



C++ Templates

Second Edition

作者: David Vandevoorde, Nicolai M. Josuttis, Douglas Gregor

译者: 陈晓伟

本书概述

模板是 C++ 中强大的特性，而对模板的误解并未随着 C++ 语言和开发社区的发展而消弭，从而无法使其无法发挥全力。本书的三位作者，作为 C++ 专家，展示了如何使用现代模板来构建干净、快捷、高效、容易维护的软件。

第二版对 C++11、C++14 和 C++17 标准进行了更新，对改进模板或与模板交互的特性进行了解释，包括可变参数模板、泛型 Lambda、类模板参数推导、编译时 if、转发引用和用户定义文字。还深入研究了一些基本的语言概念（比如值类别），并包含了所有标准类型特征。

本书从基本概念和相关语言特征开始，其余作为参考。先关注语言本书，再是编码、高级应用和复杂的惯用法。过程中，示例清楚地说明了抽象概念，并演示了模板的最佳实践。

关键特性

- 准确理解模板的行为，避免陷阱
- 使用模板编写有效、灵活、可维护的软件
- 掌握有效的习语和技巧
- 保持性能或安全的情况下重用源码
- C++ 标准库中的泛型编程
- 预览“概念”特性

作者简介

David Vandevoorde 在 20 世纪 80 年代后期开始用 C++ 编程。从伦斯勒理工学院获得博士学位后，成为惠普 C++ 编译器团队的技术负责人。1999 年，加入了爱迪生设计集团 (EDG)，该集团的 C++ 编译器技术是业界领先的。他是 C++ 标准委员会的活跃成员，也是 comp.lang.c++ 新闻组的主持人（参与创办）。也是《C++ Solutions》的作者，该书是《C++ Programming Language, 3rd Edition》的配套书籍。

Nicolai M. Josuttis 因其畅销书籍《The C++ Standard Library - A Tutorial and Reference》闻名于世，是一名独立技术顾问，为电信、交通、金融和制造业设计面向对象的软件。也是 C++ 标准委员会的活跃成员，也是 System Bauhaus 的合伙人，System Bauhaus 是一个由面向对象系统开发专家组成的德国团体。Josuttis 还写过其他几本关于面向对象编程和 C++ 的书。

Douglas Gregor 是苹果公司的高级 Swift/C++/Objective-C 编译工程师，拥有伦斯勒理工学院的计算机科学博士学位，并在印第安纳大学从事博士后工作。

本书相关

- Github 地址:

<https://github.com/xiaoweiChen/Cpp-Templates-2nd>

前言

C++ 模板在 1990 年就在“带注释的 C++ 参考手册”(ARM; 参见 [ellisstroustrup]) 中出现了，至今已经 30 多年了。在此之前，有更专业的书籍中对其详细论述过。十多年后，我们发现对于这个神秘、复杂、强大的 C++ 特性，相应的基本概念介绍和高级技术文献居然少的可怜。本书的第一版中，我们想要解决这个问题，并决定写一本关于模板的书(有点不够谦逊)。

自 2002 年底发布第一版本后，C++ 发生了很多大的变化。C++ 标准增加了很多新特性，C++ 社区不断创新也有助于展示基于模板的新编程技术。因此，本书的第二版保留了与第一版相同的目标，但使用的是“现代 C++”。

我们以不同背景和不同目的来完成写书的任务。David(又名“Daveed”)是一位经验丰富的编译器作者，也是发展核心语言的 C++ 标准委员会工作组的积极参与者，他对模板的所有功能(和问题)的精确和详细描述很感兴趣。Nico 是一名“普通”的程序员，C++ 标准委员会库工作组的成员，他对模板的技术很感兴趣。Doug 是一名模板库开发人员，成为编译器作者和语言设计师后，对收集、分类和评估用于构建模板库的技术很有兴趣。此外，我们希望与读者和整个社区积极分享这些知识，以减少误解、困惑或忧虑。

因此，会有示例概念介绍和模板行为的描述。从模板的原则开始，逐步发展到“模板的艺术”，将展现(或重新了解)静态多态性、类型特征、元编程和表达式模板等编程方式。还会更深入地了解 C++ 标准库，其中所有的代码都会涉及模板。

写这本书的过程中，我们也学到了很多，也获得了很多快乐。希望在阅读本书时，您也能享受这本书带给您的快乐。

第二版的致谢

写书很难，维护更难。过去的十年里，我们花了五年多的时间完成了第二版，如果没有其他人的支持和耐心，这将是不可能完成的任务。

首先，感谢 C++ 社区和 C++ 标准化委员会。除了添加新的语言标准和库特性之外，还耐心和热情地向我们解释和讨论他们的工作。

过去 15 年里，社区成员们为第一版的错误和可能的改进提供反馈。人数实在太多，无法在这里逐个感谢，真的很感谢你们花时间写下自己的想法和观察。有时的回复不够及时，还望能谅解。

感谢本书的审阅者们，为本书提供了宝贵的反馈和说明。这些评论对于本书的质量至关重要，这也再次证明了好东西需要许多“聪明人”。在此，非常感谢 Steve Dewhurst, Howard Hinnant, Mikael Kilpeläinen, Dietmar Kühl, Daniel Krübler, Nevin Liber, Andreas Neiser, Eric Niebler, Richard Smith, Andrew Sutton, Hubert Tong 和 Ville Voutilainen。

当然，还要感谢 Addison-Wesley/Pearson 出版社的所有人，不能再把专业人士对书籍作者的支持视为理所当然。他们很有耐心，在适当的时候还会给出建议，在需要专业知识的时候也会提供帮助。在此，非常感谢 Peter Gordon, Kim Boedigheimer, Greg Doench, Julie Nahil, Dana Wilson 和 Carol Lallier。

特别感谢 LaTeX 社区提供的文本系统，并感谢 Frank Mittelbach 解决了关于 \LaTeX 的问题（我们几个这方面真的太菜了）。

David 的致谢

第二版让大家久等了，完成最后的润色时，我由衷的感激那些让这本书出现的人。首先是我的妻子 (Karina) 和女儿们 (Alessandra 和 Cassandra)，允许我从“家庭日程”中抽出大量时间完成这本书，尤其是在工作的最后一年。我的父母也对这本书很感兴趣，每当去看望他们时，都要了解这个写作项目的情况。

显然，这是一本技术书籍，内容是关于编程的知识和经验。然而，仅有“经验”还是远远不够的。这里，非常感谢 Nico 承担了“管理”和“生产”工作（以及他所有的贡献）。如果这本书对你有用，并且有一天遇到了 Nico，一定要感谢他的付出。还要感谢 Doug 的加入，并在日程安排艰难的时刻坚持本书的编纂。

多年来，C++ 社区的许多开发者也分享了宝贵见解，感谢！这里，要特别感谢 Richard Smith，多年来一直使用“神秘的”技巧高效地回复我的邮件。同样，感谢我的同事 John Spicer, Mike Miller 和 Mike Herrick，他们也分享了他们的知识和经验，让我们学到了更多。

Nico 的致谢

首先，感谢两位专家，David 和 Doug，从他们那里学到了很多。我作为一名应用程序程序员和库使用者，经常会问出一些傻乎乎的问题。现在，我想成为一个真正的专家（当然，直到下一个问题的出现），想想就有些小激动呢！

还要感谢 Jutta Eckstein。Jutta 能够推动和支持人们实现理想、想法和目标。大多数人在 IT 行业遇到她或者和她一起工作的时候才会有这样的体验，而我却有幸能在日常生活中得到她的帮助。想想已经好多年了，希望今后也能一直这样。

Doug 的致谢

衷心感谢我的妻子 Amy，以及我们的两个女儿 Molly 和 Tessa。他们的爱和陪伴给我带来了日常的快乐和应对生活和工作中最大挑战的信心。还要感谢我的父母，感谢他们教会了我热爱学习，以及对我的鼓励。

和 David 和 Nico 工作是一件很愉快的事，他们性格迥异，却又能很好地互补。David 具有非常清晰的技术写作能力，他的描述即精确又带有启发性。Nico，除了出色的组织能力(使其他两位合著者不至于陷入混乱)外，还有一种独特的能力，可以分解复杂的技术讨论，使其变得简单、容易理解，以及清晰。

第一版的致谢

这本书提供了一些想法、概念和解决方案，还有许多例子。感谢所有在过去几年中帮助和支持我们的人和企业。

首先，要感谢所有的审稿人和所有给早期手稿意见的人员，他们保证了这本书的品质:Kyle Blaney, Thomas Gschwind, Dennis Mancl, Patrick Mc Killen 和 Jan Christiaan van Winkel。特别感谢 Dietmar Kühl 一丝不苟地审阅和编辑了本书，他的反馈对这本书的质量贡献巨大。

还要感谢给我们机会在不同平台上用不同编译器测试示例的朋友和企业。非常感谢爱迪生设计集团，感谢他们的编译器和支持。还要感谢 GNU 和 egcs 编译器的开发者 (Jason Merrill)，以及微软为 Visual C++ 提供的评估版本 (Jonathan Caves, Herb Sutter 和 Jason Shirk 是我们的联系人)。

许多现有的“C++ 智慧”是由线上 C++ 社区创造的，大部分都来自 Usenet 组的 comp.lang.c++ 和 comp.std.c++. 因此，特别感谢这些小组的积极主持，使讨论保持有益和建设性。我们也非常感谢所有花时间来描述和解释想法的人们，这让我们受益匪浅。

Addison-Wesley 团队做了一件伟大的工作。非常感谢 Addison-Wesley(我们的编辑) 对这本书的温柔鼓励、良好建议和不懈支持。感谢 Tyrrell Albaugh, Bunny Ames, Melanie Buck, Jacquelyn Doucette, Chanda Leary-Coutu, Catherine Ohala 和 Marty Rabinowitz。很感谢 Marina Lang，她是第一个赞助这本书的 Addison-Wesley 成员。Susan Winer 参与了早期的编辑工作，帮助我们塑造了后来的成品。

Nico 的致谢

首先感谢我的家人:Ulli, Lucas, Anica 和 Frederic。感谢对我的耐心，并体贴和鼓励支持这本书的写作。

另外，感谢 David。他非常非常非常的专业，而且非常有耐心(有时我会问一些傻乎乎的问题)，和他一起工作感觉棒极了。

David 的致谢

我的妻子 Karina 在本书的写作中发挥了重要作用，感激她在我生命中扮演的角色。当“业余时间”变得不稳定时，Karina 会帮助我管理时间表，教会我说“不”，以便腾出时间用于写作。最重要的是，她对这个项目非常支持，我每天都在享受她赐予我的友谊和爱意。

感谢能够与 Nico 合作。他对文书的贡献巨大，其经验和纪律性将我们凌乱的涂鸦，变成了组织良好的作品。

John “Mr. Template” Spicer 和 Steve “Mr. Overload” Adameczyk 是好朋友和好同事，他们(在一起)就是 C++ 语言的权威，他们指出了本书描述的许多问题。若本书在 C++ 语言描述中出现了错误，肯定是我们忘了向他们咨询。

最后，我想对那些支持这个项目的人表示感谢(啦啦队的力量也不可低估)。首先是我的父母：他们对我的爱和鼓励让一切变得不同。还有无数的朋友问：“书写得怎么样了？”他们也是完成本书的力量源泉:Michael Beckmann, Brett 和 Julie Beene, Jarran Carr, Simon Chang, Ho 和 Sarah Cho、Christophe De Dinechin、Ewa Deelman、Neil Eberle、Sassan Hazeghi、Vikram Kumar、Jim 和 Lindsay Long、R.J. Morgan、Mike Puritano、Ragu Raghavendra、Jim 和 Phuong Sharp、Gregg Vaughn 和 John Wiegley。

关于本书

本书第一版出版于 15 年前，目标是一本 C++ 模板权威指南。本书得到了不少读者的认可，也多次推荐为参考书目，屡获好评。

第一版中有不少现代 C++ 的内容，但 C++ 也在发展，从 C++11 到 C++14，再到 C++17，对第一版内容的修订势在必行。

第二版，初心不改：既是一本内容全面的参考书，也是一本简易的教程。因为 C++ 已经不是原来的 C++ 了，这次只针对现代 C++。

目前的 C++ 编程环境要好于本书第一版发布时，这期间有一些深入探讨模板应用的书籍。可以从互联网上获取更多的 C++ 模板知识，以及基于模板的编程技术和应用实例。第二版中，将重点关注那些可以泛化的技术。

现代 C++ 为相同功能提供了更简单的方法，所以第一版中的部分内容已经过时。因此有一部分在第二版中删除，而现代 C++ 中的更新会进行填充。

尽管 C++ 模板已经出现了 20 多年了，但目前 C++ 开发者社区中仍然会出现其在软件开发中的新理解。本书的目标是和读者分享这些内容，当然也希望能够启发读者产生新的理解。

目录

第一部分：基础知识

介绍 C++ 模板的基础概念和语言特性，通过函数模板和类模板讨论其目的和概念。再介绍其他的模板特性，比如：非类型模板参数，可变参模板，`typename` 关键字和成员模板。并且，讨论如何处理移动语义，如何声明模板参数，以及如何使用泛型进行编译时编程。本节最后会对一些术语和模板在实际中的应用，给开发工程师和泛型库的开发者们提供一些建议。

为何需要模板？

C++ 要求使用特定类型声明变量、函数和大多数其他类型的实体。但是，对于不同的类型，很多代码看起来一样。例如，算法快速排序的实现对于不同的数据结构（比如 `array<int>` 或 `vector<string>`）在结构上看就是相同的，只要所包含的类型可以相互比较。

如果编程语言不支持这种泛型特性，就只有这些（糟糕的）替代方案了：

1. 对不同类型重复实现相同的算法。
2. 公共基类（比如 `Object` 和 `void*`）里面实现通用算法。
3. 使用预处理。

若从其它语言转投 C++，可能已经使用过以上的方法了。这里来说说他们的缺点：

1. 重复实现相同算法，就是重复地造轮子！并且会犯相同的错误。为了避免犯更多的错误，不会使用复杂但高效的算法。
2. 公共基类里实现统一的代码，就等于放弃了类型检查。而且，有时候某些类必须要从特殊的基类派生出来，这会增加代码维护的成本。
3. 采用预处理的方式，就需要实现一些“笨拙的文本替换”，这很难兼顾作用域和类型检查，更容易引发奇怪的错误。

而模板就不会有这些问题，它就是为了一种或多种未明确定义的类型而定义的函数或者类。使用模板时，需要显式地或隐式地指定模板参数。由于模板是 C++ 的特性，肯定会检查类型和作用域。

目前模板使用的很广，在 C++ 标准库中，几乎所有的代码都用到了模板。标准库提供了一些针对某种特定类型的值或对象的排序算法，也提供一些数据结构（也叫容器）来维护某种特定类型的元素，对于字符串而言，“特定类型”就是“字符”。当然，这只是最基础的功能。模板还允许参数化函数或类的行为，优化代码以及参数化其他信息。这些高级特性会在后面介绍，先从简单的模板开始吧。

第 1 章 函数模板

本章将介绍函数模板。函数模板是参数化的函数，代表一组具有相似行为的函数。

1.1. 初识函数模板

函数模板代表一组函数，提供了适用于不同数据类型的行为。除了某些信息未明确指定外，看起来就和普通函数一样。这些未指定的信息就是参数化信息。

1.1.1 定义模板

下面就是一个函数模板，返回两个数的最大值：

basics/max1.hpp

```
1 template<typename T>
2 T max (T a, T b)
3 {
4     // if b < a then yield a else yield b
5     return b < a ? a : b;
6 }
```

这个模板定义指定了一个函数组，返回两个值的最大值，`a` 和 `b` 作为函数的参数。

根据 [StepanovNotes]，`max()` 模板有意地返回 $b < a ? a : b$ ，而不是 $a < b ? b : a$ ，是为了确保函数行为的正确性。

参数类型为模板参数 `T`，模板参数必须使用以下形式的进行声明：

```
1 template<逗号分割的模板参数>
```

示例中，参数列表是 `typename T`。注意 `<` 和 `>` 尖括号。关键字 `typename` 引入了一个类型参数。这是 C++ 程序中最常见的模板参数类型，也可以使用其他参数，会在后面进行讨论(参见第 3 章)。

类型参数是 `T`，也可以使用其他标识符作为参数名(`T` 是惯例罢了)。类型参数表示在调用函数时才确定的类型，可以使用支持模板使用操作的类型(基本类型、类等)。因为 `a` 和 `b` 使用小于操作符进行比较，所以类型 `T` 必须支持小于操作符。`max()` 的定义可能不太明显，类型 `T` 的值必须可复制，以便返回。

C++17 之前，类型 `T` 也必须可复制才能传递参数。C++17 以后，即使复制构造函数和移动构造函数都无效，也可以传递临时变量(右值，参见附录 B)。

由于一些历史原因，还可以使用关键字 `class` 来定义类型参数。关键字 `typename` 在 C++98 标准中出现得较晚，所以在此之前，关键字 `class` 是引入类型参数的唯一方法，现在这种方法仍然有效。因此，模板 `max()` 可以等价地定义为：

```
1 template<class T>
2 T max (T a, T b)
```

```
3 {  
4     return b < a ? a : b;  
5 }
```

即使使用 `class`, 模板参数也可以使用任意类型。但这样使用 `class` 可能会引起误会(不仅是类可以替换为 `T`), 所以更推荐使用 `typename`。但与类声明不同, 在声明类型参数时, 不能使用关键字 `struct` 来代替 `typename`。

1.1.2 使用模板

下面的展示了如何使用函数模板 `max()`:

basics/max1.cpp

```
1 #include "max1.hpp"  
2 #include <iostream>  
3 #include <string>  
4  
5 int main()  
6 {  
7     int i = 42;  
8     std::cout << "max(7,i): " << ::max(7,i) << '\n';  
9  
10    double f1 = 3.4;  
11    double f2 = -6.7;  
12    std::cout << "max(f1,f2): " << ::max(f1,f2) << '\n';  
13  
14    std::string s1 = "mathematics";  
15    std::string s2 = "math";  
16    std::cout << "max(s1,s2): " << ::max(s1,s2) << '\n';  
17 }
```

这里, `max()` 调用了三次:`int`, `double`, `std::string`。输出如下:

```
max(7,i): 42  
max(f1,f2): 3.4  
max(s1,s2): mathematics
```

代码中, 对 `max()` 的使用都用`::`限定。这是为了确保在全局命名空间中找到 `max()` 模板。因为标准库中有一个 `std::max()`, 在某些情况下可以使用, 但这里可能会产生歧义。

命名空间 `std` 中定义了一个参数类型(例如 `std::string`), 根据 C++ 的查找规则, `std` 中的 `max()` 模板函数和全局模板函数都会找到(参见附录 C)。

模板不会编译成处理任意类型的实体。相反，对于模板所使用的每个类型，会根据模板中生成不同的实体。

“一体多用”的方案可以想想，但不会在实践中使用（运行时效率较低）。所有语言规则都基于这样一个原则：对于不同的模板参数，会生成不同的实体。

因此，`max()` 对这三种类型进行了分别编译。例如，`max()` 的第一次调用

```
1 int i = 42;
2 ... max(7, i) ...
```

使用 `int` 作为模板形参 `T` 的函数模板。因此，等同于调用的如下函数：

```
1 int max (int a, int b)
2 {
3     return b < a ? a : b;
4 }
```

用具体类型替换模板参数的过程称为实例化，会产生一个模板实例。

面向对象编程中，术语实例（instance）和实例化（instantiate）用于不同的上下文——类的具体对象。本书针对模板，仅用这两个术语表示对模板的“使用”，除非另有说明。

注意，使用函数模板就可以进行实例化，开发者无需单独实例化。

类似地，`max()` 的其他调用实例化了 `double` 和 `std::string` 的 `max` 模板：

```
1 double max (double, double);
2 std::string max (std::string, std::string);
```

如果结果代码有效，`void` 也是有效的模板参数。例如：

```
1 template<typename T>
2 T foo(T*)
3 {
4 }
5
6 void* vp = nullptr;
7 foo(vp); // OK: deduces void foo(void*)
```

1.1.3 两阶段翻译

若为不支持使用操作的类型实例化模板，将导致编译时错误。例如：

```
1 std::complex<float> c1, c2; // doesn't provide operator <
2 ...
3 ::max(c1, c2); // ERROR at compile time
```

因此，模板“编译”分为两个阶段：

1. 若在定义时不进行实例化，则会检查模板代码的正确性，而忽略模板参数。这包括：

- 现语法错误，比如：缺少分号。
- 使用未知名称的模板参数（类型名、函数名……）。
- 检查（不依赖于模板参数）静态断言。

2. 实例化时，再次检查模板代码，确保生成的代码有效。特别是所有依赖于模板参数的部分，都会进行重复检查。

例如：

```

1 template<typename T>
2 void foo(T t)
3 {
4     undeclared(); // 若undeclared()未知，则在第一阶段编译时报错
5     undeclared(t); // 若undeclared(T)未知，则在第二阶段编译时报错
6     static_assert(sizeof(int) > 10, // 若sizeof(int)<=10，始终断言失败
7                   "int too small");
8     static_assert(sizeof(T) > 10, // 若示例T的大小小于等于10，则断言失败
9                   "T too small");
10 }
```

检查两次名称的情况称为两阶段查找，在第 14.3.1 节会进行详细讨论。

注意，有些编译器不在第一阶段编译时进行完全的检查。

例如，一些版本的 Visual C++ 编译器（如 Visual Studio 2013 和 2015）允许模板参数使用未声明名称，甚至允许一些语法缺陷（如缺少分号）。

因此，在模板代码第一次实例化前，可能不会看到已存在的问题。

编译和连接

两阶段翻译在实际处理模板时，会有一个问题：当以数模板实例化的方式使用函数模板时，编译器（在某些时候）需要查看该模板的定义。当函数的声明可以编译，并连接通过时，就打破了普通函数的编译和链接方式。第 9 章讨论了处理这个问题的方法。现在，我们使用最简单的方法：仅在头文件中实现每个模板。

1.2. 模板参数推导

当使用函数模板（如 `max()`）时，模板参数由传入的参数决定。如果类型 `T` 传递两个 `int` 型参数，那编译器就会认为 `T` 是 `int` 型。

然而，`T` 可能只是类型的“一部分”。若声明 `max()` 使用常量引用：

```

1 template<typename T>
2 T max (T const& a, T const& b)
3 {
4     return b < a ? a : b;
5 }
```

并传递 `int`，同样 `T` 会推导为 `int`，因为函数形参匹配的是 `int const&`。

类型推导时的类型转换

注意，自动类型转换在类型推导时有一些限制：

- 当通过引用声明参数时，简单的转换也不适用于类型推导。用同一个模板参数 T 声明的两个实参必须完全匹配。
- 当按值声明参数时，只支持简单的转换：忽略 const 或 volatile 的限定符，引用转换为引用的类型，原始数组或函数转换为相应的指针类型。对于使用相同模板参数 T 声明的两个参数，转换类型必须匹配。

例如：

```
1 template<typename T>
2 T max (T a, T b);
3 ...
4 int i = 17;
5 int const c = 42;
6 max(i, c); // OK: T推导为int
7 max(c, c); // OK: T推导为int
8 int& ir = i;
9 max(i, ir); // OK: T推导为int
10 int arr[4];
11 max(&i, arr); // OK: T推导为int*
```

下面就是反面案例了：

```
1 max(4, 7.2); // ERROR: T推导为int或double
2 std::string s;
3 max("hello", s); // ERROR: T推导为char const*或std::string
```

有三种方法可以处理此类错误：

1. 强制转换：

```
1 max(static_cast<double>(4), 7.2); // OK
```

2. 显式指定（或限定）T 的类型：

```
1 max<double>(4, 7.2); // OK
```

3. 指定参数可以有不同的类型。

第 1.3 节将进行详细的说明。第 7.2 节和第 15 章将详细讨论类型推导时的类型转换规则。

默认参数的类型推导

类型推导不适用于默认调用参数。例如：

```
1 template<typename T>
2 void f(T = "");
3 ...
4 f(); // OK: 推导出T为int, 因此其可以调用f<int>(1)
5 f(); // ERROR: 不能推断出T的类型
```

为了支持这种情况，必须为模板形参声明一个默认实参，这将在 1.4 节中进行详述：

```
1 template<typename T = std::string>
2 void f(T = "") ;
3 ...
4 f(); // OK
```

1.3. 多模板参数

函数模板可以有两组不同的类型参数：

1. 模板参数，用尖括号声明在函数模板名之前：

```
1 template<typename T> // T是模板参数
```

2. 调用参数，在函数模板名称后面的圆括号中声明：

```
1 T max (T a, T b) // a和b调用参数
```

其实，可以有任意多种模板参数。可以为两种不同类型的调用参数定义 `max()` 模板：

```
1 template<typename T1, typename T2>
2 T1 max (T1 a, T2 b)
3 {
4     return b < a ? a : b;
5 }
6 ...
7 auto m = ::max(4, 7.2); // OK, 不过返回类型与第一个参数类型一样
```

将不同类型的参数传递给 `max()` 模板这会引发一个问题。若使用其中一种形参类型作为返回类型，则其他参数可能需要向这种类型进行转换。因此，返回类型取决于调用参数的顺序。`66.66` 和 `42` 的最大值将是 `double` 类型的 `66.66`，而 `42` 和 `66.66` 的最大值将是 `int` 类型的 `66`。

C++ 提供了不同的方法来处理这个问题：

- 为返回类型引入第三个模板参数。
- 让编译器找出返回类型。
- 将返回类型声明为两个参数类型的“公共类型”。

1.3.1 返回类型的模板参数

模板参数推导可以使用普通调用函数的语法使用函数模板，所以可以不用显式地指定与模板参数对应的类型。

当然，也可以显式地指定模板参数使用的类型：

```
1 template<typename T>
2 T max (T a, T b);
3 ...
4 ::max<double>(4, 7.2); // 示例中T是double
```

若模板和调用参数之间没有关系，并且无法确定模板参数，则必须显式指定模板参数。例如，可以引入第三个模板参数类型来指定函数模板的返回类型：

```
1 template<typename T1, typename T2, typename RT>
2 RT max (T1 a, T2 b);
```

然而，模板参数不会去推导返回类型，

推导可以视为重载解析的一部分——这个过程不基于返回类型。唯一的例外是转换操作符的返回类型。

RT 不会出现在函数调用参数的类型中。因此，不能推导出 RT 的类型。

C++ 中，不可从调用代码的上下文中推断返回类型。

因此，必须显式地指定模板参数。例如：

```
1 template<typename T1, typename T2, typename RT>
2 RT max (T1 a, T2 b);
3 ...
4 ::max<int,double,double>(4, 7.2); // OK, 但很无趣
```

了解了显式地指定所有函数模板参数，或没有函数模板参数的情况。另一种方法是只显式指定第一个参数，并允许演绎过程派生其他参数。通常，必须指定最后一个不能隐式确定的参数类型之前的所有参数类型。因此，若在例子中改变了模板参数的顺序，只需要指定返回类型即可：

```
1 template<typename RT, typename T1, typename T2>
2 RT max (T1 a, T2 b);
3 ...
4 ::max<double>(4, 7.2) // OK: 返回类型是double, T1和T2可以进行推演
```

例子中，对 `max<double>` 的调用显式地将 RT 设置为 `double`，但是参数 `T1` 和 `T2` 根据实参推导为 `int` 和 `double`。

注意，`max()` 的这些版本也没什么特别。单参数版本时，若传递了两个不同类型的参数，则可以指定参数(和返回)类型。因此，使用 `max()` 单参数版本就好。

关于推导过程的详细信息，请参见第 15 章。

1.3.2 推导返回类型

若返回类型依赖于模板参数，最好和简单的方法是让编译器判断返回类型。从 C++14 开始，可以不显式声明返回类型(需要声明返回类型为 `auto`)：

basics/maxauto.hpp

```
1 template<typename T1, typename T2>
2 auto max (T1 a, T2 b)
3 {
4     return b < a ? a : b;
```

```
5 }
```

返回类型中使用 `auto`, 而没有相应的尾部返回类型 (`->`) 表明需要从函数体中的 `return` 语句推导出实际的返回类型。当然, 从函数体推断返回类型是可行的。因此, 函数体代码必须有效, 有多个返回语句时也必须匹配。

C++14 之前, 只能让编译器通过将函数的实现作为其声明的一部分来确定返回类型。C++11 中, 尾部的返回类型允许使用调用时的参数, 声明的返回类型是从三元操作符某个分支派生的:

basics/maxdecltype.hpp

```
1 template<typename T1, typename T2>
2 auto max (T1 a, T2 b) -> decltype(b < a ? a : b)
3 {
4     return b < a ? a : b;
5 }
```

这有些复杂, 返回类型由三元操作符决定, 但会产生简答的结果(如果 `a` 和 `b` 是不同的类型, 则会为返回值确定一个通用的类型)。

注意:

```
1 template<typename T1, typename T2>
2 auto max (T1 a, T2 b) -> decltype(b < a ? a : b);
```

是一个声明, 使编译器使用三元操作符, 可以在编译时找出 `max()` 的返回类型, 实现也不一定要匹配。事实上, 使用 `true` 作为操作符的条件就够了:

```
1 template<typename T1, typename T2>
2 auto max (T1 a, T2 b) -> decltype(true ? a : b);
```

然而, 这个定义有一个缺陷: 返回类型可能是引用类型, 在某些条件下 `T` 可能是引用。因为这个原因, 应该返回 `T` 衰变的类型:

basics/maxdecltypedecay.hpp

```
1 #include <type_traits>
2
3 template<typename T1, typename T2>
4 auto max (T1 a, T2 b) -> typename std::decay<decltype(true ? a : b)>::type
5 {
6     return b < a ? a : b;
7 }
```

这里使用了 `std::decay<>`, 修饰了返回值的类型。其在标准库中的 `<type_traits>` 头文件中定义(参见第 D.4 节)。因为是类型, 必须用 `typename` 进行限定才能访问(参见第 5.1 节)。

`auto` 类型的初始化是衰变的, 这也适用于返回类型为 `auto` 的返回值。`auto` 作为返回类型的行为就像下面的代码一样, 其中 `a` 由 `i`(衰变类型) 声明:

```
1 int i = 42;
2 int const& ir = i; // ir 引用于 i
```

```
3 auto a = ir; // a的类型为int
```

1.3.3 返回类型为公共类型

C++11 后，标准库提供了一种方法来指定选择“公共类型”。`std::common_type<>::type` 会产生由两个(或多个)不同类型作为模板类型的“公共类型”。例如：

basics/maxcommon.hpp

```
1 #include <type_traits>
2
3 template<typename T1, typename T2>
4 std::common_type_t<T1,T2> max (T1 a, T2 b)
5 {
6     return b < a ? a : b;
7 }
```

`std::common_type` 在 `<type_traits>` 头文件中定义，其生成一个具有返回类型的结构体。其核心用法如下所示：

```
1 typename std::common_type<T1,T2>::type // since C++11
```

C++14 起，可以通过在特性名称后面添加`_t`跳过 `typename` 和`::type` 来简化 trait 的使用(详见第 2.8 节)，这样返回类型定义就变成了：

```
1 std::common_type_t<T1,T2> // C++14起可用
```

`std::common_type<>` 使用了一些模板编程技巧，会在第 26.5.2 节中讨论。内部根据特定类型的三元操作符或特化的语言规则选择结果类型。因此，`::max(4,7.2)` 和`::max(7.2, 4)` 都返回了相同的 `double` 类型值 7.2。注意 `std::common_type<>` 也会衰变，详见 D.5 节。

1.4. 默认模板参数

还可以为模板参数定义默认值。这些值称为默认模板参数，可以用于任何类型的模板。

C++11 前，因为历史原因，默认模板参数只允许在类模板中使用。

甚至可能引用前面的模板参数。

若希望定义返回类型的方法与具有多个参数类型的能力结合起来(如前一节所述)，那么可以为返回类型引入模板参数 RT，将两个参数的公共类型作为默认类型。同样，有多种选择：

1. 可以直接使用三元操作符。但在使用三元操作符之前，只能使用参数的类型：

basics/maxdefault1.hpp

```
1 #include <type_traits>
2
```

```

3 template<typename T1, typename T2,
4 typename RT = std::decay_t<decltype(true ? T1() : T2())>>
5 RT max (T1 a, T2 b)
6 {
7     return b < a ? a : b;
8 }

```

注意 `std::decay_t<>` 的用法以确保返回的不是引用类型。

C++11 中，必须使用 `typename std::decay<...>::type`，而非 `std::decay_t<...>`(见第 2.8 节)。

此实现要求能够为传递的类型调用默认构造函数。还有另一种解决方案，使用 `std::declval`，但是这会使声明更加复杂。参见第 11.2.3 节中的示例。

2. 也可以使用 `std::common_type<>` 来指定返回类型的默认值：

basics/maxdefault3.hpp

```

1 #include <type_traits>
2
3 template<typename T1, typename T2,
4 typename RT = std::common_type_t<T1, T2>>
5 RT max (T1 a, T2 b)
6 {
7     return b < a ? a : b;
8 }

```

注意 `std::common_type<>` 的衰变，因此返回值不会是引用。

可以使用返回类型的默认值：

```

1 auto a = ::max(4, 7.2);

```

或者显式地指定返回的类型：

```

1 auto b = ::max<double, int, long double>(7.2, 4);

```

不过，目前必须指定三种类型才能只明确指定返回类型。所以，将返回类型移动到第一个模板参数，就可以对其类型进行推导。原则上，即使后面没有默认参数，推导的函数模板参数也可以有默认类型：

```

1 template<typename RT = long, typename T1, typename T2>
2 RT max (T1 a, T2 b)
3 {
4     return b < a ? a : b;
5 }

```

通过定义，可以这样使用：

```

1 int i;
2 long l;

```

```
3 ...
4 max(i, l); // 返回long(返回类型的模板默认类型)
5 max<int>(4, 42); // 显式返回int
```

但这种方法只在模板参数有默认值时才有意义。这里，需要模板参数的默认值依赖于之前的模板参数。原理上这可行（在 26.5.1 节中会进行讨论），但是这种技术依赖于类型特征，会使定义复杂化。

由于所有这些原因，最好和最简单的解决方案是让编译器推导出第 1.3.2 节中提出的返回类型。

1.5. 重载函数模板

与普通函数一样，函数模板也可以重载。可以使用相同的函数名来声明不同的函数体，当使用该函数名称时，编译器会决定调用哪一个候选函数。即使没有模板，这个决策规则也会相当复杂。本节中，将讨论涉及模板的重载。如果不熟悉无模板重载的基本规则，请参阅附录 C，其中提供了关于重载解析规则相当详细的介绍。

下面的程序示例演示了如何重载函数模板：

basics/max2.cpp

```
1 // maximum of two int values:
2 int max (int a, int b)
3 {
4     return b < a ? a : b;
5 }
6
7 // maximum of two values of any type:
8 template<typename T>
9 T max (T a, T b)
10 {
11     return b < a ? a : b;
12 }
13
14 int main()
15 {
16     ::max(7, 42); // calls the nontemplate for two ints
17     ::max(7.0, 42.0); // calls max<double> (by argument deduction)
18     ::max('a', 'b'); // calls max<char> (by argument deduction)
19     ::max<>(7, 42); // calls max<int> (by argument deduction)
20     ::max<double>(7, 42); // calls max<double> (no argument deduction)
21     ::max('a', 42.7); // calls the nontemplate for two ints
22 }
```

非模板函数可以与相同名称和相同类型的函数模板共存。其他相同的情况下，重载解析将优先使用非模板方式。第一个调用验证了这个规则：

```
1 ::max(7, 42); // 两个int值完全匹配非模板函数
```

若模板可以生成匹配更好的函数实例，则选择模板。这在 `max()` 的第二次和第三次调用中得到了验证：

```
1 ::max(7.0, 42.0); // 调用max<double>(通过参数推导)
2 ::max('a', 'b'); // 调用max<char>(通过参数推导)
```

这里，因为不需要将 `double` 或 `char` 转换成 `int`(参见 C.2 节的重载解析规则)，模板匹的更好。

也可以显式指定一个空的模板参数列表。这种语法表明只有模板才能解析调用，但所有模板参数都应该根据调用参数进行推导：

```
1 ::max<>(7, 42); // 调用max<int>(通过参数推导)
```

对推导的模板参数不进行自动类型转换，从而会自动的对普通函数参数进行类型转换，因此最后一次调用使用了非模板函数 ('`a`' 和 42.7 都转换为 `int`)：

```
1 ::max('a', 42.7); // 只有非模板函数允许这种类型转换
```

一个有趣的例子是重载 `maximum` 模板，其只能显式指定返回类型：

basics/maxdefault4.hpp

```
1 template<typename T1, typename T2>
2 auto max (T1 a, T2 b)
3 {
4     return b < a ? a : b;
5 }
6
7 template<typename RT, typename T1, typename T2>
8 RT max (T1 a, T2 b)
9 {
10    return b < a ? a : b;
11 }
```

现在，可以调用 `max()`：

```
1 auto a = ::max(4, 7.2); // 使用第一个函数模板
2 auto b = ::max<long double>(7.2, 4); // 使用第二个函数模板
```

当这样使用时：

```
1 auto c = ::max<int>(4, 7.2); // ERROR: 两个函数模板都可以匹配
```

这两个模板都匹配，这导致重载解析过程出现歧义。因此，重载函数模板时，应该确保只有一个函数模板与调用匹配。

例子是重载指针和普通 C 字符串的最大值比较函数模板：

basics/max3val.cpp

```
1 #include <cstring>
2 #include <string>
3
```

```

4 // 任意类型的两个最大值:
5 template<typename T>
6 T max (T a, T b)
7 {
8     return b < a ? a : b;
9 }
10
11 // 两个指针内容的最大值:
12 template<typename T>
13 T* max (T* a, T* b)
14 {
15     return *b < *a ? a : b;
16 }
17
18 // 两个C字符串的最大值:
19 char const* max (char const* a, char const* b)
20 {
21     return std::strcmp(b,a) < 0 ? a : b;
22 }
23
24 int main ()
25 {
26     int a = 7;
27     int b = 42;
28     auto m1 = ::max(a,b); // max():两个int类型的值
29
30     std::string s1 = "hey";
31     std::string s2 = "you";
32     auto m2 = ::max(s1,s2); // max():两个std::string类型的值
33
34     int* p1 = &b;
35     int* p2 = &a;
36     auto m3 = ::max(p1,p2); // max():两个指针类型的值
37
38     char const* x = "hello";
39     char const* y = "world";
40     auto m4 = ::max(x,y); // max():两个C字符串类型的值
41 }

```

注意，`max()`的所有重载中，参数都是通过值传递的。重载函数模板时，最好不要进行不必要的更改，可以将更改限制在参数的数量或显式地指定模板参数。否则，可能会出现意想不到的后果。若实现了`max()`模板，通过引用传递参数，对两个通过值传递C字符串进行重载实现，从而就不能使用三个参数的版本来比较三个C字符串：

basics/max3ref.cpp

```

1 #include <cstring>
2
3 // 任意类型的两个最大值(引用调用)

```

```

4 template<typename T>
5 T const& max (T const& a, T const& b)
6 {
7     return b < a ? a : b;
8 }
9
10 // 最多两个C字符串(值调用)
11 char const* max (char const* a, char const* b)
12 {
13     return std::strcmp(b,a) < 0 ? a : b;
14 }
15
16 // 任何类型的最多三个值(引用调用)
17 template<typename T>
18 T const& max (T const& a, T const& b, T const& c)
19 {
20     return max (max(a,b), c); // 如果max(a,b)使用了按值调用就会出错
21 }
22
23 int main ()
24 {
25     auto m1 = ::max(7, 42, 68); // OK
26
27     char const* s1 = "frederic";
28     char const* s2 = "anica";
29     char const* s3 = "lucas";
30     auto m2 = ::max(s1, s2, s3); // 运行时错误(未定义的行为)
31 }
```

问题是，若对三个 C 字符串调用 `max()`，

```
1 return max (max(a,b), c);
```

就会导致运行时错误，因为对于 C 字符，`max(a,b)` 创建了一个新临时变量，并通过引用返回。但这个临时值越过返回语句完成就会销毁，留给 `main()` 一个悬空引用。不幸的是，这个错误非常微妙，可能不必现。

一般来说，符合标准的编译器可以编译这段代码。

与此相反，`main()` 中对 `max()` 的第一次调用不会出现相同的问题。虽为参数 (7、42 和 68) 创建了临时变量，但这些临时变量在 `main()` 中创建，`main()` 中会一直存在，直到程序执行完毕。

这是由于重载解析规则，而导致行为与预期不同的代码示例。此外，需要在调用函数之前声明函数的所有重载版本。因为在进行相应的函数调用时，并非所有重载函数都可见（这很重要）。例如，定义三个参数版本的 `max()`，而没有声明用于 `int` 类型的两个参数版本，就会导致使用三个参数版本的函数时，使用到两个参数的模板：

basics/max4.cpp

```

1 #include <iostream>
2
3 // 两个任意类型的最大值:
4 template<typename T>
5 T max (T a, T b)
6 {
7     std::cout << "max<T>() \n";
8     return b < a ? a : b;
9 }
10
11 // 三个任意类型的最大值:
12 template<typename T>
13 T max (T a, T b, T c)
14 {
15     return max (max(a,b), c); // 对整型也使用模板版本
16 } // 因为下面的声明来了太晚了
17
18 // 两个int值的最大值:
19 int max (int a, int b)
20 {
21     std::cout << "max(int,int) \n";
22     return b < a ? a : b;
23 }
24
25 int main()
26 {
27     ::max(47,11,33); // OOPS: 这里使用的是max<T>(), 而非max(int,int)
28 }

```

我们将在第 13.2 节中讨论其中细节。

1.6. 常见问题

即使是这些简单的函数模板示例，也可能引发进一步的问题。有三个常见问题，在这里简单的讨论下。

1.6.1 使用值，还是引用传递参数？

为什么通常声明函数按值传递参数，而不是使用引用。一般来说，除了简单类型（比如基本类型或 `std::string_view`），因为不会创建副本，所以推荐使用引用传递。

然而，按值传递在下面的情况下会更好：

- 语法简单。
- 编译器会进行很好的优化。
- 移动语义会使复制成本降低。
- 没有复制或移动操作。

此外，对于模板来说：

- 模板可能同时用于简单类型和复杂类型，因此为复杂类型选择这种方法时，可能会对简单类型产生反效果。
- 作为调用者，可以通过引用来传递参数，可以使用 `std::ref()` 和 `std::cref()` (参见 7.3 节)。
- 虽然传递字符串字面值或原始数组可能会产生问题，但通过引用传递通常会有更多的问题。

这些将在第 7 章中详细讨论。目前，我们使用值传递参数 (除非某些功能只有在使用引用时才使用引用)。

1.6.2 为什么不用内联？

通常，函数模板不必使用内联声明。与普通的非内联函数不同，我们可以在头文件中定义非内联函数模板，并在多个翻译单元中包含该头文件。

该规则的唯一例外是，对特定类型的模板进行完全特化，从而产生的代码不再是泛型 (定义了所有模板参数)。参见 9.2 节了解更多细节。

从严格的语言定义角度来看，内联意味着函数的定义可以在程序中出现多次。也表示编译器对该函数的调用应该“内联展开”：某些情况下可以产生更有效的代码，但在许多其他情况下反而会降低代码的效率。现在，编译器通常能够更好地决定是否采纳使用 `inline` 关键字的提示。但是，编译器仍然要考虑在该决策中是否存在内联。

1.6.3 为什么不用 `constexpr`？

C++11 后，可以使用 `constexpr` 提供在编译时计算某些值的能力。对于很多模板来说，这很有意义。

例如，为了能够在编译时使用 `maximum` 函数，必须声明它：

basics/maxconstexpr.hpp

```
1 template<typename T1, typename T2>
2 constexpr auto max (T1 a, T2 b)
3 {
4     return b < a ? a : b;
5 }
```

这样，就可以在有编译时使用 `maximum` 函数模板，比如在声明原始数组的大小时：

```
1 int a[::max(sizeof(char),1000u)];
```

或者定义 `std::array<>` 的大小：

```
1 std::array<std::string, ::max(sizeof(char),1000u)> arr;
```

注意，将 1000 作为 `unsigned int` 传递是为了避免在模板中比较有符号值和无符号值时，编译器发出的警告。

第 8.2 节将讨论使用 `constexpr` 的例子。为了让我们的注意力集中在基本特性上，在讨论其他模板特性时，我们通常会跳过 `constexpr`。

1.7. 总结

- 函数模板为不同的模板参数定义了一个函数族。
- 当根据模板参数向函数传递参数时，函数模板会根据相应的参数类型推导出要实例化的模板参数。
- 可以显式给定模板参数。
- 可以为模板参数定义默认参数。这些参数可以使用前面的模板参数，后面跟着没有默认参数的参数。
- 函数模板可以重载。
- 函数模板与其他函数模板重载时，应确保只有一个函数模板与调用匹配。
- 重载函数模板时，应将更改限制为显式指定模板参数。
- 确保编译器在调用函数模板前，了解函数模板所有的重载版本。

第 2 章 类模板

与函数类似，类也可以用一个或多个类型参数化。用于管理特定类型元素的容器类就是个例子。通过使用类模板，可以在元素类型开放的情况下实现容器类。本章中，我们将使用堆栈作为类模板的示例。

2.1. 实现栈类模板

与函数模板一样，在头文件中声明并定义 `Stack<T>` 类，如下所示：

basics/stack1.hpp

```
1 #include <vector>
2 #include <cassert>
3
4 template<typename T>
5 class Stack {
6 private:
7     std::vector<T> elems; // elements
8
9 public:
10    void push(T const& elem); // push element
11    void pop(); // pop element
12    T const& top() const; // return top element
13    bool empty() const { // return whether the stack is empty
14        return elems.empty();
15    }
16 };
17
18 template<typename T>
19 void Stack<T>::push (T const& elem)
20 {
21     elems.push_back(elem); // append copy of passed elem
22 }
23
24 template<typename T>
25 void Stack<T>::pop ()
26 {
27     assert(!elems.empty());
28     elems.pop_back(); // remove last element
29 }
30
31 template<typename T>
32 T const& Stack<T>::top () const
33 {
34     assert(!elems.empty());
35     return elems.back(); // return copy of last element
36 }
```

类模板使用 C++ 标准库的 `vector`<> 实现。因此，不必实现内存管理、复制构造函数和赋值操作符，可以专注于该类模板的接口。

2.1.1 声明类模板

声明类模板类似于声明函数模板。声明前，必须将一个或多个标识符声明为类型参数。同样，`T` 作为常用标识符：

```
1 template<typename T>
2 class Stack {
3     ...
4 };
```

这里，关键字 `class` 也可以替换 `typename`：

```
1 template<class T>
2 class Stack {
3     ...
4 };
```

类模板内部，可以像使用其他类型一样使用 `T` 来声明成员和成员函数。这个例子中，`T` 用来声明元素的类型为 `T` 的 `vector`，将 `push()` 声明为使用 `T` 作参数的成员函数，并将 `top()` 声明为返回 `T` 的函数：

```
1 template<typename T>
2 class Stack {
3     ...
4     std::vector<T> elems; // elements
5
6     ...
7     void push(T const& elem); // push element
8     void pop(); // pop element
9     T const& top() const; // return top element
10    bool empty() const { // return whether the stack is empty
11        return elems.empty();
12    }
13};
```

这个类的类型是 `Stack<T>`，`T` 是模板参数。因此，只要在使用该类的类型，就必须使用 `Stack<T>` 声明，可以推导模板参数的情况除外。但在类模板中使用类名（不带模板参数），表明这个内部类的模板参数类型和模板类的参数类型相同（详见第 13.2.3 节）。

例如，必须声明自定义的复制构造函数和赋值操作符：

```
1 template<typename T>
2 class Stack {
3     ...
4     Stack (Stack const&); // copy constructor
5     Stack& operator= (Stack const&); // assignment operator
6     ...
7 };
```

形式上等价于：

```
1 template<typename T>
2 class Stack {
3     ...
4     Stack (Stack<T> const&); // copy constructor
5     Stack<T>& operator= (Stack<T> const&); // assignment operator
6     ...
7 };
```

通常 `<T>` 表示特殊模板参数，所以最好使用第一种方式。

这里，

```
1 template<typename T>
2 bool operator== (Stack<T> const& lhs, Stack<T> const& rhs);
```

在需要类名而不是类类型的地方，只能使用 `Stack`。尤其是，指定构造函数的名称(而不是参数)和析构函数时。

与非模板类不同，不能在函数或块作用域内声明或定义类模板。通常，模板只能在全局/命名空间作用域或类中声明内部定义(参见 12.1 节了解详细信息)。

2.1.2 成员函数

要定义类模板的成员函数，必须将其指定为模板，并且使用类模板对类型进行限定。因此，`Stack<T>` 类型的成员函数 `push()` 实现如下所示：

```
1 template<typename T>
2 void Stack<T>::push (T const& elem)
3 {
4     elems.push_back(elem); // append copy of passed elem
5 }
```

使用 `vector` 的 `push_back()`，将元素添加到 `vector` 的末尾。

注意，`vector` 的 `pop_back()` 会删除最后一个元素，但不返回这个元素，这种行为是异常安全的。使用 `pop()` 的完全异常安全版本，来返回删除元素是不可能的(这个问题由 Tom Cargill 在 [CargillExceptionSafety] 中首次提出，并在 [sutterexception] 中的第 10 项中进行讨论)。这里可以忽略这个危险，可以实现 `pop()` 返回刚刚删除的元素。为此，只需使用 `T` 声明元素类型的局部变量即可：

```
1 template<typename T>
2 T Stack<T>::pop ()
3 {
4     assert(!elems.empty());
5     T elem = elems.back(); // save copy of last element
6     elems.pop_back(); // remove last element
7     return elem; // return copy of saved element
8 }
```

当 `vector` 中没有元素时，`back()`(返回最后一个元素) 和 `pop_back()`(删除最后一个元素) 会导致未定义行为，所以先检查堆栈是否为空。如果为空，则触发断言，因为在空堆栈上调用 `pop()` 是一个使用错误。`top()` 也是这样，当试图删除一个不存在的 `top` 元素时，`top()` 会返回但不删除 `top` 元素：

```

1 template<typename T>
2 T const& Stack<T>::top () const
3 {
4     assert(!elems.empty());
5     return elems.back(); // return last element
6 }

```

当然，对于成员函数，可以在类声明中以内联的形式，实现类模板的成员函数。例如：

```

1 template<typename T>
2 class Stack {
3     ...
4     void push (T const& elem) {
5         elems.push_back(elem); // append copy of passed elem
6     }
7     ...
8 };

```

2.2. 使用栈类模板

C++17 前，要使用类模板的对象，必须显式指定模板参数。

C++17 引入了类参数模板推导，若模板参数可以从构造函数派生，则可以跳过这些参数。
这会在第 2.9 节中进行讨论。

下面的例子展示了如何使用类模板 Stack<>：

basics/stack1test.cpp

```

1 #include "stack1.hpp"
2 #include <iostream>
3 #include <string>
4
5 int main()
6 {
7     Stack<int> intStack; // stack of ints
8     Stack<std::string> stringStack; // stack of strings
9
10    // manipulate int stack
11    intStack.push(7);
12    std::cout << intStack.top() << '\n';
13
14    // manipulate string stack
15    stringStack.push("hello");
16    std::cout << stringStack.top() << '\n';
17    stringStack.pop();
18 }

```

通过声明类型 `Stack<int>`, `int` 在类模板中作为类型 `T`。因此, `intStack` 是一个对象, 使用的是 `vector<int>` 类型, 并且调用相应的成员函数。类似地, 通过声明和使用 `Stack<std::string>`, 将创建使用 `vector<std::string>` 的对象, 使用相应的成员函数。

注意, 代码只对调用的模板(成员)函数实例化。对于类模板, 只有在使用成员函数时才实例化。当然, 这节省了时间和空间, 并且只允许使用部分地类模板, 这会在 2.3 节中详细讨论。

例子中, 默认构造函数 `push()` 和 `top()` 为 `int` 和 `string` 实例化, 但 `pop()` 只对 `string` 实例化。如果类模板具有静态成员, 则对于使用类模板的每个类型实例, 这些成员也会实例化一次。

可以像使用其他类型一样使用实例化的类模板类型。可以使用 `const`、`volatile` 或从中派生数组和引用类型对其进行限定。也可以将其作为类型定义的一部分, 使用 `typedef` 或 `using`(请参阅第 2.8 节了解关于类型定义的详细信息), 或者在构建另一个模板类型时将其用作类型参数。例如:

```
1 void foo(Stack<int> const& s) // parameter s is int stack
2 {
3     using IntStack = Stack<int>; // IntStack is another name for Stack<int>
4     Stack<int> istack[10]; // istack is array of 10 int stacks
5     IntStack istack2[10]; // istack2 is also an array of 10 int stacks (same type)
6     ...
7 }
```

模板参数可以是任何类型, 例如 `float` 指针, 甚至是 `Stack<int>`:

```
1 Stack<float*> floatPtrStack; // stack of float pointers
2 Stack<Stack<int>> intStackStack; // stack of stack of ints
```

这里的要求是, 需要这种类型支持所使用的操作。

注意, 在 C++11 之前, 必须在两个模板右括号之间放空格:

```
1 Stack<Stack<int> > intStackStack; // 所有C++版本都可以用
```

如果没有这样做, 就使用 `>>`, 会导致语法错误:

```
1 Stack<Stack<int>> intStackStack; // C++11之前会报错
```

旧行为可以帮助 C++ 编译器对独立于代码语义源码进行标记。然而, 由于缺少空格是一个错误, 这需要相应的错误消息, 因此无论如何都必须考虑代码的语义。因此, 在 C++11 中, 在两个模板右括号之间放一个空格的规则被“尖括号黑客”删除了(详见 13.3.1 节)。

2.3. 部分使用栈类模板

类模板通常对其实例化的模板参数使用多个操作(包括构造和析构)。这可能会使, 这些模板参数必须为所有成员函数, 提供所有必要的操作, 而模板参数只需要提供所需的必要操作(而不是可能需要)即可。

例如, 类 `Stack<>` 提供成员函数 `printOn()` 打印堆栈中的全部内容, 并对每个元素使用操作符 `<<`:

```
1 template<typename T>
2 class Stack {
3     ...
```

```
4 void printOn(std::ostream& strm) const {
5     for (T const& elem : elems) {
6         strm << elem << ' ';// 对每个元素使用<<
7     }
8 }
9 };
```

对于没有定义操作符 `<<` 的类型，仍然可以使用这个类：

```
1 Stack<std::pair<int,int>> ps; // 注意：std::pair<>不支持<<
2 ps.push({4, 5}); // OK
3 ps.push({6, 7}); // OK
4 std::cout << ps.top().first << '\n'; // OK
5 std::cout << ps.top().second << '\n'; // OK
```

只在使用 `printOn()` 时代码才会出错，因为不能实例化特定元素类型的操作符 `<<`：

```
1 ps.printOn(std::cout); // ERROR: 元素类型不支持操作符<<
```

2.3.1 概念

先来看一个问题：如何知道模板需要哪些操作才能实例化？“概念”通常用来表示模板库中需要的约束。例如，C++ 标准库依赖于随机访问迭代器和默认可构造函数等概念。

目前（如 C++17），概念只能通过文字进行表达（如代码注释）。这可能会成为一个严重的问题，因为不遵守约束可能会导致大量的错误消息（参见 9.4）。

多年来，也有一些方法和试验支持将概念定义和验证为一种语言特性。然而，到 C++17 为止，还这样的方法还没标准化。

从 C++11 开始，可以通过使用 `static_assert` 和预定义类型特征来检查约束。例如：

```
1 template<typename T>
2 class C
3 {
4     static_assert(std::is_default_constructible<T>::value,
5                  "Class C requires default-constructible elements");
6     ...
7 };
```

使用默认构造函数，若没有这个断言，编译会失败。然而，错误消息描述的是整个模板实例化的过程，从最初的实例化原因到检测到错误的实际模板的定义（参见章节 9.4）。

但需要更复杂的代码来检查，例如：T 类型的对象是否提供了特定的成员函数，或者是否有可用的小于操作符。相关的示例，请参见第 19.6.3 节。

有关 C++ 概念的详细讨论，请参阅附录 E。

2.4. 友元

使用 `printOn()` 打印堆栈内容，不如为堆栈实现 `<<` 操作符。然而，通常 `<<` 操作符会实现为非成员函数，然后内联调用 `printOn()`：

```

1 template<typename T>
2 class Stack {
3 ...
4     void printOn(std::ostream& strm) const {
5         ...
6     }
7     friend std::ostream& operator<< (std::ostream& strm,
8     Stack<T> const& s) {
9         s.printOn(strm);
10    return strm;
11 }
12 };

```

这意味着用于类 `Stack<T>` 的 `<<` 操作符不是一个函数模板，而是在需要时用类模板实例化的“普通”函数。

其是一个模板实体，参见 12.1 节。

但当试图声明友元函数并实现时，事情会变得更加复杂。这里有两种选择：

1. 隐式声明一个新的函数模板，但要使用不同的模板参数，比如 `U`:

```

1 template<typename T>
2 class Stack {
3 ...
4     template<typename U>
5     friend std::ostream& operator<< (std::ostream&, Stack<U> const&);
6 };

```

再次使用 `T` 或跳过模板参数都不起作用 (要么内部的 `T` 隐藏外部的 `T`，要么在命名空间作用域中声明一个非模板函数)。

2. 可以将 `Stack<T>` 的输出操作符转发声明为模板，这样就需要转发声明 `Stack<T>`:

```

1 template<typename T>
2 class Stack;
3 template<typename T>
4 std::ostream& operator<< (std::ostream&, Stack<T> const&);

```

然后，将该函数声明为友元：

```

1 template<typename T>
2 class Stack {
3 ...
4     friend std::ostream& operator<< <T> (std::ostream&,
5     Stack<T> const&);
6 };

```

注意“函数名”操作符 `<<` 后面的 `<T>`。因此，需要将非成员函数模板的特化声明为友元。若没有 `<T>`，需要声明新的非模板函数。详细信息请参见 12.5.2。

其他情况下，可以对没有定义 << 操作符的元素使用这个类。只有对这个堆栈调用操作符 << 时才会出现错误：

```
1 Stack<std::pair<int,int>> ps; // std::pair<> has no operator<< defined
2 ps.push({4, 5}); // OK
3 ps.push({6, 7}); // OK
4 std::cout << ps.top().first << '\n'; // OK
5 std::cout << ps.top().second << '\n'; // OK
6 std::cout << ps << '\n'; // ERROR: operator<< not supported
```

2.5. 类模板的特化

可以为某些模板参数特化类模板。类似于函数模板的重载(参见第 1.5 节)，特化类模板允许优化特定类型的实现，或者为类模板的实例化修复特定类型的错误行为。但若特化类模板，则必须特化所有成员函数。虽然可以特化类模板的单个成员函数，但若这样做了，就不能再特化该特化成员所属的整个类模板实例。

要特化类模板，必须用 template<> 和类模板特化的类型声明类。这些类型可用作模板参数，必须直接在类名之后指定：

```
1 template<>
2 class Stack<std::string> {
3 ...
4 };
```

对于这些特化，成员函数的定义必须是一个“普通的”成员函数，每次出现 T 都会使用特化类型替换：

```
1 void Stack<std::string>::push (std::string const& elem)
2 {
3     elems.push_back(elem); // append copy of passed elem
4 }
```

下面是 std::string 类型的 Stack<> 特化：

basics/stack2.hpp

```
1 #include "stack1.hpp"
2 #include <deque>
3 #include <string>
4 #include <cassert>
5
6 template<>
7 class Stack<std::string> {
8 private:
9     std::deque<std::string> elems; // elements
10 public:
11     void push(std::string const&); // push element
12     void pop(); // pop element
```

```

13     std::string const& top() const; // return top element
14     bool empty() const { // return whether the stack is empty
15         return elems.empty();
16     }
17 };
18
19 void Stack<std::string>::push (std::string const& elem)
20 {
21     elems.push_back(elem); // append copy of passed elem
22 }
23
24 void Stack<std::string>::pop ()
25 {
26     assert(!elems.empty());
27     elems.pop_back(); // remove last element
28 }
29
30 std::string const& Stack<std::string>::top () const
31 {
32     assert(!elems.empty());
33     return elems.back(); // return last element
34 }

