# 100 C++ Mistakes and How to Avoid Them

Rich Yonts

MANNING

# C++ 编程避坑指南：100 个常见错误及解决方案

作者： Rich Yonts

译者：陈晓伟

# 前言

学习 C++ 的核心目标在于运用其独特的语言特性，以解决特定环境下的问题。无论是大学教育，还是工作场所对初级开发者的指导，虽然场景不同，但所使用的语言基础是相同的。可以将 C++ 视为开发者在最底层所使用的通用语言。然而，在实际应用中，设计模式、常规使用方法、具体问题领域的细节，以及公司内部流程则为更高层次的沟通方式，而这些高层次的内容才是最为关键的部分。正如 Alan Turing 所证明的，“任何一台计算机都能够计算其他计算机能够解决的所有可解问题，只不过实现的方法和所需的时间可能有所不同”。同样地，任何编程语言都可以处理 C++ 所能解决的所有问题。

这并非旨在批评 C++(或者其他的编程语言)，而是强调这样一个事实：即使是在企业环境中使用 C++，对于公司整体发展方向和解决的问题而言，几乎不会产生影响。成为一名资深开发者的技能，远比掌握某一特定编程语言的知识重要得多。

尽管如此，为什么这本书仍然专注于 C++ 呢？在此之前，我们已经确定各位读者在使用 C++ 的工作环境中生存。基于对这门语言的兴趣与技能，将有机会运用自己的知识和技能来应对公司关注的问题。换句话说，你将通过 C++ 来完成某项任务。

因此，了解如何识别 C++ 中常见的错误，可以帮助你减少这些错误的发生。更重要的是，避免再次犯同样的错误。如果必须为一个特定方法主导的代码库工作，那么就失去了以最具表现力和最合适的方式使用 C++ 的机会。例如，许多字符串处理方案仍然采用 C 语言风格的习惯用法，如 strcat、strstr 和 strcpy。如果这种做法在团队中普遍存在，你也可能会倾向于甚至被迫继续沿用这些方法。其实，应该避免使用这些函数，而优先考虑 C++ 提供的更安全的方法。

C++ 是一种极其灵活的语言，允许开发者执行机器所能完成的任何操作。许多较新的编程语言通过抽象隐藏了大部分机器细节和复杂性，包括 Go 和 Rust。相比之下，C++ 提供了操纵机器特性的最低粒度，使其成为处理底层操作的理想选择。

然而，这种灵活性和精细控制并非没有代价，无法像使用 Python 或 Java 那样以“宽松”的方式编写 C++ 代码。例如，C++ 并没有提供默认的垃圾回收机制来自动管理内存和其他资源，开发者必须手动处理。在这种资源管理过程中出现的错误，往往会对程序的性能和正确性造成严重影响。引用《蜘蛛侠》(2002) 电影中的台词似乎非常恰当，其中 Peter Park 的叔叔 Ben 警告过 Peter Park：“能力越大，责任越大。”这句话完全可以表达 C++ 的开发哲学。

我对 C++ 的兴趣起源于 Bjarne Stroustrup 博士开发的从 C++ 到 C 的转译器 CFront。在此之前，我已经学习并使用过 C 语言，并一直在寻找机会实践 C++ 项目。最终，在 20 世纪 80 年代末，随着工作站革命的到来，我开始正式学习 C++。随后大约在 1995 年，Java 的出现中断了我的 C++ 之旅。直到 2020 年代初，我又重新开始用 C++ 进行项目开发。同时，我也在大学教授编程课程，并开设了许多关于 C++ 的课程。

这两个阶段的教学经历中，我逐渐意识到 C++ 的一些缺点、经典问题及惯用法的存在，同时注意到现代 C++ 的特性应用相对较少；很多教科书几乎没有提及 C++ 的现代特性。因此，我决定以书籍的形式分享我的经验和发现。我希望，学生和从业者能够受到鼓舞，去审视现有的代码，理解其中存在的问题，并探索改进的方法。如果他们能够成功避开这些陷阱，则在学习和实践中就已经领先于许多人。

# 致谢

英国诗人 John Donne 于 1624 年在其散文中写道：“没有人是一座孤岛。”这句话深刻地揭示了人类之间复杂而多样的联系。撰写一本书籍不仅能够展现知识，更能揭示作者与读者、与世界之间的相互关联，这种连接是其他形式难以企及的。

在此，我要向 Manning 出版社的编辑团队及其支持人员表达诚挚的感谢，尤其是 Doug Rudder 和 Michael Stephens 两位先生。他们给予了我充分的信任和支持，使我得以将一个初步的概念发展成为现实，并在整个过程中保持方向正确。他们的专业指导帮助我整理思路，修正了许多不完善的地方。与他们合作是一段愉快且富有成效的经历。此外，还有许多幕后默默付出的团队成员，他们的工作如此出色，以至于他们的努力几乎未被特别提及。但这恰恰证明了他们的高效与专业，正是这些“无名英雄”确保了项目的顺利进行。我对所有参与此书制作的人员表示深深的谢意。

我的技术编辑 Matt Godbolt 在整个创作过程中给予了我极大的支持和细致入微的帮助。作为一名资深的 C++ 开发人员，Matt 不仅对技术有着深刻的见解，还致力于在代码的可读性、可维护性与高性能解决方案之间寻求最佳平衡点。他以敏锐的眼光发现了书中存在的若干错误、不一致之处，并提出了宝贵的改进建议。尽管书中可能仍存在一些不足，但这一切责任应由我本人承担。Matt 的专业精神和技术实力为本书质量的提升做出了重要贡献。

还要感谢所有审稿人：Abe Taha、Abel Sen、Abir Qasem、Aldo Biondo、Andre Weiner、Dmitri Nesteruk、Doyle Turner、Frances Buontempo、Francesco Basile、Ganesh Harke、Greg Wagner、Ivanuki、Jialin Jiao、John Donoghue、John Villarosa、Juan José Durillo Barrionuevo、Juan Rufes、Keith Kim、Leon Pfletschinger、Luke Kupka、Marcus Geselle、Michael Wall、Narendra Merla、Ofek Shilon、Piotr Jastrzebski、Rajat Kant Goel、Richard Meinsen、Ronaldo Scarpate、Ruud Gijsen、Saboktakin Hayati、Sachin Handiekar、Shantanu Kumar、Shawn Smith、Simone Sguazza、Srinivas Vamsi Parasa、Sriram Macharla、Stanley Anozie、Stefan Turalski、Timothy Jaap van Deurzen、Tr oi Eisler 和 Walt Stoneburner。你们的建议让这本书变得更好。

整个创作过程中，我的家人始终如一地给予我坚定的支持。我的妻子 Kim 以她充满同情心的话语，无数次地帮助我克服困难，坚持完成每一项任务。她的理解与支持如同一股温暖的力量，推动着我向前迈进。与此同时，我的孩子们也表现出了极大的热情和支持，他们总是为父亲写作书籍而感到自豪和兴奋。通过他们的鼓励，我得以在面对挑战时保持乐观的态度，这种积极的情绪不仅贯穿于本书的创作过程，也渗透到了生活的方方面面。

家庭的支持不仅仅体现在情感上，更在于它提供了一种无形的动力，使得个人能够在追求梦想的路上走得更远、更稳。正如资料中提到的，“家长是儿童诚信品质形成的重要他人”，而在成年人的世界里，伴侣和子女同样可以成为我们价值观和行为模式形成的关键影响者。一个充满关怀、尊重和信任的家庭氛围，能够极大地促进个体的成长与发展，无论是儿童时期的道德教育还是成人阶段的职业成就，都离不开这样的环境滋养。

最后，我深信，个人所拥有的一切——包括才华、教育背景、工作经验等等——皆源于至高无上的天父之恩赐。这份恩情浩瀚无边，无法完全偿还。我们唯一能做的，就是怀着感恩之心去享受这

一切，并努力将这份善意传递给更多的人（雅各书[1] 1:17）。

[1]译者注：《圣经》新约的一卷书

# 关于本书

C++ 作为一种支持多范式的编程语言，自计算机科学发展的早期阶段便已存在。它最初以“更好的 C”（即带有类的 C 语言）面貌出现，如今已经演进为一种基于国际标准、持续积极开发的语言。截至本文撰写之时，C++20 标准已经正式发布，而 C++23 标准正处于最终审查阶段，同时 C++26 标准的相关制定工作也在紧锣密鼓地进行中。C++ 的应用范围极其广泛，无论是规模微小还是庞大的项目，无论开发者经验深浅，都能在软件开发的大多数领域找到其身影。

据保守估计，当前正在生产环境中运行的 C++ 代码量达到了数百亿行之巨。这些代码由来自不同背景的无数团队以及数百万名开发人员，在过去的几十年间共同书写完成。每位开发者对于何为正确的编程模型、优质的 C++ 代码，以及适宜的开发方法都有着各自独特的见解。

这些代码在正确性、可读性、效率及性能等方面的表现参差不齐。大量代码遵循命令式编程范式；部分代码采用面向对象的方式构建；仅有少量代码体现了函数式编程的特点。其中一部分是由初学者编写，但更多的则出自专家之手，尽管这些专家未必都是 C++ 语言设计与实现方面的权威。值得注意的是，相当大比例的现存代码在 1998 年首个 C++ 标准化进程启动之前创建，而在那之后编写的代码虽然可能符合某一种特定的标准，但从外观上来看仍保留着前标准化时代的风格特征。

本书聚焦于使用 C++ 进行程序开发过程中所遇到的一些关键问题。所谓现代 C++，通常是指从 C++11 标准开始的一系列版本。鉴于前现代（或称经典）C++ 遗留下来的庞大代码库，深入理解这些问题变得尤为重要，而解决这些问题更是势在必行。通过阅读本书，开发者应当能够更加精准地识别并修正书中提及的问题。此外，对于那些未详细探讨的错误，开发者也将具备更强的识别能力和修复技巧。深入思考这些问题有助于开发人员，更好地把握语言特性和开发流程中的细微差异，从而提升代码质量。

## 适读人群

无论代码的起源、遵循的标准、采用的范式以及技术成熟度如何，总需要有人积极地对其进行维护。这包括添加新功能、修改或增强现有功能、修复缺陷以及优化性能低下之处。尽管代码库中可能存在诸多问题，但程序依然能够运行并生成有意义的结果。

考虑到读者可能是 C++ 开发领域的新人，其背景可能源于个人自学、大学教育或只接触过其他编程语言。对于大多数开发者而言，很少有机会从零开始构建一个全新的项目；因此，读者很可能会更多地参与到既有代码的工作当中。所以各位的职责将涉及为这些庞大的旧代码库开发新特性，并解决其中已有的问题。

各位这时面临的挑战在于，如何在这种环境下编写 C++ 代码。实际上，C++ 很少是在完全自由无约束的环境中开发出来的，在这样的环境中开发者可以独自做出所有的决策。相反，真实的开发环境与单纯学习如何用 C++ 编写代码有所不同，因为公司或者团队通常会制定各种指导原则、风格指南、命名规则以及其他约束。此外，现有的代码库已经确立了可接受的架构模式、命名约定、使用规范，以及针对常见问题的解决方案。不过，这些听起来的确与实际的 C++ 编码工作没有什么直接关联……

## 本书的路线图

第一部分所指出的错误展示了对 C++ 语言及其特性的不当使用，而这些问题可以通过采用更新的语言特性和标准模板库（STL）功能来加以改善。具体而言，第 2 章深入探讨了类与数据类型的正确运用，特别强调了类的设计原则以及智能指针的应用。第 3 章则聚焦于编程过程中常见的陷阱，并探讨了可能需要进一步挖掘和利用的语言特性。第 4 章则详细介绍了 C++ 语言近年来的重要变革，同时提出了几种能够有效管理和缓解常见问题的技术手段。

由于 C++ 的历史渊源可以追溯到 C 语言，本书第二部分展示的错误反映了现代 C++ 的一些深层次问题。许多此类问题可以通过引入现代技术和实践得到显著改善，从而为开发者带来巨大的便利和效率提升。第 5 章分析了一些在不知不觉间融入 C++ 开发中的传统习惯与做法，尽管这些方法往往并非最优解。第 6 章则着重分析了早期 C++ 编程中流行的一些不良实践，这些做法至今仍广泛存在于众多代码库中。

第三部分讨论的错误类型主要集中在遗留代码库中普遍存在的挑战，开发者往往难以直接借助现代技术手段进行彻底解决。尽管如此，我们仍然可以针对若干问题实施改进措施，进而逐步提高整体代码质量。第 7 章深入研究了良好类设计中存在的若干关键议题，以及构建坚固可靠的类时可能遭遇的各种麻烦。第 8 章继续围绕良好类设计展开讨论，揭示了那些可能意外导致设计方案失败的细节问题。第 9 章则集中探讨了类所提供的多种操作方式；若处理不当，可能会不经意间引发复杂的问题。第 10 章关注系统资源管理方面的内容，列举了几种因操作失误可能导致的问题。第 11 章则专注于函数调用及参数传递环节可能出现的各类状况。最后，第 12 章回顾了使用前现代 C++ 进行常规编码时可能遇到的一系列问题。

## 源码库

您可以从本书的 liveBook（在线）版本 https://livebook.manning.com/book/100-c-plus-plus-mistakes-and-how-to-avoid-them 获取可执行代码片段。书中示例的完整代码可从 Manning 的网站下载：https://www.manning.com/books/100-c-plus-plus-mistakes-and-how-to-avoid-them。

## liveBook 论坛

购买本书的读者可免费访问 Manning 的在线阅读平台 liveBook。使用 liveBook 的独
可免费访问 Manning 的在线阅读平台 liveBook。使用 liveBook 的独家讨论功能，可以对书籍全局或特定章节或段落发表评论。可以做笔记、提出和回答技术问题以及获得作者和其他用户的帮助。要访问论坛，请访问 https://livebook.manning.com/book/100-c-plus-plus-mistakes-and-how-to-avoid-them/discussion。还可以在 https://livebook.manning.com/discussion 上了解有关 Manning 论坛和行为准则的更多信息。

Manning 对我们读者的承诺是提供一个平台，让读者之间以及读者和作者之间能够进行有意义的对话。这并不是对作者参与论坛的任何具体次数的承诺，作者对论坛的贡献仍然是自愿的（并且是无偿的）。我们建议您尝试向作者提出一些有挑战性的问题，以免他失去兴趣！只要这本书还在印刷中，就可以从出版商的网站上访问论坛和之前讨论的存档

# 关于作者

Rich Yonts 是 Teradata 的高级软件工程师，也是一位长期使用 C++、Java 和 Python 进行软件开发的工程师。他在 IBM 和 Sony 工作多年，曾担任过许多技术职务和领导职位，在大型代码库方面拥有丰富的经验。

# 封面插图

封面上的人物被称为“L’Infirmier”或“护士”，这一形象来源于 Louis Curmer 在 1841 年出版的一本书籍 11。每一幅插图都是经过精心的手工绘制与着色，展现了那个时代艺术创作的细致与匠心。

在那个时期，一个人的着装往往成为识别其居住地、职业乃至社会地位的重要线索。这种通过服饰来区分身份的做法，在历史上具有深远的文化意义和社会功能。Manning 对计算机行业的创造力和创新精神给予了高度评价，而这些书籍的封面设计正是受到了几个世纪前丰富多彩地域文化的启发，并借助历史图像收藏中的资料得以重现。这样的设计不仅连接了过去与现在，也体现了技术与艺术之间的和谐共生，使读者能够在享受现代科技带来的便利的同时，也能感受到历史文化的深厚底蕴。

通过将传统手工着色技术与现代设计理念相结合，这些封面不仅仅是对过去的致敬，更是对当今数字时代中人类创造力的一种颂扬。它们提醒我们，即使是在高度数字化的世界里，手工艺的价值依然不可替代，它能够赋予作品独特的个性和温度，这是机器生产难以企及的。因此，无论是从历史文化的角度还是从当代艺术设计的视角来看，这些封面都具有重要的研究价值和审美意义。

# 目录

# 第 1 章 C++：能力越大责任越大

本章内容

- 了解 C++ 编程错误的根源
- 分析错误的四种方式
- 检查和解决编程错误的方法论
- 鼓励直面和纠正错误

C++ 开发者面临着一项充满挑战的任务：编写和维护成千上万行（甚至数百万行）的代码。幸运的是，虽然这项任务困难重重，但它并非无解。我们可以通过从错误中吸取教训来不断进步，甚至可以从前辈 C++ 开发者的经验中汲取智慧。维护 C++ 代码是一项长期的工程，无法在短短几个月内完全掌握这门语言或彻底理解整个代码库。由于历史决策可能存在疑点或不够理想，应对挫折成为每位 C++ 开发者日常工作中不可或缺的一部分。与此同时，发现他人代码中的优点，并将其融入到自己的代码实践中，是一种有效的成长方式。

本书深入剖析了现代与传统 C++ 代码中可能出现的 100 个常见错误。我们的目标是将这些潜在的陷阱转化为读者成长的契机，帮助大家构建和维护更加清晰、高效的代码库。通过深入理解每种问题的分析方法及其修复策略，您不仅能够更敏锐地识别并解决已有的问题，还能在开发过程中提前规避潜在的风险。现在，就让我们一起踏上这段探索与提升的旅程吧！

## 1.1. 编程错误

每个代码库都可能包含错误，而这些错误的成因多种多样。多年来，C++ 开发者创建了无数应用程序，用以检测错误、过时的编码方式，弥补语言的不足，改进糟糕的设计，并优化低效的工具。如今，我们已经找到了一些方法来改善这些问题！通过研究开发方法学来应对失败，并引入新的语言特性来填补短板，我们正在逐步提升开发体验。

C++ 代码库中蕴含着大量的改进机会。一年或十年前编写的代码，通常基于当时的最佳实践和可用的语言特性。然而，随着时间推移，新的技术方法不断涌现，旧的设计与现代需求之间的不匹配也愈发明显。尽管知识和技能的进步有时能够直接转化为现有代码的改进，但技术债务和技术不一致却也在持续积累。这些问题不仅需要被支持和维护，还需要不断地修改、更新和调试。

遗憾的是，并非所有开发者都有机会或资源对代码进行重新设计或重写。业务场景中，当新功能实现成为优先事项时，改进现有代码往往会被忽视。然而，早期设计中的不良决策和做法所累积的技术债务，会直接影响到新功能的开发。这种技术债务通常表现为脆弱的代码：难以修改和扩展，甚至难以与其他新代码集成。面对这种情况，与其采取非此即彼的极端方式（修复旧代码或完全舍弃），更理想的做法是识别并解决那些反复出现的问题模式。通过这种方式，我们可以逐步优化代码库，同时避免不必要的重构成本。

### 1.1.1. 简单的示例

错误 *57* 中，如果构建了完整的产品并研究了结果输出，则会输出大量警告信息。分析编译输出时，有些警告信息会反复出现。一种选择是专注于特定的编译器警告，并在整个或大部分代码库

中进行修复。以下代码显示了一个最小测试用例，其中通过三种方式发现了地址警告-Waddress。

清单 1.1 中的代码错误地进行了如下尝试：

- 调用函数但不使用必要的括号
- 测试函数调用的返回值
- 比较两个对象的值

清单 *1.1* 带有编译器警告的代码示例

```
int cleanup() {
  return 0;
}

int main() {
  cleanup; // 1

  if (!(cleanup)) { // 2
    ...
  }
  const char* message = "Hello, world";
  if (message == "Hello, world") { // 3
    ...
  }
  return 0;
}
```

注释 1：尝试不使用括号调用函数，结果只是一个地址，不会执行任何代码；需要通过添加括号进行修复

注释 2：尝试使用返回值。使用函数地址，该地址始终为真（非零），否定使其始终为假（零），代码主体永远不会运行；需要通过添加括号来修复

注释 3：将一个 const 指针与另一个 const 指针进行比较，该指针始终为假（不匹配）；需要通过 strcmp 函数或 C++ 字符串来比较进行修复

对于产生警告的代码库，如果团队已经在处理这些问题，并将其作为日常工作的一部分逐步解决，请明确告知其他开发人员和代码审阅者：相关代码在未来会有所改进。同时，可以寻求团队中其他成员的支持与协助，共同完成这一过程。需要注意的是，某些修复工作虽然表面上看起来简单，但实际上可能涉及复杂的情况。所以，在进行修复时，请务必保持谨慎，充分测试每一步改动，确保不会引入新的问题。此外，必须克制自己一次性修复过多问题的冲动（尽管这种想法可能非常诱人），而是以缓慢而稳定的速度，每次只处理可控数量的问题。这种方式能够有效降低风险，确保代码库的稳定性。

另一种可行的方法是，在处理某个代码文件时，同时修复该文件中的多个问题。例如，正在修改某段代码以实现新功能或修复某个特定问题，可以借此机会有选择性地解决该文件中的其他问题。不过，与前面提到的方法一样，务必保持谨慎，避免因过度改动而导致意外后果。通过这种方式，可以有效地提升代码质量，同时减少对代码库整体的影响。

## 1.2. 错误分析

本书讨论每个错误相互独立，但在实际开发中，某些错误可能会对其他代码产生影响。这种关联性增加了诊断和解决问题的复杂性。为了便于理解，本书中的示例简化了这些复杂性，从而缩小了错误的影响范围。大多数章节专注于介绍特定错误的概念，但有些错误由于其广泛的适用性会出现在多个章节中。这表明 C++ 的许多特性之间存在交叉关系，单一特性和错误往往不仅限于影响语言的某一部分。每个错误分析都基于开发和执行中的四个关键特征展开。

### 1.2.1. 正确性

衡量代码是否无误的解决了预期问题。如果代码是正确的，我们就可以确信问题已被妥善解决。尽管优雅且学术化的代码可能更具吸引力，但它们首先必须保证正确性。经验法则是：编写代码时应始终以解决问题为目标。虽然在实际开发中，完全正确的代码并不总是能够实现，但这仍然是一个值得追求的理想目标。

需要注意的是，在某些情况下，错误代码可能会引发未定义行为（UB）。未定义行为是一种极其危险的现象，应不惜一切代价避免（在此播放 Shaft[2] 的主题曲）。UB 指的是语言标准未明确定义的行为，可能导致不可预测的结果、程序崩溃或安全漏洞。编译器可以以任意方式处理 UB，甚至对其进行优化。

### 1.2.2. 可读性

衡量开发人员如何通过代码向其他开发者清晰传达意图的能力。每位开发者都有自己的风格，但如果这种风格妨碍了代码的清晰性和简洁性，则需要进行改进。建议参考通用的或团队内部的风格指南。经验法则是：编写代码时应始终以其他开发者为受众，而非编译器。通过表达自己的思路，使他人能够快速、清楚地理解代码逻辑，并严格遵守强制性的编程风格规范。尽管老练的开发者倾向于编写优雅的代码，但过于复杂的实现可能会给经验不足的开发者带来阅读困难。

### 1.2.3. 有效性

衡量开发人员是否充分利用了语言特性来节省开发时间。紧凑的表达和对某些特性的熟练运用是有效性的核心。开发时间直接影响开发者的生产力，而生产力通常与代码的正确性相关。经验法则是：将正确的语言特性应用于适当的场景。

### 1.2.4. 能效性

能效性衡量的是开发者选择最优方法、算法或数据结构以确保代码与硬件高效协作的能力。这是四个特征中最具挑战性的一个，因为直觉往往需要经过验证和调整才能实现最佳性能。建议使用分析工具定位实际的瓶颈和热点区域。

[2]译者注：电影《杀戮战警》

这一特征也是学术性最强的一个。选择合适的算法通常是提升效率的最佳途径，而这需要一定的计算机科学知识作为支撑。经验法则是：了解不同方法或算法的计算成本，并根据具体问题选择最合适的解决方案。

## 1.3. 以错为师

每位作者都希望自己的作品能够准确把握读者的需求，同时弥补现有文献中的不足。阅读完本书后，你将能够识别书中讨论的错误以及其他潜在问题，理解它们为何成为问题，并掌握解决方法，从而避免在未来犯下类似的错误。

### 1.3.1. 留意错误

提高编程水平的第一步是培养对错误的敏感意识。当我向学生讲解某些例子时，他们常常表现出迷茫的眼神。这些学生努力理解概念，但尚未实现突破。为了减轻他们的困惑，我并未要求他们独立发展这些想法，而是通过展示和解释具体示例，帮助他们对特定问题保持警觉，学会识别这些问题，并探索解决之道。

无论我多么专业，我的能力都离不开那些曾经教导过我、为我提供示例的人。尽管我可能显得知识渊博、聪明睿智，但实际上，我也需要不断学习（和其他人一样）。模式识别在成长过程中至关重要：当我们注意到熟悉的模式时，就获得了应对问题的有效工具。这本书将帮助你识别代码中的错误，并为你提供解决问题的指导。

### 1.3.2. 理解错误

认识到错误的存在是重要的一步，但要深入其本质，我们必须更好地理解为什么它是一个问题。一些长期以来认为是最佳实践的方法，如今可能已经不再适用。编程遵循科学模型：我们提出解决方案，使用它们，并评估其有效性。有时我们成功了，而有时则失败了。

没有人会对这种进步感到惊讶——这是每个人的学习方式。当我们在实践中犯错时，这实际上为填补了知识的空白（或者说是对缺乏知识的认识）。这种填补不仅涉及当前领域，还可能扩展到其他相关领域，深化对编程及其内在关系的理解。我想传达的是，每一个错误都可以成为推动智力与情感发展的契机。只有当我们选择放弃或拒绝学习时，真正的失败才会降临。本书将帮助各位读者深入理解，某种编码方法为何会视为错误。

### 1.3.3. 修正错误

纠正错误是撰写本书的核心目的。虽然修正错误是必不可少的，但这仅仅是整个过程的一部分。如果仅停留在修改源代码的层面，而没有真正理解错误产生的原因，那么我们不过是一台经过训练的“代码修复机器”（甚至可以假设一只经过训练的猴子也能完成类似任务）。

之所以修正错误，是因为我们清楚这些错误为何会导致问题。这种理解对编程团队而言具有重要意义。毕竟，每个人都会犯错，但在将错误推送到生产环境之前，及时发现并修正它们，将为开

发人员、客户以及公司带来更优质的体验。这本书将帮助你掌握如何自行修正错误，并避免未来再次犯下类似的失误。

### 1.3.4. 前车之鉴

检测、理解和修复错误是编写高质量代码的关键环节，但这一过程带来的收获远不止于此。随着经验的积累，各位将更容易分析其他潜在的编程问题。模式识别与理解的能力使你能够发现更多错误和隐患。这就像在健身房锻炼身体——通过全面的训练，身体会逐渐适应各种运动，而不仅仅局限于某一项特定的运动。

对这些错误的深刻理解将使你成为团队中不可或缺的资源，无论是在开发阶段还是代码审查期间。当分享自己的知识时，不仅帮助他人成长，还能促进他们培养类似的技能。

本书的目标是帮助每一位读者做好准备，超越当前的能力边界，为团队和社会作出更大的贡献。

## 1.4. 错迹可循

有经验的 C++ 开发者都会告诉你，C++ 中可能出现的错误远不止 100 种。我所选择的这些错误，是我亲身经历、深入研究或通过其他方式发现的典型问题。可以将这份列表视为我的“前 100 名”，它们大致可分为以下几类。

### 1.4.1. 类型设计

C++ 的核心目标之一是将面向对象编程范式引入流行的 C 语言。类型设计本质上是对新数据类型的定义过程，目的是让编译器能够像处理内置类型一样理解和操作这些新类型。开发者需要完全负责确定新类型的意义、行为，以及其语义。

尽管编译器会严格遵循语言规则，但它对新创建的数据类型几乎没有任何限制。这种自由度虽然强大，但也成为了错误滋生的温床。因此，开发者需要深刻理解这些问题，并主动采取措施加以缓解，而不是单纯依赖语言本身的约束。

### 1.4.2. 程序实现

在编程过程中，设计或实现中的不足往往会导致各种困难。C++ 的灵活性是一把双刃剑，它要求开发者必须深入了解语言特性，才能有效避免或改进潜在的问题。本部分中的一些错误与类型设计相关，但由于其适用范围更广，因此归类在此标题下。正确运用语言特性是生产高质量软件的关键所在。

### 1.4.3. 库的问题

开发者的代码通常只是整个程序的一小部分，而大部分功能依赖于库和函数的支持。此外，某些语言特性（如模板）可以更通用的方式解决问题。例如，模板可以通过描述通用解决方案并让编译器生成最终代码，从而解决一系列相关问题。

库函数提供了丰富的功能，如果开发者尝试自行实现这些功能，不仅耗时耗力，还容易引入错误。然而，正确使用这些库函数的前提是，开发者必须充分理解其用法。如果误解了如何正确调用这些库函数，则可能导致严重问题，甚至影响程序的稳定性和性能。

### 1.4.4. 现代 C++

C++ 的一个重要目标是确保语言在不断发展的同时，仍保持显著的向后兼容性。这种兼容性使得现代编译器能够顺利编译旧代码，但这也并不意味着旧代码一定是高质量或理想的。随着语言的演进，许多过去的错误和问题得到了缓解，并出现了更优的解决方案。

现代 C++ 提供了许多改进，使开发者能够编写更正确、更灵活且更简洁的代码，同时完成与过去相同的功能。这些改进通常会影响多个关键方面：正确性、可读性、效率和性能。然而，为了充分利用这些改进，开发者需要投入时间学习新特性，并理解它们如何改变了我们的思维方式和编码习惯。

### 1.4.5. 旧标准和用法

在这些新特性出现之前编写的代码，显然无法享受到这些改进带来的好处。即使在这些特性推出之后，许多开发者可能仍然沿用旧的编程风格，导致代码功能陈旧、表达能力受限，甚至更容易出错。例如，C++ 字符串可以有效缓解 C 风格字符串的诸多问题，但开发者只有真正使用 C++ 字符串，才能获得这些优势，并避免潜在的错误。

### 1.4.6. 陈旧的经验和过时的知识

随着经验丰富的 C++ 开发者逐渐退出行业，新一代开发者需要填补他们的位置。然而，这些新人不得不面对一些较旧、质量较差且更具挑战性的代码。遗憾的是，许多学术机构提供的现代 C++ 培训并不完善，进一步加剧了这一问题。一些最新的 C++ 教科书中的示例和建议仍然过时，与当前的最佳实践脱节，甚至在实际开发中无效。

有些错误源于过时的方法，有些则是因为糟糕的教学方式，还有一些是因为过去的最佳实践已不再适用。无论这些错误的根本原因是什么，保持对这些问题及其潜在解决方案的敏感性，都将有助于 C++ 开发者编写出更加正确、可读、高效且性能优异的代码。

## 1.5. 本书结构

本书按照时间倒序组织内容，首先探讨现代 C++ 中的问题，随后分析影响现代 C++(C++11 及更高版本）与前现代 C++(C++98 和 C++03）之间的过渡问题。最后，我们将聚焦于前现代 C++，这部分内容涉及许多受限于现代方法的遗留代码库。

书中展示的代码主要基于 C++11 标准之前的代码 (C++98 和 C++03 的结合体)。生成这些代码所使用的编译器是 GNU C++ 工具集 (版本 11.3.0)，运行环境为 Ubuntu 22.04 LTS(通过 Windows Subsystem for Linux [WSL] 2 实现)。选择这一编译器版本是为了确

保在该平台上实现简单性和稳定性，尽管后续版本已发布并将继续更新。较新的编译器版本都可以重现本书中讨论的问题和编码缺陷，但使用较旧版本的编译器可能会导致结果有所不同。

业务限制可能迫使部分读者继续使用 C++98 进行开发，从而错失现代 C++ 功能带来的巨大优势。这确实令人遗憾，但却是现实情况。我的大部分代码库目前仍局限于 C++03，原因在于企业优先考虑稳定性，而操作系统及工具团队对更高版本的审查过程往往漫长且痛苦。

尽管以下列举的许多错误出现在经典 C++（即 C++11 之前的标准）中，但其中一些问题同样存在于现代 C++ 中。由于这些问题对现代开发者具有重要意义，因此会安排在较早的章节中进行讨论。

在本书的后半部分，所有问题及其解决方案均采用经典 C++ 方法进行阐述。尽管某些问题也可能出现在更现代的代码中，但应优先考虑使用现代技术解决它们。然而，对于那些无法使用现代编译器的开发者而言，掌握如何在受限环境中处理这些问题至关重要。这种情况虽然令人遗憾，但确实是当前实际开发环境中的挑战。

## 1.6. 总结

本书可在以下几方面帮助到您：

- 了解一些常见 C++ 编程错误背后的原因。
- 识别安装和运行的 C++ 代码库中发现的常见编程错误。
- 分析代码的正确性、可读性、有效性和能效性。
- 了解在哪里以及如何更改代码，以解决这些常见错误。
- 编写更好的代码，以避免这些错误并在适用时使用现代方法。
- 向他人传授所学到的问题、遇到的问题，以及这些问题的解决方案。

# 第一部分 现代 C++

现代 C++ 的核心在于其对稳健类和类型管理的严格方法。C++ 中的类已不再仅仅是简单的数据容器，而是演变为构建稳健软件架构的基本组件。通过智能指针实现动态内存管理，可以有效减少内存泄漏和悬空指针问题。结合先进的类设计技术与现代类型功能，开发人员能够创建灵活且适应性强的结构，为构建更可靠的应用程序奠定基础。

现代 C++ 引入了诸多强大的编程工具，例如 Lambda 表达式、基于范围的循环，以及上下文关键字，这些工具使得代码编写更加简洁且精确。将这些尚未充分挖掘的功能融入日常编码实践，可以显著降低传统方法中常见的错误和低效问题。通过从传统技术向现代技术过渡，开发人员不仅能以更高的信心和创造力应对现代软件开发中的挑战，还能确保代码的正确性与前瞻性，使其能够适应未来的持续发展需求。

# 第 2 章 更好的现代 C++：类和类型

本章内容

- 移动语义
- 类成员
- 基于范围的循环
- 智能指针

随着 C++11 标准的出现，C++ 增加了一些更改以增强语言功能、改进标准模板库（STL）、提高性能，以及简化语法和表达能力。这些改进包括针对并发性的重大更新，例如线程和任务管理、错误检测机制、时间顺序与日历功能的增强，以及编译时计算的支持。

C++11 中引入的 auto 关键字实现了类型推断，从而简化代码并提高了可读性和有效性。基于范围的 for 循环使集合上的迭代更加简便，提升了代码清晰度，并减少了错误的发生。nullptr 文字通过明确区分空指针和整数零，增强了代码的安全性。

STL 的功能得到了进一步增强。智能指针有助于实现安全的内存管理，并有效减少内存泄漏的风险。移动语义优化了资源处理流程，通过减少不必要的复制操作显著提高了性能。新容器的引入丰富了数据结构选项，提升了数据处理的灵活性和效率。Lambda 表达式以其简洁而富有表现力的特点，进一步提高了代码的可读性和有效性。

性能方面的改进显著提升了程序的效率和资源利用率。右值引用能够更高效地处理临时对象，最大限度地减少不必要的开销，从而缩短执行时间并提高资源管理效率，进而提升 C++ 程序的整体性能。

C++11 在语法简化和表达能力方面也取得了重要进展，使代码更具可读性和可写性。Lambda 表达式支持内联函数定义，提高了代码清晰度并减少了对辅助函数的需求。可变参数模板可以灵活处理可变数量的模板参数，实现了更通用和可重用的代码。基于范围的 for 循环为迭代容器提供了更直观的语法。这些改进使开发人员能够编写更具表现力、更高效和更灵活的代码。

本章和接下来的两章讨论了这些改进中的大部分。未涵盖重大改进和增强，但可以在许多书籍、课程和互联网搜索中找到。C++11 标准中的增强范围非常值得研究，它们为 C++17 和 C++20 标准提供了基础，添加了增强、添加、弃用、删除和新语言功能。随着时间的推移，C++ 在不断演进和完善。

现代 C++ 无法避免开发人员犯错，但它支持一些使错误变得更加困难的功能。例如，基于范围的 for 循环不受 off-by-one[3] 的影响。以下所列的优点旨在通过改进语言特性来解决经典错误，即使只是其中的一小部分改进，也能在开发过程中带来显著的好处。

## 2.1. 错误 1： 未使用移动语义

这个错误主要针对性能。因为数据通常会被复制，所以将信息从一个区域传输到另一个区域可能代价高昂，从而产生两个（或更多）的数据版本。

[3]译者注：一种特殊的溢出漏洞

## 问题

数据必须在一个地方创建或构建，而在另一个地方进行分析或操作。将数据从一个地方复制到另一个地方效率很低，复制少量数据无关痛痒，但复制大量数据就会造成问题，尤其是频繁复制的情况下。

如果多个对象必须拥有数据，则可将数据共享。可以通过传递指针副本来实现大量共享数据，从而允许各种代码段访问数据，而无需复制数据。并且，数据有独占性。因为指针是一种共享机制，所以传递这种数据的指针可能很危险。现代 C++ 为这种情况提供了独享所有权的智能指针。

数据通常由值，而非指针拥有和管理。清单 2.1 中，TextSection 在一个实例中创建并初始化，然后传递给另一个实例。p1 对象独占 TextSection，所有权和数据“转移”到 p2，这是意图。结果是 p2 拥有数据的单个副本，但成本高昂。请注意，源对象的数据和 headers 被销毁，保留了唯一所有权要求，但打破了传递 const 引用的做法。

代码清单 *2.1* 保持唯一性的赋值操作符，但代价高昂

```
class TextSection {
  // 假设有一段巧妙的实现
};

class Page {
private:
  TextSection* headers;
  TextSection* body;
public:
  Page(TextSection* h) : headers(h), body(new TextSection()) {}
  Page(Page& o);
  ~Page() { if (body) delete body; }
};

Page::Page(Page& o) {
  if (this == &o)
    return;
  body = new TextSection(*o.body);
  delete o.body;
  o.body = 0;
  headers = o.headers;
  o.headers = 0;
}

int main() {
  Page p1(new TextSection());
  Page p2 = p1; // 1
  return 0;
}
```

注释 1：未定义行为的赋值操作

**分析**

数据的副本归 p2 所有，但并非原始数据。这会导致产生两份副本，并且需要花时间来复制。如果数据量很大，复制的时间可能会相当长。复制构造函数会销毁复制的数据，从而将副本数减少为一份，但无法最大限度地降低复制成本。转移所有权的努力值得称赞，但可以更高效。我们需要的是一种无需复制，即可转移数据的方法。

**解决**

现代 C++ 提供了一种称为右值引用的新引用类型（“右引用”），引用操作符右侧的值。右值是赋值的源，不能是赋值的目标。尽管右侧不是赋值目标，也可以对其进行修改。临时变量也是右值，所以修改它们是没有意义的，但“窃取”它们的值是合法的。新语法将包含一个双引用字符（&&）来指示移动语义。

该语言提供了添加移动构造函数和移动赋值操作符的功能。这些新功能旨在在资源从源转移到目标时保留独占所有权，并增加了移动数据而不是复制数据的好处。由于目标在移动后拥有转移的数据，所以源中数据失效，并避免错误访问。

清单 2.2 开销低且保持唯一性的赋值操作符

```
class TextSection {
  // 假设有一段巧妙的实现
};

class Page {
private:
  TextSection* headers;
  TextSection* body;
public:
  Page(TextSection* h) : headers(h), body(new TextSection()) {}
  Page(const Page& o) : headers(o.headers), body(new // 1
  TextSection(*o.body)) {}
  Page(Page&& o) : headers(o.headers), body(o.body) { // 2
    o.headers = nullptr; // 3
    o.body = nullptr;
  }
  ~Page() { delete body; }
};

int main() {
  Page p1(new TextSection());
  Page p2 = std::move(p1); // 4
  return 0;
}
```

注释 1：可移动的复制构造函数

注释 2：源资源重新分配给目标。

注释 3：源资源失效。

注释 4：这看起来有点奇怪，但语义正确。

移动意味着源对象将其内容释放到目标对象。一些实现使用交换函数将源数据移动到目标，将

目标数据移动到源。由于数据重新分配，源对象不再拥有其原始数据，但拥有目标的数据通常可以接受。如果不使用交换操作，最好的方法是确保源中所有移动的数据无效。

源对象未破坏的情况下，其先前的数据不能访问。因此，无效化源对象中的数据，以避免在移动后的无意使用。有时，可以将指针设置为 nullptr(或 0)。

另一个不错的功能是，当提供移动构造函数和赋值操作符时，编译器会尽一切努力使用它们，开发人员不必决定何时可以使用。当源是左值（操作符左侧的可赋值值）时，将使用标准复制构造函数和赋值操作符。

std::move 函数模板对于正确移动至关重要。业界对此的流行语是“移动不动”(Move doesn’t move.)。那么，移动为何如此重要？移动将左值转换为右值，这使得其符合移动语义。如果不这样做，将发生标准赋值，而不是移动赋值。使用复制和移动操作符，类可以提供共享赋值（或构造）或独占赋值（或构造），这使得语义更加清晰。

**建议**

- 向类添加移动构造函数和赋值操作符，以独占方式转移资源所有权。
- 通过包含移动构造函数和移动赋值操作符，将三规则（复制构造函数、赋值操作符、析构函数）扩展为五规则。

## 2.2. 错误 2: 使用空异常规范

这个错误与有效性、性能，以及可读性有关。异常规范告诉编译器，函数可以抛出哪些异常。

**问题**

自 2011 年以来，异常规范已被弃用，因其有效性和性能未能让开发人员更轻松。规范很复杂，而且令人困惑，以至于现代 C++ 在 C++11 规范中弃用，并在 C++17 标准中完全删除。先要了解完整的故事，请仔细阅读 P0003R5 (https://mng.bz/N1mE)。然而，源于异常规范的想法仍然有其价值和意义。

异常是将正常功能流程与错误处理区分开的绝妙方法，但这种灵活性是有代价的。大量使用异常和不使用异常一样讨厌，而我们必须找到一个合理的平衡点，既能提供明确的错误处理，又能提供良好的性能。许多函数不会抛出异常，它们不应该因为在运行时检查是否处理异常而受到性能惩罚。异常规范有一种情况是指定了 throw()，这意味着函数不会抛出异常。此规范的目标是提高性能。随着规范的删除，这种方法也消失了。

删除异常规范的提议中承认了 throw() 的价值，并主张不要删除该功能，而是对其进行转换。C++11 标准添加了一个新操作符，该操作符实际上执行了与空抛出规范相同的操作。引入了 noexcept，其含义与空抛出相同，允许编译器执行优化以删除运行时的异常检查。以下代码演示了前现代 C++ 方法，告诉编译器消除不必要的异常检查。

清单 2.3 传统 C++ 中表示不抛出异常函数的方式

```
1  const double pi = 3.1415927;
2  struct Circle {
3    double radius;
4    Circle(double r) : radius(r) {}
```

```
  double perimeter() const throw() { return 2 * pi * radius; } // 1
  double area() const throw() { return pi * radius * radius; } // 1
};

int main() {
  Circle c(3);
  std::cout << "perimeter " << c.perimeter() << ", area " << c.area() << '\n';
  return 0;
}
```

注释 1：因为没有抛出异常，可尝试进行优化

**分析**

清单 2.3 展示了传统方法，但会在现代 C++ 中引发警告。因为删除异常规范的提案认为，空的 throw() 等同于 noexcept(true)。该参数是一个表达式，如果将其设置为 true，则编译器确信该函数不会引发异常。还请各位不要欺骗编译器。

如果标记为 noexcept 的函数撒谎并抛出异常，则会调用 terminate 函数，这会立即终止整个程序，并且不会执行堆栈展开（不会调用析构函数！）。开发人员必须说实话，否则编译器会使用未定义（意外异常）行为对谎言进行惩罚。

**解决**

清单 2.4 使用现代方法，用新的语言关键字 noexcept 替换 throw()，更新代码应该很简单。

```
struct Circle {
  double radius;
  Circle(double r) : radius(r) {}
  double perimeter() const noexcept { return 2 * std::numbers::pi * radius; } // 1
  double area() const noexcept { return std::numbers::pi * radius * radius; } // 1
};

int main() {
  Circle c(3);
  std::cout << "perimeter " << c.perimeter() << ", area " << c.area() << '\n';
  return 0;
}
```

注释 1：因为没有抛出任何异常，可进行更好的优化尝试

接受一个参数的构造函数称为转换构造函数。清单 2.4 中的代码可以将 double 值转换为 Circle 值。当编译器查找此类转换时，会确定是否有可用的构造函数或转换操作，如果找到一个或多个，则使用最合适的。

**NOTE**

现代 C++ 提供了 explicit 关键字，以避免在隐式转换中使用单参数构造函数。有时，需要隐式转换行为，但不受控制的转换构造函数应该受到限制。向构造函数添加 explicit 关键字可避免编译器对其参数进行转换。

**建议**

- 将 throws() 更改为 noexcept 以使代码现代化。
- 确保声称不会抛出异常的函数，真的不会抛出异常。

## 2.3. 错误 3： 未重写派生的虚函数

这种错误会影响效率，并且会影响可读性。在派生类中实现虚函数，提供自定义继承行为的方法。错误的派生虚函数已经是一个很大的问题，所以标准化委员会同意为编译器添加功能，来避免这种情况发生。

**问题**

包含虚函数的类旨在被继承，virtual 关键字表明了这一意图。在派生类中实现的虚函数，允许在基类中修改行为（或完全替换），这种方法是多态行为的核心。

在大型代码库中，基类和派生类可能相距甚远。如果距离很远，要知道某个方法是否是虚拟的就很困难了。派生类不必在其函数声明中重复 virtual 关键字，但这种重复是确定给定方法为虚的合理方式。但这种文档无法解释虚函数的最基类版本在哪里，开发人员只能探索类型层次结构才能弄清楚。

清单 2.5 展示了一个快速而愉快的尝试，试图在周五晚上添加一些功能。代码编译成功，开发人员很满意，代码发布成功。遗憾的是，下周，客户抱怨 Square 对象无法正常工作。开发人员很惊讶，因为一切“看起来都很好”。

清单 2.5 看起来不错且编译无误的代码

```
struct Shape {
  std::string type;
  explicit Shape(const std::string& t) : type(t) {}
  virtual double area() const { return 0; }
};

struct Square : public Shape {
  double side;
  Square(double s) : Shape("square"), side(s) {}
  double aria() const { return side*side; } // 1
};

const static double pi = 3.1415927;
struct Circle : public Shape {
  double radius;
  Circle(double r) : Shape("circle"), radius(r) {}
  double area() const { return pi * radius * radius; }
};

int main() {
  Square s(2);
  Circle c(2);
```

```
  std::vector<Shape*> shapes;
  shapes.push_back(&s);
  shapes.push_back(&c);
  for (int i = 0; i < shapes.size(); ++i)
    std::cout << shapes[i]->area() << '\n';
  return 0;
}
```

注释 1：优美文字中也可能会有错别字。

### 分析

问题很简单：一个手误。因为没有犯任何语法错误，编译器会将新函数添加到类中。但开发人员应该注意，成功编译并不意味着正确重新定义了相应虚函数。

错误的派生类虚函数在语法上没有错误——至少不是因为命名错误！而开发人员是否打算重新定义虚函数，编译器并不在乎。

### 解决

C++11 标准添加了一个新关键字，称为 override。虚函数的问题非常普遍，必须采取一些措施来应对。这个新关键字添加到方法的参数列表之后，该方法应该是虚函数的重新定义。当编译器发现带有 override 关键字的方法，与某些基类中的虚函数不匹配，则会报错。这个关键字不是必需的，但其实现了两个重要的功能。首先，记录了此类重新定义了虚函数，从而提高了可读性。其次，编译器验证该方法是否重新定义了已知的虚函数。

因为在 aria 函数后添加了关键字，所以清单 2.5 中的代码无法通过编译。当编译器卡住时，开发人员可以看到错误；很快，问题就解决了。周五晚上的啤酒味道从来没有这么好过。

override 从技术上讲是一个上下文关键字，而不是常规关键字。仅在特定上下文中才是关键字（例如，在方法的参数列表之后）。在其他上下文中，它是有效的（但不推荐！）标识符。之所以做出这种区分，是因为 override 在许多现有代码库中都作为标识符存在。标准化委员会对此非常友善。

清单 *2.6* 使用 *override*

```
struct Shape {
  std::string type;
  Shape(const std::string& t) : type(t) {}
  virtual double area() const { return 0; }
};

struct Square : public Shape {
  double side;
  Square(double s) : Shape("square"), side(s) {}
  double area() const override { return side*side; } // 1
};

struct Circle : public Shape {
```

```
  double radius;
  Circle(double r) : Shape("circle"), radius(r) {}
  double area() const override { return
    std::numbers::pi * radius * radius; } // 1
};

int main() {
  Square s(2);
  Circle c(2);
  std::vector<Shape*> shapes;
  shapes.push_back(&s);
  shapes.push_back(&c);
  for (auto& shape : shapes)
    std::cout << shape->area() << '\n';
  return 0;
}
```

注释 1：添加 override，让编译器检查虚函数的情况

**建议**

- 使用 override 关键字来表明重新定义继承虚函数的意图。
- 使用 override 让编译器强制重新定义虚函数。
- override 是一个上下文关键字，但请避免在其他地方使用它——会使代码变得复杂。

## 2.4. 错误 4： 编写简单的或隐藏不需要的提供类成员

这个错误影响了有效性和可读性。当定义一个类时，一些成员若开发人员未编写，则由编译器提供。

**问题**

编译器可以轻松处理没有开发人员编写的成员来处理构造、复制和赋值的简单类。开发人员未提供的内容，编译器将自动生成。通常，这是有帮助的，并且提供的版本可以完成必要的操作；当自动提供的方法不正确或不足时，开发人员必须编写自己的版本。

如果提供了构造函数，编译器将不会生成默认构造函数。如果编写了复制构造函数、移动构造函数或析构函数，则不会自动生成这些函数。如果不应提供某些成员，一种简单的方法是将它们设为私有，且不实现任何功能。

这个问题的另一面是，一些开发人员在不明确需要时编写了许多这些提供的成员。开发人员编写了一个简单的 Container 类，提供了一些基本功能。此外，他们决定可以复制，但不能赋值。因此，以下代码提供了默认（必需）和复制构造函数，并隐藏了赋值操作符。

清单 2.7 带有显式默认构造函数的简单类

```
class Container {
private:
  std::vector<int> values;
```

```
  Container& operator=(const Container& o); // 1
public:
  Container() {}
  Container(const Container& o) values(o.values) {}
  void add(int n) { values.push_back(n); }
};

int main() {
  Container c1;
  c1.add(42);
  Container c2(c1); // 2
  // c2 = c1; // 3
  return 0;
}
```

注释 1：隐藏起来，赋值操作无法使用

注释 2：调用复制构造函数

注释 3：赋值会出现错误

**分析**

代码可以运行，但编写和阅读起来都很有挑战性。构造函数和赋值操作符通常是公共成员，将它们隐藏在私有部分中会很别扭。此外，这种方法不能很好地传达默认构造函数或赋值操作符的用途——必须熟悉该模式才能理解它。如果此示例稍微复杂一些，则默认构造函数中的代码会更复杂。

**解决**

更好的方法是使用 `=delete` 和 `=default` 关键字。`=delete` 关键字表示成员是故意省略的，不能使用它。`=default` 关键字表示编译器应该生成一个以默认方式运行的成员。这种用法的价值在于，类可以更改其实现，而不会影响用户的代码，前提是这些更改对编译器非常明确。

清单 *2.8* 带有默认和删除成员的简单类

```
class Container {
private:
  std::vector<int> values;
public:
  Container() = default; // 1
  Container(const Container& o) : values(o.values) {}
  Container& operator=(const Container& o) = delete; // 2
  void add(int n) { values.push_back(n); }
};

int main() {
  Container c1;
  c1.add(42);
  Container c2(c1);
  // c2 = c1; // 3
  return 0;
```

```
}
```

注释 1：编译器知道自己在做什么，会自动编写默认构造函数
注释 2：没有必要隐藏该操作符；编译器会阻止使用
注释 3：赋值错误

**建议**

- 使用 =delete 消除成员，从而删除该功能。
- 使用 =default 让编译器生成成员，节省时间并消除出错的可能。

## 2.5. 错误 5： 未使用类内初始化器

此错误主要针对可读性，但可能会提高效率。许多编程语言能在类定义中初始化实例变量；C++ 直到 C++11 才可以。

**问题**

大多数开发人员发现，编写带有默认值的构造函数很方便。这些值通常很有意义，提供它们的唯一原因是默认值对于特定实例并不正确。

通常，开发人员会在构造函数的参数列表中使用默认值，如清单 2.9 所示。这种方法效果很好，但存在可读性问题。假设类中有多个构造函数，并且有许多实例变量。其中一些构造函数会默认很少更改的值，但读者可能需要帮助来确定给定实例变量的值。清单 2.9 中的代码易于阅读，请考虑它在具有 4、5 个或更多参数的类。作为示例，来阅读此代码，看看其中几个参数的默认值：

```
Demo(int n=0, double d=1.0, double e=0.0, double f=10.5, bool b=false) :
n(n), d(d), e(e), f(f), b(b) {}
```

默认值应限制在可读且易懂的列表中；参数过多则表明类的责任过重，应将其重构为多种数据类型。以下代码没有犯此错误，但仍需改进可读性。

清单 2.9 在参数列表中使用默认值

```
class Complex {
private:
  double real;
  double imag;
public:
  Complex(double r=0, double i=0) : real(r), imag(i) {} // 1
  double getReal() const { return real; }
  double getImag() const { return imag; }
};

int main() {
  Complex c1;
  Complex c2(3);
  Complex c3(-2, -2);
  return 0;
```

```
}
```

注释 1：使用默认参数

### 分析

虽然这段代码一切正常，但当提供多个构造函数时，可读性会有所下降。此外，编写的代码越多，引入错误的可能性就越大。如果两个或多个构造函数中的默认值参数不匹配，读者将很难确定实例变量的正确值。

### 解决

现代 C++ 提供了一种不依赖默认参数值的替代方案。考虑到可读性，其他一些语言提供默认值，C++ 添加了类内初始化器。类中声明的每个实例变量都可以分配一个默认值，方法是使用赋值操作符或括号初始化形式（称为统一初始化）。此形式可避免歧义和自动窄化转换，并且更加一致。

清单 *2.10* 使用类内初始化以提高可读性

```
class Complex {
private:
  double real = 0; // 1
  double imag{0}; // 2
public:
  Complex() {}
  Complex(double r, double i) : real(r), imag(i) {}
  double getImag() const { return imag; }
};

int main() {
  Complex c1; // 3
  // Complex c2(3); // 4
  Complex c3(-2, -2);
  return 0;
}
```

注释 1：使用赋值操作符进行类内初始化

注释 2：使用带括号的类内初始化

注释 3：使用默认值

注释 4：错误，不知道该值打算初始化哪个成员变量

如果构造函数为类内初始化变量提供了一个值，则将使用所提供的值进行初始化。如果没有提供值，则使用类内的值。

### 建议

- 选择类内初始化来本地化实例变量的默认值。
- 使用传统 C++ 时，才在构造函数参数列表中使用默认值。

## 2.6. 错误 6： 过度使用基于索引的循环

这种错误会影响效率和可读性，并且会对性能产生负面影响。使用循环对容器进行索引是一种常见的习惯做法，但这种传统方法过于复杂。

### 问题

开发人员经常必须遍历容器才能访问每个元素，这是通过创建一个循环控制变量来实现的，该变量从第一个索引开始，并在每次循环中单调递增 (+1)。最后一个索引始终是容器的长度减一。清单 2.11 演示了一个通用的 sum 函数，它接受一个值 vector，并返回它们的总和。我们假设这些是测试分数，然后计算平均值。循环没有什么特别之处；它是索引容器的通用解决方案。还有什么比这更简单的呢?

清单 *2.11* 使用索引值遍历容器

```
template <typename T>
T sum(const std::vector<T>& values) {
  T sum = (T)0;
  for (int i = 0; i < values.size(); ++i) // 1
    sum += values[i];
  return sum;
}

int main() {
  std::vector<int> values;
  for (int i = 0; i < 10; ++i)
    values.push_back(10*i + i); // 1
  std::cout << sum(values) << '\n';
  return 0;
}
```

注释 1：传统基于索引的方法

### 分析

清单 2.11 中的代码可以工作，并且反映了当前代码库中的大多数循环。对于其传统实现，需要编写一些代码才能实现，并且专注于实现循环机制。即使一个人非常熟悉解决方案，读起来也很会尴尬。且需要输入的代码字符越多，引入错误的几率就会变大。此外，对于那些不熟悉这种习惯用法的人来说，可能会将延续条件改为

```
i <= values.size()
```

我们知道这是错误的。直觉上感觉正确，但实际上却是一个差 1 错误。

### 解决

现代 C++ 提供了基于范围的 for 形式来简化这种模式。基于范围的 for 删除了延续条件和更新部分，并修改了初始化部分。将冒号读作“in”[4]，例如“对于值中的（每个）元素，执行某事”，其中“某事”是循环体。

[4]译者注：类似于 Python 语言。

循环的机制很简单。声明一个变量，该变量按顺序分配每个容器元素的值。值就像索引方法一样，从索引 0 处的元素开始赋值，然后是索引 1，依此类推，直到最后一个元素。循环的每次迭代都会将元素复制到变量中。可以在循环中修改变量，但由于是副本，相应的修改不会影响容器中的元素值，可将此循环视为只读语义。如果需要修改元素值，请对变量使用引用类型。

若元素类型是类，则每次都会调用复制构造函数——这可能比必要的代价高得多。最好将变量设为元素类型的引用。这时只复制数据的位置，从而最大限度地减少数据量。除非循环会修改元素，否则请使用 const 关键字。对于数字类型，如清单 2.12 所示，引用可能过多。当元素类型是类时，这种方法性能会很好。某些情况下可以使用指针变量，但引用形式消除了对指针语法的需要，从而提高了可读性。

清单 *2.12* 使用基于范围的 *for* 循环遍历容器

```
template <typename T>
T sum(const std::vector<T>& values) {
  T sum {}; // 1
  for (const T& value : values) // 2
    sum += value;
  return sum;
}

int main() {
  std::vector<int> values;
  for (int i = 0; i < 10; ++i) // 3
    values.push_back(10*i + i);
  std::cout << sum(values) << '\n';
  return 0;
}
```

注释 1：使用初始化程序获取类型的正确零形式
注释 2：基于范围的 for 循环使用简单
注释 3：基于索引的 for 循环感觉笨重

**建议**

- 使用基于范围的 for 语句，可以消除传统基于索引循环的大量编码工作。
- 元素值会复制到变量中，循环本质上为只读。
- 如果元素复制成本高昂，请使用引用变量，最大程度减少数据移动。

## 2.7. 错误 7：未使用 nullptr

这个错误与可读性有关，并且在一定程度上与有效性有关。NULL 宏继承自 C，而在前现代 C++ 中最好使用用 nullptr 初始化指针。

**问题**

清单 2.13 展示了一个 Student 类的简单模型。其中一个实例变量初始化为 0。此外，由于代码需要指针，其也初始化为 0。

清单 *2.13* *0* 值的混合使用

```
struct Student {
  std::string name;
  int id;
  double gpa;
  Student(const std::string& name, int id, double gpa) : name(name),
          id(id), gpa(0) {
    if (gpa < 0)
      throw std::invalid_argument("gpa is negative");
  }
};

int main() {
  Student* sammy = 0; // 1
  Student* ginny = 0;
  Student* gene = 0;
  sammy = new Student("Samuel", 0, 3.75);
  ginny = new Student("Virginia", 1, 3.8);
  gene = new Student("Eugene", 2, 0);
  return 0;
}
```

注释 1：比 NULL 宏好点，但没好多少

**分析**

经典 C++ 需要表示不代表任何对象的指针，所以将指针初始化为 0 是个好的方法。但是，0 不是指针，其可能会导致混乱。虽然许多开发人员都熟悉这种用法，而作为指针来说，0 比 NULL 宏更难阅读。因为编译器足够聪明，可以获取整数文字值并正确使用指针。

**解决**

现代 C++ 提供了 nullptr，即表示空值，且专用于指针。其优点是意图清晰，易于理解。其另一个明显优点是，编译器不会将其与类型为 int 的函数参数混淆。

清单 *2.14* 使用 *nullptr* 明确指针状态

```
struct Student {
  std::string name;
  int id;
  double gpa;
  Student(const std::string& name, int id, double gpa) : name(name),
          id(id), gpa(0) {
    if (gpa < 0)
      throw std::invalid_argument("gpa is negative");
  }
};

int main() {
```

```
  Student* sammy = nullptr; // 1
  Student* ginny = nullptr;
  Student* gene = nullptr;
  sammy = new Student("Samuel", 0, 3.75);
  ginny = new Student("Virginia", 1, 3.8);
  gene = new Student("Eugene", 2, 0);
  return 0;
}
```

注释 1：比 NULL 宏好，比 0 值明确。

**建议**

- 使用 nullptr 初始化未指向有效对象的指针。
- 删除分配内存后，最好将相应指针赋值为 nullptr，以避免出现使用错误。

## 2.8. 错误 8：未使用 unique_ptr 独占所有权

此错误会影响正确性，并增强可读性。RAII 模式应该管理动态资源，但可能需要实现多个管理类。经典 C++ 提供了 auto_ptr 选项，由于容易出错而弃用，后来在标准中删除。

**问题**

动态资源的独占所有权对于跟踪独占资源至关重要。此类资源无法复制，这样做则表示源目标独占此资源。此时，拥有和管理资源的引用尚不确定。此外，有两种通过修改或删除来管理方式；后者可能会导致双重删除错误。第一个 delete 可以正常工作，但第二个会出现未定义行为。

以下代码展示了管理独占动态资源的简单案例。由于资源在一个地方创建，在另一个地方操作，因此必须传递其指针值。尽管代码按顺序编写，但阅读时要将每行分隔开，这就会造成干扰，导致流程不明显。

清单 2.15 管理独占资源的失败尝试

```
struct Buffer {
private:
  int* data;
public:
  Buffer() : data(new int[10]) {}
  ~Buffer() { delete[] data; }
};

int main() {
  Buffer* b1 = new Buffer();
  Buffer* b2 = b1; // 1
  if (b1)
    delete b1; // 2
  if (b2)
    delete b2; // 3
  return 0;
```

```
}
```

注释 1：糟糕！独占资源不应复制或分配
注释 2：这似乎合理；原始所有者
注释 3：这似乎不合理；第二位所有者?!

**分析**

开发人员的本意是在一个地方创建动态资源，然后稍后将其转移给另一个所有者。但与许多善意的想法一样，代码需要更好地落实。由于复制指针，因此变为共享而非独占。原始所有者 b1 删除了资源看起来合理，但忽略了所有权转移的尝试。如果所有权正确转移，则删除 b2 也合理。但由于资源已删除，此尝试会导致未定义行为。

Buffer 结构设计合理，已经实现了 RAII 模式，所以能够正确管理其动态资源。但还有改进的空间。

**解决**

正常和异常情况下管理指针很复杂，RAII 模式旨在处理这种情况，但需要对每个动态资源进行实现。auto_ptr 认识到了这个问题，并试图提供通用解决方案；然而，这种拙劣的努力导致了重大问题，并产生了很多奇怪的代码。

现代 C++ 提供 std::unique_ptr 以实现独占所有权。此模板设计精良，在任何情况下都能按预期工作。清单 2.16 中的代码演示了它与指针和数组的用法（auto_ptr 无法处理数组）。Buffer 类的内部数据使用数组形式，并让 unique_ptr 管理动态数据。此类不需要用户编写的析构函数来删除数组。

与使用 new 相比，std::make_unique 函数模板（见清单 2.16 ）更受欢迎，具有更高的异常安全性，避免了裸指针的使用，并且性能更好。

所有权转移中，代码使用 std::move 函数模板。虽然这似乎将对象从一个所有者移动到另一个所有者，但事实并非如此—换句话说，“移动不会移动”。std::move 所做的是将左侧（左值）引用转换为右侧（右值）引用。右值引用通常是一个临时值，不可寻址，保存值，并复制到接收实体。移动操作使源实体丢弃所有权，编译器将知晓何时使用右值实体的分配（或副本），并调用移动语义。值从源移动到目标对象，使源（概念上）为空值。一些实现使用交换来传输值，可使源保留目标的先前值。无论如何，除非重新初始化，否则不应使用源实体。我认为，最好不访问和不使用移动后的实体。

清单 *2.16* 使用 *unique_ptr* 实现独占所有权

```
struct Buffer {
private:
  std::unique_ptr<int[]> data; // 1
public:
  Buffer() : data(std::make_unique<int[]>(10)) {} // 2
};

int main() {
  auto b1 = std::make_unique<Buffer>(); // 3
  auto b2 = std::move(b1); // 4
```

```
  return 0;
}
```

注释 1：数组形式
注释 2：初始化数组
注释 3：指针形式，使用首选的 make_unique 函数模板
注释 4：使用 std::move 函数模板进行所有权转移（非复制或赋值）

清单 2.16 中提供的解决方案更加简洁，可能需要一些练习才能掌握。这种方法的价值比之前的方法都高效和简洁。可以使用 auto 关键字来指定 b1 和 b2 的类型，编译器可以推断它们的类型。

**建议**

- 拒绝对动态资源使用原始（裸）指针，对除共享资源之外的指针使用 unique_ptr。
- 使用 unique_ptr 表示独占所有权，分配会将资源从源实体转移到目标实体，从而使源实体不可用。
- 出于效率和简单性的考虑，最好使用 unique_ptr，因为使用 shared_ptr 的成本更高。

## 2.9. 错误 9：未使用 shared_ptr 共享所有权

这个错误主要涉及正确性和有效性。共享指针比原始指针成本更高，当了解了该方法，代码的可读性会大大增强。

**问题**

有几种数据结构，需要处理以各种方式排列的动态分配实体。这些动态资源最好使用 RAII 进行管理，当需要多次引用实体时，了解何时以及谁应该释放资源会变的很复杂。

清单 2.17 是开发人员的典例，他在大学课堂上学会了使用指针，并实现了双端队列。代码运行良好，但在管理实体资源时不鼓励使用原始指针。如果犯了错误，内存泄漏无法避免。此外，在抛出异常的应用程序中，管理的要求会更高。

清单 2.17 使用原始指针实现的双端队列（deque）

```
class Node {
public:
  Node(int value) : data(value), prev(NULL), next(NULL) {}
  int data;
  Node* prev;
  Node* next;
};

class Deque {
private:
  Node* front;
  Node* back;
public:
  Deque() : front(NULL), back(NULL) {}
```

```
15    ~Deque() { // 1
16      while (front != NULL) {
17        Node* temp = front;
18        front = front->next;
19        delete temp;
20      }
21    }
22    void push_front(int value) {
23      Node* newNode = new Node(value);
24      if (front == NULL)
25        front = back = newNode;
26      else {
27        newNode->next = front;
28        front->prev = newNode;
29        front = newNode;
30      }
31    }
32    void push_back(int value) {
33      Node* newNode = new Node(value);
34      if (back == NULL)
35        front = back = newNode;
36      else {
37        newNode->prev = back;
38        back->next = newNode;
39        back = newNode;
40      }
41    }
42    void pop_front() {
43      if (front != NULL) {
44        Node* temp = front; // 2
45        front = front->next;
46        if (front != NULL)
47          front->prev = NULL;
48        else
49          back = NULL;
50        delete temp; // 3
51      }
52    }
53    void pop_back() {
54      if (back != NULL) {
55        Node* temp = back; // 2
56        back = back->prev;
57        if (back != NULL)
58          back->next = NULL;
59        else
60          front = NULL;
61        delete temp; // 3
62      }
```

```
  }
};

int main() {
  Deque deque;
  deque.push_front(3);
  deque.pop_front();
  return 0;
}
```

注释 1：用于消除剩余动态实体的强制析构函数
注释 2：对单个节点的两个引用
注释 3：删除过期实体

### 分析

要将节点放入双端队列或从双端队列中移除节点，至少需要两个指针来引用该节点。如果使用其他指针（索引队列），可多个指针引用单个节点。原始指针为多个指针引用单个节点提供了一种简单的方法，但这种方式使节点的管理变得复杂。

需要删除多个节点时，每个需要删除的节点都必须正确编码，否则很有可能出现内存泄漏，所以析构函数和这些单独的删除是一种认知负担。首先，必须正确设计和编码双端队列。其次，强制双端队列支持资源管理。这种方法会让代码变得更复杂、更难懂。

### 解决

消除管理动态资源可以解决其中一个问题。现代 C++ 使用 `std::shared_ptr` 来管理资源，尽可能使用 `std::make_shared`，而非 `new` 操作符。共享指针使用 RAII 模式，可对单个资源进行多次引用。`std::shared_ptr` 通过跟踪对资源的引用计数来管理资源。每次复制或分配时计数都会增加，每次超出范围的共享指针时计数都会减少。当计数达到 `0` 时，资源将删除。因此，不需要用户编写管理代码。太棒了！

`std::unique_ptr` 成本很低，应该用于不需要共享的资源。`std::shared_ptr` 成本相对较高，应该只在需要共享时使用。首先，共享指针（参见清单 2.18）比独占指针大，因其需要维护一个计数器。因此，只在必要时使用相应的智能指针，并尽可能优先使用独占指针。

清单 *2.18* 使用 *shared_ptr* 实现的双端队列 *(deque)*

```
class Node {
public:
  Node(int value) : data(value) {}
  int data;
  std::shared_ptr<Node> prev; // 1
  std::shared_ptr<Node> next; // 1
};

class Deque {
private:
  std::shared_ptr<Node> front;
  std::shared_ptr<Node> back;
```

```
public:
  Deque() : front(nullptr), back(nullptr) {} // 2
  void push_front(int value) {
    std::shared_ptr<Node> newNode = std::make_shared<Node>(value);
    if (front == nullptr)
      front = back = newNode;
    else {
      newNode->next = front;
      front->prev = newNode;
      front = newNode;
    }
  }
  void push_back(int value) {
    std::shared_ptr<Node> newNode = std::make_shared<Node>(value);
    if (back == nullptr)
      front = back = newNode;
    else {
      newNode->prev = back;
      back->next = newNode;
      back = newNode;
    }
  }
  void pop_front() {
    if (front != nullptr) {
      front = front->next;
    if (front != nullptr)
      front->prev = nullptr;
    else
      back = nullptr;
    }
  }
  void pop_back() {
    if (back != nullptr) {
      back = back->prev;
      if (back != nullptr)
        back->next = nullptr;
      else
        front = nullptr;
    } // 2
  }
};

int main() {
  Deque deque;
  deque.push_front(3);
  deque.pop_front();
  return 0;
}
```

注释 1：shared_ptr 会隐式初始化

注释 2：不需要析构函数和删除函数

std::shared_ptr 的另一个优点是不需要显式初始化，在构造时即初始化为 nullptr。

使用共享指针时会出现一个问题——循环引用，即一个实体使用共享指针引用另一个实体，反之亦然。共享指针会跟踪对其共享实体的引用数量。每次构造引用它的共享指针时，计数都会增加，每次销毁共享指针时，计数都会减少。如果计数达到零，则删除共享资源；但在循环引用中，计数可能永远不会达到零，从而导致资源泄漏。

清单 2.18 中的代码，很容易复现循环引用问题。如果遇到这种情况，请考虑使用 std::weak_pointer 来打破循环。弱指针的工作方式类似于共享指针，但不共享资源的所有权。这种类型的指针必须在取消引用之前，转换为 std::shared_ptr 。如果转换成功，则创建一个有效的指针；如果转换失败，则转换返回初始化为 nullptr 的共享指针。

虽然代码演示了双端队列，但标准模板库提供了自己的版本，称为 std::deque。请使用此双端队列，尽可能避免编写自己的双端队列，除非出于学习练手的目的！

**建议**

- 使用智能指针无需管理动态资源，可将注意力集中在真正的问题上。
- 当动态资源需要多个引用时，请使用 std::shared_ptr；可使用 std::weak_ptr 打破循环引用。
- 请确保在取消引用之前转换 std::weak_ptr。
- 大多数情况下仍倾向于使用 std::unique_ptr。

# 第 3 章 更好的现代 C++：通用编程

本章内容

- 统一初始化
- 增强的 `if` 语句
- 元组
- 结构化绑定

C++ 编程的动态环境中，编码实践通常由一些习惯所塑造，从而影响我们解决问题的方法。当探索传统循环、选择语句和变量初始化技术的复杂性时，遵守传统有时只能保证可读性或有效性。这些熟悉的方法最初看起来很直观，但往往掩盖了正确性、可读性和整体代码效率挑战。

C++ 建立在 C 的基本原则之上，因此引入了新的初始化技术和数据聚合方法，这种演变带来了一系列复杂性。初始化数组的新方法，可能会导致认知过载和代码库中源文件之间不一致。在简单性至关重要的情况下，依赖结构体或类来聚合各种数据类型可能会很麻烦。虽然标准模板库模板提供了另一种解决方案，但这可能会削弱代码的可读性和有效性。

本章讨论了几个错误，以展示 C++ 编程中的常见陷阱并提供更好的编码解决方案。通过解决这些错误，可以改善我们的编码习惯，并加深对 C++ 语言和 STL 的理解。目标是让开发人员，通过改进实践并采用更高效、更易于维护的编码技术来改进代码。

## 3.1. 错误 10： 初始化时未使用 if

这种错误可能会对效率产生积极影响，但在某些情况下可能会对可读性产生负面影响。`for` 循环可以使用范围进行初始化，但 `if` 语句还不行。

**问题**

需要搜索一个容器，以查找相应的键值。这会创建一个索引循环并用于索引每个值，直到找到键或到达容器末尾，且最好使用标准模板库中提供的算法。以下代码显示了一种典型的方法。预计需要编写或使用一个函数来搜索并返回一个布尔值，指示是否找到；这可以在条件中完成。`if` 语句使用返回的值在两个选项之间进行选择。这种情况下，不会返回布尔值，但会返回迭代器。需要测试迭代器，以确定它是否有效。

列表 3.1 基于索引的关键字搜索

```
int main() {
  std::vector<int> values;
  for (int i = 0; i < 10; ++i)
    values.push_back(i*2);
  int key = 4;
  std::vector<int>::iterator it =
  std::find(values.begin(), values.end(), key); // 1
  if (it != values.end()) // 2
    std::cout << key << " was found\n";
  else
```

```
    std::cout << key << " was not found\n";
  return 0;
}
```

注释 1：确定是否找到了键

注释 2：测试迭代器以确定是否找到了相应键

### 分析

设置搜索、搜索本身，以及处理搜索返回结果，所需的代码冗长且包含大量关键字和结构。虽然这些都是必需的，但还可以更简单。感觉就像用 C 语言在编写代码（除了 vector）。

### 解决

现代 C++ 增强了 if 语句以允许初始化，可将变量限制在与 for 语句非常相似的语句中，此功能称为“带初始化器的 if 语句”。它可以减少代码的行数，并且不会引入可能在函数中意外重用的变量。此外，逻辑包含在条件中，读者能更容易找到它。

以下代码演示了如何在初始化程序中查找键值。此代码进一步演示了，如何使用标准模板库算法进行搜索。迭代器变量在分号之前声明和初始化 - 这是初始化程序部分。分号后面的条件代码检测迭代器是否引用结尾（未找到键）；如果没有，则找到键值。条件值决定了语句其余部分所采用的执行路径。

使用带有初始化器的 if 语句，比传统的 if 语句更紧凑、更具表现力。在使用某些函数式语言时，也是一种温和的过渡。

列表 *3.2* 使用带有初始化器的 *if* 语句来搜索关键字值

```
int main() {
  std::vector<int> values;
  for (int i = 0; i < 10; ++i)
    values.push_back(i*2);
  int key = 4;
  if (auto it = std::find(values.begin(),
        values.end(),key); it != values.end()) // 1
    std::cout << key << " was found\n";
  else
    std::cout << key << " was not found\n";
  return 0;
}
```

注释 1：初始化迭代器并判断是否找到键

### 建议

- 使用带有初始化程序的 if 语句，减少设置决策所需的代码行数。
- 变量应仅在其使用范围内，而不能超出该范围；带有初始化程序的 if 语句的好处是，限制变量的范围。

## 3.2. 错误 11： 未对变量使用类型推断

这个错误会影响效率。可读性通常也会得到增强，但当开发人员希望了解确切数据类型时，这些信息不透明。

### 问题

以下清单中的代码展示了，索引循环和测试搜索结果的典型方法。这些传统用法已刻在我们的大脑中，但传统的并不总是正确的。

清单 *3.3* 典型的基于索引的循环和测试变量

```
1  int main() {
2    std::vector<int> values;
3    for (int i = 0; i < 10; ++i) // 1
4      values.push_back(i*2);
5    int key = 4;
6    std::vector<int>::iterator it =
7    std::find(values.begin(), values.end(), key) // 2
8    if (it != values.end())
9      std::cout << key << " was found\n";
10   else
11     std::cout << key << " was not found\n";
12   return 0;
13 }
```

注释 1：带有循环控制变量的循环

注释 2：数据类型正确，但直观否?

