以从断点中删除一个条件，如下所示：

```
1 (gdb) cond 1 //break point 1 is unconditional now  
2
```

- Watchpoint：当表达式的值发生变化时，Watchpoint 可以停止执行，而不必预测它可能发生在哪一行。观察点有三种：

- watch：当发生写操作时，gdb 将中断

- rwatch : 当读取发生时, gdb 将中断
- awatch : 当发生写操作或读操作时, gdb 将会中断

下面的代码显示了一个例子:

```

1 (gdb) watch v //watch the value of variable v
2 (gdb) watch *(int*)0x12345678 //watch an int value pointed by an
3 address
4 (gdb) watch a*b + c/d // watch an arbitrarily complex
5 expression
6

```

- continue: 当我们完成了对断点处变量值的检查后, 我们可以使用 continue 或 c 命令继续程序执行, 直到调试器遇到断点、信号、错误或正常的进程终止。
- finish: 一旦进入一个函数内部, 我们可能想要连续地执行它, 直到它返回到调用者行。这可以使用 finish 命令完成。

现在, 让我们把这些命令放在一起调试 ch13_gdb_2.cpp。我们把它分为三个部分:

```

1 //gdb output of example ch13_gdb_2.out -- part 1
2 ~/wus1/Chapter-13$ gdb ch13_gdb_2.out //cmd 1
3 ...
4 Reading symbols from ch13_gdb_2.out ... done.
5
6 (gdb) b dotproduct(float const*, float const*, int) //cmd 2
7 Breakpoint 1 at 0xa5c: file ch13_gdb_2.cpp, line 20.
8 (gdb) b ch13_gdb_2.cpp:24 if i==1 //cmd 3
9 Breakpoint 2 at 0xa84: file ch13_gdb_2.cpp, line 24.
10 (gdb) i b //cmd 4
11 Num Type Disp Enb Address What
12 1 breakpoint keep y 0x000000000000a5c in dotproduct(float const*, float
13 const*, int) at ch13_gdb_2.cpp:20
14 2 breakpoint keep y 0x000000000000a84 in dotproduct(float const*, float
15 const*, int) at ch13_gdb_2.cpp:24
16 stop only if i==1
17 (gdb) cond 2 //cmd 5
18 Breakpoint 2 now unconditional.
19 (gdb) i b //cmd 6
20 Num Type Disp Enb Address What
21 1 breakpoint keep y 0x000000000000a5c in dotproduct(float const*, float
22 const*, int) at ch13_gdb_2.cpp:20
23 2 breakpoint keep y 0x000000000000a84 in dotproduct(float const*, float
24 const*, int) at ch13_gdb_2.cpp:24

```

在第一部分中, 我们有以下六个命令:

- cmd 1 : 我们以构建的可执行文件 ch13_gdb_2, 并结合参数使用 gdb。简要地向我们展示了它的版本、文档和使用信息, 然后告诉我们读取符号的过程已经完成, 正在等待下一个命令。
- cmd 2 : 我们设置了一个断点函数 (dotproduct())。
- cmd 3 : 设置了条件断点。

- cmd 4：它列出了关于断点的信息，并告诉我们有两个断点。
- cmd 5：我们将断点 2 设置为无条件的
- cmd 6：我们再次列出断点信息。在这一点上，我们可以看到两个断点。它们位于 ch13_gdb_2.cpp 中的第 20 行和第 24 行。

接下来，让我们看看第二部分中的 gdb 输出：

```

1 //gdb output of example ch13_gdb_2.out -- part 2
2 (gdb) r //cmd 7
3 Starting program: /home/nvidia/wus1/Chapter-13/ch13_gdb_2.out
4 [Thread debugging using libthread_db enabled]
5 Using host libthread_db library '/lib/aarch64-linux-gnu/libthread_db.so.1'.
6
7 Breakpoint 1, dotproduct (x=0x7fffffed68, y=0x7fffffed68, n=5) at
8 ch13_gdb_2.cpp:20
9 20 const float *p = x;
10 (gdb) p x //cmd 8
11 $1 = (const float *) 0x7fffffed68
12 (gdb) c //cmd 9
13 Continuing.
14
15 Breakpoint 2, dotproduct (x=0x7fffffed68, y=0x7fffffed68, n=5) at
16 ch13_gdb_2.cpp:24
17 24 s += (*p) * (*q);
18 (gdb) p i //cmd 10
19 $2 = 0
20 (gdb) n //cmd 11
21 23 for (int i=0; i<n; ++i, ++p, ++q){
22 (gdb) n //cmd 12
23
24 Breakpoint 2, dotproduct (x=0x7fffffed68, y=0x7fffffed68, n=5) at
25 ch13_gdb_2.cpp:24
26 24 s += (*p) * (*q);
27 (gdb) p s //cmd 13
28 $4 = 1
29 (gdb) watch s //cmd 14
30 Hardware watchpoint 3: s

```

第二部分有以下 cmd:

- cmd 7：通过给出 run 命令，程序开始运行，并在第 20 行中的第一个断点处停止。
- cmd 8：我们打印 x 的值，它显示了它的地址。
- cmd 9：继续运行，将在第 24 行中的第二个断点处停止。
- cmd 10：输出 i 的值，即 0
- cmd 11-12：我们使用下一个命令两次。此时，执行 $s += (*p) * (*q)$ 语句。
- cmd 13：输出 s 的值，即 1。
- cmd 14：打印 s 的值。

最后，第三部分内容如下：

```

1 //gdb output of example ch13_gdb_2.out -- part 3
2 (gdb) n //cmd 15
3 Hardware watchpoint 3: s
4
5 Old value = 1
6 New value = 5
7 dotproduct (x=0x7fffffed68 , y=0x7fffffed68 , n=5) at ch13_gdb_2.cpp:23
8 23 for (int i=0; i<n; ++i, ++p, ++q){
9 (gdb) finish //cmd 16
10 Run till exit from #0 dotproduct (x=0x7fffffed68 , y=0x7fffffed68 , n=5) at ch13_gdb_2
   .cpp:23
11
12 Breakpoint 2, dotproduct (x=0x7fffffed68 , y=0x7fffffed68 , n=5) at
ch13_gdb_2.cpp:24
13 24 s += (*p) * (*q);
14 (gdb) delete breakpoints 1-3 //cmd 17
15 (gdb) c //cmd 18
16 (gdb) c //cmd 18
17 Continuing.
18
19 dot(x,x) = 55.000000
20 dot(x,y) = 55.000000
21 [Inferior 1 (process 31901) exited normally]
22 [Inferior 1 (process 31901) exited normally]
23 (gdb) q //cmd 19
24 ~/wus1/Chapter-13$
```

在这部分中，我们有以下命令：

- cmd 15：如果执行下一个迭代，我们将使用下一个命令来查看 s 的值。它表明 s 的旧值为 1 ($s=1*1$)，新值为 5 ($s=1*1+2*2$)。到目前为止，一切顺利！
- cmd 16：finish 命令用于继续运行程序，直到它退出该函数。
- cmd 17：我们删除断点 1 到 3。
- cmd 18：使用 continue 命令。
- cmd 19：我们退出 gdb，回到终端窗口。

将 GDB 记录输出到文本文件中

在处理长堆栈跟踪或多线程堆栈跟踪时，从终端窗口查看和分析 gdb 输出可能会很不方便。但是，我们可以先将整个会话或特定输出记录到一个文本文件中，然后再使用其他文本编辑器工具脱机浏览它。为此，我们需要使用以下命令：

```
1 (gdb) set logging on
```

当我们执行这个命令时，gdb 将把所有终端窗口的输出保存到一个名为 gdb.txt 的文本文件中，该文件位于当前运行的 gdb 文件夹中。如果我们想停止日志记录，我们可以输入以下内容：

```
1 (gdb) set logging off
```

GDB 的一个优点是，我们可以随心所欲地打开或关闭 set logging 命令，而不必担心转储文件名。这是因为所有输出都连接到 gdb.txt 文件中。

下面是要返回 ch13_gdb_2.out 的 gdb 输出：

```
1 ~/wus1/Chapter-13$ gdb ch13_gdb_2.out //cmd 1
2 ...
3 Reading symbols from ch13_gdb_2.out... done.
4 (gdb) set logging on //cmd 2
5 Copying output to gdb.txt.
6 (gdb) b ch13_gdb_2.cpp:24 if i==1 //cmd 3
7 Breakpoint 1 at 0xa84: file ch13_gdb_2.cpp, line 24.
8 (gdb) r //cmd 4
9 ...
10 Breakpoint 1, dotproduct (x=0x7fffffed68, y=0x7fffffed68, n=5) at ch13_gdb_2.cpp:24
11 24 s += (*p) * (*q);
12 (gdb) p i //cmd 5
13 $1 = 1
14 (gdb) p s //cmd 6
15 $2 = 1
16 (gdb) finish //cmd 7
17 Run till exit from #0 dotproduct (x=0x7fffffed68, y=0x7fffffed68, n=5) at ch13_gdb_2
     .cpp:24
18 0x00000055555559e0 in main () at ch13_gdb_2.cpp:11
19 11 sxx = dotproduct( x, x, 5);
20 Value returned is $3 = 55
21 (gdb) delete breakpoints 1 //cmd 8
22 (gdb) set logging off //cmd 9
23 Done logging to gdb.txt.
24 (gdb) c //cmd 10
25 Continuing.
26 dot(x,x) = 55.000000
27 dot(x,y) = 55.000000
28 [Inferior 1 (process 386) exited normally]
29 (gdb) q //cmd 11
30 ~/wus1/Chapter-13$ cat gdb.txt //cmd 12
```

上述代码中使用的命令如下：

- cmd 1：启动 gdb。
- cmd 2：我们将日志标志设置为 on。此时，gdb 说输出将被复制到 gdb.txt 文件中。
- cmd 3：设置了条件断点。
- cmd 4：我们运行程序，当程序在第 24 行到达条件断点时停止。
- cmd 5 和 cmd 6：打印 i 和 s 的值。
- cmd 7：通过执行函数的步出命令，它显示 sxx 是 55(在调用 sxx=dotproduct(x, x, 5))，程
序在 sxy =dotproduct(x, y, 5) 行停止。
- cmd 8：删除断点 1。
- cmd 9：将日志标志设置为 off。

- cmd 10：当给出了 continue 指令，它就会执行 main 函数，gdb 会等待一个新命令。
- cmd 11：输入 q 退出 gdb
- cmd 12：当它返回到终端时，我们在操作系统中运行 cat 命令打印记录的 gdb.txt 文件的内容。

到目前为止，我们已经学习了足够的调试程序的 GDB 命令。你可能已经注意到的，这样做很耗时。有时，由于在错误的地方进行调试，情况会变得更糟。为了有效地调试，我们需要遵循正确的策略。我们将在下面的小节中讨论这个问题。

实际调试策略

由于调试是软件开发生命周期中成本最高的阶段，查找 bug 并修复它们是不可行的，特别是对于大型、复杂的系统。但是，也有一些策略可以在实际的过程中使用，其中一些策略如下：

- 使用 printf() 或 std::cout：这是很古老的方式。通过将一些信息打印到终端，我们可以检查变量的值，并执行 where 和 when 类型的日志配置文件，以便进一步分析。
- 使用调试器：虽然学习使用 GDB 类调试器工具不是一夜之间的事情，但它可以节省大量时间。所以，试着慢慢地熟悉它。
- 重现错误：每当在字段中报告错误时，记录运行环境和输入数据。
- 转储日志文件：应用程序应该将日志消息转储到一个文本文件中。当崩溃发生时，我们应该首先检查日志文件，看看是否发生了异常事件。
- 猜测：粗略猜测一个 bug 的位置，然后证明它是对的还是错的。
- 分而治之：即使在最坏的情况下，我们不知道有什么 bug，我们仍然可以使用二分搜索策略来设置断点，然后缩小范围，最终找到它们。
- 简化：总是从最简化的场景开始，然后逐渐添加外围设备、输入模块等，直到可以重现错误。
- 源代码版本控制：如果一个 bug 在发布版中突然出现，但它之前运行得很好，那么首先做一个源代码树检查。可能有人做了改变！
- 不要放弃：有些 bug 真的很难定位和/或修复，特别是对于复杂的和多团队参与的系统。暂时把它们放在一边，在回家的路上重新考虑一下——这个恍然大悟的时刻可能最终会显现出来。

到目前为止，我们已经学习了使用 RCA 进行宏观层面的问题定位，以及可以遵循的防止问题发生的好编码实践。此外，通过使用最先进的调试器工具（如 GDB），我们可以逐行控制程序的执行，这样我们就可以在微观层面分析和修复问题。所有这些活动都是程序员集中的和手动的。是否有任何自动化工具可以帮助我们诊断程序的潜在缺陷？我们将在下一节中研究静态和动态分析。

了解静态和动态分析

我们了解了根本原因分析过程以及如何使用 GDB 调试缺陷。本节将讨论如何分析有执行和没有执行的程序。前者称为动态分析，后者称为静态分析。

静态分析

静态分析在不执行计算机程序的情况下评估计算机程序的质量。虽然这通常可以通过自动工具和代码审查/检查来检查源代码来完成，但我们在本节中只关注自动工具。

自动静态代码分析工具被设计用来根据一组或多组编码规则或准则分析一组代码。通常，人们可以互换地使用静态代码分析、静态分析或源代码分析。通过扫描所有可能的代码执行路径的整个代码库，我们可以在测试阶段之前发现大量潜在的 bug。但是，它也有几个局限性，如下：

- 会产生假阳性和假阴性报警。
- 只应用扫描算法内部实现的规则，其中一些规则可能会被主观解释。
- 无法找到在运行时环境中引入的漏洞。
- 会给人一种所有问题都得到解决的虚假安全感。

在商业和免费的开源类别下，大约有 30 个自动的 C/C++ 代码分析工具。这些工具的名称包括 Clang、Clion、CppCheck、Eclipse、Visual Studio 和 GNU g++，这只是其中的一些。作为例子，我们将介绍-Wall，-Weffc++ 和-Wextra 选项，它们内置在 GNU 编译器 g++ 中：

- -Wall：这允许所有的编译警告，这对一些用户来说是有问题的。这些警告很容易避免或修改，即使与宏结合在一起也是如此。它还支持在 C++ 方言选项和 Objective-C/C++ 方言选项中描述的一些特定于语言的警告。
- -Wextra：顾名思义，它检查-Wall 没有检查的某些额外警告标志。以下任何情况的警告信息将被打印：
 - 用 <、<=、> 或 >= 操作数将指针与整数 0 进行比较
 - 一个非枚举数和一个枚举数出现在条件表达式中。
 - 模棱两可的虚基数。
 - 下标寄存器类型数组。
 - 使用寄存器类型变量的地址。
 - 派生类的复制构造函数不初始化其基类。
- -Weffc++：它检查是否违反了 Scott Meyers 撰写的《Effective and more Effective c++》中建议的一些准则。这些准则包括以下内容：
 - 为具有动态分配内存的类定义复制构造函数和赋值操作符。
 - 在构造函数中初始化而不是赋值。
 - 在基类中使析构函数为虚函数。
 - = 操作符是否返回了对 *this 的引用
 - 当须返回一个对象时，不要返回引用。
 - 区分自增和自减操作符的前缀和后缀形式。
 - 不要重载 &&，||，或，

为了探索这三个选项，让我们看看下面的例子：

```
1 //ch13_static_analysis.cpp
2 #include <iostream>
3 int *getPointer(void)
4 {
5     return 0;
```

```

6 }
7
8 int &getVal() {
9     int x = 5;
10    return x;
11 }
12
13 int main()
14 {
15     int *x = getPointer();
16     if( x> 0 ){
17         *x = 5;
18     }
19     else{
20         std::cout << "x is null" << std::endl;
21     }
22     int &y = getVal();
23     std::cout << y << std::endl;
24     return 0;
25 }
```

首先，让我们在没有任何选择的情况下构建：

```
1 g++ -o ch13_static.out ch13_static_analysis.cpp
```

这可以成功构建，但如果我们像预期的那样运行将崩溃，并带有一个段错误（核心转储）消息。

接下来，让我们添加-Wall，-Weffc++ 和-Wextra 选项，并重新构建它：

```
1 g++ -Wall -o ch13_static.out ch13_static_analysis.cpp
2 g++ -Weffc++ -o ch13_static.out ch13_static_analysis.cpp
3 g++ -Wextra -o ch13_static.out ch13_static_analysis.cpp
```

-Wall 和-Weffc++ 都给了我们以下信息：

```
1 ch13_static_analysis.cpp: In function ‘int& getVal()’ :
2 ch13_static_analysis.cpp:9:6: warning: reference to local variable ‘x’
3 returned [-Wreturn-local-addr]
4 int x = 5;
5 ^
```

这里，编译器抱怨说，在 `int & getVal()` 函数 (cpp 文件的第 9 行) 中，返回了对局部变量的引用。这将不起作用，因为一旦程序离开函数，`x` 就是死亡了 (`x` 的生命周期仅在函数的作用域内受限)。引用死亡变量没有任何意义。

-Wextra 给了我们以下信息：

```
1 ch13_static_analysis.cpp: In function ‘int& getVal()’ :
2 ch13_static_analysis.cpp:9:6: warning: reference to local variable ‘x’
3 returned [-Wreturn-local-addr]
4 int x = 5;
5 ^
6 ch13_static_analysis.cpp: In function ‘int main()’ :
```

```
7 ch13_static_analysis.cpp:16:10: warning: ordered comparison of pointer
8 with integer zero [-Wextra]
9 if( x> 0 ){
10 ^
```

前面的输出显示-Wextra 不仅向我们提供了来自-Wall 的警告，而且还检查了我们前面提到的 6 个东西。在这个例子中，它警告我们在代码的第 16 行中有一个指针和整数 0 之间的比较。

现在我们知道了如何在编译时使用静态分析选项，我们将通过执行程序来了解动态分析。

动态分析

动态分析是动态程序分析的缩写，它通过在真实或虚拟处理器上执行软件程序来分析软件程序的性能。与静态分析类似，动态分析也可以自动或手动完成。例如，单元测试、集成测试、系统测试和验收测试通常都是人工参与的动态分析过程。另一方面，内存调试、内存泄漏检测和概要分析工具（如 IBM purify、Valgrind 和 Clang sanitizers）都是动态分析工具。在本小节中，我们将重点关注动态分析工具。

动态分析过程包括准备输入数据、启动测试程序、收集必要的参数和分析输出等步骤。粗略地说，动态分析工具的机制是它们使用代码工具和/或模拟环境来在被分析的代码执行时执行检查。我们可以通过以下方式与程序交互：

- 源代码插植：在编译之前，在原始源代码中插入一个特殊的代码段。
- 目标代码插植：一个特定的二进制代码被直接添加到可执行文件中。
- 编译阶段检测：通过特殊的编译器开关添加检查代码。
- 不会改变源代码，使用特殊的执行阶段库来检测错误。

动态分析有以下优点：

- 不存在假阳性或假阴性结果，因为将检测到模型没有预测到的错误。
- 不需要源代码，这意味着私有代码可以由第三方组织进行测试。

动态分析的缺点如下：

- 只检测与输入数据相关的路由缺陷。其他缺陷可能不会被发现。
- 一次只能检查一个执行路径。为了获得一个完整的报告，我们需要尽可能多地运行测试。这需要大量的计算资源。
- 无法检查代码的正确性。从错误的操作中得到正确的结果是可能的。
- 在处理器上执行不正确的代码可能会产生意想不到的结果。

现在，让我们使用 Valgrind 来查找下面例子中给出的内存泄漏和越界问题：

```
1 //ch13_dynamic_analysis.cpp
2 #include <iostream>
3 int main()
4 {
5     int n=10;
6     float *p = (float *)malloc(n * sizeof(float));
7     for( int i=0; i<n; ++i){
8         std::cout << p[i] << std::endl;
```

```
9     }
10    //free(p); //leak: free() is not called
11    return 0;
12 }
```

使用 Valgrind 进行动态分析，需要执行以下步骤：

- 首先，我们需要安装 valgrind。我们可以使用下面的命令来做到这一点：

```
1 sudo apt install valgrind //for Ubuntu, Debian, etc.
```

- 当成功安装，我们可以通过传递可执行文件作为参数来运行 valgrind，以及其他参数，如下所示：

```
1 valgrind --leak-check=full --show-leak-kinds=all --track-
2 origins=yes \
3 --verbose --log-file=valgrind-out.txt ./myExeFile myArgumentList
4
```

- 接下来，让我们构建这个程序，如下所示：

```
1 g++ -o ch13_dyn -std=c++11 -Wall ch13_dynamic_analysis.cpp
2
```

- 然后，我们运行 valgrind，就像这样：

```
1 valgrind --leak-check=full --show-leak-kinds=all --track-
2 origins=yes \
3 --verbose --log-file=log.txt ./ch13_dyn
4
```

最后，我们可以检查 log.txt 的内容。粗体和斜体表示内存泄漏的位置和大小。通过检查地址 (0x4844BFC) 及其对应的函数名 (main())，我们可以看到这个 malloc 在 main() 函数中：

```
1 ... //ignore many lines at begining
2 by 0x108A47: main (in /home/nvidia/wus1/Chapter-13/ch13_dyn)
3 ==18930== Uninitialised value was created by a heap allocation
4 ==18930== at 0x4844BFC: malloc (in /usr/lib/valgrind/vgpreload_memcheck-
5 arm64-linux.so)
6 ... //ignore many lines in middle
7 ==18930== HEAP SUMMARY:
8 ==18930== in use at exit: 40 bytes in 1 blocks
9 ==18930== total heap usage: 3 allocs, 2 frees, 73,768 bytes allocated
10 ==18930==
11 ==18930== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
12 ==18930== at 0x4844BFC: malloc (in /usr/lib/valgrind/vgpreload_memcheck-
13 arm64-linux.so)
14 ==18930==
15 ==18930== LEAK SUMMARY:
16 ==18930== definitely lost: 40 bytes in 1 blocks
17 ==18930== indirectly lost: 0 bytes in 0 blocks
```

```
18 ==18930== possibly lost: 0 bytes in 0 blocks
19 ==18930== still reachable: 0 bytes in 0 blocks
20 ==18930== suppressed: 0 bytes in 0 blocks
```

在这里，我们可以看到调用 `malloc()` 来在地址 `0x4844BFC` 分配一些内存。堆摘要部分指出我们在 `0x4844BFC` 处有 40 个字节的内存丢失。最后，泄漏摘要部分显示，肯定有一个 40 字节的内存丢失块。通过搜索 `log.txt` 文件中 `0x4844BFC` 的地址值，我们最终发现在原始代码中没有调用 `free(p)` 行。在取消注释这一行之后，我们重新进行 `valgrind` 分析，这样泄漏问题就不会出现在报告中了。

总之，在静态和动态分析工具的帮助下，程序的潜在缺陷可以自动大大减少。然而，为了确保软件的质量，必须有人参与最终的测试和评估。现在，我们将探索软件工程中的单元测试、测试驱动开发和行为驱动开发概念。

探索单元测试、TDD 和 BDD

在前一节中，我们学习了自动静态和动态程序分析。本节将重点讨论涉及人的（准备测试代码）测试，这是动态分析的另一部分。它们是单元测试、测试驱动开发和行为驱动开发。

单元测试假定，如果我们已经有了单个代码单元，那么我们需要编写一个测试驱动程序，并准备输入数据，以检查其输出是否正确。在那之后，我们执行集成测试来测试多个单元，然后是验收测试，测试整个应用程序。由于集成和验收测试比单元测试更难维护，而且与项目更相关，所以在本书中介绍它们非常具有挑战性。

感兴趣的同学可以登录 <https://www.iso.org/standard/45142.html> 了解更多信息。

与单元测试不同，TDD 认为我们应该先测试代码和数据，开发一些代码并使其快速通过，最后重构直到客户满意。另一方面，BDD 的理念是我们不应该测试程序的实现，而应该测试它想要的行为。为此，BDD 强调，还应该建立软件生产人员之间的交流平台和语言。

我们将在下面的小节中详细讨论这些方法。

单元测试

单元是更大或更复杂的应用程序中的单个组件。通常，一个单元有它自己的用户接口，比如一个函数，一个类，或者一个完整的模块。单元测试是一种软件测试方法，用于确定代码单元是否按照其设计需求的预期行为。单元测试的主要特点如下：

- 它小巧而简单，编写和运行速度很快，因此，它可以在开发周期的早期发现问题，因此可以很容易地修复这些问题。
- 因为它与依赖项隔离，所以每个测试用例都可以并行运行。
- 单元测试驱动程序帮助我们理解单元接口。
- 在集成测试单元时，会极大地帮助集成和验收测试。
- 通常由开发人员准备和执行。

虽然我们可以从头开始编写单元测试包，但是社区中已经开发了很多单元测试框架（UTFs）。提振。Test、CppUnit、GoogleTest、Unit++ 和 CxxTest 是最流行的。这些 UTFs 通常有以下特性：

- 只需要少量的工作来设置一个新的测试。

- 依赖于标准库并支持跨平台，很容易移植和修改。
- 支持测试固件，这允许我们为多个不同的测试重用对象的相同配置。
- 能很好地处理异常和崩溃。这意味着 UTF 可以报告异常，但不能报告崩溃。
- 有很好的断言功能。每当断言失败时，应该打印它的源代码位置和变量的值。
- 支持不同的输出，这些输出可以方便地由人工或工具进行分析。
- 支持测试套件，每个套件可能包含几个测试用例。

现在，让我们看一个 Boost UTF 的示例 (从 v1.59.0 开始)。它支持三种不同的使用变体：仅头库、静态库和共享库。它包括四种类型的测试用例：没有参数的测试用例、数据驱动的测试用例、模板测试用例和参数化测试用例。

它还有七种类型的检查工具：

- BOOST_TEST()
- BOOST_CHECK()
- BOOST_REQUIRE()
- BOOST_ERROR()
- BOOST_FAIL()
- BOOST_CHECK_MESSAGE()
- BOOST_CHECK_EQUAL()。

它还以多种方式支持 fixture 和控制测试输出。在编写测试模块时，我们需要遵循以下原则：

1. 定义我们的测试程序的名称。这将在输出消息中使用。
2. 请选择一个用法变体：仅头文件，链接静态库，或作为共享库。
3. 选择并向测试套件添加测试用例。
4. 对测试代码执行正确性检查。
5. 在每个测试用例之前初始化测试代码。
6. 定制报告测试失败的方式。
7. 控制构建测试模块的运行时行为，也称为运行时配置。

例如，下面的示例将介绍步骤 1-4。如果你有兴趣，你可以在

https://www.boost.org/doc/libs/1_70_0/libs/test/doc/html/

上获得步骤 5-7 的例子：

```

1 //ch13_unit_test1.cpp
2 #define BOOST_TEST_MODULE my_test //item 1, "my_test" is module name
3 #include <boost/test/include/unit_test.hpp> //item 2, header-only
4
5 //declare we begin a test suite and name it "my_suite"
6 BOOST_AUTO_TEST_SUITE( my_suite )
7
8 //item 3, add a test case into test suit, here we choose
9 // BOOST_AUTO_TEST_CASE and name it "test_case1"
10 BOOST_AUTO_TEST_CASE(test_case1) {
11     char x = 'a';
12     BOOST_TEST(x); //item 4, checks if c is non-zero

```

```

13 BOOST_TEST(x == 'a'); //item 4, checks if c has value 'a'
14 BOOST_TEST(x == 'b'); //item 4, checks if c has value 'b'
15 }
16
17 //item 3, add the 2nd test case
18 BOOST_AUTO_TEST_CASE( test_case2 )
19 {
20     BOOST_TEST( true );
21 }
22
23 //item 3, add the 3rd test case
24 BOOST_AUTO_TEST_CASE( test_case3 )
25 {
26     BOOST_TEST( false );
27 }
28
29 BOOST_AUTO_TEST_SUITE_END() //declare we end test suite

```

为此，我们可能需要安装 boost，如下所示：

```
1 sudo apt-get install libboost-all-dev
```

然后，我们可以构建并运行它，如下所示：

```
1 ~/wus1/Chapter-13$ g++ -g c
```

上述代码的结果如下：

```

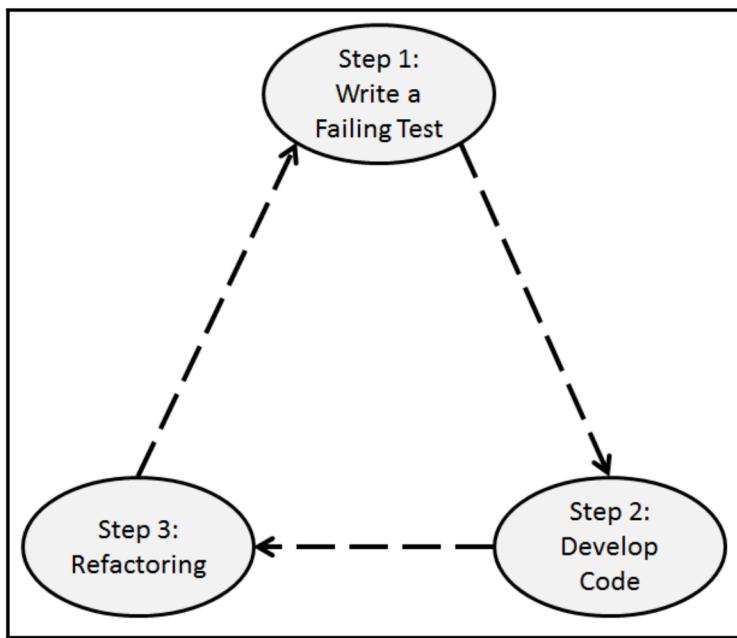
1 Running 3 test cases ...
2 ch13_unit_test1.cpp(13): error: in "my_suite/test_case1": check x == 'b'
3 has failed [ 'a' != 'b' ]
4 ch13_unit_test1.cpp(25): error: in "my_suite/test_case3": check false has
5 failed
6 *** 2 failures are detected in the test module "my_test"

```

这里，我们可以看到 test_case1 和 test_case3 中有失败。特别是，在 test_case1 中，x 的值不等于 b，而且很明显，检查不能通过 test_case3 中的测试。

TDD

如下图所示，TDD 流程首先编写失败的测试代码，然后添加/修改代码，让测试通过。在那之后，我们重构测试计划并编写代码，直到所有的需求都得到满足。让我们来看看下面的图表：



第 1 步是编写一个失败的测试。TDD 不是首先开发代码，而是首先开始编写测试代码。因为我们还没有代码，所以我们知道，如果我们运行测试，它将会失败。在这一阶段，定义测试数据格式和接口，并设想代码实现细节。

第 2 步的目标是用最少的开发工作使测试尽可能快地通过。我们不想完美地实现每件事，我们只想让它通过测试。一旦它变成绿色，我们就有东西可以展示和告诉客户，这时客户可能会在看到最初的产品后改进需求。然后，我们进入下一阶段。

第三个阶段是重构。在这一阶段，我们可能会走进去，看看我们想要改变什么，以及如何改变它。

对于传统的开发人员来说，TDD 最困难的事情就是从“编码-> 测试”模式转变为“测试-> 编码”模式。为了对测试套件有一个模糊的概念，J. Hartikainen 建议开发人员从一开始考虑以下五个步骤：

1. 首先决定输入和输出。
2. 选择类/函数签名。
3. 只决定要测试的功能的一个微小方面。
4. 实现测试。
5. 实现的代码。

当我们完成了这个迭代，我们就可以逐步重构它，直到实现了整体的全面目标。

TDD 的例子

接下来，我们将通过一个案例研究的实现来演示 TDD 过程。过程中中，我们将开发一个 Mat 类来执行二维矩阵代数，就像我们在 Matlab 中所做的那样。这是一个类模板，可以保存所有数据类型的 $m \times n$ 矩阵。矩阵代数包括矩阵的加、减、乘、除，还具有元素运算能力。