```

例子中，特化使用引用语义将字符串参数传递给 push()，这对这个特定类型有意义（不过，应该更好地传递转发引用，将在第 6.1 节中讨论）。

另一个区别是使用 deque(而非 vector) 来管理堆栈内的元素。尽管这在这里没有什么区别，但说明了特化实现可能与主模板实现完全不同。

2.6. 偏特化

类模板可偏特化。可以为特定的情况提供特殊的实现，但有些模板参数仍需要由用户定义。例如，可以为指针定义 Stack<> 类的特殊实现：

basics/stackpartspec.hpp

```

1 #include "stack1.hpp"
2
3 // partial specialization of class Stack<> for pointers:
4 template<typename T>
5 class Stack<T*> {
6 private:
7     std::vector<T*> elems; // elements
8
9 public:
10    void push(T*); // push element
11    T* pop(); // pop element
12    T* top() const; // return top element
13    bool empty() const { // return whether the stack is empty

```

```

14     return elems.empty();
15 }
16 };
17
18 template<typename T>
19 void Stack<T*>::push (T* elem)
20 {
21     elems.push_back(elem); // append copy of passed elem
22 }
23
24 template<typename T>
25 T* Stack<T*>::pop ()
26 {
27     assert(!elems.empty());
28     T* p = elems.back();
29     elems.pop_back(); // remove last element
30     return p; // and return it (unlike in the general case)
31 }
32
33 template<typename T>
34 T* Stack<T*>::top () const
35 {
36     assert(!elems.empty());
37     return elems.back(); // return copy of last element
38 }
```

```

1 template<typename T>
2 class Stack<T*> {
3 }
```

先定义了一个类模板，仍然参数化了 T，但特化了指针 (Stack<T*>)。

特化可能提供 (略微) 不同的接口。例如，这里的 pop() 返回存储的指针，当用 new 创建值时，用户就可以使用 delete 删除：

```

1 Stack<int*> ptrStack; // stack of pointers (special implementation)
2
3 ptrStack.push(new int{42});
4 std::cout << *ptrStack.top() << '\n';
5 delete ptrStack.pop();
```

多个参数的偏特化

类模板也可以特化多个模板参数之间的关系。例如，对于以下类模板：

```

1 template<typename T1, typename T2>
2 class MyClass {
3     ...
4 };
```

以下几种可能的偏特化：

```

1 // partial specialization: both template parameters have same type
2 template<typename T>
3 class MyClass<T,T> {
4     ...
5 };
6
7 // partial specialization: second type is int
8 template<typename T>
9 class MyClass<T,int> {
10    ...
11 };
12
13 // partial specialization: both template parameters are pointer types
14 template<typename T1*, typename T2*>
15 class MyClass<T1*,T2*> {
16     ...
17 };

```

下面的例子显示了哪个声明使用了哪个模板:

```

1 MyClass<int,float> mif; // uses MyClass<T1,T2>
2 MyClass<float,float> mff; // uses MyClass<T,T>
3 MyClass<float,int> mfi; // uses MyClass<T,int>
4 MyClass<int*,float*> mp; // uses MyClass<T1*,T2*>

```

若多个偏特化都匹配调用，则声明有歧义:

```

1 MyClass<int,int> m; // ERROR: matches MyClass<T,T>
2                         // and MyClass<T,int>
3 MyClass<int*,int*> m; // ERROR: matches MyClass<T,T>
4                         // and MyClass<T1*,T2*>

```

要解决歧义，可以为相同类型的指针提供偏特化实现:

```

1 template<typename T>
2 class MyClass<T*,T*> {
3     ...
4 };

```

有关偏特化的详细信息，请参见第 16.4 节。

2.7. 类模板的默认参数

对于函数模板，可以为类模板参数设置默认值。例如，在 `Stack<>` 类中，可以将用于管理元素的容器定义为第二个模板参数，并使用 `std::vector<>` 作为默认值:

basics/stack3.hpp

```

1 #include <vector>
2 #include <cassert>

```

```

3
4 template<typename T, typename Cont = std::vector<T>>
5 class Stack {
6 private:
7     Cont elems; // elements
8
9 public:
10    void push(T const& elem); // push element
11    void pop(); // pop element
12    T const& top() const; // return top element
13    bool empty() const { // return whether the stack is empty
14        return elems.empty();
15    }
16 };
17
18 template<typename T, typename Cont>
19 void Stack<T,Cont>::push (T const& elem)
20 {
21     elems.push_back(elem); // append copy of passed elem
22 }
23
24 template<typename T, typename Cont>
25 void Stack<T,Cont>::pop ()
26 {
27     assert(!elems.empty());
28     elems.pop_back(); // remove last element
29 }
30
31 template<typename T, typename Cont>
32 T const& Stack<T,Cont>::top () const
33 {
34     assert(!elems.empty());
35     return elems.back(); // return last element
36 }
```

现在有了两个模板参数，因此成员函数的每个定义都必须用这两个参数定义：

```

1 template<typename T, typename Cont>
2 void Stack<T,Cont>::push (T const& elem)
3 {
4     elems.push_back(elem); // append copy of passed elem
5 }
```

可以像以前那样使用这个堆栈。若将第一个也是唯一的参数作为元素类型传递，则会使用 `vector` 来管理该类型的元素：

```

1 template<typename T, typename Cont = std::vector<T>>
2 class Stack {
3 private:
4     Cont elems; // elements
5     ...
```

```
6 };
```

当在程序中声明 Stack 对象时，可以指定元素的容器类型：

basics/stack3test.cpp

```
1 #include "stack3.hpp"
2 #include <iostream>
3 #include <deque>
4
5 int main()
{
6
7     // stack of ints:
8     Stack<int> intStack;
9
10    // stack of doubles using a std::deque<> to manage the elements
11    Stack<double, std::deque<double>> dblStack;
12
13    // manipulate int stack
14    intStack.push(7);
15    std::cout << intStack.top() << '\n';
16    intStack.pop();
17
18    // manipulate double stack
19    dblStack.push(42.42);
20    std::cout << dblStack.top() << '\n';
21    dblStack.pop();
22}
```

```
1 Stack<double, std::deque<double>>
```

上面的代码声明一个 double 数的栈，使用 std::deque<> 在内部管理元素。

2.8. 类型别名

通过为整个类型定义一个新名称，可以使类模板使用起来更简单。

类型定义和别名声明

有两种方法类型定义新名称：

1. 通过使用关键字 `typedef`：

```
1 typedef Stack<int> IntStack; // typedef
2 void foo (IntStack const& s); // s is stack of ints
3 IntStack istack[10]; // istack is array of 10 stacks of ints
```

`typedef` 产生的名称称为 `typedef-name`。

这里使用了 `typedef`, 而不是“类型定义”。其实, 关键字 `typedef` 最初是用来建议“类型定义”的。在 C++ 中, “类型定义”实际上意味着其他类型(例如, 类或枚举类型的定义)。相反, 应该将 `typedef` 视为现有类型的替代名称(“别名”), 从而使用 `typedef`。

2. 通过使用关键字 `using`(C++11 起):

```
1 using IntStack = Stack<int>; // alias declaration
2 void foo (IntStack const& s); // s is stack of ints
3 IntStack istack[10]; // istack is array of 10 stacks of ints
```

由 [dosreismarcusaliastemplate] 引入, 称为别名声明。

两种情况下, 都会为现有类型定义新名称。因此,

```
1 typedef Stack<int> IntStack;
```

或

```
1 using IntStack = Stack<int>;
```

`IntStack` 和 `Stack<int>` 是同一类型的两种名称。

作为为现有类型定义新名称的通用术语, 可以用其来声明别名。因此, 新名称就是原类型的别名。

将已定义的类型名放在等号左边, 阅读起来更容易, 所以在本书声明类型别名时, 更倾向使用别名声明语法。

模板别名

与 `typedef` 不同, 别名声明可以模板化。自 C++11 起, 这种方式就称为别名模板。

别名模板有时(不正确地)会称为 `typedef` 模板, 若将其制成模板, 将与 `typedef` 的行为相同。

别名模板 `DequeStack`, 对元素类型 `T` 进行参数化, 展开为一个栈, 将其元素实例存储在 `std::deque`:

```
1 template<typename T>
2 using DequeStack = Stack<T, std::deque<T>>;
```

因此, 类模板和别名模板都可以作为参数化类型。但别名模板只是为现有类型提供一个新名称, 该类型仍然可以使用。`DequeStack<int>` 和 `Stack<int, std::deque<int>>` 表示的就是相同类型。

通常情况下, 模板只能在全局/命名空间作用域或类声明内部声明和定义。

成员类型的别名模板

别名模板对于制作类模板成员类型的快捷方式特别合适。比如:

```
1 template<typename T> struct MyType {
2     typedef ... iterator;
3     ...
4 };
```

或

```
1 template<typename T> struct MyType {  
2     using iterator = ...;  
3     ...  
4 };
```

可以这样的定义

```
1 template<typename T>  
2 using MyTypeIterator = typename MyType<T>::iterator;
```

也可以这样

```
1 MyTypeIterator<T> pos;
```

而不像下面这样:

```
1 typename MyType<T>::iterator pos;
```

因为成员是类型，所以 `typename` 是必要的。参见 5.1 节。

类型特征后缀`_t`

C++14 后，标准库使用这种方式为库中的所有类型定义了快捷方式。比如：

```
1 std::add_const_t<T> // since C++14
```

而非

```
1 typename std::add_const<T>::type // since C++11
```

标准库定义：

```
1 namespace std {  
2     template<typename T> using add_const_t = typename add_const<T>::type;  
3 }
```

2.9. 类模板的参数推导

C++17 前，必须将所有模板参数类型传递给类模板（除非有默认值）。C++17 后，指定模板参数的约束放宽了。相反，若构造函数能够推导出所有模板参数（没有默认值），则可以不用显式定义模板参数。

例如，前面所有的代码示例中，可以使用复制构造函数，而非指定模板参数：

```
1 Stack<int> intStack1; // stack of ints  
2 Stack<int> intStack2 = intStack1; // OK in all versions  
3 Stack intStack3 = intStack1; // OK since C++17
```

通过接受参数的构造函数，支持推导堆栈的元素类型。例如，提供由单个元素初始化的堆栈：

```
1 template<typename T>
2 class Stack {
3 private:
4     std::vector<T> elems; // elements
5 public:
6     Stack () = default;
7     Stack (T const& elem) // initialize stack with one element
8     : elems({elem}) {
9     }
10    ...
11};
```

可以这样声明栈:

```
1 Stack intStack = 0; // Stack<int> deduced since C++17
```

使用整数 0 初始化堆栈，模板参数 T 推导为 int，因此会实例化一个 stack<int> 类型。

注意以下几点:

- 根据 int 构造函数模板的定义，必须请求默认构造函数以其默认行为可用。因为只有在没有定义其他构造函数时，默认构造函数才可用:

```
1 Stack() = default;
```

- 参数 elem 传递给带大括号的 elems，以初始化初始化列表，以 elem 作为唯一参数:

```
1 : elems({elem})
```

vector 的构造函数中无法接受单个参数作为初始元素。

更糟糕的是，vector 的构造函数会将一个整型参数作为初始大小，因此对于初始值为 5 的堆栈，当使用:elems(elem) 时，vector 将获得 5 个元素的初始大小。

与函数模板不同，类模板参数不能只推导部分参数(通过显式地只指定部分模板参数)。详细信息请参见 15.12。

用字符串字面值推导类模板参数

原则上，可以用字符串字面值初始化堆栈:

```
1 Stack stringStack = "bottom"; // Stack<char const[7]> deduced since C++17
```

但会带来很多麻烦，通过引用传递模板类型 T 的参数时，参数不会衰变(衰变是指将原始数组类型转换为相应的原始指针类型的机制)。这意味着会初始化了一个实例

```
1 Stack<char const[7]>
```

因为，在使用 T 的地方使用了 char const[7]。例如，不能 push 不同大小的字符串，因为其有不同的类型。相关的详细讨论，请参阅第 7.4 节。

但当按值传递模板类型 T 的参数时，参数会衰变，将原始数组类型转换为相应的原始指针类型。也就是说，构造函数的调用参数 T 推导为 `char const*`，因此整个类将推导为 `Stack<char const*>`。由于这个原因，可以声明构造函数，直接传递值：

```
1 class Stack {  
2     private:  
3         std::vector<T> elems; // elements  
4     public:  
5         Stack (T elem) // initialize stack with one element by value  
6         : elems({elem}) { // to decay on class ttmpl arg deduction  
7     }  
8     ...  
9 };
```

有了这个，下面的初始化就可以工作了：

```
1 Stack stringStack = "bottom"; // Stack<char const*> deduced since C++17
```

在这种情况下，最好将临时元素移到堆栈中，以避免不必要的复制：

```
1 template<typename T>  
2 class Stack {  
3     private:  
4         std::vector<T> elems; // elements  
5     public:  
6         Stack (T elem) // initialize stack with one element by value  
7         : elems({std::move(elem)}) {  
8     }  
9     ...  
10 };
```

推导策略

除了声明构造函数由值调用之外，还有一种不同的解决方案：在容器中处理指针是噩梦的开始，所以应该禁止为容器类自动推导字符指针。

可以定义特定的推导策略来提供额外的辅助，或修复现有类模板参数的推导。例如，可以定义只要传入一个字符串字面值或 C 字符串，栈就会实例化为 `std::string`：

```
1 Stack(char const*) -> Stack<std::string>;
```

这个策略必须出现在与类定义相同的作用域（命名空间）中，它遵循类定义。将-> 后面的类型称为推导策略的引导类型。

现在，声明

```
1 Stack stringStack{"bottom"}; // OK: Stack<std::string> deduced since C++17
```

将堆栈推断为一个 `Stack<std::string>` 类。然而，在以下情况下仍然不能正常工作：

```
1 Stack stringStack = "bottom"; // Stack<std::string> deduced, but still not valid
```

推导出 `std::string`，以便实例化一个 `Stack<std::string>` 类：

```
1 class Stack {
2     private:
3         std::vector<std::string> elems; // elements
4     public:
5         Stack (std::string const& elem) // initialize stack with one element
6             : elems({elem}) {
7         }
8         ...
9     };

```

但根据语言规则，不能通过将字符串字面值传递给 `std::string` 的构造函数来复制对象，从而完成初始化。所以必须像这样初始化堆栈：

```
1 Stack stringStack{"bottom"}; // Stack<std::string> deduced and valid
```

此时的疑问在于，类模板参数推导的副本。将 `stringStack` 声明为 `Stack<std::string>` 之后，下面的初始化声明了相同的类型（因此，调用了复制构造函数），而不是通过字符串堆栈元素对堆栈进行初始化：

```
1 Stack stack2{stringStack}; // Stack<std::string> deduced
2 Stack stack3(stringStack); // Stack<std::string> deduced
3 Stack stack4 = {stringStack}; // Stack<std::string> deduced
```

有关类模板参数推导的更多细节，请参见 15.12 节。

2.10. 模板聚合

聚合类（不由用户提供、显式或继承的构造函数的类/结构，没有 `private` 或 `protected` 的非静态数据成员，没有虚函数，也没有 `virtual`、`private` 或 `protected` 基类）也可以是模板。例如：

```
1 template<typename T>
2 struct ValueWithComment {
3     T value;
4     std::string comment;
5 };
```

定义一个聚合，参数化了 `value` 的类型。可以像其他类模板一样声明对象，并且可以以聚合的方式进行使用：

```
1 ValueWithComment<int> vc;
2 vc.value = 42;
3 vc.comment = "initial value";
```

C++17 后，甚至可以为聚合类模板定义推导策略：

```
1 ValueWithComment(char const*, char const*)
2     -> ValueWithComment<std::string>;
3 ValueWithComment vc2 = {"hello", "initial value"};
```

因为 `ValueWithComment` 没有用于执行推导的构造函数，所以若没有推导策略，初始化将不可能完成。

标准库类 `std::array`< >也是一个聚合，参数化了元素类型和大小。C++17 标准库还为其定义了推导策略，这将在 4.4.4 节中讨论。

2.11. 总结

- 类模板是在实现时保留一个或多个类型参数的类。
- 要使用类模板，需要将类型作为模板参数传递。并为这些类型，实例化（并编译）类模板。
- 对于类模板，只有调用的成员函数会实例化。
- 可以为某些类型特化类模板。
- 可以偏特化某些类型的类模板。
- C++17 后，可以从构造函数中自动推导出类模板的参数。
- 可以定义聚合类模板。
- 若声明为按值调用，则模板类型的调用参数会衰变。
- 模板只能在全局/命名空间作用域或类声明内部声明和定义。

第3章 非类型模板参数

对于函数和类模板来说，模板参数可以是类型，也可以是普通值。与使用类型参数的模板一样，定义在使用之前。使用这样的模板时，必须显式地指定值。然后，实例化生成的代码。本章演示了新版栈类模板的特性。此外，还会展示非类型函数模板参数的示例，并讨论了这种技术的限制。

3.1. 类模板参数

与前几章的栈实现不同，这里可以通过使用固定大小的元素数组来实现栈。这种方法的优点是避免了内存管理开销，但这为堆栈确定最佳容量带来了困难。指定的容量越小，堆栈越有可能快速填满。指定的容量越大，有可能会浪费内存。好的解决方案是让堆栈的用户指定数组的大小，作为堆栈的最大容量。

为此，可以将 size 定义为模板参数：

basics/stacknontype.hpp

```
1 #include <array>
2 #include <cassert>
3
4 template<typename T, std::size_t Maxsize>
5 class Stack {
6 private:
7     std::array<T,Maxsize> elems; // elements
8     std::size_t numElems; // current number of elements
9 public:
10    Stack(); // constructor
11    void push(T const& elem); // push element
12    void pop(); // pop element
13    T const& top() const; // return top element
14    bool empty() const { // return whether the stack is empty
15        return numElems == 0;
16    }
17    std::size_t size() const { // return current number of elements
18        return numElems;
19    }
20};
21
22 template<typename T, std::size_t Maxsize>
23 Stack<T,Maxsize>::Stack ()
24 : numElems(0) // start with no elements
25 {
26     // nothing else to do
27 }
28
29 template<typename T, std::size_t Maxsize>
30 void Stack<T,Maxsize>::push (T const& elem)
31 {
```

```

32     assert(numElems < Maxsize);
33     elems[numElems] = elem; // append element
34     ++numElems; // increment number of elements
35 }
36
37 template<typename T, std::size_t Maxsize>
38 void Stack<T,Maxsize>::pop ()
39 {
40     assert(!empty());
41     --numElems; // decrement number of elements
42 }
43
44 template<typename T, std::size_t Maxsize>
45 T const& Stack<T,Maxsize>::top () const
46 {
47     assert(!empty());
48     return elems[numElems-1]; // return last element
49 }
```

第二个模板形参 Maxsize 是 std::size_t 类型，指定了堆栈元素内部数组的容量：

```

1 template<typename T, std::size_t Maxsize>
2 class Stack {
3 private:
4     std::array<T,Maxsize> elems; // elements
5     ...
6 };
```

另外，在 push() 中，可以使用它来检查堆栈是否已满：

```

1 template<typename T, std::size_t Maxsize>
2 void Stack<T,Maxsize>::push (T const& elem)
3 {
4     assert(numElems < Maxsize);
5     elems[numElems] = elem; // append element
6     ++numElems; // increment number of elements
7 }
```

要使用这个类模板，必须同时指定元素类型和最大容量：

basics/stacknontype.cpp

```

1 #include "stacknontype.hpp"
2 #include <iostream>
3 #include <string>
4
5 int main()
6 {
7     Stack<int,20> int20Stack; // stack of up to 20 ints
8     Stack<int,40> int40Stack; // stack of up to 40 ints
9     Stack<std::string,40> stringStack; // stack of up to 40 strings
```

```

10
11 // manipulate stack of up to 20 ints
12 int20Stack.push(7);
13 std::cout << int20Stack.top() << '\n';
14 int20Stack.pop();
15
16 // manipulate stack of up to 40 strings
17 stringStack.push("hello");
18 std::cout << stringStack.top() << '\n';
19 stringStack.pop();
20 }

```

每个模板实例化都是独立的类型。因此，int20Stack 和 int40Stack 是两种不同的类型。不能使用其中一个来代替另一个，也不能将其中一个分配给另一个。

同样，可以指定模板参数的默认值：

```

1 template<typename T = int, std::size_t Maxsize = 100>
2 class Stack {
3     ...
4 };

```

但从设计的角度来看，这样的写法在本例中可能不太合适。默认参数是正确的，但对于一般的堆栈类型来说，int 类型和最大容量为 100 似乎都不直观。因此，最好显式地指定这两个值，以便在声明期间进行记录。

3.2. 函数模板参数

还可以为函数模板定义非类型参数。例如，下面的函数模板定义了一组函数，可以为其加一个数：

basics/addvalue.hpp

```

1 template<int Val, typename T>
2 T addValue (T x)
3 {
4     return x + Val;
5 }

```

如果函数或操作当作参数，这些类型的函数可能很有用。例如使用 C++ 标准库，可以通过函数模板的实例化来给集合的每个元素加一个数：

```

1 std::transform (source.begin(), source.end(), // start and end of source
2 dest.begin(), // start of destination
3 addValue<5,int>); // operation

```

最后一个参数实例化函数模板 addValue<5,int>，将 5 加到传入的 int 值上。对 source 中的每个元素调用生成的函数，同时将其转换为目标 dest。

注意，必须为模板参数 `addValue<>()` 指定为 `int`。推导仅适用于即时调用，而 `std::transform()` 需要完整的类型来推导其第四个参数的类型。不支持只替换/推导一些模板参数，并查看哪些参数适合，然后推导剩下的参数。

同样，也可以指定模板参数是从前面的参数推导出来的。例如，从传递的非类型派生返回类型：

```
1 template<auto Val, typename T = decltype(Val)>
2 T foo();
```

或者确保传递的值与传递的类型相同：

```
1 template<typename T, T Val = T{}>
2 T bar();
```

3.3. 模板参数的限制

非类型模板参数有一些限制，只能是整型常量值（包括枚举），指向对象/函数/成员的指针，指向对象或函数的左值引用，或者 `std::nullptr_t(nullptr)` 的类型。

浮点数和类型对象不允许作为非类型模板参数：

```
1 template<double VAT> // ERROR: floating-point values are not
2 double process (double v) // allowed as template parameters
3 {
4     return v * VAT;
5 }
6
7 template<std::string name> // ERROR: class-type objects are not
8 class MyClass { // allowed as template parameters
9 ...
10 };
```

当向指针或引用传递模板参数时，对象不能是字符串字面值、临时对象或数据成员和其他子对象。在 C++17 前，每个 C++ 版本都放宽了这些限制，所以还有其他的限制：

- C++11 中，对象必须具有外部链接。
- C++14 中，对象必须具有外部或内部链接。

因此，以下情况会报错：

```
1 template<char const* name>
2 class Message { // OK
3 ...
4 };
5
6 Message<"hello"> x; // ERROR: string literal "hello" not allowed
```

但也有解决方法（同样取决于 C++ 的版本）：

```
1 extern char const s03[] = "hi"; // external linkage
2 char const s11[] = "hi"; // internal linkage
3 int main()
```

```

4 {
5     Message<s03> m03; // OK (all versions)
6     Message<s11> m11; // OK since C++11
7     static char const s17[] = "hi"; // no linkage
8     Message<s17> m17; // OK since C++17
9 }

```

三种情况下，常量字符数组都由”hi” 初始化，该对象用作用 `char const*` 声明的模板参数。若对象具有外部链接 (s03)，这在所有 C++ 版本中都有效；若对象具有内部链接 (s11)，则在 C++11 和 C++14 中也是有效的；若对象没有链接，则需要 C++17 的支持。

请参阅第 12.3.3 节和第 17.2 节讨论了该领域在未来变化的可能。

避免无效的表达式

非类型模板参数可以是编译时表达式。例如：

```

1 template<int I, bool B>
2 class C;
3 ...
4 C<sizeof(int) + 4, sizeof(int)==4> c;

```

但是，若在表达式中使用 `>`，必须将整个表达式放入圆括号中，以便编译器确定 `>` 在哪里结束：

```

1 C<42, sizeof(int) > 4> c; // ERROR: first > ends the template argument list
2 C<42, (sizeof(int) > 4)> c; // OK

```

3.4. 模板参数类型 `auto`

C++17 后，可以定义非类型模板参数。使用这个特性，可以实现一个确定大小的堆栈类：

basics/stackauto.hpp

```

1 #include <array>
2 #include <cassert>
3
4 template<typename T, auto Maxsize>
5 class Stack {
6     public:
7         using size_type = decltype(Maxsize);
8     private:
9         std::array<T,Maxsize> elems; // elements
10        size_type numElems; // current number of elements
11    public:
12        Stack(); // constructor
13        void push(T const& elem); // push element
14        void pop(); // pop element
15        T const& top() const; // return top element
16        bool empty() const { // return whether the stack is empty
17            return numElems == 0;

```

```

18    }
19    size_type size() const { // return current number of elements
20        return numElems;
21    }
22};

23

24// constructor
25template<typename T, auto Maxsize>
26Stack<T,Maxsize>::Stack ()
27: numElems(0) // start with no elements
28{
29    // nothing else to do
30}
31

32template<typename T, auto Maxsize>
33void Stack<T,Maxsize>::push (T const& elem)
34{
35    assert(numElems < Maxsize);
36    elems[numElems] = elem; // append element
37    ++numElems; // increment number of elements
38}

39

40template<typename T, auto Maxsize>
41void Stack<T,Maxsize>::pop ()
42{
43    assert(!empty());
44    --numElems; // decrement number of elements
45}

46

47template<typename T, auto Maxsize>
48T const& Stack<T,Maxsize>::top () const
49{
50    assert(!empty());
51    return elems[numElems-1]; // return last element
52}

```

定义

```

1 template<typename T, auto Maxsize>
2 class Stack {
3     ...
4 };

```

通过使用占位符类型 `auto`, 可以将 `Maxsize` 定义为尚未指定类型的值, 可以是任何允许为非类型模板参数的类型。

内部可以同时使用这两个值:

```

1 std::array<T,Maxsize> elems; // elements

```

其类型为:

```
1 using size_type = decltype(Maxsize);
```

例如，用作 size() 成员函数的返回类型：

```
1 size_type size() const { // return current number of elements
2     return numElems;
3 }
```

C++14 后，也可以在这里使用 auto 作为返回类型，让编译器确定返回类型：

```
1 auto size() const { // return current number of elements
2     return numElems;
3 }
```

通过这个类声明，当使用堆栈时，元素数量由使用类型定义：

basics/stackauto.cpp

```
1 #include <iostream>
2 #include <string>
3 #include "stackauto.hpp"
4
5 int main()
6 {
7     Stack<int,20u> int20Stack; // stack of up to 20 ints
8     Stack<std::string,40> stringStack; // stack of up to 40 strings
9
10    // manipulate stack of up to 20 ints
11    int20Stack.push(7);
12    std::cout << int20Stack.top() << '\n';
13    auto size1 = int20Stack.size();
14
15    // manipulate stack of up to 40 strings
16    stringStack.push("hello");
17    std::cout << stringStack.top() << '\n';
18    auto size2 = stringStack.size();
19
20    if (!std::is_same_v<decltype(size1), decltype(size2)>) {
21        std::cout << "size types differ" << '\n';
22    }
23 }
```

```
1 Stack<int,20u> int20Stack; // stack of up to 20 ints
```

因为传递了 20u，所以内部大小类型是 unsigned int。

```
1 Stack<std::string,40> stringStack; // stack of up to 40 strings
```

因为传递了 40，内部大小类型是 int。

两个堆栈的 size() 将有不同的返回类型，因此

```
1 auto size1 = int2Stack.size();
2 ...
3 auto size2 = stringStack.size();
```

size1 和 size2 的类型不同。可以使用标准类型特征 std::is_same(参见 D.3.3 节) 和 decltype，进行检查：

```
1 if (!std::is_same<decltype(size1), decltype(size2)>::value) {
2     std::cout << "size types differ" << '\n';
3 }
```

因此，输出是：

```
size types differ
```

C++17 后，对于返回值可以使用后缀_v 并省略::value(详见 5.6 节)：

```
1 if (!std::is_same_v<decltype(size1), decltype(size2)>) {
2     std::cout << "size types differ" << '\n';
3 }
```

对非类型模板参数类型的其他约束仍然有效。特别是，第 3.3 节中讨论的非类型模板参数类型的限制仍然适用。例如：

```
1 Stack<int, 3.14> sd; // ERROR: Floating-point nontype argument
```

并且，可以将字符串作为常量数组传递 (C++17 中甚至是静态的局部声明；见第 3.3 节)：

basics/message.cpp

```
1 #include <iostream>
2
3 template<auto T> // take value of any possible nontype parameter (since C++17)
4 class Message {
5 public:
6     void print() {
7         std::cout << T << '\n';
8     }
9 };
10
11 int main()
12 {
13     Message<42> msg1;
14     msg1.print(); // initialize with int 42 and print that value
15
16     static char const s[] = "hello";
17     Message<s> msg2; // initialize with char const[6] "hello"
18     msg2.print(); // and print that value
```

还要注意的是，`template<decltype(auto) N>` 也可以，其允许将 N 实例化为引用：

```
1 template<decltype(auto) N>
2 class C {
3     ...
4 };
5
6 int i;
7 C<(i)> x; // N is int&
```

详情参见 15.10.1。

3.5. 总结

- 模板的模板参数可以是值，而非类型。
- 不能将浮点数或类类型对象作为非类型模板的参数。对于指向字符串字面量、临时对象和子对象的指针/引用，有一些限制。
- 使用 `auto` 可使模板具有泛型值的非类型模板参数。

第4章 可变参数模板

C++11 后，模板可以使用可变数量的模板参数。该特性允许在传递任意数量的类型参数的地方使用模板。典型的应用是通过类或框架传递任意数量的类型参数，另一个应用是提供泛型来处理任意数量的类型参数。

4.1. 介绍

可以接受任意数量的模板参数，具有这种能力的模板称为可变参数模板。

4.1.1 示例

可以使用 print() 来获取可变类型的参数：

basics/varprint1.hpp

```
1 #include <iostream>
2
3 void print ()
4 {
5 }
6
7 template<typename T, typename... Types>
8 void print (T firstArg, Types... args)
9 {
10    std::cout << firstArg << '\n'; // print first argument
11    print(args...); // call print() for remaining arguments
12 }
```

若传递了一个或多个参数，则使用函数模板，通过指定第一个参数，可以在递归调用其余参数的 print() 前，打印第一个参数。剩下的 args 是一个函数参数包：

```
1 void print (T firstArg, Types... args)
```

使用由模板参数包指定的不同“类型”：

```
1 template<typename T, typename... Types>
```

要结束递归，需要提供 print() 的非模板重载，该重载在参数包为空时使用。

例如，

```
1 std::string s("world");
2 print (7.5, "hello", s);
```

将会输出以下内容：

```
7.5
hello
world
```

调用首先展开为

```
1 print<double, char const*, std::string> (7.5, "hello", s);
```

- firstArg 的值为 7.5，因此类型 T 是一个 double 类型
- args 是可变参数模板参数，其值为 char const* 型的”hello” 和 std::string 型的”world”。

将 7.5 作为 firstArg 打印后，再次对其余参数调用 print()，然后展开为：

```
1 print<char const*, std::string> ("hello", s);
```

- firstArg 的值为”hello”，所以 T 的类型是 char const*
- args 是可变参数，其值类型为 std::string。

将”hello” 作为 firstArg 打印后，再次对其余参数调用 print()，然后展开为：

```
1 print<std::string> (s);
```

- firstArg 的值为 “world”，所以类型 T 现在是 std::string
- args 是空的可变参数模板参数。

因此，在将”world” 打印作为 firstArg 之后，调用 print() 时不带参数，这将调用 print() 的非模板重载。

4.1.2 重载可变和非可变模板

也可以这样实现上面的例子：

basics/varprint2.hpp

```
1 #include <iostream>
2
3 template<typename T>
4 void print (T arg)
5 {
6     std::cout << arg << '\n'; // print passed argument
7 }
8
9 template<typename T, typename... Types>
10 void print (T firstArg, Types... args)
11 {
12     print(firstArg); // call print() for the first argument
13     print(args...); // call print() for remaining arguments
14 }
```

若两个函数模板的区别仅在于末尾参数包的不同，则首选没有末尾参数包的函数模板。

起初在 C++11 和 C++14 中，这是一个不明确的问题，后来得到了解决（参见 [CoreIssue1395]），并且所有版本的编译器都会这样处理。

C.3.1 节解释了这里应用的重载解析规则。

4.1.3 操作符 sizeof...

C++11 还为可变参模板引入了一种新的操作符:sizeof...，其值为参数包包含的元素数量。因此，

```
1 template<typename T, typename... Types>
2 void print (T firstArg, Types... args)
3 {
4     std::cout << sizeof...(Types) << '\n'; // print number of remaining types
5     std::cout << sizeof...(args) << '\n'; // print number of remaining args
6     ...
7 }
```

传递给 print() 的第一个参数之后，输出剩余两次参数的数量。对于模板参数包和函数参数包可以使用 sizeof...。

我们可能会认为，递归结束时可以跳过该函数，并在没有参数的情况下不调用它：

```
1 template<typename T, typename... Types>
2 void print (T firstArg, Types... args)
3 {
4     std::cout << firstArg << '\n';
5     if (sizeof...(args) > 0) { // error if sizeof...(args)==0
6         print(args...); // and no print() for no arguments declared
7     }
8 }
```

但因为函数模板中所有 if 的两个分支都要实例化。这里，是否运行是运行时确定，而调用的实例化是编译时确定。因此，如果为一个（最后一个）参数使用 print() 函数模板，则调用 print(args...) 的语句在没有参数的情况下仍然会实例化，若没有参数的情况下不提供 print() 函数，则会报错。

但请注意，由于 C++17 提供了编译时 if，就可以实现了这里的期望，但在语法上略有不同，这将在第 8.5 节中讨论。

4.2. 折叠表达式

C++17 后，有一个特性可以对参数包的所有参数使用二元运算符计算结果（初始值可选）。

例如，下面的函数会返回所有入参的和：

```
1 template<typename... T>
2 auto foldSum (T... s) {
3     return (... + s); // ((s1 + s2) + s3) ...
4 }
```

若参数包为空，表达式通常是错误格式的(除了操作符`&&`的值为`true`，操作符`||`的值为`false`，逗号操作符空参数包的值为`void()`)。

表 4.1 列出了可能的折叠表达式。

折叠表达式	展开
(... op pack)	((pack1 op pack2) op pack3) ... op packN)
(pack op ...)	(pack1 op (... (packN-1 op packN)))
(init op ... op pack)	((init op pack1) op pack2) ... op packN)
(pack op ... op init)	(pack1 op (... (packN op init)))

表 4.1. 折叠表达式 (C++17)

几乎所有的二元运算符都可以使用折叠表达式(详见 12.4.6 节)。例如，可以使用折叠表达式来遍历一个二叉树的路径，使用操作符`->*`:

basics/foldtraverse.cpp

```
1 // define binary tree structure and traverse helpers:
2 struct Node {
3     int value;
4     Node* left;
5     Node* right;
6     Node(int i=0) : value(i), left(nullptr), right(nullptr) {
7     }
8     ...
9 };
10
11 auto left = &Node::left;
12 auto right = &Node::right;
13
14 // traverse tree, using fold expression:
15 template<typename T, typename... TP>
16 Node* traverse (T np, TP... paths) {
17     return (np ->* ... ->* paths); // np ->* paths1 ->* paths2 ...
18 }
19
20 int main()
21 {
22     // init binary tree structure:
23     Node* root = new Node{0};
24     root->left = new Node{1};
25     root->left->right = new Node{2};
26     ...
27     // traverse binary tree:
28     Node* node = traverse(root, left, right);
29     ...
30 }
```

这里，

```
1 (np ->* ... ->* paths)
```

使用折叠表达式遍历从 np 开始的可变元素路径。

这种使用初始化器的折叠表达式，可以简化可变参数模板来打印上面的所有参数：

```
1 template<typename... Types>
2 void print (Types const&... args)
3 {
4     (std::cout << ... << args) << '\n';
5 }
```

但无法为参数包中的每个元素输出添加打印空格。要添加空格，需要一个额外的类模板，确保参数的输出都会添加一个空格：

basics/addspace.hpp

```
1 template<typename T>
2 class AddSpace
3 {
4     private:
5         T const& ref; // refer to argument passed in constructor
6     public:
7         AddSpace(T const& r): ref(r) {
8     }
9         friend std::ostream& operator<< (std::ostream& os, AddSpace<T> s) {
10             return os << s.ref << ' '; // output passed argument and a space
11         }
12     };
13
14 template<typename... Args>
15 void print (Args... args) {
16     ( std::cout << ... << AddSpace(args) ) << '\n';
17 }
```

表达式 AddSpace(args) 使用类模板参数推断 (参见 2.9 节) 产生 AddSpace<Args>(args)，这为每个参数创建一个 AddSpace 对象，该对象引用传递的参数，并在输出表达式时使用该参数添加一个空格。

关于折叠表达式的详细信息请参见 12.4.6 节。

4.3. 应用

可变参数模板在实现通用库 (如 C++ 标准库) 时扮演着重要的角色。

典型的应用是转发可变数量的类型参数。例如：

- 向指针中堆对象的构造函数传递参数：

```
1 // create shared pointer to complex<float> initialized by 4.2 and 7.7:
2 auto sp = std::make_shared<std::complex<float>>(4.2, 7.7);
```

- 将参数传递给由线程库启动的线程:

```
1 std::thread t (foo, 42, "hello"); // call foo(42,"hello") in a separate thread
```

- 将参数传递给进入 vector 中元素的构造函数:

```
1 std::vector<Customer> v;
2 ...
3 v.emplace_back("Tim", "Jovi", 1962); // insert a Customer initialized by three arguments
```

通常，参数使用移动语义“完美转发”(见第 6.1 节)，相应的声明为:

```
1 namespace std {
2     template<typename T, typename... Args> shared_ptr<T>
3         make_shared(Args&&... args);
4     class thread {
5         public:
6             template<typename F, typename... Args>
7                 explicit thread(F&& f, Args&&... args);
8             ...
9     };
10
11     template<typename T, typename Allocator = allocator<T>>
12     class vector {
13         public:
14             template<typename... Args> reference emplace_back(Args&&... args);
15             ...
16     };
17 }
```

可变参数函数模板参数与普通参数适用相同的规则。通过值传递，参数复制并衰变(例如，数组变成指针)，若通过引用传递，参数指向原始参数，而不衰变:

```
1 // args are copies with decayed types:
2 template<typename... Args> void foo (Args... args);
3 // args are nondecayed references to passed objects:
4 template<typename... Args> void bar (Args const&... args);
```

4.4. 类模板和表达式

参数包还可以出现在其他地方，例如表达式、类模板、using 声明，甚至推导策略。第 12.4.2 节有一个完整的列表。

4.4.1 表达式

不仅仅是转发所有参数，可以使用它们进行计算。也就是，可以使用参数包中的参数进行计算。

例如，下面的函数将参数包 args 的每个参数都加倍，并将每个加倍的参数传递给 print():

```
1 template<typename... T>
2 void printDoubled (T const&... args)
```

```
3 {  
4     print (args + args...);  
5 }
```

例如,

```
1 printDoubled(7.5, std::string("hello"), std::complex<float>(4,2));
```

该函数具有以下效果 (除了构造函数的副作用):

```
1 print(7.5 + 7.5,  
2       std::string("hello") + std::string("hello"),  
3       std::complex<float>(4,2) + std::complex<float>(4,2));
```

如果想给每个参数加 1, 请注意省略号中的点不能直接跟在数字后面:

```
1 template<typename... T>  
2 void addOne (T const&... args)  
3 {  
4     print (args + 1...); // ERROR: 1... is a literal with too many decimal points  
5     print (args + 1 ...); // OK  
6     print ((args + 1)...); // OK  
7 }
```

编译时表达式可以以同样的方式包含模板参数包。例如, 下面的函数模板返回所有参数的类型是否相同:

```
1 template<typename T1, typename... TN>  
2 constexpr bool isHomogeneous (T1, TN...)  
3 {  
4     return (std::is_same<T1,TN>::value && ...); // since C++17  
5 }
```

这是折叠表达式的一种应用 (参见第 4.2 节)

```
1 isHomogeneous(43, -1, "hello")
```

返回值的表达式展开为

```
1 std::is_same<int,int>::value && std::is_same<int,char const*>::value
```

结果是 false

```
1 isHomogeneous("hello", " ", "world", "!")
```

会产生 true, 因为传递的参数都推导为 char const* (因为调用参数按值传递, 所以参数类型不变)。

4.4.2 索引

下面的函数使用可变索引列表来访问传递给第一个参数的对应元素:

```
1 template<typename C, typename... Idx>
2 void printElems (C const& coll, Idx... idx)
3 {
4     print (coll[idx]...);
5 }
```

当使用

```
1 std::vector<std::string> coll = {"good", "times", "say", "bye"};
2 printElems(coll, 2, 0, 3);
```

则调用

```
print (coll[2], coll[0], coll[3]);
```

还可以将非类型模板参数声明为参数包。例如:

```
1 template<std::size_t... Idx, typename C>
2 void printIdx (C const& coll)
3 {
4     print(coll[Idx]...);
5 }
```

也可以使用

```
1 std::vector<std::string> coll = {"good", "times", "say", "bye"};
2 printIdx<2, 0, 3>(coll);
```

这与前面的例子有相同的效果。

4.4.3 类模板

可变参数模板也可以是类模板，其中任意数量的模板参数指定了相应成员的类型：

```
1 template<typename... Elements>
2 class Tuple;
3
4 Tuple<int, std::string, char> t; // t can hold integer, string, and character
```

这将在第 25 章中讨论。

另一个例子是能够指定对象可能的类型：

```
1 template<typename... Types>
2 class Variant;
3
4 Variant<int, std::string, char> v; // v can hold integer, string, or character
```

这将在第 26 章中讨论。

也可以定义类，作为类型来表示一组索引：

```
1 // type for arbitrary number of indices:  
2 template<std::size_t...>  
3 struct Indices {  
4 };
```

可以用来定义函数，该函数在编译时对给定索引使用 `get<>()` 访问 `std::array` 或 `std::tuple` 的元素调用 `print()`:

```
1 template<typename T, std::size_t... Idx>  
2 void printByIdx(T t, Indices<Idx...>)  
3 {  
4     print(std::get<Idx>(t)...);  
5 }
```

该模板的使用方法如下:

```
1 std::array<std::string, 5> arr = {"Hello", "my", "new", "!", "World"};  
2 printByIdx(arr, Indices<0, 4, 3>());
```

或者这样:

```
1 auto t = std::make_tuple(12, "monkeys", 2.0);  
2 printByIdx(t, Indices<0, 1, 2>());
```

这只元编程的开始，在第 8.1 节和第 23 章中会更加详细的讨论元编程。

4.4.4 推导策略

推导策略 (参见第 2.9 节) 也可以应用于可变参数。例如，C++ 标准库定义了以下 `std::arrays` 的推导策略:

```
1 namespace std {  
2     template<typename T, typename... U> array(T, U...)  
3         -> array<enable_if_t<(is_same_v<T, U> && ...), T>,  
4             (1 + sizeof...(U))>;  
5 }
```

初始化,

```
1 std::array a{42, 45, 77};
```

策略中推导出元素类型的 `T`，以及各种 `U...` 类型转换为后续元素的类型。因此，元素总数为 `1 + sizeof...(U)`:

```
1 std::array<int, 3> a{42, 45, 77};
```

第一个数组参数的 `std::enable_if<>` 表达式是一个折叠表达式 (4.4.1 节中引入的 `isHomogeneous()`) 展开为:

```
1 is_same_v<T, U1> && is_same_v<T, U2> && is_same_v<T, U3> ...
```

若结果不为 `true`(即并非所有元素类型都相同)，则丢弃推导策略，从而整体推导失败。标准库需要所有元素具有相同的类型时，推导策略才能成功执行。

4.4.5 基类和 using

看看下面的例子：

basics/varusing.cpp

```
1 #include <string>
2 #include <unordered_set>
3
4 class Customer
5 {
6     private:
7         std::string name;
8     public:
9         Customer(std::string const& n) : name(n) { }
10        std::string getName() const { return name; }
11    };
12
13 struct CustomerEq {
14     bool operator()(Customer const& c1, Customer const& c2) const {
15         return c1.getName() == c2.getName();
16     }
17 };
18
19 struct CustomerHash {
20     std::size_t operator()(Customer const& c) const {
21         return std::hash<std::string>()(c.getName());
22     }
23 };
24
25 // define class that combines operator() for variadic base classes:
26 template<typename... Bases>
27 struct Overloader : Bases...
28 {
29     using Bases::operator()...; // OK since C++17
30 };
31
32 int main()
33 {
34     // combine hasher and equality for customers in one type:
35     using CustomerOP = Overloader<CustomerHash,CustomerEq>;
36
37     std::unordered_set<Customer,CustomerHash,CustomerEq> coll1;
38     std::unordered_set<Customer,CustomerOP,CustomerOP> coll2;
39     ...
40 }
```

首先定义了一个类 Customer 和一些独立的函数对象，用于计算哈希值和比较 Customer 对象。

```
1 template<typename... Bases>
2 struct Overloader : Bases...
```

```
3 {  
4     using Bases::operator()...; // OK since C++17  
5 };
```

可以定义一个派生自可变数量基类的类，该类从每个基类引入函数操作符的声明。

```
1 using CustomerOP = Overloader<CustomerHash, CustomerEq>;
```

使用这个特性从 CustomerHash 和 CustomerEq 派生 CustomerOP，并在派生类中启用函数操作符的实现。

参见第 26.4 节了解该技术的另一种应用场景。

4.5. 总结

- 通过使用参数包，可以为任意数量、类型的模板参数定义模板。
- 要处理参数，需要递归和/或匹配的非变参函数。
- 操作符 sizeof... 可为参数包提供的参数数量。
- 可变参数模板的一个应用是转发任意数量的类型参数。
- 通过使用折叠表达式，可以对参数包中的所有参数使用相应的操作符。

第 5 章 基础技巧

本章进一步介绍了与模板使用相关的一些基本知识:typename 关键字, 将成员函数和嵌套类定义为模板, 双重模板参数, 零值初始化, 以及在函数模板中使用字符串字面值作为实参的一些细节。这些都很基础, 每个开发者都应该了解一下。

5.1. 关键字 typename

关键字 typename 是在 C++ 标准化过程中引入的, 目的是说明模板内的标识符是类型。比如下面的例子:

```
1 template<typename T>
2 class MyClass {
3     public:
4     ...
5     void foo() {
6         typename T::SubType* ptr;
7     }
8 };
```

这里, 第二个 typename 用于说明 SubType 是在类 T 中定义的类型。因此, ptr 是指向类型 T::SubType 的指针。

若没有 typename, SubType 将假定为非类型成员(例如, 静态数据成员或枚举数常量)。因此, 表达式

```
1 T::SubType* ptr
```

表示的是类 T 的静态 SubType 成员与 ptr 的乘积, 这不是一个错误, 因为对于 MyClass<> 的某些实例化, 这可能有效。

当模板参数是类型时, 必须使用 typename。这将在第 13.3.2 节中详细讨论。

typename 的一种应用是在泛型代码中声明标准容器的迭代器:

basics/printcoll.hpp

```
1 #include <iostream>
2
3 // print elements of an STL container
4 template<typename T>
5 void printcoll (T const& coll)
6 {
7     typename T::const_iterator pos; // iterator to iterate over coll
8     typename T::const_iterator end(coll.end()); // end position
9     for (pos=coll.begin(); pos!=end; ++pos) {
10         std::cout << *pos << ' ';
11     }
12     std::cout << '\n';
13 }
```

这个函数模板中，调用参数是一个 T 类型的标准容器。要遍历容器的所有元素，需要使用容器的迭代器类型，在每个标准容器类中声明为 `const_iterator` 的类型：

```
1 class stlcontainer {
2     public:
3         using iterator = ...; // iterator for read/write access
4         using const_iterator = ...; // iterator for read access
5         ...
6     };
```

因此，要访问模板类型 T 的 `const_iterator` 类型，必须用 `typename` 进行限定：

```
1 typename T::const_iterator pos;
```

有关在 C++17 前需要 `typename` 的更多细节，请参阅第 13.3.2 节。注意，C++20 在许多常见情况下可能会删除对 `typename` 的需要（请参阅第 17.1 节了解详细信息）。

5.2. 零值初始化

对于基本类型，如 `int`、`double` 或指针类型，没有默认构造函数可以初始化。相反，任何未初始化的局部变量都有一个未定义的值：

```
1 void foo()
2 {
3     int x; // x has undefined value
4     int* ptr; // ptr points to anywhere (instead of nowhere)
5 }
```

若编写模板，并想让模板类型的变量用默认值初始化，就会遇到一个问题，简单的定义对内置类型并没有进行初始化：

```
1 template<typename T>
2 void foo()
3 {
4     T x; // x has undefined value if T is built-in type
5 }
```

因此，可以显式调用内置类型的默认构造函数，该构造函数用 0 初始化内置类型（`bool` 为 `false`，指针为 `nullptr`）。因此，即使是内置类型，也可以通过编写以下代码来确保正确的初始化：

```
1 template<typename T>
2 void foo()
3 {
4     T x{}; // x is zero (or false or nullptr) if T is a built-in type
5 }
```

这种初始化方式称为值初始化，要么调用提供的构造函数，要么对对象进行零初始化。即使构造函数是显式的，也可以这样做。

C++11 前，正确初始化的语法是

```
1 T x = T(); // x is zero (or false or nullptr) if T is a built-in type
```

C++17 前，这种机制（现在仍受支持）只有在为复制初始化选择的构造函数不是显式的情况下才有效。在 C++17 中，省略了强制复制，从而避免了这种限制，而且两种语法都有效。但若没有默认构造函数，带大括号的初始化表示法可以使用初始化列表构造函数。

对于某种类型 X，有参数类型为 std::initializer_list<X> 的构造函数。

为了确保将类型参数化的类模板成员初始化，可以定义默认构造函数，使用带大括号的初始化式来初始化成员：

```
1 template<typename T>
2 class MyClass {
3 private:
4     T x;
5 public:
6     MyClass() : x{} // ensures that x is initialized even for built-in types
7 }
8 ...
9 };
```

C++11 前的语法

```
1 MyClass() : x{} // ensures that x is initialized even for built-in types
2 }
```

在后续标准中使用，仍然有效。

C++11 后，还可以为非静态成员提供默认初始化，这样也可以实现以下操作：

```
1 template<typename T>
2 class MyClass {
3 private:
4     T x{}; // zero-initialize x unless otherwise specified
5 ...
6 };
```

但是，请注意默认参数不能使用该语法。例如，

```
1 template<typename T>
2 void foo(T p{}) { // ERROR
3     ...
4 }
```

必须这样写：

```
1 template<typename T>
2 void foo(T p = T{}) { // OK (must use T() before C++11)
3     ...
4 }
```

5.3. 使用 this->

对于具有依赖于模板参数的基类类模板，即使成员 x 被继承，使用名称 x 本身并不总是等同于 this->x。例如：

```
1 template<typename T>
2 class Base {
3 public:
4     void bar();
5 };
6
7 template<typename T>
8 class Derived : Base<T> {
9 public:
10    void foo() {
11        bar(); // calls external bar() or error
12    }
13 };
```

本例中，解析 foo() 内部的符号 bar，不会考虑 Base 中定义的 bar()。因此，要么出现错误，要么调用另一个 bar() 实现（例如全局 bar()）。

我们将在第 13.4.2 节详细讨论这个问题。目前，建议始终对基类中声明的符号进行限定，这些符号在某种程度上依赖于模板参数 this-> 或 Base<T>::。

5.4. 原始数组和字符串字面量的模板

将原始数组或字符串字面值传递给模板时，务必小心。若模板参数声明为引用，则参数不会衰变，传入的“hello”参数为 char const[6] 类型。若因类型不同而传递不同长度的原始数组或字符串参数，这可能会成为一个问题。只有当按值传递参数时，类型才会衰变，因此字符串字面值转换为 char const*。这将在第 7 章中详细讨论。

注意，还可以提供专门处理原始数组或字符串字面量的模板。例如：

basics/lessarray.hpp

```
1 template<typename T, int N, int M>
2 bool less (T(&a) [N], T(&b) [M])
3 {
4     for (int i = 0; i < N && i < M; ++i) {
5         if (a[i] < b[i]) return true;
6         if (b[i] < a[i]) return false;
7     }
8     return N < M;
9 }
```

这里使用

```
1 int x[] = {1, 2, 3};
2 int y[] = {1, 2, 3, 4, 5};
```

```
3 std::cout << less(x,y) << '\n';
```

less<>() 实例化，T 为 int, N 为 3, M 为 5。

也可以将此模板用于字符串字面值：

```
1 std::cout << less("ab","abc") << '\n';
```

这种情况下，less<>() 实例化，T 为 char const, N 为 3, M 为 4。

若想为字符串(和其他字符数组)提供一个函数模板，可以这样：

basics/lessstring.hpp

```
1 template<int N, int M>
2 bool less (char const(&a) [N], char const(&b) [M])
3 {
4     for (int i = 0; i < N && i < M; ++i) {
5         if (a[i] < b[i]) return true;
6         if (b[i] < a[i]) return false;
7     }
8     return N < M;
9 }
```

注意，对于边界未知的数组，可以(有时必须)重载或偏特化。下面的代码演示了数组所有可能的重载：

basics/arrays.hpp

```
1 #include <iostream>
2
3 template<typename T>
4 struct MyClass; // primary template
5
6 template<typename T, std::size_t SZ>
7 struct MyClass<T[SZ]> // partial specialization for arrays of known bounds
8 {
9     static void print() { std::cout << "print() for T[" << SZ << "]\n"; }
10 };
11
12 template<typename T, std::size_t SZ>
13 struct MyClass<T(&) [SZ]> // partial spec. for references to arrays of known bounds
14 {
15     static void print() { std::cout << "print() for T(&) [" << SZ << "]\n"; }
16 };
17
18 template<typename T>
19 struct MyClass<T[]> // partial specialization for arrays of unknown bounds
20 {
21     static void print() { std::cout << "print() for T[]\n"; }
22 };
```

```

24 template<typename T>
25 struct MyClass<T(&) []> // partial spec. for references to arrays of unknown bounds
26 {
27     static void print() { std::cout << "print() for T(&) []\n"; }
28 };
29
30 template<typename T>
31 struct MyClass<T*> // partial specialization for pointers
32 {
33     static void print() { std::cout << "print() for T*\n"; }
34 };

```

类模板 `MyClass<>` 用于各种类型: 已知和未知边界的数组、对已知和未知边界的数组的引用, 以及指针。每种情况都不同, 可以在使用数组时产生:

basics/arrays.cpp

```

1 #include "arrays.hpp"
2
3 template<typename T1, typename T2, typename T3>
4 void foo(int a1[7], int a2[], // pointers by language rules
5     int (&a3)[42], // reference to array of known bound
6     int (&x0)[], // reference to array of unknown bound
7     T1 x1, // passing by value decays
8     T2& x2, T3&& x3) // passing by reference
9 {
10    MyClass<decltype(a1)>::print(); // uses MyClass<T*>
11    MyClass<decltype(a2)>::print(); // uses MyClass<T*>
12    MyClass<decltype(a3)>::print(); // uses MyClass<T(&) [SZ]>
13    MyClass<decltype(x0)>::print(); // uses MyClass<T(&) []>
14    MyClass<decltype(x1)>::print(); // uses MyClass<T*>
15    MyClass<decltype(x2)>::print(); // uses MyClass<T(&) []>
16    MyClass<decltype(x3)>::print(); // uses MyClass<T(&) []>
17 }
18
19 int main()
20 {
21     int a[42];
22     MyClass<decltype(a)>::print(); // uses MyClass<T[SZ]>
23
24     extern int x[]; // forward declare array
25     MyClass<decltype(x)>::print(); // uses MyClass<T[]>
26     foo(a, a, a, x, x, x, x);
27 }
28
29 int x[] = {0, 8, 15}; // define forward-declared array

```

注意, 由语言规则声明为数组(带或不带长度)的调用参数实际上具有指针类型。未知边界的数

组模板可以用于不完整类型，例如

```
1 extern int i[];
```

当通过引用传递时，就变成了一个 `int(&)[]`，也可以用作模板参数。

类型 `X(&)[]`——对于某些任意类型 `X`——参数仅在 C++17 中有效，通过核心问题 393 的解决。在语言的早期版本中，许多编译器也可以接受这样的参数。

泛型代码中使用不同数组类型的另一种例子可参见 19.3.1 节。

5.5. 成员模板

类成员可以是模板，对于嵌套类和成员函数都有可能。`Stack<>` 类模板可以再次展示这种能力的应用和优势。通常，只有具有相同的类型时，才能相互分配堆栈，这意味着元素具有相同的类型。但不能将其他类型的元素赋值给堆栈，即使对定义的元素类型有隐式的类型转换：

```
1 Stack<int> intStack1, intStack2; // stacks for ints
2 Stack<float> floatStack; // stack for floats
3 ...
4 intStack1 = intStack2; // OK: stacks have same type
5 floatStack = intStack1; // ERROR: stacks have different types
```

默认赋值操作符要求赋值操作符的两端具有相同的类型。

将赋值操作符定义为模板，可以使用定义了适当类型转换的元素对堆栈赋值。需要进行如下声明：

basics/stack5decl.hpp

```
1 template<typename T>
2 class Stack {
3 private:
4     std::deque<T> elems; // elements
5
6 public:
7     void push(T const&); // push element
8     void pop(); // pop element
9     T const& top() const; // return top element
10    bool empty() const { // return whether the stack is empty
11        return elems.empty();
12    }
13
14    // assign stack of elements of type T2
15    template<typename T2>
16    Stack& operator= (Stack<T2> const&);
17};
```

进行了以下两处修改：

1. 为另一种类型 T2 的元素堆栈添加了赋值操作符声明。
2. 栈现在使用 std::deque<> 作为元素的内部容器。同样，这也是新赋值操作符实现的结果。

新赋值操作符的实现如下所示：

basics/stack5assign.hpp

```

1 template<typename T>
2 template<typename T2>
3 Stack<T>& Stack<T>::operator= (Stack<T2> const& op2)
4 {
5     Stack<T2> tmp(op2); // create a copy of the assigned stack
6
7     elems.clear(); // remove existing elements
8     while (!tmp.empty()) { // copy all elements
9         elems.push_front(tmp.top());
10        tmp.pop();
11    }
12    return *this;
13 }
```

这只是一个模板特性的基本实现。像异常处理的实现，这里没有添加。

首先看看定义成员模板的语法。模板参数为 T 的模板内部，定义了一个模板参数为 T2 的内部模板：

```

1 template<typename T>
2 template<typename T2>
3 ...
```

成员函数中，可能只希望访问所赋堆栈 op2 的必要数据。但这个堆栈有不同的类型(若为两种不同的参数类型实例化一个类模板，将得到两种不同的类类型)，因此只能使用公共接口。所以，访问元素的唯一方法是 top()。然而，每个元素都必须成为顶部元素。因此，必须创建 op2 的副本，以便通过调用 pop() 从该副本中获取元素。因为 top() 返回压入堆栈的最后一个元素，所以可能更倾向于使用支持在集合的另一端插入元素的容器。所以，这里使用 std::deque<> 的 push_front()，将元素放在集合的另一端。

要访问 op2 的所有成员，可以将所有其他堆栈实例声明为友元：

basics/stack6decl.hpp

```

1 template<typename T>
2 class Stack {
3 private:
4     std::deque<T> elems; // elements
5
6 public:
7     void push(T const&); // push element
```

```

8 void pop(); // pop element
9 T const& top() const; // return top element
10 bool empty() const { // return whether the stack is empty
11     return elems.empty();
12 }
13
14 // assign stack of elements of type T2
15 template<typename T2>
16 Stack& operator= (Stack<T2> const&);
17 // to get access to private members of Stack<T2> for any type T2:
18 template<typename> friend class Stack;
19 };

```

因为模板参数的名字没有使用，所以可以省略：

```
1 template<typename> friend class Stack;
```

现在，模板赋值操作符的实现可能是这样：

basics/stack6assign.hpp

```

1 template<typename T>
2 template<typename T2>
3 Stack<T>& Stack<T>::operator= (Stack<T2> const& op2)
4 {
5     elems.clear(); // remove existing elements
6     elems.insert(elems.begin(), // insert at the beginning
7                  op2.elems.begin(), // all elements from op2
8                  op2.elems.end());
9     return *this;
10 }

```

无论实现什么样，有了这个成员模板，现在可以将一堆 int 型对象赋值给一堆 float 型对象：

```

1 Stack<int> intStack; // stack for ints
2 Stack<float> floatStack; // stack for floats
3 ...
4 floatStack = intStack; // OK: stacks have different types,
5 // but int converts to float

```

当然，这个赋值不会改变堆栈及其元素的类型。赋值之后，floatStack 的元素仍然是浮点数，因此 top() 仍然返回一个浮点数。

这个函数可能会禁用类型检查，这样就可以将任何类型的元素赋给堆栈，但事实并非如此。当源堆栈（副本）的元素移动到目标堆栈时，就会进行类型检查：

```
1 elems.push_front(tmp.top());
```

若一堆 string 赋值给一堆 float 数，这一行的编译会产生一个错误信息，表示 tmp.top() 返回的字符串不能作为参数传递给 elems.push_front()（错误信息会随编译器而变化，但这是其含义的重点）：

```
1 Stack<std::string> stringStack; // stack of strings
```

```

2 Stack<float> floatStack; // stack of floats
3 ...
4 floatStack = stringStack; // ERROR: std::string doesn't convert to float

```

同样，可以改变实现来参数化内部容器类型：

basics/stack7decl.hpp

```

1 template<typename T, typename Cont = std::deque<T>>
2 class Stack {
3 private:
4     Cont elems; // elements
5
6 public:
7     void push(T const&); // push element
8     void pop(); // pop element
9     T const& top() const; // return top element
10    bool empty() const { // return whether the stack is empty
11        return elems.empty();
12    }
13
14    // assign stack of elements of type T2
15    template<typename T2, typename Cont2>
16    Stack& operator=(Stack<T2,Cont2> const&);
17    // to get access to private members of Stack<T2> for any type T2:
18    template<typename, typename> friend class Stack;
19 };

```

所以模板赋值操作符就可以这样实现：

basics/stack7assign.hpp

```

1 template<typename T, typename Cont>
2 template<typename T2, typename Cont2>
3 Stack<T,Cont>&
4 Stack<T,Cont>::operator=(Stack<T2,Cont2> const& op2)
5 {
6     elems.clear(); // remove existing elements
7     elems.insert(elems.begin(), // insert at the beginning
8                  op2.elems.begin(), // all elements from op2
9                  op2.elems.end());
10    return *this;
11 }

```

对于类模板，只有那些调用的成员函数才会实例化。因此，若需要避免在堆栈中分配不同类型的元素，可以使用 vector 作为内部容器：

```

1 // stack for ints using a vector as an internal container
2 Stack<int, std::vector<int>> vStack;
3 ...