### 分析

许多解决日常问题的传统方法（例如：循环和变量类型）反映了语言的原始状态。过去，编译器不太擅长与开发人员交互，并且依赖于静态方法来声明变量 - 编译器要么批准，要么拒绝开发人员的选择。基于索引的循环非常典型，甚至不会引起注意，但其循环控制变量的方式，在技术上是错误的。

容器的索引始终从 0 开始，增加到容器到达末尾。当然，负值肯定不正确，但 int 变量可以模拟正值、零值和负值。如果吹毛求疵一下，整数不可能是正确的数据类型。用于测试 find 算法的结果，要求我们了解容器、其元素及其迭代器的类型——这三部分需要去确认。更糟的是，这些与正在解决的问题无关；这就是无谓是开销。这种开销会对效率产生负面影响，并影响代码的表达力。

### 解决

现代 C++ 为编译器提供了更多的智能和灵活性。编译器知道数据类型是否正确，可以利用这些来确定类型并将其插入到代码中。清单 3.4 解决了前面提到的两个问题。

首先，编译器确定循环控制变量的有效类型—理想情况下是无符号的。不过，由于其初始化器为零将推断为 int。无符号不仅会给出两倍的范围，而且还会排除负索引值。从数学上讲是正确的类型，实际类型是 size_t，通常实现为 unsigned long 或 unsigned long long。可以表示任何对象的大小，请放心使用。使用 auto ，并让编译器确定其类型会更好。

其次，类型的复杂性可能会分散对所要解决的问题的注意力，并引入其他错误。编译器有确定正确数据类型的能力，可以减轻这种复杂性并避免引入异常。使用 auto 关键字可以利用编译器推断正确数据类型的能力，还可以将这些信息添加到代码中。开发人员专注于所要解决的问题，语言负责处理日常事务，这是一种协同作用——专业的人做专业的事。

清单 *3.4* 使用 *auto* 让编译器推断出正确的类型

```
int main() {
  std::vector<int> values;
  for (auto i = 0; i < 10; ++i) // 1
    values.push_back(i*2);
  int key = 4;
  if (auto it = std::find(values.begin(),
          values.end(), key); it != values.end()) // 2
    std::cout << key << " was found\n";
  else
    std::cout << key << " was not found\n";
  return 0;
}
```

注释 1：编译器会为循环控制变量选择正确的数据类型

注释 2：让编译器找出确切的类型

auto 关键字有一个注意事项：倒计时循环。以下代码展示了以相反顺序显示元素的简单方法。

清单 *3.5* 对 *auto* 的误用

```
int main() {
  std::vector<int> values;
  for (int i = 0; i < 10; ++i)
    values.push_back(i*2);
  for (auto i = values.size() - 1; i >= 0; --i) // 1
    std::cout << values[i] << '\n';
  return 0;
}
```

注释 1：反向实现循环

此代码在系统上运行时会崩溃，出现了段错误。为什么？循环控制变量的推导类型 size_t 是 unsigned。是否有不等于或大于 0 的无符号值？没有。因此，当 i 变为零时，下一个值是数据类型的最大值—超出了容器的边界。这种情况最容易解决的方法是将数据类型改为 int，而不是 auto。这时使用 size_t 是错误的，这是类型推导可能引入的错误之一。由于应用程序二进制接口（ABI）的稳定性和向后兼容性，标准不会对此进行修改。

另一个影响性能的问题出现在使用 auto 推断值类型时：

```
auto name = student->getName();
```

name 变量是学生姓名的副本，但引用会更好。引用代表学生姓名 std::string 的别名，而不是副本，从而节省了对复制构造函数的调用。代码如下所示：

```
auto& name = student->getName();
```

这种区别也出现在使用 auto 在基于范围的 for 循环中推导循环控制变量时。变量始终是元素的副本。虽然这鼓励元素的只读特性，但如果元素类型大于指针，则效率会很低。此代码演示了对 students 列表进行迭代的首选方式：

```
for (auto& student : students) ...
```

**建议**

- 使用 auto 进行变量声明。
- 使用 auto 关键字可使模板受益匪浅。
- 在数据类型大于指针的情况下，请使用 auto 引用，避免复制。
- 开发人员很少需要知道确切的类型 - 如果需要，应将其写出来。

## 3.3. 错误 12： 使用 typedef

这个错误会影响可读性和有效性，类型别名通常用于从现有数据类型定义新数据类型。新类型更能表达类型的用途，并提高可读性。

**建议**

清单 3.6 中的代码使用 typedef 来定义一个可以更好地表达概念的新类型。新类型是一个接受两个整数并返回一个整数的函数，MathFunction 的类型是一个函数指针，因此可以像函数一样分配和调用符合要求的函数。这时的别名嵌入在定义中，既不明显，也不直观。

清单 *3.6* 使用 *typedef* 定义别名

```
typedef int (*MathFunction)(int, int); // 1

int add(int a, int b) {
  return a + b;
}

int sub(int a, int b) {
  return a - b;
}
int main() {
  MathFunction f = add; // 2
  std::cout << "Result of addition: " << f(5, 3) << std::endl;
  f = sub; // 2
  std::cout << "Result of subtraction: " << f(5, 3) << std::endl;
  return 0;
}
```

注释 1：以函数形式定义新类型

注释 2：赋值给新类型的变量

### 分析

别名定义的尴尬之处在于，是从 C 中继承下来的，C++ 仍支持这种语法。如果新类型基于内置类型（例如 int），源类型会位于目标类型之前，这有些反直觉。例如，将别名 age 定义为无符号整数：

```
typedef unsigned int age;
```

### 解决

现代 C++ 引入了 using 关键字，这在很大程度上简化了别名类型。此关键字遵循更直观的方法，其中别名名称位于左侧，其定义位于右侧。赋值操作符使人们认为别名是从定义中分配的，就像常规算术赋值一样，增强了可读性。此外，编写别名更有效，其遵循更直观的语法。

以下代码与清单 3.6 几乎完全相同；唯一的区别在于别名定义。这种相似性表明了两点：首先，使用别名类型与 typedef 版本相同；其次，没有理由继续使用 typedef 别名。

清单 *3.7* 使用 *typedef* 为类型创建别名

```
using MathFunction = int (*)(int, int); // 1
int add(int a, int b) {
  return a + b;q
  }
  int sub(int a, int b) {
  return a - b;
}
int main() {
  MathFunction f = add; // 2
  std::cout << "Result of addition: " << f(5, 3) << std::endl;
  f = sub; // 2
  std::cout << "Result of subtraction: " << f(5, 3) << std::endl;
  return 0;
}
```

注释 1：类型名称与类型定义不同

注释 2：用法相同

如果别名基于内置类型（例如 int），则可读性会增强。例如，可以将别名 age 定义为无符号整数：

```
using age = unsigned int;
```

不要使用全局命名空间

```
using namespace std;
```

很多代码中（在源文件的顶部），包括大多数教科书和许多在线教师都忽略了这条建议，使用这种全局命名空间显然是为了避免冗余的编码，比如：

```
std::cout << ...
```

std(标准）命名空间相当大，包含许多标识符。如果使用全局包含，代码具有同名标识符的可能性很高。猜猜编译器会使用哪一个？我也不知道。冲突和歧义的可能性很大，可能会导致未定义的行为。

**建议**

- 始终使用 `using` 关键字而不是 `typedef`。
- 将 `typedef` 替换为 `using`。
- 值得为代码的可阅读性付出努力。
- 避免在代码中包含全局命名空间。

## 3.4. 错误 13： 通用算法

这个错误关系到有效性和可读性，并提高了正确性。在我们从事的每个项目中编写几个算法可视为标准做法，但这会阻碍了代码重用。

**问题**

当开发人员能够熟练地编写几种标准算法时，其倾向于为每个项目编写算法。其中许多算法都很简单，不需要花费时间——肌肉记忆在这些情况下会有所帮助。以下代码演示了几个需要经常编写的算法。这些算法既不困难也不耗时，也不需要花太多心思。可以工作并满足程序的要求。

清单 *3.8* 两个典型的快速编写算法

```
template <typename T>
T sum(const std::vector<T>& values) { // 1
  T total = (T)0;
  for (auto val : values)
  total += val;
  return total;
}
template <typename T>
bool find(const std::vector<T> values, T key) { // 2
  for (auto val : values)
    if (val == key)
      return true;
  return false;
}
int main() {
  std::vector<int> v { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
  std::cout << sum(v) << '\n';
  if (find(v, 4))
    std::cout << "the key 4 was found\n";
  else
    std::cout << "the key 4 was not found\n";
}
```

注释 1：通用的、编写速度快的求和函数

注释 2：通用的、编写速度快的查找函数

**分析**

大多数开发人员习惯于每次都以某种方式做事，随着时间的推移形成许多固定的习惯。许多习惯有帮助，但有时这些习惯会使效率下降。每个算法大约需要七行代码，才能使其通用且格式良好。但在许多情况下，一些简化提高了开发人员专注于他们问题的能力，而不是解决子问题所需的功能，例如 `sum` 和 `find`。

**解决**

通常，最好的代码，错误最少，且不需要你来写。代码行数越少，出错的机会就越少。当开发人员不负责编写日常工作和开销代码，而是专注于主要问题时，效率会提高。清单 3.9 显示了 `algorithm` 和 `functional` 头文件中的标准模板库代码。了解使用这些需要一些时间，但当理解了之后就很简单了。

此解决方案中还包含一个节省时间的偶然因素，即统一初始化语法，这使得编写简单的 `vector` 非常容易（注意，没有调用 `push_back`）。

清单 *3.9* 两个不需要编写的库函数

```
int main() {
  std::vector<int> v { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
  std::cout << std::accumulate(v.begin(), v.end(), 0) << '\n'; // 1
  auto it = std::find(v.begin(), v.end(), 4); // 2
  if (it != v.end())
    std::cout << "the key 4 was found\n";
  else
    std::cout << "the key 4 was not found\n";
}
```

注释 1：通用的、不需要编写的求和函数

注释 2：通用的查找函数，不需要编写

标准算法已经为开发人员编写并调试完毕。学会依赖它们，如果想要正确性和性能，这些算法足以满足需求。此外，它们的使用非常简单，可以进行更高速的编码。

**建议**

- 对于大多数问题，请使用标准模板库算法。
- 专注于编程问题，而不是次要问题。

## 3.5. 错误 14： 未使用统一初始化

这种错误可以提高正确性、可读性和有效性；某些情况下，也会影响性能。初始化变量的方法有很多种，每种方法都有优点和缺点；但是，这些方法必须一致。

**问题**

C++ 使用从 C 继承的相同技术来初始化变量。但在某些情况下，C++ 超越了 C 的限制，引入了其他形式的初始化。已经引入了几种方法，但没有一种方法能以相同的方式完成其他方法所做

的事情。最终结果是 C++ 代码有多种初始化变量的方法，增加了认知负担。请记住，编程思维应该专注于正在解决的问题，并使用语言来表达解决方案——跟踪语言细节会增加不必要的负担，而不一致会降低效率。

以下代码是初始化整数 vecotr 的简单示例。经典 C++ 没有其他方法可以做到这一点，因此需要代码来设置。真是无语！

清单 *3.10* 容器和简单变量的典型初始化

```
int main() {
  std::vector<int> v;
  for (int i = 0; i < 10; ++i)
    v.push_back(i); // 1
  v.push_back(33); // 2
  v.push_back(42); // 2
  int count = v.size(); // 3
  std::cout << "the vector has " << count << " elements\n";
  return 0;
}
```

注释 1：使用循环来计算一致的值
注释 2：不一致的值需要单独插入
注释 3：通过赋值进行初始化

**分析**

如果只有 push_back，那么每个初始化问题看起来都像 push_back 问题。这种情况表明 C++ 有动力添加其他初始化方法；代码很笨重，需要付出很多才能完成一个简单的任务。

**解决**

使用统一初始化有几个原因；最优秀的是其一致性，在所有情况下都易于理解和适用。特别是对于容器，使用统一初始化的能力都非常出色。清单 3.11 中的代码演示了对 vector 和独立变量使用这种方法。当初始化值放在括号内时，就会成为首选方法。

统一初始化使用括号将初始化值括起来。vector 使用此方法，count 变量也使用此方法。随着开发人员变得更加熟练，这种统一性使编程变得更容易。

清单 *3.11* 容器和简单变量的统一初始化

```
int main() {
  std::vector<int> v {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 33, 42}; // 1
  auto count {v.size()}; // 2
  std::cout << "the vector has " << count << " elements\n";
  return 0;
}
```

注释 1：容器（复合变量）的统一初始化
注释 2：变量的统一初始化（简单变量）

统一初始化使用一种形式来初始化变量，这种一致性使读写代码更容易。

使用此方法还有其他优点：

- 不同变量类型之间的一致性
- 不包括窄化转换 - 例如，double 类型的值不能初始化 int 类型的变量
- 复杂的数据结构更容易初始化
- 构造函数和容器支持初始化列表

**建议**

- 统一初始化可用于初始化变量，甚至是复杂变量。
- 理解并使用这种方法，一致且简单。

## 3.6. 错误 15： 未使用就地初始化

这个错误主要针对性能。向容器中添加大型或复杂元素时，成本会很高。

**问题**

管理对象集合时，应优先使用 vector 之类的容器。通常创建对象并将其复制到容器，如清单 3.12 所示。创建对象需要使用构造函数，之后将其内容复制到容器的元素中；如果使用复制赋值操作符，需要开销也相当于构造函数。临时变量的成本会很高，但这无法避免。

就地创建旨在在特定内存位置创建对象。局部变量分配在运行时栈上，动态创建的对象分配在堆上。开发者不会影响这些变量的位置，只需确保它们存在即可。另外，就地创建可让开发人员指定创建对象的位置。

以下代码演示了创建对象，并将其添加到 vector 的典型方式。通常，创建对象时会调用一次构造函数调用。将对象添加到 vector 时，vector 会将其复制到其元素中，从而再调用一次构造函数。

清单 *3.12* 创建和添加元素到容器

```
class Resource {
private:
  std::string name;
  int instance;
  static int handle;
public:
  Resource(const std::string& n) : name(n), instance(++handle) {}
  int id() const { return instance; }
};
int Resource::handle = 0;

int main() {
  std::vector<Resource> resources;
  resources.push_back(Resource("resource 1")); // 1
  resources.push_back(Resource("resource 2"));
  for (int i = 0; i < resources.size(); ++i)
    std::cout << resources[i].id() << '\n';
  return 0;
```

```
}
```

注释 1：创建一个临时变量，然后将其复制到 vector 的元素中

### 分析

创建 Resource 对象需要调用构造函数。要将对象添加到 vector，容器必须进行复制或赋值，以将源对象的内容传输到目标容器元素。然而，将创建对象的成本增加一倍实在不应该。

### 解决

现代 C++ 改变了规则，以避免两步将对象插入容器的成本过高。如清单 3.12 所示，实例的创建只是为了插入容器；否则，不会使用。这种情况下，可以直接在容器的元素中构造对象，如清单 3.13 所示。就地创建使用单个构造函数在 vector 元素的位置创建对象。这种技术减少了一个构造函数或赋值调用，从而提高了性能。还需注意的是，定义移动构造函数和移动赋值操作符，可将不可复制的对象放入容器中。

清单 *3.13* 直接在容器中创建对象

```
class Resource {
private:
  std::string name;
  int instance;
  static int handle;
public:
  Resource(const std::string& n) : name(n), instance(++handle) {}
  int id() const { return instance; }
};
int Resource::handle = 0;

int main() {
  std::vector<Resource> resources;
  resources.emplace_back("resource 1"); // 1
  resources.emplace_back("resource 2");
  for (int i = 0; i < resources.size(); ++i)
    std::cout << resources[i].id() << '\n';
  return 0;
}
```

注释 1：在 vector 中创建一个对象

并非每个类都可以使用这种技术。如果对象不可复制或不可分配，则编译器在编译代码时会大声抱怨。如果这些对象定义了移动语义，则编译器会报错。容器中的对象可能必须在容器管理期间移动或复制。如果 vector 必须增长，则必须将前一个 vector 的元素复制到新 vector 中，所以就地创建仅适用于某些数据类型。

### 建议

- 当对象不独立于容器使用时，请考虑使用就地创建。
- 指针容器（最好是智能指针容器！）可能比值容器更好。

## 3.7. 错误 16： 未使用元组

这个错误关注的是效率。创建结构体或类来保存多个值，会分散开发人员对解决问题的注意力。

### 问题

一个典型的问题是将多个数据类型聚合到一个单元中，以便于操作。通常使用 `struct` 或 `class`，这需要开发人员创建代码来定义聚合。这种方法已经行之有效多年，但对于普通旧数据（POD）来说，当结构体仅在少数地方使用时，这往往有些矫枉过正。

清单 3.14 中需要对学生的三个属性进行建模——姓名、年龄和平均成绩。代码中还必须定义几个类似的类来承载其他聚合组，增加结构数量并使用不同的数据类型。这种方法的一个明显优势是，可以通过字段名称访问 POD——稍后会详细介绍。

清单 *3.14* 将简单 *POD* 定义为结构体

```
struct Student { // 1
  std::string name;
  int age;
  double gpa;
  Student(const std::string& n, int a, double g) : name(n), age(a), gpa(g) {}
};

// other well-defined PODs
struct Employee {};
struct Teacher {};
struct Admin{};

int main() {
  Student s("Susan", 23, 3.85); // 2
  std::cout << "student " << s.name << ", " << s.age << " years old, carries a "
            << s.gpa << '\n'; // 3
  return 0;
}
```

注释 1: 定义简单的 POD

注释 2: 初始化 POD

注释 3: 使用 POD

### 分析

每个聚合都需要定义一种新的数据类型，必须定义实例变量及其数据类型。最简单的方法是使用构造函数来初始化实例。通过实例变量名访问变量，这是一种自然而有效的方式。但是，类似代码的重复，以及编写每个新 POD 的负担，使这种方法繁琐且容易出错。此外，对数据类型的理解会因代码量、定义之间的距离，以及类似部分的重复而减弱。

### 解决

如清单 3.15 所示，添加了 `using` 关键字以简化数据类型名称的创建及其定义。

此代码中的关键是元组，`tuple`。`std::tuple` 模板通过枚举其类型来声明多个不同类型的实例变量，它们没有名称。结构体中的这种差异最大限度地减少了命名和编码。实例的初始化与结

构版本完全一样，但无需提供构造函数—另一种节省时间和减少混乱的手段。

最后一个区别是负面的；访问字段不是通过名称，而是通过索引。每个实例变量都按规范顺序进行索引，从索引零开始。给定一个索引，`std::get` 函数模板有助于确定要访问哪个字段。

清单 *3.15* 将简单 *POD* 定义为元组

```
using Student = std::tuple<std::string, int, double>; // 1

// other well-defined PODs
// using Employee = std::tuple<...>;
// using Teacher = std::tuple<...>;
// using Admin = std::tuple<...>;

int main() {
  Student s("Susan", 23, 3.85); // 2
  std::cout << "student " << std::get<0>(s) << ", " << std::get<1>(s)
      << " years old, carries a " << std::get<2>(s) << '\n'; // 3
  return 0;
}
```

注释 1：低开销的 POD 定义方法

注释 2：初始化 POD

注释 3：使用 POD

不需要类的完整语义时，`std::tuple` 模板提供了一种紧凑且可用的方法，来定义数据聚合；这种方法特别适用于 POD。模板可用于进一步概括元组。例如，可以使用此替代方法来定义和使用 Student：

```
template <typename T1, typename T2, typename T3>
using Student = std::tuple<T1, T2, T3>;
```

初始化将是

```
Student<std::string, int, double> s("Susan", 23, 3.85);
```

对字段的访问保持不变。

这里有一个警告：如果 Student 和 Employee 定义相同，则它们是同一类型。任何使用这种 Student 的函数或容器也可以使用 Employee —这不是里氏替换原则，而是一个严重的设计缺陷。请仔细考虑如何使用。

关于以不同方式使用元组的最后一点意见是，当函数需要返回多个值时，请考虑使用 `std::tuple` 模板。此应用程序将是考虑使用 `using` 关键字的模板的一个很好的理由。输出参数应替换为元组以增强可读性。

**建议**

- 对于简单结构（通常是 POD），使用 `std::tuple` 来简化编码并方便阅读。
- 考虑模板化元组，以实现更通用的用途。
- 使用元组返回多个值。

## 3.8. 错误 17： 未使用结构化绑定

这个错误主要针对可读性和有效性。元组是一种方便的模板，可以传递或返回多个值。

### 问题

错误 16 中的示例展示了清单 3.16 中的代码，消除了编写结构体或类的麻烦，并依赖于 std::tuple 模板。这种方法的简单性是其最显著的吸引力，但访问值时必须使用函数模板 std::get。此代码编写和阅读起来很不方便，这引起了人们对其实用性的担忧。

清单 3.16 使用函数模板提取元组值

```
using Student = std::tuple<std::string, int, double>;

int main() {
  Student s("Susan", 23, 3.85);
  std::cout << "student " << std::get<0>(s) << ", " << std::get<1>(s) <<
        " years old, carries a " << std::get<2>(s) << '\n'; // 1
  return 0;
}
```

注释 1: 使用带有索引的 std::get 模板函数

### 分析

Student 元组很容易初始化，看起来像是对构造函数的调用。我们认为这种简单性将继续存在，但对字段的访问却证明并非如此。元组的字段按定义顺序排列。由于它们没有名称，因此访问的唯一选项是 get 函数使用索引值。这种方法效果很好，但字段的“名称”与索引之间的脱节的确令人不安。

### 解决

std::tuple 模板的灵活性因其访问字段的方法过多，且不连贯而减弱。编码正确的索引值时很容易出错，或者稍后重新排列字段，并忘记已经编码的访问函数。

结构化绑定将元组，或其他具有公共数据成员的对象，分解为单个值。有些语言将此称为“解构”。其语法很简单：以 auto 关键字开头，添加括号，并为每个数据成员添加一个变量。然后，将元组实例分配给它。以下代码显示了使用结构化绑定的简易性。

清单 3.17 使用结构化绑定提取元组值

```
using Student = std::tuple<std::string, int, double>;
int main() {
  Student s("Susan", 23, 3.85);
  auto [name, age, gpa] = s; // 1
  std::cout << "student " << name << ", " << age <<
      " years old, carries a " << gpa << '\n'; // 2
  return 0;
}
```

注释 1: 使用结构化绑定来分解元组

注释 2: 使用变量名代替索引值

如果结构中字段的顺序发生变化，结构化绑定将存在严重的正确性和可读性问题。如果重新排列了清单 3.17 中的变量名称，则它们与字段的顺序将不一致。如果在结构中重新排列了 age 和 gpa，则变量名称将具有错误的值。如果重新排列使得类型可转换，则清单 3.17 中的代码可以工作（没有编译器或运行时错误），但值将不正确。因此，请注意确保结构字段和结构化绑定变量保持协调一致。

**建议**

- 使用结构化绑定来分解 POD 和简单结构或类。
- 对象必须具有公共数据成员，结构化绑定才能正常工作。

# 第 4 章 更好的现代 C++：附加主题

本章内容

- 文本格式的改进
- 正则表达式
- Lambda 表达式
- 可变参数模板
- 可移植文件系统代码

现代 C++ 扩展了可供开发人员使用的功能范围和深度。许多领域都得到了改进，精心设计和清晰的建议已纳入标准。本章只对一些功能的简单介绍，这些功能使日常编程更简单、更正确、更易读。

所有 C++ 开发人员都会从这些附加主题中受益，但有一些关键点被忽略了，因为有太多漂亮的功能无法一一列举。但是，请将这项工作视为一种通用方法的起点，以获得一些方便的改进。

文本处理已得到改进，使用尚未实现的格式头文件。通过编写代码来解析文本非常复杂且容易出错，但正则表达式功能可以简化开发人员的工作。

出于对函数式编程概念的关注，标准委员会采用了范围，并更好地与现场定义的函数（称为 Lambda）进行交互。Lambda 可以聊的东西太多了，这里只要稍微了解一下如何使用即可。范围实现了 Linux 的“过滤”概念（通常与管道符号一起使用：|），其中一个的输出成为下一个的输入，这些输出可以组合成非常强大的管道。

文件系统代码传统上是为特定平台开发的。对于跨平台代码，使用某种条件编译机制来选择正确的平台相关代码。现代 C++ 对这些区别进行了抽象，允许单个代码单元在任何平台上工作。最后，通过定义数学常数（例如 pi[5]）的新功能，使用千位分隔符可以更简单地理解多位文字数字，以及用户定义的文字值（提高可读性并有助于单位转换），可读性得到了极大增强。

## 4.1. 错误 18： 未使用可变参数模板

这个错误影响了效率和可读性。对于许多问题来说，变长参数列表是很自然的，但处理它们会增加开销和复杂性。

**问题**

可变参数列表非常常见，C 语言很早就提供了一种处理它们的方法。C++ 继承了这种方法，但这种方法存在问题。

C++ 提供了一种更简洁的方法来处理这种情况，但并非没有复杂性和开销。清单 4.1 将参数捆绑到容器中，并将容器传递给 sum 函数（可以使用 std::accumulate 算法更好地解决此问题）。没有特殊的函数调用（例如，va_list），处理容器很简单。此解决方案模板化，以处理加法操作符有意义的类型，但这种好处需要付出代价：开发人员必须创建容器，将参数打包到其中，并将其作为参数传递给函数。所有这些努力都是解决问题之外的工作；打包数据只是必要的日常工作。阅读此代码需要分散注意力，专注于解决问题和开销。

[5]译者注：pi 表示数学中的 π(圆周率)

清单 *4.1* 使用 *vector* 实现可变形参列表

```
template <typename T>
T sum(T initial, const std::vector<T>& vals) { // 1
  T sum = initial;
  for (int i = 0; i < vals.size(); ++i)
    sum += vals[i];
  return sum;
}

int main() {
  std::vector<int> intvalues; // 2
  for (int i = 1; i < 10; ++i) // 3
    intvalues.push_back(i);
  std::cout << sum(0, intvalues) << '\n'; // 4
  std::vector<double> doublevalues;
  for (int i = 1; i < 4; ++i)
    doublevalues.push_back(i);
  std::cout << sum(5.0, doublevalues) << '\n';
  return 0;
}
```

注释 1：选择模板化，实现具有更高的灵活性

注释 2：创建容器

注释 3：填充容器

注释 4：解决问题

### 分析

清单 4.1 中所示的解决方案是上一节的，它给开发人员的效率留下了一个重大问题，但也为改进提供了机会。开发者负责构建一个包含要处理的值列表的结构。此结构的构建纯粹是开销，不是要解决的问题的一部分，而是将值聚合为可用形式的必要步骤。代码越不直接关注问题，开发人员的效率就越低，出错的机会就越大。

### 解决

现代 C++ 提供了一种递归方法解决此问题，为通用模板函数提供长度可变的参数列表。模板将可变参数列表解析为第一个参数和其余参数形式的组合，将组合复制到上述对中的其余参数进行递归。此过程将持续到最后一个参数独立，而没有剩余参数为止。编写了第二个接受一个参数的模板函数；这是通用模板的特化。模板的显着优势是在编译时可以进行计算，从而节省了运行时成本。

清单 4.2 展示了这两个模板函数。特化（或基础）模板很简单；它接受一个参数，并返回它。递归（或通用）模板也很简单。每个参数对都是最左边的参数和其余参数的可变参数包。当前模板（概念上）会推送到堆栈上，其剩余参数成为对模板的下一次递归调用的参数列表。此模板调用将参数列表分为最左边的参数以及剩余参数，当剩余参数恰好包含一个参数时，将调用专用模板。然后（概念上）使用单个参数弹出堆栈模板调用，并从每个参数构建函数调用。

要实现这种模板化的可变参数模式，首先处理基本情况，然后处理递归情况。就是这样！当编译器解析函数调用时，会查找可以处理多个参数的 sum 函数，将发现具有单个参数和可变参数包

的模板。如果编译器可以生成参数对（最左边和剩余），就知道继续调用哪个模板版本。当只剩下一个参数时，可变参数模板不适用。编译器将选择具有单个参数的 sum 模板。如果没有该情况的函数，这种技术将无法工作。但它确实有效，而且效果非常出色。

清单 *4.2* 使用模板化的可变参数列表

```
template <typename T> // base case
T sum(T t) { return t; }

template <typename T, typename... Pack> // general (recursive) case
T sum(T t, Pack... remaining) { return t + sum(remaining...); }

int main() {
  std::cout << sum(3, 4) << '\n'; // 1
  std::cout << sum(3, 4, 5) << '\n'; // 1
  std::cout << sum(3, 4, 6, 7) << '\n'; // 1
  return 0;
}
```

注释 1：以自然的形式解决了问题

使用折叠表达式（C++17 及更高版本）来实现扩展目标。清单 4.2 中的模板可以压缩为

```
template <typename T, typename... Pack>
T sum(T t, Pack... remaining) {
  return (t + ... + remaining);
}
```

**建议**

- 使用带有可变参数包的模板，来处理可变长度的参数列表。
- 避免使用经典 C/C++ 方法处理可变参数列表。

## 4.2. 错误 19： 使用全局命名空间枚举

这种错误会影响有效性和可读性，并且会对正确性产生重大的影响。枚举允许开发人员命名以增强可读性，但必须正确且清晰。

**问题**

变量和常量可用于命名概念，例如：可以为整数常量分配一些映射到红色或蓝色的值。除非开发人员小心谨慎，否则这种映射可能会非常随意，阅读时可能会很难懂。C++ 语言提供了一种简化方法，即使用 enum 常量。枚举中的命名对象可以表示集合中的可能值。

枚举常量以整数形式实现，因此很容易与整数相互转换。清单 4.3 显示了一个表示颜色的整数变量，该变量为 2，这是什么意思呢?

开发人员正在解决一个与汽车及其相互作用有关的问题。汽车和交通信号灯都有颜色。枚举颜色以便使用符号名称是有意义的。编译器默认为这些常量分配值，并且会自动考虑更改（即添加、

删除或重新排列）。然而，开发人员遇到了一些问题。以下代码显示了对汽车和交通信号灯的颜色属性进行建模（简陋）的尝试。

清单 4.3 有问题的全局命名空间枚举

```
enum TrafficColor {
  Green,
  Yellow, // 1
  Red // 2
};

enum PaintColor {
  Gray,
  White, // 2
  Black,
  Paint_Red, // 3
  Blue,
  Paint_Yellow,
  Paint_Green
};

int main() {
  int color = 2; // 4
  switch (color) {
  case Red:
    std::cout << "we have a red one\n";
    break;
  case Blue:
    std::cout << "we have a blue one\n";
    break;
  case Yellow:
    std::cout << "we have a yellow one\n";
    break;
  default:
    std::cout << "we have a white one\n";
    break;
  // case White: // 5
  };
  return 0;
}
```

注释 1：红色交通信号灯进入全局命名空间

注释 2：两个枚举中的索引默认为 1

注释 3：不能重复使用完全相同的枚举名称；必须对其进行修改

注释 4：为枚举分配和使用 int

注释 5：枚举中的重复索引值——无法定义

**分析**

枚举常量位于全局命名空间中。如果声明的符号不包含在特定命名空间中，则该命名空间是每

个符号所在的位置。开发人员在使用 enum 常量时发现了几个问题，如清单 4.3 所示。

首先，所有常量都在一个命名空间中，因此重复值会造成错误。重命名油漆颜色的努力表明开发人员，必须弱化符号名称才能解决这个问题。

其次，由于整数支持常量，任何混合两个 enum 集的代码都存在多个冲突值的风险，会产生歧义或错误。switch 语句不能有两个枚举常量值相同的情况，例如 Yellow 对应 TrafficColor 和 White 对应 PaintColor（每个都分配了相同的值；在本例中为 1）。

第三，可以给已分配颜色的变量分配任意整数值。此示例为颜色分配了 2。开发人员必须努力清楚地传达分配的含义。需要在示例中明确说明，值为 2 的枚举常量指的是哪个。

最后，支持整数变量无法知道某个值是否超出范围。该变量可能赋值为负数或大于最大有效常数值，很难搞定清楚这代表了什么。更糟糕的是，代码可能会继续运行，但会意外或错误地处理不正确的赋值（更多未定义行为）。

**解决**

现代 C++ 来了！青天就有了！枚举现在可以封装在类命名空间中，这让全局命名空间井然有序，允许在不同命名空间中使用相同的常量名称。尽管支持变量类型是整数，但代码无法为该变量分配整数值，分配必须在该类的枚举值集内，不能分配非法值。

以下代码显示了如何使用 enum 类，将所有定义的值隔离到一个类中，而不会与另一个类冲突。一个类（和命名空间）的常量不能分配给另一个变量，从而避免了前面遇到的一些问题。

清单 *4.4* 单独的命名空间枚举

```
enum class Traffic { // 1
  Green,
  Yellow,
  Red
};

enum class Paint { // 2
  Gray,
  Black,
  White,
  Red,
  Blue,
  Yellow,
  Green
};

int main() {
  auto color = Traffic::Yellow; // 3
  switch (color) {
  case Traffic::Red: // 4
    std::cout << "we have a red light\n";
    break;
  case Traffic::Yellow:
    std::cout << "we have a yellow light\n";
```

```
      break;
    case Traffic::Green:
    default:
      std::cout << "we have a green light\n";
      break;
    };

    Paint can = Paint::Blue;
    switch (can) {
    case Paint::Red:
      std::cout << "we have a red cube\n";
      break;
    case Paint::Yellow:
      std::cout << "we have a blue cube\n";
      break;
    case Paint::White:
      [[fallthrough]];
    default: // 5
      std::cout << "we have a white cube\n";
      break;
    };
    return 0;
  }
```

注释 1：命名空间绑定枚举

注释 2：不同命名空间绑定枚举

注释 3：使用 auto 作为数据类型，该数据类型由初始化值推导而来

注释 4：与同名其他类中的枚举字面量不同

注释 5：消除故意“贯穿”行为的警告

[[fallthrough]] 注释记录了不使用 break 关键字的有意失败行为的情况。switch 语句很容易出错，因此请使用可用的技术来避免错误。不要忘记最后一个右括号（}）后面的语句结尾分号（;）。

枚举类将值隔离到一个命名空间中，并防止将任意值分配给类变量。使用这一功能，enum 的经典问题即得到解决并消除了歧义。如果需要指定 enum 类的底层表示类型，可以通过基本类型扩展来完成：

```
enum class Traffic : uint8_t {
  ...
};
```

**建议**

- 将全局 enum 定义更改为 enum 类。

## 4.3. 错误 20： 未使用新的格式化功能

这种错误会影响效率和可读性，在某些情况下还会影响正确性。格式化输出是一种艺术形式，与 C++ 笨拙的方法相比，C 风格的格式化具有一些明显的优势。

### 问题

C 为细粒度的输出文本格式化提供了一种有用但危险的解决方案。printf 系列函数提供了一个格式说明符，其中包含可读文本和用于格式化变量的嵌入说明。说明符使用特殊字符来表示变量的插入位置，后跟变量或表达式的列表。但当格式说明符变量占位符与变量类型不匹配时，就会出现问题。编译器不会检查以确保类型与说明符匹配，也不会检查变量的数量是否等于说明符的数量。

假设正在为一所学校建模，此示例中一个基本数据类型是 Student。有了学生，我们就想知道他们的姓名、年龄和 GPA[6]。要输出出这些，格式说明符应包含三个占位符，每个属性一个。代码必须确保占位符针对属性的数据类型正确定义，并且参数的顺序正确。以下代码演示了一个干净、没有奇怪的示例。

清单 *4.5* 使用 *sprintf* 实现细粒度的格式化功能

```
struct Student {
  std::string name;
  int age;
  double gpa;
  Student(const std::string& n, int a, double g) : name(n), age(a), gpa(g) {}
};

int main() {
  Student dinah("Dinah", 26, 3.3);
  char buffer[256];
  sprintf(buffer, "%s is a student aged %d carrying a gpa of %.2f",
    dinah.name.c_str(), dinah.age, dinah.gpa); // 1
  std::cout << buffer << '\n';
  return 0;
}
```

注释 1: 格式易于理解（但 sprintf 存有危险）

### 分析

虽然此代码写得正确，但在维护期间交换一些格式说明符或变量，会导致一些奇怪的行为（未定义行为）。任何变体版本都能通过编译，这是一个危险信号。麻烦的问题是%s 说明符需要 char*，而不是 std::string。

经典 C++ 为提供了 std::stringstream 模板，该模板与 operator<< 配合使用，可以干净地完成大多数输出（主要是将数值转换为字符串值）。但此解决方案也并非所有情况下都好

[6]GPA 通常代表 “Grade Point Average”，即平均成绩点数，是一种用来衡量学生学术表现的标准化计算方法，广泛应用于教育机构中，特别是在高等教育里用于评估学生的整体学业成就。GPA 通过将每门课程的成绩转换为一个数值（通常是 4.0 或 5.0 为满分的尺度），然后根据各课程的学分加权平均计算得出。这种方法使得不同学科和课程之间的成绩具有可比性。在不同的国家和地区，以及不同的教育系统中，GPA 的具体计算方式可能有所不同。

使，当格式变得更加复杂，字符串流就会显得过于笨拙。

以下代码提供了解决此问题的方法，但其可读性非常糟糕。只是不会因错误指定变量，而导致未定义的行为；无论使用哪种数据类型，底层字符流都会正确格式化。

来添加一个新要求：学生的 GPA 必须以这样的方式格式化：需要有一个前导数字、一个小数点和两个尾随数字 - 没有人会吹嘘说“我的 GPA 是 3.8”，而实际上他们的 GPA 是 3.85。此外，GPA 需要有两位尾随数字以保持一致性和列对齐。学校管理部门坚持这一要求；他们不想手动对齐 4 和 3.85 - 他们想要 4.00 和 3.85。

因此，必须在 GPA 4 上添加两个尾随零，并且还需要一个尾随零以确保 GPA 3.5 输出为 3.50。如下面的代码所示，指定该要求比在代码中实现更直接。让 gpa 变量正确地用尾随零填充是一项繁琐的工作，解决方案需要一些研究，也不像看起来那么明显。

清单 *4.6* 使用 *stringstream* 实现细粒度的格式化功能

```
struct Student {
  std::string name;
  int age;
  double gpa;
  Student(const std::string& n, int a, double g) : name(n), age(a), gpa(g) {}
};

int main() {
  Student dinah("Dinah", 26, 3.3);
  std::stringstream str;
  str << dinah.gpa;
  std::string gpa_str(str.str());
  str.str("");
  str << dinah.name << " is a student aged " << dinah.age << " carrying a gpa of "
      << gpa_str << std::setw(4 - gpa_str.size()) << std::setfill('0') << ""; // 1
  std::cout << str.str() << '\n';
  return 0;
}
```

注释 1：格式复杂，收效甚微

**解决**

现代 C++ 提供了一个巧妙的解决方案，将 printf 系列函数的所有易于指定性与字符串流的类型安全性相结合。清单 4.7 演示了与 printf 系列函数相同的形式，即使用 {} 作为占位符，并在格式说明符后使用变量或表达式列表[7]。使用此法的好处是输出表现良好，变量的位置也很突出。更好的是，编译器会找出数据类型并自动转换为字符串，并且具有类型安全性。微调变量输出的能力让人想起 C 风格的方法，但使用方式更统一。不可否认，需要进行一些实验才能掌握这些符号及其交互，但这并不困难。结果是输出格式简单易读，不存在类型不匹配问题。现在以简单的方式满足了两位尾随数字的要求。学校对前导数字、小数点和两位尾随数字的要求由列宽（4）和填充字符（0）指定。因此，GPA 为 4 将输出为 4.00；GPA 为 3.5 将输出为 3.50；GPA 为 3.85 将输出为 3.85，列都一致对齐并根据需要进行填充。

[7]译者注：这很 Python。

注意：format 头文件的 std::format 功能部分在有限的编译器上可用。C++20 标准定义了该功能，但一些编译器仍需要实现它。我下载 Ubuntu 23.10 并安装其 GCC 编译器套件（版本 13+）才能获得此功能，其在 Clang 和 MSVC 中已经实现。

清单 *4.7* 使用 *std::format* 实现细粒度格式化功能

```
struct Student {
  std::string name;
  int age;
  double gpa;
  Student(const std::string& n, int a, double g) : name(n), age(a), gpa(g) {}
};

int main() {
  Student dinah("Dinah", 26, 3.3);
  std::cout << std::format("{} is a student aged {} carrying a gpa of {:0^4}", dinah.name,
  ↪  dinah.age, dinah.gpa) << '\n'; // 1
  return 0;
}
```

注释 1：具有类型安全行为的 sprintf 功能

**建议**

- 使用 std::format 可获得 printf 样式函数的优势。
- 字符串流比 printf 样式函数更安全，但细粒度格式化会很尴尬。
- 某些编译器尚未实现 std::format 功能；请提前确定编译器是否实现了。
- C++23 提供了 std::print 和 std::println 来简化编码输出语句；与使用类似形式的语言（例如 Java 和 Python）一致。

## 4.4. 错误 21： 未使用容器的范围功能

这个错误会严重影响效率。开发人员编写映射、过滤或减少数据的方法通常非常乏味，而且大部分代码都集中在琐事上。

**问题**

数据容器代表一种高效而紧凑的处理值序列的方法。vector 是一种很好的选择，易于使用、灵活且功能强大。然而，使用它们所需的代码本质上需要完成一些琐碎的任务，这会分散开发人员对所要解决的问题的注意力。基于范围的 for 循环是朝着更函数式编程迈出的重要一步，并使用了数据流。

开发人员会获取一系列值，将所有负数转换为其绝对值，消除奇数值，并对剩余值求和。清单 4.8 是完成这项工作的直接方法，代表了处理序列的常用方法。前两个循环可以压缩为一个，但操作会更加晦涩难懂。一次执行一个功能方面更直接，但性能很差。与往常一样，我更关注可读性，而非性能。

清单 *4.8* 值序列上的几个独立功能

```cpp
int main() {
  std::vector<int> vals { 42, 3, 7 -9, 0, 22, 23, -7, 22 };

  for (auto v : vals) // 1
    if (v < 0)
      v = -v;

  std::vector<int> evens; // 2
  for (const auto v : vals) // 3
    if (v % 2 == 0)
      evens.push_back(v);

  int sum = 0;
  for (const auto v : evens) // 4
    sum += v;

  std::cout << sum << '\n';
  return 0;
}
```

注释 1：将负值转换为对应的正值
注释 2：简单起见，新建一个容器
注释 3：过滤掉奇数
注释 4：对第二个容器中的所有值求和

**分析**

代码执行了开发人员需要实现的功能。发生了什么、如何工作，以及结果意味着什么，这些都很简单明。循环意味着应该使用更好的方法，但选择很少。可以压缩前两个循环，但会对可读性产生负面影响。第二个容器也感觉不是很舒服。正确地修改 vector 很有挑战性；使用第二个 vector 来保持干净和正确，也会影响性能。

**解决**

数据序列的优点是可以作为流进行处理，cin 和 cout 对象使用此方法。文件输入和输出遵循数据流模型。此模型引人注目，应尽可能使用其方法，并且操作其所需的代码也进行了最小化。C++ 引入了范围头文件，其中包含范围和视图来操作值流。尤其是在使用 Lambda 时，可将代码量减少到最低。

范围的明显优势是函数可组合，可以按顺序逐个调用函数。第一个过滤函数的输出可送到下一个过滤函数，作为其输入；其输出成为下一个过滤函数的输入，依此类推。通常，这种过滤器组合称为“流水线”。

下面的代码介绍了处理数据流的三种基本函数形式：

- 映射
- 过滤
- 归约

映射是一种方法，可获取输入流中的每个元素，对元素执行相应的操作，并生成另一

个元素，该元素将放入输出流中。此功能是一对一生成—每个输入值恰产生一个输出值。std::views::transform 范围适配器附带一个函数，该函数将输入值映射或转换为输出值。以下代码调用 abs 函数将任何负值转换为正值。正值不修改，表明转换可能不会影响值，但会将其传递。每个值都会映射到其绝对值。

过滤是一个函数，接受输入流中的每个元素，并确定是否应将该元素传递到输出流。它采用一个确定包含或拒绝的谓词，是一个返回布尔值的函数。如果返回值为 true，则保留该元素；如果返回值为 false，则拒绝（消除）该元素。

理解过滤的第二种用法很重要。Linux 示例的传统名称；在范围上下文中，其表示一种功能。两种用法相关，但不等同。清单 4.9 中的代码演示了两个函数的组合，即绝对值和偶数谓词。

清单 4.9 过滤掉奇数值，只保留偶数值。此功能是一对一或一对无的产生式；要么保留值，要么什么都不保留。std::views::filter 范围适配器使用谓词函数，来确定元素包含或排除。根据谓词的返回值，每个值都可过滤掉或保留。

最后，归约是一个功能，获取输入流中的每个元素并对其执行某些操作，从而产生单个结果值。该函数使用输入流中的每个元素来得出最终值，典型的例子是求和。清单 4.9 中的代码将过滤后的值相加以得出其总和。此功能是多对一的产生——每个输入元素都会对最终的单个值做出贡献。std::accumulate 函数模板采用迭代器作为第一个和最后一个元素以及一个初始值（求和时为零），所有输入值都缩减为它们的和。

C++ 范围和范围适配器功能强大，但尚未达到其他语言的性能水平。accumulate 函数无法与映射和过滤功能的结果组合，所以开发人员必须提供开始和结束迭代器。通过创建第二个容器来保存流水线流值的结果，未来可能会开发累积式范围适配器。几种函数式语言提供了许多可组合函数，使用起来很融洽。

清单 4.9 值序列上的组合函数

```
int main() {
  std::vector<int> vals { 42, 3, 7 -9, 0, 22, 23, -7, 22 };

  auto evens = vals
    | std::views::transform([](int x) { return abs(x); })
    | std::views::filter([](int x){ return x % 2 == 0; }); // 1
  auto sum = std::accumulate(evens.begin(), evens.end(), 0); // 2

  std::cout << sum << '\n';
  return 0;
}
```

注释 1：创建一个流，将值绝对化，过滤掉奇数，并存储在一个新容器中

注释 2：对第二个容器中的值求和

**建议**

- 范围可以显著减少传统容器处理代码的开销。
- 尽可能使用范围和范围适配器来处理容器，以简化编写和读取代码。
- 并非所有面向范围的函数都可以在管道中组合。
- 函数式编程功能强大，可以使用管道法过渡到该范式。

## 4.5. 错误 22： 编写非可移植的文件系统代码

此错误会影响可读性和有效性。文件系统在不同平台上有所不同，开发与其交互的代码可能很复杂、晦涩难懂，且不可移植。

**问题**

许多程序必须访问文件系统才能输入和输出数据，以进行配置、处理、记录结果或其他目的。多年来，实现此目的所需的代码在复杂性和功能性方面不断增长。这一进程的一个重大问题是，不同的平台已经朝着不同的、不兼容的方向发展。

编写一个需要检查文件是否存在并在不存在时创建文件的程序很简单，但要考虑到平台文件系统的复杂性。开发人员需要编写一个简单的程序来检查文件是否存在；如有必要，创建文件；并输出一行文本。该程序必须可在 Windows 和 Linux 上运行。虽然为任一系统编写功能都很简单，但为两者编写功能却很复杂。有两种方法：预处理器指令和独立的源文件。第一种方法没有演示，但多个代码库使用了这种技术。清单 4.10 和 4.11 中的代码演示了使用单独文件的第二种方法。Windows 文件处理依赖于几个特定于平台的函数和常量。

清单 *4.10* 创建和填充文件：*Windows*

```
#include <windows.h> // 1

void createFileIfNotExists(const std::string& filename) {
  std::wstring wideFilename = std::wstring(filename.begin(), filename.end());
  if (GetFileAttributesW(wideFilename.c_str()) == INVALID_FILE_ATTRIBUTES) {
    HANDLE hFile = CreateFileW(wideFilename.c_str(), GENERIC_WRITE, 0,
    NULL, CREATE_NEW,
    FILE_ATTRIBUTE_NORMAL, NULL); // 2
    if (hFile != INVALID_HANDLE_VALUE) { // 3
      std::wcout << L"File created: " << wideFilename << std::endl;
      const wchar_t* content = L"Hello, world!\r\n";
      DWORD bytesWritten;
      WriteFile(hFile, content, wcslen(content) * sizeof(wchar_t),
      &bytesWritten, NULL);
      CloseHandle(hFile);
    } else
      std::cerr << "Error creating file: " << filename << std::endl; // 4
    } else
      std::wcout << L"File already exists: " << wideFilename << std::endl;
}

int main() {
  createFileIfNotExists("example_file.txt");
  return 0;
}
```

注释 1: 特定于 Windows 的包含文件
注释 2: Windows 特有的常量
注释 3: 针对 Windows

注释 4：使用宽字符流时使用 wcerr

Linux 拥有一些特定于平台的函数和常量，与 Windows 中不同。这种差异使得跨平台处理变得困难且容易出错。专门从事某一平台的开发者，可能无法理解另一平台开发人员的代码。

清单 4.11 创建和填充文件：Linux

```
#include <unistd.h> // 1
#include <fcntl.h> // 1

void createFileIfNotExists(const std::string& filename) {
  if (access(filename.c_str(), F_OK) == -1) {
    int fd = open(filename.c_str(), O_CREAT | O_WRONLY, S_IRUSR
        | S_IWUSR | S_IRGRP |S_IROTH); // 2
    if (fd != -1) { // 3
      std::cout << "File created: " << filename << std::endl;
      const char* content = "Hello, world!\n";
      write(fd, content, strlen(content));
      close(fd);
    } else
      std::cerr << "Error creating file: " << filename << std::endl;
    } else
      std::cout << "File already exists: " << filename << std::endl;
}

int main() {
  createFileIfNotExists("example_file.txt");
  return 0;
}
```

注释 1：Linux 特定的头文件
注释 2：Linux 特定的常量
注释 3：Linux 特定的用法

**分析**

常见的方法是使用预处理器指令，这些指令可以根据平台选择性地包含或排除代码。所有代码都包含在一个源文件中，并按平台分隔，并通过 #ifdef 指令启用，这样做的好处是简化了构建过程。单个源文件易于编写和构建；但使用可选包含来复制代码通常更难阅读。

这里演示的第二种方法是使用单独的文件，并通过一些特定于平台的宏或其他技术确定哪些文件包含在构建过程中。将一个平台的所有代码放在一个源文件中，这样做的简单性使阅读更容易，但会使构建变得复杂。

主要问题是跨平台代码很少用于特定于平台的操作，例如：文件系统操作。同样，开发人员必须花费大量时间来处理，那些不直接影响正在解决的问题，但属于内部细节的错误。

**解决**

现代 C++17 提供了一组标准函数，适用于任何符合要求的平台上的文件系统。通用性允许直接编写以通用方式执行文件系统功能的代码。清单 4.12 中的代码将前面的代码示例，重新编写为一种适用于 Windows 和 Linux 的通用形式。此解决方案无需使用预处理器指令，或多个源

文件进行条件编译，来隔离特定于平台的代码。

清单 *4.12* 创建和填充文件：通用

```
namespace fs = std::filesystem; // 1

void createFileIfNotExists(const
    std::string& filename) { // 2
  fs::path filePath = filename;
  if (fs::exists(filePath)) { // 3
    std::cout << "File already exists: " << filePath << std::endl;
    return;
  }
  std::ofstream file(filePath);
  if (!file) {
    std::cerr << "Error creating file: " << filePath << std::endl;
    return;
  }
  if (!file.is_open()) {
    std::cerr << "File cannot be opened: " << filePath << std::endl;
    return;
  }
  std::cout << "File created: " << filePath << std::endl;
  file << "Hello, world!\n"; // 4
}

int main() {
  createFileIfNotExists("example_file.txt");
  return 0;
}
```

注释 1：为更简单的使用，对命名空间进行别名的便捷方法；代码中的 fs 别名为 std::filesystem

注释 2：适用于 Windows 和 Linux 的通用代码

注释 3：检查文件地址是否存在

注释 4：将相应内容填充到文件中

将文件系统功能概括为标准头文件，使开发人员能够专注于使用文件系统原语解决问题。文件系统的简化抽象以与特定系统无关的方式提供了所有基本功能。该指令指定一个别名；此形式可用于多种上下文（例如：模板）。这里，简化了在 path 变量和 exists 函数之前写入 std::filesystem 的过程；只需使用 fs 别名和范围解析操作符（::）即可指定。清单 4.12 中的代码有一个重要的警告：该示例仅适用于单线程环境。

并发增加了复杂性，代码中没有演示这一点。问题是另一个线程可能在检查时间（TOC）和使用时间（TOU）之间删除、更新或以其他方式影响文件的特性，这个问题也称为 TOCTOU；代表了一种竞争条件，可能会让粗心的开发人员犯错。

**建议**

- 使用新文件系统功能来处理文件和目录。
- 尽可能更新现有代码以使用新功能。

- 访问文件系统后记得检查错误；虽然代码更容易编写，但错误仍会带来问题。

## 4.6. 错误 23： 编写过多的独立函数

这个错误主要针对可读性和有效性，但需要经过一段陡峭的学习曲线才能解决。一次性且不常调用的函数往往会使源码变得混乱，并使查找变得困难。

**问题**

假设，开发人员编写代码来管理美国一所学校。有几名学生在读，每个学生都必须在数据库中。与任何由多人组成的组织一样，必须进行排名或排序。我们的开发人员要求按姓名、年龄和 GPA 对学生进行排名。

为了便于处理，学生名单保存在容器中。必须使用某种方法来迭代容器并确定首选顺序，可以修改容器内的顺序。

开发人员编写了三个独立函数来实现基于给定标准的排名，代码库比我们想的要大得多。这些新函数必须放在某个地方，因此它们在许多其他代码中的文件中定义，并且在其他地方调用这些函数来执行其行为。

清单 *4.13* 几个独立函数

```
class Student {
private:
  std::string name;
  int age;
  double gpa;
public:
  Student(const std::string& n, int a, double g) : name(n), age(a), gpa(g) {}
  std::string getName() const { return name; }
  int getAge() const { return age; }
  double getGpa() const { return gpa; }
};

// assume lots of code

bool by_name(const Student& s1, const Student& s2)
{ return s1.getName() < s2.getName(); } // 1
bool by_age(const Student& s1, const Student& s2)
{ return s1.getAge() < s2.getAge(); } // 1
bool by_gpa(const Student& s1, const Student& s2)
{ return s1.getGpa() < s2.getGpa(); } // 1
void output_by_student_name(const std::vector<Student>& s) {
  for (int i = 0; i < s.size(); ++i)
    std::cout << s[i].getName() << '\n';
  std::cout << '\n';
}
// assume even more code

```

```
int main() {
  std::vector<Student> s;
  s.push_back(Student("Susan", 23, 3.85));
  s.push_back(Student("James", 24, 3.35));
  s.push_back(Student("Annette", 25, 3.75));
  s.push_back(Student("Wilson", 26, 3.8));

  std::sort(s.begin(), s.end(), by_name); // 2
  output_by_student_name(s);
  std::sort(s.begin(), s.end(), by_age); // 2
  output_by_student_name(s);
  std::sort(s.begin(), s.end(), by_gpa); // 2
  output_by_student_name(s);
  return 0;
}
```

注释 1：几个隐藏在代码中的函数，只使用一次

注释 2：调用函数，但没有参数或实现的提示

**分析**

命名函数使得描述其行为的句柄易于记忆，但当混入其他代码时，其实现就很难找到。由于定义和调用站点相距甚远，开发人员必须记住其实现细节，才能充分理解函数的预期行为。

现在，做一个大胆的假设：开发人员最近有机会学习函数式编程。一个想法突然出现在他们的脑海中，认为这些函数可以重新设计成一种更具功能性的方法。一个新的需求出现了，开发人员认为这是尝试创建函子的绝佳机会。

函子是实现 operator() 的结构（或类）。此操作符有多种名称，我们将其称为 apply 操作符（以下函数术语），其是一个执行行为的普通函数—本例中，确定最大元素。为了实现这一点，创建了一个 Maximum 结构，其应用操作符用于查找 vector 中的最大值。将 apply 视为将函数应用于数据，也许感觉很落后，但这就是函数式思维方式。传统的命令式编程更多地考虑，将数据传递给函数并执行其行为；函数式编程考虑，将函数印记或应用于数据——一种以数据为中心的模型。

清单 4.14 中的代码展示了，确定最大值的两种实现。第一种是开发人员在创建函子后想出的。函子的实例化会创建一个函数对象 f，f 对数据的应用是常规实现的；函数对象后面跟着一个参数列表，并返回一个结果（如果有）。开发人员的团队负责人提供了另一种方法，如下所示。团队负责人建议使用 Lambda 来简化代码，消除命名函子及其实例化，并让读者理解调用点的行为。

清单 *4.14* 演示就地函数定义

```
template <typename T>
struct Maximum { // 1
  T operator()(const std::vector<T>& vals) { // 2
    T max = vals[0];
    for (const auto& v : vals)
      if (v > max)
        max = v;
```

```
    return max;
  }
};

int main() {
  std::vector<int> v {3, 9, 6, 2 -1, 0, 8};
  Maximum<int> f; // 3
  std::cout << f(v) << '\n'; // 4
  auto max = std::accumulate(v.begin(), v.end(), v[0],
    [](auto a, auto b){return std::max(a, b); }); // 5
  std::cout << max << '\n';
  return 0;
}
```

注释 1：定义函数对象类（函子）
注释 2：定义应用操作符
注释 3：实例化函数对象类，生成函数对象
注释 4：将函数对象应用于数据；调用应用操作符
注释 5：使用带有局部最大值 Lambda 函数的数值算法

**解决**

团队负责人展示了 Lambda 的一个优势，能够在需要的地方准确定义行为。这种方法消除了回滚和寻找函数定义，以及定义函子的尴尬。清单 4.15 中的代码演示了按姓名、年龄或 GPA 对学生进行排名的原始三个函数。此示例中，行为是在使用时定义的，清楚地传达了正在执行的操作。诚然，最初几次编写和阅读 Lambda 需要更长的时间，但理解了 Lambda 就会变得简单且自动化。

由于 Lambda 是函数对象，因此可以将它们分配给变量，并在调用点传递该变量。GPA 对学生进行排名的地方就证明了这种方法。

清单 *4.15* 原位函数定义 *(Lambda)*

```
class Student {
private:
  std::string name;
  int age;
  double gpa;
public:
  Student(const std::string& n, int a, double g) : name(n), age(a), gpa(g) {}
  std::string getName() const { return name; }
  int getAge() const { return age; }
  double getGpa() const { return gpa; }
};

// assume lots of code

void output_by_student_name(const std::vector<Student>& s) {
  for (int i = 0; i < s.size(); ++i)
```

```
    std::cout << s[i].getName() << '\n';
  std::cout << '\n';
}

int main() {
  std::vector<Student> s;
  s.push_back(Student("Susan", 23, 3.85));
  s.push_back(Student("James", 24, 3.35));
  s.push_back(Student("Annette", 25, 3.75));
  s.push_back(Student("Wilson", 26, 3.8));

  std::sort(s.begin(), s.end(),
    [](const Student& s1, const Student& s2){
      return s1.getName() < s2.getName(); }); // 1
  output_by_student_name(s);

  std::sort(s.begin(), s.end(),
    [](const Student& s1, const Student& s2){
      return s1.getAge() < s2.getAge(); }); // 1
  output_by_student_name(s);

  auto f = [](const Student& s1, const Student& s2) {
    return s1.getGpa() < s2.getGpa(); }; // 2
  std::ranges::sort(s, f); // 3
  output_by_student_name(s);
  return 0;
}
```

注释 1：在需要的地方定义函数

注释 2：在需要之前定义函数

注释 3：使用范围并将函数对象作为参数传递

GPA 的排序在姓名和年龄方面略有不同，展示了使用范围 API 的替代方法，其更紧凑、更易读且更易于编写。那么为什么要等到开始使用范围呢?

Lambda 具有广泛的适用性；当一次性且不常调用的函数在编写，或研究的代码上下文中，难以找到和记住时，Lambda 函数提供了在需要时准确定义函数的机会，使开发人员和维护人员无需寻找其实现。

Lambda 函数最好只使用一次；如果多次调用，则将 Lambda 赋给变量。当然，在需要的地方定义函数的好处就消失了；与几乎所有事情一样，必须考虑竞争值的权衡。

**建议**

- 将 Lambda 用于简单、不常调用的函数；提供的文档可显著提高理解力，而不会占用过多的短期记忆。
- 考虑在现有代码使用简单函数的地方添加 Lambda。
- 标准模板库有几个函数可用于代替编写实现（例如，`std::max_element` 和 `std::less`、`std::sort`）。

- 考虑使用范围 API 来简化编码并方便阅读。开发者不必指定 begin 或 end 迭代器；范围会计算出这一点—只需提供容器和一个函数。

## 4.7. 错误 24： 手动定义笨拙的常量

这个错误会影响可读性、有效性，并且在一定程度上影响正确性。常量是必不可少的，但它们通常会导致代码难以阅读、值被截断，以及使用不灵活。

### 问题

许多程序必须定义常量或使用将单位按常量转换的函数，通常可以使用 #define 或 const 定义来指定常量或函数的名称和值，这里使用其将值从一种单位类型转换为另一种单位类型。

下面的代码展示了使用定义常量的三个例子。第一个是一年中的秒数（为简单起见，假设为非闰年）。但这个定义有一个问题：太大了。

清单 *4.16* 常数值的传统用法

```
const int SEC_PER_YEAR = 315360000; // 1

const double pi = 3.1415927; // 2
struct Circle {
  double radius;
  Circle(double r) : radius(r) {}
  double perimeter() throw() { return 2 * pi * radius; }
  double area() throw() { return pi * radius * radius; }
};

int main() {
  Circle c(3);
  std::cout << "perimeter " << c.perimeter() << ", area " << c.area() << '\n';

  double rads = 90 * pi / 180; // 3
  std::cout << "90 degrees is " << rads << " radians\n";

  std::cout << "There are " << SEC_PER_YEAR << " seconds in a year\n";
  return 0;
}
```

注释 1: 这正确吗? 还是 10 倍太大了?

