C++20 模板元编程

[罗] 马里乌斯 • 班西拉(Marius Bancila) 著
何荣华 王文斌 张毅峰 杨文波 译

清華大學出版社

北 京

北京市版权局著作权合同登记号　图字：01-2024-4706

图书在版编目（CIP）数据

C++20 模板元编程 / (罗) 马里乌斯·班西拉著 ;
何荣华等译. -- 北京 : 清华大学出版社, 2025. 6.
ISBN 978-7-302-69436-6

I. TP312.8

中国国家版本馆 CIP 数据核字第 2025MV5773 号

责任编辑：王　军
封面设计：高娟妮
版式设计：恒复文化
责任校对：成凤进
责任印制：丛怀宇

出版发行：清华大学出版社
网　　址：https://www.tup.com.cn，https://www.wqxuetang.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-83470000　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn

印 装 者：北京同文印刷有限责任公司
经　　销：全国新华书店
开　　本：170mm×240mm　　印　　张：20　　字　　数：435 千字
版　　次：2025 年 7 月第 1 版　　印　　次：2025 年 7 月第 1 次印刷
定　　价：99.80 元

产品编号：108440-01

玄之又玄，众妙之门

大家都知道，AI 可以写代码了。你用过吗？感觉如何？

前不久，我在准备 GPU 训练营的试验程序时，确实用 AI 写了一些代码，既有传统的 CPU 端代码，也有更现代的 GPU 端代码。我用 AI 写代码的目的有两个，一是提高工作效率，二是亲身测试 AI 写代码的能力。

亲身测试一番之后，我有两个比较强烈的感受。第一个感受是对于比较简单的任务，AI 确实可以写出质量不错的代码，不仅速度快，而且准确度很高，没有误拼等人类常犯的低级错误。第二个感受是，随着代码量的上升，AI 写的代码也开始具有人类代码常有的问题，先是重复，啰嗦，然后是有 bug(错误)。

众所周知，AI 领域吸引了大量的投资和优秀的人才，新的成果不断涌现。因此，我们比较难预测 AI 的代码能力在 2 年后会怎么样？在 5 年和 10 年后又会怎么样？

AI 技术的发展速度难以预测，但是我觉得以下三个趋势是比较确定的。首先，AI 技术确实会改变软件产业的格局，一些简单的软件开发任务将 AI 化，因为使用 AI 技术能大大提高编码的效率，不再需要那么多的程序员来写代码。第二，随着 AI 技术不断被应用到软件开发领域，软件的产量和软件的代码量都将随之上升。而且，AI 产生的代码也是不完美并且存在瑕疵的。软件团队里将需要很多调试工程师来定位各种稀奇古怪的问题。第三，在追求高性能、高可靠性的某些领域里，仍需要优秀的人类程序员来编写极端精致的代码。就像在机器可以包饺子的今天，仍有某些饺子店使用人工包。

其实，不管我的预测是否对，一名好的程序员都应该不断锤炼自己的编码能力，提高技术水平，让自己写出的代码越来越好。

于是，可能有人问，我已经能写出很漂亮的代码，什么样的代码算是更好呢？

的确，评价代码好坏的标准有很多。在我看来，第一个硬指标就是 generic，也就是通用性。展开来说，很多代码都有的一个通病就是长相类似的代码有很多份，结构类似，但有差异，不完全相同。

我是信儒家的，但偶尔也会读一点道家的作品，一般是在睡前读，因为读道家的作品读着读着就昏昏欲睡了。为什么呢？因为道家的话一般都比较“虚空”。用时髦的话说，就是不接地气，难以琢磨。比如一句“道可道，非常道”就有很多种解释。

我对道家的这种态度持续了很多年，直到有一天，当我领悟了计算机世界的一系列经典案例和一个永恒规律后，我又看到了“玄之又玄，众妙之门”。这八个字归纳得太好了，说出我心中所有，笔下所无，改变了我对道家的态度。

什么是玄而又玄呢？传统的解释有很多种，对多数程序员来说，都不大好理解。

在我看来，玄就是抽象。玄之又玄，就是抽象了再抽象。

人类的大脑喜欢生动具体的东西，比如小孩子都喜欢听故事，无论是“小马过河”还是“后羿射日”都有具体的场景、“人”和物。长大了以后喜欢刷剧也是类似的原因。每部剧都在一个具体的时空中讲一个故事。没有哪部剧没有人物，只有“道可道，非常道”。

因此，做抽象是很难的事情。也因此，很多代码都是不够抽象的，今天需要 int 类型的 max()函数，那么就写个 int 类型的；明天需要 float 类型的，就把 int 类型的复制一份，改成 float 类型的。日积月累，整个项目里就有很多长相类似的代码了。

如何提炼这样的代码，消除重复，把它们合众为一呢？

传统 C++中的模板技术就是为解决这个问题而设计的，现代 C++将其发扬光大，去除约束，增加功能，使其成为现代 C++语言的一大亮点。

我认识文波和荣华多年，他们都在 C ++语言和编程技术领域耕耘多年，孜孜不倦，满怀深情。更加可贵的是，他们把热爱转化为实际的行动，以各种形式推动技术的传播和发展。他们在翻译《C++模板》(第 2 版)之后，又将另一本模板编程的好书《C++20 模板元编程》翻译成中文，功莫大焉。

张银奎

《软件调试》和《软件简史》的作者

译 者 序

C++的演化与模板的历史

自 1979 年 Bjarne Stroustrup 创建 C++以来，这门语言经历了多个重要的标准化版本，每一次演进都带来了新的特性和改进。从 C++98 的标准化到 C++11 迎来现代 C++编程范式，再到 C++14、C++17 的稳定和扩展，现在 C++20 作为一个里程碑式的更新，引入了概念(Concepts)、范围(Ranges)、协程(Coroutines)等强大特性。其中，C++20 对模板系统的扩展和改进，使得泛型编程更加直观、高效。

C++模板的历史可以追溯到 20 世纪 80 年代后期，它最初是为了解决代码复用的问题。1998 年的 C++标准(C++98)正式引入了模板，随后在 C++11 中得到了重要增强，如变参模板(Variadic Templates)、模板别名(Template Aliases)等。C++17 进一步引入了折叠表达式(Fold Expressions)和类模板实参推导(Class Template Argument Deduction，CTAD)。到了 C++20，概念(Concepts)的加入使得模板的可读性、可维护性大幅提升。

C++模板的优势

C++是一门支持多种编程范式的语言，包括：

- **过程式编程(Procedural Programming)**——基于函数和过程的结构化编程。
- **面向对象编程(OOP)**——通过类、继承和多态实现模块化与复用。
- **泛型编程(Generic Programming)**——借助模板编写类型无关的代码，提高代码复用性和灵活性。
- **函数式编程(Functional Programming)**——使用不可变数据和高阶函数，提升代码可测试性和并发能力。
- **元编程(Metaprogramming)**——利用编译期计算优化程序，提高运行效率。

在这些范式中，**模板技术是 C++的核心特性**，它赋予 C++强大的泛型编程能力，使代码适用于多种数据类型，而不需要冗余编写。例如，标准模板库(STL)的容器(如 std::vector、std::map)和算法(如 std::sort、std::find)均依赖模板实现。

C++模板的主要优势包括：

- **编译时多态(Compile-time Polymorphism)**——相比运行时多态(如继承与虚函数)，模板允许编译期进行类型推导和优化，从而提高执行效率。
- **编译时计算(Compile-time Computation)**——利用模板元编程(TMP)，C++能在编译期执行计算，减少运行时开销。例如，std::integral_constant 和 std::conditional 可用于选择编译期代码路径。
- **代码复用**——模板减少了重复代码，提高了通用性。例如，std::enable_if 可用于 SFINAE(替换失败非错误)，实现条件编译。

模板的强大使其在现代 C++开发中占据重要地位，特别是在高性能计算、游戏开发、底层系统编程等领域。掌握模板不仅能提升代码质量，还能帮助程序员深入理解 C++语言的底层机制。

C++程序员必备的技能

对于希望深入掌握 C++的开发者而言，理解模板是进阶 C++编程的必经之路。从泛型编程(Generic Programming)、模板元编程(Template Metaprogramming)，到 C++20 概念(Concepts)，这些技术都在现代 C++开发中占据了重要地位。

无论是编写高效的库函数，还是优化应用程序的性能，模板都是必不可少的工具。例如，在高性能计算(HPC)、游戏开发、底层系统编程等领域，模板能够提供无与伦比的灵活性和效率。掌握模板不仅能够提高代码质量，还能帮助程序员更深入地理解 C++语言的底层机制。

模板技术的学习建议

模板技术属于编译期编程，在学习过程中，建议结合反汇编，并善用 Cpp Insights 等工具来观察模板实例化和生成的代码。

要系统学习模板技术，仅靠一本书是不够的。推荐阅读以下书籍：

- 《C++ Templates (第 2 版・中文版)》[*C++ Templates: The Complete Guide, 2nd Edition*, (美)David Vandevoorde、(德)Nicolai M. Josuttis、(美)Douglas Gregor 著，何荣华、王文斌、张毅峰、杨文波译，人民邮电出版社]——经典的 C++模板书籍，全面介绍了模板技术。
- 《编程原本》[*Elements of Programming*，(美)Alexnader Stepanov、(美) Paul McJones 著，裘宗燕译，人民邮电出版社]和亚历山大的系列博文(https://www.stepanovpapers.com/)——进一步深入泛型编程。
- 《C 实战：核心技术与最佳实践》(吴咏炜著，人民邮电出版社)——现代 C 最佳实践。

此外，学习模板技术不能只依赖理论，还需要实践。建议阅读并改造以下模板库：

1. fmt(std::format 的实现)——不依赖领域知识，适合作为入门教材。
2. blaze 高性能数学库——作者 Klaus Iglberger 是模板设计模式专家。
3. folly 库——Andrei Alexandrescu 主导，适合学习模板元编程。
4. cutlass C++模板库——Nvidia 出品，适用于深度学习优化。

结语与感谢

C++模板的强大使得它成为现代 C++开发的基石，而 C++20 的更新更是让模板变得更易用、更强大。本书的目标是帮助读者全面理解 C++20 模板的核心概念，并掌握如何在实际开发中高效地应用模板技术。希望本书能为你打开 C++模板编程的大门，帮助你在 C++领域更进一步。

本书的出版离不开各方的支持。我们衷心感谢 Bjarne Stroustrup 教授，他的贡献不仅塑造了 C++语言，也为全球开发者提供了深远的技术指导。

特别感谢清华大学出版社的编辑团队，他们在术语规范、技术表达及出版质量方面提供了宝贵的支持，付出了大量的辛勤工作，确保本书得以高质量呈现。我们也感谢 C++社区的开发者们，你们的深入讨论与实践经验为我们提供了极大的启发。

尽管我们力求精确，但面对 C++如此庞大而复杂的体系，难免仍有不足之处。我们诚恳欢迎读者通过出版社反馈意见，以便我们进一步完善后续的修订工作。希望本书能够帮助广大开发者更深入地理解 C++，更高效地运用这门强大语言。

献给那些总是渴望学习更多的好奇心灵。

——Marius Bancila

贡 献 者

关于作者

Marius Bancila 是一位拥有 20 年经验的软件工程师，在业务应用程序开发和其他领域都有丰富的解决方案经验。他是 *Modern C++ Programming Cookbook* 和 *The Modern C++ Challenge* 的作者。他目前担任软件架构师，专注于微软技术，主要使用 C++和 C#开发桌面应用程序。他热衷于与他人分享技术专长，因此自 2006 年起一直被认定为微软 C++ MVP，后来还获得了开发者技术领域的 MVP 称号。Marius 居住在罗马尼亚，活跃于各种在线社区。

