TURING 图灵原创

C++实战

核心技术与最佳实践

吴咏炜 ○ 著

人民邮电出版社
北 京

图书在版编目（CIP）数据

C++实战 ： 核心技术与最佳实践 / 吴咏炜著.

北京 ： 人民邮电出版社，2024. -- (图灵原创).

ISBN 978-7-115-65769-5

Ⅰ. TP312.8

中国国家版本馆 CIP 数据核字第 2024XH1337 号

内 容 提 要

这是一本面向实战的现代 C++ 指南，由作者结合 30 余年 C++ 编程经验倾力打造。书中聚焦开发者日常高频使用的语言特性，重点讲解惯用法（而非罗列语言里的琐碎细节），展示代码示例及其技术原理，旨在帮助大家又快又好地使用 C++。

作者精选了对象生存期与 RAII、移动语义、标准模板库（STL）、视图、智能指针、错误处理、并发与异步编程等核心主题，深入浅出地剖析语言特性，并针对实际开发中的常见问题提供解决方案。

本书面向 C/C++ 程序员（特别是遇到困难、希望深入理解并优化 C++ 开发的读者），以及其他需要提升 C++ 编程能力的开发者。

◆ 著　　吴咏炜
责任编辑　刘美英
责任印制　胡　南

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　https://www.ptpress.com.cn
北京　　印刷

◆ 开本：800×1000 1/16　　彩插：2
印张：23　　2024 年 12 月第 1 版
字数：445 千字　　2024 年 12 月北京第 1 次印刷

定价：99.80 元

读者服务热线：(010)84084456-6009 印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东市监广登字 20170147 号

目　录

推荐序

欣闻吴咏炜老师的新书即将出版。自 C++11 以来，C++ 通过引入多种抽象机制不断演进而步入“现代”之列。秉承“零开销抽象”的原则，在追求“极致性能”和“使用抽象管理复杂性”的双重目标下，同时支持面向过程、面向对象、泛型编程以及函数式编程等多种编程范式，现代 C++ 构建了全新而又博大的语言系统。

学习和掌握现代 C++ 这样一门博大精深的编程语言，是非常有挑战的。如果从语言特性集合入手，很容易陷入 C++ 纷繁芜杂的特性字典中。我一直认为，要真正理解和掌握现代 C++，不能仅仅停留在“语言特性”的层面，更要深入到“为何设计”与“如何高效使用”的层面。C++ 语言的每一个特性都是为解决某个具体问题和应对特定场景而发展演进的，如果不理解其背景和设计理念，不理解实战中的应用场景，很容易导致误用或滥用。这本书从“实战”场景出发，不仅讲授基础原理，而且展示在实战场景中的核心原则和最佳实践。这种“实战派”风格的 C++ 教本，是技术修炼过程中不可多得的。

我经常讲，一本书选择“讲什么和不讲什么”很重要，也是作者的功夫所在。这本书并没有长篇大论地罗列所有语言特性，而是在有限的篇幅里简明扼要地抓住现代 C++ 的“主线”，选择那些“重要又常用”的语言特性，深入这些特性的“来龙去脉”，确保“好钢用在刀刃上”。同时，语言特性讲解的次序和脉络也很重要。这本书从 C++ 最核心的基础机制讲起，内容包括对象生存期和 RAII、值类别和移动语义、模板等。了解现代 C++ 的朋友都知道，这些都是现代 C++ 的“基石”。从这里入手，才能循序渐进、渐入佳境。

选择一本图书，就是选择一位老师，因此，图书的作者非常重要。我和这本书的作者吴咏炜相识许久，也共事多年。吴老师既是 C++ 社区备受赞叹的高手，也是在合作伙伴中广受尊重的顾问，还是 C++ 及系统软件技术大会（CPP-Summit）常年的专家讲师和专题出品人。他对 C++ 孜孜以求、精益求精的态度一直为众多同好所称道。我很高兴见证吴老师将他多年在 C++ 领域的宝贵积累整理成书。

阅读这本书，恰似跟高手切磋、向良师问道，我相信大家会有如沐春风之感。我向每一位渴望掌握现代 C++的读者推荐这本书。

李建忠
C++ 及系统软件技术大会主席
ISO 国际 C++ 标准委员会委员

前　言

从 Bjarne Stroustrup[①] 博士把他的“带类的 C”重命名为“C++”开始算，C++ 已经有超过 40 年的历史了。这些年来，虽然也有不少起起落落，但 C++ 一直是一种非常重要的编程语言，是计算机世界里很多基础设施背后的基石。根据 TIOBE 网站，C++ 自 1987 年来的最差排名是第 5 名，并多次占据榜眼和探花之位。在多次起起伏伏之后，C++ 在 2023 年又回升到了第 3 名的位置，超过了 Java，并赢得 2022 年“年度编程语言”（Programming Language of the Year）的称号，展示了它长盛不衰的生命力[②]。

已有好些年，我每年在 C++ 大会上进行演讲，介绍 C++ 方面的知识。在 2019 年，我开始在“极客时间”上开设专栏[③]，专门介绍现代 C++[④] 的一些知识点。在随后的几年里，我也做了不少 C++ 方面的培训和技术咨询工作。于是，在写了几十年的 C++ 代码之后，我终于觉得，可以把自己对 C++ 的认知写成书，让后来者可以少走一些弯路。

本书特色与目标读者

当然，C++ 的书籍已经有很多了。本书的特点是什么？或者说，什么样的 C++ 学习者应该学习本书？

简单来说，本书偏向实战，从代码和使用场景的角度来讲解 C++ 特性，而非试图去对所有 C++ 的特性进行系统描述。**我认为编程语言的学习方式应该更接近外语，而不是数学。**我不会像 cppreference.com[⑤] 一样描述所有的语法细节，而是根据我在软件项目获得的

① 英文发音一般是 /ˈbjɑːk(r)nə ˈstraʊstrʌp/（对于包括我在内的大多数人，丹麦语发音太难了）。注意“本贾尼”绝对是错误的音译。

② 在 2024 年 6 月，C++ 又超过了 C，成为第 2 名。但我觉得这一成绩主要是因为 C 的热度下降，反而意义没那么重大了。

③ 链接是 https://time.geekbang.org/column/256。该专栏始于 2019 年 11 月，迄今为止（2024 年 9 月）已有三万多人学习。

④ 即自 C++11 标准以来的“新”C++ 语言，代码风格可以跟 C++98 有相当大的不同。

⑤ 英文版网站的内容最全，主页是 https://en.cppreference.com/。但中文版网站的内容也相当全面，主页是 https://zh.cppreference.com/。在语法细节方面，本书推荐读者自行参考该网站中的内容，而不会重复地摘抄其中的知识点。

经验，来阐释某种特性存在的原因和适用的场景。当读者了解了我讲解的概念、方法之后，要进一步了解语法上的细节，可自行查阅网站——全面、详尽不是我的意图。

C++ 已经大到不需要学习“完整”的语言，正如我们学英语不需要把整本词典背下来一样。因此，本书是从一个实践者的角度出发，挑出了我认为最值得学习的基本方面：

- 读本书所需的预备知识（复习和快速检查）
- 对象生存期和 RAII（C++ 最重要的特性）
- 移动语义（现代 C++ 最重要的新特性，全面影响代码组织）
- 模板基础（现代 C++ 里到处是模板，即使你不会写，也得会用）
- 标准模板库（STL）和相关知识（效率利器）
- 视图（略新，也有一定使用风险，但至少比指针要好用）
- 智能指针（全面替代有所有权的裸指针和手工 `new/delete`）
- 现代 C++ 的其他一些重要改进（初学者友好的特性和重要的新库）
- 不同的错误处理方式（契约、异常的概念，及不使用异常的一些方法）
- 如何传递对象（引用传参，还是值传参？）
- 并发编程（应对多核世界的挑战）
- 异步编程（一种不同的思维方式，但可以更好地利用资源）
- C++ 程序员应当掌握的一些基础工具（最后稍稍讨论一下工程问题）

因此，本书的目标受众是有一定编程经验的程序员。我期望你有基本的 C 和 C++ 知识。我也期望你在使用 C++ 时遇到过问题、踩过坑，并希望本书能提供给你一些解决方案。