注释 2: 不准确的近似值

注释 3: 不灵活的方法，仅适用于 90 度

### 分析

定义第一个常量时很容易犯错误。开发人员很难区分零的数量，错误地多加了一个。第二个是 pi 的截断值。开发人员为许多用途编写了一个合理的近似值，但考虑到数据类型可以表示的内容，这个近似值相对不精确。最后，将 90 度转换为弧度不够灵活——只适用于 90 度。如果能用函数来表示会更好。

### 解决

现代 C++ 为这三个问题提供了解决方案。首先，将解决 pi 的值。numbers 头文件定义了此值和其他几个值。系统架构会考虑这些值，并选择最佳值；无需手动定义。

使用千位分隔符改进了每年秒数的定义。撇号将常数值分成几组，这些组与书籍或文章中的读法相近。逗号已定义为操作符，因此选择了类似的符号。使用它简化了零的数量计算（在本例中），并使错误更加明显。

最后，将度数转换为弧度使用用户定义文字（UDL）。operator"" 是一个接受参数值的函数并对其进行转换。结果是一个可读的用法，可以处理任何合法的值。

清单 *4.17* 改进常数值的使用

```
constexpr double operator"" _deg_in_rad(long double d) { // 1
  return d * std::numbers::pi / 180;
}

constexpr int SEC_PER_YEAR = 31'536'000; // 2

struct Circle {
  double radius;
  Circle(double r) : radius(r) {}
  double perimeter() noexcept { return 2 *
  std::numbers::pi * radius; } // 3
  double area() noexcept { return std::numbers::pi * radius * radius; }
};

int main() {
  Circle c(3);
  std::cout << "perimeter " << c.perimeter() << ", area " << c.area()
    << '\n';

  std::cout << "90 degrees is " << 90.0_deg_in_rad << " radians\n";

  std::cout << "There are " << SEC_PER_YEAR << " seconds in a year\n";
  return 0;
}
```

注释 1：执行转换的用户定义文字

注释 2：易于阅读的精确分隔的数字

注释 3：该架构所能提供的最佳近似值

定义常量通常是必要的。可以使用的现代 C++ 技术，使代码更易读且更易于实现。

### 建议

- 使用千位分隔符来分隔长数字序列。
- 使用 operator"" 可将一个单位转换为另一个单位；可为数据类型创建文字 - 查看 std::chrono，具有强类型 minutes、hours 等，展示了这一强大的概念。
- 使用 numbers 头文件中定义的常量，来简化使用并获取系统的最佳近似值。

## 4.8. 错误 25： 编写模式匹配代码

这种错误会影响有效性和正确性，但对可读性也有负面影响。字符串数据中的模式匹配很常见；模式匹配代码费力又复杂，很难做到恰到好处。

### 问题

一个常见问题是文本数据的解析。输入以字符串数据的形式输入，而从文件读取的信息通常是文本。当所需的数据类型是字符串时，这种方法很好用，但需要验证文本或将文本分解为组成部分。除了最简单的情况外，分析数据格式、内容或类似内容的代码在所有情况下都具有挑战性。

清单 4.18 演示了电子邮件地址的验证。规则是单个单词（用户名）由 at 符号（@）分隔，后跟另一个单词（域），由句点分隔，后跟另一个单词（顶级域）。代码检查这两个分隔符和三个单词，其中一个单词是一个或多个字母（不包括 at 符号或句点）。如果有任何不妥，则电子邮件地址视为无效。

清单 *4.18* 手工编码的电子邮件验证函数（有 *bug*）

```
bool isValidEmail(const std::string& e) {
  size_t pos_at = e.find('@'); // 1
  if (pos_at == std::string::npos) // no '@' found
  return false;
  if (pos_at == 0) // no first word // 2
    return false;
  size_t pos_period = e.find('.', pos_at); // 1
  if (pos_period == std::string::npos) // no '.' found
    return false;
  if (pos_at + 1 >= pos_period) // no middle word // 2
    return false;
  if (pos_period == e.length() - 1) // no last word // 2
    return false;
  return true;
}
int main() {
  std::cout << isValidEmail("prof@nu.edu") << '\n';
  return 0;
}
```

注释 1: 解析分隔符

注释 2: 解析单词

### 分析

代码确定组成用户名、域和顶级域的字符（字符，甚至是无效字符）的存在。必须用 at 符号和句点将它们分开。这种匹配存在不精确性，但它可以正确解析表单，而无需检查其他有效性。恶意或错误的代码可能会让错误的电子邮件地址逃过此检查。增强此代码以确保用户名的字母、数字和标点符号正确，两个域部分（没有标点符号）的字母、数字和标点符号正确。编写解析器很困难，还要理解代码，必须对其进行大量文档记录带有注释。注释有一个令人讨厌的习惯，即在第一次维护代码后变得陈旧或错误。

**解决**

包含 regex 头文件，现代 C++ 中添加了文本模式匹配。正则表达式是一种计算机，是公认的最简单的类型。通常，其由状态机实现，每个字母（符号）都会导致从一个状态到另一个状态的转换。如果模式匹配，则认为已接受。清单 4.19 显示了该行为的实际效果。我们看到了五个基本部分，从模式的左侧开始向右移动：

1. 用户名，由一个或多个字母、数字和有限标点符号组成
2. at 符号分隔符
3. 域名，由一个或多个字母或数字组成
4. 句点（句点前的反斜杠转义符）
5. 顶级域名，由两个或多个字母组成

此解析器模式比清单 4.17 中的代码更精确，确保顶级域名至少有两个字母（从技术上讲，顶级域名可以有一个字符，但尚未注册；请参阅 http://data.iana.org/TLD/tlds-alpha-by-domain.txt 以获取当前列表）。此外，用户名和域名经过验证，仅包含允许的符号。这种精度对正确性有很大影响。

清单 *4.19* 使用正则表达式进行电子邮件地址验证

```
bool isValidEmail(const std::string& e) {
  std::regex pattern(
      R"([a-zA-Z0-9._%+-][CA}+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,})"; // 1
  return std::regex_search(e, pattern);
}

int main() {
  std::cout << isValidEmail("prof@nu.edu") << '\n';
  return 0;
}
```

注释 1：用于匹配电子邮件地址的 5 段正则表达式

另一个演示的功能是原始字符串。原始字符串以 R"( 开头，以)" 结尾。其目的是表达（通常是正则表达式模式）而无需转义任何字符。眼尖的读者会注意到正则表达式中有一个转义字符；转义是正则表达式的重要组成部分。通常，句点匹配单个字符（换行符除外），必须对其进行转义才能告诉正则表达式匹配句点。

当需要转义（非原始字符串）时，正则表达式的阅读难度更大，必须将模式与通常控制 C++ 格式的字符区分开来。一个简单的示例清楚地显示了其好处。假设匹配的是双反斜杠。使用转义字符，模式将如下所示：

```
std::string pattern = "\\\\";
```

使用原始字符串

```
std::string pattern = R"(\\)"
```

后者精确地显示了匹配的内容，而前者将其隐藏在转义字符中。

正则表达式旨在匹配文本模式。其语言很复杂，需要进行学习，但开发人员发现这种努力是值得的。理解了之后，模式就很容易编写、测试简单且表达紧凑。使用正则表达式可以获得更强大的功能。可以提取匹配模式的部分，例如用户名或所有三个部分。可以使用正则表达式用新值替换文本。

**建议**

- 正则表达式值得学习和使用；确保定制的文本模式匹配代码的正确性是一项挑战。
- 正则表达式表达能力很强，但读写却很困难；请使用能想到的每种数据变体。对其进行彻底测试。

# 第二部分 经典 C++

C++ 沿用了很多 C 风格的惯用法。虽然这些习惯曾经几乎不可或缺，但引入现代 C++ 功能使它们变得过时。许多这些习惯会带入当前的工作中，为当代开发带来了潜在的低效和陷阱。通过识别并用现代替代方案，替换这些过时的做法，开发人员可以清理他们的代码库，并充分利用 C++ 的全部功能，从而提高代码的清晰度和有效性。

虽然 C 和早期 C++ 编程的遗留问题，仍然存在于许多代码库中，但这些设计选择和范例已不再符合最佳实践的要求。这些曾经一流的方法，需要根据新技术和语言特性重新评估。从 C 和早期 C++ 过渡到现代实践需要重构代码以遵守当前标准，从而提高可维护性、可读性和效率。这种现代化工作可确保应用程序，仍能满足当今的期望和性能标准，并适应未来的发展。

# 第 5 章 C 语言的惯用法

本章内容

- 宏概述
- 使用特定宏
- C 风格字符串
- 布尔值的错误数据类型

C 是 C++ 的主要前身，其影响力遍及广泛的项目。许多 C++ 的早期采用者都是 C 开发者，他们也在寻找“更好的 C”。C++ 提供了与 C 的兼容性，但有少数例外。这些例外对于更好的类型检查、正确性和语言一致性是必不可少的。

然而，在 C++ 问世之后，C 编码的许多方面也从未改变。许多开发者学习了 C 中的风格和技术，并将它们带到了 C++ 中。编译器很少抱怨这些方法。生成的代码可以编译并运行—还能期待什么呢？但这种转变并不一定意味着，开发人员开始抛弃 C 语言的习语，并采用 C++ 方法。许多代码的编写并没有考虑到表达相同意图的新方法。

在这些年里，C++ 和 C 开始越来越分化。C99 标准明确并标准化了 C，同时从 C++ 中吸取了一些内容；C++ 采纳了 C 的一些创新。这两种语言仍然独立，但彼此之间相互促进。

语言的差异导致了一系列问题和错误，这些问题和错误是 C++ 代码中 C 风格的特点。这些错误大多无害，因为代码可以运行，但很难读写。

本章介绍了 C 开发人员用 C++ 编写的代码库中存在的几个问题。尽管存在更好的方法，但开发人员必须使用它来解决这些问题。为了改进 C++ 代码，必须了解 C 的影响，以及如何纠正其与 C 向后兼容性的常见误用。下一章将继续这个主题，但重点介绍误用 C++ 构造，这些构造通常以 C 语言的方式编写。

## 5.1. 错误 26： 总是在函数的顶部声明变量

此错误分别影响正确性、可读性和有效性。C 要求在函数顶部声明所有变量。

**问题**

许多语言要求在函数顶部声明所有变量，有时也定义所有变量。这是对编译器的让步，而不是开发人员的需求，这种做法存在一些问题。首先，当所有变量都在函数顶部声明时，读者和开发人员必须参考顶部来查看名称、记住它们的含义，并查看变量的使用位置。这种方法需要一定的认知负荷，这可能会很繁重。其次，开发人员必须在使用变量之前确定变量的初始化；方便起见，此初始化可能是一个简单的值（通常为 0），但不一定反映后面的代码所依赖的正确值。但如果开发者花时间正确地初始化该值，读者可能要到很晚才明白为什么。维护开发者可能需要稍后更改此值，从而引入问题。最后，这些初始化值可能经常看起来像是神奇的数字；为什么 1 应该被初始化为 0、1 或（显然）任意值？

清单 5.1 演示了其中的一些问题。开发人员需要初始化 max 变量，但由于“膝跳”值为 0，由于初始化不当，它会计算出错误的答案。此外，pos 变量在循环之前的代码中使用，并保留了一些值。原始代码假设 pos 将在（某处）初始化并在循环中正确使用。后来的开发人员忽略了这

个未记录的假设，并将“完美无缺”的变量用于其他目的。

清单 5.1 在函数的顶部声明变量

```
int maximum(const std::vector<int>& values) {
  int max = 0; // 1
  int pos; // 2

  // assume code here that uses pos ...
  pos = 1; // 3
  // assume more code here...
  for (; pos < values.size(); ++pos) // 4
    if (values[pos] > max)
      max = values[pos];
  return max;
}

int main() {
  std::vector<int> values;
  values.push_back(1);
  values.push_back(-2);
  values.push_back(-3);
  std::cout << maximum(values) << '\n';
  return 0;
}
```

注释 1：变量的初始化选择不当
注释 2：没有初始化；假定将在稍后发生
注释 3：添加的一些计算的结束值
注释 4：使用 pos 时假定其值有意义

**分析**

变量声明与变量使用之间的分离，使得变量在预期使用之前会发生奇怪的事情。pos 变量对于一些计算来说是一个不错的选择，但它处于不适合循环的状态。max 变量的初始化使用了一个通常正确且合适的值，但每个问题都可以使用这个初始值。由于 vector 中所有测试的值都是负数，所以没有一个大于错误初始化的值。这个错误引入了一个容器中未包含的外部值作为其最大值。

**解决**

考虑在使用变量之前声明它——尽可能缩短声明、定义和使用之间的距离。在某些情况下（for 循环），可以在结构范围内声明变量，确保它仅在那里可见（现代 C++ 允许在 if 语句中这样做）。

初始化变量是一项有趣的练习。许多学生倾向于始终使用 0 值来初始化变量。大多数情况下，这非常合适，但并非全部。这部分将 0 视为错误的问题将导致微妙的问题。清单 5.2 中的 maximum 函数通过在一组负值中搜索最大值来演示这一点。出于某种原因，太多开发人员在编写代码时只考虑正值。整数和实数有一个令人讨厌的习惯，就是经常为负数，不容忽视。在扫描值集合之前，一种简单的初始化方法是将第一个元素的值复制到变量，并比较其他值。这种方法可确保使用实际数据，而不是开发人员假设的值初始化变量。如前所述，使用开发人员选择的值初始化，

可能会使用数据集中不存在的值，从而导致错误。

清单 5.2 需要时声明变量

```cpp
int maximum(const std::vector<int>& values) {
  // assume code here...
  int max = values[0]; // 1
  for (int pos = 1; pos < values.size(); ++pos) // 2
    if (values[pos] > max)
      max = values[pos];
  return max;
}

int main() {
  std::vector<int> values;
  values.push_back(1);
  values.push_back(-2);
  values.push_back(-3);
  std::cout << maximum(values) << '\n';
  return 0;
}
```

注释 1：使用集合值之一初始化变量

注释 2：将循环控制变量的范围限制在循环内

记住变量的含义和值，对于代码的可读性和推理至关重要。在需要的地方准确声明变量，并在使用前初始化它们，可以减轻读者的认知负担。

**建议**

- 限制每个变量的范围；声明后立即用有意义的值进行初始化。
- 可在 `for` 循环范围和允许这种方法的其他构造中声明变量（现代 C++ 增加了更多机会）。

## 5.2. 错误 27： 宏依赖

这种错误会影响正确性、可读性、有效性，有时还会影响性能。宏是一种向程序源代码添加信息，而不直接由编译器检查的方法。编译受影响的源代码时会，检查宏的文本，所以某些错误可能不明显、难以调试。

**问题**

清单 5.3 中引入了 C++ 宏。通常，宏旨在通过赋予文字值一个有意义的名称来定义文字值，该名称记录了它们在程序中的用途。

另一个用途是生成函数式结构，来记录它们的使用。与大多数函数一样，可能需要参数，宏也可以很好地处理参数。

此外，宏可以通过在一个符号中，定义大量重复代码来减少编码工作量。以下代码尝试通过在一个符号中指定初始化、延续和更新部分，最大限度地减少编写 `for` 循环所需的工作量。有了所有这些（明显的）优势，建议少使用它们（如果有的话）似乎有些危言耸听。

清单 5.3 使用宏定义、减少和添加类型

```
#define PI 3.1415927 // 1
#define SQUARE(n) n * n // 2
#define FOR(a, b) for (i = a; i < b; ++i); // 3
#define FALSE 0 // 4
#define TRUE !FALSE

int main() {
  int n = 3;
  std::cout << SQUARE(n) << '\n';
  std::cout << SQUARE(n + (n - 1)) << '\n';
  std::cout << SQUARE(++n) << '\n';

  int i;
  FOR(0, 10)
    std::cout << i << '\n';

  int truth = FALSE;
  ++truth; // 5
  ++truth;

  if (truth == FALSE)
    std::cout << "smooth\n";
  else
    std::cout << "dismay\n";
  return 0;
}
```

注释 1：引入没有特定类型的文字值

注释 2：类似函数的构造，便于计算

注释 3：通过定义结构来减少编码工作量，但其中有一个错误

注释 4：另一个无类型值

注释 5：开发人员正在尝试转换 truth 的值

**分析**

宏最明显的问题是，在编译之前就会替换到代码中；预处理器在找到宏的地方进行文本替换，并在该位置插入文字宏定义。如果宏中出现错误，编译器可能会在稍后发现它，并报出一些与源代码不太相关的信息，编译的内容与编写的内容不同。这种情况在调试过程中会造成一些困难。

清单 5.3 中的代码存在几个与宏使用相关的问题。首先，PI 没有特定类型，通常会默认为 double；但有时，这是错误的，编译器可能会错误地确定其含义。

其次，SQUARE 宏看起来不错，但其中潜伏着危险。第一次调用平方后，结果正确。第二次调用失败，因为在宏扩展中，调用中的代码替换了两次。应该是 5 × 5，但变成了 3 + 2 × 3 + 2。第三次调用的结果为表达式 ++n * ++n，其中 n 为平方后的 5。

第三，FOR 宏有错误。for 循环头以分号结尾，所以没有循环体（实际上，它有一个空循环体）。因此，宏后面的代码是独立的，只执行一次，而不是预期的 10 次。另外，请注意，循环控

制变量必须在 FOR 宏之前定义；这一点必须清楚，以免造成误解。

第四，尝试定义布尔数据类型只打算使用 TRUE 和 FALSE 的值，这没问题，但开发人员希望对其进行类型转换。遗憾的是，由于 int 数据类型表示布尔值，所以递增会将值加 1。预期行为和实际行为不匹配，使用宏会欺骗大脑“看到”布尔约束，而编译器却完全不知道这些约束。

**解决**

这些错误可以使用 C++ 语言功能解决，确保编译器参与每一步并在使用错误时发出错误。PI 的文字值定义为双精度值。这种类型在大多数情况下可能是正确的，但开发人员必须进行确认。如果编译器发现不匹配将报错，开发人员需要重新评估他们的假设。现代 C++ 提供了 constexpr 形式，这可以缓解任何此类问题。这些表达式在编译期间进行计算，并保证是正确的类型。

其次，可以创建一个名为 square 的新函数。由于不确定需要什么数据类型，因此将其构建为模板。编译器将推断出正确的参数类型，并将该类型替换为参数和返回类型。这种方法可确保使用正确的代码，而无需进行可能影响精度和性能的隐式类型转换。

第三，删除了 FOR 宏，改用 for 循环。该宏并没有减少太多代码量，但却为循环控制变量引入了一个奇怪的作用域。在循环头中定义循环控制变量，可以将其作用域控制在循环内，从而提供更好的可读性和正确性。

第四，用 C++ 提供的 true 值类型 bool 替换奇怪的布尔值。检测到切换变量的错误，并发出错误，要求开发人员解决错误。必须根据定义使用该类型，并且假设必须与行为相匹配。

清单 *5.4* 使用 *C++* 结构代替宏的用法

```
const double PI = 3.1415927; // 1
template <typename T> // 2
T square(T n) {
  return n * n;
}
int main() {
  int n = 3;
  std::cout << square(n) << '\n';
  std::cout << square(n + (n - 1)) << '\n'; // 3
  std::cout << square(++n) << '\n';

  for( int i = 0; i < 10; ++ i) // 4
    std::cout << i << '\n';

  bool truth = false;
  truth = !truth; // 5
  truth = !truth;

  if (truth == false)
    std::cout << "smooth\n";
  else
    std::cout << "dismay\n";
  return 0;
}
```

注释 1：使用命名常量可以很好地记录
注释 2：模板允许推断参数和返回类型
注释 3：参数表达式在调用之前进行计算
注释 4：变量的作用域仅限于循环
注释 5：布尔值的切换（仅限于数据类型）

这些问题足以表明宏的危害大于帮助。互联网上有很多帖子认为在 C++ 中可以使用宏，在有限的范围内，这个建议当然正确。

我唯一同意使用宏通常合适的地方是，那些已经存在的（不幸的）具有条件编译要求的代码库。这种情况下，应删除一个条件部分和另一个条件部分之间不同的代码，并将其放入头文件中的函数中。然后，使用条件编译在函数之间进行选择。

**建议**

- 限制或消除宏以保证安全，并生成可预测的代码。
- 使用提供宏预期好处的 C++ 功能；编译器可以对其进行错误检查，并发出有意义的消息。

## 5.3. 错误 28： NULL 的误解

此错误会影响可读性，有时还会影响效率。误解 NULL 宏可能会导致正确性问题。

**问题**

一般建议是尽可能避免使用宏。此建议至关重要，因为 NULL 宏值和类型不确定。此代码是在 64 位架构上编写的，其中指针的大小为 64 位，这也是 long 的大小。您的系统可能具有不同的大小和不同的 NULL 定义。我的系统上的 unicode/utypes.h 中的定义如下：

```
#define NULL ((void *)0)
```

清单 5.5 中 NULL 有两种使用方式，但这两种方式都不明显。以下清单中的第一个重载 compute 函数采用 long 值，而第二个重载函数采用 long 指针。对 compute 的第一次调用传递了初始化为 NULL 的 long，该调用的明显选择是匹配第一个函数。第二次调用传递了 NULL 值，该值应与第二个 compute 函数匹配。

清单 *5.5* 以两种方式误用 *NULL*

```
long compute(long n) { // increment the value
  return ++n;
}

long compute(long* p) { // increment valid dereferenced pointer value
  if (p)
    return ++*p;
  return 0;
}

int main() {
  long x = NULL; // 1
```

```
  long n = compute(x); // 2
  std::cout << n << '\n';
  n = compute(NULL); // 3
  std::cout << n << '\n';
  return 0;
}
```

注释 1：使用 NULL 作为整数零值，这不是好的方式
注释 2：long 值应该与第一个函数匹配
注释 3：NULL 指针应该与第二个函数匹配

**分析**

代码似乎先调用第一个函数，然后调用第二个函数，但这并不正确。第一个调用匹配 long 值（第一个函数），第二个调用也匹配 long 值（第一个函数）。这不明显吗？NULL 宏读作“未引用有效内存对象的指针”。清单 5.5 中的代码使用 NULL 的定义，即 long 整数值 0；它是 not 指针。但这是该宏的部分问题，读起来像一个指针，并用作指针，但它不是指针。读者很容易被误导。

此外，不同架构上的不同编译器对 NULL 的定义可能不同，一个系统上的任何“有效”使用不一定适用于另一个系统。如果开发人员在其系统上使用 NULL，他们可能会假设另一个系统会以相同的方式工作。他们错了。NULL 没有明确的定义，因此难以一致地使用。在未将其定义为指针的系统中，这并不正确（C 确实将其定义为指针）。一些实现将 NULL 定义为 0 的整数值，其大小与指针大小匹配。但如果不检查，您能确定吗？

第一行代码将 NULL 用作 0 值。这看起来像是一个专门用于教学目的的人为错误，但这种情况经常发生，一些（也许是大多数）编译器会检测并警告这种不正确的使用。我曾在代码库中看到过这种情况，其中初始化器和参数值需要为 0——太可怕了！

**解决**

首先，不要将 NULL 用作巧妙的 0 值。其次，请注意 NULL 不是指针，不能用作指针。描述不指向有效内存对象的指针，经典 C++ 中其值为 0。如果开发人员想要使用 NULL 来实现指针，则必须将其转换为适当大小的指针。使用 NULL 的一个优点是，它能比 0 值更好地传达了指针语义。

清单 *5.6* 以更好的方式使用 *NULL*

```
long compute(long n) { // increment the value
  return ++n;
}

long compute(long* p) { // increment the value
  if (p)
    return ++*p;
  return 0;
}

int main() {
```

```
  long m = 0; // 1
  long n = compute(m); // 2
  std::cout << n << '\n';
  n = compute((long*)NULL); // 3
  std::cout << n << '\n';
  return 0;
}
```

注释 1：在需要 0 值的地方使用 0
注释 2：匹配第一个函数，保留值语义
注释 3：与第二个函数匹配，保留指针语义

NULL 宏明确指出了分配指针的意图，即不指向任何对象的值，但在转换宏时使用 0 值来消除歧义会更好。

现代 C++ 提供了一个语言关键字来解决 NULL 宏或零值的缺点，以下代码片段显示了如何正确地将指针变量分配给空值。nullptr 关键字是一个指针文字，因此不会发生隐式转换：

```
long* p = new long(42);
...
delete p;
p = nullptr;
```

**建议**

- 使用 0 值声明一个不指向有效内存对象的指针。
- 避免使用 NULL 作为 0 值。
- 避免使用 NULL 作为指针；它不是指针。

## 5.4. 错误 29： FILE 访问文件

这个错误影响了效率和可读性。开发人员多年来一直在使用 C 语言的 FILE 对象，但它通常比预想的更复杂。

**问题**

开发人员经常需要打开文件并读取文本行，而 C 语言的 FILE 对象已在 C++ 代码中广泛使用。以下清单中的代码显示了一种常见方法，即尝试打开文件；测试此假设是否准确；如果正确，则继续逐行读取文件。

清单 5.7 使用 FILE 读取文本行

```
const int SIZE = 100; // 1
int main() {
  FILE* file;
  file = fopen("data.txt", "r");
  if (!file) { // 2
    std::cerr << "Error opening file\n";
    return 1;
```

```
  }
  char buffer[SIZE];
  while (!feof(file)) {
    if (!fgets(buffer, SIZE, file)) // 3
      break;
    std::cout << buffer;
  }
  return 0;
}
```

注释 1：希望每行不超过 99 个字符

注释 2：首次使用负逻辑

注释 3：负逻辑，希望读取整行

### 分析

清单 5.7 中的代码运行良好，实现了编写它的目的。代码在三个地方使用了负逻辑：测试是否成功打开文件、测试行尾条件以及测试是否已读取行。负逻辑总是比正逻辑更耗费认知。当负逻辑嵌套时，负载会增加，如本例所示，编写负逻辑会更加困难。看似自然的方法必须反转，并且结果流必须保存在短期记忆中。

代码的另一个问题是行长。代码假设每行的长度不会超过 99 个字符，这个假设合理吗？无论使用什么值，都存在这样的可能：要么太小，导致每行读取多次；要么太大，导致内存使用效率低下。

虽然从技术上来说这不是错误，但缓冲区定义应该大一个字符以预测行尾字符。假设一行文本恰好是 SIZE 个字符，则将在循环的下一次迭代中读取行尾字符。这种方法对性能有轻微影响，因为每行都应该进行一次读取操作，且 SIZE 没有很好定义。

最后，FILE 对象保持打开状态。长期运行的程序如果存在此错误，可能会对系统资源产生负面影响，并可能导致资源耗尽。

### 解决

这些问题可以使用 C++ 的 std::ifstream 对象来解决。清单 5.8 中没有负逻辑，更容易编写和阅读。缺乏负逻辑意味着代码读起来很简单。如果未打开输入文件，则失败测试读起来很自然。

此解决方案没有缓冲区；而是使用 std::string 对象。每行输入所需的大小都是 line 变量的长度。此外，在创建每行时，字符串中不存在先前的数据。执行部分读取时，缓冲区总是容易受到其他数据的影响。

std::ifstream 对象在 main 函数结束时超出范围。这一事实确保文件将关闭。这是 RAII 模式的一个示例，其中构造函数-析构函数对管理动态资源。可以关闭流，但没有必要。这种方法简化了写入和读取。开发人员必须确保记录自动销毁；否则，他们只能猜测会发生什么。标准模板库在这方面非常出色。

清单 *5.8* 使用 *ifstream* 读取文本行

```cpp
int main() {
  std::ifstream file("data.txt");
  if (file.fail()) { // 1
    std::cerr << "Error opening file\n";
    return 1;
  }
  std::string line;
  while (std::getline(file, line)) // 2
  std::cout << line << '\n';
  return 0;
}
```

注释 1：积极逻辑使阅读和写作更容易

注释 2：更积极的逻辑，不覆盖旧数据

`std::ifstream` 方法是一种读取文本数据，并测试流中是否存在影响其操作的错误的简单方法。此方法优于从 C 语言继承的 `FILE*` 方法。

**建议**

- 使用 `std::ifstream` 和 `std::ofstream` 对象简化文件数据的读取和写入。
- 尽可能使用正逻辑。

## 5.5. 错误 30： 将整数值转为布尔值

此错误影响正确性和可读性。C 语言没有提供表示布尔值的数据类型；C++ 后来进行了支持。早期代码必须使用该方式，并且通常将其实现为整数。

**问题**

使用整数来捕获 `FALSE` 和 `TRUE` 的值是有意义的；`0` 表示 `FALSE`，而 `1` 表示 `TRUE`(实际上，非零值都可用作 `TRUE`)。清单 5.9 提出了这一概念，并展示了一些使用这一想法的代码。使定义保持不变至关重要，这样真值就不会改变。当然，由于布尔值是作为整数实现的，因此整数可以做的事情，布尔变量也可以做到。这会导致一些奇怪的代码的出现。

清单 *5.9* 使用整数模拟布尔值

```cpp
const int TRUE = 1;
const int FALSE = 0;
int main() {
  int truth = FALSE;
  std::cout << "truth is " << (truth ? "real" : "illusory") << "\n";
  ++truth; // 1
  std::cout << "truth is " << (truth ? "real" : "illusory") << "\n";
  truth = 42; // 2
  std::cout << "truth is " << (truth ? "real" : "illusory") << "\n";
  return 0;
}
```

注释 1：增加布尔值意味着什么？

注释 2：分配除 FALSE 或 TRUE 之外，其他值意义是什么？

许多早期的开发者发现，使用枚举可以更好地实现布尔值的意图。这种方法限制了对表示布尔值的变量执行的操作。以下代码演示了这种方法。

清单 *5.10* 用枚举模拟布尔值

```
enum BOOL { FALSE, TRUE };
int main() {
  BOOL truth = FALSE;
  std::cout << "truth is " << (truth ? "real" : "illusory") << "\n";
  truth = TRUE; // 1
  std::cout << "truth is " << (truth ? "real" : "illusory") << "\n";
  return 0;
}
```

注释 1：分配仅限于 FALSE 和 TRUE。

**分析**

将布尔值表示为其他数据类型的主要问题在于意图。整数表示允许执行没有意义但合法的操作，枚举解决方案要好得多，并且提供了一组更受限制的操作。枚举解决方案的主要问题是它不标准，并且并非所有开发人员都必须以这种方式实现它。缺乏共同的理解，使得这个解决方案的结果没那么理想。

**解决**

C++ 语言开发人员意识到上述两种解决方案存在问题，并添加了 bool 数据类型，表示一个布尔值，其意图和内容应该更清晰。他们的意图是正确的，但实现可能更成功。bool 是一种数据类型，但编译器将 bool 实现为整数，这允许进行诸如清单 5.9 中的恶意操作。语言标准没有定义除 0 或 1 以外的值的行为；其他值都是未定义行为，但此定义并不能防止滥用。提供单独的数据类型，记录了变量的目的和意图，但在滥用时不会强制执行严格的语义。

从 C++17 开始，删除了增加 bool 变量的功能；此更改提高了数据类型操作集的正确性。以下代码显示了 bool 变量的用法，该变量应仅采用 false 和 true 值。

清单 *5.11* 使用 *bool* 数据类型

```
int main() {
  bool truth = false;
  std::cout << "truth is " << (truth ? "real" : "illusory") << "\n";
  ++truth;
  std::cout << "truth is " << (truth ? "real" : "illusory") << "\n";
  truth = 42;
  std::cout << "truth is " << (truth ? "real" : "illusory") << "\n";
  return 0;
}
```

bool 数据类型必须按照其定义的操作集使用，并且仅表示布尔值。

**建议**

- 使用提供的 `bool` 数据类型来表示真值概念；其工作良好，并且能够很好地记录意图。
- 除了值 `false` 和 `true` 之外，不要将其他值分配给 `bool` 变量。
- 如果可能，请使用 C++17 或更高版本进行编译，以获得更好的 `bool` 语义。

## 5.6. 错误 31： C 风格的强制转换

此错误会影响正确性和可读性，并对效率产生轻微的负面影响。对效率的负面影响是故意的；C++ 风格的强制类型转换很大、很丑陋，而且很难编写，因此建议谨慎使用。

**问题**

需要进行类型转换的主要情况有四种：

- 绕过 `const` 限制
- 将底层数据重新解释为新类型
- 将一种类型转换为另一种类型
- 将基指针向下转换为派生指针

C 风格的强制转换可以完成所有四种行为，而不会导致编译问题，但成功的编译与有意义的编译不同。考虑清单 5.12 中的代码。第一部分围绕字符数组的 `const` 性进行强制转换，并允许数据修改。第二部分将整数值视为字符序列。第三部分是伪装成双指针的字符数组，这毫无意义。第四部分采用基类对象，并将其视为派生对象。

清单 *5.12* 使用危险地 *C* 风格强制转换

```
struct B {
  virtual int compute() { return 0; }
};
struct D : public B {
  int n;
  D() : n(42) {}
  int compute() { return n; }
};

int main() {
  const char msg[] = "Hello, world";
  char* p = (char*)msg; // 1
  *(p+1) = 'a';
  std::cout << msg << '\n';

  int n = 0x2A2A2A2A;
  char* c = (char*)(&n); // 2
  for (int i = 0; i < sizeof(int); ++i)
    std::cout << *(c+i);
  std::cout << '\n';

```

```
    std::cout << *((double*)msg) << '\n'; // 3

    B* b = new B();
    D* d = (D*)b; // 4
    std::cout << d->compute() << '\n';

    return 0;
}
```

注释 1：绕过字符数组的 const 属性
注释 2：将整数视为字节集合
注释 3：这没有任何意义，也可以通过编译
注释 4：D 是 B，但 B 可能不是 D

**分析**

第一个问题违反了数据类型的语义，应该很少发生。编译器默认不会发现错误，但可以配置为发现错误。这种“松散性”，可让开发者对数据做任何事情。

第二个问题很“聪明”，但通常有点牵强；虽然它能工作，但需要更好的类型和设计。

第三个错误令人厌恶，完全破坏语义——两种类型不兼容且不可翻译，但这段代码可以编译，导致未定义的行为。

最后，第四个问题可能有应用（很少），并且以这种方式完成时很危险。派生类中的虚函数没有正确调用，如果正确调用，则会访问不存在的实例变量——未定义的行为！

**解决**

每种强制类型转换情况，都需要比 C 风格强制类型转换更有意义的东西替代。C 风格强制类型转换存在两个主要问题：首先，需要传达开发者的意图，其次，需要澄清一些转换。C++ 风格强制类型转换故意设计的很长，感觉很笨拙，但它们可以很好地记录意图，让开发人员三思而后行。通常，设计更好的代码可以消除强制类型转换的需要，但有时也必须要使用。

const_cast 有意除去（或添加，如果不是常量）实体的 const 性。混合 C 和 C++ 代码时，这种强制类型转换通常是必要的—请负责任地使用它。

reinterpret_cast 应谨慎使用。此强制类型转换告诉编译器，忽略它所知道的指向数据（其类型），并假装它是要强制类型转换的类型；这是一种“假装”的情况。编译器会屈从于开发者的愿望，但不承担任何责任。使用数字数据写入和读取二进制文件就是这种情况，强制类型转换很好地记录了意图。

static_cast 将一种兼容的数据类型转换为另一种。编译器检查数据类型的兼容性；如果不兼容，会发出错误。在所有强制类型转换中，这是最有可能需要的。例如，使用此强制类型转换可以快速将双精度值截断为整数。

最后，dynamic_cast 将基类指针或引用转换为派生类指针或引用。如果使用此技术，请确保至少有一个虚函数（没有虚函数就不要继承！）；否则，没有规范的方法来存储用于向下转换的类类型。转换的结果在运行时进行检查，因此可以干净地编译但在运行时崩溃。兼容转换的结果是一个指针，而不兼容转换的结果是一个空指针，但开发人员应该极力避免采用这种方法。如果引用转换失败，则会引发异常。C++ 为此提供了更好的机制，尤其是具有多态行为的继承。

清单 5.13 更谨慎、更负责地使用 C++ 风格强制转换

```cpp
struct B {
  virtual int compute() { return 0; }
};

struct D : public B {
  int n;
  D() : n(42) {}
  int compute() { return n; }
};

int main() {
  const char msg[] = "Hello, world";
  char* p = const_cast<char*>(msg); // 1
  *(p+1) = 'a';
  std::cout << msg << '\n';

  int n = 0x2A2A2A2A;
  char* c = reinterpret_cast<char*>(&n); // 2
  for (int i = 0; i < sizeof(int); ++i)
    std::cout << *(c+i);
  std::cout << '\n';

  // std::cout << static_cast<double*>(msg) << '\n'; // 3
  std::cout << static_cast<double>(n) << '\n'; // 4

  B* b = new B();
  D* d = dynamic_cast<D*>(b); // 5
  if (d)
    std::cout << d->compute() << '\n';
  else
    std::cout << "incompatible downcast\n";
  return 0;
}
```

注释 1：假设这是正确的，并且意图可查

注释 2：另一个可查的案例

注释 3：编译器确定类型不兼容

注释 4：合理的类型转换

注释 5：如果兼容，则指针有效，否则为 NULL(nullptr)

如果使用不当，C 风格强制类型转换和 reinterpret_cast 可能会导致未定义行为。如果发现强制类型转换是必要的，请考虑重新设计代码。如果必须使用强制类型转换，请仅在转换尽可能安全的情况下才这样做。

**建议**

- 尽可能避免强制类型转换。

- 将 C++ 代码与 C 函数接口时使用 const_cast。
- 检查 dynamic_cast 的结果以确保指针兼容。
- 使用 reinterpret_cast 之前请仔细考虑；并非所有架构都以相同的方式表示数据，因此转换结果可能不同；这样做很容易陷入未定义行为的陷阱。

## 5.7. 错误 32： atoi 转换文本

此错误主要关注正确性，并包含大量可读性和有效性。键盘输入始终以字符流的形式完成，但这些字符没有固有含义。换句话说，没有上下文来了解数据类型应该是什么。

C 为编译器无法直接转换的数据类型提供了转换函数，atoi 函数用于将 C 样式字符串转换为整数值。sprintf 类函数可反转方向，获取数值数据并构建相应的 C 样式字符串。

### 问题

atoi 函数分析 C 样式字符串并跳过前置空格。从第一个非空格字符开始，扫描与整数值一致的字符。这些字符包括前置加号或减号和十进制数字。发现的第一个不符合此模式的字符可视为分隔符，转换过程结束。

如果输入的是空字符串或者没有找到有效的整数类型字符，函数会返回 0 值。以下代码需要输入年龄。

清单 *5.14* 输入表示年龄的信息

```
int main() {
  std::cout << "Enter your age: ";
  std::string input;
  std::cin >> input;

  int age = atoi(input.c_str()); // 1
  std::cout << "On your next birthday, you will be " << age + 1 << " years
  old\n";
  return 0;
}
```

注释 1：将无效输入和零值转换为 0

假设输入的是有效数字，则一切正常。但如果收到无效输入，转换函数将失败，并通过返回 0 来表示。如果输入恰好是 0，转换函数还会返回 0。缺少的是返回的 0 是有效（例如，输入了 0 数字）还是无效（例如，输入了 cat）的基本知识。

### 分析

atoi 类函数的根本问题是 0 返回值。如果清单 5.14 中代码的用户恰好是婴儿，则输出结果是正确的。如果用户在键盘上乱摸，输入了 0 和 9 之间的数字以外的内容，则输出结果不正确。

添加测试以确定值是否合法为 0，需要检查每个输入数字以确保它们在有效范围内。即使添加了此检查功能，结果也会很混乱且难以阅读。由于 C++ 提供了为转换添加智能的方法，因此请这样做以验证输入。

**解决**

虽然解决这个问题的方法不止一种，但最简单的解决方案是使用 stringstream 类，使用熟悉的流和提取操作符来解析字符串数据。此方法的一个好处是能够测试输入字符串流是否失败，而 atoi 类型函数不提供此功能。

清单 *5.15* 检查输入的年龄

```
int main() {
  std::cout << "Enter your age: ";
  std::string input;
  std::cin >> input;

  std::istringstream is(input);
  int age;
  is >> age;
  if (is.fail()) { // 1
    std::cout << "Invalid input\n";
    return 1;
  }
  std::cout << "On your next birthday, you will be " << age + 1 << " years old\n";
  return 0;
}
```

注释 1：可以检查无效输入，并进行处理

如果未选中 fail 函数，则默认行为与 atoi 相同—对于无效数据，返回零值。检查流中是否存在不正确的数据；对象很聪明，会使用它们的知识，可确保程序正确。使用提取操作符，可以轻松检查将文本转换为预期数据类型是否失败。C++11 提供 std::stoi 函数，用于将 C 样式或 C++ 样式的字符串转换为数值，并在失败时抛出异常。C++17 提供 std::from_chars 以有效地将文本转换为数字，并允许检查错误。要查看这些选项的实际效果，请考虑以下代码。

清单 *5.16* 检查年龄的输入（现代方式）

```
int main() {
  std::cout << "Enter your age: ";
  std::string input;
  std::cin >> input;
  int age;

  // C++11
  try {
    age = std::stoi(input);
  } catch (const std::invalid_argument& e) {
    std::cout << "stoi: Invalid input\n";
    return 1;
  }
  std::cout << "On your next birthday, you will be " << age + 1 << " years old\n";

```

```
  // C++17
  auto result = std::from_chars(input.data(), input.data() + input.size(), age);
  if (result.ec != std::errc()) {
    std::cout << "from_chars: Invalid input\n";
    return 1;
  }
  std::cout << "On your next birthday, you will be " << age + 1 << " years old\n";
  return 0;
}
```

**建议**

- 使用输入流，可以测试问题。
- 尽可能使用抛出异常的函数，很容易检查。
- 尽可能使用 C++ 功能，对 atoi 的调用更改为其他更智能的解决方案。
- 如果可以使用现代 C++，请考虑使用 std::stoi 或 std::from_chars 函数。

## 5.8. 错误 33： C 风格的字符串

此错误会影响可读性和有效性；有时，性能也会受到影响。C 风格字符串是字符数组，其最后一个字符为零值，零值是标记值，表示字符串的结尾。

**问题**

C 没有 class 或所谓的智能对象的概念。相反，程序代码往往充满了用于管理愚蠢对象的特定命令。每个开发人员都倾向于在很多地方处理 C 风格的字符串，经常重复代码和重复（通常通过复制和粘贴）功能。

清单 5.17 中的代码是众多可能示例之一。较旧的代码倾向于始终使用 C 样式的字符串，许多函数都有 char* 参数，鼓励在整个代码库中使用它们。

考虑以下代码，开发人员计算 C 风格字符串中某个字符的出现次数。该代码很简洁，并且完全满足了需要。看起来性能很好，并且是一个 O(n) 解决方案。谁会对此不满意呢?

清单 *5.17* 简单的搜索和追加算法

```
int freq(const char* s, char k) {
  int count = 0;
  for (int i = 0; i < strlen(s); ++i) // 1
    if (s[i] == k)
      ++count;
  return count;
}

char* concat(const char* lhs, const char* rhs) {
  char* buffer = new char[strlen(lhs) + strlen(rhs)]; // 2
  strcpy(buffer, lhs);
  strcat(buffer, rhs);
```

```
  return buffer;
}

int main() {
  const char* msg = "Hello, world";
  char letter = 'l';
  std::cout << letter << " occurs " << freq(msg, letter) << " times\n";
  const char* msg2 = ", come on in!";
  std::cout << concat(msg, msg2) << '\n';
}
```

注释 1：对字符串进行线性搜索

注释 2：动态内存分配——正确吗?

## 分析

对于相对较短的字符串，这种方法效果很好；但对于长字符串，其性能损失可能不可接受。可以使用分析器来查找，需要更好的解决方案的地方。循环分析似乎显示线性搜索，其时间复杂度为线性 O(n)。算法是二次 O(n2)。为什么? C 样式字符串很笨；代码不能简单地向字符串询问信息以了解其长度。相反，每次请求时都必须计算大小。循环必须在循环的每次迭代中计算字符串的长度；所以每次执行 strlen 本身都是线性 O(n)，并且执行 n 次，从而得到二次解。值得庆幸的是，大多数编译器都会检测到这种情况，并将 strlen 函数移出循环，但不要相信编译器会为所有可能性找出答案。并且这种方法并不理想，它仍然专注于 C 样式字符串。清单 5.18 中的代码提高了性能，但并没有真正改进方法。

循环范围之外的其他代码，都必须重新计算字符串的长度。如果字符串处理代码遍布整个代码库，那么在一个地方进行长度计算不是件好事。

管理字符数组也很困难。必须了解 strcpy 和 strcat 之间的区别，并记住如何处理空字符。需要确保目标缓冲区的长度，至少为字符串的长度加上终止字符的长度。问题是开发人员必须记住并管理所有细节，大多数情况下这些细节都是正确的。但在极少数情况下，如果操作失误，就会出现未定义的行为。通常，如果目标太小，复制或附加似乎仍然有效，但会损坏数据。好的系统会崩溃（并非所有系统都是好的）。

freq 函数中的字符串长度计算不会使 concat 函数中的相同计算受益。此外，concat 函数中获得的动态内存未释放。这可能看起来无害，但这个错误会影响正确性。

清单 5.18 改进的搜索和追加算法

```
int freq(const char* s, char k) {
  int count = 0;
  int len = strlen(s); // 1
  for (int i = 0; i < len; ++i)
    if (s[i] == k)
      ++count;
  return count;
}

char* concat(const char* lhs, const char* rhs) {
```

```
  char* buffer = new char[strlen(lhs) + strlen(rhs) + 1]; // 2
  strcpy(buffer, lhs);
  strcat(buffer, rhs);
  return buffer;
}

int main() {
  const char* msg = "Hello, world";
  char letter = 'l';
  std::cout << letter << " occurs " << freq(msg, letter) << " times\n";
  const char* msg2 = ", come on in!";
  std::cout << concat(msg, msg2) << '\n';
  delete [] msg;
}
```

注释 1：计算一次该函数的长度

注释 2：确保空终止符有足够的空间

**解决**

C++ 比 C 风格的字符串有了显著的改进。它们的问题非常严重，以至于自 C++ 早期以来提供的首批主要用户编写的类之一就是 string 类。C++ 风格的 string 是一个智能对象；开发人员可以向它提问并获得答案。以下代码利用了这一事实来查询 string 的长度。

清单 *5.19* 使用 *c++* 字符串解决检测到的问题

```
int freq(const std::string& s, char k) {
  int count = 0;
  for (int i = 0; i < s.length(); ++i) // 1
    if (s[i] == k)
      ++count;
  return count;
}
std::string concat(const std::string& lhs, const std::string& rhs) {
  return lhs + rhs; // 2
}

int main() {
  std::string msg = "Hello, world";
  char letter = 'l';
  std::cout << letter << " occurs " << freq(msg, letter) << " times\n";
  std::string msg2 = ", come on in!";
  std::cout << concat(msg, msg2) << '\n'; // 3
}
```

注释 1：长度的恒定时间查询

注释 2：开发人员不必关心数据移动的长度

注释 3：自动处理为临时对象获取的内存

该解决方案可以更好，但在改进之前的代码的同时，也符合之前的代码。使用字符串长度初始

化的局部变量更好，但这种方法展示了智能对象的好处。将字符串长度查询留在循环中表明，尽管该函数调用了 n 次，但循环的计算成本为 O(n)。由于查询长度不会触发长度计算，因此保留了此时间特性。字符串在构造时确定其长度，并将结果存储在实例变量中，因此长度查询在 O(1) 时间内完成。

通过让对象本身确定所需的内存，两个字符串的连接解决了管理结果字符串长度的难题。开发人员不需要知道数据是如何处理的——是否使用空终止符？此外，因为字符串使用 RAII 模式管理其数据，所以不需要删除动态内存。

**建议**

- 学习使用字符序列时，可以使用 C++ 样式的字符串；字符串类经过高度优化且易于使用。
- 尽可能消除 C 样式的字符串，并引入 C++ 样式的字符串进行代替。
- 字符串自动处理动态内存；当字符串超出范围时，将调用其析构函数。

## 5.9. 错误 34： exit 函数

这个错误与正确性有关。使用 exit 函数可能会导致有限或动态资源泄漏，从而影响系统的稳定性。

**问题**

假设开发人员需要遵循一些可疑的要求。如果发现有员工解雇，程序必须在工资单处理期间立即崩溃。开发人员理解这一要求，并记得以前的 C 程序会立即使用 exit 调用来终止；因此，开发人员决定在 C++ 程序中使用此技术。后来，在出现一些奇怪的问题后，系统开发者会询问开发人员，为什么这个程序会导致系统稳定性问题。

清单 5.20 使用 *exit* 结束程序

```
class Connection {
private:
  int conn;
public:
  Connection(const std::string& name) : conn(0) {
    conn = 1; // assume: database connection resource is returned
  }
  ~Connection() {
    if (conn)
      conn = 0; // assume: destroys database connection resource
    std::cerr << "Connection destroyed\n";
  }
};

struct Employee {
  bool isTerminated() { return true; }
  double computePay() { return 42.0; }
};

```

```
int main() {
  Connection c("payroll");
  Employee emp;
  if (emp.isTerminated())
    exit(-1); // 1
  std::cout << emp.computePay() << '\n';
  return 0;
}
```

注释 1：程序必须在此终止

### 分析

程序分配数据库连接、处理员工，并在发现解雇的人员时崩溃。清理数据库后重新运行程序，可能又崩溃了几次。处理结束时，数据库清理干净，程序运行完成，每个人都拿到了工资——快乐的一天！但为什么会出现系统稳定性问题？问题出在 Connection 类，其构造函数访问了有限的资源，而析构函数是为了释放这些资源而设计的。正常情况下，此代码可以正常工作。但问题会在特殊情况下出现，exit 调用强制程序立即终止。Connection 对象将不会进行清理，所以其析构函数无法运行。

动态资源已分配，但没有使用——典型的资源泄漏。

### 解决

解决这个问题需要两个步骤。首先，不要使用 exit 调用；而是抛出异常。其次，将最外层的代码包装在一个通用的 try/catch 块中，其异常操作是重新抛出异常。当然，catch 子句可以更具体，但在解决这个问题时，代码的智能是有限的。

这种方法可以防止资源泄漏，因为析构函数有机会运行。当 Connection 对象超出范围时，其析构函数会将数据库连接返回到池中，从而防止分配但未使用的资源。

代码很笨拙，但在许多情况下，开发人员发现自己需要改进整体结构或实现。这些情况下，“廉价”解决方案是必要的，即使设计很差。开发人员很少能负担得起，改进旧代码所需的时间和预算——这是一个在他们可以适应的时间和地点，并在有限的领域寻求改进的问题。现实有时很残酷。

清单 *5.21* 使用异常终止程序

```
class Connection {
private:
  int conn;
public:
  Connection(const std::string& name) : conn(0) {
    conn = 1; // assume: database connection resource is allocated
  }
  ~Connection() {
    if (conn)
      conn = 0; // assume: returns database connection to pool
    std::cerr << "Connection destroyed\n";
  }
};
```

```
14
15  struct Employee {
16    bool isTerminated() { return true; }
17    double computePay() { return 42.0; }
18  };
19
20  struct TerminatedEmployee {
21    std::string message;
22    TerminatedEmployee(const std::string& msg) : message(msg) {}
23  };
24
25  int main() {
26    try {
27      Connection c("payroll");
28      Employee emp;
29      if (emp.isTerminated())
30      throw TerminatedEmployee("Employee was terminated");
31      std::cout << emp.computePay() << '\n';
32    } catch (...) {
33      throw; // 1
34    }
35    return 0;
36  }
```

注释 1：此时终止程序

某些情况下，有必要提前退出程序。需要时，请确保对 `exit` 进行编码，以便在退出之前调用所有析构函数。只有在不需要堆栈展开（调试）的情况下，程序才应该 `exit` ，而不清理资源。请注意不要泄漏系统资源，例如：套接字、数据库连接和类似的实体或对象。

**建议**

- 避免调用 `exit` 。
- 抛出异常，一般性地（或具体地）捕获它，然后重新抛出，以便让析构函数有机会运行。
- 尽其所能地改进；很可能不会有很好的机会来改进大量代码。

## 5.10. 错误 35： 优先选择数组

这个错误与有效性和可读性有关；会影响正确性，如果是这样，会产生不利影响。C 提供了一个内置数据容器，数组是可以用一个名称寻址，并通过索引区分的元素序列。

**问题**

许多编程问题都需要集合，无论是内置的还是用户编写的。如果对象是用户编写的并存储在数组中，则必须存在默认构造函数。在某些情况下，此构造函数有意义，但在许多情况下，它没有意义。毫无意义的默认构造函数表明需要更正或完成设计，但使用数组不是一种选择。

创建静态数组时，必须在编译时知道元素的数量。许多问题没有清楚地表达必须处理的元素数量，所以可以选择任意大小。如果开发人员的猜测太大，就会浪费空间；如果猜测太小，这通常会

导致崩溃，甚至更糟。这里一个很好的选择是使用动态数组，代码可以在需要容器之前确定元素的数量。开发人员必须记住管理容器的内存——动态数组也需要默认构造函数。

无论基于数组的容器有多少元素，添加过多的元素都是可能的。使用动态数组允许开发人员通过分配较大的数组，并复制值来管理过短的数组。如果该数组被其他实体或对象引用，开发人员通常会感到意外，因为他们不知道分配的内存已转移。

最后，删除不再需要或有效的数组元素。使用数组的所有代码都必须清楚，哪些元素无效无效（或已删除）。这个问题会将知识分散到使用该数组的函数中，并使工作量重复。由于数组是哑对象，因此无法询问哪些元素有效。开发人员必须确定一种方案，通过某种方式将各个元素标记为无效——读者可能不清楚此代码的含义或为什么它会与某些元素发生冲突。以下代码演示了静态数组的简单情况，该数组强制 `Person` 类包含伪默认构造函数。

清单 *5.22* 静态数组强制使用默认构造函数

```
struct Person {
  std::string name;
  int age;
  Person(const std::string& n, int a) : name(n), age(a) {}
  Person() : name("", 0) {}
};

int main() {
  Person people[3]; // 1
  Person suzy("Susan", 25);
  people[0] = suzy;
  Person anna("Annette", 32);
  people[2] = anna;
  std::cout << people[1].name << '\n';

  int count = 5; // assume this is computed
  Person* others = new Person[count]; // 1
  return 0;
}
```

注释 1：每个元素可通过调用默认构造函数进行初始化

### 分析

第一个数组是静态的；元素数量必须在编译时知道，此示例表明元素数量太少。尝试了另一种方法，即在确定元素数量后分配动态数组。这两种情况下，都需要默认构造函数，但实例变量没有合理的默认值。如果五个元素中有四个有意义地初始化，则缺少的元素仍将是“合法的”Person 元素；但将包含非法信息，因其数据不代表任何人。

### 解决

用 `vector` 代替数组几乎总是正确的选择。Stroustrup 博士（C++ 的发明者）推荐这种方法，所以我对此深信不疑。即使他没有这样说过，使用 `vector` 的理由本身也足够令人信服。

首先，`vector` 是动态的，使用 `vector` 在某些方面类似于数组。数组用于实现称为支持数组的 `vector`。当支持数组填满，并且没有多余的元素时，就会发生神奇的事情。添加新元素时不

会发生崩溃，vector 将分配一个新的、更大的支持数组；从前一个数组复制元素；并将新元素添加到第一个未使用的索引。开发人员只需享受轻松使用的乐趣，并且不需要用户编写内存管理代码即可获得此结果。请不要假设 vector 不会滥用或使用而不会产生后果；错误的代码可能会导致异常！

其次，可以将元素推送到 vector 中，而无需考虑特定的索引值。如果索引有效，则可以添加元素。对于开发人员而言，删除元素就是删除该元素；无需设置哨兵来指示无效元素。

第三，如果对后备数组进行多次重新分配，则可能会出现严重的性能下降，这是由于元素被复制的次数所致。最好估算所需元素的数量，并调用 reserve 函数。reserve 函数会为该数量的元素分配足够的内存。选择正确的值意味着不会重新分配或浪费空间遇到这种情况时，即使猜测错误，vector 也会正确运行。如果猜测值太低，则会发生重新分配；如果猜测值太高，则会浪费空间。但开发人员不必管理内存或为 vector 的机制担心。

以下代码改进了数组实现，使编码更加流畅，错误更少。开发人员不会凭空得到任何东西；只要明智地使用，vector 就可以接近这个不可能的梦想。

清单 5.23 使用 *vector* 替换笨拙的数组

```
struct Person {
  std::string name;
  int age;
  Person(const std::string& n, int a) : name(n), age(a) {}
};

int main() {
  std::vector<Person> people;
  Person suzy("Susan", 25);
  people.push_back(suzy);
  Person anna("Annette", 32);
  people.push_back(anna);
  std::cout << people[people.size()-1].name << '\n';

  int count = 5; // assume this is computed
  std::vector<Person> others;
  others.reserve(count);
  return 0;
}
```

**建议**

- 大多数情况下，用 vector 代替数组。
- 仔细阅读并理解使用 vector 的空间和时间含义；许多情况下，不会出现问题，并且具有出色的性能。
- vector 有许多可用的方法；研究它们以了解它们的能力和可能性。

# 第 6 章 更好的经典 C++

本章内容

- 输入和输出
- 内存分配
- 可变长度参数列表
- 迭代器

从 C 到 C++ 的演进过程中，过去编程实践的经验继续影响着软件开发格局。我们从第 5 章中对 C 习语的考察过渡到本章，将重点关注早期 C++ 编程中根深蒂固的习惯。

尽管 C++ 带来了诸多进步，但开发人员仍难以摆脱面向 C 的构造和思维方式。这种对现有方法的倾向，延伸到了库函数的选择，有时会妨碍使用较新的 C++ 功能。即使是那些接受 C++ 最新功能的人，也不确定如何有效地使用一些新功能。

这些模式的影响体现在使用代码库的几个问题上。本章讨论了这些挑战，并针对几个陷阱和误解提供了见解和解决方案。通过检测这些问题并了解其解决方案，可以更好地驾驭 C++ 编程。

## 6.1. 错误 36: 使用 scanf 和 printf 进行输入和输出

这个错误主要关注有效性，其次是可读性。当只有 scanf 和 printf 类型的函数时，必须使用。然而，它们很难正确使用，而且阅读起来很别扭。

**问题**

许多问题需要从键盘或文件输入文本数据，并将其转换为各种数据类型。scanf 和 printf 函数使用格式说明符来确定数据类型。此说明符通常是一个字符，以符号形式表示数据类型。如果使用了错误的说明符，则会导致未定义行为。某些情况下，错误可以容忍；其他情况下，会发生崩溃，甚至更糟。如果错误的说明符不会导致崩溃，则数据很可能是错误的。

只有当开发人员了解格式说明符的可能性时，才有可能读取格式说明符。清单 6.1 对 sscanf(字符串扫描）函数使用了复杂的格式说明符。有时，该说明符比正则表达式更难读，读者必须停下来，在脑海中解析各个部分。如果说明符不为人知，开发人员必须通过搜索互联网，或其他资源来了解其含义和正确用法。编程的许多方面都是如此，而不仅仅是格式说明符。关键是让我们停下阅读，并进行一些研究的事情，都是改进的机会。格式说明符就是这样的机会。

清单 6.1 使用复杂的格式说明符确定转换

```
int main() {
  const char* str = "3.14159 042 boxes .3";
  double pi;
  int cats;
  int mice;
  char buffer[5];

  int count = sscanf(str, "%lf%*c%i%s%d", &pi, &cats,
    buffer, &mice); // 1
```

```
  if (count != 4) // 2
    std::cout << "error reading value " << count+1 << '\n';

  printf("%f being eaten by %d cats in %s along with %d mice\n",
    pi, cats, mice, mice); // 3
  return 0;
}
```

注释 1：所有输入规范都在一次操作中应用；这……正确吗？

注释 2：开发者必须确定正确的数字

注释 3：文本、规范和变量混合在一起很别扭——其中有一个错误

**分析**

格式说明符有一些奇怪的地方。double 变量上可以使用 d 字符进行扫描和转换，但这样做会导致错误。此处的 d 字符指的是 decimal，而不是 double。f 字符表示浮点值，直观上是 float 数据类型。要将文本读入 double 变量，需要使用 long 浮点说明符 lf。我们当然希望看到更直观的东西。

另一个奇怪的是，第一个整数变量使用 i 说明符（表示整数？），而第二个整数变量使用 d 说明符。第一个整数变量的输出结果是 34，而不是 42。

只要理解了 d 表示以 10 为基数的整数，谜团就解开了。相比之下，i 表示一个整数，其基数由数据的前导输入字符决定（0 表示八进制，0 x 表示十六进制，否则为十进制），前导 0 表示文本是八进制值，这些细节很容易被忘记。

最后，buffer 变量表示 C 样式字符串并使用 s 说明符—一个具有直观含义的说明符了！开发人员知道容器是五个字符，因此为数组分配了该数量的元素。很好，但 scanf 类型函数将相关字符传输到目标中并添加终止空字符。如果使用 c 说明符，则不会添加该空终止符。这些差异必须记住，或每次都查找都进行。

printf 输出也存在一些问题。某些情况下，其说明符与 scanf 的说明符不同。此示例中，double 变量使用 f 说明符，但没有使用 lf，就像在 scanf 中一样。可以理解将输出文本与说明符混合在一个字符串中，但它确实引入了一些阅读的复杂性，因为变量是在格式字符串之后列出的。从左到右阅读，还需要在字符串和变量之间来回跳转（并且必须严格保持它们的顺序）。

最后，如果说明符和变量类型之间出现变量不匹配，会发生什么情况？清单 6.1 中的示例两次错误指定了 mice；第一次应该是 buffer。当使用整数变量作为 C 风格字符串？肯定不会有什么好事发生，所以未定义行为才是正解。我的系统发出了段错误，提醒我有些东西坏了，但不会继续执行。

这些函数系列不是类型安全的，使用起来存在风险。很容易出现预期格式说明符，与实际数据类型不匹配的情况。鉴于这些有限的错误检测能力，有意义的恢复代码非常复杂。

**解决**

C 语言为提供了 scanf 和 printf 功能，但我们使用 C++ 进行编程，它有更好的选项。将字符数据移入或移出程序时，应使用流。流提供了非常有用的插入和提取操作符（分别为 << 和 >>）。这些操作符确定如何将文本转换为正确的数据类型（假设输入数据与数据类型一致），并使开发人员不再需要数据类型说明符。

清单 6.2 中的错误编码（其中 mice 输出两次）不会导致运行时，错误或歪曲数据——输出不正确，并且没有像清单 6.1 中那样从一种类型到另一种类型的错误转换。如果数据一致，则将文本数据提取到各种变量中，可以正常工作；但当数据不一致时会失败。scanf 的问题在于输入和转换是一次性发生的；如果发生错误，开发人员必须从返回值中确定哪个转换失败了。清单 6.1 中的检测逻辑更加比清单 6.2 更简单，但至少第二个版本为开发人员提供了对错误检测的更细粒度的控制。

清单 *6.2* 使用类型确定转换

```
int main() {
  std::istringstream str("3.14159 042 boxes .3");
  double pi;
  int cats;
  int mice;
  std::string buffer;

  str >> pi; // 1
  if (str.fail())
    std::cout << "error reading value 1\n";
  str >> cats;
  if (str.fail())
    std::cout << "error reading value 2\n";
  str >> buffer;
  if (str.fail())
    std::cout << "error reading value 3\n";
  str >> mice;
  if (str.fail())
    std::cout << "error reading value 4\n";

  std::cout << pi << " being eaten by " << cats << " cats in " << mice
    << " along with " << mice << " mice\n"; // 2
  return 0;
}
```

注释 1：输入和转换工作正常或检查 fail() 函数

注释 2：转换基于实际数据

整数值 042 正确转换为值 42；前导 0 不会影响其含义。如果数据确实是八进制，则可以将输入流设置为 8 进制以进行输入操作。此代码将输入并将 cats 转换为八进制：

```
str >> std::setbase(8) >> cats;
```

**建议**

- 尽可能用更适合流式传输的提取和插入操作符，替换 scanf 和 printf 调用。
- 避免使用复杂的输入和输出格式字符串；它们即难用，又难阅读。

## 6.2. 错误 37: 过度使用 endl

这个错误主要与性能有关。操作系统通常会缓冲输入和输出数据,以尽量减少输入/输出（I/O）操作。误解缓冲可能会对性能产生重大影响。

### 问题

下面的代码很简单，循环迭代一千次，每次输出循环控制变量的值。最后输出终止消息——很简单，对吧?

清单 *6.3* 不加区分地使用 *std::endl*

```
int main() {
  for (int i = 0; i < 1000; ++i)
    std::cout << i << std::endl; // 1
  std::cout << "finished!" << std::endl;
  return 0;
}
```

注释 1: 输出到 std::cout 缓冲区并刷新

### 分析

代码按预期工作，只是性能可能比预期的更差。简单的问题是 std::endl 输出行尾字符，并刷新输出缓冲区。

更少的 I/O 操作等同于更高的吞吐量，std::cout 流是使用更少 I/O 操作的缓冲流。其缓冲区会累积输出值和换行符，直到填满为止，然后将缓冲区作为一个大块刷新。使用 std::endl 会抵消这种缓冲优化，无论缓冲区包含多少数据，都会刷新缓冲区。各种优化可以抵消终端输出的这种明显差异，但不能抵消文件输出的这种明显差异，因此请考虑对此分析进行更细致的理解。

### 解决

如果将 std::endl 更改为 '\n'，缓冲将按预期工作。某些情况下，应刷新部分填充的缓冲区以完成 I/O 操作。清单 6.4 中的代码优化了缓冲区填充，直到输出所有值和换行符。然后，终止消息刷新缓冲区，以确保任何缓冲数据都写出，从而干净地完成输出操作。

清单 *6.4* 谨慎地使用 *std::endl*

```
int main() {
  for (int i = 0; i < 1000; ++i)
    std::cout << i << '\n'; // 1
  std::cout << "finished!" << std::endl; // 2
  return 0;
}
```

注释 1: 输出到 std::cout 缓冲区

注释 2: 输出并将缓冲区刷新

所有情况下都使用缓冲输出，它可以与操作系统配合使用以提供最佳性能。如果要在不使用缓冲的情况下输出数据，请考虑使用 std::cerr 流。

**建议**

- 尽可能将 `std::endl` 更改为 `\n`，以防止不必要的缓冲区刷新。
- 当需要刷新输出缓冲区时，请使用 `std::endl`。
- 如果输出大量数据，则对除最后一个操作之外的所有操作使用 `\n`；对最后一个操作使用 `std::endl`。
- 考虑使用 `std::cerr` 进行无缓冲输出。

## 6.3. 错误 38：使用 malloc 和 free 进行动态分配

这个问题主要集中在正确性上，其次是有效性。内存分配自 C 语言以来已经取得了长足的进步，但这并不是每个人都能接受这些变化。

**问题**

C 提供 `malloc` 和 `free` 内存分配和释放操作符对。与使用 `new` 和 `delete` 的 C++ 动态资源分配一样，使用 `malloc` 获得的任何内容都应使用 `free` 释放。这听起来很简单，很多情况下也确实如此。但 C 和 C++ 代码可能会使这个“简单”的过程变得复杂，尤其是当资源所有权发生变化时。

清单 6.5 展示了 `malloc` 代码中出现的两个常见问题：

- 大小计算不正确
- 无法初始化获取的内存

清单 *6.5* 不规范地使用 *malloc* 和 *free*

```
struct Buffer {
  char* str;
  Buffer(int size) : str(new char[size+1]) { str[0] = '\0'; }
  ~Buffer() { free(str); }
};

int main() {
  double* val = (double*)malloc(sizeof(int)); // 1
  std::cout << val << '\n';
  Buffer* buf = (Buffer*)malloc(sizeof(Buffer)); // 2
  std::cout << buf->str << ", size " << strlen(buf->str) << '\n';
  free(buf);
  return 0;
}
```

注释 1：睡眼惺忪的错别字

注释 2：动态对象空间已分配，但未初始化

**分析**

除了前面提到的两个 `malloc` 问题之外，上述代码至少还有三个其他错误。首先，获取了一个动态内存块来保存 `double` 值。由于熬夜和咖啡因水平下降，开发人员错误地请求了足够的内

存来存储整数（通常是 double 值的一半大小）。编译器对这种不匹配保持沉默，这是一个典型的类型安全错误。对存储在这个太短的块中的数据的访问，都将访问其边界之外的数据。

第二个问题出现在创建 Buffer 对象时。这次，分配的大小是正确的，但缺少 char* 变量的初始化。构造函数应该处理获取动态内存块，以保存字符数据，并确保将终止字符写入第一个位置。然而，这种情况并未发生过。调用 malloc 时，不会执行构造函数，也不会初始化。

第三，val 变量没有相应的释放。内存的动态分配在主函数退出时泄漏；没有析构函数或其他管理实体在监视内存。在长期运行的程序中，几次这样的泄漏可能会导致严重的问题。

第四个问题是，虽然 buf 对象已正确释放，但该对象获取的动态内存并未释放。诚然，从未分配，但更好的代码会进行分配，只会在封闭实体释放时泄漏。free 调用从不调用析构函数，导致动态资源都处于困境之中。

最后，通过 new 分配内存并调用构造函数获得的 Buffer 对象的动态内存，与通过调用 free 释放不匹配，这种不匹配的行为没有定义。使用复制和赋值（和移动）语义将缓解其中许多问题。

**解决**

new 和 delete 操作符并非只是为了互逆而开发，malloc 和 free 的不足之处足以说明需要一种新的方法。

如清单 6.6 所示，第一个问题由 new 处理，以确保获得的对象可以接收变量的正确类型——不会出现类型安全错误。第二个问题通过确保每次分配时，都调用构造函数来处理。正确设计的代码将实现 RAII 模式，以确保分配与正确的释放配对。第三，仍然有可能无法删除 new 对象；开发人员必须确保其配对。如果设计正确（RAII 模式），则在调用析构函数时会解决第四个问题。第五，也是最后，使用 new 和 delete 操作符时就能很好的匹配。混合使用 new/ delete 与 malloc/free 不仅是不好的形式，而且还会产生严重后果：更多未定义的行为。

清单 *6.6* 以规范的方式使用 *new* 和 *free*

```
struct Buffer {
  char* str;
  Buffer(int size) : str(new char[size+1]) { str[0] = '\0'; }
  ~Buffer() { delete[] str; }
};
int main() {
  // double* val = new int; // 1
  double* val = new double; // 2
  std::cout << val << '\n';
  Buffer* buf = new Buffer(25); // 3
  std::cout << buf->str << ", size " << strlen(buf->str) << '\n';
  delete buf; // 4
  return 0;
}
```

注释 1：不再可能，编译器会报错
注释 2：类型安全得到保证
注释 3：调用构造函数
注释 4：调用析构函数；记得删除它！

Buffer 类可以删除或隐藏复制构造函数和复制赋值操作符，以使其实现更加健壮。这一点并未得到演示，但在使用动态资源时应始终考虑。

**建议**

- 将 malloc 调用替换为 new，将 free 调用替换为 delete。
- 每个 new 都有对应的 delete，每个 new[] 都有匹配的 delete[]。

## 6.4. 错误 39： 使用联合（union）进行类型转换

这个问题主要集中在正确性和可读性上。编译器理解 C++ 类型，开发者试图转换它们，可能会让类型检查机制措手不及。

**问题**

编译器中的类型检查系统，旨在避免将一种类型与另一种类型错误匹配。虽然人类可能看不出某些类型之间的区别，但编译器更了解情况，并会阻止许多转换，除非开发者坚持要转换，并告诉编译器退出。这种方法非常有效 - 编译器会闭嘴 - 但可能会导致难以发现，且调试起来很麻烦的错误。

清单 6.7 中的代码在某些系统上运行良好，使用 union 来描述一个包含四个字符和一个整数的数组。开发者打算将整数转换为字符并将其输出到二进制文件。联合 mixer 简化了小字符数组和整数之间的转换。整数写入联合实例；其值在 write 函数调用中读取。在我的系统上，这运行良好。

清单 *6.7* 使用联合进行类型转换

```
static const int bytes_per_int = 4; // 1
  union mixer {
  char ch[bytes_per_int]; // 2
  int n;
};
int main() {
  mixer converter;
  converter.n = 42;
  std::ofstream out("data.txt", std::ios::binary);
  out.write(converter.ch, bytes_per_int);
  return 0;
}
```

注释 1: 整数总是四个字节

注释 2: 这里只进行了假设

**分析**

union 为字符数组保留四个字节，并用整数覆盖。这在具有四字节整数的系统上有效，但在任何地方都有效吗？当编译器版本更改时，是否有效？也许；答案是 yes 和 yes，但不要指望在别人的系统上也能工作。不能保证整数占用四个字节。当然，开发者可以检查这个值并对其进行硬

编码，但这种解决方案是不可移植的。union 不能有效地处理动态内存，自动将字符数组调整为整数大小。

**解决**

非提升的类型转换的根本问题是开发者对大小、位和字节的底层布局做出了假设。这些假设说明代码僵化且不可移植。由于这些假设，开发者引发的转换是有问题的，但如果必须编码，则应让编译器参与进来提供帮助。

清单 6.8 中显示的 reinterpret_cast 带有编译器提供的整数大小，是一种更好的方法。没有开发者必须确定和验证的硬编码大小；编译器在编译时，会根据特定架构计算出其大小。

二进制流有意义的少数几个地方之一，是用户定义类型转换同样有意义的地方。转换为字符（实际上是字节）的数据永远不会以转换后的形式使用。使用 reinterpret_cast 转换的指针，应该完全反转为原始类型，以避免出现异常和错误。

清单 *6.8* 合理地使用 *reinterpret cast*

```
int main() {
  int n = 42;
  std::ofstream out("data.txt", std::ios::binary);
  out.write(reinterpret_cast<const char*>(&n),
    sizeof(n)); // 1
  return 0;
}
```

注释 1：编译器确定正确的大小；不做任何假设

尽管如此，以不需要用户定义类型转换的方式设计程序仍然是不错的选择。信任底层数据类型，并使用类来聚合不同的数据类型。将实现细节推送给编译器，并尽可能在更高的级别上工作。

另外两个可能需要用户定义类型转换的地方是，API 的接口边界和与其他语言的模块集成。这些情况下，尽可能使用编译器来协助转换工作。此外，将转换代码隔离为执行转换的文档齐全、小型、单一用途的函数。每次编译器、系统级别和架构发生变化时，都要检查这些函数。

**建议**

- 尽可能避免用户定义的、依赖于实现的类型转换。
- 需要时，尽可能避免硬编码假设；使用编译器来帮助。
- 记录每个出现对话代码的地方；明确每个假设，以便开发者可以在变化发生时，根据需要对其进行修改。

## 6.5. 错误 40： 使用 varargs 实现可变参数列表

这个错误主要针对正确性，影响有效性和可读性。当开发人员不知道函数需要多少个参数时，可以使用可变参数列表。

**问题**

无需编写多个采用越来越多参数的重载函数，只需使用一个使用可变参数（varargs）的函数即可处理任意数量的参数。通常，该函数将具有一个或多个命名参数，其中一个是参数列表的长

度。最后一个参数是省略号（…），其向编译器表明后面有未知数量的参数。从技术上讲，更正确的是后面有未知数量的值，但无需在命名上吹毛求疵。

看看清单 6.9 中的代码。开发人员希望有一个函数可以对值列表进行求和。为了灵活性，添加了初始起始值，以防包含前一个求和的结果（通常为零）。需要传递值的数量，函数必须知道读取值列表的次数；编译器不会自动确定这一点。这会出什么问题呢?