关于审校者

Aleksei Goriachikh 拥有超过 8 年的 C++编程经验。2012 年从俄罗斯的新西伯利亚国立大学获得数学硕士学位后，Aleksei 曾参与一些计算数学和优化领域的研究项目、某 CAD 系统的几何内核开发，以及自动驾驶的多线程库开发。Aleksei 最近的专业兴趣是硅前建模。

前　言

几十年来，C++一直是世界上使用最广泛的编程语言之一。它的成功不仅仅归功于其提供的性能或者说它的易用性(许多人对此持不同意见)，而更可能是由于它的多功能性。C++是一种通用的多范式编程语言，它融合了过程式、函数式和泛型编程。

泛型编程是一种编写代码的方式，例如函数和类等实体是按照稍后实例化的类型编写的。这些泛型实体仅在需要作为实参具化为特定类型时才会实例化，这些泛型实体在C++中称为模板。

元编程是一种编程技术，它使用模板(以及 C++中的 constexpr 函数)在编译期生成代码，然后将其与剩余源代码合并以便编译最终程序。元编程意味着其输入或输出中至少有一个是类型。

正如《C++核心指南》(Bjarne Stroustrup 和 Herb Sutter 维护的一份关于应该做什么和不应该做什么的文档)中所描述的那样，C++中的模板可谓声名狼藉。然而，它们使得泛型库成为可能，比如 C++开发人员一直使用的 C++标准库。无论你是自己编写模板，还是只使用他人编写的模板(比如标准容器或算法)，模板都很可能是你日常编码的一部分。

本书旨在让读者对 C++中可用的所有范围内的模板都有很好的理解(从基本语法到C++20 中的概念)，这是本书前两部分的重点内容。第III部分会帮助你将新获得的知识付诸实践，并使用模板进行元编程。

本书适读人群

本书适合想要学习模板元编程的初学者、中级 C++开发人员，以及希望快速掌握与模板相关的 C++20 新功能和各种惯用法和模式的高级 C++开发人员。在开始阅读本书之前，必须具备基本的 C++编程经验。

本书涵盖内容

第 1 章“模板简介”。通过几个简单的例子介绍了 C++中模板元编程的概念，讨论了为什么我们需要模板以及模板的优缺点。

第 2 章“模板基础”。探讨了 C++中所有形式的模板：函数模板、类模板、变量模

板和别名模板。我们讨论了其中每一个的语法和它们如何工作的细节。此外，还讨论了模板实例化和特化的关键概念。

第 3 章“变参模板”。专门介绍了变参模板，即具有可变数量模板形参的模板。我们详细讨论了变参函数模板、变参类模板、变参别名模板和变参变量模板、形参包及其展开方式，以及帮助我们简化编写变参模板的折叠表达式。

第 4 章“高级模板概念”。对一系列高级模板概念进行了分组，比如依赖名称和名称查找、模板实参推导、模板递归、完美转发、泛型和模板 lambda 函数。通过了解这些主题，读者将能够极大地扩展他们可以阅读或编写的模板的种类。

第 5 章“类型特征和条件编译”。专门讨论类型特征。读者将了解类型特征、标准库提供的特征以及如何使用它们解决不同的问题。

第 6 章“概念和约束”。介绍了新的 C++20 机制，通过概念和约束定义模板实参的需求。你将了解指定约束的各种方法。此外，还概述了 C++20 标准概念库的内容。

第 7 章“模式和惯用法”。探讨了一系列独立的高级主题，即利用迄今为止学到的知识实现各种模式。我们探讨了静态多态、类型擦除、标签派发和模式的概念，比如奇异递归模板模式、表达式模板、混入和类型列表。

第 8 章“范围和算法”。专注于理解容器、迭代器和算法，它们是标准模板库的核心组件。你将在这里学习如何为其编写泛型容器和迭代器类型以及通用算法。

第 9 章“范围库”。探讨了新的 C++20 范围库及其关键特性，例如范围、范围适配器和约束算法。这些使我们能够编写更简单的代码来处理范围。此外，你还将在这里学习如何编写自己的范围适配器。

附录是一个简短的结语，提供了本书的总结。

问题答案包含了所有章节中习题的答案。

充分利用本书

要开始阅读本书，首先需要对 C++编程语言有一些基本的了解。需要了解有关类、函数、运算符、函数重载、继承、虚函数等的语法和基础知识。不过，对模板知识不做要求，因为本书将从头开始教你一切。

本书中的所有代码示例都是跨平台的。这意味着你可以使用任何编译器来构建和运行它们。然而，尽管许多代码段适用于 C++11 编译器，但也有一些代码段需要兼容 C++17 或 C++20 的编译器。因此，建议你使用支持 C++20 的编译器版本，以便运行所有示例。书中的示例已使用 **MSVC 19.30 (Visual Studio 2022)**、**GCC 12.1/13** 和 **Clang 13/14** 进行了测试。如果你的机器上没有这样一个兼容 C++20 的编译器，可以试着网上下载一个。我们推荐以下几个方案：

- Compiler Explorer (https://godbolt.org/)

- Wandbox (https://wandbox.org/)
- C++ Insights (https://cppinsights.io/)

本书多次引用 C++ Insights 在线工具来分析编译器生成的代码。

如果你想检查编译器对不同版本的 C++ 标准的支持，应该参考页面 https://en.cppreference.com/w/cpp/compiler_support。

提及标准和延伸阅读

在本书中，我们会多次提到 C++标准。此文件版权归**国际标准化组织**所有。官方的 C++标准文档可以从这里购买：https://www.iso.org/standard/79358.html。但是，C++标准的多个草案以及相应源码可以在 GitHub 上免费获得，网址为 https://github.com/cplusplus/draft。可以在 https://isocpp.org/ std/the-standard 链接上找到有关 C++标准的更多信息。

C++ Reference 网站是 C++开发人员的一个很好的在线资源，网址为 https://en.cppreference.com/。它提供了直接派生自 C++标准的 C++语言的详尽文档。本书多次引用了 C++参考中的内容。C++参考的内容是基于 CC-BY-SA 协议的，https://en.cppreference.com/w/Cppreference:Copyright/CC-BY-SA。

(在每一章的末尾，你会发现一个名为“延伸阅读”的部分，该部分包含一份用作参考书目的阅读材料清单，推荐阅读以加深对所介绍主题的理解。)

下载示例代码文件

可以从 GitHub 上下载本书的示例代码文件，网址为 https://github.com/PacktPublishing/Template-Metaprogramming-with-CPP。也可以扫描封底二维码下载。如果代码有更新，将会在 GitHub 仓库中更新它。

下载彩图

我们还提供了一个 PDF 文件，其中包含本书中使用的屏幕截图和图表的彩图。可以在此处下载：https://packt.link/Un8j5。也可以扫描封底二维码下载。

使用的约定

本书使用了一些文本约定。

文本中的代码：表示文本中的代码词汇、数据库表名、文件夹名、文件名、文件扩展名、路径名、虚拟 URL、用户输入和 Twitter 用户名/账号标识。这里有一个例子：“这个问题可以通过将 init 设置为依赖名称来解决。”

代码块的格式如下：

```
template<typenameT>
structparser:base_parser<T>
{
   voidparse()
   {
      this->init(); // 正确
      std::cout<<"parse\n";
   }
};
```

按如下方式编写任意命令行的输入或输出：

```
fatal error:recursive template instantiation exceeded maximum depth of 1024
use -ftemplate-depth=N to increase recursive template instantiation depth
```

粗体：表示一个新术语、一个重要单词或你在屏幕上看到的单词。例如，菜单或对话框中的单词以**粗体**显示。这里有一个例子：“容量为 8，大小为 0，**头部**和**尾部**都指向索引 0。”

提示或重要说明

迭代器概念在第 6 章“概念与约束”中进行了简要讨论。

本书的参考文献和问题答案可扫描封底二维码下载。

目　录

第 I 部分　模板的核心概念

在这部分中，你将首先从模板简介开始，了解模板的优点。然后，你将学习编写函数模板、类模板、变量模板和别名模板的语法。你还将探索模板实例化和模板特化等概念，并学习如何编写参数数量可变的模板。

包含以下章节：

- 第 **1** 章　模板的简介
- 第 **2** 章　模板的基础
- 第 **3** 章　变参模板

第1章
模板的简介

作为一名 C++开发人员，你应该至少熟悉甚至精通**模板元编程(template metaprogramming)**，通常简称为**模板(template)**。模板元编程是一种编程技术，它使用模板作为编译器的蓝图来生成代码，帮助开发人员避免编写重复代码。尽管通用库大量使用模板，但 C++语言中模板的语法和内部工作机制可能令人望而生畏。即使是由 C++语言之父 Bjarne Stroustrup 和 C++标准化委员会主席 Herb Sutter 编写的 C++核心准则(*C++ Core Guidelines*)，这部包含了各种建议和禁忌的 C++语言的指南汇总，也称模板“相当恐怖”。

本书旨在阐明 C++语言中的这一领域，并帮助你熟练掌握模板元编程。

在本章中，我们将讨论以下主题：

- 理解模板的必要性
- 编写你的第一个模板
- 理解模板术语
- 模板的简史
- 模板的优缺点

学习如何使用模板的第一步是理解它们实际解决的问题。让我们从这里开始。

1.1 理解模板的必要性

每种语言特性设计出来，都旨在帮助开发人员解决使用该语言时遇到的问题或任务。模板的目的是帮助我们避免编写只有细微差异的重复代码。

为了说明这一点，我们以经典的 `max` 函数为例。函数接收两个数值实参，并返回其中较大的一个。我们可以很容易地实现，参见如下代码。

```
int max(int const a, int const b)
{
  return a > b ? a : b;
}
```

这能满足需求，但显然，它只适用于 int 类型(或可转换为 int 的类型)。如果我们需要同样的函数，但实参类型为 double，怎么办？这时，可以为 double 类型重载这个函数(创建一个具有相同名称但实参数目或者类型不同的函数)。

```
double max(double const a, double const b)
{
  return a > b ? a : b;
}
```

但是，数值类型不仅有 int 和 double。还有 char、short、long、long long 和它们的无符号对应类型 unsigned char、unsigned short、unsigned long、unsigned long long。还有 float 和 long double 类型，以及其他类型，如 int8_t、int16_t、int32_t 和 int64_t。还可能有其他可以进行比较的类型，如 bigint、Matrix、point2d 以及任何重载了 operator>的用户定义类型。通用库如何能提供一个通用的 max 函数来处理所有这些类型？它可以为所有内置类型和一些库中的类型重载这个函数，但无法为任何用户自定义类型重载这个函数。

一种替代的方案是使用 void* 传递不同类型的实参。请记住，这是一种糟糕的实践，下面的示例仅仅是在没有模板的世界中可能的一种替代方案。于是，出于讨论的需要，我们可以设计一个排序函数，它可在提供严格弱序的类型的元素组成的数组上运行快速排序算法。快速排序算法的细节可以在线上查找，例如在维基百科上查找，网址是 https://en.wikipedia.org/wiki/Quicksort。