```

```
4 vStack.push(42);
5 vStack.push(7);
6 std::cout << vStack.top() << '\n';
```

因为赋值操作符模板不是必需，所以不会出现缺少成员函数 `push_front()` 的错误消息。

要获得上一个示例的完整实现，请参阅子目录基础中以 `stack7` 开头的文件。

成员函数模板的特化

成员函数模板也可以全特化，但不能偏特化。例如，对于以下类：

basics/boolstring.hpp

```
1 class BoolString {
2 private:
3     std::string value;
4 public:
5     BoolString (std::string const& s)
6     : value(s) {
7     }
8     template<typename T = std::string>
9     T get() const { // get value (converted to T)
10         return value;
11     }
12 };
```

可以为如下所示的成员函数模板提供全特化：

basics/boolstringgetbool.hpp

```
1 // full specialization for BoolString::getValue<>() for bool
2 template<>
3 inline bool BoolString::get<bool>() const {
4     return value == "true" || value == "1" || value == "on";
5 }
```

不需要也不能对特化进行声明，只需要定义。因为它是全特化的，且位于头文件中，所以若定义包含在不同的转译单元中，必须使用内联声明，以避免错误。

可以使用类的全特化，如下所示：

```
1 std::cout << std::boolalpha;
2 BoolString s1("hello");
3 std::cout << s1.get() << '\n'; // prints hello
4 std::cout << s1.get<bool>() << '\n'; // prints false
5 BoolString s2("on");
6 std::cout << s2.get<bool>() << '\n'; // prints true
```

特殊成员函数的模板

只要特殊成员函数允许复制或移动对象，就可以使用模板成员函数。与上面定义的赋值操作符类似，也可以是构造函数。但模板构造函数或模板赋值操作符，不能取代预定义构造函数或赋值操作符，成员模板不作为复制或移动对象的特殊成员函数。本例中，对于相同类型的堆栈赋值，仍然使用默认赋值操作符。

这种影响有好有坏：

- 模板构造函数或赋值操作符，可能比预定义的复制/移动构造函数或赋值操作符更匹配，尽管只提供了用于初始化其他类型的模板版本。请参见 6.2 节。
- 要对复制/移动构造函数进行“模板化”以约束其存在是非常困难的。详见章节 6.4。

5.5.1 .template 构造

有时，在调用成员模板时，需要显式限定模板参数。这种情况下，必须使用 `template` 关键字来确保 < 是模板参数列表的开头。考虑下面使用标准 `bitset` 类型的例子：

```
1 template<unsigned long N>
2 void printBitset (std::bitset<N> const& bs) {
3     std::cout << bs.template to_string<char>, std::char_traits<char>,
4         std::allocator<char>>>();
5 }
```

对于 `bitset` `bs`，使用 `to_string()` 的成员函数模板，同时显式指定字符串类型的信息。如果没有使用 `.template`，编译器就不知道后面的小于标记 (<) 是模板参数列表的开头。注意，只有在句点之前的构造依赖于模板参数时才会出现问题。在例子中，参数 `bs` 依赖于模板参数 `N`。

`.template` 表示法（以及类似的表示法，如 `->template` 和 `::template`）应该只在模板内部使用，只有当它们遵循依赖于模板的某些参数时才会使用。详细信息请参见 13.3.3。

5.5.2 泛型 Lambda 和成员模板

注意，C++14 中引入的泛型 Lambda 是成员模板的快捷方式。计算任意类型的两个参数的“加和”的简单 Lambda 表达式：

```
1 [] (auto x, auto y) {
2     return x + y;
3 }
```

是以下类默认构造对象的快捷方式：

```
1 class SomeCompilerSpecificName {
2     public:
3     SomeCompilerSpecificName(); // constructor only callable by compiler
4     template<typename T1, typename T2>
5     auto operator() (T1 x, T2 y) const {
6         return x + y;
7     }
8 };
```

详细信息请参见 15.10.6。

5.6. 变量模板

C++14 后，变量也可以通过特定类型参数化，成为做变量模板。

我们有非常相似的术语来描述非常不同的事情：变量模板是一个变量，它是一个模板（变量在这里是一个名词）。可变参数模板是用于可变数量模板参数的模板（可变参数在这里是形容词）。

例如，可以使用以下代码定义 π 的值，而不定义值的类型：

```
1 template<typename T>
2 constexpr T pi{3.1415926535897932385};
```

与所有模板一样，此声明不能出现在函数或块作用域内。

要使用变量模板，必须指定它的类型。例如，下面的代码使用了声明 $\pi <>$ 作用域的两个不同变量：

```
1 std::cout << pi<double> << '\n';
2 std::cout << pi<float> << '\n';
```

也可以声明用于不同翻译单元的变量模板：

```
1 // header.hpp:
2 template<typename T> T val{}; // zero initialized value
3
4 // translation unit 1:
5 #include "header.hpp"
6
7 int main()
8 {
9     val<long> = 42;
10    print();
11 }
12
13 // translation unit 2:
14 #include "header.hpp"
15
16 void print()
17 {
18     std::cout << val<long> << '\n'; // OK: prints 42
19 }
```

变量模板也可以有默认模板参数：

```
1 template<typename T = long double>
2 constexpr T pi = T{3.1415926535897932385};
```

可以使用默认或其他类型：

```
1 std::cout << pi<> << '\n'; // outputs a long double
2 std::cout << pi<float> << '\n'; // outputs a float
```

但请注意，必须始终指定尖括号。仅使用 pi 就会导致错误：

```
1 std::cout << pi << '\n'; // ERROR
```

变量模板也可以用非类型参数进行参数化，非类型参数也可以用来对初始化式进行参数化。例如：

```
1 #include <iostream>
2 #include <array>
3
4 template<int N>
5 std::array<int, N> arr{}; // array with N elements, zero-initialized
6
7 template<auto N>
8     constexpr decltype(N) dval = N; // type of dval depends on passed value
9
10 int main()
11 {
12     std::cout << dval<'c'> << '\n'; // N has value 'c' of type char
13     arr[0] = 42; // sets first element of global arr
14     for (std::size_t i=0; i<arr.size(); ++i) { // uses values set in arr
15         std::cout << arr[i] << '\n';
16     }
17 }
```

同样需要注意的是，即使在不同的翻译单元中对同一变量 `std::array<int,10> arr` 进行初始化和迭代时，仍使用全局作用域。

数据成员的变量模板

变量模板的一种应用是定义表示类模板成员的变量。例如，若类模板定义如下：

```
1 template<typename T>
2 class MyClass {
3     public:
4         static constexpr int max = 1000;
5 };
```

允许为 `MyClass<>` 的不同特化定义不同的值，然后可以定义

```
1 template<typename T>
2 int myMax = MyClass<T>::max;
```

这样就可以直接写

```
1 auto i = myMax<std::string>;
```

而不是

```
1 auto i = MyClass<std::string>::max;
```

对于标准类，如

```
1 namespace std {
2     template<typename T> class numeric_limits {
3         public:
4             ...
5             static constexpr bool is_signed = false;
6             ...
7     };
8 }
```

可以定义

```
1 template<typename T>
2 constexpr bool isSigned = std::numeric_limits<T>::is_signed;
```

可以这样写

```
1 isSigned<char>
```

而不是

```
1 std::numeric_limits<char>::is_signed
```

类型特征后缀_v

C++17 后，标准库使用变量模板技术为标准库中产生 (布尔) 值的所有类型特征定义快捷方式。比如，

```
1 std::is_const_v<T> // since C++17
```

而不是

```
1 std::is_const<T>::value // since C++11
```

标准库定义为

```
1 namespace std {
2     template<typename T> constexpr bool is_const_v = is_const<T>::value;
3 }
```

5.7. 双重模板参数

允许模板参数本身是类模板。同样，以堆栈类模板作为示例。

要为堆栈使用不同的内部容器，必须两次指定元素类型。因此，要指定内部容器的类型，必须再次传递容器的类型及其元素的类型：

```
1 Stack<int, std::vector<int>> vStack; // integer stack that uses a vector
```

使用模板模板参数，可以通过指定容器的类型来声明 Stack 类模板，无需重新指定容器元素的类型：

```
1 Stack<int, std::vector> vStack; // integer stack that uses a vector
```

为此，必须将第二个模板参数指定为一个双重模板参数。如下所示：

C++17 前，这个版本实现有一个问题(稍后会进行解释)。但是，这影响默认值 std::deque。
因此，在讨论如何在 C++17 前处理它前，可以用这个默认值来说明模板参数的一般特性。

basics/stack8decl.hpp

```
1 template<typename T,
2     template<typename Ele> class Cont = std::deque>
3 class Stack {
4 private:
5     Cont<T> elems; // elements
6
7 public:
8     void push(T const&); // push element
9     void pop(); // pop element
10    T const& top() const; // return top element
11    bool empty() const { // return whether the stack is empty
12        return elems.empty();
13    }
14    ...
15};
```

不同的是，第二个模板参数声明为类模板：

```
1 template<typename Ele> class Cont
```

默认值从 std::deque<T> 变为 std::deque。这个参数必须是类模板，其作为第一个模板参数类型的实例化：

```
1 Cont<T> elems;
```

使用第一个模板参数实例化第二个模板参数是本例的特别之处。通常，可以使用类模板中的类型实例化双重模板参数。

通常，可以使用关键字 `class` 来代替 `typename` 作为模板参数。C++11 前，`Cont` 只能用类模板名称替代。

```
1 template<typename T,
2     template<class Ele> class Cont = std::deque>
3 class Stack { // OK
4     ...
5};
```

C++11 后，也可以用别名模板的名称来替换 `Cont`。直到 C++17 才做了相应的更改，允许使用关键字 `typename`，而不是 `class`，来声明一个模板模板参数：

```
1 template<typename T,
2     template<typename Ele> typename Cont = std::deque>
3 class Stack { // ERROR before C++17
```

```
4 ...  
5 };
```

这两种的含义完全相同: 使用 `class` 或 `typename` 并不阻止指定别名模板作为与 `Cont` 对应的参数。

因为双重模板参数没有使用, 习惯上会省略其名字(除非提供了文档):

```
1 template<typename T,  
2     template<typename> class Cont = std::deque>  
3 class Stack {  
4     ...  
5 };
```

必须修改相应地成员函数。因此, 必须将第二个模板参数指定为双重模板参数, 这同样适用于成员函数的实现。例如, `push()` 成员函数的实现:

```
1 template<typename T, template<typename> class Cont>  
2 void Stack<T,Cont>::push (T const& elem)  
3 {  
4     elems.push_back(elem); // append copy of passed elem  
5 }
```

虽然双重模板参数是类模板或别名模板的占位符, 但函数模板或变量模板没有相应的占位符。

双重模板参数的匹配

若使用新版 `Stack`, 可能会得到一个错误消息, 默认值 `std::deque` 与双重模板参数 `Cont` 不兼容。问题是在 C++17 之前, 双重模板参数必须与它所替代的模板参数完全匹配, 一些例外与可变参数模板参数相关(参见 12.3.4 节)。没有考虑双重模板参数的默认模板参数, 因此不考虑具有默认值的参数无法实现匹配(在 C++17 中, 会考虑默认参数)。

本例中, C++17 前的问题是 `std::deque` 模板有多个参数: 第二个参数(用于描述分配器)有一个默认值, 但在 C++17 前, 将 `std::deque` 与 `Cont` 参数匹配时没有考虑到这一点。

不过, 有一个解决办法。可以重写类声明, 使 `Cont` 参数包含两个模板参数的容器:

```
1 template<typename T,  
2     template<typename Elemt,  
3         typename Alloc = std::allocator<Elemt>>  
4     class Cont = std::deque>  
5 class Stack {  
6     private:  
7     Cont<T> elems; // elements  
8     ...  
9 };
```

同样, 可以省略 `Alloc`, 因为这里没有使用它。

堆栈模板的最终版本(包括不同元素类型堆栈赋值的成员模板)是这样的:

basics/stack9.hpp

```
1 #include <deque>  
2 #include <cassert>
```

```

3 #include <memory>
4
5 template<typename T,
6         template<typename Elemt,
7                  typename = std::allocator<Elemt>>
8     class Cont = std::deque<Elemt>;
9
10 class Stack {
11
12     private:
13
14     Cont<T> elems; // elements
15
16     public:
17
18     void push(T const&); // push element
19     void pop(); // pop element
20     T const& top() const; // return top element
21     bool empty() const { // return whether the stack is empty
22         return elems.empty();
23     }
24
25     // assign stack of elements of type T2
26     template<typename T2,
27              template<typename Elemt2,
28                      typename = std::allocator<Elemt2>>
29             >class Cont2>
30     Stack<T,Cont>& operator= (Stack<T2,Cont2> const&);
31
32     // to get access to private members of any Stack with elements of type T2:
33     template<typename, template<typename, typename> class>
34     friend class Stack;
35 };
36
37
38 template<typename T, template<typename, typename> class Cont>
39 void Stack<T,Cont>::push (T const& elem)
40 {
41     elems.push_back(elem); // append copy of passed elem
42 }
43
44
45 template<typename T, template<typename, typename> class Cont>
46 T const& Stack<T,Cont>::top () const
47 {
48     assert(!elems.empty());
49     return elems.back(); // return last element
50 }
51

```

```

52 template<typename T, template<typename, typename> class Cont>
53     template<typename T2, template<typename, typename> class Cont2>
54     Stack<T, Cont>&
55     Stack<T, Cont>::operator=(Stack<T2, Cont2> const& op2)
56     {
57         elems.clear(); // remove existing elements
58         elems.insert(elems.begin(), // insert at the beginning
59                     op2.elems.begin(), // all elements from op2
60                     op2.elems.end());
61         return *this;
62     }

```

为了访问 op2 的所有成员，将所有其他堆栈实例都声明为友元 (省略模板参数的名称):

```

1 template<typename, template<typename, typename> class>
2 friend class Stack;

```

不过，并不是所有的标准容器模板都可以用于 Cont 参数。例如，`std::array` 就不行，因为它是包含了数组长度的非类型模板参数，在我们的模板参数声明中并不匹配。

下面的代码使用了最终版本的所有功能:

basics/stack9test.cpp

```

1 #include "stack9.hpp"
2 #include <iostream>
3 #include <vector>
4
5 int main()
6 {
7     Stack<int> iStack; // stack of ints
8     Stack<float> fStack; // stack of floats
9
10    // manipulate int stack
11    iStack.push(1);
12    iStack.push(2);
13    std::cout << "iStack.top(): " << iStack.top() << '\n';
14
15    // manipulate float stack:
16    fStack.push(3.3);
17    std::cout << "fStack.top(): " << fStack.top() << '\n';
18
19    // assign stack of different type and manipulate again
20    fStack = iStack;
21    fStack.push(4.4);
22    std::cout << "fStack.top(): " << fStack.top() << '\n';
23
24    // stack for doubles using a vector as an internal container
25    Stack<double, std::vector> vStack;
26    vStack.push(5.5);
27    vStack.push(6.6);

```

```
28 std::cout << "vStack.top(): " << vStack.top() << '\n';
29
30 vStack = fStack;
31 std::cout << "vStack: ";
32 while (! vStack.empty()) {
33     std::cout << vStack.top() << ' ';
34     vStack.pop();
35 }
36 std::cout << '\n';
37 }
```

相应的输出为：

```
iStack.top(): 2
fStack.top(): 3.3
fStack.top(): 4.4
vStack.top(): 6.6
vStack: 4.4 2 1
```

关于模板参数的进一步讨论和示例，请参见第 12.2.3 节、第 12.3.4 节和第 19.2.2 节。

5.8. 总结

- 要访问依赖于模板参数的类型名称，必须使用 `typename` 对名称进行限定。
- 要访问依赖于模板参数的基类成员，必须通过 `this->` 或类名访问。
- 嵌套类和成员函数也可以是模板，一种应用是能够实现具有内部类型转换的泛型操作。
- 模板的构造函数或赋值操作符，不能替换预定义的构造函数或赋值操作符。
- 通过使用带大括号的初始化或显式调用默认构造函数，即使使用内置类型实例化，也可以使用默认值初始化模板的变量和成员。
- 可以为原始数组提供特定的模板，这些模板也可以应用于字符串字面量。
- 当传递原始数组或字符串字面量时，且参数不是引用时，类型在推导过程中会衰变(执行数组到指针的转换)。
- 可以定义变量模板(C++14 起)。
- 也可以使用类模板作为模板参数，或作为双重模板参数。
- 模板参数必须精确匹配。

第 6 章 移动语义与 enable_if<>

C++11 引入的最亮眼的特性是移动语义。通过将内部资源从源对象移动(“窃取”)到目标对象，而不是复制这些内容，可以使用它来优化复制和赋值。若原始值不再需要内部值或状态(因为即将丢弃)，就可以移动。

移动语义对模板的设计有很重要的影响，在通用代码中需要引入特殊的规则来支持移动语义。本章将对这些特性进行介绍。

6.1. 完美转发

假设想使用泛型代码来转发传递参数：

- 可修改的对象应该在转发时，保持可修改的属性，以便它们可以修改。
- 常量对象应作为只读对象转发。
- 可移动的对象(可以“窃取”的对象，因为其即将销毁)应该作为可移动对象转发。

要在没有模板的情况下实现此功能，必须对这三种情况进行编程。例如，将 f() 的调用转发给函数 g()：

basics/move1.cpp

```
1 #include <utility>
2 #include <iostream>
3
4 class X {
5     ...
6 };
7
8 void g (X&) {
9     std::cout << "g() for variable\n";
10 }
11 void g (X const&) {
12     std::cout << "g() for constant\n";
13 }
14 void g (X&&) {
15     std::cout << "g() for movable object\n";
16 }
17 // let f() forward argument val to g():
18 void f (X& val) {
19     g(val); // val is non-const lvalue => calls g(X&)
20 }
21 void f (X const& val) {
22     g(val); // val is const lvalue => calls g(X const&)
23 }
24
25 void f (X&& val) {
26     g(std::move(val)); // val is non-const lvalue => needs std::move() to call g(X&&)
27 }
```

```

28
29 int main()
30 {
31     X v; // create variable
32     X const c; // create constant
33
34     f(v); // f() for nonconstant object calls f(X&) => calls g(X&)
35     f(c); // f() for constant object calls f(X const&) => calls g(X const&)
36     f(X()); // f() for temporary calls f(X&&) => calls g(X&&)
37     f(std::move(v)); // f() for movable variable calls f(X&&) => calls g(X&&)
38 }

```

这里, f() 将参数转发给 g() 的三种不同实现:

```

1 void f (X& val) {
2     g(val); // val is non-const lvalue => calls g(X&)
3 }
4 void f (X const& val) {
5     g(val); // val is const lvalue => calls g(X const&)
6 }
7 void f (X&& val) {
8     g(std::move(val)); // val is non-const lvalue => needs std::move() to call g(X&&)
9 }

```

移动对象的代码(通过右值引用)与其他代码不同: 需要 `std::move()`, 根据语言规则, 移动语义无法传递。

移动语义不传递是有意为之的。若不是这样, 就会在函数中第一次使用可移动对象时丢失其内部值。

尽管在第三个 f() 中 val 声明为右值引用, 但当用作表达式时, 值类别是非常量左值(参见附录 B), 并且在第一个 f() 中表现为 val。若没有 move(), 将调用 g(X&) 而不是 g(X&&) 来表示非常量左值。

想在通用代码中合并这三种情况, 就要面对一个问题:

```

1 template<typename T>
2 void f (T& val) {
3     g(val);
4 }

```

适用于前两种情况, 但不适用于传递移动对象的(第三种)情况。

C++11 为此引入了完善转发参数的特殊规则。代码如下所示:

```

1 template<typename T>
2 void f (T&& val) {
3     g(std::forward<T>(val)); // perfect forward val to g()
4 }

```

`std::move()` 没有模板参数, 并且为传递的参数“触发”移动语义, 而 `std::forward<>()` 会根据传递的模板参数“转发”移动语义。

不要认为模板参数 `T&&` 的行为就像特定类型 `T` 一样，它们应用完全不同的规则！但在语法上看起来一样：

- 对于特定类型 `X&&` 声明一个参数为右值引用，只能绑定到一个可移动的对象 (`prvalue`，比如临时对象；`xvalue`，比如通过 `std::move()` 传递的对象；详见附录 B)。它总是可变的，所以可以“窃取”其值。

`X const&&` 这样的类型合法，因为“窃取”可移动对象的内部表示需要修改该对象，但在实践中没有提供通用的语义。不过，也可以强制传递带有 `std::move()` 标记的临时变量或对象，但不能修改它们。

- 模板参数 `T` 的 `T&&` 声明为转发引用（也称为通用引用）。

通用引用这个术语由 Scott Meyers 创造，其是一个常见的术语，可以指代“左值引用”或“右值引用”。因为“通用”太通用了，并且使用这种引用的主要原因是转发对象，所以 C++17 标准引入了术语转发引用，但它不会自动转发。这个术语没有描述它是什么，但描述了用于做什么。

可以绑定到可变、不可变（即 `const`）或可移动对象。在函数定义内部，参数可以是可变的、不可变的，也可以引用一个可以“窃取”内部函数的值。

注意，`T` 必须是模板参数的名称。仅依赖模板参数是不够的，对于模板参数 `T`，像 `typename T::iterator&&` 这样的声明只是一个右值引用，而不是转发引用。

因此，完善转发的整个程序如下所示：

basics/move2.cpp

```
1 #include <utility>
2 #include <iostream>
3
4 class X {
5     ...
6 };
7
8 void g (X&) {
9     std::cout << "g() for variable\n";
10 }
11 void g (X const&) {
12     std::cout << "g() for constant\n";
13 }
14 void g (X&&) {
15     std::cout << "g() for movable object\n";
16 }
17
18 // let f() perfect forward argument val to g():
19 template<typename T>
```

```

20 void f (T&& val) {
21     g(std::forward<T>(val)); // call the right g() for any passed argument val
22 }
23
24 int main()
25 {
26     X v; // create variable
27     X const c; // create constant
28
29     f(v); // f() for variable calls f(X&) => calls g(X&)
30     f(c); // f() for constant calls f(X const&) => calls g(X const&)
31     f(X()); // f() for temporary calls f(X&&) => calls g(X&&)
32     f(std::move(v)); // f() for move-enabled variable calls f(X&&) => calls g(X&&)
33 }
```

当然，完美转发也可以与可变参数模板一起使用(参见4.3节的一些示例)。关于完美转发的详细介绍请参见15.6.3节。

6.2. 特殊成员函数模板

成员函数模板也可以用作特殊的成员函数，包括用作构造函数，但这可能会导致意想不到的结果。

看看下面的例子：

basics/specialmemtmpl1.cpp

```

1 #include <utility>
2 #include <string>
3 #include <iostream>
4
5 class Person
6 {
7 private:
8     std::string name;
9 public:
10    // constructor for passed initial name:
11    explicit Person(std::string const& n) : name(n) {
12        std::cout << "copying string-CONSTR for '" << name << "'\n";
13    }
14    explicit Person(std::string&& n) : name(std::move(n)) {
15        std::cout << "moving string-CONSTR for '" << name << "'\n";
16    }
17    // copy and move constructor:
18    Person (Person const& p) : name(p.name) {
19        std::cout << "COPY-CONSTR Person '" << name << "'\n";
20    }
21    Person (Person&& p) : name(std::move(p.name)) {
22        std::cout << "MOVE-CONSTR Person '" << name << "'\n";
```

```

23     }
24 }
25
26 int main()
27 {
28     std::string s = "sname";
29     Person p1(s); // init with string object => calls copying string-CONSTR
30     Person p2("tmp"); // init with string literal => calls moving string-CONSTR
31     Person p3(p1); // copy Person => calls COPY-CONSTR
32     Person p4(std::move(p1)); // move Person => calls MOVE-CONST
33 }
```

这里，有一个具有 string 成员名的类 Person，我们为其提供了初始化构造函数。为了支持移动语义，重载了接受 std::string 参数的构造函数：

- 提供了 string 的构造函数，其名称由传递参数的副本进行初始化：

```

1 Person(std::string const& n) : name(n) {
2     std::cout << "copying string-CONSTR for '" << name << "'\n";
3 }
```

- 提供了一个可移动 string 对象的构造函数，使用 std::move() 来“窃取”下面的值：

```

1 Person(std::string&& n) : name(std::move(n)) {
2     std::cout << "moving string-CONSTR for '" << name << "'\n";
3 }
```

正如预期的那样，对于传入的字符串对象 (lvalues) 调用第一种方法，而对于可移动的对象 (rvalues) 调用后者：

```

1 std::string s = "sname";
2 Person p1(s); // init with string object => calls copying string-CONSTR
3 Person p2("tmp"); // init with string literal => calls moving string-CONSTR
```

除了这些构造函数之外，示例还对复制和移动构造函数有特定的实现，以确定 Person 作为一个整体何时复制/移动：

```

1 Person p3(p1); // copy Person => calls COPY-CONSTR
2 Person p4(std::move(p1)); // move Person => calls MOVE-CONSTR
```

现在用一个泛型构造函数替换两个需要传入 string 的构造函数，完美转发的参数将传递给成员 name：

basics/specialmemtmp2.hpp

```

1 #include <utility>
2 #include <string>
3 #include <iostream>
4
5 class Person
6 {
```

```

7 private:
8     std::string name;
9 public:
10    // generic constructor for passed initial name:
11    template<typename STR>
12    explicit Person(STR&& n) : name(std::forward<STR>(n)) {
13        std::cout << "TMPL-CONSTR for '" << name << "'\n";
14    }
15
16    // copy and move constructor:
17    Person (Person const& p) : name(p.name) {
18        std::cout << "COPY-CONSTR Person '" << name << "'\n";
19    }
20    Person (Person&& p) : name(std::move(p.name)) {
21        std::cout << "MOVE-CONSTR Person '" << name << "'\n";
22    }
23}

```

传递 string 的构造函数工作正常:

```

1 std::string s = "sname";
2 Person p1(s); // init with string object => calls TMPL-CONSTR
3 Person p2("tmp"); // init with string literal => calls TMPL-CONSTR

```

在这种情况下 p2 的构造并没有创建临时字符串, 参数 STR 推导为 char const[4]。将 std::forward<STR> 应用于构造函数的指针参数没有太大的效果, 因此 name 成员是由一个以 null 结尾的字符串 (进行构造)。

但当我们试图调用复制构造函数时, 会得到一个错误:

```
1 Person p3(p1); // ERROR
```

当用一个可移动对象初始化一个新的 Person 时, 仍然可以正常工作:

```
1 Person p4(std::move(p1)); // OK: move Person => calls MOVE-CONST
```

复制一个常量 Person 也可以正常工作:

```

1 Person const p2c("ctmp"); // init constant object with string literal
2 Person p3c(p2c); // OK: copy constant Person => calls COPY-CONSTR

```

问题是, 根据 C++ 的重载解析规则 (参见 16.2.4 节), 对于一个非常量左值 Person p, 成员模板

```

1 template<typename STR>
2 Person (STR&& n)

```

要比 (通常是预定义的) 复制构造函数匹配的更好:

```
1 Person (Person const& p)
```

STR 只是替换为 Person&, 而对于复制构造函数, 需要转换为 const。

可以考虑通过提供一个非常量复制构造函数来解决这个问题:

```
1 Person (Person& p)
```

这只是部分的解决方案，对于派生类的对象，成员模板仍然是更好的匹配。当传递的参数是 Person 或可以转换为 Person 的表达式时，这里需要的是禁用成员模板。这可以通过使用 std::enable_if<> 来禁用，这将在下一节中介绍。

6.3. 使用 enable_if<> 禁用模板

从 C++11 开始，标准库提供了辅助模板 std::enable_if<>，以在特定的编译时条件下忽略函数模板。

例如，函数模板 foo<>() 有如下定义：

```
1 template<typename T>
2 typename std::enable_if<(sizeof(T) > 4>::type
3 foo() {
4 }
```

如果 sizeof(T)>4 生成 false，则忽略 foo<>() 的定义。

不要忘记将条件放入圆括号中，否则条件中的> 将视为模板参数列表的结束。

如果 sizeof(T)>4 的结果为 true，则函数模板实例展开为

```
1 void foo() {
2 }
```

也就是说，std::enable_if<> 是一种类型特征，计算作为其(第一个)模板参数传递的给定编译时表达式：

- 若表达式结果为 true，其类型成员类型将产生一个类型：
 - 若没有传递第二个模板参数，则该类型为 void。
 - 否则，该类型就是第二个模板参数类型。
- 若表达式结果为 false，则没有定义成员类型。由于名为 SFINAE 的模板特性(替换失败不为过)(请参阅 8.4 节)，这将忽略使用 enable_if 表达式的函数模板。

对于自 C++14 产生类型的类型特征，有一个对应的别名模板 std::enable_if_t<>，允许跳过 typename 和 ::type(请参阅第 2.8 节了解详细信息)。因此，从 C++14 起就使用

```
1 template<typename T>
2 std::enable_if_t<(sizeof(T) > 4>
3 foo() {
4 }
```

若第二个参数传递至 enable_if<> 或 enable_if_t<>：

```
1 template<typename T>
2 std::enable_if_t<(sizeof(T) > 4), T>
3 foo() {
4     return T();
5 }
```

如果表达式为 true，则 `enable_if` 构造展开为第二个参数。因此，若 `MyType` 是传递或推导为 `T` 的具体类型，其大小大于 4，则效果为

```
1 MyType foo();
```

在声明中间使用 `enable_if` 表达式非常笨拙。由于这个原因，使用 `std::enable_if<>` 的常见方法是使用带有默认值的函数模板参数：

```
1 template<typename T,
2         typename = std::enable_if_t<(sizeof(T) > 4>>
3 void foo() {
4 }
```

其会扩展为

```
1 template<typename T,
2         typename = void>
3 void foo() {
4 }
```

当 `sizeof(T) > 4` 时。

若感觉还是太笨拙，并且想让需求/约束更明确，可以使用别名模板为它命名：

```
1 template<typename T>
2 using EnableIfSizeGreater4 = std::enable_if_t<(sizeof(T) > 4>;
3
4 template<typename T,
5         typename = EnableIfSizeGreater4<T>>
6 void foo() {
7 }
```

请参阅第 20.3 节，以了解如何实现 `std::enable_if`。

6.4. 使用 `enable_if<>`

可以使用 `enable_if<>` 来解决在 6.2 节中引入的构造函数模板的问题。

必须解决的问题是，禁用模板构造函数的声明

```
1 template<typename STR>
2 Person(STR&& n);
```

若传递的参数 `STR` 具有正确的类型（即 `std::string`，或可转换为 `std::string` 的类型）。

为此，使用另一个标准类型 `std::is_convertible<FROM,TO>`。C++17 中，相应的声明如下所示：

```
1 template<typename STR,
2         typename = std::enable_if_t<
3             std::is_convertible_v<STR, std::string>>>
4 Person(STR&& n);
```

如果 `STR` 类型可转换为 `std::string` 类型，则整个声明展开为

```
1 template<typename STR,
2         typename = void>
3 Person(STR&& n);
```

如果 STR 类型不能转换为 std::string 类型，将忽略整个函数模板。

为什么不检查 STR 是否“不可转换为 Person”？我们正在定义一个函数，该函数可能允许将字符串转换为 Person。因此构造函数必须知道它是否启用，这取决于是否可转换，而可转换取决于是否启用，循环往复。永远不要在影响 enable_if 使用的条件的地方使用 enable_if。这是编译器不一定能检测到的逻辑错误。

同样，可以使用别名模板定义名称：

```
1 template<typename T>
2 using EnableIfString = std::enable_if_t<
3             std::is_convertible_v<T, std::string>>;
4 ...
5 template<typename STR, typename = EnableIfString<STR>>
6 Person(STR&& n);
```

因此，Person 类如下所示：

basics/specialmentmpl3.hpp

```
1 #include <utility>
2 #include <string>
3 #include <iostream>
4 #include <type_traits>
5
6 template<typename T>
7 using EnableIfString = std::enable_if_t<
8             std::is_convertible_v<T, std::string>>;
9
10 class Person
11 {
12 private:
13     std::string name;
14 public:
15     // generic constructor for passed initial name:
16     template<typename STR, typename = EnableIfString<STR>>
17     explicit Person(STR&& n)
18     : name(std::forward<STR>(n)) {
19         std::cout << "TMPL-CONSTR for '" << name << "'\n";
20     }
21
22     // copy and move constructor:
23     Person (Person const& p) : name(p.name) {
24         std::cout << "COPY-CONSTR Person '" << name << "'\n";
25     }
26
27     Person (Person& p) : name(p.name) {
28         std::cout << "MOVE-CONSTR Person '" << name << "'\n";
29     }
30
31     ~Person() {
32         std::cout << "DESTRUCTOR Person '" << name << "'\n";
33     }
34 }
```

```

25 }
26 Person (Person&& p) : name(std::move(p.name)) {
27     std::cout << "MOVE-CONSTR Person '" << name << "'\n";
28 }
29 };

```

现在，所有行为都符合预期：

basics/specialmemtmpl3.cpp

```

1 #include "specialmemtmpl3.hpp"
2
3 int main()
4 {
5     std::string s = "sname";
6     Person p1(s); // init with string object => calls TMPL-CONSTR
7     Person p2("tmp"); // init with string literal => calls TMPL-CONSTR
8     Person p3(p1); // OK => calls COPY-CONSTR
9     Person p4(std::move(p1)); // OK => calls MOVE-CONST
10 }

```

C++14 中，必须如下声明别名模板，因为_v 版本没有为类型定义 value:

```

1 template<typename T>
2 using EnableIfString = std::enable_if_t<
3     std::is_convertible<T, std::string>::value>;

```

C++11 中，必须声明特殊成员模板，因为_t 版本并没有定义 type:

```

1 template<typename T>
2 using EnableIfString
3 = typename std::enable_if<std::is_convertible<T, std::string>::value
4             >::type;

```

但这些现在都隐藏在 EnableIfString<> 的定义中。

因为它要求类型是隐式可转换的，所以使用 std::is_convertible<> 还有另一种选择。通过使用 std::is_constructible<>，允许显式转换用于初始化。然而，在这种情况下，参数的顺序是反的：

```

1 template<typename T>
2 using EnableIfString = std::enable_if_t<
3     std::is_constructible_v<std::string, T>>;

```

关于 std::is_constructible<> 和 std::is_convertible<> 的详细信息请参见 D.3.2 节。关于在可变参数模板上应用 enable_if<> 的详细信息和示例，请参见 D.6 节。

禁用特殊成员函数

注意，通常不能使用 enable_if<> 来禁用预定义的复制/移动构造函数和/或赋值操作符。原因是成员函数模板永远不会算作特殊成员函数，并且在需要复制构造函数等情况下会忽略。因此，在此声明：

```

1 class C {
2 public:
3     template<typename T>
4     C (T const&) {
5         std::cout << "tmpl copy constructor\n";
6     }
7     ...
8 };

```

当请求一个 C 的副本时，预定义的复制构造函数仍然可以使用：

```

1 C x;
2 C y{x}; // still uses the predefined copy constructor (not the member template)

```

(实际上没有办法使用成员模板，因为没有办法指定或推导它的模板参数 T。)

删除预定义的复制构造函数不是解决方案，因为复制 C 的尝试将导致错误。

不过，有一个解决方案：

可以为 `const volatile` 参数声明复制构造函数，并将其标记为“已删除”(即用 `= delete` 修饰)。这样做可以防止隐式声明另一个复制构造函数。有了这些，就可以定义一个构造函数模板，对于非 `volatile` 类型，该构造函数将优先于(已删除的)复制构造函数：

```

1 class C
2 {
3 public:
4     ...
5     // user-define the predefined copy constructor as deleted
6     // (with conversion to volatile to enable better matches)
7     C(C const volatile&) = delete;
8
9     // implement copy constructor template with better match:
10    template<typename T>
11    C (T const&) {
12        std::cout << "tmpl copy constructor\n";
13    }
14    ...
15 };

```

现在模板构造函数会用于“通常”的复制：

```

1 C x;
2 C y{x}; // uses the member template

```

在模板构造函数中，可以使用 `enable_if<>` 进行约束。例如，若模板参数是整型，为了防止复制类模板 C<> 的对象，可以实现以下方法：

```

1 template<typename T>
2 class C
3 {
4 public:
5     ...

```

```

6 // user-define the predefined copy constructor as deleted
7 // (with conversion to volatile to enable better matches)
8 C(C const volatile&) = delete;
9 // if T is no integral type, provide copy constructor template with better match:
10 template<typename U,
11 typename = std::enable_if_t<!std::is_integral<U>::value>>
12 C (C<U> const&) {
13     ...
14 }
15 ...
16 };

```

6.5. 使用概念简化 `enable_if`<> 表达式

因为它使用了一种变通方法，即使在使用别名模板时，`enable_if` 语法也相当笨拙：为了获得所需的效果，添加了一个的模板参数，并“滥用”该参数来提供函数模板可用的特定要求。这样的代码很难读懂，也使函数模板的其他部分难以理解。

原则上，只需要一种语言特性，允许制定函数的需求或约束，如果需求/约束没有得到满足，函数就会忽略。

这是期待已久的语言特性概念的应用，其允许用自己简单的语法定制模板的需求/条件。但尽管经过长时间的讨论，概念仍然没有成为 C++17 标准的一部分。然而，一些编译器提供了对这种特性的实验性支持，而且概念可能会成为 C++17 后的下一个标准的一部分。

对于概念，正如其作用，只需写下以下内容：

```

1 template<typename STR>
2 requires std::is_convertible_v<STR, std::string>
3 Person(STR&& n) : name(std::forward<STR>(n)) {
4     ...
5 }

```

甚至可以将需求指定为一般概念

```

1 template<typename T>
2 concept ConvertibleToString = std::is_convertible_v<T, std::string>;

```

把这个概念表述为一种需求

```

1 template<typename STR>
2 requires ConvertibleToString<STR>
3 Person(STR&& n) : name(std::forward<STR>(n)) {
4     ...
5 }

```

也可以这样表述：

```

1 template<ConvertibleToString STR>
2 Person(STR&& n) : name(std::forward<STR>(n)) {
3     ...
4 }

```

关于 C++ 概念的详细讨论，请参阅附录 E。

6.6. 总结

- 模板中，通过将参数声明为转发引用 (声明为模板参数名称后跟 `&&` 形成的类型) 并在转发调用中使用 `std::forward<>()`，就可以“完美”地转发参数了。
- 使用完美转发成员函数模板时，可能会比预定义用于复制或移动对象的特殊成员函数更匹配。
- 使用 `std::enable_if<>`，可以在编译时条件为 `false` 时禁用函数模板 (当条件确定，将忽略模板)。
- 通过 `std::enable_if<>`，可以避免为单个参数调用的构造函数模板，或赋值操作符模板，以及比隐式生成的特殊成员函数更好匹配的问题。
- 通过删除 `const volatile` 预定义的特殊成员函数，可以对特殊成员函数进行模板化 (并应用 `enable_if<>`)。
- 概念允许对函数模板需求使用更直观的语法。

第 7 章 使用值还是引用?

从一开始, C++ 就提供了按值和按引用调用, 但要决定选择哪一种并不那么容易: 通常按引用调用对于重要的对象来说成本更低, 但更复杂。C++11 添加了移动语义, 现在有了不同的通过引用传递的方式:

常量右值引用 `X const&&` 也可以, 但没有既定的语义。

1. `X const&` (常量左值引用):

参数引用传递的对象, 但不能修改。

2. `X&` (非常数的左值引用):

参数引用传递的对象, 并能够修改。

3. `X&&` (右值引用):

参数引用传递的对象, 带有移动语义, 可以修改或“窃取”值。

决定如何声明已知具体类型的参数已经够复杂的了。模板中类型是未知的, 因此很难决定哪种传递机制更为合适。

第 1.6.1 节中, 我们确实建议在函数模板中按值传递参数, 除非有很好的理由:

- 不能复制

由于 C++17, 即使没有可用的复制或移动构造函数, 也可以通过值传递临时实体(右值)(参见 B.2.1 节)。因此, 由于 C++17 的约束, 不可能复制左值。

- 参数用于返回数据。
- 模板只是通过保留原始参数的所有属性, 将参数转发到其他地方
- 有显著的性能改进

本章讨论了在模板中声明参数的不同方法, 提出了通过值传递参数的建议, 并为不这样做的原因提供了参数。本文还讨论了在处理字符串字面值和其他数组时遇到的棘手问题。

阅读本章时, 熟悉值类别 (`lvalue`、`rvalue`、`prvalue`、`xvalue` 等) 的术语对理解本章内容会有帮助, 这些在附录 B 中都有解释。

7.1. 按值传递

按值传递参数时, 原则上必须复制每个参数, 每个参数都成为所传递实参的副本。对于类, 作为副本创建的对象通常由复制构造函数初始化。

调用复制构造函数的代价可能会很高。然而, 即使在按值传递参数时, 也有方法来避免复制: 编译器可能会优化复制对象的复制操作, 并且通过移动语义, 对复杂对象的操作也可以变得廉价。

来看一个实现的简单函数模板, 参数通过值传递:

```
1 template<typename T>
2 void printV (T arg) {
3     ...
```

```
4 }
```

为整数调用函数模板时，结果代码为

```
1 void printV (int arg) {  
2     ...  
3 }
```

参数 `arg` 成为传入参数的副本，无论它是否对象、文字还是函数返回的值。

若定义一个 `std::string` 类型，并为此调用函数模板：

```
1 std::string s = "hi";  
2 printV(s);
```

模板参数 `T` 实例化为 `std::string`，得到

```
1 void printV (std::string arg)  
2 {  
3     ...  
4 }
```

传递字符串时，`arg` 变成了 `s` 的副本。这次的副本是由 `string` 类的复制构造函数创建的，这是一个昂贵的操作，因为这个复制操作创建了一个完整的“深”副本，以便该副本内部分配自己的内存来保存该值。

`string` 类的实现本身可能进行了一些优化，以降低复制成本。一种是小字符串优化 (SSO)，只要值不太长，就直接使用对象内部的一些内存来保存值，而不分配内存。另一种是写时复制优化，只要源文件和副本都没有修改，就会使用与源文件相同的内存创建副本。但是，写时复制优化在多线程代码中有明显的缺陷。由于这个原因，C++11 的标准字符串是禁止使用写时复制优化。

但是，复制构造函数并不总调用。考虑以下代码：

```
1 std::string returnString();  
2 std::string s = "hi";  
3 printV(s); // copy constructor  
4 printV(std::string("hi")); // copying usually optimized away (if not, move constructor)  
5 printV(returnString()); // copying usually optimized away (if not, move constructor)  
6 printV(std::move(s)); // move constructor
```

第一次调用中，传递了一个左值，使用了复制构造函数。然而，第二次和第三次调用中，当直接调用函数模板来获取 `prvalues`(动态创建或由另一个函数返回的临时对象；参见附录 B)，编译器通常会优化传递参数，这样就不会调用复制构造函数了。C++17 起，这种优化是必需的。C++17 前，不能优化复制的编译器，至少需要使用移动语义，这通常会使复制成本降低。最后一次调用中，当传递 `xvalue`(一个现有的带有 `std::move()` 的非常量对象) 时，不再需要 `s` 的值来强制使用移动构造函数。

因此，可以调用 `printV()` 的实现，来声明按值传递的参数，通常只在传递左值(之前创建的对象，因为没有使用 `std::move()` 来传递它，所以在之后仍然可用)时才会很昂贵。不幸的是，这是一个常见的情况。在早期创建对象时，在稍后(经过一些修改)将其传递给其他函数的是常规操作。

值传递的类型衰变

还有一个按值传递的属性: 当按值传递参数时, 类型会衰变。从而数组将转换为指针, 并删除 const 和 volatile 等限定符(就像使用值作为使用 auto 声明的对象的初始化式一样):

术语衰变来自 C 语言, 也适用于从函数到函数指针的类型转换(参见 11.1.1 节)。

```
1 template<typename T>
2 void printV (T arg) {
3     ...
4 }
5
6 std::string const c = "hi";
7 printV(c); // c decays so that arg has type std::string
8
9 printV("hi"); // decays to pointer so that arg has type char const*
10
11 int arr[4];
12 printV(arr); // decays to pointer so that arg has type int*
```

因此, 当传递字符串字面值"hi" 时, 类型 char const[3] 衰变为 char const*, 从而成为 T 的推导类型。

```
1 void printV (char const* arg)
2 {
3     ...
4 }
```

这种行为有利有弊, 简化了对传递的字符串字面量的处理, 但缺点是在 printV() 中, 无法区分传递单个元素的指针和传递数组。因此, 将在 7.4 节中将讨论如何处理字符串字面值和其他数组。

7.2. 按引用传递

来讨论一下引用传递的不同风格。所有情况下, 都不会创建副本(因为参数只引用传递的实参)。另外, 传递参数永不衰变。然而, 有些传递是不可行的, 若传递对这种参数进行了, 在某些情况下, 参数的结果类型可能会出现问题。

7.2.1 传递常量引用

为了避免(不必要的)复制, 传递非临时对象时, 可以使用常量引用。例如:

```
1 template<typename T>
2 void printR (T const& arg) {
3     ...
4 }
```

通过这样的声明, 传递的对象永远不会创建副本:

```
1 std::string returnString();
2 std::string s = "hi";
3 printR(s); // no copy
4 printR(std::string("hi")); // no copy
5 printR(returnString()); // no copy
6 printR(std::move(s)); // no copy
```

即使是 int 也是通过引用传递的，这有点适得其反，但并不重要。

因此：

```
1 int i = 42;
2 printR(i); // passes reference instead of just copying i
```

printR() 会实例化为：

```
1 void printR(int const& arg) {
2     ...
3 }
```

底层实现中，通过引用传递参数是通过传递参数地址实现的。地址编码紧凑时，将地址从调用方传递给被调用方的效率很高。然而，在编译代码时，传递地址会给编译器带来不确定性：使用该地址做什么？理论上，可以对该地址“可达”的值进行修改。因此，编译器必须假设其可能缓存的值（通常在机器寄存器中）在调用之后都无效。重新加载这些值的成本可能非常高。有些读者可能会认为我们传递的是常量引用：难道编译器不能推断出这样的数值其实不会发生更改的吗？不幸的是，情况并非如此，因为调用者可以通过自己的非常量引用修改引用的对象。

使用 `const_cast` 是另一种显式修改引用对象的方法。

内联可以缓和一下：如果编译器可以内联展开调用，就可以推断调用和被调用在一起，并且在许多情况下“看到”地址，除了传递底层值外，没有其他用途。函数模板通常很短，因此可以用于内联扩展。但若模板封装了更复杂的算法，就不太可能使用内联了。

传递引用类型不会衰变

当通过引用将参数进行传递时，就不会衰变。从而不会将数组转换为指针，并且不删除 `const` 和 `volatile` 等限定符。因为调用参数声明为 `T const&`，所以模板参数 `T` 本身并没有推导为 `const`。例如：

```
1 template<typename T>
2 void printR (T const& arg) {
3     ...
4 }
5
6 std::string const c = "hi";
7 printR(c); // T deduced as std::string, arg is std::string const&
8
9 printR("hi"); // T deduced as char[3], arg is char const(&)[3]
10
11 int arr[4];
```

```
12 printR(arr); // T deduced as int[4], arg is int const(&)[4]
```

因此，printR() 中使用 T 类型声明的局部对象不是常量。

7.2.2 传递非常量引用

当通过传递的参数作为返回值时(例如，使用 out 或 inout 参数时)，必须使用非常量引用(除非通过指针传递)。在传递参数时，不会创建副本，被调用函数模板的参数只能直接访问传递的参数。

来看看以下代码：

```
1 template<typename T>
2 void outR (T& arg) {
3     ...
4 }
```

通常不允许对一个临时的(prvalue)或一个通过 std::move()(xvalue)传递的现有对象调用 outR()：

```
1 std::string returnString();
2 std::string s = "hi";
3 outR(s); // OK: T deduced as std::string, arg is std::string&
4 outR(std::string("hi")); // ERROR: not allowed to pass a temporary (prvalue)
5 outR(returnString()); // ERROR: not allowed to pass a temporary (prvalue)
6 outR(std::move(s)); // ERROR: not allowed to pass an xvalue
```

可以传递非常量类型的数组，也不会衰变：

```
1 int arr[4];
2 outR(arr); // OK: T deduced as int[4], arg is int(&)[4]
```

因此，可以修改元素(处理数组的大小)。例如：

```
1 template<typename T>
2 void outR (T& arg) {
3     if (std::is_array<T>::value) {
4         std::cout << "got array of " << std::extent<T>::value << " elems\n";
5     }
6     ...
7 }
```

然而，模板在这里有点棘手。若传递 const 参数，可能导致 arg 变成一个常量引用的声明，这意味着允许传递右值，但这里需要左值：

```
1 std::string const c = "hi";
2 outR(c); // OK: T deduced as std::string const
3 outR(returnConstString()); // OK: same if returnConstString() returns const string
4 outR(std::move(c)); // OK: T deduced as std::string const
5 outR("hi"); // OK: T deduced as char const[3]
```

当传递 std::move(c) 时，std::move() 首先将 c 转换为 std::string const&&，然后将 T 推导为 std::string const。

这种情况下，修改函数模板中传递的参数是错误的。在表达式中传递常量对象是可能的，但当函数完全实例化时（这可能发生在编译的后期），修改该值的尝试都将触发错误（然而，这可能发生在被调用模板的内部；见 9.4 节）。

如果想禁用将常量对象传递给非常量引用，可以这样做：

- 使用静态断言触发编译时错误：

```
1 template<typename T>
2 void outR (T& arg) {
3     static_assert(!std::is_const<T>::value,
4                 "out parameter of foo<T>(T&) is const");
5     ...
6 }
```

- 使用 `std::enable_if`（参见章节 6.3）禁用此模板：

```
1 template<typename T,
2         typename = std::enable_if_t::value>
3 void outR (T& arg) {
4     ...
5 }
```

或概念，若支持（见章节 6.5 和附录 E）：

```
1 template<typename T>
2 requires !std::is_const_v<T>
3 void outR (T& arg) {
4     ...
5 }
```

7.2.3 通过转发引用进行传递

使用引用调用的原因是能够完美地转发参数（参见第 6.1 节）。但当使用转发引用（定义为模板形参的右值引用）时，需要使用特殊的规则。考虑以下代码：

```
1 template<typename T>
2 void passR (T&& arg) { // arg declared as forwarding reference
3     ...
4 }
```

可以把所有的东西都传递给转发引用，和通过引用传递一样，不会创建副本：

```
1 std::string s = "hi";
2 passR(s); // OK: T deduced as std::string& (also the type of arg)
3 passR(std::string("hi")); // OK: T deduced as std::string, arg is std::string&&
4 passR(returnString()); // OK: T deduced as std::string, arg is std::string&&
5 passR(std::move(s)); // OK: T deduced as std::string, arg is std::string&&
6 passR(arr); // OK: T deduced as int(&)[4] (also the type of arg)
```

但是，类型推导的特殊规则可能会导致一些意外发生：

```

1 std::string const c = "hi";
2 passR(c); // OK: T deduced as std::string const&
3 passR("hi"); // OK: T deduced as char const(&)[3] (also the type of arg)
4 int arr[4];
5 passR(arr); // OK: T deduced as int (&)[4] (also the type of arg)

```

每一种情况下，`passR()` 内部的参数 `arg` 有一个类型，“知道”传递的是右值(使用移动语义)还是常量/非常量左值。这是传递参数的唯一方法，这样就可以用来区分这三种情况的行为。

这给人的印象是，将参数声明为转发引用几乎完美。但注意，天下没有免费的午餐。

例如，模板参数 `T` 隐式地成为引用类型的唯一情况。因此，使用 `T` 声明局部对象而不进行初始化可能会出错：

```

1 template<typename T>
2 void passR(T&& arg) { // arg is a forwarding reference
3     T x; // for passed lvalues, x is a reference, which requires an initializer
4     ...
5 }
6
7 passR(42); // OK: T deduced as int
8 int i;
9 passR(i); // ERROR: T deduced as int&, which makes the declaration of x in passR() invalid

```

有关如何处理这种情况的详细信息，请参阅第 15.6.2 节。

7.3. 使用 `std::ref()` 和 `std:: cref()`

C++11 起，可以让调用者决定函数模板参数是通过值传递，还是通过引用传递。当模板声明为按值接受参数时，调用者可以使用在头文件 `<functional>` 中声明的 `std::cref()` 和 `std::ref()`，通过引用传递参数。例如：

```

1 template<typename T>
2 void printT (T arg) {
3     ...
4 }
5
6 std::string s = "hello";
7 printT(s); // pass s by value
8 printT(std::cref(s)); // pass s "as if by reference"

```

注意 `std::cref()` 不会改变模板中参数的处理，之类使用了一个技巧：用一个引用的对象来包装传递的参数。事实上，这会创建了一个 `std::reference_wrapper<T>` 类型的对象，引用原始参数，并按值传递了这个对象。包装器或多或少支持一种操作：返回原始类型的隐式类型转换，生成原始对象。

还可以在引用包装上调用 `get()` 并将其用作函数对象。

因此，只要对传递的对象有有效的操作符，就可以使用引用包装器。例如：

basics/cref.cpp

```
1 #include <functional> // for std::cref()
2 #include <string>
3 #include <iostream>
4
5 void printString(std::string const& s)
6 {
7     std::cout << s << '\n';
8 }
9
10 template<typename T>
11 void printT (T arg)
12 {
13     printString(arg); // might convert arg back to std::string
14 }
15
16 int main()
17 {
18     std::string s = "hello";
19     printT(s); // print s passed by value
20     printT(std::cref(s)); // print s passed "as if by reference"
21 }
```

最后一次调用通过值传递 `std::reference_wrapper<string const>` 类型的对象到参数 `arg`, 然后传递并转换回其底层类型 `std::string`。

编译器必须知道返回原始类型必要的隐式转换。因此, 只有通过泛型代码将对象传递给非泛型函数时, `std::ref()` 和 `std::cref()` 才能正常工作。例如, 因为没有为 `std::reference_wrapper<T>` 定义输出操作符, 直接尝试输出传递的泛型类型 `T` 对象将失败:

```
1 template<typename T>
2 void printV (T arg) {
3     std::cout << arg << '\n';
4 }
5 ...
6 std::string s = "hello";
7 printV(s); // OK
8 printV(std::cref(s)); // ERROR: no operator << for reference wrapper defined
```

此外, 因为不能比较一个引用包装与 `char const*` 或 `std::string`, 所以会失败:

```
1 template<typename T1, typename T2>
2 bool isless(T1 arg1, T2 arg2)
3 {
4     return arg1 < arg2;
5 }
6 ...
7 std::string s = "hello";
8 if (isless(std::cref(s), "world")) ... // ERROR
```

```
9 if (isless(std::cref(s), std::string("world"))) ... // ERROR
```

将 arg1 和 arg2 声明为通用类型 T 也没有帮助:

```
1 template<typename T>
2 bool isless(T arg1, T arg2)
3 {
4     return arg1 < arg2;
5 }
```

因为当编译器试图为 arg1 和 arg2 推导 T 时，类型会冲突。

因此，类 `std::reference_wrapper`<> 的作用是将引用用作“第一个类对象”，可以复制，并通过值传递给函数模板。也可以在类中使用它，例如：保存容器中对象的引用。但是最终需要转换回基础类型。

7.4. 处理字符串字面值和数组

已经看到了模板参数在使用字符串字面值和数组时不同的效果：

- 按值调用会衰变，使其成为指向元素类型的指针。
- 引用调用都不会衰变，因此参数成为仍然是数组。

两者都可能是可工作的，也可能是不工作的。当数组衰变为指针时，将失去区分处理指向元素的指针和处理传递的数组的能力。另一方面，处理可能传递字符串字面值的参数时，因为不同大小的字符串字面值有不同的类型，所以不衰变可能有问题。例如：

```
1 template<typename T>
2 void foo (T const& arg1, T const& arg2)
3 {
4     ...
5 }
6
7 foo("hi", "guy"); // ERROR
```

这里，`foo("hi", "guy")` 无法进行编译，因为”hi”的类型是 `char const[3]`，而”guy”的类型是 `char const[4]`，但模板要求它们具有相同的类型 T。只有当字符串字面值具有相同的长度时，这样的代码才能编译。出于这个原因，强烈建议在测试用例中使用不同长度的字符串字面值。

通过声明函数模板 `foo()` 来通过值传递参数：

```
1 template<typename T>
2 void foo (T arg1, T arg2)
3 {
4     ...
5 }
6
7 foo("hi", "guy"); // compiles, but ...
```

但是，这并不意味着所有的问题都消失了。更糟糕的是，编译时问题可能变成了运行时问题。考虑下面的代码，使用 `operator==` 比较传递的参数：

```

1 template<typename T>
2 void foo (T arg1, T arg2)
3 {
4     if (arg1 == arg2) { // OOPS: compares addresses of passed arrays
5         ...
6     }
7 }
8
9 foo("hi", "guy"); // compiles, but ...