大部分的读者应该在通过其他渠道学过、用过 C++ 之后再来阅读本书；而如果你有扎实的 C 编程基础，但从来没用过 C++，也可以尝试一下直接上手本书。在第 1 章，我描述了阅读本书需要的基础知识，也可以作为一个快速的知识复习吧。

本书内容按循序渐进、由浅入深的方式组织，因此推荐的阅读方式是从头到尾、按顺序阅读。在初次阅读时，你可以考虑跳过所有的脚注、插文、交叉引用，以及节标题尾部带上标星号（*）表示难度超出本章其他部分的内容。对于 C++ 老手，也许你会觉得需要跳过一些已经了解的章节，但此时你也有可能会漏掉一些有用的说明信息。因此，我建议对已经了解的内容也快速浏览一下。需要快速查找信息时，本书尾部的索引应该可以给你帮助。

本书是我计划写作的 C++ 系列图书的第一本。这本书是关于“使用 C++”的，是基础，适合任何使用 C++ 进行开发的程序员。这本书的目的是告诉读者使用现代 C++ 的“正确”方式——换句话说，如何用好标准 C++ 提供的“制式武器”，高效地进行开发并避免一

些常见的陷阱。

至少我还会写作第二本，主题是“扩展 C++”，算是高级课题，适合于使用 C++ 搭建项目的框架和公用工具的程序员和架构师。在这一本书中，我会讲解 C++ 的高级特性，通过讲解例子介绍 C++ 标准库和类似于 C++ 标准库的工具该如何搭建——换句话说，我们要在用好“制式武器”的基础之上，研究它们是如何打造的，并进一步打造适合特定项目的称手兵刃。

引用和拓展阅读

有一个挺有意思的事情是，该怎么处理引用和拓展阅读。以前我读书的时候，觉得部分图书列上好多页的参考资料真是无聊：作者真的参考了这么多资料吗？读者真会去读这些参考资料吗？在真正开始技术写作之后，我知道了，对于前者，回答多半确实是“是”。对于后者，就比较复杂了。对于学术论文，参考资料是必要的，既说明了某概念/思想/理论的来源，也为别人进一步查阅信息提供了方向。而对于技术书籍的读者而言，两者的需要都要弱上不少，特别是，参考资料太多对读者相当不友好（参考资料是英文的话，那更是雪上加霜）。无论如何，书的内容需要较为完整，而不是让读者自己去阅读各种参考资料。因此，我目前的处理方式是：

- 在书中尽量完整描述技术问题，而不需要读者去查阅额外的资料。其他给出的资料和链接，只作为拓展阅读使用。
- 对于 cppreference.com、C++ 核心指南（C++ Core Guidelines）[①] 和维基百科中的内容，只给出来源和词条名称，不直接给链接。读者在需要时自己搜索多半比手工输入链接更快。cppreference.com 的给出方式是 [CppReference: *标题名称*][②]；C++ 核心指南的给出方式是 [Guidelines: *条目编号*]；维基百科的给出方式是 [Wikipedia: *条目名称*]。
- 对于很容易通过标题搜索到的资料，只给出标题。同样，搜索是比键入链接更快、更有效的方式。
- 对于 C++ 的标准文档和标准提案，以 [N*nnnn*] 或 [P*nnnn*] 这样的编号形式给出。读者需要了解，这类文档可以通过在编号前拼接链接“http://wg21.link/”[③] 来获得。比

① https://github.com/isocpp/CppCoreGuidelines/

② 一般我给出的是英文标题。但是，和维基百科不同，cppreference.com 上通常每个英文条目都有对应的中文条目，且相比英文条目来说，内容并没有什么缺失。请读者根据自己的需要查阅英文或中文的条目内容。

③ C++ 标准委员会在 ISO 组织里被称作 WG21，因为它的完整名称是 ISO/IEC JTC1 (Joint Technical Committee 1) / SC22 (Subcommittee 22) / WG21 (Working Group 21)。

如，C++23 的标准草案是 [N4950]，这意味着我们可以通过链接 `http://wg21.link/n4950` 来获取该文档。顺便说一句，这也是本书主要参考使用的 C++ 标准文档。

- 对于其他有链接的参考资料，一般直接在提到时（在正文或脚注中）给出链接。
- 我在书末给出了一份推荐阅读材料的清单，供读者进一步学习参考。需要说明的是，本书不一定参考其中的内容。

如果原始资料是英文，在正文里引用时，我会要么使用官方中文版（如果有中文版且没有额外说明的话），要么自己翻译成中文。

传统上对于引用资料要求稳定，因此对于会变化的内容（如网页）还经常会限定访问的时间。但从实用的角度看，对大部分学习者，最新的资料最有用——对上面几种特殊引用尤其如此。自然，我也不会限定这些内容的访问时间。在极少数场合，链接可能会过期（这时候，就需要使用 Internet Archive 这样的手段了）。但大部分情况，其中的内容只是被更新。而习惯这些变化，甚至拥抱变化，是任何一个程序员——或者学习者——都需要去做的事。

中文术语

编程术语基本是从英文翻译过来的。有时候，一个英文术语有多种译法，该采用哪种呢？本书的术语选用标准是贴近英文原文、形象、易懂和可组合。有时候你可能觉得某种译法不常见，但这是我的选择，而不是发明。原创并非我的追求：虽然我认为 vector 翻译成“多量”更为合适，但我并没有在本书中这样翻译，因为我觉得此路不通（该词我尽量不译，实在需要也只能译为“向量”）。不过，在可能的范围里（至少包含了常见 C++ 书籍、网络文章和 CppReference），我会选择最符合我的标准的中文术语。比如，我选择使用“转型”（cast），而不是“强制类型转换”，因为前者贴近原文（且考虑到 cast 是单音节词）、形象，且可组合——named cast 如果在中文里说成“有名字的强制类型转换”，那真会令人发疯。我选择使用“名空间”（namespace），而非“命名空间”，因为后者长而没有道理，并不贴近原文——英文里并没有用 *naming* space 作为编程术语。但我也不是一味求短。我选择使用“部分特化”（partial specialization），而不是“偏特化”，因为前者更贴近用法（部分指定类型）、容易理解。我选择啰唆地说“默认提供”（defaulted），而不是 CppReference 中文版里使用的“预置”，因为前者贴近原文，且不需要解释就可以理解。诸如此类。此外，在某一术语存在常见的其他中文说法时，我一般也会在第一次使用时加以说明。

一些约定

本书正文里一般使用等宽字体来表示代码。这应当是大家熟知的惯例了，可以适当强调，也易于对齐代码后面的注释（但在标题、图题等地方，出于排版、美观等原因，不使用等宽字体）。对于一些较长的名字，本书可能会在下划线（“_”）之后插入连字符（“-”）来进行换行，注意在这种情况下连字符不是名字的一部分。

我们讨论的内容大部分是标准 C++，因此，为简洁起见，本书正文里的代码示例中一般不写出标准名空间“std”[①]（但部分可能有歧义或需要强调的地方仍会使用）。在代码示例有所省略的地方，我采用 Scott Meyers 在 *Effective Modern C++* 中引入的惯例，使用“…”表示，以便跟标准 C++ 中的“...”（在 C 的变参数函数和 C++ 的变参模板中需要用到）有较为明显的区别。

插文说明

在正文中会出现一些插文，以目前这样的灰底文字形式出现。这些插文不是正文的“正式”部分，但会为正文的讨论提供有益的补充。略过插文的内容通常不影响对正文的理解，但我真心希望读者从插文中也能得到知识或者愉悦——兴许兼而有之。

为了尽可能使书中的内容可以在实际的软件项目中使用，我在大部分时间会使用不超过 C++17 标准的语言特性。但从学习的角度看，仍建议读者尽可能使用高版本的编译器，以便试验 C++20 的功能，并在代码有问题时有望得到更友好的错误和告警信息。目前推荐使用的编译器是：

- GCC 13 或更新版本
- Clang 17 或更新版本
- MSVC（Visual Studio 2022 或更新版本）

书中的代码跟正文一样重要。除非我展示的代码你已经非常熟悉，否则请务必像阅读正文一样仔细阅读代码——就像阅读学术论文不能忽略其中的公式一样。当然，正文里的代码由于通常省略了头文件包含和名空间，一般不能直接编译。如果你需要立即可以编译的代码，很多示例在 GitHub 上有完整版本，其链接是：

```
https://github.com/adah1972/cpp_book1
```