快速排序算法需要比较和交换任意两个元素。但是，由于我们不知道它们的类型，在函数实现中就不能直接进行这些操作。解决方案是依赖**回调函数**，把它作为实参传入，在必要时调用。一种可能的实现如下所示。

```
using swap_fn = void(*)(void*, int const, int const);
using compare_fn = bool(*)(void*, int const, int const);

int partition(void* arr, int const low, int const high,
              compare_fn fcomp, swap_fn fswap)
{
  int i = low - 1;

  for (int j = low; j <= high - 1; j++)
  {
    if (fcomp(arr, j, high))
    {
      i++;
      fswap(arr, i, j);
    }
  }
  fswap(arr, i + 1, high);

  return i + 1;
}
```

```
void quicksort(void* arr, int const low, int const high,
               compare_fn fcomp, swap_fn fswap)
{
  if (low < high)
  {
    int const pi = partition(arr, low, high, fcomp,
      fswap);
    quicksort(arr, low, pi - 1, fcomp, fswap);
    quicksort(arr, pi + 1, high, fcomp, fswap);
  }
}
```

为了调用 quicksort 函数，需要为传递的每种类型的数组提供这些比较和交换函数的实现。如下代码所示是 int 类型的实现。

```
void swap_int(void* arr, int const i, int const j)
{
  int* iarr = (int*)arr;
  int t = iarr[i];
  iarr[i] = iarr[j];
  iarr[j] = t;
}

bool less_int(void* arr, int const i, int const j)
{
  int* iarr = (int*)arr;
  return iarr[i] <= iarr[j];
}
```

有了这些定义，可以编写如下代码对整数数组进行排序。

```
int main()
{
  int arr[] = { 13, 1, 8, 3, 5, 2, 1 };
  int n = sizeof(arr) / sizeof(arr[0]);
  quicksort(arr, 0, n - 1, less_int, swap_int);
}
```

这些示例针对的是函数的问题，但同样也适用于类。假设你要编写一个类，该类模拟一个大小可变的数值集合，并将元素连续存储在内存中。可以为存储整数提供以下实现(这里只简要给出了声明)。

```
struct int_vector
{
  int_vector();

  size_t size() const;
  size_t capacity() const;
  bool empty() const;

  void clear();
  void resize(size_t const size);

  void push_back(int value);
  void pop_back();
```

```
    int at(size_t const index) const;
    int operator[](size_t const index) const;
private:
    int* data_;
    size_t size_;
    size_t capacity_;
};
```

这看上去都挺好，但是一旦需要存储 double 类型、std::string 类型或任何用户定义类型的值，你就必须编写相同的代码，而每次只改变元素的类型。这是没有人愿意干的活，因为它属于重复劳动，并且当需要改变某些内容(例如添加新功能或修复 bug)时，就需要在多个地方应用相同的改变。

最后，类似的问题在需要定义变量时也可能遇到，尽管并不常见。让我们考虑一个保存换行字符的变量。可以按如下方式声明它：

```
constexpr char NewLine = '\n';
```

如果你需要相同的常量，但用于不同的编码，如宽字符串字面量、UTF-8 等，该怎么办？可以定义多个变量，使用不同的名称，如下所示。

```
constexpr wchar_t NewLineW = L'\n';
constexpr char8_t NewLineU8 = u8'\n';
constexpr char16_t NewLineU16 = u'\n';
constexpr char32_t NewLineU32 = U'\n'
```

模板是一种技术，支持开发人员编写蓝图，让编译器能够为我们生成所有这些重复的代码。在下一节中，我们将看到如何将前面的代码片段转换为 C++模板。

1.2　编写你的第一个模板

现在是时候看看如何在 C++语言中编写模板了。在本节中，我们将从 3 个简单的示例开始，分别对应前面介绍的代码片段。

前面讨论过的 max 函数的模板版本如下所示。

```
template <typename T>
T max(T const a, T const b)
{
    return a > b ? a : b;
}
```

你会注意到，这里类型名(如 int 或 double) 已被替换为 T (表示类型)。T 称为**类型模板形参**，由语法 template<typename T>或 typename<class T>引入。请记住，T 是形参，因此它可以有任何名称。我们将在第 2 章中了解更多关于模板形参的内容。

到目前为止，你在源代码中放置的这个模板只是蓝图。编译器将根据它的使用情况生成代码。更准确地说，它将为使用模板的每种类型实例化一个函数重载。下面是一个例子。

```
struct foo{};

int main()
{
   foo f1, f2;
   max(1, 2);      // 正确, 比较 int
   max(1.0, 2.0); // 正确, 比较 double
   max(f1, f2);    // 错误, foo 没有重载 operator>
}
```

在以上代码片段中，首先传入两个整数来调用 max，这是可以的，因为 int 类型有 operator>。这将生成一个重载 int max(int const a, int const b)。之后，传入两个 double 来调用 max，这也是正确的，因为 double 类型支持 operator>。因此，编译器将生成另一个重载 double max(double const a, double const b)。但是，对 max 的第三次调用将生成一个编译错误，因为 foo 类型没有重载 operator>。

需要指出，调用 max 函数的完整语法如下，其细节暂不深究。

```
max<int>(1, 2);
max<double>(1.0, 2.0);
max<foo>(f1, f2);
```

这里编译器能推导出模板形参的类型，写出类型反而变得多余。但是，在某些情况下编译器无法进行推导，那你就需要用以上语法明确指定类型。

第二个示例涉及 1.1 节“理解模板的必要性”中以 void*传参的 quicksort()实现。它只需要很少的改动就能轻而易举地转换为模板版本。正如以下代码所示。

```
template <typename T>
void swap(T* a, T* b)
{
   T t = *a;
   *a = *b;
   *b = t;
}

template <typename T>
int partition(T arr[], int const low, int const high)
{
   T pivot = arr[high];
   int i = (low - 1);

   for (int j = low; j <= high - 1; j++)
   {
      if (arr[j] < pivot)
      {
```

```
        i++;
        swap(&arr[i], &arr[j]);
      }
    }

    swap(&arr[i + 1], &arr[high]);

    return i + 1;
}

template <typename T>
void quicksort(T arr[], int const low, int const high)
{
    if (low < high)
    {
      int const pi = partition(arr, low, high);
      quicksort(arr, low, pi - 1);
      quicksort(arr, pi + 1, high);
    }
}
```

quicksort 函数模板使用起来同之前非常类似，只是不再需要传递回调函数的指针。

```
int main()
{
    int arr[] = { 13, 1, 8, 3, 5, 2, 1 };
    int n = sizeof(arr) / sizeof(arr[0]);
    quicksort(arr, 0, n - 1);
}
```

前一节中的第三个示例是 `vector` 类。它的模板版本如下所示：

```
template <typename T>
struct vector
{
    vector();

    size_t size() const;
    size_t capacity() const;
    bool empty() const;

    void clear();
    void resize(size_t const size);

    void push_back(T value);
    void pop_back();

    T at(size_t const index) const;
    T operator[](size_t const index) const;
private:
    T* data_;
    size_t size_;
    size_t capacity_;
};
```

与 `max` 函数的情况一样，改动很小。模板声明在类上方一行，元素的 `int` 类型被

类型模板形参 T 替换。可以按如下方式使用：

```
int main()
{
  vector<int> v;
  v.push_back(1);
  v.push_back(2);
}
```

需要注意的是，在声明变量 v 时，必须指定元素的类型，在这里是 int，因为如果不这样做，编译器就无法推断它们的类型。在某些情况下，可以不指定类型(在 C++17 中)，这被称为**类模板实参推导**，将在第 4 章“高级模板概念”中讨论。

第四个也是最后一个示例是关于声明多个只有类型不同的变量。我们可以用一个模板替换所有这些变量，如下面的代码片段所示。

```
template<typename T>
constexpr T NewLine = T('\n');
```

可以如以下代码所示使用此模板。

```
int main()
{
  std::wstring test = L"demo";
  test += NewLine<wchar_t>;
  std::wcout << test;
}
```

本节的示例表明，无论用于表示函数、类还是变量，声明和使用模板的语法都一样。下节讨论模板的类型和术语。

1.3　理解模板术语

到目前为止，我们用的都是通用术语“模板”。其实，上文编写的模板可以用 4 个不同的术语分别描述。

- **函数模板(function template)**是指模板化的函数。之前看到的 max 模板就是一个例子。
- **类模板(class template)**是指模板化的类(可以使用关键字 class、struct 或 union 定义)。在上一节我们编写的 vector 类就是一例。
- **变量模板(variable template)**是指模板化的变量，比如上一节中的 NewLine 模板。
- **别名模板(alias template)**是指模板化的类型别名。我们将在下一章中见到别名模板的例子。

模板由一个或多个形参参数化(到目前为止，我们看到的例子都只有一个形参)。这

些形参称为**模板形参**，可以分为 3 类。

- **类型模板形参(type template parameters)**，如 `template<typename T>`，形参表示的是使用模板时指定的类型。
- **非类型模板形参(non-type template parameters)**，如 `template<size_t N>` 或 `template<auto n>`，每个形参必须有一个结构化类型，包括整型、浮点型(C++20 支持)、指针类型、枚举类型、左值引用类型等。
- **模板模板形参(template template parameters)**，如 `template<typename K, typename V, template<typename> typename C>`，形参的类型是另一个模板。

可以通过提供替代实现来特化模板。这些实现可以依赖于模板形参的性质特点。特化的目的是实现优化或减少代码膨胀。特化有以下两种形式。

- **部分特化(partial specialization)**：只为部分模板形参提供替代实现。
- **(显式)完全特化[(explicit)full specialization]**：为所有模板形参提供特化实现。

编译器通过将模板定义中的模板形参替换成实参，从而自模板生成代码的过程称为**模板实例化(template instantiation)**。例如，在使用 `vector<int>` 的例子中，编译器会在 `T` 出现的每个地方都将它替换为 `int` 类型。

模板实例化有以下两种形式。

- **隐式实例化(implicit instantiation)**：编译器由于代码中用到了模板而对其进行实例化。只会为实际使用到的组合或者实参进行实例化。例如，如果编译器遇到 `vector<int>`和 `vector<double>` 用在了代码中，它会分别为 `int` 和 `double` 类型实例化 `vector` 类模板，而不会实例化其他类型。
- **显式实例化(explicit instantiation)**：这种方式要显式地告诉编译器需要实例化哪些模板，即便这些实例化在代码中没有显式使用。这在创建库文件时很有用，因为未实例化的模板不会被放入目标文件中。它们还可以帮助减少编译时间和目标文件大小，我们将在后面的章节中见到这种方式。

本节提到的所有术语和概念将在本书的其他章节中详细介绍。本节旨在为模板术语提供一个简短的参考指南。请记住，还有许多其他与模板相关的术语将在适当的时候引入。

1.4 模板的简史

模板元编程是泛型编程的 C++实现。这种编程范式的探索始于 20 世纪 70 年代，最早支持它的主要语言是 20 世纪 80 年代前半叶的 Ada 和 Eiffel。David Musser 和 Alexander Stepanov 在 1989 年的论文《泛型编程》中定义了这一范式。

“泛型编程的核心思想是从具体、高效的算法抽象出泛型的算法，它们可以与不同的数据表示相结合来产生种类繁多的有用的软件。”

这就定义了一种编程范式，其中算法基于以后指定的类型来定义，并根据使用情况实例化。

模板并不是由 Bjarne Stroustrup 开发的**带类的 C 语言(C with Classes)**的初始组成部分。Stroustrup 在 1986 年首次发表了描述 C++模板的论文，这是在《C++编程语言(第Ⅱ版)》出版之后的一年。模板在 1990 年就成为 C++语言的一部分，那还是在 ANSI 和 ISO C++标准委员会成立之前。