```

因为模板还必须处理来自自己衰变的字符串字面量参数 (例如, 通过 value 调用的函数或赋值给使用 auto 声明的对象), 所以编译器必须知道应该将传递的字符指针解释为字符串。

许多情况下衰变是有用的, 特别是用于检查两个对象 (都作为参数传递, 或者一个作为传递参数, 另一个作为期待参数) 是否具有或转换为相同的类型。典型的用法是完美转发, 但想使用完美转发, 必须将参数声明为转发引用。这种情况下, 可以使用类型特征 std::decay<>() 显式地衰变参数。具体的例子请参阅第 120 页 7.6 节中的 std::make_pair()。

其他类型特征有时也隐式衰变, 例如 std::common_type<>, 会产生两个传递参数类型的通用类型 (参见章节 1.3.3 和章节 D.5)。

7.4.1 字符串字面值和数组的特殊实现

可能需要根据传递的是指针, 还是数组来区分实现。当然, 这要求传递的数组没有衰变。

要区分这些情况, 必须检测是否传递了数组。基本上有两种选择:

- 可以声明模板参数, 使其只对数组有效:

```

1 template<typename T, std::size_t L1, std::size_t L2>
2 void foo(T (&arg1)[L1], T (&arg2)[L2])
3 {
4     T* pa = arg1; // decay arg1
5     T* pb = arg2; // decay arg2
6     if (compareArrays(pa, L1, pb, L2)) {
7         ...
8     }
9 }

```

这里, arg1 和 arg2 必须是数组, 具有相同的元素类型 T, 但 L1 和 L2 大小不同。但请注意, 可能需要多个实现来支持数组 (参见第 5.4 节)。

- 可以使用类型特征来检测是否传递了数组 (或指针):

```

1 template<typename T,
2     typename = std::enable_if_t<std::is_array_v<T>>
3 void foo (T& arg1, T& arg2)
4 {
5     ...
6 }

```

由于这些属于特殊的处理方式，而常用处理数组方式就是使用不同的函数名。当然，更好的方法是确定模板的调用者使用 std::vector 或 std::array。只要字符串字面值是数组，就必须考虑这种情况。

7.5. 处理返回值

对于返回值，还可以决定是通过值返回，还是通过引用返回。但返回引用可能是麻烦的来源，因为引用的东西超出了控制范围。一些情况下，返回引用是常见的编程实践：

- 返回容器或字符串元素 (例如，通过 operator[] 或 front())
- 授予类成员写访问权限
- 返回链式调用的对象 (流的 operator<< 和 operator>>，类对象的 operator=)

此外，通过返回 const 引用来授予成员读权限。

若使用不当，可能会产生麻烦。例如：

```
1 std::string* s = new std::string("whatever");
2 auto& c = (*s)[0];
3 delete s;
4 std::cout << c; // run-time ERROR
```

这里，获得了一个字符串元素的引用，但是当使用这个引用时，底层字符串已经不存在了(创建了一个悬空引用)，并且出现了未定义行为。这个例子有些做作(有经验的程序员可能马上就会注意到问题)，但是事情可能会变得不那么明显。例如：

```
1 auto s = std::make_shared<std::string>("whatever");
2 auto& c = (*s)[0];
3 s.reset();
4 std::cout << c; // run-time ERROR
```

因此，应该确保函数模板按值返回结果。正如本章所讨论的，使用模板参数 T 并不能保证它不是引用，因为 T 有时可能会隐式推导为引用：

```
1 template<typename T>
2 T retr(T&& p) // p is a forwarding reference
3 {
4     return T{...}; // OOPS: returns by reference when called for lvalues
5 }
```

即使 T 是由按值调用推导而来的模板参数，当显式指定模板参数为引用时，也可能成为引用类型：

```
1 template<typename T>
2 T retr(T p) // Note: T might become a reference
3 {
4     return T{...}; // OOPS: returns a reference if T is a reference
5 }
6
7 int x;
8 retr<int&>(x); // retr() instantiated for T as int&
```

安全起见，这里有两个选择：

- 使用类型特征 `std::remove_reference`(参见 D.4 节) 将类型 T 转换为非引用：

```
1 template<typename T>
2 typename std::remove_reference<T>::type retV(T p)
3 {
4     return T{...}; // always returns by value
5 }
```

其他特征，如 `std::decay`(参见 D.4 节)，因为隐式地移除引用，在这里也可能会有用。

- 编译器通过声明返回类型 `auto` 来推断返回类型 (C++14 起；参见第 1.3.2 节)，因为 `auto` 总会衰变：

```
1 template<typename T>
2 auto retV(T p) // by-value return type deduced by compiler
3 {
4     return T{...}; // always returns by value
5 }
```

7.6. 推荐的模板参数声明

正如前几节所述，声明依赖于模板参数类型有不同的方式：

- 通过值传递参数：

这种方法很简单，衰变字符串字面值和数组，但不能为大型对象提供最佳性能。调用者可以决定使用 `std::cref()` 和 `std::ref()` 通过引用传递，但是必须确定这样做的必要性。

- 通过引用传递参数：

这种方法通常可以为大型对象提供更好的性能，特别是在传递参数时

- 现有对象 (lvalue) 到左值引用，
- 临时对象 (prvalue) 或标记为可移动 (xvalue) 的对象将引用右值，
- 或者两者都为转发引用。

这些情况下，参数都不会衰变，所以在传递字符串字面值和其他数组时，可能需要特别注意。对于转发引用，还必须注意使用模板参数隐式推导出引用类型的方法。

不要过于泛化

实践中，函数模板通常不支持任意类型的参数，可以进行了一些约束。例如，可能知道只传递某种类型的 `vector`。这种情况下，最好不要太泛化地声明这样的函数。如前所述，这可能会出现令人惊讶的副作用，可以使用以下声明：

```
1 template<typename T>
2 void printVector (std::vector<T> const& v)
3 {
4     ...
5 }
```

通过在 `printVector()` 中声明参数 `v`, 可以确定传递的 `T` 不能成为引用, 因为 `vector` 不能使用引用作为元素类型。另外, 因为 `std::vector<T>` 的复制构造函数会创建元素副本, 所以按值传递 `vector` 的成本很高。出于这个原因, 将这样的 `vector` 参数声明为按值传递可能不合适。若将参数 `v` 的声明交由类型 `T` 来决定, 那么按值调用和按引用调用之间的区别就不那么明显了。

`std::make_pair()` 的实例

`std::make_pair()` 是一个很好的例子, 演示了决定参数传递机制的缺陷。C++ 标准库中, 可以使用其进行类型推导, 并创建 `std::pair<T1, T2>` 对象 (一个方便的函数模板)。它的声明在不同版本的 C++ 标准中有所不同:

- C++98 中, `make_pair()` 在命名空间 `std` 中声明, 使用引用调用来避免不必要的复制:

```
1 template<typename T1, typename T2>
2 pair<T1,T2> make_pair (T1 const& a, T2 const& b)
3 {
4     return pair<T1,T2>(a,b);
5 }
```

然而, 使用字符串字面值对或不同大小的数组时, 这会导致严重的问题。

请参见 C++ 库的 181 号 issue[LibIssue181]

- C++03 中, 函数定义改为使用按值调用:

```
1 template<typename T1, typename T2>
2 pair<T1,T2> make_pair (T1 a, T2 b)
3 {
4     return pair<T1,T2>(a,b);
5 }
```

如同在问题解决方案的基本原理中所了解到的那样, “与其他两个建议相比, 这似乎是对标准的一个小修改, 而且任何效率方面的担忧都由该解决方案的优点所抵消。”

- C++11 中, `make_pair()` 必须支持移动语义, 因此参数必须成为转发引用。因此, 该定义又发生了如下变化:

```
1 template<typename T1, typename T2>
2 constexpr pair<typename decay<T1>::type, typename decay<T2>::type>
3 make_pair (T1&& a, T2&& b)
4 {
5     return pair<typename decay<T1>::type,
6                 typename decay<T2>::type>(forward<T1>(a),
7                                         forward<T2>(b));
8 }
```

完整的实现甚至更加复杂: 为了支持 `std::ref()` 和 `std:: cref()`, 该函数还将 `std::reference_wrapper` 的实例展开为实际的引用。

C++ 标准库现在在许多地方以类似的方式完美地转发传递的参数, 并与 `std::decay<T>` 一起使用。

7.7. 总结

- 测试模板时，可以使用不同长度的字符串字面值。
- 通过值传递的模板参数会衰变，而通过引用传递的模板参数不会衰变。
- 类型特征 `std::decay<>` 允许在引用传递的模板中衰变参数。
- 某些情况下，函数模板声明参数通过值传递时，允许通过 `std::cref()` 和 `std::ref()` 传递参数的引用。
- 按值传递模板参数很简单，但可能无法获得最佳性能。
- 按值传递参数给函数模板，除非有很好的理由不这样做。
- 确保返回值通常按值传递（模板参数不能直接指定返回类型）。
- 有时需要衡量性能。不要依赖直觉，因为直觉很可能是错的。

第 8 章 编译时编程

C++ 有一些在编译时计算的方法。模板添加了更多编译时计算的方法，而语言的发展也对此进行了增强。

并且可以决定是否使用某些模板代码，或者在不同的模板代码之间进行选择。但若所有必要的输入都可用，编译器可以在编译时计算控制流的结果。

事实上，C++ 可以通过多种特性来支持编译时编程：

- C++98 前，模板提供了编译时计算的能力，包括使用循环和执行路径选择（因为使用了非直观的语法，所以有些人认为这是对模板特性的“滥用”）。
- 使用偏特化，可以在编译时根据约束或要求在不同的类模板实现之间进行选择。
- 使用 SFINAE，可以针对类型或约束在函数模板实现之间进行选择。
- C++11 和 C++14 中，通过使用“直观的”执行路径选择，以及自 C++14 后的大多数语句类型（包括 for 循环、switch 语句等）的 constexpr 特性，编译时计算得到了更好的支持。
- C++17 引入了“编译时 if”解除依赖于编译时条件或约束的语句，其甚至可以在模板外工作。

本章将介绍这些特性，特别关注模板的角色和上下文。

8.1. 模板元编程

模板在编译时实例化（与动态语言相反，动态语言在运行时处理泛型）。C++ 模板的一些特性可以与实例化过程结合，成为一种递归“编程语言”。

Erwin Unruh 通过在编译时计算素数，成为第一个发现编译时计算的人。详见第 23.7 节。

因此，模板可以用来“计算”。第 23 章将详细介绍所有功能，这里只是举一个简单的例子。

下面的代码，在编译时找出给定的数字是否为素数：

basics/isprime.hpp

```
1 template<unsigned p, unsigned d> // p: number to check, d: current divisor
2 struct DoIsPrime {
3     static constexpr bool value = (p%d != 0) && DoIsPrime<p,d-1>::value;
4 };
5
6 template<unsigned p> // end recursion if divisor is 2
7 struct DoIsPrime<p,2> {
8     static constexpr bool value = (p%2 != 0);
9 };
10
11 template<unsigned p> // primary template
12 struct IsPrime {
13     // start recursion with divisor from p/2:
14     static constexpr bool value = DoIsPrime<p,p/2>::value;
15 };
16
```

```
17 // special cases (to avoid endless recursion with template instantiation):
18 template<>
19 struct IsPrime<0> { static constexpr bool value = false; };
20 template<>
21 struct IsPrime<1> { static constexpr bool value = false; };
22 template<>
23 struct IsPrime<2> { static constexpr bool value = true; };
24 template<>
25 struct IsPrime<3> { static constexpr bool value = true; };
```

`IsPrime<>` 模板返回成员值，无论传递的模板参数 p 是否是一个素数。为了实现，实例化 `DoIsPrime<>`，其递归地展开为一个表达式，检查 $p/2$ 和 2 之间的每个除数 d 是否能整除 p 。

例如，表达式为

```
1 IsPrime<9>::value
```

扩展为

```
1 DoIsPrime<9, 4>::value
```

继续扩展

```
1 9%4!=0 && DoIsPrime<9, 3>::value
```

继续扩展

```
1 9%4!=0 && 9%3!=0 && DoIsPrime<9, 2>::value
```

继续扩展

```
1 9%4!=0 && 9%3!=0 && 9%2!=0
```

计算结果为 `false`，因为 $9\%3$ 是 0。

正如这个实例链所示：

- 使用递归展开 `DoIsPrime<>` 来遍历从 $p/2$ 到 2 的所有除数，以确定这些除数是否能整除给定整数。
- 当 d 等于 2 时，`DoIsPrime<>` 的偏特化作为结束递归。

注意，所有这些都在编译时完成。也就是说，

```
1 IsPrime<9>::value
```

在编译时展开为 `false`。

模板语法可以说是笨拙的，但类似的代码自 C++98(或更早) 就一直有效，并且可以提高运行库的效率。

C++11 前，通常将值成员声明为枚举数常量，而不是静态数据成员，以避免静态数据成员需要在类外定义(参见第 23.6 节了解详细信息)。例如：

```
1 enum f value = (p%d != 0) && DoIsPrime<p, d-1>::value g;
```

详见第 23 章。

8.2. 使用 `constexpr` 进行计算

C++11 引入了一个新特性 `constexpr`, 简化了各种形式的编译时计算。特别是, 如果有输入适当, 可以在编译时对 `constexpr` 函数求值。虽然在 C++11 中引入 `constexpr` 函数有严格限制(每个 `constexpr` 函数本质上都需要包含一个 `return` 语句), 但在 C++14 中, 这些限制大部分都取消了。当然, 成功地计算 `constexpr` 函数仍然需要所有的计算步骤, 在编译时是可行和有效的: 这排除了堆分配或抛出异常的情况。

我们测试一个数字是否是素数的例子, 可以使用 C++11 进行如下实现:

basics/isprime11.hpp

```
1 constexpr bool
2 doIsPrime (unsigned p, unsigned d) // p: number to check, d: current divisor
3 {
4     return d!=2 ? (p%d!=0) && doIsPrime(p,d-1) // check this and smaller divisors
5     : (p%2!=0); // end recursion if divisor is 2
6 }
7
8 constexpr bool isPrime (unsigned p)
9 {
10    return p < 4 ? !(p<2) // handle special cases
11    : doIsPrime(p,p/2); // start recursion with divisor from p/2
12 }
```

由于只有一条语句的限制, 只能使用条件操作符作为选择机制, 并且需要递归来遍历元素。但其语法是普通的 C++ 函数代码, 这使得它比依赖于模板实例化的第一个版本更容易使用。

C++14 中, `constexpr` 函数可以使用通用 C++ 代码中的控制结构。因此, 不用编写笨拙的模板代码或有些“奇怪的”单行程序, 现在只使用普通的 `for` 循环:

basics/isprime14.hpp

```
1 constexpr bool isPrime (unsigned int p)
2 {
3     for (unsigned int d=2; d<=p/2; ++d) {
4         if (p % d == 0) {
5             return false; // found divisor without remainder
6         }
7     }
8     return p > 1; // no divisor without remainder found
9 }
```

使用 C++11 和 C++14 版本的 `constexpr isPrime()` 实现, 可以直接调用

```
1 isPrime(9)
```

找出 9 是否为质数。可以在编译时这样做，但不一定要这样做。在需要编译时值的上下文中（例如，数组长度或非类型模板参数），编译器将尝试在编译时计算对 `constexpr` 函数的调用。若无法计算，则会产生错误（因为最后必须生成一个常量）。其他上下文中，编译器在编译时可能尝试或不尝试求值，但若这样的求值失败，是不会产生错误信息，而是将问题留给运行时。

2017 年写这本书的时候，编译器似乎确实在尝试编译时求值，即使不严格的情况下。

例如：

```
1 constexpr bool b1 = isPrime(9); // evaluated at compile time
```

编译时计算该值，也适用于

```
1 const bool b2 = isPrime(9); // evaluated at compile time if in namespace scope
```

假设 `b2` 是全局定义的或在命名空间中定义的。块作用域中，编译器可以决定是在编译时计算，还是在运行时计算。

理论上，即使使用了 `constexpr`，编译器也可以决定在运行时计算 `b` 的初始值。编译器只需要在编译时检查它是否可计算即可。

例如，这样：

```
1 bool fiftySevenIsPrime() {
2     return isPrime(57); // evaluated at compile or running time
3 }
```

编译器可能会在编译时计算对 `isPrime` 的调用。

另外：

```
1 int x;
2 ...
3 std::cout << isPrime(x); // evaluated at run time
```

将生成在运行时计算 `x` 是否为素数的代码。

8.3. 使用偏特化的执行路径选择

`isPrime()` 等编译时测试的一种应用是，在编译时使用偏特化在不同实现之间进行选择。

例如，可以根据模板参数是否是素数来选择不同的实现：

```
1 // primary helper template:
2 template<int SZ, bool = isPrime(SZ)>
3 struct Helper;
4
5 // implementation if SZ is not a prime number:
6 template<int SZ>
7 struct Helper<SZ, false>
8 {
```

```

9   ...
10 }
11
12 // implementation if SZ is a prime number:
13 template<int SZ>
14 struct Helper<SZ, true>
15 {
16   ...
17 };
18
19 template<typename T, std::size_t SZ>
20 long foo (std::array<T,SZ> const& coll)
21 {
22   Helper<SZ> h; // implementation depends on whether array has prime number as size
23   ...
24 }
```

根据 `std::array` 参数的大小是否为素数，使用了 `Helper` 类的两种不同实现。这种偏特化的应用广泛适用于函数根据模板参数，选择不同的实现。

上面，使用了两个偏特化来实现两个可能的替代方案，也可以对其中一个替代（默认）情况使用主模板，并对其他情况使用偏特化实现：

```

1 // primary helper template (used if no specialization fits):
2 template<int SZ, bool = isPrime(SZ)>
3 struct Helper
4 {
5   ...
6 };
7
8 // special implementation if SZ is a prime number:
9 template<int SZ>
10 struct Helper<SZ, true>
11 {
12   ...
13 };
```

因为函数模板不支持偏特化，所以必须使用其他机制根据某些约束来更改函数实现。可供的选择包括：

- 带有静态函数的类，
- `std::enable_if`，在第 6.3 节中介绍。
- SFINAЕ 特性，
- 编译时 `if` 特性，该特性从 C++17 引入，将在第 8.5 节中介绍。

第 20 章讨论了基于约束选择函数实现的技术。

8.4. SFINAE(替换失败不为过)

C++ 中，以各种参数类型重载的函数很常见。因此，当编译器看到对重载函数的调用时，必须考虑每个候选函数，评估调用参数，并选择最匹配的候选函数(有关此过程的细节，请参阅附录 C)。

候选集包括函数模板的情况下，编译器首先必须确定为该候选对象使用哪些模板参数，然后在函数参数列表及其返回类型中替换这些参数，然后评估匹配程度(就像普通函数一样)。但替换过程可能会遇到问题：可能产生毫无意义的构造。语言规则并不认为这种无意义的替换会导致错误，而具有这种问题的候选则会直接忽略。

这里，称这个原则为 SFINAE(发音类似于 sfee-nay)，“替换失败不为过”。

这里描述的替换过程不同于按需实例化过程(请参阅第 2.2 节)：即使是不需要实例化，也可以进行替换(编译器评估是否需要)，其会直接替换函数声明(但不是函数体)中的内容。

考虑下面的例子：

basics/len1.hpp

```
1 // number of elements in a raw array:  
2 template<typename T, unsigned N>  
3 std::size_t len (T(&) [N])  
4 {  
5     return N;  
6 }  
7  
8 // number of elements for a type having size_type:  
9 template<typename T>  
10 typename T::size_type len (T const& t)  
11 {  
12     return t.size();  
13 }
```

这里，定义了两个函数模板 len()，接受一个泛型参数：

没有将该函数命名为 size()，避免与 C++ 标准库的命名冲突，C++17 引入了标准函数模板 std::size()。

1. 第一个函数模板将参数声明为 T(&)[N]，从而参数必须是由 N 个 T 类型元素组成的数组。
2. 第二个函数模板将参数声明为 T，没有对参数施加任何约束，而是返回类型 T::size_type，这要求传递的参数类型具有 size_type 成员变量。

当传递数组或字符串字面量时，只有数组的函数模板匹配：

```
1 int a[10];  
2 std::cout << len(a); // OK: only len() for array matches  
3 std::cout << len("tmp"); // OK: only len() for array matches
```

根据签名，第二个函数模板在(分别)用 int[10] 和 char const[4] 替换 T 时也会匹配，这些替换会导致返回类型 T::size_type 中出现错误。因此，对于这些调用，第二个模板将会直接忽略。

当传递 std::vector<> 时，只有第二个函数模板匹配：

```
1 std::vector<int> v;
2 std::cout << len(v); // OK: only len() for a type with size_type matches
```

当传递指针时，两个模板都不匹配（不会失败）。结果，编译器会抱怨没有找到匹配的 len() 函数：

```
1 int* p;
2 std::cout << len(p); // ERROR: no matching len() function found
```

注意，这与传递一个具有 size_type 成员，但没有 size() 成员函数的类型对象不同，例如：std::allocator<>：

```
1 std::allocator<int> x;
2 std::cout << len(x); // ERROR: len() function found, but can't size()
```

当传递这种类型的对象时，编译器会找到第二个函数模板作为匹配的函数模板。因此，这将导致编译错误，即对 std::allocator<int> 的调用 size() 无效，而不是没有找到匹配的 len() 函数。这一次，没有忽略第二个函数模板。

当替换候选对象的返回类型没有意义时，忽略它会导致编译器选择另一个参数匹配更差的候选对象。例如：

basics/len2.hpp

```
1 // number of elements in a raw array:
2 template<typename T, unsigned N>
3 std::size_t len (T (&) [N])
4 {
5     return N;
6 }
7
8 // number of elements for a type having size_type:
9 template<typename T>
10 typename T::size_type len (T const& t)
11 {
12     return t.size();
13 }
14
15 // fallback for all other types:
16 std::size_t len (...)
17 {
18     return 0;
19 }
```

这里，还提供了一个通用的 len() 函数，该函数总是匹配，但其匹配度也是最糟糕的那个（重载解析中使用省略号匹配 …，参见 C.2 节）。

对于数组和 vector，这里有两个匹配，其中特化的匹配更好。对于指针，只有保底候选可以匹配，这样编译器就不会再抱怨这个调用缺少 len() 了。

实践中，保底候选函数通常会提供更有用的默认值、抛出异常或包含静态断言输出更有用的错误消息。

但对于分配器，第二个和第三个函数模板匹配，第二个函数模板会进行更好的匹配。因此，这会导致无法调用 `size()` 成员函数的错误：

```
1 int a[10];
2 std::cout << len(a); // OK: len() for array is best match
3 std::cout << len("tmp"); // OK: len() for array is best match
4
5 std::vector<int> v;
6 std::cout << len(v); // OK: len() for a type with size_type is best match
7
8 int* p;
9 std::cout << len(p); // OK: only fallback len() matches
10
11 std::allocator<int> x;
12 std::cout << len(x); // ERROR: 2nd len() function matches best,
13 // but can't call size() for x
```

关于 SFINAE 的更多细节，请参见 15.7 节和 SFINAE 应用的章节 19.4。

SFINAE 和重载解析

随着时间的推移，SFINAE 原则在模板设计者中变得重要和普遍，以至于它的缩写已经成为一个动词。若打算应用 SFINAE 机制来确保函数模板在某些约束条件下进行忽略，可以说“SFINAE 出了一个函数”，方法是对模板代码进行测试，导致这些约束条件的代码无效。当在 C++ 标准中读到函数模板“不应该参与重载解析，除非…”，这意味着 SFINAE 去除了这个函数模板。

例如，`std::thread` 类声明了一个构造函数：

```
1 namespace std {
2 class thread {
3 public:
4 ...
5 template<typename F, typename... Args>
6 explicit thread(F&& f, Args&&... args);
7 ...
8 };
9 }
```

附注如下：

备注：如果 `decay_t<F>` 与 `std::thread` 类型相同，此构造函数将不参与重载解析。

这意味着如果使用 `std::thread` 作为第一个也是唯一的参数调用模板构造函数，将忽略。原因是成员模板有时可能比预定义的复制或移动构造函数更匹配（请参阅 6.2 节和 16.2.4 节了解详细信息）。当调用线程时，通过 SFINAE 输出构造函数模板，确保另一个线程构造一个线程时，总是使用预定义的复制或移动构造函数。

由于删除了类线程的复制构造函数，这也禁止了复制。

这种技术可能很麻烦。幸运的是，标准库提供了更容易禁用模板的工具。这种特性是 std::enable_if<>，在 6.3 节中介绍。允许通过使用包含禁用条件的构造替换类型来禁用模板。

因此，std::thread 的实际声明如下所示：

```
1 namespace std {
2     class thread {
3         public:
4             ...
5             template<typename F, typename... Args,
6                     typename = enable_if_t<!is_same_v<decay_t<F>,
7                     thread>>>
8             explicit thread(F&& f, Args&&... args);
9             ...
10        };
11    }
```

请参阅第 20.3 节了解如何使用偏特化和 SFINAE 实现 std::enable_if<> 的详细信息。

8.4.1 SFINAE 与 decltype

对于某些条件，找出并制定正确的表达式来 SFINAE 出函数模板并不容易。

例如，想确保函数模板 len() 对于具有 size_type 成员，但没有 size() 成员函数的参数类型就会忽略。函数声明中没有对 size() 成员函数的要求，最终会在实例化时出错：

```
1 template<typename T>
2 typename T::size_type len (T const& t)
3 {
4     return t.size();
5 }
6
7 std::allocator<int> x;
8 std::cout << len(x) << '\n' ; // ERROR: len() selected, but x has no size()
```

有一种常见的模式或习语可以用来处理这种情况：

- 使用尾部返回类型语法指定返回类型（前面使用 auto，在末尾返回类型之前使用 ->）。
- 使用 decltype 和逗号操作符定义返回类型。
- 给出以逗号操作符开头的表达式（在重载逗号操作符时转换为 void）。
- 在逗号操作符的末尾定义一个实际返回类型的对象。

例如：

```
1 template<typename T>
2 auto len (T const& t) -> decltype( (void)(t.size()), T::size_type() )
3 {
4     return t.size();
5 }
```

返回类型是

```
1 decltype( void(t.size)(), T::size_type() )
```

decltype 构造的操作数是一个逗号分隔的表达式列表，因此最后一个表达式 T::size_type() 会产生期望的返回类型的值 (decltype 使用该值转换为返回类型)。在 (最后一个) 逗号之前，必须有有效的表达式，本例中就是 t.size()。将表达式强制转换为 void，是为了避免表达式使用了用户重载定义的逗号操作符。

注意，decltype 的参数是一个未计算的操作数，所以可以创建“虚拟对象”而不调用构造函数，这将在第 11.2.3 节中讨论。

8.5. 编译时 if

偏特化、SFINAE 和 std::enable_if 允许启用或禁用模板。C++17 引入了编译时 if 语句，允许我们根据编译时条件启用或禁用特定语句。使用 if constexpr(…) 语法，编译器使用编译时表达式来决定是应用 then 部分，还是 else 部分 (如果有的话)。

作为第一个例子，4.1.1 节中介绍的可变参数函数模板 print()，使用递归打印参数 (任意类型)。与提供一个单独的函数来结束递归不同，constexpr if 特性决定是否继续递归：

虽然代码读取 if constexpr，但该特性称为 constexpr if，因为它是 if 的“constexpr”形式 (由于历史原因)。

```
1 template<typename T, typename... Types>
2 void print (T const& firstArg, Types const&... args)
3 {
4     std::cout << firstArg << '\n' ;
5     if constexpr(sizeof... (args) > 0) {
6         print(args...); // code only available if sizeof... (args)>0 (since C++17)
7     }
8 }
```

这里，若只对一个参数调用 print()，args 将成为空的参数包，因此 sizeof…(args) 将变为 0。结果，print() 的递归调用变成了一个丢弃的语句，代码没有实例化。因此，相应的函数不需要存在，递归结束。

代码没有实例化意味着只执行第一个翻译阶段，检查正确的语法和不依赖于模板参数的名称 (参见第 1.1.3 节)。

```
1 template<typename T>
2 void foo(T t)
3 {
4     if constexpr(std::is_integral_v<T>) {
5         if (t > 0) {
6             foo(t-1); // OK
7         }
8     }
9     else {
```

```

10 undeclared(t); // error if not declared and not discarded (i.e. T is not integral)
11 undeclared(); // error if not declared (even if discarded)
12 static_assert(false, "no integral"); // always asserts (even if discarded)
13 static_assert(!std::is_integral_v<T>, "no integral"); // OK
14 }
15 }

```

若 `constexpr` 可以用于任何函数，而不仅在模板中。只需要一个产生布尔值的编译时表达式。例如：

```

1 int main()
2 {
3     if constexpr(std::numeric_limits<char>::is_signed) {
4         foo(42); // OK
5     }
6     else {
7         undeclared(42); // error if undeclared() not declared
8         static_assert(false, "unsigned"); // always asserts (even if discarded)
9         static_assert(!std::numeric_limits<char>::is_signed,
10           "char is unsigned"); // OK
11     }
12 }

```

有了这个特性，若给定的大小不是质数，可以使用第 8.2 节中的 `isPrime()` 在编译时执行：

```

1 template<typename T, std::size_t SZ>
2 void foo (std::array<T,SZ> const& coll)
3 {
4     if constexpr(!isPrime(SZ)) {
5         ... // special additional handling if the passed array has no prime number as size
6     }
7     ...
8 }

```

详见 14.6 节。

8.6. 总结

- 模板提供了在编译时进行计算的能力 (使用递归进行迭代，使用偏特化或三元操作符进行选择)。
- 使用 `constexpr` 函数，可以将大多数编译时计算替换为，可在编译时上下文中调用的“普通函数”。
- 使用偏特化，可以根据特定的编译时约束，在类模板的不同实现之间进行选择。
- 模板只在需要的时候使用，在函数模板声明中的替换不会导致代码无效。这个原则称为 SFINAЕ(替换失败不为过)。
- SFINAЕ 只能用于为特定类型和/或约束提供函数模板。
- C++17 起，编译时 `if` 允许根据编译时条件 (甚至在模板外部) 启用或丢弃语句。

第9章 实际使用模板

模板代码与普通代码略有不同，模板介于宏和普通(非模板)声明之间。这可能是一种过度简化，但这不仅影响使用模板编写算法和数据结构的方式，还影响了表达和对模板的分析。

本章中，将讨论其中的一些比较实用的内容，而不探究研究背后的技术细节，这些细节将在第14章中进行探讨。这里，假设C++编译系统由传统的编译器和链接器组成(不属于这一类的C++系统很少)。

9.1. 包含模型

有几种方法可以组织模板源码。本节介绍一种流行的方法：包含模型。

9.1.1 连接错误

大多数C和C++开发者的非模板代码大致组织如下：

- 类和其他类型声明完全放在头文件中。通常，这个文件名扩展名为.hpp(或.h, .H, .hh, ..hxx)的文件。
- 对于全局(非内联)变量和(非内联)函数，只有声明放在头文件中，定义放在实现单元编译文件中。这样的文件通常的展名为.cpp(或.c, .C, .cc, 或.cxx)的文件。

其使所需的类型定义在整个工程中都可以获得，并避免了链接器中变量和函数的重复定义的错误。

了解了这些习惯，下面的(错误的)程序展示了开始学习模板开发者所抱怨的常见错误。和“普通代码”一样，可以在头文件中声明模板：

basics/myfirst.hpp

```
1 #ifndef MYFIRST_HPP
2 #define MYFIRST_HPP
3
4 // declaration of template
5 template<typename T>
6 void printTypeof (T const&);
7
8 #endif // MYFIRST_HPP
```

printTypeof()是一个辅助函数的声明，打印一些类型信息。函数实现放在另一个CPP文件中：

basics/myfirst.cpp

```
1 #include <iostream>
2 #include <typeinfo>
3 #include "myfirst.hpp"
4
5 // implementation/definition of template
6 template<typename T>
```

```
7 void printTypeof (T const& x)
8 {
9     std::cout << typeid(x).name() << '\n';
10}
```

该示例使用 typeid 操作符打印一个字符串，该字符串描述传递表达式的类型。返回一个静态类型 std::type_info 的左值，提供了一个成员 name()，显示表达式的类型。C++ 标准上并没有规定 name() 必须返回有意义的东西，但在定义良好的 C++ 实现中，应该获得传递给 typeid 的表达式类型的字符串，该字符串对表达式的类型进行了很好的描述。

某些实现中，这个字符串是混乱的（使用参数类型和作用域的名称进行编码，以区别于其他名称），存在解析器可以将其转换为人类可读的文本。

最后，在另一个 CPP 文件中使用模板，模板声明通过 #include 包含：

basics/myfirstmain.cpp

```
1 #include "myfirst.hpp"
2 // use of the template
3 int main()
4 {
5     double ice = 3.0;
6     printTypeof(ice); // call function template for type double
7 }
```

C++ 编译器可能会接受这个程序，但是链接器可能会报错：printTypeof() 函数没有定义。

这个错误的原因是函数模板 printTypeof() 的定义没有实例化。为了实例化模板，编译器必须知道应该实例化哪个定义，以及应该为哪个模板参数实例化它。但前面的示例中，这两个信息在单独编译的文件中。因此，当编译器看到对 printTypeof() 的调用，但没有看到为 double 实例化这个函数的定义时，只能假设在其他地方提供了这样的定义，并创建了对该定义的引用（供链接器解析）。另一方面，当编译器处理 myfirst.cpp 文件时，没有为特定参数实例化模板指明定义。

9.1.2 头文件中的模板

对于这个问题，常见的解决方案是使用与宏或内联函数相同的方法：在声明模板的头文件中包含模板的定义。

也就是，不提供文件 myfirst.cpp，而是重写 myfirst.hpp，使其包含所有模板声明和模板定义：

basics/myfirst2.hpp

```
1 #ifndef MYFIRST_HPP
2 #define MYFIRST_HPP
3
4 #include <iostream>
5 #include <typeinfo>
6
```

```

7 // declaration of template
8 template<typename T>
9 void printTypeof (T const&);
10
11 // implementation/definition of template
12 template<typename T>
13 void printTypeof (T const& x)
14 {
15     std::cout << typeid(x).name() << '\n';
16 }
17
18 #endif // MYFIRST_HPP

```

这种组织模板的方法称为“包含模型”，程序可以正确地编译、链接和执行。

这种方法大大增加了包含头文件 myfirst.hpp 的成本。本例中，代价是必须包含模板定义所使用的头文件——本例中是 `<iostream>` 和 `<typeinfo>`。这相当于多了数万行代码，因为像 `<iostream>` 这样的头文件其中包含许多模板定义。

这在实践中是一个问题，因为它增加了编译器编译重要程序所需的时间。因此，我们将研究一些方法来解决这个问题，包括预编译头文件(参见 9.3 节)和使用显式模板实例化(参见 14.5 节)。

尽管存在构建时间问题，但仍然建议尽可能使用“包含模型”的方式来组织模板代码，除非有更好的机制可用。写这本书时，这样的机制已经有了：模块，在 17.11 节中会介绍。它们是一种语言机制，允许开发者以某种方式组织代码：编译器可以单独编译所有声明，然后在需要时有效、有选择地导入处理过的声明。

关于“包含模型”的另一个(更微妙的)结果，非内联函数模板与内联函数和宏有所不同，它们没有在调用点展开。当实例化时，会创建函数副本。因为这是一个自动过程，所以编译器最终可能会在两个不同的文件中创建两个副本，而一些链接器在发现同一个函数的两个不同定义时可能会发出错误。理论上，这不是我们所关心的问题：这是 C++ 编译系统要解决的问题。实践中，大多数情况下都没问题，根本不需要处理这个问题。然而，对于创建代码库的大型项目，偶尔会出现这种奇怪的问题。第 14 章中对实例化方案的讨论，并对 C++ 翻译系统(编译器)文档的研究应该有助于解决这些问题。

最后，示例中适用于普通函数模板的内容，也适用于成员函数和类模板的静态数据成员，以及成员函数模板。

9.2. 模板和内联

将函数声明为内联是提高程序运行时间的常用方式。内联说明符旨在提示实现：在调用点内联替换函数体优于通常的函数调用机制。

但是，实现可能会忽略这个提示。因此，内联唯一可以保证的是，允许函数定义在程序中出现多次(出现在需要多次包含的头文件中)。

与内联函数一样，函数模板可以在多个翻译单元中定义。可以通过将定义放在由多个 CPP 文件包含的头文件中来实现。

但这并不意味着函数模板默认使用内联替换，以及何时在调用点内联替换函数模板体是否优于通常的函数调用机制，这完全取决于编译器。也许，在评估内联调用是否会让性能提高方面，编译

器会比人类做得更好。因此，关于内联的具体策略会因编译器而异，甚至取决于特定的编译选项。

然而，可以使用性能监视工具，让开发者比编译器拥有更多的信息，可能有希望重写编译器(例如，在针对特定平台(如手机或特定输入)调优软件时)。有时，这可能与编译器特定的属性有关，如 `noinline` 或 `always_inline`。

函数模板的全特化在这方面就像普通函数一样：定义只能出现一次，除非以内联方式定义(参见第 16.3 节)。有关本主题的更广泛、详细的概述，请参见附录 A。

9.3. 预编译头文件

即使没有模板，C++ 头文件也会变得非常大，因此需要很长时间来编译。模板增加了这种趋势，由于开发者的强烈抗议，驱使供应商了一种称为预编译头文件(PCH)的方案。该方案在标准范围之外运行，并依赖于特定于供应商的选项。尽管如何创建和使用预编译头文件的信息，在具有此特性的各种 C++ 编译系统的文档中有介绍，但对其工作原理多少还要了解一下。

当编译器翻译一个文件时，其会从文件的开头开始，一直翻译到末尾。当处理来自文件(可能来自 `#include` 的文件)的标记时，会调整它的内部状态，包括像向符号表添加条目这样的事情，以便后续查找。这样做的同时，编译器也可以在目标文件中生成代码。

预编译头方案依赖于：许多文件以相同的代码行开始。为了方便讨论，假设每个要编译的文件都以相同的 N 行代码开始。可以编译这 N 行，并将编译器的完整状态保存在一个预编译头中。然后，对于程序中的每个文件，可以重新加载保存的状态，并在第 N+1 行开始编译，重新加载保存的状态的操作比实际编译前 N 行要快几个数量级。然而，首先保存状态的开销通常比编译 N 行代码更大，成本的增加大约在 20% 到 200% 之间。

有效使用预编译头的关键是确保(尽可能多的)文件以最大数量的公共代码行开始。实践中，文件必须以相同的 `#include` 指令开始，这些指令(如前所述)消耗了构建时间的很大一部分。因此，头文件的包含顺序也很重要。例如以下两个文件：

```
1 #include <iostream>
2 #include <vector>
3 #include <list>
4 ...
```

和

```
1 #include <list>
2 #include <vector>
3 ...
```

禁止使用预编译头文件，因为源文件中没有常见的初始状态。

一些开发者决定，与其传递使用预编译头来加速文件的翻译，不如包含一些额外的不必要的头文件。这个决定可以大大简化包含政策的管理。例如，创建一个名为 `std.hpp` 的头文件通常是相对简单的，包含所有标准头文件：

理论上，标准头文件实际上不需要与物理文件相对应。在实践中，确实如此，而且文件非常大。

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 #include <deque>
5 #include <list>
6 ...
```

可以对这个文件进行预编译，然后每个使用标准库的程序文件就可以按如下方式启动：

```
1 #include "std.hpp"
2 ...
```

通常，这将需要一段时间来编译。但若系统有足够的内存，预编译头文件的处理速度快于不进行预编译的标准头文件。标准头文件在这种方式下特别方便，因为它们很少更改，因此 std.hpp 文件的预编译头文件只需要构建一次。否则，预编译头文件通常是项目依赖配置的一部分（例如，会根据需要由构建工具或集成开发环境（IDE）的项目构建工具进行更新）。

管理预编译头文件的方法是创建预编译头文件层，这些头文件层从最广泛使用和最稳定的头文件（例如，std.hpp 头文件）到那些不会随时更改的头文件，因此仍然可以使用预编译的头文件。但若头文件需要大量开发，那创建预编译的头文件所花费的时间，可能比重用它们要多。这种方法的关键概念是，可以重用较稳定层的预编译头，以提高较不稳定头文件的预编译时间。例如，假设除了 std.hpp 头文件（已经预编译过了），还要定义了一个 core.hpp 头文件，包含了一些特定于项目的附加工具，但仍然具有一定程度的稳定性：

```
1 #include "std.hpp"
2 #include "core_data.hpp"
3 #include "core_algos.hpp"
4 ...
```

因为该文件以 #include "standard.hpp" 开始，编译器可以加载相关的预编译头文件，并继续下一行，而无需重新编译所有标准头文件。当文件完全处理后，可以生成一个新的预编译头文件。因为编译器可以加载后一个预编译头文件，多以应用可以使用 #include "core.hpp" 来快速提供对大量功能的访问。

9.4. 解析编译错误

普通的编译错误通常是非常简洁和直指要害的。例如，当编译器说“类 X 没有 ‘fun’ 成员”时，通常不难找出我代码中的问题（例如，可能把 run 错输入为 fun）。模板就不是这样了，来看一些例子。

类型不匹配

考虑以下使用 C++ 标准库例子：

basics/errornovell.cpp

```
1 #include <string>
2 #include <map>
3 #include <algorithm>
```

```

4 int main()
5 {
6     std::map<std::string, double> coll;
7     ...
8     // find the first nonempty string in coll:
9     auto pos = std::find_if (coll.begin(), coll.end(),
10     [] (std::string const& s) {
11         return s != "";
12     });
13 }

```

包含一个相当小的错误: 用于查找集合中第一个匹配字符串的 Lambda 中, 针对给定的字符串进行检查。因为 map 中的元素是键/值对, 所以期望得到 std::pair<std::string const, double>。

GNU C++ 编译器的某个版本会报告以下错误:

```

1 In file included from /cygdrive/p/gcc/gcc61/include/bits/stl_algobase.h:71:0,
2                 from /cygdrive/p/gcc/gcc61/include/bits/char_traits.h:39,
3                 from /cygdrive/p/gcc/gcc61/include/string:40,
4                 from errornovell.cpp:1:
5 /cygdrive/p/gcc/gcc61/include/bits/predefined_ops.h: In instantiation of ' bool __gnu_cxx
6 ::_ops::_Iter_pred<_Predicate>::operator()(_Iterator) [with _Iterator = std::_Rb_tree_i
7 terator<std::pair<const std::__cxx11::basic_string<char>, double> >; _Predicate = main()
8 ::<lambda(const string&)>]':
9 /cygdrive/p/gcc/gcc61/include/bits/stl_algo.h:104:42: required from '_InputIterator
10 std::find_if(_InputIterator, _InputIterator, _Predicate, std::input_iterator_tag)
11 [with _InputIterator = std::_Rb_tree_iterator<std::pair<const std::__cxx11::basic_string
12 <char>, double> >; _Predicate = __gnu_cxx::__ops::__Iter_pred<main()::<lambda(const
13 string&)> >]
14 /cygdrive/p/gcc/gcc61/include/bits/stl_algo.h:161:23: required from '_Iterator std::_
15 find_if(_Iterator, _Iterator, _Predicate) [with _Iterator = std::_Rb_tree_iterator<std::
16 pair<const std::__cxx11::basic_string<char>, double> >; _Predicate = __gnu_cxx::__ops::__
17 Iter_pred<main()::<lambda(const string&)> >]
18 /cygdrive/p/gcc/gcc61/include/bits/stl_algo.h:3824:28: required from '_IIter std::find
19 _if(_IIter, _IIter, _Predicate) [with _IIter = std::_Rb_tree_iterator<std::pair<const
20 std::__cxx11::basic_string<char>, double> >; _Predicate = main()::<lambda(const string&)
21 >]
22 errornovell.cpp:13:29: required from here
23 /cygdrive/p/gcc/gcc61/include/bits/predefined_ops.h:234:11: error: no match for call to
24   ' (main()::<lambda(const string&)>) (std::pair<const std::__cxx11::basic_string<char>,
25   double>&)'
26 { return bool(_M_pred(*__it)); }
27 ^~~~~~
28 /cygdrive/p/gcc/gcc61/include/bits/predefined_ops.h:234:11:
29 note: candidate: bool (*)(const string&) {aka bool (*)
30       (const std::__cxx11::basic_string<char>&)} <conversion>
31 /cygdrive/p/gcc/gcc61/include/bits/predefined_ops.h:234:11:
32 note: candidate expects 2 arguments, 2 provided
33 errornovell.cpp:11:52: note: candidate: main()::<lambda(
34   const string&)>
35                               [] (std::string const& s) {
36                                         ^
37

```

```
18 errornovel1.cpp:11:52: note: no known conversion for
argument 1 from 'std::pair<const std::__cxx11::basic_string<char>,
double>' to 'const string& {aka const std::__cxx11::basic_string
<char>&}'
```

这样的信息开始看起来更像一本小说，这可能会让模板新手感到沮丧。通过对这样的消息进行管理，错误会相对更容易定位。

错误消息的第一部分说错误发生在内部 `predefined_ops.h` 头文件中的函数模板实例中，`errornovel1.cpp` 通过其他头文件包含其中。下面几行中，编译器报告用哪个参数实例化了什么。本例中，一切都从 `errornovel1.cpp` 的第 13 行语句开始：

```
1 auto pos =
2 std::find_if (coll.begin(), coll.end(),
3 [] (std::string const& s) {
4     return s != "";
5 }) ;
```

这导致在 `stl_go.h` 头文件的第 115 行实例化了一个 `find_if` 模板

```
1 _IIter std::find_if(_IIter, _IIter, _Predicate)
```

实例化为

```
1 _IIter = std::__Rb_tree_iterator<std::pair<
2     const std::__cxx11::basic_string<char>, double> >
3 _Predicate = main()::<lambda(const string&)>
```

编译器都会进行报告，不期望这些模板实例化。

我们的例子中，所有类型的模板都需要实例化，只是想知道为什么它不工作。这个信息出现在消息的最后一部分：表示“调用不匹配”意味着函数调用无法解析，因为实参类型和形参类型不匹配。

```
1 (main()::<lambda(const string&)>) (std::pair<const std::__cxx11::basic_string<char>, double>&)
```

导致调用代码错误：

```
1 { return bool (_M_pred(*__it)); }
```

之后，包含“note: candidate:”的行解释了有一个候选类型需要一个 `const` 字符串 `&`，这个候选类型在 `errornovel1.cpp` 的第 11 行定义为 Lambda 函数 `[](std::string const& s)`，并说明了为什么候选类型不适合：

```
no known conversion for argument 1
from 'std::pair<const std::__cxx11::basic_string<char>, double>'
to 'const string& {aka const std::__cxx11::basic_string<char>&}'
```

这也描述了我们的问题。

毫无疑问，错误消息可以呈现的更好。实际问题会在实例化之前发出，而不是使用完全扩展的模板实例化名称，如 std::__cxx11::basic_string<char>，只使用 std::string 可能就足够了。然而，诊断中的所有信息在某些情况下也可能有用。因此，其他编译器提供类似的信息也就不足为奇了（尽管有些使用了上面提到的结构化技术）。

例如，Visual C++ 编译器会输出如下错误信息：

```
1 c:\tools_root\cl\inc\algorithm(166): error C2664: 'bool main::<lambda_b863c1c7cd07048816f454330789acb4>::operator ()(const std::string &) const' : cannot convert argument 1 from 'std::pair<const _Kty,_Ty>' to 'const std::string &'  
2     with  
3     [  
4         _Kty=std::string,  
5         _Ty=double  
6     ]  
7 c:\tools_root\cl\inc\algorithm(166): note: Reason: cannot convert from 'std::pair<const _Kty,_Ty>' to 'const std::string'  
8     with  
9     [  
10        _Kty=std::string,  
11        _Ty=double  
12    ]  
13 c:\tools_root\cl\inc\algorithm(166): note: No user-defined-conversion operator available  
     that can perform this conversion, or the operator cannot be called  
14 c:\tools_root\cl\inc\algorithm(177): note: see reference to function template instantiation '_InIt std::_Find_if_unchecked<std::_Tree_unchecked_iterator<_Mytree>,_Pr>(_InIt,_InIt,_Pr &)' being compiled  
15     with  
16     [  
17         _InIt=std::_Tree_unchecked_iterator<std::_Tree_val<std::_Tree_simple_types<std::pair<const std::string,double>>>,  
18         _Mytree=std::_Tree_val<std::_Tree_simple_types<std::pair<const std::string,double>>>,  
19         _Pr=main::<lambda_b863c1c7cd07048816f454330789acb4>  
20     ]  
21 main.cpp(13): note: see reference to function template instantiation '_InIt std::find_if<std::_Tree_iterator<std::_Tree_val<std::_Tree_simple_types<std::pair<const _Kty,_Ty>>>,> main::<lambda_b863c1c7cd07048816f454330789acb4>(<lambda_b863c1c7cd07048816f454330789acb4>,_InIt,_Pr)' being compiled  
22     with  
23     [  
24         _InIt=std::_Tree_iterator<std::_Tree_val<std::_Tree_simple_types<std::pair<const std::string,double>>>,  
25         _Kty=std::string,  
26         _Ty=double,  
27         _Pr=main::<lambda_b863c1c7cd07048816f454330789acb4>  
28     ]
```

再次提供了实例化链，并告诉哪些参数实例化了代码中的哪些位置，我们看到了两次

```
cannot convert from 'std::pair<const _Kty,_Ty>' to 'const std::string'
with
[
    _Kty=std::string,
    _Ty=double
]
```

缺少 `const` 的编译器

有时泛型代码的问题只是因为某些编译器。考虑下面的例子：

basics/errornovel2.cpp

```
1 #include <string>
2 #include <unordered_set>
3 class Customer
4 {
5     private:
6         std::string name;
7     public:
8         Customer (std::string const& n)
9             : name(n) {
10         }
11         std::string getName() const {
12             return name;
13         }
14     };
15
16 int main()
17 {
18     // provide our own hash function:
19     struct MyCustomerHash {
20         // NOTE: missing const is only an error with g++ and clang:
21         std::size_t operator() (Customer const& c) {
22             return std::hash<std::string>()(c.getName());
23         }
24     };
25
26     // and use it for a hash table of Customers:
27     std::unordered_set<Customer,MyCustomerHash> coll;
28     ...
29 }
```

Visual Studio 2013 或 2015 中，这段代码按预期编译。但是，使用 `g++` 或 `clang` 时，代码编译时会出错。例如，在 `g++ 6.1` 上，第一个错误消息如下所示：

紧接着是 20 多条其他错误消息：

```
32 /cygdrive/p/gcc/gcc61-include/bits/unordered_set.h:95:63: error: 'value' is not a member
of 'std::__not<std::__and<std::__is_fast_hash<main()::MyCustomerHash>, std::__detail::__is_noexcept_hash<Customer, main()::MyCustomerHash> >'^~~~~~
33     typedef __uset_hashtable<_Value, _Hash, _Pred, _Alloc> _Hashtable;
34                                         ^~~~~~
35 /cygdrive/p/gcc/gcc61-include/bits/unordered_set.h:102:45: error: 'value' is not a member
of 'std::__not<std::__and<std::__is_fast_hash<main()::MyCustomerHash>, std::__detail::__is_noexcept_hash<Customer, main()::MyCustomerHash> >'^~~~~~
36 typedef typename _Hashtable::key_type key_type;
37                                         ^~~~~~
...
...
```

同样，阅读错误消息也很困难（甚至找到每个消息的开头和结尾都很麻烦）。其要表达的是头文件中 hashtable_policy.h 在实例化 std::unordered_set<> 中需要

```
1 std::unordered_set<Customer, MyCustomerHash> coll;
```

在实例化中

```
1 noexcept(declval<const _Hash&>() (declval<const _Key&>()))>
```

有匹配的调用

```
1 const main()::MyCustomerHash (const Customer&)
```

(declval<const _Hash&>() 是 main()::MyCustomerHash 类型的表达式。)一个可能的候选是

```
1 std::size_t main()::MyCustomerHash::operator() (const Customer&)
```

声明为

```
1 std::size_t operator() (const Customer& c) {
```

错误信息最后说明了这个问题：

```
passing 'const main()::MyCustomerHash*' as 'this' argument discards qualifiers
```

能看出问题出在哪里吗？

这个 std::unordered_set 类模板的实现要求哈希对象的函数调用操作符是 const 成员函数（参见第 11.1.1 节）；否则，算法内部就会出现错误。

所有其他错误消息从第一个开始级联，并在将 const 限定符简单地添加到哈希函数操作符时消失：

```
1 std::size_t operator() (const Customer& c) const {
2     ...
3 }
```

Clang 3.9 在第一个错误消息的末尾给出了更好的提示，哈希函数的 operator() 没有标记为 const：

```
...
errornovel2.cpp:28:47: note: in instantiation of template class 'std::unordered_set<Customer
, MyCustomerHash, std::equal_to<Customer>, std::allocator<Customer> >' requested here
std::unordered_set<Customer,MyCustomerHash> coll;
^
errornovel2.cpp:22:17: note: candidate function not viable: 'this' argument has type 'const
MyCustomerHash', but method is not marked const
std::size_t operator() (const Customer& c) {
^
```

clang 在这里提到了默认的模板参数，比如 `std::allocator<Customer>`，而 gcc 跳过了它们。

使用多个编译器来测试代码是有帮助的。不仅有助于编写更可移植的代码，而且当一个编译器产生特别难以理解的错误消息时，另一个编译器可能会提供易懂的信息。

9.5. 后记

在头文件和 CPP 文件中组织源代码具有一定规则或以 ODR 形式的结果。附录 A 对此规则进行了讨论。

包含模型是一个实用的方式，主要由 C++ 编译器的现有实践决定。第一个 C++ 实现很特殊：包含的模板定义是隐式的，从而造成了某种分离的错觉（参见第 14 章了解这个原始模型的详细信息）。

第一个 C++ 标准 ([C++98]) 通过导出的模板对模板编译的分离模型提供了支持。分离模型允许标记导出的模板声明在头文件中声明，而它们相应的定义放在 CPP 文件中，非常像非模板代码的声明和定义。与包含模型不同的是，这个模型不基于现有实现的理论模型，而且实现本身比 C++ 标准化委员预期的要复杂得多。它花了五年多的时间才发布了第一个实现（2002 年 5 月），此后的几年里没有出现其他实现。为了更好地使 C++ 标准与现有的实践保持一致，标准化委员会在 C++11 中删除了导出模板。有兴趣了解更多分离模型细节（和陷阱）的读者可以阅读本书第一版的 6.3 和 10.3 节 ([VandevoordeJosuttisTemplates1st])。

有时很容易想象扩展预编译头概念的方法，以便在编译中加载多个头文件，这允许使用更细粒度的方法进行预编译。这里的障碍主要是预处理器：头文件中的宏可能会改变后续头文件的含义。然而，当文件预编译，宏处理就完成了，而试图为其他头文件引入的预处理器效果修订预编译头文件不现实。不久的将来，会有一个称为“模块”的新语言特性（参见 17.11 节）添加到 C++ 中，来解决这个问题（宏定义不能泄露到模块接口中）。

9.6. 总结

- 模板的包含模型是组织模板代码广泛使用的方式。备选方案将在第 14 章中讨论。
- 只有在类或结构外部的头文件中定义函数模板的特化时才需要内联。
- 要利用预编译的头文件，请确保对 #include 保持相同的顺序。
- 调试使用模板的代码很有挑战性。

第 10 章 基本模板的术语

我们已经介绍了 C++ 中模板的基本概念。了解细节之前，来看看会使用到的术语。在 C++ 社区中（甚至在标准的早期版本中），有时术语缺乏准确性。

10.1. “类模板”还是“模板类”？

C++ 中，结构体、类和联合体统称为类类型。如果没有附加的限定条件，纯文本类型中的单词“class”包含通过关键字 `class` 或关键字 `struct` 引入的类类型。

C++ 中，`class` 和 `struct` 之间的唯一区别是，`class` 的默认访问权限是 `private`，而 `struct` 的默认访问权限是 `public`。然而，对于使用新的 C++ 特性的类型，我们更倾向使用 `class`，而对于普通的 C 数据结构，使用 `struct`，这些数据结构可以用作“普通数据”（POD）。

注意，“类类型”包括联合，但“类”不包括。

对于模板类的调用方式有一些混乱：

- 术语“类模板”表示类是一个模板。也就是说，它是一族类的参数化描述。
- 另一方面，已经使用了术语模板类
 - 类模板的同义词。
 - 引用从模板生成的类。
 - 使用模板 id(模板名称后跟在 < 和 > 之间指定的模板参数的组合) 来引用类。

第二和第三个之间的区别有些微妙，不过这并不重要。

由于这种确定性，所以在本书中会避免使用模板类这个术语。

类似地，使用函数模板、成员模板、成员函数模板和变量模板，但不使用模板函数、模板成员、模板成员函数和模板变量。

10.2. 替换、实例化和特化

处理使用模板的源代码时，C++ 编译器必须用具体的模板实参替换模板中的模板形参。有时，这种替换是暂时的：编译器可能需要检查替换是否有效（参见 8.4 节和 15.7 节）。

通过替换模板的具体参数，为模板中的常规类、类型别名、函数、成员函数或变量实际创建定义的过程，称为模板实例化。

但目前还没有标准或公认的术语，来表示通过模板参数替换来创建非定义声明的过程。我们已经看到了一些团队使用的部分实例化或声明的实例化，但这不通用。也许更直观的术语是不完全实例化（类模板的情况下，会生成不完整的类）。

由实例化或不完全实例化（即类、函数、成员函数或变量）产生的实体一般称为特化。

C++ 中实例化过程并不是产生特化的唯一方法。替代机制允许开发者显式地指定与模板参数的特殊替换绑定的声明。如在 2.5 节中看到的，这样的特化是通过前缀 `template<>` 引入的：

```

1 template<typename T1, typename T2> // primary class template
2 class MyClass {
3     ...
4 };
5
6 template<> // explicit specialization
7 class MyClass<std::string, float> {
8     ...
9 };

```

严格地说，这称为显式特化(与实例化或生成的特化相对)。

如 2.6 节所述，仍有模板参数的特化称为偏特化：

```

1 template<typename T> // partial specialization
2 class MyClass<T, T> {
3     ...
4 };
5
6 template<typename T> // partial specialization
7 class MyClass<bool, T> {
8     ...
9 };

```

当说到(显式或部分)特化时，通用模板也称为主模板。

10.3. 声明和定义

目前，“声明”和“定义”这两个词在本书中只出现过几次。然而，这些词在标准 C++ 中有相当精确的含义。

声明是一种 C++ 构造，在 C++ 作用域中引入或重新引入一个名称。此介绍会包含该名称的部分类别，但进行有效声明时不需要详细信息。例如：

```

1 class C; // a declaration of C as a class
2 void f(int p); // a declaration of f() as a function and p as a named parameter
3 extern int v; // a declaration of v as a variable

```

注意，宏定义和 goto 标签在 C++ 中不是声明。

当声明的结构已知时，或对于变量，必须分配存储空间时，声明就变成了定义。对于类类型定义，必须提供带括号的主体。对于函数定义，这就必须提供(一般情况下)用大括号括起来的函数体，或者必须将函数指定为 =default 或 =delete。对于变量，初始化或缺少 extern 说明符会导致声明变成定义。下面是补充上述非定义声明的示例：

默认函数是由编译器提供默认实现的特殊成员函数，例如：默认复制构造函数。

```

1 class C {} // definition (and declaration) of class C
2
3 void f(int p) { // definition (and declaration) of function f()

```

```
4     std::cout << p << '\n' ;
5 }
6
7 extern int v = 1; // an initializer makes this a definition for v
8
9 int w; // global variable declarations not preceded by
10 // extern are also definitions
```

通过扩展，类模板或函数模板的声明若有主体，就称为定义。因此，

```
1 template<typename T>
2 void func (T);
```

声明不是定义，

```
1 template<typename T>
2 class S {};
```

实际上是一个定义。

10.3.1 完整类型与不完整类型

类型可以完整的或不完整的，这是一个与声明和定义之间的区别密切相关的概念。有些语言构造需要完整的类型，而其他语言构造也可以使用不完整的类型。

不完整类型是以下类型之一：

- 已声明但尚未定义的类类型。
- 未指定边界的数组类型。
- 不完整类型的数组类型。
- 无效类型
- 枚举类型，基础类型或枚举值未定义。
- 上面应用 `const` 和/或 `volatile` 的类型。

其他类型都是完整的。例如：

```
1 class C; // C is an incomplete type
2 C const* cp; // cp is a pointer to an incomplete type
3 extern C elems[10]; // elems has an incomplete type
4 extern int arr[]; // arr has an incomplete type
5 ...
6 class C { }; // C now is a complete type (and therefore cp and elems
7           // no longer refer to an incomplete type)
8 int arr[10]; // arr now has a complete type
```

有关如何处理模板中不完整类型的提示，请参阅 11.5 节。

10.4. 定义规则

C++ 语言定义对各种实体的声明施加了一些约束。这些约束的总和称为单一定义规则或 ODR。这个规则的细节有点复杂，并且涉及到很多不同的情况。后面的章节将说明在每个适用的上下文中

产生的各种情况，可以在附录 A 中找到 ODR 的完整描述。现在，记住 ODR 的基础知识就足够了：

- 普通(即，不是模板)非内联函数和成员函数，以及(非内联)全局变量和静态数据成员在整个程序中应该只定义一次。

C++17 后，全局变量和静态变量，以及数据成员都可以定义为内联。这样就不需要在同一个翻译单元中定义它们。

- 类类型(包括结构和联合)、模板(包括偏特化，但不包括全特化)以及内联函数和变量在每个翻译单元中最多定义一次，而且所有这些定义应该相同。

翻译单元是对源文件进行预处理的结果；也就是说，包含了由 #include 指令命名并由宏展开生成的内容。

本书的其余部分中，可链接实体指的是以下任何一种：一个函数或成员函数，一个全局变量或一个静态数据成员，包括从模板生成的，对链接器可见的东西。

10.5. 模板实参与模板形参

比较下面的类模板：

```
1 template<typename T, int N>
2 class ArrayInClass {
3     public:
4         T array[N];
5 }
```

使用普通类：

```
1 class DoubleArrayInClass {
2     public:
3         double array[10];
4 }
```

如果将参数 T 和 N 分别替换为 double 和 10，则后者在本质上等价于前者。C++ 中，这个替换可表示为