该函数首先调用 va_list 来建立变量参数列表，通过调用 va_start 来初始化流程；然后，使用 va_arg 迭代每个参数；最后，通过调用 va_end 来清理混乱局面。

此代码演示了一个错误，该错误已在生产代码中发现。va_start 宏需要知道两个事实；第一是参数列表的位置，第二是变量参数之前的最后一个参数。在这种情况下，应该使用 initial，而不是 len。虽然不推荐，但这种奇怪的方法似乎有效（为什么会这样，留给好奇的读者吧）。

清单 *6.9* 为可变长度的值列表使用变量

```
int sum(int len, int initial, ...) {
  int sum = initial;
  va_list args;
  va_start(args, len); // 1
  for (int i = 0; i < len; ++i) // 2
    sum += va_arg(args, int);
  va_end(args);
  return sum;
}

int main() {
  std::cout << sum(9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9) << '\n';
  std::cout << sum(3, 5, 1, 2, 3) << '\n';
  return 0;
}
```

注释 1：哎呀，有个 bug！不过这已经很接近正确了

注释 2：访问每个传递的值

**分析**

某些系统上，两次调用的预期总和可以正确生成，但其他系统可能会遇到问题。如果其中一个变量参数值是 double 或 Person 实例，会发生什么情况。编译器不会对参数进行类型检查，这些参数不会导致错误或警告。实际上，使用基于 C 的 varargs 会关闭验证，将结果留给开发人员，而不匹配的类型会导致未定义行为。更糟糕的是，sum 函数可能会返回一个不会泄露其不愿崩溃的值。

注意：系统按“预期”运行，并等于代码正确或按预期运行，只是没有发生明显的故障。提供此类示例的好处是，错误的编程可能看似有效，但却掩盖了潜在的致命问题。不要因为“一切似乎都没有问题”而忽略警告、错误的技术等。演示代码有误，这是当整个系统中的某些东西（任何东西）发生变化（编译器、平台、太阳耀斑等）时可能突然失败的情况之一。买者慎之！

**解决**

简而言之，使用 varargs 是个坏主意。这个结论并不意味着它们总是无法工作，但确实意

味着开发人员无法避免错误。C 几乎没有其他方法来实现此功能，但出现类型相关问题的可能性较小。这种情况下，C++ 很容易搞砸事情。一种可能的解决方案是传递一个值数组，并完全删除 varargs。删除 va_* 宏后，sum 的基本功能仍将保留，但使用 vector 存在更好的解决方案。这两种解决方案都具有明显的优势，可以避免添加元素类型以外的数据类型，所以编译器可以实现类型安全。考虑清单 6.10 中的代码，其中使用了 vector，并且不需要传递长度。

以下代码使用模板来增加元素类型的灵活性。该解决方案增加了使用未知数量的值及类型的灵活性——一种可爱的小方法。

清单 *6.10* 变长列表中使用 *vector*

```
template <typename T>
T sum(T initial, const std::vector<T>& vals) { // 1
  T sum = initial;
  for (int i = 0; i < vals.size(); ++i)
    sum += vals[i];
  return sum;
}

int main() {
  std::vector<int> intvalues; // 2
  for (int i = 1; i < 10; ++i)
    intvalues.push_back(i);
  std::cout << sum(0, intvalues) << '\n';
  std::vector<double> doublevalues;
  for (int i = 1; i < 4; ++i)
    doublevalues.push_back(i);
  std::cout << sum(5.0, doublevalues) << '\n';
  return 0;
}
```

注释 1：选择模板化的参数列表，以获得更大的灵活性

注释 2：选择具有更大灵活性的容器

**建议**

- 尽可能避免使用 varargs；删除用更好的解决方案进行替换。
- 使用数组或更好的 vector，来获得类型安全的（相同）结果。

## 6.6. 错误 41： 类的初始化顺序

这个错误主要针对性能和效率，初始化列表可增强初始化实例变量的性能。

**问题**

清单 6.11 中的代码，显示了尝试从提供的参数构建实例。完整“标题”是 title、first、middle 和 last 名称的组合。在构造函数中用这些名称初始化非常合理，所有其他变量都从提供的值初始化。这种情况下，可以在访问器中构建标题。尽管如此，为了挽救轻微的性能差异（假

设从未访问过标题），所做的努力是微不足道，并迫使开发人员编写更多代码。这种方法影响了可读性，却没有提供足够的好处。

当代码在我的系统上运行的时候，会抛出一个 `std::bad_alloc` 异常，开发人员会修复这个错误。这个结果是这个问题的一个可能的影响，系统可能会有不同的行为。无论如何，代码存在导致未定义行为的问题。如果碰巧这个代码在给定的系统上运行，则会产生奇怪的输出，可能很难调试。

清单 *6.11* 混淆初始化顺序

```
struct Person {
  std::string full;
  std::string first;
  std::string middle;
  std::string last;
  std::string title;
  Person(std::string f, std::string m, std::string l, std::string t) :
    first(f), middle(m), last(l), title(t),
    full(title + ' ' + first + ' ' + ' ' + middle + ' ' + last ) {} // 1
};

int main() {
  Person judge("Hank", "M.", "Hye", "Hon.");
  std::cout << judge.full << '\n';
  return 0;
}
```

注释 1：构建完整的标题

**分析**

这段代码大部分都没问题，但请注意实例变量的声明顺序和初始化顺序。首先声明了 `full` 实例变量，然后是其余变量。因为各个实例变量（似乎）已经初始化了，所以开发者对此进行了推理，并决定最后初始化这个实例变量。

开发人员在这里犯了一个错误；尽管完整的实例变量似乎最后才初始化，但编译器会编写代码按声明顺序初始化变量。因此，`full` 会先初始化，然后才是其他变量。这些未初始化变量的连接，会导致异常或其他不良行为。

理解了这个强制顺序之后，开发人员可以通过将代码放在构造函数主体中来控制初始化。清单 6.12 显示了结果。

清单 *6.12* 将初始化推入构造函数体

```
struct Person {
  std::string full;
  std::string first;
  std::string middle;
  std::string last;
  std::string title;
```

```
  Person(std::string f, std::string m, std::string l, std::string t) { // 1
    first = f; middle = m; last = l; title = t;
    full = title + ' ' + first + ' ' + ' ' + middle + ' ' + last;
  }
};

int main() {
  Person judge("Hank", "M.", "Hye", "Hon.");
  std::cout << judge.full << '\n';
  return 0;
}
```

注释 1：将所有初始化都放在构造函数主体中（性能较差）

此解决方案有效，但需要提高性能。这可能并不明显，但所有变量仍然使用初始化列表形式进行初始化

```
Person(std::string f, std::string m, std::string l, std::string t) :
  full(), first(), middle(), last(), title() {}
```

编译器确保所有类实例都使用其默认构造函数进行初始化，所有实例变量都初始化为初始化列表中的默认值。初始化之后，执行构造函数主体，并将参数值分配给现有实例。这是每次操作成本的两倍（暂时不要挑剔复制构造函数和复制赋值操作符之间的性能差异）。不过，这种方法并非最优。

**解决**

由于编译器将使用初始化列表形式，来初始化此类声明中的每个基于类的实例变量（而非基元变量）。如果顺序很重要，理想的方法是按照变量应初始化的顺序对其进行排序。此示例对顺序很敏感，但应尽可能避免这种情况。如果顺序不敏感，请选择有意义的顺序（按字母顺序、按类型分组等）。以下代码考虑了顺序，并确保只有在其所有组成部分都完全初始化后，才初始化 `full`。

清单 *6.13* 严格使用初始化顺序

```
struct Person {
  std::string first;
  std::string middle;
  std::string last;
  std::string title;
  std::string full;
  Person(std::string f, std::string m, std::string l,
         std::string t) : // 1
    first(f), middle(m), last(l), title(t),

  full(title + ' ' + first + ' ' + ' ' + middle + ' ' + last ) {}
};

int main() {
  Person judge("Hank", "M.", "Hye", "Hon.");
```

```
  std::cout << judge.full << '\n';
  return 0;
}
```

注释 1：重新排序变量，以确保首先初始化组件部分

**建议**

- 如果可能的话，避免使用复合实例变量；如果不能，请对它们进行排序，以便首先初始化所有组成部分。
- 对所有实例变量使用初始化列表；如果不使用它们，编译器会生成它们。
- 确保编译器警告已打开；这是最早、最广泛检测到的问题方法（在编译时始终使用 `-Wall`）。

## 6.7. 错误 42： 将非值类型添加到容器

这个问题影响正确性。大多数情况下，容器是数组的首选替代方案，但这一事实并不能消除使用容器的问题。

**问题**

通常，编程问题涉及通常处理或处理的数据组。数组是将单个数据元素作为一个单元，保存和管理的经典解决方案，但很少使用。标准模板库提供一些容器比数组更加智能，并且提供了值得使用的功能。

假设开发人员想要打印某个活动的人员列表。其中会混合使用 `Persons` 和 `Employees`，但必须全部列出。清单 6.14 中的代码展示了这一尝试。由于其所有权语义，开发人员被鼓励在编程期间使用自动指针（`auto_ptr`）。大胆前进，代码开发和测试，其结果可能会更令人欣喜。

清单 *6.14* 滥用多态元素的容器

```
struct Person {
  std::string name;
  Person(const std::string& n) : name(n) {}
  virtual std::string toString() { return name; }
};

struct Employee : public Person {
  double salary;
  Employee(const std::string& n, double s) : Person(n), salary(s) {}
  std::string toString() {
    std::stringstream ss;
    ss << Person::name << " gets paid " << salary;
    return ss.str();
  }
};

int main() {
  Person p("Sue");
```

```
  Employee e("Jane", 123.45);

  std::vector<Person> people;
  people.push_back(p);
  people.push_back(e);
  for (int i = 0; i < people.size(); ++i)
    std::cout << people[i].toString() << '\n'; // 1

  std::vector<std::auto_ptr<Person> > persons;
  // persons.push_back(p); persons.push_back(e); // 2

  std::vector<Person> peeps; // 3

  peeps.push_back(p);
  for (int i = 0; i < peeps.size(); ++i)
    std::cout << peeps[i].toString() << '\n';

  std::vector<Employee> emps; // 3
  emps.push_back(e);
  for (int i = 0; i < emps.size(); ++i)
    std::cout << emps[i].toString() << '\n';
  return 0;
}
```

注释 1：多态行为不起作用

注释 2：无法编译；具有不同的元素类型

注释 3：单独的 vector 解决了该问题

**分析**

尝试使用 std::vector<std::auto_ptr<Person>> 无法通过编译。开发人员对此代码进行了评论，希望以后对其进行研究。由于开发人员迫切希望代码能够正常工作，因此编写了单独的容器来保存每种数据类型。这种方法有效，但需要优化。

单个 vector 的问题在于保存的是基类型的对象。每个元素只分配了足够容纳基类对象的空间，从而切断了派生类数据。使用数组解决同一问题时也会出现此问题。多态行为就是这样！

第二次尝试是使用新奇的 auto_ptr 类型，独占容器内的实例。这种方法很有意义，但会遇到麻烦。只有可复制构造和可分配的数据类型才能插入容器中。auto_ptr 类型两者都不是，它支持独占所有权语义，所以无法将实例添加到 vector 中——新奇的想法就这么多。（现代 C++ 有一个更比使用 auto_ptr 更好的方法；出于充分的理由，此类型在 C++11 中已弃用）。开发人员的最后一次尝试解决了该问题；但若没有单个容器的灵活性，解决方案可能会更好。

**解决**

容器应包含值类型，可以是基本元单元、不与派生类实例混合的类型或指针（在本例中为原始指针）。现代 C++ 已经解决了原始指针问题，因此此建议仅应在前现代 C++ 中使用。选择原始指针作为容器的元素类型，是为了保持开发人员想要的灵活性、消除切片问题，并允许复制构造和赋值。以下代码展示了解决方案，并展示了开发人员的所有首选行为。

清单 *6.15* 正确使用容器——多态元素

```
struct Person {
  std::string name;
  Person(const std::string& n) : name(n) {}
  virtual std::string toString() const { return name; }
  virtual ~Person() {}
};

struct Employee : public Person {
  double salary;
  Employee(const std::string& n, double s) : Person(n), salary(s) {}
  std::string toString() const {
    std::stringstream ss;
    ss << Person::name << " gets paid " << salary;
    return ss.str();
  }
};
int main() {
  Person p("Sue");
  Employee e("Jane", 123.45);
  std::vector<Person*> people;
  people.push_back(&p); // 1
  people.push_back(&e); // 1
  for (int i = 0; i < people.size(); ++i)
    std::cout << people[i]->toString() << '\n'; // 2
  return 0;
}
```

注释 1：每一个都可以添加；相同的元素类型

注释 2：多态行为有效

容器在使用多态元素时很有用，但必须正确管理。确保基类添加虚析构函数，容器在构造和析构期间会复制和删除元素。代码安全可确保运行时稳定性！

**建议**

- 确保添加到容器中的内容都是可复制构造和可分配的；消除其中一个或两个的类不符合纳入条件。
- 容器应保存值类型或指针类型，以避免切片或其他问题。
- 如果可能，请考虑使用现代 C++。

## 6.8. 错误 43： 首选索引

这个错误注重效率，影响性能。许多开发者都熟悉使用索引来迭代数组，并将这种方法应用于容器。

### 问题

开发使用数组的代码，会将开发人员的思维集中到一种迭代方法中。循环中使用索引很常见，并且适用于许多问题；但使用索引可能会不必要地限制一个人的思维，并在使用不同的容器时导致问题。

数组很少是容器的最佳选择。标准模板库提供了几个更专业的容器，比数组更实用。缺少这些容器会人为地限制开发人员，并导致他们不必要地设计和重写代码。优先使用索引，而非迭代器会影响性能。循环体通常足够大，以至于这种影响可以忽略不计——但不一定。

清单 6.16 展示了索引以迭代几个容器。仅包括几个有用的容器；其他容器的工作方式类似（例如，deque、list 和 map）。这种缺失不是懒惰；其中一些缺少的容器无法索引，这使得处理它们与从 C 或课堂上学到的方法截然不同。将数组传递给函数的不便之处在于，需要将元素数量作为附加参数传递。很容易弄错这个值，导致数据丢失或访问超出数组末尾的内容，并且调用者需要负责处理函数应该处理的细节。总的来说，数组通常不是容器的良好选择。

清单 *6.16* 对函数中的数组进行索引

```
double sum(const double* values, int size) {
  double sum = 0.0;
  for (int i = 0; i < size; ++i)
    sum += values[i];
  return sum;
}

int main() {
  double vals[] = { 3.14, 2.78, 3.45, 7.77 };
  std::cout << sum(vals, 3) << '\n'; // 1
  return 0;
}
```

注释 1：糟糕！使用了最后一个元素的索引值，而非数组长度

### 分析

C 语言编程没有索引的替代方案，会导致很多错误。虽然错误不是索引固有的特性，但在 C 语言中，出错的风险巨大。主要原因是开发人员必须跟踪值列表和大小，并将它们传递给函数或在循环中使用它们。

关于数组的使用及其局限性的话题已经有很多文献记载，许多教科书都侧重于该技术。一旦深入记忆，使用迭代器似乎尴尬和“不同”。我们首先学到的东西通常会视为正确、规则和有效的。索引具有这种效果，在大多数情况下是一个糟糕的选择。

### 解决

在大多数情况下，使用迭代器是理想的选择。首先，所有标准模板库容器都使用它们。其次，它们经过了优化，通常比索引更高效。第三，不会出现索引与长度差 1 的错误。清单 6.17 演示了使用各种容器的迭代器。它们提供的一致性很重要；如果一个人学会了在一种情况下如何迭代，那么其他情况下也会类似。这种方法应该足够熟悉，并形成肌肉记忆。

与数组不同，STL 容器非常智能；它们知道自己的大小，并可以与迭代器交互，从第一个元素

开始迭代到末尾。该方法不需要根据容器进行更改。

清单 *6.17* 对 *vector* 对象使用迭代器

```
double sum(const std::vector<double>& values) {
  double sum = 0.0;
  for (std::vector<double>::const_iterator it = values.begin();
      it != values.end(); ++it)
    sum += *it;
  return sum;
}

int main() {
  std::vector<double> vals;
  vals.push_back(3.14); vals.push_back(2.78);
  vals.push_back(3.45); vals.push_back(7.77);
  std::cout << sum(vals) << '\n';
  return 0;
}
```

有人可能会抱怨循环中的迭代器类型很复杂。这是一个合理的反对意见；现代 C++ 提供了使用 auto 关键字进行类型推导的能力，这通过让编译器推导正确的类型来消除这种复杂性。这确实节省了时间！

为了证明其他容器的工作方式相同，请使用 set 的清单 6.18。唯一的变化是容器的类型，集合而不是 vector，以及向集合添加元素的变化，即 insert 而不是 push_back。添加了第三个集合 insert，以展示 set 消除了重复值。

清单 *6.18* 使用带有 *set* 的迭代器

```
double sum(const std::set<double>& values) {
  double sum = 0.0;
  for (std::set<double>::const_iterator it = values.begin();
      it != values.end(); ++it)
    sum += *it;
  return sum;
}

int main() {
  std::set<double> vals;
  vals.insert(3.14); vals.insert(2.78);
  vals.insert(3.45); vals.insert(7.77);
  vals.insert(3.45); vals.insert(7.77); // 1
  std::cout << sum(vals) << '\n';
  return 0;
}
```

注释 1：集合中将忽略重复的值

随着时间的推移，这项技术变得更加简单，并通过标准化迭代容器的方法使开发人员受益。希

望这足以激励人们停止使用数组！

现代 C++ 引入了三种功能来简化或替代编码循环。

**建议**

- 尽可能使用标准模板库容器。
- 在容器上使用迭代器来消除差 1 错误；让数据类型决定起点和终点。
- 一般情况下，不要使用数组。

# 第三部分 前现代 C++

软件开发中，处理历史遗留代码是一项挑战，特别是对于 C++ 编程而言。C++ 的经典或前现代的特点是在现代语言功能建立之前开发的实践，通常缺乏当代功能提供的安全性、效率和简单性。本节深入探讨这些经典的编码问题，提供在遗留系统范围内提高代码质量和稳健性的见解和策略。

一个关键的关注领域是类不变量的设计和维护，这是创建稳健且稳定的应用程序的基本原则。建立强大的类不变量对于确保应用程序的稳定性和可预测性至关重要，但随着时间的推移，尤其是随着代码库的发展，维护这些不变量会带来挑战。本节探讨过去的陷阱，并提供通过精心设计来制定和维持类完整性的指导，帮助开发人员避免设计质量的逐渐下降和新代码的改进。

此外，本部分还讨论了有关类操作、资源管理和函数使用的问题。精心实现类操作（例如：构造函数、析构函数和赋值操作符）对于减少错误和优化性能至关重要。正确处理异常和系统资源管理，对于避免资源泄漏和确保应用程序可靠性至关重要。有必要改进函数和参数的设计，以避免前现代 C++ 实践中的低效。最后，对经典 C++ 中的一般编码实践的全面检查为开发人员提供了路线图，以提高其遗留代码的质量和可持续性，确保其保持扩展性以应对未来的需求。

# 第 7 章 创建类的不变量

本章内容

- 类的不变量，一个关于类是什么和用来做什么的基本观点
- 从基本函数的最小集合构建类行为
- 将类视为新的数据类型
- 良好的继承设计，从类不变式可以看出
- 确保类正确工作并与编译器配合

类代表数据类型。C++ 标准化委员会永远无法预测所需的每种数据类型，所以该语言设计为可扩展的。如果需要新的数据类型，开发人员可以通过创建新的类来编写。

虽然编写类是一个简单的过程，但编写一个好的类却更具挑战性。多年来，人们给出了不同的建议，并发现了许多问题。了解其中一些问题，有助于开发者在现有代码中发现这些和类似的问题。彻底消除这些问题可以指导开发人员更好地设计类。

## 7.1. 类不变量确保类的正确设计

一个类至少有四个基本特征。一些学者可能会提出其他建议，但应该将这四个视为作为基本：

- 表示方式
- 操作集合
- 取值范围
- 内存机制

### 7.1.1. 表示方式

类不变量的主要特征是对类数据的表示。类是数据和函数的集合。数据是状态，函数是类实例的行为。在此上下文中，实例用于表示，从程序的角度对类进行创建的对象，而对象则指存储在内存中的值。各种组件数据类型的含义（单独（每个数据）和聚合（对象））取决于类的设计。每个值都有特定的类型，具有特定的含义，每个值都有助于对实例的整体理解。

类设计者必须确定数据的属性，除了类型之外还表达了什么呢？例如，在 Student 类中，std::string 可以保存学生的姓名。数据类型为 std::string，具有其所有内置含义，但在 Student 对象的上下文中，特定值表示特定于给定学生的内容，并且可能具有特定形式。假设格式如下：姓氏、逗号、空格、名字。std::string 实例无法知道或强制执行这样的方案，所以由类来维护。这种方法很糟糕；最好有多个表示每条数据的字符串。这个例子表明，好的类设计也可以很复杂。

std::string 的表示是学生的姓氏、一些分隔标点符号和学生的名字之一。总体而言，其他数据字段将增加 Student 实例的含义。该实例的结果足以代表某个学术机构中的学生。解决问题所需的所有数据，都有望在课堂上建模并在实例中进行表示。

## 7.1.2. 操作集合

操作实例数据应该仅限于类本身。如果外部代码（用户）代码可以直接使用实例数据，则很快就会出现错误或漏洞。错误是不可避免的，所以需要控制对数据的访问和修改。必须控制对值的含义（表示）、合法值是什么（值的范围）以及如何操作值的了解，以避免出现错误。

公共接口是用户可以使用的一组操作（构造函数和函数），仅将部分类成员设为公共可限制对实例的访问，并确保访问正确完成每页可能存在不被视为公共接口一部分的其他成员（受保护和私有的函数和析构函数）。总之，这些功能成员是一组操作。典型的操作包括加法、减法、乘法和除法 - 所有这些操作都可以对整数进行。同样，公共接口是可以对类的实例进行的操作。

思考正确的操作集需要考虑几个用例：用户如何使用实例并与之交互？默认情况下，类会提供一些操作，但通常这些操作不足以执行任何有意义的操作。由于类是为表示概念或实体而编写的，所以开发人员必须提供有意义的行为和与其实例交互的方式。

类设计者和开发者确定这组操作。设计好一组的操作时，开发者应该考虑可行的最小操作集（基础）。其他非基础操作可以用这个最小集来实现。好的设计会减少执行计算或操作的函数数量，消除重复代码，最大限度地减少维护工作，并简化类。

## 7.1.3. 取值范围

值的范围与解释的概念有关，有效值通常有一端或两端有界。有一个 unsigned char 变量：最小值是 0，最大值是 255，字节的性质决定了这些界限。其他数据类型的值通常位于变量可能值的子范围内。我的机器上的整数是一个 32 位实体，其界限为正负 21.4 亿，但大多数问题不需要这个范围。例如，如果要对一个人的年龄进行建模，则值的范围需要限制为非负值，最大值可能为 150（很难说这个最大值应该是多少）。

假设 Student 类有一个名为 age 的整数变量，则可能的值范围将受到限制；使用无符号值会更好。首先，有效值应为非负值——年龄为负值是没有意义的。其次，最大值很难确定，最好忽略上限范围。此范围意味着任何年龄都不应超出指定值；范围允许有效值并排除所有无效值。有时，范围有时很明显，有时很模糊。

精心设计的类可强制每个变量可以容纳的值的范围，考虑每个数据字段以及所有数据字段的总体情况。每个数据字段都包含一些有关实例的重要信息，必须协同工作并有意义。例如，考虑一个 Movie 类，其中两个日期分别代表可供查看的第一个日期和最后一个日期。每个日期都必须保持有意义的值（不是 2 月 30 日），并正确地与另一个日期相关联。即使两个日期格式正确，最后一个查看日期早于第一个查看日期也不正确。

## 7.1.4. 内存机制

如果使用动态资源，内存的分配和使用对许多类来说都是一个问题。通常，实例变量的布局遵循开发人员的偏好，可能按数据类型或字母顺序排列。然而，C++ 允许开发人员根据需要确定内存的分配位置和方式。自定义行为的分配器是一种强大非凡的工具。内存布局取决于数据类型的大小和对齐方式。某些类型将落在 4 字节的起始边界上，而其他类型则落在 2 字节或 8 字节的边界上。如果实例变量定义不当，某些变量将存在未使用的内存间隙。当今的大内存机器中，这很少

会成为空间考虑的问题，但会成为机器性能的问题。根据对象的大小，可能需要两条或更多条缓存线，并且移动效率低下，浪费了内存。理想情况下，应将很少或根本不应将冗余的内存引入对象中。

### 7.1.5. 性能影响

虽然性能不是数据类型的特征，但性能受数据类型设计和实现的影响。性能优化应该是开发程序的最后步骤，这并不是因为它不重要，而是因为过早的优化可能会影响正确性和有效性。设计具有良好布局的高效算法、数据结构和最小尺寸实例要好得多。性能调优应该发现瓶颈和热点并设法进行纠正。正如许多人所建议的那样，必须使用适当的工具而不是直觉来实现性能。必要时，C++ 提供了优化内存分配和对象布局的机制。如果在正确的时间有节制地使用这些选项，其可能会发挥重要作用。

## 7.2. 类设计中的错误

很多时候，课程设计都是从一个相对纯粹的想法开始的，并且这个想法得到了很好的实现，但随后却面临着不可避免的变化。变化通常是因为类有效，但似乎不完整而需要。然后，需要进行额外的更改，设计受到损害，权宜之计胜过精确性，并且会发生混乱。类腐烂但仍然有用。代码异味（非理想技术或实现，通常是由于时间限制或对问题空间或代码影响的理解不足）不断增加，但时间和预算决定了其他优先事项。

检测这些问题通常比确定解决方案要简单。常见错误通常可以修复，对代码库的其他部分的影响很小，但这些错误仅代表现有问题的一小部分。关于类设计错误的内容可以写成一整本书。

良好的类设计是理解和维护类不变性的基础。类表示一种数据类型，必须保持一致性和正确性，实例才有意义。忽略不变性，会导致实例出现不一致和可能失败的情况。客户期望正确性，作为开发人员，我们有责任满足这一期望。

### 7.2.1. 类的不变量

“类的不变量”是类对象在构造之后，以及在该对象上的方法调用序列之间，必须始终满足的条件。其表示对象的一致、有效状态，通常通过构造函数、析构函数和成员函数强制执行。类不变量可确保对象数据的完整性和正确性。

每个类都应该代表一个概念或一个实体；类是一种新的数据类型。有意义地描述其概念或实体，聚合其他数据类型，从不同的类到原始类型。良好的设计和理解类不变量是同一枚硬币的两面。其抽象原则为：首先，类将概念或实体的细节减少到最低限度，用于定义类。其次，抽象表示整个实例可以作为一个单元来处理。这种实体或概念的抽象应该由用户代码有意义地表现。类应该避免以意想不到的表现，而让用户感到意外。

由于类表示概念或实体，所以必须将其内部数据限制为对其表示有意义的特定值。如前所述，`Student` 应该具有非负的 `age`。类是有关所表示实体或概念的知识的来源，不应依赖任何其他类或外部数据来传达。封装原则在某种程度上意味着，类所知道和所做的一切都应完全包含在类中。用户代码应该能够与实例交互或从实例获取任何必要的数据，而无需操作或查询其他对象或变量。

这些特征伴随着责任。类必须确保其所有数据成员（无论是单独还是整体）都是合法的，并且始终有意义，并且与它的交互可预测，不会出现意外。此外，它还必须确保在程序执行期间，这种敏感性和可预测性永远不会受到损害。这个类的责任称为“类的不变量”。不变量是实例的一个属性，它始终无论如何初始化或修改实例，都是 `true`。类绝不能创建不遵守不变量责任的实例，并且保护现有实例避免不变量的变化。

正确性特征对于程序的正确行为最为重要。如果程序代码不正确，则无法保证任何事情；无论多大的性能都无法弥补此错误，结果也是可疑的，保持类不变是确保正确性的重要手段。当类的行为受控且可预测时，用户可以放心地使用它，其结果也是有意义的。遗留代码中的类通常设计时，考虑了不变量以外的其他因素。存在大量机会来清理许多现有类，但修改（即使是使类更可预测和更强大的修改）可能会导致意外行为。一如既往，理想和理论必须尊重现实。目标是改进，但道路可能有些（或非常）坎坷。

### 7.2.2. 建立类的不变量

类表示一致、有凝聚力的概念或实体。类要求必须分别和总体指定每个实例变量的界限。总体表示实例的状态。一个重要的目标是确保实例的状态一致。继续使用 `Person` 的示例，如果 `age` 的值为负数，则 `Person` 的含义将受到威胁——负数年龄代表什么？

建立类不变量是构造函数的工作；维护不变量是修改器或其他改变状态函数的责任。每个实例变量必须初始化并保持有意义且正确的值。实例的解释取决于是否维持不变量。变量类型的值范围可能会超出不变量的范围，因此构造函数和变量必须确保没有值超出范围。我观察到一些代码，这些代码使用访问器来验证范围，并仅返回有意义的值，从而将实例变量初始化或设置为无效值。如果实例变量表示无效值，则必须考虑违反类不变量。

本章中的错误主要集中在建立类不变性上，所以本章重点介绍构造函数。下一章将单独讨论修改器，修改器是维持类不变性所必需的。这两章截然不同，但必须协调一致，共同努力才能使类不变性保持正确。

## 7.3. 错误 44： 未能维护类不变量

此错误主要针对类设计和开发中的正确性和有效性。实例变量必须初始化为类确定的边界值。这可确保对象在使用前定义明确且有效。

变量器会修改实例变量的状态。为了保持类不变性，必须确保检查输入参数的值是否在该变量的可接受范围内。此外，影响实例变量状态的代码都必须确保它不会损害不变性。确保构造函数、变量器和支持代码维护类的不变性至关重要。

每个构造函数负责初始化所有实例变量。如果是默认值，则该值通常必须有意义，并且如果默认值稍后未修改，则不会导致实例无效。每个变量必须确保其输入参数值在可接受值范围内，并且仅在这种情况下修改其实例变量。假设构造函数或变量的参数值无效。这种情况下，类的行为必须确保实例的状态符合不变量。许多情况下，无效参数输入应通过引发异常来处理；其他情况下，不应修改实例变量。最后一种方法可能会导致构造函数出现问题。使用构建器或工厂模式时，可能会出现进一步的复杂情况。其他情况下，实例变量在使用实例之前必须有效。

## 问题

正确编码的构造函数或构造函数集，将使用有意义的值初始化每个实例变量，否则不允许实例化对象（即抛出异常）。保持类不变性需要满足这个条件，而且不能低于这个条件。每个变量都必须验证输入参数，并确保变量不会因无效值而更新，否则会抛出异常。影响实例变量的所有其他方法都必须保持类不变性。当不遵守这些条件时，就会出现困难，如清单 7.1 所示。

许多类都有由编译器或开发人员编写的默认或无参数构造函数。假设开发人员没有为该类提供任何构造函数。这种情况下，编译器将编写一个默认构造函数，其中每个实例变量在调用时都将初始化为该类型的 0 值。该语句的条件部分表明，是否调用默认构造函数取决于变量的定义或赋值方式。

清单 *7.1* 典型的教科书结构

```
struct Person {
  int age;
  std::string name;
};

int main() {
  Person p1;
  std::cout << p1.age << " '" << p1.name << "'\n"; // 1
  Person p2 = Person();
  std::cout << p2.age << " '" << p2.name << "'\n"; // 2
}
```

注释 1：age 会输出垃圾信息（未定义），name 输出空字符串

注释 2：这将输出：0 ”

编译器提供了无参数构造函数，但在第一种情况下不会调用。第二种情况下，将调用提供的构造函数，这演示了将原始（或内置）变量初始化为该类型的零值的行为。这两种情况下，实例都会调用其默认构造函数。

清单 7.1 中的代码的输出是不可预测的——对于 age 变量，这是未定义的行为。内存的每个字节都有一些值；输出是代表该变量的字节可解释为的任何值，这让这段代码不确定。

## 分析

由于构造函数的职责是初始化每个实例变量，因此使用第一种方法会导致其失败。如果构造函数失败，就无法保持类不变。更糟糕的是，编译器对此从不抱怨；实例变量是内存中已有的未定义值，一切似乎都正常。如果使用这些值，将导致未定义行为。如何调用编译器编写的默认构造函数，取决于实例的设置方式。如果开发人员编写了默认构造函数，则无论哪种情况都会调用它。如果需要保持类不变，开发人员必须编写默认构造函数。例如，这是一个默认构造函数：

```
Person() : name("Joey"), age(21) {}
```

但这是一个好的、有意义的默认构造函数吗？这个构造函数很容易编写，但需要回答一个好的默认名称和年龄的问题。大多数情况下，默认构造函数会导致类不变量失败，实例是用无意义的数据初始化的——根据变量类型的约束，不是无效的，只是没有意义。虽然默认构造函数将名称 Joey 和年龄 21 分配给 Person 实例，但这有什么意义呢？用户代码稍后可以将值更改为对其问题有

意义的值，但如果忽略了实例的变量会发生什么？类不变量无法得到保证。

开发人员必须编写接受多于零个参数的构造函数。用户代码可以创建一个 Person 实例，并传递一个将姓名和年龄传入具有合理数据的双参数构造函数。

考虑 Person 结构的构造函数：

```
Person(const std::string& n, int a) : name(n), age(a) {}
```

如果为该类编写了任何构造函数，编译器将不会提供默认构造函数。如果还需要默认构造函数，开发人员必须明确包含它。如果不假设（或希望）用户以后会更改，则获取适合问题的默认构造函数默认值是一项负担。如果无法做到这一点，默认构造函数将违反约束。

如果不小心，变量器也可能违反类不变量。考虑以下 age 实例变量的修改器：

```
void setAge(int a) { age = a; }
```

年龄可以设置为负值或不合理的大值，这两种情况都没有意义。将下限限制为非负值可以解决取值范围的问题，但要真正解决上限问题却很有挑战性。什么值才是最大年龄？

**解决**

理想情况下，构造函数和修改器以协调的方式工作。构造函数有责任建立类不变量，但不一定直接初始化每个实例变量。如果构造函数和修改器都初始化或修改同一个实例变量，很可能重复。这些应该封装在变量中。构造函数可以调用每个变量，传递其初始化参数值，变量验证参数值并进行适当处理。构造函数必须确保调用变量来执行初始化，而不一定初始化实例变量。

这时，开发人员或编译器提供的默认构造函数将无效—无法为 name 或 age 变量默认有意义的正确值。尽管使用默认构造函数可能很诱人，但请尽可能避免使用，方法是编写至少一个需要每个实例变量参数的构造函数。

使用数组时没有默认构造函数会产生负面影响。定义类实例数组必须调用默认构造函数来初始化每个元素，所以没有默认构造函数则是一个错误。

清单 7.2 演示了构造函数和变量器协同工作，以使用部分年龄范围检查来初始化实例。需要注意的是，在这个略显复杂的代码中，age 设置了两次，其旨在演示限制允许的值范围——教师可以做这样的事情！将范围检查代码移动到私有验证函数，并在初始化程序和 setAge 函数中调用它是一种更好的方法。

清单 *7.2* 要求每个实例变量的初始化值

```
struct Person {
  int age;
  std::string name;
  Person(std::string n, int a) : name(n),
    age(setAge(a)) {} // 1
  int setAge(int);
};

int Person::setAge(int a) {
  if (a < 0)
    throw std::out_of_range("age must be non-negative");
```

```
12    age = a;
13    return age;
14  }
15
16  int main() {
17    Person annie("Annette", 25);
18    Person floyd("Floyd", -1); // 2
19  }
```

注释 1：构造期间调用成员函数的技术可能会产生未定义的行为，维持简单或选择其他方法

注释 2：引发 out_of_range 异常

当函数或构造函数无法有意义地执行其应执行的操作时，最好的解决方案是抛出异常。此方法不同于返回错误返回代码。错误返回代码说明由于某些错误条件而无法执行某些操作，但对象或函数仍处于良好状态。对于构造函数，不能使用无效值来初始化实例变量。如果没有合理的默认值可用，则不能实例化该对象，无法初始化该变量。因此，最好抛出异常。异常会强制调用者处理不存在对象的可能性。返回不正确的返回代码意味着已实例化该对象并保持类不变量。对于无效数据，两者都无效。

NOTE

本书中的几个解决方案都使用了在初始化列表中调用成员函数的方法。某些情况下，这可能会产生未定义行为。请参阅 https://compiler-explorer.com/z/PPes7vPYd 以获取有意义的示例。我将它们用于简单变量。如果该方法依赖于其他变量的状态，事情可能会很快变得一团糟，请谨慎使用此技术。

**建议**

- 确保构造函数正确初始化每个实例变量。
- 构造函数应调用实例变量赋值器进行初始化，以防止代码重复。
- 赋值器应在将值分配给实例变量之前验证输入参数值；如果值不正确，则抛出异常。
- 如果无法完全正确地初始化实例，则抛出异常——不要创建部分或错误构造的对象。
- 大多数情况下，避免使用默认构造函数；如果需要，请自己编写。

## 7.4. 错误 45： 未将类视为数据类型

内置的 C++ 数据类型（例如 int 和 double）直观、语法简单，且性能卓越。开发人员设计类时，这些特性必不可少。类设计就是类型设计；必须考虑构成正确，且有用类型的所有因素。很少有其他语言能够提供 C++ 为类设计者提供的灵活性，但必须考虑几个问题，包括内存分配和释放、对象实例化、初始化和销毁；重载和友元函数，以及类的不变量。

类型决定了实例所执行的一组操作。经常会需要相应的子类型，因此必须仔细考虑继承的正确使用。如果类或类型是基类，则必须定义一个对所有子类型都有意义的接口。必须确保子类型不会破坏类的不变量。

正确且有意义的类型设计，为读者和开发人员提供了一种易于使用且定义清晰的感觉。正确性对于类设计至关重要，但必须重视实例使用结果的可读性。良好的设计还可以有效地使用类型，这

样开发者就不必弥补缺失或尴尬的功能。最后，良好的类型设计会在以下情况下的使用方式：集合或大型集合；这会影响性能。请仔细考虑运行时间和空间成本，以减少与算法选择不当相关的问题。

类的四个主要特征影响其设计、编码和使用方式。必须仔细考虑，以确保所有相关人员都能有效使用该类。有效使用通常归结为类提供的一组操作。设计这组操作时，应尽可能简单让开发者可以直接使用。

**问题**

清单 7.3 是一个人为的例子，展示了考虑不周的数据类型实现为类时，出现的几个问题。开发者认为，通过开发一个简单的 Rational 类，有理数的概念就可以轻松抽象和使用。乍一看，代码看起来很合理，但它包含几个错误。该类旨在允许对 Rational 实例进行加法、乘法和输出，但使用此类会产生尴尬的代码。

其理念是使用有理数进行计算，然后输出结果。使用此代码的客户很快就会发现，没有提供常规的预期操作（例如，使用 plus 方法，而非 + 操作符），并且使用起来变得混乱且不清楚。这种难以理解的方法，还会浪费开发人员的时间。

清单 *7.3 Rational* 类的原生实现

```
class Rational {
private:
  double num;
  double den;
public:
  Rational() : num(0), den(1) {}
  Rational(double n, double d) : num(n), den(d) { reduce(); }
  void setNumerator(double n) { num = n; }
  void setDenominator(double d) { den = d; }
  double getNumerator() { return num; }
  double getDenominator() { return den; }
  static int gcd(int a, int b) { return a == 0 ? b : gcd(b % a, a); }
  void reduce() {
    int div = gcd(num, den);
    num = (den > 0 ? 1 : -1) * num / div;
    den = abs(den) / div;
  }
  Rational plus(const Rational& o) const { // 1
    int n = num * o.den + den * o.num;
    int d = den * o.den;
    return Rational(n, d);
  }
  Rational times(const Rational& o) const { // 2
    int n = num * o.num;
    int d = den * o.den;
    return Rational(n, d);
  }
  void print() { std::cout << num << '/' << den; } // 3
};
```

```
30
31 int main() {
32   Rational r1(1, 3);
33   Rational r2(2, 4);
34   Rational r3 = r1.plus(r2);
35   r3.print();
36   std::cout << '\n';
37   Rational(1, 0);
38   return 0;
39 }
```

注释 1：命名不当的加法操作符
注释 2：命名不当的乘法操作符
注释 3：命名不当的输出操作符

**分析**

这一类 Rational 最突出的特点是误解了有理数的本质。从本质上讲，有理数由比率组成(因此得名 rational，它并不像我最初认为的那样，指的是“合理”的数字)。数论断言分子和分母是整数，而不是双精度数。遵循合理的数学推理至关重要。接下来，考虑主函数中返回之前的最后一行代码。这种情况未定义，因为分母是一个除以零的问题。

必须提出一个重要问题：有理数会改变吗？我更喜欢不可变数据；这是该原则的一个典型例子。可变版本必须使用不同的考虑因素。数字永远不会改变，所以有理数构造之后，就不应该改变其值(但这取决于要解决的问题)。尽可能在所有地方删除变量。考虑独立于分母获取分子是否有意义——这时，可能没有意义。需要删除访问器。

确定用户是否应该能够调用 gcd 或 reduce。这些函数是作为 Rational 类的辅助函数提供，不应成为公共接口的一部分—将其标记为 private。最后，确定这些函数是否应该内联；通常，递归函数不会内联，但实际结果取决于编译器。通过在类内部定义，可将其隐式内联。这种方法将按照编译器认为合适的方式实现。可以让 gcd 保持隐式内联，其很简单并且在类中实现；编译器将确定其实际情况，但 reduce 不太可能通过内联来节省重大开销。

考虑一下 plus 和 times 函数的含义。设计者希望其是加法和乘法，但不清楚这些函数是否会改变调用对象（对 r1 使用 plus），或者是否会创建一个用计算结果初始化的新对象。源码可以说明这个问题，但因为功能实现的反直觉会严重影响可读性。

print 函数意图良好，但不太直观。人们可能认为 std::cout 是使用 Rational 实例输出的自然流，而好的设计在形式上通用，但这种设计是具体且不灵活的。这三个设计不良的函数严重影响了效率，因为其不直观，也不遵循已知的使用模式。

**解决**

此代码是重新设计的版本，考虑了有理数的含义，并在代码中进行实现。plus 和 times 方法已重新实现，以使用标准算术 + 和 * 操作符。用户可以使用它们直观地使用预期计算符号的代码。print 方法已更改为为 ostream 类重载 operator<<。这种方法能让开发人员使用标准插入操作符，将数据添加到输出流 - 就像所有内置数据类型一样。这种一致性使代码更加自然有效。

清单 7.4 提供更加直观的实现

```
class Rational {
  private:
  int num;
  int den;
  static int gcd(int a, int b) { return a == 0 ? b : gcd(b % a, a); }
  void reduce();
  int validate(int v) {
    return v != 0 ? v : throw
      std::invalid_argument("zero denominator");
  }
public:
  Rational(int n, int d=1) : num(n), den(validate(d)) { reduce(); }
  Rational operator+(const Rational& o) const;
  Rational operator*(const Rational& o) const;
  friend std::ostream& operator<<(std::ostream&, const Rational&);
};
void Rational::reduce() {
  int div = gcd(num, den);
  num = (den > 0 ? 1 : -1) * num / div;
  den = abs(den) / div;
}
Rational Rational::operator+(const Rational& o) const { // 1
  return Rational(num * o.den + den * o.num, den * o.den);
}
Rational Rational::operator*(const Rational& o) const { // 2
  return Rational(num * o.num, den * o.den);
}
std::ostream& operator<<(std::ostream& out, const Rational& r) { // 3
  out << r.num << '/' << r.den;
  return out;
}

int main() {
  Rational r1(1, 3);
  Rational r2(2, 4);
  std::cout << r1 + r2 << '\n';
  //Rational(1, 0);
  return 0;
}
```

注释 1：自然加法操作符

注释 2：自然乘法操作符

注释 3：自然插入操作符

通过将实例变量设为整数，可以体现有理数的含义。辅助函数会巧妙地隐藏起来，仅供类使用。除以零的运算得到适当处理。操作符直观易懂，用户了解加法和乘法不会影响调用（或左侧）对象。插入操作符的工作方式与其他操作符一样，所以用户可以像使用其他操作符一样使用—显著提高了可读性和有效性。

考虑内联与外联函数定义时，有几个问题需要考虑。以下是需要考虑的一些问题：

- 内联函数可以通过将函数代码，直接嵌入到调用点来减少函数调用开销，这可能会提高频繁调用小函数的性能。但是，过度内联会增加代码大小并降低缓存性能。外联函数通常具有函数调用的开销，但不会增加调用点的代码大小，这有助于保持更好的指令缓存局部性。
- 如果过度使用内联函数或将其用于大型函数，因为代码会插入到每个调用点，会导致代码膨胀。外联函数有助于保持可执行文件的大小较小，因为函数代码是重复使用的，而非重复。
- 内联函数会使调试更加困难，因为函数代码在多个位置重复，使堆栈跟踪和调试工作变得复杂。外联函数将实现集中起来，使其更易于维护和调试。
- 内联函数通常在头文件中定义，如果内联函数发生更改，则会增加重新编译多个翻译单元的风险。外联函数在源文件中定义，对这些函数的更改不会影响头文件和依赖于头文件的文件，有助于最大限度地减少重新编译。
- 内联函数通常在头文件中定义，实现的细节会暴露，很不理想的封装。外联函数可以隐藏源文件中的实现细节，从而改善封装，以及接口和实现的分离。
- 内联函数最适合简单函数，以内联方式定义可以将相关代码放在一起，提高可读性。外联函数最适合复杂函数，分离实现有助于保持类定义的清晰易读。

**建议**

- 请记住，实现类设计就是设计一种新的数据类型。对其含义有基本了解的人来说，使用类型时操作不应反直觉。
- 当基于符号的操作符，能够在数据类型上下文中准确传达操作的含义时，勿必对其进行定义。切勿仅仅因为它们“看起来很酷”而使用。
- 保持数据流和代码设计的通用性，尽可能不要将用户锁定在一条特定的路径上。最好的方法是设计一种数据类型，使其接近内置类型的自然操作。
- 明智地使用内联，仔细考虑内联和外联方法之间的权衡。

## 7.5. 错误 46： 未对方法建立基准

这个错误关注的是正确性和有效性。多个构造函数通常作为单独的操作实现。此外，计算方法经常重复计算代码。这种重复应该导致考虑确定基准。并非所有重复都必须确定基准部分，但这是一个好兆头，表明可能找到。基准是一个数学概念，定义了一组最小函数 - 所有函数都是必不可少的，并且没有重叠 - 可以从中实现其他非基础函数。don’t repeat yourself(DRY) 原则与本次讨论相关，但必须服务于比通常呈现的更细微的目的 - 稍后会详细介绍。

实现多个构造函数或函数的差异，可能会导致错误的行为。在一个地方支持基本代码，可以节省开发者的时间。

本讨论分析了构造函数，但适用于方法和独立函数。主要想法是减少行为实现的数量，避免代码随着时间的推移而出现分歧。当功能重复并以不同方式维护时，事情很快就会出错。

通常，类会有多个构造函数，每个构造函数接受不同数量的参数。如果实例化调用中未提供某些信息，则需要提供默认值。在其他情况下，将应用默认行为来填补知识空白。

**问题**

一个简单的例子可能有助于更好地理解基函数的含义。考虑一个包装内置 `int` 类型的 `integer` 类。这四种基本数学运算可能彼此之间有足够的差异，因此所有这四种运算都可以唯一地实现。然而，仔细分析表明事实并非如此。减法问题 `a - b` 在数学上等同于 `a + -b`。减法可以用加法和反相来实现。因此，减法不需要是单独的运算。乘法是重复的加法，除法是重复的减法——或者重复加一个反相值。因此，只需要两个基函数来实现这四个算术函数——将两个整数值相加并对一个整数值取反。其他三个运算可以根据这两个进行定义。

这种操作以其他函数形式实现的思想，确立了如何区分基函数，并直接以这种方式实现。通过在尽可能少的地方编写计算代码，来确保正确性。以这些方式实现的其他方法将正常工作，并且不会重复可能在发生变化时变得不匹配的代码。通过为每个函数编写核心代码来保持有效性，并且通过在其他非基函数中反复使用来证明其正确性。

`Cylinder` 代码示例（清单 7.5）展示了一种典型的方法，即在未适当考虑基函数的情况下。其构造函数和操作是单独且独立地编码的。在类开发的早期，很容易假设代码在未来保持不变。很多情况下，这并不正确。通过添加一两个新要求，实现这些构造函数和函数可能会彼此分歧。任何分歧都会影响正确性、可读性和有效性。

清单 7.5 类中重复的知识

```
class Cylinder {
private:
  double radius;
  double height;
  double area;
  double volume;
public:
  const double PI = 3.1415927;
  Cylinder() {
    radius = 1;
    height = 1;
    area = PI;
    volume = PI;
  }
  Cylinder(double h) {
    radius = 1;
    height = h;
    area = PI;
    volume = PI * h;
  }
  Cylinder(double r, double h) {
    radius = r;
    height = h;
    area = PI * r * r; // 1
    volume = PI * r * r * h; // 2
  }
  double getBaseArea() const { return area; }
```

```
28    double getVolume() const { return volume; }
29  };
```

注释 1：使用标准公式计算面积

注释 2：体积等于面积 * 高度；再次计算面积

### 分析

这些构造函数和函数的实现会重复知识，DRY 原则旨在防止这种情况。然而，DRY 通常实现为“不重复代码”方法，而不是更有帮助的“不重复知识”方法。随着时间的推移，功能类将容易受到新需求的影响，从而导致代码的添加和修改。这些变化是不可避免的，知识重复的可能性也会增加。当复制的知识在一个地方发生变化，而另一个地方没有变化时，就会发生分歧。一段时间后，就不清楚正确的版本是哪个了。

每个构造函数都会重复实例变量的初始化，有时使用默认值，有时使用参数。这种重复表明了一种更好的方法。代码展示了一种典型的模式，可以将其重构为单个辅助函数。辅助函数的好处是，所有构造函数都可以使用，并避免知识和代码的重复。当添加新需求时，更改会隔离到辅助函数中。

### 解决

解决这个问题的最佳方法是，了解如何根据其他函数来实现函数，然后将该依赖关系缩小到最低限度。构造函数应该将标准代码分解出来，并将其放在私有辅助方法中。此辅助方法将成为根据辅助方法，实现的构造函数的基础。getBaseArea 方法计算圆柱体圆形底面的面积。volume 方法计算圆柱体圆形底面的面积，然后将其乘以其高度。能否分解出通用代码，并进行单一面积计算决定了必要的基础，这时体积方法应根据 basearea 方法实现，而非辅助函数。

方法重用也发生在类继承中，其中派生类继承基类方法功能并对其进行添加。许多情况下，重写的派生类方法可以调用基类方法，作为其计算的一部分，并根据需要对其进行修改。这种方法可以防止功能重复，并允许派生类从基类中受益。设计得当的基类方法，应该是派生类基础的一部分。

这些重构步骤已在清单 7.6 中执行。已创建一个名为 init 的辅助方法，来处理构造函数中的先前重复。

volume 方法现在使用 basearea 来计算圆柱体的面积，并将该值乘以高度。体积以底面积的形式实现，以避免在变化发生时出现分歧。简单的情况下，知识重复可能不是问题。然而，这个问题可以扩展到更复杂的类，其中知识重复的影响更大。

清单 *7.6* 函数的最小集合

```
1   class Cylinder {
2   private:
3     double radius;
4     double height;
5     void init(double r, double h) {
6       radius = r;
7       height = h;
8     }
9   public:
10    const double PI = 3.1415927;
11    Cylinder() { init(1, 1); }
```

```
  Cylinder(double h) { init(1, h); }
  Cylinder(double r, double h) { init(r, h); }
  double basearea() { return PI * radius * radius; } // 1
  double volume() { return basearea() * height; } // 2
};
```

注释 1：计算面积

注释 2：计算体积

**建议**

- 将通用构造函数代码分解为辅助函数，该函数将成为类的基础集的一部分。
- 考虑如何根据其他基础函数实现函数，以简化编码、避免发散，并为该知识维护真实来源。
- 最小化基础函数集，确保功能不重叠。
- 添加新函数时，以基础函数为基础，并重新评估是否需要新的基础函数。
- 如果这种方法变得难以操作或过于笨拙，请放宽限制。请记住，原则有助于最大限度地减少技术债务，并加快开发速度，但盲目执行可能会导致更糟糕的后果。
- 寻找机会将重叠的方法，重写为基础函数或集合。

## 7.6. 错误 47： 未能实现“三大”函数

资源管理影响的主要特征是正确性。不正确的行为会影响程序，并且可能对程序本身（例如：系统）产生不利影响。大多数其他正确性问题的范围有限—这种错误可能无处不在，影响其他程序和系统。

三大函数——复制构造函数、复制赋值操作符和析构函数——在 C ++ 的每个版本中都可用，并且在使用动态资源时应进行实现。缺少其中一个或多个，可能会对程序产生不利影响。

许多类都有一个或多个基于值的实例变量，示例包括整数或双精度数来表示数值。有些使用 `std::string` 或 `std::vector`，这些变量在实例化时初始化，在程序执行期间使用一段时间，然后在实例超出作用域时销毁。这些变量可自动管理，不会带来不利影响。

但那些更复杂的类使用动态或有限的资源，并使用变量来进行表示。典型的例子是基于指针的变量，这些变量通常表示动态内存资源。同样，有限资源通常由基于值的变量表示，这时资源必须妥善管理。这可能包括数据库连接、套接字或文件句柄。虽然当实例超出范围时会自动管理实例变量，但引用的资源不会自动管理。这种管理缺失增加了不正确的行为的概率，开发者必须在这些情况下保持警惕并正确管理资源。

**问题**

当类具有动态资源时，必须正确地管理这些资源，可将其管理视为类不变量的一部分。该类承诺在其操作中建立、管理和删除资源。动态内存的示例展示了，正确的分配和释放步骤。

某些用户代码调用构造函数。构造函数中发生 new 操作以获取动态内存，从而建立部分不变量。其他实例方法使用此内存，保持不变量完好无损。最后，当实例超出范围（或删除）时，将调用其析构函数以释放内存。如果此方法正确执行，则不会出现内存泄漏，也不会出现双重释放和释放无效指针的情况，保证了程序和系统的稳定性。

另一个问题出现在与构造函数和析构函数操作无关的动态资源上。复制赋值操作符存在影响正确性的问题，必须仔细考虑。这里区分一下复制构造函数和复制赋值操作符；在创建实例时获取原始内存块时会调用构造函数，此内存可以容纳任何东西；其状态未定义。构造函数必须使用有意义的数据初始化原始内存，以建立类的不变量。

另一方面，复制赋值操作符不会初始化实例；可使用另一个已初始化实例的值来改变现有实例。必须小心确保复制赋值操作符处理自赋值（应该只是返回）。复制构造函数必须获取新的动态资源，并初始化其指针或相对于该值的句柄变量。复制赋值操作符必须执行相同的操作，但要多加一步。如果复制赋值操作符的目标已经具有动态资源，则在许多情况下必须将其销毁以避免内存或资源泄漏。某些情况下，可以复制源实例值，但其他情况下需要修改此方法。

复制构造函数和复制赋值操作符中的许多代码将会重复，可将此代码重构为两者都可以调用的私有辅助函数。必须考虑调用析构函数的影响，以确保重复的销毁代码得到正确处理。RAII 模式演示了这种成对的分配和释放。

假设已经编写了一个用于操作 wiki 页面的项目，负责该项目的公司要求每个页面都有一组特定的标题。提出了一个新要求，即开发一个复制操作来复制一个页面作为新页面的基础，复制构造函数是此操作的理想选择。

添加此复制构造函数代码是为了包含新的复制功能，但开发人员忘记添加复制赋值操作符的实现。

清单 7.7 共享唯一资源的复制构造函数

```
class TextSection {
  // assume a clever implementation
};

class Page {
private:
  TextSection* headers; // 1
  TextSection* body;
public:
  Page(TextSection* h) : headers(h), body(new TextSection()) {}
  Page(const Page& o) : headers(o.headers), body(o.body) {} // 2
  Page& operator=(const Page&);
};

int main() {
  Page p1(new TextSection());
  Page p2 = p1;
  return 0;
}
```

注释 1：动态资源

注释 2：复制动态资源指针；两个对象使用一个动态资源

## 分析

开发人员需要解决复制构造函数使用共享语义这一事实。头文件可以共享，公司规定的，且不可

更改。共享的正文是一个问题。作为正常操作的一部分，复制页面将对其正文进行更改。由于是共享的，这将影响复制自的页面。此外，当 Page 因超出范围而销毁时，堆内存中的正文 TextSection 将隔离且无法访问，从而导致内存泄漏。

正确的复制构造函数必须考虑，指针浅复制实现的共享语义，这对于页面主体来说不利。复制的页面必须分配一个从复制自页面初始化的新 TextSection。由于主体文本是由指针处理的动态资源，因此析构函数必须确保当实例超出范围时，在销毁实例之前删除动态资源。添加析构函数可确保实现正确的销毁。最后，由于已添加复制构造函数和析构函数，因此也需要实现复制赋值操作符。

复制构造函数假定新实例使用原始内存（填充了垃圾值）。复制赋值操作符假定受影响的实例已正确构造，并且每个实例变量都具有合法值。Page 类显示将源主体指针复制到目标（浅复制），会导致设计不良的复制构造函数中演示的错误。复制赋值操作符必须确保对动态资源使用深复制语义，还必须确保在深度复制发生之前正确删除或以其他方式处理现有动态资源。复制赋值操作符必须在创建新主体之前，添加代码以删除现有主体。

**解决**

清单 7.8 展示了这些改进。复制构造函数假设没有现有的动态资源，可分配一个新的 TextSection 对象。类似地，复制赋值操作符执行相同的分配，但首先通过删除现有资源来进行处理。最后，析构函数确保删除所有动态资源。

三大系统之间的资源协调可以避免资源泄漏（通常以内存泄漏的形式出现），保留了特定对象的唯一性，确保了程序运行的正确性。

清单 *7.8* 保持唯一性的复制构造函数

```
class TextSection {
  // assume a clever implementation
};

class Page {
private:
  TextSection* headers;
  TextSection* body;
// 1
public:
  Page(TextSection* h) : headers(h), body(new TextSection()) {}
  Page(const Page& o) : headers(o.headers), body(new TextSection(*(o.body))) {}
// 2
  Page& operator=(const Page&);
  ~Page() { delete body; }
};
Page& Page::operator=(const Page& o) {
  if (this == &o)
    return *this;
  headers = o.headers;
  delete body;
  body = new TextSection(*(o.body));
```

```
  return *this;
}

int main() {
  Page p1(new TextSection());
  Page p2 = p1;
  return 0;
}
```

注释 1：动态资源

注释 2：从现有副本创建，并初始化新的动态资源

现代 C++ 将“三大”更改为“五大”，另外两个成员是移动构造函数和移动赋值操作符。对三大函数的建议保持不变，但增加了移动语义。清单 7.9 显示了清单 7.8 中的代码，使用现代移动语义和智能指针进行了改进。注意 =default 的使用。

清单 7.9 添加唯一性保护、移动构造和赋值

```
class TextSection {
  // assume a clever implementation
};

class Page {
private:
  TextSection* headers;
  std::unique_ptr<TextSection> body;
public:
  Page(TextSection* h) : headers(h),
  body(std::make_unique<TextSection>()) {}
  Page(const Page& o) : headers(o.headers),
  body(std::make_unique<TextSection>(*o.body)) {}
  Page& operator=(const Page& o) {
    headers = o.headers;
    body = std::make_unique<TextSection>(*o.body);
    return *this;
  }
  Page(Page&&) = default; // 1
  Page &operator=(Page&&) = default; // 2
};

int main() {
  Page p1(new TextSection());
  Page p2 = p1;
  return 0;
}
```

注释 1：现代的移动构造函数

注释 2：现代的移动赋值操作符

**建议**

- 如果在类中使用了动态资源，请务必编写三大函数：析构函数、复制构造函数和复制赋值操作符。
- 构造函数用于初始化原始内存，而赋值操作符用于更新现有的、已构造的实例。务必妥善处理现有资源。
- 确保已移动对象无法重用。

## 7.7. 错误 48： 仅为了代码复用而使用继承

正确使用继承对于正确性、可读性和有效性至关重要。如果使用不当，继承会引入异常，难以阅读和推理，并且需要花时间来整理细节。

在面向对象编程的早期，继承被吹捧为解决所有软件工程问题的灵丹妙药。DRY 原则很快被采纳为鼓励继承的典范。教科书赞扬了它的使用，并教导人们代码重用可以实现。

主要思想是只在基类（如果使用其他语言，则为 superclass）中编写一次代码，然后在派生类中重用该代码。这个承诺很诱人，我们中的许多人都上钩了。我们不得不编写奇怪的代码，为这个类而不是那个类产生的异常分散注意力，所以这里需要理解 is-a 关系的概念。

所有的童话故事都包含一些事实，但幻想的细节在现实世界面前似乎黯淡无光。多年来，代码重用的童话故事已经黯淡了许多。教科书仍然在继承的背景下提到代码重用，这种建议是使用继承的最糟糕的理由。代码重用是继承的一个好处，但它绝不是继承的理由。

**问题**

如果使用得当，继承是一个好主意，也是一种非常强大的技术。派生类是基类，派生类和基类之间存在一种是关系。这种关系正是 Barbra Liskov 博士在现在著名的 Liskov 替换原则（LSP，Liskov substitution principle）中所描述的关系。当需要基类型的实例时，可以用派生类型的实例代替。我们正在研究类，其是就是数据类型。

> **NOTE**
>
> 此错误仅讨论公共继承。受保护、私有和虚拟继承完全不同，不应用于 is-a 关系。本书未涉及这些内容。

在 LSP 中可能需要明确的是，在处理对象集合、一些基类和各种派生类时，可以特定的方式对待它们。每个实例都可视为基类：“基类知晓和实现的一切，派生类同样知晓并实现。”当然，这并不意味着基类知道和做的一切都可以直接被派生类访问（私有成员不可访问）。所以，最好这样说：“基类的所有非私有成员（数据与方法），派生类均可直接访问并调用。”

第一个含义是，只有基类接口才能用于处理对象和子对象的集合—如果某个行为未在基类上定义，则无法在派生类上调用它。第二个含义是，集合通常必须处理指向对象的指针，而不是对象本身。许多容器按值保存对象，值的大小固定。试图将（很可能更大的）派生对象挤进基类大小的空间注定会失望—复制过程悄无声息地进行，但对象的派生类部分会被切掉，只留下基类部分。

考虑从 Person 类派生出 Student 类的情况，如以下清单所示。Person 类的所有知道（并且做），Student 类知道（并且做）。

清单 7.10 简单的继承层次结构

```
class Person {
  std::string name;
  int age;
};

class Student : public Person {
  double gpa;
};
```

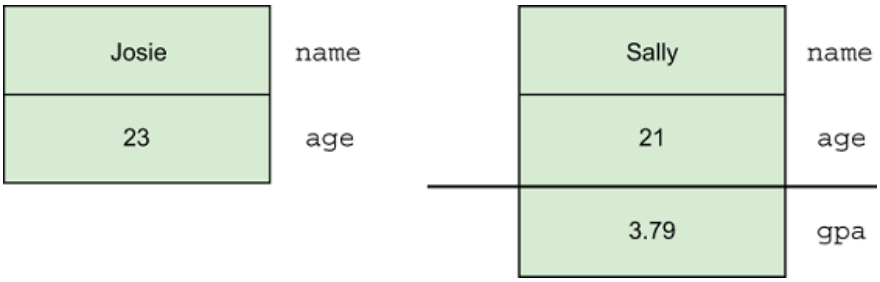


图 7.1 Person 和 Student 对象显示实例变量及其布局

要理解为什么派生类可以替代基类，请研究图 7.1。在左侧，有一个类 Person 的对象，有两个实例变量：name 和 age。右侧的 Student 对象派生自 Person 并添加了一个 gpa 实例变量。Student 的顶部看起来像 Person，正是因为 Student 是 Person。Student 的底部是附加的实例变量。Student 可以视为 Person 或 Student，具体取决于如何访问，所以其可以表现可以是多态的。

Josie 是 Person，Sally 也是。如果将 Sally 引用为 Person，则有关 Person 的所有内容都可以访问。只有将 Sally 视为 Student 时，底部部分才可访问，因为学生具有 gpa 值。图 7.1 还说明了为什么将 Sally 存储在 Person 元素数组中，只有空间容纳基类部分；Student 部分被切掉，Sally 突然只有 Person 部分了—对象失去了 Student 性。

**分析**

多态性是将类排列成层次结构的主要原因，唯一的其他原因是重用代码。然而，这个原因是反面原因；造成的麻烦和尴尬，远节省的多。

以下清单中，基类具有派生类想要重用的功能。但派生类无法通过“is-a”关系与基类相关联。此示例旨在展示当类之间没有紧密联系时，尝试重用方法会暴露一些缺陷（通常较晚显现）。

清单 7.11 根据外观关联类

```
class Square {
private:
  double side;
  double offset;
public:
  Square(double side) : side(side), offset(0) {}
  virtual void move(int n) { offset += n*side; } // 1
  double getOffset() { return offset; }
```

```
  double area() { return side*side; }
};

class Horse : public Square { // 2
public:
  Horse(double height) : Square(height) {}
};

int main() {
  std::vector<Square*> squares;
  squares.push_back(new Square(1));
  squares.push_back(new Horse(15));
  for (int i = 0; i < squares.size(); ++i) {
    squares[i]->move(2);
    Horse* h = dynamic_cast<Horse*>(squares[i]); // 3
    if (h) {
      std::cout << "Horse moved " << h->getOffset()
        <<" meters ahead\n";
    } else { // 4
      std::cout << "Square moved to (" << squares[i]->getOffset()
        << ", " << squares[i]->getOffset() << ")\n";
      std::cout << "Square area is " << squares[i]->area() << '\n';
    }
  }
  return 0; // 5
}
```

注释 1：正方形可以在图形空间中移动
注释 2：马可以移动，但应该以不同的方式移动
注释 3：强制类型转换可能是一个不好的迹象（代码异味）
注释 4：这是一个烂摊子；随着更多类的添加，会变得越来越复杂
注释 5：动态对象已泄漏；应该将其删除

首先，当迭代这些相关性较差的类的容器时，必须添加定制处理代码，以确保 Horse 实例不会调用现在毫无意义的 area 方法。这种负担迫使开发者，将每个派生对象作为特殊情况来处理，从而破坏了相关类容器的顺畅流动。此外，这种特殊处理可能只会添加到一些新开发的代码中。

其次，move 方法表示将方块移动到某个点 n 个单位上方和 n 个单位右侧（如果使用负值）。但对于马来说，getOffset 则意味着完全不同的东西，并且方法名称无法传达其意图。

第三，Horse 对象无法有意义地替代 Square 对象。Horse 实例将同时具有数据字段，并继承这三种方法；Horse 对象不能拥有合法区域，也不能像 Square 那样移动。虽然这个例子是人为的，但这些关键的评论点对于在使用继承，进行代码重用来说非常有效。

**解决**

有两种方法可以解决这个问题。第一种方法是拒绝以这种方式编码。当派生类与其基类正确相关且在所有情况下可替换时才使用继承。这种关系的参考点是派生类是否始可以在需要基类实例的地方替换 - 再次强调，每次都是如此。可置换性表示每个数据字段和行为在基类和派生类的上下文中都有意义。

清单 7.12 中的代码通过打破垂直继承层次结构，将其水平展开。程序可能需要 Horse 和 Square 对象，但并非直接相关。每个类都定义了自己的一组实例变量（是什么）和方法（做什么）。两者都有一个 move 方法，但事实证明这只是偶然的。更恰当的命名方法 getOffset 和 getPosition，可以更好地沟通每种方法的目的。最后，对象集合会分成不同的容器，这更直观，也更接近现实。

清单 7.12 分离不相关的类

```
class Square {
private:
  double side;
  double offset;
public:
  Square(double side) : side(side), offset(0) {}
  void move(int n) { this->offset += n*side; } // 1
  double getOffset() { return offset; }
  double area() { return side*side; }
};

class Horse {
  double height;
  double position;
public:
  Horse(double height) : height(height), position(0) {}
  void move(int offset) { position += offset; } // 2
  double getPosition() { return position; }
};

int main() {
  std::vector<Square*> squares; // 3
  squares.push_back(new Square(1));
  for (int i = 0; i < squares.size(); ++i) {
    squares[i]->move(2);
    std::cout << "Square moved to (" << squares[i]->getOffset()
      << ", " << squares[i]->getOffset() << ")\n";
    std::cout << "Square area is " << squares[i]->area() << '\n';
  }
  std::vector<Horse*> horses; // 4
  horses.push_back(new Horse(15));
  for (int i = 0; i < horses.size(); ++i) {
    horses[i]->move(10);
    std::cout << "Horse moved " << horses[i]->getPosition()
    << " meters ahead\n";
  }
  return 0;
}
```

注释 1：方格可以以自己的方式移动
注释 2：马可以按照自己的方式移动

注释 3：将方格放在一起

注释 4：将马放在一起

第二种方法使用仅定义纯虚函数的基类。Java 等语言提供称为 Interface 的类状结构，没有实现细节（这在 Java 8 及更高版本中被放宽）。其思想是提供行为（抽象函数）的接口声明，而不提供实现。每个派生的具体类都实现该接口，并且必须为每个抽象方法提供代码主体。使用这种类的最显著优势是，以前不相关的类可以在更抽象的级别上相关联，但必须非常小心地确保这些类必须通过这种共同点相关联。

此示例中，如果问题主要涉及移动对象（在图形绘制程序中），则有必要通过每个类实现一个典型的抽象基类，并声明一个移动方法将这两个类关联起来。注意不要在没有模式的地方看到模式；不要因为一些相同的方法或变量名称而“关联”类。关注语义（含义），而不是语法（语法和命名）。例如，Square 和 Horse 类可以使用可移动的概念相关联。

此概念不同于 C++20 中引入的技术语言功能。可移动的对象表明其实现了移动方法，而可移动对象的容器表明每个元素都可以移动（但没有其他共同点）。以下代码演示了此概念。

清单 *7.13* 使用类似接口的类

```
class Moveable { // 1
public:
  virtual void move(int) = 0;
  virtual double getPosition() = 0;
};

class Square : public Moveable {
private:
  double side;
  double offset;
public:
  Square(double side) : side(side), offset(0) {}
  void move(int n) { this->offset += n*side; } // 2
  double getPosition() { return offset; }
  double area() { return side*side; }
};

class Horse : public Moveable {
private:
  double height;
  double position;
public:
  Horse(double height) : height(height), position(0) {}
  void move(int offset) { position += offset; } // 3
  double getPosition() { return position; }
};

int main() {
  std::vector<Moveable*> movers; // 4
  movers.push_back(new Square(1));
```

```
  movers.push_back(new Horse(15));
  for (int i = 0; i < movers.size(); ++i) {
    movers[i]->move(2);
    std::cout << "Mover moved " << movers[i]->getPosition() << " units \n";
  }
  return 0;
}
```

注释 1：通过创建抽象方法来声明行为的混合类

注释 2：方格可以按自己的方式移动

注释 3：马可以挡路

注释 4：一组可移动的物体；这些不应该认为是方块和马

当使用类似接口的类时，确保每个方法对于该类的意图和含义都是有意义的捕获。Liskov 替换原则在这里的应用与以前一样严格——需要基类实例的地方，都可以正确且有意义地替换派生类实例。

**建议**

- 多态性应该是继承的主要动机。
- 如果没有多态性，继承的目的就会退化为代码重用。
- 代码重用会导致书写和阅读上的尴尬。
- 将使用继承的类分开以实现代码重用，并以相关的方式编写方法。

## 7.8. 错误 49： 过度使用默认构造函数

这个错误严重影响正确性，因为它影响了类的不变性。因为阅读使用默认构造函数的代码，无需检查便可知道初始化值为何，所以可读性也会受到影响。

确保对象的实例化保持类的不变量，是构造函数的主要职责。许多类需要对实例变量进行特定的初始化，以使其处于有意义的状态。当未通过参数指定初始化值时，提供默认构造函数以向变量提供有效（但不一定有意义）的值。如果无法确定正确的默认值，则默认构造函数将实例初始化为不一致的状态。假设类设计者希望开发人员稍后用有意义的值填充这些值。如果没有调用一个或多个变量器来满足这一期望，就会出现问题。

**问题**

默认构造函数可能看起来更有效，但通常情况并非如此。默认值必须在稍后修改，以添加有意义的数据，但会使有效性和可读性变得复杂。假设在使用实例之前更改初始化值，性能会就会受到影响。

实例的状态中，不应包含无效或无意义的数据。如果使用默认构造函数，则必须将类设计为有意义的默认值，这些默认值无需进一步修改即可使用。

清单 7.14 中的代码有些就这样。由于没有给出开发者提供的默认构造函数，因此编译器提供了其默认版本，其将所有类实例初始化为其默认值。内置实例根本没有初始化，name 为空字符串（一个奇怪的名称），而 age 未定义（垃圾）。只有调用容易忘记的 setter 来提供有意义

的值，才能实现对此类对象的有意义使用。如果没有验证，无效值可能会分配给实例变量，所以此类 setter 也很危险。

清单 7.14 默认构造函数，导致无意义的结果

```
class Person { // 1
private:
  std::string name; // 2
  int age; // 3
public:
  std::string getName() { return name; }
  int getAge() { return age; }
};

int main() {
  Person p1; // 4
  std::cout << p1.getName() << ' ' << p1.getAge() << '\n';
}
```

注释 1：开发者没有提供默认构造函数；因此，编译器会编写一个
注释 2：类实例将调用其默认构造函数
注释 3：内置实例将不会初始化（将包含垃圾数据）
注释 4：将调用编译器编写的默认构造函数—name 将初始化，但 age 不会

添加显式的默认构造函数也并没有什么好处：

```
Person() : name(""), age(0) {}
```

但 age 实例变量将初始化，从而避免未定义的行为。

**分析**

不可能出现没有姓名，且年龄为零的人；但如果默认构造函数是必要的，则必须允许这种危险情况发生。默认值可以更改为 Lakshmi 和 21，但这样一来，所有默认的人的名字都是 Lakshmi，年龄都是 21。这种方法没有任何好处。

唯一有意义的构造函数是

```
Person(const std::string& name, int age) : name(name), age(age) {}
```

需要两个参数来创建一个 Person 对象；实例变量用这些值初始化，从而避免使用（可能）毫无意义的默认值。

类不变量要求每个数据成员都具有有意义的数据，默认构造函数经常违反这一原则。

创建对象数组时必须使用默认构造函数。在这种情况下创建数组，并通过调用默认构造函数初始化每个元素。除了极少数情况外，每个元素都会修改类的不变量。

**解决**

创建对象时，初始化所有实例变量对于保持正确性至关重要。少数情况下，默认构造函数可能是一种合理的方法。由于元素的初始化数据尚未确定，必须调用默认构造函数来初始化实例变量。有时，可以使用数组，但会出现问题。代价是效率略有提高，而不正确。数组通常分配在堆栈上（除非使用 new 关键字创建）。并且，教科书隐含地权威低位，使它们在学生心中占据首要地位。当有

选择时，先学到的东西通常会首先使用。所以，在几乎所有情况下，都应该首先教授 vector，并且优先使用 vector 而非数组。

vector 使用一个间接层（指针）来访问数据，数组在访问过程中会稍微快一些。但开发人员必须能够证明，这种微小的增益对正确性的影响处于合理的范围。数组非常典型也经常使用，但其缺点很大。更好的方法是选择一个不进行默认构造的容器。

清单 7.15 展示了一个使用数组和可比较 vector 的简单示例。数组使用默认构造函数，导致混乱。vector 需要为每个元素使用双参数构造函数，以保证正确（非默认）初始化；vector 要求元素可复制。现代 C++ 可以在创建数组时，使用初始化列表来缓解默认构造函数，但前现代 C++ 就没有这么幸运了。

清单 *7.15* 优先使用 *vector* 而非数组

```
class Person {
  private:
  std::string name;
  int age;
public:
  Person() : name(""), age(0) {}
  Person(const std::string& name, int age) : name(name), age(age) {}
  std::string& getName() { return name; }
  int getAge() { return age; }
};

int main() {
  Person people[2]; // 1
  for (int i = 0; i < 2; ++i) // 2
    std::cout << people[i].getName() << " is " << people[i].getAge() << '\n';

    std::vector<Person> peeps;
  peeps.push_back(Person("Susan", 21));
  peeps.push_back(Person("Jason", 25)); // 3

  for (int i = 0; i < peeps.size(); ++i) // 4
    std::cout << peeps[i].getName() << " is " << peeps[i].getAge() << '\n';
}
```

注释 1：调用默认构造函数，会让每个元素错误地初始化
注释 2：元素的数量可能会改变，但这个值可能会被忽略
注释 3：向 vector 中添加任意数量的元素，并跟踪其数量
注释 4：添加或删除元素后，此循环总能正常工作

**建议**

- 不要仅因为默认构造函数很熟悉或很常见就添加它们，它们也可能很危险。
- 请仔细考虑是否要添加默认构造函数，除非每个实例变量都正确初始化。
- 优先使用 vector 或其他容器而非数组，因为数组需要使用默认构造函数来初始化每个元素。

- 切记类不变量和默认数据成员的含义——如果默认构造对象是有意义的，那么就这样做；如果没有，请避免使用默认构造函数。

## 7.9. 错误 50： 未能保持”is-a” 关系

可读性（或缺乏可读性）是本次讨论的主要动机，但继承关系不当会对正确性产生不利影响，并影响有效性。

公有继承提供了从现有类创建新类的能力。这些新派生的类必须通过“is-a”关系与基类相关。受保护继承和私有继承本质上是不同的，本错误中不对其进行讨论。当函数具有基类实例的参数（或指向或引用）时，派生类对象将按预期工作。此属性是 Liskov 替换原则。继承向读者传达信息；因此，为了传达正确且有意义的信息，公有派生类实例必须可以替代基类实例，并且表现得像基类实例。

**问题**

正如其他地方提到的，一些教科书仍然认为继承是一种共享代码的巧妙方法，可以为开发人员节省一些时间。这种方法听起来像是一种编写更少代码，并获得更多好处的有效方法。然而，这个建议会把粗心的开发人员逼入绝境。

公共继承意味着基类是一个通用的概念或概念，可以通过多种方式进行特化。例如，一个描述二维多边形的 Shape 类。此外，程序需要操作 Circle、Square 和 Triangle 对象。由于所有这些都与 Shapes 相关并且具有典型行为（例如，area 和 perimeter），因此将其建模为 Shape 类的派生类很有意义。这样做之前，必须考虑是否存在 Circle、Square 或 Triangle 对象无法满足“is-a”关系的情况。这种情况下，答案是否定的，但有些情况远没有那么清晰或设计得那么干净。

相反，考虑这样一种情况：我们有一个 Bird 类，并希望从中派生出一个 Ostrich 类，因为鸵鸟肯定是鸟类。这种继承似乎满足了 is-a 属性，但这里有一个问题。我们直觉地认为鸟类是会飞的动物，但鸵鸟不能飞。要解决的问题决定了类之间的适当关系；有时，必须最小化或忽略自然关系。