在 20 世纪 90 年代初，Alexander Stepanov、David Musser 和 Meng Lee 尝试在 C++中实现各种泛型概念。这促成了**标准模板库(STL)**的首次实现。当 ANSI/ISO 委员会在 1994 年了解到这个库时，他们迅速将其添加到了起草的规范中。STL 与 C++语言一起于 1998 年标准化，称之为 C++98。

C++标准的较新版本，统称为**现代 C++**，引入了对模板元编程的各种改进，如表 1-1 所示。

表 1-1　现代 C++的特性和描述

版本	特性	描述
C++11	变参模板	模板可以有可变数目的模板形参
	模板别名	借助 `using` 声明来定义模板的同义词
	外部模板	告诉编译器在某个翻译单元中不要实例化某个模板
	类型特征	新的头文件`<type_traits>` 包含了标准的类型特征，用于标识对象的分类和类型的特点
C++14	变量模板	支持定义变量或者静态数据成员为模板
C++17	折叠表达式	以二元运算符规约变参模板的形参包
	模板形参中的 `typename`	在模板形参中可以用 `typename` 关键字取代 class
	非类型模板形参支持 `auto`	`auto` 关键字可以用于非类型模板形参
	类模板实参推导	编译器从对象初始化的方式推断出模板形参的类型
C++20	模板 lambda 表达式	lambda 表达式也和一般函数一样，可以作为模板
	字符串字面量作为模板形参	字符串字面量可以作为非类型模板的实参，以及用户定义字面量运算符的新形式
	约束	在模板实参中定义需求
	概念	具名约束的集合

所有这些特性以及模板元编程的其他方面构成了本书的主要内容，这些内容将在接

下来的章节中详细展开。现在让我们看看使用模板的优缺点是什么。

1.5 模板的优缺点

在开始使用模板之前，重要的一步是了解使用它们的优点以及可能带来的缺点。

首先是优点：

- 模板可以帮助我们避免编写重复的代码。
- 模板有利于创建提供算法和类型的泛型库，如标准 C++库(有时被错误地称为 STL)，这些库可以在许多应用程序中使用，不拘类型。
- 使用模板可以产生更少且更好的代码。例如使用标准库中的算法可以帮助编写更短小的代码，这些代码往往更易于理解和维护，此外，由于这些算法在开发和测试中投入了大量精力，它们通常会更加健壮。

谈到缺点，以下几点值得一提。

- 语法被认为很复杂而笨拙，尽管稍加练习就不会真正成为开发和使用模板的障碍。
- 与模板相关的编译错误常常长且晦涩，很难找出错误的原因。较新版本的 C++编译器在简化这类错误方面有所进步，但通常仍然是一个重要的问题。C++20 标准中引入的概念(concepts)也被看作一种努力，旨在提供更好的编译错误诊断。
- 它们会增加编译时间，因为模板完全在头文件中实现。每当对模板进行更改时，包含该头文件的所有翻译单元都必须重新编译。
- 模板库作为一个或多个头文件的集合提供，必须与使用它们的代码一起编译。
- 模板在头文件中实现的另一个缺点是缺乏信息隐藏。整个模板代码都在头文件中可读。库开发者通常会使用诸如 detail 或 details 等名空间来包含应放在库内部的代码，它们不应该被使用库的人直接调用。
- 由于未使用的代码不会被编译器实例化，这些代码就可能更难验证。因此，在编写单元测试时，务必确保良好的代码覆盖率。对于库尤其如此。

尽管缺点列表看起来更长，但使用模板并非坏事，也不应该被回避。相反，模板是 C++语言的一个强大特性。模板并不总是被正确理解，有时也会被误用或滥用。但是，请审慎地使用，模板确实有不可否认的优点。本书将试图提供对模板及其使用的更好的理解。

1.6 总结

本章介绍了 C++编程语言中的模板概念。

我们首先学习了用模板要解决的问题，然后我们看到了函数模板、类模板和变量模板的简单示例。我们介绍了模板的基本术语，将在接下来的章节中进一步讨论。在本章的末尾，简要回顾了 C++语言中模板的历史。最后讨论了使用模板的优缺点。所有这些主题都将帮助我们更好地理解下一章的内容。

在下一章，我们将探讨 C++中模板的基础。

1.7　问题

1. 我们为什么需要模板？模板有什么优势？
2. 如何称呼模板化的函数？如何称呼模板化的类？
3. 有多少种模板形参？它们分别是什么？
4. 什么是部分特化？什么是完全特化？
5. 使用模板的主要缺点有哪些？

第2章 模板的基础

在上一章中我们简要介绍了模板，其中包括模板的概念及用处，使用模板的利弊，以及一些函数模板和类模板的示例。在本章中，我们将深入探索这个领域，并了解诸如模板形参、实例化、特化、别名等方面的内容。

在本章中，我们将讨论以下主题：

- 如何定义函数模板、类模板、变量模板和别名模板
- 有哪几种模板形参
- 什么是模板实例化
- 什么是模板特化
- 如何使用泛型 lambda 和 lambda 模板

在本章结束时，你将熟悉 C++中模板的核心基础知识，并能够理解大段模板代码，也能自己编写模板。

下面将探讨定义和使用函数模板的细节。

2.1 定义函数模板

函数模板的定义方式与普通函数类似，只是在函数声明之前加上了关键字 template，然后在紧跟的尖括号中列出模板形参。下面是一个函数模板的简单示例：

```
template <typename T>
T add(T const a, T const b)
{
   return a + b;
}
```

本例中的函数有两个形参 a 和 b，它们均为相同的 T 类型。该类型列在模板形参列

表中，通过关键字 typename 或 class 引入(本例和本书中使用的是前者)。这个函数只是将两个实参相加并返回操作的结果，并且操作结果应该具有相同的 T 类型。

函数模板只是创建实际函数的蓝图，并只存在于源代码中。除非在源代码中显式调用，否则函数模板不会出现在编译后的可执行文件中。但是，当编译器遇到对函数模板的调用，并能将提供的实参及其类型与函数模板的形参相匹配时，编译器就会根据模板和用于调用模板的实参生成一个实际的函数。为了理解这一点，我们来看几个示例：

```
auto a = add(42, 21);
```

在这个代码片段中，我们使用两个 int 形参 42 和 21 来调用 add 函数。编译器可以根据所提供的实参类型推导出模板形参 T，因此不必显式地提供它。但是，以下两次调用也是可能的，而且实际上与之前的调用完全相同。

```
auto a = add<int>(42, 21);
auto a = add<>(42, 21);
```

通过这次调用，编译器将生成以下函数(请记住对于不同的编译器，实际代码可能会有所不同)。

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

但是，如果我们将调用更改为下面的形式，就可以显式地为模板形参 T 提供实参，即 short 类型。

```
auto b = add<short>(42, 21);
```

在这种情况下，编译器将生成该函数的另一个实例，并使用 short 代替 int。新的实例如下所示：

```
short add(const short a, const int b)
{
   return static_cast<short>(a + b);
}
```

如果这两个形参的类型不明确，则编译器将无法自动推导出它们。以下调用就属于这种情况：

```
auto d = add(41.0, 21);
```

本例中，41.0 是 double 类型，而 21 是 int 类型。add 函数模板有两个相同类型的形参，因此编译器无法将其与所提供的实参进行匹配，将会报错。为了避免这种情况，假设你预料到会将编译器实例化为 double，那么就必须显式地指定类型，代码如下。

```
auto d = add<double>(41.0, 21);
```

只要两个实参的类型相同，且实参类型中的“+”运算符可用，则可以按照前面所

示的方式调用函数模板 add。但是，如果“+”运算符不可用，那么即使模板形参已正确解析，编译器也无法生成实例。下面的代码段就说明了这一点。代码如下：

```
class foo
{
   int value;
public:
   explicit foo(int const i):value(i)
   { }

   explicit operator int() const { return value; }
};

auto f = add(foo(42), foo(41));
```

在这种情况下，编译器将会因为没有为 foo 类型的实参找到二元“+”运算符而报错。当然，对于不同的编译器，给出的实际信息是不同的，所有错误都是如此。为了能够对 foo 类型的实参调用函数 add，必须重载该类型的“+”运算符。可能的实现方法如下。

```
foo operator+(foo const a, foo const b)
{
   return foo((int)a + (int)b);
}
```

到目前为止，我们看到的所有示例都只是具有单个模板形参的模板。但是，模板可以有任意数量的形参，甚至数量可变的形参。数量可变形参将在第 3 章“变参模板”中讨论。下面这个函数是一个包含两个类型模板形参的函数模板：

```
template <typename Input, typename Predicate>
int count_if(Input start, Input end, Predicate p)
{
   int total = 0;
   for (Input i = start; i != end; i++)
   {
      if (p(*i))
         total++;
   }
   return total;
}
```

此函数接收两个输入迭代器，分别指向范围的开头和结尾，以及一个谓词，并返回范围中与谓词匹配的元素个数。至少在概念上，这个函数与标准库中头文件 <algorithm> 中的 std::count_if 通用函数非常相似，因此，你应该始终优先使用标准算法而不是手工实现。但是，就本主题而言，这个函数是一个很好的示例，可以帮助理解模板的工作原理。

可以使用 count_if 函数，如下所示。

```
int main()
{
```

```
    int arr[]{ 1,1,2,3,5,8,11 };
    int odds = count_if(
                  std::begin(arr), std::end(arr),
                  [](int const n) { return n % 2 == 1; });
    std::cout << odds << '\n';
}
```

同样，不必显式指定类型模板形参的实参(输入迭代器的类型和一元谓词的类型)，因为编译器能够从调用中推导出这些参数。

尽管关于函数模板还有很多内容需要学习，但本节已经介绍了如何使用它们。现在让我们学习定义类模板的基础知识。

2.2 定义类模板

类模板的声明方式非常相似，都是在类声明之前加上关键字 template 和模板形参列表。下面的代码片段展示了一个名为 wrapper 的类模板。它只有单个模板形参(名为 T 的类型)，用作数据成员、形参和函数返回类型的类型。

```
template <typename T>
class wrapper
{
public:
    wrapper(T const v): value(v)
    { }

    T const& get() const { return value; }
private:
    T value;
};
```

如果源代码中没有使用该类模板，那么编译器就不会从中生成代码。要做到这一点，必须要实例化类模板，并由用户显式或编译器隐式地将其所有形参与实参正确匹配。下面是实例化该类模板的示例：

```
wrapper a(42);              // 包装一个 int
wrapper<int> b(42);         // 包装一个 int
wrapper<short> c(42);       // 包装一个 short
wrapper<double> d(42.0);    // 包装一个 double
wrapper e("42");            // 包装一个 char const *
```

由于类模板实参推导(Class Template Argument Deduction，CTAD)的功能的存在，该代码段中的 a 和 e 的定义仅在 C++17 及以后的版本中有效。只要编译器能够推导出所有模板实参，我们就可以在不指定任何模板实参的情况下使用该类模板。这将在第 4 章“高级模板概念”中讨论。在此之前，所有引用类模板的示例都会显式列出实参，如 wrapper<int>或 wrapper<char const*>。

类模板可以在不定义的情况下声明，并在允许不完整类型的语境中使用，例如函数的声明，示例如下。

```
template <typename T>
class wrapper;

void use_foo(wrapper<int>* ptr);
```

但是，类模板必须在模板实例化发生处定义，否则编译器将会报错。下面的代码段展示了这一点：

```
template <typename T>
class wrapper;                          // 正确

void use_wrapper(wrapper<int>* ptr); // 正确

int main()
{
   wrapper<int> a(42);                 // 错误，不完整类型
   use_wrapper(&a);
}

template <typename T>
class wrapper
{
   // 模板定义
};

void use_wrapper(wrapper<int>* ptr)
{
   std::cout << ptr->get() << '\n';
}
```