```
1 ArrayInClass<double, 10>
```

注意模板名称后面是如何用尖括号将模板参数括起来的。

不管这些实参本身是否依赖于模板形参，模板名后跟尖括号中的实参的组合称为模板标识。

可以像使用非模板一样使用这个名称。例如：

```
1 int main()
2 {
3     ArrayInClass<double, 10> ad;
4     ad.array[0] = 1.0;
5 }
```

区分模板形参和模板实参很重要。简而言之，“参数由实参初始化”。

学术界中，参数有时称为实际参数，而声明参数称为形式参数。

或者更准确地说：

- 模板参数是那些列在模板声明或定义中的关键字 `Template` 之后的参数（示例中是 `T` 和 `N`）。
- 模板实参是替代模板形参的项（示例中是 `double` 和 `10`）。与模板形参不同，模板实参可以不仅仅是“名称”。

当使用模板标识表示时，模板实参对模板形参是显式替换的。但在许多情况下，替换是隐式的（例如，如果模板形参被默认实参替换）。

基本原则是，模板参数必须在编译时确定。这个需求对于模板实体运行时成本有巨大的好处。因为模板形参最终会被编译时的值替代，所以其本身可以形成编译时表达式。这在 `ArrayInClass` 模板中可以用来调整成员数组的大小。数组的大小必须是一个常量表达式，而模板参数 `N` 刚好符合此条件。

可以进一步推进这种方式：因为模板形参是编译时实体，所以也可以用来创建有效的模板形参。下面是一个例子：

```
1 template<typename T>
2 class Dozen {
3     public:
4         ArrayInClass<T, 12> contents;
5 }
```

本例中，名称 `T` 既是模板形参，又是模板实参。因此，可以使用一种机制从更简单的模板构建更复杂的模板。当然，这与组装类型和函数的机制没有区别。

10.6. 总结

- 对于属于模板的类、函数和变量，请分别使用类模板、函数模板和变量模板。
- 模板实例化是通过将模板形参替换为具体实参，来创建常规类或函数的过程。产生的实体是特化模板类型。
- 类型可以是完整的或不完整的。
- 根据单一定义规则 (ODR)：程序中，非内联函数、成员函数、全局变量和静态数据成员应该只定义一次。

第 11 章 通用库

目前，对模板的讨论集中在特定特性、功能和约束上，同时考虑到直接使用和应用程序(这是应用程序开发者会遇到的事情)。然而，模板在用于编写通用库和框架时效率很高。在这些地方，设计必须考虑潜在的用途，这些用途是通用的，所以不受限制。虽然本书中所有的内容都适用于这样的设计，但在编写可移植组件时，应该考虑一些通用性问题，这些组件要适用于尚未想到的类型。

这里提出的问题并不完整，但它总结了迄今为止介绍的一些特性，介绍了一些附加特性，并引用了本书后面涉及的一些特性。我们希望本章除能推动读者阅读后面的章节。

11.1. 可调用类型

许多库包括外部代码向其传递接口。示例包括必须在另一个线程上调度的操作、如何将哈希值存储在哈希表中的函数、对集合中的元素排序的顺序的对象，以及提供默认参数值的通用包装器。标准库在这里也不例外，定义了许多使用此类的组件。

这种情况下使用的术语是回调，作为函数调用参数传递的实体(与模板参数相反)，我们保持了这个传统。例如，`sort` 函数可以包含一个回调参数作为“排序条件”，调用该参数来确定一个元素是否在所需排序顺序的另一个元素之前。

C++ 中，有几种类型可以很好地用于回调，它们既可以作为函数调用参数传递，也可以以 `f(...)` 方式直接使用：

- 函数指针类型
- 具有重载 `operator()`(函数操作符，有时称为函子) 的类类型，包括 Lambda
- 使用转换函数生成指向函数的指针或指向函数引用的类类型

这些类型统称为函数对象类型，这种类型的值就是函数对象。

C++ 标准库引入了更宽泛的可调用类型概念，可以是函数对象类型或成员指针。可调用类型的对象是可调用对象，这里将其称为可调用对象。

泛型代码通常是能够接受任何类型的可调用代码。

11.1.1 函数对象

来看看标准库的 `for_each()` 算法是如何实现的(使用我们的“foreach”以避免名称冲突)：

basics/foreach.hpp

```
1 template<typename Iter, typename Callable>
2 void foreach (Iter current, Iter end, Callable op)
3 {
4     while (current != end) { // as long as not reached the end
5         op(*current); // call passed operator for current element
6         ++current; // and move iterator to next element
7     }
8 }
```

下面的程序演示了该模板与各种函数对象的使用：

basics/foreach.cpp

```
1 #include <iostream>
2 #include <vector>
3 #include "foreach.hpp"
4 // a function to call:
5 void func(int i)
6 {
7     std::cout << "func() called for: " << i << '\n';
8 }
9
10 // a function object type (for objects that can be used as functions):
11 class FuncObj {
12 public:
13     void operator()(int i) const { // Note: const member function
14         std::cout << "FuncObj::op() called for: " << i << '\n';
15     }
16 };
17
18 int main()
19 {
20     std::vector<int> primes = { 2, 3, 5, 7, 11, 13, 17, 19 };
21     foreach(primes.begin(), primes.end(), // range
22             func); // function as callable (decays to pointer)
23
24     foreach(primes.begin(), primes.end(), // range
25             &func); // function pointer as callable
26
27     foreach(primes.begin(), primes.end(), // range
28             FuncObj()); // function object as callable
29
30     foreach(primes.begin(), primes.end(), // range
31             [] (int i) { // lambda as callable
32                 std::cout << "lambda called for: " << i << '\n';
33             });
34 }
```

来仔细看看每种情况:

- 将函数名作为函数参数传递时，实际上传递的不是函数本身，而是指向函数的指针或引用。与数组一样(参见 7.4 节)，函数实参在按值传递时衰变为指针，对于模板参数类型，将推导出指向函数的指针类型。

就像数组一样，函数可以通过引用传递，但函数类型不能用 `const` 限定。若用可调用的 `const&` 类型声明 `foreach()` 的最后一个参数，那么 `const` 将会忽略。(主流 C++ 编码中很少使用函数引用。)

- 第二个调用通过传递函数名的地址显式接受函数指针。这相当于第一次调用(其中函数名隐式衰变为指针值)，但可能更清晰一些

- 传递函数时，将类类型对象作为可调用对象传递。通过类类型调用通常相当于调用其函数操作符。因此，

```
1 op(*current);
```

通常会转化为

```
1 op.operator() (*current); // call operator() with parameter *current for op
```

定义函数操作符时，应该将其定义为常量成员函数。否则，当框架或库希望此调用不会改变传递对象的状态时，就会产生错误消息(详见 9.4 节)。

类类型对象也可以隐式转换为指向代理调用函数的指针或引用(在 C.3.5 节讨论)。这种情况下，调用

```
1 op(*current);
```

会变成

```
1 (op.operator F()) (*current);
```

其中 F 是类类型对象可以转换为的函数指针或函数引用的类型。

- Lambda 表达式产生函数(称为闭包)，这种情况与函数情况没有区别。然而，Lambda 是引入函数的一种非常方便的快捷方式。C++11 后，经常出现在 C++ 代码中。

有趣的是，以 [] 开头的 Lambda(没有捕获) 产生到函数指针的转换操作符。因为闭包的普通函数操作符会匹配得更好，所以从来没将其作为代理函数。

11.1.2 处理成员函数和附加函数

前面的例子中没有使用的实体：成员函数。这是因为调用非静态成员函数通常需要使用 object.memfunc(...) 或 ptr->memfunc(...) 这样的语法指定调用的对象，而这与通常的模式 function-object(...) 不匹配。

C++17 后，标准库提供了一个实用工具 std::invoke()，会将这种情况与普通的函数调用语法统一，从而允许以单一形式调用可调用对象。以下 foreach() 模板的实现使用了 std::invoke()：

basics/foreachinvoke.hpp

```
1 #include <utility>
2 #include <functional>
3 template<typename Iter, typename Callable, typename... Args>
4 void foreach (Iter current, Iter end, Callable op, Args const&... args)
5 {
6     while (current != end) { // as long as not reached the end of the elements
7         std::invoke(op, // call passed callable with
8             args..., // any additional args
9             *current); // and the current element
10        ++current;
11    }
12 }
```

这里除了可调用的参数外，还接受任意数量的附加参数。然后，`foreach()` 模板使用给定的可调用对象调用 `std::invoke()`，再加上附加的给定参数和所引用的元素。`std::invoke()` 处理如下：

- 如果可调用对象是指向成员的指针，则使用第一个附加参数作为 `this` 对象。所有附加参数只是作为参数传递给可调用对象。
- 否则，所有附加参数都只是作为参数传递给可调用对象。

不能在这里对可调用参数或附加参数使用完美转发：第一次调用可能“窃取”值，导致在后续迭代中调用 `op` 时可能会出现意外行为。

有了这个实现，仍然可以编译上面对 `foreach()` 的调用。另外，还可以向可调用对象传递额外的参数，可调用对象也可以是成员函数。

`std::invoke()` 允许将指向数据成员的指针作为回调类型。它不调用函数，而是返回附加参数引用的对象中相应数据成员的值。

下面的代码说明了这一点：

basics/foreachinvoke.hpp

```
1 #include <iostream>
2 #include <vector>
3 #include <string>
4 #include "foreachinvoke.hpp"
5
6 // a class with a member function that shall be called
7 class MyClass {
8 public:
9     void memfunc(int i) const {
10         std::cout << "MyClass::memfunc() called for: " << i << '\n';
11     }
12 };
13
14 int main()
15 {
16     std::vector<int> primes = { 2, 3, 5, 7, 11, 13, 17, 19 };
17     // pass lambda as callable and an additional argument:
18     foreach(primes.begin(), primes.end(), // elements for 2nd arg of lambda
19             [] (std::string const& prefix, int i) { // lambda to call
20                 std::cout << prefix << i << '\n';
21             },
22             "- value: "); // 1st arg of lambda
23
24     // call obj.memfunc() for/with each elements in primes passed as argument
25     MyClass obj;
26     foreach(primes.begin(), primes.end(), // elements used as args
27             &MyClass::memfunc, // member function to call
28             obj); // object to call memfunc() for
29 }
```

`foreach()` 的第一次调用将其第四个参数 (字符串字面量"- value: ") 传递给 Lambda 的第一个参数，而 `vector` 中的当前元素绑定到 Lambda 的第二个参数。第二次调用传递成员函数 `memfunc()` 作为传递 `obj` 作为第四个参数的第三个参数。

有关产生 `std::invoke()` 是否可以使用可调用对象的类型特征，请参见 D.3.1 节。

11.1.3 使用包装函数

`std::invoke()` 的一个常见应用是封装单个函数调用 (例如，记录调用，测量持续时间，或准备一些上下文，例如启动一个新线程)。现在，可以通过完美转发可调用参数和传递参数来支持移动语义：

basics/invoke.hpp

```
1 #include <utility> // for std::invoke()
2 #include <functional> // for std::forward()
3
4 template<typename Callable, typename... Args>
5 decltype(auto) call(Callable&& op, Args&&... args)
6 {
7     return std::invoke(std::forward<Callable>(op), // passed callable with
8                        std::forward<Args>(args)...); // any additional args
9 }
```

另一个有趣的地方在于，如何处理调用函数的返回值，以便将其“完美地”转发回调用者。为了支持返回引用 (比如 `std::ostream&`)，必须使用 `decltype(auto)` 而不是 `auto`：

```
1 template<typename Callable, typename... Args>
2 decltype(auto) call(Callable&& op, Args&&... args)
```

`decltype(auto)(C++14)` 是一个占位符类型，根据相关表达式的类型 (初始化器、返回值或模板参数) 确定变量、返回类型或模板参数的类型。详见 15.10.3 节。

若将 `std::invoke()` 返回的值临时存储在变量中，以便在执行其他操作 (例如，处理返回值或记录调用结束) 后返回，还必须使用 `decltype(auto)` 声明临时变量：

```
1 decltype(auto) ret{std::invoke(std::forward<Callable>(op),
2                               std::forward<Args>(args)...)};
3 ...
4 return ret;
```

注意，用 `auto&&` 声明 `ret` 并不正确。作为一个引用，`auto&&` 扩展返回值的生命周期直到作用域结束 (参见第 11.3 节)，但不超出函数调用者的返回语句。

使用 `decltype(auto)` 也有一个问题：若可调用对象的返回类型为 `void`，则不允许将 `ret` 初始化为 `decltype(auto)`，因为 `void` 是一个不完整的类型。现在，有以下选择：

- 在语句的前一行声明一个对象，其析构函数执行希望实现的可观察行为。例如：

```
1 struct cleanup {
2     ~cleanup() {
3         ... // code to perform on return
```

```

4     }
5 } dummy;
6 return std::invoke(std::forward<Callable>(op),
7                     std::forward<Args>(args)...);

```

- 以不同的方式实现 void 和非 void 的情况:

basics/invokeret.hpp

```

1 #include <utility> // for std::invoke()
2 #include <functional> // for std::forward()
3 #include <type_traits> // for std::is_same<> and invoke_result<>
4
5 template<typename Callable, typename... Args>
6 decltype(auto) call(Callable&& op, Args&&... args)
7 {
8     if constexpr(std::is_same_v<std::invoke_result_t<Callable, Args...>,
9                 void>) {
10         // return type is void:
11         std::invoke(std::forward<Callable>(op),
12                     std::forward<Args>(args)...);
13         ...
14     }
15     else {
16         // return type is not void:
17         decltype(auto) ret{std::invoke(std::forward<Callable>(op),
18                             std::forward<Args>(args)...)};
19         ...
20     }
21     return ret;
22 }
23 }

```

```

1 if constexpr(std::is_same_v<std::invoke_result_t<Callable, Args...>, void>)

```

编译时测试调用的返回类型 Args... 是否有 void。有关 std::invoke_result<> 的详细信息，请参阅 D.3.1 节。

C++17 起可用 std::invoke_result<>。C++11 起，要获得返回类型，可以使用 :**typename std::result_of<Callable(Args...)>::type**

未来的 C++ 可能会避免对 void 的特殊处理 (参见 17.7 节)。

11.2. 实现通用库

std::invoke() 只是标准库为实现泛型库提供的有用实用程序的一个例子。接下来，将看看其他一些重要的特性。

11.2.1 类型特征

标准库提供了各种类型特征工具，允许计算和修改类型。支持泛型，即代码必须适应实例化类型的功能或对其作出反应。例如：

```
1 #include <type_traits>
2 template<typename T>
3 class C
4 {
5     // ensure that T is not void (ignoring const or volatile):
6     static_assert(!std::is_same_v<std::remove_cv_t<T>, void>,
7                   "invalid instantiation of class C for void type");
8 public:
9     template<typename V>
10    void f(V&& v) {
11        if constexpr(std::is_reference_v<T>) {
12            ... // special code if T is a reference type
13        }
14        if constexpr(std::is_convertible_v<std::decay_t<V>, T>) {
15            ... // special code if V is convertible to T
16        }
17        if constexpr(std::has_virtual_destructor_v<V>) {
18            ... // special code if V has virtual destructor
19        }
20    }
21};
```

正如这个例子所示，通过检查某些条件，可以在不同的模板实现之间进行选择。这里，使用编译时 if 特性，从 C++17 开始就可用了(请参阅 8.5 节)，但也可以使用 std::enable_if、偏特化或 SFINAE 来启用或禁用助手模板(请参阅第 8 章的详细信息)。

必须特别小心地使用类型特征：其行为可能与(菜鸟期)开发者所期望有所不同。例如：

```
1 std::remove_const_t<int const&> // yields int const&
```

这里，因为引用不是 const(尽管不能修改)，所以调用没有效果，并生成传递类型。

因此，删除引用和 const 的顺序很重要：

```
1 std::remove_const_t<std::remove_reference_t<int const&>> // int
2 std::remove_reference_t<std::remove_const_t<int const&>> // int const
```

相反，可以只调用

```
1 std::decay_t<int const&> // yields int
```

但也可以将数组和函数转换为相应的指针类型。有一些情况下，类型特征是有限制的，不满足这些条件会导致未定义行为。

有人建议 C++17，要求违反类型特征的前提条件必须导致编译时错误。然而，因为一些类型特征有过度约束的条件，例如总是需要完整的类型，这种建议还未完全付诸于行动。

例如:

```
1 make_unsigned_t<int> // unsigned int
2 make_unsigned_t<int const&> // undefined behavior (hopefully error)
```

有时结果可能会令人惊讶。例如:

```
1 add_rvalue_reference_t<int> // int&&
2 add_rvalue_reference_t<int const> // int const&&
3 add_rvalue_reference_t<int const&> // int const& (lvalue-ref remains lvalue-ref)
```

这里可能期望 `add_rvalue_reference` 总是进行右值引用,但是 C++ 的引用折叠规则(参见第 15.6.1 节)导致左值引用和右值引用的组合产生左值引用。

另一个例子:

```
1 is_copy_assignable_v<int> // yields true (generally, you can assign an int to an int)
2 is_assignable_v<int,int> // yields false (can't call 42 = 42)
```

`is_copy_assignable` 只是检查是否可以给另一个 `int` 赋值(检查 `lvalues` 的操作), `is_assignable` 将值类别(参见附录 B)考虑进去(这里检查是否可以给 `prvalue` 赋值)。也就是说,第一个表达式等价于

```
1 is_assignable_v<int&,int&> // yields true
```

出于同样的原因:

```
1 is_swappable_v<int> // yields true (assuming lvalues)
2 is_swappable_with_v<int&,int&> // yields true (equivalent to the previous check)
3 is_swappable_with_v<int,int> // yields false (taking value category into account)
```

请仔细注意类型特征的准确定义,附录 D 中详细描述了标准的情况。

11.2.2 std::addressof()

`std::addressof<>()` 函数模板生成对象或函数的实际地址。即使对象类型有重载操作符 `&`,也能工作。尽管后者很少使用,但可能会发生(例如,在智能指针中)。因此,如果需要任意类型对象的地址,建议使用 `addressof()`:

```
1 template<typename T>
2 void f (T&& x)
3 {
4     auto p = &x; // might fail with overloaded operator &
5     auto q = std::addressof(x); // works even with overloaded operator &
6     ...
7 }
```

11.2.3 std::declval()

`std::declval<>()` 函数模板可以用作特定类型的对象引用的占位符。该函数没有定义,因此不能调用(也不创建对象)。因此,只能用于未求值的操作数(`decltype` 和 `sizeof` 构造的操作数)。因此,与其尝试创建一个对象,可以假设有一个相应类型的对象。

例如，下面的声明从传递的模板参数 T1 和 T2 推导出默认返回类型 RT:

basics/maxdefaultdeclval.hpp

```
1 #include <utility>
2 template<typename T1, typename T2,
3           typename RT = std::decay_t<decltype(true ? std::declval<T1>()
4                                         : std::declval<T2>())>>
5 RT max (T1 a, T2 b)
6 {
7     return b < a ? a : b;
8 }
```

为了避免必须调用 T1 和 T2 的(默认)构造函数才能在表达式中调用三元操作符来初始化 RT，这里使用 std::declval 来“使用”对应类型的对象。不过，这可能在 decltype 未求值的上下文中实现。

不要忘记使用 std::decay<> 类型来确保默认的返回类型不是一个引用，因为 std::declval() 本身会产生右值引用。否则，像 max(1,2) 这样的调用将得到一个 int&& 的返回类型。详见 19.3.4 节。

11.3. 完美转发临时变量

如 6.1 节所示，可以使用转发引用和 std::forward<> 来“完美转发”泛型参数：

```
1 template<typename T>
2 void f (T&& t) // t is forwarding reference
3 {
4     g(std::forward<T>(t)); // perfectly forward passed argument t to g()
5 }
```

然而，有时必须完美地转发不通过参数的泛型代码中的数据。可以使用 auto&& 来创建一个可以转发的变量，假设对函数 get() 和 set() 进行了链接调用，其中 get() 的返回值应该完美地转发给 set()：

```
1 template<typename T>
2 void foo(T x)
3 {
4     set(get(x));
5 }
```

进一步假设需要更新代码，以便对 get() 产生的中间值执行一些操作。通过将值保存在使用 auto&& 声明的变量中来实现：

```
1 template<typename T>
2 void foo(T x)
3 {
4     auto&& val = get(x);
5     ...
6     // perfectly forward the return value of get() to set():
7     set(std::forward<decltype(val)>(val));
8 }
```

这样可以避免对中间值的复制。

11.4. 模板参数的引用

虽然不常见，但模板类型参数可以成为引用类型。例如：

basics/tmplparamref.cpp

```
1 #include <iostream>
2
3 template<typename T>
4 void tmplParamIsReference(T) {
5     std::cout << "T is reference: " << std::is_reference_v<T> << '\n' ;
6 }
7 int main()
8 {
9     std::cout << std::boolalpha;
10    int i;
11    int& r = i;
12    tmplParamIsReference(i); // false
13    tmplParamIsReference(r); // false
14    tmplParamIsReference<int&>(i); // true
15    tmplParamIsReference<int&>(r); // true
16 }
```

即使引用变量传递给 `tmplParamIsReference()`，模板参数 `T` 也会推导为所引用类型的类型（因为，对于引用变量 `v`，表达式 `v` 具有所引用的类型；表达式的类型绝不是引用）。但是，可以通过显式指定 `T` 的类型来强制使用引用：

```
1 tmplParamIsReference<int&>(r);
2 tmplParamIsReference<int&>(i);
```

这样做可以从根本上改变模板的行为，而且模板在设计时可能没有考虑到这种情况，从而触发错误或意外行为。考虑下面的例子：

basics/referror1.cpp

```
1 template<typename T, T Z = T{}>
2 class RefMem {
3 private:
4     T zero;
5 public:
6     RefMem() : zero{Z} {
7     }
8 };
9
10 int null = 0;
11
12 int main()
```

```

13 {
14     RefMem<int> rm1, rm2;
15     rm1 = rm2; // OK
16
17     RefMem<int&> rm3; // ERROR: invalid default value for N
18     RefMem<int&, 0> rm4; // ERROR: invalid default value for N
19
20     extern int null;
21     RefMem<int&,null> rm5, rm6;
22     rm5 = rm6; // ERROR: operator= is deleted due to reference member
23 }
```

这里，有一个具有模板参数类型 T 成员的类，用非类型模板参数 Z 初始化，Z 具有初始值的默认值为零。用 int 类型实例化类符合预期。然而，当试图用引用其实例化时，麻烦出现了：

- 默认初始化不再工作。
- 不能再传递 0 作为 int 的初始值。
- 因为非静态引用成员的类已经删除了默认的赋值操作符，赋值操作符不再可用。

此外，对非类型模板参数使用引用类型很麻烦，可能会很危险。考虑一下这个例子：

basics/referror2.cpp

```

1 #include <vector>
2 #include <iostream>
3
4 template<typename T, int& SZ> // Note: size is reference
5 class Arr {
6 private:
7     std::vector<T> elems;
8 public:
9     Arr() : elems(SZ) { // use current SZ as initial vector size
10    }
11    void print() const {
12        for (int i=0; i<SZ; ++i) { // loop over SZ elements
13            std::cout << elems[i] << ', ';
14        }
15    }
16 };
17
18 int size = 10;
19
20 int main()
21 {
22     Arr<int&,size> y; // compile-time ERROR deep in the code of class std::vector<>
23
24     Arr<int,size> x; // initializes internal vector with 10 elements
25     x.print(); // OK
26     size += 100; // OOPS: modifies SZ in Arr<>
27     x.print(); // run-time ERROR: invalid memory access: loops over 110 elements
```

这里，为引用类型的元素实例化 Arr 的尝试，会导致 std::vector<> 类的错误，因为它不能用引用作为元素实例化：

```
1 Arr<int&,size> y; // compile-time ERROR deep in the code of class std::vector<>
```

这个错误经常会形成在 9.4 节中看到的“错误小说”，编译器提供了从初始化原因到检测到错误实际模板定义的整个模板实例化过程。

更糟糕的可能是由于将 size 参数作为引用而导致的运行时错误：其允许记录的 size 值在容器不知道的情况下发生变化（也就是说，size 值可能会失效）。因此，使用 size 的操作（如 print() 成员）必然会出现未定义行为（导致程序崩溃，或更糟）：

```
1 int int size = 10;
2 ...
3 Arr<int,size> x; // initializes internal vector with 10 elements
4 size += 100; // OOPS: modifies SZ in Arr<>
5 x.print(); // run-time ERROR: invalid memory access: loops over 110 elements
```

将模板参数 SZ 更改为 int const& 类型并不能解决这个问题，因为 size 本身可以修改。

这个例子有些牵强。在更复杂的情况下，这样的问题确实会发生。另外，C++17 中可以推导出非类型参数。例如：

```
1 template<typename T, decltype(auto) SZ>
2 class Arr;
```

使用 decltype(auto) 可以很容易地产生引用类型，因此在上下文中最好不要使用（默认使用 auto）。详见 15.10.3 节。

出于这个原因，标准库有时会有奇怪的规范和约束。例如：

- 为了在模板参数为引用实例化时仍然具有赋值操作符，std::pair<> 和 std::tuple<> 类实现了赋值操作符，而不是使用默认行为。例如：

```
1 namespace std {
2     template<typename T1, typename T2>
3     struct pair {
4         T1 first;
5         T2 second;
6         ...
7         // default copy/move constructors are OK even with references:
8         pair(pair const&) = default;
9         pair(pair&) = default;
10        ...
11        // but assignment operator have to be defined to be available with references:
12        pair& operator=(pair const& p);
13        pair& operator=(pair& p) noexcept(...);
14        ...
15    };
16 }
```

- 由于副作用的复杂性，C++17 标准库类模板 `std::optional` 和 `std::variant` 的实例化是“病态”的(至少在 C++17 中是这样)。

要禁用引用，简单的静态断言就足够了：

```

1 template<typename T>
2 class optional
3 {
4     static_assert(!std::is_reference<T>::value,
5         "Invalid instantiation of optional<T> for references");
6     ...
7 };

```

引用类型通常与其他类型不同，并且受几种语言规则的约束。这会影响调用参数的声明(参见第 7 节)，以及定义类型特征的方式(参见第 19.6.1 节)。

11.5. 缓式评估

实现模板时，有时会出现这样的问题：代码是否能够处理不完整的类型(参见 10.3.1 节)。来看看下面的类模板：

```

1 template<typename T>
2 class Cont {
3 private:
4     T* elems;
5 public:
6     ...
7 };

```

这个类可以与不完整类型一起使用。例如：当类引用自己类型的元素时：

```

1 struct Node
2 {
3     std::string value;
4     Cont<Node> next; // only possible if Cont accepts incomplete types
5 };

```

然而，仅通过使用一些特性，就会失去处理不完整类型的能力。例如：

```

1 template<typename T>
2 class Cont {
3 private:
4     T* elems;
5 public:
6     ...
7     typename std::conditional<std::is_move_constructible<T>::value,
8         T&&,
9         T&
10        >::type
11     foo();
12 };

```

这里，使用特征 `std::conditional`(参见 D.5 节)来决定成员函数 `foo()` 的返回类型是 `T&&` 还是 `T&`。这取决于模板参数类型 `T` 是否支持移动语义。

问题是特性 `std::is_move_constructible` 要求参数是一个完整的类型(不是 `void` 或未知边界的数组;参见的 D.3.2 节)。在 `foo()` 的这个声明中，`struct Node` 的声明失败了。

如果 `std::is_move_constructible` 是一个完整的类型，并不是所有的编译器都会产生错误。因为对于这种错误，不需要进行诊断。所以，在需要平台移植时需要考虑这个问题。

可以将 `foo()` 替换为成员模板来解决这个问题，这样 `std::is_move_constructible` 的计算就会延迟到 `foo()` 的实例化点：

```
1 template<typename T>
2 class Cont {
3 private:
4     T* elems;
5 public:
6     template<typename D = T> std::conditional<std::is_move_constructible<T>::value,
7             T&&,
8             T&
9             >::type
10    foo();
11 }
```

现在，特性依赖于模板参数 `D`(默认为 `T`，我们想要的值)，编译器必须等到 `foo()` 调，如 `Node` 之前，再评估特性(那时 `Node` 是一个完整的类型，只是在定义时不完整)。

11.6. 编写泛型库

让我们列出一些在实现泛型库时需要记住的事情(注意，其中一些可能会在后面会介绍到)：

- 模板中使用转发引用来转发值(参见第 91 页 6.1 节)。如果值不依赖于模板参数，使用 `auto&&`(参见 11.3 节)。
- 当参数声明为转发引用时，模板参数在传递左值时要有引用类型(参见 15.6.2 节)。
- 当需要依赖于模板形参的对象地址时，使用 `std::addressof()`，以避免当对象绑定到带有重载操作符 `&` 的类型时出现意外(11.2.2 节)
- 对于成员函数模板，确保不会比预定义的复制/移动构造函数或赋值操作符更好地匹配(6.4 节)。
- 模板参数可能是字符串字面值，且不通过值传递时(7.4 节和 D.4 节)，请考虑使用 `std::decay`。
- 如果模板参数有 `out` 或 `inout`，请准备好处理参数可能指定为 `const` 类型的情况(参见 7.2.2 节)。
- 准备好处理模板参数引用的副作用(参见 11.4 节了解详细信息，19.6.1 节为示例)。特别是，要确保返回类型不能是引用(参见 7.5 节)。
- 准备好处理不完全类型，从而进行以支持，例如：递归数据结构(参见 11.5 节)。
- 重载所有数组类型，而不仅仅是 `T[SZ]`(参见 5.4 节)。

11.7. 总结

- 模板允许将函数、函数指针、函数对象、函子和 Lambda 作为可调用对象传递。
- 使用重载 operator() 定义类时，将其声明为 const(除非调用改变了状态)。
- 使用 std::invoke()，可以处理所有可调用对象的代码，包括成员函数。
- 使用 decltype(auto) 来完美地转发返回值。
- 类型特征是检查类型属性和功能性函数。
- 当需要模板中对象的地址时，可以使用 std::addressof()。
- 使用 std::declval() 在未计算的表达式中创建特定类型的值。
- 如果对象的类型不依赖于模板参数，可以使用 auto&& 在泛型代码中完美转发。
- 准备好处理模板参数作为引用的副作用。
- 可以使用模板来推迟对表达式的求值(例如，支持在类模板中使用不完整类型)。

第二部分：深入了解模板

本书的第一部分提供了 C++ 模板底层的大多数语言概念，这些表述足以回答日常 C++ 编程中可能出现的大多数问题。本书的第二部分提供了参考，回答了将语应用于高级软件时出现的问题。可以在第一次阅读时跳过这一部分，然后根据后面章节中的引用或在索引中查找某个概念后返回到特定的章节。

我们的目标是清晰且完整，但也保持讨论的简洁。所以例子很简短，而且常常比较有代表性。这也确保了我们的讨论不会偏离主题，从而避免涉及到不相关的话题。

此外，我们还将探讨 C++ 中模板语言特性未来可能的更改和扩展。

这部分的主题包括：

- 基础的模板声明问题
- 模板中名称的含义
- C++ 模板实例化机制
- 模板参数的推导规则
- 特化和重载
- 未来的可能性

第 12 章 模板基础

这章中，将深入了解本书第一部分中介绍的基础知识：模板声明、模板形参和实参的限制等。

12.1. 参数化的声明

C++ 目前支持四种基本模板：类模板、函数模板、变量模板和别名模板。这些模板类型都可以出现在名称空间中，也可以出现在类中。在类作用域中，作为嵌套的类模板、成员函数模板、静态数据成员模板和成员别名模板。这些模板的声明与普通的类、函数、变量和类型别名（或类成员）相似，不过需要参数化

```
1 template<parameters here>
```

C++17 引入了通过参数化子句引入的构造：推导策略（请参阅 2.9 节和 15.12.1 节）。本书中，它们不叫做模板（例如，没有实例化），但是选择这个语法是为了让人联想到函数模板。

将在后面的章节中回到模板参数声明。首先，使用一些示例展示这四种模板，其可以出现在命名空间中（全局或在一个命名空间中）：

details/definitions1.hpp

```
1 template<typename T> // a namespace scope class template
2 class Data {
3     public:
4         static constexpr bool copyable = true;
5     ...
6 };
7
8 template<typename T> // a namespace scope function template
9 void log (T x) {
10     ...
11 }
12
13 template<typename T> // a namespace scope variable template (since C++14)
14 T zero = 0;
15
16 template<typename T> // a namespace scope variable template (since C++14)
17 bool dataCopyable = Data<T>::copyable;
18
19 template<typename T> // a namespace scope alias template
20 using DataList = Data<T*>;
```

本例中，尽管是通过类模板 Data 的参数化间接参数化的，但静态数据成员 Data<T>::copyable 不是变量模板。变量模板可以出现在类中，所以其是一个静态数据成员模板。

下面的例子展示了，四种模板作为成员，在如何在父类中进行定义：

details/definitions2.hpp

```

1 class Collection {
2     public:
3         template<typename T> // an in-class member class template definition
4             class Node {
5                 ...
6             };
7
8         template<typename T> // an in-class (and therefore implicitly inline)
9             T* alloc() { // member function template definition
10                ...
11            }
12
13        template<typename T> // a member variable template (since C++14)
14            static T zero = 0;
15
16        template<typename T> // a member alias template
17            using NodePtr = Node<T>*;
18    };

```

C++17 中，变量(包括静态数据成员)和变量模板可以“内联”，定义可以跨翻译单元进行重复。这对于变量模板来说是多余的，因为变量模板可以定义在多个翻译单元中。但与成员函数不同，在封闭类中定义的静态数据成员不会使其内联：必须使用关键字 `inline` 指定。

最后，下面的代码演示了如何在类外定义非别名模板的成员模板：

details/definitions3.hpp

```

1 template<typename T> // a namespace scope class template
2 class List {
3     public:
4         List() = default; // because a template constructor is defined
5
6         template<typename U> // another member class template,
7             class Handle; // without its definition
8
9         template<typename U> // a member function template
10        List (List<U> const&); // (constructor)
11
12        template<typename U> // a member variable template (since C++14)
13        static U zero;
14    };
15
16 template<typename T> // out-of-class member class template definition
17     template<typename U>
18     class List<T>::Handle {
19     ...
20    };
21
22 template<typename T> // out-of-class member function template definition

```

```

23 template<typename U>
24 List<T>::List (List<U> const& b)
25 {
26     ...
27 }
28
29 template<typename T> // out-of-class static data member template definition
30     template<typename U>
31 U List<T>::zero = 0;

```

定义在类外的成员模板需要多个 `template<...>` 参数化子句：每个外围作用域的类模板一个，成员模板本身也需要一个，子句从类模板最外层开始逐行展示。

构造函数模板（一种特殊的成员函数模板）禁用默认构造函数的隐式声明（因为只有在没有声明其他构造函数的情况下，才会隐式声明）。添加默认声明

```
1 List() = default;
```

确保 `List<T>` 的实例是默认可构造的，具有隐式声明构造函数的语义。

联合模板

联合模板也可用（一种类模板）：

```

1 template<typename T>
2 union AllocChunk {
3     T object;
4     unsigned char bytes[sizeof(T)];
5 };

```

默认参数

函数模板可以像普通函数声明一样有默认的参数：

```

1 template<typename T>
2 void report_top (Stack<T> const&, int number = 10);
3
4 template<typename T>
5 void fill (Array<T>&, T const& = T{}); // T{} is zero for built-in types

```

后面的声明表明默认参数可以依赖于模板参数，也可以定义为（C++11 前唯一的方法，参见 5.2 节）

```

1 template<typename T>
2 void fill (Array<T>&, T const& = T()); // T() is zero for built-in types

```

当使用 `fill()` 时，若提供了第二个函数调用参数，则不会实例化默认参数。这确保了默认参数不能为特定的 `T` 进行实例化，从而不会出现错误。例如：

```

1 class Value {
2 public:
3     explicit Value(int); // no default constructor

```

```

4 } ;
5
6 void init (Array<Value>& array)
7 {
8     Value zero(0);
9     fill(array, zero); // OK: default constructor not used
10    fill(array); // ERROR: undefined default constructor for Value is used
11 }

```

类模板的非模板成员

除了在类内部声明的四种基本模板之外，还可以通过成为类模板的一部分来将普通类成员参数化，偶尔（错误地）也称为成员模板。尽管可以进行参数化，但这样的定义并不是模板，其参数完全由所属的模板决定。例如：

```

1 template<int I>
2 class CupBoard
3 {
4     class Shelf; // ordinary class in class template
5     void open(); // ordinary function in class template
6     enum Wood : unsigned char; // ordinary enumeration type in class template
7     static double totalWeight; // ordinary static data member in class template
8 };

```

因为不是模板，所以对应的定义只为父类模板指定参数化子句（参数化子句与出现在最后一个::之后的名称相关联）：

```

1 template<int I> // definition of ordinary class in class template
2 class CupBoard<I>::Shelf {
3     ...
4 };
5
6 template<int I> // definition of ordinary function in class template
7 void CupBoard<I>::open ()
8 {
9     ...
10 }
11
12 template<int I> // definition of ordinary enumeration type class in class template
13 enum CupBoard<I>::Wood {
14     Maple, Cherry, Oak
15 };
16
17 template<int I> // definition of ordinary static member in class template
18 double CupBoard<I>::totalWeight = 0.0;

```

C++17 后，静态的 totalWeight 成员可以在类模板中使用 inline 来初始化：

```

1 template<int I>
2 class CupBoard

```

```
3 ...
4     inline static double totalWeight = 0.0;
5 }
```

尽管这种参数化的定义通常称为模板，但这个术语并不适用于。对这些实体的一个术语是 **temploid**。C++17 起，标准确实定义了模板化实体的概念，包括模板和 **temploid**，以及递归地在模板化实体中定义或创建的实体(例如，包括在类模板中定义的友元函数(参见 2.4 节)或模板中出现的 Lambda 表达式的闭包类型)。目前，**temploid** 和模板实体都没有获得太多的关注，但将来可以使用这些使用这些术语，在讨论与 C++ 模板的话题时进行精确的表达。

12.1.1 虚拟成员函数

成员函数模板不能进行虚声明。这个约束是因为虚函数调用机制的实现，是使用一个固定大小的表，每个虚函数只有一个条目。但成员函数模板的实例化，直到整个程序翻译后才固定。因此，支持虚成员函数模板需要在 C++ 编译器和链接器中支持一套全新的机制。

因为在实例化类时其数量固定，所以类模板的普通成员可以为虚函数：

```
1 template<typename T>
2 class Dynamic {
3     public:
4         virtual ~Dynamic(); // OK: one destructor per instance of Dynamic<T>
5
6         template<typename T2>
7         virtual void copy (T2 const&); // ERROR: unknown number of instances of copy()
8             // given an instance of Dynamic<T>
9
10    };
```

12.1.2 链接模板

每个模板都必须有一个名字，并且这个名字在其作用域内必须唯一(除了函数模板可以重载(参见第 16 章))。与类类型不同，类模板不能与其他类型的实体共享名称：

```
1 int C;
2 ...
3 class C; // OK: class names and nonclass names are in a different "space"
4
5 int X;
6 ...
7 template<typename T>
8 class X; // ERROR: conflict with variable X
9 struct S;
10 ...
11 template<typename T>
12 class S; // ERROR: conflict with struct S
```

模板名称有链接，但不能有 C 链接存在。非标准名称链接的含义可能与实现有关(然而，目前还不清楚有哪个实现支持模板的非标准名称链接)：

```

1 extern "C++" template<typename T>
2 void normal(); // this is the default: the linkage specification could be left out
3
4 extern "C" template<typename T>
5 void invalid(); // ERROR: templates cannot have C linkage
6
7 extern "Java" template<typename T>
8 void javaLink(); // nonstandard, but maybe some compiler will someday
      // support linkage compatible with Java generics

```

模板通常具有外部链接。唯一的例外是具有静态说明符的命名空间作用域函数模板、未命名命名空间的直接或间接成员模板(具有内部链接)，以及未命名类的成员模板(没有链接)。例如：

```

1 template<typename T> // refers to the same entity as a declaration of the
2 void external(); // same name (and scope) in another file
3
4 template<typename T> // unrelated to a template with the same name in
5 static void internal(); // another file
6
7 template<typename T> // redeclaration of the previous declaration
8 static void internal();
9
10 namespace {
11     template<typename> // also unrelated to a template with the same name
12     void otherInternal(); // in another file, even one that similarly appears
13 } // in an unnamed namespace
14
15 namespace {
16     template<typename> // redeclaration of the previous template declaration
17     void otherInternal();
18 }
19
20 struct {
21     template<typename T> void f(T) {} // no linkage: cannot be redeclared
22 } x;

```

注意，由于后一个成员模板没有链接，因为没有办法在类之外提供定义，所以必须在未命名的类中定义。

目前模板不能在函数作用域或局部类作用域内声明，但是泛型 Lambda(请参阅 15.10.6 节)可以出现在局部作用域中，具有包含成员函数模板的相关闭包，这是一种局部成员函数模板。

模板实例链接就是模板链接。从上面声明的模板内部实例化的函数 `internal<void>()` 将进行内部链接。对于变量模板，有一个有趣的结果，看看下面的例子：

```

1 template<typename T> T zero = T{};


```

所有 0 的实例化都有外部链接，甚至像 `zero<int const>` 这样的东西都有。这可能是反直觉的

```

1 int const zero_int = int{};


```

具有内部链接，因为使用 `const` 类型声明。模板的所有实例化为

```
1 template<typename T> int const max_volume = 11;
```

有外部链接，所有这些实例化也有 int const 类型。

12.1.3 主模板

模板的普通声明为主模板，就不需要模板名称后加上尖括号中的模板参数：

```
1 template<typename T> class Box; // OK: primary template
2 template<typename T> class Box<T>; // ERROR: does not specialize
3
4 template<typename T> void translate(T); // OK: primary template
5 template<typename T> void translate<T>(T); // ERROR: not allowed for functions
6
7 template<typename T> constexpr T zero = T{}; // OK: primary template
8 template<typename T> constexpr T zero<T> = T{}; // ERROR: does not specialize
```

非主模板是在声明类或变量模板的偏特化时发生，这些将在第 16 章中讨论。并且函数模板必须是主模板（参见 17.3 节的讨论）。

12.2. 模板形参

模板参数有三种基本类型：

1. 类型参数(最常见的)
2. 非类型参数
3. 双重模板参数

这些基本类型的模板参数都可以作为模板参数包的基础（参见 12.2.4 节）。

模板参数在模板声明的介绍性参数化子句中进行声明。

C++14 以来的例外是泛型 Lambda 的隐式模板类型参数，参见 15.10.6 节。

这样的声明不一定需要命名：

```
1 template<typename, int>
2 class X; // X<> is parameterized by a type and an integer
```

若参数在模板中引用，则需要参数名。模板参数名称可以在后续的参数声明中引用（但不能在前面引用）：

```
1 template<typename T, // the first parameter is used
2      T Root, // in the declaration of the second one and
3      template<T> class Buf> // in the declaration of the third one
4 class Structure;
```

12.2.1 类型参数

类型参数是通过关键字 `typename` 或关键字 `class` 引入的: 两者等价。

关键字 `class` 并不意味着替换参数应该是类类型, 还可以是一个可访问的类型。

关键字后面必须跟着一个简单的标识符, 该标识符后面必须跟着一个逗号, 表示下一个参数声明的开始, 一个结束尖括号 (`>`) 表示参数化子句的结束, 或者一个等号 (`=`) 表示默认模板参数的开始。

模板声明中, 类型形参的作用类似于类型别名 (参见 2.8 节)。当 `T` 是模板参数时, 不能使用参数 `T` 的名称:

```
1 template<typename Allocator>
2 class List {
3     class Allocator* allocptr; // ERROR: use "Allocator* allocptr"
4     friend class Allocator; // ERROR: use "friend Allocator"
5     ...
6 };
```

12.2.2 非类型参数

非类型模板参数表示可在编译或链接时确定的常量。

模板参数也不表示类型, 但不同于非类型参数。这也是个历史问题: 双重模板参数在类型参数和非类型参数之后添加。

这样的参数类型 (换句话说, 代表的类型) 必须是下列之一:

- 整数类型或枚举类型
- 指针类型

撰写本文时, 只允许“指向对象的指针”和“指向函数的指针”类型, 这就排除了像 `void*` 这样的类型。然而, 所有的编译器似乎也能接受 `void*`。

- 成员指针类型
- 左值引用类型 (对对象的引用和对函数的引用都可以)
- `std::nullptr_t`
- 包含 `auto` 或 `decltype(auto)` 的类型 (仅 C++17 中可用, 参见 15.10.1 节)

所有其他类型目前都排除在外 (尽管浮点类型可能会在将来添加, 参见 17.2 节)。

非类型模板参数的声明, 某些情况下也使用关键字 `typename` 开头:

```
1 template<typename T, // a type parameter
2         typename T::Allocator* Allocator> // a nontype parameter
3 class List;
```

或者使用关键字 class:

```
1 template<class X*> // a nontype parameter of pointer type
2 class Y;
```

这两种情况很容易区分，因为第一个后面跟着一个简单的标识符，然后是一段标记（‘=’表示默认实参，‘,’表示后面跟着另一个模板参数，或者用>结束模板参数列表）。5.1 节和的 13.3.2 节解释了，在第一个非类型参数中使用 typename 关键字的必要性。

函数和数组类型可以指定，但会隐式衰变为指针类型：

```
1 template<int buf[5]> class Lexer; // buf is really an int*
2 template<int* buf> class Lexer; // OK: this is a redeclaration
3 template<int fun()> struct FuncWrap; // fun really has pointer to
4 // function type
5 template<int (*) ()> struct FuncWrap; // OK: this is a redeclaration
```

非类型模板参数的声明很像变量，但它们不能有静态、可变等非类型修饰符。它们可以有 const 和 volatile 限定符，但若这样的限定符出现在参数类型的最外层，就会忽略：

```
1 template<int const length> class Buffer; // const is useless here
2 template<int length> class Buffer; // same as previous declaration
```

最后，非引用非类型参数在表达式中使用时，始终是 prvalue。

参见附录 B 中关于值类别的讨论，比如右值和左值。

地址不能取走，也不能赋予。另一方面，左值引用类型的非类型参数可用于表示左值：

```
1 template<int& Counter>
2 struct LocalIncrement {
3     LocalIncrement() { Counter = Counter + 1; } // OK: reference to an integer
4     ~LocalIncrement() { Counter = Counter - 1; }
5 };
```

并且，不允许右值引用。

12.2.3 双重模板参数

双重参数是类模板或别名模板的占位符。声明很像类模板，但关键字 struct 和 union 不能使用：

```
1 template<template<typename X> class C> // OK
2 void f(C<int>* p);
3
4 template<template<typename X> struct C> // ERROR: struct not valid here
5 void f(C<int>* p);
6
7 template<template<typename X> union C> // ERROR: union not valid here
8 void f(C<int>* p);
```

C++17 允许使用 typename，而非 class。因为双重模板参数不仅可以使用类模板替换，还可以使用别名模板（别名模板实例化为任意类型）替换。C++17 中，上面的例子也可以写成

```
1 template<template<typename X> typename C> // OK since C++17
2 void f(C<int>* p);
```

双重模板参数声明的作用域内，使用方式和其他类或别名模板一样。

模板参数可以有默认值。这些默认值适用于没有指定相应参数的情况：

```
1 template<template<typename T,
2           typename A = MyAllocator> class Container>
3 class Adaptation {
4     Container<int> storage; // implicitly equivalent to Container<int,MyAllocator>
5     ...
6 };
```

T 和 A 是模板参数 Container 的模板参数名，只能在该模板参数的其他参数声明中使用：

```
1 template<template<typename T, T*> class Buf> // OK
2 class Lexer {
3     static T* storage; // ERROR: a template template parameter cannot be used here
4     ...
5 };
```

然而，通常模板参数名在声明其他模板参数时是不需要的，因此可以不命名。例如，之前的 Adaptation 模板的声明可以这样写：

```
1 template<template<typename,
2           typename = MyAllocator> class Container>
3 class Adaptation {
4     Container<int> storage; // implicitly equivalent to Container<int,MyAllocator>
5     ...
6 };
```

12.2.4 模板参数包

C++11 后，任何类型的模板参数都可以通过在模板参数名之前引入省略号 (...) 转换为模板参数包。若板参数未命名，需要对模板参数进行命名：

```
1 template<typename... Types> // declares a template parameter pack named Types
2 class Tuple;
```

模板参数包的行为与其底层模板参数相似，但有一个关键区别：普通模板参数只能匹配一个模板参数，而模板参数包可以匹配任意数量的模板参数。所以，上面声明的 Tuple 类模板接受任意数量（可能是不同的）的类型作为模板参数：

```
1 using IntTuple = Tuple<int>; // OK: one template argument
2 using IntCharTuple = Tuple<int, char>; // OK: two template arguments
3 using IntTriple = Tuple<int, int, int>; // OK: three template arguments
4 using EmptyTuple = Tuple<>; // OK: zero template arguments
```

类似地，非类型参数和模板参数包可以接受任意数量的非类型参数和模板参数：

```

1 template<typename T, unsigned... Dimensions>
2 class MultiArray; // OK: declares a nontype template parameter pack
3
4 using TransformMatrix = MultiArray<double, 3, 3>; // OK: 3x3 matrix
5
6 template<typename T, template<typename, typename>... Containers>
7 void testContainers(); // OK: declares a template template parameter pack

```

MultiArray 要求所有非类型模板参数具有无符号类型。C++17 引入了推导非类型模板参数的可能，在一定程度上可以绕过这个限制——请参阅 15.10.1 节了解详细信息。

主模板、变量模板和别名模板最多可以有一个模板参数包。若模板参数包是最后一个模板参数，则函数模板有一个较弱的限制：允许多个模板参数包，只要模板参数包后面的每个模板参数要么有一个默认值（参见下一节），要么可以推导（参见第 15 章）：

```

1 template<typename... Types, typename Last>
2 class LastType; // ERROR: template parameter pack is not the last template parameter
3
4 template<typename... TestTypes, typename T>
5 void runTests(T value); // OK: template parameter pack is followed
6 // by a deducible template parameter
7
8 template<unsigned...> struct Tensor;
9 template<unsigned... Dims1, unsigned... Dims2>
10 auto compose(Tensor<Dims1...>, Tensor<Dims2...>);
11 // OK: the tensor dimensions can be deduced

```

最后一个例子使用了返回类型推导——C++14 的特性。请参见第 15.10.1 节。

类和变量模板的偏特化声明（参见第 16 章）可以有多个参数包。与主模板对应的参数包不同，这是因为偏特化是通过推导选择的，该推导过程与函数模板的推导过程相同。

```

1 template<typename...> Typelist;
2 template<typename X, typename Y> struct Zip;
3 template<typename... Xs, typename... Ys>
4   struct Zip<Typelist<Xs...>, Typelist<Ys...>>;
5   // OK: partial specialization uses deduction to determine
6   // the Xs and Ys substitutions

```

类型参数包不能在自己的参数子句中展开。例如：

```

1 template<typename... Ts, Ts... vals> struct StaticValues {};
2 // ERROR: Ts cannot be expanded in its own parameter list

```

然而，嵌套模板可以创建类似且有效的代码：

```

1 template<typename... Ts> struct ArgList {
2   template<Ts... vals> struct Vals {};
3 };
4 ArgList<int, char, char>::Vals<3, 'x', 'y'> tada;

```

因为能接受可变数量的模板参数，所以包含模板参数包的模板称为可变参数模板。第 4 章和第 12.4 节在描述了可变参数模板的使用。

12.2.5 默认模板参数

非模板参数包中的模板参数都可以配备一个默认参数，必须在类型上与相应的参数匹配(例如，类型参数不能有非类型的默认参数)。默认参数不能依赖于自己的参数，因为参数名直到默认参数之后才在作用域中。但是，其可能取决于之前的参数：

```
1 template<typename T, typename Allocator = allocator<T>>
2 class List;
```

只有在后续参数也提供了默认参数的情况下，类模板、变量模板或别名模板的模板参数才可以有默认模板参数(对于默认函数调用参数也存在类似的约束)。随后的默认值通常在同一个模板声明中提供，但可以在该模板的声明中进行声明：

```
1 template<typename T1, typename T2, typename T3,
2 typename T4 = char, typename T5 = char>
3 class Quintuple; // OK
4
5 template<typename T1, typename T2, typename T3 = char,
6 typename T4, typename T5>
7 class Quintuple; // OK: T4 and T5 already have defaults
8
9 template<typename T1 = char, typename T2, typename T3,
10 typename T4, typename T5>
11 class Quintuple; // ERROR: T1 cannot have a default argument
12 // because T2 doesn't have a default
```

函数模板的参数，默认模板参数不需要后续模板参数有默认值：

后续模板参数的模板参数，可以通过模板参数推导来确定，参见第 15 章。

```
1 template<typename R = void, typename T>
2 R* addressof(T& value); // OK: if not explicitly specified, R will be void
```

默认模板参数不能重复：

```
1 template<typename T = void>
2 class Value;
3
4 template<typename T = void>
5 class Value; // ERROR: repeated default argument
```

许多上下文不允许默认模板参数：

- 偏特化：

```
1 template<typename T>
2 class C;
3 ...
4 template<typename T = int>
5 class C<T*>; // ERROR
```

- 参数包:

```
1 template<typename... Ts = int> struct X; // ERROR
```

- 类模板成员类外定义:

```
1 template<typename T> struct X
2 {
3     T f();
4 }
5
6 template<typename T = int> T X<T>::f() { // ERROR
7     ...
8 }
```

- 友元类模板声明:

```
1 struct S {
2     template<typename = void> friend struct F;
3 }
```

- 友元函数模板声明, 除非是一个定义, 并且在编译单元的其他地方没有声明:

```
1 struct S {
2     template<typename = void> friend void f(); // ERROR: not a definition
3     template<typename = void> friend void g() { // OK so far
4     }
5 };
6 template<typename> void g(); // ERROR: g() was given a default template argument
7 // when defined; no other declaration may exist here
```

12.3. 模板实参

实例化模板时, 模板应用模板参数。参数可以使用几种不同的机制来确定:

- 显式模板参数: 模板名称后面可以跟着用尖括号括起来的显式模板参数。产生的名称称为 template-id。
- 注入类名: 类模板 X 的作用域内, 模板参数 P1, P2, …, 该模板的名称 (X) 等价于 template-id X<P1, P2, …>。详见 13.2.3 节。
- 默认模板参数: 若默认模板参数可用, 则可以从模板实例中省略模板参数。但是, 对于类模板或别名模板, 即使所有模板参数都有默认值, 也必须提供 (可能为空) 尖括号。
- 参数类型推导: 未显式指定的函数模板参数, 可以从调用中的函数调用参数类型推导出来 (这在第 15 章中有详细描述), 其他几种情况下也可以进行推导。若所有模板参数都可以推导, 则不需要在函数模板的名称后指定尖括号。C++17 还引入了从变量声明或函数表示法, 可以从类型转换初始化表达式中推导类模板参数类型; 请参阅 15.12 节。

12.3.1 函数模板参数

函数模板参数可以显式指定，可以根据模板的使用方式推导出来，也可以作为默认模板参数提供。例如：

details/max.cpp

```
1 template<typename T>
2 T max (T a, T b)
3 {
4     return b < a ? a : b;
5 }
6
7 int main()
8 {
9     ::max<double>(1.0, -3.0); // explicitly specify template argument
10    ::max(1.0, -3.0); // template argument is implicitly deduced to be double
11    ::max<int>(1.0, 3.0); // the explicit <int> inhibits the deduction;
12    // hence the result has type int
13 }
```

因为对应的模板参数没有出现在函数参数类型中，或者出于其他原因（参见第 15.2 节），有些模板参数永远无法推导。相应的参数通常放在模板参数列表的开头，以便在推导其他参数的同时显式地指定。例如：

details/implicit.cpp

```
1 template<typename DstT, typename SrcT>
2 DstT implicit_cast (SrcT const& x) // SrcT can be deduced, but DstT cannot
3 {
4     return x;
5 }
6
7 int main()
8 {
9     double value = implicit_cast<double>(-1);
10 }
```

若在本例中颠倒了模板参数的顺序（写成了 `template<typename SrcT, typename DstT>`），则 `implicit_cast` 的调用必须显式指定两个模板参数。

此外，因为没有办法显式地指定或推导，这样的参数不能放在模板参数包之后或出现在偏特化中。

```
1 template<typename ... Ts, int N>
2 void f(double (&) [N+1], Ts ... ps); // useless declaration because N
3 // cannot be specified or deduced
```

因为函数模板可以重载，显式地提供函数模板的参数可能不足以标识单个函数：某些情况下，可以标识一组函数：

```

1 template<typename Func, typename T>
2 void apply (Func funcPtr, T x)
3 {
4     funcPtr(x);
5 }
6
7 template<typename T> void single(T);
8
9 template<typename T> void multi(T);
10 template<typename T> void multi(T*);
11
12 int main()
13 {
14     apply(&single<int>, 3); // OK
15     apply(&multi<int>, 7); // ERROR: no single multi<int>
16 }
```

本例中，因为表达式 `&single<int>` 的类型明确，对 `apply()` 的第一次调用有效。因此，`Func` 的模板参数值很容易推导。然而，在第二个调用中，`&multi<int>` 可能是两种不同类型中的一种，这种情况下不能推导 `Func` 的模板参数类型。

此外，在函数模板中替换模板参数，可能会导致构造无效的 C++ 类型或表达式。考虑以下重载函数模板 (RT1 和 RT2 是未指定类型):

```

1 template<typename T> RT1 test(typename T::X const*) ;
2 template<typename T> RT2 test(...);
```

表达式 `test<int>` 对于两个函数模板中的第一个没有意义，因为 `int` 类型没有成员类型 `x`。然而，第二个模板没有这样的问题。因此，表达式 `&test<int>` 为单个函数的地址。将 `int` 替换到第一个模板失败的事实并不意味着表达式无效。SFNAE(替换失败不为过) 原则是实现函数模板重载的重要因素，在 8.4 节和 15.7 节中进行了讨论。

12.3.2 类型参数

模板类型实参为模板类型形参指定的“值”，任何类型 (包括 `void`、函数类型、引用类型等) 都可以用作模板实参，但对模板形参的替换必须进行有效的构造：

```

1 template<typename T>
2 void clear (T p)
3 {
4     *p = 0; // requires that the unary * be applicable to T
5 }
6
7 int main()
8 {
9     int a;
10    clear(a); // ERROR: int doesn't support the unary *
11 }
```

12.3.3 非类型模板参数

非类型模板实参是替代非类型形参的值。这样的值必须是以下条件之一：

- 具有正确类型的另一个非类型模板的参数。
- 整型(或枚举)的编译时常量值。只有当相应参数的类型与值的类型匹配，或者值可以隐式转换(不收缩)时，这才可以接受。可以为 int 形参提供一个 char 值，但是对于 8 位 char 形参，500 是一个无效的输入。
- 外部变量或函数的名称前面加上内置的一元 &("address of") 操作符。对于函数和数组变量，可以省略 &。这样的模板实参匹配指针类型的非类型形参。C++17 放宽了这一要求，允许产生指向函数或变量指针的常量表达式。
- 对于引用类型的非类型形参来说，前面不带 & 操作符的实参是有效的实参。这里，C++17 也放宽了限制，允许函数或变量使用任意常量表达式 glvalue。
- 成员指针常数，形式为 &C::m 的表达式，其中 C 是类类型，m 是非静态成员(数据或函数)，这将匹配指针成员类型的非类型形参。C++17 中，实际语法形式不再受到限制：允许将常量表达式求值匹配为相应的成员常量指针。
- 空指针常量，是指针或指针成员类型的非类型形参的有效实参。

对于整型非类型形参(可能是最常见的非类型形参类型)，将考虑到形参类型的隐式转换。随着 C++11 中 constexpr 的引入，转换前的实参可以具有类类型。

C++17 之前，当将实参与指针或引用形参匹配时，用户定义的转换(一个实参的构造函数和转换操作符)和派生是不考虑到转换为基类的。使参数更具有 const 和/或 volatile 的隐式转换是可以的。

下面是一些非类型模板参数的示例：

```
1 template<typename T, T nontypeParam>
2 class C;
3
4 C<int, 33>* c1; // integer type
5
6 int a;
7 C<int*, &a>* c2; // address of an external variable
8
9 void f();
10 void f(int);
11 C<void (*)(int), f>* c3; // name of a function: overload resolution selects
12 // f(int) in this case; the & is implied
13
14 template<typename T> void templ_func();
15 C<void(), &templ_func<double>>* c4; // function template instantiations are functions
16
17 struct X {
18     static bool b;
19     int n;
20     constexpr operator int() const { return 42; }
21 };
22
```