让我们开始吧！

步骤 1 - 编写一个失败的测试

首先，我们只需要以下内容：

1. 根据给定的行数和 cols 创建一个 Mat 对象 (默认值应该是 0×0 ，这是一个空矩阵)。
2. 逐行打印它的元素。
3. 从 rows() 和 cols() 中获取矩阵大小

基于这些需求，我们可以有失败的单元测试代码来增强 USTF，如下所示：

```
1 // ch13_tdd_boost_UTF1.cpp
2 #define BOOST_TEST_MODULE tdd_test
3 #include <boost/test/included/unit_test.hpp>
4 #include "ch13_tdd_v1.h"
5
6 BOOST_AUTO_TEST_SUITE(tdd_suite) //begin a test suite: "tdd_suite"
7
8 BOOST_AUTO_TEST_CASE(test_case1) {
9     Mat<int> x(2, 3); //create a 2 x 3 int matrix
10    x.print("int x=");
11    BOOST_TEST(2 == x.rows());
12    BOOST_TEST(3 == x.cols());
13
14    Mat<float> y; //create a 0 x 0 empty float matrix
15    y.print("float y=");
16    BOOST_TEST(0 == y.rows());
17    BOOST_TEST(0 == y.cols());
18
19    Mat<char> z(1, 10); //create a 1 x 10 char matrix
20    z.print("char z=");
21    BOOST_TEST(1 == z.rows());
22    BOOST_TEST(10 == z.cols());
23 }
24 BOOST_AUTO_TEST_SUITE_END() //end test suite
```

既然我们的测试代码已经准备好了，我们就可以开发代码了。

步骤 2 - 开发代码让测试通过

实现最小代码段的一种方法是通过前面的测试，如下所示：

```
1 // file: ch13_tdd_v1.h
2 #ifndef __ch13_TDD_V1__
3 #define __ch13_TDD_V1__
4 #include <iostream>
5 #include <assert.h>
6 template< class T>
7 class Mat {
8 public:
9     Mat(const uint32_t m=0, const uint32_t n=0);
10    Mat(const Mat<T> &rhs) = delete;
11    ~Mat();
```

```

12
13 Mat<T>& operator = (const Mat<T> &x) = delete;
14
15 uint32_t rows() { return m_rows; }
16 uint32_t cols() { return m_cols; }
17 void print(const char* str) const;
18
19 private:
20 void creatBuf();
21 void deleteBuf();
22 uint32_t m_rows; //# of rows
23 uint32_t m_cols; //# of cols
24 T* m_buf;
25 };
26 #include "ch13_tdd_v1.cpp"
27 #endif

```

我们有了前面的头文件，就可以开发它对应的 cpp，如下所示：

```

1 //file: ch13_tdd_v1.cpp
2 #include "ch13_tdd_v1.h"
3 using namespace std;
4
5 template< class T>
6 Mat<T>::Mat(const uint32_t m, const uint32_t n)
7 : m_rows(m)
8 , m_cols(n)
9 , m_buf(NULL)
10 {
11     creatBuf();
12 }
13
14 template< class T>
15 Mat<T> :: ~Mat()
16 {
17     deleteBuf();
18 }
19
20 template< class T>
21 void Mat<T>::creatBuf()
22 {
23     uint32_t sz = m_rows * m_cols;
24     if (sz > 0) {
25         if (m_buf) { deleteBuf(); }
26         m_buf = new T[sz];
27         assert(m_buf);
28     }
29     else {
30         m_buf = NULL;
31     }

```

```

32 }
33
34 template< class T>
35 void Mat<T>::deleteBuf()
36 {
37     if (m_buf) {
38         delete [] m_buf;
39         m_buf = NULL;
40     }
41 }
42
43 template< class T>
44 void Mat<T> :: print(const char* str) const
45 {
46     cout << str << endl;
47     cout << m_rows << " x " << m_cols << "[ " << endl;
48     const T *p = m_buf;
49     for (uint32_t i = 0; i < m_rows; i++) {
50         for (uint32_t j = 0; j < m_cols; j++) {
51             cout << *p++ << ", ";
52         }
53         cout << "\n";
54     }
55     cout << "] \n";
56 }

```

假设我们使用 g++ 来构建和执行它，g++ 支持-std=c++11 或更高版本：

```

1 ~/wus1/Chapter-13$ g++ -g ch13_tdd_boost_UTF1.cpp
2 ~/wus1/Chapter-13$ a.out

```

输出如下：

```

1 Running 1 test case ...
2 int x=2 x[3[
3 1060438054, 1, 4348032,
4 0, 4582960, 0,
5 ]
6 float y=0 x[0[
7 ]
8 char z=1 x[10[
9 s,s,s,s,s,s,s,s,s,
10 ]

```

在 test_case1 中，我们创建了三个矩阵，并测试了 rows()、cols() 和 print() 函数。第一个是一个 2x3 的 int 型矩阵。由于它没有初始化，其元素的值是未知的，这就是为什么我们可以从 print() 中看到这些随机数。此时我们还传递了 rows() 和 cols() 测试（两次 BOOST_TEST() 调用都没有错误）。第二个是一个空的浮点型矩阵，它的 print() 函数什么也不给出，cols() 和 rows() 都是零。最后，第三个是一个 1x10 的 char 类型未初始化矩阵。同样，这三个函数的所有输出都符合预期。

步骤 3 -重构

目前为止，一切顺利——我们通过了测试！但是，在将上述结果显示给我们的客户后，他/她可能会要求我们再增加两个接口，例如：

- 为所有元素创建一个具有给定初始值的 $m \times n$ 矩阵。
- 添加 `numel()` 返回矩阵的元素总数。
- 添加 `empty()`，如果矩阵的行数为零或列数为零，则返回 `true`，否则返回 `false`。

当将第二个测试用例添加到测试套件中，整个重构的测试代码如下：

```
1 // ch13_tdd_Boost_UTF2.cpp
2 #define BOOST_TEST_MODULE tdd_test
3 #include <boost/test/included/unit_test.hpp>
4 #include "ch13_tdd_v2.h"
5
6 //declare we begin a test suite and name it "tdd_suite"
7 BOOST_AUTO_TEST_SUITE(tdd_suite)
8
9 //add the 1st test case
10 BOOST_AUTO_TEST_CASE(test_case1) {
11     Mat<int> x(2, 3);
12     x.print("int x=");
13     BOOST_TEST(2 == x.rows());
14     BOOST_TEST(3 == x.cols());
15
16     Mat<float> y;
17     BOOST_TEST(0 == y.rows());
18     BOOST_TEST(0 == y.cols());
19
20     Mat<char> z(1, 10);
21     BOOST_TEST(1 == z.rows());
22     BOOST_TEST(10 == z.cols());
23 }
24
25 //add the 2nd test case
26 BOOST_AUTO_TEST_CASE(test_case2) {
27     Mat<int> x(2, 3, 10);
28     x.print("int x=");
29     BOOST_TEST( 6 == x.numel() );
30     BOOST_TEST( false == x.empty() );
31     Mat<float> y;
32     BOOST_TEST( 0 == y.numel() );
33     BOOST_TEST( x.empty() ); //bug x --> y
34 }
35
36 BOOST_AUTO_TEST_SUITE_END() //declare we end test suite
37
```

下一步是修改代码以通过这个新测。简单起见，我们在这里不展示 ch13_tdd_v2.h 和 ch13_tdd_v2.cpp 文件。可以从本书的 GitHub 存储库中下载它们。在构建并执行 ch13_tdd_Boost_UTF2.cpp 之后，我们得到如下输出：

```
1 Running 2 test cases ...
2     int x=2x3 [
3         1057685542, 1, 1005696,
4         0, 1240624, 0,
5     ]
6     int x=2x3 [
7         10, 10, 10,
8         10, 10, 10,
9     ]
10    .../Chapter-13/ch13_tdd_Boost_UTF2.cpp(34): error: in
11    "tdd_suite/test_case2": che
12        ck x.empty() has failed [(bool)0 is false]
```

第一个输出中，因为我们只是定义了一个 2x3 的整数矩阵，并且没有在 test_case1 中初始化它，所以将输出未定义的行为——即 6 个随机数。第二个输出来自 test_case2，其中 x 的所有 6 个元素都初始化为 10。在我们对前面的结果进行了展示和说明之后，我们的客户可能会要求我们添加其他新功能或修改当前现有的功能。经过几次迭代，最终，我们将到达满意点并停止分解。

现在我们已经了解了 TDD，我们将来了解 BDD。

BDD

软件开发中最困难的部分是与业务参与者、开发人员和质量分析团队进行沟通。一个项目很容易超出预算，错过最后期限，或者因为误解或模糊的需求、技术争论和缓慢的反馈周期而完全失败。

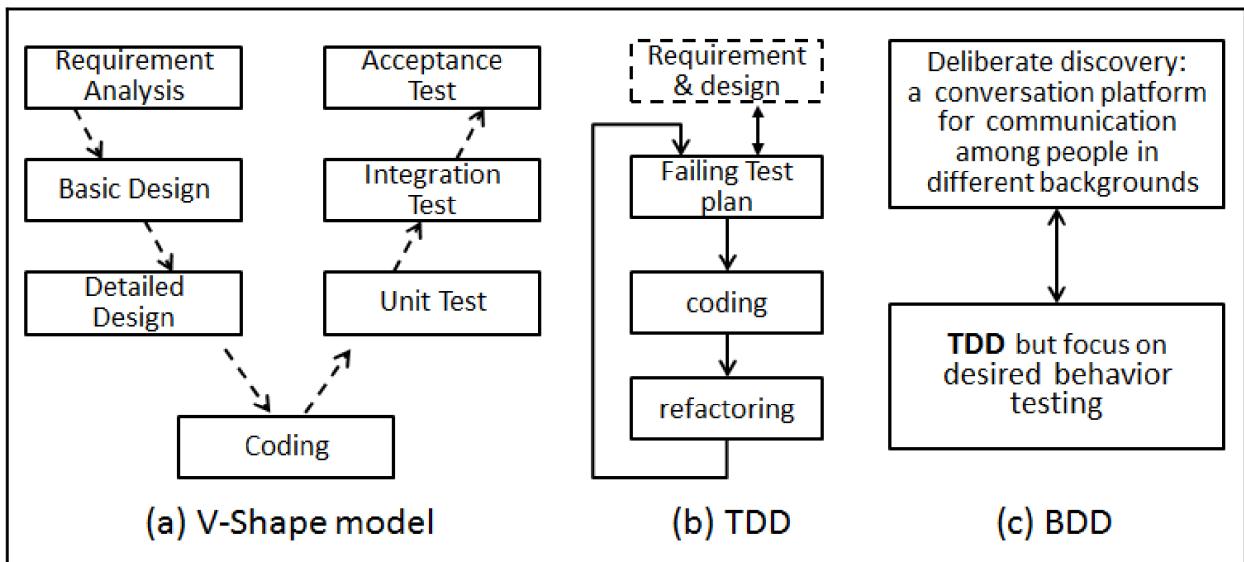
BDD 是一个敏捷开发过程，它包含一组实践，旨在减少沟通差距/障碍和其他浪费的活动。它还鼓励团队成员在生产生命周期中不断地与真实的示例进行交流。

BDD 包含两个主要部分：问题发现和 TDD。为了让不同组织和团队中的人理解所开发软件的正确行为，有意的在发现阶段引入了一个示例映射技术，使不同角色的人通过具体的示例进行对话。这些例子将成为系统行为的自动化测试和活生生的文档。在 TDD 阶段，BDD 指定任何软件单元的测试都应该根据单元的期望行为来指定。

有几种针对不同平台和编程语言的 BDD 框架工具 (JBehave、RBehave、Fitness、Cucumber 等)。一般来说，这些框架执行以下步骤：

1. 阅读业务分析人员在有意发现阶段准备的规范格式文档。
2. 将文档转换成有意义的子句。每个单独的子句都可以设置到 QA 的测试用例中。开发人员也可以从子句实现源代码。
3. 自动为每个子句场景执行测试。

总之，我们已经了解了应用程序开发管道中应该涉及什么、何时、如何以及测试过程的策略。如下图所示，传统的 v 形模型强调需求模式-> 设计-> 编码-> 测试。TDD 认为开发过程应该由测试驱动，而 BDD 将不同背景和角色的人之间的交流添加到 TDD 框架中，侧重于行为测试：



此外，单元测试强调在编码完成时对单个组件进行测试。TDD 更侧重于如何在编写代码之前编写测试，然后通过下一个级别的测试计划添加/修改代码。BDD 鼓励客户、业务分析人员、开发人员和质量保证分析人员之间的协作。尽管可以单独使用它们，但应该将它们结合起来。

总结

在本章中，我们简要介绍了软件开发过程中与测试和调试相关的主题。测试可以发现问题，而根本原因分析有助于在宏观层次上定位问题。然而，良好的编程习惯可以在早期阶段防止软件缺陷。此外，称为 GDB 的命令行接口调试工具，可以帮助我们设置断点并逐行执行程序，同时在程序运行期间打印变量的值。

我们还讨论了自动分析工具和涉及人工测试的过程。静态分析在不执行程序的情况下评估程序的性能。另一方面，动态分析工具可以通过执行程序来发现缺陷。最后，我们了解了在软件开发管道中应该涉及什么、何时以及如何涉及测试过程的策略。单元测试强调在编码完成时对单个组件进行测试。TDD 更侧重于如何在开发代码之前编写测试，然后通过下一个级别的测试计划来重复这个过程。BDD 鼓励客户、业务分析人员、开发人员和质量保证分析人员之间的协作。

在下一章中，我们将学习如何使用 Qt 为跨平台应用程序创建图形用户界面 (GUI) 程序，这些应用程序运行在 Linux、Windows、iOS 和 Android 系统上。首先，我们将深入研究跨平台 GUI 编程的基本概念。然后我们将介绍 Qt 及其小部件的概述。最后，通过一个案例研究的例子，我们将学习如何使用 Qt 设计和实现一个网络应用。

扩展阅读

- J. Rooney and L. Vanden Heuvel, Root Cause Analysis For Beginners , Quality Progress, July 2004, p.45-53.
- T. Kataoka, K. Furuto and T. Matsumoto, The Analyzing Method of Root Causes for Software Problem s, SEI Tech. Rev., no. 73, p. 81, 2011.
- K. A. Briski, et al. Minimizing code defects to improve software quality and lower development costs , IBM Rational Software Analyzer and IBM Rational PurifyPlus software.

- <https://www.learncpp.com/cpp-programming/eight-c-programming-mistakes-the-compiler-wont-catch> .
- B. Stroustrup and H. Sutter, C++ Core Guidelines: <https://isocpp.github.io/CppCoreGuidelines>

练习和问题

1. 使用 gdb 对 ch13_gdb_2.cpp 进行断点、条件断点的设置，以及 watchpoint, continue 和 finish 命令的使用。
2. 使用 `g++ -c -Wall - Weffc++ -Wextra x.cpp -o x.out` 编译 cpp 文件 ch13_rca*.cpp。警告中看到了什么？
3. 为什么静态分析会产生误报，而动态分析不会？
4. 下载 ch13_tdd_v2.h/.cpp 并执行下一阶段的重构。这时，我们将添加复制构造函数、赋值操作符和元素操作符，如 +、-、*、/ 等。具体来说，要做到以下几点：
 - (a) 将第三个测试用例添加到我们的测试套件中，即 ch13_tdd_Boost_UTF2.cpp。
 - (b) 将这些函数的实现添加到文件中，例如：ch13tdd_v2.h / .cpp
 - (c) 运行测试套件进行测试。

第 14 章：使用 Qt 开发图形界面

C++ 不提供即用的图形用户界面 (GUI) 编程。首先，我们应该理解 GUI 与特定的操作系统 (OS) 紧密相连。可以在 Windows 中使用 Windows API 编写 GUI 应用程序，也可以在 Linux 中使用特定于 Linux 的 API 编写 GUI 应用程序等。每个操作系统都有自己特定形式的 Windows 和 GUI 组件。

我们在第 1 章中讨论了不同的平台及其差异。当讨论 GUI 编程时，平台之间的差异更加令人生畏。跨平台开发已经成为 GUI 开发人员生活中的一大难题。他们必须专注于特定的操作系统。为其他平台实现相同的应用程序再次花费了几乎相同的工作量。这是对时间和资源的巨大浪费。Java 等语言提供了在虚拟环境中运行应用程序的智能模型。这允许开发人员专注于一种语言和一个项目，因为环境负责在不同平台上运行应用程序。这种方法的一个主要缺点是强迫用户安装虚拟机，并且与特定于平台的应用程序相比，执行时间较慢。

为了解决这些问题，创建了 Qt 框架。在本章中，我们将了解 Qt 框架如何支持跨平台 GUI 应用程序开发。要做到这一点，您需要熟悉 Qt 及其关键特性。这将允许您使用您最喜欢的编程语言——C++ 开发 GUI 应用程序。我们将从理解 Qt 的 GUI 开发方法开始，然后我们将涵盖它的概念和特性，如信号和槽，以及模型/视图编程。

本章中，我们将了解以下内容：

- 跨平台 GUI 编程基础
- Qt 核心组件
- 使用 Qt widget
- 用 Qt 网络设计一个网络应用程序

编译器要求

您将需要安装最新的 Qt 框架来运行本章中的示例。我们建议使用 Qt Creator 作为你的项目的 IDE。要下载 Qt 以及相应的工具，请访问 [Qt .io](https://www.qt.io) 网站并选择该框架的开源版本。可以从这里获取本章的源码文件：<https://github.com/PacktPublishing/Expert-CPP>

理解跨平台 GUI 编程

每个操作系统都有自己的 API，并与 GUI 有关。当公司计划设计、实现和发布桌面应用程序时，他们应该决定关注哪个平台。在一个平台上工作的开发团队需要花费几乎相同的时间为另一个平台编写相同的应用程序。最大的原因是 OS 提供了不同的方法和 API，API 的复杂性也可能在及时实现应用程序方面扮演重要角色。例如，官方文档中的以下代码片段展示了如何在 Windows 中使用 C++ 创建按钮：

```
1 HWND hwndButton = CreateWindow(
2     L"BUTTON", // Predefined class; Unicode assumed
3     L"OK", // Button text
4     WS_TABSTOP | WS_VISIBLE | WS_CHILD | BS_DEFPUSHBUTTON, // Styles
5     10, // x position
6     10, // y position
7     100, // Button width
8     100); // Button height
```

```
9 m_hwnd, // Parent window  
10 NULL, // No menu.  
11 (HINSTANCE)GetWindowLong(m_hwnd, GWL_HINSTANCE),  
12 NULL); // Pointer not needed.
```

处理 Windows GUI 编程需要你使用 HWND、HINSTACNCE 和许多其他命名怪异、令人困惑的组件。

.NET 框架对 Windows GUI 编程进行了重大改进。如果你想要支持 Windows 以外的操作系统，在使用.NET 框架之前，必须三思而后行。

然而，为了支持多个操作系统，仍然需要深入研究 API 来实现相同的应用程序，以覆盖所有的操作系统用户。下面的代码展示了在 Linux 中使用 Gtk+GUI 工具包创建按钮的示例：

```
1 GtkWidget* button = gtk_button_new_with_label("Linux button");
```

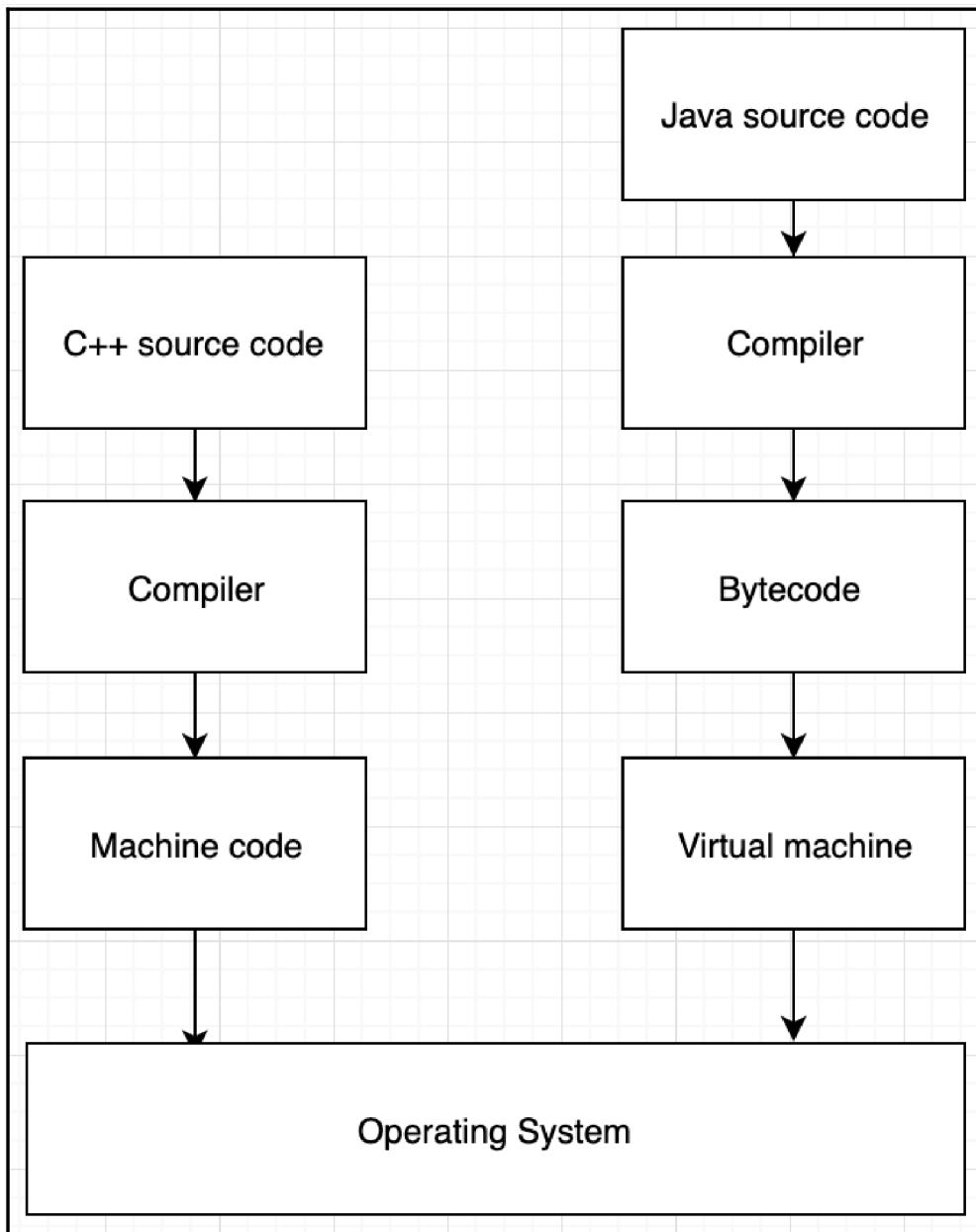
与 Windows API 相比，它似乎更容易理解。但是，您应该深入研究带有 Gtk 前缀的 GtkWidgets 和其他组件，以了解有关它们的更多信息。

正如我们已经提到的，跨平台语言，如 Java 和 .Net Core 使用虚拟机在不同的平台上运行代码。Qt 框架使用基于平台的编译方法支持跨平台 GUI 编程。让我们就 C++ 语言来讨论这两种方式吧。

如 Java 一样使用 C++

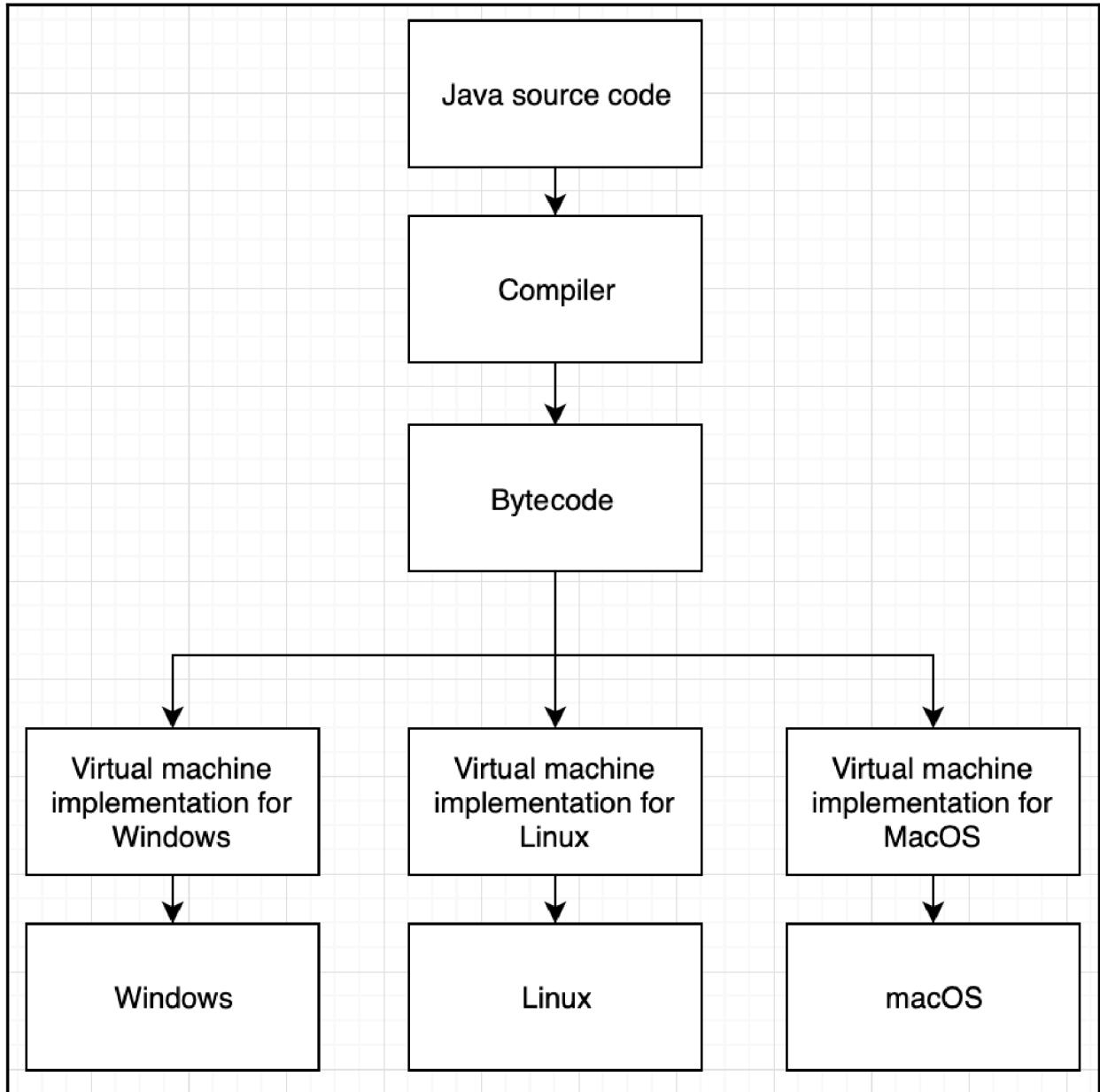
像 Java 或 C# 这样的语言有不同的编译模型。本书第一章介绍了 C++ 的编译模型。首先，我们认为 C++ 是一种完全可编译的语言，而 Java 是一种混合模型语言。它将源代码编译为称为字节码的中间表示，然后虚拟机通过将其转换为特定平台的机器码来运行它。

下图描述了 C++ 和 Java 编译模型之间的区别：



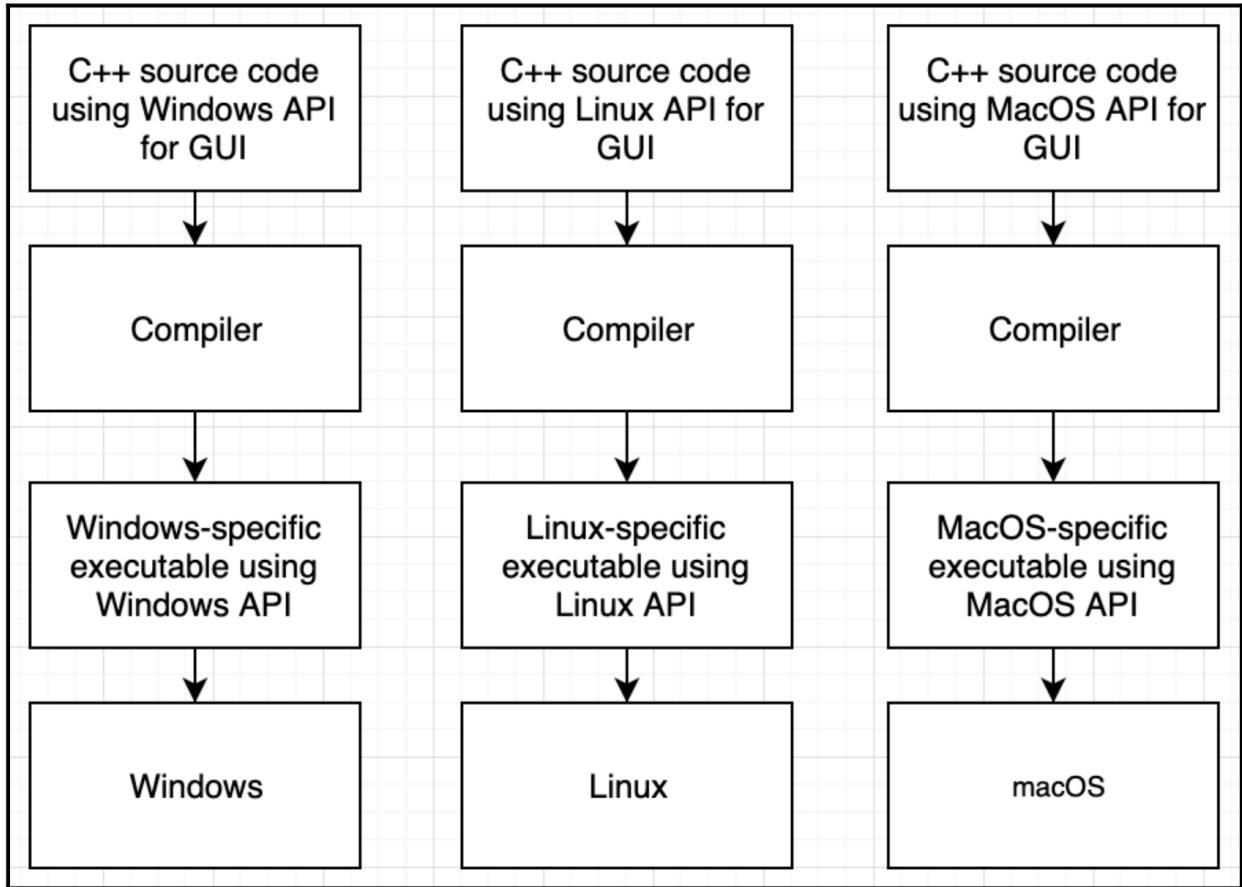
Java 虚拟机 (JVM) 充当中间层，对每个平台都有一个独特的实现。用户需要在运行 Java 程序之前安装虚拟机的特定实现，安装过程只发生一次。另一方面，C++ 程序被翻译成机器码，这些机器码在没有 JVM 等中间层环境的情况下运行。这就是为什么 C++ 应用程序通常更快的原因之一。当我们在某个平台上编译 C++ 程序时，编译器输出一个可执行文件，该文件由特定于该平台的格式的指令组成。当我们把应用程序转移到另一个平台时，它就无法运行了。

另一个平台不能识别它的格式，也不能识别它的指令（尽管它们在某些方面可能相似）。Java 方法的工作原理是提供一些字节码，这些字节码对于虚拟机的所有实现都是相同的。但是虚拟机知道它们作为输入的字节码生成什么指令。相同的字节码可以在许多安装了虚拟机的计算机上运行。下面的图表演示了 Java 应用程序编译模型：