在声明 use_wrapper 函数时，仅仅只声明了类模板 wrapper 而没有定义它。但是，在这种情况下允许使用不完整类型，因而此时使用 wrapper<T>是正确的。然而，在 main 函数中我们尝试实例化 wrapper 类模板的一个对象，这会导致编译器报错，因为此时类模板的定义必须可用(而这里尚未得到类模板的完整定义)。要修复这个特殊的例子，我们必须把 main 函数的定义搬移至(源码)末尾，即搬移到 wrapper 类与 use_wrapper 函数的完整定义之后。

本例中的类模板使用关键字 class 定义，但是在 C++中，使用关键字 class 或关键字 struct 来声明类的差异微乎其微。

- 用 struct 时，默认的成员访问是 public，而使用 class 时则是 private。
- 用 struct 时，基类继承的默认访问说明符是 public，而使用 class 时则是 private。

使用关键字 struct 定义类模板的方法与之前使用关键字 class 定义类模板的方式完全相同。使用关键字 struct 或 class 定义的类之间的区别，同样适用于使用关键字 struct 或 class 定义的类模板。

无论某类是否为模板，它也同样能包含成员函数模板。下一节将讨论如何定义它们。

2.3　定义成员函数模板

到目前为止，我们已经学习了函数模板和类模板。在本节中，我们还将学习如何在非模板类或者类模板中定义成员函数模板。想要理解其中的差异，让我们从以下示例开始。

```
template <typename T>
class composition
{
public:
   T add(T const a, T const b)
   {
      return a + b;
   }
};
```

composition 类是一个类模板，它只包含一个使用类型形参 T 的成员函数 add。该类可以通过如下方式使用：

```
composition<int> c;
c.add(41, 21);
```

在本例中，首先需要实例化 composition 类的一个对象，注意必须显式指定类型形参 T 的实参，因为编译器无法自行确定它(这里没有可供编译器推导的语境)，但当我们调用函数 add 时，只需要提供实参即可。因为其类型已知(由之前解析为 int 的 T 类型模板形参表示)。像 c.add<int>(41, 21)这样的调用将触发编译错误，因为函数 add 不是模板函数，而是作为 composition 类的普通成员函数。

下面的例子中，composition 类的变化不大，但影响显著。让我们先看看其定义：

```
class composition
{
public:
   template <typename T>
   T add(T const a, T const b)
   {
      return a + b;
   }
};
```

这次，类 composition 不再是模板类。但是，其 add 函数却是一个函数模板。因此，为了调用这个函数，我们必须执行以下操作。

```
composition c;
c.add<int>(41, 21);
```

把类型模板形参 T 显式指定为 int 是多余的，因为编译器可以从调用的实参自己推

导出来。不过，为了更好地理解这两种实现之间的差异，我们在这里进行了说明。

除了类模板的成员函数与类的成员函数模板这两种情况外，我们还可以使用类模板的成员函数模板。但在这种情况下，成员函数模板的模板形参必须与类模板的模板形参不同；否则编译器将会报错。让我们回到类模板 wrapper 的例子，并对其进行如下修改。

```
template <typename T>
class wrapper
{
public:
   wrapper(T const v) :value(v)
   {}

   T const& get() const { return value; }

   template <typename U>
   U as() const
   {
      return static_cast<U>(value);
   }
private:
   T value;
};
```

这个实现引入了函数 as 这个新成员。这是一个函数模板，有一个名为 U 的类型模板形参。该函数用于把封装的值从类型 T 转化为类型 U，并把其返回给调用者。我们可以如下使用这个实现。

```
wrapper<double> a(42.0);
auto d = a.get();       // double
auto n = a.as<int>();   // int
```

在实例化 wrapper 类(double)(尽管在 C++17 中这是多余的)和调用 as 函数(int)执行强制类型转换时都指定了模板形参的实参。

在继续学习实例化、特化和其他形式的模板(包括变量模板和别名模板)等其他主题前，我们有必要花点时间更进一步了解模板形参的知识。而这正是下一节的主题。

2.4　理解模板形参

到目前为止，我们在书中已经见过不少带有一个或多个形参的模板示例。在所有这些示例中，形参代表了在实例化时所提供的具体类型，这些类型既可以是用户显式提供的，也可以是编译器在推导时隐式提供的。这类参数称为类型模板形参(**type template parameter**)。此外，模板还可以有非类型模板形参(**non-type template parameter**)和模板模板形参(**template template parameter**)，在接下来的章节中，我们将对这些参数进行深入探讨。

2.4.1 类型模板形参

如前所述，这些参数代表在模板实例化过程中作为实参提供的具体类型。它们可以使用关键字 typename 或关键字 class 引入。使用这两个关键字没有区别。类型模板形参可以带有默认值，而这个默认值同样是某种类型。指定默认值的方式与指定函数形参的默认值的方式完全相同。示例如下：

```
template <typename T>
class wrapper { /* ... */ };

template <typename T = int>
class wrapper { /* ... */ };
```

还可以省略类型模板形参的名称，这在前置声明(forward declaration)中非常有用。

```
template <typename>
class wrapper;

template <typename = int>
class wrapper;
```

C++11 引入了变参模板，即实参数量可变的模板。接受零个或多个实参的模板形参称为形参包(**parameter pack**)。类型模板形参包(**type template parameter pack**)具有以下形式：

```
template <typename... T>
class wrapper { /* ... */ };
```

变参模板将在第 3 章“变参模板”中进行讨论，因此，本节不作过多深入介绍。

C++20 引入了概念(**concept**)和约束(**constraint**)。约束对于模板实参提出了要求，而一个命名的约束集合称为概念。概念可以用作类型模板形参，但语法略有不同。我们使用概念的名称(如果需要，还要跟随一个用尖括号括起来的模板实参列表)代替关键字 typename 或关键字 class。下面展示一些示例，其中包括具有默认值的概念和受约束的类型模板形参包。

```
template <WrappableType T>
class wrapper { /* ... */ };

template <WrappableType T = int>
class wrapper { /* ... */ };

template <WrappableType... T>
class wrapper { /* ... */ };
```

概念和约束将在第 6 章“概念和约束”中进行讨论，届时我们将更进一步了解这些类型的参数。现在，介绍第二种模板形参，即非类型模板形参。

2.4.2　非类型模板形参

模板实参并不总是必须表示某种类型，它们也可以是编译期表达式，例如常量、函数地址、具有外部链接的函数或对象的地址，或者静态类成员的地址等。使用编译期表达式提供的形参称为非类型模板形参(**non-type template parameter**)。这类形参只能有一种结构化类型(**structural type**)。结构化类型如下：

- 整数类型
- C++20 中的浮点类型
- 枚举类型
- 指针类型(指向对象或函数)
- 成员类型的指针(指向成员对象或成员函数)
- 左值引用类型(指向对象或函数)
- 满足下列要求的字面类：
  - 所有基类都是公有且不可变(non-mutable)的
  - 所有非静态数据成员都是公有且不可变的
  - 所有基类和非静态数据成员的类型都是结构化类型或其数组

这些类型的 cv 限定形式也可用于非类型模板形参。

可以用不同的方式指定非类型模板形参。代码如下：

```
template <int V>
class foo { /*...*/ };

template <int V = 42>
class foo { /*...*/ };

template <int... V>
class foo { /*...*/ };
```

在这些示例中，非类型模板形参的类型都是 int。前两个例子很相似，只不过在第二个例子中使用了默认值。但第三个例子明显不同，因为该形参实际上是一个形参包，这将在下一章中进行讨论。

为了更好地理解非类型模板形参，让我们看一看下面的示例。在这个示例中，我们创建了一个名为 buffer 的固定大小的数组类。

```
template <typename T, size_t S>
class buffer
{
   T data_[S];
public:
   constexpr T const * data() const { return data_; }

   constexpr T& operator[](size_t const index)
   {
      return data_[index];
```

```
    }

    constexpr T const & operator[](size_t const index) const
    {
        return data_[index];
    }
};
```

该 buffer 类具有一个包含 S 个 T 类型元素的内部数组，因此，S 必须是一个可在编译期确定的值。这个类可以用以下方式实例化：

```
buffer<int, 10> b1;
buffer<int, 2*5> b2;
```

这两个定义是等价的，且 b1 和 b2 同为存放 10 个整数的 buffer。事实上二者是同一种类型，因为 2*5 与 10 这两个表达式在编译期可求得相同的值，可以通过下面的语句简单验证这一点。

```
static_assert(std::is_same_v<decltype(b1), decltype(b2)>);
```

对于下面的 b3 而言，情况却有所不同。

```
buffer<int, 3*5> b3;
```

本例中的 b3 是一个包含 15 个整数的 buffer，这与前一个示例中持有 10 个整数的 buffer 不同。从概念上说，编译器会生成以下代码：

```
template <typename T, size_t S>
class buffer
{
    T data_[S];
public:
    constexpr T* data() const { return data_; }

    constexpr T& operator[](size_t const index)
    {
        return data_[index];
    }

    constexpr T const & operator[](size_t const index) const
    {
        return data_[index];
    }
};
```

这是主模板的代码，接下来是几个特化版本：

```
template<>
class buffer<int, 10>
{
  int data_[10];

public:
  constexpr int * data() const;
  constexpr int & operator[](const size_t index);
```

```
    constexpr const int & operator[](
      const size_t index) const;
};

template<>
class buffer<int, 15>
{
  int data_[15];
public:
  constexpr int * data() const;
  constexpr int & operator[](const size_t index);
  constexpr const int & operator[](
    const size_t index) const;
};
```

上述这些示例中展示了模板特化的概念，将在本章的 2.6 节“理解模板特化”中进一步详细讨论。目前，你应该注意到两种 buffer 是不同的类型。同样，我们可以通过下面的语句验证 b1 和 b3 的类型是否不同。

```
static_assert(!std::is_same_v<decltype(b1), decltype(b3)>);
```

在实际项目中，诸如整型、浮点型和枚举类型等结构化类型的使用要比其他类型更为常见。这样相对更容易理解并找到相关有帮助的示例。但是，也同样存在使用指针或引用的场景。在接下来的例子中我们将探讨使用指向函数形参的指针，代码如下。

```
struct device
{
   virtual void output() = 0;
   virtual ~device() {}
};

template <void (*action)()>
struct smart_device : device
{
   void output() override
   {
      (*action)();
   }
};
```

在本例中，device 是一个带有纯虚函数 output(和虚析构函数)的基类，它也是类模板 smart_device 的基类。该类模板通过函数指针调用某个函数并以此实现了虚函数 output。该函数指针作为类模板的非类型模板形参的实参传入。下面的代码展示了其用法：

```
void say_hello_in_english()
{
   std::cout << "Hello, world!\n";
}

void say_hello_in_spanish()
{
```

```
    std::cout << "Hola mundo!\n";
}

auto w1 =
    std::make_unique<smart_device<&say_hello_in_english>>();
w1->output();

auto w2 =
    std::make_unique<smart_device<&say_hello_in_spanish>>();
w2->output();
```

本例中的 w1 和 w2 是两个 unique_ptr 对象。虽然表面上看它们指向相同类型的对象，但事实并非如此，因为 smart_device<&say_hello_in_english>和 smart_device<&say_hello_in_spanish>是不同的类型，它们在实例化时使用了不同的函数指针值。利用下面的语句可以轻松地验证这一点：

```
static_assert(!std::is_same_v<decltype(w1), decltype(w2)>);
```