① 发音为 /stʊd/ 或 /ˌestiːˈdiː/。

致　　谢

首先，我想感谢 C++ 之父 Bjarne Stroustrup，他以其睿智为我们带来了 C++ 这门虽有缺陷却又非常强大的语言——如同世间的每一个英雄。没有 C++，本书完全没有存在的意义。

其次，我要感谢 C++ 技术的布道者们，尤其是 Herb Sutter、Scott Meyers、Andrei Alexandrescu 和侯捷。他们的书籍、文章、演说在我学习、成长时一路相随。

我要感谢 Boolan，她在中国筹办了全球 C++ 及系统软件技术大会，让我有机会能跟 Bjarne Stroustrup、Stan Lippmann、Andrei Alexandrescu 等世界级大师面对面交流，并从一名听众成长为分享嘉宾，逐步走到了今天，乃至可以为 C++ 社区做一点微薄的贡献。

感谢极客时间，让我走上了技术分享这条道路。也是从那时起，我逐渐萌生了从事独立咨询、培训工作及写书的想法。同时，也要感谢我的极客时间专栏的学员、我的咨询客户、其他线上/线下培训的学员、一些技术讨论群的群友，以及在知乎上提出问题的知友——问题和解答都需要来自实践，没有他们的输入，我也无法深入了解 C++ 学习者可能面对的各种困惑。

感谢编辑英子，她很早就来“劝诱”我写书，并直接护持了本书的诞生。嗯，抓本书文字里的“虫子”也是她的工作，希望没有让她太痛苦。

本书的初稿经过了一些朋友和 C++ 爱好者的审读，他们是：倪文卿（IceBear）、罗能（netcan）、冯畅、何荣华、李冠锋、琳哒 6、张德龙、杨凡、吴长盛、杨文波和杨红尘。我收到了各位宝贵的反馈意见，并修正了书中很多技术和文字问题。在此，谨向各位表示深深的谢意。特别鸣谢倪文卿，他给出了最详尽的反馈意见，还花了不少时间跟我讨论。

感谢我的母亲在我上初中时就让我参加计算机学习班，让我走上软件开发的道路。感谢我的妻子的付出和陪伴，让我没有后顾之忧。感谢我的孩子，在让我欢喜让我忧的同时，让我更多地理解了生命的意义。

书里几乎不可避免仍会有错误残留，这当然都是我的责任。如果你在阅读本书过程中发现了问题，请给我发邮件：wuyongwei@gmail.com。本书的 GitHub 仓库里届时会记录勘误信息[①]。

---

① 另见图灵社区本书页面：https://www.ituring.com.cn/book/3410。

绪　论

在进入本书正文之前，我想讨论两个问题：

- 我们为什么要学习 C++？
- 我们该如何学习 C++？

为什么要学习 C++

我们学习 C++，通常是因为要用到 C++。用一门语言通常有两种理由。在不同的场合两种理由的占比会不同，但多多少少肯定都存在：

1. 当前的项目里已经用到了 C++；
2. 出于性能或相关的其他考虑，我们需要使用 C++。

换句话说，**历史原因**，以及**技术原因**。

历史原因非常容易理解，如果已有代码是用 C++ 编写的，换其他语言总会增加很多麻烦——不管是重写代码，还是拼接不同语言组成的模块。单一语言比较简单。

技术原因就复杂些了。目前甚至存在某些观点，认为新项目应该不再使用 C++ 这样的“不安全”语言。但这种观点在我看来相当片面。它夸大了 C++ 的不安全性，低估了 C++ 的灵活性和强大功能。事实上，使用现代 C++ 的惯用法，犯内存方面的错误已经相当不容易，而 C++ 在某些领域的优势（如模板和编译期编程），在主流编程语言里尚无竞争对手。

在比较主流的编程语言里，C、C++ 和 Rust 是“唯三”的适合系统级编程的高性能语言。我作为一个从业近三十年的 C 和 C++ 开发者，朋友李沫南作为一个 Rust 的拥趸，经常进行对嘲。常规对话如下：

——哦，C++ 没有这个检查吗？垃圾啊。

——咦，Rust 写这个这么费劲？失败啊。

这虽然一半是玩笑，却也有几分真理。Rust 以安全为核心理念，比 C++ 在类型系统上更为复杂。虽然 Rust 不像 C++ 一样有 C 兼容性这个历史包袱，但仍不能算好写。而 C++ 则

在庞杂之中给了开发者相当大的自由度：它给了用户强有力的抽象，但又允许用户深入底层细节，控制内存布局之类的细节；但它也缺乏 Rust 那样的安全检查。对于熟稔 C++ 的人来说，C++ 的强大和灵活性是任何其他语言无法比拟的。内存安全性是一柄双刃剑：虽然我们不能说内存安全的语言不好（当然是好的），但它有自己的代价。目前看起来，想要内存安全，要么使用垃圾收集（garbage collection）[①]，要么使用借用检查（borrow checker），两者似乎必居其一。目前 C++ 没有使用其中任何一种方法，而是要求开发者承担起更多的责任，这是语言的选择，也是演化的结果。此外，C++ 的“后继”语言，如 Cpp2 [②] 和 Carbon [③]，都把跟 C++ 的互操作性当作基本要求——要谈 C++ 是否会“过时”，现在还为时尚早。

C++ 的特点

Bjarne 老爷子认为，C++ 最主要的特点在于对以下两方面的持续关注：

- 语言构件到硬件功能的直接映射
- 零开销抽象

跟 C 语言一样，C++ 提供非常底层的数据操作能力，为开发者提供了灵活性。跟“高级”语言一样，C++ 提供了强大的抽象能力（可以说超越了大部分语言）。但跟大部分高级语言不同的是，C++ 有点太灵活了，也因此容易被误用。后来者在提供抽象能力的同时，也往往施加了很多额外限制，使得语言在保留表达能力的前提下，能够更容易被使用和更不容易被误用。C++ 作为一名历史悠久的“先行者”，在这方面就比较吃亏了——向后兼容性保证导致某些不安全的用法不那么容易从语言中被消除。不过，相比 C，C++ 还是安全得多。在语言诞生的初期就是如此，现在就更不用说了。

C++ 的类型安全性

C++ 的类型系统比 C 更加严格。虽然一直有 C++ 是 C 的超集的说法，但严格说来，这个说法从来就没成立过。最近（2023 年）我碰到过一个程序崩溃的案例，简化来讲，就是开发者使用了一个 `char` 的二维数组（`char names[MAX_NAMES][MAX_NAME_LEN]`），然后把它传给了一个接收 `char**` 参数的函数……这样的代码当然是错的，但 C 编译器只是

① 本书跟随一般用法，用“垃圾收集”指代跟踪收集器这样的垃圾回收机制，不包括引用计数收集器。

② https://github.com/hsutter/cppfront

③ https://github.com/carbon-language/carbon-lang

给了个告警，编译还是没有失败。如果这是 C++ 代码的话，编译器就会直接报告错误，拒绝编译通过了。[①]

C++ 凭借它的“零开销抽象”，可以让“使用者”非常安全地使用这门语言，同时还可以让“定制者”根据自己的需求来写出更贴近使用场景的库，进一步方便“使用者”。

更具体地说，C++有以下几方面的优点：

1. **和 C 兼容**。这在以前是个很大的优势，这意味着 C 程序员可以自然而然往 C++ 的方向转，同时现有的代码仍然能够在不需要任何改动或只需少量改动的情况下，即可在 C++ 里直接编译使用。这点是 C++ 的大部分其他竞争者所没有的。今天，C 程序员比以前略少了点，优势不那么大了，但毕竟现有的 C 项目的数量仍然是惊人的。在 TIOBE 上，C 也仍然处于第 3 名的位置（2024 年中）。
2. **接近硬件**。这是和 C 共享的特点。要跟硬件密切交互，完成各种稀奇古怪的底层功能，没有比 C 和 C++ 更合适的语言了。像 CUDA 这样在人工智能年代火热的编程接口，主要编程语言仍然只是 C 和 C++，其他的语言支持多多少少有些限制。
3. **零开销抽象**。声称这一点的，也就是 C++ 和 Rust 两种语言了，也只有 Rust 可以算是 C++ 的真正竞争者。如果你不使用某种抽象，这种抽象就不会给你带来性能损失。这既给了开发者强大的武器，又让开发者在性能方面有最大的控制能力。大家熟悉的 LLVM 是用 C++ 写的，而 GCC 也早就从纯 C 转向了混合使用 C 和 C++。
4. **编译期编程**。我不知道还有哪种语言有像 C++ 一样强大的编译期编程功能，即在编译时而不是运行时完成某种运算。这不仅仅是理论，我可以定义一个用户定义字面量，写下 `"192.168.1.0"_ipv4` 就能让编译器直接为我生成 IPv4 地址的常量，字节内容是 `0xC0A80100`——完全没有运行期的开销。
5. **高成熟度**。Rust 可以去芜存菁，但它的工具链没有 C++ 成熟。不是所有使用 C++ 的地方都可以换成 Rust，虽然它们的理论应用场合较为一致。成熟的 C++ 代码，不管是开源还是闭源，数量都是惊人的。在很多地方，对 C 或 C++ 已有完整的支持，而对 Rust 要么不支持，要么存在较多的问题。
6. **高稳定性**。绝大多数完全符合 C++98 标准的代码，也仍然是合法的 C++20 代码。这种稳定性（不同版本之间兼容）有它的代价，但好处也是惊人的。