源代码编译成可以在每个操作系统上运行的字节码。但是，必须为每个操作系统提供自己的虚拟机实现。这意味着我们可以在任何操作系统上运行 Java 应用程序，只要我们安装了专门为该操作系统实现的 JVM。

尽管 C++ 是一种跨平台语言，这意味着我们不需要修改代码来在其他平台上编译它，但这种语言并不支持 GUI 编程。如前所述，要编写 GUI 应用程序，我们需要直接从代码访问 OS API。这使得 C++ GUI 应用程序依赖于平台，因为您需要修改代码库以在另一个平台上编译它。下图显示了 GUI 如何破坏了语言的跨平台特性：



尽管应用程序逻辑、名称和任务可能是相同的，但它现在有三个不同的实现和三个不同的可执行文件。要将应用程序交付给最终用户，我们需要发现他们的操作系统并交付正确的可执行文件。在 Web 上下载应用程序时，可能遇到过类似的情况。他们提供基于操作系统的下载应用程序。这就是 Qt 要一展身手的地方，让我们一起来看看。

Qt 跨平台的模式

Qt 是一个流行的用于创建 GUI 应用程序的工具包。它允许我们创建运行在各种系统上的跨平台应用程序。Qt 由以下模块组成：

- Qt Core: 核心类
- Qt GUI: GUI 组件的基类
- Qt Widgets: 用 C++ widget 扩展 Qt GUI 的类
- Qt Multimedia: 音频、视频、广播和相机功能的课程
- Qt Multimedia Widgets: 用于实现多媒体功能的类
- Qt Network: 网络编程类 (我们将在本章中使用)
- Qt Modeling Language (QML): 用于构建具有自定义用户界面的应用程序的声明性框架
- Qt SQL: 使用 SQL 集成数据库的类
- Qt Quick family of modules: 本书中不会讨论的 QML 相关的模块。
- Qt Test: 用于单元测试 Qt 应用程序的类

程序中使用的每个模块都通过一个后缀为.pro 的项目文件附加到编译器中。这个文件描述了

qmake 构建应用程序所需的所有内容，qmake 是一个旨在简化构建过程的工具。描述项目的.pro 文件中的项目组件（源代码、Qt 模块、库等），例如：一个使用 Qt Widgets 和 Qt 网络的项目，由 main.cpp 和 test.cpp 文件组成，.pro 文件的内容如下：

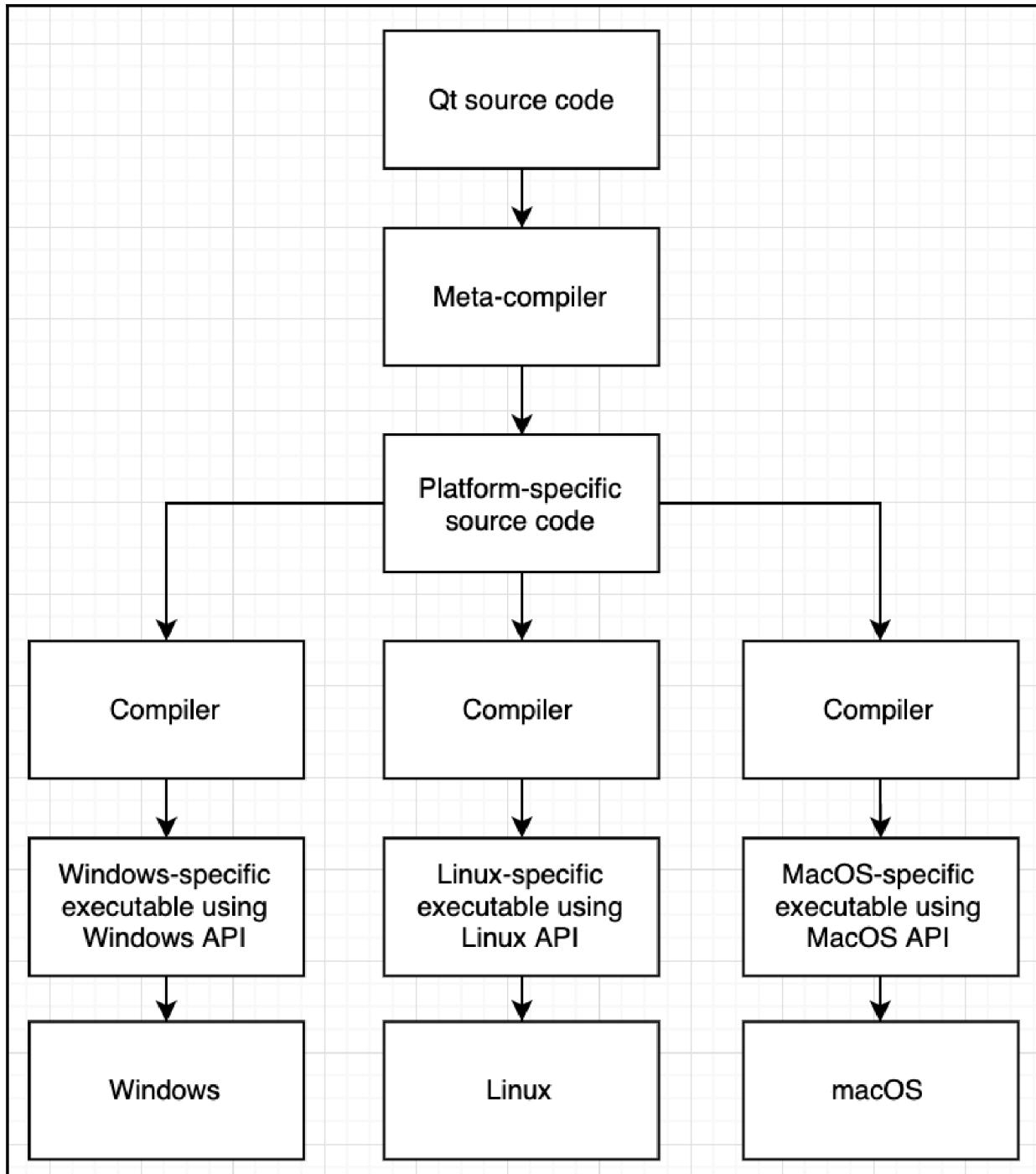
```
1 QT += widgets
2 QT += network
3 SOURCES += test.cpp
4 SOURCES += main.cpp
```

我们也可以在.pro 文件中指定特定于平台的源文件，如下所示：

```
1 QT += widgets
2 QT += network
3 SOURCES += test.cpp
4 SOURCES += main.cpp
5 win32 {
6     SOURCES += windows_specific.cpp
7 }
8 unix {
9     SOURCES += linux_world.cpp
10 }
```

当我们在 Windows 环境中构建应用程序时，windows_specific.cpp 文件将参与构建。相反，在 Unix 环境中构建时，将包含 linux_world.cpp 文件，忽略 windows_specific.cpp 文件。至此，我们遇到了 Qt 应用程序的编译模型。

Qt 提供跨平台编程的强大能力的全部意义在于元编译源代码，在代码传递给 C++ 编译器之前，Qt 编译器通过引入或替换特定于平台的组件来清理代码。例如，当我们使用一个按钮组件（QPushButton）时，如果在 Windows 环境中编译，它将被 Windows 的按钮组件所取代。这就是为什么.pro 文件也可以包含项目特定于平台的修改。下面的图表描述了这一过程：



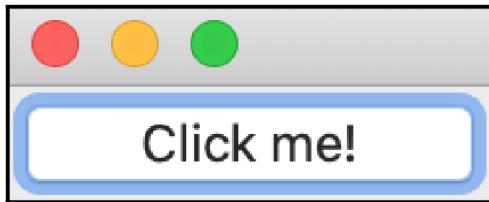
元编译器通常称为元对象编译器 (MOC)。这种方法的美妙之处在于，生成的输出代表了在没有虚拟机的情况下运行的相同的机器代码，我们可以马上发布这个可执行文件。这种方法的缺点是，对于不同的平台，我们有不同的可执行文件。然而，我们只编写一个应用程序——不需要使用不同的语言，不需要钻研特定于操作系统的 API，也不需要学习特定于操作系统的 GUI 组件类名。就像 Qt 说的，*Write once, compile everywhere*。现在，让我们继续构建一个简单的 GUI 应用程序。

编写一个简单的应用程序

我们不会讨论提到的所有模块，因为这需要一本全新的书来进行。你可以参考本章末尾列出的书，在扩展阅读部分，以获得更多信息。主函数是这样的：

```
1 #include <QtWidgets>
2
3 int main( int argc, char** argv )
4 {
5     QApplication app(argc, argv);
6
7     QPushButton btn( "Click me!" );
8     btn.show();
9     return app.exec();
10 }
```

让我们看看代码中使用的各种组件。第一个是 QtWidgets 头文件，它包含 widget 组件，我们可以使用这些组件为应用程序构建细粒度的 GUI。接下来是 QPushButton 类，它表示可单击按钮的包装器。我们在这里有意地把它作为一个包装器来介绍，以便在本章后面讨论 Qt 程序的编译过程时能够解释它。下面是运行上述代码的结果：



我们只声明了 QPushButton 类，但它显示为一个带有操作系统标准的关闭和最小化按钮的窗口（示例环境为 macOS）。这样做的原因是 QPushButton 间接继承自 QWidget，它是一个带有框架的 widget，也就是一个窗口。这个按钮几乎占据了整个窗口的空间。我们可以调整窗口的大小，看看按钮是如何随它一起调整大小的。我们将在本章后面更详细地讨论 widget。

当我们运行 app.exec() 时，GUI 就会构建。注意 app 对象的类型。它是一个 QApplication 对象。这是 Qt 应用程序的起点。当我们调用 exec() 函数时，我们开始 Qt 的事件循环。为了理解 GUI 应用程序的生命周期，我们应该稍微改变一下对程序执行的看法。在第 7 章之后，重新定义程序构造和执行的概念并不奇怪这里需要了解的主要内容是 GUI 应用程序有一个与主程序一起运行的实体。这个实体称为事件循环。



TIP 回想一下我们在第 11 章中讨论过的事件循环。这个游戏代表了一个带有视觉组件的程序，用户可以与这些组件进行密集的交互。这同样适用于带有按钮、标签和其他图形组件的常规 GUI 应用程序。

用户与应用程序交互，用户的每个操作都解释为一个事件，然后将每个事件推入队列。事件循环一个接一个地处理这些事件。处理事件意味着调用附加在事件上的特殊处理函数，例如：每当单击一个按钮时，就会调用 keyPressEvent() 函数。它是一个虚函数，所以我们在设计自定义按钮时覆盖它，代码如下所示：

```
1 class MyAwesomeButton : public QPushButton
2 {
```

```
3     Q_OBJECT
4
5     public:
6     void keyPressEvent(QKeyEvent* e) override
7     {
8         // anything that we need to do when the button is pressed
9     }
9;
```

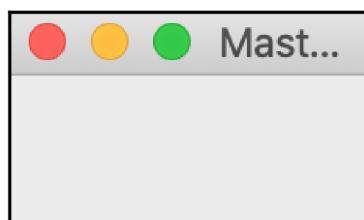
事件的唯一参数是指向 QKeyEvent 的指针，QEvent 的子类型。QEvent 是 Qt 中所有事件类的基类，注意在类的开始块后面放置了一个奇怪的 Q_OBJECT。这是一个 Qt 特有的宏，如果想让 Qt 的 MOC 发现，应该把它放在自定义类的第一行。

下一节中，我们将介绍 Qt 对象特有的信号和槽机制。为了使我们的自定义对象支持这种机制，我们在类定义中放置 Q_OBJECT 宏。

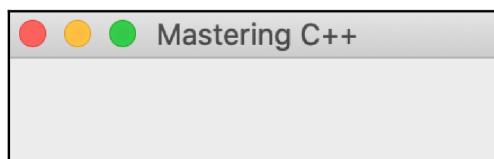
现在，让我们构建一个比简单按钮更大的东西。下面的例子创建了一个标题为 Mastering C++ 的窗口：

```
1 #include <QtWidgets>
2 int main(int argc, char** argv)
3 {
4     QApplication app(argc, argv);
5
6     QWidget window;
7     window.resize(120, 100);
8     window.setWindowTitle("Mastering C++");
9     window.show();
10
11     return app.exec();
12 }
```

下面是我们通过执行前面的程序得到的结果：



标题部分没有显示完全。现在，如果我们手动调整它的大小或更改源代码，使其在 resize() 函数的第二个参数中具有更大的值，我们会得到以下结果：



窗口对象是 QWidget 类型的。QWidget 是所有用户界面对象的中心类。无论何时您想要创建一个自定义 widget 或扩展一个现有的 widget，您都可以直接或间接地从 QWidget 继承。它为每个

用例提供了很多功能。您可以使用 move() 函数在屏幕中移动它，也可以通过调用 showFullScreen() 使窗口全屏等。在前面的代码中，我们调用了 resize() 函数，该函数获取 widget 的宽度和高度以调整其大小。另外，请注意 setWindowTitle() 函数——将传递的字符串参数设置为窗口的标题。在代码中使用字符串值时，最好使用 QApplication::translate() 函数。它使程序的本地化变得更容易，因为当语言设置改变时，Qt 会自动用正确的翻译替换文本。QObject::tr() 提供了相同的功能。



QObject 是所有 Qt 类型的基类。在 Java 或 C# 这样的语言中，每个对象都直接或间接地继承自泛型类型，大多数是命名对象（C++ 不包含公共基类）。Qt 引入了 QObject，它附带了所有对象都应该支持的基本功能。

现在我们已经接触了 Qt 应用程序开发的基础知识，让我们深入了解一下这个框架并发现它的关键特性。

研究 Qt

Qt 随着时间的推移不断发展，在撰写本书时，它的版本是 5.14。它的第一个公开预发行版本是在 1995 年宣布的。20 多年过去了，现在 Qt 拥有很多功能强大的功能，这些功能几乎适用于所有平台，包括 Android 和 iOS 等移动系统。除了少数例外，我们可以满怀信心地用 C++ 和 Qt 为所有平台编写功能齐全的 GUI 应用程序。这是一个重大的游戏规则改变者，因为公司雇佣的是专注于一种技术的小型团队，而不是针对每个特定平台组建多个团队。

如果您是 Qt 的新手，强烈建议您尽可能地熟悉它（参阅本章末尾的书籍参考）。除了 GUI 框架提供的常规组件外，Qt 还引入了一些新的或在框架中巧妙实现的概念。其中一个概念是使用信号和插槽的对象之间的通信。

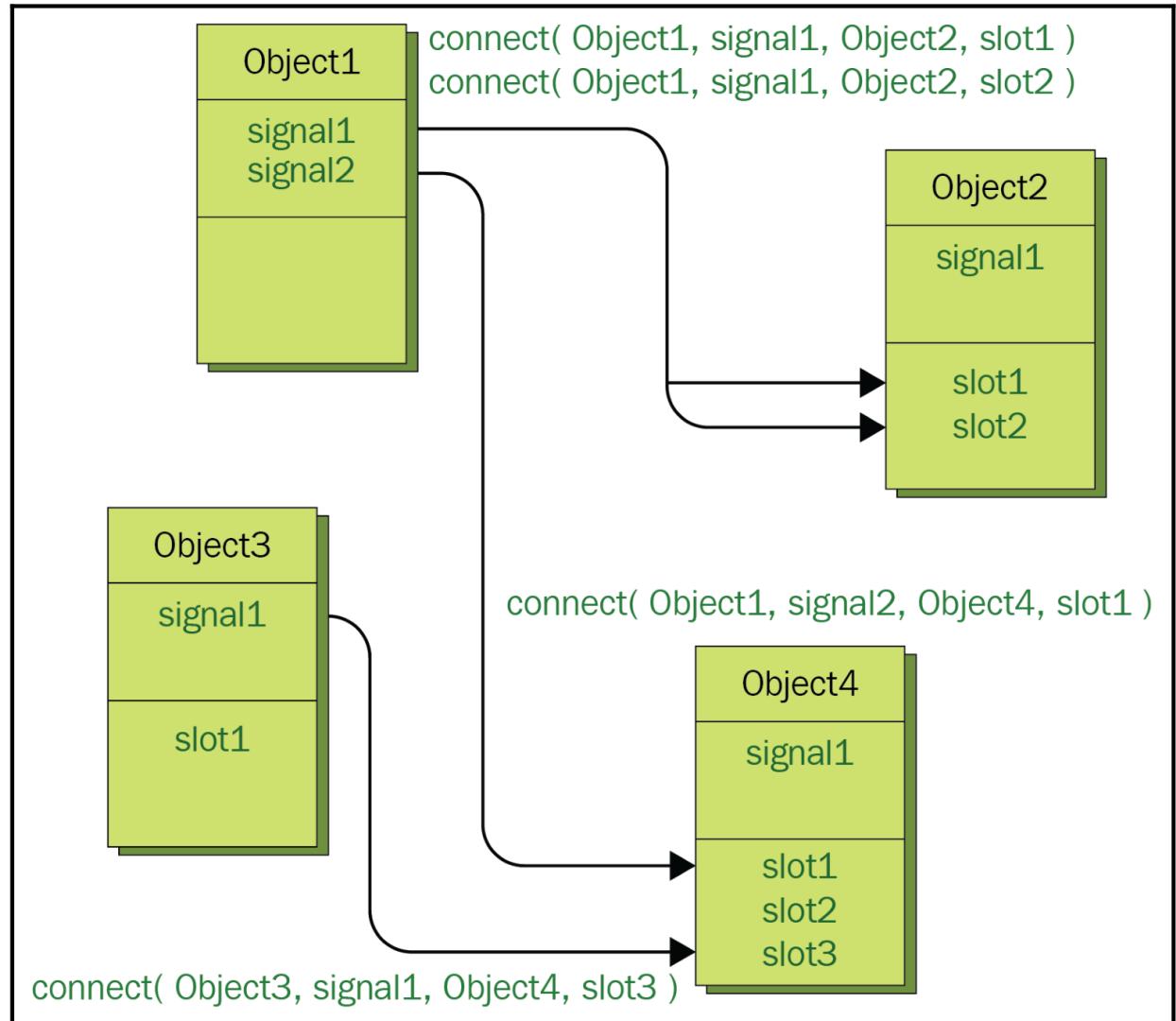
抓取信号和槽

Qt 引入了信号和槽的概念，作为对象之间的通信机制。信号和槽的概念及其实现机制是 Qt 与其他 GUI 框架区别开来的特性之一。前面的章节中，我们讨论了观察者模式。该模式的主要思想是使用一个对象将事件通知给其他对象（订阅者）。信号和槽的机制类似于观测器模式的实现。它是一个对象通知另一个对象它的变化的一种方式。Qt 提供了一个通用接口，可以通过将一个对象的信号绑定到另一个对象的插槽来将对象连接在一起。信号和槽都是对象的常规成员函数。信号是针对对象的指定操作调用的函数。槽位是作为订阅器的另一个功能。它由信号函数调用。

正如我们前面提到的，Qt 向我们介绍了所有对象的基类型 QObject。支持信号和槽的基本功能在 QObject 中实现。在代码中声明的任何对象、QWidget、QPushButton 等都继承自 QObject，因此它们都支持信号和槽。QObject 为我们提供了两个管理对象通信的函数，分别是 connect() 和 disconnect():

```
1 bool connect(const QObject* sender, const char* signal,
2               const QObject* receiver, const char* method,
3               Qt::ConnectionType type = Qt::AutoConnect);
4
5 bool disconnect(const QObject* sender, const char* signal,
6                 const QObject* receiver, const char* method);
```

`connect()` 函数将接收方和发送方对象作为参数。此外，它还接受信号和槽的名称。信号是与发送方相关联的，而插槽是接收方提供的。下图说明了这一点：



编写 Qt 应用程序时，使用信号和槽操作将变得很自然。另外，注意信号和槽在 `connect()` 和 `disconnect()` 函数中处理为字符串。要在连接对象时指定信号和槽，我们需要使用另外两个宏:`signal()` 和 `slot()`(从现在开始不会再引入更多的宏)。

下面是我们如何将两个物体连接在一起。假设我们想要更改标签 (QLabel 的一个实例) 的文本，以便在单击按钮时接收到一个信号。为此，我们将 QPushButton 的 `clicked()` 信号连接到 QLabel 槽位，如下所示：

```
1 QPushButton btn( "Click me!" );
2 QLabel lbl;
3 lbl.setText( "No signal received" );
4 QObject::connect(&btn, SIGNAL(clicked()), &lbl, SLOT(setText(const
5 QString&)));
```

前面的代码可能看起来有点冗长，可以把它看作是一种方便的信号和槽机制的付出。然而，前面的例子不会给出我们需要的结果，它不会设置标签的文本来声明它收到了信号。我们应该以某种

方式将该字符串传递给标签的槽，`clicked()` 信号不会为我们做这些。实现这一点的方法之一是扩展 `QLabel`，使其实现一个自定义槽，设置文本接收信号。我们可以这样做：

```
1 class MyLabel : public QLabel
2 {
3     Q_OBJECT
4     public slots:
5     void setCustomText() {
6         this->setText("received a signal");
7     }
8 };
```

为了声明一个槽，就像我们在前面的代码中所做的那样。信号的声明方式几乎相同：用 `Signals` 指定，唯一的区别是信号不能是私有的或受保护的。我们就这样声明它们：

```
1 class Example
2 {
3     Q_OBJECT:
4     public:
5         // member functions
6     public slots:
7         // public slots
8     private slots:
9         // private slots
10    signals: // no public, private, or protected
11        // signals without any definition, only the prototype
12};
```

现在，我们只需要更新前面的代码来改变标签的信号（以及标签对象的类型）：

```
1 QPushButton btn("Click me!");
2 MyLabel lbl;
3 lbl.setText("No signal received");
4 QObject::connect(&btn, SIGNAL(clicked()), &lbl, SLOT(setCustomText()));
```

当信号发出时，将调用槽。也可以从对象中声明和发射信号。与信号和插槽相关的一个重要细节是，它们独立于 GUI 事件循环。

当信号发出时，连接的槽立即执行。我们可以通过传递 `Qt::ConnectionType` 作为 `connect()` 函数的第五个参数来指定连接类型。它由以下值组成：

- `AutoConnection`
- `DirectConnection`
- `QueuedConnection`
- `BlockingQueuedConnection`
- `UniqueConnection`

在 `DirectConnection` 中，信号发出时立即调用槽。另一方面，当使用 `QueuedConnection` 时，当执行返回到接收方对象的线程的事件循环时，将调用槽。`BlockingQueuedConnection` 与 `QueuedConnection` 类似，不同之处在于会让线程被阻塞，直到槽返回一个值。自动连接可以是 `DirectCon-`

nection 或 QueuedConnection。如果接收端和发送端在同一个线程中，使用 DirectConnection，否则，连接将使用 QueuedConnection。最后，UniqueConnection 与前面描述的任何连接类型一起使用（它使用位或与其中的一个组合），唯一目的是让 connect() 函数在信号和线程之间已经建立了连接时失败。

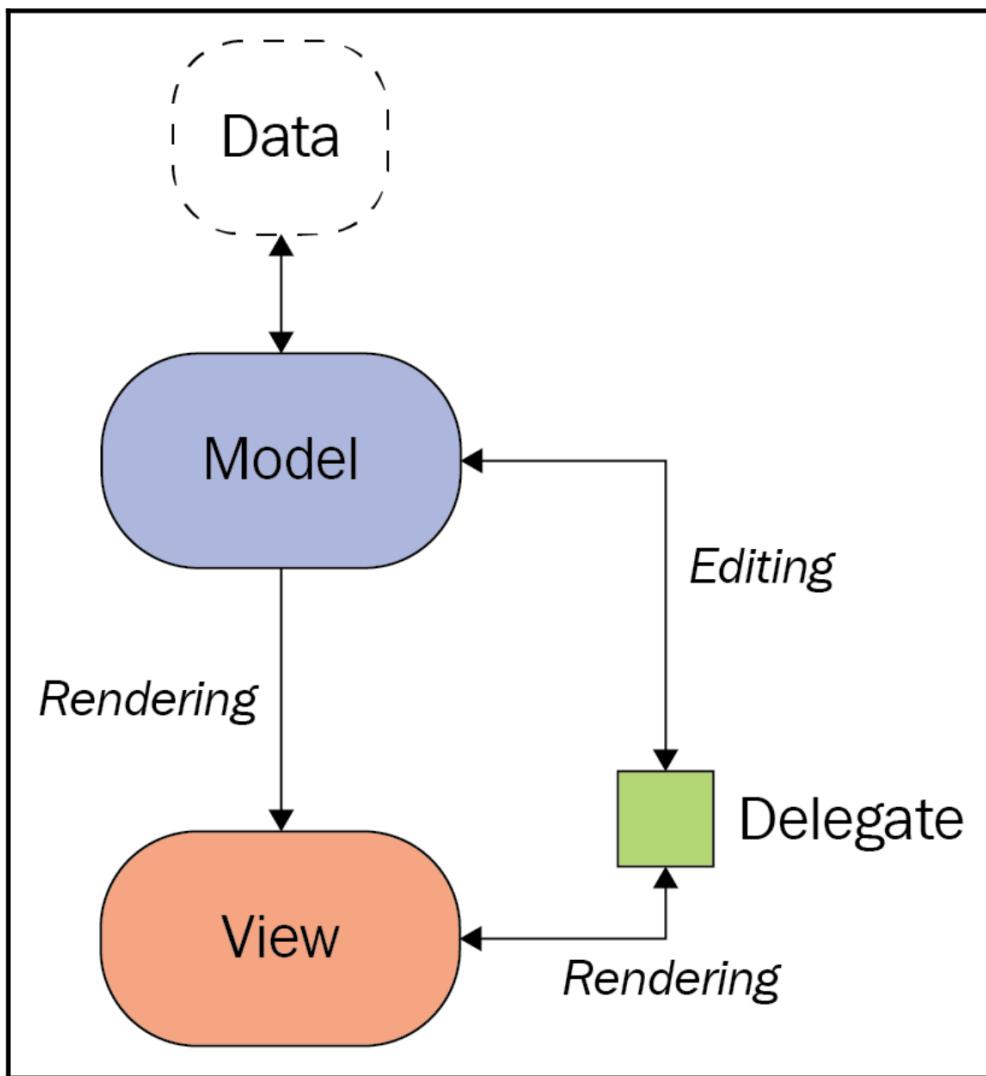
信号和槽形成了一种强大的机制，使 Qt 成为 GUI 编程中出色的框架。我们要介绍的下一个机制在框架中很流行，它与我们在应用程序中操作数据的方式有关。

了解编程模型/视图

模型/视图编程起源于模型-视图-控制器 (MVC) 设计模式。该模式背后的主要思想是将问题分解为三个松散耦合的组件，如下所示：

- 模型，负责存储和操作数据
- 视图，负责呈现和可视化数据
- 控制器，负责额外的业务逻辑，并提供从模型到视图的数据

通过它的发展，我们现在有了一种简化和更方便的编程方法，称为模型/视图编程。它与 MVC 模式相似，只是它省略了控制器，使视图和模型更关心手边的功能。我们可以说视图和控制器在模型/视图架构中结合在一起。看看下面的架构图：



模型表示数据，数据与其源通信，并为体系结构中的其他组件提供了一个方便的接口。模型的实现及其与其他组件的通信基于数据类型。

视图通过获得所谓的模型索引来获得对数据项的引用。视图可以检索并向模型提供数据。关键是，可以使用视图编辑数据项，委托扮演与模型通信以保持数据同步的角色。

引入的每个组件——模型、视图和委托——都是由提供公共接口的抽象类定义。某些情况下，类还提供特性的默认实现。为了编写专门化的组件，我们从抽象类派生子类。当然，模型、视图和委托使用信号和槽进行通信，我们在前一节中介绍了这一点。

当模型遇到数据中的更改时，它会通知视图。另一方面，用户与呈现数据项的交互是由视图发出的信号通知的。最后，来自委托的信号将数据编辑的状态通知模型和视图。

模型基于 `QAbstractItemModel` 类，该类定义了视图和委托使用的接口。Qt 提供了一组不需要修改就可以使用的现有模型类，但如果您需要创建新的模型，应该从 `QAbstractItemModel` 继承您的类。例如，`QStringListModel`、`QStandardItemModel` 和 `QFileSystemModel` 类可以处理数据项。`QStringListModel` 用于存储字符串项列表（表示为 `QString` 对象）。此外，还有一些方便使用 SQL 数据库的模型类。 `QSqlQueryModel`、 `QSqlTableModel` 和 `QSqlRelationalTableModel` 允许我们在模型/视图约定的上下文中访问关系数据库。

视图和代理也有相应的抽象类，即 `QAbstractItemView` 和 `QAbstractItemDelegate`。Qt 提供了

可以立即使用的现有视图，如 QListView、QTableView 和 QTreeView。这些是大多数应用程序处理的基本视图类型。QListView 显示项目列表，QTableView 显示表中的数据，QTreeView 显示层次化列表中的数据。如果你想使用这些视图类，Qt 建议从 QAbstractListModel 或 QAbstractTableModel 继承自定义模型，而不是子类化 QAbstractItemModel。

QListView、QTreeView 和 QTableView 认为是核心类和低级类。还有一些更方便的类可以为新手 Qt 使用者提供更好的可用性——QListWidget、QTreeWidget 和 QTableWidget。我们将在本章的下一节看到使用 widget 的例子。在此之前，让我们看一个简单的 QListWidget 的运行示例：

```
1 #include <QListWidget>
2
3 int main( int argc, char** argv )
4 {
5     QApplication app( argc, argv );
6     QListWidget* listWgt{ new QListWidget };
7     return app.exec();
8 }
```

将项目添加到列表 widget 的方法之一是创建它们，我们可以通过将列表 widget 设置为其所有者来实现。下面的代码中，我们声明了三个 QListWidgetItem 对象，每个对象都有一个名称，并与前面声明的列表 widget 相关联：

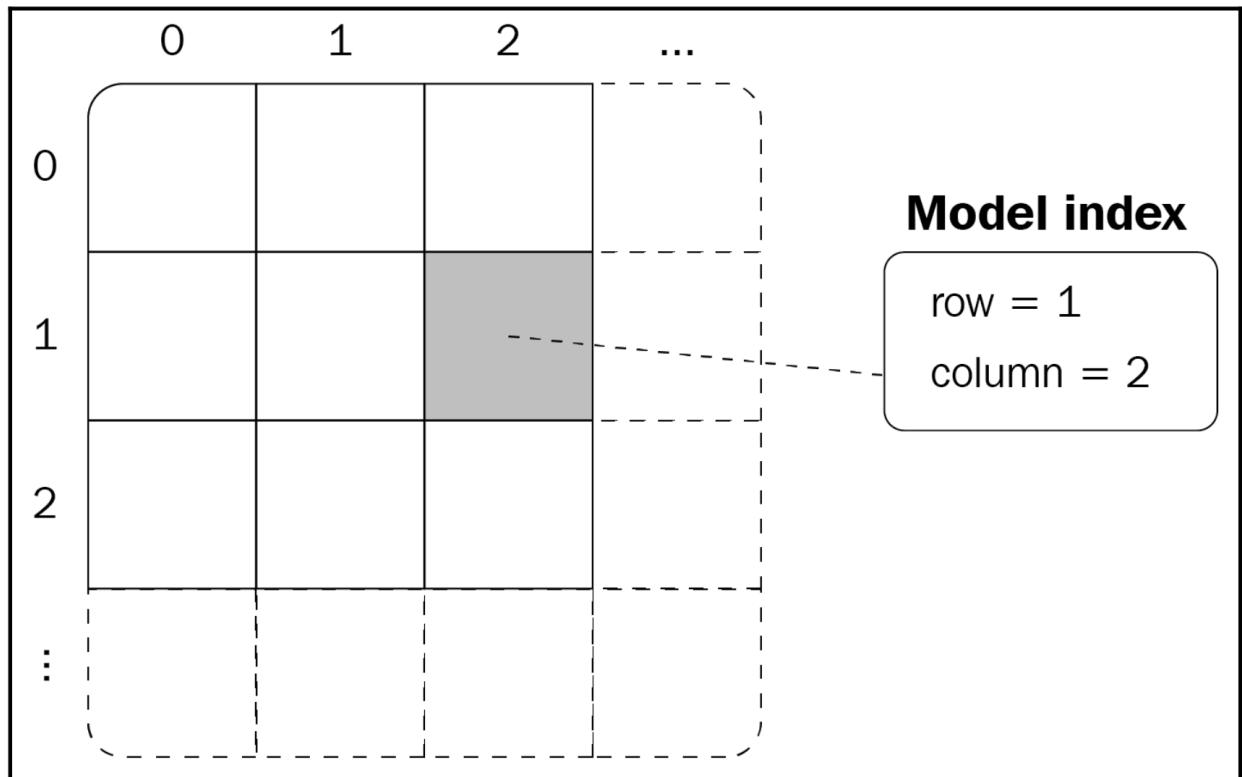
```
1 new QListWidgetItem( "Amat" , listWgt );
2 new QListWidgetItem( "Samvel" , listWgt );
3 new QListWidgetItem( "Leia" , listWgt );
```