另一方面，如果我们像下面的代码那样用 std::unique_ptr<device>替代 auto 说明符，那么 w1 和 w2 现在都是指向基类 device 的智能指针，因而具有相同的类型。

```
std::unique_ptr<device> w1 =
    std::make_unique<smart_device<&say_hello_in_english>>();
w1->output();

std::unique_ptr<device> w2 =
    std::make_unique<smart_device<&say_hello_in_spanish>>();
w2->output();

static_assert(std::is_same_v<decltype(w1), decltype(w2)>);
```

虽然本例中使用的是函数指针，但类似的例子也可用于指向成员函数的指针。可以将前面的示例代码改写为如下所示(仍使用相同的基类 device)。

```
template <typename Command, void (Command::*action)()>
struct smart_device : device
{
    smart_device(Command& command) : cmd(command) {}

    void output() override
    {
       (cmd.*action)();
    }
private:
    Command& cmd;
};

struct hello_command
{
    void say_hello_in_english()
    {
       std::cout << "Hello, world!\n";
    }
```

```
    void say_hello_in_spanish()
    {
      std::cout << "Hola mundo!\n";
    }
};
```

这些类的使用方法如下所示。

```
hello_command cmd;

auto w1 = std::make_unique<
    smart_device<hello_command,
       &hello_command::say_hello_in_english>>(cmd);
w1->output();

auto w2 = std::make_unique<
    smart_device<hello_command,
      &hello_command::say_hello_in_spanish>>(cmd);
w2->output();
```

C++17 中引入了一种指定非类型模板形参的新形式，即使用 auto 说明符(包括 auto* 和 auto&形式)或 decltype(auto)代替类型名称。这样，编译器就能从实参提供的表达式中推导出模板形参的类型，而如果推导出的类型不适用于模板形参，则编译器将报错。示例如下：

```
template <auto x>
struct foo
{ /* … */ };
```

这个类模板使用的方法如下所示。

```
foo<42>     f1;     // foo<int>
foo<42.0>   f2;    // C++20 会解析为 foo<double>，老版本则报错
foo<"42">   f3;    // 错误
```

在第一个例子中，对于 f1，编译器推导出的实参类型为 int；在第二个例子中，对于 f2，编译器推导出的实参类型为 double，但这只是 C++20 的情况。在 C++20 之前的标准中，这一行代码将会产生错误，因为 double 型在 C++20 之前的标准中不能用作非类型模板形参；而最后关于 f3 的例子会产生错误，因为“42”是一个字符串字面量，而字符串字面量不能用作非类型模板形参的实参。

不过在 C++20 中，最后一个示例中的问题可以通过将字面量的字符串封装为一个结构化的字面量类的方法来解决。该类使用一个固定长度的数组来保存字符串字面量的每个字符，代码如下。

```
template<size_t N>
struct string_literal
{
    constexpr string_literal(const char(&str)[N])
    {
      std::copy_n(str, N, value);
```

```
    }

    char value[N];
};
```

同时还需要修改之前示例中展示过的 foo 类模板，其方法是显式使用 string_literal 而不是 auto 说明符。

```
template <string_literal x>
struct foo
{
};
```

这样，之前示例中的 foo<"42"> f3; 声明在C++20中编译时就可以正常通过编译了。

auto 说明符也可能用于非类型模板形参包。在这种情况下，每个模板实参的类型都可以相互独立地推导出来。模板实参的类型不必相同。下面的代码段展示了这种用法：

```
template<auto... x>
struct foo
{ /* ... */ };

foo<42, 42.0, false, 'x'> f;
```

在本例中编译器会把模板实参的类型依次推导为 int、double、bool 和 char。

第三种也是最后一种模板形参是**模板模板形参**，我们接下来将讨论它。

2.4.3　模板模板形参

虽然“模板模板形参”的名称听起来有点奇怪，但它指的是一类本身就是模板的模板形参。指定这些参数的方法与类型模板形参一样，可以带名称，也可以不带名称；可以带默认值，也可以不带默认值；可以作为带名称的形参包，也可以作为不带名称的形参包。从 C++17 开始，关键字 class 和 typename 都可以用来引入模板模板形参。但在此版本之前，只能使用关键字 class。

为了展示模板模板形参的用法，首先考虑以下两个类模板。

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

template <typename T>
class fancy_wrapper
{
public:
    fancy_wrapper(T const v) :value(v)
    {
    }
```

```
    T const& get() const { return value; }

    template <typename U>
    U as() const
    {
        return static_cast<U>(value);
    }
private:
    T value;
};
```

simple_wrapper 类是一个非常简单的类模板，它持有一个类型模板形参 T 的值。另一方面，fancy_wrapper 则是一个更复杂的 wrapper 实现，它隐藏了封装的值，并提供了一些用于数据访问的成员函数。接下来，我们要实现类模板 wrapping_pair，它包含两个封装类型的值，这个类型可以是 simpler_wrapper，也可以是 fancy_wrapper 或其他任何相似类型。

```
template <typename T, typename U,
          template<typename> typename W = fancy_wrapper>
class wrapping_pair
{
public:
    wrapping_pair(T const a, U const b) :
        item1(a), item2(b)
    {
    }

    W<T> item1;
    W<U> item2;
};
```

类模板 wrapping_pair 有 3 个参数，前两个是类型模板形参 T 和 U，第三个形参 W 则是模板模板形参，W 带有默认值，其类型默认为 fancy_wrapper 类型。可以按照如下方式使用这个类模板。

```
wrapping_pair<int, double> p1(42, 42.0);
std::cout << p1.item1.get() << ' '
          << p1.item2.get() << '\n';

wrapping_pair<int, double, simple_wrapper> p2(42, 42.0);
std::cout << p2.item1.value << ' '
          << p2.item2.value << '\n';
```

本例中，p1 是一个 wrapping_pair 对象，该对象包含两个值(一个 int 和一个 double)，每个值都封装在一个 fancy_wrapper 对象中。这不是显式指定的，却是模板模板形参的默认值。另一方面，p2 也是一个 wrapping_pair 对象，它同样包含一个 int 和一个 double，但这些值都封装在 simple_wrapper 对象中，这是在模板实例化中显式指定的。

我们在这个例子中看到了为模板形参设定默认模板实参的方式，下一小节将会详细地讨论这个主题。

2.4.4　默认模板实参

默认模板实参的指定方式与默认函数实参类似，都是在形参列表中等号之后指定。以下规则适用于默认模板实参。

- 它们可以与任何类型的模板形参一起使用，形参包除外。
- 如果为类模板、变量模板或类型别名的模板形参指定了默认值，那么形参列表中所有其后续的模板形参也都必须有默认值。如果是模板形参包，那么最后一个形参例外。
- 如果在函数模板中为模板形参指定了默认值，那么形参列表中所有其后续的模板形参不强制要求提供默认值。
- 在函数模板中，只有在形参包具有默认实参或编译器可以从函数实参中推导出其值时，才可以在形参包后面添加更多的类型形参。
- 友元类模板的声明中不允许使用形参包。
- 只有当友元函数模板的声明也是定义，并且在同一翻译单元中没有其他该函数声明的情况下，才允许在友元函数模板的声明中使用它们。
- 不允许在函数模板或成员函数模板的显式特化的声明或定义中使用它们。

下面的代码展示了默认模板实参的用法：

```
template <typename T = int>
class foo { /*...*/ };

template <typename T = int, typename U = double>
class bar { /*...*/ };
```

如前文所述，在声明类模板时带有默认实参的模板形参之后不能跟着没有默认值的形参，但这一限制条件不适用于函数模板，正如下方代码段所示。

```
template <typename T = int, typename U>
class bar { };  // 错误

template <typename T = int, typename U>
void func() {}  // 正确
```

一个模板可以有多个声明(但只能有一个定义)。所有声明和定义中的默认模板实参都会被合并(合并方法与合并默认函数实参相同)。让我们通过下面这个示例理解它是如何工作的。

```
template <typename T, typename U = double>
struct foo;

template <typename T = int, typename U>
struct foo;

template <typename T, typename U>
struct foo
```

```
{
   T a;
   U b;
};
```

这在语义上等同于下面的定义：

```
template <typename T = int, typename U = double>
struct foo
{
   T a;
   U b;
};
```

但是，这些带有不同默认模板实参的多重声明不能用任意顺序提供。毕竟前面提到的那些规则依然适用。因此，如果类模板的一个声明中第一个形参带有一个默认实参，而后面的形参没有默认实参，那么这个类模板的声明就是非法的。

```
template <typename T = int, typename U>
struct foo;   // 错误! 形参 U 没有默认实参

template <typename T, typename U = double>
struct foo;
```

默认模板实参的另一个限制是：同一模板形参不能在相同作用域中被赋予多个默认值。因此，下面的代码是错误的。

```
template <typename T = int>
struct foo;

template <typename T = int> // 错误! 重复指定默认参数
struct foo {};
```

当默认模板实参使用了类中的名称时，编译器会在声明时检查其成员访问限制，而不是在模板实例化时进行检查。

```
template <typename T>
struct foo
{
protected:
   using value_type = T;
};

template <typename T, typename U = typename T::value_type>
struct bar
{
   using value_type = U;
};

bar<foo<int>> x;
```

在定义变量 x 时，bar 类模板会被实例化，但由于 foo::value_type 的访问修饰符是 protected，因此不能在 foo 外部使用它，结果是编译器在声明 bar 类模板时出错。

综上所述，我们结束了模板形参的主题。下一节我们将探讨的是模板实例化，即根据模板定义和一组模板实参来创建函数、类或变量的新定义。

2.5 理解模板实例化

正如之前提到的，模板只是一张蓝图，编译器在使用模板时才会根据它们创建实际的代码。根据模板声明生成函数、类或变量定义的行为称为模板实例化**(template instantiation)**。模板实例化可以是显式**(explicit)**的，即主动告诉编译器何时生成定义；也可以是隐式**(implicit)**的，即编译器根据需要自动生成新的定义。我们将会在接下来的章节中详细介绍这两种形式。

2.5.1 隐式实例化

隐式实例化发生在编译器根据模板的使用生成定义，而没有显式实例化的情况下。隐式实例化的模板在与模板相同的名空间中定义。但是，编译器从模板创建定义的方式可能会有所不同。我们将在下面的示例中看到这一点。看看下面这段代码：

```
template <typename T>
struct foo
{
  void f() {}
};

int main()
{
  foo<int> x;
}
```

本例中我们有一个名为 foo 的类模板，其中包含了一个成员函数 f。在 main 函数中，我们定义了一个 foo<int>类型的变量，但没有使用它的任何成员。由于使用了 foo，编译器会隐式地为 int 类型定义 foo 的一个特化。如果你利用运行了 Clang 的 cppinsights.io，你将会看到如下代码。

```
template<>
struct foo<int>
{
  inline void f();
};
```

因为我们的代码中没有调用函数 f，所以它只是被声明却没有被定义。而如果我们在 main 函数中添加一个对函数 f 的调用，那么特化将如下所示。

```
template<>
struct foo<int>
{
```

```
  inline void f() { }
};
```

但是，如果再添加一个包含错误的实现函数 g，那么在不同的编译器中将会出现不同的行为。

```
template <typename T>
struct foo
{
  void f() {}
  void g() {int a = "42";}
};

int main()
{
  foo<int> x;
  x.f();
}
```

g 的函数体中包含一个错误(也可以用 static_assert(false)语句作为替代)。VC++能够正常编译这段代码，但 Clang 和 GCC 却编译失败。这是因为只要代码的语法正确，VC++就会忽略模板中未使用的部分，而其他编译器则会在模板实例化前执行语义检查。

对于函数模板，隐式实例化发生在用户代码在需要函数定义存在的语境中使用函数时。对于类模板，隐式实例化发生在当用户代码在上下文中引用了一个需要完整类型模板，或者当类型的完整性会影响代码时。这类语境的典型例子就是构造该类型的对象时。然而，在声明指向类模板的指针时，情况却并非如此。为了理解其工作原理，让我们看看下面的示例。