回到 Shape 示例，继承（尤其是公共继承）最令人信服的论据是，程序通常必须以通用方式处理相关对象的集合。基类接口函数指定通用行为——为什么它是基类。派生类可以重写这些函数以提供特有的行为。如果问题不需要处理相关对象的集合，则可能不需要继承层次结构。

**分析**

开发人员尝试使用继承来关联几何形状，通用行为处理各种类似乎合理。因此，他们认为将使用具有通用功能的基类，而派生类可以根据实例的实际类型特化行为。由于矩形和正方形非常相似，因此从矩形派生正方形，并强制其高度和宽度之间建立更严格的关系是一个好主意。

Square 类是为了共享来自 Rectangle 类的代码而开发的，这保证了有效性，即使可读性略有下降。但共享代码的愿望（可以理解为节省一些击键次数的愿望），很快就在现实的岩石上搁浅。Square 似乎是 Rectangle 的一个特例，直到人们意识到，高度和宽度不能独立变化的矩形并不是矩形——只是具有高度和宽度，而这并没有定义一个真正的矩形。虽然矩形的高度和宽度可以具有相同的值，但这种罕见性并不能证明将 Square 设计为 Rectangle 的一个特例是合

理的。矩形必须保持具有独立高度和宽度属性的能力。这种独立性是 Rectangle 类对每个实例不变量的一部分。

清单 *7.16* 不正确的继承，似乎节省了很多编码工作

```
class Rectangle {
private:
  double height;
  double width;
public:
  Rectangle(double h, double w) : height(h), width(w) {}
  double getHeight() { return height; }
  void setHeight(double h) { height = h; }
  double getWidth() { return width; }
  void setWidth(double w) { width = w; }
  virtual void validate() { assert(height >= 0 && width >= 0); }
};

class Square : public Rectangle { // 1
public:
  Square(double s) : Rectangle(s, s) {}
  void validate() override {
    Rectangle::validate();
    assert(getHeight() == getWidth());
  }
};

int main() {
  std::vector<Rectangle*> shapes { new Rectangle(3, 4),
      new Square(2) };
  for (auto shape : shapes) {
    // guarantee different lengths
    shape->setHeight(shape->getWidth() + 1); // 2
    shape->validate();
  }
  return 0;
}
```

注释 1：尝试保存一些击键操作会导致出现此派生

注释 2：Square 实例将具有不同的高度和宽度值！

虽然此代码正常运行期间不会出现问题，但 validate 函数暴露了问题。Square 实例必须具有相等的高度和宽度（根据定义），但 Rectangle 实例必须可以自由地具有不同的大小。当测试类不变量时，问题就暴露出来了—Square 不是 Rectangle（反之亦然）。

**解决**

清单 7.17 展示了继承的良好用法，每个派生类在需要时都可以像 Shape（主函数中的循环）一样工作。每个派生类都通过继承与其他派生类相关，但代表不同的想法。每个类都可以验证其类不变量，但都不会偏离基类的接口。

这种方法不是从 Rectangle 开始，而是从一个抽象基类开始（理想情况下，每个函数都是纯函数；至少有一个必须是纯函数），该基类定义所有派生类的行为。然后，矩形和正方形没有从基类派生的“is-a”关系。Rectangle 和 Square 类是兄弟关系，而非父子关系。每个类都可以强制执行其对边长的约束，而不会影响其他类。想象一下将 Triangle 类添加到这个组合中，将具有更多不同的约束。将类添加到正确排序的继承层次结构，不会导致令人头疼和黑客式的解决方法。

清单 *7.17* 将要点抽象为基类

```
class Shape { // 1
public:
  virtual double area() = 0;
  virtual double perimeter() = 0;
  virtual void validate() = 0;
};

class Rectangle : public Shape { // 2
private:
  double height;
  double width;
public:
  Rectangle(double h, double w) : height(h), width(w) {}
  double area() override { return height * width; }
  double perimeter() override { return (height + width) * 2; }
  void validate() override { assert(height >= 0 && width >= 0); }
};

class Square : public Shape { // 3
private:
  double side;
public:
  Square(double s) : side(s) {}
  double area() override { return side * side; }
  double perimeter() override { return side * 4; }
  void validate() override { assert(side >= 0); }
};

int main() {
  std::vector<Shape*> shapes { new Square(2), new Rectangle(3, 4) };
  for (auto shape : shapes)
    hape->validate();
  return 0;
}
```

注释 1：抽象基类仅定义行为

注释 2：此类以其特定的方式实现行为

注释 3：此类以其惯用的方式实现行为，与其他类不同

节省一些击键次数来共享代码是一种危险的诱惑，但最终却带来了灾难。只有当派生类是基类，

因此在所有情况下可以替代基类时，才能适当地维护公共继承关系。在许多实际层次结构中，一些代码在派生类和基类之间共享，从而导致其重用，从而提高效率。请记住，代码重用（包括节省击键次数）是继承的优点，而非使用继承的原因。

**建议**

- 代码共享（重用）是公共继承的一个好处，但绝不是其原因。
- 仅当派生类在每种情况下都可以正确替代基类实例，并且为每个基类行为都具有有意义的定义函数时，才使用公共继承。
- 如果现实世界中的行为不需要在基类中建模，请避免实现该行为的诱惑，即使直觉表明应该对其进行建模。删除所有当前不需要的行为—这是 you ain’t gonna need it（YAGNI）原则的实际应用。
- 类应该通过继承来关联，它们是相关实体，并且用于集合中，其中每个实例都像基类对象一样运行，也许具有其行为的一些特化。

# 第 8 章 保护类的不变量

本章内容

- 如何确保在程序设计中维护类的不变量
- 由于不加区别地使用旧的面向对象设计建议而产生的困难
- 复制构造函数和复制赋值操作符之间的区别
- 无法初始化内置类型变量的问题

在完成第 7 章中讨论的建立类不变量后，开发人员必须小心对其进行维护。程序开发的几个方面提供了破坏不变量的机会。重要的是要意识到其中一些可能性，并保持警惕以避免使用。

每当使用数据初始化或修改状态变量时，都有可能破坏类的不变性。C++ 构造的灵活性和细粒度为修改数据提供了充足的机会，有时甚至会以令人惊讶的方式发生。最明显破坏不变性的地方是在数据状态发生变化时，但这时问题也会最明显。对象的构造和删除在某种程度上是一个隐蔽的过程，但在确定对象的内容（即不变性）方面起着至关重要的作用。

继承提供了更多滥用数据和影响不变量的机会。C++ 对对象的创建、每个部分的构造顺序，以及在构造（或销毁）的哪个阶段，可以使用哪些状态数据有着严格的规定。使用错误可能会带来灾难性的后果，数据复制函数提供了一些弄乱类实例的机会。不当使用可能很难发现，但影响可能巨大。复制有两个方面：浅层和深层；不当使用会潜移默化地威胁类的不变量。

使用数组和其他数据容器取决于复制语义。正确使用可确保对象保持原始状态，并保留对象数据的含义。正确使用虚函数对于多态性至关重要。此外，正确使用对于正确形成对象至关重要。使用这些函数时，可能犯的错误各不相同，每次错误使用都会对正确的操作和/或状态产生不利影响。

## 8.1. 保护类的不变量

保持类不变需要注意细节，并了解构造函数和虚函数之间的一些联系。C++ 语言的灵活性为开发人员提供了自由，来设计特定行为，但必须充分理解，并需要正确使用。每当初始化或修改状态变量时，都存在损坏的可能，需要仔细注意一些较模糊或较脆弱的区域以避免这些错误。

技术上讲，对象构造属于类不变量的建立。继承是一种强大的技术，可以构建相关类共享公共接口的行为。派生对象的构造，可能会通过错误地使用虚函数，或错误地在数组中使用派生对象，来影响已经建立的不变量。如果操作错误，会使类不变量无效。

## 8.2. 错误 51： 编写非必要的访问器方法

这个错误主要针对正确性、可读性和有效性。不必要的访问器打破了类和用户之间的隔离。这些不必要的访问器需要花时间编写，但真正维护起来却很少；其会带来额外的认知负担，而且没有任何优点。

在面向对象编程的时代，许多教科书和作者都提倡为每个私有实例变量提供访问器和修改器。流行的说法是，每个用户都应该能够访问实例内的每个状态。这种代码淡化了更重要的方法，使阅读和记住类变得更加困难。遵循这一建议的代码，充斥着返回实例变量值副本的访问器。这种方法打破了公共/私有封装，过度的暴露了实现细节。

### 问题

这种方法在很多方面都存在语义问题。如果变量存在非必要访问器，则变量和类实现可能会危险地暴露给用户。使用简单的访问器和修改器，公共变量和私有变量之间没有区别。泄露状态变量的实现细节表明，用户代码应该负责理解和管理它们，这个缺陷直接违反了维护类不变性的规定。修改类实现时，正确性和有效性会受到影响，因为用户代码可能依赖于已公开类的行为方式。

访问器的另一个不好的做法是，使用实例变量来存储从其他实例变量派生的对象状态。假设 `Circle` 类中计算面积，如清单 8.1 所示。遗留代码示例定义了一个名为 `area` 的浮点变量，该变量在构造函数中初始化。编写了一个简单的访问器，来返回 `area` 变量的计算值。

清单 *8.1* 带有计算值和简单访问器的类

```
class Circle {
private:
  const static double PI;
  double radius;
  double area;
public:
  Circle() : radius(1), area(PI) {}
  Circle(double radius) : radius(radius),
    area(radius*radius*PI) {} // 1
  double getArea() const { return area; } // 2
  double getRadius() const { return radius; }
};
const double Circle::PI = 3.1415927;

int main() {
  Circle c1(3);
  std::cout << c1.getArea() << '\n';
  return 0;
}
```

注释 1：需要计算的值

注释 2：一个简单的访问器，只访问计算值

非必需访问器会导致代码膨胀。实例由其构造函数初始化后，部分或大部分实例变量可能对用户而言没必要了解的。如上所述，这些访问器会让用能依赖于变量的实现细节，从而阻止简单的重新实现。另一个结果是用户可能承担理解和维护实例状态的责任。这种情况违反了类不变性，并将类的封装传播到类外。

### 分析

正确编写的类仅显示必要的状态信息，而不一定以类中使用的形式显示。公共接口定义了用户如何访问该信息，但没有定义该信息的实现方式、是缓存还是直接计算，或其他实现细节。用户不得依赖实现细节，类编写者有责任防止这种情况发生。

`Circle` 类使用传入的半径信息构造一个实例。访问器为用户提供了对 `radius` 值，及其实现的无限制可视化。`area` 方法依赖于预先计算的实例变量；其访问器仅返回该值。

应避免随意编写 `radius` 访问器；如果用户需要此信息，则应提供。需要使用用户代码验证

该值，以查看是否需要；如果是，则提供访问器。由于似乎只有圆的面积才有意义，因此计算该值很重要，而半径则不重要。但确实需要存在，并且具有所有假定的优点。在添加半径访问器之前，应确定用户要求。用户已经提供了半径；如果是必需的，用户应该记住它。

第二个问题是面积的自动计算。用户可能需要这些信息，因此预先计算是一个好主意。但作为一种习惯，预先计算应该推迟到第一次（或每次）使用时。如果类更复杂，则不确定是否需要该面积—类设计者应该按照意图工作，并完成工作。

**解决**

应将非必要的访问器删除。用户可能需要这些值，但不应将其作为“前提”来使用。扫描代码库，可以看到现有代码中是否使用了访问器。如果没有，则将假设需要值全部删除。如果使用了访问器，则确定用户代码是否因使用它而超越了其职责。示例代码中，如果用户是需要绘制和排列圆形的图形程序，则半径将是必需的。如果用户正在计算圆形铺路石的重量，则半径将无关紧要；只需计算面积。

清单 *8.2* 带有计算访问器的类

```
class Circle {
private:
  const static double PI;
  double radius;
public:
  Circle() : radius(1) {}
  Circle(double radius) : radius(radius) {}
  double getArea() const { return radius*radius*PI; } // 1
};
const double Circle::PI = 3.1415927;
```

注释 1：仅在需要时计算值

开发人员在编写计算访问器时必须采取平衡的方法，考虑计算成本。计算操作通常成本低廉，可按需计算。假设计算成本高昂（例如，使用大量数据、访问数据库或在线通信），则应尽可能少地计算其结果。这种情况下，更好的方法是仅在第一次访问时计算结果，然后缓存结果。后续访问将快速访问缓存的值，从而减轻计算成本。

由于大多数访问器不会改变实例变量，最好将其标记为 `const`。编译器将强制承诺不会修改任何值；此外，这也是一种意图的良好文档式方案。

**建议**

- 不要编写访问器，除非对于用户使用必不可少；必不可少的访问器是必需的。
- 不要维护可以计算其值的实例变量，除非这是计算量很大的操作。
- 出于性能原因，应该缓存昂贵的计算结果。
- 将访问器标记为 `const` 以传达其不变性。

## 8.3. 错误 52： 提供脆弱的修改器

这个错误过于注重正确性而忽视了有效性，许多编程教科书延续了一种古老而熟悉的反模式。解释如何构建类时，作者经常建议为每个实例变量提供一个访问器和修改器。面向对象编程的早期，这认为是一种很好的做法。当前的教科书作者倾向于将这种做法延续。然而，这种做法往往会使类不变量失效；所以，我们需要一种更好的方法。

### 问题

让我们考虑具有单个实例变量 `radius` 的 `Circle` 类，如以下清单所示。构造函数初始化变量，访问器返回其副本，而修改器修改它。

清单 *8.3* 非验证的可修改类

```
1  class Circle {
2  private:
3    double radius;
4  public:
5    Circle() : radius(1) {}
6    Circle(double radius) : radius(radius) {}
7    double getRadius() { return radius; }
8    void setRadius(double radius) { this->radius = radius; } // 1
9  };
10
11 int main() {
12   Circle c1(2);
13   std::cout << c1.getRadius() << '\n';
14   c1.setRadius(-1);
15   std::cout << c1.getRadius() << '\n';
16 }
```

注释 1：由于没有验证检查，值的范围不受限制

根据教科书的方法，这是正确的设计；但存在三个问题：

- 未经验证的输入值
- 坚持所有实例变量可变
- 设置器和构造函数参数间的重复

遗留代码中充满了这种模式，了解这些问题将有助于确定何时进行代码改进。

脆弱的设置器允许对实例变量进行无限制的修改，从而显著影响正确性。用户不应负责了解变量的正确值范围。类必须保持类的这一部分不变，并提醒用户注意不正确的值。引入无关的方法可能会影响可读性，从而掩盖重要的方法。使实例变量可变会影响正确性，某些类在构造后不应更改其实例变量。

坚持要求开发人员编写不必要的方法，并引入破坏类不变性的代码会影响有效性。有效值范围的知识通常在构造函数和变量之间重复，很容易出现分歧。

有效的编码试图在一个地方消除和隔离这些重复。

### 分析

虽然脆弱的访问器会分散注意力但相对无害，但脆弱的修改器可能很危险。修改器负责维护构造函数建立的类不变量。任何错误的值引入类都会导致实例不一致，并可能导致未定义行为。所以，修改器必须验证输入值的正确性。

此外，实例变量可能不需要修改。虽然我们可能倾向于改变实例变量，但这条路比看上去更曲折。在编写变量之前，必须采取两个基本步骤。首先，确定实例变量是否应可变。其次，确定合法值的范围。

许多情况下，第一个问题的答案是否定的。如果是这样，就没有理由编写一个变量设置器；这样的变量器是危险的和多余的。开发人员应该确定半径在 `Circle` 类中是否正确可变。这种情况下，不太可能；如果需要具有不同半径的圆，用户应该创建一个新实例。

对于剩余的变量，请仔细考虑变量的适当值范围。在将参数值分配给实例变量之前，编写验证代码以确保满足这些界限。考虑如何解决超出范围的参数值问题。使值无效，这通常是最好的方法。另一种选择是忽略无效参数值，而不执行任何操作；但当构造函数调用变量来设置初始值时，这种方法不起作用。

### 解决

以下代码展示了一种更好，但并不理想的方法。它试图通过确保构造函数和变量在用户传递无效值时，抛出异常来解决验证问题。没有解决可变性或重复问题。

清单 *8.4* 具有重复验证的类

```
class Circle {
private:
  double radius;
public:
  Circle() : radius(1) {}
  Circle(double radius) {
    if (radius < 0) // 1
      throw std::invalid_argument("radius is negative");
    this->radius = radius;
  }
  double getRadius() const { return radius; }
  void setRadius(double radius) {
    if (radius < 0) // 1
      throw std::invalid_argument("radius is negative");
    this->radius = radius;
  }
};
int main() {
  Circle c1(2);
  std::cout << c1.getRadius() << '\n';
  c1.setRadius(-1);
  std::cout << c1.getRadius() << '\n';
}
```

注释 1：重复确定有效值

通常，构造函数和修改器有共同的代码。构造函数负责建立类不变量，包括初始化实例变量（当前未初始化）的值。修改器负责修改实例变量（现在已初始化）的值。可以通过让构造函数调用修改器来消除这种重复，都必须保持类不变量，因此知识重复是不可避免的。误读“不要重复自己”（DRY）原则可能会导致人们仅关注代码重复；开发人员应该关注知识重复。范围检查代码应放在修改器中，构造函数应调用修改器。

以下代码解决了前面讨论的三个问题（未验证的参数值、不加区分的可变性和重复知识），并以最小的努力解决了每个问题。

清单 *8.5* 一个具有验证和单一知识源的类

```
class Circle {
private:
  double radius;
  static double validateRadius(double radius) { // 1
    if (radius < 0)
      throw std::invalid_argument("radius is negative");
    return radius;
  }
public:
  Circle() : radius(1) {}
  Circle(double radius) : radius(validateRadius(
    radius)) {} // 2
  double getRadius() const { return radius; }
  void setRadius(double r) { radius =
  validateRadius(r); } // 3
};

int main() {
  Circle c1(2);
  std::cout << c1.getRadius() << '\n';
  Circle c2(-1); // 4
  std::cout << c2.getRadius() << '\n';
}
```

注释 1：验证知识的单一来源
注释 2：代码依赖于知识来源
注释 3：代码依赖于知识来源；如果需要不变性，请删除此方法
注释 4：这会引发异常，提醒开发人员注意问题

私有的 `validateRadius` 方法隔离了对适当半径值的知识，构造函数通过引用此方法初始化实例变量，并实例化具有不同半径的圆。每个不可变的实例变量必须没有变量；消除现有的变量。完成此步骤是一种练习，可以摆脱思维混乱，并让代码免除不必要的维护成本。

**建议**

- 使尽可能多的实例变量不可变，并消除变量。
- 验证每个输入参数，以确保其值在类不变量定义的范围内。
- 如果可能，抛出无效值的异常；否则，忽略无效值。让构造函数和变量调用一个通用的验证方

法，将知识隔离到一个地方。

- 确保每个实例变量都初始化为有意义的值。

## 8.4. 错误 53： 过度使用受保护的实例变量

这个错误主要集中在正确性上。基类建立并维护了类不变性，但容易受到派生类所做的更改的影响。派生类有义务遵守基类不变性。可读性无疑得到了改善，有效性也可以得到改善。

作为最佳实践，所有实例变量都应为私有变量。这种方法可确保，没有外部代码可以在没有类严格控制的情况下，访问或修改它们。引入继承后，派生类会发现需要的基类信息无法访问。与用户代码一样，派生类必须使用基类访问器和修改器，但这种限制可能感觉过于严格。C++ 为基类提供了 `protected` 关键字，以允许派生类直接访问这些实例变量 - 外部代码仍然被禁止访问或修改这些变量。虽然这种放宽看起来不错，但也引入了违反类不变量的可能性。

### 问题

基类通过严格控制每个数据成员的值，来维护其状态。构造函数和变量器应设计为初始化类不变量，将值保持在其适当的边界，并避免越界突变的企图。如果基类已将这些变量设为受保护，则派生类可以直接访问这些变量。

派生类应保持其类不变量，并尽力遵守其基类不变量。不幸的是，编译器无法保护不变量；开发者必须付出努力。此外，为了使派生类正确维护基类不变量，开发人员必须掌握如何正确处理基类限制的外部知识。这些知识已存在于基类中，不应在派生类中重复。

清单 *8.6* 受保护实例变量的漏洞

```
class Person {
protected:
  std::string name;
  int age; // 1
public:
  Person(const std::string& name, int age) : name(name) {
    if (age < 0) // 2
      throw std::invalid_argument("negative age");
    this->age = age;
  }
  std::string getName() const { return name; }
  int getAge() const { return age; }
};
class Student : public Person {
private:
  double gpa;
public:
  Student(const std::string& name, int age, double gpa) : Person
  (name, age), gpa(gpa) {}
  double getGpa() const { return gpa; }
  void setAge(int age) { this->age = age; } // 3
};
```

```
23
24 int main() {
25   Student jane("Jane", 26, 3.85);
26   jane.setAge(-26);
27   std::cout << "Jane is " << jane.getAge() << " years old\n";
28   return 0;
29 }
```

注释 1：易受攻击的实例变量
注释 2：经过适当验证，并保持类不变量
注释 3：危及基类不变量的权宜之计

### 分析

对于不理解或未充分维护基类不变量的派生类，一个或多个受保护的基类变量可能会强制进入超出范围的状态，且不会发出警告，使拒绝无效值变得不可能。Student 的 setAge 方法不限制参数值，而是直接设置 age 实例变量；发生这种情况是因为该变量是受保护的，而非私有。

Student 类的设计者认为使用 Person 基类的受保护变量是有利的。Student 开发者决定添加一个 setAge 方法，其知道一些较新的学校规则会根据年龄而有所不同，并且学生可能会在学年期间过生日。

Person 类正确验证了年龄输入值并拒绝了无效值。派生类需要充分了解添加 setAge 方法的含义，但未能做到这一点。由于基类构造函数验证了 age，因此一切最初看起来都很好。但基类不会改变 age 实例变量，需要一种快捷方式来直接改变变量。为了在 Student 中正确实现 setAge 方法，派生类必须复制有关 age 基类变量。这种知识重复违反了 DRY 原则，并且给变量的封装带来了压力。

### 解决

Person 基类负责维护其类的不变性。部分责任是通过拒绝其他类，在没有某些验证逻辑保护的情况下，直接访问其实例变量来处理的，如下面的清单所示。

清单 8.7 通过将实例变量设置为私有来增强安全性

```
1  class Person {
2  private:
3    std::string name;
4    int age; // 1
5  public:
6    Person(std::string name, int age) : name(name) { setAge(age); }
7    std::string getName() const { return name; }
8    int getAge() const { return age; }
9    void setAge(int age) {
10     if (age < 0) // 2
11       throw std::invalid_argument("negative age");
12     this->age = age;
13   }
14 };
15
```

```cpp
class Student : public Person {
private:
  double gpa;
public:
  Student(std::string name, int age, double gpa) :
    Person(name, age), gpa(gpa) {} // 3
  double getGpa() const { return gpa; }
};

int main() {
  Student jane("Jane", 26, 3.85);
  jane.setAge(-26);
  std::cout << "Jane is " << jane.getAge() << " years old\n";
  return 0;
}
```

注释 1：不易受到派生类突变的影响

注释 2：经过适当验证，并保持类不变

注释 3：必须使用经过适当验证的基类方法

基类正确地验证了年龄值，该值从构造函数、派生类和用户代码中调用。派生类不需要知道有关有效年龄的信息，它将保证年龄正确这一责任委托给基类。

这次讨论并不意味着永远不使用受保护的变量；有些情况下它们已经存在，无法改变这一事实。但可以（并且必须）尊重基类不变性并使用其变量。如果基类无法更改且没有验证变量，请在派生类中编写一个并注释原因。记住里氏替换原则（LSP）；如果基类变量未抛出而派生类变量抛出，则不满足替换要求。

如果基类不打算向用户代码公开其某些实例变量，则可以为派生类提供受保护的方法，以供其进行变异。这些方法确保基类能够控制其状态的任何更改，同时为派生类提供一个接口，以用户代码无法实现的方式改变状态。

**建议**

- 消除尽可能多的受保护变量。
- 始终让变量器验证其参数。
- 了解派生类使用访问器和修改器的轻微尴尬，完全可以通过基类保持其类不变性来抵消。
- 考虑使用受保护的访问器和变量设置器，来提供用户代码无权使用的访问权限。
- 不要将变量设为虚（从基类继承）；它们只能在所属类中实现。

## 8.5. 错误 54： 混淆 operator= 和复制构造函数

这个错误会影响正确性和可读性，编码不当会对类不变量和代码的适用性产生不利影响。

经常会将信息从一个类的实例复制到另一个类的实例。此操作的常用方法是使用复制赋值操作符，但也必须考虑复制构造函数。如果误解了两者的语义，很容易认为两者在做同样的事情，而其中一个是多余的。然而，虽然操作非常相似，但含义却不同。

**问题**

创建类时，必须考虑赋值和从现有对象复制的语义。如果开发者既没有编写复制构造函数，也没有编写复制赋值操作符，则编译器默认提供这两者。如果提供了其中之一，则另一个自动生成。

两者的默认实现都是对每个实例变量，从源到目标对象进行成员级复制。某些情况下，这已经足够了，而且是正确的。但是，开发人员仍然必须仔细考虑其中的含义。忽略它们并轻率地让编译器完成繁重的工作是一种糟糕的方法。

使用动态资源（如内存、句柄、连接、套接字等）时，应使用编译器默认方法的替代方法。开发人员必须确保在复制操作期间，正确处理动态资源。

当创建新对象且当前未初始化时（无论是基于堆栈还是基于堆），将调用复制构造函数。当使用源对象值覆盖现有的已初始化目标对象时，将调用复制赋值操作符。区别在于目标对象是未初始化的，还是已初始化的——两者的含义略有不同。

清单 8.8 中的代码使用动态资源来处理读取文件（省略大部分代码），并使用现有的文件读取器作为初始化源。编译器提供了其默认版本，因为开发人员没有提供复制构造函数或复制赋值操作符。每个版本都使用浅复制语义，将值从源实例变量复制到目标实例变量。

清单 *8.8* 由默认复制构造函数和复制赋值操作符处理的资源

```
class InputReader {
private:
  std::string* filenames; // 1
  int count;
public:
  InputReader() : filenames(new std::string[5]), count(0) {}
  ~InputReader() { delete[] filenames; }
  void addFileName(const std::string& n) {
    if (count == 5)
      throw std::runtime_error("too many files specified");
    filenames[count++] = n;
  }
  int getCount() const { return count; }
  const std::string& operator[](int index) {
    return filenames[index];
  }
};
int main() {
  InputReader ir1;
  ir1.addFileName("statistics.txt");
  InputReader ir2(ir1); // 2
  ir2.addFileName("numbers.txt");
  InputReader ir3 = ir2; // 3
  ir3.addFileName("categories.txt");
  ir1.addFileName("questions.txt");

  for (int i = 0; i < ir1.getCount(); ++i)
    std::cout << ir1[i] << '\n';
```

```
29    for (int i = 0; i < ir2.getCount(); ++i)
30      std::cout << ir2[i] << '\n';
31    for (int i = 0; i < ir3.getCount(); ++i)
32      std::cout << ir3[i] << '\n';
33    return 0;
34  }
```

注释 1：为了简单而采用的糟糕做法；STL 提供了更好的方法

注释 2：使用浅复制语义调用编译器默认的复制构造函数

注释 3：使用浅复制语义调用编译器默认的复制赋值操作符

**分析**

第一个读取器添加 statistics.txt 文件，然后用于初始化第二个读取器；第二个读取器添加 numbers.txt 文件，然后用于初始化第三个读取器；第三个读取器添加 categories.txt 文件，预计有三个文件需要读取。第一个读取器然后添加一个名为 questions.txt 的新文件。以下输出显示事情没有按预期进行（为清晰起见添加了空行）：

```
statistics.txt
questions.txt

statistics.txt
questions.txt

statistics.txt
questions.txt
categories.txt
Segmentation fault
```

第一个读者应该是唯一可以访问 questions.txt 文件的人；但是，输出显示二级和三级的读者也可以访问 questions.txt。由于复制和赋值仅复制指针值，因此每个都指向相同的共享数据。ir2 读者已失去对 numbers.txt 文件的访问权限。当 ir1 添加另一个文件时，它认为该数组有一个有效元素。因此，当添加 questions.txt 文件时，它将成为共享数组中的第二个元素，覆盖 numbers.txt 元素。

默认复制构造函数和复制赋值操作符对 filenames 指针进行了浅拷贝，共享了数组数据。每个读取器都会独立于其他读取器更新下一个元素，从而导致数据损坏。这种情况会导致意外输出和危险的错误行为，包括重复删除。

**解决**

要使此示例正常工作，必须编写复制构造函数和复制赋值操作符，来正确处理文件名的动态数组。复制构造函数将分配一个新数组，执行现有元素的复制，并设置计数变量。

对于复制构造函数，新实例未初始化，必须在初始化列表中创建一个新的动态数组。现有值的复制在构造函数主体中完成。经过此更改后，代码可以正常工作并符合预期。复制赋值操作符必须采用不同的方法，尽管大部分功能相同。由于目标对象已初始化，已经具有其非共享动态数组，无需分配新的动态数组。只需要复制元素并初始化计数值，如以下清单所示。

清单 8.9 正确编写的复制构造函数和赋值操作符

```
class InputReader {
private:
  std::string* filenames;
  int count;
public:
  InputReader() : filenames(new std::string[5]), count(0) {}
  ~InputReader() { delete[] filenames; }
  InputReader(const InputReader& r) : filenames(new std::string[5]),
      count(r.count) { // 1
    for (int i = 0; i < r.count; ++i)
      filenames[i] = r.filenames[i];
  }
  InputReader& operator=(const InputReader& r) { // 2
    for (int i = 0; i < r.count; ++i)
      filenames[i] = r.filenames[i];
    count = r.count;
    return *this;
  }
  void addFileName(const std::string& n) {
    if (count == 5)
      throw std::runtime_error("too many files specified");
    filenames[count++] = n;
  }
  int getCount() const { return count; }
  const std::string& operator[](int index) {
  return filenames[index];
}
};
int main() {
  InputReader ir1;
  ir1.addFileName("statistics.txt");
  InputReader ir2(ir1); // 3
  ir2.addFileName("numbers.txt");
  InputReader ir3 = ir2; // 4
  ir3.addFileName("categories.txt");
  ir1.addFileName("questions.txt");

  for (int i = 0; i < ir1.getCount(); ++i)
    std::cout << ir1[i] << '\n';
  for (int i = 0; i < ir2.getCount(); ++i)
    std::cout << ir2[i] << '\n';
  for (int i = 0; i < ir3.getCount(); ++i)
    std::cout << ir3[i] << '\n';
  return 0;
}
```

注释 1：分配一个新数组并复制元素，而非指针

注释 2：复制元素，而非指针

注释 3：调用用户编写的复制构造函数

注释 4：调用用户编写的复制赋值操作符

通过这些更改，行为得到纠正。以下输出符合预期（为清晰起见添加了空行）：

```
statistics.txt
questions.txt

statistics.txt
numbers.txt

statistics.txt
numbers.txt
categories.txt
```

对于某些类，这种复制赋值操作符方法可能需要修改。清单 8.9 中的代码展示了一种情况，其中所有实例的数组大小都相同。假设有一个类，其中数组大小不同。复制构造函数的工作方式相同，会创建一个与源对象大小相同的新数组。复制赋值操作符的目标对象，将具有不同大小的数组，因此需要删除现有数组，并且需要根据源的大小和复制的每个元素创建一个新数组。请保持警惕，以正确销毁动态元素。如果元素是类实例，则需要销毁。尽管它可能会自动发生，但在极少数情况下可能需要调用析构函数。

清单 8.10 中的代码展示了解决此问题的变体，其中假设类添加了一个表示最大数组大小的容量实例变量（这是比硬编码大小更好的设计）。此示例有一个潜在的问题；如果进行自赋值，此代码将失败。

清单 8.10 处理不同大小的源数组和目标数组长度

```
InputReader& operator=(const InputReader& r) {
  if (capacity != r.capacity) {
    delete[] filenames;
    filenames = new std::string[r.capacity];
    capacity = r.capacity;
  }
  for (int i = 0; i < r.capacity; ++i)
    filenames[i] = r.filenames[i];
  count = r.count;
  return *this;
}
```

现代 C++ 修改了默认成员的生成方式。如果仅提供了移动构造函数，则编译器将不会自动生成复制构造函数、复制赋值操作符或移动赋值操作符。如果仅提供了移动赋值操作符，则编译器将不会自动生成复制构造函数、复制赋值操作符或移动构造函数。期望编译器填充缺失的成员时，请务必了解这些限制。以下代码是对清单 8.9 中代码的补充，演示了移动构造函数和移动赋值操作符：

```
InputReader(InputReader&& r) : filenames(r.filenames), count(r.count) {
  r.filenames = nullptr;
  r.count = 0;
```

```
}

InputReader& operator=(InputReader&& r) {
  if (this != &r) {
    delete[] filenames;
    filenames = r.filenames;
    count = r.count;
    r.filenames = nullptr;
    r.count = 0;
  }
  return *this;
}
```

**建议**

- 仔细考虑复制赋值操作符和复制构造函数的机制，并决定逐个成员复制是否足够。
- 记住处理自我赋值；如果忽略，可能会出现未定义的行为。
- 复制构造函数针对未初始化的内存，而复制赋值操作符针对已初始化的内存。
- 仔细考虑如何在初始化期间，正确处理动态资源。

## 8.6. 错误 55： 误解浅复制与深复制

此错误主要针对正确性，其他特性不受其影响。典型的操作是将一个变量的值赋给另一个变量，或将一个对象赋给同一类型的另一个对象。此操作的代码看起来很简单，但除非理解清楚，否则会导致错误行为。

当一个对象复制或赋值给另一个对象时，每个数据成员都会收到一个覆盖其现有值的值。默认复制构造函数和复制赋值操作符实现从源到目标的逐个成员复制。如果成员是指针，则源对象和目标对象将指向同一内存。虽然指针分享可能是人们所期望的行为，但通常并非如此，而且会导致严重的问题。

**问题**

复制或赋值之前，源对象和目标对象中已经建立了类不变量。复制或赋值之后，源对象和目标对象应该与不变量保持一致。但如果需要深度复制语义，情况可能并非如此。

有两种情况需要考虑：指针和独占资源。错误 54 解释了第一种情况，并推荐了确保不变量保持正确建立的技术。本讨论涵盖了独占资源的情况。

考虑使用通信通道传输消息的情况。每个通道都连接到不同的终端，可能是付费信息的订阅者。开发人员意识到，由于指针用于消息，因此应注意确保动态资源不被共享。现有消息得到正确处理，但需要解决独占资源的使用问题。

独占资源通常用作一组有限实体的值。例如，一个办公室可能有三台共享打印机。每台打印机都是独占资源，因为只有三个实例存在，每次只能由一个用户使用。非动态分配的资源可能看起来只是“另一个”值。默认情况下，这些值会毫无保留地复制，但必须谨慎管理独占资源。当独占资源从一个实例复制到另一个实例时，不能再存在或使用——资源已转移。

举一个不同的例子，假设开发人员想要将现有消息从一个通道（假设在终端上出现某种错误情况）传输到另一个通道并更改连接的终端。这本质上是一个移动操作；因此，他们实现了复制赋值操作符并谨慎管理动态消息。但是，当通道尝试处理 `Messages` 时，它会因终端错误而失败，如以下清单所示。

清单 *8.11* 注意动态资源，而非独占资源

```
int createSocket() {
  // assume valid implementation; this is for an example only!
  static int sock_num = 0;
  return ++sock_num;
}

class Message {
  private:
  std::string payload;
  public:
  Message(const std::string& msg) : payload(msg) {}
  const std::string& getMessage() const { return payload; }
};

class Channel {
private:
  std::queue<Message*> messages;
  int socket;
public:
  Channel(int sock) : socket(sock) {}
  void setMessage(std::string msg) { messages.push(new Message(msg)); }
  int getSocket() const { return socket; }
  std::string getMessage() {
    std::string msg = messages.front()->getMessage();
    messages.pop();
    return msg;
  }
  Channel& operator=(Channel& o) {
    for (int i = 0; i < messages.size(); ++i)
      messages.pop();
    for (int i = 0; i < o.messages.size(); ++i) {
      messages.push(o.messages.front());
      o.messages.pop();
    }
    socket = o.socket; // 1
    return *this;
  }
};

int main() {
  Channel one(createSocket());
```

```
  Channel two(createSocket());
  two = one;
  std::cout << one.getSocket() << ' ' << two.getSocket() << '\n';
}
```

注释 1：现在，套接字在源和目标之间共享

### 分析

源队列中的指针使用元素的消息，来构造新的 `Message` 实例（设计不佳；这只是一个示例）。直接传输指针会导致问题，因为对 `pop` 的调用会从源队列中删除元素，并调用该元素的析构函数—指针将引用已删除的内存。套接字资源按原样复制，留下两个对象共享（都不拥有）该资源。由于套接字不是指针，很容易忽略这一事实，并直接复制。

复制操作将套接字值从源复制到目标，但源对象不得保留资源的副本。源对象意外使用资源可能会导致系统出现未定义、不正确或意外的行为；但假定套接字代表失败的终端。在这种情况下，复制可能会使目标留下一个无法操作的套接字，从而使分配本应解决的问题长期存在。

独占资源需要了解其复制行为。资源独占意味着一次只能有一个对象拥有，将资源从一个对象转移到另一个对象必须失去对该资源的引用。这种情况下，可能必须重新创建独占资源。一定要了解正在使用哪种方法，并进行适当地使用。

### 解决

最直接的方法是将独占资源变量设置为无效值或默认值，具体取决于数据成员的类不变量。指针应清零（具体方法取决于 C++ 版本），值类型变量应具有明确且可检测的零值，意味着没有分配资源。虽然变化很小，但是影响很大。

清单 *8.12* 传输动态和独占资源

```
int createSocket() {
  // assume valid implementation; this is for an example only!
  static int sock_num = 0;
  return ++sock_num;
}

class Message {
private:
  std::string payload;
public:
  Message(const std::string& msg) : payload(msg) {}
  const std::string& getMessage() const { return payload; }
};

class Channel {
private:
  std::queue<Message*> messages;
  int socket;
public:
  Channel(int sock) : socket(sock) {}
```

```
21    void setMessage(std::string msg) { messages.push(new Message(msg)); }
22    int getSocket() const { return socket; }
23    std::string getMessage() {
24      std::string msg = messages.front()->getMessage();
25      messages.pop();
26      return msg;
27    }
28    Channel& operator=(Channel& o) {
29      for (int i = 0; i < o.messages.size(); ++i) {
30        messages.push(new Message(o.messages.front()->getMessage()));
31      o.messages.pop();
32      }
33      socket = createSocket(); // 1
34      o.socket = 0; // 2
35      return *this;
36    }
37  };
38
39  int main() {
40    Channel one(createSocket());
41    Channel two(createSocket());
42    two = one;
43    std::cout << one.getSocket() << ' ' << two.getSocket() << '\n';
44  }
```

注释 1：终端的唯一版本

注释 2：源终端无效，无法共享，如果无效则清除

**建议**

- 确保动态资源得到适当处理，并且不会在实例之间共享。
- 确保独占资源从源移动到目标，并将源设置为其零值。NULL 是一种常见的 C 习惯用法，但其定义和资源类型之间可能会出现大小不匹配的情况。现代 C++ 有 nullptr 关键字，用于将指针分配给空值。
- 确保分配给值数据成员的零值，不会使类不变量无效；如果零是合法值，请选择另一个值来表示无效值。

## 8.7. 错误 56： 未调用基类的操作符

此错误主要针对正确性。其他特性不会受到此错误明显影响。

继承是指从现有类中定义新类，从而允许统一处理相关对象的集合。层次结构中的每个类都会向组合的派生类添加一个数据部分，派生类的对象是这些数据部分的组合。多态是将这些派生类视为基类，但行为与派生类相同的运行时能力。相应的类构造函数、析构函数和操作符，必须正确管理层次结构中每个类的各个数据部分。很容易忽视复制赋值操作符的正确性。

**问题**

复合对象（派生类实例）的每个部分，都必须正确实现每个相关操作符。构造函数和析构函数经常在此上下文中提及，如果不引用基类构造函数，就很难编写构造函数。应更频繁地提及、理解此上下文中操作符的正确操作。

考虑以下 Person 和 Student 之间的继承层次结构。Student 派生自 Person，因为学生是人，但学生又不仅仅是人。Student 类的开发人员会小心地实现代码来处理该类中的专业知识，但可能会忽略基类中的类似知识。基类中的实例变量是私有的，派生类无法访问；所以派生类无法充分复制这些值。

清单 8.13 展示了从一个实例到另一个实例的复制操作。这个例子有些牵强，但这种类型的操作经常发生在未命名的变量中，例如 vector 或数组元素，并且没有明显的警告标志，例如：将 Thelma 复制到 Louise，如下面的清单所示。

清单 *8.13* 派生类复制数据，但忽略其基类

```
class Person {
private:
  std::string name;
public:
  Person(const std::string& name) : name(name) {}
  const std::string& getName() { return name; }
};

class Student : public Person {
private:
  double gpa;
public:
  Student(const std::string& name, double gpa) : Person(name), gpa(gpa) {}
  Student& operator=(const Student& o) {
    if (this == &o)
      return *this;
    gpa = o.gpa; // 1
    return *this;
  }
double getGpa() const { return gpa; }
};

int main() {
  Student thelma("Thelma", 3.85);
  Student louise("Louise", 3.75);
  louise = thelma;
  std::cout << thelma.getName() << ' ' << thelma.getGpa() << '\n';
  std::cout << louise.getName() << ' ' << louise.getGpa() << '\n';
}
```

注释 1：Student 实例变量可正确复制，但 Person 实例变量则没有

该问题的结果从其输出中可以看出：

```
Thelma 3.85
Louise 3.85
```

### 分析

当完成创建 `louise` 对象时，它会分配给 `thelma` 对象。现在，我们有两个 `thelma` 值，由于对 `louise` 实例的赋值操作，每个值都与不同的变量名相关联。赋值不会导致编译或运行时问题，因此代码按预期工作。但请注意，`louise` 对象仍然保留其名称 `Louise`，但使用 `Thelma` 的 `gpa` —对于学生来说，这可能是最好的选择。但对于我们的编程完整性而言，事实并非如此。

派生类定义了复制赋值操作符，以确保参数的 `gpa` 复制到自身，代码按预期工作。但基类中的 `name` 字段也需要正确复制，其在 `Person` 中为私有，因此 `Student` 无法复制——这很容易忽视。只有从代码中删除复制赋值操作符定义，复制操作才能按预期工作。默认的复制赋值操作符和我们的定义不同。

编译器编写的默认复制赋值操作符，确保在对派生类的数据字段进行成员级复制之前，调用基类复制赋值操作符。这个微小但关键的事实，对于正确操作至关重要。由于派生对象是所有基类和派生类的组合，当复制或赋值组合的一部分时，必须确保每个部分都执行了复制或赋值操作。

### 解决

理解了这个问题，就可以轻松解决。清单 8.14 中更新的代码使赋值操作正确执行。复制赋值操作符确保首先调用基类版本，然后继续其复制行为。复制赋值操作符确保调用其基类复制赋值操作符，并将参数作为参数传递，以确保处理层次结构中所有更高级别的实例。对象构造顺序所确立的主题再次体现：基类始终先构造、复制或赋值，然后在派生类中执行相同的操作。

清单 *8.14* 适当考虑其基类的派生类

```
1  class Person {
2  private:
3    std::string name;
4    public:
5    Person(const std::string& name) : name(name) {}
6    const std::string& getName() { return name; }
7  };
8
9  class Student : public Person {
10 private:
11   double gpa;
12 public:
13   Student(const std::string& name, double gpa) : Person(name), gpa(gpa) {}
14   Student& operator=(const Student& o) {
15     if (this == &o)
16       return *this;
17     Person::operator=(o); // 1
18     gpa = o.gpa; // 2
19     return *this;
20   }
21   double getGpa() const { return gpa; }
```

```
};

int main() {
  Student thelma("Thelma", 3.85);
  Student louise("Louise", 3.75);
  louise = thelma;
  std::cout << thelma.getName() << ' ' << thelma.getGpa() << '\n';
  std::cout << louise.getName() << ' ' << louise.getGpa() << '\n';
}
```

注释 1：首先调用基类复制赋值操作符

注释 2：随后复制派生类实例变量

**建议**

- 如果在派生类中定义了任何操作符，请确保在执行数据移动操作之前，实现包括对等效基类操作符的调用。
- 构造函数和操作符的顺序必须遵循严格的模式，即最基类先执行，派生类最后执行，而析构函数则反转此顺序。

## 8.8. 错误 57： 未在多态基类中使用虚析构函数

此错误重点在于正确性。正确和不正确的实现之间的差异，可能不会影响可读性、有效性或性能（如果影响，则取决于编译器的实现）。

必须仔细考虑构造函数，正确初始化实例变量对于保持类不变至关重要。同样，必须仔细考虑析构函数。C++ 赋予开发人员类设计的权力和责任。必须考虑类以确定它是否应该使用继承、从继承而来，或成为独立的数据类型。

**问题**

独立的类是最容易规划。这并不意味着设计一定很容易，只是没有继承，事情会更简单，而且有更多的控制权。基类的设计会影响派生类的设计。在继承层次结构中犯下的错误，可能会给派生类的开发人员带来痛苦。除了谨慎编程和考虑例外情况外，可能对此无能为力。但如果正在编写基类，不仅要考虑类型，还要考虑所有的子类型——基类的行为将影响每一个派生类。

如果将某个类设计为基类，则它应遵循一些规则。其中一条规则是，基类应提供虚拟析构函数，应至少具有一个虚函数。反过来说，如果没有虚函数，则不要提供虚析构函数。除非打算将类设计为基类，否则类不应包含虚函数；因此，每个基类都应提供虚析构函数。

使用基类指针或引用删除派生对象时，C++ 会让我们感到惊讶。多态行为只有使用指针或引用才有可能，必须使用这种方法。指针和引用并不意味着基于堆的对象，但这是一种非常常见的用例。那么问题是什么？尝试使用基类指针或引用删除派生类，会发生什么。一切似乎都应该没问题，但结果谁也说不准；更糟糕的是，这是未定义的行为。

清单 8.15 中的代码有一个基类和一个使用动态资源的派生类。基类的析构函数未定义为虚，主函数创建一个动态 Square 对象并将其赋值给 Shape 指针。使用完派生对象后，将其删除。

清单 *8.15* 没有虚析构函数的基类

```cpp
class Point {
private:
  double x;
  double y;
public:
  Point(double x, double y) : x(x), y(y) {}
};

class Shape {
private:
  Point* location;
public:
  Shape(double x, double y) : location(new Point(x, y)) {}
  ~Shape() { delete location; std::cout << "~Shape\n"; } // 1
  virtual double area() const = 0; // 2
};

class Square : public Shape {
private:
  double* side;
public:
  Square(double side, double x, double y) : Shape(x, y),
  side(new double(side)) {}
  ~Square() { delete side; std::cout << "~Square\n"; }
  double area() const { return *side * *side; } // looks weird
};

int main() {
  Shape* s = new Square(2.5, 0, 0); // place at origin
  std::cout << s->area() << '\n';
  delete s; // 3
  return 0;
}
```

注释 1：非虚析构函数，这似乎是正确的

注释 2：用于多态行为的虚函数

注释 3：删除对象的基类部分，但不删除派生类部分

此代码的输出为

```
6.25
~Shape
```

这表明基类析构函数运行，但泄漏了 `side` 值——每台机器的结果可能会有所不同。

## 分析

`main` 函数的输出显示，只调用了 `Shape` 析构函数。尽管这两个类的设计很不合理，但 `Square` 析构函数未调用这一事实表明，存在内存泄漏。

基类应该至少有一个虚函数，因为只有虚函数才能参与多态行为。否则，继承可能用于代码重

用。没有虚析构函数的基类可能看起来设计得当，因为它可以工作。当通过基类指针操作派生对象时，一切都按预期工作，工作到销毁。行为变得奇怪，但不太可能注意到。这种情况完全取决于编译器如何实现此功能；调试和发布版本也可能不同。

通过基类指针销毁派生类可能只会删除定义基类的部分，基类的析构函数将调用并执行其任务。基类不知道（也不应该知道）应该删除对象的派生类部分。最终结果可能是派生部分成为内存泄漏。这种情况无疑是一个问题，但可能会避免短期运行的程序出现重大问题 - 这是欺骗行为。但当派生类具有动态资源（例如：数据库连接、文件句柄或其他独占或受限资源）时，问题更有可能影响操作。

**解决**

解决方法很简单：为具有一个或多个虚函数的基类提供虚析构函数。此外，不要使用没有虚函数的基类或具有虚函数但不是基类的类。构造顺序始终是从继承层次结构中最基类向下到最派生类。相反，销毁顺序是从层次结构中最派生类向上到最基类—与构造相反。必须首先构造基类，派生类可能依赖于基类知识，所以应该首先销毁最派生类。除非使用 `virtual` 关键字，否则不会发生这种情况。

清单 *8.16* 带有虚析构函数的基类

```
class Point {
private:
  double x;
  double y;
public:
  Point(double x, double y) : x(x), y(y) {}
};

class Shape {
private:
  Point* location;
public:
  Shape(double x, double y) : location(new Point(x, y)) {}
  virtual ~Shape() { delete location; // 1
  std::cout << "~Shape\n"; }
  virtual double area() const = 0;
};

class Square : public Shape {
private:
  double* side;
public:
  Square(double side, double x, double y) : Shape(x, y),
    side(new double(side)) {}
  ~Square() { delete side; std::cout << "~Square\n"; }
  double area() const { return *side * *side; } // still looks weird
};

```

```
29  int main() {
30    Shape* s = new Square(2.5, 0, 0); // place at origin
31    std::cout << s->area() << '\n';
32    delete s;
33    return 0;
34  }
```

注释 1：添加 virtual 关键字；现在，首先调用派生类的析构函数

此执行的输出显示预期的派生类销毁，随后是基类销毁：

```
6.25
~Square
~Shape
```

基类析构函数中添加了 virtual 关键字，结果显示两个析构函数均被调用。这一令人欣慰的事实表明，两个类中的动态分配内存均正确删除。如果基类和派生类使用动态资源（例如：堆内存、系统资源句柄或类似实体），请确保其具有析构函数。

**建议**

- 多态行为需要指针或引用和虚函数；值对象（即基类元素）将分割派生部分。
- 如果某个类具有虚函数，则该类应为基类并从中派生；始终遵循此模式。
- 抵制从没有虚析构函数的现有类派生类的诱惑（或修复该现有类以进行继承）。
- 如果类具有虚函数，请始终定义一个虚析构函数，以防止仅破坏基类部分，造成内存泄漏。
- 如果类没有虚函数，则不要提供虚析构函数。这遵循了避免使用不打算继承虚函数类的建议。

## 8.9. 错误 58： 构造函数和析构函数中调用虚函数

这个错误主要针对正确性。在虚函数中误用构造函数和析构函数，会严重影响其他特性。

面向对象编程的显著优势是，能够创建相关类的层次结构。这些类通过扩展其基础类，使其具有特定行为，从而与其基础类相关联。这种能力源于根据定义特定行为的接口来处理相关类。专门的类会修改接口函数，从而为其特定类型产生有意义的结果。

这种特化允许对一般函数的行为进行细粒度控制。例如，Shape 基类可能具有 Circle 派生类。一般而言，所有形状都应该具有 area 方法，但如何实现计算完全取决于正在处理的形状类型。这种定义一般概念（在基类中）并用具体内容（在派生类中）覆盖该行为的想法称为运行时多态。多态性是面向对象编程的三大公认支柱。了解特定编译器如何实现多态性没有帮助；从概念上了解如何实现多态性，有助于理解为什么不合适在构造函数中调用虚函数。

考虑一下继承是一种自上而下的信息流。基类知道什么（状态，如实例变量）和做什么（行为，如方法）向下流向派生类。多态行为是自下而上搜索特定函数定义。从概念上讲，对适当虚函数的搜索从特定的派生类类型开始，并沿着层次结构向上移动。使用该函数的第一个定义版本。

编译器确定何时需要非虚拟实例方法的地址，并留出空间用于插入该方法的地址。当实例调用这些方法之一时，编译器或链接器会将地址插入可执行代码中。在运行时，该地址用于定位方法的代码。但对于虚函数，需要的不仅仅是这种方法。

虚函数的存在会导致编译器为该类建立一个新表，该表保存每个虚函数的地址在其中声明。这些地址不会像非虚函数那样在编译或链接期间直接插入到代码中。创建实例时，每个虚函数的指针用于引用正确的虚函数。实际细节很复杂，但以下概念模型是一个可行的想法。构造函数确定哪个类表代表正确的虚函数，并调整指针以引用。

构造顺序是精确定义的。假设一个三级层次结构：基类是 A，中间类是 B，最外层派生类是 C。当创建 C 的实例时，将调用其构造函数。

构造函数做的第一件事是调用 B 的相应基类构造函数。B 构造函数通过调用其基类构造函数来初始化 A 部分，从而开始执行。在 A 构造函数完成初始化 A 部分后，B 构造函数继续执行并完成初始化 B 部分。

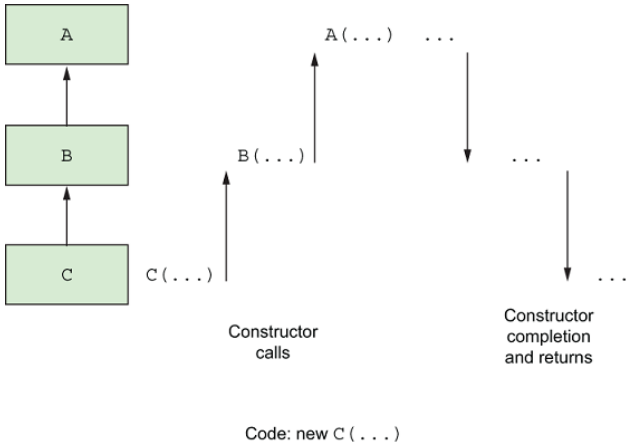


图 8.1 显示构造顺序的三级层次结构

只有在 B 构造函数返回后，C 构造函数才会恢复执行并初始化对象的 C 部分。图 8.1 显示了此构造顺序，其中构造函数首先调用其基类构造函数。

对于具有虚函数的类，认为构造函数负责将指针调整到正确的类表；这在技术上并不正确，但其解释了为什么派生类有最终决定权。由于 A 构造函数首先运行完成，所以将使用定义虚函数的 A 类表初始化表。如果 B 也定义了虚函数，则 A 类的表指针将使用 B 类表的位置覆盖，C 类也是如此。因此，当通过基类指针处理 C 实例时，用于查找正确虚函数的表将是最近调整的表，即使 C 如果实例被视为通用的 A 对象，则写入实例的最后一个表指针就是调用虚函数时调用的指针。当调用虚函数时，通用的 A 指针或引用可以表现得像 C 实例一样。

这种构造顺序的结果是，实际对象（本例中为 C 实例）将仅在其构造的某个时刻调整其类表指针。调整之前的时刻，对虚函数的调用会调用不正确且意外的版本。当构造函数或析构函数正在执行时，对象要么正在构造，要么正在销毁。由于对象仍然需要完全构造，不能认为已经准备好，并且不能保证类的不变性。

**问题**

假设定义了一个继承层次结构，其中不同的人被赋予不同的尊称。在这所学校，研究生受到高度尊敬，学生受到一定程度的尊敬，而普通人则几乎不受到尊敬。由于这个层次结构需要表现得一般化，因此 GradStudent 和 Student 实例将根据需要替换为 Person 实例。但在这种情况下，这些替换的实例必须通过确定适当的尊称来具体表现。

清单 8.17 中的示例代码展示了一个案例，本来应该调用实际类型的 getHonorific 虚函数并输出正确的信息。但是，由于这个错误，它失败了。

此示例显示了一个三级层次结构。每个构造函数都会初始化其实例变量，然后调用 print 方法。当此方法执行时，它会调用虚 getHonorific 方法。输出显示每次调用 print 都会输出不同的消息。getHonorific 虚函数在每个部分的构造过程中将其函数指针添加到表中，这是 print 方法调用的特定函数。

清单 *8.17* 从构造函数调用虚函数

```
class Person {
private:
  std::string name;
public:
  Person(const std::string& name) :
  name(name) { print(); }
  void print() const { std::cout << getHonorific() << name << '\n'; }
  const std::string getName() const { return getHonorific() + name; }
  virtual std::string getHonorific() const { return ""; }
};

class Student : public Person {
private:
  std::string year;
public:
  Student(const std::string& name, const std::string& year) : Person(name),
    year(year) { print(); }
  std::string getHonorific() const { return year + " "; }
};

class GradStudent : public Student {
private:
  bool candidate;
public:
  GradStudent(const std::string& name, const std::string& year,
    bool candidate) : Student(name, year), candidate(candidate)
  { print(); } // 1
  std::string getHonorific() const {
    return candidate ? "candidate " : "";
  }
};

int main() {
  GradStudent aimee("Aimee", "second year", true);
  return 0;
}
```

注释 1：构造函数中调用虚函数；函数版本很可能不正确

**分析**

代码的结果表明，构造每个部分后，将调用其版本的 getHonorific，而不是实际类型的版

本，实例尚未完全构造。因此，虚函数的版本和结果完全取决于对象的哪个部分正在调用该方法。以下输出显示了构造对象时的各种敬语，每个类类型一行：

```
Aimee
second year Aimee
candidate Aimee
```

另一个问题是虚函数可能依赖于给定部分的状态数据。如果构造函数尚未完成该部分，则无法保证数据处于有效状态。如果调用虚函数并且它依赖于此不完整的数据，则结果可能不确定。

**解决**

解决方案是在对象完全构造后调用虚函数。只有这样，虚函数表才会正确初始化，行为才会正确实现多态。以下代码演示了此修复。

清单 8.18 没有虚函数调用的构造函数

```
class Person {
private:
  std::string name;
public:
  Person(const std::string& name) : name(name) {}
  void print() const { std::cout << getHonorific() << name << '\n'; }
  std::string getName() const { return getHonorific() + name; }
  virtual std::string getHonorific() const { return ""; }
};

class Student : public Person {
private:
  std::string year;
public:
  Student(const std::string& name, const std::string& year) : Person(name), year(year) {}
  std::string getHonorific() const { return year + " "; }
};

class GradStudent : public Student {
private:
  bool candidate;
public:
  GradStudent(const std::string& name, const std::string& year,
      bool candidate) :
    Student(name, year), candidate(candidate) {}
  std::string getHonorific() const { return candidate ? "candidate " : ""; }
};

int main() {
  GradStudent aimee("Aimee", "second year", true);
  aimee.print(); // 1
  return 0;
}
```

注释 1：对象构造后调用虚函数

对于正确设计的类，析构函数的工作顺序与构造函数相反。当对指针或引用调用析构函数时，最先调用派生类的析构函数。其执行结束时，会调用其基类的析构函数。这会一直渗透到层次结构的顶部，从而保证对象以与构造相反的顺序销毁。

如果在析构过程中调用虚函数，则最外层派生类中的状态信息可能无效，具体取决于其析构函数的行为。但虚函数会期望这些数据处于有效状态，由于无法保证这一点，因此应始终避免在析构函数中使用虚函数。

**建议**

- 确保在实例完全构造之前未调用类的虚函数；切勿从其构造函数中调用类的虚函数。
- 构造顺序会影响状态信息的有效性，以及将调用哪个版本的虚函数；在调用任何虚函数之前，先让构造完成。
- 销毁顺序会影响虚函数可用状态信息的有效性，其中一些信息可能会销毁或以其他方式无效；当销毁开始时，就永远不要调用虚函数。

## 8.10. 错误 59： 尝试使用多态数组元素

这个错误集中在正确性上，以带有基类元素的数组（或容器）作为参数的函数似乎允许多态行为。可以将派生元素的数组传递给函数——不会发出编译时错误。这提高了效率，但隐藏的危险会让粗心的用户绊倒。

对象容器是收集相关对象，并根据基类公共接口对其进行通用操作的自然方式——这是多态性的核心。但当将派生对象数组传递给函数时，使用可能会出现问题。数组是易于编写和使用的内置容器，其使用方式简单而直观。

**问题**

接受指向基类对象的指针的函数，也可以传递派生类指针或引用。LSP 在这里成立，派生类可以在需要基类的地方替换。这是一种强大的技术，可以编写正确处理传递对象的通用代码。

很容易想到，如果指向对象的指针有效，那么指向对象数组的指针也应该有效。在 C++ 中，这并不意味着派生元素数组可以替换为基类元素数组，并且仍然有效。

C++ 代码使用指针算法访问数组中的元素。假设有一个名为 arr 的整数数组，其中包含多个元素；第三个元素由 arr[2] 索引。此表示法是语法糖，因为访问实际上是 *(arr+2)。指向第一个元素 arr 的 const 指针添加了整数 2，以跳过两个元素并指向第三个元素，而要跳过的字节数是 2*sizeof(element_type)。正是这种指针算法导致传递数组时出现问题。

考虑清单 8.19 中的代码，其中打印了派生类元素的数组。类 D 派生自基类 B，而 D 在需要打印其元素时使用 B 中定义的 getN 方法。B 将 n 实例变量默认为 0，而 D 将其 m 实例变量默认为 1。这些易于区分的值使输出分析变得清晰。

清单 8.19 传递带有派生元素的数组

```
const int SIZE = 4;
class B {
private:
```

```
  int n;
public:
  B(int n=0) : n(n) {}
  ~B() { std::cout << "destroying B\n"; } // 1
int getN() const { return n; } // 2
};
class D : public B {
  int m;
public:
  D() : B(0), m(1) {}
  ~D() { std::cout << "destroying D\n"; }
};

void printArray(const B a[]) {
  for (int i = 0; i < SIZE; ++i)
    std::cout << a[i].getN() << '\n'; // 3
}

void deleteArray(const B a[]) { // 4
  delete [] a;
}

int main() {
  B* bs = new B[SIZE];
  printArray(bs);
  deleteArray(bs);

  D* ds = new D[SIZE];
  printArray(ds);
  deleteArray(ds);
  return 0;
}
```

注释 1：此处故意不写成 virtual，是为了更好地演示整个问题。通常情况下，请不要这样做！

注释 2：数组元素的通用访问函数

注释 3：输出值时使用通用访问函数

注释 4：使用通用删除函数

执行此代码时，B 元素数组按预期工作；n 部分的输出为 0，销毁为 B 元素。但对于 D 元素数组，输出可能会更好。getN 函数交替输出 D 元素的 n 和 m 部分，由于 n 和 m 部分都是整数（大小相同），因此输出显示值的索引是增加 B 元素的大小，而不是必要的 D 元素的大小。此外，元素销毁时，只会调用 B 析构函数，所以会返回到操作系统的内存块可能与实际数组大小不一致，这是内存泄漏。虽然预计不同的系统可能会产生不同的结果，但无论如何这都是未定义的行为。

以下输出表明使用派生元素无法按预期工作。首先，析构函数是基类版本独有的。其次，getN 的结果在 n 和 m 实例变量之间有所不同：

```
0
0
0
0
destroying B
destroying B
destroying B
destroying B
0
1
0
1
destroying B
destroying B
destroying B
destroying B
```

**分析**

printArray 函数将索引数组元素并调用 getN 函数。第一次调用该函数会传递一个 B 元素数组。函数中的索引从第一个元素开始，到下一个元素添加一个 B 元素的大小。此过程持续进行，直到每个元素都已迭代完毕。

第二种情况下，数组包含 D 个元素。由于 D 派生自 B，因此传递数组成功，编译器没有报错—LSP 在这里正常工作。printArray 函数接收指向数组的 const 指针作为其初始点；第一个元素从该点开始。在索引数组并调用 getN 函数时，访问从此元素开始，并使用 B 部分从中提取值；到目前为止一切顺利。它找到 1 的值并输出它。

仔细注意第二个打印输出；它是 m 变量的值。B 元素使用一个整数的空间（通常为四个字节），但 D 元素使用两倍的空间。当循环更新索引值时，会将指针增加一个 B 元素的大小（一个整数），但实际值是第一个 m 变量，恰好是指针的 sizeof(B)(或 *(ptr+sizeof(B))。D 元素的大小是 B 元素的两倍，索引以交替模式引用它们。因此，索引 1 的指针指向第一个元素的 D 部分，而不是第二个元素的 B 部分。如果这听起来不对，因为这是错的。

指针算法对于 B 元素的数组正确运行，但对于派生类（例如 D）则不正确。没有编译器警告，运行时也没有发现段错误（访问无法访问的内存）。此错误与访问越界内存相反，其没有足够的内存可进行访问！

注意：对操作和结果的解释基于同一环境中的非优化编译和执行，优化或编译器更改可能会导致不同的行为。更重要的是，这种行为未定义。

现在，考虑删除数组。当使用包含 B 个元素的数组调用 deleteArray 时，此代码可以正常工作。不明显的是，delete 操作符还将索引数组，并在每个元素上调用 B 版本的删除操作—输出为“destroying B ”。当数组包含 B 个元素时，此方法可以正常工作。

当使用 D 个元素调用此函数时，循环将调用元素的 B 部分的析构函数，并根据 B 元素的大小进行索引。第二个 delete 引用第一个元素的 D 部分。这种情况下发生的情况尚待推测，只有 B 大小的元素销毁。指针算术问题最终导致了这个问题，所以虚析构函数无法解决问题。

现在，考虑一下派生元素比整数更复杂的情况。如果 D 元素包含 std::string 引用或指

向其他数据类型的指针，则不清楚会清理什么。需要澄清这一事实非常令人担忧；编程语言都不应该提供可能有效或可能无效的操作，这取决于无法控制的事物。

在最初的问题中，删除了一个包含 D 元素的数组（或者代码尝试删除），但只有前四个 B 大小的元素删除。如果数组位于堆栈中，最终，这个混乱局面将会清除，但如果内存基于堆，那么释放器将如何处理它，以及回收多少数据仍不清楚。

如果使用 valgrind(用于检测多种内存问题（尤其是泄漏）的出色工具）运行此代码，它会运行干净，因为分配和删除的数量正确配对。地址清理器可以发现一些问题；请参阅 Matt Godbolt 的 Compiler Explorer（https://compiler-explorer.com/z/6d73hscoE）以获取示例。这个错误非常隐蔽，即使是复杂的工具似乎也无法完全检测到问题。开发者有责任永远不要使用基类数组，作为节省代码的通用技术。

**解决**

正确的方法是，拒绝将指向基类和派生类对象数组的指针传递给函数。让函数只处理实际元素类型的数组，复制这些函数是一种简单的补救措施。使用添加函数模板的通用编程技术是确保这种复制的理想方法，这种技术既可读又有效。另一种可能性是将指针数组传递给基类。这种方法的缺点是，代码可能必须处理指向指针的指针，从而使调用和处理语法变得有点混乱——如果愿意，那就去做吧！

现代 C++ 提供了 std::array 类，也能解决这个问题。如果可以的话，请使用该功能来解决这个问题。

这时，代码重用不是一个好主意，如果过于严格地遵循这个概念，将会导致困难。以下代码通过根据模板建议，制作明确的非多态输出和删除函数，来解决前面提到的问题。

清单 *8.20* 没有多态数组传递的函数

```
const int SIZE = 4;

class B {
private:
  int n;
public:
  B(int n=0) : n(n) {}
  ~B() { std::cout << "destroying B\n"; }
  int getN() const { return n; }
};

class D : public B {
  int m;
public:
  D() : B(1), m(2) {}
  ~D() { std::cout << "destroying D\n"; }
};

template <typename T> // 1
void printArray(T a[]) {
  for (int i = 0; i < SIZE; ++i)
```

```
    std::cout << a[i].getN() << '\n';
}

template <typename T> // 1
void deleteArray(T a[]) {
  delete [] a;
}

int main() {
  B* bs = new B[SIZE];
  printArray(bs); // 2
  deleteArray(bs);

  D* ds = new D[SIZE];
  printArray(ds); // 2
  deleteArray(ds);

  return 0;
}
```

注释 1：函数模板处理所需的代码重复

注释 2：不同元素类型的通用处理

以下代码显示修复后的代码的输出：

```
0
0
0
0
destroying B
destroying B
destroying B
destroying B
1
1
1
1
destroying D
destroying B
destroying D
destroying B
destroying D
destroying B
destroying D
destroying B
```

元素显示和删除均正确，并且没有内存丢失。非常重要的是，B 和 D 元素的输出正确，并且 D 元素的每个部分均销毁。元素的销毁涉及正确的类型，不会留下剩余内存或奇怪的操作系统分配和释放不匹配。

**建议**

- 如果必须将基类和派生类数组作为参数传递给函数，则编写单独的函数来处理；编写类似的数组可避免出现奇怪的行为。考虑使用函数模板让编译器编写重复的代码。
- 没有运行时错误并不表示编程方法是正确的，也不表示没有潜在的内存问题。
- 要知道，即使是复杂的工具也无法检测到所有问题；`valgrind` 是一个很棒的工具，但它并不是为检测这种奇怪的极端情况而设计的。
- 正确编码是开发者的责任，并了解使用快捷方式可能引起的问题。

## 8.11. 错误 60： 未初始化所有实例变量

这个错误主要关注正确性、有效性和性能。理解这个习语之前，开发人员可能会说这会对可读性产生负面影响。然而，理解了这个习语，大多数人都会更喜欢这种方法。

类将具有两种类型的实例变量：原始（内置）和类实例。许多现代语言强制初始化所有变量，无论其位置或用途如何。C++ 并不总是确保初始化发生。某些情况下，实例变量已声明但没有定义的初始值。内存中的每个字节都有一些状态，因此这些变量将具有未定义的值。读取这些变量会导致未定义行为，先赋值然后读取就可以了。初始化的目的是，确保状态由程序明确定义。

初始化原始类型的三种常用方法：使用文字值分配变量；输入分配给变量的值；从另一个已初始化的变量分配变量，这些方法确保变量已声明并初始化。开发人员必须确保在第一次使用局部变量之前都保证这一点。

对于类，初始化是构造函数的职责；其主要工作是初始化变量，建立类不变量。构造函数必须接收每个参数的参数值，并使用其初始化实例变量或为其分配默认值。如果不是在每种情况下都这样做，则可能导致未定义行为。

**问题**

许多问题都以某种方式与人有关。假设需要用 `name` 和 `age` 来建模 `Person` 类。这个简单类的以下实现编译后似乎运行良好，只是它使用了未定义的数据。除非手动检查人员的年龄，否则它不会使其错误显现出来。

清单 *8.21* 带有未初始化内置实例变量的类

```
class Person {
private:
  std::string name; // 1
  int age; // 2
public:
  Person(const std::string& name) { this->name = name; } // 3
  void setAge(int age) { this->age = age; }
  friend std::ostream& operator<<(std::ostream&, const Person&);
};

std::ostream& operator<<(std::ostream& o, const Person& p) {
  o << p.name << " is " << p.age << " years old";
  return o;
```

```
}

int main() {
  Person joey("Joey");
  std::cout << joey << '\n';
  return 0;
}
```

注释 1：一个实例变量，是一个类实例

注释 2：一个实例变量，是一个原始类型

注释 3：看似无辜的构造函数

开发人员应该坚持在构造函数中提供年龄（忽略变量是错误的格式）。目的是让用户代码通过调用 `setAge` 方法初始化年龄数据，从而确保 `age` 实例变量有效。当开发者需要记住这样做时，可能会出现问题。编译器无法警告未初始化问题，需要开发者自己找出问题，而这个问题很可能会在代码使用一段时间后出现。

有些编译器在调试模式下运行时会初始化变量，但在发布模式下不会。当然，“在我的机器上可以运行”的说辞并不会让用户满意。

**分析**

类中的两个实例变量属于两个类别。`std::string` 对象是类实例，如果未尝试将值传递给它的构造函数。其默认初始化确保字符串为空，是一个合法的对象，但不包含对开发者的问题有用的值。等一下，名称怎么会是空字符串呢？构造函数主体将其初始化为参数值，该参数值不太可能是空字符串。确实如此，但这超出了实际操作的范围。

构造函数主体中的代码是赋值，而非初始化。此错误的介绍部分中，术语 `initialization` 的使用有些含糊，其中赋值用于初始化。

构造函数确保在调用构造函数主体之前，初始化所有类实例变量。在进入 `Person` 构造函数主体之前会调用默认的 `std::string` 构造函数，在构造函数主体中，赋值将参数值的副本绑定到实例变量。非常低效，变量初始化并赋值。执行了两个操作，但实际上只需要一个操作。更糟糕的是，如果参数不引用调用者的参数值，则会进行另一次复制以初始化参数。

`age` 实例变量是内置类型。构造函数在进入构造函数主体之前会做什么？什么也不做。（谢谢 C，你的遗产在这里非常受用。）`age` 实例变量将是恰好位于其所在内存中的随机位模式。允许访问该值，但其内容未知 - 其含义和用途未定义。这种情况可能会导致漫长，而（并不）有趣的调试。

**解决**

编写构造函数的正确方法是确保，每个实例变量都明确初始化。为每个实例变量提供一个参数值，或在初始化列表中提供一个有意义的默认值。坚持这种方法将确保每次都初始化每个实例变量。