```

23 C<bool&, X::b>* c5; // static class members are acceptable variable/function names
24
25 C<int X::*>* c6; // an example of a pointer-to-member constant
26
27 C<long, X{}>* c7; // OK: X is first converted to int via a constexpr conversion
28 // function and then to long via a standard integer conversion

```

模板参数需要在编译器或链接器在构建程序时，能够表示它们的值。在程序运行前不知道的值（例如，局部变量的地址）与构建时实例化模板的概念不兼容。

即便如此，有些常量值目前还是无效的：

- 浮点数
- 字符串字面值

(C++11 之前，空指针常量也不允许)

字符串字面值的问题是，两个相同的字面值可以存储在两个不同的地址。常量字符串上实例化模板的另一种（很麻烦）方法，引入一个额外变量来保存字符串：

```

1 template<char const* str>
2 class Message {
3     ...
4 };
5
6 extern char const hello[] = "Hello World!";
7 char const hello11[] = "Hello World!";
8
9 void foo()
10 {
11     static char const hello17[] = "Hello World!";
12
13     Message<hello> msg03; // OK in all versions
14     Message<hello11> msg11; // OK since C++11
15     Message<hello17> msg17; // OK since C++17
16 }

```

要求是声明为引用或指针的非类型模板形参，可以是一个常量表达式，在旧 C++ 版本中具有外部链接，C++11 开始的内部链接，或者 C++17 开始的任意链接方式。

请参阅第 17.2 节，了解关于该领域未来可能的变化。

以下是其他一些无效的例子：

```

1 template<typename T, T nonTypeParam>
2 class C;
3
4 struct Base {
5     int i;
6 } base;
7
8 struct Derived : public Base {
9 } derived;
10

```

```

11 C<Base*, &derived>* err1; // ERROR: derived-to-base conversions are not considered
12
13 C<int&, base.i>* err2; // ERROR: fields of variables aren't considered to be variables
14
15 int a[10];
16 C<int*, &a[0]>* err3; // ERROR: addresses of array elements aren't acceptable either

```

12.3.4 双重模板参数

双重模板实参通常必须是类模板或别名模板，其形参必须与它所替换的双重模板参数的形参精确匹配。C++17 前，双重模板参数的默认模板实参会忽略(但如果双重模板参数有默认实参，会在模板实例化期间考虑)。C++17 放宽了匹配规则，只要求模板形参至少与对应的模板形参一样特化(参见 16.2.2 节)。

这使得以下示例在 C++17 前无效：

```

1 #include <list>
2 // declares in namespace std:
3 // template<typename T, typename Allocator = allocator<T>>
4 // class list;
5
6 template<typename T1, typename T2,
7          template<typename> class Cont> // Cont expects one parameter
8 class Rel {
9 ...
10};
11
12 Rel<int, double, std::list> rel; // ERROR before C++17: std::list has more than
13                                // one template parameter

```

本例中的问题是标准库的 `std::list` 模板有多个参数。第二个参数(用于描述分配器)有默认值，但在 C++17 前，将 `std::list` 匹配到 `Container` 参数时，不会考虑默认值。

可变参数模板参数是上述 C++17 前“精确匹配”规则的例外，其为这个限制提供了一个解决方案：支持对双重模板参数进行更通用的匹配。一个双重模板参数包可以匹配零个或多个在双重模板参数中同类的模板形参：

```

1 #include <list>
2
3 template<typename T1, typename T2,
4          template<typename...> class Cont> // Cont expects any number of
5 class Rel { // type parameters
6 ...
7};
8
9 Rel<int, double, std::list> rel; // OK: std::list has two template parameters
10                                // but can be used with one argument

```

模板参数包只能匹配同类的模板实参。下面的类模板可以用只有模板类型参数的类模板或别名模板实例化，因为作为 `TT` 传递的模板类型参数包可以匹配零个或多个模板类型参数：

```

1 #include <list>
2 #include <map>
3 // declares in namespace std:
4 // template<typename Key, typename T,
5 // typename Compare = less<Key>,
6 // typename Allocator = allocator<pair<Key const, T>>>
7 // class map;
8 #include <array>
9 // declares in namespace std:
10 // template<typename T, size_t N>
11 // class array;
12
13 template<template<typename...> class TT>
14 class AlmostAnyTmpl {
15 };
16
17 AlmostAnyTmpl<std::vector> withVector; // two type parameters
18 AlmostAnyTmpl<std::map> withMap; // four type parameters
19 AlmostAnyTmpl<std::array> withArray; // ERROR: a template type parameter pack
20 // doesn't match a nontype template parameter

```

C++17 前，只有关键字 `class` 可以用来声明双重模板参数，这并不意味着只有用关键字 `class` 声明的类模板才允许作为替换参数。实际上，`struct` 模板、`union` 模板和 `alias` 模板都是双重模板参数的有效参数（C++11 引入别名模板后）。这类似于任何类型都可以使用关键字 `class` 声明的模板类型参数。

12.3.5 等价参数

当参数的值——相同时，两组模板实参是等价的。对于类型参数，类型别名并不重要：要比较的是类型别名声明的最终类型。对于整型非类型实参，比较实参的值；如何表达这个值并不重要：

```

1 template<typename T, int I>
2 class Mix;
3
4 using Int = int;
5
6 Mix<int, 3*3>* p1;
7 Mix<Int, 4+5>* p2; // p2 has the same type as p1

```

（从这个例子可以清楚地看出，建立模板参数列表的等价性不需要定义模板）

然而上下文中，模板参数的“值”不能确定，因此等价规则会变得稍微复杂一些。考虑下面的例子：

```

1 template<int N> struct I {};
2
3 template<int M, int N> void f(I<M+N>); // #1
4 template<int N, int M> void f(I<N+M>); // #2
5
6 template<int M, int N> void f(I<N+M>); // #3 ERROR

```

仔细研究 #1 和 #2 声明，通过将 M 和 N 分别重命名为 N 和 M，会得到相同的声明：因此，这两个表达式等价，声明了相同的函数模板 f。这两个声明中的表达式 M+N 和 N+M 等价。

然而，声明 #3 略有不同：操作数的顺序反了。这使得 #3 中的表达式 N+M 不等于其他两个表达式。由于表达式将为所涉及的模板参数的值产生相同的结果，这些表达式功能等效。如果模板声明的方式不同，只是因为声明包含了不相等的功能等效表达式，那就是错误的。这样的错误不需要编译器诊断，因为有些编译器可能在内部以与 N+2 完全相同的方式表示 N+1+1，而其他编译器可能不会这样。标准方面并没有强制一种特定的实现方法，而是允许其中任何一种方法，所以开发者在这方面需要三思而后行。

从函数模板生成的函数永远不等同于普通函数，即使具有相同的类型和相同的名称。这对类成员有两个重要的影响：

1. 从成员函数模板生成的函数不重写虚函数。
2. 从构造函数模板生成的构造函数不是复制或移动构造函数。

默认构造函数可以构造函数模板。

类似地，从赋值模板生成的赋值操作不是复制赋值操作符或移动赋值操作符。（然而，因为隐式调用复制赋值或移动赋值操作符不太常见，所以这不太容易出现问题。）

这种方式优缺点并存。请参阅 6.2 节和 6.4 节了解详细信息。

12.4. 可变参数模板

可变参数模板，已经在 4.1 节中介绍过了，是包含至少一个模板参数包的模板（参见 12.2.4 节）。

可变参数这个术语是从 C 的可变参数函数借鉴来的，可变参数函数接受任意数量的函数参数。可变参数模板还借鉴了 C 语言中使用省略号来表示零个或多个参数，并在某些应用程序中作为 C 语言可变参数函数的（类型）安全替代。

当模板的行为可泛化为任意数量的参数时，可变参数模板是有用的。在 12.2.4 节中介绍的 Tuple 类模板就是这样一种类型，因为 Tuple 可以有任意个元素，所有元素的处理方式都是相同的。还可以看下 print() 函数，其也能接受任意数量的参数，并按顺序显示每个参数。

当为可变参数模板确定模板参数时，可变参数模板中的每个模板参数包将匹配一个由零个或多个模板参数组成的序列，我们将这个模板参数序列称为参数包。下面的例子说明了模板形参包 Types，如何为 Tuple 提供的模板参数匹配不同的参数包：

```
1 template<typename... Types>
2 class Tuple {
3     public:
4         static constexpr std::size_t length = sizeof...(Types);
5     };
6
7 int a1[Tuple<int>::length]; // array of one integer
```

```
8 int a3[Tuple<short, int, long>::length]; // array of three integers
```

12.4.1 包扩展

sizeof…表达式是包扩展的一个例子，是将参数包展开为独立参数的构造。虽然 sizeof…执行这种扩展只是为了计算单独参数的数量，其他形式的参数包（即 C++ 所期望的列表形式）可以扩展为该列表中的多个元素。此类包扩展由列表中元素右侧的省略号（…）标识。下面是一个简单的例子，创建了一个新的类模板 MyTuple，派生自 Tuple，并传递相应参数：

```
1 template<typename... Types>
2 class MyTuple : public Tuple<Types...> {
3     // extra operations provided only for MyTuple
4 };
5
6 MyTuple<int, float> t2; // inherits from Tuple<int, float>
```

模板参数 Types…是一个包扩展，生成一个模板参数序列，在替换为 Types 的参数包中，每个参数对应一个参数。如示例所示，类型 MyTuple<int, float> 的实例化实参包 int, float 替换模板类型参数包 Types。当这发生在包扩展 Types…时，得到一个用于 int 的模板参数和一个用于 float 的模板参数，因此 MyTuple<int, float> 继承自 Tuple<int, float>。

理解包扩展的一种直观方法是将其视为语法扩展，其中模板参数包替换为相应数量的（非包）模板参数，而包扩展则作为单独的参数列出。例如，将 MyTuple 展开为两个参数：

这种对包扩展的语法理解是一个有用的工具，但是当模板参数包的长度为 0 时，就失效了。12.4.5 节提供了关于零长度包扩展解释的更多细节。

```
1 template<typename T1, typename T2>
2 class MyTuple : public Tuple<T1, T2> {
3     // extra operations provided only for MyTuple
4 };
```

对于三个参数：

```
1 template<typename T1, typename T2, typename T3>
2 class MyTuple : public Tuple<T1, T2, T3> {
3     // extra operations provided only for MyTuple
4 };
```

不能直接通过名称访问参数包中的单个元素，因为诸如 T1、T2 等名称在可变参数模板中没有定义。若需要这些类型，惟一能做的就是将它们（递归地）传递给另一个类或函数。

每个包展开都有一个模式，该模式将针对参数包中的每个参数重复的类型或表达式，通常出现在表示包展开的省略号之前。之前的例子只有一些简单的模式（参数包的名称），但模式的复杂度可以调整。可以定义一个新类型 PtrTuple，派生自指向其参数类型的指针元组：

```
1 template<typename... Types>
2 class PtrTuple : public Tuple<Types*...> {
```

```

3 // extra operations provided only for PtrTuple
4 };
5
6 PtrTuple<int, float> t3; // Inherits from Tuple<int*, float*>

```

包扩展的模式 Types*…在上面的例子中是 Types*。对这个模式的重复替换会产生一系列模板类型参数，所有这些参数都是相应类型的指针。在包扩展的语法解释下，若使用三个参数扩展 PtrTuple：

```

1 template<typename T1, typename T2, typename T3>
2 class PtrTuple : public Tuple<T1*, T2*, T3*> {
3     // extra operations provided only for PtrTuple
4 };

```

12.4.2 何处进行包扩展？

目前的示例都集中在，使用包扩展来生成一系列模板参数。事实上，包扩展可以在语法中提供逗号分隔列表的地方使用：

- 基类列表中。
- 构造函数中的基类初始化式列表中。
- 调用参数列表中（模式是参数表达式）。
- 初始化式列表中（例如，带括号的初始化式列表中）。
- 类、函数或别名模板的模板参数列表中。
- 函数可能抛出的异常列表中（在 C++11 和 C++14 中弃用，在 C++17 中不允许使用）。
- 属性中，如果属性本身支持包扩展（尽管 C++ 没有定义这样的属性）。
- 指定声明的对齐时。
- 指定 Lambda 的捕获列表时。
- 函数类型的参数列表中。
- 使用声明（自 C++17；参见 4.4.5 节）。

已经提到了 sizeof…作为一种包扩展机制，不会产生列表。C++17 还添加了折叠表达式，这是一种不会产生逗号分隔的机制（参见第 12.4.6 节）。

其中一些包扩展上下文仅仅为了完整性而包含，因此可以只关注那些在实践中可能有用的包扩展上下文。由于所有上下文中的包扩展都应遵循相同的原则和语法，因此若发现需要更深的包扩展上下文，能够从这里给出的示例进行推导。

基类列表中的包扩展扩展到一些直接基类。这样的扩展可以通过 mixin 聚合外部提供的数据和功能，mixin 是打算“混合”到一个类层次结构中的类，以提供新的行为。例如，下面的 Point 类在几种不同的上下文中使用包扩展，从而使用任意数量的 mixin：

mixin 将在 21.3 节中详细讨论。

```

1 template<typename... Mixins>
2 class Point : public Mixins... { // base class pack expansion
3     double x, y, z;

```

```

4 public:
5   Point() : Mixins()... { } // base class initializer pack expansion
6
7   template<typename Visitor>
8   void visitMixins(Visitor visitor) {
9     visitor(static_cast<Mixins&>(*this)...); // call argument pack expansion
10  }
11 }
12
13 struct Color { char red, green, blue; };
14 struct Label { std::string name; };
15 Point<Color, Label> p; // inherits from both Color and Label

```

Point 类使用包扩展来获取每个 mixin，并将其扩展为一个公共基类。然后，Point 的默认构造函数在基类初始化器列表中使用包展开，通过 mixin 机制，对每个基类进行值初始化。

成员函数模板 visitMixins 是最有趣的，因为它使用包展开的结果作为调用的参数。通过将 *this 强制转换为 mixin 类型，包扩展产生的调用参数指向与 mixin 对应的每个基类。实际上，可以编写访问器来使用 visitMixins，使用任意数量的函数调用参数，这在 12.4.3 节中有介绍。

包扩展也可以在模板参数列表中使用，以创建非类型参数包或模板参数包：

```

1 template<typename... Ts>
2 struct Values {
3   template<Ts... Vs>
4   struct Holder {
5     };
6   };
7
8 int i;
9 Values<char, int, int*>::Holder<'a', 17, &i> valueHolder;

```

当 Values<…> 已指定，则 Values<…>::Holder 非类型参数列表长度也就固定了；因此参数包 Vs 不是一个变长参数包。

Vs 是一个非类型模板参数包，其每个实际模板参数都可以具有不同的类型，由模板类型参数包 Ts 指定类型。Vs 声明中的省略号起着双重作用，既可以将模板参数声明为模板参数包，又可以将模板参数包的类型声明为包展开。虽然这样的模板参数包很罕见，但同样的原则适用于更一般的上下文：函数参数。

12.4.3 函数参数包

函数参数包是匹配零个或多个函数调用模板参数。与模板参数包一样，函数参数包使用省略号 (...) 在函数参数名称之前（或位置），与模板参数包一样，函数参数包在使用时必须使用包展开符来展开。模板参数包和函数参数包可以一并称为参数包。

与模板参数包不同，函数参数包始终是包扩展，因此声明的类型必须至少包含一个参数包。下面的例子中，引入了一个新的 Point 构造函数，提供的构造函数参数会复制初始化每个 mixin：

```

1 template<typename... Mixins>
2 class Point : public Mixins...

```

```

3 {
4     double x, y, z;
5 public:
6     // default constructor, visitor function, etc. elided
7     Point(Mixins... mixins) // mixins is a function parameter pack
8         : Mixins(mixins)... {} // initialize each base with the supplied mixin value
9     ;
10
11 struct Color { char red, green, blue; };
12 struct Label { std::string name; };
13 Point<Color, Label> p({0x7F, 0, 0x7F}, {"center"});

```

函数模板的函数参数包，可能依赖于该模板中声明的模板参数包，这允许函数模板接受任意数量的调用参数，而不丢失类型信息：

```

1 template<typename... Types>
2 void print(Types... values);
3
4 int main()
5 {
6     std::string welcome("Welcome to ");
7     print(welcome, "C++ ", 2011, '\n'); // calls print<std::string, char const*, 
8 } // int, char>

```

调用带有参数的函数模板 `print()` 时，参数的类型将放在参数包中以替代模板类型的参数包 `Types`，而实际参数值将放在参数包中以替代函数参数包的值。从调用中确定参数的过程将在第 15 章详细描述。现在，`Types` 中的第 i 个类型是 `values` 中的第 i 个值的类型，并且这两个参数包都可以在函数模板 `print()` 中使用。

`print()` 的实际使用递归模板实例化，这种模板元编程技术在第 8.1 节和第 23 章中有描述。

参数列表末尾的未命名函数参数包，和 C 风格的“vararg”参数之间存在语法上的歧义。例如：

```

1 template<typename T> void c_style(int, T...);
2 template<typename... T> void pack(int, T...);

```

第一种情况下，“`T…`”当作“`T, …`”：未命名的 `T` 类型参数，后面跟着一个 C 风格的可变参数。第二种情况，“`T…`”视为一个函数参数包，因为 `T` 是一个有效的扩展。若要消除歧义可以通过在省略号之前添加逗号(可以确保省略号视为 C 风格的“vararg”参数)或在…后添加标识符，这可使它成为一个命名函数参数包。在泛型 Lambda 中，末尾的…如果紧接在它前面的类型(没有中间的逗号)包含 `auto`，则将视为参数包。

12.4.4 多重和嵌套包扩展

包扩展的模式的复杂可变，可能包括多个不同的参数包。当实例化包含多个参数包的展开时，所有参数包必须具有相同的长度。通过将每个参数包的第一个参数代入模式，然后代入每个参数包的第二个参数，依此类推，形成类型或值的结果序列。下面的函数在将所有参数转发给函数对象 `f` 前，会复制它们：

```

1 template<typename F, typename... Types>

```

```

2 void forwardCopy(F f, Types const&... values) {
3     f(Types(values)...);
4 }
```

调用参数包扩展命名两个参数包，类型和值。实例化此模板时，Types 和 values 参数包的元素级扩展将生成一系列对象构造，这些构造通过在 Types 中将第 i 个值转换为第 i 个类型来构建值中的第 i 个值的副本。在包扩展的语法解释下，forwardCopy 的三个参数如下所示：

```

1 template<typename F, typename T1, typename T2, typename T3>
2 void forwardCopy(F f, T1 const& v1, T2 const& v2, T3 const& v3) {
3     f(T1(v1), T2(v2), T3(v3));
4 }
```

包扩展也可以嵌套，参数包的每次出现都由其最近的封装包展开来“展开”（且仅是该包展开）。下面的例子演示了包含三个不同参数包的嵌套扩展：

```

1 template<typename... OuterTypes>
2 class Nested {
3     template<typename... InnerTypes>
4     void f(InnerTypes const&... innerValues) {
5         g(OuterTypes(InnerTypes(innerValues)...)...);
6     }
7 };
```

对 g() 的调用中，带有模式 InnerTypes(innerValues) 的包展开是最内层的包展开，同时展开 InnerTypes 和 innerValues，并生成函数调用参数序列，用于初始化由 OuterTypes 表示的对象。外部包展开的模式包括内部包展开，为函数 g() 生成一组调用参数，这些参数根据内部展开生成的函数调用参数序列，初始化 OuterTypes 中的每个类型而创建的。在这个包扩展的语法解释下，OuterTypes 有两个参数，InnerTypes 和 innerValues 都有三个参数，嵌套变得更加明显：

```

1 template<typename O1, typename O2>
2 class Nested {
3     template<typename I1, typename I2, typename I3>
4     void f(I1 const& iv1, I2 const& iv2, I3 const& iv3) {
5         g(O1(I1(iv1), I2(iv2), I3(iv3)),
6          O2(I1(iv1), I2(iv2), I3(iv3)));
7     }
8 };
```

多重和嵌套包扩展是一个强大的工具（例如，参见第 26.2 节）。

12.4.5 扩展空参数包

对于理解可变参数模板的实例化如何使用不同数量的参数，包扩展解析是一个有用的工具。然而，在零长度参数包的情况下，语法解析通常会失败。考虑 12.4.2 节中的 Point 类模板，在语法上用零参数替换：

```

1 template<>
2 class Point : {
3     Point() : {}
```

```
4 };
```

上面的代码是错误的，因为模板参数列表为空，而空的基类和基类初始化器列表都有一个冒号。

包扩展实际上是语义结构，替换参数包并不影响如何解析包扩展（或其外围的可变参数模板）。当包展开为空列表时，行为（语义上）就好像列表不存在。实例化点 \diamond 没有基类，默认构造函数没有基类初始化器，但在其他方面没有问题。即使零长度包扩展的语法解释是定义良好（但不同）的，这种语义规则仍然有效。例如：

```
1 template<typename T, typename... Types>
2 void g(Types... values) {
3     T v(values...);
4 }
```

可变参数函数模板 g() 根据给定的值序列创建一个直接初始化的值 v。若这个值序列为空，v 的声明在语法上看起来就像函数声明 T v()。但由于是在语义上对包扩展进行替换，所以不会影响解析产生的实体的类型，可以用零参数初始化 v，即值初始化。

对于类模板的成员和类模板内嵌套类的成员也有类似的限制：若成员声明的类型看起来不是函数类型，实例化后该成员的类型是函数类型，则程序格式错误。因为该成员的语义解释已从数据成员变为成员函数了。

12.4.6 折叠表达式

编程中重复出现的一种模式是对一系列值进行操作的折叠。例如，函数 fn 对序列 x[1], x[2], …, x[n - 1], x[n] 的折叠可以表示为 fn(x[1], fn(x[2], fn(..., fn(x[n-1], x[n])…)))。

探索一种新的语言特性时，C++ 委员会遇到了为应用于包扩展的逻辑二进制操作符（即 `&&` 或 `||`），从而需要处理这种构造。若没有其他的特性，可以编写以下代码来实现 `&&` 操作符：

```
1 bool and_all() { return true; }
2 template<typename T>
3     bool and_all(T cond) { return cond; }
4 template<typename T, typename... Ts>
5     bool and_all(T cond, Ts...conds) {
6         return cond && and_all(conds...);
7     }
```

在 C++17 中，添加了名为折叠表达式的新特性（参见 4.2 节的介绍），适用于除`.`、`->` 和 `[]` 之外的所有二进制操作符。

给定一个未展开的表达式模式包和一个非模式表达式值，C++17 也允许为任何此类操作符 op 编写

```
1 (pack op ... op value)
```

对于右折算子（称为二元右折），或

```
1 (value op ... op pack)
```

或者是左折(称为二元左折), 这里需要括号。参见 4.2 节的一些基本示例。

折叠操作适用于展开包并增加值的序列, 其要么是序列的最后一个元素(适用于右折叠), 要么是序列的第一个元素(适用于左折叠)。

有了这个特性, 代码就可以像这样为每个传递的类型 T 调用一个特征

```
1 template<typename... T> bool g() {  
2     return and_all(trait<T>()...);  
3 }
```

(其中 and_all 是上面定义的), 可以写成

```
1 template<typename... T> bool g() {  
2     return (trait<T>() && ... && true);  
3 }
```

折叠表达式是包的扩展。如果包为空, 折叠表达式的类型可以根据非包操作数(上面表单中的值)确定。

然而, 该特性的设计者还希望有一个选项来省略值操作数。因此, C++17 还提供了另外两种形式: 一元右折

```
1 (pack op ...)
```

一元左折

```
1 (... op pack)
```

同样, 括号是必需的。显然, 这给空扩展带来了一个问题: 如何确定类型和值? 答案是一元折叠的空展开是错误的, 有三个例外:

- 对 `&&` 的一元折叠的空展开产生的值为 `true`
- 对 `||` 的一元折叠进行空展开将产生值 `false`
- 逗号操作符 `(,)` 的一元折叠的空展开将产生一个空的表达式

若以一种不寻常的方式重载这些特殊操作符中的一个, 将会产生意外。例如:

```
1 struct BooleanSymbol {  
2     ...  
3 };  
4  
5 BooleanSymbol operator|| (BooleanSymbol, BooleanSymbol);  
6  
7 template<typename... BTs> void symbolic(BTs... ps) {  
8     BooleanSymbol result = (ps || ...);  
9     ...  
10 }
```

假设使用派生自 `BooleanSymbol` 的类型来调用 `symbolic`。对于所有展开, 结果将产生一个布尔符号值, 除了空展开, 其将产生一个 `bool` 值。

因为重载这三个特殊操作符很罕见，所以这个问题很少（但很微妙）。最初的折叠表达式建议为更常见的操作符（如 + 和 *）包含空展开值，这将导致更严重的问题。

因此，通常会对一元折叠表达式的使用提出警告，并建议使用二元折叠表达式（显式指定空展开的值）。

12.5. 友元

友元声明的基本思想很简单：标识与出现友元声明的类有特权连接的类或函数。然而，由于两个原因，事情变得有些复杂：

1. 友元声明可能是实体的唯一声明。
2. 友元函数的声明可以是定义。

12.5.1 类模板的友元类

友元类声明不能是定义，因此很少有问题。在模板环境中，友元类声明的唯一方式是能够将一个特定的类模板实例命名为友元：

```
1 template<typename T>
2 class Node;
3
4 template<typename T>
5 class Tree {
6     friend class Node<T>;
7     ...
8 }
```

注意，当类模板的实例成为类或类模板的友元时，类模板必须可见。对于普通类，则没有这样的要求：

```
1 template<typename T>
2 class Tree {
3     friend class Factory; // OK even if first declaration of Factory
4     friend class Node<T>; // error if Node isn't visible
5 }
```

13.2.2 节对此有更多的说明。

在 5.5 节中介绍的一个应用是声明其他类模板实例化为友元：

```
1 template<typename T>
2 class Stack {
3     public:
4     ...
5     // assign stack of elements of type T2
6     template<typename T2>
7     Stack<T>& operator= (Stack<T2> const&);
8     // to get access to private members of Stack<T2> for any type T2:
```

```
9 template<typename> friend class Stack;
10 ...
11 };
```

C++11 添加了使模板参数成为友元的语法:

```
1 template<typename T>
2 class Wrap {
3     friend T;
4     ...
5 };
```

这对任何类型 T 都有效，若 T 不是实际的类类型，则会忽略。

12.5.2 类模板的友元函数

函数模板的实例可以成为友元函数，方法是确保友元函数的名称后面加上尖括号。尖括号可以包含模板参数，若参数可以推导，则尖括号可以为空:

```
1 template<typename T1, typename T2>
2 void combine(T1, T2);
3
4 class Mixer {
5     friend void combine<>(int&, int&);
6     // OK: T1 = int&, T2 = int&
7     friend void combine<int, int>(int, int);
8     // OK: T1 = int, T2 = int
9     friend void combine<char>(char, int);
10    // OK: T1 = char T2 = int
11    friend void combine<char>(char&, int);
12    // ERROR: doesn't match combine() template
13    friend void combine<>(long, long) { ... }
14    // ERROR: definition not allowed!
15 };
```

不能定义模板实例(最多只能定义特化)，因此命名实例的友元声明不能成为定义。

若名称后面没有尖括号，则有两种可能:

1. 若名称没有限定(不包含::)，永远不会引用模板实例。若在友元声明处没有匹配的非模板函数可见，则友元声明是该函数的第一个声明。声明也可以是定义。
2. 若该名称是限定的(包含::)，则该名称必须引用先前声明的函数或函数模板。匹配的函数优先于匹配的函数模板。但是，这样的友元声明不能是定义。

举一个例子:

```
1 void multiply(void*) ; // ordinary function
2
3 template<typename T>
4 void multiply(T) ; // function template
5
6 class Comrades {
```

```

7  friend void multiply(int) { }
8  // defines a new function ::multiply(int)
9
10 friend void ::multiply(void*);
11 // refers to the ordinary function above,
12 // not to the multiply<void*> instance
13
14 friend void ::multiply(int);
15 // refers to an instance of the template
16
17 friend void ::multiply<double*>(double*);
18 // qualified names can also have angle brackets,
19 // but a template must be visible
20
21 friend void ::error() { }
22 // ERROR: a qualified friend cannot be a definition
23 };

```

在一个普通类中声明了友元函数。同样的规则也适用于在类模板中的声明，但是模板参数可以参与识别友元函数：

```

1 template<typename T>
2 class Node {
3     Node<T>* allocate();
4     ...
5 };
6
7 template<typename T>
8 class List {
9     friend Node<T>* Node<T>::allocate();
10    ...
11 };

```

友元函数也可以在类模板中定义，其只在实际使用时实例化。这通常要求友元函数在类型中使用类模板本身，这使得在类模板上更容易表示，像在命名空间中调用函数一样：

```

1 template<typename T>
2 class Creator {
3     friend void feed(Creator<T>) { // every T instantiates a different function ::feed()
4         ...
5     }
6 };
7
8 int main()
9 {
10     Creator<void> one;
11     feed(one); // instantiates ::feed(Creator<void>)
12     Creator<double> two;
13     feed(two); // instantiates ::feed(Creator<double>)
14 }

```

这个例子中，每次 Creator 的实例化都会生成一个不同的函数。尽管这些函数是作为模板实例化的一部分生成的，但这些函数本身是普通函数，而不是模板的实例。但它们是模板实体(参见 12.1 节)，定义只有在使用时才会实例化。因为这些函数体是在类定义内部定义的，所以是隐式内联的。因此，在两个不同的编译单元中生成相同的函数不会出错。13.2.2 节和 21.2.1 节对这个主题有更多的说明。

12.5.3 友元模板

通常，当声明函数或类模板实例的友元时，可以明确表示哪个实体是友元。尽管如此，有时表示模板的所有实例都是类的友元也可以。这需要一个友元模板：

```
1 class Manager {
2     template<typename T>
3     friend class Task;
4
5     template<typename T>
6     friend void Schedule<T>::dispatch(Task<T>*);
7
8     template<typename T>
9     friend int ticket() {
10         return ++Manager::counter;
11     }
12
13 static int counter;
14 };
```

就像普通的友元声明一样，友元模板只有在命名了一个非限定的、后面没有尖括号的函数名时才可以定义。

友元模板只能声明主模板和主模板的成员。与主模板关联的偏特化和显式特化也认为是友元。

12.6. 后记

自 20 世纪 80 年代后期，C++ 模板的一般概念和语法保持相对稳定。类模板和函数模板是初始模板工具的一部分，类型参数和非类型参数也是。

然而，最初的设计中也有一些重要功能的添加，主要是出于 C++ 标准库的需要。成员模板可能是这些新增功能中最基本的。奇怪的是，只有成员函数模板正式纳入了 C++ 标准。由于编辑上的疏忽，成员类模板成为标准的一部分。

友元模板、默认模板参数和双重模板参数，是在 C++98 标准化过程中出现的。声明双重模板参数的能力有时称为高阶泛型。最初引入的原因是为了支持 C++ 标准库中的某个分配器模型，但这个分配器模型后来使用了不依赖于双重模板参数的模型。后来，双重模板参数几乎要从该语言中删除了，直到 1998 年标准的标准化过程的时候，规范仍然是不完整。最终，委员会的大多数成员决定保留它们，从而形成了新的标准。

别名模板是 2011 年标准的一部分。别名模板与经常要求的“类型定义模板”特性具有相同的需求，它使编写一个模板变得容易，而这个模板只不过是现有类模板的不同拼写。使之成为标准的规范(N2258)是由 Gabriel Dos Reis 和 Bjarne Stroustrup 编写的，Mat Marcus 也参与了该提案的早期

草案。Gaby 还为 C++14(N3651) 制定了变量模板建议的细节。最初，该提案只打算支持 `constexpr` 变量，但在标准草案中采用时，这一限制已经取消了。

可变参数模板是由 C++11 标准库和 Boost 库 (参见 [Boost]) 的需求驱动的，其中 C++ 模板库使用越来越高级 (和复杂) 的技术可以接受任意数量模板参数。Doug Gregor, Jäkko Jarvi, Gary Powell, Jens Maurer 和 Jason Merrill 提供了标准 (N2242) 的初始规范。开发规范的同时，Doug 还开发了该特性的原始实现 (在 GNU 的 GCC 中)，这对在标准库中使用该特性有很大帮助。

折叠表达式是 Andrew Sutton 和 Richard Smith 的成果，通过 N4191 将其添加到 C++17 中。

第 13 章 模板中的名称

大多数编程语言中，名称是一个基本概念。它们是开发者可以引用之前构造的实体的方法。C++ 编译器遇到一个名称时，必须“查找”以识别所引用的实体。从实现者的角度来看，C++ 在这方面是一门很难的语言。C++ 语句 `x*y;`，如果 `x` 和 `y` 是变量名，这条语句就是一个乘法语句，但如果 `x` 是类型名，那么这条语句将 `y` 声明为指向类型为 `x` 的实体的指针。

这个小示例表明 C++(像 C 一样)是一种上下文敏感的语言：若不了解上下文，就不能总是理解一个构造。这与模板有什么关系？模板构造必须处理多个上下文：(1) 模板出现的上下文，(2) 模板实例化的上下文，(3) 模板实例化的模板参数相关的上下文。因此，在 C++ 中必须非常小心地处理“名称”。

13.1. 名称的分类

C++ 用各种方法对名称进行分类——方法非常多。为了处理如此丰富的术语，我们提供了表 13.1 和表 13.2，其中描述了这些分类。幸运的是，通过熟悉两个主要的命名概念，就可以很好地了解大多数 C++ 模板问题：

1. 若名称所属的作用域是使用范围解析操作符 `(::)` 或成员访问操作符 `()` 显式表示的，则该名称为限定名称。或 `->`。例如，`this->count` 是限定名，但 `count` 不是（即使普通计数实际上可能引用类成员）。
2. 若名称在某种程度上依赖于模板参数，那么就是从属名称。如果 `T` 是模板参数，那么 `std::vector<T>::iterator` 通常是一个依赖名称，但如果 `T` 是一个已知的类型别名（例如使用 `T=int` 时的 `T`），那么就是一个非依赖名称。

通读这些表以熟悉用来描述 C++ 模板问题的术语，但没必要记住每个术语的确切含义。需要的时候，可以回来进行查找。

分类	解释及注意事项
标识符 Identifier	只能由不间断的字母、下划线和数字组成的名称。不能以数字开头，并且有些标识符是保留的：不应该在程序中引入（根据经验，避免开头下划线和双下划线）。“字母”的概念应该得到广泛的理解，包括从非字母语言编码字形的特殊通用字符名称（UCNs, universal character names）。
操作符函数标识 Operator-function-id	关键字 <code>operator</code> 后面跟着运算符的符号——例如， <code>operator new</code> 和 <code>operator []</code> 。许多运算符都有不同的表示。例如，运算符 <code>&</code> 可以等价地写作操作符位与，即使它表示的是一元运算取地址操作。

表 13.2. 名称的分类

转换函数标识 Conversion-function-id	用于表示用户定义的隐式转换操作符——例如，操作符 <code>int&</code> ，很容易与位与操作 (<code>bitand</code>) 混淆。
字面操作标识 Literal-operator-id	用于表示用户定义的文字操作符例如——操作符” <code>_km</code> ，可用来表示字面值 (比如 <code>100_km</code>)(C++11).
模板标识 Template-id	模板的名称，后面跟着用尖括号括起来的模板参数，例如： <code>List<T, int, 0></code> 。也可以是一个操作符函数标识或字面操作标识，后面加上用尖括号括起来的模板参数；例如： <code>operator+<X<int>></code>
非限定性标识 Unqualified-id	标识符的泛化，可以是上面的标识符 (标识符，操作符函数标识，转换函数标识符，转换函数标识，字面操作标识或模板标识) 或是“析构函数名称”(例如，像 <code>~Data</code> 或 <code>~List<T, T, N></code>)。
限定性标识 Qualified-id	非限定标识由类、枚举或名称空间的名称限定，或仅由全局作用域解析操作符限定。这样的名称本身就是限定的。例如， <code>::X</code> , <code>S::x</code> , <code>Array<T>::y</code> 和 <code>::N::A<T>::z</code> 。
限定名 Qualified name	这个术语在标准中没有定义，我们用它来指代经过限定查找的名称。具体来说，这是一个限定性标识或一个非限定标识，用于显式成员访问操作符 (. <code>或</code> - <code>></code>)。例如 <code>S::x</code> , <code>this->f</code> 和 <code>p->A::m</code> 。但在隐式等效于此的上下文中只使用 <code>class_mem</code> 等价于 <code>this->class_mem</code> 不是限定名：成员访问必须显式。
非限定名称 Unqualified name	非限定标识不是限定名称。这不是一个标准术语，而对应于标准所称的非限定名称的查找。
名称 Name	一个限定或不限定的名字。
相关名称 Dependent name	在某种程度上依赖于模板参数的名称。通常，显式包含模板参数的限定名或非限定名是依赖的。此外，成员访问操作符 (. <code>或</code> - <code>></code>) 通常是依赖于访问操作符左侧表达式的类型，这个概念在 13.3.6 节中讨论过。特别地，当 <code>b</code> 出现在模板中时 (<code>this->b</code>)，它通常是一个依赖名称。最后，需要根据参数进行查找的名称 (见第 13.2 节)，例如调用 <code>ident(x, y)</code> 形式中的 <code>ident</code> 或表达式 <code>x + y</code> 中的加法操作符，仅当参数表达式是类型依赖时，才是一个依赖名称。
不相关名称 Nondependent name	不属于上述描述的名称。

13.2. 查找名称

C++ 中查找名称有许多小技巧，但这里只关注几个主要的。只有在下面两种情景中才有必要确认名称查找的细节：(1) 直接处理会犯错的普通例子 (2) C++ 标准给出的错误例子。

限定名在限定构造所隐含的范围内查找。若该作用域是一个类，那么也可以搜索基类。但查找限定名称时，不考虑封闭作用域。基本原则如下：

```

1 int x;
2
3 class B {

```

```

4   public:
5     int i;
6   };
7
8 class D : public B {
9 };
10
11 void f(D* pd)
12 {
13   pd->i = 3; // finds B::i
14   D::x = 2; // ERROR: does not find ::x in the enclosing scope
15 }

```

相比之下，不限定名通常会依次在外围作用域中查找(尽管在成员函数定义中，类及其基类的作用域会在其他外围作用域之前搜索)，这称为常规查找。下面是一个基本的例子，展示了常规查找的主要思想：

```

1 extern int count; // #1
2
3 int lookup_example(int count) // #2
4 {
5   if (count < 0) {
6     int count = 1; // #3
7     lookup_example(count); // unqualified count refers to #3
8   }
9   return count + ::count; // the first (unqualified) count refers to #2 ;
10 } // the second (qualified) count refers to #1

```

对于非限定名称的查找，最近增加了一项新的查找机制——除了常规的查找之外——有时可能要进行依赖参数的查找 (ADL)。

在 C++98/C++03 中，这也称为 Koenig 查找(或扩展 Koenig 查找)，以 Andrew Koenig 命名，他首先提出了这种机制的变体。

继续研究 ADL 之前，用 max() 模板来触发这个机制：

```

1 template<typename T>
2 T max (T a, T b)
3 {
4   return b < a ? a : b;
5 }

```

假设现在需要将此模板应用于，定义在另一个命名空间中的类型：

```

1 namespace BigMath {
2   class BigNumber {
3     ...
4   };
5   bool operator < (BigNumber const&, BigNumber const&);
6   ...

```

```

7 }
8
9 using BigMath::BigNumber;
10
11 void g (BigNumber const& a, BigNumber const& b)
12 {
13     ...
14     BigNumber x = ::max(a,b);
15     ...
16 }
```

这里的问题是 `max()` 模板不知道 `BigMath` 名称空间，但是常规查找无法找到适用于 `BigNumber` 类型值的小于操作符。如果没有一些特殊的规则，这将大大降低模板在 C++ 名称空间中的适用性。ADL 就是对这些“特殊规则”的回答。

13.2.1 ADL

ADL 主要适用于非限定名称，这些名称看起来像是在函数调用或操作符调用中命名非成员函数。若在常规查找中，就不会发生 ADL 了

- 成员函数的名称，
- 变量名，
- 类型的名称，或
- 块作用域函数声明的名称。

若要调用的函数名在括号内，则 ADL 会受到抑制。

否则，若名称后面跟着用圆括号括起来的参数表达式列表，ADL 将继续在与调用参数类型“相关”的命名空间和类中查找名称。这些关联的命名空间和关联类的精确定义将在后面给出，但它们可以认为是将给定类型，直接相连的所有命名空间和类。如果类型是指向类 X 的指针，那么相关的类和命名空间将包括 X，以及 X 所属的命名空间或类。

对于给定类型的关联命名空间和关联类的精确定义，由以下规则确定：

- 内置类型，这是空集。
- 指针和数组类型，关联的命名空间和类的集合是基础类型的集合。
- 枚举类型，关联的命名空间是在其中声明枚举的命名空间。
- 类成员，外围类是关联类。
- 类类型（包括联合类型），相关类的集合是类型本身、封闭类，以及直接和间接基类。关联命名空间的集合在其中声明关联类的命名空间。如果类是类模板实例，则模板类型参数的类型，以及声明模板参数的类和命名空间也包括在内。
- 函数类型，关联的命名空间和类集合包含，与所有参数类型和返回类型关联的命名空间和类。
- 指向类 X 成员类型的指针，关联的命名空间和类集除了与成员类型关联的命名空间和类外，还包括与 X 关联的命名空间和类（如果是指向成员函数类型的指针，那么参数和返回类型也有作用）。

然后，ADL 在所有关联的命名空间中查找该名称，就好像该名称已经通过这些命名空间进行了限定一样（除非忽略 using 的指示）：

details/adl.cpp

```
1 #include <iostream>
2
3 namespace X {
4     template<typename T> void f(T);
5 }
6
7 namespace N {
8     using namespace X;
9     enum E { e1 };
10    void f(E) {
11        std::cout << "N::f(N::E) called\n";
12    }
13 }
14
15 void f(int)
16 {
17     std::cout << "::f(int) called\n";
18 }
19
20 int main()
21 {
22     ::f(N::e1); // qualified function name: no ADL
23     f(N::e1); // ordinary lookup finds ::f() and ADL finds N::f(),
24 } // the latter is preferred
```

本例中执行 ADL 时，命名空间 N 中的 using 指令会忽略。因此，X::f() 不是 main() 中函数调用的候选。

13.2.2 友元声明的 ADL

友元函数声明可以是指定函数，假定函数是在包含友元声明的类的最近的命名空间作用域（可能是全局作用域）中声明的。但是，这样的友元声明在该作用域中不可见。

```
1 template<typename T>
2 class C {
3     ...
4     friend void f();
5     friend void f(C<T> const&);
6     ...
7 };
8
9 void g (C<int>* p)
10 {
11     f(); // is f() visible here?
```

```
12     f(*p); // is f(C<int> const&) visible here?  
13 }
```

如果友元声明在封闭的命名空间中可见，实例化类模板可能会使普通函数的声明可见。这将导致令人惊讶的行为：调用 `f()` 将导致编译错误，除非在程序的早些时候实例化了类 `C`！

另一方面，只在友元声明中声明（和定义）函数也可以（参见 21.2.1 节，了解这种行为的方式）。当是友元的类在 ADL 的关联类中时，就可以找到这样的函数了。

重新看下一个例子。调用 `f()` 没有相关类或命名空间，因为没有参数，所以是无效的调用。但调用 `f(*p)` 确实有关联类 `C<int>`（因为类型是 `*p`），并且全局命名空间也相关联（因为这是声明 `*p` 类型的命名空间）。因此，若调用之前完全实例化 `C<int>`，就可以找到第二个友元函数声明。为了确保这一点，假设涉及查找关联类中的友元的调用会导致类的实例化（若没有这样做的话）。

虽然这是 C++ 标准作者们的意图，但在标准中并没有明确说明。

ADL 查找友元声明和定义的能力，有时称为友元名称注入。然而，这个术语有一定的误导性，因为它是 C++ 标准之前的一个特性名称，该特性“注入”了友元声明的名称到外围作用域，使它们对普通的名称查找可见。所以在上面的例子中，两个调用是定义良好的。本章的后记会进一步详细介绍友元注入的历史。

13.2.3 注入类名

类名注入到该类本身的作用域中，可以作为该作用域中的非限定名称访问（只是用来表示构造函数的符号，所以不能作为限定名访问）：

details/adl.cpp

```
1 #include <iostream>  
2  
3 int C;  
4  
5 class C {  
6 private:  
7     int i[2];  
8 public:  
9     static int f() {  
10         return sizeof(C);  
11     }  
12 };  
13  
14 int f()  
15 {  
16     return sizeof(C);  
17 }  
18  
19 int main()  
20 {
```

```

21 std::cout << "C::f() = " << C::f() << ', '
22 << " ::f() = " << ::f() << '\n';
23 }

```

成员函数 C::f() 返回类型 C 的大小，而函数::f() 返回变量 C 的大小 (int 的大小)。

类模板也有注入类名，但比普通的方式更奇怪：它们后面可以跟着模板参数（这种情况下，注入的是类模板名），但是如果后面没有模板参数，就表示将其形参作为实参的类（对于偏特化，则表示其特化的实参），如果是上下文需要类型，则表示为模板。这就解释了以下情况：

```

1 template<template<typename> class TT> class X {
2 };
3
4 template<typename T> class C {
5     C* a; // OK: same as "C<T>* a;"
6     C<void>& b; // OK
7     X<C> c; // OK: C without a template argument list denotes the template C
8     X<::C> d; // OK: ::C is not the injected class name and therefore always
9         // denotes the template
10};

```

注意非限定名称是如何引用注入名称的，如果没有模板参数，就不会认为是模板名称。为了弥补这一点，可以使用作用域限定符:: 强制找到模板名称。

可变参数模板注入的类名还有一个问题：若通过使用可变参数模板的模板参数直接注入类名，那么注入的类名将包含没有展开的模板参数包（参见 12.4.1 节了解包展开的详细信息）。因此，当为可变参数模板参数注入类名时，对应的模板参数包是一个模式为该模板参数包的包展开：

```

1 template<int I, typename... T> class V {
2     V* a; // OK: same as "V<I, T...>* a;"
3     V<0, void> b; // OK
4 };

```

13.2.4 当前实例化类

为类或类模板的注入类名实际上是定义类型的别名。对于非模板类，这个属性很明显，因为类本身是具有该名称，且是在该作用域中的类型。但在类模板或类模板内嵌套的类中，每个模板实例都是不同的类型。这个属性在上下文中特别有趣，因为它注入的类名引用了类模板的相同实例化类，而不是该类模板的其他特化（对于类模板的嵌套类也是如此）。

类模板中，注入类名或任何与外围类，或类模板注入类名等价的类型（包括查看类型别名声明）都指向当前的实例。依赖于模板参数的类型（即依赖类型），但不引用当前实例化类型则称为引用未知特化，该特化可以从相同的类模板或完全不同的类模板实例化。下面的例子说明了其中的区别：

```

1 template<typename T> class Node {
2     using Type = T;
3     Node* next; // Node refers to a current instantiation
4     Node<Type>* previous; // Node<Type> refers to a current instantiation
5     Node<T*>* parent; // Node<T*> refers to an unknown specialization
6 };

```

嵌套类和类模板的情况下，确定类型是否引用当前实例化可能会造成混淆。外围类和类模板（或与其等价的类型）的注入类名引用当前实例化类，而其他嵌套类或类模板的名称则不引用：

```
1 template<typename T> class C {
2     using Type = T;
3     struct I {
4         C* c; // C refers to a current instantiation
5         C<Type>* c2; // C<Type> refers to a current instantiation
6         I* i; // I refers to a current instantiation
7     };
8
9     struct J {
10        C* c; // C refers to a current instantiation
11        C<Type>* c2; // C<Type> refers to a current instantiation
12        I* i; // I refers to an unknown specialization,
13            // because I does not enclose J
14        J* j; // J refers to a current instantiation
15    };
16};
```

当类型引用当前实例化时，其内容保证从当前定义的类模板或其嵌套类中实例化。这对解析模板时的名称查找有影响，但也产生了另一种更类似游戏的方法，以确定类模板定义中的类型 X 是引用当前的实例化，还是未知的特化：若另一个开发者编写显式特化（第 16 章中详细描述），使 X 引用该特化，那么 X 引用未知的特化。上面的例子中考虑类型 $C<\text{int}>::J$ 的实例化：用于实例化 $C<T>::J$ 的定义（因为这是正在实例化的类型）。此外，由于显式特化需要在了解所有外围模板或成员的情况下特化模板或模板成员，因此 $C<\text{int}>$ 将从外围类定义实例化。所以，对 J 和 $C<\text{int}>$ （其中 Type 为 int）的引用指向当前的实例化类。另外，可以为 $C<\text{int}>::I$ 编写显式特化类：

```
1 template<> struct C<int>::I {
2     // definition of the specialization
3 };
```

这里，为 $C<\text{int}>::I$ 的特化提供了一个与 $C<T>::J$ 定义中可见的定义完全不同的定义。因此 $C<T>::J$ 定义中，I 指的是一个未知的特化类。

13.3. 解析模板

大多数编程语言的编译器的两个基本活动是标记化（也称为扫描或词法）和解析。标记化过程将源代码读取为字符序列，并从中生成一系列标记。例如，当看到字符序列 $\text{int}^* \text{p} = 0;$ 时，“分词器”会将为关键字 int、符号/运算符 *、标识符 p、符号/运算符 =、整数文字 0 和符号/运算符；生成记号描述。

然后，解析器通过递归减少标记或先前模式到更高级别的构造中，从而在标记序列中找到已知的模式。例如，词牌 0 是一个有效的表达式，后跟标识符 p 的组合 * 是一个有效的声明符，声明符后跟 “=” 再后跟表达式 “0” 是一个初始化声明符。最后，关键字 int 是已知的类型名称，当后跟初始化声明器 $*\text{p} = 0$ 时，即得到 p 的初始化声明。

13.3.1 非模板中的上下文相关性

词法化比解析更容易，词法解析是一门已经发展出坚实理论的学科，许多语言都使用该理论进行解析。然而，该理论最适合上下文无关的语言，我们知道 C++ 是上下文敏感的。为了处理这个问题，C++ 编译器会将符号表与分词器和解析器耦合起来：当解析一个声明时，会使用符号表进行解析。当分词器找到一个标识符时，会进行查找，并在找到匹配类型时对结果词法进行注释。

例如，C++ 编译器看到

```
1 x*
```

分词器会查找 x，如果找到类型，解析器将看到

```
1 identifier, type, x
2 symbol, *
```

并得出结论说，这是一个声明。但是，如果 x 不是类型，则解析器从分词器接收到的信息为

```
1 identifier, nontype, x
2 symbol, *
```

该构造只能通过乘法进行有效解析，这些细节取决于具体实现策略。

下面的表达式说明了上下文相关的另一个例子：

```
1 X<1>(0)
```

如果 X 是类模板的名称，表达式将整数 0 强制转换为从该模板生成的类型 X<1>。如果 X 不是模板，则表达式等价于

```
1 (X<1>)0
```

换句话说，X 与 1 比较，比较的结果（下隐式转换为 1 或 0）与 0 比较。虽然这样的代码很少使用，但它是有效的 C++（和有效的 C）代码。因此，C++ 解析器将查找出现在 < 之前的名称，当知道该名称是模板名称时，才将 < 作为尖括号处理；否则，< 将视为小于操作符。

这种形式的上下文相关性，是选择尖括号来分隔模板参数列表的结果。这是另一种情况：

```
1 template<bool B>
2 class Invert {
3 public:
4     static bool const result = !B;
5 };
6
7 void g()
8 {
9     bool test = Invert<(1>0)>::result; // parentheses required!
10 }
```

如果省略了 Invert<(1>0)> 中的圆括号，则大于符号将误认为模板参数列表的结束符号。这将使代码无效，因为编译器将其视为 ((Invert<1>))0>::result。

使用双括号是为了避免将 (Invert<1>)0 作为强制转换操作进行解析——这也是导致语法歧义的另一个原因。

分词器也不能避免尖括号表示法的问题。例如：

```
1 List<List<int>> a;
2 // ^ -- no space between right angle brackets
```

两个>字符组合成右移标记`>>`，因此分词器不会将其视为两个单独的标记。这是最大蒙克标记化原则的结果：C++ 实现必须将尽可能多的连续字符收集到标记中。

可以引入了特定的异常来解决本节中描述的标记化问题。

正如在 2.2 节中提到的，C++11 标准特别调用了这种情况——嵌套的模板标识使用`>>`标记作为结束——在解析器中，等同于两个独立的尖括号，一次性关闭两个模板。

1998 年和 2003 年版本的 C++ 标准不支持这种“尖括号黑客”。然而，需要在两个连续的尖括号之间加入一个空格，这对模板初学者来说是一个障碍，因此委员会决定在 2011 年的标准中对这种写法进行改变。

这一变化改变了某些程序(诚然，这些程序是人为设计的)的意义。考虑下面的例子：

details/anglebrackethack.cpp

```
1 #include <iostream>
2
3 template<int I> struct X {
4     static int const c = 2;
5 };
6
7 template<> struct X<0> {
8     typedef int c;
9 };
10
11 template<typename T> struct Y {
12     static int const c = 3;
13 };
14
15 static int const c = 4;
16
17 int main()
18 {
19     std::cout << (Y<X<1> >::c >::c>::c) << ' ';
20     std::cout << (Y<X< 1>>::c >::c>::c) << '\n';
21 }
```

这是一套 C++98 代码，输出为 0 3。在使用 C++11 标准时，尖括号使用方式的变革使得括号内的两个语句等价，所以输出为 0 0。

一些提供 C++98 或 C++03 模式的编译器会保持 C++11 在这些模式下的行为，因此即使使用 C++98/C++03 编译也会打印 0 0。

由于“<:”是“[”(在一些传统键盘上不可用)的替代，因此也存在类似的问题。看看下面的例子：

```
1 template<typename T> struct G {};
2 struct S;
3 G<::S> gs; // valid since C++11, but an error before that
```

C++11 前，最后一行代码相当于 G[:S> gs;，这显然是无效的代码标识。为了解决这个问题，添加了另一个“词法黑客”：当编译器看到字符 <: 没有立即跟在: 或 > 后面时，前导“<:”不再等同于 “[”。

这是前面提到的最大蒙克原则的一个例外。

这个黑客技巧使得以前有效(但有些人为)的代码，编程无效代码：

```
1 #define F(X) X ## :
2
3 int a[] = { 1, 2, 3 }, i = 1;
4 int n = a F(<::) i]; // valid in C++98/C++03, but not in C++11
```

要理解这一点，“词法黑客”适用于预处理标记，这是预处理程序可以接受的标记类型(在预处理完成后可能无法接受)，它们在宏展开完成之前决定。考虑到这一点，C++98/C++03 在宏调用 F(<::) 中无条件地将“<:”转换为 “[”，n 的定义扩展为

```
1 int n = a [ :: i];
```

这完全没问题。因为在宏展开之前，“<::”后面不是“:”或“>”，而是“)”，所以 C++11 则不会进行字符转换。没有转换，连接操作符必须尝试将“::”和“)”粘合到新的预处理标记中。但这不起作用，因为“::”不是有效的连接标记。C++11 标准导致了这种未定义行为，有些编译器会诊断这个问题，只是将两个预处理标记分开，这是一个语法错误，因为它导致 n 的定义扩展为

```
1 int n = a < :: : i];
```

13.3.2 依赖型类型名称

模板名称的问题在于，不能进行合理的分类。特别是，模板不能看到另一个模板，因为另一个模板的内容可以通过显式特化使其无效：

```
1 template<typename T>
2 class Trap {
3     public:
4         enum { x }; // #1 x is not a type here
5     };
6
7 template<typename T>
8 class Victim {
```

```

9  public:
10 int y;
11 void poof() {
12     Trap<T>::x * y; // #2 declaration or multiplication?
13 }
14 };
15
16 template<>
17 class Trap<void> { // evil specialization!
18 public:
19     using x = int; // #3 x is a type here
20 };
21
22 void boom(Victim<void>& bomb)
23 {
24     bomb.poof();
25 }

```

当编译器解析 #2 行时，必须判断看到的是声明还是乘法，这取决于依赖的限定名 `Trap<T>::x` 是否是类型名。可能很容易在模板 `Trap` 中查找，并根据 #1 行，知道 `Trap<T>::x` 不是一个类型，这将使 #2 行是一个乘法，但其为 `T` 为 `void` 的情况重写了通用的 `Trap<T>::x`。这种情况下，`Trap<T>::x` 实际上是 `int` 类型。

例子中，类型 `Trap<T>` 是一个依赖类型，该类型依赖于模板参数 `T`。而且，`Trap<T>` 引用了一个未知的特化(在第 13.2.4 节中描述)，从而编译器不能安全地查看模板内部，来确定名称 `Trap<T>::x` 是否是一个类型。如果”`::`”前面的类型引用当前实例化类——`Trap<T>::y`——编译器就可以看到模板定义，从而确定没有其他特化介入。因此，当”`::`”前面的类型指向当前实例化类时，模板中的限定名查找与非依赖类型的限定名查找非常类似。

但对未知特化的名称查找仍是一个问题。解决问题的方法是，指定依赖的限定名不表示类型，除非该名称以关键字 `typename` 作为前缀。若替换模板参数之后，发现名称不是类型的名称，则程序无效，并且 C++ 编译器应该在实例化时报错。注意，`typename` 的这种用法与表示模板类型参数的用法不同。与类型参数不同，不能等价地将 `typename` 替换为 `class`。

当名称满足以下条件时，`typename` 前缀是必需的：

C++20 在大多数情况下可能会删除对 `typename` 的需要(请参阅第 17.1 节了解详细信息)。

1. 是有限定的，而不是在后面添加”`::`”形成一个更有限定的名称。
2. 不是详细类型说明符(即以关键字 `class`、`struct`、`union` 或 `enum` 之一开头的类型名)的一部分。
3. 不在基类规范列表中或引入构造函数定义的成员初始化式列表中使用。

从语法上讲，这些上下文中只允许使用类型名，因此可以假设使用限定名来命名类型。

4. 依赖于模板参数。
5. 是未知特化的成员，由限定符命名的类型指向未知的特化类型。

此外，若不满足前两个条件，则不可使用 `typename` 前缀。为了说明这一点，看看以下的错误例子：

改编自 [VandevoordeSolutions]，多次证明 C++ 代码重用的重要性。

```
1 template<typename1 T>
2 struct S : typename2 X<T>::Base {
3     S() : typename3 X<T>::Base(typename4 X<T>::Base(0)) {
4 }
5     typename5 X<T> f() {
6         typename6 X<T>::C * p; // declaration of pointer p
7         X<T>::D * q; // multiplication!
8     }
9     typename7 X<int>::C * s;
10
11     using Type = T;
12     using OtherType = typename8 S<T>::Type;
13 }
```

每次出现 `typename`(不管正确与否) 都用一个下标进行编号，以便于引用。第一个 `typename1` 表示模板参数。前面的规则不适用于第一次使用。

前面规则中的第二项不允许使用第二个和第三个类型名。这两个上下文中基类的名称之前不能有 `typename`。但是，`typename4` 是必需的。这里，基类的名称不表示要初始化或派生的对象，而该名称是表达式的一部分，该表达式从其参数 0 构建临时 `X<T>::Base`(可以认为是一种转换)。禁止使用第五个 `typename`，因为后面的名称 `X<T>` 不是限定名称。若此语句要声明指针，则 `typename` 需要出现第六次。下一行省略了 `typename` 关键字，因此编译器将其解释为乘法。第七个 `typename` 可选，因为它满足前两个规则，但不满足后两个规则。第八个 `typename` 也可选，因为它引用当前实例化的一个成员(因此不满足最后一个规则)。

确定是否需要 `typename` 前缀的最后一个规则有时很难评估，因为它依赖于确定类型是引用当前实例化，还是未知特化的规则。所以，最安全的做法是添加 `typename` 关键字，以表明希望后面的限定名是类型。`typename` 关键字，即使它是可选的，也会为开发者的意图增加可读性。

13.3.3 依赖型模板名称

当模板名称是依赖型名称时，会出现与上一节中遇到的问题非常相似的问题。通常，C++ 编译器需要将模板名称后面的 `<` 作为模板参数列表的开头。与类型名的情况一样，除非开发者使用关键字 `template` 提供额外的信息，否则编译器必须假设依赖型名称不指向模板：

```
1 template<typename T>
2 class Shell {
3     public:
4         template<int N>
5         class In {
6             public:
7                 template<int M>
8                 class Deep {
```