---

① 这个 `names` 在传参时会退化成为 `char (*)[MAX_NAMES_LEN]`（参见 1.1.9 节），但 C 语言将传递不兼容的指针看作未定义行为，而没有规定为错误。GCC 要到 14.0 才默认把这种情况当作编译错误（除非你指定 C 标准为 C89，此时只告警不报错；使用 C99 或更新标准时，GCC 都会把不兼容的指针类型当作错误）。

我个人非常欣赏 C++ 综合这些优点带来的高度灵活性。尤其是它的模板功能，可以认为，其强大程度在主流编程语言中是独一无二的。在本系列的第二本书（主题是“扩展 C++”）中，将有大量的篇幅讨论模板和模板带来的无限可能性。某种意义上，C++ 的一些特性具有实验性。也许不是每个开发人员都需要触碰这些特性，但是，如果你因为性能或其他原因需要这些特性，就会觉得它们非常有用——一些让很多人觉得没用的 C++ 特性，实际上，可能真的是因为只有少数人需要直接使用它们。但是，很可能发生的一件事是，如果真的去掉这些“不常用”的特性，那另外一些常用的特性也就没法使用了。

当然，“优点”的代价就是 C++ 非常复杂，成为一个合格的 C++ 程序员需要付出极大的努力。不过，抽象机制也是可以掩盖很大一部分复杂性的；C++ 新特性虽然不能从根本上降低语言的复杂性，但至少可以让初学者上手更快。事实上，现代 C++ 的很多特性之所以存在，很大程度上是为了让使用者更加方便：比如 auto（4.2.3 节），比如类模板参数推导（4.6.1 节），比如用户定义字面量（12.3.1 节），等等。

C++ 的复杂内存模型

写到这里，我觉得一定得引用一下 Bjarne 老爷子提到的一件轶事。他在 HOPL4 论文《在纷繁多变的世界里茁壮成长：C++ 2006—2020》的 4.1.1 节讨论内存模型的时候提到：

“我们知道 Java 有一个很好的内存模型，并曾希望采用它。令我感到好笑的是，来自英特尔和 IBM 的代表坚定地否决了这一想法，他们指出，如果在 C++ 中采用 Java 的内存模型，那么我们将使所有 Java 虚拟机的速度减慢至少一半。因此，为了保持 Java 的性能，我们不得不为 C++ 采用一个复杂得多的模型。可以想见而且讽刺的是，C++ 此后因为有一个比 Java 更复杂的内存模型而受到批评。”

要在不牺牲灵活性的前提下，正确地用好 C++，我们需要语法之外的使用指南。C++ 核心指南就是这样的一份重要资料，我在本书中也常常会引用。C++ 的工具也在一直进化，很多不违反语法的编程错误，现在编译器已经可以直接发现[①]，另外很多问题也可以使用静态扫描工具检查出来，如开源的 Clang-Tidy（参见 18.2.2 节）。

C++ 不是一门停滞不前的语言，它还在不断地向前发展。有人说 C++ 过于学术化——这话肯定不对，因为 C++ 一直是一门注重实际的工程化语言。但是，C++ 相比其他语言，在语言里能玩的“花样”确实多一点——C++ 确实有玩花样的能力。要说跟学术界靠拢，

① 有些需要开高告警级别，如 GCC/Clang 的 -Wall -Wextra，及 MSVC 的 /W3。

那 C++ 没法跟另外一些编程语言比，毕竟在 C++ 里实用是第一性的。

任何一门语言里都有要做和不要做的事情（do's and don'ts）。对于 C++ 而言，C++ 社区积累下来的最佳实践就是要做的事情，它们是经验积累，也是社区的共同财富；而各种编码规范里禁止（并且静态扫描常常会告警）的写法，则是不要做的事情，它们往往代表着一种历史包袱——出于向后兼容性的原因，C++ 不能从头再来，修复语言中所有已知的问题，把“不好”的写法宣布为非法。而新来者则有着更大的自由，从 C++ 里汲取历史教训，修正已有的问题。不过，问题容易修正，财富则不那么容易继承……

跟任何语言一样，C++ 语言的发展受到两种动力的影响：一是要更简单、更自然、更符合直觉；二是要符合规则。相比人类的语言，计算机语言更加不能容忍不满足规则，因为编译器只能靠规则，不能靠直觉。此外，直觉并不可靠，专家和初学者的直觉也并不相同。对于专家，C++ 里的切片（参见第 43 页的插文）行为也许挺自然、合理；但对于初学者，恐怕切片就是一个黑暗的陷阱了。

这样，总体的后果就是，C++ 里面有很多符合规则、但不那么满足直觉的东西。切片是一个例子，最令人烦恼的解析（参见 4.6.4 节）和很多未定义行为也是如此。

有最好的语言吗?

从单一维度去看，语言总是存在问题的。下面我试着从自己关心的一些维度去比较一下 C++、C、Rust、Java 和 Python，使用 1—5 的数字打分（越高越理想），如表 0-1 所示。

表 0−1：几种编程语言的多维度比较

	C++	C	Rust	Java	Python
学习快速	2	3	1	4	5
可读/可维护	3	3	2	4	5
抽象能力	5	2	4	3	3
安全性	3	2	5	4	5
静态检查能力	5	1	4	3	1
适用平台广泛	4	5	2	3	3
运行性能	5	5	5	3	1
编译速度	2	4	1	3	—
向后兼容性	4	5	2	4	4
第三方库数量	4	4	2	4	5
第三方库易集成	2	3	4	4	5

这当然是很不严肃的个人评估，但也应该做一下简单说明：

1. 虽然 C 和 C++ 在“可读/可维护”上的得分相同，但原因大不相同。C 是代码直白，容易判断背后发生的事情；缺点是简单的事情可能需要写很多代码。C++ 则相反，代码可以看似简单到接近 Python，但背后的逻辑复杂且没有对开发者完美隐藏（抽象泄漏[①]问题较严重），一旦遇到问题，有时寻找问题背后的根源较为困难。
2. Rust 的“安全性”是不使用 `unsafe` 的情况（否则至多打 4 分）。
3. “静态检查能力”不含跟安全相关的检查（已被“安全性”覆盖）。

这个比较的目的并不是评判哪种语言好，哪种语言差，而是展示编程语言特点上的复杂性。对于很多问题，Python 也许是最好的语言（我打 5 分最多的就是它了），但它的运行性能低到在很多场景下完全不会考虑它。如果既需要高性能又需要高度安全，那 Rust 是个不错的选择，但兼顾这两者也使得 Rust 难以上手、可读性下降及编译缓慢。而如果你的使用场景要求高性能、高度抽象、代码稳定兼容，C++ 可能就是最好的选择。

如何学习 C++

不过，我不鼓励你去记住 C++ 里所有的语法规则。学习用一门语言跟成为语言律师是不同的——一个不那么完美的比方是写小说不需要成为语法学家。不完美，是因为随着对语言理解、掌握、使用的深入，我们多多少少需要学会像编译器一样看问题，这时候，成为语言律师，或者说了解语言里的语法细节，也慢慢成为一种必要的事情。但无论如何，在很长的一段时间里你不需要成为语言律师。本书也不会教你成为语言律师。我会尽力避免大段地引用 C++ 标准或 CppReference 来讲述语言细节，而更着重于讲解为什么，并给出靠近实际应用的例子。