消除多余的类型实例变量初始化的解决方案是，使用参数值以初始化列表形式初始化实例变量。将构造函数主体中的代码，用于赋值给实例变量的情况作为少数例外。

清单 *8.22* 坚持为每个实例变量提供参数的类

```cpp
class Person {
private:
  std::string name;
  int age;
public:
  Person(const std::string& name, int age) :
  name(name), age(age) {} // 1
  friend std::ostream& operator<<(std::ostream&, const Person&);
};

std::ostream& operator<<(std::ostream& o, const Person& p) {
  o << p.name << " is " << p.age << " years old";
  return o;
}

int main() {
  Person joey("Joey", 27); // 2
  std::cout << joey << '\n';
  return 0;
}
```

注释 1：每个实例变量都必须初始化

注释 2：每个构造函数参数都必须有一个实参

如果需要验证（例如变量），请编写一个私有验证方法，该方法返回已验证的参数值或引发异常。在初始化列表中调用验证方法。以下清单中的代码已更新，以包含 age 实例变量的私有验证器，该验证器在构造函数的初始化列表中调用。

清单 8.23 在初始化列表中使用私有验证器验证参数

```cpp
class Person {
private:
  std::string name;
  int age;
  static int validateAge(int age) { // 1
    if (age < 0)
      throw std::invalid_argument("negative age");
    return age;
  }
public:
  Person(const std::string& name, int age) :
  name(name), age(validateAge(age)) {} // 2
  friend std::ostream& operator<<(std::ostream&, const Person&);
};

std::ostream& operator<<(std::ostream& o, const Person& p) {
  o << p.name << " is " << p.age << " years old";
  return o;
}
```

```
20
21 int main() {
22   Person joey("Joey", 27); // 3
23   std::cout << joey << '\n';
24   return 0;
25 }
```

注释 1：返回有效值或引发异常的私有验证器方法
注释 2：验证器在初始化列表中调用；不需要多余的赋值
注释 3：因验证而安全地创建实例

**建议**

- 确保每个实例变量在每个构造函数中都已初始化，无论是通过参数值还是默认值。
- 内置类型不会在构造函数主体之外隐式初始化。
- 不要依赖开发者记住初始化实例变量；在构造函数中提供参数，以便编译器提醒他们提供每个所需的参数。
- 调用参数验证代码以确保正确建立类不变量；将验证器设为私有，需要本地化实例变量的知识，并返回有效值或引发异常。

# 第 9 章 类操作

本章内容

- 对比操作符和成员函数
- 处理自赋值和返回值优化
- 与类成员的编译器协作
- 对比显式和隐式转换
- 增量和减量操作符的前缀和后缀版本

本章继续讨论如何更好地使用实例。类不变量的概念始终存在，但解决这些错误需要与前几章有所不同。这当然并不意味着它不那么重要，只是重点更广泛，涉及不一定直接影响对象状态的领域。

这些错误经常涉及性能类别，其中一些错误通过消除不必要的临时对象来强调这一方面。当表达式求值的中间步骤需要中间对象来保存部分求值的结果（这些结果将在进一步求值中使用）时，就会出现这些临时对象。了解这些临时对象的创建时间，以及如何设计一个类来消除其中的许多临时对象，会显著影响构造函数和析构函数调用的次数。

其他错误集中在误用影响常见用法或性能的操作符。C++ 提供了大量为开发人员提供灵活性，并且必须尊重这种灵活性，以避免不当使用功能。

## 9.1. 错误 61： 对变量遮蔽的误解

这种错误主要影响可读性，如果对变量的范围和含义的理解出现错误，则可能会对正确性和有效性产生不利影响。

命名变量是一项艰巨的任务。开发人员必须很好地命名实体以避免混淆。随着开发人员对代码的理解不断发展，找到良好的命名实践可能需要多次迭代。存在许多可以声明变量的范围。这些范围包括语句、本地、类、命名空间、全局和文件。

变量在写入其类型和名称时，其作用域从声明点开始，并限于声明它的文件、块或语句的末尾。通常，作用域以文件末尾、语句作用域中的右括号或分号结束。当后面的内部作用域定义另一个具有相同名称，但不一定相同类型的变量时，变量将被遮蔽。内部作用域内的代码，无法在外部作用域中看到同名变量。某些情况下，可以使用作用域解析 operator:: 来访问被遮蔽的外部作用域变量。

程序规模大于几个函数，代码中变量的名称通常相同。由于这些函数不共享作用域，不会发生遮蔽，但当内部作用域声明与外部作用域中名称相同的变量时，开发者必须在赋值或计算中使用正确的变量。

**问题**

当作用域定义同名变量时，就会发生遮蔽。作用域越广，越有可能发生遮蔽。全局变量对于编译单元中的所有函数都可见，从声明点到源文件末尾。一般不鼓励使用全局变量，但人们却常常这么做。过度使用全局变量首先会损害正确性和可读性。使用全局变量时，必须谨慎命名。只有当这些短名称是众所周知的概念（如 PI 或 E）时，为全局变量选择简单、简短的名称才合理。对于

鲜为人知的值，应使用命名方案（如用下划线分隔的全大写字母（或类似方案））来清楚地区分全局变量和非全局变量。最好使用命名空间来避免遮蔽错误。

通常，需要更好地实践清晰的命名。假设有一个名为 I 的全局变量；只有阅读过它的定义（以及有用的注释）后才能理解其含义。声明或定义离使用代码越远，变量的名称就应该越长、越有表现力。大多数公司或项目都有一些指导方针；请严格遵循这些指导方针——这关乎可读性和合作性。

许多情况下，短变量名会在多个作用域中使用。这种情况使得理解正在使用哪个变量，变得更加困难。然而，短变量名很常见且方便。

当使用变量名时出现错误时，就会发生内容类型隐藏错误；所用变量的作用域为内部，而预期变量位于外部作用域。清单 9.1 中的代码演示了全局变量的名称与实例变量相同。仅出于说明目的，sum 函数将错误地使用变量作用域。

sum 函数错误地使用了实例变量两次，其使用了内部和外部作用域的变量。应该使用作用域解析操作符，来消除变量名称的歧义。这是对内容的错误掩盖，是一种语法错误——可能只错别字，可以通过正确命名变量轻易解决。

清单 *9.1* 不同作用域中相同的变量名

```
1  double rad = 1.0; // 1
2  class Circle {
3  private:
4    double rad; // 2
5  public:
6    Circle(double rad) : rad(rad) {} // 3
7    void setRadius(double rad) { // 3
8      if (rad < 0)
9        throw std::invalid_argument("negative radius");
10     rad = rad;
11   }
12   double sum() const {
13     return rad + rad; // 4
14   }
15   double getRadius() { return rad; } // 5
16 };
17
18 int main() {
19   Circle c(3);
20   std::cout << "radius is " << c.getRadius() << '\n';
21   std::cout << "enlarged radius is " << c.sum() << '\n';
22   return 0;
23 }
```

注释 1：名称很短的全局变量，含义不明确
注释 2：名称很短的实例变量
注释 3：与实例变量同名的参数，需要消除歧义
注释 4：两个变量同名，但使用局部变量，遮蔽了全局版本
注释 5：使用实例变量，它是作用域内唯一的变量

清单 9.2 的目的是将一个随机值数组传递给 SearchAnalyzer，这将确定数组中出现多少

个值。由于此代码必须修改数组才能进行高效搜索，其他 Analyzer 派生类不需要复制数据，共享数据的思维模式建立。调用 analyze 方法时，预期结果是一行文本，解释在数据中找到多少个随机搜索值，但学生惊讶地发现没有找到任何值。那么，下一步是给教授发电子邮件求助！

清单 *9.2* 不同变量名含义的混淆

```
class Analyzer{
protected:
  int* cloneValues(int* a, int size){
    int* arr = new int[size];
    for (int i = 0; i < size;i++)
    arr[i] = a[i];
    return arr;
  }

  int* array; // 1
  int size;
public:
  Analyzer(int*values, int size) : array(values), size(size) {}
  virtual std::string analyze() = 0;
  virtual ~Analyzer() { delete[] array; }
};

class SearchAnalyzer : public Analyzer{
public:
  SearchAnalyzer(int* values, int size) : Analyzer(cloneValues(values,
      size), size) { // 2
    selection_sort(values, size); // 3
  }
  std::string analyze() {
    int count = 0;
    for (int i = 0; i < 100; ++i)
      if (binary_search(array, rand() % SIZE, size)) // 4
        ++count;
    std::stringstream ss;
    ss << "There were "<< count << " random values found.";
    return ss.str();
  }
};

int main() {
  int* numbers = createArray(SIZE);
  SearchAnalyzer searcher(numbers, SIZE);
  std::cout << searcher.analyze() << '\n';
  return 0;
}
```

注释 1：指向数组的受保护指针

注释 2：对 cloneValues 方法的调用

注释 3：错误地将 values 变量用于排序方法

注释 4：搜索数组变量；并未排序！

这个问题很简单，但做起来却很容易。参数值可理解为要搜索的随机值列表，但这个列表需要复制和排序，从而创建一个代表预期数据的新变量。人们可以专注于输入变量，而忘记那些不显眼的复制值；当其他代码没有单独的数据时，尤其如此。

修复很简单：在对 selection_sort 的调用中将 values 更改为 array，一切即可按预期工作。

不完全是遮蔽，当对相同内容使用相似的变量名时，就会发生一种常见错误。比实际遮蔽更有害的是，清单 9.2 中的代码故意遮蔽，其中同名变量不会导致问题；因此，修复内容遮蔽的简单方法并不适用。相反，两个具有相似含义的变量是错误的——语义错误。该代码取自提交了无法完成项目的学生。

**分析**

全局变量经常使用，但很少以一种容易与类或局部变量区分开来的方式进行限定。清单 9.1 中的全局变量与代码中的类和参数变量同名。本例中，很容易看出 setRadius 参数 rad 应该初始化同名的实例变量。然而，更广泛或更晦涩的代码，会极大地掩盖明显的含义和用法。开发人员倾向于在类或函数中，为全局变量提供简短的名称，使其看起来很“明显”。只要充分说明变量的含义和目的，使用就很简单了。

再来看一下 setRadius 方法：代码需要引用全局 rad 变量，但这是什么意思，它在哪里？如果类和方法没有这样的名称，则会进行搜索以查找外部范围变量；这种情况下，将使用全局变量。这种方法在时间和理解方面都不直观。

当可变参数与实例变量同名时，this 关键字必须区分实例变量和参数变量。如果没有这种区别，参数将赋值给自身，如清单 9.1 所示。有些编译器不会对这种情况发出警告，而是愉快地编译，并生成不符合预期的代码。我教学生使用这种模式作为经验法则。与正在初始化的实例变量同名的参数，是重要的文档；使用 this 指针对于消除歧义至关重要。如果使用这种方法，请养成习惯；否则，偶尔可能会无法使用 this 指针，并会需要调试。

**解决**

为了便于阅读，读者必须理解初始化列表每个部分的含义。括号外的变量是实例变量（清单 9.3 中的 this->radius 部分），括号内的变量是参数（本例中的 double radius）。效率略有提高，开发人员可以复制名称，而无需在短期记忆中保留单个字母或缩写。

清单 *9.3* 由命名空间区分的相同变量名

```
namespace global { // 1
  double radius = 1.0;
};
class Circle {
private:
  double radius;
public:
  Circle(double radius) : radius(radius) {}
  void setRadius(double radius) {
```

```
    if (radius < 0)
      throw std::invalid_argument("negative radius");
    this->radius = radius; // 2
  }
  double sum() const {
    return radius + global::radius; // 3
  }
  double getRadius() const { return radius; }
};

int main() {
  Circle c(3);
  std::cout << "radius is " << c.getRadius() << '\n';
  std::cout << "enlarged radius is " << c.sum() << '\n';
  return 0;
}
```

注释 1：使用命名空间来确保不会发生遮蔽

注释 2：记住要养成使用此模式的习惯，否则就不要使用它

注释 3：变量名的明确使用

隐藏名称是一种记录用于初始化或分配实例变量的参数的有效方法；构造函数或赋值器的参数名称和实例变量名称相同。一些开发人员建议将参数编码为单个字符（实例变量的第一个字符）或名称的缩写版本，以合理地传达意图。更好的建议是在实例变量前面或后面加上一个符号（例如：下划线），以表明该变量是私有的（只有实例变量可以是私有的）。

全局变量的命名方式应使其明显是全局变量。这种方法可避免在需要使用其他变量时，出现无意使用或混淆。有两种方法有助于确保这一点清晰。首先，考虑一种标准化的全局变量命名方式。例如，在每个变量前加上 GLOB_ 或类似的标记。另一种方法是将 _g 或 _global 作为后缀添加到变量名中。记住变量距离其使用位置越远，其名称就应该越长、越具有描述性，以传达其意图和目的；这两个选项很好地解决了这个建议。其次，考虑使用命名空间来包含所有全局变量。这使用种方法使理解全局变量会容易得多。命名空间包含可确保全局变量不会分散在源码中，并且以明显的模式命名。此外，在 enum 或类中包含全局变量，提供了具有相同好处的替代方案。代码演示了在计算连续复利时，如何使用命名空间：

```
namespace Constants {
  const double e = 2.7182828283;
}
...
amount = principal * std::pow(Constants::e, rate*time);
```

**建议**

- 名称与用途的距离越远，名称就应该越长、越具有描述性，使用方式也应该越具体（例如：命名空间）。
- 尽可能避免使用全局变量，以避免出现设计和遮蔽问题。
- 请记住，重复变量名称可能会导致难以理解代码的意图及其正确用法。

- 注意内容的遮蔽（简单版本）和意图的遮蔽（更复杂的版本）；第一个是语法的，第二个是语义的，当类似的代码使用其他变量名时，这使得它更难注意到。

## 9.2. 错误 62： 允许复制唯一对象

此错误主要针对正确性。除非读者理解问题及其解决方案，否则对可读性也会有轻微影响。

许多数据类型表示允许对象重复的概念。`std::string` 可包含产品名称，例如 `Automobile` 类的品牌和型号。在对停车场中的汽车进行建模时，可以预期会出现多辆具有相同品牌和型号的汽车；同一辆车出现多次很正常。

然而，有些数据类型代表着独特的概念，复制它们的想法违反了类不变量。经典 C++ 没有特定于语言的方法来防止复制唯一对象，因此开发者必须维护此属性。

**问题**

让我们考虑在一个系统中对艺术品进行建模，也许是一个为潜在买家制作目录的系统。每件艺术品都是独一无二的，必须如此处理。清单 9.4 中的代码没有定义复制对象的方法，表面上保留了不变量唯一性属性。最初，该类有一个公共的复制赋值操作符，必须对其进行更正。开发人员（错误地）认为将其移至私有部分，将排除创建重复项的可能性。

清单 9.4 为避免复制唯一对象而进行的错误编码

```
class ArtPiece {
private:
  int id;
  std::string description;
  ArtPiece& operator=(const ArtPiece& o) { // 1
    this->id = o.id;
    this->description = o.description;
    return *this;
  }

public: // 2
  ArtPiece(int id, std::string d) : id(id), description(d) {}
  std::string getDesc() { return description; }
};

int main() {
  ArtPiece ml(333, "Mona Lisa");
  ArtPiece ts(444, "The Scream");
  std::cout << ml.getDesc() << '\n';

  ArtPiece dup(ml); // 3
  std::cout << dup.getDesc() << '\n';
  // dup = ts; // 4
  return 0;
}
```

注释 1：已存在的复制赋值操作符移到了私有区

注释 2：没有编写公共复制构造函数或复制赋值操作符

注释 3：这似乎行不通

注释 4：私有复制赋值操作符会阻止这段代码运行

开发人员认为，随着复制赋值操作符的移动，将消除复制对象的能力；毕竟，无法调用复制赋值操作符——忘记了默认的复制构造函数！开发人员考虑了复制唯一对象的效果，并选择隐藏作为防止这种情况发生的手段。输出为

```
Mona Lisa
Mona Lisa
```

一片混乱！

### 分析

如果开发者未指定内容，编译器会编写默认的复制构造函数和复制赋值操作符。由于复制构造函数未编码，因此 default 版本会创建 dup 对象并生成副本。复制成功，而非失败，并且没有伪造的迹象（没有明显违反类不变量的唯一性属性），这种（草率的）方法存在重大问题。

因为编译器提供了默认版本的复制构造函数，所以创建 dup 的代码可以顺利编译和执行，确保对每个实例变量进行浅拷贝。但默认行为会产生伪造的副本，而无法区分。

开发者没有考虑到，如果一个类没有开发者定义的复制构造函数，编译器会很乐意免费提供一个默认版本。类必须设计时，需要避免过于依赖于编译的操作。可以抑制其他类型的不良功能，仅仅通过不编码函数即可；这两个函数不属于那种类型。开发者必须明确进行阻止。

一种已取得一定成效的方法是，显式定义复制构造函数和复制赋值操作符，设为私有。这在很大程度上解决了问题。但如果类中的代码恰好做了一些不合适的事情（复制或赋值），编译器会很乐意遵守，并破坏不变性（伪造）。

### 解决

上述方法有部分正确的想法，但需要包含另一个重要方面。缺少的部分是，提供函数体的每个操作符的私有定义都是可执行代码，所以不应指定函数体。解决方案很简单：声明但不定义这两个操作。下面清单中的代码一举解决了这两个问题。

清单 9.5 通过声明默认操作来避免重复

```
class ArtPiece {
private:
  int id;
  std::string description;
  ArtPiece(const ArtPiece&); // 1
  ArtPiece& operator=(const ArtPiece&); // 1
public:
  ArtPiece(int id, std::string d) : id(id), description(d) {}
  std::string getDesc() { return description; }
  void badMethod() { ArtPiece a = *this; } // 2
};

int main() {
  ArtPiece ml(333, "Mona Lisa");
```

```
  ArtPiece ts(444, "The Scream");
  // ArtPiece dup(ml); // 3
  // ts = ml; // 3
  std::cout << ml.getDesc() << '\n';
  ts.badMethod(); // 4
  return 0;
}
```

注释 1：私有且未实现
注释 2：基于类的赋值编译时没有警告
注释 3：因为操作是私有的，所以不会编译
注释 4：没有编译错误

这两个重复函数被声明为私有函数，且未定义。用户中任何试图使用任一操作的代码都将失败，正如预期的那样，保持类不变，因为面向用户的复制赋值操作符和复制构造函数都不会编译。编译器发现这两个操作都是私有的，并阻止用户代码使用它。但事情并不顺利。

清单 9.5 中引入了 badMethod 方法。该方法的主体执行了编译器无法阻止的赋值，而在面向用户的代码中可以这样做；语法是合法的，但其语义值得怀疑。编译器对代码没有问题，代码编译得很干净；由于没有错误，所以没有任何消息来说明发生了一个错误。这句话听起来可能不对，但它是正确的。问题出现在链接器中。链接器尝试在外部代码中找到复制赋值操作符的定义，但找不到并产生未定义的引用错误。对于独立于链接进行编译的项目来说，这个错误可能更直接。行为可能与预期不同，避免发生是最安全的途径。

如果复制构造函数和复制赋值操作符，是公共的并且设置为 =delete ，现代 C++ 编译器可以捕获此错误。以下代码清单显示了一个示例。请注意注释掉的代码；这些错误现在是非法的。

清单 9.6 使用 *delete=* 避免重复

```
class ArtPiece {
private:
  int id;
  std::string description;
public:
  ArtPiece(int id, std::string d) : id(id), description(d) {}
  ArtPiece(const ArtPiece&) = delete;
  ArtPiece& operator=(const ArtPiece&) = delete;
  const std::string& getDesc() const { return description; }
  //void badMethod() { ArtPiece a = *this; }
};

int main() {
  ArtPiece ml(333, "Mona Lisa");
  ArtPiece ts(444, "The Scream");
  // ArtPiece dup(ml); // 1
  // ts = ml; // 1
  std::cout << ml.getDesc() << '\n';
  // ts.badMethod(); // 1
  return 0;
```

```
}
```

注释 1：现代 C++ 可以避免这些错误。做得好！

**建议**

- 未实现的复制构造函数和复制赋值操作符由编译器默认，这可能会产生意外或不必要的行为。
- 确保知道哪些对象是唯一的，并且不能复制；如果源实体留空，则可以传输值。
- 成员使用现代 C++ =delete 来删除，这会强制出现编译时错误，而非链接时错误。

## 9.3. 错误 63： 未针对返回值优化进行编码

这个错误主要集中在性能上，就是效率。使用操作符的类，会更频繁地创建临时对象。在某些早期的 C++ 编译器中，由于表达式求值期间临时对象的激增，执行某些算术运算的成本更高。这个问题的核心在于，求值此类表达式所需的构造函数调用次数。

已经尝试了各种方案来消除一些临时值，其中两种尝试是返回指针和返回引用。然而，生成的代码很难读懂，有时甚至会出错。其他方法需要笨拙的语法，这些解决方案有助于减少临时值。

**问题**

一些工程师需要编写一个复杂的类。清单 9.7 是一个典型的实现，可以满足其需求。一切看起来都很干净、高效，所以工程师们很高兴；随着时间的推移，他们开始抱怨他们的性能迟缓。这段代码可能没有任何问题，所以开发人员会去别处寻找。（一定要衡量你的性能假设！）下面的清单展示了，用于创建和添加复数的交付代码。

清单 *9.7* 添加复数的典型方法

```
class Complex {
private:
  double real;
  double imag;
public:
  Complex(double r, double i) : real(r), imag(i) {std::cout<<"x\n";}
  Complex(const Complex& o) : real(o.real), imag(o.imag)
    { std::cout<<"y\n"; }
  friend const Complex operator+(const Complex&, const Complex&);
};

const Complex operator+(const Complex& lhs, const Complex& rhs) {
  Complex sum(lhs.real+rhs.real, lhs.imag+rhs.imag); // 1
  return sum; // 2
}

int main() {
  Complex c1(-2, 3.3); // 3
  Complex c2(3, -1); // 4
  Complex c3 = c1 + c2; // 5
```

```
21 }
```

注释 1：第三次调用构造函数
注释 2：第四次调用构造函数——复制构造函数
注释 3：第一次调用构造函数
注释 4：第二次调用构造函数
注释 5：第五次调用构造函数——复制构造函数

此代码有五个未进行返回值优化（RVO）的构造函数调用。前两个是构造 c1 和 c2 对象。第三个是在 operator+ 方法中创建 sum 对象。返回 sum 对象时，将创建一个临时对象，并使用复制构造函数用其值初始化该对象，这是第四次调用。最后，从临时对象创建 c3 对象，进行第五次构造函数调用。

**NOTE**

由于非 RVO 问题普遍存在，当前的编译器会自动实现它。因此，很难找到不执行 RVO 的编译器，所以这个错误只适用于较旧的编译器。但仍然建议使用 RVO 的编码建议，会更简洁。GNU g++ 编译器提供了 -fno-elide-constructors 标志来禁用 RVO 以（仅！）用于实验原因。

**分析**

已经付出了很多努力来消除一些构造函数调用，这将最大限度地减少临时对象的成本。尝试创建一个对象，并返回其指针使得调用语法很糟糕。指针方法可以按照以下清单所示实现。

清单 *9.8* 通过返回一个指针来最小化构造函数调用

```
1  class Complex {
2  private:
3    double real;
4    double imag;
5  public:
6    Complex(double r, double i) : real(r), imag(i) { std::cout << "x\n"; }
7    Complex(const Complex& o) : real(o.real), imag(o.imag) {
8      std::cout<<"y\n"; }
9    friend const Complex* operator+(const Complex&, const Complex&);
10 };
11 const Complex* operator+(const Complex& lhs, const Complex& rhs) { // 1
12   Complex* cpx = new Complex(lhs.real+rhs.real, lhs.imag+rhs.imag);
13   return cpx;
14 }
15 int main() {
16   Complex c1(-2, 3.3);
17   Complex c2(3, -1);
18   Complex c3 = *(c1 + c2); // 2
19 }
```

注释 1：消除一次构造函数调用
注释 2：通过调用构造函数的笨拙和不自然，来消除构造函数调用，但它有效——谁来删除对象？

此代码消除了一个构造函数调用，但迫使用户代码以不自然且不直观的方式编写。更糟糕的是，它分配了堆内存，这总是比自动（堆栈）内存更昂贵。这种方法很难阅读，并且浪费了开发者的时间。

尝试在操作符主体内创建对象，并返回对该对象的引用非常诱人，而且通常可行（但必须忽略非常有用的警告消息）。需要小心以下清单中显示的陷阱！

清单 *9.9* 通过返回引用来最小化构造函数调用

```
class Complex {
private:
  double real;
  double imag;
public:
  Complex(double r, double i) : real(r), imag(i) { std::cout << "x\n"; }
  Complex(const Complex& o) : real(o.real), imag(o.imag) {
    std::cout<<"y\n"; }
  friend const Complex& operator+(const Complex&, const Complex&);
};

const Complex& operator+(const Complex& lhs, const Complex& rhs) { // 1
  Complex cpx(lhs.real+rhs.real, lhs.imag+rhs.imag);
  return cpx;
}

int main() {
  Complex c1(-2, 3.3);
  Complex c2(3, -1);
  Complex c3 = c1 + c2; // 2
}
```

注释 1：消除构造函数调用

注释 2：这是一个简单自然的语法，但这是错误的！

操作符中创建的 `cpx` 对象一直存在，直到函数结束并销毁。对象是在堆栈框架中创建，堆栈框架已失效。返回对现已销毁的对象的引用，意味着它是对无效对象的引用。使用该对象会导致未定义行为，具体情况会有所不同，具体取决于各种条件。危险的意外是，对象所在的堆栈框架通常不会覆盖，代码将正常工作。但有一天，当需要工作时，其会突然神秘地失败。

**解决**

不要尝试创建一个对象并返回一个指针（很尴尬！）或一个引用（错误！），而是编写构造函数，以便编译器可以使用 RVO 消除不必要的临时对象。许多教科书演示的是“创建一个对象并返回”模式，而不是更优雅、更高效的“返回创建的对象”模式。

清单 9.10 演示了更好的方法，其中操作符在 `return` 语句中创建一个新对象。RVO 消除了在返回点和用户代码中的赋值处的两个复制构造函数调用。编译器可以使其更加高效，但并不是每个人都能从后来的技术中受益。

清单 *9.10* 为 *RVO* 编码

```cpp
class Complex {
private:
  double real;
  double imag;
public:
  Complex(double r, double i) : real(r), imag(i) {std::cout<<"x\n";}
  Complex(const Complex& o) : real(o.real), imag(o.imag)
    { std::cout<<"y\n"; }
  friend const Complex operator+(const Complex&, const Complex&);
};

const Complex operator+(const Complex& lhs, const Complex& rhs) {
  return Complex(lhs.real+rhs.real, lhs.imag+rhs.imag); // 1
}

int main() {
  Complex c1(-2, 3.3);
  Complex c2(3, -1);
  Complex c3 = c1 + c2; // 2
}
```

注释 1：编码充分利用了 RVO——消除了一次复制构造函数调用

注释 2：RVO 消除了第二次复制构造函数调用

Complex 对象的创建需要调用一次构造函数，但无论如何都必须这样做。目标不是阻止构造，而是消除临时对象。使用 RVO，可以在 c3 占用的空间中构造 c1 和 c2 的总和，从而消除所有复制构造函数调用。必须在操作符主体中构造一个对象；重点是通过编码消除复制构造函数的成本，以便编译器的 RVO 机制生效。

**建议**

- 许多当前的编译器都实现了 RVO；请仔细阅读文档以了解您的编译器是否实现了 RVO。
- 返回构造函数调用的结果，而不是在操作符主体中创建的对象，以确保代码容易成为 RVO 的目标。
- 操作符可以返回指针，但是调用语法很混乱，并且必须有人负责删除对象。
- 操作符永远不应该返回对本地对象的引用；当调用代码进行访问时，本地对象已经销毁。

## 9.4. 错误 64： 从复制赋值操作符不返回引用

此错误主要针对性能。开发类时可能会创建过多的临时对象，了解创建的原因可帮助开发人员避免不必要的创建。错误地编写复制赋值操作符，可能是创建过多临时对象的一个原因。

**问题**

假设开发人员正在编写一个用于工程计算的程序包。客户需要的数据类型之一是复数。C++ 在 complex 头文件中提供了一个 complex 模板，但是为了说明这个问题，开发人员编写了自己的类。

此代码旨在允许链式赋值，以遵守复制赋值操作符的优先级和结合性。它可以编译和运行，但在处理临时变量时存在一些问题。

清单 9.11 一个类从拷贝赋值操作符返回一个临时对象

```
class Complex {
private:
  double real;
  double imag;
public:
  Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) {}
  Complex operator=(const Complex& o) {
    Complex cpx(o.real, o.imag); // 1
    real = cpx.real;
    imag = cpx.imag;
    return cpx;
  }
};

int main() {
  Complex c1, c2, c3(1, 1);
  c1 = c2 = c3;
  return 0;
}
```

注释 1：也许是为了返回值优化而编写的?

**分析**

复制赋值操作符应该修改实例的状态，但开发人员使用了创建临时对象的方式。赋值的目标应该直接从参数中更新；在本例中，使用临时对象中创建临时对象。创建一个用该对象的值初始化的新对象并返回（可能是过度使用 RVO 了）。RVO 的一个好处是，可以避免从返回的对象中创建另一个临时对象。这种方法很昂贵，其调用构造函数来创建临时对象，并在使用后立即将其丢弃。

清单 9.11 中的用法调用了五次 Complex 构造函数。此外，初始化分配的对象，不是其右侧的对象；而是从对象的副本中获取到的。

**解决**

以下代码展示了一种更好的方法。不返回对象的副本，而是返回对修改后对象的引用，以便使用实际对象（而不是副本）进行赋值。

清单 9.12 复制赋值操作符返回引用

```
class Complex {
private:
  double real;
  double imag;
public:
  Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) {}
  Complex& operator=(const Complex& cpx) { // 1
```

```
    real = cpx.real;
    imag = cpx.imag;
    return *this; // 2
  }
};

int main() {
  Complex c1, c2, c3(1, 1);
  c1 = c2 = c3;
  return 0;
}
```

注释 1：写入修改目标对象

注释 2：返回修改后的目标对象的引用

关键因素是复制赋值操作符返回的是对象的引用，而不是对象的副本。这种方法减少了用于创建临时返回对象的构造函数调用。返回值引用新更新的对象，无需复制对象。赋值的语义得以保留，并且不会创建临时对象。实际对象用于链接赋值。

这种方法应该用于所有赋值类型的操作符，而不仅仅是裸赋值。以下代码仅显示遵循此模式的 `operator+=`。所有其他复合赋值操作符都应遵循此模式。例如，复合加法操作符将按以下清单所示实现。

清单 9.13 从复合操作符返回引用

```
class Complex {
private:
  double real;
  double imag;
public:
  Complex(double r = 0.0, double i = 0.0) :
  real(r), imag(i) {std::cout<<"x\n";}
  Complex& operator+=(const Complex& cpx) { // 1
    real += cpx.real;
    imag += cpx.imag;
    return *this;
  }
};

int main() {
  Complex c1, c2, c3(1, 1);
  c2 += c3; // 2
  c1 += c2;
  return 0;
}
```

注释 1：重复既定模式

注释 2：使用模式

**建议**

- 了解操作符的标准使用模式，并确保类可以提供相同的方法。
- 记住每个操作符的内置优先级和结合性，以确保代码遵循。
- 确保类使用与内置数据类型相同的操作符方法；不要用一些奇特的、不直观的行为让用户感到惊讶。

## 9.5. 错误 65： 忘记处理自我赋值

此错误主要涉及正确性，而很少涉及性能。这是一个问题，主要是在类中使用动态资源时。无法处理自赋值，可能会导致资源出现严重错误。

从一个实例到另一个实例的赋值经常发生。复制赋值操作符用于处理此功能。正确编写的复制赋值操作符必须正确处理动态资源，以避免破坏类的不变量、唯一对象和资源泄漏。但当赋值发生在同一个对象之间时，可能会让开发人员犯难。似乎不太可能发生自赋值，但使用指针时，实际对象比命名变量更模糊。如果自赋值处理不当，可能会丢失数据、发生崩溃或出现未定义行为。

**问题**

考虑清单 9.14 中的代码，从不同的 Auto 对象分配时创建一个新的 Engine 对象。处理动态资源的模式是正确的，但代码中隐藏着一个重大问题。假设删除现有的 Engine 是必要的（确实如此），代码在复制之前将其删除。由于分配源和目标相同，尝试获取 VIN 是对已删除数据的访问。希望代码崩溃时不会出现意外行为或数据损坏；否则，请做好做噩梦的准备。使用 valgrind 和内存清理器等工具是一种非常好的做法，正如这个错误所证明的。

清单 *9.14* 带有隐患的对象赋值

```
class Engine {
private:
  std::string vin;
public:
  Engine(std::string vin) : vin(vin) {}
  std::string getVin() { return vin; }
};

class Auto {
private:
  Engine* engine;
public:
  Auto(Engine* engine) : engine(engine) {}
  Auto& operator=(const Auto& car) {
    if (engine) // 1
      delete engine;
    engine = new Engine(car.engine->getVin()); // 2
    return *this;
  }
};

int main() {
```

```
  Engine* e1 = new Engine("123456789");
  Auto mustang(e1);
  mustang = mustang;
}
```

注释 1：删除现有引擎

注释 2：访问已删除的引擎数据

### 分析

这种资源处理在某些情况下可能是正确的，但使用复制赋值操作符时，就会出现错误。赋值代码首先删除现有的 Engine 对象。然后，从源提供的 vin 分配一个新的 Engine 对象。虽然看起来不错，但删除是一个大问题。

删除 Engine 对象可确保删除现有的 Engine 以防止资源泄漏。从参数的 vin 值分配新的 Engine 旨在仅从该数据创建新的 Engine，但数据来自删除的对象。在许多赋值情况下，这两个对象不会相同，因此此代码可以正常工作。这种情况下，源和目标是同一个对象。这种情况是自我赋值，现在代码会有问题，运行良好的代码现在会崩溃（没好的一天）或执行一些未定义的行为（糟糕的一天）。

我的系统很幸运——尝试访问已删除的引擎 vin 时，会收到有关重复释放或损坏的错误消息。此错误表明出了什么严重问题，我必须弄清楚。也许其他系统不会显示错误。

### 解决

问题是，当对象和参数是同一个实体时，删除 Engine 对象会导致问题。其他情况下（不是同一个对象），则不存在问题。需要的是检查接收对象和参数对象是否相同的代码，不应发生删除。

以下清单中的代码展示了一种更好的方法，可以处理自赋值并避免对这些动态资源进行不当处理。通过简单的测试确定源和目标是否是同一对象，可以避免不当行为。

清单 9.15 复制赋值操作符中处理自赋值

```
class Engine {
private:
  std::string vin;
public:
  Engine(std::string vin) : vin(vin) {}
  std::string getVin() { return vin; }
};

class Auto {
private:
  Engine* engine;
public:
  Auto(Engine* engine) : engine(engine) {}
  Auto& operator=(const Auto& car) {
    if (this == &car) // 1
      return *this;
    if (engine) {
      delete engine; // 2
```

```
      engine = 0; // 3
    }
    engine = new Engine(car.engine->getVin()); // 4
    return *this;
  }
};

int main() {
  Engine* e1 = new Engine("123456789");
  Auto mustang(e1);
  mustang = mustang;
}
```

注释 1：自我赋值测试
注释 2：当源不是目标时才删除
注释 3：如果可以，请使用 nullptr
注释 4：安全地访问有效数据

上述失败案例中的赋值不太可能发生；没有人会那么不负责任，对吧？但如果指针存储在容器中，则从视觉上发现，将对象赋值给自己的能力会变得模糊不清。类似这样的情况可能会导致自我赋值，但根本不明显：

```
autos[i] = autos[j];
```

此外，指针可能会引入意外的自我分配，如下例所示：

```
*pcar1 = *pcar2;
```

必须对自我分配进行测试；如果发现这种情况，则需要快速退出复制分配操作符，并且不删除任何资源。

**建议**

- 出于性能和正确性原因，始终在每个复制和复合赋值操作符中测试自我赋值。
- 确保在删除动态资源后将指针清零。
- 如果目标对象与源对象相同，则立即退出。
- 请警惕通过指针进行自我赋值，无论是直接进行还是使用容器元素进行。

## 9.6. 错误 66： 对前缀和后缀形式的误解

这个错误主要集中在性能上，对正确性关注较少。C++ 允许开发人员重载操作符，以保持编写代码的一致方法。前缀和后缀增量和减量操作符，就是一个很好的例子。编写自定义减量操作符的方法与增量操作符非常相似，因此本讨论只需涵盖增量版本。

增量操作符有两种形式：前缀和后缀。这些形式可能看起来实现相同，并且性能相同。然而，这种直觉是误导性的。它们的实现不仅不同，而且结果类型也不同。这些差异表明它们的用途不同，性能也可能不同。

**问题**

假设表示复数的数据类型在使用增量（减量）操作符时，单调增加（或减少）实部。一种简单的方法是创建一个 increment 函数，将复数值的实部加一。然而，这不是 C++ 的惯用方式，并且不能像使用增量操作符那样传达意图。

因此，开发者可能会将 operator++ 实现得更符合惯用语，他们应该谨慎决定操作符的返回类型。前缀和后缀版本必须返回不同的类型才能正确运行，这一点必须注意。

清单 9.16 中的定义以相同的方式实现了操作符的前缀和后缀版本，假设其实现相同。该语言使用空参数列表定义前缀版本，使用 int 参数定义后缀版本。该参数区分形式并且未使用，不应该有名称。

当一些“聪明”的开发者决定对变量进行双倍递增时，就会出现问题。这不仅在不应该发生的情况下起作用，而且没有按预期工作。在这种情况下，编译器会执行所要求的操作，但不会执行预期的操作。调试这个问题可能很困难。

清单 *9.16* 前缀和后缀相同地实现

```
class Complex {
private:
  double real;
  double imag;
public:
  Complex(double real = 0.0, double imag = 0.0) : real(real), imag(imag) {}
  void increment() { real += 1; }
  Complex operator++();
  Complex operator++(int);
friend std::ostream& operator<<(std::ostream&, const Complex&);
};

Complex Complex::operator++() { // 1
  real += 1;
  return *this;
}

Complex Complex::operator++(int) { // 2
  real += 1;
  return *this;
}

std::ostream& operator<<(std::ostream& out, const Complex& c) {
  out << '(' << c.real << ", " << c.imag << ')';
  return out;
}

int main() {
  Complex cpx(2.2, -1);
  cpx++;
  cpx++++; // 3
```

```
    ++cpx;
    ++++cpx; // 3
    std::cout << cpx << '\n';
    return 0;
}
```

注释 1：前缀操作符，返回对象本身
注释 2：后缀操作符，返回对象本身
注释 3：双倍递增不能正常工作，并且从语法上讲有效，但在语义上值得怀疑

**分析**

示例代码显示了多个问题。首先，增量功能不符合惯用的 C++ 用法。这段代码错失了为数字（和迭代器）类型提供一致接口的机会。就可读性和有效性而言，操作符比函数更能传达意图。

其次，前缀和后缀版本之间的返回值或对象必须不同，其用法和语义不同。前缀形式应该在值修改后返回一个值或对象，后缀形式应该在值被修改之前返回一个值或对象。

第三，后缀版本的返回值或对象存在问题，因为它不是常量。这种情况允许修改返回的值或对象。“聪明”的开发者可能会错误地认为他们可以得到双倍的增量。第一次增量返回值或对象的副本，第二次增量返回副本的修改副本 - 第二次增量不会影响原始对象，原始对象只增加一次，所以增量的期望和现实是不匹配的。编译器不会试图教育这个开发者，而是希望防止这个错误在编译中出现。如果编译器没有检测到这个问题，聪明的做法将导致意外的行为，并且可能会出现一个漫长而奇怪的错误信息。

第四，这些操作符不是根据其他操作符实现的。实际上，随着代码的修改，会采用不同的方法。这种不一致性会滚雪球般发展，产生依赖于操作符使用顺序的行为——顺序的变化会产生不同的结果。前缀和后缀增量操作符的实现完全依赖于 operator+=。关于如何修改实例数据，应该尽可能地独立。这种情况下，三个操作符不需要重复。

**解决**

正确实施后，operator++ 可以简化这些问题的解决，它遵循成熟的 C 和 C++ 习惯用法。使用操作符代替函数非常有效，而且可读性更高。

operator++（以及其他相关操作符）的实现应根据内置类型维护操作符的语义。如果操作正确，实现细节会随着时间的推移保持一致，从而避免因在类中重复知识而造成的麻烦。两种形式均根据 operator+= 实现，其中最具体的操作符（增量）依赖于最通用的操作符。这种关注点分离使操作符的知识，有利于保持独立和一致。

需要考虑这些操作符的两个方面：实现细节和性能特征。增量操作符及其操作对象的文本顺序，揭示了它们之间的语义差异。前缀版本在获取对象值之前先递增；为是一种先更新再求值的方法——求值表达式会反映更新后的值。调用代码会看到实际对象的值是什么。

后缀版本应视为先求值再更新的方法——首先进行求值，保存对象的当前值，然后进行更新，并返回保存的值。求值并不反映对象的当前值，而是反映更新前的值。调用代码看到的是过去值，而不是现在值。这两个值之间存在时间差异。

由于后缀版本使用一个临时对象，来反映增量操作之前对象的值，因此必须返回一个值对象，而不是引用——不想返回对现已销毁本地对象的引用！

两个版本的操作符的返回类型已更正。前缀版本应返回对对象本身的引用，因为它是修改后的

对象。“聪明”的开发者仍然可以执行双递增（可读性下降！），因为引用的对象被正确修改，而不是副本。后缀版本不能返回对自身的引用，但必须返回对象先前值的副本。因此，必须返回一个值，而不是引用。为了防止出现双递增问题，使返回值 `const` 禁止修改它—如下面的清单所示，这也读起来也好多了。

清单 *9.17* 语义正确的自增操作符的实现

```
class Complex {
private:
  double real;
  double imag;
public:
  Complex(double real = 0.0, double imag = 0.0) : real(real), imag(imag) {}
  Complex& operator++();
  const Complex operator++(int);
  Complex& operator+=(int);
  friend std::ostream& operator<<(std::ostream&, const Complex&);
};

Complex& Complex::operator++() { // 1
  Return *this += 1; // implemented in terms of operator+=
}

const Complex Complex::operator++(int) { // 2
  Complex temp(*this);
  *this += 1; // implemented in terms of operator+=
  return temp;
}

Complex& Complex::operator+=(int n) {
  real += n;
  return *this;
}

std::ostream& operator<<(std::ostream& out, const Complex& c) {
  out << '(' << c.real << ", " << c.imag << ')';
  return out;
}

int main() {
  Complex cpx(2.2, -1);
  cpx++;
  //cpx++++; // 3
  ++cpx;
  ++++cpx;
  std::cout << cpx << '\n';
  return 0;
}
```

注释 1：前缀版本返回修改对象的对象引用
注释 2：后缀版本返回未修改对象的副本
注释 3：双后缀增量在语法上不再有效，但在语义上合理

循环体或更新部分中的后缀版本效率低于前缀版本，每次对对象调用后缀增量操作符时，对象都必须创建一个副本，其中执行内存分配；调用复制构造函数；返回的值永远不会使用（真是浪费）；匿名对象超出范围后，会调用析构函数。使用前缀版本的成本要低得多，返回一个引用，然后就完成了。

**建议**

- 根据前缀增量（和减量）实现后缀增量（和减量），并根据 operator+=（operator-=）实现前缀增量（和减量），将如何修改值的知识本地化到单个位置。
- 尽可能使用前缀形式，尤其是在循环体和循环更新部分，这可以最大限度地减少对象的副本数量。

## 9.7. 错误 67： 误导性的隐式转换操作符

这个错误主要集中在可读性上，其次是正确性。阅读代码应该能说明全部情况；隐式转换会隐藏一些细节。

当函数调用未收到预期类型的参数时，C++ 提供了一组复杂的规则，用于将一种数据类型转换为另一种数据类型。使用两种形式的隐式转换，具体取决于是否定义了转换函数和转换构造函数。此错误主要针对转换函数。

**问题**

转换操作符使用 operator 关键字定义，后跟要转换的数据的类型，后跟括号。例如，清单 9.18 中的 Rational 类可能返回表示有理数近似值的 double 值。double() 操作符是 Rational 类成员，将 Rational 值转换为其近似值 double。

清单 *9.18* 带有转换函数的类

```
class Rational {
private:
  int num;
  int den;
public:
  Rational(int num, int den = 1) : num(num), den(den) {}
  operator double() { return (double)num/den; }
  friend std::ostream& operator<<(std::ostream&, const Rational&);
};

std::ostream& operator<<(std::ostream& out, const Rational& r) {
  out << r.num << '/' << r.den;
  return out;
}

```

```
int main() {
  Rational r1(3);
  std::cout << r1 << ' ' << (double)r1 << '\n'; \\ 1
  if (r1 == 3) \\ 2
    std::cout << "equal\n";
  else
    std::cout << "not equal\n";
  return 0;
}
```

注释 1：显式转换非常易读

注释 2：隐式转换容易引起误解；看起来像是在比较整数

清单 9.18 中的代码按预期运行，因为定义了友元函数 operator<< 。其输出为

```
3/1 3
equal
```

没有办法准确预测要比较的实际值（0.333 值）。我的学生经常听到我说：“浮点值是近似值，很少精确。”这种比较浮点值的方法是错误的，应始终使用 delta-epsilon（无穷小分析）方法进行比较。

**解决**

如果需要转换函数，最好使用函数而不是操作符。当编译器隐式尝试将值从一种类型转换为另一种类型时，不会考虑该函数。在无法进行转换的情况下，编译器将出现错误，从而提醒开发人员函数不足。开发人员可能会意识到所编码的内容没有意义，并选择不同的方法。

下面的代码通过添加转换函数，并消除隐式转换来避免这两个问题。此外，对读者来说，更明显的是发生了显式转换。

清单 *9.19* 使用显式转换函数的类

```
class Rational {
private:
  int num;
  int den;
public:
  Rational(int num, int den = 1) : num(num), den(den) {}
  double toDouble() { return (double)num/den; }
  friend std::ostream& operator<<(std::ostream&, const Rational&);
};
std::ostream& operator<<(std::ostream& out, const Rational& r) {
  out << r.num << '/' << r.den;
  return out;
}
int main() {
  Rational r1(3);
  std::cout << r1 << ' ' << r1.toDouble() << '\n';
  if (r1.toDouble() == 3)
```

```
    std::cout << "equal\n";
  else
    std::cout << "not equal\n";
  return 0;
}
```

删除 double 操作符并添加 toDouble 函数使隐式变得显式。现在，比较对读者来说更加明显，并且类型不匹配应该显而易见。由于比较双精度值，而非有理数，所以存在不精确性。使用 operator== 比较双精度是一个错误的选择。

**建议**

- 尽量少用隐式类型转换操作符，除非读者能明显看出它们的用途，但这种情况很少见。
- 编写显式类型转换函数可增强可读性，并避免意外转换和错误假设。

## 9.8. 错误 68： 过度使用隐式转换构造函数

这个错误主要影响性能；可读性和有效性可能会受到轻微的负面影响。

C++ 编译器中使用混合模式计算和函数调用非常容易，这种情况通常发生在将函数调用，或算术运算应用于两种不同类型时。如果编译器可以找到一种方法，将值从一种类型转换为另一种类型。从而使调用成功，那么它将默默地（通常）完成此操作，但这种转换可能会有性能损失。

**问题**

隐式转换感觉很好，当它们顺利且正确地工作时很有趣。清单 9.20 中的代码显示了一个适度的 Complex 类，考虑到有人可能希望将双精度值添加到实数部分。不明显的是，没有办法进行这种简单的添加。定义的 operator+ 接受两个 Complex 参数；显然，double 不是 Complex——需要静默的、可能有用的构造函数类型转换来救援！

清单 *9.20* 隐式、静默和昂贵的构造函数类型转换

```
class Complex {
private:
  double real;
  double imag;
public:
  Complex(double real, double imag=0) : real(real), imag(imag) {}
  double getReal() const { return real; }
  double getImag() const { return imag; }
};
const Complex operator+(const Complex& lhs, const Complex& rhs) {
  return Complex(lhs.getReal()+rhs.getReal(), lhs.getImag()+rhs.getImag());
}

int main() {
  Complex c1(2.2); // 1
  Complex c2 = c1 + 3.14159; // 2
```

```
  Complex c3 = 2.71828 + c1; // 2
  Complex c4 = 2.71828 + 3.14159; // 1
}
```

注释 1：一次构造函数调用

注释 2：两次构造函数调用

如预期的那样，调用 Complex 构造函数来创建对象 c1，还为对象 c2 和 c3 调用了两次。第一次调用是转换调用，其中 double 值转换为等效的 Complex 表示。double 值初始化 real 组件，而 imag 组件默认为零。最后，对象 c4 是单个构造函数调用。4 行代码中有 6 次构造函数调用。如果这是一个更复杂的数据类型，还会调用 6 次析构函数。在这种简单的纯数据类型的情况下，大多数（如果不是全部）编译器都会优化析构函数。

**分析**

Complex 对象 c1 和 c4 的构造显而易见，每个对象只需调用一次构造函数。这种行为无法消除。对象 c2 和 c3 必须先将 double 转换为 Complex 对象，然后执行加法，因为 operator+ 只接受 Complex 对象。转换需要调用一次构造函数；加法的结果需要调用第二次构造函数。将返回的对象赋值给声明的对象不需要构造函数调用，RVO 会就地复制。

此操作的所有方面都是正确的，但如果性能是关注点，构造函数（和析构函数）调用可能会成为痛点。如果不关心性能，这仍然是一种低效的方法。我们需要一种不需要调用构造函数，即可进行混合模式转换的方法。

**解决**

性能不是问题的情况下，上述方法可以接受，并且代码量最少。这提高了可读性和效率（小菜一碟）。但在不允许调用构造函数（和析构函数）的情况下，必须有一种方法可以最大限度地减少这些调用。

解决方案是使用不同的参数列表重载 operator+，这些参数列表可以接受 Complex 和 double 的组合，但需要多写几行代码（效率会降低）。两个参数均为 double 类型的情况不包括在内，也不可能包括。C++ 确保重载操作符至少具有一个用户定义的参数类型。

如果编译器允许这种情况，则将重新定义两个 double 值的相加，从而导致与内置的相加规则不一致的行为。他们考虑到了一切！性能问题的解决方案很简单：为预期使用的每个组合编写一个重载的 operator+。以下代码显示了 Complex 类的改进。再添加两个操作符，即可完成可能的重载参数列表集（Complex/Complex、double/Complex 和 Complex/double）。

清单 *9.21* 通过提供重载操作符来最小化构造函数转换

```
class Complex {
private:
  double real;
  double imag;
public:
  Complex(double real, double imag=0) : real(real), imag(imag) {}
  double getReal() const { return real; }
  double getImag() const { return imag; }
};
```

```
10
11  const Complex operator+(const Complex& lhs, const Complex& rhs) {
12    return Complex(lhs.getReal()+rhs.getReal(), lhs.getImag()+rhs.getImag());
13  }
14
15  const Complex operator+(const Complex& lhs, double rhs) { // 1
16    return Complex(lhs.getReal()+rhs, lhs.getImag());
17  }
18
19  const Complex operator+(double lhs, const Complex& rhs) { // 1
20    return Complex(lhs+rhs.getReal(), rhs.getImag());
21  }
22
23  int main() {
24    Complex c1(2.2);
25    Complex c2 = c1 + 3.14159;
26    Complex c3 = 2.71828 + c1;
27    Complex c4 = 2.71828 + 3.14159;
28  }
```

注释 1：重载操作符定义消除了混合模式的构造函数转换

确保操作符具有所有必要的重载，这将最大限度地减少临时对象的影响。构造函数（和析构函数）调用的数量已减少到 4 个。此代码表明，在实现混合模式计算、函数调用和类型转换构造函数时，性能会受到显著影响。但不要过度使用 - 只重载必要的操作符，而不是所有操作符。困惑吗？只是尽力的平衡而已。

**建议**

- 如果担心性能，则对混合模式函数或操作符调用重载每个预期模式；如果性能不是问题，这仍然是一种很好的彻底方法。
- 操作符必须至少具有一种用户定义的参数类型，以保持语义一致性并避免重新定义现有规则。

## 9.9. 错误 69： 过于关注独立操作符

这个错误主要影响性能，对可读性或有效性没有影响。算术运算通常以代数形式实现。这种默认方法在许多情况下是合理的，我们无需担心它对性能的影响。但在其他情况下，可以从阻止临时对象的创建和销毁中受益。算术表达式是一个非直观形式，可以最小化临时对象并提高执行速度的领域。

**问题**

考虑一下此代码，其中创建了一些 Complex 对象，然后将其相加以初始化另一个对象。这种方法是使用类对象创建算术表达式的常见且直观的方法。代码中有 5 个构造函数调用：3 个用于前三个对象，即 c1、c2 和 c3，这些对象无法最小化，两个是在求和表达式中创建的临时对象。消除这些临时对象将提高性能。许多现代编译器都会尝试消除它们，但可以用来理解这个问题。

清单 *9.22* 在表达式求值中具有过多临时项的类

```
class Complex {
private:
  double real;
  double imag;
public:
  Complex(double real=0, double imag=0) : real(real), imag(imag) {}
  double getReal() const { return real; }
  double getImag() const { return imag; }
};

const Complex operator+(const Complex& lhs, const Complex& rhs) {
  return Complex(lhs.getReal()+rhs.getReal(), lhs.getImag()+rhs.getImag());
}

int main() {
  Complex c1(2, 2); // 1
  Complex c2(0, -1);
  Complex c3(-2.2, 4.2);
  Complex c4 = c1 + c2 + c3; // 2
}
```

注释 1：创建每个对象都需要调用一次构造函数

注释 2：计算需要两个临时对象

**分析**

前三个构造函数调用必不可少，必须存在才能创建 `Complex` 对象。我们无法在没有构造函数调用的情况下，神奇地获得对象，但求和会创建两个临时对象。从性能角度考虑，只要创建了临时对象，就有可能消除它的创建。请记住，每个非平凡数据类型的构造函数在某个时刻，都有一个关联的析构函数调用，这会使使用临时对象的成本加倍。大多数编译器可以优化简单对象的析构函数调用。

第一个临时变量保存 `c1` 和 `c2` 的总和，第二个临时变量保存第一个临时变量和 `c3` 的总和的结果。然后，调用默认的 `operator=` 来使用第二个临时变量中保存的值初始化 `c4` 对象。问题是，是否可以减少构造函数调用的次数，同时仍提供相同的功能。

**解决**

可以通过直接使用复合赋值操作符或与独立 `operator+` 结合使用，来减少临时变量的数量。如果同时提供独立版本和复合版本，则应根据复合赋值形式实现独立版本。此外，消除了代码重复的可能性，将基本函数单独封装在复合赋值版本中。独立版本调用此函数，但不提供基本逻辑。以下代码通过根据复合赋值版本定义独立操作，并在计算中继续使用独立操作符形式展示了这些改进。

清单 *9.23* 表达式求值中具有最小化的临时类

```
class Complex {
private:
```

```
  double real;
  double imag;
public:
  Complex(double real=0, double imag=0) : real(real), imag(imag) {}
  Complex& operator+=(const Complex&);
  double getReal() const { return real; }
  double getImag() const { return imag; }
};

Complex& Complex::operator+=(const Complex& o) { // 1
  real += o.real;
  imag += o.imag;
  return *this;
}

const Complex operator+(const Complex& lhs, const Complex& rhs) { // 2
  return Complex(lhs) += rhs;
}

int main() {
  Complex c1(2, 2);
  Complex c2(0, -1);
  Complex c3(-2.2, 4.2);
  Complex c4 = c1 + c2 + c3;
}
```

注释 1：操作符逻辑包含在复合赋值版本中

注释 2：独立操作符按照复合赋值版本实现

当独立操作符以复合赋值版本实现时，就不再需要临时对象了。清单 9.23 中的实现进行了三次构造函数调用，这些调用是构造 c1、c2 和 c3 对象所必需的。求和操作符的求值不会导致创建任何临时对象，从而提高了性能，同时又不损害表达式求值的有效性。

清单 9.24 中的代码展示了实现操作符调用序列的另一种方法。独立版本单独会为每次对象调用创建一个临时对象，而此形式会就地更新对象，而不会创建临时对象。Complex 类的定义与清单 9.23 中的相同，只是 main 函数已更改。

清单 9.24 多操作符调用的另一种形式

```
int main() {
  Complex c1(2, 2);
  Complex c2(0, -1);
  Complex c3(-2.2, 4.2);
  Complex c4(c1); // 1
  c4 += c2; // 2
  c4 += c3;
}
```

注释 1：调用复制构造函数

注释 2：不需要临时变量的顺序求和

这种方法的优势在于复合赋值形式效率更高，它们会就地更新值。之前的独立版本表明它必须返回一个新对象，并且这个新对象需要通过构造函数调用创建临时对象。

**建议**

- 为了防止创建临时对象，请考虑实现算术操作符的复合赋值版本，并从独立版本调用这些版本。
- 可以有效地按顺序多次使用该操作符，而不会创建临时对象。

## 9.10. 错误 70： 未能将非变异方法标记为常量

这个错误关注正确性，并稍微提高了效率。当执行类实例方法时，代码主体可以访问实例变量，并在需要时修改它们。这种行为是好的和正常的，但并非所有方法都需要修改变量。`const` 方法永远不会修改实例变量，只直接或间接返回这些状态变量。直接方法称为访问器，间接方法称为计算访问器。

**问题**

假设有一个相当大的类，并且它的方法太大，难以阅读和理解。还要考虑一些访问器类型的方法中嵌入了逻辑，这些方法可以很容易地（也许是错误的）修改状态变量。编译器无法防止对状态的意外修改。当这种情况发生时，对象可能会破坏类的不变量；如果是无意的，对象最终会处于不正确的状态。以下代码是此问题的一个简化版本，消除了所有模糊代码，并暴露了基本框架。

清单 *9.25* 无意中修改状态的复杂访问器方法

```
class Person {
private:
  std::string name;
  int age;
public:
  Person(const std::string& name, int age) : name(name), age(age) {}
  const std::string& getName() { return name; }
  int getAge() { // 1
    ++age; // oops, unintentional
    return age;
  }
};

int main() {
  Person amy("Aimee", 26);
  std::cout << amy.getName() << " is " << amy.getAge() << " years old\n";
  return 0;
}
```

注释 1：也许这个方法应该改变状态

**分析**

虽然清单 9.25 中的代码非常简单，错误也很明显，但更复杂的代码会通过引入多行代码来

掩盖状态修改，其中一些代码很难理解。这种混乱的代码中，很容易无意中修改状态。如上所述，编译器无法发出警告，其无法知道修改是正确的还是错误的——它必须假设更改是正确的。需要的是一些方法来避免无意修改变量的状态。

**解决**

使用 `const` 关键字可以在很大程度上避免无意中修改状态变量。虽然没有关键字可以保证代码正确，但 `const` 关键字至少可以确保，实例变量不会在某些方法中修改。当方法用 `const` 关键字标记时，编译器会确保该方法不会修改实例变量。

建议将每个可以这样做的方法标记为 `const`，错误地将方法标记为 `const` 也无妨。经检查，可以确定该方法不应使用该关键字，可以删除。对于所有其他方法，即那些有意改变状态的方法，该关键字是不必要的（实际上，这将是一个错误）并且不会使用。

编译器无法确保修改是必要的，但可以保证强制执行不修改。清单 9.26 中的代码修改了 `getAge` 方法并发出错误消息，这是由于无意修改了代码。分析代码后，确定修改是一个错误（可能是错别字）并消除。由于该方法不应修改状态，因此 `const` 关键字确保编译器不允许实例数据的更改。

清单 *9.26* 将方法标记为 *static* 以避免修改状态

```
class Person {
private:
  std::string name;
  int age;
public:
  Person(const std::string& name, int age) : name(name), age(age) {}
  const std::string& getName() const { return name; }
  int getAge() const { // 1
    return age;
  }
};
int main() {
  Person amy("Aimee", 26);
  std::cout << amy.getName() << " is " << amy.getAge() << " years old\n";
  return 0;
}
```

注释 1：`const` 关键字可避免实例变量发生改变

**建议**

- 将所有非变异方法标记为 `const`，以防止无意中修改状态；成本很小，但可以保证的安全性。

## 9.11. 错误 71： 未能正确地标记类方法为静态

这个错误主要针对可读性和有效性。类由状态（实例变量）和行为（方法）组成。声明方法有两种选择：实例方法或类方法。实例方法可以访问所有实例和类变量（使用 `static` 关键字声

明)。类方法只能访问类变量，所以这些方法不能访问对象值。可能会出现一个问题：为什么任何方法都不访问实例变量，以及这样做有什么好处。

当我的学生发现所教的内容不一致或无法立即理解时，我会鼓励他们提出这样的问题，而不是盲目地接受我的教学。这种错误是提出问题，并寻求澄清的机会。

类方法的一个众所周知的优点是，执行代码不需要类实例。这种情况允许定义与类相关的辅助方法，但可以像独立函数一样执行。例如，考虑 Java 提供的 Math 类。创建该类的实例，用特定值初始化它，然后调用 sqrt 方法会非常尴尬。直接调用 sqrt 方法，传递特定值作为参数要好得多。C++ 提供独立函数，易于使用且清晰易懂；此外，通过在类中声明类方法来实现与 Java 类似的功能。

**问题**

假设正在编写一个 Rational 类，其中某个数学团队想要使用精确值，而不是浮点值提供的近似值。例如，有理数 10/3 是精确的，但无法用十进制或二进制精确表示。清单 9.27 中的代码显示了实现 Rational 子集的第一次尝试。每次对象的状态发生变化时都会使用 reduce 和 gcd 方法。该类的每个实例都需要频繁调用该方法。

清单 *9.27* 对理性类的第一次尝试

```
class Rational {
private:
  int num;
  int den;
public:
  Rational(int num, int den = 1) : num(num), den(den)
    { reduce(); } // 1
  friend std::ostream& operator<<(std::ostream&, const Rational&);
  Rational& operator*=(const Rational&);
  int gcd(int a, int b) {
    if (b == 0)
      return a;
    return gcd(b, a % b);
  }
  void reduce() {
    int div = gcd(num, den);
    num /= div;
    den /= div;
  }
};

std::ostream& operator<<(std::ostream& out, const Rational& r) {
  out << r.num << '/' << r.den;
  return out;
}

Rational& Rational::operator*=(const Rational& o) {
  num *= o.num;
```

```
  den *= o.den;
  reduce(); // 2
  return *this;
}

int main() {
  Rational r1(3, 9);
  std::cout << r1 << '\n';
  return 0;
}
```

注释 1：reduce 和 gcd 在构造函数中调用

注释 2：reduce 和 gcd 在 operator*= 中调用

**分析**

reduce 和 gcd 方法是公共实例方法。从正确性角度看，这种实现没有问题，但这一事实会影响可读性。代码暗示这些方法是必要地实例方法，所以只有存在实例时才应调用这些方法，并且每个方法都需要访问至少一个实例变量。reduce 方法按编写方式访问 num 和 den，但 gcd 方法仅对其参数进行操作。

函数式编程定义了一个术语，用于描述 gcd 的实现方式；它是一个纯函数，表示该方法不会访问或影响其范围之外的任何状态；换句话说，没有副作用。完全根据其参数传递给它的数据来计算其结果 - 相同的输入始终产生相同的输出。由于 gcd 是纯函数，所以没有理由坚持它需要一个对象来操作。可通过使用 static 关键字标记此方法，该方法可以作为类方法的候选。

reduce 方法使用实例变量，它必须是实例方法。或者它确实是实例方法？可以重写该方法，以消除该要求并使其成为纯函数。如果可以将其变为纯函数，那么应该这样做吗？假设答案是肯定的。

**解决**

清单 9.28 中的代码重新编写了对 Rational 类的第一次尝试，其立即将 gcd 标记为静态方法。此标记现在表明该方法在实例之间共享，并且不影响任何实例变量。通过添加 static 关键字，可以更清楚地描述其含义和范围。gcd 方法是纯方法（它不会改变自身之外的状态）。不幸的是，reduce 方法不能是纯方法，它会修改实例变量。由于这两种方法现在都是类方法，所以在语义上与以前不同，并传达了不同的含义。最好将它们设为私有，仅供在类中使用。

清单 *9.28 Rational* 重新设计了两个类方法

```
class Rational {
  private:
  int num;
  int den;
  static int gcd(int a, int b) { // 1
    if (b == 0)
      return a;
    return gcd(b, a % b);
  }
```

```
  static void reduce(int& num, int& den) { // 1
    int div = gcd(num, den);
    num /= div;
    den /= div;
  }
public:
  Rational(int num, int den = 1) : num(num), den(den) {
    reduce(this->num, this->den); }
  friend std::ostream& operator<<(std::ostream&, const Rational&);
  Rational& operator*=(const Rational&);
};

std::ostream& operator<<(std::ostream& out, const Rational& r) {
  out << r.num << '/' << r.den;
  return out;
}

Rational& Rational::operator*=(const Rational& o) {
  num *= o.num;
  den *= o.den;
  reduce(num, den);
  return *this;
}

int main() {
  Rational r1(3, 9);
  std::cout << r1 << '\n';
  return 0;
}
```

注释 1：现在，一个私有的纯函数

为了看起来更简洁，两个方法都设为私有；但这真的是考虑此语义特征的最佳方法吗？这个疑问让我们有时间思考这些变化。在盯着 `reduce` 方法时，可能会质疑它是否正确传达了正确的行为。毕竟，`reduce` 旨在与实例变量交互。传递引用当然允许它成为纯函数，但它几乎与其实际意图相冲突，并且引用对于影响两个变量是必要的。最好将 `reduce` 返回到实例方法，而不是使用有点难以阅读的引用。引用非常方便，但在这种情况下，往往会掩盖方法的含义。将实现返回到实例方法更有意义。清单 9.29 表明，按照这种推理进行的修改已经取得了成果。

清单 *9.29 Rational* 再次进行了修改，以更好地指示一个类方法

```
class Rational {
private:
  int num;
  int den;
  void reduce(); // 1
public:
  Rational(int num, int den = 1) : num(num), den(den) { reduce(); }
```

```
  friend std::ostream& operator<<(std::ostream&, const Rational&);
  Rational& operator*=(const Rational&);
};

std::ostream& operator<<(std::ostream& out, const Rational& r) {
  out << r.num << '/' << r.den;
  return out;
}

Rational& Rational::operator*=(const Rational& o) {
  num *= o.num;
  den *= o.den;
  reduce();
  return *this;
}

static int gcd(int a, int b) { // 2
  if (b == 0)
    return a;
  return gcd(b, a % b);
}

void Rational::reduce() {
  int div = gcd(num, den);
  num /= div;
  den /= div;
}

int main() {
  Rational r1(3, 9);
  std::cout << r1 << '\n';
  return 0;
}
```

注释 1：私有，因为只有实例可以使用此方法

注释 2：公开，因为用户可能希望使用该功能

清单 9.29 中考虑了另一个方面。数学团队表示，希望 gcd 可供外部使用。static 关键字的主要特征之一是 sharing 的概念。使用类方法，类可以与外界共享其某些行为或知识。

**建议**

- 仔细考虑是否可以将方法设为纯方法；如果可以，请将其标记为 static，这样可以更好地传达其意图，它不会影响任何实例变量。
- 考虑静态方法是否应设为 public 或 private；public 版本可以与用户共享以供一般使用，而 private 版本应仅供类使用。
- 如果派生类中需要 private 方法，请考虑将其设为 protected，这样继承类就可以直接访问，也可以避免在用户代码使用。

## 9.12. 错误 72： 错误的选择成员函数和非成员函数

这个错误关注的是语义的正确性（而不是正确或错误的结果）和有效性。最初，有效性似乎受到了负面影响，但当实施要求发生变化时，积极的一面就会显现出来。当要求发生变化时，修改方法应该尽量减少受影响方法的数量。

函数有三种类型：成员、非成员和非成员友元（以下称为 friend）。面向对象编程强调封装、继承和多态这三大支柱。封装的理念是向用户代码隐藏实现细节（实例变量和方法主体），以确保它仅使用类的公共接口。

良好封装的好处是可以重新实现类以改进一个或多个类别。一个例子是 Date 类的概念；应该按照计算时代开始后的年、月、日或秒来实现，还是其他方式？没有答案放之四海而皆准，但在某些情况下，一个答案通常比其他答案更好。封装允许重新实现以使用新技术、技巧、要求或知识，而不会将实现者锁定在特定方法中；用户应该始终不了解更改（尤其是在接口中）。

### 问题

考虑开发一个 Date 类并添加用于打印格式化的方法，清单 9.30 中的代码显示了所选方法。选择成员方法的理由很简单：封装。每个方法都访问和修改实例变量，应该是类的一部分。重载 operator<< 通常是通过定义一个友元函数来直接访问实例变量来完成，可以为其他语言环境定义其他格式样式的成员函数。

友元函数可以直接访问私有数据成员，可视为本分析的函数成员。最佳做法是朋友永远不会改变数据。观察类的函数成员，并计算其中有多少直接访问实例变量：有 5 个。

清单 *9.30* 带有实例方法的类

```
class Date {
private:
  int year;
  int month;
  int day;
public:
  Date(int year, int month, int day) : year(year), month(month), day(day) {}
  std::string formatUS(); // 1
  friend std::ostream& operator<<(std::ostream&, const Date&);
  int getYear() { return year; }
  int getMonth() { return month; }
  int getDay() { return day; }
};

std::ostream& operator<<(std::ostream& o, const Date& d) { // 2
  o << d.year << '/' << d.month << '/' << d.day;
  return o;
}

std::string Date::formatUS() { // 3
  std::stringstream ss;
  ss << month << '/' << day << '/' << year;
```

```
  return ss.str();
}

int main() {
  Date birthday(1970, 1, 1); // smart AI will understand this
  std::cout << birthday << '\n';
  std::cout << birthday.formatUS() << '\n';
  return 0;
}
```

注释 1：5 个成员直接访问实例变量
注释 2：友元函数可以直接访问私有实例变量
注释 3：成员函数还可以直接访问私有实例变量

现在，需求突然发生了变化。在紧迫的期限内，必须将课程改为使用基于自 1970/01/01（或 01/01/1970）以来的秒数的纪元日期实现。

> **NOTE**
>
> Date 类非常难以正确实现，强烈建议使用比此示例更好的实现！

**分析**

Date 类的实现合理（仅作为示例！）并且按客户期望的方式工作，但更改的需求要求更改 private 实例变量，更糟糕的是，要求更改 5 个实例方法。封装表明这种设计是最佳的，但重新考虑这种方法将证明是有帮助的。采用不同的指标（直接访问实例变量的方法数量），并确定是否可以减少它们将证明会更好。

必须更改这三种访问器方法来处理纪元后的秒数；几乎无法改变这一事实。但如果 operator<< 和 formatUS 方法更改为完全依赖于访问器，则无需修改。具有更多方法的类中，受影响的方法数量使这个问题更加复杂。因此，衡量封装的更好指标是计算访问实例变量的方法数量。原因很简单：如果方法不访问实例变量，就无法公开封装的数据。影响的方法越少，隐藏性就越好，并且更改实现细节对其他方法的影响就越小。

**解决**

为了尽量减少实例方法的数量，尽可能多地将方法设为非成员方法。按照剩下的几个成员方法可以减少必须发生更改时的修改量。清单 9.31 展示了一个（不切实际的）实现，其中计算访问器作为 operator<< 和 formatUS 非成员函数的基础。

清单 *9.31* 最小化成员方法并实现非成员

```
class Date {
private:
  static const long sec_in_year = 31536000;
  static const long sec_in_mon = 2592000;
  static const long sec_in_day = 86400;
  long seconds;
public:
  Date(int year, int month, int day) {
```

```
    seconds = (year-1970) * sec_in_year;
    seconds += (month-1) * sec_in_mon;
    seconds += (day-1) * sec_in_day;
  }
  int getYear() const {
    return seconds/sec_in_year + 1970; } // 1
  int getMonth() const {
    int sec = seconds % sec_in_year;
    return sec/sec_in_mon + 1;
  }
  int getDay() const {
    int sec = seconds/sec_in_year/sec_in_mon;
    return sec/sec_in_day + 1;
  }
};

std::ostream& operator<<(
  std::ostream& o, const Date& d) {
  o << d.getYear() << '/' << d.getMonth() // 2
    << '/' << d.getDay();
  return o;
}

std::string formatUS(const Date& d) { // 2
  std::stringstream ss;
  ss << d.getMonth() << '/' << d.getDay() << '/' << d.getYear();
  return ss.str();
}

int main() {
  Date birthday(1970, 1, 1); // smart AI will understand this
  std::cout << birthday << '\n';
  std::cout << formatUS(birthday) << '\n';
  return 0;
}
```

注释 1：计算 getter 必须反映实现的变化

注释 2：当实现发生变化时，也要保持免疫

访问实例变量的最小方法集指导开发人员确定基集，其他方法应根据这些方法实现。这种方法将影响实例变量的方法数量降至最低，确保封装最大化。应保持面向对象编程的 this 值。

**建议**

- 通过减少成员和友元方法，最大限度地减少直接访问实例变量的方法数量。
- 访问最少数量成员方法的非成员方法是，保持封装的最佳方式——直接访问实例变量的方法越少，封装越好。
- 友元函数可能很诱人，但可能无意中修改私有实例变量的点，并且破坏封装。