或者，我们可以声明一个 item，然后将它插入到 listwidget 中：

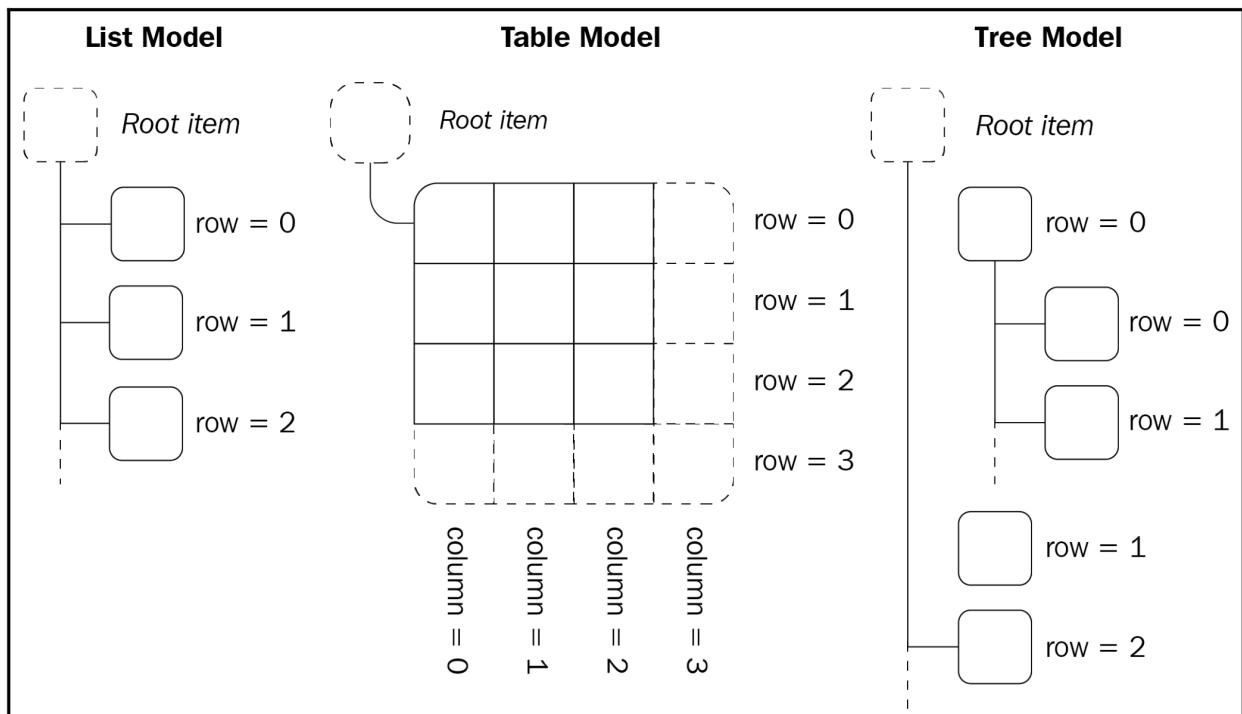
```
1 QListWidgetItem* newName{ new QListWidgetItem };
2 newName->setText( "Sveta" );
3 listWgt->insertItem( 0 , newName );
```

insertItem() 成员函数的第一个参数是要插入项目的行数。我们将 Sveta 项目放在列表的第一个位置。

既然我们已经接触了行的概念，我们应该回到模型及其索引。模型将数据封装为数据项的集合。模型中的每个项都有一个由 QModelIndex 类指定的唯一索引。这意味着模型中的每个项都可以通过关联的模型索引访问。为了获得模型索引，我们需要使用 index() 函数。下图描述了一个以表状结构组织数据的模型：



视图使用这种约定来访问模型中的数据项。但请注意，视图不受如何向用户显示数据的限制。由视图实现以方便用户的方式呈现和呈现数据。下图显示了数据在模型中的组织方式：



下面是我们如何使用模型索引访问第 1 行第 2 列的特定数据项：

```
1 QModelIndex itemAtRow1Col2 = model->index(1, 2);
```

最后，让我们声明一个视图，并为它设置一个模型，以查看模型/视图编程的实际操作：

```
1 QStringList lst;
2 lst << "item 1" << "item 2" << "item 3";
3
4 QStringListModel model;
5 model.setStringList(lst);
6
7 QListView lview;
8 lview.setModel(model);
```

在熟悉 Qt 提供的各种 widget 之后，我们将在下一节继续这个示例。

使用 QWidget

widget 是可视的 GUI 组件。如果一个 widget 没有父部件，它将被视为一个窗口，或者称为顶级 widget。在本章的前面，我们在 Qt 中创建了一个尽可能简单的窗口，如下所示代码：

```
1 #include <QtWidgets>
2
3 int main(int argc, char** argv)
4 {
5     QApplication app(argc, argv);
6
7     QWidget window;
8     window.resize(120, 100);
9     window.setWindowTitle("Mastering C++");
10    window.show();
11
12    return app.exec();
13 }
```

window 对象没有父对象。问题是，QWidget 的构造函数将另一个 QWidget 作为当前 QWidget 的父 QWidget。因此，当我们声明一个按钮并希望它成为 window 对象的子对象时，我们可以这样做：

```
1 #include <QtWidgets>
2
3 int main(int argc, char** argv)
4 {
5     QApplication app(argc, argv);
6
7     QWidget window;
8     window.resize(120, 100);
9     window.setWindowTitle("Mastering C++");
10    window.show();
11
12    QPushButton* btn = new QPushButton("Click me!", &window);
13    return app.exec();
14 }
```

观察 QPushButton 构造函数的第二个参数。我们传递了一个窗口对象的引用作为它的父对象。当父对象被销毁时，它的子对象会自动销毁。Qt 还支持许多其他 widget，让我们来看看其中的一些。

常见的 QWidget

上一节中，我们介绍了 QPushButton 类，并说明了它间接继承了 QWidget 类。为了创建一个窗口，我们使用 QWidget 类。结果是，QWidget 代表了呈现到屏幕的能力，它是所有 widget 继承的基本类。它有很多属性和函数，比如 enabled，一个 boolean 属性，如果 widget 启用，该属性为 true。要访问它，我们使用 isEnabled() 和 setEnabled() 函数。为了控制 widget 的大小，我们使用它的高度和宽度，这表示 widget 的高度和宽度。为了获得它们的值，我们分别调用 height() 和 width()。要设置新的高度和宽度，应该使用 resize() 函数，该函数接受两个参数——宽度和高度。还可以使用 setMinimumWidth()、setMinimumHeight()、setMaximumWidth() 和 setMaximumHeight() 函数来控制 widget 的最小和最大尺寸。当在布局中设置 widget 时，这可能会很有用（请参阅下一节）。除了属性和函数外，我们主要对 QWidget 的公共槽感兴趣，具体如下：

- close()：关闭 widget。
- hide()：与 setVisible(false) 相同，这个函数隐藏了 widget。
- lower() 和 raise()：在父部件的堆栈中移动 widget(到底部或顶部)。每个 widget 都可以有一个父部件。没有父窗口 widget 的 widget 将成为一个独立窗口。我们可以使用 setWindowTitle() 和 setWindowIcon() 函数为这个窗口设置一个标题和一个图标。
- style: 该属性保存 widget 的样式。为了修改它，我们使用 setStyleSheet() 函数传递描述 widget 样式的字符串。另一种方法是调用 setStyle() 函数并传递封装与样式相关属性的 QStyle 类型对象。

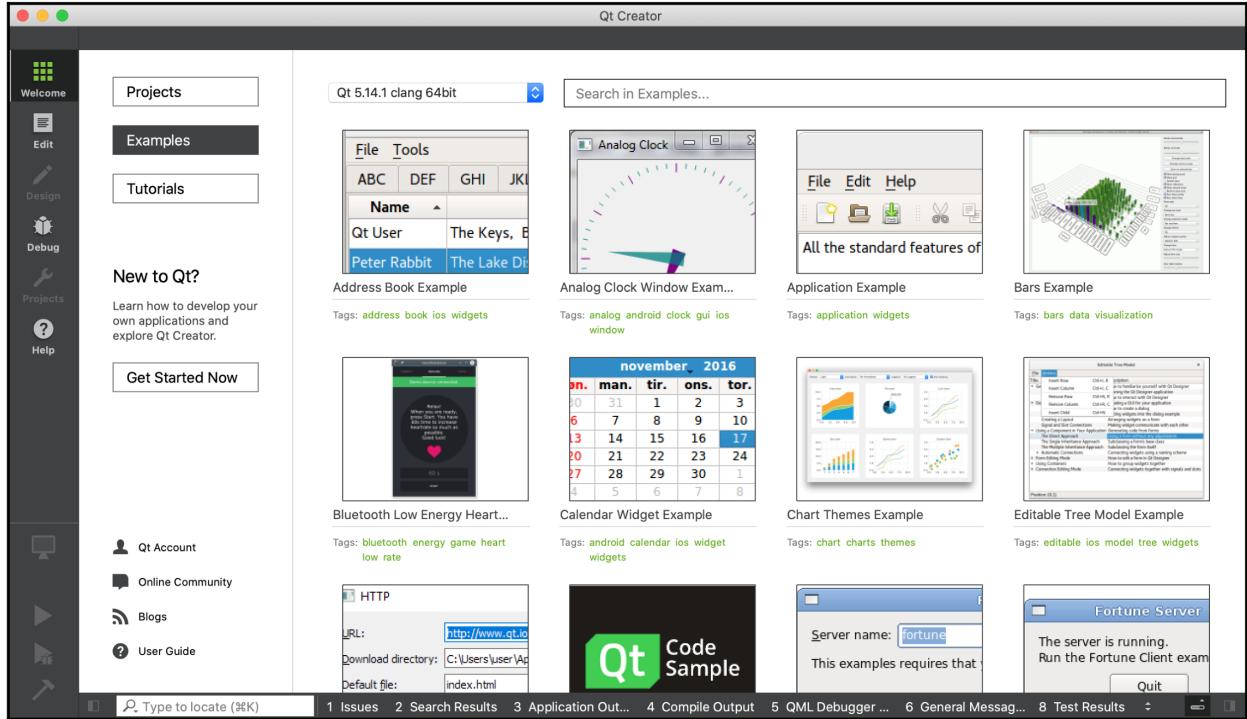
Qt 部件几乎拥有所有必要的属性，可以开箱即用。很少会遇到必须构建自己的 widget 的情况。然而，有些团队为他们的软件创建了一整套定制 widget。如果您计划为程序定制外观，那么这是很好的。例如，可以合并平面样式的 widget，这意味着您必须修改框架提供的默认 widget 的样式。自定义 widget 应该继承自 QWidget(或其任何后代)，如下所示：

```
1 class MyWidget : public QWidget  
2 { };
```

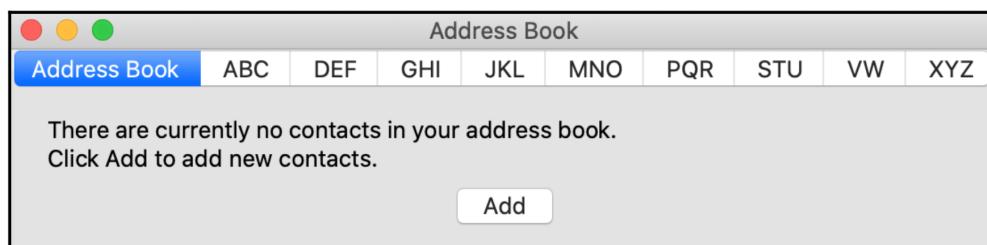
如果希望 widget 公开信号和槽，则需要在类声明的开头使用 Q_OBJECT 宏。更新后的 MyWidget 类的定义如下：

```
1 class MyWidget : public QWidget  
2 {  
3     Q_OBJECT  
4     public:  
5         // public section  
6         signals:  
7             // list of signals  
8         public slots:  
9             // list of public slots  
10    };
```

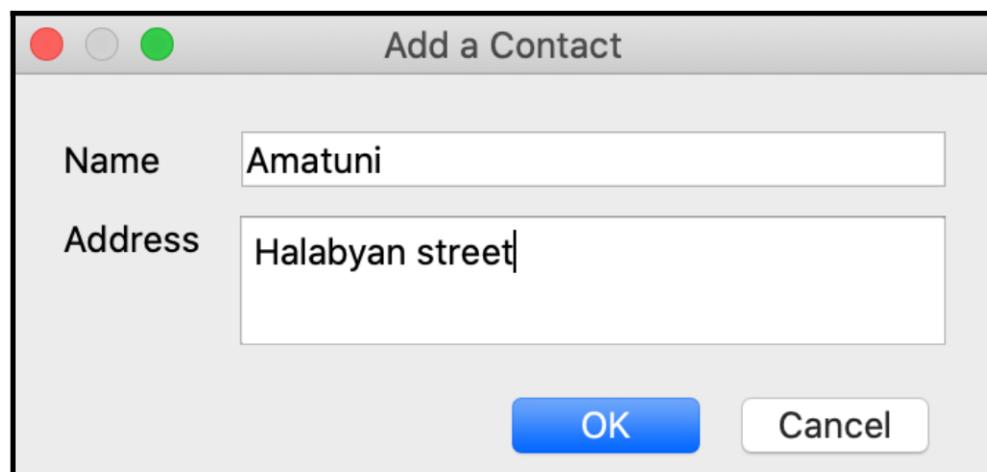
正如您可能已经猜到的，信号没有访问修饰符，而槽可以分为公共、私有和受保护的部分。正如我们前面提到的，Qt 提供了足够的开箱即用的 widget。为了研究 widget 集，Qt 提供了一组将 widget 组合在一起的示例。如果您已经安装了 Qt Creator(用于开发 Qt 应用程序的 IDE)，那么您应该能够在一次单击中浏览这些示例。下面是它在 Qt Creator 中看起来的样子：



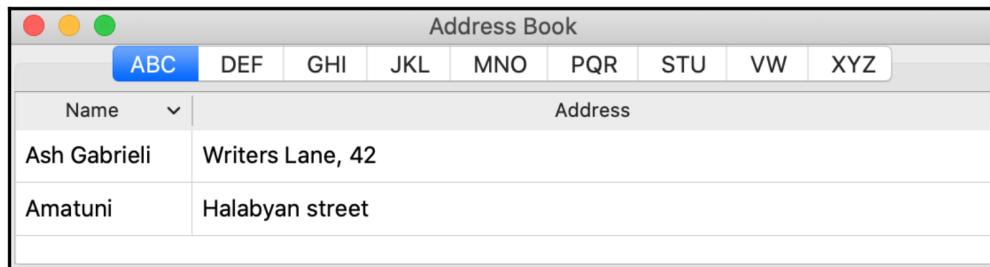
配置和运行地址簿的例子将给我们如下的接口：



点击 Add 按钮将打开一个对话框，我们可以添加一个新的条目到地址簿，如下所示：



在添加了一些条目后，主窗口以表格的形式显示条目，如下所示：



前面的屏幕截图显示了在一个应用程序中组合在一起的各种 widget。以下是我们在 GUI 应用程序开发中经常使用的一些最常见的 widget：

- QCheckBox：表示带有文本标签的复选框。
- QDateEdit：表示可用于输入日期的 widget。如果想要输入时间，也可以使用 QDateTimeEdit。
- QLabel：文本显示。也用于显示图像。
- QLineEdit：一个单行编辑框。
- QProgressBar：呈现一个垂直或水平的进度条。
- QTabWidget：作为标签 widget 的堆栈。这是许多组织者 widget 中的一个。其他一些是 QButtonGroup、QGroupBox 和 QStackedWidget。

前面的列表不是最终的，但是它给出了 Qt 功能的基本概念。我们这里使用的地址簿示例使用了许多这样的 widget。QTabWidget 表示一个组织 widget，将几个 widget 分组在一起。另一种组织 widget 的方法是使用布局。在下一节中，我们将介绍如何将 widget 组织在一起。

使用布局组合 widget

Qt 为我们提供了一个灵活而简单的平台，我们可以在这个平台上以布局的形式使用 widget 排布。这有助于我们确保 widget 内部的空间得到有效利用，并提供友好的用户体验。

让我们来看看布局管理类的基本用法。使用布局管理类的优点是，当容器 widget 改变其大小时，它们会自动调整 widget 的大小和位置。Qt 的布局类的另一个优点是，它们允许我们通过编写代码而不是使用 UI 编辑器来安排 widget。虽然 Qt Creator 提供了一种很好的手工组合 widget(在屏幕上拖放 widget) 的方法，但大多数程序员在实际编写代码来安排 widget 的外观和感觉时，会感觉更舒服。假设你也喜欢后一种方法，我们将引入以下布局类：

- QHBoxLayout
- QVBoxLayout
- QGridLayout
- QFormLayout

所有这些类都继承自 QLayout，它是几何管理的基类。QLayout 是继承自 QObject 的抽象基类。它没有继承自 QWidget，因为它与渲染没有任何关系，但它负责组织应该呈现在屏幕上的 widget。你可能不需要实现你自己的布局管理器，但如果做了，应该继承你的类从 QLayout 和提供实现以下函数：

- addItem()

- sizeHint()
- setGeometry()
- itemAt()
- takeAt()
- minimumSize()

这里列出的类足以组成几乎任何复杂性的 widget。更重要的是，我们可以将一种布局放置到另一种布局中，从而实现更灵活的 widget 组合。使用 QHBoxLayout，我们可以从左到右水平地组织 widget，如下图所示：



为了实现上述组合，我们需要使用以下代码：

```
1 QWidget *window = new QWidget;
2 QPushButton *btn1 = new QPushButton("Leia");
3 QPushButton *btn2 = new QPushButton("Patrick");
4 QPushButton *btn3 = new QPushButton("Samo");
5 QPushButton *btn4 = new QPushButton("Amat");
6
7 QHBoxLayout *layout = new QHBoxLayout;
8 layout->addWidget(btn1);
9 layout->addWidget(btn2);
10 layout->addWidget(btn3);
11 layout->addWidget(btn4);
12
13 window->setLayout(layout);
14 window->show();
```

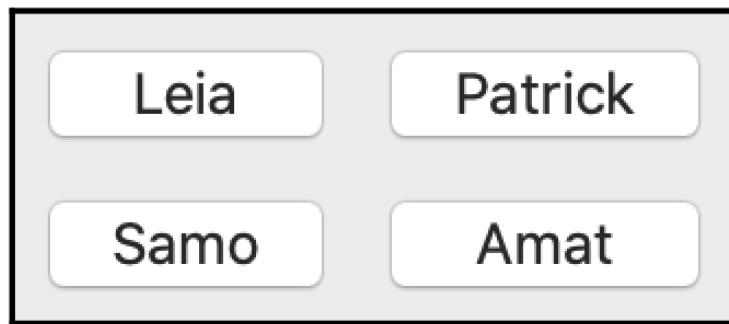
看一下我们在 widget 上调用 setLayout() 函数的那一行。每个 widget 都可以分配一个布局。如果没有容器，布局本身就不能发挥多大作用，因此我们需要将其设置为一个 widget，作为有组织的 widget(在本例中是按钮) 的容器。QHBoxLayout 继承自 QHBoxLayout，它有我们前面列出的另一个后代——QVBoxLayout。它类似于 QHBoxLayout，但垂直组织部件，如下图所示：



前面的代码中，我们唯一需要做的是用 QVBoxLayout 替换 QHBoxLayout，如下所示：

```
1 QVBoxLayout* layout = new QVBoxLayout;
```

GridLayout 允许我们将 widget 组织成网格，如下图所示：



这是相应的代码：

```
1 QGridLayout *layout = new QGridLayout;
2 layout->addWidget(btn1, 0, 0);
3 layout->addWidget(btn2, 0, 1);
4 layout->addWidget(btn3, 1, 0);
5 layout->addWidget(btn4, 1, 1);
```

最后，与 QGridLayout 类似，QFormLayout 在设计输入表单时更有帮助，因为它以两列描述性样式布局 widget。

我们可以将一个布局组合到另一个布局中。为此，我们需要使用 addItem() 函数，如下所示：

```
1 QVBoxLayout *vertical = new QVBoxLayout;
2 vertical->addWidget(btn1);
3 vertical->addWidget(btn2);
4
5 QHBoxLayout *horizontal = new QHBoxLayout;
6 horizontal->addWidget(btn3);
7 horizontal->addWidget(btn4);
8
9 vertical->addItem(horizontal);
```

布局管理器足够灵活，并且可以构建复杂的用户界面。

总结

如果你是 Qt 的新手，这一章可以作为对这个框架的一般介绍。我们讨论了 GUI 应用程序开发的基础知识，并比较了 Java 方法和 Qt 方法。使用 Qt 最大的优点之一是它支持跨平台开发。虽然 Java 做了同样的事情，但 Qt 通过生成平台本地的可执行文件超越了这一点。这使得使用 Qt 编写的应用程序比使用虚拟机编写的应用程序要快得多。

我们还讨论了 Qt 的信号和插槽作为对象间通信的灵活机制。通过使用它，可以在 GUI 应用程序中设计复杂的通信机制。我们在本章中看到了相当简单的例子，我们也可以自由地试验使用信号和槽的各种方法。我们还熟悉了常见的 QWidget 和布局管理机制。现在，已经基本了解了如何设计最复杂的 GUI 布局。可以通过应用本章介绍的技术和 QWidget 自由地实现复杂的 Qt 应用程序。在下一章中，我们将讨论当今的一个热门话题——人工智能和机器学习。

问题

1. 为什么 Qt 不需要虚拟机？
2. QApplication::exec() 函数是做什么的？
3. 如何更改顶级 widget 的标题？
4. 给定 m 模型，如何访问第 2 行和第 3 列的项？
5. 对于 QWidget，如何将其宽度更改为 400，高度更改为 450？
6. 当从 QLayout 继承来创建自己的布局管理器类时，应该实现哪些函数？
7. 如何将信号连接到槽上？

扩展阅读

- Qt5 C++ GUI Programming Cookbook by Lee Zhi Eng: <https://www.packtpub.com/application-development/qt5-c-gui-programming-cookbook-second-edition>
- Mastering Qt5 by Guillaume Lazar, Robin Penea:
<https://www.packtpub.com/web-development/mastering-qt-5-second-edition>

3 人工智能领域中的 C++

本节概述人工智能和机器学习的最新进展。我们将使用 C++ 进行机器学习，并设计一个基于对话框的搜索引擎。

本节包括以下章节：

- 第 15 章，使用 C++ 进行机器学习
- 第 16 章，实现一个交互式搜索引擎

第 15 章：使用 C++ 进行机器学习

近年来，人工智能 (AI) 和机器学习 (ML) 越来越受到人们的青睐。从简单的送餐网站，到复杂的工业机器人，人工智能已成为软件和硬件的主要功能之一。虽然在大多数情况下，这些术语都是为了让产品看起来更严肃，但一些公司正在深入研究并将人工智能融入自己的系统。

在进一步讨论之前，请考虑以下事实：本章从 C++ 开发者的角度对 ML 进行了介绍。要了解更全面的文献，请参阅本章末尾的书单。在这一章中，我们将介绍 AI 和 ML 的概念。虽然我们更倾向于有数学背景，但在这一章中我们几乎不使用任何数学。如果你正计划扩大你的技能集和潜入 ML，必须首先考虑学习数学。

除了介绍概念，本章还提供了 ML 任务的例子。我们将实现它们，并给你一个基本的想法，你应该如何研究和前进，以解决更复杂的任务。

本章中，我们将了解以下内容：

- 介绍 AI 和 ML
- ML 的分类和应用
- 设计用于计算的 C++ 类
- 神经网络的结构与实现
- 回归分析与聚类

编译器要求

g++ 编译器需要添加编译选项 `-std=c++2a` 来编译本章的代码。可以从这里获取本章的源码件：<https://github.com/PacktPublishing/Expert-CPP>

人工智能导论

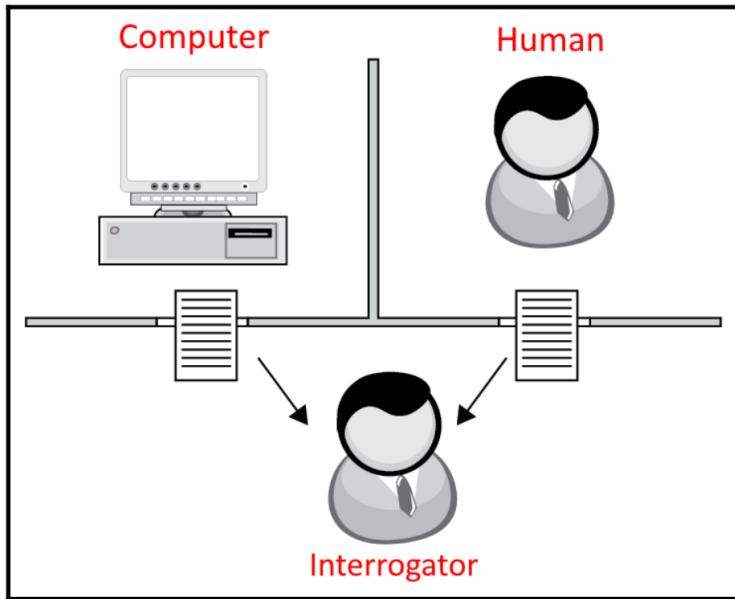
人工智能最简单的定义是机器人像人类一样行动，是由机器表现出来的智能。下面是关于智力定义的讨论。我们如何为机器定义它？我们应该与什么水平上的智能机器打交道？

如果您不熟悉太多的机器智能测试，那么你一定听说过图灵测试。这个想法是让审讯者向两个人提问，其中一个是机器，另一个是人类。如果审讯者不能明确区分这两者，那么这台机器就应该认为是智能的。



图灵测试是以艾伦·图灵的名字命名的。他在 1950 年的论文《计算机与智能》中引入了这个测试。他提议使用模仿游戏来确定机器是否像人类一样思考。

被审讯的人在一堵墙后面，这样审讯者就看不见他们了。然后审讯者向两名参与者问几个问题。下图展示了审问者是如何与人类和机器沟通的，但却看不到他们：



当你开始深入人工智能领域时，智能的定义就变得越来越模糊。可以以任何形式向机器提问：文本、音频、可视形式等。有许多东西可能永远无法在机器中实现，比如：面部表情。有时候人们通过对方的表情来理解对方的心情。你不能确定机器人是否能够理解甚至能够模仿脸上的情绪。没人教过我们在生气的时候要表现出生气的样子，没人教过我们要有情感，它们就在那里。很难说是否有一天，类似的东西是否也会出现在机器上。

说到人工智能，大多数时候我们都认为它是一个说话和行为与人类相似的机器人。但是当你作为一个程序员去分析它的时候，你会遇到很多子领域，每一个都需要花很多时间去理解。许多领域都有很多任务在进行中或处于早期研究阶段。以下是一些你可能会在职业生涯中感兴趣的人工智能子领域：

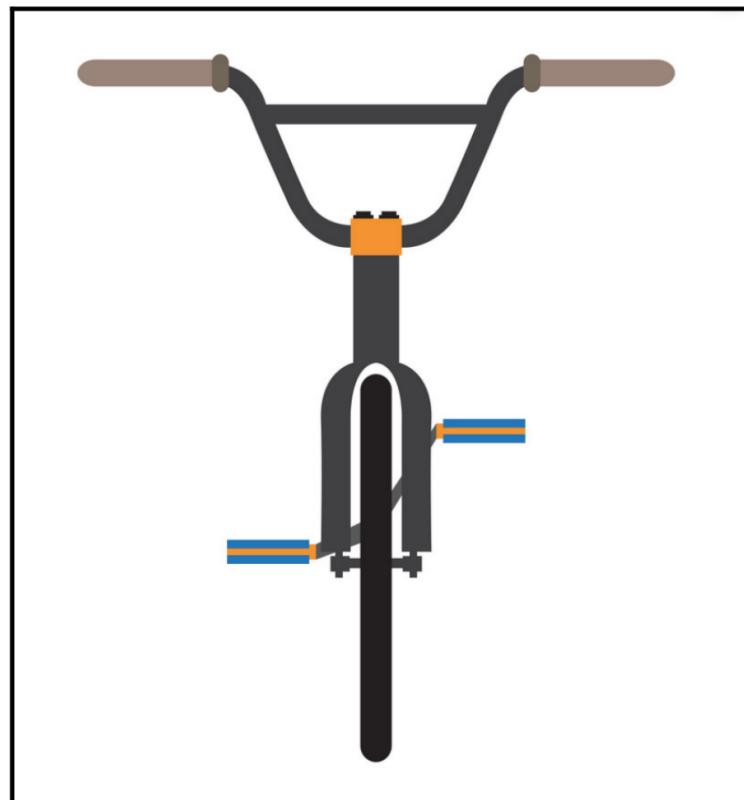
- **计算机视觉:** 设计视觉物体识别的算法，并通过分析物体的视觉表现来理解物体。人类很容易在人群中发现一张熟悉的面孔，但为机器实现类似的功能可能需要花费很多时间来获得与人类相同的精度。
- **自然语言处理 (NLP):** 机器对文本的语言分析。在很多领域都有应用，比如机器翻译。想象一下，计算机完全理解人类书写的文本，这样我们就可以告诉它该做什么，而不是花几个月的时间学习编程语言。
- **知识推理:** 这似乎是机器智能行为的目标。知识推理是关于使机器进行推理，并根据它们所拥有的信息提供解决方案，例如：通过检查医疗状况提供诊断。
- **ML:** 一个研究算法和统计模型的领域，利用机器在没有明确指令的情况下执行任务。ML 算法依赖于模式和推理，而不是直接指令。也就是说，ML 允许机器自己完成工作，无需人类参与。

计算机视觉

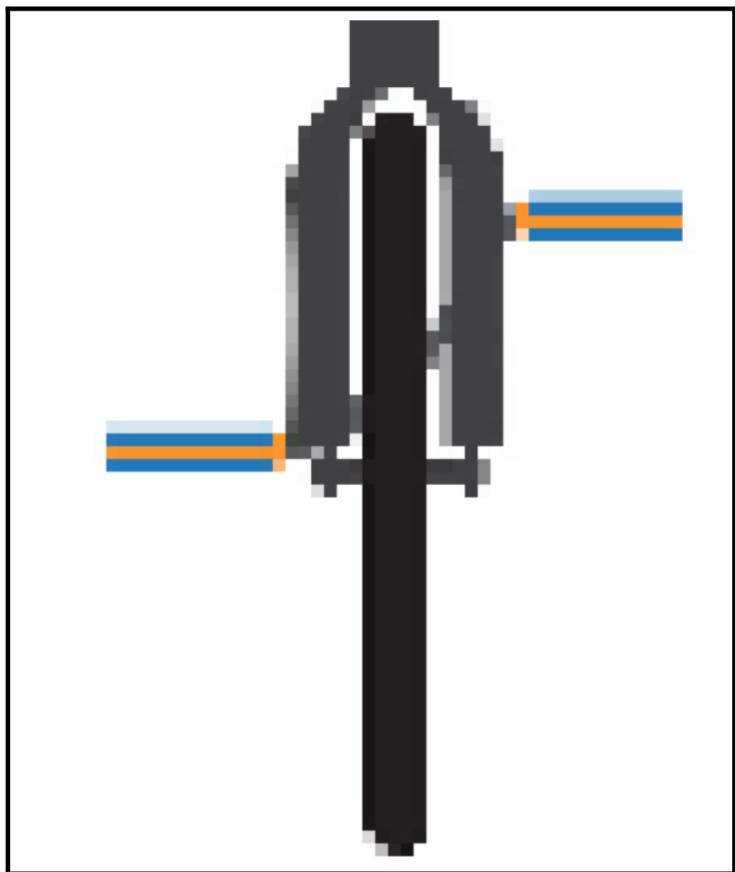
计算机视觉是一个综合性的研究领域，有很多正在进行的研究项目。它几乎涉及所有与视觉数据处理相关的内容。它在各个领域有着广泛的应用，例如：人脸识别软件处理来自城市各个摄像头的数据，以寻找和确定犯罪嫌疑人，或者光学字符识别软件从包含该数据的图像中生成文本。结合

一些增强现实 (AR) 技术，该软件能够将图像中的文本翻译成用户熟悉的语言。

这一领域的研究正日益取得进展。与人工智能系统相结合，计算机视觉是一个让机器像我们一样感知世界的领域。然而，对我们来说，一个简单的任务，在计算机视觉方面却具有挑战性，例如：当我们在图像中看到一个物体时，我们很容易看出它的尺寸。例如，下面的图片代表了一辆自行车的前视图：



即使我们没有提到这是一辆自行车，人类也不难判断出来。很明显，底部中间的黑线是自行车的前轮。很难让电脑明白这是一个轮子。电脑看到的都是像素的集合，其中一些有相同的颜色：



除了理解自行车的车轮之外，还应该推断出这辆自行车一定有另一个在图像中看不到的车轮。同样，我们可能会猜测自行车的大致大小，而计算机从图像中确定它是一个综合性的任务。也就是说，在我们看来，这个简单的事情可能会成为对计算机视觉的真正挑战。

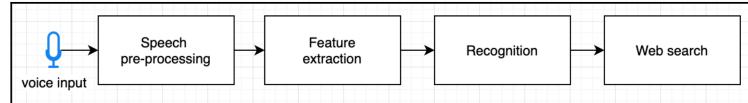


我们建议在计算机视觉任务中使用 OpenCV 库。这是一个用 C 和 C++ 编写的跨平台库。OpenCV 代表了一组针对实时计算机视觉的功能，包括但不限于面部识别、手势识别、运动理解、运动跟踪等特征。

计算机视觉的典型任务包括物体识别、识别和检测。物体识别是理解物体是前一幅图像中的一辆车。识别是对一个对象的单个实例的识别，例如：前面图像中自行车的轮子。目标检测任务可能包括从自行车的图像中发现损坏的区域。所有这些任务与 ML 算法结合起来，可能组成一个全面的软件，它可以像人类一样了解周围环境。

NLP

另一个有趣的研究领域是自然语言处理。NLP 致力于让计算机理解人类语言，更普遍的方法是自动语音识别和自然语言理解，这是目前虚拟助手的一个关键特性。今天，使用语言控制手机要求它在网络上搜索东西已经不再是魔术了。所有的过程都是由复杂的算法在语音和文本分析。下图显示了会话代理背后发生的流程的高级视图：



许多语言处理任务都与 Web 有关。搜索引擎处理用户输入，以便在 Web 上的数百万文档中进行搜索，这是自然语言处理最热的应用之一。搜索引擎设计的主要关注点之一是处理文本数据。搜索引擎不能仅仅存储所有的网站并对用户的第一个匹配的查询作出响应。在 NLP 中有许多具有复杂实现的任务。假设我们正在设计一个程序，它被输入一个文本文档，我们应该输出文档中的句子。识别一个句子的开头和结尾是一项复杂的任务。下面的句子是一个简单的例子：

I love studying C++. It's hard, but interesting.