```
template <typename T>
struct foo
{
  void f() {}
  void g() {}
};

int main()
{
  foo<int>* p;
  foo<int> x;
  foo<double>* q;
}
```

在这段代码中，使用了与前面示例相同的 foo 类模板，并声明了几个变量：p 是 foo<int> 类型的指针，x 是 foo<int> 类型的对象，q 变量是指向 foo<double> 类型的指针。本例中由于声明了 x，此时编译器只需要实例化 foo<int>。现在，让我们考虑如何调用成员函数 f 和 g。

```
int main()
{
  foo<int>* p;
```

```
    foo<int> x;
    foo<double>* q;

    x.f();
    q->g();
}
```

对于这些更改，编译器需要实例化以下内容。

- 在声明变量 x 时实例化 foo<int>
- 在调用 x.f()时实例化 foo<int>::f()
- 在调用 q->g()时实例化 foo<double>和 foo<double>::g()

另一方面，编译器不需要在声明指针 p 时实例化 foo<int>，也不需要在声明指针 q 时实例化 foo<double>。但是，当类模板特化涉及指针转换时，编译器确实需要隐式实例化类模板。代码如下：

```
template <typename T>
struct control
{};

template <typename T>
struct button : public control<T>
{};

void show(button<int>* ptr)
{
   control<int>* c = ptr;
}
```

在函数 show 中发生了从 button<int>* 到 control<int>* 的转换。因此，此时编译器必须实例化 button<int>。

当类模板包含静态成员时，这些成员在编译器隐式实例化类模板时不会被隐式实例化，而只会在编译器确实需要它们的定义时才会实例化。另一方面，类模板的每个特化都有自己独立的静态成员的一个副本，正如下面的代码所示。

```
template <typename T>
struct foo
{
   static T data;
};

template <typename T> T foo<T>::data = 0;

int main()
{
   foo<int> a;
   foo<double> b;
   foo<double> c;

   std::cout << a.data << '\n'; // 0
   std::cout << b.data << '\n'; // 0
```

```
    std::cout << c.data << '\n'; // 0

    b.data = 42;
    std::cout << a.data << '\n'; // 0
    std::cout << b.data << '\n'; // 42
    std::cout << c.data << '\n'; // 42
}
```

类模板 foo 有一个静态成员变量 data，在 foo 定义之后初始化该变量。在 main 函数中，我们将变量 a 声明为 foo<int>的对象，变量 b 和 c 则声明为 foo<double> 的对象。最初，这些变量的 data 成员字段都被初始化为 0。但是，由于变量 b 与 c 共享相同的 data 副本，因此，在执行操作 b.data = 42 之后，a.data 仍旧是 0，而 b.data 和 c.data 却都是 42。

在介绍了隐式实例化的工作原理后，是时候更进一步去了解模板实例化的另一种形式——显式实例化。

2.5.2　显式实例化

程序员可以明确告诉编译器去实例化一个类模板或函数模板，这称为显式实例化，它有两种形式：显式实例化定义(**explicit instantiation definition**)和显式实例化声明(**explicit instantiation declaration**)。我们将依次讨论它们。

1. 显式实例化定义

显式实例化定义可以出现在程序中的任意位置，但必须在它所引用的模板定义之后。显式模板实例化定义的语法形式如下所示。

- 类模板的语法如下：

```
template class-key template-name <argument-list>
```

- 函数模板的语法如下：

```
template return-type name<argument-list>(parameter-list);
template return-type name(parameter-list);
```

如你所见，无论哪种情况，显式实例化定义都是由关键字 template 引入，但后面没有任何形参列表。对于类模板，class-key 可以是关键字 class、struct 或 union 中的任意一个。对于类模板和函数模板，带有给定实参列表的显式实例化定义在整个程序中只能出现一次。

我们将通过几个示例来了解其工作原理。下面是第一个例子：

```
namespace ns
{
    template <typename T>
    struct wrapper
    {
        T value;
    };
```

```
    template struct wrapper<int>;        // [1]
}

template struct ns::wrapper<double>;    // [2]

int main() {}
```

在上述代码中，wrapper<T>是定义在名空间 ns 中的一个类模板。代码中标有 [1] 和[2] 的语句分别代表 wrapper<int>和 wrapper<double>的显式实例化定义。显式实例化定义只能与其所引用的模板定义位于相同的名空间中(如 [1])，或者就必须是完全限定的(如 [2])。也可以为函数模板编写类似的显式模板定义。

```
namespace ns
{
    template <typename T>
    T add(T const a, T const b)
    {
       return a + b;
    }

    template int add(int, int);            // [1]
}

template double ns::add(double, double);  // [2]

int main() { }
```

第二个例子与第一个例子非常相似：[1]和[2]都表示 add<int>()和 add<double>() 的显式模板定义。

如果显式实例化定义不属于模板所在的名空间，则必须使用完全限定的名称。即使使用了 using 语句，也不会使得该名称在当前名空间中可见。代码示例如下：

```
namespace ns
{
    template <typename T>
    struct wrapper { T value; };
}

using namespace ns;

template struct wrapper<double>;   // 错误!
```

本例最后一行将会引发一个编译错误，因为 wrapper 是一个未知标识符；如果需要使用，则必须使用名空间名称来限定，如 ns::wrapper。

当把类成员用于返回类型或形参类型时，在显式实例化定义中将会忽略成员访问说明符。代码示例如下：

```
template <typename T>
class foo
{
```

```
    struct bar {};

    T f(bar const arg)
    {
      return {};
    }
};

template int foo<int>::f(foo<int>::bar);
```

类 X<T>::bar 和函数 foo<T>::f() 都是类 foo<T>的私有成员，在显式实例化定义中却能直接使用它们，正如本例最后一行所示。

在了解了显式实例化的定义及其工作原理之后，人们很容易对其使用场景产生疑问。为什么需要告知编译器去实例化一个模板？答案是它有助于分发库(distribute libraries)、减少编译时间和可执行程序的大小。如果你正在编译一个要以.lib 文件发布的库，而该库使用了模板，那么没有实例化的模板定义不会放入库中。而这会导致每次使用库时用户代码的构建时间增加。如果在库中强制实例化模板，这些定义就会被放入目标文件和发布的 .lib 文件中。因此，用户代码只需要链接到库文件中的可用函数，而不必次次编译。这正是微软 MSVC CRT 库为所有流、locale 和字符串类所做的工作。libstdc++库为字符串类和一些其他类也做了同样的处理。

模板实例化可能会产生一个问题，那就是每个翻译单元一个定义，最终可能会得到多个定义。如果包含模板的头文件为多个翻译单元(即 .cpp 文件)所包含，并且使用了相同的模板实例(如前例中的 wrapper<int>)，那么编译器会将这些实例化的相同副本放入每个翻译单元中。这将导致目标文件的大小增加。而这个问题可以通过显式实例化声明来解决，这是我们接下来要讨论的话题。

2. 显式实例化声明

显式实例化声明(自C++11起可用)是一种告诉编译器模板实例化的定义可以在不同的翻译单元中找到，并且不应生成新定义的方式。其语法与显式实例化定义相同，只需要在声明前使用关键字 extern。

- 类模板的语法如下：

```
extern template class-key template-name <argument-list>
```

- 函数模板的语法如下：

```
extern template return-type name<argument-list>(parameter-list);
extern template return-type name(parameter-list);
```

如果你提供了显式实例化声明，但在程序的任何翻译单元中都没有实例化定义，那么结果就是编译器警告和链接器错误。这里有个技巧，是在一个源文件中声明显式模板实例化，并在其余文件中声明显式模板声明。这样做既能节省编译时间，又能减少目标文件的大小。

让我们看看下面的示例。

```
// wrapper.h
template <typename T>
struct wrapper
{
   T data;
};

extern template wrapper<int>;   // [1]

// source1.cpp
#include "wrapper.h"
#include <iostream>

template wrapper<int>;          // [2]

void f()
{
   ext::wrapper<int> a{ 42 };
   std::cout << a.data << '\n';
}

// source2.cpp
#include "wrapper.h"
#include <iostream>

void g()
{
   wrapper<int> a{ 100 };
   std::cout << a.data << '\n';
}

// main.cpp
#include "wrapper.h"
int main()
{
   wrapper<int> a{ 0 };
}
```

从本例中我们能够看出：

- 头文件 wrapper.h 包含一个模板类 wrapper<T>。标有[1]的行中有 wrapper<int> 的显式实例化声明，它告知编译器在编译包含此头文件的源文件(翻译单元)时不要为此实例化生成新的定义。
- 文件 source1.cpp 包含了头文件 wrapper.h，并且在标有[2]的行中包含 wrapper<int> 的显式实例化定义。这是整个程序中对该实例化的唯一定义。
- 源文件 source2.cpp 和 main.cpp 都使用了 wrapper<int>，但没有任何明确的实例化定义或声明。这是因为当它们各自包含头文件 wrapper.h 之后，其头文件中的显式声明对它们可见。

此外，我们也可以从头文件中移除显式实例化声明，但这样就必须将其添加到包含

头文件的每个源文件中，而我们很可能会忘记这么做。

在显式声明模板后，需要记住的是那些在类内定义的类成员函数总是被视为内联的，因此它们总是会被实例化。因此，只有在类外定义的成员函数才需要使用关键字 extern。

既然我们已经了解了模板实例化，现在可以继续讨论另一个重要的主题，即模板特化(**template specialization**)，该术语用于模板实例化所创建的定义，以处理一组特定的模板实参。

2.6　理解模板特化

模板特化是模板实例化所创建的定义。被特化的模板称为主模板(**primary template**)。你可以为一组给定的模板实参提供显式的特化定义，从而覆盖掉编译器隐式生成的代码。该技术为类型特征和条件编译等特性提供了支持，我们将在第 5 章中继续深入探讨这些元编程概念。

模板特化有两种形式：**显式(完全)特化**与**部分特化**。我们将在接下来的章节中进一步深入讨论它们。

2.6.1　显式特化

显式特化(亦称完全特化)是指为模板实例化提供完整模板实参集的定义时所发生的特化。以下几种情况可以完全特化：

- 函数模板
- 类模板
- 变量模板(自 C++14 起)
- 类模板的成员函数、类和枚举值
- 类或类模板的成员函数模板和类模板
- 类模板的静态数据成员

让我们先看下面的示例。

```
template <typename T>
struct is_floating_point
{
   constexpr static bool value = false;
};

template <>
struct is_floating_point<float>
{
   constexpr static bool value = true;
};
```

```
template <>
struct is_floating_point<double>
{
   constexpr static bool value = true;
};

template <>
struct is_floating_point<long double>
{
   constexpr static bool value = true;
};
```

在本例中，is_floating_point 是主模板(primary template)。它包含一个名为 value 的 constexpr 静态布尔型数据成员，该成员的初始值为 false。然后，我们为 float、double 和 long double 类型提供了该主模板的 3 个完全特化。这些新定义使用 true 而非 false 来初始化 value。因此，可以使用该模板编写如下代码。