## 9.13. 错误 73： 从访问器方法中错误地返回字符串

此错误主要影响性能，并对正确性有影响。这是一个经常发生的特殊问题。

std::string 类很有用，提供了重要的行为，可轻松处理文本。现代 C++ 增加了许多功能，每个新标准似乎都添加了更多有用的方法。充分利用该类非常重要，许多用户编写的类至少有一个此类型的实例变量。C++ 是少数几种不使字符串不可变的语言，因此这为正确性带来了一个问题。

### 问题

我们正在为学校设计一个记录保存系统，需要对个人进行建模。我们选择从 Person 类开始，最终从该类派生出其他类（当然，还要开发出良好的继承设计）。目前，正在处理少量的类功能。

负责这项任务的开发人员最近了解了引用的强大功能，并选择返回对 name 实例变量的引用以提高性能。引用比副本更快，但也有一些缺点。以下代码显示了第一次尝试和一些意外行为—不知何故，name 实例变量被修改了。

清单 *9.32* 对一个人进行建模并尝试改进性能

```
class Person {
private:
  std::string name;
  int age;
public:
  Person(const std::string& name, int age) : name(name), age(age) {}
  std::string& getName() { return name; } // 1
  int getAge() { return age; }
};

int main() {
  Person sam("Samantha", 26);
  std::string& name = sam.getName();
  name += "x"; // 2
  std::cout << sam.getName() << " is " << sam.getAge() << " years old\n";
  return 0;
}
```

注释 1：为了提高性能，返回实例变量的引用

注释 2：用户代码不应该直接更改私有实例数据

开发人员本意是修改数据的副本，而不是修改实例变量。没有注意到问题，代码就通过了测试。

### 分析

开发人员试图编写高性能代码，应该注意到 getName 访问器返回了对实例变量的引用；它不是数据的副本。引用是同一事物（实体）的不同名称 - 别名。如果修改了别名，则修改了实体。对高效实现的渴望掩盖了实际实例变量暴露的事实。

### 解决

返回对 std::string 实例变量的引用可防止复制该变量，从而节省内存分配和复制。但如果返回副本，则实例变量将不受用户修改的影响。开发人员的意图值得称赞，但他们的实现却不尽

如人意。这不仅限于 std::string；通过引用（或指针！）返回成员的类，都会遇到同样的问题。

获得引用的性能和私有实例变量的不可攻击性方法，可确保 std::string 变量返回 const 引用，如清单 9.33 所示。如果引用存储在某个局部变量中，则编译器也必须确保它为常量。因此，尝试通过引用修改数据会导致编译错误。此代码修复了急切开发人员的方法，同时保持了数据的完整性。

另一方面，用户代码试图更改访问器获取数据的行为非常可疑。如果修改了数据的本地副本，将不再与对象的值匹配，语义会发生变化。某些情况下，这是有意为之，但要非常小心。最好复制或引用实例数据以保留对象的语义。代码始终可以在需要时访问实例变量值，并且始终与对象保持一致。本地保存数据允许修改和更改其含义。

清单 9.33 使用常量避免修改并提高性能

```
class Person {
private:
  std::string name;
  int age;
public:
  Person(const std::string& name, int age) : name(name), age(age) {}
  const std::string& getName() const { return name; } // 1
  int getAge() const { return age; }
};

int main() {
  Person sam("Samantha", 26);
  const std::string& name = sam.getName(); // 2
  // name += "x";
  std::cout << name << " is " << sam.getAge() << " years old\n";
  return 0;
}
```

注释 1：返回对 std::string 实例变量的常量引用

注释 2：使局部变量不可变，以便编译并保留语义

**建议**

- 为了提高效率，请通过引用返回 std::string 实例变量；为了保证正确性，请确保引用是常量，以防止无意中修改实例变量。
- 请记住，引用是别名；代表实际数据，而非数据副本。

# 第 10 章 异常与资源管理

本章内容

- 异常的缺点
- 异常的优点
- 资源处理和异常
- 构建异常层次结构以实现精确控制

异常这一话题在整个领域都有支持者和反对者，开发人员主张从很少使用异常（如果有的话）到经常使用异常。多年来，一些作者发表了文章，为这一领域的各种立场辩护。其中许多论点是相关的，但其他观点不太吻合。这种情况，让我们在处理异常使用时陷入困境。

许多情况下，一种政策是有意义的，但在其他情况下却毫无意义。需要在整个程序的范围内考虑异常，解决函数行为的策略可能无法有意义地扩展到更大的单元。许多当前在生产中运行的程序需要重新设计以处理统一的异常策略。在这些情况下，任何设计都比没有设计要好，但将本地化策略集成到更大的单元中可能会容易出错。

众多的案例中，最好的方法是更好地理解例外情况，并应用这些例外情况的直觉，应用于所开发的代码库。因此，我们将从对异常，及其存在原因的进行一般性了解。从这一点开始，可以更好地理解如何在资源管理等关键情况下，如何更好地使用它们。异常和资源紧密联系在一起（以及构造函数和析构函数），以提供全面的管理模式，尤其是在夜间发生意外时。Bjarne Stroustrup 发明的惯用模式 resource allocation is initialization（RAII）将重点介绍为协调资源管理的 C++ 方式。

使用异常的一个重要方面是明确什么是错误，什么不是错误。函数是执行某种行为的一段命名代码。每个代码都有三个必须保持的特征—未能保持这些特征的行为都应视为错误：

- 先决条件失败
- 不变量失败
- 后置条件失败

函数中发现的其他问题应在本地处理。如果这些问题是或成为这三个特征的失败，则应抛出异常。明确区分错误（以异常表示）和应在本地处理的问题可能很棘手。下面给出了每种类型的快速示例，以帮助阐述各种可能性：

- 先决条件失败 – 向函数传递一个指向链接列表的指针以搜索键，但指针为 NULL(可能等于零)。由于函数不可能继续，因此必须发出错误信号。如果函数设计为处理空列表（NULL 指针），则这不是错误，应该返回一个值，表明未找到该键。
- 不变量失败 – 类实例维护表示日期的状态。构造函数使用先决条件检查验证了范围。实例尚未被销毁，所以类不变量有效且不可违反。但某些方法修改了日期状态，导致日期超过了 31。另一种方法尝试转换此日期，但使用不变量检查发现了错误。由于检测到错误，因此应该发出信号。由于类本身修改了值，因此这是一个糟糕的编程案例！幸运的是，不变量检查传达了这种情况。
- 后置条件失败 – 函数接收指向包含要转换为数字的文本的缓冲区的指针，但文本与数值不一致。由于函数无法返回有意义的数值，因此必须发出错误信号。如果函数旨在返回非值的

数字表示形式（假设为浮点 NaN），则返回该值，并且不会出现错误。

这三个特征一致且可预测地考虑其他情况。应该清楚的是，函数设计需要考虑这些，并且应仔细考虑什么是错误，什么不是错误。避免临时的错误策略——同情所有有遗留问题并必须清理的人。

## 10.1. 使用异常

异常的主要目的是断言发生了错误，并且调用的代码无法成功完成请求。此错误以开发人员无法忽略的方式出现。传统的错误处理通常是设置某种状态值，来指示发生了错误，并且调用代码会针对这种情况进行测试；但没有办法强制执行错误检查。

异常会引起开发人员的注意。如果抛出异常而开发人员忽略，程序最终会崩溃。开发人员必须处理异常以避免崩溃——不可忽略的特性迫使开发人员解决这个问题。希望开发人员能够仔细考虑错误的含义，以及从错误中恢复可能需要做什么。换句话说，异常为设计具有弹性的软件提供了重要的机会。

需要考虑的异常时空方面有三个，依次为：

1. 功能调用点
2. 错误检测点
3. 恢复点

调用点首先发生。调用一段功能性代码，调用者等待结果（我们这里只考虑同步执行）。错误检测点随后发生。这是直接检测错误情况的操作发生的地方。调用点后面的代码开始可选发生的恢复点。

对于经典的返回代码，这将是对返回值的检查；对于异常，是相关的捕获块。

这三个站点在空间上是分开的（不同的功能），当调用代码确定它无法有意义地继续时，就会检测到错误。调用者要求被调用者执行一个行为，但调用者被到绝境，必须承认它无法执行。许多情况下，调用者的返回值都没有意义，必须在替代控制路径中发出错误信号。最后，恢复代码是空间和时间上的第三个站点。这个可选执行的代码试图从错误中恢复。如果可以，程序继续；如果不能，应该正常终止。

以下是对异常的几种肯定和反对意见。我们考虑了每种反对意见，并给出了使用异常的支持。简而言之，大多数对异常的反对意见与异常的正确使用关系不大；相反，它们反对的是不当使用。与任何技术一样，不当使用是有问题的，但不能成为拒绝正确使用的理由。

### 10.1.1. 肯定：混合控制和恢复路径

传统的错误处理将控制和恢复路径混在一起；异常将它们分开。没有问题的正常执行路径（快乐路径）通常设计得很好，大多数开发者编写代码来执行行为。太频繁了，只有在快乐路径编码并工作后，错误检测和处理路径才会优先考虑——有时，只会得到一个粗略的一目了然。坦白地说，错误检测和错误处理代码经常会遮掩住主控制路径，从而难以理解和推理。

这种代码混淆，可能会让一些开发人员不太注意错误检测和处理的必要性。人们很容易希望事情能够正常工作（至少在大多数情况下），而不必过分强调错误。此外，当调用的代码不太容易理

解，或者调用代码隐藏在可能不太容易理解的条件时，处理错误通常只是猜测。无法清楚地理解错误的含义，可能会降低处理导致错误的能力。

### 10.1.2. 反对：混淆异常处理和正常错误处理

在错误处理情况下，异常会过度使用。这里，区分调用点和检测点至关重要。检测错误的代码不应该直接抛出异常。相反，当检测点确定发生了错误时，必须决定是否可以处理它。如果可以，就没有必要抛出异常。当抛出异常时，读者可以正确地争辩说检测点代码应该处理这个问题。然而，当检测站点无法处理错误时，应该抛出异常。包括检测点在内的代码应该执行某种行为。当不能处理时，必须通过某种机制来发出信号表明其无能为力。通常，这是由错误返回码表示。但同样，没有什么可以强制调用点检查返回码。

使用异常来表示函数无法满足调用者的需求，是使用它们的正确理由。使用异常来要求调用者修复调用代码本应存在的问题是令人反感的，不应这样做。

### 10.1.3. 反对：难以改进异常处理

大量现有代码中添加异常处理太难了。无论是否反对改造现有代码以添加异常处理，这都不是一个好主意。需要通过引入不一致、低效率和阅读和理解挑战，来将异常处理代码添加到逻辑中。现有代码可能没有返工的预算，但在某些情况下，可能会有返工的预算。新代码中设计一致的异常处理策略具有挑战性，在现有代码中则更困难。请尽你所能，但在这些情况下不要做得过火。

### 10.1.4. 肯定：歧义值

异常将正常值与错误值区分开来。返回代码取决于定义的数据类型、该类型的特定值，以及用于检查该值的代码。该函数返回其定义的返回类型的值。返回代码是哨兵值。哨兵值与数据类型一致，但与问题不一致。假设一个函数返回一个整数值，确定与问题不一致的整数值可能很容易，但更有可能不会如此。必须确定单个值或一组值来表示良好的返回；必须指定不同的值或一组值来解决错误的返回。哪个是哪个？谁需要记住这些？并且能保证解释这些的注释和代码一致吗？

抛出异常是明确的，并且不依赖于函数的返回类型。异常类型包含信息。此外，异常可能还包含描述错误细节的附加文本信息。

### 10.1.5. 肯定：模棱两可的数据类型

异常可确保值的数据类型在正常和错误条件下不同，使用返回代码或设置全局变量的传统方法，必须遵循返回代码的类型，该类型也用于正常处理。异常可以抛出与返回类型无关的完全不同的类型，异常的类型消除了返回数据类型的含义。

### 10.1.6. 肯定：强制错误处理

使用异常可确保错误得到处理。异常不会强制执行适当的错误处理，仅确保捕获每个异常或终止程序。提供了有意义地处理错误的机会；开发人员必须充分利用。异常的一个显著好处是，简化了将错误渗透到更高级别的过程。这些级别通常可以更好地处理问题。使用返回代码会使渗透变得复杂，并且在每个级别进行测试时会令人困惑。

如果第一级错误处理不能充分处理错误，则只需稍微努力就可以重新抛出异常，或将其转换为另一个键入并抛出该异常。唯一的编码要求是每个级别，都有一些可以匹配抛出的异常的 `catch` 块。错误处理的替代控制结构简化了代码，并使推理更加清晰。

### 10.1.7. 反对：提供恢复处理程序

`Catch` 块不必具有有意义的恢复处理。这几乎不是一个重大的反对意见，编写错误恢复代码都没必要，无论是异常还是经典方式。捕获异常的第一个块可以确定出了什么问题，以及是否可以恢复。恢复可能意味着调用函数修改的所有状态要么回滚，要么初始化到有意义的点。毫无疑问，可以忽略更改并继续前进，开发人员必须考虑这一点。必须遵守类的不变量，并在必要时恢复。调用代码之后的恢复代码，总是能更好地掌握可能已更改的内容，并需要清理的内容。发现本地的代码需要更多知识，才能进行有意义的恢复。

### 10.1.8. 肯定：转变失败类型

为正常和错误情况使用单独的控制路径，允许开发人员在飞行过程中转换异常类型。如果检测点以一种方式理解错误，则恢复点可以将该语义转换为另一种含义。假设正在从磁盘文件读取一条记录，输入验证例程在将文本转换为特定字段的整数时检测到无效数据。它会抛出异常，指出发现了无效数据，转换无法成功。如果恢复点决定不恢复，可能希望将此故障推到另一个级别，添加元数据或其他诊断信息。较高级别应该了解一些比低级别细节更一般的信息（例如，输入不匹配错误）。因此，第一个恢复点可以将异常类型更改为更适合下一级恢复点的内容，可能是无效记录错误。日志记录可用于捕获细节，这可能不适合实际恢复。

### 10.1.9. 肯定：分离关注点

能够在错误检测和错误处理之间分离关注点，让多个开发人员能够在不干扰彼此代码的情况下就近工作。混合正常和错误处理控制路径会混淆关注点，处理正常路径的代码与处理错误路径的代码的工作方式不同。因此，将这些路径分离为两个不同问题的能力，可以使每个关注点彼此隔离。这是一个强大的软件工程原则，只要可以应用，就应该尽可能地加以利用。

### 10.1.10. 反对：鼓励预先设计

代码必须从一开始就设计为处理异常，这需要花费时间和精力。这种反对意见可能是允许在未进行适当设计之前，或未进行适当设计的情况下编写代码的原因，这与异常无关。新代码和尽可能

多的遗留代码必须经过适当设计，以处理正常和错误路径处理的分离。

## 10.2. 错误 74： 构造函数中抛出异常

这个错误关注的是正确性和有效性。由于保持类不变性至关重要，从失败的构造函数中抛出异常是避免使用部分初始化对象的最佳方法。异常使创建对象的代码更简洁，但更重要的是，构造失败都会发出信号并必须处理。很容易忘记检查返回代码和失败标志，或一致地实施其他方案来测试对象的有效性。这种情况下，开发人员使用异常时花费的开发时间比不使用时少。

**问题**

除非经过专门的设计，否则许多现有代码很少能很好地使用异常。一个值得注意的领域是对象的构造。默认构造函数可能被过度使用，导致某些变量实例初始化不当。糟糕的是，构造缺少初始化数据的对象可以接受，因为能够提供（稍后！）其余的初始化数据。

这种错误的方法通常使用默认构造函数，或采用初始化数据子集的构造函数。当初始化数据缺失时，将重置有效位以指示对象不完整或部分初始化。当提供缺失数据时，将设置有效位，指示已完成初始化。如果始终如一地应用，这种方法是有效的，但可能表明决策和设计不当。以下代码显示了此假设的实际作用。

清单 *10.1* 部分初始化的类

```
class Person {
private:
  std::string name;
  int age;
public:
  Person(const std::string& name) : name(name) {} // 1
  Person(const std::string& name, int age) : name(name), age(age) {}
  int getAge() { return age; }
  const std::string& getName() const { return name; }
  bool isValid() const { return age > -1; }
};

int main() {
  Person sally("Sally"); // 2
  std::cout << sally.getName() << " is " << sally.getAge()
    << " years old\n";
  Person brian("Brian", -1); // 3
  std::cout << brian.getName();
  if (brian.isValid()) // 4
    std::cout << " is " << brian.getAge() << " years old\n";
  else
    std::cout << " is invalid\n";
  std::cout << brian.getName() << " is " << brian.getAge() <<
    " years old\n"; // 5
  return 0;
}
```

注释 1：部分默认构造函数无法初始化一个实例变量
注释 2：使用部分默认构造函数；age 未初始化
注释 3：实例状态中引入了不良数据
注释 4：开发人员检查了有效位
注释 5：开发人员忘检查有效位

sally 对象已部分初始化，age 实例变量将受制于当前内存中的值，其行为未定义。brian 对象已完全初始化，但将坏数据引入实例。这两种情况下，都需要测试这些对象的有效性。假设 sally 的 age 的随机值为正，则该对象看起来有效。如果执行检查，brian 对象将无法通过有效性检查，但没有办法强制检查。

### 分析

使用这些对象的代码无法保证知道这些对象是有效的，并且类不变量已得到遵守。尽管引入了有效性检查来提供这种保证，但可能不会调用——这将是另一个未经过深思熟虑的好主意。

### 解决

第一道防线是仅在知道所有必要的初始化数据时，才创建对象。这一点是过度使用默认构造函数的另一个理由。类不变量依赖于两部分数据，这两者都是必需的。此外，age 实例变量的范围是有界的。除非这些检查得到验证，否则对象的状态未知。

该类应该验证两个参数的值，以确保状态可以得到保证，如有任何不妥，应该抛出一个异常，以明确说明无法在有意义的状态。调用代码应始终清楚对象的有效性；类负责确保和执行它。以下代码显示了执行类不变式，并传达建立时的失败。

清单 *10.2* 为构造失败抛出异常

```
class Person {
private:
  std::string name;
  int age;
  static const std::string& validateName(const std::string& name) { // 1
    if (name.empty())
      throw std::invalid_argument("name must not be empty");
    return name;
  }
  static int validateAge(int age) { // 1
    if (age < 0)
      throw std::out_of_range("age must be non-negative");
    return age;
  }
public:
  Person(const std::string& name, int age) :
    name(validateName(name)), age(validateAge(age)) {} // 2
  int getAge() const { return age; }
  const std::string& getName() const { return name; }
  bool isValid() const { return age > -1; }
};

```

```
int main() {
  // Person sally("Sally"); // 3
  Person sally("Sally", 27);
  std::cout << sally.getName() << " is " << sally.getAge()
    << " years old\n";
  Person brian("Brian", -1); // 4
  std::cout << brian.getName() << " is " << brian.getAge()
    << " years old\n";
  return 0;
}
```

注释 1：用于验证参数数据的验证方法

注释 2：构造函数负责建立类不变量

注释 3：不完整的参数值将无法通过编译

注释 4：构造期间抛出异常；不允许模棱两可

由于删除了该函数，无法再使用部分默认构造函数创建 sally 对象。现在必须提供两个参数值。brian 对象提供两个参数值，但其中一个无效。validateAge 方法确保不使用负值来初始化 age 实例变量。但在 brian 中，提供的信息会导致抛出异常。此对象的有效性没有歧义，调用者应该提供恢复点来处理这种情况。

**建议**

- 所有需要的信息都已知之前，切勿构造对象；如果信息不足，则无法建立类不变量。
- 如果构造信息无效或其他故障导致构造函数无法正确完成，则抛出异常。
- 预测构造故障；相应地规划和处理。

## 10.3. 错误 75： 析构函数中抛出异常

这个错误关注的是正确性，会影响有效性，但这是确保正确操作必须付出的代价。异常是发现和处理错误的一种宝贵手段。有人认为，当参数值无效时，构造函数应该快速抛出。部分构造的对象对正确性来说是一个真实的危险；因此，在析构函数中遵循这个建议是可行的。毕竟，如果析构函数检测到错误，有什么比抛出异常更好的方法来发现它呢？这种直觉值得称赞，但实现却并非如此。

**问题**

假设一个热切的开发者已经学会了一些关于抛出异常的要点，这些知识被其扩展到了析构函数。清单 10.3 中的代码展示了开发者尝试应用这些知识，来为他们的代码生成一个健壮，且一致的错误处理策略的结果。假设在构造之后的某个时间会添加一个 Paragraph 对象。析构时，该类会删除一个动态的 Paragraph 对象。如果在没有有效 Paragraph 的情况下销毁了 Page ，则将其视为错误。这种情况似乎是抛出异常的理想时机。

清单 *10.3* 异常的过度使用

```cpp
struct Paragraph {};

class Page {
private:
  std::string title;
  Paragraph* pgph;
public:
  Page(const std::string& title) : title(title), pgph(0) {}
  ~Page() {
    if (pgph == 0)
      throw std::string("destructor"); // 1
    delete pgph;
  }
};

int main() {
  try {
    Page p("Catching Up"); // 2
  } catch (const std::string& ex) {
    std::cout << "Exception caught: " << ex << '\n';
  }
  try {
    Page p("Trouble Ahead"); // 3
    throw std::string("try block");
  } catch (const std::string& ex) {
    std::cout << "Exception caught: " << ex << '\n';
  }
  return 0;
}
```

注释 1：从析构函数中抛出异常
注释 2：对象正常销毁
注释 3：对象错误销毁

代码产生以下输出（不同的编译器和系统可能看起来有些不同）：

```
Exception caught: destructor terminate called after throwing an instance of
↪  'std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >'
Aborted
```

程序在"Trouble Ahead" try 块中毫不客气地终止。第一个 try 块成功了——或者更准确地说，没有失败——但它的执行是一个幸运的意外，允许一些险恶的东西潜伏着。其他编译器和系统可能会有不同的行为。

**分析**

第一个 try 块之所以能正常工作，是因为没有正在进行的异常。第一个 catch 块可以捕获异常并按预期进行处理。这种看似正确的操作具有误导性；当策略失败时，会让开发人员吃惊。

第二个 try 块是那个暂停的惊喜。Page 对象是在 try 块中创建的，但在正常退出块时销毁它之前会引发异常。以下 catch 块应该处理该异常；但必须先销毁 Page 对象。调用 Page

析构函数，并且它遇到错误。因此，它引发异常以表明，它缺少 Paragraph 对象。

当异常正在进行时，第二个异常抛出会导致调用 terminate 函数。std::terminate 函数调用的默认行为是，通过调用 std::abort 在该点中止执行而不展开堆栈。abort 函数确保调试器（例如，dbg）可以在终止点看到程序的确切状态。如果堆栈展开并且要执行其他清理函数，调试将缺少其大部分上下文。

出乎意料的是，这个程序应该以这种方式运行。开发者没有意识到抛出一个当另一个正在执行时，将引发异常。析构函数特别容易受到这种危险的影响，它们在两种情况下调用：正常终止和错误终止。当在错误条件下调用抛出的析构函数时，就会出现这种意外行为。

**解决**

最好的方法是避免从析构函数中抛出异常；这可以完全避免问题，因为异常发生的概率大到足以成为问题。但某些情况下，析构函数中抛出异常。如果析构函数执行抛出的代码，则析构函数无法控制该行为，并可能成为终止调用的受害者。

对于可能（或实际）抛出的析构函数代码，必要的方法是将代码包装在 try 块中。清单 10.4 中的代码展示了这种方法；它捕获异常，并大概记录问题，但不会重新抛出当前异常或抛出另一个异常。通用的 catch 规范将捕获任何异常；如果特定处理需要更多特异性，请在通用 catch 之前添加。关键是任何异常都不应逃过析构函数。

清单 *10.4* 析构函数中抛出异常，而不是从析构函数抛出异常

```
struct Paragraph {};

class Page {
private:
  std::string title;
  Paragraph* pgph;
public:
  Page(const std::string& title) : title(title), pgph(0) {}
  ~Page() {
    try {
      if (pgph == 0)
      throw std::string("destructor"); // 1
    delete pgph;
    } catch (...) { // 2
      std::cout << "ERROR: exception captured in destructor\n";
    }
  }
};

int main() {
  try {
    Page p("Catching Up"); // 3
  } catch (const std::string& ex) {
    std::cout << "Exception caught: " << ex << '\n';
  }
  try {
```

```
    Page p("Trouble Ahead");
    throw std::string("try block"); // 4
  } catch (const std::string& ex) {
    std::cout << "Exception caught: " << ex << '\n';
  }
  return 0;
}
```

注释 1：仍在析构函数中抛出
注释 2：捕获析构函数中的抛出，避免其离开函数体
注释 3：对象的正常销毁
注释 4：对象的错误销毁

**建议**

- 不要从析构函数中抛出异常；异常很可能已经发生，这将导致程序立即终止。
- 如果析构函数中的代码可以抛出异常，则将其包装在 `try` 块中以捕获它，并将其转换为日志（或类似的）消息；不要重新抛出异常。

## 10.4. 错误 76： 使用异常时的资源泄漏

此错误会影响正确性。当从已分配动态资源的构造函数中抛出异常时，会导致资源泄漏，从而使系统处于受损状态。正确性不仅仅是程序内的正确计算，还必须考虑对整个系统的影响。

动态资源是一种简单的方法来访问和使用各种资源，而这些资源作为值实体时效果不佳。一个简单的例子是长度未知的数组，其大小在运行时确定，并且数组是动态分配的。如果出现异常，处理这些资源会变得更加复杂。

**问题**

假设开发人员正在构建一个需要页眉和页脚的 `Page` 实体，组成每个文本的文本位于硬盘文件中。系统资源是变体行为的来源。文本文件可能存在也可能不存在；并且可能会出现读取或写入错误、意外数据和其他一些问题。

清单 10.5 中的代码展示了一种处理页眉和页脚文本的幼稚尝试。首先，应该优先使用 `std::string`，而非 C 样式的字符串，但不是每个人都有这种特权。其次，预期的错误（例如读取文本文件时）应该在本地处理，并且只有在没有很好的问题解决方案时才应抛出异常。虽然开发人员对这个问题都有不同的看法，但本章中的代码接受了开发人员的方法。第三，输入缓冲区的硬编码长度充满了潜在的问题。黑客喜欢这种错误，这样的安全漏洞会让他们有机会破坏原本安全的代码。

清单 *10.5* 尝试在析构函数中处理动态资源

```
class OneLiner {
private:
  std::string filename;
  char* text;
public:
```

```
  OneLiner(const std::string& filename) : filename(filename), text(0) {
    text = new char[64];
    std::cout << filename << " allocated\n";
    std::ifstream file(filename.c_str());
    if (file.fail())
      throw std::string("file " + filename + " inaccessible");
    // read data into text
  }
  ~OneLiner() { std::cout << filename <<
    " ~deallocated\n"; delete [] text; } // 1
    char* getText() const { return text; }
};

class Page {
private:
  std::string title;
  char* header;
  char* footer;
public:
  Page(const std::string& title, const std::string& headerfile, const
  std::string& footerfile) :
  title(title), header(0), footer(0) {
    try {
      OneLiner head(headerfile); // 2
      header = head.getText();
      OneLiner foot(footerfile); // 3
      footer = foot.getText();
    } catch(const std::string& ex) {
      std::cout << ex << '\n';
    }
  }
};

int main() {
  Page chapter("Introduction to C++", "header.txt", "footer.txt");
  return 0;
}
```

注释 1：调用时释放动态内存

注释 2：正确清理动态内存

注释 3：动态内存泄漏，而不是清理

正常情况下，OneLiner 的析构函数代码可以正确处理动态访问资源的释放；然而，错误条件会彻底搞乱这一切。当异常导致 try 块终止时，head 对象将销毁。foot 对象应该以相同的方式清理，但这里的表象是骗人的。以下输出显示了发生的情况；请注意，只有一个释放：

```
header.txt allocated
footer.txt allocated
header.txt ~deallocated
```

```
file footer.txt inaccessible
```

### 分析

当 head 对象创建后，动态内存被分配，文件对象成功打开，读取操作可以继续。构造函数完成，并且 head 对象完全构造。

创建 footer 对象时会分配动态内存，但打开文件对象时会失败。由于构造函数无法采取补救措施，因此失败会引发异常。它确定它无法成功，因此会抛出异常。构造函数执行不完整，这就是问题的开始。

当 main 函数运行 try 块时，会成功创建 head 对象。当尝试创建 footer 对象时，会引发异常。try 块会立即结束，并执行 catch 块。但执行 catch 块之前，必须展开堆栈以清理在其范围内创建的资源。head 对象销毁，释放动态内存。footer 对象从未完全构造，因此不会被清理。在部分构造的对象上调用析构函数会带来很多麻烦，因为不清楚如何正确销毁部分对象。因此，析构函数不会清理此对象。footer 对象泄漏了动态分配的内存。

### 解决

解决方案很简单：在从构造函数中抛出异常之前，手动释放所有分配的动态资源—简单！读起来很简单，但这种方法可能会引入重复的清理代码、奇怪或复杂的清理逻辑，并且可能会丢失一些资源，具体取决于构造的复杂性。以下代码在抛出异常之前释放动态内存，从而防止泄漏。

清单 *10.6* 抛出之前清理动态资源

```
class OneLiner {
private:
  std::string filename;
  char* text;
public:
  OneLiner(const std::string& filename) : filename(filename), text(0) {
    text = new char[64];
    std::cout << filename << " allocated\n";
    std::ifstream file(filename.c_str());
    if (file.fail()) {
      std::cout << filename << " deallocated dynamic memory\n";
      delete [] text; // 1
      throw std::string("file " + filename + " inaccessible");
    }
    // read data into text
  }
  ~OneLiner() {
    std::cout << filename << " ~deallocated\n";
    delete [] text;
  }
  char* getText() const { return text; }
};

class Page {
```

```
25  private:
26    std::string title;
27    char* header;
28    char* footer;
29  public:
30    Page(const std::string& title, const std::string& headerfile, const
31    std::string& footerfile) :
32        title(title), header(0), footer(0) {
33      try {
34        OneLiner head(headerfile); // 2
35        header = head.getText();
36        OneLiner foot(footerfile); // 3
37        footer = foot.getText();
38      } catch(const std::string& ex) {
39        std::cout << ex << '\n';
40      }
41    }
42  };
43
44  int main() {
45    Page chapter("Introduction to C++", "header.txt", "footer.txt");
46    return 0;
47  }
```

注释 1：抛出之前删除动态资源

注释 2：正常的析构函数调用来释放内存

注释 3：该错误导致基于构造函数的内存释放

以下输出显示了基于构造函数的释放的结果。结果与预期一致：

```
header.txt allocated
footer.txt allocated
footer.txt deallocated dynamic memory
header.txt ~deallocated
file footer.txt inaccessible
```

虽然这种方法很可爱，但只有在无法做更简单、更优雅的事情时才使用这种方法。下一个错误推荐了 C++ 的发明者开发的一种模式；如果 Bjarne 说要使用它，我们就用它吧！

潜在的困难在于 OneLiner 类具有多项职责，分配内存并打开和读取文件。如果有一种方法可以通过将分配和释放的琐事推给另一个类来最小化其职责，生活可能会更简单。RAII 模式就是这样做的。因此，更倾向使用 RAII ，而非这个在可以的时候使用技术，还请在必须的时候修改构造函数。

现代 C++ 提供了智能指针来更简洁地解决这个问题。std::unique_ptr 和 std::shared_ptr 实现了 RAII 模式，免去了繁重工作的必要。

**建议**

- 确保在通过异常离开构造函数之前正确删除所有分配的动态资源。
- 确保析构函数处理所有动态资源。

- 只有完全构造的对象才会调用其析构函数。
- 尽量使用 RAII。

## 10.5. 错误 77: 未使用 RAII 模式

这个错误关注的是正确性和可读性。使用简单的模式来实现重要的语义概念，通过在所有条件下删除动态或有限的资源来保持正确性。

动态和有限资源具有所有权限制，管理它们具有挑战性。编译器无法通过检测问题来提供帮助。开发人员有责任将每个分配与其相应的释放配对。当抛出异常时，这种配对会很复杂。正确的设计必须考虑正常和错误路径。错误 76 详细讨论了这种情况并提供帮助。更好的是，有一个通用解决方案，可以让开发人员从管理代码和将动态资源管理推向代码中解脱出来。最通用的解决方案是，制作遵循 RAII 模式的资源管理类。

### 问题

考虑这样一种情况：需要一个动态数组来保存一组测试分数，而这些分数的大小在编译时是未知的。用户代码确定测试分数的数量，分配数组，将分数输入数组，并将该信息传递给构造函数。构造函数（不必要地）分配其数组，复制输入值，并计算平均值。开发人员知道负值可能是一个问题，进行了适当的测试，如果发现错误值，则会引发异常。

以下代码试图做正确的事情，但必须明确在一个函数中创建的数据应如何传输到另一个函数。这种尝试部分解决了问题，但很笨拙。

清单 *10.7* 出现异常时的资源泄漏

```
class Grades {
private:
  int size;
  double* grades;
  double average;
public:
  Grades(int size, const double* grades) : size(size), grades(0),
         average(0) {
    if (size == 0)
      return;
    this->grades = new double[size]; // 1
    for (int i = 0; i < size; ++i) {
      if (grades[i] < 0) // 2
      throw std::invalid_argument("negative score");
      this->grades[i] = grades[i];
    }
    double sum = 0.0;
    for (int i = 0; i < size; ++i)
      sum += this->grades[i];
    average = sum/size;
  }
  ~Grades() { delete [] grades; }
```

```
  double getAverage() const { return average; }
};

int main() {
  int count;
  std::cout << "Enter number of items to average: ";
  std::cin >> count;
  double* scores = new double[count];
  for (int i = 0; i < count; ++i) {
    std::cout << "Enter score: ";
    std::cin >> scores[i];
  }

  try {
    Grades g(count, scores);
    std::cout << "Average score is " << g.getAverage() << '\n';
  } catch (const std::invalid_argument& ex) {
    std::cout << ex.what() << '\n';
  }
  delete [] scores;
  return 0;
}
```

注释 1：分配动态资源

注释 2：抛出前未能删除资源

### 分析

如果发现负值，构造函数将抛出异常。grades 资源不会被释放，从而导致资源泄漏。代码中至少有三个错误。首先，构造函数分配了一个动态数组，然后测试无效条件，抛出异常而不考虑释放资源。其次，构造函数正在计算平均值，这不是它的职责。发生了太多事情掩盖了代码的意图，使得查看资源问题更加困难。第三，没有必要分配动态数组。数组副本对于正确访问测试分数数据并非必须，有更好的方法来获取所有权。

勇敢的开发人员读了一篇关于 RAII 的文章，并决定更新代码以解决其建议。将输入值数组传递给实现 RAII 的 Grades 类，以使实例能够管理数据。第一次尝试如清单 10.8 所示，其中构造函数不再分配动态资源，而是得到了所有权。

有几篇文章谈到了这种方法，指出构造函数不必创建动态资源，但可以承担所有权。这是好的建议，也是正确的建议；然而，共享资源的问题仍然存在，资源的创建者必须将完全所有权传递给 RAII 类。以下代码显示了这种混乱。

清单 *10.8* 带有共享资源的 *RAII*

```
class Grades {
private:
  int size;
  double* grades;
public:
```

```
  Grades(int size, double* grades) : size(size),
  grades(grades) {} // 1
  ~Grades() { delete [] grades; } // 2
  double getAverage() const {
    if (size == 0)
      return 0.0;
    double sum = 0.0;
    for (int i = 0; i < size; ++i) {
      if (grades[i] < 0)
        throw std::invalid_argument("negative score");
      sum += grades[i];
    }
    return sum/size;
  }
};

int main() {
  int count;
  std::cout << "Enter number of items to average: ";
  std::cin >> count;
  double* scores = new double[count]; // 3
  for (int i = 0; i < count; ++i) {
    std::cout << "Enter score: ";
    std::cin >> scores[i];
  }
  try {
    Grades g(count, scores);
    std::cout << "Average score is " << g.getAverage() << '\n';
  } catch (const std::invalid_argument& ex) {
    std::cout << ex.what() << '\n';
    delete [] scores; // 4
  }
  return 0;
}
```

注释 1：通过接收获取资源

注释 2：在所有条件下删除获取的资源

注释 3：创建原始资源

注释 4：在错误条件下删除原始资源

这项努力值得称赞，但需要尊重独特资源的语义。如果调用代码希望将所有权传递给 RAII 类，绝不能事后管理或访问该资源—必须依赖于类授予的访问权限。执行此代码表明在错误条件下会发生双重释放，当代码删除指针并随后再次删除同一指针时，就会发生双重释放，从而导致未定义的行为。当发生这种情况时，我的系统会崩溃，这提醒这里真的有一个问题。

遵循 RAII 模式的析构函数正确地删除了资源，但开发人员误以为错误情况是，让原始所有者执行显式删除的原因——因此，进行了双重删除。如果没有抛出异常，RAII 类将正确处理所有事情。这种情况必须测试正常和错误路径，以确保正确性。

**解决**

RAII 背后的核心思想是构造函数应该分配动态资源。许多情况下，最好的方法是将其作为构造函数的唯一职责（除了其初始化列表中所做的事情之外）。某些情况下，这可能不切实际，因此应该修改方法，以便获取动态资源是构造函数的最后一项任务。这样，只有在获取资源（资源获取）时，初始化阶段（即初始化）才会完成，从而避免在出现异常时发生资源泄漏。如果构造函数成功完成，则将在正常和错误条件下调用析构函数。如果获取失败，则不会调用析构函数；这种方法是正确的，没有资源可以释放。这种全有或全无的方法是 RAII 模式的核心。

以下代码有一个主要警告：RAII 类不完整，没有解决复制构造函数和复制赋值操作符。此外，一个好的 RAII 类可能会实现指针访问操作符。所有此类操作都必须进行编码以避免灾难，使用默认操作符来完全实现 RAII 类。

清单 *10.9* 具有独占所有权的 *RAII*

```
class Grades {
private:
  int size;
  int next;
  double* grades;
public:
  Grades(int size) : size(size), next(0), grades(0) {
    this->grades = new double[size]; // 1
  }
  ~Grades() { delete [] grades; } // 2
  void addGrade(double grade) { grades[next++] = grade; }
    double getAverage() const {
    if (size == 0)
      return 0.0;
    double sum = 0.0;
    for (int i = 0; i < next; ++i) {
      if (grades[i] < 0)
        throw std::invalid_argument("negative score");
      sum += grades[i];
    }
    return sum/size;
  }
};

int main() {
  int count;
  std::cout << "Enter number of items to average: ";
  std::cin >> count;
  Grades g(count); // 3
  for (int i = 0; i < count; ++i) {
    std::cout << "Enter score: ";
    double grade;
    std::cin >> grade;
```

```
    g.addGrade(grade);
  }

  try { // 4
    std::cout << "Average score is " << g.getAverage() << '\n';
  } catch (const std::invalid_argument& ex) {
    std::cout << ex.what() << '\n';
  }
  return 0;
}
```

注释 1：初始化只包括资源分配
注释 2：只在构造函数成功时调用
注释 3：由 RAII 类获取动态资源
注释 4：正常或错误控制路径都调用析构函

动态和有限资源（例如，数据库连接句柄）在除最小程序之外的所有程序中都很常见。开发者会努力确保他们的代码正常运行，但很容易忽视程序与系统交互的问题。必须考虑这种交互以确保程序正确性。管理动态和有限资源可能很复杂，但 RAII 模式提供了一种将管理机制推入资源拥有类的方法。只要有可能，就应该将管理责任转移到仅用于管理资源的类中。RAII 模式确保所有成功获取的资源都会在正确的时间自动删除，而无需开发者添加逻辑。

我一直在努力完全理解短语 resource acquisition is initialization，尽管这个概念的实现很简单。有人建议改用语义更精确的短语 scope-bound resource management（SBRM，范围受限的资源管理）。我修改了这个短语，添加了一些解释性文字，使其更明确：资源获取是（此对象的目的）初始化。我听说 StackOverflow 上有人使用了短语 destruction is resource relinquishment（破坏就放弃资源），这句话非常切中要害。Phil Karlton 说过，计算机科学中只有两件难事：缓存失效和命名。所以，我们就继续吧。

**建议**

- 使用 RAII 管理动态和有限的资源；该模式可确保自动释放所有获取的资源。
- 限制 RAII 类构造函数仅尽可能获取资源；如果不能，则将获取作为最后一站。
- 如果构造函数传递了资源，请确保调用代码永远不会再次访问该资源—是 RAII 类管理的独占资源。

## 10.6. 错误 78： 使用指针管理资源

此错误重点在于正确性和有效性。不理解，不熟悉该习语的读者可能会比较难受。

当开发人员使用指针，并且必须通过所有正常和错误路径管理动态和有限资源时，管理这些资源会很困难。在自动处理管理的类中，实现的 RAII 模式是充分解决这些复杂性的一大步。但每次需要管理另一个资源时，都必须开发一个新类。

使用指针来管理动态资源是一种自 C 时代以来使用多年的模式，可以在 C++ 代码中找到。通过原始指针管理资源的困难，促使人们提出了一种称为 auto_ptr 的解决方案。这种类型的指针旨在管理原始指针的困难，但它造成了更重大的问题。

### 问题

许多遗留代码中都存在多个指向资源的原始指针，这些指针遍布整个代码库。之前的错误已经讨论了使用它们进行资源管理的一些缺陷。有人建议使用 RAII 模式，但必须开发新的类来管理每种资源类型。考虑到遵循该模式的开销，摆脱原始指针似乎是一件令人望而生畏的事情。

清单 10.10 中的代码是通过原始指针管理的一些资源的简化视图。假设进行更改很棘手，并且通常不被上层技术资源和管理层所接受。除了研究代码并模拟正常和错误路径之外，开发人员几乎没有其他选择。代码是正确的，但开发人员花费了太多时间思考和编写代码来管理 Student 实例。很有价值，但目的不是解决问题。

清单 *10.10* 使用原始指针管理资源

```
struct Student {
  std::string name;
  double gpa;
  Student(const std::string& name, double gpa) : name(name), gpa(0) {
    if (gpa < 0)
      throw std::invalid_argument("gpa is negative");
  }
};

int main() {
  Student* sammy = NULL; // 1
  Student* ginny = NULL;
  Student* gene = NULL;
  try {
    sammy = new Student("Samuel", 3.75); // 2
    ginny = new Student("Virginia", 3.8);
    gene = new Student("Eugene", -1);
  } catch (const std::invalid_argument& ex) {
    std::cout << "exception caught\n";
    if (sammy) // 3
      delete sammy;
    if (ginny)
      delete ginny;
    if (gene)
      delete gene;
    std::exit(1);
  }
  if (sammy) // 4
    delete sammy;
  if (ginny)
    delete ginny;
  if (gene)
    delete gene;
  return 0;
}
```

注释 1：正常路径和错误路径上都必须有一个指针

注释 2：创建实例并分配给指针
注释 3：错误路径清理，重复
注释 4：正常路径清理，重复

### 分析

虽然代码按设计正确运行，但相当一部分仅管理基于指针的资源。原始指针在大部分代码中随处可见，可以使用 RAII 模式将其管理最小化。此代码设计为在抛出异常时退出，所以正常路径和错误路径不，能共享清理代码。这一事实要求重复清理代码。可以编写一个包含清理的函数，并从两个地方调用，但这并不能从根本上改变这种情况。

经典 C++ 提供了一个实现 RAII 的通用类，可用于基于指针的资源。auto_ptr 模板实现 RAII 模式，并管理指向资源的指针。动态资源在 auto_ptr 之外创建，而不在其构造函数中创建。

auto_ptr 构造函数独占指针，以避免出现上一个错误中提到的问题。清单 10.11 中的代码建议使用 auto_ptr 类进行彻底修复，但此类存在几个重大问题。这些困难分为三个方面：

- 复制此类指针时，源指针将清零。
- 这些指针无法在标准模板库（STL）容器中正确使用。
- 可以创建这些指针的数组，但永远无法正确删除。

首先，复制 auto_ptr 实例时，目标指针和源指针会改变，源指针赋值为 NULL 复制之后。这是不寻常的复制语义，旨在防止访问共享资源；但从类名来看，这种行为并不明显。

其次，STL 容器要求实体可复制和可分配。如果将 auto_ptr 添加到容器中，则容器将拥有资源的独占所有权，并且源指针将为空（这是上述问题的结果）。托管指针不可复制或分配（似乎“可转让”）。

第三，数组是唯一的选择，STL 容器不适用于智能（托管）指针的集合。这种方法无法正常工作，因为 auto_ptr 析构函数使用的是 delete 形式，而不是 delete[] 形式。虽然可以创建这样的数组，但无法正确删除，从而导致资源泄漏。

清单 10.11 展示了一种使用 auto_ptr 托管资源的简单方法，但其局限性非常大，通常不应使用。C++11 标准弃用了 auto_ptr，并将其从 C++17 标准中删除。这种快速的弃用和删除表明了这些托管指针的有害性质，及用户对它们的看法。

清单 *10.11* 使用 *auto_ptr* 对象管理资源

```
struct Student {
  std::string name;
  double gpa;
  Student(const std::string& name, double gpa) : name(name), gpa(0) {
    if (gpa < 0)
      throw std::invalid_argument("gpa is negative");
  }
};

int main() {
  try {
```

```
    std::auto_ptr<Student> sammy(new Student("Samuel", 3.75));
    std::auto_ptr<Student> ginny(new Student("Virginia", 3.8));
    std::auto_ptr<Student> gene(
      new Student( "Eugene", -1)); // 1
  } catch (const std::invalid_argument& ex) {
    std::cout << "exception caught\n";
  }
  return 0;
}
```

注释 1：正常路径和错误路径在退出作用域时执行析构函数

这种简单的用法按预期工作，相当平淡。好处完全在于当指针超出范围时（无论是正常情况还是由于异常），对指针的管理。振作起来，还有更好的方法。

**解决**

理想情况下，需要使用 RAII 来管理基于指针的资源。这种方法使开发人员在编写几个特殊类来处理这些资源时的工作变得复杂，通常设计一个专门处理这种情况的类会更好。C++ 提供了 auto_ptr 类来实现这个目的，事实证明这是一个重大错误。此解决方案在现代 C++ 中弃用并删除，表明其使用始终存在问题。现代 C++ 为那些有幸能够使用它们的人提供了强大而有意义的智能指针。

这种情况并不意味着开发人员必须编写自定义 RAII 类来管理指针。Boost 项目中的一些坚定的开发人员已经为我们完成了这项工作（https://www.boost.org/）。如果不能使用现代 C++，可使用 Boost 库（即使只是为了智能指针）也是值得的。清单 10.12 中的代码演示了 boost::scoped_ptr 的用法，其语义是独占所有权；对于共享语义，请使用 boost::shared_ptr。这些 Boost 指针按预期工作，并解决了劣质 std::auto_ptr 的缺陷。这些 Boost 指针是现代 C++ std::unique_ptr 和 std::shared_ptr 的基础。请放心使用！

清单 *10.12* 使用 *auto_ptr* 对象管理资源

```
#include <boost/scoped_ptr.hpp>

struct Student {
  std::string name;
  double gpa;
  Student(const std::string& name, double gpa) : name(name), gpa(0) {
    if (gpa < 0)
      throw std::invalid_argument("gpa is negative");
  }
};

int main() {
  try {
    boost::scoped_ptr<Student> sammy(new Student("Samuel", 3.75));
    boost::scoped_ptr<Student> ginny(new Student("Virginia", 3.8));
```

```
    boost::scoped_ptr<Student> gene(
    new Student( "Eugene", -1)); // 1
  } catch (const std::invalid_argument& ex) {
    std::cout << "exception caught\n";
  }
  return 0;
}
```

注释 1：当范围退出时，正常路径和错误路径都会执行析构函数

开发人员现在只需花费最少的时间和精力来编写管理代码，其余的精力则用于解决他们正在处理的问题。最小化的代码更容易编写（即更有效），也更容易理解（即更易读）。

`boost::scoped_ptr` 实例是管理指向 `Student` 对象的指针的模板的实例化，构造函数通过获取资源的指针来取得资源的所有权。其语义是独占的；无法分配。如果需要分配，请使用共享指针版本。每当指针超出范围时，动态资源就会删除，这是预期的行为。

现代 C++ 允许使用 `std::move` 函数传输 `std::unique_ptr` 实例，该函数将源实例转换为可移动对象，然后通过复制赋值操作符将其传输到目标对象。此代码片段演示了该技术：

```
std::unique_ptr<int> ptr1 = std::make_unique<int>(42);
...
std::unique_ptr<int> ptr2 = std::move(ptr1);
```

**建议**

- 尽可能用现代 C++(或 Boost，如果没有现代）智能指针替换原始指针。
- 尽可能用至少 Boost 智能指针替换 `auto_ptr` 实例；现代 C++ 版本要好得多。
- 不要将 `auto_ptr` 用于 STL 容器；复制语义是个例外。
- 不要将 `auto_ptr` 用于数组；析构函数行为将导致未定义行为。
- 可能的情况下，请考虑使用现代 C++ 的 `unique_ptr` 和 `shared_ptr` 作为替代方案。

## 10.7. 错误 79： 混合 new 和 delete 的不同形式

此错误关注正确性。不正确使用新增和删除形式会导致未定义行为，使程序和系统处于未知状态。

动态资源的使用方式多种多样，例如：在编译时不知道实体的大小时。在运行时计算大小的动态内存的能力对于安全和正确的使用至关重要。C++ 提供了 `new` 和 `delete` 操作符来分配和释放内存。每种操作符都有两种形式：单个实体形式和数组形式。

**问题**

编程新手经常对数组着迷——它们为多个值提供一个变量名，以区分通过计算索引值来查找元素。数组的强大功能是使用它们的强大诱因，这导致许多新手（和专业人士）过度使用它们。大多数情况下，`std::vector` 是一个更好的选择；然而，数组将继续存在。

我们正在设计一个简单的程序来帮助教师计算成绩。每个学生都有几项测验和补充分数，必须将它们平均化为最终分数。由于补充工作项目是可选的（额外学分），因此项目数量未知且因学生

而异。一种直接的方法是创建一个数组，其大小远大于预期的最大元素数。立即出现两个问题：第一，如果该数字低于实际最大值怎么办？第二，为什么要为不需要的元素浪费空间？第一个问题是关于正确操作的问题，不容忽视。第二个问题是风格问题（没有正确性问题），但对于更有经验的开发人员来说，感觉不对。此外，可能会出现性能问题，因此这种方法确实存在问题。

清单 10.13 表单不匹配的简单平均代码

```
int main() {
  int count;
  std::cout << "Enter number of items to average: ";
  std::cin >> count;
  double* scores = new double[count]; // 1
  for (int i = 0; i < count; ++i) {
    std::cout << "Enter score: ";
    std::cin >> scores[i];
  }
  double sum = 0.0;
  for (int i = 0; i < count; ++i)
    sum += scores[i];

  delete scores; // 2
  if (count > 0)
    std::cout << "Average score is " << sum/count << '\n';
  else
    std::cout << "No items to average\n";
  return 0;
}
```

注释 1：分配计算出的内存量

注释 2：这会删除分配的内存吗？

**分析**

代码看起来合理，工作正常；计算出的平均值正确，但删除分配的内存存在一个大问题。

new 操作符根据实体的大小获取固定数量的内存。如果分配的是单个实体，则所需的内存量恰好是实体的大小。在这种情况下，new 的形式是朴素的，不会使用 [] 表示法。

当分配的内存基于多个实体时，数组形式用于 new 操作符，使用 [] 表示法。元素的大小乘以元素的数量，即可获得该内存量，这还不够。

编译器可以自由地实现内存管理，数组大小的信息存储在分配的内存中。这样做的编译器通常会将计数存储在返回内存的开头之前，所以第一个数组元素遵循数组大小，而数组访问代码不知道有关大小的信息。

清单 10.13 中，数组是动态分配的，程序计算得分总和。然后，删除 scores 指针。这种情况下，会发生什么谁也说不准；可能是按指针元素类型的字节数释放。由于元素数存储在开头，因此删除了它（可能还有第一个元素的一部分）或仅删除了第一个元素。谁知道呢？delete 形式（没有数组符号）释放了一个元素。数组其余部分会发生什么情况尚未定义，但可能是泄露了。这不会造成任何明显的问题，但是错误的行为。

如果 new 后面跟着 delete [] 操作符，会发生什么情况就远没有那么清楚了。那么 delete

[] 操作符会认为元素的数量是多少？哦，这看起来不太好！未定义的行为就是这样；程序可以做任何事情。无论它做什么，都不太可能和预期一致。

**解决**

将 new 和 delete 操作符视为一对；对于每个 new，必须有一个对应的 delete，对于每个 new []，必须有一个对应的 delete []。切勿混合这些对。混合它们会导致未定义行为，即使程序似乎可以运行也不正确。以下代码显示了微小但必要的更改，以正确将 delete [] 与其对应的 new [] 配对。

清单 *10.14* 正确配对 *new* 和 *delete* 操作符

```
int main() {
  int count;
  std::cout << "Enter number of items to average: ";
  std::cin >> count;
  double* scores = new double[count]; // 1
  for (int i = 0; i < count; ++i) {
    std::cout << "Enter score: ";
    std::cin >> scores[i];
  }
  double sum = 0.0;
  for (int i = 0; i < count; ++i)
    sum += scores[i];
  delete [] scores; // 2

  if (count > 0)
    std::cout << "Average score is " << sum/count << '\n';
  else
    std::cout << "No items to average\n";
  return 0;
}
```

注释 1：分配一个计算大小的数组

注释 2：删除该数组，并将其与分配的数组配对

**建议**

- 确保基于 new 形式使用正确的 delete 形式；混合使用会导致未定义行为。
- 使用 new 和 delete 时，请仔细研究代码以确保正确配对；如果单个函数中包含多个 new 和 new [] 操作符，则很容易混淆这些配对。
- 最小化函数的大小以避免混合。

## 10.8. 错误 80： 信任异常说明

此错误主要影响正确性和可读性，从而影响性能。正确性的问题在于代码可能无法按预期工作；可读性的问题在于代码看似直观，但实际上并非如此。

异常提供了一种通过备用控制路径报告错误的简洁方法，从而大大降低了代码复杂性。记录预期的异常是通过读取所有相关的捕获块来完成的，但这种方法还有待改进。一个好主意是通过在异常规范列表中指定它们，来记录函数抛出的异常。

**问题**

当开发人员编写所有受影响的代码时，异常处理会变得简单。使用库或其他模块时，查看和理解各种困难的能力会大大降低。清单 10.15 中的代码计算 vector 中一组值的标准偏差。开发人员小心地确保太少的值会引发异常，以防止在 std_dev 函数中错误计算结果。作为一名细心的开发人员，添加了异常规范来记录如果给出的值太少，函数可能会引发 std::invalid_argument。开发人员确保调用代码可以从异常中恢复，因为程序终止是有问题的。

正常情况下，此代码按预期工作，但如果没有给出任何值，情况就会变得很糟糕。如果传递给 std_dev 函数的值太少，预计会抛出异常。开发人员忽略了 arith_mean 函数是在 vector 大小测试之前执行的，调用的函数检测到值太少的问题，并抛出一个异常。尽管开发人员很小心，程序还是突然终止了。

清单 *10.15* 列出可以抛出的异常

```
double arith_mean(const std::vector<double>& values) {
  if (values.size() == 0)
    throw std::domain_error("missing values"); // 1
  double sum = 0.0;
  for (int i = 0; i < values.size(); ++i)
    sum += values[i];
  return sum/values.size();
}

double std_dev(const std::vector<double>& values)
                throw (std::invalid_argument) { // 2
  double mean = arith_mean(values);
  if (values.size() < 2)
    throw std::invalid_argument("too few values"); // 3
  double sum = 0.0;
  for (int i = 0; i < values.size(); ++i)
    sum += std::pow(values[i] - mean, 2);
  return std::sqrt(sum / (values.size() - 1));
}

int main() {
  std::vector<double> values;
  values.push_back(3.14159);
  values.push_back(2.71828);
  try {
    std::cout << "standard deviation is " << std_dev(values) << '\n';
  } catch (...) {
    std::cout << "exception caught, but we did not crash\n";
  }
```

```
30    return 0;
31  }
```

注释 1：如果存在的元素太少，则抛出 std::domain_error
注释 2：预测可能出现的 std::invalid_argument 异常
注释 3：如果存在的元素太少，抛出 std::invalid_argument 异常

开发人员对异常条件下的行为感到惊讶，并且对为什么主函数的 catch 块无法拯救程序感到困惑。

### 分析

开发人员试图通过指定异常来控制程序行为，这种做法虽然合理，但并不完善。除非彻底理解代码，否则在规范列表中指定异常就是一种猜测。如果猜对了，程序就会按预期运行；如果猜错了，程序就会毫不留情地崩溃。

编译器必须付出一些努力来处理异常规范。在不使用异常规范的程序中，抛出的异常都会随着堆栈展开而渗透到代码层次结构中。当添加了规范，行为就会以非直观的方式发生变化。编译器会添加代码来检查抛出的异常，并将其与规范进行匹配。如果匹配，则异常会按预期传播，但如果检测到的异常不在规范中，则检测代码会抛出 std::unexpected 异常，其默认行为是调用 terminate（后者会调用 abort），其他代码会影响性能。

如果开发人员希望将意外异常转换为更易于管理的异常，请编写一个抛出已知异常（标准异常或自己创建的异常）的函数，并将该转换函数作为 set_unexpected 函数的参数传递。简而言之，代码如下所示：

```
1  class MyException {};
2  void transformException() { throw MyException(); }
3  set_unexpected(transformException);
```

如果使用这种方法，则应在某个 catch 块中捕获生成的 MyException[8]。这种方法的一个重大缺点是整个程序会将 std::unexpected 异常转换为 MyException，这使得捕获其上下文非常困难。此解决方案是可行的，但其过于笼统是有害的。尝试记录异常会掩盖需要更好处理的问题。

### 解决

由于不确定是否可以知道所有可能的异常并将其列在异常规范中，因此最彻底和最有帮助的方法是消除规范。消除它们可以消除与对异常的直观理解背道而驰的意外（和不想要的）行为的可能性。库中许多调用的代码无法充分分析潜在异常，使用规范列表可能会产生意外行为。

异常规范是一个实验，已从现代 C++ 中删除。这一事实表明专家们不信任它们。我们可以将此视为一个明确的信息，采取相应行动并删除所有异常规范。

以下代码显示了此调整。作为替代方案，注释掉的函数头指定了两种可能的异常；但在实际系统中可能无法知道可能异常的完整列表。除非管理层或技术领导坚持，否则请避免采用这种方式。

清单 *10.16* 消除异常规范

[8]译者注：最好是继承于标准的 std::exception，可以让后续对异常的处理更加简单；否则 try/catch 块不能进行捕获，后续的处理会比较麻烦。

```cpp
double arith_mean(const std::vector<double>& values) {
  if (values.size() == 0)
    throw std::domain_error("missing values");
  double sum = 0.0;
  for (int i = 0; i < values.size(); ++i)
    sum += values[i];
  return sum/values.size();
}

// double std_dev(const std::vector<double>& values) throw (std::invalid_argument,
↪  std::domain_error) { // 1
double std_dev(const std::vector<double>& values) { // 2
  if (values.size() < 2) // 3
    throw std::invalid_argument("too few values");
  double mean = arith_mean(values);
  double sum = 0.0;
  for (int i = 0; i < values.size(); ++i)
    sum += std::pow(values[i] - mean, 2);
  return std::sqrt(sum / (values.size() - 1));
}

int main() {
  std::vector<double> values;
  values.push_back(3.14159);
  values.push_back(2.71828);
  try {
    std::cout << "standard deviation is " << std_dev(values) << '\n';
  } catch (...) {
    std::cout << "exception caught, but we did not crash\n";
  }
  return 0;
}
```

注释 1：可选：添加所有发现的异常（这个列表完整吗？）

注释 2：完全消除异常规范

注释 3：确保在调用其他代码之前进行本地验证

虽然异常规范列表是一个好主意（理论上），但意外行为、额外的实施成本，以及无法兑现其承诺意味着最好的方法是消除它们。

**建议**

- 如果可能的话，消除异常规范列表。
- 了解未包含在异常列表中抛出异常的意外行为。
- 记住使用异常规范的性能影响。

## 10.9. 错误 81： 未使用引用捕获异常

这个错误关注的是正确性、有效性和性能。异常可以由指针、值或引用抛出，并由同一类型捕获。但每种方法的优缺点各不相同。

异常为处理错误提供了另一种控制结构。正确实施后，可以实现可扩展的代码，并允许开发人员从可能终止程序的错误中恢复。异常比函数更昂贵，并且其行为不太直观。了解其中一些问题对于正确使用至关重要。

### 问题

我们可以想象，我们的开发人员已经了解了抛出和捕获异常的各种方法，但需要明确最佳方法。他们进行一些实验来测试多种组合并了解其中的差异。

正确地理解了抛出异常，则会将一些数据从抛出位置移动到捕获位置。其直觉认为移动最少数据的解决方案是最佳的，可以尝试通过指针抛出异常。开发人员对这种方法不满意，决定尝试各种通过指针抛出异常的方法，并将它们与通过值抛出进行比较。

清单 10.17 中的代码显示了三次这样的尝试。第一次是抛出指向本地对象的指针；第二次是抛出指向基于堆的对象的指针；为了进行比较，第三次是按值抛出。开发人员对 DerivedException 构造函数及其复制构造函数进行了检测，以计算每个构造函数调用的次数（未显示检测代码）。

清单 *10.17* 通过指针和值捕获异常

```
struct DerivedException : public std::exception {
  std::string message;
  DerivedException() : message("DerivedException") {}
  ~DerivedException() throw() {}
  const char* what() const throw() { return message.c_str(); } // 1
};

void throw_by_local_pointer() {
  DerivedException de;
  throw &de;
}

void throw_by_heap_pointer() {
  DerivedException* de = new DerivedException();
  throw de;
}

void throw_by_value() {
  DerivedException de;
  throw de;
}

int main() {
  try {
    throw_by_local_pointer();
  } catch(std::exception* ex) {
```

```
    std::cout << ex->what() << '\n';
  }
  try {
    throw_by_heap_pointer();
  } catch(std::exception* ex) {
    std::cout << ex->what() << '\n';
  }
  try {
    throw_by_value();
  } catch(std::exception ex) {
    std::cout << ex.what() << '\n';
  }
  return 0;
}
```

注释 1：重写 std::exception 虚函数

开发人员运行代码并注意到每个方法都调用了一次构造函数，并且从未调用过复制构造函数。从这些结果可以得出结论，实现抛出异常和捕获异常之间没有区别。

**分析**

可以赞扬开发人员的方法，但质疑其结论中遗漏了一些重要的东西。必须强调的一点是，当抛出一个对象（或指针）时，该对象将在抛出时超出范围。C++ 要求复制抛出的内容，并将临时对象（或指针）转移到捕获点。因此，无论选择哪种方法抛出，都会进行复制，从而增加抛出的成本。

throw_by_local_pointer 应该会引起抗议。本地对象在函数内创建，并抛出其地址。这种方法最大限度地减少了从抛出到捕获点传输的数据量，只复制指针的大小；然而，当抛出完成时，函数已经退出，破坏局部变量。catch 子句接收指针的副本，但该指针指向无效的 DerivedException 对象。输出的 what 信息是未定义的行为。

throw_by_heap_pointer 的执行速度与上一次尝试一样快，但解决了无效异常对象的问题。有一个不太明显的问题：谁删除了堆分配的对象？如果捕获点是最后一个使用该对象的地方，则此块中的代码删除该对象应该没问题。但由于无法恢复此代码，可能会再次抛出异常。那么，谁删除该对象？这种方法提出了一个关于对象删除责任的难以回答或无法回答的问题。

最后，分析按值抛出，开发人员发现构造函数调用次数没有差异，错误地认为唯一的成本是指针大小和异常对象大小之间的差异。这就产生了两个模糊的问题；开发人员忽略了第一个问题，而第二个问题则是看不见的。

对于按值捕获的异常，what 方法的输出不正确，其输出来自 std::exception 类的消息（输出是 std::exception，再聪明不过了！）——DerivedException 的多态 what 方法未执行。原因是抛出的异常的副本用于初始化 catch 参数。该参数的类型为 std::exception，因此抛出的异常切片，并且只复制了基类部分。切片了重载的方法，并执行基类版本。这是一个很容易发现的问题。

难以发现的问题基于一个显而易见的问题——它们是一对。按值抛出的明显成本是一次复制，这是开发人员（错误地）确定的。由于异常是由其基类捕获的，调用的不是派生类复制构造函数，而是基类版本。开发人员无法检测 std::exception 复制构造函数；否则，他们会看到隐藏的副本。按值抛出时，会为临时对象进行复制，然后使用临时对象的副本初始化 catch 参数——两

个副本。

**解决**

最后，开发人员通过值抛出，可避免出现指针删除问题。了解到通过值抛出会使副本数量翻倍，使用通过引用捕获来最小化成本。这种情况是少数几个（如果不是唯一）临时对象引用合法的地方之一（等待现代 C++）。在抛出点创建的临时对象，通过引用捕获参数进行传输，所以不会进行第二次复制。

此外，由于捕获了引用，其允许多态行为，并且调用了 DerviedException 版本的 what。以下代码显示需要进行简单的调整以最大限度地降低复制成本，并保留重写的方法行为，可添加 & 引用符号。

清单 10.18 通过引用捕获异常

```
struct DerivedException : public std::exception {
  std::string message;
  DerivedException() : message("DerivedException") {}
  ~DerivedException() throw() {}
  const char* what() const throw() { return message.c_str(); }
};
void throw_by_value_catch_by_reference() {
  DerivedException de;
  throw de; // 1
}

int main() {
  try {
    throw_by_value_catch_by_reference();
  } catch(const std::exception& ex) { // 2
    std::cout << ex.what() << '\n';
  }
  return 0;
}
```

注释 1：按值抛出，无论如何都必须进行复制

注释 2：按引用捕获，以避免复制并保持多态性

开发人员还从这次经历中吸取了一个教训，那就是不能想当然地认为简单的检测手段就足以全面描绘出一个程序的行为。

**建议**

- 通过值抛出异常可消除通过指针抛出的问题。
- 通过引用捕获异常可最大程度地减少复制并确保多态行为。
- 最小化派生异常类的大小可最大程度地减少复制成本。
- 切勿返回指向本地对象的指针（或引用）。

# 第 11 章 函数与编码

本章内容

- 选择正确的参数类型
- 函数返回多个值
- 优雅的函数
- 确保满足函数先决条件和后置条件

函数是模块化编程的核心，是可从不同位置多次调用的命名代码单元。函数的设计对于生成易于使用、行为有意义且无错误的可重用代码至关重要。函数设计和使用存在一些错误，主要与正确的参数定义和使用有关。许多其他错误也很常见；这些错误应该引起开发人员的警惕。

## 11.1. 设计考量

函数由名称、参数列表、返回值和函数体组成。编译器坚持所有四个部分，但提供了一种使参数列表和返回值看起来是可选的方法。有些书称构造函数（和隐含的析构函数）为函数；从技术上讲，它不是函数，但 C++ 标准称它们为“特殊成员函数”。构造函数和析构函数没有返回值，不符合函数性质。参数列表看起来是可选的，因为可以为空，但仍必须用空括号指定。返回值看起来是可选的，类型可以为 void。但这仍然是一种类型，是没有值的类型，或者其值为空。

命名是定义函数最难的部分。虽然可以使用旧名称，但有一个单一、简洁、易于沟通的名称通常很困难。必须以直观的方式解释函数的用途，使其使用起来简单明了。过长或复杂的函数几乎不可能命名。

理想情况下，函数应该是纯函数，不会访问其范围之外的变量或数据，完全根据其参数列表传递的数据执行其行为。有些函数应该在其范围之外影响系统，operator<< 和 operator>> 方法就是很好的例子。由于这些函数必须存在，目标是将它们分为两类：纯函数和产生副作用的函数。

产生副作用的函数不应进行任何计算，逻辑最少，并且影响其作用域之外的实体。纯函数应执行计算，使用必要的逻辑，并通过返回值返回其结果。某些函数需要返回多个值，而使用其返回值无法实现这一点。返回结构体（或类）的实例或使用输出参数是选项。输出参数更难以快速理解，并影响可读性。返回实例更复杂，但可以更好地将输入与输出隔离。

值、指针或引用参数可以将数据输入传递给函数。本书使用在调用站点引用值的约定 argument，参数列表中的变量称为 parameter。（有些作者和老师也有使用 parameter 和 formal parameter 的配对）

函数的局部变量通过显式编程操作初始化，通常使用赋值操作符。参数是局部变量，但开发者不会显式地初始化它们。编译器生成代码来复制参数的值，该值是参数的初始化值。否则，参数在所有方面都是常规的局部变量。局部变量的一个重要结果是，当函数退出时（通常是通过抛出异常）局部变量会销毁，但这并不意味着销毁的变量一定无法访问。有些系统允许在销毁它们之后访问（通过指针或引用），从而导致难以诊断的错误；优秀的系统会在这种情况下会崩溃。

以下错误尝试解决函数设计和使用中的一些常见问题。虽然存在许多此类错误，但有些错误更常发生。

## 11.2. 错误 82： 未使用默认参数

这个错误会影响可读性和有效性。编写多个重载函数可以以一致的方式处理不同的参数列表，但可能会产生繁琐且重复的代码。

### 问题

问题经常需要几个函数来完成相同的工作，但参数数量不同。C 语言会强制这些函数使用不同的名称，但 C++ 允许使用相同的名称，但有一个限制，即参数列表必须是唯一的（无论是参数类型、参数数量还是类型顺序）。重载函数使用相同的名称，从而保持操作的语义。

以下代码重载了三个函数，对不同的参数执行相同的操作。这种方法冗长乏味，永无止境。

清单 *11.1* 处理不同数量参数的重载函数

```
int sum(int a, int b) { return a + b; }
int sum(int a, int b, int c) { return a + b + c; }
int sum(int a, int b, int c, int d) { return a + b + c +d; }

int main() {
  std::cout << sum(3, 4) << '\n';
  std::cout << sum(3, 4, 5) << '\n';
  std::cout << sum(3, 4, 6, 7) << '\n';
  return 0;
}
```

### 分析

这些函数的功能完全符合预期，但代码重复，写完前几个之后就很累了。虽然可能性不大，但代码主体中的拼写错误可能会导致函数之间的行为不同。

### 解决

使用默认参数是一种更好的方法，可以最大限度地提高效率、阅读理解能力和可能是正确性。有些函数不适合这种方法，可能是因为没有合理的默认值。

清单 *11.2* 为多个参数使用默认参数

```
int sum(int a, int b, int c = 0, int d = 0) { return a + b + c +d; }

int main() {
  std::cout << sum(3, 4) << '\n';
  std::cout << sum(3, 4, 5) << '\n';
  std::cout << sum(3, 4, 6, 7) << '\n';
  return 0;
}
```

sum 函数消除了代码重复，但可能存在编码错误。许多情况下，可能会出现某种形式的有意义的默认值，但不要使函数过度默认并根据参数更改行为。例如，布尔参数为 false 可能会执行 this 功能，但如果为 true，则会执行 that（略有不同）功能，请保持简单。

**建议**

- 保持函数简单，允许参数有默认值。
- 不要使用默认参数来改变函数行为。

## 11.3. 错误 83： 未能使用断言

这个错误影响了有效性和正确性。开发程序有两个时段——编译时和运行时，软件构建也有两个时段——开发时和生产时。这个错误主要针对软件的构建开发时。

**问题**

程序开发过程中，会测试新想法、引入新代码，并尝试增加组件之间的交互。所有这些新事物都为漏洞的生长提供了完美的条件——漏洞确实会滋生。

确保代码按预期运行，应该通过单元测试和集成测试来确认。在此之前，确保满足函数先决条件必不可少。此外，对于开发人员来说，确保满足函数后置条件和不变量也至关重要。然而，编写代码来测试和处理这些领域似乎总是 …… 可以推迟——明日复明日。

推迟先决条件、后置条件和不变量很诱人，但也很危险。如果没有足够的测试，错误就会悄悄出现并不断增加。以下代码就是一个例子；虽然它看起来无害，但其中却隐藏着真实而现实的危险。

清单 11.3 意外的除零问题

```
// a divides b evenly
bool divides(int a, int b) {
  return b % a == 0; // 1
}

int main() {
  int x = 0;
  int y = 42;
  if (divides(x, y))
    std::cout << x << " divides " << y << '\n';
  return 0;
}
```

注释 1：这很少会出现问题，但是 ……

大多数情况下，此代码都会正常运行，开发人员对其好处的信心也会增强。有一天，当编写此代码与关注其他较新的代码之间相隔很远时，此代码就会崩溃。但为什么呢？以前总是能正常工作。

**分析**

原因是除以零的问题，错误的输入数据会导致问题浮现。这个错误一直存在，但从未暴露出来。单元测试应该能揭示这一点，但工作量是推迟编写验证代码的另一个“原因”。

**解决**

开发过程中，发现错误的最佳方法之一是，编写前置条件和后置条件断言。这些小测试非常残酷—如果失败，程序会立即崩溃。从好的方面来说，错误暴露了，并且无需展开堆栈，调试器就可以在问题的确切区域设置一个。

断言是一个简单的测试，用于检查某事是否为真。编写起来很快，不需要特殊逻辑，而且功能强大。

下列代码快速解决了清单 11.3 中的潜在错误。

清单 *11.4* 使用断言检查先决条件

```
// disable assertions by defining NDEBUG // 1
// a divides b evenly
bool divides(int a, int b) {
  assert(a!=0); // 2
  return b % a == 0;
}
int main() {
  int x = 0;
  int y = 42;
  if (divides(x, y))
    std::cout << x << " divides " << y << '\n';
  return 0;
}
```