程序将输出两个句子：

I love studying C++.

It's hard, but interesting.

就编码任务而言，我们只搜索.(点) 字符，并确保第一个单词以大写字母开头。如果一个句子具有以下形式，程序将如何表现？

I love studying C++!

由于在句子末尾有一个感叹号，我们应该重新访问我们的程序，添加另一个识别句子结尾的规则。如果句子是这样结束的呢？

It's hard, but interesting...

越来越多的规则和定义引入到具有完整功能的句子提取器中。在解决 NLP 任务时，利用 ML 将我们引向一个更明智的方向。

另一个与语言相关的任务是机器翻译，它将文档从一种语言自动翻译成另一种语言。此外，需要注意的是，建立一个全面的 NLP 系统将有利于其他领域的研究，如知识推理。

知识推理

知识推理使计算机以与人类相似的方式思考和推理。想象一下和机器对话，开头是这样的：

[Human] 你好！

[Machine] 你好！

我们可以通过编程让机器回答特定的问题或理解用户输入的复杂文本，但要根据以往的经验来判断机器的原因要困难得多。例如，以下推理是这项研究的目标之一：

[Human] 我昨天走路的时候下着雨。

[Machine] 雨中漫步很不错哦。

[Human] 下次我应该穿得暖和点。

[Machine] 你说的对。

[Human] 我好像发烧了。

[Machine] 你昨天感冒了吗？

[Human] 我想是的。

虽然人类很容易就能发现感冒和下雨之间的联系，但这个程序很难推导出这一点。它一定会把下雨和寒冷联系在一起，把发烧和感冒联系在一起。它还应该记住之前的输入，以便智能地保存。

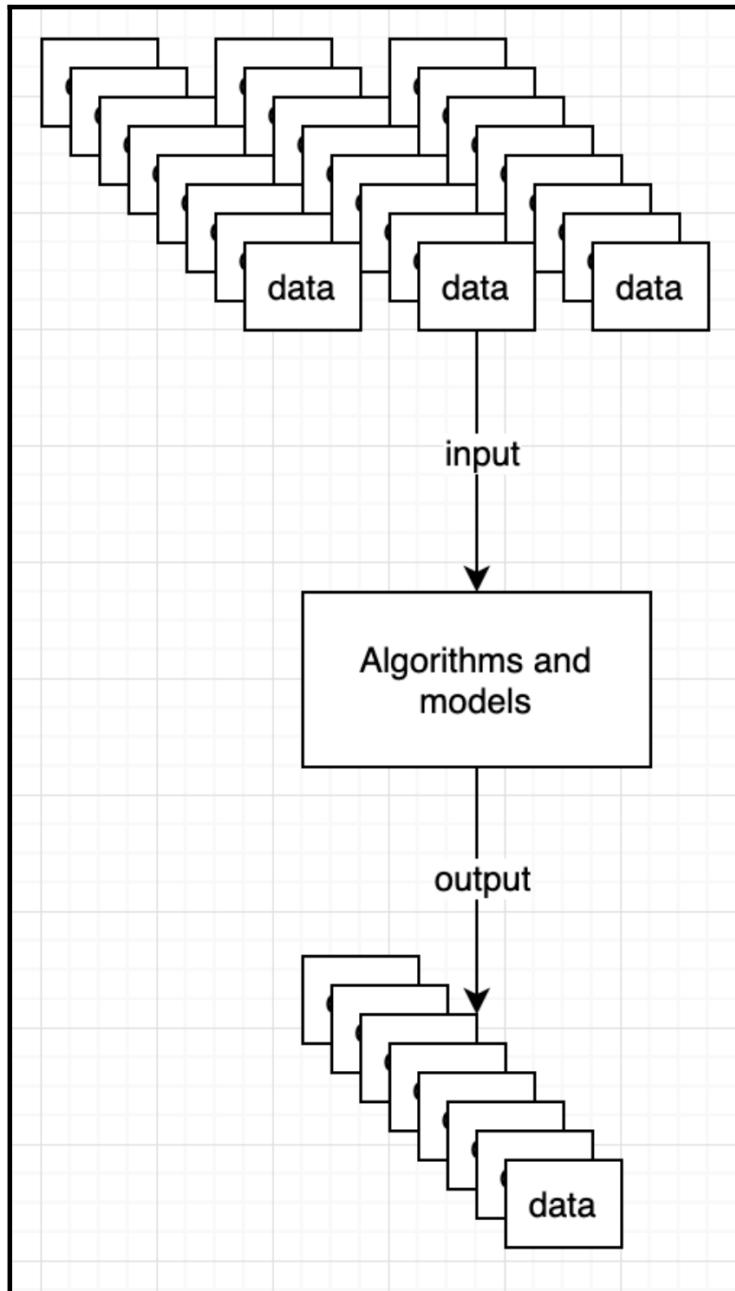
前面提到的所有研究领域都是开发者可以深入研究的领域。最后，ML 通常是所有其他领域的基础，为每个特定的应用设计算法和模型。

机器学习

ML 将我们带到了一个全新的水平，让机器像人类一样执行任务，甚至可能做得更好。与我们前面介绍的领域相比，ML 的目标是构建不需要特定指令就能完成任务的系统。在开发人工智能机器的过程中，我们应该仔细看看人类的智能。出生时，孩子不会表现出聪明的行为，但开始慢慢地熟悉周围的世界。没有记录证据表明有任何一个 1 个月大的孩子解微分方程或作曲。就像孩子学习和发现世界一样，ML 关心的是建立直接执行任务的基础模型，而不是能够学习如何去做。这就是让系统执行预先设定好的指令和让它自己去做的根本区别。

当一个孩子开始走路、拿东西、说话、问问题的时候，他们就是在一步步地获得关于世界的知识。她或他拿了一本书，尝了尝它的味道，迟早会停止把书当作可食用的东西来咀嚼。年复一年，孩子打开书页，在书中寻找图像和组成文字的小图形。又过了几年，孩子开始读这些书。多年来，大脑变得越来越复杂，神经元之间产生了越来越多的连接，成为一个有智慧的人。

想象一个系统，它有一些神奇的算法和模型。把一堆数据喂给它，它的理解能力就会越来越强，就像孩子通过视觉（眼睛看）、嗅觉、味道等形式对输入数据进行处理来认识世界一样。后来，通过发展一种提问的方式，孩子能够理解词语，并将这些词语与现实世界中的物体，甚至是无形的概念联系起来。ML 系统的作用方式几乎相同。它们对输入数据进行处理，并产生一些符合我们预期结果的输出。下图说明了这个想法：



现在让我们更深入地研究 ML。和往常一样，理解新事物的最好方法是先尝试实现它。

理解机器学习

ML 是一个大的研究领域，有很多研究正在进行中，并且正在迅速扩展。要了解 ML，首先要了解学习的本质。思考和推理是使我们人类与众不同的关键概念。ML 的核心是使系统学习和使用知识来执行任务。您可能还记得学习编程的第一步，必须学习新的概念，构建抽象，使大脑理解程序执行背后发生的事情。之后，你应该使用那些小组件构建复杂系统，这些组件包括关键词，指示，条件语句，函数，类等。

然而，ML 程序不同于我们通常创建的程序。看看下面的代码：

```

1 int calculate()
2 {
```

```
3 int a{14};  
4 int b{27};  
5 int c{a + b};  
6 return c;  
7 }
```

简单的先例程序做我们指示它做的事情。它包含了几条简单的指令，指向表示 a 和 b 的和的变量 c。我们可以修改函数，以接受用户输入如下：

```
1 int calculate(int a, int b)  
2 {  
3     int c{a + b};  
4     return c;  
5 }
```

前面的函数不会获得任何智能。无论我们调用了多少次 calculate() 函数都没有关系。输入的数字也无关紧要。该函数表示指令的集合。我们甚至可以说是硬编码指令的集合，函数永远不会修改自己的指令，使其根据输入表现出不同的行为。然而，我们可以引入一些逻辑——让它在每次接收到负数时返回 0：

```
1 int calculate(int a, int b)  
2 {  
3     if (a < 0 && b < 0) {  
4         return 0;  
5     }  
6     int c{a + b};  
7     return c;  
8 }
```

条件语句引入了最简单的决策形式，该函数根据其输入做出决策。我们可以添加越来越多的条件，这样函数就会增长，并有一个复杂的实现。然而，再多的条件语句也不能使它变得聪明，因为这不是代码自己提出的东西。这就是我们在处理程序时所面临的限制。他们会按照我们设定的程序行动。我们决定他们的行为。他们总是服从，只要我们不引入漏洞。

现在，想象一个 ML 算法在运行。假设 calculate() 函数具有某种魔力，因此它根据输入返回一个值。假设它有以下形式：

```
1 int calculate(int a, int b)  
2 {  
3     // some magic  
4     // return value  
5 }
```

现在，假设我们正在调用 calculate()，并将 2 和 4 作为它的参数传递，希望它将计算它们的和并返回 6。另外，想象一下我们可以通过某种方式告诉它结果是否符合我们的预期。一段时间后，函数的行为方式是它理解如何使用这些输入值并返回它们的和。下面我们要构建的类代表了我们理解 ML 的第一步。

设计一个可以学习的算法

下面的类代表了一台计算机器。它包含四个算术运算，并期望我们提供如何计算输入值的示例：

```
1 struct Example
2 {
3     int input1;
4     int input2;
5     int output;
6 };
7 class CalculationMachine
8 {
9 public:
10    using Examples = std::vector<Example>;
11    // pass calculation examples through the setExamples()
12    void setExamples(const Examples& examples);
13
14    // the main function of interest
15    // returns the result of the calculation
16    int calculate(int a, int b);
17
18 private:
19    // this function pointer will point to
20    // one of the arithmetic functions below
21    int (*fptr_)(int, int) = nullptr;
22
23 private:
24    // set of arithmetic functions
25    static int sum(int, int);
26    static int subtract(int, int);
27    static int multiply(int, int);
28    static int divide(int, int);
29 };
```

在使用 calculate() 函数之前，我们应该提供一个 setExamples() 函数的示例列表。下面是我们提供给 CalculationMachine 的一个例子：

```
1 3 4 7
2 2 2 4
3 5 5 10
4 4 5 9
```

每行中的前两个数字表示输入参数，第三个数字是运算的结果。setExamples() 函数是计算机如何学习使用正确的算术函数。同样的方法，我们可以从前面的例子中猜测发生了什么，同样的方法，CalculationMachine 试图找到最适合它的操作。它遍历示例并定义了当调用 calculate() 时应该使用哪些函数。实现类似如下：

```
1 void CalculationMachine::setExamples(const Examples& examples)
2 {
3     int sum_count{0};
4     int sub_count{0};
5     int mul_count{0};
```

```

6   int div_count{0};
7   for (const auto& example : Examples) {
8     if (CalculationMachine.sum(example.input1, example.input2) ==
9         example.output) {
10       ++sum_count;
11     }
12     if (CalculationMachine.subtract(example.input1, example.input2) ==
13         example.output) {
14       ++sub_count;
15     }
16     // the same for multiply() and divide()
17   }
18
19   // the function that has the maximum number of correct output results
20   // becomes the main function for called by calculate()
21   // fptr_ is assigned the winner arithmetic function
22 }
```

从前面的示例中可以看到，该函数调用所有算术函数，并将它们的返回值与示例输出进行比较。每次结果正确时，就会增加特定函数正确答案的数量。最后，将正确答案最多的函数赋值给 calculate() 函数使用的 fptr_，如下所示：

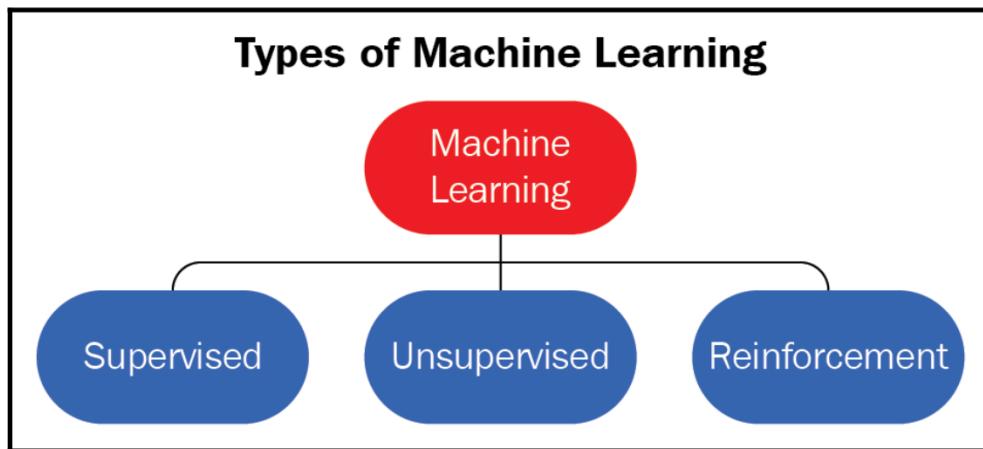
```

1 int CalculationMachine::calculate(int a, int b)
2 {
3   // fptr_ points to the sum() function
4   return fptr_(a, b);
5 }
```

我们设计了一个简单的学习算法。setExamples() 函数可以重命名为 setDataSet() 或 trainWithExample() 或类似的东西。关于 CalculationMachine 的例子的要点是，我们定义了一个模型和使用它的算法，我们可以称之为 ML。它从数据中学习，能从经验中学习。我们提供给 CalculationMachine 的例子向量中的每一个记录都可以视为经验，同时计算的性能随经验而提高。也就是说，我们提供的示例越多，它就越有信心选择正确的函数来执行任务。任务是根据两个输入参数计算值。学习的过程本身并不是任务，但学习是完成任务的关键。任务通常被描述为系统应该如何处理一个示例，而示例是功能的集合。尽管在 ML 术语中，其中每个条目都是另一个特征，矢量数据结构的选择只是一个巧合。ML 算法的基本原理之一是对系统进行训练，它可以分为监督算法和无监督算法。让我们了解一下它们的差异，然后再建立 ML 系统的各种应用。

机器学习的类别

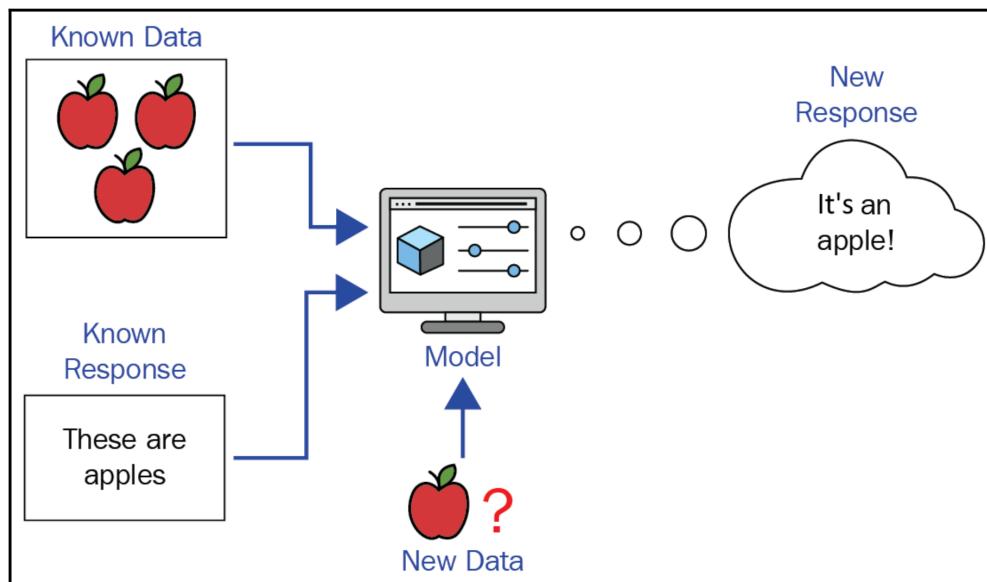
下面的图表说明了机器学习的分类：



ML 算法的分类取决于它们在学习过程中所拥有的经验。我们通常称示例集合为数据集。一些书也使用术语数据点。数据集基本上是表示对目标系统有用的任何数据的集合。可能包括一段时间内的天气测量、某个或多个公司的股票价格列表，或任何其他数据集。虽然数据集可能是未处理的或所谓的原始数据集，但也有针对每个包含的经验的附加信息的数据集。CalculationMachine 示例中，我们使用了一个原始数据集，我们已经编程让系统识别出前两个值是操作的操作数，第三个值是操作的结果。如前所述，我们将 ML 算法分为有监督的和无监督的。

监督学习算法从标记数据集学习，每个记录都包含描述数据的附加信息。calculationmachine 是一个监督学习算法的例子。监督学习也称为导师培训。类似讲师使用数据集来教授知识的系统。

有监督的学习算法将能够在学习经验后标记新的未知数据。下面的图表最好地描述了它：



监督学习算法应用的一个很好的例子是电子邮件应用中的垃圾邮件过滤器。用户将电子邮件标记为垃圾邮件或非垃圾邮件，然后系统试图在新收到的电子邮件中发现模式，以检测潜在的垃圾邮件。

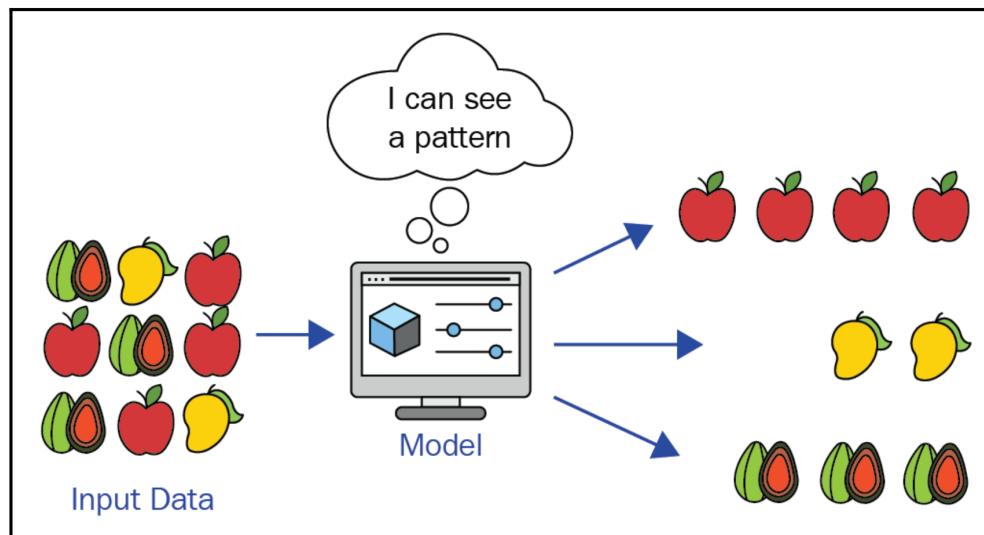
CalculationMachine 的例子是监督学习的另一个例子。我们提供了以下数据集：

1	3	4	7
2	2	2	4

3	5	5	10
4	4	5	9

我们为 CalculationMachine 编写了程序，将前两个数字作为输入参数读取，将第三个数字作为应用于输入的函数产生的输出。通过这种方式，我们提供了有关系统应该得到的结果的必要信息。

无监督学习算法甚至更复杂——处理包含一系列特征的数据集，然后找到这些特征的有用属性。无监督学习算法大多是自己定义数据集中的内容。在智能方面，无监督学习方法比监督学习算法更符合智能生物的描述。相比之下，监督学习算法试图预测哪些输入值映射到输出值，而非监督学习算法执行几个操作来发现数据集中的模式。下面的图描述了一种无监督学习算法：



无监督学习算法应用的例子是推荐系统。我们将在下一章中讨论，在那里我们设计了一个网络搜索引擎。推荐系统通过分析用户活动来推荐类似的数据，例如电影推荐。

正如你在前面的例子中看到的，还有强化学习——从错误中学习的算法。在学习系统和经验之间有一个反馈循环，因此强化学习算法与环境相互作用。可能一开始就犯很多错，然后在处理反馈后，进行自我修正以改进算法。学习过程成为任务执行的一部分。想象一下，计算机只接收输入的数字，而不接收计算的结果。对于每一次体验，它将通过应用其中一个算术运算产生一个结果，然后收到反馈。假设它减去这些数字，然后根据反馈进行自我修改，计算出总和。

机器学习的应用

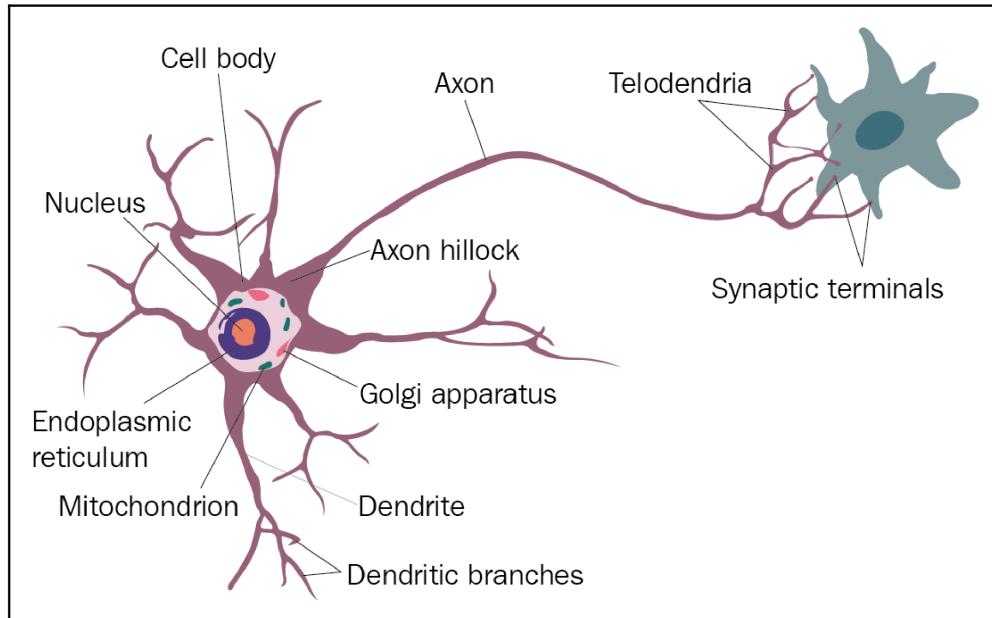
理解 ML 的分类有助于更好地将其应用于各种任务。有大量的任务可以用 ML 解决。我们已经提到分类作为 ML 算法解决的任务之一。基本上，分类就是对输入进行过滤和排序，以指定输入所属的类别的过程。用 ML 解决分类问题通常意味着它产生一个将输入映射到特定输出的函数。输出类的概率分布也是一种分类任务。分类任务最好的例子之一是物体识别。输入是一组像素值（换句话说，是一幅图像），输出是标识图像中的对象的值。想象一下，一个机器人可以识别不同种类的工具，并按命令将它们交付给工人，也就是说，一个在车库工作的机械师有一个助手机器人，能够识别螺丝刀并按命令进行工作。

更有挑战性的是在缺少输入的情况下进行分类。前面的例子中，这类似于让机器人来拧紧螺栓。当一些输入缺失时，学习算法必须与多个函数一起操作才能获得成功的结果。例如，机器人助手可能会先拿出钳子，然后再拿出螺丝刀作为正确的解决方案。

与分类类似的是回归，系统被要求预测给定输入的一个数值。不同之处在于输出的格式。回归任务的一个例子是预测股票的未来价格。ML 的这些和其他应用使它作为一个研究领域迅速发展。学习算法并不像一开始感觉的那样只是一系列条件语句。它们是基于更全面的构造，模仿人类大脑神经元及其连接。这将引导我们进入下一节，人工神经网络 (ANN) 的研究。

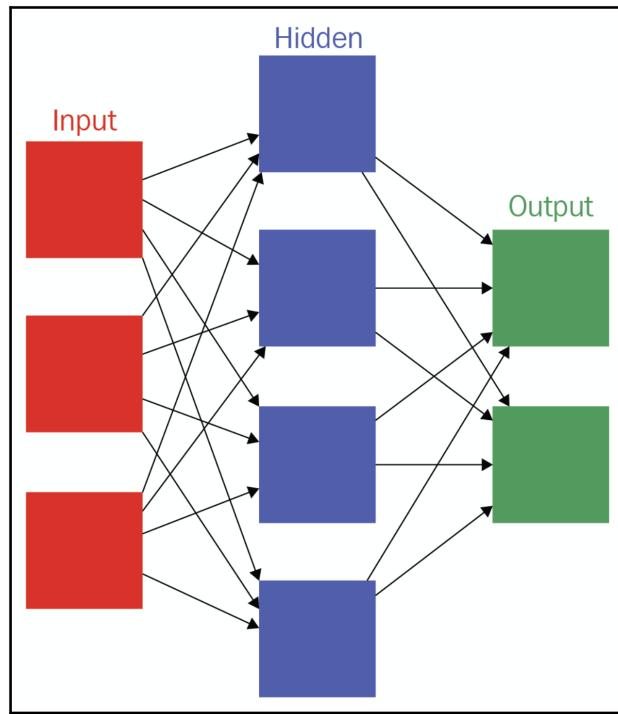
神经网络

神经网络是用来识别模式的。它们是模仿人类大脑的，或者说是大脑的神经元和它们的人工对等物——人工神经元。下面的图展示了人脑中的一个神经元：



一个神经元通过突触与其他神经元交流。神经元的基本功能是处理一部分数据并根据这些数据产生信号，神经元接受一组输入并产生输出。

下面的图表清楚地说明了为什么人工神经元与人类大脑神经元的结构相似：



神经网络是自然神经网络的一种简化模型。它代表一组相互连接的节点，每个节点代表一个神经元后的模型。每个节点连接都可以传递类似于生物大脑神经元突触的信号。神经网络是一组有助于聚类和分类的算法。从上图中可以看出，神经网络由三层组成：

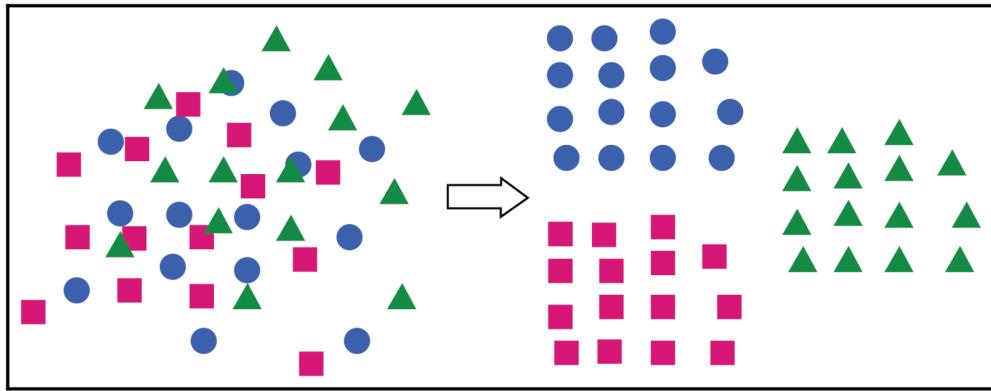
- 输入层
- 隐藏层
- 输出层

输入层表示问题的初始输入数据，例如：图像、音频或文本文件。输出层是完成任务的结果，例如：对文本内容或图像中已识别的对象进行分类。隐藏层使网络产生合理的结果。输入到输出的转换要经过隐藏层，该层要进行大量的分析、处理和修改，以产生输出。

考虑上面的图表，表明神经元可以有多个输入和输出连接。通常，每个连接都有一个权重来指定连接的重要性。上图中的神经元层告诉我们，每一层的神经元都与前一层和后一层的神经元相连。应该注意，在输入层和输出层之间可能有几个隐藏层。输入输出层的主要目的是读取外部数据并返回计算（或推导）的输出，而隐含层的目的是通过学习来适应。学习还包括调整连接和权重，以提高输出精度（这就是 ML 发挥作用的地方）。所以，如果我们创建一个包含几个隐藏层的复杂神经网络，准备学习和改进，我们就得到了一个人工智能系统。例如，我们来看一下聚类问题，然后再转向回归分析。