```

9     public:
10    virtual void f();
11  };
12 }
13 };
14
15 template<typename T, int N>
16 class Weird {
17 public:
18 void case1 (
19 typename Shell<T>::template In<N>::template Deep<N>* p) {
20   p->template Deep<N>::f(); // inhibit virtual call
21 }
22
23 void case2 (
24 typename Shell<T>::template In<N>::template Deep<N>& p) {
25   p.template Deep<N>::f(); // inhibit virtual call
26 }
27 };

```

这个有点复杂的示例展示了所有可以限定名称 (::、-> 和.) 的操作符 (可能需要跟在 template 关键字的后面)。当限定操作符前面的名称或表达式的类型，依赖于模板参数，并引用未知的特化，并且操作符后面的名称是模板标识 (模板名称) 时，就会出现这种情况。例如：

```
1 p.template Deep<N>::f()
```

p 的类型依赖于模板参数 T，所以 C++ 编译器无法通过查找 Deep 来确定它是否是模板，而必须通过前缀 template 来显式地指出 Deep 是模板名称。若没有这个前缀，p.Deep<n>::f() 会解析为 ((p.Deep)<n>)>f()。这可能会在限定名中进行多次，因为限定符本身可能使用依赖限定符进行限定 (可以通过前面示例中 case1 和 case2 参数的声明来说明)。

若省略了关键字 template，那么开始和结束尖括号将解析为小于和大于操作符。与 typename 关键字一样，可以安全地添加 template 前缀，以表明下面的名称是模板标识 (即使 template 前缀不是必须的)。

13.3.4 Using 中声明依赖型名称

使用声明可以引入来自两个地方的名称：命名空间和类。命名空间的情况与此不同，因为不存在“命名空间模板”。另一方面，使用从类引入名称的声明只能从基类引入名称到派生类。这种 using 声明的行为类似于派生类到基类声明的“符号链接”或“快捷方式”，从而允许派生类的成员访问指定的名称，就好像是派生类中声明的成员一样。简短的非模板示例会比文字更能说明这种情况：

```

1 class BX {
2 public:
3   void f(int);
4   void f(char const*);
5   void g();
6 };
7

```

```
8 class DX : private BX {
9     public:
10    using BX::f;
11};
```

前面的 `using` 声明将基类 `BX` 的名称 `f` 引入派生类 `DX`。这个名称与两个不同的声明相关联，从而强调处理的是名称机制，而不是这些名称的声明。这种 `using` 声明可以使无法访问的成员，变为可访问的。基函数 `BX`(以及它的成员)是 `DX` 类私有的，函数 `BX::f`引入到了 `DX` 的公有接口，从而可以进行访问。

当 `using` 声明从从属类引入名称时，可能会发现一个问题。虽然知道名称，但不知道它是类型、模板，还是其他什么：

```
1 template<typename T>
2 class BXT {
3     public:
4     using Mystery = T;
5     template<typename U>
6     struct Magic;
7 };
8
9 template<typename T>
10 class DXTT : private BXT<T> {
11     public:
12     using typename BXT<T>::Mystery;
13     Mystery* p; // would be a syntax error without the earlier typename
14};
```

若希望通过 `using` 声明引入依赖型名称来指定类型时，则必须通过关键字 `typename` 显式说明。奇怪的是，C++ 标准没有提供类似的机制来将依赖名称标记为模板：

```
1 template<typename T>
2 class DXTM : private BXT<T> {
3     public:
4     using BXT<T>::template Magic; // ERROR: not standard
5     Magic<T>* plink; // SYNTAX ERROR: Magic is not a
6 }; // known template
```

标准化委员会并没有想要解决这个问题的意思，但 C++11 别名模板确实提供了部分解决方案：

```
1 template<typename T>
2 class DXTM : private BXT<T> {
3     public:
4     template<typename U>
5     using Magic = typename BXT<T>::template Magic<T>; // Alias template
6     Magic<T>* plink; // OK
7 };
```

这有点笨拙，但是对于类模板来说，达到了预期的效果。但函数模板的情况(可以说不太常见)仍然没有得到解决。

13.3.5 ADL 和显式模板参数

```
1 namespace N {
2     class X {
3         ...
4     };
5     template<int I> void select(X* );
6 }
7
8 void g (N::X* xp)
9 {
10    select<3>(xp); // ERROR: no ADL!
11 }
```

这个例子中，可能期望通过调用 `select<3>(xp)` 中的 ADL 找到模板 `select()`。然而，情况并非如此，因为编译器在确定 `<3>` 是模板参数列表之前，无法确定 `xp` 是否是一个函数调用参数。在找到 `select()` 是模板之前，编译器不能确定 `<3>` 是否是模板参数列表。因为无法解决这个“先有鸡还是先有蛋”的问题，所以表达式解析为 `(select<3>)(xp)`。

这个例子可能看起来像模板标识禁用了 ADL，但事实并非如此。可以通过引入一个名为 `select` 的函数模板来修复该代码，函数模板在调用时可见：

```
1 template<typename T> void select();
```

即使 `select<3>(xp)` 没有任何意义，这个函数模板的存在确保 `select<3>` 会解析为模板标识。然后 ADL 会找到函数模板 `N::select`，使调用成功。

13.3.6 依赖型表达式

与名称一样，表达式本身也依赖于模板参数。依赖于模板参数的表达式在每次实例化和下一次实例化时的行为可能不同——选择不同的重载函数或生成不同的类型或常量值。相反，不依赖于模板参数的表达式在所有实例化中行为相同。

表达式可以以几种不同的方式依赖于模板参数。依赖型表达式最常见的形式是类型依赖，表达式本身的类型可以从一个实例化到下一个实例化，例如：一个表达式引用一个函数参数，其类型是模板参数的类型：

```
1 template<typename T> void typeDependent1(T x)
2 {
3     x; // the expression type-dependent, because the type of x can vary
4 }
```

具有依赖于类型子表达式的表达式本身通常也依赖于类型——使用参数 `x` 调用函数 `f()`：

```
1 template<typename T> void typeDependent2(T x)
2 {
3     f(x); // the expression is type-dependent, because x is type-dependent
4 }
```

注意 `f(x)` 的类型在不同的实例化中可能不同，这是因为 `f` 可能解析为一个结果类型依赖于参数类型的模板，也因为两阶段查找（在第 14.3.1 节中讨论）可能在不同的实例化中发现名为 `f` 的函数。

并非所有涉及模板参数的表达式都依赖于类型。包含模板参数的表达式，可以在每次实例化到下一次实例化时产生不同的常量值。这样的表达式称为值依赖型表达式，最简单的是引用非依赖类型的非类型模板参数表达式。例如：

```
1 template<int N> void valueDependent1()
2 {
3     N; // the expression is value-dependent but not type-dependent,
4     // because N has a fixed type but a varying constant value
5 }
```

与类型依赖表达式一样，如果表达式由其他值依赖型表达式组成，那么这个表达式也是值依赖型的，因此 $N + N$ 或 $f(N)$ 也是依赖值的表达式。

有趣的是，有些操作（如 `sizeof`）具有已知的结果类型，因此可以将依赖操作数类型的表达式，转换为不依赖值类型的表达式。例如：

```
1 template<typename T> void valueDependent2(T x)
2 {
3     sizeof(x); // the expression is value-dependent but not type-dependent
4 }
```

`sizeof` 操作会产生 `std::size_t` 类型的值，不管输入是什么。因此 `sizeof` 表达式不依赖于类型（即使子表达式依赖于类型）。但产生的常量值会因实例化的不同而不同，因此 `sizeof(x)` 是一个值依赖型的表达式。

如果对值依赖型表达式应用 `sizeof` 会怎样呢？

```
1 template<typename T> void maybeDependent(T const& x)
2 {
3     sizeof(sizeof(x));
4 }
```

这里，内部 `sizeof` 表达式依赖于值。然而，外部的 `sizeof` 表达式总是生成 `std::size_t` 类型的值，因此类型和常量值在模板的所有实例化中都是一致的。任何涉及模板参数的表达式都是实例化依赖型表达式，

C++ 标准中使用了类型和值依赖型的表达式来描述模板的语义，它们对模板实例化有影响（第 14 章）。另一方面，术语实例化依赖表达式主要供 C++ 编译器的作者使用。我们对实例化相关表达式的定义来自 Itanium C++ ABI [ItaniumABI]。

即使类型和常数值在实例化中都是不变的。然而，与实例化相关的表达式在实例化时可能会无效。例如，因为 `sizeof` 不能应用于此类类型，所以使用不完整的类实例化 `maybeDependent()` 将会触发错误。

类型、值和实例化型依赖可以看作一系列泛型的表达式分类。因为表达式的类型会在不同实例化之间变化，它的常量值自然也会在不同实例化之间变化，所以任何依赖于类型的表达式会认为是值依赖型的。类似地，表达式的类型或值在每次实例化中都不同，在某种程度上依赖于模板参数，因此类型依赖型和值依赖型的表达式都是实例化依赖型的。这种关系如图 13.1 所示。

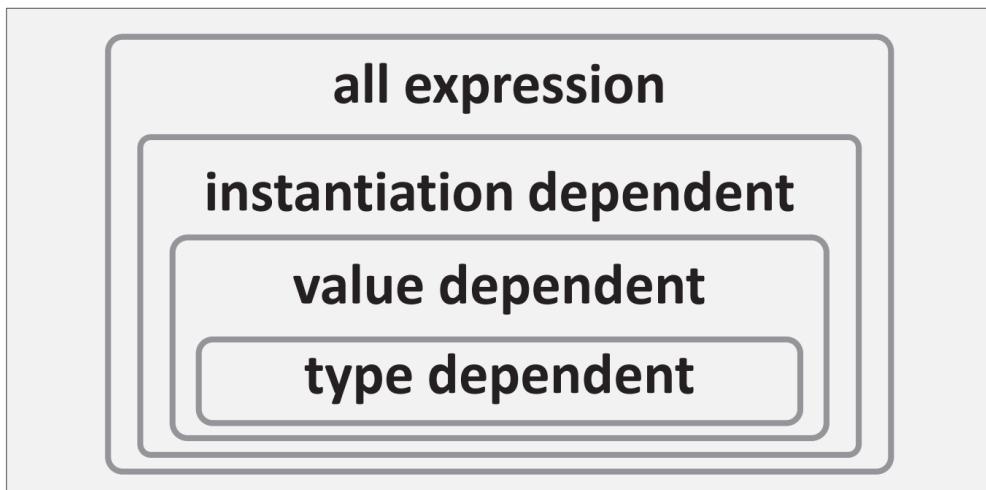


图 13.1. 类型、值和实例化型依赖表达式之间的关系

当从最内部的上下文 (依赖于类型的表达式) 到最外部的上下文时, 模板的更多行为是在模板解析时决定的, 因此不能在一个实例化到下一个实例化时进行变化。若 $f(x)$ 中的 x 依赖于类型, 那么 f 依赖于名称, 需要进行两阶段查找 (第 14.3.1 节), 而若 x 依赖于值但不依赖于类型, 那么 f 是非依赖名称, 在解析模板时可以完全确定其查找名称。

13.3.7 编译错误

当所有模板实例都会产生错误时, C++ 编译器允许 (但不是必需的!) 在解析模板时诊断错误。让我们扩展上一节中的 $f(x)$ 例子来进一步探讨这个问题:

```

1 void f() { }
2
3 template<int x> void nondependentCall()
4 {
5     f(x); // x is value-dependent, so f() is nondependent;
6     // this call will never succeed
7 }
```

这里, 调用 $f(x)$ 在每个实例中都会产生错误, 因为 f 是一个非依赖名称, 而且唯一可见的 f 不接受参数。C++ 编译器在解析模板时会产生错误, 或者会等到第一个模板实例化时才会产生错误: 即使在这个简单的例子中, 常用编译器的表现也会有所不同。可以构造相似的例子: 表达式是实例依赖的, 但并不是值依赖的。

```

1 template<int N> void instantiationDependentBound()
2 {
3     constexpr int x = sizeof(N);
4     constexpr int y = sizeof(N) + 1;
5     int array[x - y]; // array will have a negative size in all instantiations
6 }
```

13.4. 派生和类模板

类模板可以继承或被继承，模板和非模板场景之间没有显著的区别。然而，当从依赖名称引用的基类派生类模板时，有一个微妙区别。让我们首先看看非依赖型基类的情况。

13.4.1 非依赖型基类

类模板中，非依赖型基类是具有完整类型的类，可以在不知道模板参数的情况下即可确定的基类。换句话说，这个基类的名称是使用非依赖型名称表示的。

```
1 template<typename X>
2 class Base {
3     public:
4         int basefield;
5         using T = int;
6     };
7
8 class D1: public Base<Base<void>> { // not a template case really
9     public:
10    void f() { basefield = 3; } // usual access to inherited member
11 };
12
13 template<typename T>
14 class D2 : public Base<double> { // nondependent base
15     public:
16    void f() { basefield = 7; } // usual access to inherited member
17    T strange; // T is Base<double>::T, not the template parameter!
18 };
```

模板中的非依赖型基类类似于普通非模板类中的基类，但当在模板派生中查找非限定名称时，非依赖型基类会优先考虑该名称，而后才是模板参数列表。类模板 D2 的成员 strange 总是对应 Base<double>::T(就是 int) 的类型 T。例如，下面的函数是无效的：

```
1 void g (D2<int*>& d2, int* p)
2 {
3     d2.strange = p; // ERROR: type mismatch!
4 }
```

这有些违反直觉，并且要求派生模板的编写者知道派生自的非依赖型基类中的名称——即使该派生是间接的或名称是私有的。将模板参数放在“模板化”的范围中可能会更好。

13.4.2 依赖型基类

前面的例子中，基类完全确定，不依赖于模板参数。只要知道模板定义，C++ 编译器就可以在这些基类中查找非依赖性名称。另一种方法（C++ 标准不允许）是延迟对这些名称的查找，直到模板实例化时再进行查找。这种方法的缺点是，由于缺少符号而导致的错误消息延迟至实例化。因此，C++ 标准在模板中出现的非依赖名称时就立即查找。看看下面的例子：

```
1 template<typename T>
```

```

2 class DD : public Base<T> { // dependent base
3     public:
4         void f() { basefield = 0; } // #1 problem...
5     };
6
7 template<> // explicit specialization
8 class Base<bool> {
9     public:
10        enum { basefield = 42 }; // #2 tricky!
11    };
12
13 void g (DD<bool>& d)
14 {
15     d.f(); // #3 oops?
16 }

```

#1 找到了对非依赖名称 `basefield` 的引用，立即查找它。假设在 `Base` 模板中查找，并将它绑定到其中找到的 `int` 成员上。此之后不久，我们使用显式特化重写了 `Base` 的定义。恰巧，这种特化改变了已经提交的 `basefield` 成员的含义！因此，在 #3 实例化 `DD::f` 的定义时，发现太急于在 #1 绑定非依赖名称。`DD<bool>` 中没有在 #2 特化 `basefield`，并且产生错误消息。

为了规避这个问题，标准 C++ 在依赖型基类中不会查找非依赖名称（但在遇到时仍然会查找）。

这是两阶段查找规则的一部分，用来区分第一次看到模板定义时的第一阶段和模板实例化时的第二阶段（参见第 14.3.1 节）。

因此，标准 C++ 编译器将在 #1 发出错误信息。要纠正代码，使名称 `basefield` 具有依赖性就好，因为依赖型名称只能在实例化时查找，而那时必须查找的具体基类实例类型就明朗了。编译器在 #3 将知道 `DD<bool>` 的基类是 `Base<bool>`，并且这已经显式特化了。这种情况下，推荐的方式就是让名称转成依赖型：

```

1 // Variation 1:
2 template<typename T>
3 class DD1 : public Base<T> {
4     public:
5         void f() { this->basefield = 0; } // lookup delayed
6     };

```

另一种方法是使用限定名引入依赖：

```

1 // Variation 2:
2 template<typename T>
3 class DD2 : public Base<T> {
4     public:
5         void f() { Base<T>::basefield = 0; }
6     };

```

使用这种解决方案必须小心，若使用非限定的非依赖名称来形成虚函数调用，那么限定会抑制虚函数调用机制，程序的含义也会发生变化。在某些情况下，当遇到第 2 种解决方案不适用的情况，

可以使用方案 1:

```
1 template<typename T>
2 class B {
3     public:
4     enum E { e1 = 6, e2 = 28, e3 = 496 };
5     virtual void zero(E e = e1);
6     virtual void one(E& e);
7 };
8
9 template<typename T>
10 class D : public B<T> {
11     public:
12     void f() {
13         typename D<T>::E e; // this->E would not be valid syntax
14         this->zero(); // D<T>::zero() would inhibit virtuality
15         one(e); // one is dependent because its argument
16     } // is dependent
17 };
```

这个例子中是如何用 `D<T>::E` 替换 `B<T>::E` 的。这种情况下，二者皆可。然而，在多重继承的情况下，可能不知道哪个基类提供了所需的成员（使用派生类进行限定），或者多个基类可能声明相同的名称（必须使用特定的基类名称来消除歧义）。

注意，调用 `one(e)` 中的名称 `one` 依赖于模板参数，因为调用的显式参数之一是类型依赖型的。因为编译器在确定查找之前无法验证默认参数，所以对于依赖于模板形参的类型，隐式使用的默认参数无效。为了避免这些问题，我们更倾向于在允许的情况下使用 `this->` 前缀——同样适用于非模板代码。

如果发现重复的限制使代码陷入混乱，可以从派生类中的依赖型基类中引入一个名称，从而一劳永逸：

```
1 // Variation 3:
2 template<typename T>
3 class DD3 : public Base<T> {
4     public:
5     using Base<T>::basefield; // #1 dependent name now in scope
6     void f() { basefield = 0; } // #2 fine
7 };
```

#2 的查找成功并找到 #1 的使用的声明。然而，`using` 声明直到实例化时才确定。这个方案有一些限制，例如：如果派生了多个基类，开发者必须准确地选择包含所需成员的基类。

在当前实例化中搜索限定名称时，C++ 标准指定在当前实例化和所有非依赖型基类中首先搜索名称查找，类似于对该名称执行限定查找的方式。若找到，则限定名称将引用当前实例化的成员，而不是依赖于名称。

但在实例化模板时仍然要重复查找，若在该上下文中产生了不同的结果，则该程序的格式不正确。

若没有找到这样的名称，并且类还有其他的依赖型基类，那么受限名称就会指代未知特化实例的某个成员。例如：

```
1 class NonDep {
2     public:
3     using Type = int;
4 };
5
6 template<typename T>
7 class Dep {
8     public:
9     using OtherType = T;
10};
11
12 template<typename T>
13 class DepBase : public NonDep, public Dep<T> {
14     public:
15     void f() {
16         typename DepBase<T>::Type t; // finds NonDep::Type;
17         // typename keyword is optional
18         typename DepBase<T>::OtherType* ot; // finds nothing; DepBase<T>::OtherType
19         // is a member of an unknown specialization
20     }
21};
```

13.5. 后记

真正用于解析模板定义的第一个编译器是在 20 世纪 90 年代中期，由一家名为 Taligent 的公司开发的。在此之前——甚至几年后——大多数编译器都将模板视为在实例化时才处理的标记。因此，除了找到模板定义结束位置等少许操作以外，没有进行任何解析。撰写本文时，Microsoft Visual C++ 编译器仍然以这种方式工作。爱迪生设计集团 (EDG) 的编译器前端使用了一种混合技术，模板在内部视为一个带注释的标记序列，在需要的模式中执行“通用解析”来验证语法 (EDG 的产品模拟多个其他编译器，可以很好地模拟微软编译器的行为)。

Bill Gibbons 是 Taligent 在 C++ 委员会的代表，其极力主张让模板可以无二义性地进行解析。然而，直到惠普公司完成第一个完整的编译器之后，Taligent 公司的努力才真正产品化，也才有了一个真正编译模板的 C++ 编译器。和其他具有竞争性优点的产品一样，这个 C++ 编译器很快就由于高质量的错误信息而得到业界的认可。模板的错误信息不会总延迟到实例化时才发出，也要归功于这个编译器。

模板的早期开发过程中，Tom Pennello(Metaware 公司的一位著名解析专家)就意识到了尖括号所带来的一个问题。Stroustrup 也对这个话题进行了讨论 [StroustrupDnE]，而且认为人们更喜欢阅读尖括号，而不是圆括号。然而，除了尖括号和圆括号，还存在其他的一些可能性：Pennello 在 1991 年的 C++ 标准大会 (在达拉斯举办) 上特别地提议使用大括号，例如 (List::X)。

使用括号也不是完全没有问题。具体来说，特化类模板的语法需要进行重大的调整。

那时，因为嵌入在其他模板内部的模板(也称为成员模板)还是不合法的，所以问题的影响范围也非常有限，也就不会涉及到 13.3.3 节的问题。最后，委员会拒绝了这个取代尖括号的提议。

13.4.2 节中描述的非依赖型名称和依赖型基类的名称查找规则，是在 1993 年 C++ 标准中引入的。在 1994 年，Bjarne Stroustrup 的 [StroustrupDnE] 首次公开描述了这一规则。然而直到 1997 年惠普才把这一规则引入 C++ 编译器，自那以后出现了大量的派生自依赖型基类的类模板代码。当惠普工程师开始测试该实现时，发现大部分以特殊方式使用模板的代码都无法编译成功了。

幸运的是，在发布新功能之前就发现了。

特别地，STL 的所有实现都打破了这一规则。

具有讽刺意味的是，第一个实现也是由惠普开发的。

考虑到客户的转换代码的成本，对于那些“假定非依赖型名称可以在依赖型基类中进行查找”的代码，惠普弱化了相关的错误信息。例如，对于位于类模板作用域的非依赖型名称，若利用标准原则不能找到该名称，C++ 就会在依赖型基类中进行查找。若仍然找不到，才会给出一个错误而编译失败。然而，若在依赖型基类中找到了该名称，那么就会给出一个警告，对该名称进行标记，并且当成是依赖型名称，然后在实例化时再次查找。

查找过程中，“非依赖型基类中的名称，会隐藏相同名称的模板参数(13.4.1 节)”这一规则显然是一个疏忽，但修改这一规则的建议还没被 C++ 标准委员会所认可。最好的办法就是避免使用非依赖型基类中的名称作为模板参数名称。命名转换对这一类问题都是一种良好的解决方式。

友元注入一度认为是有害的，会使得程序的合法性与实例出现的顺序紧密相关。Bill Gibbons(此时他还在 Taligent 公司开发编译器)就是解决这一问题的最大支持者，因为消除实例顺序依赖性激活了一个新的、有趣的 C++ 开发领域(传闻 Taligent 正在做)。然而，Barton-Nackman 技巧(21.2.1 节)需要友元注入的形式，正是这种特殊的技术使它以基于 ADL 的(弱化)形式保留在语言中。

Andrew Koenig 首次为操作符函数提出了 ADL 查找(这就是为什么有时候 ADL 也称为 Koenig 查找)，动机主要是考虑美观性：“用外围命名空间显式地限定操作符名称”看起来很拖沓(例如，对于 `a+b`，需要这样编写:`N::operator+(a,b)`)，而为每个操作符使用 `using` 声明又会让代码看起来非常笨重。因此，才决定操作符可以在参数关联的命名空间中查找。ADL 随后扩展到普通函数名称的查找，得以容纳有限种类的友元名称注入，并为模板及其实例支持两阶段查找模型(第 14 章)。泛化的 ADL 规则也称作扩展 Koenig 查找。

David Vandevoorde 通过他的论文 N1757 在 C++11 中添加了“尖括号黑客”规范。还通过核心问题 1104 的解决方案添加了“有向图黑客”，以解决美国对 C++11 标准草案的审查要求。

第 14 章 实例化

模板实例化是从泛型模板定义中生成类型、函数和变量的过程。

术语实例化有时也用来指从类型创建对象。在本书中，用来表示模板实例化。

C++ 模板实例化的概念非常基础，但有时又错综复杂。因为，模板生成的实体定义不再局限于源代码的位置。模板本身的位置、模板使用的位置，以及模板参数类型定义的位置均在实体的含义中扮演着重要角色。

本章中，将解释如何组织源代码以正确使用模板。此外，我们调研了主流 C++ 编译器用于处理模板实例化的各种方法。尽管这些方法在语义上应该等价，但理解编译器实例化策略的基本原则是大有裨益的。构建实际的软件时，每种机制都带有一组小怪癖，并且每种机制都影响了标准 C++ 规范。

14.1. 按需实例化

当 C++ 编译器遇到模板特化时，将通过替换模板参数来创建该特化。

术语特化用于一般意义上的实体，是模板的一个特定实例（参见第 10 章），不涉及第 16 章中描述的显式特化机制。

特化是自动完成的，不需要特殊代码（或者模板定义）的指导。这种按需变化的实例化特性将 C++ 模板与其他早期编译语言不同（如 Ada 或 Eiffel；一些语言需要显式的实例化指令，而其他语言则使用运行时分发机制来避免实例化）。这有时也称为隐式或自动实例化。

按需实例化要求编译器在使用时，需要经常访问模板及其部分成员的完整定义（不仅仅是声明）。看看下面的示例：

```
1 template<typename T> class C; // #1 declaration only
2
3 C<int>* p = 0; // #2 fine: definition of C<int> not needed
4 template<typename T>
5
6 class C {
7     public:
8         void f(); // #3 member declaration
9     }; // #4 class template definition completed
10
11 void g (C<int>& c) // #5 use class template declaration only
12 {
13     c.f(); // #6 use class template definition;
14 } // will need definition of C::f()
15
16 // in this translation unit
17 template<typename T>
18 void C<T>::f() // required definition due to #6
```

#1 处只有模板的声明是可用的，而不是定义（这种声明称为前置声明）。与普通类的情况一样，不需要类模板的定义来声明指向该类型的指针或引用，就像 #2 那样。例如，函数 g() 的参数类型不需要模板 C 的完整定义。但当组件需要知道模板特化的大小时，或者访问类的特化成员时，整个类模板定义就必须可见。这解释了为什么在源代码中 #6，必须看到类模板定义；否则，编译器无法验证该成员是否存在，以及是否可访问（不是 private 或 protected）。此外，也需要成员函数的定义，因为 #6 的调用需要 C<int>::f()。

下面是另一个表达式，因为需要 C<void> 的大小，所以需要实例化之前的类模板：

```
1 C<void>* p = new C<void>;
```

这种情况下需要实例化，以便编译器可以确定 C<void> 的大小，new 表达式需要确定分配多少存储空间。对于这个特定的模板，替代 T 的参数 X 的类型不会影响模板的大小，因为 C<X> 是一个空类。但编译器不需要分析模板定义来避免实例化（编译器都会执行实例化）。此外，本例中还需要实例化来确定 C<void> 是否具有可访问的默认构造函数，并确保 C<void> 不会声明成员操作符 new 或 delete。

访问类模板成员在源代码中并不总是显式可见，例如：C++ 重载解析要求对候选函数的参数类型可见：

```
1 template<typename T>
2 class C {
3 public:
4     C(int); // a constructor that can be called with a single parameter
5 }; // may be used for implicit conversions
6
7 void candidate(C<double>); // #1
8 void candidate(int) {} // #2
9
10 int main()
11 {
12     candidate(42); // both previous function declarations can be called
13 }
```

调用 candidate(42) 时，将解析为在 #2 处的重载声明。然而，#1 的声明也可以实例化，以检查是否是匹配的候选（因为单参数构造函数可以隐式地将 42 转换为类型 C<double> 的右值）。如果编译器可以在没有实例化的情况下解析调用，则允许（但不是必需）执行此实例化（本例可能就是这种情况，因为在精确匹配上不会选择隐式转换）。还要注意，C<double> 的实例化可能会触发错误。

14.2. 延迟实例化

到目前为止所展示的这些例子，和使用非模板类相比并没有本质上的区别，例如：需要完整的类类型（参见第 10.3.1 节）。对于模板的情况，编译器将根据类模板定义生成这个完整的定义。

那模板实例化程度如何？一个模糊的答案是：达到需要的程度即可。换句话说，编译器在实例化模板时是“延迟的”。来看看这种懒惰到底意味着什么。

14.2.1 部分实例化和完全实例化

正如所看到的，编译器有时不需要替换类或函数模板的完整定义。例如：

```
1 template<typename T> T f (T p) { return 2*p; }
2 decltype(f(2)) x = 2;
```

本例中，由 `decltype(f(2))` 表示的类型不需要函数模板 `f()` 的完整实例化。因此，编译器允许替换 `f()` 的声明，而不允许替换它的“主体”。这有时称为部分实例化。

如果引用类模板的实例，而不需要该实例是完整类型，编译器不执行该类模板实例的完整实例化。看看下面的例子：

```
1 template<typename T> class Q {
2     using Type = typename T::Type;
3 };
4
5 Q<int>* p = 0; // OK: the body of Q<int> is not substituted
```

这里，因为 `T::Type` 在 `T` 为 `int` 时没有意义，所以 `Q<int>` 的实例化将触发一个错误。在这个例子中 `Q<int>` 不需要完整定义，所以没有执行完整的实例化。

变量模板也有“完整”和“部分”实例化的区别。下面的例子说明了这一点：

```
1 template<typename T> T v = T::default_value();
2 decltype(v<int>) s; // OK: initializer of v<int> not instantiated
```

`v<int>` 的完整实例化会引起错误，但只是需要变量模板实例类型的话，不需要进行完整实例化。有趣的是，别名模板没有这种区别。

C++ 谈到“模板实例化”而不确定是完整还是部分实例化时，通常说的是“完整实例化”。也就是说，实例化在默认情况下是完全实例化。

14.2.2 实例化组件

当类模板隐式(完全)实例化时，其成员的声明也会实例化，但相应的定义不会(即成员部分实例化)，当然也有一些例外。首先，若类模板包含匿名联合，则该联合定义的成员也会实例化。

匿名联合在这方面总是特别的：其成员可以认为是外围类的成员。匿名联合是一种表示某些类成员共享相同存储空间的结构。

另一个异常发生在虚成员函数中，其定义可以作为类模板实例化的结果而实例化，也可以不实例化。许多实现将实例化该定义，因为支持虚函数调用机制的内部结构，要求虚函数实际上作为可链接实体存在。

实例化模板时，默认函数调用参数会单独处理。除非调用了实际使用默认参数的函数(或成员函数)，否则不会对其实例化。另外，如果函数调用时带有覆盖默认参数的显式参数，不会实例化默认参数。

类似地，通常不会实例化异常规范和默认成员初始化器。

结合一些例子来说明这些原则：

details/lazy1.hpp

```

1 template<typename T>
2 class Safe {
3 };
4
5 template<int N>
6 class Danger {
7     int arr[N]; // OK here, although would fail for N<=0
8 };
9
10 template<typename T, int N>
11 class Tricky {
12     public:
13     void noBodyHere(Safe<T> = 3); // OK until usage of default value results in an error
14     void inclass() {
15         Danger<N> noBooMYet; // OK until inclass() is used with N<=0
16     }
17     struct Nested {
18         Danger<N> pfew; // OK until Nested is used with N<=0
19     };
20     union { // due anonymous union:
21         Danger<N> anonymous; // OK until Tricky is instantiated with N<=0
22         int align;
23     };
24     void unsafe(T (*p) [N]); // OK until Tricky is instantiated with N<=0
25     void error() {
26         Danger<-1> boom; // always ERROR (which not all compilers detect)
27     }
28 };

```

标准 C++ 编译器会检查这些模板定义，来检查语法和语义约束。在当检查涉及模板参数的约束时，将其“认为是最佳情况”。例如，成员 Danger::arr 中的参数 N 可以是零或负的（这是无效的），但假定情况并非如此。

一些编译器，如 GCC，允许零长度数组作为扩展，因此即使 N 为 0，也没问题。

因此，inclass()、嵌套 struct 和匿名联合的定义都不是问题。

只要 N 是一个未替换的模板参数，成员的不安全声明 (T (*p)[N]) 也没有问题。

因为模板 Safe<> 不能用整数初始化，所以成员 noBodyHere() 声明的默认参数 specification(=3) 看起来很诡异，但假设要么默认参数不需要用 Safe<T> 的泛型定义，要么 Safe<T> 将特化（参见第 16 章）以使用整数值初始化。然而，即使模板没有实例化，成员函数 error() 也会为其定义一个错误。因为使用 Danger<-1> 需要完整地定义 Danger<-1> 类，而生成该类会导致定义一个负大小的数组。有趣的是，虽然标准明确声明此代码无效，但也允许编译器在没有实际使用模板实例时不报错误。由于 Tricky<T,N>::error() 不用于具体的 T 和 N，因此编译器不需要为此发出错误。在编写本文时，GCC 和 Visual C++ 都不会对此进行报错。

现在来分析一下当添加以下定义时会发生什么：

```
1 Tricky<int, -1> inst;
```

这会导致编译器在模板 `Tricky<T>` 的定义中用 `int` 替换 `T`, 用`-1` 替换 `N`, 从而(完全)实例化 `Tricky<int, -1>`。不是所有成员定义都需要, 但默认构造函数和析构函数(隐式声明)肯定会调用, 因此它们的定义必须以某种方式可用。如上所述, `Tricky<int, -1>` 的成员会部分实例化(也就是声明被替换):这个过程可能会导致错误。例如, 不安全的(`T (*p)[N]`) 声明会创建一个负数的数组类型(这是错误的)。类似地, 因为类型 `Danger<-1>` 无法完成, 成员 `anonymous` 会触发错误。相比之下, 成员 `class()` 和 `struct Nested` 的定义还没有实例化, 因此在完整类型 `Danger<-1>`(包含前面讨论的无效数组)实例化时不会发生错误。

实例化模板时, 实际上也应该提供虚成员的定义。否则, 可能会发生链接错误。例如:

`details/lazy2.cpp`

```
1 template<typename T>
2 class VirtualClass {
3     public:
4     virtual ~VirtualClass() {}
5     virtual T vmem(); // Likely ERROR if instantiated without definition
6 };
7
8 int main()
9 {
10    VirtualClass<int> inst;
11 }
```

最后, 注意 `operator->`:

```
1 template<typename T>
2 class C {
3     public:
4     T operator-> ();
5 };
```

通常, `operator->` 必须返回指针类型或其他类类型。这样, 因为声明了 `operator->` 的返回类型为 `int`, `C<int>` 的补全会触发一个错误。但某些类模板定义触发了这些类型(返回类型为 `T` 或者 `T*`)的定义, 所以语言规则更灵活。

典型的例子是智能指针模板(例如, `std::unique_ptr<T>`)

只有在重载解析规则确实选择了用户自定义的 `operator->` 时, 才要求自定义 `operator->` 只能返回应用其他(内置)`operator->` 的类型。这甚至对模板之外的代码也同样有效(这种松弛法则(relaxed behavior)在上下文中用处不大)。因此, 尽管 `int` 会替代该返回类型, 但这里的声明不会触发错误。

14.3. C++ 实例化模型

模板实例化是通过替换模板参数, 从对应的模板实体获得类型、函数或变量的过程。这听起来挺简单, 但需要确定许多细节。

14.3.1 两阶段查找

第 13 章中，看到在解析模板时不能解析依赖名称。相反，它们会在实例化时再次进行查找。但是，非依赖名称会提前查找，以便在模板第一次出现时就能诊断出错误。这就引出了两阶段查找的概念：

除了两阶段查找外，有时还使用了两阶段查找或两阶段名称查找等术语进行描述。

第一个阶段是解析模板，第二个阶段是实例化模板：

1. 解析模板时，使用普通查找规则和参数依赖查找规则（如果适用）来查找非依赖名称。使用普通查找规则查找非限定的依赖名称（它们是依赖的，因为看起来像带有参数型依赖的函数调用的函数名），但是在第二阶段（模板实例化时）再次进行查找前，不会采纳第一次查找的结果。
2. 第二阶段中，当在实例化点（POI）处实例化模板时，将查找相关的限定名（使用该特定实例化的模板参数），并为在第一阶段使用普通查找查找到的非限定名称型依赖执行 ADL。

对于非限定的名称型依赖，初始的普通查找（不是完整的）用于确定该名称是否为模板。考虑下面的例子：

```
1 namespace N {  
2     template<typename> void g() {}  
3     enum E { e };  
4 }  
5  
6 template<typename> void f(T p) {}  
7  
8 template<typename T> void h(T p) {  
9     f<int>(p); // #1  
10    g<int>(p); // #2 ERROR  
11 }  
12  
13 int main() {  
14     h(N::e); // calls template h with T = N::E  
15 }
```

在 #1 行中，看到名称 `f` 后面跟着一个 < 时，编译器必须判断 < 是尖括号还是小于号。这取决于是否知道 `f` 是模板的名称；在本例中，普通查找查找 `f` 的声明，它实际上是一个模板，因此使用尖括号解析成功。

但是，第 #2 行产生了一个错误，因为使用普通查找没有找到模板 `g`；因此，< 视为小于号，这在本例中是一个语法错误。如果能克服这个问题，会在为 `T = N::E` 实例化 `h` 时，使用 ADL 找到模板 `N::g`（因为 `N` 是与 `E` 关联的名称空间）。但只有成功地解析 `h` 的泛型定义，才能进行这些操作。

14.3.2 实例化点

在模板源代码中有一些点，C++ 编译器必须访问模板实体的声明或定义。当代码构造引用模板特化时，需要实例化相应模板的定义来创建该特化，就会创建实例化点（POI）。POI 是源中的一个点，在这里可以插入替换模板。例如：

```

1 class MyInt {
2     public:
3     MyInt(int i);
4 }
5
6 MyInt operator - (MyInt const&);
7
8 bool operator > (MyInt const&, MyInt const&);
9
10 using Int = MyInt;
11
12 template<typename T>
13 void f(T i)
14 {
15     if (i>0) {
16         g(-i);
17     }
18 }
19 // #1
20 void g(Int)
21 {
22     // #2
23     f<Int>(42); // point of call
24     // #3
25 }
26 // #4

```

C++ 编译器看到调用 `f<int>(42)` 时，就知道模板 `f` 需要实例化，以便用 `MyInt` 替换 `T`: 创建一个 POI。#2 和 #3 非常接近调用点，但它们不是 POI，因为 C++ 不允许在那里插入`::f<int>(Int)` 的定义。#1 和 #4 之间的本质区别是，#4 上函数 `g(Int)` 是可见的，因此可以解析依赖于模板的调用 `g(-i)`。然而，如果 #1 是 POI，因为 `g(Int)` 还不可见，则该调用无法解析。幸运的是，C++ 将函数模板特化引用的 POI，定义在包含该引用最近的名称空间作用域声明或定义之后，例子中是 #4。

为什么这个示例是 `MyInt` 类型，而不是 `int`? 在 POI 执行的第二个查找只是一个 ADL。因为 `int` 没有关联的名称空间，所以 POI 查找不会发生，也不会找到函数 `g`。

因此，如果将 `Int` 的类型别名声明替换为

```

1 using Int = int;

```

前面的示例将不再编译。下面的例子也存在类似的问题:

```

1 template<typename T>
2 void f1(T x)
3 {
4     g1(x); // #1
5 }
6
7 void g1(int)
8 { }

```

```

9
10 int main()
11 {
12     f1(7); // ERROR: g1 not found!
13 }
14 // #2 POI for f1<int>(int)

```

调用 `f1(7)` 为 `f1<int>(int)` 在 `main()` 之外的 #2 创建一个 POI。在这个实例化中，关键问题是函数 `g1` 的查找。当第一次遇到模板 `f1` 的定义时，注意到非限定 `g1` 是名称型依赖，因此具有依赖参数的函数调用中的函数名（参数 `x` 的类型取决于模板参数 `T`）。因此，`g1` 在 #1 使用普通查找规则查找，此时 `g1` 还不可见。在 #2 函数在相关的名称空间和类中再次查找，但是唯一的参数类型是 `int`，并且没有相关的名称空间和类。因此，即使在 POI 上普通查找可以找到 `g1`，也找不到 `g1`。

变量模板实例化点与函数模板的处理类似。

在撰写本文时，标准中并没有明确规定。然而，估计这也不会成为一个有争议的问题。

对于类模板特化，情况不同：

```

1 template<typename T>
2 class S {
3     public:
4     T m;
5 };
6 // #1
7 unsigned long h()
8 {
9     // #2
10    return (unsigned long) sizeof(S<int>);
11    // #3
12 }
13 // #4

```

同样，函数作用域 #2 和 #3 不能是 POI，因为命名空间作用域类 `S<int>` 的定义不能出现在那（模板不能出现在函数作用域中）。

泛型 Lambda 的调用操作符是一个例外。

如果遵循函数模板实例的规则，POI 将在 #4 设置，但是表达式 `sizeof(S<int>)` 无效，因为 `S<int>` 的大小在点 #4 之前不确定。因此，对生成类实例引用的 POI 定义为，紧接最近的名称空间作用域声明或定义之前的点，该声明或定义包含对实例的引用。例子中就是 #1。

当模板实际实例化时，可能会出现其他实例化的需求。举一个简单的例子：

```

1 template<typename T>
2 class S {
3     public:
4     using I = int;
5 };

```

```

6
7 // #1
8 template<typename T>
9 void f()
10 {
11     S<char>::I var1 = 41;
12     typename S<T>::I var2 = 42;
13 }
14
15 int main()
16 {
17     f<double>();
18 }
19 // #2 : #2a , #2b

```

前面的讨论已经建立了 `f<double>()` 的 POI 在 #2 的基础上。函数模板 `f()` 也引用了类特化 `S<char>`, 其 POI 位于 #1。它也引用 `S<T>`, 但因为这仍然有依赖, 不能在这里实例化它。然而, 如果在 #2 实例化 `f<double>()`, 还需要实例化 `S<double>` 的定义。这种次要或可传递的 POI 定义略有不同。对于函数模板, 次要 POI 与主 POI 完全相同。对于类实体, 次要 POI 位于主 POI 之前(在最近的封闭命名空间范围内)。例子中, `f<double>()` 的 POI 可以放置在 #2b, 在它之前的点 (#2a) 是 `S<double>` 的辅助 POI, 这与 `S<char>` 的 POI 完全不同。

翻译单元通常包含同一个实例的多个 POI。对于类模板实例, 只保留每个翻译单元中的第一个 POI, 而忽略后续的 POI(实际上不认为是 POI)。对于函数和变量模板实例, 将保留所有的 POI。ODR 要求在保留的 POI 上发生的实例化都等价, 但 C++ 编译器不需要验证和诊断是否会违反该规则。这允许 C++ 编译器只选择一个非类 POI 来执行实际的实例化即可, 而不用担心另一个 POI 可能导致不同的实例化。

实际中, 大多数编译器将大多数函数模板的实例化延迟到翻译单元的末尾。有些实例化不能延迟, 包括需要实例化来确定导出返回类型的情况(参见 15.10.1 节和 15.10.4 节), 以及 `constexpr` 函数, 必须求值以产生常量结果的情况。一些编译器在首次用于内联调用时会立刻实例化内联函数。

现代编译器中, 可调用的内联通常是由编译器中一个主要独立于语言的专业优化组件(“后端”或“中端”)来处理的。然而, 早期设计的 C++ “前端”(C++ 编译器中特定于 C++ 的部分)也可能具有内联扩展调用的能力, 因为旧的后端在考虑内联扩展调用时过于保守。

C++ 标准允许的替代 POI, 这可将对应模板特化的 POI 移动到翻译单元的末尾。

14.3.3 包含模型

无论何时遇到 POI, 都必须以某种方式访问相应模板的定义。对于特化类, 类模板定义必须在翻译单元中更早看到。对于函数和变量模板(以及成员函数和类模板的静态数据成员)的 POI, 这也是需要的。通常模板定义会添加到包含在翻译单元中的头文件中, 即使是非类型模板。这个模板定义的源模型称为包含模型, 是当前 C++ 标准支持模板的自动源模型。

C++98 标准也提供了一个分离模型。但从未流行起来，在发布 C++11 标准之前就删除了。

尽管包含模型鼓励程序员将所有的模板定义放在头文件中，这样就可以满足可能出现的 POI，也可以使用显式的实例化声明和显式的实例化定义来显式地管理实例化（参见 14.5 节）。大多数时候开发者更喜欢依赖于自动实例化的机制。使用自动模式实现的挑战是处理不同翻译单元具有函数或变量模板（或类模板实例的相同成员函数或静态数据成员）的相同特化 POI 的可能性。接下来我们讨论这个问题的解决方法。

14.4. 实现方案

本节中，将回顾 C++ 实现支持包含模型的方式。所有这些实现都依赖于两个经典组件：编译器和链接器。编译器将源代码转换为目标文件，目标文件包含带有符号注释的机器码（交叉引用其他目标文件和库）。链接器通过组合目标文件并解析包含的符号交叉引用来创建可执行程序或库。接下来的内容中，假设有这样一个模型，尽管完全可能（但不流行）以其他方式实现 C++。例如，可以想象这样一个 C++ 解释器。

当在多个翻译单元中使用类模板特化时，编译器将在每个翻译单元中重复实例化过程。这没什么问题，因为类定义不会直接创建低层代码。它们仅在 C++ 实现内部使用，用于验证和解释各种其他表达式和声明。这方面，类定义的多个实例化与类定义的多个包含（通常是通过包含头文件）在各种翻译单元中没有本质区别。

但若实例化一个（非内联）函数模板，情况可能会有所不同。若要为普通的非内联函数提供多个定义，就会违反 ODR。例如，编译并链接一个由以下两个文件组成的程序：

```
1 // a.cpp:  
2 int main()  
3 { }  
4  
5 // b.cpp:  
6 int main()  
7 { }
```

C++ 编译器会分别编译模块不会有问题，因为确实是有效的 C++ 单元。但若试图将两者链接在一起，链接器很可能会抗议——不允许重复定义。

相比之下，考虑模板的情况：

```
1 // t.hpp:  
2 // common header (inclusion model)  
3 template<typename T>  
4 class S {  
5     public:  
6         void f();  
7 };  
8  
9 template<typename T>  
10 void S::f() // member definition  
11 { }
```

```

12 void helper(S<int>* );
13
14 // a.cpp:
15 #include "t.hpp"
16 void helper(S<int>* s)
17 {
18     s->f(); // #1 first point of instantiation of S::f
19 }
20
21 // b.cpp:
22 #include "t.hpp"
23 int main()
24 {
25     S<int> s;
26     helper(&s);
27     s.f(); // #2 second point of instantiation of S::f
28 }

```

如果链接器像处理普通函数或成员函数那样，处理类模板的成员函数实例化，编译器需要确保它只在两个 POI 中的一个生成代码:#1 或 #2。为了实现这一点，编译器必须将信息从一个翻译单元传递到另一个翻译单元，而在引入模板之前，C++ 编译器从来接到过这样的要求。在接下来的内容中，将讨论 C++ 实现者的三大类解决方案。

模板实例化产生的所有可链接实体，都会出现同样的问题：实例化的函数模板和成员函数模板，以及实例化的静态数据成员和实例化的变量模板。

14.4.1 贪婪实例化

第一个普及贪婪实例化的 C++ 编译器是来自一家名为 Borland 的公司。目前为止，已经发展成为各种 C++ 系统中最常用的技术。

贪婪实例化假定链接器知道某些实体（特别是可链接的模板实例化），可能在各种对象文件和库中重复出现。编译器通常会以一种特殊的方式标记这些实体。当链接器发现多个实例时，会保留一个并丢弃所有其他实例。

理论上，贪婪实例化有一些严重缺陷：

- 编译器会浪费时间来生成和优化 N 个实例，其中只有一个会保留。
- 链接器通常不会检查两个实例是否相同，因为对于一个模板特化的多个实例，生成的代码中可能会出现一些无关紧要的差异。这些小的差异不会导致链接器失败。（这些差异可能是由于实例化时编译器状态的微小差异造成的。）然而，这也常常导致链接器没有注意到更实质性的差异，例如：实例化是用严格的浮点数学规则编译的，而另一个实例化是用宽松的、性能更高的浮点数学规则编译的。

然而，目前的系统已经发展到可以检测某些其他差异。例如，如果一个实例化有相关的调试信息，而另一个实例化没有，就会报告。

- 所有目标文件的总和可能会比替代文件大得多，因为相同的代码可能会复制多次。

这些缺点似乎并没有造成重大问题，也许这是因为贪婪实例化的优势：保留了传统的源对象依赖关系。特别地，一个翻译单元只生成一个目标文件，每个目标文件包含对应源文件（包含实例化的定义）中所有可链接定义的编译代码。另一个重要的好处是，所有函数模板实例都是内联的候选对象，而无需求助于“链接时”优化机制（而且，函数模板实例通常受益于内联函数）。其他实例化机制专门处理内联函数模板实例，以确保它们可以内联扩展。并且，贪婪实例化允许非内联函数模板实例进行内联扩展。

最后，允许可链接实体的重复定义的链接机制，也用于处理重复溢出的内联函数

当编译器无法“内联”对使用关键字 `inline` 标记的函数的每次调用时，将在目标文件中产生函数的副本。这可能发生在多个目标文件中。

以及虚函数调度表。

虚函数调用通常通过函数指针表作为间接调用实现。参见 [LippmanObjMod] 了解 C++ 实现方面的深入研究。

如果这种机制不可用，可以使用内部链接来发出这些项，但代价是生成更大的代码。内联函数只有一个地址的要求，使得以符合标准的方式实现该替代变得困难重重。

14.4.2 查询实例化

20世纪90年代中叶，一家名为 Sun Microsystems 的公司

Oracle 公司后来收购了 Sun Microsystems 公司。

发布了一个重新实现的 C++ 编译器（4.0 版本），为实例化问题提供了一个新的、有趣的解决方案，我们称之为“查询实例化”。其在概念上非常简单和优雅，但它是这里回顾的最新实例化方案。这种方案中，维护由所有参与程序翻译单元编译共享的数据库。这个数据库会跟踪哪些特化已经实例化，以及它们依赖的源码。生成的特化本身会与此信息一起存储在数据库中。每当遇到可链接实体的实例化点时，可能会发生以下三种情况：

1. 没有可用的特化：将发生实例化，并将特化结果输入到数据库中。
2. 特化是可用的，但已经过时，因为生成后源就进行了更改。这里也产生了实例化，但是所产生的特化取代了以前存储在数据库中的特化。
3. 数据库中记录了最新的特化类型。

虽然概念上很简单，但这种设计在实现上有一些挑战：

- 正确地维护与源代码状态相关数据库内容的依赖关系，并不是一件容易的事情。尽管将第三种情况误认为第二种情况并没有错，但这样做会增加编译器的工作量（也会增加整个构建时间）。
- 并发编译多个源文件是很常见的。因此，编译器实现需要为数据库提供适当数量的并发控制。

尽管存在一些挑战，但该方案可以有效地实施。此外，没有明显的病态的前提会使这个解决方案的扩展性很差，相反，贪婪实例化可能会导致大量的无效工作。

数据库的使用也会给开发者带来问题，其根源在于：从大多数 C 编译器继承的传统编译模型不再适用，单个翻译单元不再产生独立的目标文件。假设希望链接最终程序，这个链接操作不仅需要与各种翻译单元相关联的每个目标文件，还需要存储在数据库中的目标文件。类似地，如果创建二进制库，则需要确保创建该库的工具（链接器或归档器）知道数据库内容。任何对目标文件进行操作的工具，都需要知道数据库的内容。通过不将实例化存储在数据库中，而是在生成初始化的对象文件中产生目标代码，可以缓解其中的问题。

库是另一个挑战。许多生成的特化可以打包在一个库中，当库添加到另一个项目时，可能需要让该项目的数据库知道可用的实例化。如果不可用，并且项目为库中存在的特化创建了实例化点，则可能会重复的进行实例化。处理这种情况的一个策略是使用与允许贪婪实例化相同的链接器技术：让链接器知道生成的特化，并让它清除重复（发生的频率应该比贪婪实例化低得多）。源文件、目标文件和库的各种其他安排可能会导致其他问题，比如：由于包含所需实例化的目标代码，没有链接到最终的可执行程序中，而丢失实例化信息。

最终，查询实例化在市场上无法生存，甚至 Sun 的编译器现在也在使用贪婪实例化。

14.4.3 迭代实例化

第一个支持 C++ 模板的编译器是 Cfront 3.0，是 Bjarne Stroustrup 为开发该语言而编写的编译器后代。

不要认为 Cfront 是一个学术原型：它用于工业环境，并成为了许多商业 C++ 编译器产品的基础。其 3.0 版本于 1991 年发布，但漏洞百出。此后，版本 3.0.1 很快就出现了，并支持模板。

Cfront 的一个约束是必须能够在不同平台之间移植，这意味着（1）使用 C 语言作为跨所有目标平台的通用目标表示，（2）使用本地目标链接器。这样链接器看不到模板，Cfront 像普通 C 函数一样生成模板的实例化，因此必须避免重复的实例化。虽然 Cfront 源模型与标准包含模型不同，但它的实例化策略可以适应包含模型。因此，它是迭代实例化的第一个实现。Cfront 迭代可以这样描述：

1. 编译源代码时不实例化必需的可链接特化。
2. 使用预链接器链接目标文件。
3. 预链接器调用链接器并解析其错误消息，以确定是否有错误消息是缺少实例化的结果。如果缺少实例化，预链接器将在包含所需模板定义的源上调用编译器，并带有生成缺失实例化的选项。
4. 若生成了定义，则重复步骤 3。

第 3 步中，在实例化一个可链接实体过程中，可能会要求“另一个仍未实例化”的实体进行实例化；最后，所有的迭代都已经完成，链接器才会成功创建一个完整的程序。

原始的 Cfront 方案的缺点相当严重：

- 感知到的链接时间不仅会因为预链接器的开销而增加，而且还会因为每次重新编译和重新链接而增加。有时使用 Cfront 系统的用户会抱怨说：“链接时间往往需要几天”，而同样的工作，

采用其他解决方案，一个小时左右就足够了。

- 诊断(错误、警告)延迟到连接时。当链接大型程序时，并且开发人员必须等待数小时才能找到模板定义中的错误时，这就特别的痛苦。
- 必须记住包含特定定义的源的位置(步骤1)。特别是Cfront使用了中央存储库，必须面对在查询实例化方法中处理中央数据库的一些挑战。特别是，最初Cfront实现并不支持并发编译。

随后，爱迪生设计集团(EDG)和惠普的aC++对迭代原理进行了改进，

惠普的aC++是在一家名为Taligent(后来被国际商业机器公司(IBM)收购)的技术基础上发展起来的。HP还在aC++中添加了贪婪实例化，并将其作为默认机制。

避免了Cfront实现的一些缺点。实践中，这些实现工作得非常好。尽管“从头开始”构建通常比其他方案更耗时，但后续的构建时间相当有竞争力。不过，使用迭代实例化的C++编译器仍然少见。

14.5. 显式实例化

可以为模板特化显式地创建实例化点。实现点的构造称为显式实例化指令。语法上，由关键字`template`和要实例化的特化声明组成。例如：

```
1 template<typename T>
2 void f(T)
3 { }
4
5 // four valid explicit instantiations:
6 template void f<int>(int);
7 template void f<>(float);
8 template void f(long);
9 template void f(char);
```

每个实例化指令都有效，模板参数可以推导(参见第15章)。

类模板的成员也可以这样显式实例化：

```
1 template<typename T>
2 class S {
3     public:
4         void f() {
5     }
6 };
7
8 template void S<int>::f();
9
10 template class S<void>;
```

此外，可以通过显式实例化类模板特化，来显式实例化类模板特化的所有成员。因为这些显式实例化指令创建了命名模板特化(或其成员)的定义，所以上面的显式实例化指令，可以称为显式实

例化定义。因为这意味着两个定义可能是不同的(因此违反了ODR),所以显式实例化的模板特化不应该显式特化,反之亦然。

14.5.1 手动实例化

许多C++开发者已经注意到,自动模板实例化对构建时间有很大的负面影响。对于实现贪婪实例化的编译器来说尤其如此(第14.4.1节),因为相同的模板特化,可能在许多不同的翻译单元中实例化和优化。

改进构建时间的技术包括,在特定位置手动实例化程序所需的那些模板特化,并在所有其他翻译单元中抑制实例化。确保这种抑制的一种可移植的方法是不提供模板定义,除非在显式实例化的翻译单元中定义。

1998年和2003年的C++标准中,这是在其他翻译单元中禁止实例化的唯一可移植方法。

例如:

```
1 // translation unit 1:  
2 template<typename T> void f(); // no definition: prevents instantiation  
3 // in this translation unit  
4  
5 void g()  
6 {  
7     f<int>();  
8 }  
9  
9 // translation unit 2:  
10 template<typename T> void f()  
11 {  
12     // implementation  
13 }  
14  
15 template void f<int>(); // manual instantiation  
16  
17 void g();  
18  
19 int main()  
20 {  
21     g();  
22 }
```

第一个翻译单元中,编译器无法看到函数模板f的定义,因此不会(不能)生成f<int>的实例化。第二个翻译单元通过显式实例化定义提供f<int>的定义;没有它,程序将无法连接。

手动实例化有一个明显的缺点:必须小心地跟踪要实例化的实体。对于大型项目来说,这很快就会成为一个负担;因此,我们不建议这样做。我们曾参与过几个最初低估了这个负担的项目,随着代码的成熟,我们后面就开始后悔开始的决定。

然而,手动实例化也有一些优点,因为可以根据程序的需要进行调整。显然,避免了巨型头文件的开销,也避免了在多个翻译单元中使用相同参数重复实例化相同模板的开销。此外,模板定义

的源代码可以隐藏，但是使用程序就不能进行额外的实例化。

通过将模板定义放置到第三个源文件(通常扩展名为.tpp)中，可以减轻手动实例化的一些负担。对于函数f，可以分解为：

```
1 // f.hpp:  
2 template<typename T> void f(); // no definition: prevents instantiation  
3  
4 // f.tpp:  
5 #include "f.hpp"  
6 template<typename T> void f() // definition  
7 {  
8     // implementation  
9 }  
10  
11 // f.cpp:  
12 #include "f.tpp"  
13  
14 template void f<int>(); // manual instantiation
```

这种结构提供了一定的灵活性。可以只包含f.hpp来获得f的声明，可以根据需要将显式实例化添加到f.cpp中。或者，若手动实例化过于繁重，还可以包含f.tpp来启用自动实例化。

14.5.2 显式实例化声明

消除冗余自动实例化的一种更有针对性的方法是使用显式实例化声明，它以关键字**extern**为前缀的显式实例化指令。显式实例化声明通常会抑制已命名模板特化的自动实例化，因为声明已命名模板特化将在程序中的某个地方定义(通过显式实例化定义)。之所以说一般，是因为有很多例外情况：

- 内联函数仍然可以实例化，以便内联扩展(不会生成单独的目标代码)。
- 具有**auto**或**decltype(auto)**类型的变量，以及具有推导返回类型的函数，可以实例化以确定类型。
- 值可用作常量表达式的变量仍然可以实例化，以便计算它们的值。
- 引用类型的变量仍然可以实例化，以便解析引用的实体。
- 类模板和别名模板仍然可以实例化，以检查产生的类型。

使用显式的实例化声明，可以在头文件(t.hpp)中提供f的模板定义，然后抑制常用特化的自动实例化，如下所示：

```
1 // t.hpp:  
2 template<typename T> void f()  
3 { }  
4  
5 extern template void f<int>(); // declared but not defined  
6 extern template void f<float>(); // declared but not defined  
7  
8 // t.cpp:  
9 template void f<int>(); // definition
```

```
10 template void f<float>(); // definition
```

每个显式实例化声明必须与相应的显式实例化定义配对，该定义必须在显式实例化声明之后。省略该定义将导致链接器报错。

在许多不同的翻译单元中使用特定的专特化时，可以使用显式实例化声明来改进编译或链接时间。手动实例化需要在需要新的特化时手动更新显式实例化定义列表。与手动实例化不同的是，可以将显式实例化声明作为一种优化引入。然而，编译时的好处可能没有手动实例化那么显著，因为可能会出现一些冗余的自动实例化，而且模板定义仍然会作为头文件的一部分进行解析。

这个优化问题的有趣部分是，确定哪些特化适合显式的实例化声明。低层实用程序（如 Unix 工具 nm）在识别程序的目标文件中的自动实例化方面非常有用。

14.6. 编译时 if 语句

正如第 8.5 节中介绍的，C++17 添加了一种新的语句类型，编写模板时非常有用：编译时 if。不过，这在实例化过程中引入了一个新的问题。

下面的例子演示了其基本操作：

```
1 template<typename T> bool f(T p) {
2     if constexpr (sizeof(T) <= sizeof(long long)) {
3         return p>0;
4     } else {
5         return p.compare(0) > 0;
6     }
7 }
8
9 bool g(int n) {
10    return f(n); // OK
11 }
```

编译时 if 是一个 if 语句，其中 if 关键字紧跟着 constexpr 关键字（如本例所示）。

虽然代码读取 if constexpr，但该特性称为 constexpr if，因为它是 if 的“constexpr”形式。

后面的圆括号条件必须有一个确定的布尔值（包含了对 bool 的隐式转换）。因此，编译器知道将选择哪个分支；另一个分支为丢弃分支。特别值得注意的是，在模板（包括泛型 Lambda）的实例化期间，不会实例化丢弃分支。我们使用 T = int 实例化 f(T)，将丢弃 else 分支。如果没有丢弃，将会实例化，并且会遇到表达式 p.compare(0) 的错误信息（当 p 是一个简单整数时，这个表达式是错误的）。

C++17 及其 constexpr if 语句可用之前，为了避免此类错误，需要显式的模板特化或重载（参见第 16 章）来达到类似的效果。

上面的例子中，在 C++14 可以这样实现：