我的个人观点是，学好编程同时需要语言能力（你的外语成绩如何？）和推理能力（你的数学成绩如何？）[②]。推理能力估计大家都能理解，抽象思维是编程的基础；语言能力可能会有点出乎意料了。但细想一下，应该也不那么奇怪。代码既是写给计算机看的，也是写给人看的——这个人可能是你的同事，也可能是三个月后的你自己。如果每次读代码都

---

① 参见 [Wikipedia: Leaky abstraction]。

② 这句话写完后很久，我才惊讶地发现（谢谢杨文波提醒），著名计算机科学家、图灵奖得主 Edsger Dijkstra 在一篇文章（“"Why is software so expensive?" An explanation to the hardware designer”）中表达过非常相似的观点：“出色掌握母语是我选择有潜力的程序员的首要标准；对数学的良好品味紧随其后。”他的文章主题是软件和硬件设计思维上的区别，虽然写在四十多年前，也非常值得一读。

跟公式推导那样累，那这样的代码，不客气地说，是无法维护的。而阅读代码，跟读外语文章一样，你如果时时刻刻想着语法规则，想着如何分析句子，那效率是极其低下的，是还没有入门的表现。你需要做的事情，是把语法知识内化，把知识变成直觉和肌肉记忆，直到最后获得“语感”。

不管是人类的语言，还是计算机语言，其中的规则（语法）都是为内容服务的。我们需要规则，是因为规则解决了一些不确定性，而人们厌恶不确定性（当然，编译器更加厌恶不确定性）。但语法并非至高无上的东西。在人类语言中，习惯成自然，语法只是事后的描述，不规则之处数不胜数。编程语言要由计算机来解释，不能那么自由，但类似地，语法是一种手段，是为了解决问题而服务的。不同的语言，可以有不同的折中。有些语言追求语法简单，让编译器/解释器能够快速处理；有些语言追求简洁的表达能力，能够非常简短地表达目的；有些语言追求安全性，让程序员不容易写出有问题的代码……没有哪种答案全对或者全错。能在自己的特定领域解决问题，就是一种成功的语言。

那 C++ 属于哪种情况呢？我的回答是，它是一种同时注重抽象表达能力、代码性能和向后兼容性的编程语言。在以前两点作为目的的情况下，我们确实可以做出一些更好的设计，尤其是有了这么多年在编程语言上的经验和教训的情况下。但要加上第三点——向后兼容性——那就没有太多选择了。**跟英语这样的自然语言一样，C++ 是演化出来，而非设计出来的**（实际上，任何关注向后兼容性的系统都大抵如此，如 Unix 和 Linux）。在很多领域，C++ 的地位就跟英语一样——英语不是最完美的语言，但仍不可替代。

好，下面正文正式开始。

第 7 章　标准容器

容器是 C++ 里非常重要的概念，也是 C++ 标准库里最常用的组件之一。本章讨论标准库的各种容器，包括序列容器、关联容器和无序关联容器，以及与容器紧密相关的容器适配器。

7.1　标准模板库和容器

Alex Stepanov 的主要作品之一，C++ 标准模板库（STL），可以认为是 C++ 面世后最重要的标准库组件。在 1993 年向 C++ 标准委员会展示之后，它一举征服了委员会成员，随后顺利进入 ISO C++ 标准，并促使 C++ 程序员的思维方式朝着泛型编程（generic programming）的方向发展。

C++ 和 STL 可以说是互相成就。C++ 通过模板为 STL 提供了所需的实现机制，是实现标准模板库的最佳语言。STL 为 C++ 提供了一套高性能、高度抽象的标准库，同时还促进了 C++ 的模板和泛型机制的发展。

以下内容摘自 *Dr. Dobb's Journal* 对 Alex Stepanov 的访谈：[①]

> 在 1988 年，我搬到了惠普实验室……到了 1992 年，我回到了泛型库的开发……那时候 C++ 已经有了模板。我发现 Bjarne 在模板设计上做得非常出色。我之前在贝尔实验室参与过几次关于如何设计模板的讨论，并且与 Bjarne 有过激烈的争论，我认为他应该使 C++ 模板尽可能接近 Ada 泛型。我想我争论得太激烈了，他反而决定不那么做……Bjarne 设计了一种模板函数机制，在这种机制下，模板通过重载机制来隐式实例化。这种特别的技巧对我的工作至关重要，因为我发现它允许我做很多在 Ada 中无法实现的事情。
>
> ……
>
> 在 STL 被接受之前，有两个[模板]变化被修改后的 STL 所使用。一个是允许模板成员函数。STL 广泛使用模板成员函数，这样可以从一种类型的容器构造另一种类型的

① http://stepanovpapers.com/drdobbs-interview.html

> 容器……在 STL 中使用的第二个重要新特性是模板参数本身可以是模板，分配器在最初提出时就是这样实现的。
>
> ……
>
> Bjarne 提出了一个对模板的重要补充，称为“部分特化”，这会让很多算法和类变得更加高效，并解决了代码大小的问题。我与 Bjarne 一起在这个提案上工作，驱动它的需求就是让 STL 更加高效。
>
> ……
>
> 目前 C++ 是这种[泛型]编程风格的最佳载体。

STL 里的基本概念是：容器（container）、迭代器（iterator）、算法（algorithm）和函数对象（function object）。我们已经在第 6 章对函数对象进行了基本介绍，接下来我们就分别在本章、第 8 章和第 9 章依次介绍标准容器、迭代器和标准算法。

鉴于读者应该对很多容器已经相当熟悉了，我采取一种非正规的讲解方式，尽量不重复已有的参考资料，而是强调需要重点关注的部分。

此外，在把玩容器时，一个常见的需求是输出容器中所有元素的内容。有很多方法可以做到这一点。我在本章的一些范例中会使用一个自己写的头文件 `ostream_range.h`，在包含它之后我们就可以直接通过流输出运算符 `<<` 来输出容器的内容。该文件的实现稍复杂，本书中暂不进行讲解。但这不妨碍大家现在就开始使用它（本书的示例代码里有）。

7.2 序列容器

序列容器（sequence container）是在概念上按顺序存储元素的一组容器。

7.2.1 vector

`vector` 是最常用的序列容器，也是最常用的容器。本书之前的部分也已经出现过它的身影。它的名字来源于数学术语，直接翻译是“向量”的意思，但在实际应用中，我们把它当成动态数组更为合适[①]。它基本相当于 Java 的 `ArrayList` 和 Python 的 `list`。C++ 里更接近数学里向量概念的对象是 `valarray`（很少有人使用，本书也不介绍）。

和 `string` 相似，`vector` 的成员在内存里连续存放，同时 `begin` 和 `end` 成员函数指向的

① Alex Stepanov 在设计 STL 时起了这个名字。他在《数学与泛型编程：高效编程的奥秘》的 7.2 节里写道：“在 STL 中，*vector* 这个名字借鉴自早期的编程语言 Scheme 和 Common Lisp。不幸的是，这与数学中这个术语更古老的含义不一致……这种数据结构就应该叫 *array*。遗憾的是，如果你犯了错……结果可能会在很长一段时间里一直保留下来。”

位置也和 string 一样，大致如图 7-1 所示。

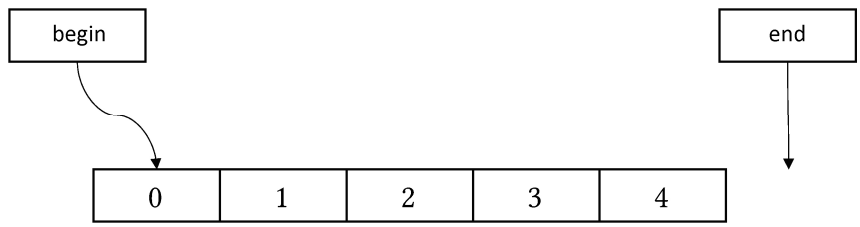

图 7-1：vector 的内存布局

注意，end() 不应该解引用。如果去解引用的话，结果是未定义行为（虽然通常不会导致程序崩溃[①]）。

vector<bool>

Alex Stepanov 和 Meng Lee 很早就引入了特化[②]，让 vector<bool> 里每个元素只占一比特，而非一字节。这个特化随后进入 C++98 标准，并一直沿用了下来。在今天看来，这有点画蛇添足了（当然，马后炮总是容易一点）。