聚类

聚类是指对一组对象进行分组，将它们分布在类似对象的分组中，也称为聚类分析。它是一组技术和算法，目的是将相似的对象分组在一起，产生聚类。最简单的说就是将一组有颜色的物体分成由相同颜色的物体组成的不同组，如下所示：



虽然本章中讨论的是人工智能任务，但我们建议你先尝试用目前拥有的知识库来解决问题，让想想我们自己如何通过相似性来给物体分类。首先，我们应该有一个物体的基本概念。在前面的例子中，对象的形状、颜色、尺寸（2D 对象的宽度和高度）等。不需要深入了解，一个基本的对象表示可能是这样的：

```

1 struct Object
2 {
3     int color;
4     int shape;
5     int width;
6     int height;
7 };

```

我们考虑这样一个情况，即颜色和形状的值在一定范围内是已定义的。我们可以使用枚举来提高可读性。聚类分析包括对对象进行分析，并对其进行分类。首先想到的是拥有一个接受对象列表的函数。让我们来定义一个：

```

1 using objects_list = std::vector<Object>;
2 using categorized_table = std::unordered_map<int, objects_list>;
3 categorized_table clusterize(const objects_list& objects)
4 {
5     // categorization logic
6 }

```

考虑一下实现细节。需要定义聚类点，有可能是颜色，也可能是形状的类型。挑战性在于，它可能是未知的。也就是说，为了防止万一，我们将每个属性的对象分类如下：

```

1 categorized_table clusterize(const objects_list& objects)
2 {
3     categorized_table result;
4     for (const auto& obj : objects) {
5         result[obj.color].push_back(obj);
6         result[obj.shape].push_back(obj);
7     }
8     return result;
9 }

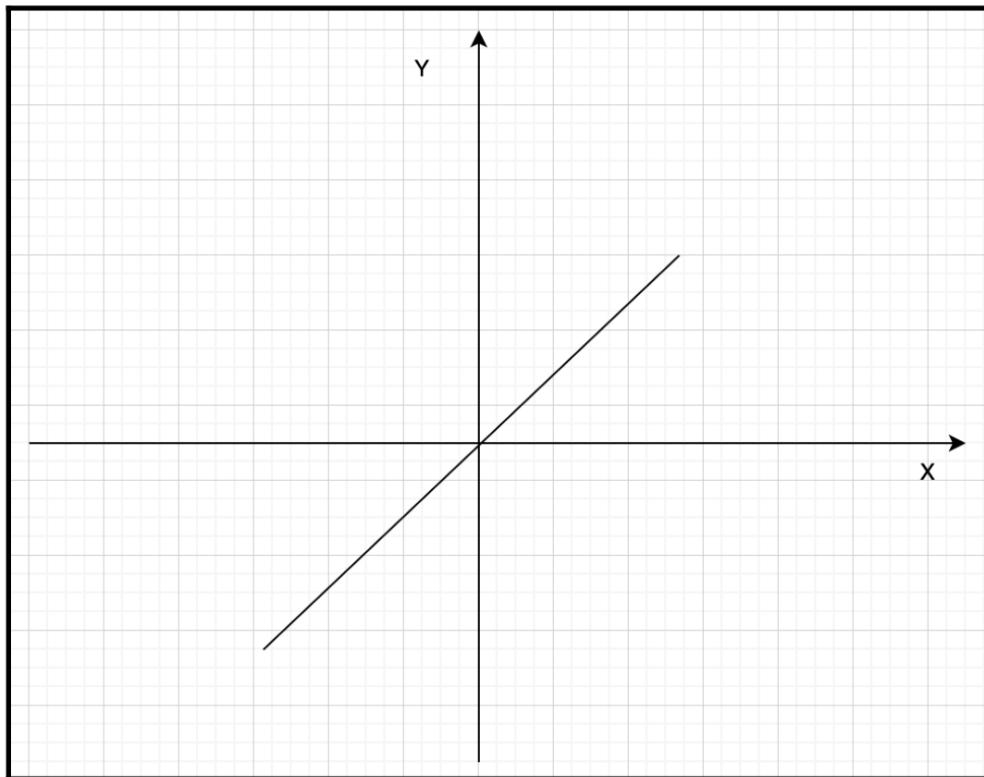
```

具有相似颜色或形状的对象分组在一个散列表中。虽然前面的代码相当简单，但它承载了根据某种相似标准，对对象进行分组的基本思想。我们在上一个示例中所做的可以描述为硬集群。一个对象要么属于一个集群，要么不属于。相反，软聚类（也称为模糊聚类）可以在一定程度上描述对象所属的聚类。

例如，`shape` 属性的对象相似性可以通过应用于对象的函数的结果来定义。也就是说，如果对象 A 的形状是正方形，而对象 B 的形状是菱形，那么该函数定义对象 A 和对象 B 是否具有相似的形状。这意味着我们应该更新前面示例中的逻辑，将对象与多个值进行比较，并将它们的形状定义为一个组。通过进一步发展这一思想，我们会得到不同的聚类策略和算法，例如 K-means 聚类。

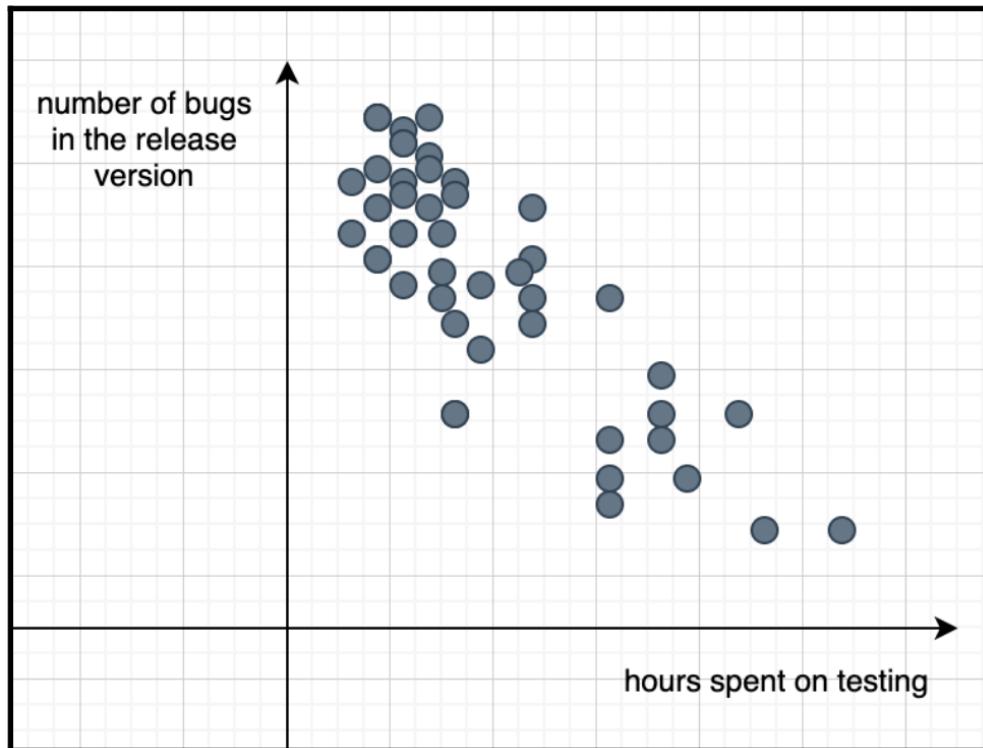
回归分析

回归分析关心的是找出一个值与另一个值的偏差。最简单的理解回归分析的方法是通过数学中的函数图。你可能还记得函数 $f(x) = y$ 的图像：



对于每一个 x 的值，函数的结果是 y 的一个固定值。回归分析有点类似于前面的图表，因为它关注的是找出变量之间的关系。更具体地说，它估计一个因变量和几个自变量之间的关系。因变量也称为结果，自变量也称为特征。功能的数量可能是一个。

最常见的回归分析形式是线性回归。它看起来与前面的图表相似。下面的例子代表了花费在测试程序上的时间和在发布版本中发现的 bug 数量之间的关系：



有两种类型的回归：负回归是如上图所示的，随着自变量的减少而因变量的增加。另一方面，正回归对自变量的值递增。

ML 中的回归分析是一种预测方法。可以开发一个程序，根据因变量的值来预测结果。ML 是一个大领域，涉及的主题非常广泛。尽管开发者倾向于尽可能少地使用数学，但在 ML 领域是不行的。你仍然需要掌握一些数学科目，以充分利用 ML。回归分析强烈依赖于统计学。

C++ 和机器学习

ML 更多的是关于数学而不是编程，这已经不再是一个秘密了。计算机科学起源于数学，在早期，计算机科学家首先是数学家。你可能熟悉几个著名的科学家，包括艾伦·图灵，约翰·冯·诺伊曼，克劳德·香农，诺贝尔特·维纳，尼克劳斯·沃斯，唐纳德·克努特还有很多人。他们都是数学家，对技术有着特殊的热爱。在计算机的发展过程中，编程成为了一个对新来者更友好的领域。过去的二三十年里，计算机开发者在开发有用的程序之前不再被迫要学习数学。语言发展成为越来越多的高级工具，几乎每个人都可以编写代码。

有很多框架可以让程序员的工作变得更容易。现在，掌握一些框架或高级编程语言并创建一个新程序需要几周的时间。然而，程序往往会重复。现在构建东西并不难，因为有很多模式和最佳实践在这一过程中帮助我们。数学的作用已经不再是必须，越来越多的人成为程序员，甚至不需要使用数学。这其实不是问题，这更像是技术发展的自然过程。最终，科技的目的是让人类生活得更舒适，工程师也是如此。虽然在 20 世纪 60 年代，NASA 的工程师们使用计算机进行计算，但那不是我们今天所知道的“计算机”。他们都是真实存在的人，他们有一种特殊的能力叫做计算机，“计算机”意味着他们在数学和解方程方面要比其他人快得多。

现在我们是计算机科学新时代的一部分，数学又回来了。ML 工程师现在使用数学的方式，就像数学家在 70 年代或 80 年代使用编程语言一样。现在，仅仅了解一种编程语言或框架来设计新

的算法或将 ML 合并到应用程序中是不够的。你也应该至少在一些数学的子领域，如线性代数，统计和概率论。

几乎同样的逻辑也适用于 C++。现代语言提供了广泛的开箱即用的功能，而 C++ 开发人员仍在努力设计具有手动内存管理的完美程序。如果你对 ML 领域做一些快速研究，你会发现大多数库或示例都在使用 Python。首先，这可能被视为 ML 任务中使用的默认语言。然而，ML 的工程师们开始触及进化的一个新门槛——性能。这并不新鲜，许多工具仍然在性能敏感的部分使用 C++。游戏开发、操作系统、关键任务系统和许多其他基本领域都使用 C++(和 C) 作为标准。现在是 C++ 征服新领域的时候了。我们给读者的最好建议是同时学习 ML 和 C++，因为对于 ML 工程师来说，可以结合 C++ 以获得最好的性能在实际应用中变得越来越关键。

总结

介绍了 ML 的分类和应用。它是一个快速发展的研究领域，在构建智能系统方面有着众多的应用。我们将 ML 分为有监督、无监督和强化学习算法。每个类别都有自己的应用，如：分类，聚类，回归，和机器翻译。

我们实现了一个简单的学习算法，它基于作为输入的经验定义了一个计算函数。我们称它为用来训练系统的数据集。使用数据集（称为经验）训练是 ML 系统的关键属性之一。

最后，我们介绍并讨论了人工神经网络在模式识别中的应用。ML 和神经网络在解决问题时是密切相关的。本章为您提供了该领域的必要介绍，以及几个任务示例，以便您可以花一些时间深入该主题。这将帮助您了解 AI 和 ML 的一般概念，因为在现实世界的应用程序开发中，工程师越来越需要 AI 和 ML。在下一章，我们将学习如何实现一个基于对话框的搜索引擎。

问题

1. ML 是什么？
2. 有监督和无监督学习算法之间有什么区别？
3. 给出一些 ML 应用的例子。
4. 在用一组不同的经验训练 CalculationMachine 类后，您将如何修改它来改变它的行为？
5. 神经网络的目的是什么？

扩展阅读

- Artificial Intelligence and Machine Learning Fundamentals, at <https://www.packtpub.com/big-data-and-business-intelligence/artificial-intelligence-and-machine-learning-fundamentals>
- Machine Learning Fundamentals, at <https://www.packtpub.com/big-data-and-business-intelligence/machine-learning-fundamentals>
- Hands-On Machine Learning for Algorithmic Trading, at <https://www.packtpub.com/big-data-and-business-intelligence/hands-machine-learning-algorithmic-trading>

第 16 章：实现一个交互式搜索引擎

已经到了这本书的最后一个章节了！我们学习了 C++ 应用程序开发的基础知识，并讨论了架构和设计实际的应用程序。我们还深入研究了数据结构和算法，这是高效编程的核心。现在是时候利用所有这些技能来设计复杂的软件了——搜索引擎。

随着互联网的普及，搜索引擎已经成为最受欢迎的产品。大多数用户通过搜索引擎开始他们的网络旅程。各种 Web 搜索服务，如谷歌、百度、Yandex 等，接收大量的流量，每天服务数万亿的请求。搜索引擎处理每个请求的时间不到一秒。尽管它们维护了数千台服务器来处理负载，但它们高效处理的核心是数据结构和算法、数据架构策略和缓存。

设计一个高效的搜索系统的问题不仅仅出现在网络搜索引擎中。本地数据库、客户关系管理 (CRM) 系统、会计软件等都需要健壮的搜索功能。在本章中，我们将发现搜索引擎的基本原理，并讨论用于构建快速搜索引擎的算法和数据结构。您将了解 Web 搜索引擎通常如何工作，并满足在需要高处理能力的项目中使用的新数据结构。你也将建立自己的搜索引擎，与现有大佬们竞争。

本章中，我们将了解以下内容：

- 了解搜索引擎的结构
- 理解并设计用于将关键字映射到搜索引擎中的反向索引
- 为搜索平台的用户设计和构建一个推荐引擎
- 利用知识图谱设计一个基于对话框的搜索引擎

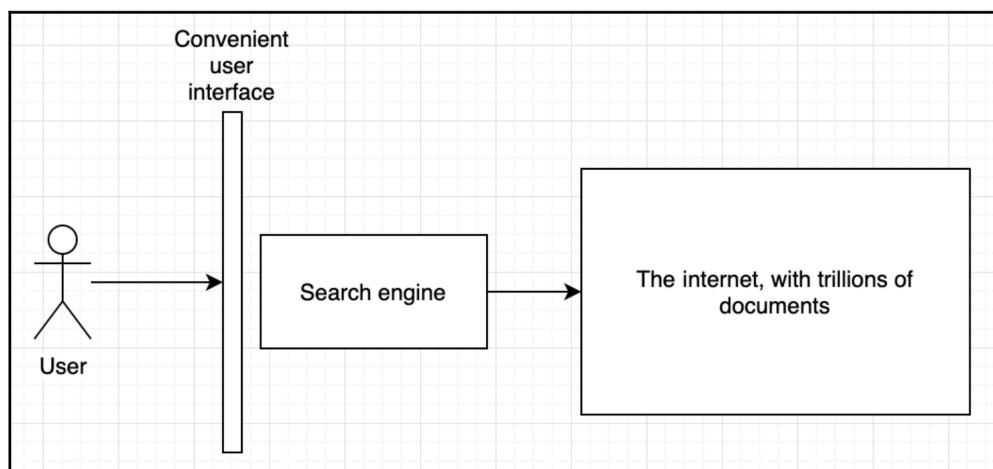
编译器要求

g++ 编译器需要添加编译选项 `-std=c++2a` 来编译本章的代码。可以从这里获取本章的源码件：<https://github.com/PacktPublishing/Expert-CPP>

了解搜索引擎的结构

想象一下，世界上有数十亿个网页。在搜索引擎界面中输入一个单词或短语，不到一秒钟就会返回一长串搜索结果。搜索引擎处理这么多网页的速度是不可思议的。它怎么能这么快找到正确的文档？为了回答这个问题，我们将做一个程序员能做的最明智的事情，设计一个我们自己的引擎。

下图展示了搜索引擎背后的基本思想：



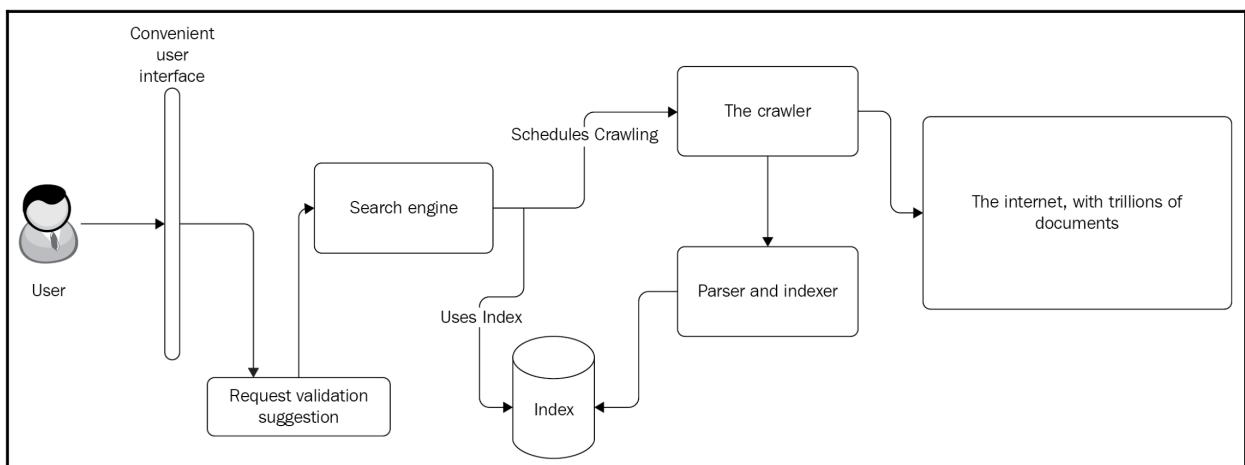
用户使用搜索引擎的用户界面输入单词。搜索引擎扫描所有文档，筛选它们，根据相关性对它们进行排序，并尽可能快地响应用户。我们的主要兴趣在于 Web 搜索引擎的实现。要想找到某样东西，就需要在数十亿份文件中搜索。

让我们尝试设计一种方法来从数十亿的文档中（为了简洁起见，我们将 Web 页面称为文档）找到 `Hello, world!`。扫描每一份文件寻找这个短语会花费大量的时间。如果我们认为每个文档至少有 500 个单词，那么搜索一个特定的单词或单词的组合将花费大量时间。事先扫描所有文件会更实际些。这个扫描过程包括为文档中出现的每个单词建立索引，并将信息存储在数据库中，这也称为索引文档。当用户输入一个短语时，搜索引擎将在其数据库中查找该词，并以满足查询的文档链接作出回应。

搜索文档之前，引擎应该验证用户的输入。用户在短语中出现拼写错误并不罕见。除了拼写错误，如果引擎自动完成单词和短语，用户体验会更好。例如，当用户输入 `hello` 时，引擎可能会建议搜索短语 `hello, world!`。一些搜索引擎跟踪用户，存储有关他们最近搜索的信息、他们用来发出请求的设备的详细信息等，例如：如果搜索引擎知道用户的操作系统，那么用户搜索如何重启计算机将得到更好的结果。如果是 Linux 发行版，搜索引擎将对搜索结果进行排序，使描述基于 Linux 的计算机重新启动的文档首先出现。

我们还应该注意网络上经常出现的新文件。后台工作可能会不断地分析网络以找到新的内容，我们称这个工作为爬虫，因为它爬虫网络和索引文档。爬虫程序下载文档以解析其内容并构建索引。已经建立索引的文档可能会更新，甚至删除。因此，另一项后台工作应该负责定期更新现有文档。您可能会遇到术语“蜘蛛”，它表示在 Web 上搜索和解析文档的任务。

下面的图表更详细地说明了搜索引擎的结构：



搜索应用广泛。想象一下最简单的搜索形式——在数组中查找单词：

```
1 using words = std::vector<std::string>;
2 words list = get_list_of_words(); // suppose the function is implemented
3
4 auto find_in_words(const std::string& term)
5 {
6     return std::find(list.begin(), list.end(), term);
7 }
```

虽然前面的例子适用于最简单的搜索引擎，但真正的问题是设计一个可伸缩的搜索引擎。我们不希望通过搜索字符串数组来满足用户请求。相反，应该努力实现一个可扩展的搜索引擎，能够搜索数百万个文档。这需要大量的思考和设计，因为从数据结构的正确选择到数据处理的高效算法，一切都很重要。现在让我们更详细地讨论搜索引擎的组件。我们将结合前面章节学到的所有技巧来设计一个好的搜索引擎。

提供方便的用户界面

投入时间和资源来构建一个细粒度的用户界面是至关重要的，它将提供惊人的用户体验。界面越简单，使用效果越好。我们将以占据市场主导地位的谷歌为例。它在页面中心有一个简单的输入字段。用户在字段中输入他们的请求，引擎会给出一些短语：

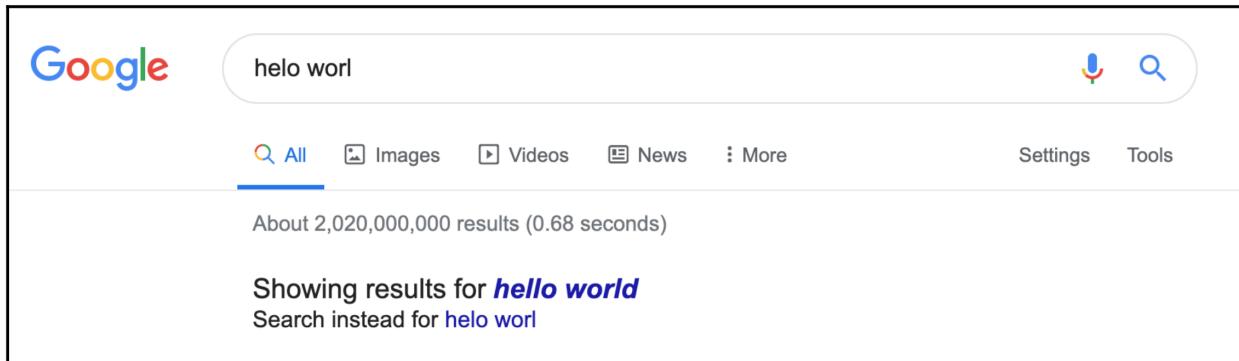


我们并不认为用户是懒惰的人，但是提供一个建议列表是很有帮助的，因为有时候用户并不知道他们想要的确切术语。让我们专注于建议列表的结构和实施。毕竟，我们感兴趣的是解决问题，而不是设计良好的用户界面。本章不讨论用户界面设计，而是专注于搜索引擎的后端。然而，在继续之前，还有一件事我们应该考虑，这里实现的搜索引擎是基于对话框的。用户查询引擎，并可以从几个答案中进行选择，以缩小结果列表。例如，假设用户查询一台计算机，搜索引擎询问一台台式机或笔记本电脑？这大大减少了搜索结果，为用户提供了更好的结果。我们将使用决策树来实现这一点。在此之前，让我们先了解一下搜索引擎的复杂性。

首先是输入标记化的问题。这与文档解析和搜索短语分析有关。您可能会构建一个很好的查询解析器，它会因为用户在查询中犯了错误而中断。让我们看看处理模糊查询的两种方法。

处理查询中的拼写错误

用户在输入时出现拼写错误的情况并不少见。虽然这看起来是一件简单的事情，但对于搜索引擎设计者来说却是一个真正的问题。如果用户输入的是 `heleo world` 而不是 `hello world`，那么搜索数百万个文档可能会出现意想不到的错误结果。您可能熟悉搜索引擎提供的自动建议，例如：当我们输入错误时，谷歌的搜索界面是这样的：



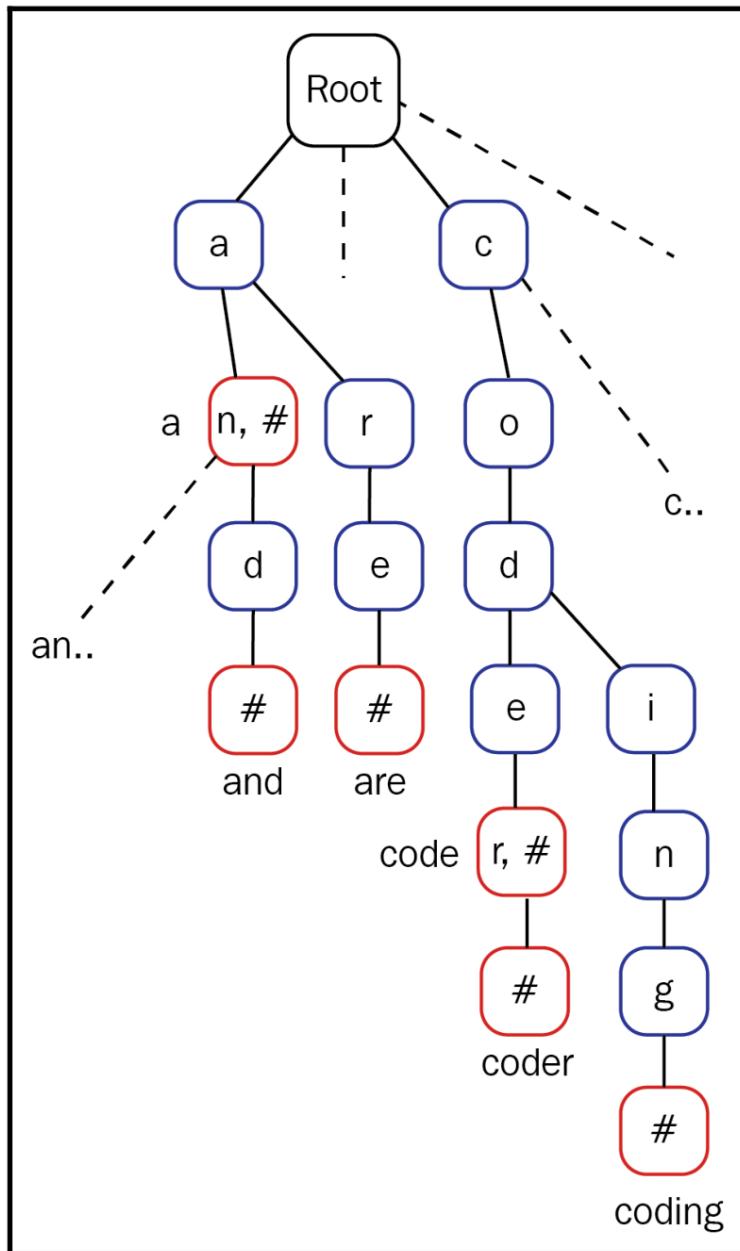
请注意截图底部的两行。其中之一是显示 `hello world` 的结果，这表明搜索引擎假设用户输入的查询有拼写错误，并主动显示正确的查询结果。然而，仍然有可能用户确实想要搜索他们输入的确切单词。因此，用户体验提供了下一行作为搜索 `heleo world` 的结果。

因此，在构建搜索引擎时，我们需要解决几个问题，首先是用户请求。首先，我们需要为用户提供一个简单的输入文本的界面。界面还应该与它们交互，以提供更好的结果，这包括基于部分输入的单词提供建议。使搜索引擎与用户交互是我们将在本章讨论的用户界面的另一个改进。

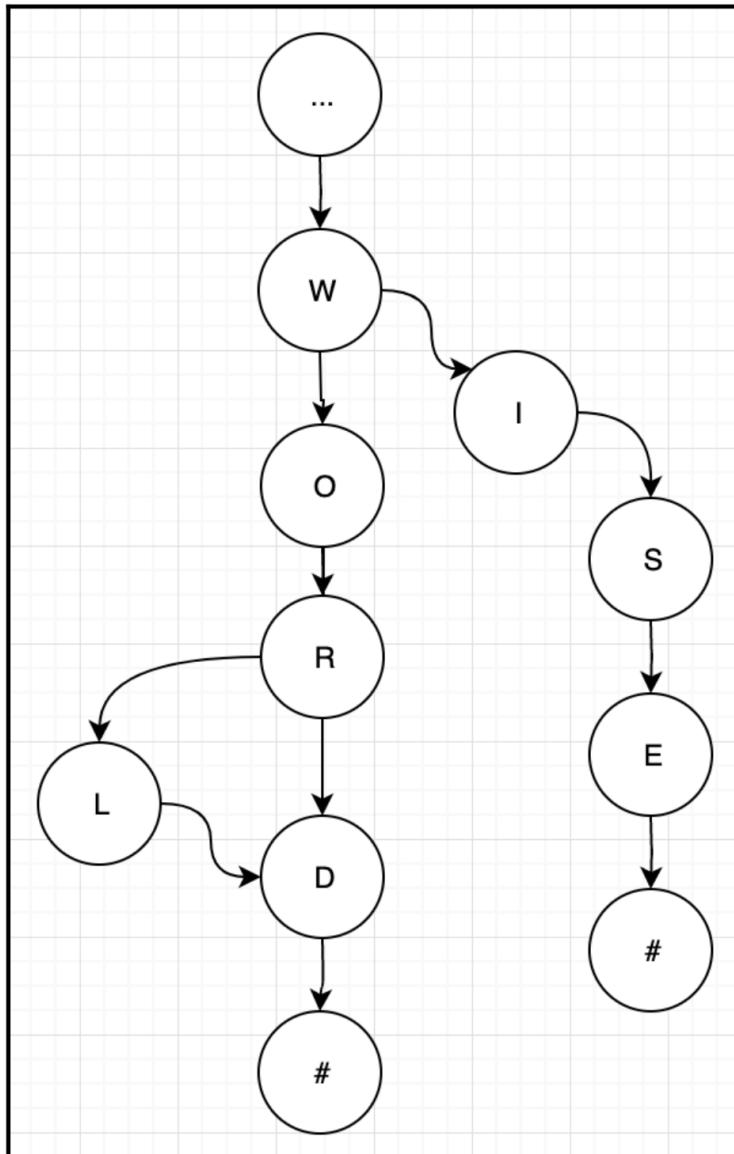
接下来是检查拼写错误或不完整的单词，这不是一项容易的任务。在字典中保存所有单词的列表并比较用户输入的单词可能需要一段时间。为了解决这个问题，必须使用特定的数据结构和算法。例如，在检查用户查询中的拼写错误时，查找单词之间的 Levenshtein 距离可能会有帮助。Levenshtein 距离是一个单词中应该添加、删除或替换的字符数，以使它等于另一个字符。例如，单词 `world` 和 `word` 之间的 Levenshtein 距离是 1，因为从 `world` 中删除字母 `d` 或将字母 `d` 添加到 `word` 中会使这两个单词相等。单词编码和坐下之间的距离是 4，因为以下四次编辑将一个单词变为另一个：

- `coding` -> `codting` (在中间插入 `t`)
- `codting` -> `cotting` (用 `t` 替换 `d`)
- `cotting` -> `citting` (用 `i` 替换 `o`)
- `citting` -> `sitting` (用 `s` 替换 `c`)

如果我们将每个用户的输入与数万个单词进行比较，找出最接近的单词，那么处理过程将需要多长时间。另一种方法是使用一个大的 `try`(数据结构) 预先发现可能的拼写错误。一个 `try` 是一个有序的搜索树，其中键是字符串。下面的图表代表了一次尝试：



每个路径代表一个有效的字。例如，a 节点指向 n 和 r 节点。注意 n 后面的 #。它告诉我们，到这个节点的路径代表一个单词 an。但是，它继续指向 d, d 后面跟着另一个 #，这意味着到这个节点的路径表示另一个单词和。同样的逻辑也适用于其余的试验。例如，想象一下 world 这个单词的部分：