注释 1：向编译器传递 NDEBUG 标志，以禁用生产代码中的断言

注释 2：证明除数非零

如果生产代码不应包含断言，请不要使用条件编译来删除它们。编译代码时向编译器定义 NDEBUG 宏，相关代码将消失—不存在运行时成本。

使用测试框架、工具和方法咨询其他软件工程和开发资源。这些远远超出了这个错误的范围，但这可以激发人们对其的研究和使用的兴趣。断言在调试期间特别有用，因此要学会很好地使用。

**建议**

- 不要将先决条件、后置条件和不变性检查推迟到以后。在开发过程中使用断言来验证条件。
- 使用单元测试框架进一步验证代码。不要将测试视为无功能的代码；相反，应将其视为证明其正确性的元代码。
- 永远不要编写具有副作用的断言；如果禁用断言，代码的行为将发生意外变化。

## 11.4. 错误 84： 返回局部对象的指针或引用

此错误与正确性和可读性有关，但对性能有轻微的负面影响。在一个地方创建值或对象，并将其转移到其他区域的能力是一种强大的代码模块化技术。

**问题**

在函数范围内创建对象有助于隔离代码，使其更易于理解和管理。清单 11.5 中的代码应该将两个 std::string 对象，合并为第三个更大的对象。开发人员认为，输出流中使用函数比使用 operator+= 创建本地对象，并引用该变量更有意义。开发人员还看到这种方法可以最大限度地减少复制（构造函数和可能的析构函数调用），从而提高性能。

清单 *11.5* 返回一个局部指针或引用

```cpp
const std::string& catenate(const std::string& a, const std::string& b) {
  std::string combined(a);
  combined += b;
  return combined;
}
// const std::string* catenate(const std::string& a,
  [SA]const std::string& b) { // 1
// std::string combined(a);
// combined += b;
// return &combined;
// }
int main() {
  std::string msg1 = "Hello, ";
  std::string msg2 = "world!";
  std::cout << catenate(msg1, msg2) << '\n'; // 2
  return 0;
}
```

注释 1：通过指针返回对象示例

注释 2：比创建、更新和引用变量更顺畅

**分析**

将代码隔离到函数中是一个好主意，尤其是在特定上下文中使用起来更顺畅时。清单 11.5 中的代码的可读性非常直观，并消除了局部变量的本地创建和更新。

创建 catenate 函数是为了减轻局部变量的创建和更新负担，以提高可读性。调用代码在其输出上下文中调用该函数，并且一切都应该正常工作，但其行为未定义。当函数返回时，组合局部变量超出范围。

catenate 函数创建一个局部 std::string 变量，组合两个参数，并返回指向该局部变量的指针（或引用）。调用点尝试访问数据。如果开发人员很幸运，尝试访问将导致分段错误（或类似错误）并使程序崩溃。不幸的开发人员不会遇到问题。更糟糕的是，不幸的开发人员可能会看到预期的结果。

创建局部变量时，会在堆栈上分配内存空间来表示变量。函数执行完成后，会返回该变量的地址，但函数的完成会做一些不可见但有影响的事情。专用于该函数的堆栈框架将失效，并可供其他代码使用该空间。指针仍然指向计算值，如果结果未被覆盖或访问不会导致崩溃，则调用代码将看到结果。

**解决**

解决方案很简单，只需返回本地值的副本即可，如以下清单所示。此方法会导致调用复制构造函数，从而产生开销（有时是析构函数调用）。但复制的结果是正确的，并且没有令人讨厌的未定义行为。大多数情况下，返回值优化（RVO）可以消除部分开销。

清单 *11.6* 返回一个值

```cpp
std::string catenate(const std::string& a, const std::string& b) {
  std::string combined(a);
```

```
  combined += b;
  return combined;
}

int main() {
  std::string msg1 = "Hello, ";
  std::string msg2 = "world!";
  std::cout << catenate(msg1, msg2) << '\n';
  return 0;
}
```

编写高性能和可读性的值得称赞，但必须遵循特定规则以确保正确性。返回本地计算的值时，返回引用或指针是不安全的。必须接受复制的影响，现代 C++ 编译器通常会尽量减少或消除这种影响。

另一种方法是在堆上创建本地对象，并将指针传回给调用者。这种方式不会发生未定义行为，但删除动态内存的问题会成为影响可读性的新问题。如果不删除，还会导致资源泄漏。

**建议**

- 切勿返回指向本地创建对象的指针或引用。
- 了解函数调用和返回对内存的影响机制。

## 11.5. 错误 85： 使用输出参数

此错误会模糊变量的用途，从而影响可读性，并且可能存在一些轻微的有效性问题。返回多个值已证明是有问题的，并且已使用了一些技术。这些方法通常难以阅读、编写或两者兼而有之。

**问题**

有时，函数需要返回两个或更多值。一种显而易见的方法是在调用代码中，创建调用函数将修改的变量。为每个变量传递一个指针或使用一个引用参数是可行的，引入的复杂性可能会混淆它们的使用并导致理解错误。

清单 11.7 中的代码执行整数除法并返回商和余数。开发人员使用了一个输出参数，该函数只能返回一个值。余数参数是一个引用变量，是调用代码中某个值的别名。该函数修改了调用代码的变量，一切正常，但读者可能不会注意到这种用法，并感到困惑。

清单 11.7 使用引用变量返回值

```
int divide(int value, int divisor, int& remainder) {
  remainder = value % divisor;
  return value/divisor;
}
int main() {
  int x = 42;
  int y = 4;
  int r; // 1
  int q = divide(x, y, r);
```

```
  std::cout << x << " divided by " << y << " is " << q
    << " with remainder " << r << '\n';
  return 0;
}
```

注释 1：不明显的返回值

### 分析

读者可能需要一段时间才能发现，调用代码中的局部变量 `r` 未初始化，调用函数中的代码对其进行了初始化。这不符合预期的模式，但这比有初始化值的情况略微更具可读性。通常，开发者不希望函数修改参数值；相反，他们希望计算结果以可以分配给变量的方式返回。

使用指向调用者局部变量的指针不会改变这种情况，只会改变其实现。语法更混乱，函数的代码也略显笨拙。

### 解决

标准模板库提供了一个由类模板实现的包含两个类型值的容器。此 `std::pair` 允许函数返回两个更符合预期行为的值。此外，无需定义结构或类来包含这些值，标准模板库已经这样做了。

清单 *11.8* 使用 *std::pair* 返回两个值

```
std::pair<int, int> divide(int value, int divisor) {
  return std::make_pair(value / divisor, value % divisor);
}

int main() {
  int x = 42;
  int y = 4;
  std::pair<int, int> res = divide(x, y); // 1
  std::cout << x << " divided by " << y << " is " << res.first
    <<" with remainder " << res.second << '\n';
  return 0;
}
```

注释 1：两个值均返回，这样读起来更顺畅

另一种方法是返回一个数组，调用者可以从中提取值，但这种方法可能晦涩难懂，影响可读性。其使用 `std::pair`，但灵活性较差。数组元素必须是同质的，而 `std::pair` 允许异质类型。

最后一种选择是返回一个结构并让调用代码从其字段中提取值。这种方法更易读，但需要编写一个新结构，可能只使用一次。

### 建议

- 尽量减少输出变量的使用；可能的话，消除它们。
- 尽可能使用标准模板库提供的功能。

## 11.6. 错误 86： 参数类型的错误使用

此错误主要影响性能，并且会影响可读性。C++ 允许通过值、指针或引用传递参数。正确选择至关重要。正确使用参数的三个方面是其大小、用法和安全性。以下讨论将重点讨论 `std::string` 参数，偶尔会引用内置类型。

参数的大小取决于其数据类型。值参数作为来自调用站点的参数值的副本传递。内置类型往往很小，将其作为副本传递通常很有效，但用户编写的数据类型可能很大。`std::string` 对象的大小取决于运行时库；可能存在各种大小，但将假定大小为 32 字节。如果字符串通过值传递，则堆栈必须为副本分配 32 个字节，并执行代码以将这些字节从源复制到堆栈。这项工作会影响性能和调用堆栈上的可用空间。

如果使用指针传递参数，则堆栈必须为架构的指针大小分配足够的字节——为了便于讨论，假设为 8 个字节。就大小和速度而言，通过指针传递字符串可能比通过值传递更高效——与大多数性能声明一样，测量对于验证至关重要。

最后，如果参数通过引用传递，代码的作用就如同通过指针传递的参数。传递的数据大小为 8 个字节。引用语法与指针语法不同，但更易于使用。

因此，按值传递值的大小变化最大。如果数据类型较小，8 个字节或更少（在 64 位架构上），则按值传递的成本与按指针或引用传递的成本完全相同。但如果数据类型的大小超过 8 个字节，或具有非平凡的复制构造函数，则会产生额外的开销。

参数的用法就是其语义，确定如何使用参数对于正确传递至关重要。假设参数是一个输入值；任何传递方式都可以。应考虑可读性和性能来确定最佳方法；但如果要将参数用于函数的输出，则不能使用按值传递。必须通过指针或引用传递参数。另一种错误试图阻止人们使用输出参数。

通过指针或引用传递参数不仅限于简单的输出参数；否则，无法用于修改调用方范围内的源数据。如果调用方希望对数据容器进行排序，则向其传递指针或引用是最佳选择，可以避免复制容器的内容。在选择使用哪种传递类型时，必须彻底了解参数的用途。

最后，数据的安全性至关重要。由于指针或引用参数通常比值参数更有效，因此很容易假设所有参数都应该是这种类型。在接受这种效率之前，请仔细考虑可读性。这很关键，确定函数是否应该修改数据。如果函数不应该更改源数据，则必须适当使用 `const` 关键字。考虑到引用的易用性（以及可以修改调用方数据的函数），在引用上使用 `const` 非常自由。由于指针语法更难编写和读取，因此引用是首选。此外，引用不能为空，不需要测试有效性。

### 问题

通过值传递字符串比通过指针或引用传递字符串占用更多的空间。优点是参数是源数据的副本，函数无法修改该数据。引用的优点是语法简单。指针的一个优点是它熟悉旧代码库。指针的另一个优点是，在用于修改值时不会出错；引用看起来就像值修改，很容易忽略正在更新的内容。以下代码使用了每种可能性，而没有考虑何时应使用每种参数类型。

清单 *11.9* 参数类型使用不当

```
1  const std::string catenate(std::string a, std::string& b, std::string* c) {
2    std::string combined(a);
3    b[0] = '-'; // 1
4    combined += b;
```

```
  c[0] = 'z'; // 2
  combined += *c;
  return combined;
}

int main() {
  std::string msg1 = "Hello";
  std::string msg2 = ", ";
  std::string msg3 = "world!";
  std::cout << catenate(msg1, msg2, &msg3) << '\n';
  return 0;
}
```

注释 1：无意的拼写错误影响了原始数据；语法上没问题，但存在错误

注释 2：另一个无意的拼写错误影响了原始数据；可以编译，但有缺陷

### 分析

第一个参数效率低下；传递消耗超过指针大小的数据类型时应仔细考虑。

此外，必须将复制的构造函数和析构函数计入使用此方法的成本中。无论其易用性和对调用方数据的内置安全性如何，按值传递仅应用于较小的数据和内置类型。第二个参数高效但不安全，并且无法区分其用途是仅用于输入还是包含输出方面。第三个参数与第二个参数非常相似，而且增加了使用指针语法的复杂性。虽然这种语法可能很熟悉，但使用时可能需要费很大力气才能确定其确切含义。开发人员应考虑函数参数列表中每一项的大小、用法和安全性的影响。

### 解决

清单 11.10 考虑了这三个特征，并为每个参数使用了最有意义的方法。由于字符串参数仅是输入，每个参数都将它们指定为 const 引用。现在显然不允许其修改调用者的数据，从而消除了作为输出参数的可能性。使用引用将最少量的数据从调用点传递到函数，从而对性能产生了积极影响。最后，当编译器强制执行其 const 性时，会阻止试图通过引用修改调用者数据的无意拼写错误。

清单 *11.10* 正确使用参数类型

```
const std::string catenate(const std::string& a, const std::string& b, const
std::string& c) {
  std::string combined(a);
  // b[0] = '-'; // 1
  combined += b;
  // c[0] = 'z';
  combined += c; // 1
  return combined;
}

int main() {
  std::string msg1 = "Hello";
  std::string msg2 = ", ";
```

```
  std::string msg3 = "world!";
  std::cout << catenate(msg1, msg2, msg3) << '\n';
  return 0;
}
```

注释 1：现在这是一个错误；参数是 const，消除了无意修改的可能性

**建议**

- 从大小方面考虑传递参数的效率。从调用点复制到函数的字节数是一般经验法则；如有疑问，请进行测试。
- 考虑参数的用途；是仅输入、输入/输出还是仅输出；以及函数是否应修改调用方的数据。
- 考虑数据的安全性—仅在明确需要时才允许修改调用方的数据；否则，请指定 const。

## 11.7. 错误 87： 依赖参数求值顺序

这个错误关注的是正确性，对可读性有积极影响。许多情况下，求值顺序是指定的，开发人员很快就会对如何编写代码来使用此顺序产生直觉。例如，更复杂的 for 循环对每个部分求值都有特定的顺序。求值参数似乎很简单，它们通常是某个值的副本、指针或对变量的引用。

> **NOTE**
>
> 本节讨论中，术语 parameter 的使用相当宽泛。有两组公认的术语来描述参数。第一组是在调用点使用的值称为 parameter，函数参数列表中的变量称为 formal parameter。第二组是在调用点使用的值称为 argument，函数参数列表变量称为 parameter。我更喜欢第二种，但我将使用第一种来与其他文献保持一致。

**问题**

假设开发人员热衷于减少调用函数时使用敲击键盘的次数，并决定在参数传递中引入一些副作用。表面上，这将减少代码行数，因为对参数的更新是函数调用的一部分。以下代码显示了使用两个值调用函数的简单方法。该函数确定相邻值是否增加。Increasing 表示左侧值小于或等于右侧值。

清单 11.11 既有副作用的函数调用

```
bool isIncreasing(int a, int b) {
  return a <= b;
}

int main() {
  std::vector<int> values;
  for (int i = 0; i < 10; ++i)
    values.push_back(i+1);
  int loc = 0;
  while (loc < 9) {
    if (!isIncreasing(values[loc], values[++loc])) // 1
```

```
      break;
  }
  if (loc == 9)
    std::cout << "success\n";
  else
    std::cout << "failure\n";
  return 0;
}
```

注释 1：计算参数

**分析**

函数代码是正确的，但调用代码包含一个重大错误。开发人员假设最左边的参数将首先计算，获得 `loc` 处的值，然后在增加其位置后获得右侧参数的值。最右边的参数将获得 `loc` 处的值并更新它。一切都应该按预期工作，但循环在第一次迭代时失败。C++ 不保证哪个参数首先计算；在我的环境中，最右边的参数首先计算，增加 `loc` 然后确定其值。之后，建立 `loc` 处的值并将其作为第一个参数传递给函数。结果是每次循环都会比较相同的值，而不是相邻的值。此外，第一个元素从未测试过。副作用导致程序失败，尽管它并不明显，并且函数是正确的。

**解决**

尽管在表达式求值中使用副作用的诱惑很大，但这总是个坏主意。直观的感觉是参数将按从左到右的顺序进行求值，但 C 之后的 C++ 没有定义任何顺序，不能假设任何顺序。

要解决此问题，请确保参数不涉及表达式。在调用函数之前计算表达式将需要一些敲入额外的代码，但正确性至关重要。以下代码更冗长，但恒正确。

清单 11.12 将表达式用作参数之前求值

```
bool isIncreasing(int a, int b) {
  return a <= b;
}

int main() {
  std::vector<int> values;
  for (int i = 0; i < 10; ++i)
    values.push_back(i+1);
  int loc = 0;
  while (loc < 9) {
    int x = values[loc];
    int y = values[++loc]; // 1
    if (!isIncreasing(x, y))
      break;
  }
  if (loc == 9)
    std::cout << "success\n";
  else
    std::cout << "failure\n";
  return 0;
```

```
}
```

注释 1：在调用函数之前计算表达式的值

清单 11.12 中的代码比较相邻的值，从第一个值开始。前缀增量操作符的副作用的时间隔离得很好。虽然代码可以更好地获取这两个值，并且应该编写更好的代码，但副作用代码的隔离得到了证明。

**建议**

- 切勿在参数中使用副作用，参数求值顺序未指定。
- 尽量减少（或消除！）节省按键次数的巧妙编码；最好多花一些时间来清晰地传达意图并确保正确性。

## 11.8. 错误 88： 传递参数过多

这个错误会影响可读性，对有效性有轻微影响。命名参数的好处已经在 Python 和 JavaScript 等语言中得到证实。C++ 目前没有这个强大的功能，必须严格遵循参数列表。有时，参数的顺序应该更直观，但长列表会使这样做变得困难。

**问题**

有些函数的参数列表很长，可能需要明确说明其顺序。以下代码演示了一个类，其中的几个参数对开发人员来说（显然！）具有一些有意义的顺序。但类用户可能需要明确顺序，这可能会导致错误使用。

清单 *11.13* 调用参数顺序不明显的构造函数

```
class Person {
private:
  std::string first;
  std::string middle;
  std::string last;
  int year;
  int month;
  int day;
public:
  Person(const std::string& f, const std::string& l, const std::string& m,
  int y, int d, int mo) :
    first(f), middle(m), last(l), year(y), month(mo), day(d) {} // 1
  const std::string& getFirst() const { return first; }
  int getYear() const { return year; }
};

int main() {
  Person p("Ann", "Konda", "A.", 2000, 14, 3); // 2
  std::cout << p.getFirst() << " is " << 2024 - p.getYear() << " years old\n";
  return 0;
```

```
}
```

注释 1：命名不当且参数过多

注释 2：很难跟踪哪个参数是哪个，以及到底要包含多少个参数

**分析**

创建 Person 类的实例需要调用者传递 6 个参数：3 个用于名称，3 个用于出生日期。构造函数可能会更有帮助，因为它命名了参数使用单个字母。开发人员清楚地了解它们的含义，但需要通过更好的命名来传达这些知识。

由于这 6 个参数是必需的，调用者必须按正确的顺序提供每个参数。很容易错误地按第一个、中间和最后一个的顺序指定名称（这对许多开发人员来说是直观的）。尽管如此，开发人员还是按第一个、最后一个和中间的顺序排列它们。虽然这种方法本身并没有错，但必须澄清一下。

需要的是一种方法来更好地按照调用者喜欢的顺序指定参数，同时又不损害构造函数的正确性。

**解决**

命名参数将使这种方法更易于管理。由于 C++ 不支持，另一种方法是使用普通旧数据（POD）结构；有些人称之为参数对象。POD 是一个传输层 - 允许用户按照对他们有意义的顺序使用字段名称，并将该数据传送到构造函数。构造函数按照有意义的顺序从 POD 中提取数据。此层允许顺序变化而不影响正确性 - 事实上，其允许直接使用不严格遵守的特定顺序来确保正确性。此外，使用 POD 可以消除混乱的调用点，单个参数可用于传递多个值。

很难对参数数量做出严格的限制，短期记忆和直观使用表明参数数量最少为 3 个。开发者必须记住，其他人会使用他们的代码，因此无论设计“简单而明显”，其他人都可能不同意。请将参数数量保持在较小水平。

这里必须说明一点：POD 不能确保所有值都已初始化。正确的构造函数设计对于防止数据丢失至关重要。

清单 *11.14* 使用 *POD* 作为命名变量技术

```
struct personPOD { // 1
  std::string first;
  std::string middle;
  std::string last;
  int year;
  int month;
  int day;
};

class Person {
private:
  std::string first;
  std::string middle;
  std::string last;
  int year;
  int month;
```

```
  int day;
public:
  Person(const std::string& f, const std::string& l, const std::string& m,
      int y, int d, int mo) :
    first(f), middle(m), last(l), year(y), month(mo), day(d) {}
  Person(const personPOD& pp) : first(pp.first), middle(pp.middle),
    last(pp.last), year(pp.year), month(pp.month),
    day(pp.day) {} // 2
  const std::string& getFirst() const { return first; }
  int getYear() const { return year; }
};

int main() {
  personPOD pp;
  pp.month = 3; // 3
  pp.day = 14;
  pp.year = 2000;
  pp.last = "Konda";
  pp.first = "Ann";
  pp.middle = "A.";

  Person p(pp);
  std::cout << p.getFirst() << " is " << 2024 - p.getYear() << " years old\n";
  return 0;
}
```

注释 1：包含每个参数的 POD

注释 2：接受 POD 的构造函数

注释 3：使用命名字段以方便的顺序初始化 POD 字段

**建议**

- 设置参数数量的限制，并使用 POD 传输超过该限制的参数。为不同的概念设置单独的 POD 有助于保持一致性和连贯性。
- 确保每个字段都已初始化，以防止破坏类的不变量。
- 当命名变量合适但不可用时，请使用 POD。

## 11.9. 错误 89： 函数过长且行为复杂

这种错误最显著的影响是可读性（可理解性），在每个函数中放入少量代码会对效率产生负面影响。不过，还是可以提出一个论点从长远来看，编写简短的函数可以提高开发人员的生产力。

**问题**

许多遗留函数很长，有些甚至很长。项目包含超过 1,000 行的函数并不罕见。我曾经使用过一个有 1,200 行代码的函数，但我从未理解它的大部分功能。通常，函数开始时很短，但随着时间的推移，会积累越来越多的代码来满足新的需求和变化。清单 11.15 中的代码需要重新设计，

随着时间的推移，添加了越来越多的代码。开发人员需要编写所有代码，并（错误地）认为既然它们一起工作，就应该一起编写。

假设标准偏差代码仅适用于非负值——需要一些数据清理的理由！在处理之前必须清理数据，应消除所有负值。此后，计算算术平均值。最后，计算标准偏差。代码从上到下逻辑流畅，该函数也应该更容易阅读。

更糟糕的是，函数名称 `std_dev` 并未传达将涉及数据清理的信息，这种错误命名很容易让读者误以为只计算标准差。正确命名函数很复杂，每个名称都应该简明扼要地传达函数的用途。这个函数应有简短、易懂的名称，准确地传达其所有功能。像 `compute_standard_deviation_after_removing_negative_values` 这样的内容传达效果很好，但冗长；坦率地说，也很难阅读。

清单 11.15 一个包含三个部分的过长的函数

```
double std_dev(const std::vector<double>& values) { // 1
  std::vector<double> new_values;
  for (int i = 0; i < values.size(); ++i) // 2
    if (values[i] >= 0)
      new_values.push_back(values[i]);

      double sum = 0.0;
  for (int i = 0; i < new_values.size(); ++i)
    sum += new_values[i];
  double mean = sum/new_values.size();

  sum = 0.0;
  for (int i = 0; i < new_values.size(); ++i) // 3
    sum += std::pow(new_values[i] - mean, 2);
  return std::sqrt(sum / (new_values.size() - 1));
}
int main() {
  std::vector<double> values;
  values.push_back(3.14159);
  values.push_back(-1.23456);
  values.push_back(2.71828);
  std::cout << "standard deviation is " << std_dev(values) << '\n';
  return 0;
}
```

注释 1：顾名思义，只计算标准差

注释 2：数据清理令人意外；没有记录

注释 3：只有此代码执行函数名称所暗示的操作

**分析**

此函数有三种不同的行为，这些行为合乎逻辑；必须每做一次才能得出答案。很容易假设，这些行为是必需的，所以可以或应该逐个地写出来。

虽然本书没有解决可测试性问题，但这仍然是编写简短、易于理解的函数的重要动机。随着越来越多的需求接受，新功能可能会添加到这个函数中，从而加速问题的解决。

### 解决

函数应该简短并执行一个明确定义的行为。如果需要一系列行为（一系列函数调用），则主要有两种选择：错误选择和正确选择。错误的选择是将 `std_dev` 函数重写为三个函数，并让用户（`main` 函数）按正确顺序调用这三个函数。

我的学生反复听到，`main` 函数应该视为老板代码。老板负责指挥操作，但不负责处理细节——从来不想要事无巨细的管理者。如此指挥的函数必须处理细节，老板代码应该只提供足够的数据来解决问题，而非更多。要求老板正确命令函数以获得标准偏差，是在推卸开发人员的责任。

正确的解决方案是编写包含所需功能的三个函数，并添加第四个函数来协调。以下代码展示了这种方法，`boss` 代码像以前一样调用 `std_dev` 函数，而无需了解执行此操作所需的详细信息计算结果。我们假设这个名字已经确立；否则，应该更完整地拼写出来——有时，必须接受我们无法改变的事情。

清单 *11.16* 拆分长函数并添加编排器

```
std::vector<double> cleanse(const std::vector<double>& values) {
  std::vector<double> good;
  for (int i = 0; i < values.size(); ++i)
    if (values[i] >= 0)
      good.push_back(values[i]);
  return good;
}

double arith_mean(const std::vector<double>& values) {
  double sum = 0.0;
  for (int i = 0; i < values.size(); ++i)
    sum += values[i];
  return sum/values.size();
}

double std_deviation(const std::vector<double>& values, double mean) {
  double sum = 0.0;
  for (int i = 0; i < values.size(); ++i)
    sum += std::pow(values[i] - mean, 2);
  return std::sqrt(sum / (values.size() - 1));
}

double std_dev(const std::vector<double>& values) { // 1
  std::vector<double> new_values = cleanse(values);
  double mean = arith_mean(new_values);
  return std_deviation(new_values, mean);
}

int main() {
  std::vector<double> values;
  values.push_back(3.14159);
  values.push_back(-1.23456);
```

```
  values.push_back(2.71828);
  std::cout << "standard deviation is " <<
    std_dev(values) << '\n'; // 2
  return 0;
}
```

注释 1：管理详细信息的编排器功能

注释 2：用户（老板）代码没有变化

**建议**

- 将扩展函数拆分为小的、易于理解的函数。
- 添加协调器函数来协调新的小函数。
- 为函数命名，使其行为清晰易懂。
- 如果可以使用重构代码的自动化工具（例如 IDE 插件），请学会使用；有些工具会将多个参数转换为参数对象—太棒了！

## 11.10. 错误 90： 职能过多的函数

这种错误会影响可读性和有效性，并可能对正确性产生负面影响。函数是编写可重用、可读代码的核心，但如果编程不当，则可能会成为负担。

**问题**

与上一个错误相关，过于负责的函数承载了太多功能，让人无法清楚理解。开发者又走了一条捷径，结果如下代码。对数据进行清理的需求发展为两个要求：过滤低于 0 的数据和过滤高于 0 的数据。由于这些行为相似，因此编写了一个函数来执行这两项操作。调用代码需要传递一个值 vector 和一个布尔标志，以确定哪些值可消除。

清单 *11.17* 在一个函数中过滤 *0* 以上和 *0* 以下的数据

```
std::vector<double> cleanse(const std::vector<double>& values,
    bool less_than) {
  std::vector<double> new_values;
  for (int i = 0; i < values.size(); ++i)
    if (less_than) { // 1
      if (values[i] < 0)
        new_values.push_back(values[i]);
    } else
        if (values[i] > 0)
          new_values.push_back(values[i]);
  return new_values;
}

int main() {
  std::vector<double> values;
  values.push_back(3.14159);
```

```
  values.push_back(-1.23456);
  values.push_back(2.71828);
  values.push_back(-3.14159);

  std::vector<double> above = cleanse(values, false); // 2
  for (int i = 0; i < above.size(); ++i)
    std::cout << above[i] << ' ';
  std::cout << '\n';

  std::vector<double> below = cleanse(values, true); // 2
  for (int i = 0; i < below.size(); ++i)
    std::cout << below[i] << ' ';
  std::cout << '\n';
  return 0;
}
```

注释 1：基于布尔标志的过滤器

注释 2：指定用于过滤大于或小于 0 的布尔标志

**分析**

cleanse 函数对 vector 进行迭代；确定是否过滤高于或低于的值；检查值的符号；如果匹配所需范围，则将其复制到结果 vector 中。虽然操作很简单，很明显该函数正在完成两种行为，由标志切换。开发人员可能记得这种行为的微妙之处，但新开发人员必须花时间弄清楚像 cleanse(values, false) 这样的调用意味着什么。这位开发人员必须找到该函数的源代码，并进行阅读；并且在没有有意义的注释（如本例所示）的情况下，跟踪代码，直到清楚发生了什么。

这种编程风格需要付出很多努力才能正确（有效）和理解（可读性）。这是一种糟糕的设计，希望节省精力和避免重复。现有的代码库中充满了例子。

**解决**

类似函数中的代码重复是不可避免的，而消除重复的努力可能会导致更严重的问题。目标是尽量减少或消除多个函数或方法之间的知识传播。过滤器下方和过滤器上方的代码之间没有共享知识，它们之间存在相当多的重复。

当过滤之间的控制流（控制结构）相同时，通常会引入问题。这种情况导致开发人员将不同的部分（过滤功能）组合成一个通用结构，通常包含重复的部分（我们正在努力消除这种重复）。引入由布尔值选择的控制流路径，则要努力将不同函数中的控制流重复合并为一个函数。这不仅浪费开发时间并使测试更加困难，而且也无法解决其编写目标。

几个简短的函数是理想的，但并非丑陋的重复。简短的函数更易于命名，从而导致调用代码中的函数直观。这些函数易于测试和验证，并且易于阅读和编写，往往会重复控制流逻辑。正如 Meat loaf 所说，“Two Out Of Three Ain’t Bad(三分之二也不错)”[9]。

以下代码将过滤逻辑拆分为两个函数，并适当地命名它们。消除结构化代码的冲动，并将每个函数的目的（其逻辑）分离为一个单元。

[9]译者注：“Two Out Of Three Ain’t Bad”是 Meat Loaf 演唱的歌曲，由 Jim Steinman 作词作曲，收录于《The Collection》专辑中。

清单 *11.18* 使用单独的函数进行过滤

```
std::vector<double> filter_above(const
    std::vector<double>& values) { // 1
  std::vector<double> new_values;
  for (int i = 0; i < values.size(); ++i)
    if (values[i] > 0)
      new_values.push_back(values[i]);
  return new_values;
}

std::vector<double> filter_below(const
    std::vector<double>& values) { // 2
  std::vector<double> new_values;
  for (int i = 0; i < values.size(); ++i)
    if (values[i] < 0)
      new_values.push_back(values[i]);
  return new_values;
}

int main() {
  std::vector<double> values;
  values.push_back(3.14159);
  values.push_back(-1.23456);
  values.push_back(2.71828);
  values.push_back(-3.14159);

  std::vector<double> above = filter_above(values); // 3
  for (int i = 0; i < above.size(); ++i)
    std::cout << above[i] << ' ';
  std::cout << '\n';

  std::vector<double> below = filter_below(values); // 3
  for (int i = 0; i < below.size(); ++i)
    std::cout << below[i] << ' ';
  std::cout << '\n';
  return 0;
}
```

注释 1：具有特定行为的单一用途函数

注释 2：具有不同特定行为的另一个单一用途函数

注释 3：使用简单，没有可疑参数

作为一项挑战，弄清楚如何使用函数模板来实现比较功能。编写一个接受 `vector` 和函数模板参数的单个过滤器函数。可以在此示例中使用 `std::greater<double>` 和 `std::less<double>`。有关提示，请参阅以下 C++ 参考网页：https://mng.bz/gaan。

**建议**

- 让每个函数在其职责上独立，并明确说明明确的目的。
- 不要害怕在函数之间复制控制结构—如果在一个函数中有效，那么在另一个函数中也能有效。
- 不要过分担心跨函数重复代码；可以将知识隔离到一个函数或一组协调的函数 （可能是个类）。

# 第 12 章 通用编码

本章内容

- 非 C++ 的异常
- 正确的循环设置和执行
- 冗长的编码和关键字的过度使用
- 使用已删除的指针

不当的设计和实现并不局限于 C++。以下许多错误都是普遍存在的，并且出现在许多语言中。这种普遍性并不意味着以下问题不重要，只是它们可能出现在任何代码库中。语言不能脱离其运行的架构和机器，尽管大多数语言都试图尽可能多地抽象这些细节。Java 消除了许多机器细节，但仍然显示出约束的迹象。例如，数据类型的位大小限制了可能值的范围——无论尝试抽象多少，似乎都无法绕过这些机器细节。

C++ 比大多数语言更接近机器，所以经常出现与机器相关的细节和问题也就不足为奇了。以下错误解决了语言或机器细节影响正确实现的程序的几种情况。

## 12.1. 错误 91： 不正确处理除以零

这种错误会影响可读性和有效性。当恢复是可能且可行的情况下，处理异常是一种处理问题的有效技术。

一些学生对计算机无法处理整数（或表示实数的浮点值）这一数字类别中的大量数值感到惊讶。计算机速度快、精确且准确，但功能有限，尤其是在表示数值范围方面。

包含除法步骤的计算问题，有一个需要解决的特殊问题。当尝试用零除整数时，数学家们也会变得敏感。

**问题**

许多计算都涉及除法。这包括除法操作符（/）和不太明显的模数操作符（%）。经验丰富的开发者可能会通过惨痛的经历了解到整数除以零会导致程序以浮点异常（FPE）终止。许多新手开发者尚未理解这一点，并且很少通过在尝试计算之前检查除数的值来保护他们的除法代码。以下代码由一位有意识的开发者编写，考虑了这种数学错误的可能性。

清单 *12.1* 未检查的整数除法

```
bool divides(int b, int a) {
  return a % b == 0; // 1
}

int main() {
  int x = 0;
  int y = 42;
  try {
  if (divides(x, y))
    std::cout << x << " divides " << y << '\n';
```

```
  } catch(...) { // 2
    std::cout << "probably divide by zero issue\n";
  }
  return 0;
}
```

注释 1：浮点异常的潜在来源

注释 2：通用 catch 块

当运行此代码时，开发者惊讶地发现他们的程序仍然因 FPE 而崩溃。尽管有良好的意愿，但问题并没有解决，不可能恢复。有些系统不会崩溃，但这并不意味着代码运行正常。

### 分析

我的系统上运行清单 12.1 中的代码会导致出现以下意外消息：

```
Floating point exception
```

其他系统和编译器可能会产生不同的消息。

在这里，catch-all 异常处理程序不会响应此异常。catch-all 处理程序的目的是处理未捕获的异常。

### 解决

这里有个谜语：什么时候异常不再是异常？当它不是 C++ 异常时。

开发者需要处理 IEEE 754 标准（定义计算中的浮点运算）和 C++ 版本之间的术语混淆。术语浮点数异常让许多学习过编程语言的开发人员感到困惑。

CPU 确定除以零的运算有效，因此拒绝执行。在 C++ 中，这会引发一个信号，从而触发信号浮点异常（SIGFPE）处理程序。此信号不是 C++ 异常，通用 catch 块不会执行。SIGFPE 处理程序位于开发人员代码之外，是 C++ 运行时代码的一部分，这种替代控制流完全绕过了 catch 块。

一种可能性是让开发者定义一个除以零的处理函数，并设置 SIGFPE 处理程序来调用该函数。但在调用该函数后，将控制权返回到有意义的点很困难，甚至不可能。

最好理解术语异常的双重用途，并避免除以零的问题。以下代码通过抛出异常，来避免该问题。这是一个 C++ 异常，因此所有典型的异常处理技术均按预期工作。

清单 12.2 预期除零并抛出 C++ 异常

```
bool divides(int b, int a) {
  if (b == 0) // 1
    throw std::invalid_argument("divisor is zero"); // 2
  return a % b == 0;
}

int main() {
  int x = 0;
  int y = 42;
  try {
    if (divides(x, y))
```

```
      std::cout << x << " divides " << y << '\n';
    } catch(std::invalid_argument ex) { // 3
      std::cout << ex.what() << '\n';
    }
    return 0;
  }
```

注释 1：预测除以零的问题
注释 2：如果发生，则抛出 C++ 异常
注释 3：处理情况

**建议**

- 请谨慎理解术语，不要假设 C++ 对所有东西总有定义。
- 如果想了解有关计算机中浮点数的表示、限制和用途的更多信息，请查看 IEEE 754 标准。
- 计算是有界的；数据类型不代表无限的值集—32 位有符号整数的范围是 –2,147,483,648 到 2,147,4 83,647，超出此范围的值都无效。
- 值可能会下溢和上溢，在接近数值边界时要小心。

## 12.2. 错误 92： 不正确地使用循环中的 continue

这种错误会影响正确性和有效性。编写循环通常是靠肌肉记忆来完成的，注意力不集中可能会导致问题，这些问题可能会潜伏一段时间后才会显现出来。

**问题**

一些循环具有用于处理迭代的附加逻辑，C++ 提供了 break 和 continue 关键字来修改循环控制。

每个循环都包含四个部分，其中一些部分可能为空。了解这些部分对于设计正确的循环至关重要。第一个是“初始化部分”，其中设置循环控制变量或其他值 - 是循环的先决条件。第二个是“延续性测试”，回答是否执行循环体 - 通常测试循环控制变量。第三个是“执行体”，循环的原因 - 每次迭代时执行的代码。最后一个是“更新部分”，其中修改循环控制变量，使延续测试越来越接近终止条件。此顺序是为 while 循环（包括 for 循环）指定，延续测试成为 do 循环中的第四部分，其他部分保持其现有顺序。

许多开发者倾向于使用 for 循环，循环控制变量的作用域仅限于循环内，并且编写速度更快。熟悉 for 循环和替代控制流（break 和 continue 关键字）可能会让粗心的开发人员在使用 while 或 do 循环时遇到问题。当 for 循环执行 continue 关键字时，控制流会回到循环的“顶部”（参见以下列表）或更新部分，而更新部分恰好位于循环的顶部。

清单 *12.3* 带有 *continue* 关键字的 *while* 循环

```
int main() {
  int x = 10;
  while (x > 0) {
    if (x % 3 != 0)
```

```
      continue; // 1
    std::cout << x << " is divisible by 3\n";
    --x;
  }
  return 0;
}
```

注释 1：控制流至循环顶部——while 关键字

该代码是一个无限循环，将一直执行。

### 分析

在 while 或 do 循环中，continue 关键字使控制流向循环顶部，即连续测试或 do 关键字。这些循环和 for 循环之间的这种变化至关重要。

### 解决

for 循环与 do 和 while 循环之间的差别在于，for 循环在循环的“顶部”具有初始化、延续和更新部分；只有主体未包括在内。其他两个循环将这些部分分散开来，顶部要么是单独的 do 关键字，要么是带有延续测试的 while 关键字。for 循环在执行 continue 关键字后立即调用更新部分。执行 continue 关键字时，while 和 do 循环从不调用更新部分。这一差异对于正确理解如何编写这些代码至关重要循环，以下代码复制了 continue 关键字之前的更新部分。

清单 *12.4* 在 *continue* 关键字之前正确执行更新部分

```
int main() {
  int x = 10;
  while (x > 0) {
    if (x % 3 != 0) {
      --x; // 1
      continue;
    }
    std::cout << x << " is divisible by 3\n";
    --x;
  }
  return 0;
}
```

注释 1：在 continue 关键字之前执行更新部分

这对于继续推动循环控制变量达到其终止条件至关重要。如果没有这个重复的部分，循环控制变量在本次迭代中不会发生变化，并产生无限循环。

### 建议

- 对于 while 和 do 循环，在执行 continue 关键字之前，始终先执行更新部分。
- 最好使用 for 循环，这种行为对它们来说不是问题；while 或 do 循环中的重复更新部分可能是一种代码异味，这意味着设计较差或不正确。

## 12.3. 错误 93： 未能将已删除的指针设置为 NULL

此错误会影响正确性。指针经常用于管理动态资源；删除它们对于正常运行至关重要。通过指针访问已删除的资源通常可行，但其行为未定义。

### 问题

开发人员面临着许多通过原始指针管理动态资源的情况，现代 C++ 使用智能指针解决了这个问题。在没有智能指针的时代，开发人员必须小心管理动态资源。资源管理通常可以正确完成，并且不会遇到未定义的行为。然而，存在几种情况，在删除资源后访问资源，因为代码复杂，需要显式删除，但事实并非如此。当从异常中恢复的代码错误处理资源时，就会发生这种情况。

就我个人而言，曾经参与的一个项目就出现了 12 次此问题。静态代码分析器发现了这些情况，但开发人员或维护人员却没有发现。以下代码是一个非常简化的案例，说明了这个问题。

清单 *12.5* 指针删除后访问的动态资源

```
class Person {
private:
  std::string name;
  int age;
public:
  Person(const std::string& name, int age) : name(name), age(age) {}
  const std::string& getName() { return name; }
  int getAge() { return age; }
};

int main() {
  Person* anne = new Person("Annette", 28);
  if (anne) // 1
    std::cout << anne->getName() << " is " << anne->getAge()
      << " years old\n";
  delete anne;
  if (anne) // 2
    std::cout << anne->getName() << " is " << anne->getAge()
      << " years old\n";
  return 0;
}
```

注释 1：一个有效的测试，确保对象已创建

注释 2：一个看似有效的测试；对象已销毁且可能访问。

### 分析

创建动态资源，测试其有效性，处理并删除。问题出现在删除之后，其中测试指针的有效性，并错误地用于访问资源。不同的编译器和系统的组合对此代码的反应不同，但通过已删除指针访问资源的尝试都需要纠正。在最好的情况下，程序崩溃，阻止进一步访问。这种未定义行为的情况可能看起来有效，但却是相当危险的错误。

**解决**

解决这个问题最简单的方法是删除指针后将其置空。如果删除操作是在作用域的末尾，则可以忽略此建议，但一般情况下最好这样做。通过空指针访问比成功访问删除了资源，访问空指针导致的崩溃是很明显的编程问题。以下代码显示了这一微小的更改，产生了显著的效果。

清单 *12.6* 指针为空以避免删除后的访问

```
class Person {
private:
  std::string name;
  int age;
public:
  Person(const std::string& name, int age) : name(name), age(age) {}
  const std::string& getName() { return name; }
  int getAge() { return age; }
};

int main() {
  Person* anne = new Person("Annette", 28);
  if (anne)
    std::cout << anne->getName() << " is " << anne->getAge() << " years old\n";
  delete anne;
  anne = NULL; // 1
  if (anne) // 2
    std::cout << anne->getName() << " is " << anne->getAge() << " years old\n";
  return 0;
}
```

注释 1：将已删除的指针清零

注释 2：阻止通过已删除的指针进行访问的测试

**建议**

- 始终将已删除资源的指针清空；在经典 C++ 中使用 0，在现代 C++ 中使用 nullptr。

## 12.4. 错误 94： 未能直接返回计算得到的布尔值

这个错误会影响可读性和有效性。我们经常会编写函数来确定某事物的真假，这样的函数称为“谓词”—返回布尔值 true 或 false。

**问题**

一些谓词以一些值作为参数，根据这些值进行计算以确定结果，然后返回该结果。通常使用局部变量，将局部变量初始化为某个状态，执行计算，确定结果并将其保存在变量中，再返回结果。

然而，这种冗长的方法背后隐藏着一个重大问题。请考虑以下代码，确定数字是偶数还是奇数。

清单 *12.7* 计算布尔值并返回

```cpp
bool isEven(int n) {
  bool even = false; // 1
  if (n % 2 == 0) // 2
    even = true;
  else
    even = false;
  return even; // 3
}

int main() {
  int n = 42;
  if (isEven(n))
    std::cout << n << " is even\n";
  else
    std::cout << n << " is odd\n";
  ++n;
  if (isEven(n))
    std::cout << n << " is even\n";
  else
    std::cout << n << " is odd\n";
  return 0;
}
```

注释 1：声明变量
注释 2：计算其值
注释 3：返回值

`isEven` 函数的编码正确但冗长。隐藏的问题是开发人员需要认识到，模数计算的结果就是返回的结果。`if` 测试确定计算的值，找出计算值，并将变量设置为该结果。变量精确地包含计算所做的操作—`if` 测试没有任何收获。

### 分析

使用局部变量来保存计算的中间值是经常需要的。但如果计算就是结果，则无需保留该值。

### 解决

`isEven` 函数应删除 `if` 测试代码并返回计算结果。这种方法更易读，也更容易写，并更好地理解计算和返回值之间的关系。

这个问题不仅限于布尔值的计算，许多其他函数（非谓词）也会计算返回值而无需进一步修改。这种技术也适用于这些情况，尽量使函数非常简短且非常清晰。

清单 *12.8* 直接返回计算的布尔值

```cpp
bool isEven(int n) {
  return n % 2 == 0; // 1
}
int main() {
  int n = 42;
  if (isEven(n))
```

```
    std::cout << n << " is even\n";
  else
    std::cout << n << " is odd\n";
  ++n;
  if (isEven(n))
    std::cout << n << " is even\n";
  else
    std::cout << n << " is odd\n";
  return 0;
}
```

注释 1：直接返回计算结果

**建议**

- 保持函数简短而简洁。
- 尽可能直接返回计算值。

## 12.5. 错误 95： 对表达式的利用不足

这个错误主要针对的是效率和可读性。很多开发者更喜欢使用 `if` 语句，而不是强大的三元操作符。

**问题**

教科书会教授 `if/else` 条件结构，并在代码示例中使用它们。通常，三元操作符很少使用，但包含通常是偶然的。需要包含有关 `if/ else` 语句和三元表达式的区别和使用的详细讨论。语句不返回值；表达式总是返回值。

考虑以下代码，其中谓词 `isEven` 决定从 `if` 测试中遵循哪条路径。真路径执行 `if` 关键字后面的语句，假路径执行 `else` 关键字后面的语句（如果存在）。

清单 *12.9* 过于冗长的条件代码

```
bool isEven(int n) {
  return n % 2 == 0;
}
int main() {
  int n = 42;
  if (isEven(n)) // 1
    std::cout << n << " is even\n";
  else
    std::cout << n << " is odd\n";
  ++n;
  if (isEven(n))
    std::cout << n << " is even\n";
  else
    std::cout << n << " is odd\n";
  return 0;
```

```
}
```

注释 1：冗长的 if/else 语句

**分析**

代码确定值的均匀性并输出结果。由于使用 if 语句进行此确定，输出发生在 true 或 false 路径上，或者该路径必须使用测试结果设置局部变量。如果选择第二个选项，则输出使用局部变量的内容。if/ else 语句无法返回值，但只能计算值。计算代码通常在真路径和假路径上重复，使代码冗长且阅读起来有一些复杂。

**解决**

if/else 结构无法返回值，三元操作符本质上也是一个 if/else 结构，是一个可以产生值的表达式。使用操作符作为表达式意味着可以直接计算值，而无需分配局部变量或重复代码。

清单 12.10 显示了操作符返回值的用法。表达式应该括在括号中，以防止编译器混淆。同样重要的是，括号将表达式与其周围的代码区分开来，使其更易于阅读。本讨论并不是说所有 if/else 结构都应强制放入表达式中，而是说，在许多情况下，这种方法很合适。输出语句尤其受益于使用三元操作符。

清单 *12.10* 最小化条件代码

```
bool isEven(int n) {
  return n % 2 == 0;
}
int main() {
  int n = 42;
  std::cout << n << " is " << (isEven(n) ? "even" : "odd") << '\n'; // 1
  ++n;
  std::cout << n << " is " << (isEven(n) ? "even" : "odd") << '\n';
  return 0;
}
```

注释 1：使用三元操作符最小化 if/else 逻辑

**建议**

- 使用三元操作符直接计算值，消除局部变量和一些重复代码；此示例代码并不理想，最好将其放入函数中。
- 查找可以从使用表达式形式中受益的输出语句。

## 12.6. 错误 96： 使用多余的 else

这个错误主要针对的是有效性和可读性。许多教科书倾向于展示完整的 if/else 结构，而不描述它们何时会掩盖代码的意图。

**问题**

当开发人员学习新的语言技术时，学习的风格经常会融入到其使用中。以下代码是确定一组值

的算术平均值的简单示例。C++ 语言提供了使用 if/else 构造从一组选项（在本例中为三个）中进行选择的方法。第一个选项是没有值，第二个选项是一个值，第三个选项是多个值。每个选项必须以不同的方式处理，使用演示的结构也很自然。但是，读者必须在这些选项之间做出决定，并确定哪些是短路，哪些是主要计算。

短路选项是直接计算回报的选项（递归函数中的基本情况），将这些选项与计算混合在一起，通常不如此代码所示那么明显。

清单 *12.11* 使用 *if/else* 链

```
double mean(const std::vector<double>& values) {
  unsigned int size = values.size();
  if (size == 0) // 1
    throw std::invalid_argument("no values to average");
  else if (size() == 1)
    return values[0];
  else {
    double sum = 0.0;
    for (int i = 0; i < size; ++i)
      sum += values[i];
    return sum/size;
  }
}
int main() {
  std::vector<double> values;
  values.push_back(3.14159);
  values.push_back(2.71828);
  std::cout << mean(values) << '\n';
  return 0;
}
```

注释 1：一个 if/else 链，使计算的返回条件变得模糊

**分析**

返回直接计算结果的两个选项（好吧，抛出异常可能不是计算，但还是接受它），与计算选项暗中混合在一起。逻辑从无到一，再到多个值。不需要按照这种逻辑顺序编写，并且在代码更广泛的情况下，计算可能会完全混合所掩盖。最好采用一种更好的方法，来理解单独的选项并使其清晰明了。

**解决**

清单 12.12 对清单 12.11 中的代码进行了轻微修改，但这样做使结果更容易理解。处理了两种短路情况—要么是 true，并且处理了 throw 或 return，要么不处理。如果两种情况都是 false，则执行主计算情况。

这两个例子在视觉上有所区别。前者使用了更多的缩进，这总是会给短期记忆带来负担。

后者省去了缩进，更有力地阐明了如果前两种情况不成立，它们很快就会遗忘。计算与 if 关键字的缩进级别相同，清楚地表明如果控制流到达那么远，此代码将执行。减少 else 关键字的数量可让读者更快地省去短路情况并专注于主要计算任务。

清单 *12.12* 使用短路逻辑

```
double mean(const std::vector<double>& values) {
  unsigned int size = values.size();
  if (size == 0) // 1
    throw std::invalid_argument("no values to average");
  if (size() == 1) // 1
    return values[0];
  double sum = 0.0;
  for (int i = 0; i < size; ++i)
    sum += values[i];
  return sum/size;
}

int main() {
  std::vector<double> values;
  values.push_back(3.14159);
  values.push_back(2.71828);
  std::cout << mean(values) << '\n';
  return 0;
}
```

注释 1：返回条件的短路计算

**建议**

- 组织一个函数，首先测试无效或失败的情况，如果通过，则将主要计算放在后面。
- 尽可能减少缩进级别。
- 只要有意义就使用短路情况；这些短路情况会直接计算或抛出，并且在尝试理解主代码时很快就会遗忘。

## 12.7. 错误 97： 未使用辅助函数

这种错误会影响有效性和可读性，在某些情况下，可能会对正确性产生负面影响。用户代码永远不必知道算法或函数如何工作的具体细节；相反，它应该直观地调用函数，只提供解决问题所需的数据。

**问题**

考虑以递归方式实现二分搜索的情况。为了尽量减少性能问题，数据集合通过引用传递。起始和结束索引值会针对每个后续递归调用进行调整。

用户代码调用递归函数，并提供初始的起始和结束索引值。用户代码倾向于对索引值进行硬编码，必须确定函数调用的机制以及数据容器与该调用的关系。如果这样做，值就很脆弱，如果在未适当改变索引值的情况下改变容器大小，将导致不正确的行为。

希望客户始终使用正确的起始和结束索引值。他们可能会猜测他们的理解是否错误，或者经验是否有限。这种方法很危险。

清单 *12.13* 公开实现细节的递归函数

```
bool bin_search(const std::vector<int>& values, int key, int start, int end)
{
  int mid = start + (end-start) / 2;
  if (values[mid] == key) // mention previous 'else' issue
    return true;
  else if (values[mid] < key)
    return bin_search(values, key, start+1, end);
  else
    return bin_search(values, key, start, end-1);
}

int main() {
  std::vector<int> values;
  for (int i = 0; i < 100; ++i)
    values.push_back(i);
  std::cout << bin_search(values, 55, 0, 99) << '\n'; // 1
}
```

注释 1：用户代码必须了解起始和结束索引值。信息量太大了！

**分析**

编写 `main` 函数的开发人员得到的索引值是正确的，尚不清楚是否理解结束索引必须是容器大小（`100`）或最大索引（`99`）。在略有不同的情况下，可能会使用结束索引 `100`。要更正索引值，开发人员必须了解递归搜索如何使用该值，此信息负载会对有效性产生负面影响。

理想情况下，搜索函数的开发人员应该对其进行编码，以便用户无需了解索引值或函数的实现方式。在类的开发中，封装会抽象这些细节，并提供一个最小的、希望是直观的界面，函数也应如此。

**解决**

可以通过创建辅助函数来封装递归函数的功能。此函数采用搜索函数的名称，但消除了有关索引值或其他实现细节的任何知识。函数重载允许名称保持不变。递归函数名称无需更改即可表明递归（除非这对开发人员来说是有用的文档细节），辅助函数（可能是由递归函数的开发人员编写的）确定有关起始和结束索引值的详细信息。

辅助函数的另一个好处是，结束索引是根据容器计算的，因此消除了对该值进行硬编码。如何调用递归函数的实现细节抽象出来，用户的接口最小化为仅包含基本数据 - 值的容器和键。

清单 `12.14` 在用户代码和递归函数之间引入了辅助函数，与类的公共方法非常相似。这样做的目的是减少用户开发人员的认知负担，并允许在以后需要时重新实现该函数。

清单 *12.14* 从抽象的辅助函数调用的递归函数

```
bool bin_search(const std::vector<int>& values, int key, int start, int end) {
  int mid = start + (end-start) / 2;
  if (values[mid] == key)
    return true;
```

```
  if (values[mid] < key)
    return bin_search(values, key, start+1, end);
  return bin_search(values, key, start, end-1);
}

bool bin_search(const std::vector<int>& values, int key) { // 1
  return bin_search(values, key, 0, values.size()-1);
}

int main() {
  std::vector<int> values;
  for (int i = 0; i < 100; ++i)
    values.push_back(i);
  std::cout << bin_search(values, 55) << '\n'; // 2
}
```

注释 1：在用户和递归函数调用之间进行转换

注释 2：使调用尽可能简单

**建议**

- 尽量减少函数调用者需要提供的数据量；坚持只提供必要的数据。
- 寻找辅助函数可以最小化界面，并简化调用函数的情况。

## 12.8. 错误 98： 错误地比较浮点数值

这个错误影响正确性，同时对可读性和有效性产生轻微的负面影响。比较数字似乎简单明了，但浮点值应该是近似值，而不是精确值。

**问题**

数学直觉可能会告诉我们，除数除以一个数，结果乘以除数应该得到原始数。代数强调了这一真理，大多数人毫不犹豫地接受了它。当开始用浮点数编程时，这种直觉可能会让我们走上歧途。

考虑清单 12.15 中的代码，其中一个简单的除法问题产生了意想不到的结果。下面的比较使用了这种直觉，并期望结果相等。结果非常非常接近但不精确，开发人员熟悉使用相等操作符来确定值的等价性，这种熟悉会导致他们误用该操作符。

开发人员原本预计比较会返回 `true` 结果，但却惊讶地发现结果竟然是 `false`。直觉告诉我，比较应该是 `true`。

清单 *12.15* 使用相等操作符进行比较

```
int main() {
  double amount = 100.0 / 3;
  std::cout << (amount == 33.3333333333 ? "true" : "false") << '\n'; // 1
  return 0;
}
```

注释 1：预期结果是正确的，但事实并非如此

### 分析

计算机是功能有限的机器，只有少量的位分配来表示浮点值。浮点值是一组最小实数的近似值。然而，从代数到编程，却遇到了意想不到的惊喜。必须记住的是，浮点数无法表达大多数实数。

如果开发者将除法结果乘以 3，则与值 100 的比较会成功，但并不正确。除法结果的实际值不是以数字 3 结尾，但最后一位数字可能是 4。IEEE 754 标准定义了此行为；所有符合标准的编译器都会表示尊重。在我的计算机上，将输出精度设置为 18 位，结尾数字是... 57 ；更多数字将以出乎意料的形式表示。这些非 3 数字不是我们所期望的，也不是我们的直觉告诉我们的。如果有人提出反对意见，说增加更多数字可以解决这个问题，那么问题就变成了到底需要多少位数字？没有数字可以精确地表示该值。关键的想法是浮点数是近似值，很少是正确的，但对于大多数应用来说足够接近。这个问题不仅限于计算机；三分之一的精确值无法用十进制（基数为 10）数来表示——有人喜欢十二进制（基数为 12）数制吗？

### 解决

由于浮点数是近似值，不能使用 operator== 进行比较。需要进行比较以确定两个近似值之间的接近程度。如果它们之间的差异足够小，则应考虑等效。清单 12.16 中的代码演示了 delta-epsilon 比较方法。我将其作为“足够接近”函数来讲述。

Delta 是数学家用来表示值之间的差异的希腊字母——可以将其视为从一个值中减去另一个值。Epsilon 是一个希腊字母，表示非常小的值（字符串中，代表空字符串）。其目的是找出两个值之间的差异，看看它是否小于一个微小的数字。如果是这样，这两个值就“足够接近”，即使它们的最后数字可能不同，也可视为等价。

必须从较大的值中减去较小的值才能产生有意义的差值。由于不知道哪个值更大，可计算差值的绝对值，确保结果为正数或零。Epsilon 通常定义为 $10^{-14}$，大致是可靠地确定 64 位双精度值的最小有意义值。但请确保系统可以有效地使用，较小的位大小将需要较大的 epsilon 值。此外，如果可以降低所需的精度，请使用较大的 epsilon 值。以下代码使用精度 $10^{-10}$，因为对于这个问题来说，这已经足够接近了。

清单 *12.16* 使用 *delta-epsilon* 方法进行比较

```
bool close_enough(double value, double target, double epsilon=1e-14) { // 1
  return fabs(value-target) < epsilon; // 2
}

int main() {
  double amount = 100.0 / 3;
  std::cout << (close_enough(amount, 33.3333333333, 1e-10) ? "true" : "false") << '\n';
  return 0;
}
```

注释 1：默认为一个合理的小值

注释 2：确定差异是否足够接近，可以视为相等

### 建议

- 代数和编程是相关的，但并不等同。
- 通过比较浮点值与任意小值的差来比较浮点值。

## 12.9. 错误 99： 浮点数到整数的赋值

这个错误影响正确性。将浮点值转换为整数值其实比听起来要复杂得多。

### 问题

许多学生学习过 round 函数，应该始终使用浮点值和整数值之间的转换。他们可能还学习过使用强制类型转换进行截断。这两种方法当然会将浮点值转换为整数，但并不代表所有可能性。

下面的代码演示了这一基本理解，但缺少两个重要的转换函数。

清单 *12.17* 使用截断和舍入将浮点数转换为整数

```
int main() {
  std::vector<double> values;
  values.push_back(3.14); values.push_back(2.71); values.push_back(1.5);
  values.push_back(-1.5); values.push_back(-2.71);
  values.push_back(-3.14);
  std::cout << "value trunc round\n";
  for (int i = 0; i < values.size(); ++i) {
    double v = values[i];
    std::cout << std::setw(5) << v
      << std::setw(6) << static_cast<int>(v)
      << std::setw(6) << round(v)
      << '\n';
  }
  return 0;
}
```

清单 12.17 中的代码输出演示了这两个转换函数：

```
value trunc round
 3.14     3     3
 2.71     2     3
  1.5     1     2
 -1.5    -1    -2
-2.71    -2    -3
-3.14    -3    -3
```

### 分析

上述代码的结果显示了这些常用函数，但这些结果不足以解决开发人员的所有问题。如果计算搬运 3.14 件物品所需的包装箱数量，这些转换函数将不会返回正确的值 4。翻转问题的符号，如果 -$3.14 是余额（即欠款金额），需要对账户采取什么操作来偿还债务？提供的函数表明该账户需要 -$3 操作（提取 3 美元），而正确的响应将是 -$4。计算必要结果的这些差距表明，需要其他函数来完成转换。

换句话说，如果只使用这两个函数，上述两个问题将需要其他代码来确定正确的行为，有时甚至会导致解决方案出错。更好的方法是使用一个对转换有意义的函数，而无需很复杂的逻辑。

**解决**

除了典型的截断和舍入之外，将浮点值转换为整数时还必须考虑两个附加函数。我们将分析和描述这四个函数中的每一个，并给出一个示例来巩固解释。许多解释中，忽略了值的符号，这是一个错误。数值必须始终考虑值的全部范围。

截断应被认为是向零截断。这种添加意味着当值的小数部分截断时，结果值会朝特定方向移动。无论初始值是什么，当小数部分删除时，结果都会比以前更接近零。有关此移动的直观表示，请参见图 12.1。

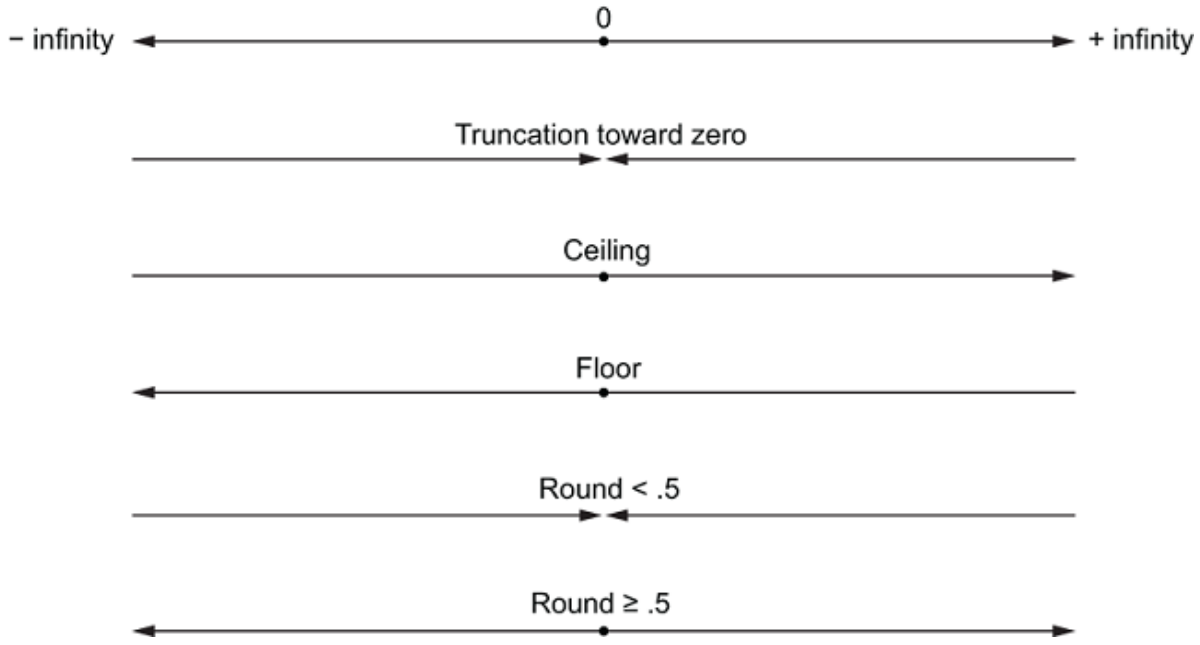


图 12.1 几个函数的转换方向

截断的一个例子是人的年龄（尽管很小的孩子似乎对小数年份很着迷）。生日之间的整年可视为精确浮点值的年龄，小数部分删除，所以截断值将向零移动，直到达到整数。一个 3.14 岁的人是三岁，直到他变成四岁。

转换中必须考虑 std::floor 函数。与截断函数不同，floor 函数从精确值移向负无穷大。图 12.1 演示了这一点。对于正值，看起来就像截断，但负值的工作原理却大不相同。

制造商就是底线函数的一个例子。考虑一家生产冰淇淋三明治的公司。如果一盒三明治有 12 件，而该公司生产了 150 件，那么只能制作 12 个盒子。没有人愿意买 1 盒后发现里面只有 6 个三明治（尽管如果他们将其作为减肥辅助品出售，可能会起作用）。

round 函数的工作原理与我们在算术中学到的相似 – 如果小于一半，则向较低值移动；如果大于一半，则向较高值移动；当恰好为一半时，通过向较高值移动来打破平局。图 12.1 显示了使用此函数移动的方向。将舍入描述为一种移动非常困难，这取决于值和符号。

一个简单的例子就是测验成绩：如果学生得分 89.6，就认为自己应该得 90 分。我总是尽量给学生打出有利的分数，但没有理由给得分 89.4 的人打 90 分。

最后，还必须考虑 std::ceil 函数，移动方向与 floor 函数相反。此函数始终朝正无穷大方向移动。任何小数部分都使其有资格移动到下一个最高整数值。有关此函数移动的图形描述，请参见图 12.1。

`std::ceil` 的一个例子是购买几加仑油漆用于装饰。假设一面墙的面积为 165 平方英尺，一加仑油漆可以覆盖 144 平方英尺。遗憾的是，家装中心不出售零碎加仑，所以必须购买 2 加仑油漆来粉刷这面墙。

清单 12.18 中的代码，在前面提到的函数中引入了 `std::floor` 和 `std::ceil` 函数。问题的性质决定了使用这四个选项中的哪一个。处理小数部分对于现实世界的问题来说至关重要，若忽视它则会产生负面影响。

清单 *12.18* 从浮点型到整型

```
int main() {
  std::vector<double> values;
  values.push_back(3.14); values.push_back(2.71); values.push_back(1.5);
  values.push_back(-1.5); values.push_back(-2.71);
  values.push_back(-3.14);
  std::cout << "value trunc floor round ceil\n";
  for (int i = 0; i < values.size(); ++i) {
    double v = values[i];
    std::cout << std::setw(5) << v
      << std::setw(6) << static_cast<int>(v)
      << std::setw(6) << std::floor(v)
      << std::setw(6) << round(v)
      << std::setw(5) << std::ceil(v)
      << '\n';
  }
  return 0;
}
```

上述代码的输出结果如下：

```
value trunc floor round ceil
 3.14     3     3     3    4
 2.71     2     2     3    3
  1.5     1     1     2    2
 -1.5    -1    -2    -2   -1
-2.71    -2    -3    -3   -2
-3.14    -3    -4    -3   -3
```

回到分析部分中提到的问题，快速回顾一下就会发现选择适当的转换函数的好处，并且这样做无需添加其他逻辑。包装箱问题是存在非零小数部分的情况，需要一个完整的盒子，而不管小数部分的大小如何；正确的选择是使用天花板函数。账户问题，其中余额为 `-$ 3.14`，需要对账户（再次提款）中存入 4 美元以支付费用。此问题使用 `floor` 函数来提取正确的金额。

值得庆幸的是，银行和其他金融机构不使用浮点数来表示货币单位。他们从惨痛经历中了解到，当这种情况发生时，狡猾的开发人员可以“赚大钱”。这种技术就像切香肠一样，很容易掩盖——哪个金融机构愿意承认这样的内部欺诈行为?

需要更广泛的转换函数才能正确选择适当的转换方法，而无需确定边界条件并添加潜在的复杂逻辑。可以通过选择单个函数解决问题时，意图表达得更好（可读性），并且更容易正确编码（有效性）。

**建议**

- 理解要解决的问题，以及小数部分的含义。
- 选择正确的函数将浮点值转换为整数。
- 理解转换函数向负无穷、正无穷或零的移动方式。

## 12.10. 错误 100： 忽略编译器警告

此错误主要影响正确性，对可读性和有效性影响较小。编译器警告用于目的；忽略是不合理的，有些会导致未定义的行为。许多警告不会影响程序的运行特性，其可能会影响程序的运行特性；保留它们可能会很危险。更重要的是，这些问题会影响未来的开发，并让粗心的维护开发者陷入困境。

**问题**

清单 12.19 中的代码编译良好，并且只生成一个警告（在我的系统上；您的系统可能有所不同）。编译未指定额外的检查。鉴于这个相对无害的警告，继续开发是非常诱人的选项。在忽略这些看似无害的警告足够长的时间后，它们会累积起来，直到变得无法处理。

清单 *12.19* 带有多个问题的简单程序

```
int compute(int* x, int y) {
  int* pos = x;
  if (*x > NULL) // 1
    return *x + *x;
  return 0;
}

int main() {
  int* x; // 2
  int res = compute(x, 0); // 3
  std::cout << "hello, world\n";
  return 0;
}
```

注释 1: NULL 使用不正确
注释 2: x 具有未定义的值
注释 3: 这个未定义的值被传递给了函数

启用警告检查（使用 -Wall 和 -Wextra）后，输出突然变得很糟糕。使用此警告检查进行编译的结果如下：

```
warning-bad.cpp: In function 'int compute(int*, int)':
warning-bad.cpp:6:14: warning: NULL used in arithmetic [-Wpointer-arith]
    6 | if (*x > NULL) ## 1
      |    ^~~~
warning-bad.cpp:5:10: warning: unused variable 'pos' [-Wunused-variable]
    5 | int* pos = x;
      |    ^~~
warning-bad.cpp:4:25: warning: unused parameter 'y' [-Wunused-parameter]
```

```
    4 | int compute(int* x, int y) {
      | ~~~~^
warning-bad.cpp: In function 'int main()':
warning-bad.cpp:13:9: warning: unused variable 'res' [-Wunused-variable]
    13 | int res = compute(x, 0); ## 2
       | ^~~
warning-bad.cpp:14:5: warning: label 'std' defined but not used [-Wunusedlabel]
    14 | std:cout << "hello, world\n";
       | ^~~
warning-bad.cpp:13:22: warning: 'x' is used uninitialized [-Wuninitialized]
    13 | int res = compute(x, 0); ## 3
       | ~~~~~~~^~~~~
```

注释 1：编译器会抱怨错误的用法

注释 2：编译器会问为什么变量没有被使用；是不是少了什么？

注释 3：编译器知道未初始化的使用将会是一个问题

这表明了一些需要认真倾听的事情。

两个警告选项可能有不同的名称，具体取决于所使用的工具。它们直接适用于 g++ 并添加到命令行；MSVC 和其他编译器使用不同的选项和方法来启用它们。选择尽可能高的警告级别，以暴露编译器可以检测到的尽可能多的错误。

### 分析

看似无害的警告可能表明存在几个问题。首先，未使用的参数或变量可能是重构代码以简化代码库的证据。如果是这样，每个函数调用都必须提供无用参数，从而混淆其含义并增加复杂性。其次，未初始化的变量几乎总是需要修复。有人可能会认为这些应该是错误。第三，开发人员可能需要进一步了解代码机制，并且可能犯了一个错误。第四，总是有无意的输入错误。还有许多其他原因，发现的警告将取决于代码库和开发人员的成熟度。最佳做法是在编译时启用尽可能多的警告。

**小故事**

我参与过一个项目，有 21,283 个编译器警告，涵盖 48 个类别。其中一个类别（未使用的变量）出现了 5,176 次。我开始系统地减少这些警告，每次解决一个特定类别。5 个月后，我被调到一个新项目，但这个项目未完成——我怀疑这与我付出的努力有什么关系！但谁知道呢？也许代码不希望得到改进吧。

### 解决

打开最高级别的警告可能会产生很多问题，请查阅编译器的文档以了解如何实现。可以通过互联网搜索来快速找到答案，每个编译器都有自己的一组附加警告。

每个警告都应视为一个需要解决的问题。在出现拼写错误的情况下，纠正程序文本并继续前进通常很容易。考虑未使用的变量，或参数是否不再需要或重构期间需要清理步骤。当发现未初始化的变量时，立即修复！不修复警告是积累技术债务的快速步骤，技术债务可能会变得如此之大，以至于人们想要避免面对负载。避免这种心理障碍并修复警告。对于真正的挑战，找出如何设置编译器设置（如果可用）以将警告转换为错误。这可以防止构建带有警告的代码，并激励开发人员清理所有检测到的问题。代码干净后执行此操作，可能是防止新错误弄乱代码的最佳方法。对于不太

严重的方法，一次启用一个额外的警告，并逐步清理。

如果有其他静态代码分析工具可用，请考虑使用它们进一步分析代码库，以查找编译器未检测到的错误。C++ 的一个不错的选择是开源 Cppcheck 工具（https://cppcheck.sourceforge.io/）。也有几种商业选择，您的公司可能已经有一个或多个可用。如果没有，请申请购买一个用于开发。

清单 *12.20* 这个简单的程序完成了清理

```
int compute(int* x, int y) {
  int* pos = x;
  if (*x > 0)
    return *x + y;
  return *pos;
}
int main() {
  int n = 42;
  int* x = &n;
  int res = compute(x, 0);
  std::cout << res << " hello, world\n";
  return 0;
}
```

启用警告的情况下，编译此代码不会产生任何错误。这种情况并不能证明程序是正确的，但与清单 12.19 相比，消除了潜在和实际的问题。

**建议**

- 开启团队在项目中可以处理的最高级别的警告；如果级别太低，错误就会忽略，而如果级别太高，可能会出现警报疲劳（淹没在过多的冗长信息中！）。
- 切实可行的情况下，不要忽略代码中的错误；越早修复错误，其成本就越低——管理层必须决定切实可行的水平，这总是与开发人员的定义不同。
- 外部库或项目代码中的警告，可能需要仅针对该代码关闭特定警告；考虑将这些文件包装在包含文件中，并禁用特定警告。
- 如果可能的话，使用静态代码分析工具来发现错误；有些免费，很多收费，但贵有贵的价值。