这样的特殊容器很有用，在需要减少内存占用的时候你会发现非常方便。但它跟其他 vector 的行为有明显的不同之处，如 operator[] 和 *begin() 返回的不是 bool 的引用，而是个代理类（proxy）对象。在通用代码中，如果无意使用 vector<bool>，而是用 vector<T> 一不小心组合出了 vector<bool> 类型的话，就可能会导致各种意想不到的模板错误——因为写代码的人很可能会自然假设，如果需要修改元素就应当用 auto& 或类似的方式去获得一个元素的引用，而这在 vector<bool> 上无法工作。这种时候，有时候你会不得不做反向特化，在发现参数类型是 bool 的情况下努力不要生成 vector<bool>，而改用 vector<char>、vector<BoolObj> 之类的东西来替代。[③]

事实上，STL 实现里最早用的名字 bit_vector[④] 就挺好。

---

① 程序崩溃一般是由于访问非法地址所致的，而迭代器越界之后的地址通常仍为合法地址。至少在调试时，我们希望能尽早捕获这种情况，此时可以使用 Address Sanitizer（ASan）或启用标准库调试模式来达到这样的效果。参见第 18 章。

② 参见 STL 最早的文档：http://stepanovpapers.com/STL/DOC.PDF。

③ 另外，可参考 Herb Sutter 的讨论文章（http://www.gotw.ca/publications/mill09.htm）和 Howard Hinnant 的讨论文章（https://isocpp.org/blog/2012/11/on-vectorbool）。

④ 参见 SGI STL 的文档：https://www.boost.org/sgi/stl/bit_vector.html。

vector 跟 C 风格数组的主要区别在两点：

- 大小不在编译期决定，且可以动态增长和缩小
- 元素放在堆上而不是栈上

因为 vector 的元素放在堆上，它也自然可以受益于现代 C++ 的移动语义——移动 vector 开销很低，通常只是操作六个指针而已（见 3.3.1 节）。

下面的代码展示了 vector 的基本用法（vector.cpp）：

```
vector<int> v{1, 2, 3, 4};
v.push_back(5);
v.insert(v.begin(), 0);
for (size_t i = 0; i < v.size(); ++i) {
    cout << v[i] << ' ';  // 输出 0 1 2 3 4 5
}
cout << '\n';

int sum = 0;
for (auto it = v.begin(); it != v.end(); ++it) {
    sum += *it;
}
cout << sum << '\n';      // 输出 15
```

我们首先构造了一个内容为 {1, 2, 3, 4} 的 vector，然后在尾部追加一项 5，在开头插入一项 0。接下来，我们使用传统的下标方式来遍历，并输出其中的每一项。随即我们展示了 C++ 里通用的使用迭代器遍历的做法，对其中的内容进行累加。最后输出结果。

当一个容器存在 push_… 和 pop_… 成员函数时，说明容器对指定位置的删除和插入性能较高。vector 适合在尾部操作，这是它的内存布局决定的（它只支持 push_back 而不支持 push_front）。只有在尾部插入和删除时，其他元素才会不需要移动，除非内存空间不足，需要重新分配。

除了容器类的共同点，vector 允许下面的操作（不完全列表）：

- 可以使用方括号（下标运算符）来访问其成员（同 string）
- 可以使用 data 来获得指向其内容的裸指针（同 string）
- 可以使用 capacity 来获得当前分配的存储空间的大小，以元素数量计（同 string）
- 可以使用 reserve 来改变所需的存储空间的大小，成功后 capacity() 会改变（同 string）
- 可以使用 resize 来改变其大小，成功后 size() 会改变（同 string）

- 可以使用 pop_back 来删除最后一个元素（同 string）
- 可以使用 push_back 在尾部插入一个元素（同 string）
- 可以使用 insert 在指定位置前插入一个元素（同 string）
- 可以使用 erase 在指定位置删除一个元素（同 string）
- 可以使用 emplace 在指定位置构造一个元素
- 可以使用 emplace_back 在尾部新构造一个元素

关键操作的强异常安全保证

当 push_back、insert、reserve、resize 等函数导致内存重分配时，或当 insert、erase 导致元素位置移动时，vector 会试图把元素“移动”到新的内存区域。vector 的一些重要操作（如 push_back）试图提供强异常安全保证，即如果操作失败（发生异常）的话，vector 的内容完全不发生变化，就像数据库事务失败发生了回滚一样。如果元素类型没有提供一个保证不抛异常的移动构造函数，vector 此时通常会使用拷贝构造函数。因此，如果需要用移动来优化自己的元素类型而又不能应用零法则，那我们不仅要定义移动构造函数（和移动赋值运算符，虽然 push_back 不要求），还应当将其标为 noexcept。当然，我们可以使用现成的支持移动的对象，如容器（本章）和智能指针（第 11 章）。

下面的代码可以演示这一行为（vector_move.cpp）：

```
class Obj {
public:
    Obj() { cout << "Obj()\n"; }
    Obj(const Obj&) { cout << "Obj(const Obj&)\n"; }
    Obj(Obj&&) { cout << "Obj(Obj&&)\n"; }
};

int main()
{
    vector<Obj> v;
    v.reserve(2);
    v.emplace_back();
    v.emplace_back();
    v.emplace_back();
}
```

你可能会期望最后一次对 emplace_back 的调用会触发对移动构造函数的调用，但实际程序输出为：

```
Obj()
```

```
Obj()
Obj()
Obj(const Obj&)
Obj(const Obj&)
```

即拷贝构造函数被调用，而不是移动构造函数。

具体说来，push_back 在容量不足时做了以下动作：

1. 分配更大的内存块（一般是两倍）来容纳所有的元素。如果内存分配失败，则向外抛异常（结束当前流程），现有元素完全不受影响。
2. 在新内存块的指定位置构造新元素。如果构造失败，则释放新内存块并向外抛异常（结束当前流程），现有元素完全不受影响。
3. 把现有的元素“搬移”到新内存块中。

问题就在最后一步搬移中：vector 应该使用元素类型的移动构造函数还是拷贝构造函数呢？

答案是：如果元素类型的移动构造函数被标为 noexcept，从而保证不抛异常，那 vector 就会使用移动构造函数；否则，vector 会优先使用拷贝构造函数[①]。当使用保证不抛异常的移动构造函数时，显然搬移一定会成功，所以强异常安全能得到保证。当使用拷贝构造函数时，拷贝动作有可能失败，但因为原有的元素没有受到影响，在发生异常时 vector 仍能够回滚到原先的状态。

只要在移动构造函数的声明后加上 noexcept，我们就会得到一个不同的结果：

```
Obj()
Obj()
Obj()
Obj(Obj&&)
Obj(Obj&&)
```

emplace

C++11 开始提供的 emplace… 系列函数是为了提升容器的插入性能而设计的。如果把前面代码里的 v.emplace_back() 改成 v.push_back(Obj())，那结果相同，但性能方面有所不同——使用 push_back 会额外生成临时对象，多一次（移动或拷贝）构造和析构。如果是移动的情况，那会有小幅性能损失；如果对象没有实现移动，那性能差异就可能比较大了。

---

① 在只有移动构造函数而没有拷贝构造函数时，vector 会不得不使用移动构造函数。此时 push_back 成员函数就无法提供强异常安全保证了。

我们直观地对比一下。在加上移动构造函数的 noexcept 之后，运行结果是：

```
Obj()
Obj()
Obj()
Obj(Obj&&)
Obj(Obj&&)
```

如果把 v.emplace_back() 改成 v.push_back(Obj())，则结果变成：

```
Obj()
Obj(Obj&&)
Obj()
Obj(Obj&&)
Obj()
Obj(Obj&&)
Obj(Obj&&)
Obj(Obj&&)
```

作为简单的使用指南，当且仅当见到 v.push_back(Obj(…)) 这样的代码时，我们应将其改为 v.emplace_back(…)。

使用初始化器列表

在 C++98 里，标准容器比起 C 风格数组至少有一个明显的劣势：不能在代码里方便地初始化容器的内容。比如，对于数组你可以写：

```
int a[] = {1, 2, 3};
```