```
std::cout << is_floating_point<int>::value        << '\n';
std::cout << is_floating_point<float>::value      << '\n';
std::cout << is_floating_point<double>::value     << '\n';
std::cout << is_floating_point<long double>::value << '\n';
std::cout << is_floating_point<std::string>::value << '\n';
```

其中第一行和最后一行输出 0(表示 false)，其余各行输出 1(表示 true)。这个示例也演示了类型特征的工作原理。事实上，标准库在 std 名空间中包含一个名为 is_floating_point 的类模板，它定义在头文件<type_traits>中。我们将在第 5 章中进一步深入探讨这个主题。

正如本例所示，静态类成员可以完全特化，但每个特化各自拥有自己的静态成员的副本，下面的示例展示了这一点。

```
template <typename T>
struct foo
{
   static T value;
};

template <typename T> T foo<T>::value = 0;
template <> int foo<int>::value = 42;

foo<double> a, b;  // a.value=0, b.value=0
foo<int> c;        // c.value=42

a.value = 100;     // a.value=100, b.value=100, c.value=42
```

本例中，foo<T> 是带有一个静态成员 value 的类模板。对于主模板，value 初始化为 0；而对于其 int 类型的特化，value 初始化为 42。在声明了变量 a、b 和 c 之后，a.value 和 b.value 的值都是 0，而 c.value 的值为 42。但是，在给 a.value 赋值 100 后，b.value 的值也变为 100，而 c.value 的值依旧是 42。

显式特化必须出现在主模板的声明之后。它不需要在显式特化前提供主模板的定义。因此，下面的代码是正确的。

```
template <typename T>
struct is_floating_point;

template <>
struct is_floating_point<float>
{
   constexpr static bool value = true;
};

template <typename T>
struct is_floating_point
{
   constexpr static bool value = false;
};
```

模板特化也可以只声明而不定义，这样的模板特化可以像其他不完整类型(incomplete type)一样使用。示例如下：

```
template <typename>
struct foo {};          // 主模板

template <>
struct foo<int>;        // 显式特化声明

foo<double> a;          // 正确
foo<int>* b;            // 正确
foo<int> c;             // 错误! foo<int> 的类型不完整
```

本例中，foo<T> 是主模板，而只声明了 int 类型的显式特化。这样就允许我们使用 foo<double>和 foo<int>*(支持声明不完整类型的指针)，但在声明变量 c 的时候，由于仍旧缺少 int 类型的完全特化的定义，因此 foo<int> 的完整类型不可用，而这会导致编译出错。

在特化函数模板时，如果编译器能够根据函数实参的类型推导出模板实参，那么该模板实参是可选的。示例如下：

```
template <typename T>
struct foo {};

template <typename T>
void func(foo<T>)
{
   std::cout << "primary template\n";
}

template<>
void func(foo<int>)
{
```

```
    std::cout << "int specialization\n";
}
```

函数模板 func 的 int 类型的完全特化的语法应该是 template<> func<int>(foo<int>)。但是，编译器能够从函数实参中推导出 T 所表示的实际类型。因此，我们在定义特化时无须指定它。

另一方面，函数模板和成员函数模板的声明或定义不允许包含默认函数实参。因此，编译下面的示例将会报错。

```
template <typename T>
void func(T a)
{
    std::cout << "primary template\n";
}

template <>
void func(int a = 0) // 错误！不支持默认实参
{
    std::cout << "int specialization\n";
}
```

在所有这些示例中，模板只有一个模板实参。但事实上，模板可以有多个实参。而显式特化要求定义时必须指定完整的实参集。代码如下：

```
template <typename T, typename U>
void func(T a, U b)
{
    std::cout << "primary template\n";
}

template <>
void func(int a, int b)
{
    std::cout << "int-int specialization\n";
}

template <>
void func(int a, double b)
{
    std::cout << "int-double specialization\n";
}

func(1, 2);       // int-int 特化
func(1, 2.0);     // int-double 特化
func(1.0, 2.0);   // 主模板
```

在了解了这些知识之后，我们可以继续研究“部分特化”，它基本上是显式(完全)特化的一种泛化。

2.6.2　部分特化

如果在特化主模板时仅指定部分模板实参，那么这种特化称为部分特化(partial specialization)。这意味着部分特化将同时拥有模板形参列表(后跟关键字 template)和模板实参列表(后跟模板名称)。但是，只有模板类可以部分特化。

让我们通过下面的示例理解其工作原理。

```
template <typename T, int S>
struct collection
{
   void operator()()
   { std::cout << "primary template\n"; }
};

template <typename T>
struct collection<T, 10>
{
   void operator()()
   { std::cout << "partial specialization <T, 10>\n"; }
};

template <int S>
struct collection<int, S>
{
   void operator()()
   { std::cout << "partial specialization <int, S>\n"; }
};

template <typename T, int S>
struct collection<T*, S>
{
   void operator()()
   { std::cout << "partial specialization <T*, S>\n"; }
};
```

我们的主模板名为 collection，它有两个模板实参(一个类型模板实参，一个非类型模板实参)，还有如下 3 个部分特化。

- 对非类型模板实参 S 值为 10 的特化
- 对 int 类型的特化
- 对指针类型 T*的特化

这些模板的使用方式如下所示。

```
collection<char, 42> a;     // 主模板
collection<int,  42> b;     // <int, S> 部分特化
collection<char, 10> c;     // <T, 10> 部分特化
collection<int*, 20> d;     // <T*, S> 部分特化
```

如同注释所示，a 根据主模板来实例化，b 根据 int 型的部分特化版本(collection<int, S>)来实例化，c 根据 10 的部分特化版本(collection<T, 10>)来实例化，d 根据指针的部分

特化版本(collection<T*, S>)来实例化。但是，有些组合却因存在歧义而不可用，因为编译器无法选择使用哪个模板来实例化。示例如下：

```
collection<int,   10> e;    // 错误! collection<T,10> 或
                            //      collection<int,S>
collection<char*, 10> f;    // 错误! collection<T,10> 或
                            //      collection<T*,S>
```

在第一种情况下，collection<T, 10>和 collection<int, S> 两个部分特化都匹配 collection<int, 10> 这种类型。而在第二种情况下，它既可以是 collection<T, 10>，又可以是 collection<T*, S>。

在定义主模板的特化时需要注意以下几点：

- 部分特化的模板形参列表中的形参不能有默认值。
- 模板形参列表隐含了模板实参列表中实参的顺序，而这种顺序只在部分特化中才有。部分特化的模板实参列表不能与模板形参列表所隐含的模板实参列表相同。
- 在模板实参列表中，只能为非类型模板形参使用标识符，而不能使用表达式。示例如下：

```
template <int A, int B> struct foo {};
template <int A> struct foo<A, A> {};       // 正确
template <int A> struct foo<A, A + 1> {};   // 错误
```

当一个类模板有部分特化时，编译器必须决定生成定义的最佳匹配。为此，在实际特化时，编译器会将模板特化中的模板实参与主模板和部分特化中的模板实参列表进行匹配。根据匹配的结果，编译器将执行以下操作：

- 如果未能匹配，则根据主模板生成定义。
- 如果只找到一个部分特化，则根据该特化生成定义。
- 如果不止一个匹配的部分特化，则根据最匹配特化版本生成定义，但前提是该特化版本是唯一的；否则，编译器将会报错(正如我们之前所看到的)。其中，如果模板 A 接受的类型是模板 B 接受的类型的子集，且反之则不成立时，则认为模板 A 比模板 B 更具有特化性。

但是，部分特化不能通过名称查找被找到，只有通过名称查找机制找到主模板后，编译器才会考虑部分特化。

为了理解部分特化的用途，让我们看一个真实的例子。

在这个例子中，我们希望能够创建一个函数以一种优雅的方式格式化数组的内容，并将其输出到流中。格式化后数组的内容具有 [1,2,3,4,5] 这种形式。但对于元素类型为 char 的数组而言，元素之间不应该用逗号分隔，而应该展示为中括号内的字符串，例如 [demo]。为此，我们考虑使用 std::array 类。下面的实现将数组内容格式化，并在元素之

间加上分隔符。

```
template <typename T, size_t S>
std::ostream& pretty_print(std::ostream& os,
                           std::array<T, S> const& arr)
{
   os << '[';
   if (S > 0)
   {
      size_t i = 0;
      for (; i < S - 1; ++i)
        os << arr[i] << ',';
      os << arr[S-1];
   }
   os << ']';

   return os;
}

std::array<int, 9> arr {1, 1, 2, 3, 5, 8, 13, 21};
pretty_print(std::cout, arr);     // [1,1,2,3,5,8,13,21]

std::array<char, 9> str;
std::strcpy(str.data(), "template");
pretty_print(std::cout, str);     // [t,e,m,p,l,a,t,e]
```

在这段代码中，pretty_print 是一个带有两个模板形参的函数模板，它与 std::array 类的模板形参匹配。当将数组 arr 作为实参进行调用时，结果输出 [1,1,2,3,5,8,13,21]。当将数组 str 作为实参进行调用时，结果输出 [t,e,m,p,l,a,t,e]。但是，我们第二次调用的本意是输出 [template]。为此，需要另一种专门针对 char 类型的实现。

```
template <size_t S>
std::ostream& pretty_print(std::ostream& os,
                           std::array<char, S> const& arr)
{
   os << '[';
   for (auto const& e : arr)
     os << e;
   os << ']';

   return os;
}

std::array<char, 9> str;
std::strcpy(str.data(), "template");
pretty_print(std::cout, str);  // [template]
```

在第二种实现中，pretty_print 是一个只有单一模板形参的函数模板，该参数是用于表示数组大小的非类型模板形参。而类型模板形参则在 std::array<char, S> 中被显式指定为 char。这样在使用数组 str 调用函数 pretty_print 时，编译器会将 [template] 打印到控制台。

这里的关键在于需要理解被部分特化的不是函数模板 pretty_print，而是类模板

std::array。函数模板不能被部分特化，因而这里只有函数重载。但是 std::array<char, S> 是主类模板 std::array<T, S> 的部分特化。

我们在本章中看到的所有示例都是函数模板或类模板。然而，变量也可以是模板，这将是下一节将要讨论的主题。

2.7 定义变量模板

C++14 引入了变量模板，它允许我们在名空间作用域内定义变量模板。定义在名空间中的变量模板表示一族全局变量，定义在类作用域中的变量模板表示静态数据成员。

变量模板在名空间作用域内声明，如下面的代码片段所示。这是一个可以在文献中找到的典型示例，能够很好地展示变量模板的优势。

```
template<class T>
constexpr T PI = T(3.1415926535897932385L);
```

语法类似于声明变量(或数据成员)，但结合了声明模板的语法。

随之而来的问题是变量模板究竟有什么用。让我们通过构建一个例子来回答这个问题。假设我们要编写一个函数模板，在给定球体半径的情况下，返回球体的体积。球体的体积是 4πr^3 / 3。因此，可能的实现如下所示。

```
constexpr double PI = 3.1415926535897932385L;

template <typename T>
T sphere_volume(T const r)
{
   return 4 * PI * r * r * r / 3;
}
```

本例中，我们将 PI 定义为 double 类型的编译期常量。假如我们使用 float 作为类型模板形参 T，编译器就会告警。

```
float v1 = sphere_volume(42.0f); // 告警
double v2 = sphere_volume(42.0); // 正确
```

解决这个问题的一种方案是将 PI 作为模板类的静态数据成员，其类型由模板类的类型模板形参决定。具体实现如下：

```
template <typename T>
struct PI
{
   static const T value;
};

template <typename T>
const T PI<T>::value = T(3.1415926535897932385L);
```