当引擎遇到 world 时，它将经历之前的尝试。w 很好，o 也很好，直到单词 l 中的倒数第二个字符为止，其他的一切都很好。在前面的图中，l 之后没有终端节点，只有 d。这意味着我们确定不存在 world 这样的单词，所以它可能是 world。为了提供好的建议和检查拼写错误，我们应该有一个完整的用户语言词汇词典。当您计划支持多种语言时，这将变得更加困难。然而，虽然收集和存储字典是一项容易的任务，但更难的任务是收集 Web 上的所有文档并相应地存储它们以执行快速搜索。收集和解析网站以构建搜索引擎数据库（如前所述）的搜索引擎的工具、程序或模块称为爬虫程序。在深入了解存储这些网站页面的方式之前，让我们先快速了解一下爬虫程序的功能。

爬虫

每次用户输入查询时搜索数百万个文档是不现实的。想象一个搜索引擎，在用户点击系统 UI 上的搜索按钮之后，它解析网站以搜索用户查询。那将永远无法完成。搜索引擎对网站的每个请求都需要一些时间。即使它小于 1 毫秒（0.001 秒），在用户等待查询完成时，分析和解析所有网站也需要很长的时间。为了让事情更清楚，让我们假设访问和搜索一个网站大约需要 0.5 毫秒（即使这样，那也是不合理的快）。这意味着搜索 100 万个网站大约需要 8 分钟。现在想象一下你打开一个

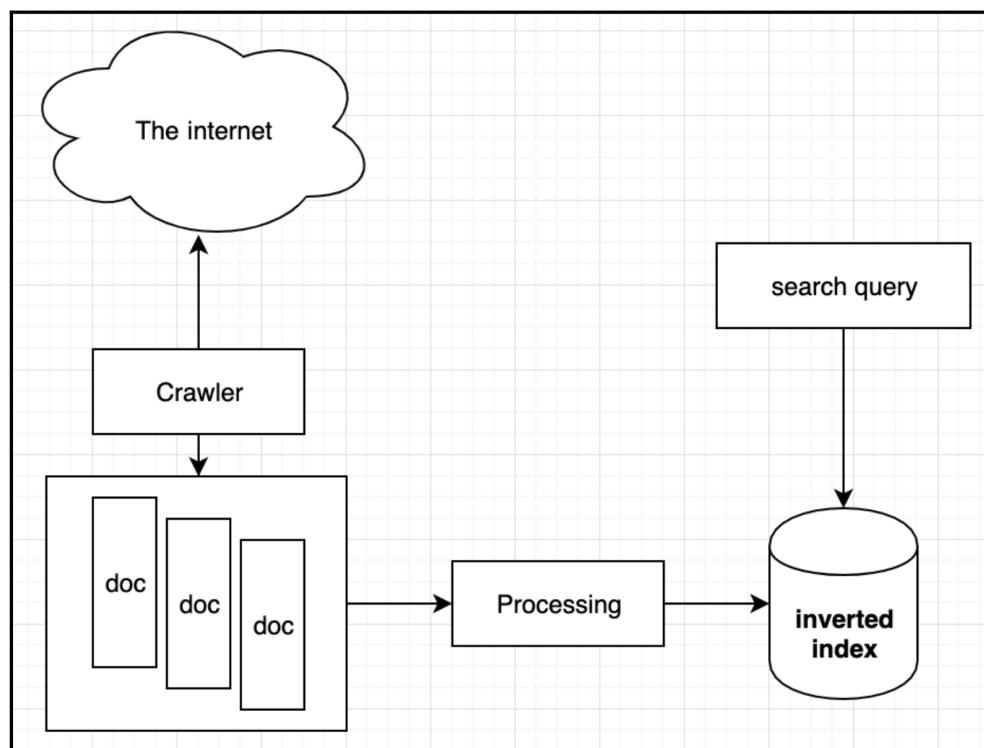
谷歌搜索并进行查询——你会等待 8 分钟吗？

正确的方法是将所有信息存储在数据库中，以便搜索引擎有效地访问。爬虫程序下载网站页面并将其存储为临时文档，直到进行解析和建立索引。复杂的爬虫程序还会对文档进行解析，以使它们保持对索引器更方便的格式。这里重要的一点是，下载一个网页不是一个只发生一次的操作。Web 页面的内容可能会更新。同时，在此期间可能会出现新的页面。因此，搜索引擎必须保持其数据库的更新。为了实现这一点，它安排爬虫程序定期下载页面。聪明的爬虫程序可能会在将内容传递给索引器之前比较内容的不同之处。

通常，爬虫程序作为多线程应用程序工作。开发人员应该注意使爬行尽可能快，因为保持数十亿文档的最新不是一项容易的任务。正如我们已经提到的，搜索引擎并不直接搜索文档。它在所谓的索引文件中执行搜索。虽然爬行是一项有趣的编码任务，但在本章中，我们将关注于索引。下一节将介绍搜索引擎中的索引。

索引文件

搜索引擎的关键功能是索引。下面的图表显示了爬虫程序如何处理下载的文档来构建索引文件：



上面的图表中，索引显示为反向索引，用户查询被定向到反向索引。虽然我们在本章中交替使用索引和倒序索引，但反向索引是一个更准确的名称。首先，让我们看看搜索引擎的索引是什么。建立文档索引的全部原因是为了提供快速搜索功能。其思想很简单：每次爬虫程序下载文档时，搜索引擎都会处理它的内容，将其划分为引用该文档的单词。这个过程称为标记化。假设我们从维基百科下载了一个包含以下文本的文档（为了简洁起见，我们只取一部分作为例子）：

¹ In 1979, Bjarne Stroustrup, a Danish computer scientist, began work on "C with Classes", the predecessor to C++. The motivation for creating a new language

originated from Stroustrup's experience in programming for his PhD thesis.
Stroustrup found that Simula had features that were very helpful for large
software development...

搜索引擎将前面的文档分成几个单独的单词，如下所示（为了简洁起见，这里只显示前几个单词）：

```
1 In
2 1979
3 Bjarne
4 Stroustrup
5 a
6 Danish
7 computer
8 scientist
9 began
10 work
11 ...
```

在将文档划分为单词之后，引擎将为文档中的每个单词分配一个标识符（ID）。假设前面文档的 ID 为 1，下表显示了单词引用（出现在）ID 为 1 的文档中：

In	1
1979	1
Bjarne	1
Stroustrup	1
a	1
Danish	1
computer	1
scientist	1
...	

可能有几个文档包含相同的单词，所以上面的表可能看起来更像下面的表：

In	1,4,14,22
1979	1,99,455
Bjarne	1,202,1314
Stroustrup	1,1314
a	1,2,3,4,5,6,7,8,9,10,11,...
Danish	1,99,102,103
computer	1,4,5,6,24,38,...
scientist	1,38, 101, 3958, ...

下面的表表示反向索引。它将单词与爬虫程序下载的文档的 id 映射在一起。现在查找包含用户作为查询输入的单词的文档变得更快了。现在，当用户通过键入 computer 来查询引擎时，结果

将基于从索引中检索到的 ID 生成，即 1, 4, 5, 6, 24, 38 ... 在上面的例子中。索引还有助于为更复杂的查询找到结果。例如，“计算机科学家”匹配以下文件：

computer	1,4,5,6,24, 38 ,...
scientist	1,38 , 101, 3958, ...

要用包含两个术语的文档响应用户，我们应该找到引用文档的交集（参见上表中的粗体数字），例如 1 和 38。

注意，用户查询在与索引匹配之前也需要标记化。标记化通常涉及到词的规范化。如果没有规范化，计算机科学家查询将不会给出任何结果（注意查询中的大写字母）。让我们进一步了解一下。

分词文件

您可能还记得第 1 章中标记化的概念，其中我们讨论了编译器如何通过将源文件标记为更小的、不可分割的单元（称为标记）来解析源文件。搜索引擎以类似的方式解析和标记文档。

我们不会过多地讨论这个问题，但是应该考虑到，文档的处理方式意味着标记（在搜索引擎上下文中具有意义的不可分割的术语）是规范化的。例如，我们正在查看的所有单词都是小写的。因此，索引表应该如下所示：

in	1,4,14,22
1979	1,99,455
bjarne	1,202,1314
stroustrup	1,1314
a	1,2,3,4,5,6,7,8,9,10,11,...
danish	1,99,102,103
computer	1,4,5,6,24,38,...
scientist	1,38, 101, 3958, ...

作为 C++ 程序员，看到小写的 bjarne 或 stroustrup 可能会感到不舒服。然而，当我们用倒排的索引键匹配用户输入时，我们应该考虑到用户输入可能没有我们期望的形式。因此，我们需要对用户输入应用相同的规则，使其与倒排索引匹配。

接下来，注意 a，毫不夸张地说，这是每个文档中都会出现的一个词。其他的例子还有 the, an, in 等，我们把它们称为停止词。通常，搜索引擎会忽略它们，所以倒排索引会更新为以下形式：

1979	1,99,455
bjarne	1,202,1314
stroustrup	1,1314
danish	1,99,102,103
computer	1,4,5,6,24,38,...
scientist	1,38, 101, 3958, ...

您应该注意，规范化不仅仅是使单词小写。它还包括将单词变为正常形式。



将单词规范化为词根形式 (或词干) 也称为词干提取。

看看我们在本节开始时所使用的文件中的以下句子:

```
1 The motivation for creating a new language originated from Stroustrup's experience  
in programming for his PhD thesis.
```

creation、originated 和 Stroustrup 都是规范化的，所以反向索引的形式如下:

motivation	1
create	1
new	1
language	1
originate	1
stroustrup	1
experience	1
programming	1
phd	1
thesis	1

请注意，我们忽略了停止词，也没有在前面的表中包含。

标记化是创建索引的第一步。除此之外，我们可以自由地以任何使搜索更好的方式处理输入。

结果排序

相关性是搜索引擎最重要的特性之一。仅响应与用户输入匹配的文档是不够的。我们应该对它们进行排序，使最相关的文档排在前面。

一种策略是记录每个单词在文档中出现的次数。例如，描述计算机的文档可能包含单词 computer 的多次出现，如果用户搜索计算机，结果将首先显示包含最多的 computer 的文档。下面是一个索引表的例子:

computer	1{18}, 4{13}, 899{3}
map	4{9}, 1342{4}, 1343{2}
world	12{1}

花括号中的值定义了文档中每个单词出现的次数。

向用户展示搜索结果时，我们可以考虑许多因素。一些搜索引擎存储用户相关信息，以响应个性化的结果。甚至用户用于访问搜索引擎（通常是 Web 浏览器）的程序也可能改变搜索平台的结果，例如：在 Linux 操作系统上搜索重新安装操作系统的用户会得到列表顶部包含重新安装 Ubuntu 的结果，因为浏览器向搜索引擎提供了操作系统类型和版本信息。然而，考虑到隐私问题，有些搜索引擎完全不使用个性化用户数据。

文档的另一个属性是更新日期，新鲜内容总是具有更高的优先级。因此，当向用户返回一个文档列表时，我们还可以按照其内容更新的顺序对它们重新排序。考虑到文档的相关排名，我们将进入下一节，在这里我们将讨论推荐引擎。

构建推荐引擎

在前一章中，我们介绍了人工智能 (AI) 和机器学习 (ML)。推荐引擎可以视为人工智能驱动的解决方案或简单的条件语句集合。构建一个接受用户数据并返回最满足输入的选项的系统是一项复杂的任务。将 ML 合并到这样的任务中，可能很合理。

但是，您应该考虑这样一个事实：推荐引擎可能包含一组规则，在将数据输出给最终用户之前，根据这些规则对数据进行处理。推荐引擎可以在预期和意外的地方运行。例如，在 Amazon 上浏览产品时，推荐引擎会根据我们当前浏览的产品向我们推荐产品。电影数据库会根据我们之前看过的电影或分级的电影来推荐新的电影。这对许多人来说可能是意想不到的，但推荐引擎也运行在搜索引擎之后。

你可能熟悉一些电子商务平台推荐产品的方式。大多数时候，建议面板的标题是类似于购买了这个的客户，也购买了.... 回想一下我们在前一章介绍过的聚类分析。现在，如果我们试图理解这些建议是如何在幕后工作的，我们很可能会发现一些聚类算法。

让我们简单地看一看，并尝试设计一些推荐机制。让我们以一个书店网站为例。John 买了一本名为《精通 Qt5》的书，所以让我们把这些信息放在下面的表格中：

	Mastering Qt5
John	yes

接下来，约翰决定买一本 C++ 书《精通 c++ 编程》。莱娅买了一本名为《设计模式》的书。卡尔买了三本书，分别是《学习 Python》、《掌握机器学习》和《用 Python 学习机器》。表格会进行更新，现在看起来像这样：

	Mastering Qt5	Mastering C++ Programming	Design Patterns	Learning Python	Mastering Machine Learning	Machine Learning with Python
John	yes	yes	no	no	no	no
Leia	no	no	yes	no	no	no
Karl	no	no	no	yes	yes	yes

现在，让我们想象一下 Harut 访问了网站并购买了前面列出的两本书，《学习 Python》和《用 Python 学习机器》。那么向他推荐《精通 Qt5》这本书合理吗？我们不这么认为。但是我们知道他买了什么书，我们也知道另外一个用户 Karl 买了三本书，其中两本和 Harut 买的书是一样的。所以，向 Harut 推荐精通机器学习可能是合理的，告诉他买了其他两本书的顾客也买了这本书。这是一个简单的例子，说明了从高层次的角度推荐引擎是如何工作的。

使用知识图谱

现在，让我们回到搜索引擎。一个用户正在搜索一位著名的计算机科学家——比如 Donald Knuth。他们在搜索字段中输入名称，然后从所有的 Web 中获得排序的结果，以提供最好的结果。我们再来看看谷歌搜索。为了充分利用用户界面，谷歌向我们展示了一些关于搜索主题的简要信息。在本例中，它在结果页面的右侧显示了这位伟大科学家的几张图片和一些关于他的信息。这部分看起来是这样的：

Donald Knuth

American computer scientist

Donald Ervin Knuth is an American computer scientist, mathematician, and professor emeritus at Stanford University. He is the 1974 recipient of the ACM Turing Award, informally considered the Nobel Prize of computer science. He is the author of the multi-volume work *The Art of Computer Programming*. [Wikipedia](#)

Born: January 10, 1938 (age 82 years), [Milwaukee, Wisconsin, United States](#)

Spouse: [Jill Knuth](#) (m. 1961)

Education: [California Institute of Technology](#) (1960–1963), [MORE](#)

Awards: [Turing Award](#), [Kyoto Prize](#), [MORE](#)

通过这种方式，搜索引擎试图满足用户的基本需求，让他们在不访问任何网站的情况下更快地找到信息。在这种情况下，我们最感兴趣的是放在前面信息框下面的建议框。它的标题是“人们也会搜索”，它是这样的：

People also search for

[View 10+ more](#)



Robert
Sedgewick



Alan Turing



Edsger W.
Dijkstra



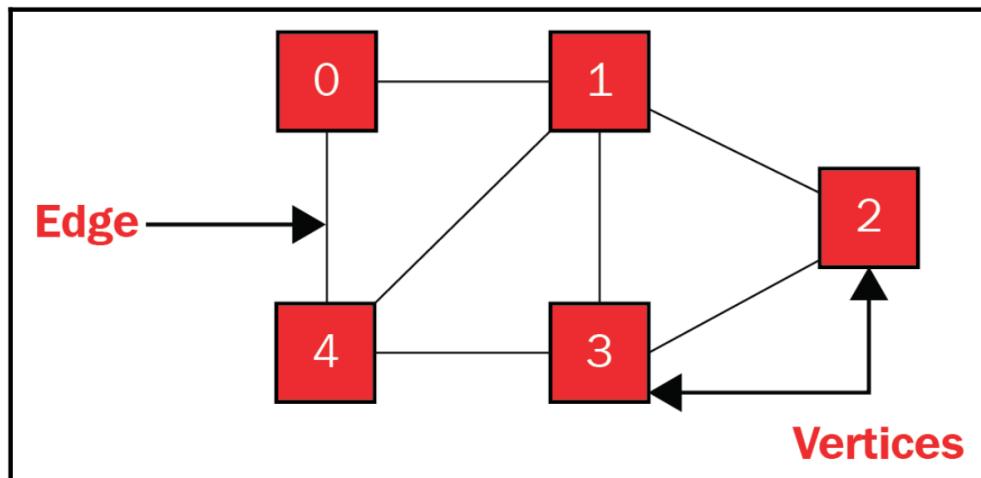
Vaughan
Pratt



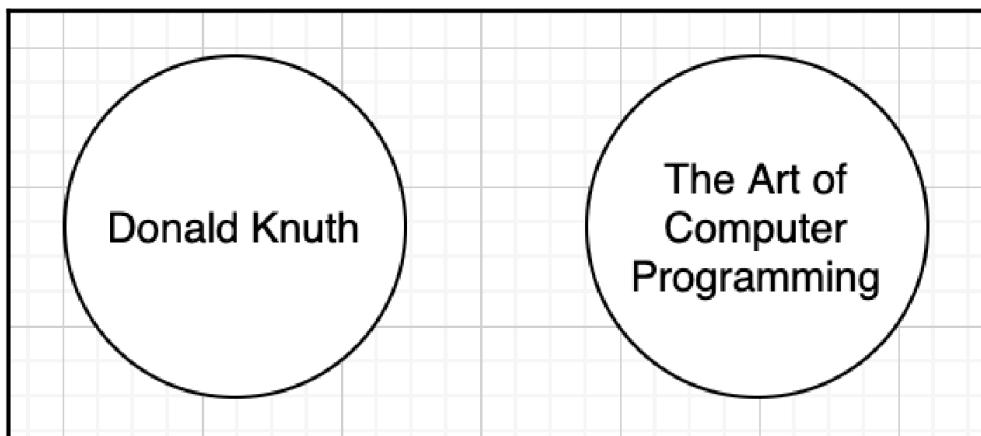
Ronald
Graham

这些是基于用户搜索 Alan · Turing 的推荐，就在他们搜 Donald · Knuth 之后。这促使推荐引擎提出了一个建议，如果有人在搜索 Donald · Knuth，他们可能也对 Alan · Turing 感兴趣。

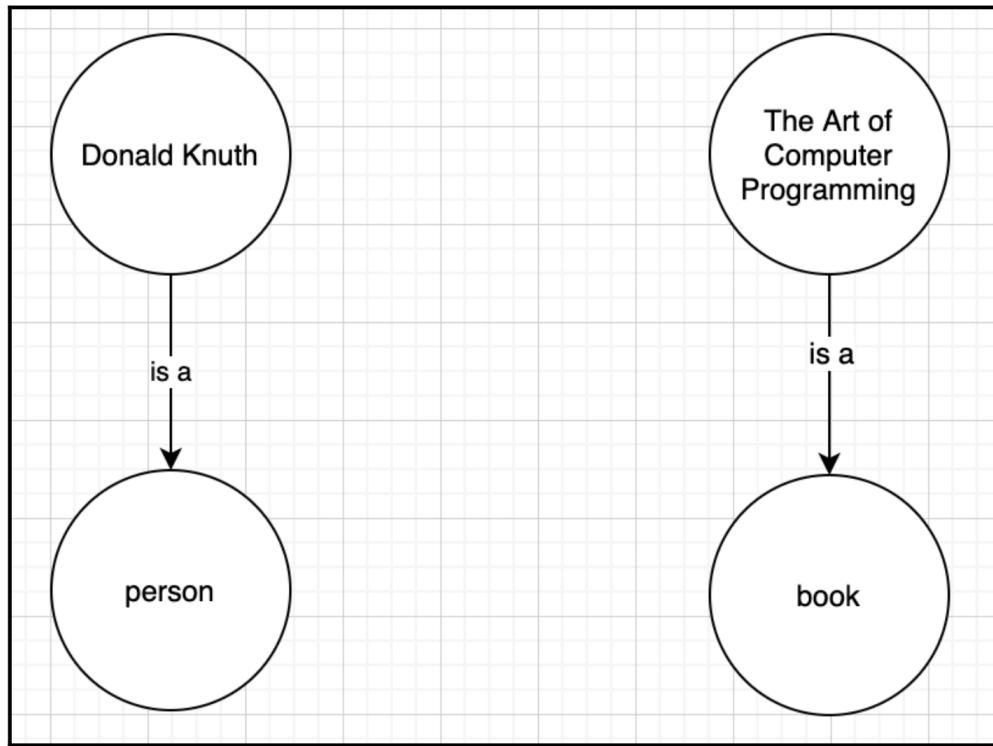
我们可以通过谷歌称之为知识图谱的东西来组织类似的建议机制。这是一个由节点组成的图，每个节点表示某个主题、人物、电影或其他可搜索的内容。图数据结构是节点和连接这些节点的边的集合，如下图所示：



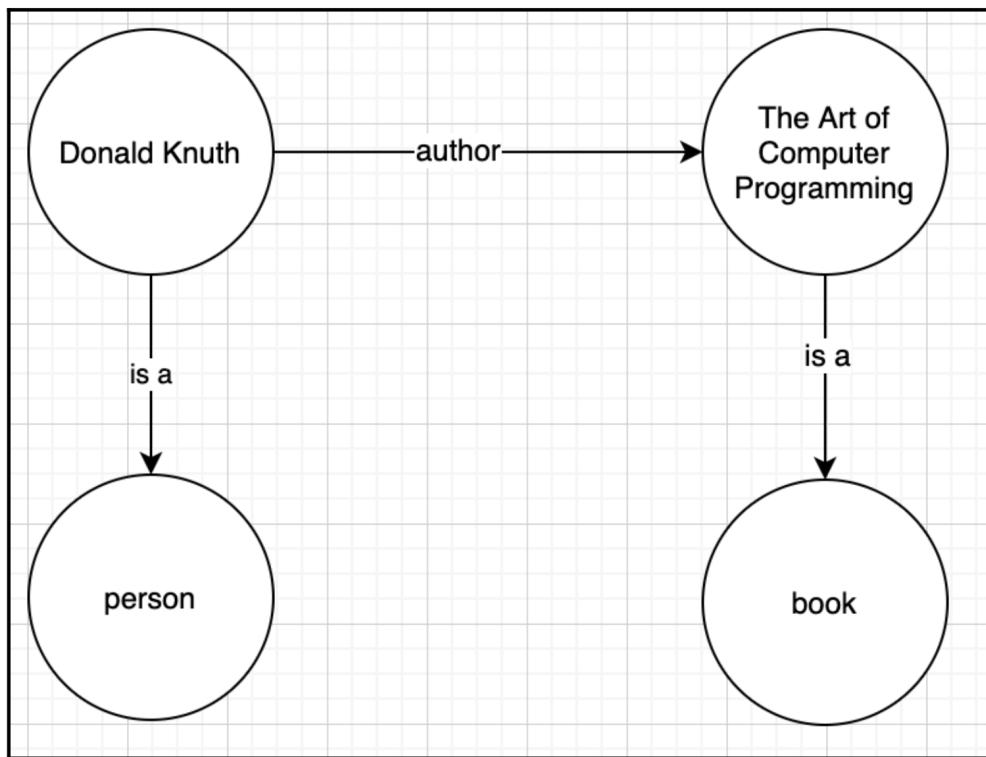
在知识图谱中，每个节点代表一个单独的实体。所谓实体，我们指的是是一座城市，一个人，一只宠物，一本书，或者几乎任何你能想象到的东西。现在，图中的边表示实体之间的连接。每个节点可以通过多个节点连接到另一个节点。例如，看看这两个节点：



这两个节点只包含文本。我们可能会认为 Donald · Knuth 是一个名字，而《计算机编程的艺术》是某种艺术。构建知识图的本质是，我们可以将每个节点与表示其类型的另一个节点关联起来。下面的图是对前一图的扩展：

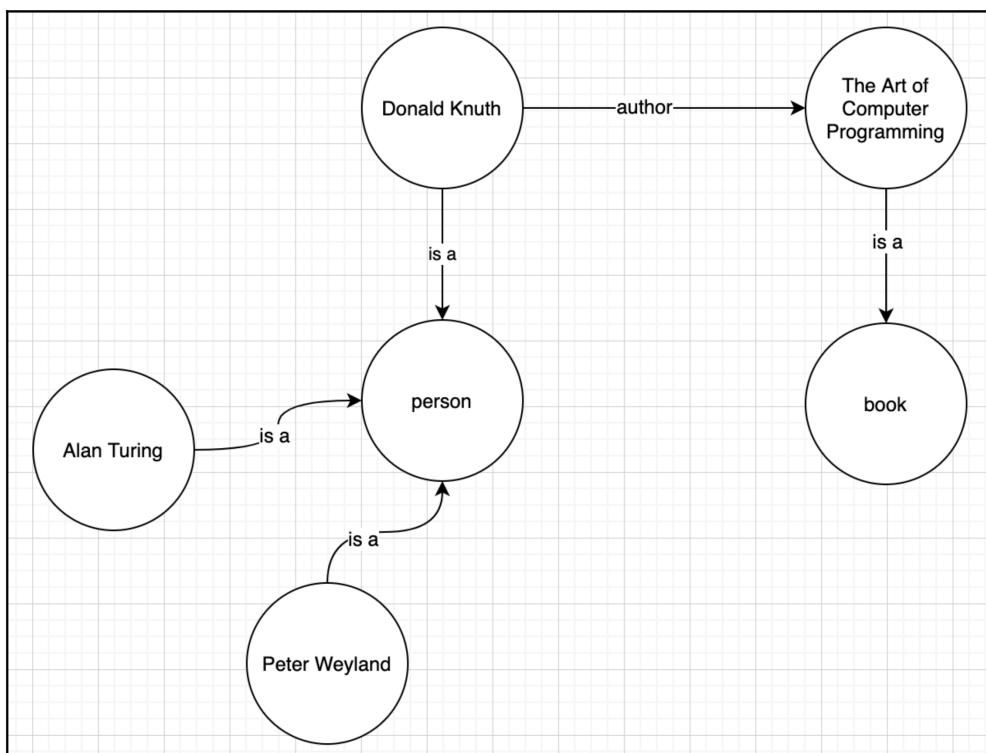


看看我们添加的两个新节点。其中一个代表一个人，而另一个代表一本书。更令人兴奋的是我们用一条边把 Donald · Knuth 节点和 person 节点连接起来并将其标记为'是'的关系。同样地，我们已经将《计算机编程艺术》节点与书籍节点连接起来，所以我们可以说明计算机编程艺术是一本书。现在让我们把 Donald · Knuth 和他写的书联系起来：



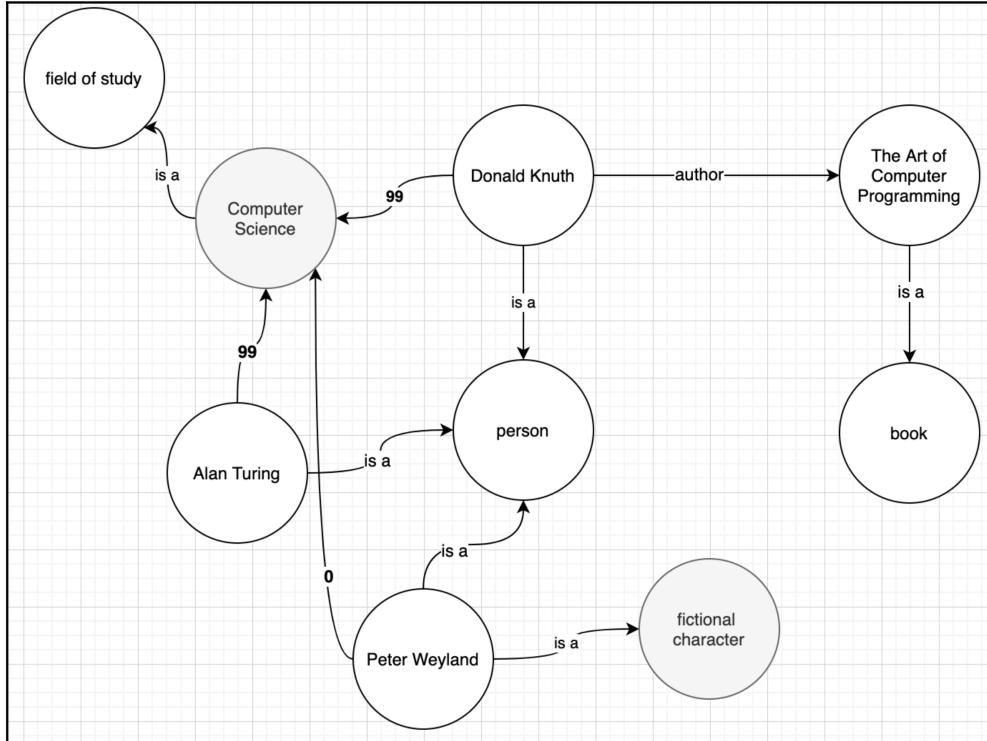
所以，现在我们有了一个完整的关系因为我们知道 Donald · Knuth 是《计算机编程艺术》的作者。

让我们再添加几个节点来表示人。下面的图表显示了我们是如何添加 Alan · Turing 和 Peter · Weyland 节点的：

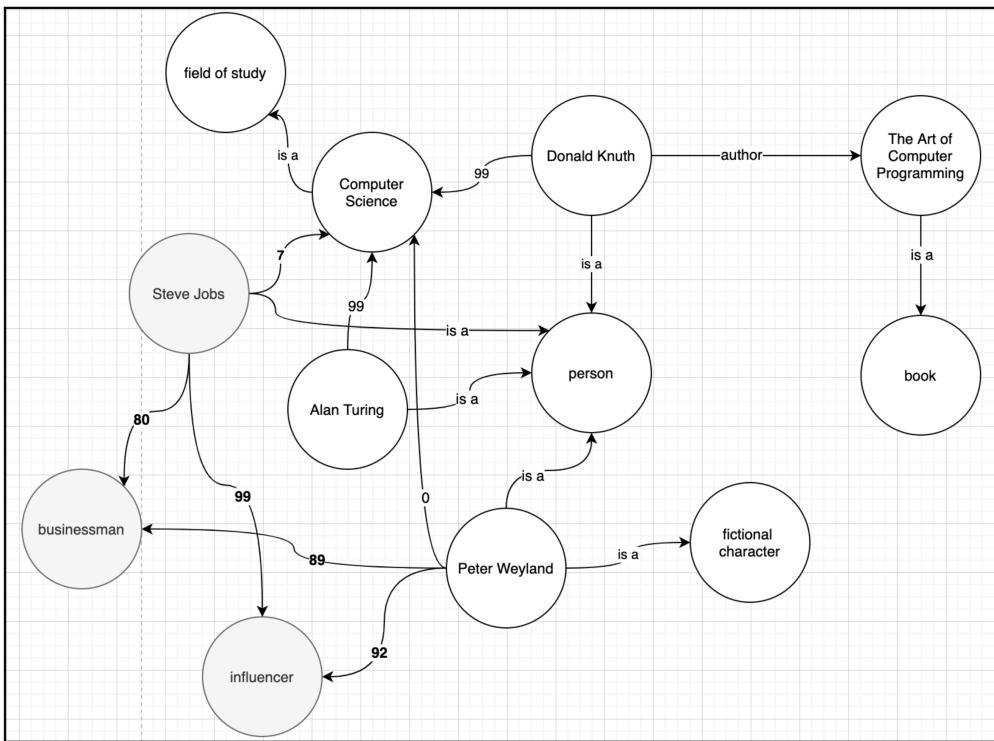


Alan · Turing 和 Peter · Weyland 都是人。现在，如果这是搜索引擎知识库的一部分，那么它给了我们一个很好的洞察用户的搜索意图。当我们看到 Donald · Knuth 的结果时，我们知道这是关