而对于 vector 你却得写：

```
vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
```

这样又啰唆，性能又差，无法让人满意。在 C++11 标准引入列表初始化后，现在初始化容器也可以和初始化数组一样简单了：

```
vector<int> v{1, 2, 3};
```

同样重要的是，这不是针对标准容器的特殊魔法，而是一个通用的、可以用于各种类的方法。从技术角度看，编译器的魔法只是对 {1, 2, 3} 这样的表达式自动生成了一个初始化器列表，在这个例子里其类型是 initializer_list<int>（详情可参考 [CppReference:

std::initializer_list])。程序员只需要声明一个接受 initializer_list 的构造函数即可使用。从效率角度看，至少在动态对象的情况下，容器和数组也并无二致，都是通过拷贝（构造）进行初始化。

鉴于现代 C++ 里允许使用花括号来调用构造函数和进行结构体聚合初始化，我们通常建议在构造对象时优先使用花括号初始化。除了一些个人偏好的情况[①]，这个推荐还有一个明显的例外：当一个类同时具有 initializer_list 构造函数和其他构造函数时，如需调用非 initializer_list 构造函数，应使用小括号语法。使用不同的括号此时可能产生不同的结果：vector<int> v{3, 5} 会产生元素 {3, 5}，而 vector<int> v(3, 5) 则会产生元素 {5, 5, 5}！

容器及其元素的移动行为

容器可以被移动，被移动之后的容器的元素全部转移到了新容器上。一个被移动了的对象在标准里通常规定为“有效但未指定的状态”，我们建议对这样的对象后续只应析构或重新初始化。Clang-Tidy 有一个检查项 bugprone-use-after-move，就是用来检查代码有没有这类问题，在对象被移动之后仍继续被使用。容器的重新初始化可通过以下方式实现：

- 赋值运算符
- clear 成员函数
- assign 成员函数

容器的迭代器在容器发生变化时常常会失效。对于 vector 的 erase 操作，所有指向被删除位置及其后面元素的迭代器（包括尾后的 end()）都会失效——因为这些元素被移动了。对于 insert，所有指向插入位置及其后面元素的迭代器都会失效；并且，当 vector 因扩容而移动元素时，这个 vector 的所有现有迭代器都将失效。不过，容器的交换操作保证不会让除 end() 之外的现有迭代器失效；而容器在被移动后，虽然标准里尚未提供这样的迭代器不失效保证，C++ 标准委员会在讨论中也认为这一般应当予以支持[②]。也就是说，下面的代码应该是合法有效的：

```
vector<int> v1{1, 2, 3};
auto it = v1.begin();
vector<int> v2 = std::move(v1);
cout << *it << '\n';         // 输出 1
cout << v1.empty() << '\n';  // 合法，但会触发 bugprone-use-after-move
```

① 比如，我一般在构造函数的成员初始化列表部分仍优先使用小括号，以便和构造函数的函数体部分有明显区别。

② https://cplusplus.github.io/LWG/issue2321

注意 string 因为有小字符串优化，不支持类似的用法。而 array 以外的容器都可以支持这样的移动后迭代器仍然有效的用法。

7.2.2 deque

deque 是一个内存部分连续的序列容器，它的意思是 double-ended queue（双端队列）。它主要用来满足下面这个需求：

- 容器既可以从尾部，也可以从头部，自由地添加和删除元素。

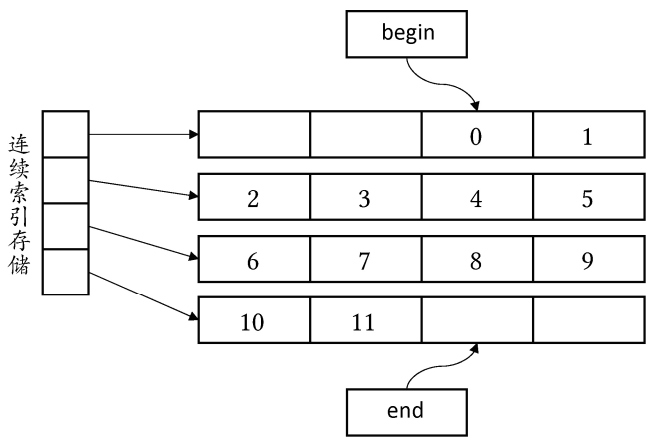

图 7-2：deque 的内存布局

deque 的内存布局一般如图 7-2 所示。可以看到：

- 如果只从头、尾两个位置对 deque 进行增删操作的话，容器里的元素永远不需要移动。[①]
- 容器里的元素只是部分连续。
- 由于元素的存储大部分仍然连续，它的遍历性能比较高。
- 由于每一段存储大小相等，deque 支持使用下标访问容器元素，大致相当于 index[i / chunk_size][i % chunk_size]，也保持高效。

① 因此，在首尾增删元素时，除了被删除的元素外，所有其他元素的引用都保持有效。注意，标准里规定迭代器还是会失效（虽然常见非调试实现里迭代器通常仍保持有效）。

因此，deque 的接口和 vector 相比，有如下主要区别：

- deque 提供 push_front、emplace_front 和 pop_front 成员函数。这表明它支持高效的开头插入动作。
- deque 不提供 data、capacity 和 reserve 成员函数。它没有容量和预分配容量的概念；因为内存只是部分连续的，提供 data() 也没有意义——通常提供 data() 的容器，就是为了把指针（和大小）传给类 C 的接口，会要求内存完全连续。

如果你需要一个经常在两端增删元素的容器，那 deque 会是个合适的选择。

在首尾增删元素时，deque 里的元素完全不会移动；这与 vector 不同，即使只在尾部添加元素，仍可能导致容器内的元素发生移动。因为这个原因，容器适配器 queue（见 7.5.1 节）和 stack（见 7.5.2 节）的默认底层容器都是 deque。

7.2.3 list

list 在 C++ 里代表链表。我们也可以说 list 是双向链表，因为它跟 string、vector、deque 一样，可以正向遍历，也可以反向遍历。它的内存布局如图 7-3 所示[①]。

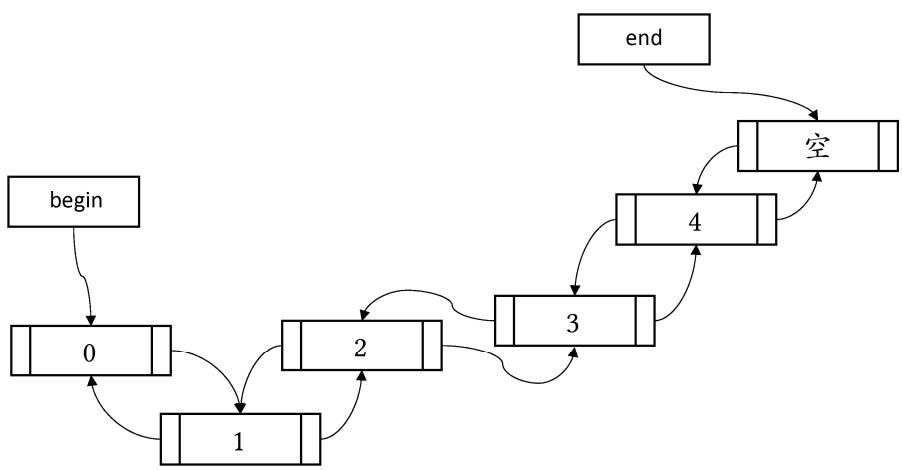

图 7-3：list 的内存布局

① 仅概念示意。真正的实现里一般不会浪费一个尾部的“空”结点。forward_list 的内存布局也是如此。

和 vector 相比，它优化了在容器中间的插入和删除：

- list 提供高效的、$O(1)$ 复杂度的任意位置的插入和删除操作。
- list 不提供使用下标访问其元素。
- list 提供 push_front、emplace_front 和 pop_front 成员函数（和 deque 相同）。
- list 不提供 data、capacity 和 reserve 成员函数（和 deque 相同）。

虽然 list 提供了任意位置插入新元素的灵活性，但由于每个元素的内存空间都是单独分配、不连续的，它的遍历性能比 vector 和 deque 都要低。这在很大程度上抵消了它在插入和删除操作时不需要移动元素的理论性能优势。如果你不太需要遍历容器、又需要在中间频繁插入或删除元素，可以考虑使用 list。

另外一个需要注意的地方是，因为某些标准算法不能在 list 上工作但仍存在适用于 list 的算法，或者算法能工作但潜在可能低效，list 提供了成员函数作为替代。这些成员函数只调整 list 结点的指针，而不会真正移动/交换元素，因此通常比可用的标准算法（注意标准 sort 算法此处不可用）更加高效。

- merge
- remove
- remove_if
- reverse
- sort
- unique

下面是一个完整的程序示例（list_member_func.cpp）：

```
#include <algorithm>        // std::sort
#include <iostream>         // std::cout/endl
#include <list>             // std::list
#include <vector>           // std::vector
#include "ostream_range.h"  // operator<< for ranges

using namespace std;

int main()
{
    list<int> lst{1, 7, 2, 8, 3};
    vector<int> vec{1, 7, 2, 8, 3};
```

```
    sort(vec.begin(), vec.end());     // 正常
    // sort(lst.begin(), lst.end());  // 不能编译
    lst.sort();                       // 正常

    cout << lst << endl;  // 输出 { 1, 2, 3, 7, 8 }
    cout << vec << endl;  // 输出 { 1, 2, 3, 7, 8 }
}
```

7.2.4 forward_list

既然 list 是双向链表，那么 C++ 里有没有单向链表呢？答案是肯定的。从 C++11 开始，前向列表 forward_list 成了标准的一部分。顾名思义，这个容器不支持反向遍历了，它没有 rbegin、rend、crbegin、crend 这些成员函数。

我们先看一下它的内存布局，如图 7-4 所示。

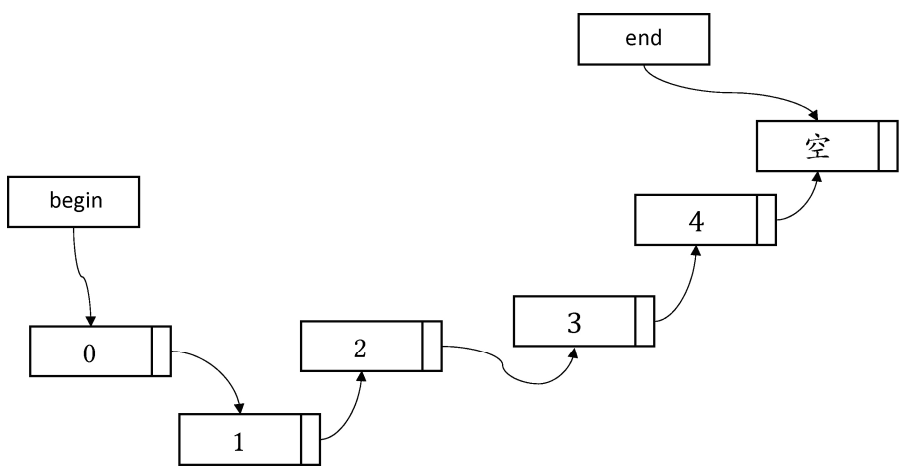

图 7-4：forward_list 的内存布局

大部分 C++ 容器支持 insert 成员函数，语义是从指定的位置之前插入一个元素。对于 forward_list，这不容易做到（想一想这是为什么），因此它提供了 insert_after 作为替代。类似地，它没有 emplace 和 erase，而有 emplace_after 和 erase_after。此外，跟 list 相比它还缺了下面这些成员函数：

- back
- size
- push_back

- emplace_back
- pop_back

为什么会需要这么一个阉割版的 list 呢？原因是，在元素大小较小的情况下，forward_list 能节约的内存是非常可观的（约三分之一）。提高内存利用率，往往就能提升程序性能。

这个容器相对来说使用机会较少一点。C++ 里的很多特性并非所有人都需要，但当你需要的时候，你就会觉得标准库提供这些特性非常有用了。

7.2.5 array

最后介绍一下 C++11 中新增用来替代 C 风格数组的序列容器，array。它和数组一样，内存一般在栈上分配（除非你手工使用 new），性能方面没有差异。也跟数组一样，你需要在编译期确定数组的长度，而不能在运行时动态决定。在声明一个 array 时，你需要写出两个模板参数：第一个是类型，第二个是 array 的长度（一个非类型模板参数）。

那我们为什么要使用 array 而不是 C 风格数组呢？最主要的理由是：

- 解决 C 数组的怪异行为：不能按值拷贝（除非放在结构体里）；值传参有退化行为，被调用函数不再能获得 C 数组的长度（除非使用引用）。
- 能够直接支持 ==、< 等比较运算（如果元素支持的话），使用更加方便。

下面的代码不管在 C 还是 C++ 里都是有问题的：

```
int a[3] = {1, 2, 3};
int b[3] = a;  // 不能编译
b = a;         // 不能编译
```

一个函数如果参数是 int a[3] 的话，实际跟 int a[] 或 int* a 完全等效：数组会退化成为指针。同理，a < b 这样的比较也可能跟你的想象相去甚远：它只是做了一个极其无聊的指针比较而已。

在 C 的年代，大家有时候会定义这样一个宏来获得数组的长度：

```
#define ARRAY_LEN(a) (sizeof(a) / sizeof((a)[0]))
```

如果在函数内部对数组参数使用这个宏，那代码虽然合法，结果却是错的：

```
void test(int a[8])
{
```

```
    cout << ARRAY_LEN(a) << endl;
}
```

幸好，现代编译器一般已经会友好地发出告警，如：

```
warning: 'sizeof' on array function parameter 'a' will return size of 'int
*' [-Wsizeof-array-argument]
cout << ARRAY_LEN(a) << endl;
```

可以看到，数组传参跟其他类型的传参有着明显的不同，更容易被误用。

次要一点，array 对所有人而言都应该可读性更高。试比较下面两种写法（函数指针的数组）：

```
int (*fpa[3])(const char*);
array<int (*)(const char*), 3> fpa;
```

以及这两种（返回整数数组指针的函数的指针）：

```
int (*(*fp)(const char*))[3];
array<int, 3>* (*fp)(const char*);
```

最后，跟普通容器一样，array 提供了 begin、end、size 等通用成员函数。因此更容易在泛型代码中使用。对于下面的代码，你可以传 array、vector、list 等容器，但不能传 C 风格的数组：

```
template <typename T>
double average(const T& container)
{
    double sum = 0.0;
    for (auto it = container.begin(); it != container.end();
         ++it) {
        sum += *it;
    }
    return sum / container.size();
}
```

不过，这一优点到了 C++17 有所减弱，因为在 std 名空间里已经定义了独立的 begin、end、size 等函数[①]，可以用于容器，也可以用于数组：

```
template <typename T>
double average(const T& container)
{
```

① C++11 时已经有了独立的 begin/end 函数模板，但 size 要到 C++17 才加入。

```
    using std::begin;
    using std::end;
    using std::size;
    double sum = 0.0;
    for (auto it = begin(container); it != end(container);
         ++it) {
        sum += *it;
    }
    return sum / size(container);
}
```

上面的代码具有最高的通用性：不仅可以用于标准容器和数组，也可以用于用户在自己名空间里定义的特殊容器或结构体。前提是类型合理地支持了这三种操作——它们可以作为成员函数提供，也可以在类型所在的名空间下作为独立函数提供（这样可通过实参依赖查找发现）。

顺便说一句，`std::size` 只能应用于数组，而不能应用于指针。假设我们在本节开头的错误代码中使用的是 `std::size` 的话，编译就会直接失败（而不仅是告警）。所以，如果你还在 C++ 里用 `ARRAY_LEN` 宏的话，是时候替换掉它们了！

7.3 关联容器

如果历史可以重来，关联容器（associative container）实际上应该叫有序关联容器，这样我们可以：

- 明确强调它们的内部元素排列有序
- 可以用“关联容器”一词来统一称呼本节和 7.4 节里所有的容器

很遗憾，历史不能重来，术语也已经基本定型。现在我们只能记住，“关联容器”表示的是一组元素排列有序的容器。因为排列有序，它们既支持普通的（正向）遍历，也支持反向遍历。

7.3.1 排序问题

既然元素是有序的，那一个基本问题就是如何进行排序。所有关联容器都有一个表示排序方式的比较（`Compare`）模板参数，其默认值是键类型的 `less`。比如，一个整数的集合 `set<int>`，这个比较模板参数默认就是 `less<int>`。