于一个人的。如果有必要，可以建议用户看看在知识图谱中已经积累的其他人。建议搜索 Donald · Knuth 的用户也看看 Alan · Turing 和 Peter · Weyland 的页面是否合理？好了，棘手的部分来了：尽管他们都是人，但他们之间的联系并不紧密。所以，我们需要一些额外的东西来定义两个不同的人之间的联系的相关性。看看图表中添加的以下内容：



现在很清楚的是，Donald · Knuth 和 Alan · Turing 的活动是相同的，作为计算机科学节点，代表了一个研究领域，而 Peter · Weyland 原来是一个虚构的人物。Peter · Weyland 和 Donald · Knuth 唯一的联系就是他们都是人。看看我们放在从人节点到计算机科学节点的边缘上的数字。假设我们从 0 到 100 对这种关系进行评级，后者意味着这种关系是最强的。所以，我们给 Alan · Turing 和 Donald · Knuth 都加了 99。我们应该忽略从 Peter · Weyland 到计算机科学的连线，而不是输入 0，但我们这样做是为了显示对比。这些数字就是权重。我们在边缘添加权重以强调连接性因素，Donald · Knuth 和 Alan · Turing 有相同的东西，而且彼此有很强的联系。如果我们把 Steve · Jobs 作为一个新人加入到知识图谱中，图谱会是这样的：



看看这些边的权值。Steve · Jobs 与计算机科学有某种联系，但他主要与商人和有影响力的节点有关。同样，我们现在可以看到，Peter · Weyland 与 Steve · Jobs 的共同点多于 Donald · Knuth。现在，推荐引擎建议搜索 Donald Knuth 的用户也应该看看 Alan · Turing，因为他们都是人和计算机科学相关，权重相等或接近相等。这是一个将这样的图表整合到搜索引擎中的很好的例子。接下来，我们将向您介绍如何使用类似的知识图来构建一个更智能的框架，以提供相关的搜索结果。我们称之为基于对话框的搜索。

实现基于对话框的搜索引擎

最后，让我们着手设计搜索引擎的一部分，它将为我们提供细粒度的用户界面。正如我们在本章开头提到的，基于对话框的搜索引擎涉及到构建一个用户界面，向用户询问与他们的查询相关的问题。这种方法最适用于结果不明确的情况。例如，搜索 Donald 的用户可能想到以下情况之一：

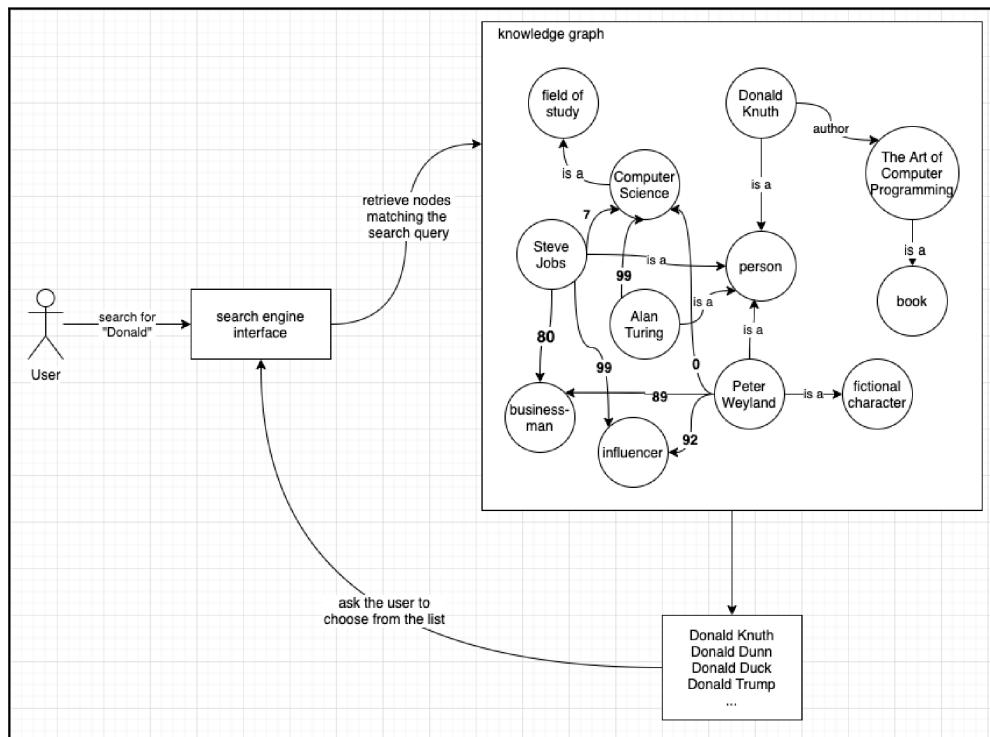
- *Donald Knuth*, 伟大的计算机科学家
- *Donald Duck*, 卡通人物
- *Donald Dunn*, 真实姓名是杰瑞德 · 邓恩, 虚构人物
- *Donald Trump*, 商人和美国第 45 任总统

前面的列表只是 Donald 搜索词潜在结果的一个小示例。那么，缺少基于对话的搜索引擎会做什么呢？它们为用户输入提供最佳匹配的相关结果列表。例如，在我写这本书的时候，搜索 Donald 的结果是一系列与 Donald Trump 有关的网站，尽管我脑子里想的是 Donald Knuth。这里，我们可以看到用户的最佳匹配和最佳匹配之间的关系。

搜索引擎收集了大量的数据用于个性化的搜索结果。如果用户在网站开发领域工作，他们的大多数搜索请求将以某种方式与该特定领域相关。这对于为用户提供更好的搜索结果非常有帮助，例如：用户有一个很大的搜索历史，主要包括与网站开发相关的请求，当搜索 zepelin 时，会得到更

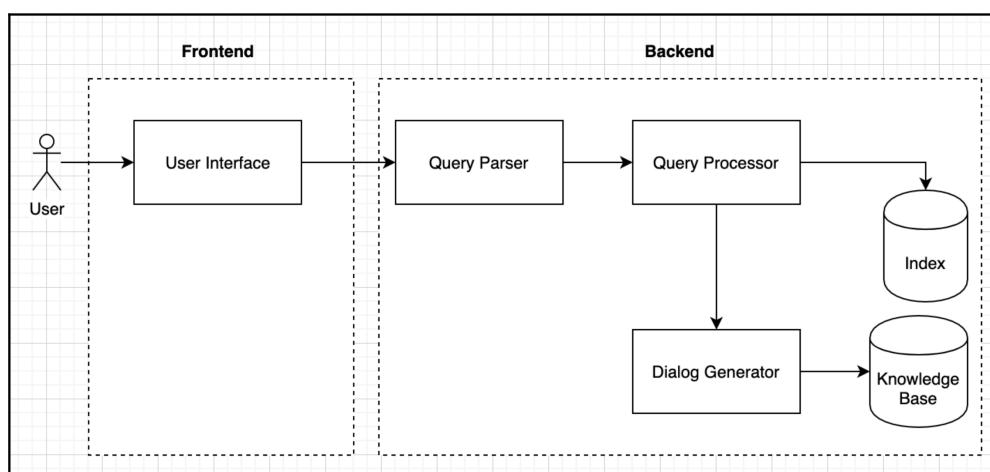
好、更集中的结果。理想的搜索引擎将提供到 Zeplin 应用程序的网站链接，以构建 Web UI，而对于其他用户，该引擎将提供关于名为 Led Zeppelin 的摇滚乐队的信息。

设计一个基于对话框的搜索引擎是为用户提供更好界面的下一步。现在，如果我们已经有了一个强大的知识库，构建它就足够简单了。我们将使用上一节中描述的知识图的概念。让我们假设当用户输入一个搜索词时，我们从知识图中获取所有匹配的主题，并得到用户的潜在搜索列表，如下图所示：



现在用户可以更容易地选择一个主题，并节省回忆全名的时间。当用户输入查询时，来自知识图的信息可以（对于某些搜索引擎来说是）并入自动建议中。此外，我们将处理搜索引擎的主要组成部分。显然，本章不能涵盖实现的各个方面，但我们将讨论的基本组件足以让你跳进自己的搜索引擎的设计和实现。

我们不考虑搜索引擎的 UI 部分。我们最关心的是后台。当谈到应用程序的后端时，我们通常指的是用户不可见的部分。更具体地说，来看看下面的图表：



大部分引擎位于后端。虽然 UI 看起来很简单，但它是整个搜索系统的重要组成部分。这是用户开始他们的旅程的地方，用户界面越多的被设计来提供最好的可能的体验，当用户搜索时，不舒服的体验就越少。我们将专注于后端，下面是我们将要讨论的几个主要模块：

- **查询解析器**: 分析用户查询、规范化单词，并收集查询中每个词汇的信息，以便稍后传递给查询处理器。
- **查询处理器**: 使用索引和补充数据库检索与查询关联的数据，并进行响应。
- **对话生成器**: 提供更多的选项供用户在搜索时选择。对话框生成器是一个补充模块。发出请求的用户可以省略这个对话框，或者使用它进一步缩小搜索结果的范围。

我们跳过了一些在搜索引擎中很常见的组件（比如爬虫），我们将把重点放在那些与基于对话框的搜索引擎密切相关的组件上。现在让我们从查询解析器开始。

实现查询解析器

查询解析器如其名称所示：解析查询。作为查询解析器的基本任务，我们应该用空格分隔单词。例如，用户查询 zeplin best album 会分为以下几个术语：zeplin、best 和 album。下面的类表示基本的查询解析器：

```
1 // The Query and Token will be defined in the next snippet
2 class QueryParser
3 {
4 public:
5     static Query parse(const std::string& query_string) {
6         auto tokens = QueryParser::tokenize(query_string);
7         // construct the Query object and return
8         // see next snippet for details
9     }
10
11 private:
12     static std::vector<Token> tokenize(const std::string& raw_query) {
13         // return tokenized query string
14     }
15 };
```

看一下前面的 parse() 函数。它是这个类中唯一的公共函数。我们将添加从 parse() 函数调用的更多私有函数，以完全解析查询并作为查询对象获取结果。Query 表示一个简单的结构体，包含关于查询的信息，如下所示：

```
1 struct Query
2 {
3     std::string raw_query;
4     std::string normalized_query;
5     std::vector<Token> tokens;
6     std::string dialog_id; // we will use this in Dialog Generator
7 };
```

raw_query 是用户输入的查询的文本表示，normalized_query 是规范化后的相同查询。例如，如果用户输入了 good books, a programmer should read.，_query 是精确的文本，normalized_query

是 good books programmer should read。在下面的代码片段中，我们不使用 normalized_query，但在完成实现后，您将需要它。我们还将符号存储在 Token 的 vector 中，Token 是一个结构体，如下所示：

```
1 struct Token
2 {
3     using Word = std::string;
4     using Weight = int;
5     Word value;
6     std::unordered_map<Word, Weight> related;
7 };
```

相关属性表示与该标记语义相关的单词列表。如果两个词在概念上表达的意思相似，我们称之为语义相关的词。例如，“best” 和 “good” 或者 “album” 和 “collection” 可以认为是语义相关的。您可能已经猜到了哈希表值中权重的用途，我们用它来存储相似度的权重。



权重的范围应该在使用搜索引擎时进行配置。假设我们选择的范围是 0 到 99。best 和 good 两个词的相似度权重可以用接近 90 的数字表示，而 album 和 collection 两个词的相似度权重可以在 40~70 之间偏离。选择这些数字是很棘手的，它们应该在引擎的开发和开发过程中进行调整。

最后，如果用户选择了生成器建议的路径，查询结构的 dialog_id 表示生成的对话框的 ID。我们很快就会讲到这个问题。现在让我们继续结束 parse() 函数。

在 QueryParser 类添加的以下内容：

```
1 class QueryParser
2 {
3     public:
4         static Query parse(const std::string& query_string,
5                             const std::string& dialog_id = "")
6         {
7             Query qr;
8             qr.raw_query = query_string;
9             qr.dialog_id = dialog_id;
10            qr.tokens = QueryParser::tokenize(query_string);
11            QueryParser::retrieve_word_relations(qr.tokens);
12            return qr;
13        }
14     private:
15         static std::vector<Token> tokenize(const std::string& raw_string) {
16             // 1. split raw_string by space
17             // 2. construct for each word a Token
18             // 3. return the list of tokens
19         }
20         static void retrieve_word_relations(std::vector<Token>& tokens) {
21             // for each token, request the Knowledge Base
22             // to retrieve relations and update tokens list
23         }
24 }
```

虽然在前面的代码片段中没有实现两个私有函数 (tokenize 和 retrieve_word_relations)，但基本思想是它们对搜索查询进行规范化和收集信息。在实现查询处理器之前，先看一下前面的代码。

实现查询处理器

查询处理器完成搜索引擎的主要工作，从搜索索引中检索结果，并根据搜索查询响应一个相关的文档列表。在本节中，我们还将介绍对话框的生成。

正如在前一节中看到的，查询解析器构造了一个包含令牌和 dialog_id 的查询对象。我们将在这里的查询处理程序中使用这两种方法。



TIP 考虑到可伸缩性，建议为对话框生成器使用单独的组件。出于学习目的，我们将保持实现的简洁，但您可以自由地重新设计基于对话框的搜索引擎，并完成实现以及爬虫和其他补充模块。

查询对象中的标记用于向搜索索引发出请求，以便检索与每个单词相关联的一组文档。下面是相应的 QueryProcessor 类的样子：

```

1 struct Document {
2     // consider this
3 };
4 class QueryProcessor
5 {
6 public:
7     using Documents = std::vector<Document>;
8     static Documents process_query(const Query& query) {
9         if (!query.dialog_id.empty()) {
10             // request the knowledge graph for new terms
11         }
12         // retrieve documents from the index
13         // sort and return documents
14     }
15 }
```

请将前面的代码片段视为对实现的介绍。我们想表达 QueryProcessor 类的基本思想。它有 process_query() 函数，该函数根据 query 参数中的标记从索引中检索文档。这里的关键角色是搜索索引。就进行快速查询而言，定义其构造的方式和存储文档的方式是必不可少的。同时，作为附加参数提供的对话框 ID 允许 process_query() 函数请求知识库（或知识图谱）来检索与查询相关的更多相关标记。

同样重要的是，考虑到 QueryProcessor 还负责生成对话框（即定义一组路径，为用户提供查询的可能场景）。生成的对话框被发送给用户，当用户进行另一个查询时，使用的对话框将通过我们已经看到的对话框 ID 与该查询关联。

虽然前面的实现主要是介绍性的（因为代码太大，不适合本章），但它进一步设计和实现引擎的基础。

总结

对于经验丰富的程序员来说，从零开始构建搜索引擎也是一项艰巨的任务。我们在这本书中涉及了很多话题，并在本章中通过设计一个搜索引擎将它们中的大部分结合起来。

我们已经了解到，Web 搜索引擎是由几个组件组成的复杂系统，如爬虫、索引器和用户界面。爬虫程序负责定期检查网页，下载网页给搜索引擎索引。索引导致了被称为倒排索引的大数据结构的产生。倒排索引，或者仅仅是索引，是一种数据结构，它将单词映射到它们所在的文档中。

接下来，我们定义了什么是推荐引擎，并尝试为我们的搜索引擎设计一个简单的推荐引擎。推荐引擎与本章讨论的基于对话框的搜索引擎特性相连接。基于对话框的搜索引擎旨在向用户提供有针对性的问题，以了解更多关于用户实际想要搜索的内容。

本书的最后，我们从 C++ 的角度讨论了计算机科学的各种主题。从 C++ 程序的细节开始，然后简要地讨论了使用数据结构和算法有效地解决问题。掌握一门编程语言并不足以在编程方面取得成功。而且，需要解决在数据结构、算法、多线程等方面需要大量技能的编码问题。此外，处理不同的编程范式可以改变对计算机科学的看法，并允许以全新的视角来解决问题。本书中，我们已经接触了几种编程范式，如函数式编程。

最后，软件开发不仅仅局限于编码。构建和设计项目是成功应用程序开发的关键步骤之一。第 10 章到第 16 章，主要是关于设计现实世界应用程序的方法和策略。让这本书成为你以 C++ 开发人员的角度了解编程世界的入门指南。通过开发更复杂的应用程序来提高你的技能，并与同事和那些刚刚开始职业生涯的人分享知识。

问题

1. 爬虫程序在搜索引擎中的角色是什么？
2. 为什么我们称搜索索引为反向索引？
3. 在索引单词之前对单词进行标记的主要规则是什么？
4. 推荐引擎的作用是什么？
5. 什么是知识图谱？

扩展阅读

更多信息，请参阅以下书籍：

- Introduction to Information Retrieval, Christopher Manning, et al.,
<https://www.amazon.com/Introduction-Information-Retrieval-Christopher-Manning/dp/0521865719/>

4 评估

第 1 章

1. 从源代码生成可执行文件的过程称为编译。编译 C++ 程序是一项复杂的任务。通常，C++ 编译器解析和分析源代码，生成中间代码，优化它，最后在目标文件的文件中生成机器码。另一方面，解释器不产生机器代码，它会逐行执行源代码中的指令。
2. 首先是预处理，然后编译器通过解析代码来编译代码，执行语法和语义分析，然后生成中间代码。优化生成的中间代码之后，编译器生成最终目标文件（包含机器码），然后可以将其与其他目标文件链接起来。
3. 预处理器的目的是处理源文件，使它们为编译做好准备。预处理程序使用预处理器指令，比如 `#define` 和 `#include`。指令不代表程序语句，但它们是预处理器的命令，告诉它如何处理源文件的文本。编译器无法识别这些指令，因此无论何时在代码中使用预处理器指令，预处理器都会在代码开始实际编译之前解析它们。
4. 编译器为每个编译单元输出一个目标文件。链接器的任务是将这些目标文件组合成单个目标文件。
5. 库可以作为静态库或动态库与可执行文件链接。当为静态库时，将成为最终可执行文件的一部分。动态链接库会被操作系统加载到内存中，以便为您的程序提供使用相应的功能。

第 2 章

1. 通常，`main()` 函数有两个形参，`argc` 和 `argv`，其中 `argc` 是程序的输入参数数，`argv` 构成这些输入参数。偶尔会看到一个得到广泛支持但没有标准化的第三个参数，最常见的名称是 `envp`。`envp` 的类型是一个 `char` 指针数组，它保存着系统的环境变量。
2. `constexpr` 说明符声明函数的值可以在编译时求值。同样的定义也适用于变量。名称由 `const` 和表达式组成。
3. 递归会为函数调用分配额外的空间。与迭代解决方案相比，为函数和调用分配空间代价会很大。
4. 堆栈可以自动存储对象，开发者不需要关心相应用对象的构造和销毁。通常，堆栈用于函数参数和局部变量。另一方面，堆允许在程序执行期间分配新的内存。然而，适当的内存空间回收也是开发者的责任。
5. 指针的大小不依赖于指针的类型，因为指针是表示内存中地址的值。地址的大小取决于系统。通常，它不是 32 位就是 64 位。因此，我们说指针的大小是 4 字节或 8 字节。
6. 就项目位置而言，数组具有独特的结构。它们被连续地放置在内存中，第二项放在第一项的右边，第三项放在第二项的右边，以此类推。考虑到这个特性，以及数组由相同类型的元素组成的事，访问任何位置的项都需要固定的时间。
7. 如果在 `case` 语句中忘记 `break` 关键字，执行将传递给下一个 `case` 语句，而不检查其条件。
8. 例如，`operations['+'] = [](int a, int b) { return a + b; }`

第 3 章

1. 类型、状态和行为。

2. 当移动对象而不是复制对象时，我们省略了临时变量。
3. 在 C++ 中，除了默认的访问修饰符之外，结构和类之间没有任何区别。这对于结构体是公共的，对于类是私有的。
4. 在聚合的情况下，可以在没有聚合的情况下实例化包含一个或多个其他类实例的类。
5. 私有继承从派生类的客户端代码中隐藏继承的成员。受保护的继承也执行同样的操作，但允许链中的派生类访问这些成员。
6. 通常，虚函数的引入会导致使用指向虚函数表的扩充。通常，这加起来有 4 或 8 个字节的空间（根据指针的大小）指向类对象。
7. 单例设计模式允许构造该类的单个实例。在许多项目中，我们需要确保类的实例数量限制在一个以内，这是很有帮助的。例如，如果一个数据库连接类作为一个单例来实现，它的工作效果最好。

第 4 章

1. 如果使用得当，宏是强大的工具。然而，以下几个方面限制了宏的使用。(1) 你不能调试宏;(2) 宏观扩张会导致奇怪的副作用;(3) 宏没有命名空间，所以如果你有一个宏与其他地方使用的名字冲突，你会得到你不想要的宏替换，这通常会导致奇怪的错误消息;(4) 宏可能会影响你不知道的事情。有关详情，请浏览 <https://stackoverflow.com/questions/14041453> .
2. 类/函数模板是指一种用来生成模板类/函数的模板。它只是一个模板，而不是一个类/函数，因此编译器不会为它生成任何目标代码。模板类/函数是类/函数模板的一个实例。因为它是一个类/函数，相应的目标代码由编译器生成。
3. 定义类/函数模板时，在关键字 template 后面有一个 <> 符号，其中必须给出一个或多个类型形参。<> 内部的类型形参称为模板形参列表。当实例化一个类/函数模板时，所有模板形参都必须替换为它们对应的模板实参，这就是模板实参列表。
隐式实例化按需发生。但是，当提供库文件 (.lib) 时，您不知道用户将来将使用哪种类型的参数列表，因此，需要显式实例化所有可能的类型。
4. 多态性意味着某些东西以不同的形式存在。具体来说，在编程语言中，多态性意味着一些函数、操作或对象在不同的上下文中具有几种不同的行为。在 C++ 中，有两种多态性：动态多态性和静态多态性。动态多态性允许用户确定要在运行时执行的实际函数方法，而静态多态性意味着要调用的实际函数（通常是要运行的实际代码）在编译时是已知的。
函数重载意味着函数用相同的名称定义，但参数集合不同（不同的签名）。
函数重写是指子类重写父类中定义的虚方法的能力。
5. 类型特征是一种用于收集有关类型信息的技术。在它的帮助下，我们可以做出更智能的决策，开发高质量的泛型编程优化算法。类型特征可以通过部分或全部模板特化来实现。
6. 我们可以在 g() 中写一条错误语句，然后构建代码。如果一个未使用的函数被实例化，编译器将报告错误，否则它将被成功构建。您可以在以下文件 ch4_5_class_template_implicit_inst_v2.h 和 ch4_5_class_template_implicit_inst_B_v2.cpp 和 ch4_q7.cpp 中找到示例代码，网址是 <https://github.com/PacktPublishing/Mastering-Cpp-Programming./tree/master/Chapter-4>
7. 这是一个开放练习，不需要标准答案。

第 5 章

1. 计算机内存可以被描述为一个单一的概念-动态 RAM(DRAM)，或作为计算机包含的所有内存单元的组合，从寄存器和高速缓存开始，到硬盘驱动器结束。从程序员的角度来看，DRAM 是最有趣的，因为它保存着计算机中运行的程序的指令和数据。
2. 虚拟内存是一种有效管理计算机物理内存的方法。通常，操作系统集成了虚拟内存来处理来自程序的内存访问，并有效地将内存块分配给特定的程序。
3. 在 C++ 中，我们使用 new 和 delete 操作符来分配和释放内存空间。
4. delete 释放为单个对象分配的空间，而 delete[] 用于动态数组，释放堆上数组的所有元素。
5. 垃圾收集器是一种工具或一组工具和机制，在堆上提供自动资源回收。对于垃圾收集器，需要环境支持，比如虚拟机。C++ 可以编译生在没有支持环境下运行的机器码。

第 6 章

1. 当向 vector 对象插入新元素时，它将被放置在 vector 对象中已经分配的空闲槽位上。如果 vector 的大小与容量相等，则意味着 vector 没有存放新元素的空闲槽位。在这些（罕见的）情况下，vector 会自动调整自身的大小，这涉及到分配新的内存空间并将现有元素复制到新的更大的空间。
2. 当在链表的前面插入元素时，我们只创建新元素并更新链表指针，以有效地将新元素放入链表中。在 vector 的前端插入新元素需要将所有 vector 元素向右移动，从而为该元素腾出一个槽位。
3. 选择排序搜索最大（或最小）元素，并用该最大（或最小）元素替换当前元素。插入排序将集合分成两个部分，遍历未排序的部分，并将其每个元素放置在已排序部分的适当槽中。
4. 参考 GitHub 中这一章的源代码。

第 7 章

1. C++ 中的 ranges 库允许处理元素的范围，使用视图适配器来操作它们，这要高效得多，因为它们不将整个集合作为适配器结果存储。
2. 如果一个函数不修改状态，并且为相同的输入产生相同的结果，那么它就是纯函数。
3. 纯虚函数是没有实现的函数的特征。纯虚函数用于描述派生类的接口函数。函数式编程中的纯函数是那些不修改状态的函数。
4. 折叠表达式（或简化）是将一组值组合在一起以生成较少数量的结果的过程。
5. 尾递归允许编译器通过忽略为每个递归调用分配新的内存空间来优化递归调用。

第 8 章

1. 如果两个操作的开始时间和结束时间在任意点交叉，那么它们将并发运行。
2. 并行性意味着任务同时运行，而并发性并不强制任务同时运行。
3. 过程是程序的映像。它是程序指令和装入计算机内存的数据的组合。
4. 线程是进程范围内的一段代码，可以由操作系统调度器调度，而进程是正在运行的程序的镜像。

5. 参考本章中的例子。
6. 通过使用双重检查锁定。
7. 参考 GitHub 中这一章的源代码。
8. C++20 引入协程作为对经典异步函数的补充。协程将代码的后台执行移动到下一个级别，允许一个函数在必要时被暂停和恢复。`co_await` 是一个告诉代码等待异步执行代码的构造。这意味着函数可以在此时挂起，并在结果就绪时继续执行。

第 9 章

1. 双重检查锁定是使单例模式在多线程环境中完美工作的一种方法。
2. 这是一种确保在复制另一个堆栈的基础数据时不会被修改的方法。
3. 原子操作是一种不可分割的操作，原子类型利用较低级别的机制来确保指令的独立和原子执行。
4. `load()` 和 `store()` 利用低级机制来确保写和读操作以原子的方式完成。
5. 除了 `load()` 和 `store()` 之外，还有诸如 `exchange()`、`wait()` 和 `notify_one()` 等操作

第 10 章

1. TDD 代表测试驱动开发，其目标是在项目的实际实现之前编写测试。这有助于更清楚地定义项目需求，并预先避免代码中的大多数错误。
2. 交互图描绘了对象之间交流的准确过程。这允许开发人员对任何给定时刻的实际程序执行有一个高级视图。
3. 在聚合的情况下，可以在没有聚合的情况下实例化包含一个或多个其他类实例的类。
4. 简单地说，Liskov 替换原则确保任何函数接受某种类型 `T` 的对象作为参数，如果 `K` 扩展了 `T`，也将接受类型 `K` 的对象。
5. 开闭原则表示类应该对扩展开放，对修改关闭。在上述例子中，`Animal` 是可扩展的，因此它与从 `Animal` 继承 `monkey` 类的原则并不矛盾。
6. 参考 GitHub 中这一章的源代码。

第 11 章

1. 重写私有虚函数允许通过保持类的公共接口不变来修改类的行为。
2. 它是一种行为设计模式，其中对象封装了一个操作以及执行该操作所需的所有信息。
3. 通过尽可能多地与其他对象共享数据。当我们有很多具有相似结构的对象时，在对象之间共享重复的数据可以最小化内存的使用。
4. 观察器将事件通知订阅者对象，而中介扮演互通对象之间的连接中心的角色。
5. 将游戏循环设计成无限循环是合理的，因为从理论上讲，游戏可能永远不会结束，只有在玩家命令时才会结束。

第 12 章

1. 物理层、数据链路层、网络层、传输层、会话层、表示层和应用层。

2. 端口号提供了一种区分在同一环境中运行的几个网络应用程序的方法。
3. 套接字是一种抽象，为程序员提供了一种通过网络发送和接收数据的方法。
4. 首先，我们需要创建和绑定一个 IP 地址的套接字。接下来，我们应该侦听传入的连接，如果说有的话，我们应该接受连接以进一步处理数据通信。
5. TCP 是一种可靠的协议。它处理端点之间的强连接，并通过重新发送未被接收方接收到的包来处理包丢失。另一方面，UDP 是不可靠的。几乎所有的处理工作都由程序员来完成。UDP 的优势在于它的速度，因为它忽略了握手、检查和包丢失处理。
6. 宏定义会导致代码中的逻辑错误，很难发现。使用 const 表达式总是比使用宏更好。
7. 客户端应用程序必须具有惟一的标识符以及用于授权和/或验证它们的令牌（或密码）。

第 13 章

1. 这是一个实验练习。
2. 以下是 Ubuntu 18.04 在 NVIDIA Jetson Nano 上的输出：

```

1  swu@swu-desktop:~/ch13$ g++ -c -Wall -Weffc++ -Wextra
2  ch13_rca_compound.cpp
3  ch13_rca_compound.cpp: In function ‘int main()’:
4  ch13_rca_compound.cpp:11:17: warning: operation on ‘x’ may be
5  undefined [-Wsequence-point]
6  std::cout << f(++x, x) << std::endl; //bad, f(4,4) or f(4,3)?
7  ^
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32

```

```

33 int v1; //good, here the declaration order is clear
34 ^
35 ch13_rca_order_of_evaluation.cpp:25:2: warning: when initialized
36 here [-Wreorder]
37 B(float x) : v2(x), v1(v2) {};
38 ^
39 swu@swu-desktop:~/ch13$ g++ -c -Wall -Weffc++ -Wextra
40 ch13_rca_uninit_variable.cpp
41 ch13_rca_uninit_variable.cpp: In function ‘int main()’:
42 ch13_rca_uninit_variable.cpp:7:2: warning: ‘x’ is used
43 uninitialized in this function [-Wuninitialized]
44 if (x) {
45 ^
46

```

3. 因为静态分析工具从它们的模型中预测错误，而动态分析工具通过程序的执行来检测错误。
4. 请参考示例代码 ch13_tdd_v3.h、ch13_tdd_v3.cpp 和 ch13_tdd_Boost_UTF3.cpp
<https://github.com/PacktPublishing/Mastering-Cpp-Programming./tree/master/Chapter-13>

第 14 章

1. Qt 的编译模型不用虚拟机。它使用元对象编译器 (MOC) 将其翻译成 C++，然后将其编译成特定平台的机器码。
2. QApplication::exec() 是应用程序的起点。它开始 Qt 的事件循环。
3. 通过调用 setWindowTitle()
4. m->index (2, 3) .
5. wgt->resize (400, 450) .
6. 继承 QLayout 时，应该提供 addItem()、sizeHint()、setGeometry()、itemAt()、takeAt() 和 minimumsize() 函数的实现。
7. 通过使用 connect() 函数，该函数以源和目标对象以及信号和槽的名称作为参数。

第 15 章

1. ML 代表机器学习，是一个研究算法和统计模型的领域，计算机系统使用这些算法和模型来执行特定任务，而不需要使用明确的指令，而是依赖模式和推理。
2. 监督学习算法（也称为导师训练）从标记的数据集学习，每个记录都包含描述数据的附加信息。无监督学习算法甚至更加复杂——它们处理包含一组特征的数据集，然后试图找到这些特征的有用属性。
3. ML 应用包括机器翻译、自然语言处理、计算机视觉和电子邮件垃圾检测。
4. 其中一种方法是对每个结果加一个权重，如果减法运算的权重大于其他运算的权重，则它将成为主导运算。
5. 神经网络的目的是识别模式。

第 16 章

1. 爬虫程序下载网页并存储其内容，以便搜索引擎建立索引。
2. 我们称它为反向索引，因为它将单词映射回它们在文档中的位置。
3. 在建立索引之前，标记化对单词进行规范化。
4. 推荐引擎验证并推荐适合特定请求的最佳结果。
5. 知识图谱是这样一种图，其中节点是主题（知识），边是主题之间的连接。