```
1 template<bool b> struct Dispatch { // only to be instantiated when b is false
2     static bool f(T p) { // (due to next specialization for true)
3         return p.compare(0) > 0;
```

```

4     }
5 };
6
7 template<> struct Dispatch<true> {
8     static bool f(T p) {
9         return p > 0;
10    }
11 };
12
13 template<typename T> bool f(T p) {
14     return Dispatch<sizeof(T) <= sizeof(long long)>::f(p);
15 }
16
17 bool g(int n) {
18     return f(n); // OK
19 }

```

显然, `constexpr if` 替代方案使我们的意图, 更加清楚和简洁。然而, 需要实现来细化实例化的单元: 以前的函数定义总是作为一个整体实例化, 现在必须抑制部分实例化。

`constexpr if` 的另一个用法是表示处理函数参数包所需的递归。为了推广这个例子, 在第 8.5 节中进行了介绍:

```

1 template<typename Head, typename... Remainder>
2 void f(Head&& h, Remainder&&... r)
3     doSomething(std::forward<Head>(h));
4     if constexpr (sizeof... (r) != 0) {
5         // handle the remainder recursively (perfectly forwarding the arguments):
6         f(std::forward<Remainder>(r)...);
7     }
8 }

```

如果没有 `constexpr if` 语句, 就需要对 `f()` 模板进行重载, 以确保递归终止。

即使在非模板上下文中, `constexpr if` 语句也有些独特的效果:

```

1 void h();
2 void g() {
3     if constexpr (sizeof(int) == 1) {
4         h();
5     }
6 }

```

在大多数平台上, `g()` 中的条件为 `false`, 因此会丢弃 `h()` 的调用。因此, `h()` 不需要定义(当然, 除非在其他地方使用)。如果本例中省略了关键字 `constexpr`, 那么缺少 `h()` 的定义通常会在链接时引起错误。

优化可能会掩盖错误。如果保证不存在问题, 则可以使用 `constexpr`。

14.7. 标准库中的显式实例化

C++ 标准库包含许多模板，这些模板通常只用于基本类型。例如，`std::basic_string` 类模板最常与 `char(std::string)` 是 `std::basic_string<char>` 的类型别名) 或 `wchar_t` 一起使用，可以用其他类似字符的类型实例化。因此，标准库实现通常会为这些常见情况引入显式的实例化声明。例如：

```
1 namespace std {
2     template<typename charT, typename traits = char_traits<charT>,
3              typename Allocator = allocator<charT>>
4     class basic_string {
5         ...
6     };
7     extern template class basic_string<char>;
8     extern template class basic_string<wchar_t>;
9 }
```

实现标准库的源文件将包含显式实例化定义，这些公共实现可以在标准库的所有用户间共享。类似的显式实例化通常存在于各种流类中，例如 `basic_iostream`、`basic_istream` 等。

14.8. 后记

本章讨论两个相关但不同的问题：C++ 模板编译模型和 C++ 模板实例化机制。

编译模型在程序转换的各个阶段决定模板的含义。特别地，它确定了模板中各种构造在实例化时的含义。名称查找是编译模型的重要组成部分。

标准 C++ 只支持一种编译模型，即包含模型。然而，1998 年和 2003 年的标准也支持模板编译的分离模型，允许在与实例化不同的翻译单元中编写模板定义。这些导出的模板只由爱迪生设计团队 (EDG) 实现过。

具有讽刺意味的是，将该技术添加到标准文档中时，EDG 是最强烈的反对者。

实现工作确定了：(1) 实现 C++ 模板的分离模型比预期的要困难得多，耗时也多；(2) 分离模型的假设优势（比如：改进编译时间），由于模型的复杂性，并没有实现。随着 2011 年标准的开发接近尾声，并且其他实现者不打算支持该特性，C++ 标准委员会投票决定从该语言中删除导出模板的特性。对分离模型的细节感兴趣的读者，推荐本书的第一版 ([VandevoordeJosuttisTemplates1st])，其中描述了导出模板的具体行为。

实例化机制允许 C++ 实现正确创建实例化的机制，这些机制可能会受到链接器和其他软件构建工具的限制。虽然实例化机制在不同的实现中不同（每个实现都有其优缺点），但通常不会对 C++ 的编程产生重大影响。

C++11 完成后不久，Walter Bright、Herb Sutter 和 Andrei Alexandrescu 提出了一个与 `constexpr if` 类似的“静态 if”特性（通过 N3329）。然而，它是一个更普遍的特性，甚至可以出现在函数定义之外（Walter Bright 是 D 编程语言的主要设计者和实现者，D 语言也有类似的功能）。例如：

```
1 template<unsigned long N>
2 struct Fact {
3     static if (N <= 1) {
```

```
4     constexpr unsigned long value = 1;
5 } else {
6     constexpr unsigned long value = N*Fact<N-1>::value;
7 }
8 };
```

本例中，类作用域声明是有条件的。然而，这种强大能力具有争议，一些委员会成员担心它会滥用，而另一些则不喜欢该提议使用的某些技术(例如，大括号没有引入范围，并且丢弃的分支根本没有解析)。

几年后，Ville Voutilainen 带着提案(P0128)回来了，就是后来的 `constexpr if` 语句。经过了一些设计迭代(涉及到暂定的关键字 `static_if` 和 `constexpr_if`)，在 Jens Maurer 的帮助下，Ville 最终将该提案引入语言(通过 P0292r2)。

第 15 章 模板参数推导

每次调用函数模板时显式地指定模板参数(例如, `concat<std::string, int>(s, 3)`), 代码很快就会变得很笨重。幸运的是, C++ 编译器可以使用模板参数推导功能自动确定模板参数。

本章中, 我们会解释模板参数推导的详细过程。就像在 C++ 中出现的情况一样, 许多规则通常会产生直观的结果。

尽管模板参数推导最初是为了简化函数模板调用添加的, 但后来扩展到了其他用途, 包括从初始化式确定变量的类型。

15.1. 推导过程

基本推导过程是, 将函数调用的参数类型与函数模板相应的类型参数进行比较, 并尝试对推导出的一个或多个参数类型进行替换。每个参数对都进行独立分析, 若最后得出的结论不同, 则推导失败。看看下面的例子:

```
1 template<typename T>
2 T max (T a, T b)
3 {
4     return b < a ? a : b;
5 }
6
7 auto g = max(1, 1.0);
```

第一个调用参数是 `int` 类型, 因此初始 `max()` 模板的参数 `T` 试探性地推导为 `int` 类型。然而, 第二个调用参数是 `double`, 因此对于这个参数 `T` 应该是 `double` 型参数: 这与前面的结论冲突。这里说的是“推导失败”, 而不是“程序无效”。毕竟, 对于另一个名为 `max` 的模板, 推导可能会成功(函数模板可以像普通函数一样重载; 请参阅第 1.5 节和第 16 章)。

若推导出的所有模板参数结论一致, 那么若在函数声明的其余部分中替换参数导致无效构造, 推导过程仍可能失败。例如:

```
1 template<typename T>
2 typename T::ElementT at (T a, int i)
3 {
4     return a[i];
5 }
6
7 void f (int* p)
8 {
9     int x = at(p, 7);
10 }
```

这里 `T` 推断为 `int*`(只有一种参数类型出现了 `T`, 因此显然没有分析冲突)。但在返回类型 `T::ElementT` 中用 `int*` 替换 `T` 显然无效, 所以推导失败。

这种情况下, 推导失败会导致错误。但这属于 SFINAE(参见第 8.4 节): 如果另一个函数推导成功, 代码可能是有效的。

继续探索参数匹配是如何进行的。我们将其描述为匹配类型 A(从调用参数类型派生) 到参数化类型 P(从调用参数声明派生)。若调用参数是用引用声明符声明的, P 为引用的类型, A 是具体的参数类型。但在其他情况下, P 是声明的参数类型, 而 A 是通过(忽略 const 和 volatile 限定符)将数组和函数类型衰变为指针类型。

衰变是指函数和数组类型隐式转换为指针类型的术语。

例如:

```
1 template<typename T> void f(T); // parameterized type P is T
2 template<typename T> void g(T&); // parameterized type P is also T
3
4 double arr[20];
5 int const seven = 7;
6
7 f(arr); // nonreference parameter: T is double*
8 g(arr); // reference parameter: T is double[20]
9 f(seven); // nonreference parameter: T is int
10 g(seven); // reference parameter: T is int const
11 f(7); // nonreference parameter: T is int
12 g(7); // reference parameter: T is int => ERROR: can't pass 7 to int&
```

对于调用 `f(arr)`, `arr` 的数组类型衰变为 `double*` 类型, 这是 `T` 的类型。`f(7)` 中去掉了 `const` 限定, 因此 `T` 推导为 `int`。相反, 调用 `g(x)` 可以将 `T` 推导为 `double[20]` 类型(没有发生衰变)。类似地, `g(7)` 的左值参数类型为 `int const`, 在匹配引用参数时不删除 `const` 和 `volatile` 限定符, 因此 `T` 推导为 `int const`。但注意 `g(7)` 会推导 `T` 为 `int`(非类右值表达式没有 `const` 或 `volatile` 限定类型), 并且调用会失败, 因为 7 不能传递给 `int&` 类型的参数。

当参数是字符串字面值时, 绑定到引用的参数不会发生衰变。重新考虑用引用声明的 `max()` 模板:

```
1 template<typename T>
2 T const& max(T const& a, T const& b);
```

可以合理地预期, 对于表达式 `max("Apple", "Pie")`, `T` 推导为 `char const*`。然而, “Apple”的类型是 `char const[6]`, 而 “Pie” 的类型是 `char const[4]`。没有发生数组到指针的衰变(推导涉及到引用参数), 因此 `T` 必须同时是 `char[6]` 和 `char[4]`, 才能推导成功, 但这是不可能的。参见第 7.4 节关于如何处理这种情况的讨论。

15.2. 推导上下文

比 “T” 复杂得多的参数化类型可以匹配到给定的参数类型。下面是一些例子:

```
1 template<typename T>
2 void f1(T*);
3
4 template<typename E, int N>
5 void f2(E(&)[N]);
```

```

6
7 template<typename T1, typename T2, typename T3>
8 void f3(T1 (T2::*)(T3*)) ;
9
10 class S {
11     public:
12     void f(double*);
13 };
14
15 void g (int*** ppp)
16 {
17     bool b[42];
18     f1(ppp); // deduces T to be int**
19     f2(b); // deduces E to be bool and N to be 42
20     f3(&S::f); // deduces T1 = void, T2 = S, and T3 = double
21 }

```

复杂类型声明有基本的构造(指针、引用、数组和函数声明符;指向成员的声明符;模板标识;等等),匹配过程从顶层构造开始,递归地遍历各个元素。大多数类型声明构造都可以用这种方式匹配,这称为推导上下文。然而,有一些结构不可推导上下文。例如:

- 限定类型名称。例如,永远不会使用像 $\text{Q}\langle\text{T}\rangle::\text{X}$ 这样的类型名来推导模板参数 T 。
- 非类型参数不只有非类型表达式,像 $\text{S}\langle\text{I}+1\rangle$ 这样的类型名永远不会用于推断 I ,也不会通过匹配类型为 $\text{int}(\&)[\text{sizeof}(\text{S}\langle\text{T}\rangle)]$ 的参数来推断 T 的类型。

这些限制并不奇怪,因为推导过程通常不唯一(甚至是有限的),尽管限定类型名的这种限制有时容易忽略。非推导上下文不会表示程序出错,甚至不会说明分析的参数不能参与类型推导。为了说明这一点,看看下面这个更复杂的例子:

basics/fppm.cpp

```

1 template<int N>
2 class X {
3     public:
4     using I = int;
5     void f(int) {
6     }
7 };
8
9 template<int N>
10 void fppm(void (X<N>::*p) (typename X<N>::I));
11
12 int main()
13 {
14     fppm(&X<33>::f); // fine: N deduced to be 33
15 }

```

函数模板 $fppm()$ 中,子构造 $X\langle N \rangle::I$ 不用推导上下文。然而,成员指针类型的成员类组件 $X\langle N \rangle$ 可推导上下文,当它推导出的参数 N 放入到非推导上下文时,将获得与实际参数类型兼容的类型

`&X<33>::f`。因此，参数匹配，推导成功。

相反，对于完全由推导上下文构建的参数类型，推导也可能出现矛盾。例如，声明的类模板 X 和 Y：

```
1 template<typename T>
2 void f(X<Y<T>, Y<T>>);
3
4 void g()
5 {
6     f(X<Y<int>, Y<int>>()); // OK
7     f(X<Y<int>, Y<char>>()); // ERROR: deduction fails
8 }
```

第二次调用函数模板 `f()` 的问题是，两个参数 T 会推导出不同的类型，这是无效的（两种情况下，函数调用参数是通过调用类模板 X 的默认构造函数，获得的临时对象）。

15.3. 特殊的推导情况

几种情况下，用于推导的一对 (A, P) 不能从函数调用的参数和函数模板参数中获得。第一种情况发生在获取函数模板地址时，P 是函数模板声明的参数化类型，A 是初始化或赋值给指针的函数类型。例如：

```
1 template<typename T>
2 void f(T, T);
3
4 void (*pf)(char, char) = &f;
```

例子中，P 是 `void(T, T)`，A 是 `void(char, char)`。`char` 替换 `T` 后，推导成功，`pf` 初始化为特化的地址 `f<char>`。

类似地，函数类型用于 P 和 A 的一些特殊情况：

- 确定重载函数模板之间的部分顺序
- 显式特化与函数模板匹配
- 显式实例化与模板匹配
- 友元函数模板特化与模板进行匹配
- 替换 `delete` 操作符或 `delete[]` 操作符，`new` 操作符或 `new[]` 操作符的模板进行匹配

其中一些话题，以及在类模板偏特化中使用模板参数推导，将在第 16 章中进一步讨论。

另一种特殊情况发生在转换函数模板中。例如：

```
1 class S {
2     public:
3         template<typename T> operator T&();
4 };
```

这种情况下，获得 (P, A) 对，就好像包含一个要转换的类型和一个作为转换函数返回类型的类型：

```
1 void f(int (&)[20]);
2
3 void g(S s)
4 {
5     f(s);
6 }
```

这里，试图将 S 转换为 int[&][20]。因此，类型 A 为 int[20]，类型 P 为 T。int[20] 替换 T，推导成功。

最后，还需要对推导自动占位符类型进行一些特殊处理。这在第 15.10.4 节中会进行讨论。

15.4. 初始化列表

当函数调用的参数是初始化列表时，参数没有特定的类型，通常不会从一对给定的 (A, P) 进行推断，因为没有 A。例如：

```
1 #include <initializer_list>
2
3 template<typename T> void f(T p);
4
5 int main() {
6     f({1, 2, 3}); // ERROR: cannot deduce T from a braced list
7 }
```

然而，若参数类型 P 在除去引用，以及 const 和 volatile 限定符后，等价于某些类型 P0 的 std::initializer_list，这种类型的 P0 可推导。通过将 P0 与初始化列表中每个元素的类型进行比较，只有当所有元素都具有相同类型时，推导才会成功：

basics/initlist.cpp

```
1 #include <initializer_list>
2
3 template<typename T> void f(std::initializer_list<T>);
4
5 int main()
6 {
7     f({2, 3, 5, 7, 9}); // OK: T is deduced to int
8     f({'a', 'e', 'i', 'o', 'u', 42}); // ERROR: T deduced to both char and int
9 }
```

类似地，若参数类型 P 是一个类型数组的引用 (可推导)，推导过程也会将初始化列表的每个元素的类型与 P 类型数组的元素类型进行比较，当所有元素都有相同的类型时，推导才成功。此外，若 (数组) 边界可推导 (即，只指定非类型模板参数)，那么该边界会推导为初始化列表中元素的数量。

15.5. 参数包

推导过程将每个参数进行匹配，以确定模板参数的值。当对可变参数模板执行模板参数推导时，参数之间不再具有一一对应的关系，因为一个参数包可以匹配多个参数。这种情况下，同一个参数包 (P) 匹配多个参数 (A)，每次匹配都会为 P 中的模板参数包产生额外的值：

```
1 template<typename First, typename... Rest>
2 void f(First first, Rest... rest);
3
4 void g(int i, double j, int* k)
5 {
6     f(i, j, k); // deduces First to int, Rest to {double, int*}
7 }
```

第一个函数参数的推导很简单，不涉及参数包。第二个函数参数 rest 是一个函数参数包，类型是一个包扩展 (Rest...)，模式是 Rest 类型：该模式用作 P，与第二个和第三个调用参数的类型 A 进行比较。当与第一个 A(double 类型) 比较时，模板参数包 Rest 中的第一个值推导为 double。类似地，当与第二个这样的 A(类型为 int*) 比较时，模板参数包 Rest 中的第二个值推导为 int*。因此，推导决定了参数包 Rest 的值是序列 {double, int*}。将该推导结果和第一个函数参数的推导结果结合，替换为函数类型 void(int, double, int*)，与调用点的参数类型匹配。

因为函数参数包的推导使用展开模式进行比较，所以模式可以随意，并且可以从每个参数类型确定多个模板参数和参数包的值。考虑 h1() 和 h2() 函数的推导行为：

```
1 template<typename T, typename U> class pair { };
2
3 template<typename T, typename... Rest>
4     void h1(pair<T, Rest> const&...);
5
6 template<typename... Ts, typename... Rest>
7     void h2(pair<Ts, Rest> const&...);
8
9 void foo(pair<int, float> pif, pair<int, double> pid,
10 pair<double, double> pdd)
11 {
12     h1(pif, pid); // OK: deduces T to int, Rest to {float, double}
13     h2(pif, pid); // OK: deduces Ts to {int, int}, Rest to {float, double}
14     h1(pif, pdd); // ERROR: T deduced to int from the 1st arg, but to double from the 2nd
15     h2(pif, pdd); // OK: deduces Ts to {int, double}, Rest to {float, double}
16 }
```

对于 h1() 和 h2()，P 是一个引用类型，为引用的非限定版本（分别为 pair<T, Rest> 或 pair<Ts, Rest>），用于对每个参数类型进行推导。所有参数都是类模板 pair 的特化，因此需要比较模板参数。对于 h1()，第一个模板参数 (T) 不是一个参数包，因此其值是针对每个参数可以进行独立推导。如果类型推导不同，如第二次调用 h1()，则推导失败。对第二个在 h1() 和 h2() 的 pair 模板参数 Rest、以及 h2() 的第一个模板实参 Ts 来说，推导会根据 A 的每个参数类型来确定模板参数包的值。

参数包的推导不限于来自函数参数包的参数对，当包扩展位于函数参数列表或模板参数列表的末尾时，就会使用这种推导。

如果包展开发生在函数参数列表或模板参数列表的位置，则该包展开不可推导上下文。

例如， Tuple 类型上的两个类似操作：

```
1 template<typename... Types> class Tuple { };
2
3 template<typename... Types>
4 bool f1(Tuple<Types...>, Tuple<Types...>);
5
6 template<typename... Types1, typename... Types2>
7 bool f2(Tuple<Types1...>, Tuple<Types2...>);
8
9 void bar(Tuple<short, int, long> sv,
10          Tuple<unsigned short, unsigned, unsigned long> uv)
11 {
12     f1(sv, sv); // OK: Types is deduced to {short, int, long}
13     f2(sv, sv); // OK: Types1 is deduced to {short, int, long},
14         // Types2 is deduced to {short, int, long}
15     f1(sv, uv); // ERROR: Types is deduced to {short, int, long} from the 1st arg, but
16         // to {unsigned short, unsigned, unsigned long} from the 2nd
17     f2(sv, uv); // OK: Types1 is deduced to {short, int, long},
18         // Types2 is deduced to {unsigned short, unsigned, unsigned long}
19 }
```

f1() 和 f2() 中，模板参数包通过比较嵌在 Tuple 类型中的包展开模式 (例如，类型为 h10) 与调用参数提供 Tuple 类型的每个模板参数来推导的，依次为相应的模板参数包推导出相应的值。函数 f1() 在两个函数参数中使用相同的模板参数包 Types，确保只有两个函数调用参数，具有与其类型相同的 Tuple 特化时，推导才能成功。另一方面，函数 f2() 在其每个函数参数中为 Tuple 类型使用不同的参数包，因此函数调用参数的类型可以不同——只要两者都是 Tuple 的特化即可。

15.5.1 字面量操作符模板

文字操作符模板的参数以一种独特的方式确定。下面的例子说明了这一点：

```
1 template<char...> int operator "" _B7(); // #1
2 ...
3 int a = 121_B7; // #2
```

这里，#2 的初始化式包含用户定义的字面量，转换成对字面操作符模板 #2 的调用，其中包含模板参数列表 <'1', '2', '1'>。因此，文字操作符的实现：

```
1 template<char... cs>
2 int operator"" _B7()
3 {
4     std::array<char, sizeof...(cs)> chars{cs...}; // initialize array of passed chars
5     for (char c : chars) { // and use it (print it here)
6         std::cout << " " << c << " ";
7     }
8     std::cout << '\n' ;
```

```
9  return ...;  
10 }
```

会为 121.5_B7 输出'1' '2' '1' '.' '5'。

这种技术只支持那些没有后缀也有效的数字字面值。例如:

```
1 auto b = 01.3_B7; // OK: deduces <' 0' , ' 1' , ' .' , ' 3' >  
2 auto c = 0xFF00_B7; // OK: deduces <' 0' , ' x' , ' F' , ' F' , ' 0' , ' 0' >  
3 auto d = 0815_B7; // ERROR: 8 is no valid octal literal  
4 auto e = hello_B7; // ERROR: identifier hello_B7 is not defined  
5 auto f = "hello"_B7; // ERROR: literal operator _B7 does not match
```

请参阅第 25.6 节, 了解该特性的应用: 在编译时计算整数字面值。

15.6. 右值引用

C++11 引入了右值引用来支持新的特性, 包括移动语义和完美转发。本节描述右值引用和类型推导之间的交互。

15.6.1 引用折叠规则

开发者不允许直接声明“引用的引用”:

```
1 using RI = int&;  
2 int i = 42;  
3 RI r = i;  
4 RI const& rr = r; // OK: rr has type int&
```

确定由这样的组合产生的类型的规则称为引用折叠规则。

当人们注意到标准 pair 类模板不能使用引用类型时, 引用折叠就引入了 C++2003 标准中。2011 年的标准进一步扩展了右值引用的规则。

首先, 应用于内部引用之上的 const 或 volatile 限定符都会丢弃(只有内部引用之下的限定符会保留)。根据表 15.1, 这两个引用会简化为一个引用, 可以总结为“如果其中一个引用是左值引用, 得到的类型也是左值引用; 否则, 就是一个右值引用。”

Inner reference	Outer reference	Resulting reference
&	+	&
&	+	&
&&	+	&
&&	+	&&

表 15.1. 引用折叠规则

还有一个例子展示了这些规则的实际应用:

```

1 using RCI = int const&;
2 RCI volatile&& r = 42; // OK: r has type int const&
3 using RRI = int&&;
4 RRI const&& rr = 42; // OK: rr has type int&&

```

volatile 应用于引用类型 RCI(int const& 的别名), 因此丢弃。然后, 右值引用位于该类型的顶部, 但是由于基础类型是左值引用, 并且左值引用在引用折叠规则中“优先”, 所以整体类型保持 int const&(或 RCI, 是一个等效别名)。类似地, 丢弃 RRI 上的 const, 并且在产生的右值引用类型上应用右值引用, 最后留下一个右值引用类型(能像 42 那样绑定右值)。

15.6.2 转发引用

6.1 节中介绍的, 当函数参数是转发引用(对该函数模板参数的右值引用)时, 模板参数推导会以一种特殊的方式进行。在这种情况下, 模板参数推导不仅要考虑函数调用参数的类型, 还要考虑该参数是左值还是右值。参数是左值时, 由模板参数推导确定的类型为参数类型的左值引用, 引用折叠规则(见上面)确保替换的参数是左值引用。否则, 为模板参数推导出的类型就是参数类型(不是引用类型), 而替换的参数是该类型的右值引用。例如:

```

1 template<typename T> void f(T&& p); // p is a forwarding reference
2
3 void g()
4 {
5     int i;
6     int const j = 0;
7     f(i); // argument is an lvalue; deduces T to int& and
8     // parameter p has type int&
9     f(j); // argument is an lvalue; deduces T to int const&
10    // parameter p has type int const&
11    f(2); // argument is an rvalue; deduces T to int
12    // parameter p has type int&&
13 }

```

调用 f(i) 时, 模板参数 T 推导为 int&, 因为表达式 i 是 int 类型的左值。将 int& 替换为 T 的参数类型 T&& 需要引用折叠, 这里使用 & + && -> & 的规则, 得到的参数类型是 int&, 适合接受 int 类型的左值。相反, f(2) 中参数 2 是一个右值, 因此模板参数推导为该右值的类型(即 int)。生成的函数参数不需要引用折叠, 类型为 int&&(同样, 适合的参数)。

将 T 推导为引用类型会对模板的实例化产生一些影响。例如, 用 T 类型声明的局部变量在左值实例化后具有引用类型, 因此需要初始化式:

```

1 template<typename T> void f(T&&) // p is a forwarding reference
2 {
3     T x; // for passed lvalues, x is a reference
4     ...
5 }

```

上面函数 f() 的定义需要注意如何使用 T 类型, 否则函数模板本身在使用左值参数时, 将无法正常工作。为了处理这种情况, 常使用 std::remove_reference 类型特性来确定 x 不是一个引用:

```
1 template<typename T> void f(T&&) // p is a forwarding reference
2 {
3     std::remove_reference_t<T> x; // x is never a reference
4     ...
5 }
```

15.6.3 完美转发

右值引用的特殊推导规则和引用折叠规则的结合，使编写一个函数模板成为可能，其参数可以接受任何参数，并捕获其属性(类型，以及它是左值还是右值)。

位域是个例外。

然后，函数模板可以将参数“转发”到另一个函数，如下所示：

```
1 class C {
2 ...
3 };
4
5 void g(C&);
6 void g(C const&);
7 void g(C&&);
8
9 template<typename T>
10 void forwardToG(T&& x)
11 {
12     g(static_cast<T&&>(x)); // forward x to g()
13 }
14
15 void foo()
16 {
17     C v;
18     C const c;
19     forwardToG(v); // eventually calls g(C&)
20     forwardToG(c); // eventually calls g(C const&)
21     forwardToG(C()); // eventually calls g(C&&)
22     forwardToG(std::move(v)); // eventually calls g(C&&)
23 }
```

上面演示的技术称为完美转发，通过forwardToG()间接调用g()的结果将与直接调用g()的代码相同：不生成副本。

forwardToG()函数中使用static_cast需要一些解释。forwardToG()的实例化中，参数x要么具有左值引用类型，要么具有右值引用类型。无论如何，表达式x将是引用所引用类型的左值。

将右值引用类型的参数作为左值处理是出于安全考虑，因为带有名称的参数可以很容易地在函数中多次引用。若这些引用中的每一个都可以隐式地作为右值，那么其值就可以在开发者不知道的情况下销毁。因此，必须显式声明命名实体何时视为右值。为此，C++ 标准库函数 `std::move` 将其操作的值视为右值（或者更准确地说，`xvalue`；详见附录 B）。

`static_cast` 将 `x` 转换为原始类型和左值或右值。类型 `T&&` 要么折叠为左值引用（如果原始参数是左值，则 `T` 是左值引用），要么会是右值引用（如果原始参数是右值），因此 `static_cast` 的结果具有与原始参数相同的类型和左值或右值性，从而实现完美转发。

正如在第 6.1 节中介绍的，C++ 标准库在头文件 `<utility>` 中提供了一个函数模板 `std::forward<T>()`，该函数模板应用于替代 `static_cast` 以实现完美转发。使用该程序模板比上面的不透明 `static_cast` 更好地记录了开发者的意图，并防止了遗漏 `&` 等错误。也就是说，上面的例子可以写得更清楚：

```
1 #include <utility>
2
3 template<typename T> void forwardToG(T&& x)
4 {
5     g(std::forward<T>(x)); // forward x to g()
6 }
```

完美转发可变参数模板

完美转发与可变参数模板可以很好的结合，允许函数模板接受任意数量的函数调用参数，并将每个参数转发给另一个函数：

```
1 template<typename ... Ts> void forwardToG(Ts&&... xs)
2 {
3     g(std::forward<Ts>(xs)...); // forward all xs to g()
4 }
```

`forwardToG()` 调用中的参数将（独立地）推导参数包 `Ts` 的连续值（参见第 15.5 节），以便捕获每个参数的类型和左值或右值。调用 `g()` 中的包扩展（参见第 12.4.1 节）将使用完美转发，转发每个参数。

尽管称为完美转发，但并不是“完美”的，因为没有捕捉到一个表达式的所有属性。例如，不区分左值是否是位域左值，也不捕获表达式是否具有特定的常量值。后者会引起问题，特别是处理空指针常量时，这是一个整型值，计算结果为常量 0。由于完美转发无法捕获表达式的常量值，因此在下面的例子中，对 `g()` 的直接调用与对 `g()` 的转发调用会有不同的行为：

```
1 void g(int*);
2 void g(...);
3
4 template<typename T> void forwardToG(T&& x)
5 {
6     g(std::forward<T>(x)); // forward x to g()
7 }
8
```

```

9 void foo()
10 {
11     g(0); // calls g(int*)
12     forwardToG(0); // eventually calls g(...)
13 }

```

这也是使用 `nullptr`(C++11 中引入), 而不是空指针常量的另一个原因:

```

1 g(nullptr); // calls g(int*)
2 forwardToG(nullptr); // eventually calls g(int*)

```

所有的完美转发示例都专注于转发函数参数, 同时保持它们类型, 以及是左值还是右值。当将调用的返回值转发给另一个函数时, 也会出现同样的问题, 该函数的类型和值类型完全相同, 即在附录 B 中讨论的左值和右值。C++11 中引入的 `decltype` 功能(在第 15.10.2 节)允许使用这种有点冗长的惯用法:

```

1 template<typename... Ts>
2 auto forwardToG(Ts&&... xs) -> decltype(g(std::forward<Ts>(xs)...))
3 {
4     return g(std::forward<Ts>(xs)...); // forward all xs to g()
5 }

```

注意, `return` 语句中的表达式会逐字复制到 `decltype` 类型中, 以便计算返回表达式的确切类型。此外, 还使用了末尾的返回类型特征(即, 函数名之前的自动占位符和`->`来指示返回类型), 以便函数参数包 `xs` 在 `decltype` 类型的范围内。这个转发函数“完美”地将所有参数转发给 `g()`, 然后“完美”地将其结果转发回调用者。

C++14 引入了其他特性进一步简化这种情况:

```

1 template<typename... Ts>
2 decltype(auto) forwardToG(Ts&&... xs)
3 {
4     return g(std::forward<Ts>(xs)...); // forward all xs to g()
5 }

```

使用 `decltype(auto)` 作为返回类型表明编译器应该从函数的定义, 推导出返回类型。参见 15.10.1 节和 15.10.3 节。

15.6.4 惊奇推导

对右值引用的特殊推导规则结果, 对完善转发具有重要意义。然而, 也可能会令人惊讶, 因为函数模板通常在函数签名中泛化类型, 而不影响参数类型(左值或右值)。考虑一下这个例子:

```

1 void int_lvalues(int&); // accepts lvalues of type int
2 template<typename T> void lvalues(T&); // accepts lvalues of any type
3
4 void int_rvalues(int&&); // accepts rvalues of type int
5 template<typename T> void anything(T&&); // SURPRISE: accepts lvalues and
6 // rvalues of any type

```

将具体函数(如 `int_rvalues`)抽象到模板等价的开发者，可能会对函数模板接受左值感到惊讶。幸运的是，这种推导行为仅适用于以下情况：函数参数是用表单模板参数 `&&` 编写的，是函数模板的一部分，并且命名的模板参数由该函数模板声明。因此，该推导规则不适用于以下情况：

```
1 template<typename T>
2 class X
3 {
4     public:
5         X(X&&); // X is not a template parameter
6         X(T&&); // this constructor is not a function template
7
8     template<typename U> X(X<U>&&); // X<U> is not a template parameter
9     template<typename U> X(U, T&&); // T is a template parameter from
10    // an outer template
11};
```

尽管这个模板推导规则给出了令人惊讶的行为，但在实际中，这种行为导致问题的情况并不常见。发生时，可以使用 SFINAE(参见第 8.4 节和第 15.7 节) 和类型特征，如 `std::enable_if`(参见第 6.3 节和第 20.3 节) 来限制模板只能接收右值：

```
1 template<typename T>
2 typename std::enable_if<!std::is_lvalue_reference<T>::value>::type
3 rvalues(T&&); // accepts rvalues of any type
```

15.7. SFINAE(替换失败不为过)

SFINAE(替换失败不为过) 原则在第 8.4 节中介绍，其对模板参数类型的推导有很大的影响，可以避免不相关的函数模板在重载解析期间引起的错误。

SFINAE 也适用于部分类模板特化的替换。参见第 16.4 节

例如，考虑一对函数模板，用于提取容器或数组的起始迭代器：

```
1 template<typename T, unsigned N>
2 T* begin(T (&array) [N])
3 {
4     return array;
5 }
6
7 template<typename Container>
8 typename Container::iterator begin(Container& c)
9 {
10    return c.begin();
11 }
12
13 int main()
14 {
15     std::vector<int> v;
```

```
16 int a[10];
17
18 ::begin(v); // OK: only container begin() matches, because the first deduction fails
19 ::begin(a); // OK: only array begin() matches, because the second substitution fails
20 }
```

对 `begin()` 的第一次调用，其中参数是 `std::vector<int>`，尝试对 `begin()` 函数模板进行模板参数推导：

- 数组 `begin()` 的模板参数推导失败，因为 `std::vector` 不是数组，所以忽略。
- 容器 `begin()` 的模板实参推导成功，将 `Container` 推导为 `std::vector<int>`，从而实例化和调用函数模板。

对 `begin()` 的第二次调用，参数是一个数组，也会部分失败：

- 数组 `begin()` 推导成功，`T` 推导为 `int`, `N` 推导为 `10`。
- 对容器 `begin()` 的推导决定了将 `Container` 替换为 `int[10]`。虽然通常来说这种替换没有问题，因为数组类型没有名为 `iterator` 的迭代器类型，所以产生的返回类型 `Container::iterator` 无效。在其他上下文中，访问不存在的迭代器类型将导致编译错误。SFINAE 在替换模板参数时，将这些错误转化为推导失败，不再考虑函数模板。因此，忽略第二个 `begin()` 候选，并调用第一个 `begin()` 模板的特化。

15.7.1 即时上下文

SFINAE 可以避免形成无效类型或表达式的尝试，包括在函数模板替换的即时上下文中发生的由歧义或违反访问控制引起的错误。通过定义函数模板替换的即时上下文，可以更容易地定义不在该上下文中的内容。

即时上下文包括很多东西，各种查找、别名模板替换、重载解析等。但这个术语有不恰当，因为包含的一些活动与所替换的函数模板没有关系。

函数模板替换过程中，为了推导而发生的实例化，都不在函数模板替换的即时上下文中：

- 类模板的定义 (即“主体”和基类列表)
- 函数模板的定义 (函数模板本身，对于构造函数，则是构造函数初始化式)
- 变量模板的初始化式
- 默认参数
- 默认成员初始化式
- 异常规范

不是函数模板替换的即时上下文的一部分。由替换过程触发的特殊成员函数的隐式定义，也不是替换的即时上下文的一部分。其他一切都是上下文的一部分。

因此，若替换函数模板声明的模板参数需要实例化类模板的主体，该类的一个成员正在引用，那么实例化期间的错误并不在函数模板替换的即时上下文中，因此是一个真正的错误 (即使另一个函数模板匹配没有错误)。例如：

```

1 template<typename T>
2 class Array {
3     public:
4     using iterator = T*;
5 };
6
7 template<typename T>
8 void f(Array<T>::iterator first, Array<T>::iterator last);
9
10 template<typename T>
11 void f(T*, T*);
12
13 int main()
14 {
15     f<int&>(0, 0); // ERROR: substituting int& for T in the first function template
16 } // instantiates Array<int&>, which then fails

```

此示例与上一个示例的主要区别在于错误位置。前面的例子中，错误发生在 typename Container::iterator 类型时，该类型位于函数模板 begin() 替换的即时上下文中。本例中，失败发生在 Array<int&> 的实例化中，尽管由函数模板的上下文触发，但实际上发生在类模板 Array 的上下文中。因此，SFINAE 原理在这里不适用，编译器将产生错误。

下面是 C++14 的例子——依赖于推导的返回类型 (参见 15.10.1 节)——在函数模板定义的实例化过程中会产生一个错误：

```

1 template<typename T> auto f(T p) {
2     return p->m;
3 }
4
5 int f(...);
6
7 template<typename T> auto g(T p) -> decltype(f(p));
8
9 int main()
10 {
11     g(42);
12 }

```

调用 g(42) 将 T 推导为 int。在 g() 的声明中进行替换需要确定 f(p) 的类型 (p 现在已知是 int 类型)，因此需要确定 f() 的返回类型。f() 有两个候选。非模板候选匹配，但不是很好，因为它匹配一个省略号参数。不幸的是，模板候选有一个推导的返回类型，因此必须实例化定义来确定返回类型。这个实例化失败的原因是 p->m 在 p 是 int 时无效，而且由于失败在替换的即时上下文之外 (因为在函数定义的后续实例化中)，失败产生了一个错误。因此，若可以显式指定推导返回类型，这里建议不使用。

SFINAE 最初的目的是为了消除由于函数模板重载意外匹配而导致的错误，比如：容器 begin() 的例子。但检测无效表达式或类型的能力可以实现更卓越的编译技术，允许判断特定语法是否有效。这些技术将在第 19.4 节中讨论。

请参阅第 19.4.4 节，以获得使类型 SFINAE 友好的例子，以避免由于即时上下文问题造成的问题。

15.8. 推导的限制

模板参数推导是一个强大的特性，消除了在大多数函数模板调用中显式指定模板参数的需要，并且启用了函数模板重载（参见第 1.5 节）和部分类模板特化（参见第 16.4 节）。

但是，开发者在使用模板时可能会遇到一些限制，本节来讨论一下这些限制。

15.8.1 合法的参数转换

通常，模板推导试图找到函数模板参数的替换类型，使参数化的类型 P 与类型 A 相同。然而，当这不可能时，当 P 在推导上下文中包含模板参数时，以下差异合理的：

- 若原始参数是用引用声明符声明的，那么替换的 P 类型可能比 A 类型更符合 const/volatile 限定。
- 如果 A 类型是指针或成员指针类型，则可以通过限定转换转换为替换的 P 类型（换句话说，就是添加 const 和/或 volatile 限定符的转换）。
- 除非对转换操作符模板进行推导，否则替换的 P 类型可以是 A 类型的基类类型，或者是指向 A 为基类类型的指针。例如：

```
1 template<typename T>
2 class B {
3 };
4
5 template<typename T>
6 class D : public B<T> {
7 };
8
9 template<typename T> void f(B<T>*);
10 void g(D<long> dl)
11 {
12     f(&dl); // deduction succeeds with T substituted with long
13 }
```

若 P 在推导的上下文中不包含模板参数，那么所有隐式转换都是可以进行。例如：

```
1 template<typename T> int f(T, typename T::X);
2 struct V {
3     V();
4     struct X {
5         X(double);
6     };
7 } v;
8 int r = f(v, 7.0); // OK: T is deduced to V through the first parameter,
9                     // which causes the second parameter to have type V::X
10                    // which can be constructed from a double value
```

只有在不能精确匹配的情况下才考虑宽松匹配要求。即便如此，只有找到一个替换使 A 类型与添加了这些转换的 P 类型相匹配，推导才会成功。

这些规则的作用范围非常窄，忽略了（例如）可以应用于函数参数以使调用成功的各种转换。例如，下面对 max() 函数模板的调用，如 15.1 节所示：

```
1 std::string maxWithHello(std::string s)
2 {
3     return ::max(s, "hello");
4 }
```

这里由第一个参数推导的模板参数，将 T 推导为 std::string，而由第二个参数推导的模板参数 T 推导为 char[6]，因此模板参数推导失败，因为两个参数使用相同的模板参数。这个失败可能令人惊讶，因为字符串字面量“hello”可以隐式转换为 std::string

```
1 ::max<std::string>(s, "hello")
```

这样就成功了。

当两个参数从一个公共基类派生出不同的类类型时，推导并不认为该公共基类是推导出的类型的候选者。参见 1.2 节对此问题的讨论和可能的解决方案。

15.8.2 类模板参数

C++17 前，模板参数推导只适用于函数和成员函数模板。特别地，类模板的参数并没有从参数导出到对其构造函数的调用。例如：

```
1 template<typename T>
2 class S {
3     public:
4     S(T b) : a(b) {
5     }
6     private:
7     T a;
8 };
9
10 S x(12); // ERROR before C++17: the class template parameter T was not deduced from
11 // the constructor call argument 12
```

这个限制在 C++17 中取消了，参见第 15.12 节。

15.8.3 默认调用参数

默认函数调用参数可以在函数模板中指定，就像普通函数一样：

```
1 template<typename T>
2 void init (T* loc, T const& val = T())
3 {
4     *loc = val;
5 }
```

这个例子所示，默认函数调用参数可以依赖于模板参数。这种依赖的默认参数只有在没有提供显式参数的情况下，才会实例化——这一原则使得下面的例子有效：

```
1 class S {
2     public:
3     S(int, int);
4 }
5
6 S s(0, 0);
7
8 int main()
9
10{
11    init(&s, S(7, 42)); // T() is invalid for T = S, but the default
12    // call argument T() needs no instantiation
13    // because an explicit argument is given
14}
```

即使不依赖默认调用参数，也不能用它来推导模板参数。所以，以下代码无效：

```
1 template<typename T>
2 void f(T x = 42)
3 {
4
5 int main()
6 {
7    f<int>(); // OK: T = int
8    f(); // ERROR: cannot deduce T from default call argument
9 }
```

15.8.4 异常规范

与默认调用参数一样，异常规范只在需要时才实例化，不参与模板参数推导。例如：

```
1 template<typename T>
2 void f(T, int) noexcept(nonexistent(T())); // #1
3
4 template<typename T>
5 void f(T, ...); // #2 (C-style vararg function)
6 void test(int i)
7 {
8    f(i, i); // ERROR: chooses #1, but the expression nonexistent(T()) is ill-formed
9 }
```

#1 的函数中的 noexcept 规范试图调用一个不存在的函数。通常，在函数模板声明中直接出现这样的错误，会触发模板参数推导失败 (SFINAE)，通过选择 #2 的匹配较小的函数（与省略号参数匹配从重载解析的角度来看，是最糟糕的匹配类型；详见附录 C）但是，由于异常说明不参与模板参数推导，所以重载解析选择 #1，当 noexcept 实例化时，程序就会呈现出一种病态的形式。

相同的规则适用于潜在异常类型的异常规范：

```

1 template<typename T>
2 void g(T, int) throw(typename T::Nonexistent); // #1
3
4 template<typename T>
5 void g(T, ...); // #2
6
7 void test(int i)
8 {
9     g(i, i); // ERROR: chooses #1 , but the type T::Nonexistent is ill-formed
10}

```

然而，这些“动态”异常规范自 C++11 以来已弃用，并在 C++17 中删除。

15.9. 显式函数模板参数

当不能推导函数模板参数时，可以在函数模板名称后面显式地指定。例如：

```

1 template<typename T> T default_value()
2 {
3     return T{};
4 }
5
6 int main()
7 {
8     return default_value<int>();
9 }

```

对于可推导的模板参数也可以这样做：

```

1 template<typename T> void compute(T p)
2 {
3     ...
4 }
5
6 int main()
7 {
8     compute<double>(2);
9 }

```

当显式指定了模板参数，对应参数就不再需要推导了。这允许对函数调用参数进行转换，而这是不能在推导式调用中进行。上面的例子中，`compute<double>(2)` 调用中的参数 2 将隐式转换为 `double`。

可以显式地指定一些模板参数，同时推导出其他参数。但显式指定的参数总是与模板参数从左到右匹配。因此，应该首先指定不能推导（或显式指定）的参数。例如：

```

1 template<typename Out, typename In>
2 Out convert(In p)
3 {
4     ...
5 }

```

```

6
7 int main() {
8     auto x = convert<double>(42); // the type of parameter p is deduced,
9     // but the return type is explicitly specified
10 }

```

有时，可以指定空模板参数列表，以确保所选函数是一个模板实例，同时可以使用推导来确定模板参数：

```

1 int f(int); // #1
2 template<typename T> T f(T); // #2
3 int main() {
4     auto x = f(42); // calls #1
5     auto y = f<>(42); // calls #2
6 }

```

`f(42)` 选择非模板函数，因为重载解析在其他条件都相同的情况下更匹配普通函数。但对于 `f<>(42)`，因模板参数列表的存在，排除了非模板函数（即使没有指定实际的模板参数）。

友元函数声明的上下文中，显式模板参数列表存在一个有趣的效果。看看下面的例子：

```

1 void f();
2 template<typename> void f();
3 namespace N {
4     class C {
5         friend int f(); // OK
6         friend int f<>(); // ERROR: return type conflict
7     };
8 }

```

使用普通标识符命名友元函数时，该函数只会在最近的封闭范围内查找。若找不到，则会在该范围内声明一个新实体（但仍然是“不可见”的，除非通过参数依赖查找（ADL）；参见第 13.2.2 节）。这就是上面第一个友元声明所发生的情况：没有 `f` 在名称空间 `N` 中声明，所以新的 `N::f()` 是“不可见”的声明。

但为友元命名的标识符后面跟着模板参数列表时，此时模板必须通过常规查找可见，而常规查找将向可能需要的任意数量的作用域扩展。因此，上面的第二个声明会找到全局函数模板 `f()`，因为返回类型不匹配（这里没有执行 ADL，由前面友元函数声明创建的声明会忽略），所以是编译器会报错。

使用 SFINAE 原则替换显式指定的模板参数：若替换在即时上下文中导致错误，则丢弃函数模板，但其他模板仍可能成功。例如：

```

1 template<typename T> typename T::EType f(); // #1
2 template<typename T> T f(); // #2
3
4 int main() {
5     auto x = f<int*>();
6 }

```

将 `int*` 替换为候选 #1 中的 `T` 导致替换失败，但在候选 #2 中成功了，因此这就是选中的候选。若在替换后只剩下一个候选函数，那么带有显式模板参数的函数模板行为与普通函数非常相似，包括在许多上下文中衰变为指向函数类型的指针。也就是说，将上面的 `main()` 替换为

```
1 int main() {
2     auto x = f<int*>; // OK: x is a pointer to function
3 }
```

生成一个有效的翻译单元。然而，下面的例子

```
1 template<typename T> void f(T);
2 template<typename T> void f(T, T);
3
4 int main() {
5     auto x = f<int*>; // ERROR: there are two possible f<int*> here
6 }
```

无效，因为 `f<int*>` 在这种情况下没有标识单个函数。

可变参数函数模板也可以与显式的模板参数一起使用：

```
1 template<typename ... Ts> void f(Ts ... ps);
2
3 int main() {
4     f<double, double, int>(1, 2, 3); // OK: 1 and 2 are converted to double
5 }
```

有趣的是，参数包可以部分显式地指定和推导：

```
1 template<typename ... Ts> void f(Ts ... ps);
2
3 int main() {
4     f<double, int>(1, 2, 3); // OK: the template arguments are <double, int, int>
5 }
```

15.10. 初始化式和表达式的推导

C++11 包含了声明变量的能力，该变量的类型是由其初始化式推导出来的，还提供了一种机制来表示命名实体（变量或函数）或表达式的类型。这些工具非常方便，C++14 和 C++17 在这个基础上增加了其他的版本。

15.10.1 `auto` 类型指示符

`auto` 类型说明符可以用于许多地方（主要是命名空间作用域和局部作用域），以从变量的初始化式推导其类型。`auto` 称为占位符类型（另一种占位符类型 `decltype(auto)` 将在第 15.10.2 节介绍）。例如：

```
1 template<typename Container>
2 void useContainer(Container const& container)
3 {
```

```

4 auto pos = container.begin();
5 while (pos != container.end()) {
6     auto& element = *pos++;
7     ... // operate on the element
8 }
9 }
```

上面的例子中，`auto` 的两次使用避免了书写迭代器类型和迭代器的值类型这两种冗长类型的必要：

```

1 typename Container::iterator pos = container.begin();
2 ...
3 typename std::iterator_traits<typename Container::iterator>::reference
4 element = *pos++;
```

`auto` 的推导使用与模板参数推导机制相同。类型指示符 `auto` 取代模板类型参数 `T`，然后继续推导，就好像该变量是一个函数参数，初始化式是相应的函数参数一样。对于第一个 `auto`：

```

1 template<typename T> void deducePos(T pos);
2 deducePos(container.begin());
```

其中 `T` 为 `auto` 的推导类型，结果是 `auto` 类型的变量永远不会是引用类型。示例中还使用了 `auto&`，说明了如何生成对推导类型的引用。这里的推导等价于以下函数模板和调用：

```

1 template<typename T> void deduceElement(T& element);
2 deduceElement(*pos++);
```

元素将始终为引用类型，其初始化式不能产生临时变量。

也可以将 `auto` 和右值引用结合使用，因为推导模型会使其行为类似于转发引用，

```

1 auto&& fr = ...;
```

基于函数模板：

```

1 template<typename T> void f(T&& fr); // auto replaced by template parameter T
```

这解释了下面的例子：

```

1 int x;
2 auto&& rr = 42; // OK: rvalue reference binds to an rvalue (auto = int)
3 auto&& lr = x; // Also OK: auto = int& and reference collapsing makes
4 // lr an lvalue reference
```

这种技术在泛型代码中经常用于绑定值类别(左值与右值)未知的函数或操作符调用的结果，而无需复制结果。例如，在基于范围的 `for` 循环中声明迭代值通常是首选方式：

```

1 template<typename Container> void g(Container c) {
2     for (auto&& x: c) {
3         ...
4     }
5 }
```

这里不知道容器迭代接口的签名，但是通过使用 `auto&&` 可以确定，正在迭代的值不会产生多余的副本。`std::forward<T>()` 可以像往常一样对变量调用，若希望完美地转发绑定值。这就实现了一种“延迟”型完美转发。参见第 11.3 节中的示例。

除了引用之外，还可以结合 `auto` 说明符来创建常量、指针、成员指针等，但 `auto` 必须是声明的“主”类型说明符，不能嵌套在模板参数或类型说明符后面声明符的一部分中。下面的例子说明了各种可能性：

```
1 template<typename T> struct X { T const m; };
2 auto const N = 400u; // OK: constant of type unsigned int
3 auto* gp = (void*)nullptr; // OK: gp has type void*
4 auto const S::*pm = &X<int>::m; // OK: pm has type int const X<int>::*
5 X<auto> xa = X<int>(); // ERROR: auto in template argument
6 int const auto::*pm2 = &X<int>::m; // ERROR: auto is part of the "declarator"
```

C++ 不支持最后一个例子(这并不是技术上的原因)，但 C++ 委员会认为，额外的实现成本和潜在的滥用会超过带来的好处。

为了避免同时搞晕开发者和编译器，在 C++11(以及后来的标准) 中不再允许使用 `auto` 作为“存储类说明符”：

```
1 int g() {
2     auto int r = 24; // valid in C++03 but invalid in C++11
3     return r;
4 }
```

这种 `auto`(继承自 C) 的旧用法总是多余的。大多数编译器通常可以将新用法和旧用法区分为占位符(不必这样做)，提供从旧的 C++ 代码到新 C++ 代码的转换。然而，`auto` 在 C++11 前用的非常少。

推导返回类型

C++14 增加了另一种可以出现可推导的自动占位符类型的情况：函数返回类型。

```
1 auto f() { return 42; }
```

定义一个返回类型为 `int`(类型为 42) 的函数。这也可以用尾部返回类型语法表示：

```
1 auto f() -> auto { return 42; }
```

第一个 `auto` 声明尾部的返回类型，第二个 `auto` 是要推导的占位符类型。

默认情况下，Lambda 存在相同的机制：若没有显式指定返回类型，Lambda 的返回类型会推导为 `auto`：

虽然 C++14 引入了推导返回类型，但 C++11 的 Lambda 已经可以使用了，规范没有使用推导。C++14 中，该规范更新为使用通用的自动推导机制(从开发者的角度来看，两者没有区别)。

```
1 auto lm = [] (int x) { return f(x); };
2 // same as: [] (int x) -> auto f return f(x); g
```

函数声明可以与其定义分开。对于可以推导出返回类型的函数也是如此:

```
1 auto f(); // forward declaration
2 auto f() { return 42; }
```

前置声明在这种情况下用途非常有限，因为定义必须在使用函数的地方可见。提供具有“已解析”返回类型的前置声明无效。例如:

```
1 int known();
2 auto known() { return 42; } // ERROR: incompatible return type
```

由于风格上的偏好，使用推导的返回类型前置声明函数的能力，只在能够将成员函数定义移出类定义时有用:

```
1 struct S {
2     auto f(); // the definition will follow the class definition
3 };
4 auto S::f() { return 42; }
```

可推导的非类型参数

C++17 前，非类型模板参数必须用特定的类型声明，该类型可以是模板参数类型。例如:

```
1 template<typename T, T V> struct S;
2 S<int, 42>* ps;
```

本例中，必须指定非类型模板参数的类型——即除了 42 之外还要指定 int。因此，C++17 增加了声明非类型模板参数的能力，其实际类型由对应的模板参数推导而来。声明如下:

```
1 template<auto V> struct S;
1 S<42>* ps;
```

因为 42 具有 int 类型，所以这里 S<42> 的 V 类型推导为 int 类型。若写成 S<42u>，V 的类型就会推导为 uint(有关推导自动类型说明符的细节，请参阅 15.10.1 节)。

注意，对非类型模板参数类型的常规约束仍然有效。例如:

```
1 S<3.14>* pd; // ERROR: floating-point nontype argument
```

具有这种可推导非类型参数的模板定义，通常还需要表示相应参数的实际类型。这可以使用 decltype 完成(参见第 15.10.2 节)。例如:

```
1 template<auto V> struct Value {
2     using ArgType = decltype(V);
3 };
```

自动非类型模板参数在对类成员的模板进行参数化时也很有用。例如:

```
1 template<typename> struct PMClassT;
2 template<typename C, typename M> struct PMClassT<M C::*> {
3     using Type = C;
4 };
```

```

5
6 template<typename PM> using PMClass = typename PMClassT<PM>::Type;
7 template<auto PMD> struct CounterHandle {
8     PMClass<decltype(PMD)>& c;
9     CounterHandle(PMClass<decltype(PMD)>& c) : c(c) {
10    }
11    void incr() {
12        ++(c.*PMD);
13    }
14 };
15
16 struct S {
17     int i;
18 };
19
20 int main() {
21     S s{41};
22     CounterHandle<&S::i> h(s);
23     h.incr(); // increases s.i
24 }
```

这里，使用了一个助手类模板 PMClassT，使用类模板的部分特化（在第 16.4 节中描述）从成员指针类型的“父”类类型中检索。

可以使用相同的技术来提取关联的成员类型：使用 using Type= M；替换 using Type = C；

对于自动模板参数，只需要指定指向成员的指针常量 &S::i 作为模板参数。C++17 前，还必须指定成员指针类型；这有点像

```
1 OldCounterHandle<int S::*>
```

即笨拙又冗余。

该特性也可以用于非类型参数包：

```

1 template<auto... VS> struct Values {
2 };
3 Values<1, 2, 3> beginning;
4 Values<1, 'x', nullptr> triplet;
```

三元操作符的示例表明，包中的每个非类型参数元素都可以推导出不同的类型。与多变量声明符的情况不同（见 15.10.4 节），不要求所有推导都等价。

若想强制使用同构的非类型模板参数包，可以这样：

```

1 template<auto V1, decltype(V1)... VR> struct HomogeneousValues {
2 };
```

然而，需要模板参数列表不能为空在特定情况下。

使用 auto 作为模板参数类型的完整示例，请参见第 3.4 节。

15.10.2 用 decltype 表示表达式的类型

虽然 auto 避免了写变量类型的需要，但想要使用该变量类型时就没那么容易了。decltype 关键字解决了这个问题，其允许开发者使用精确类型来表示表达式或进行声明。然而，开发者应该注意 decltype 产生类型的差别，这取决于传递的参数是声明还是表达式：

- decltype(e) 中，若 e 是实体（如变量、函数、枚举器或数据成员）或类成员访问的名称，则 decltype(e) 生成该实体声明的类型或指定的类成员。因此，decltype 可以用来检查变量的类型。这用在需要精确匹配现有声明的类型时。例如，有以下变量 y1 和 y2：

```
1 auto x = ...;
2 auto y1 = x + 1;
3 decltype(x) y2 = x + 1;
```

根据 x 的初始化式，y1 的类型可能与 x 相同，也可能不同：它取决于加法操作符的行为。若 x 推导为 int 型，y1 也会是 int 型。若 x 推导为 char 类型，y1 将是 int 类型，因为 char 类型与 1（根据定义为 int 类型）的和是一个 int 类型。在 y2 的类型中使用 decltype(x)，可以确保它与 x 具有相同的类型。

- 否则，若 e 是其他表达式，decltype(e) 会产生一个类型，表示该表达式的类型和值别，如下所示：

- 若 e 是类型 T 的左值，则 decltype(e) 生成 T&。
- 若 e 是类型 T 的 xvalue，则 decltype(e) 生成 T&&。
- 若 e 是类型 T 的 prvalue，则 decltype(e) 生成 T。

有关值类别的详细讨论请参见附录 B。

差别可以通过下面的例子来了解：

```
1 e difference can be demonstrated by the following example:
2 void g (std::string&& s)
3 {
4     // check the type of s:
5     std::is_lvalue_reference<decltype(s)>::value; // false
6     std::is_rvalue_reference<decltype(s)>::value; // true (s as declared)
7     std::is_same<decltype(s), std::string&>::value; // false
8     std::is_same<decltype(s), std::string&&>::value; // true
9
10    // check the value category of s used as expression:
11    std::is_lvalue_reference<decltype((s))>::value; // true (s is an lvalue)
12    std::is_rvalue_reference<decltype((s))>::value; // false
13    std::is_same<decltype((s)), std::string&>::value; // true (T& signals an lvalue)
14    std::is_same<decltype((s)), std::string&&>::value; // false
15 }
```

前四个表达式中，对变量 s 使用了 decltype：

```
1 decltype(s) // declared type of entity e designated by s
```

`decltype` 会产生 `s` 的声明类型，`std::string&&`。最后四个表达式中，因为每种情况下表达式都是 `(s)`，一个带圆括号的名称，所以 `decltype` 构造的操作数不仅是一个名称。这种情况下，类型将反映 `(s)` 的值别：

```
1 decltype((s)) // check the value category of (s)
```

表达式通过名称指向一个变量，因此是左值：

将右值引用类型的参数当作左值，而不是 `xvalue` 是一种安全特性，因为带有名称的参数可以在函数中多次引用。若其是一个 `xvalue`，第一次使用可能会导致值被“移走”。参见第 6.1 节和第 15.6.3 节。

根据上面的规则，`decltype(s)` 是 `std::string` 的普通引用（即左值）（因为 `(s)` 的类型是 `std::string`）。C++ 中除了影响运算符的结合性外，这是少数几个用圆括号括表达式会改变程序含义的地方。

`decltype` 会计算表达式 `e` 的类型，可能会惠及很多方面。具体来说，`decltype(e)` 保留足够的信息对表达式可以描述函数的返回类型，“完美地”返回表达式 `e` 本身：`decltype` 计算表达式的类型，也可以将表达式的值类别传递给函数的调用者。例如，一个简单的转发函数 `g()`，返回调用 `f()` 的结果：

```
1 ??? f();
2
3 decltype(f()) g()
4 {
5     return f();
6 }
```

`g()` 的返回类型取决于 `f()`。如果 `f()` 返回 `int&`，`g()` 的返回类型的计算将首先确定表达式 `f()` 的类型为 `int`。这个表达式是左值，因为 `f()` 返回左值引用，所以 `g()` 声明的返回类型变成 `int&`。类似地，如果 `f()` 的返回类型是右值引用类型，调用 `f()` 将返回一个 `xvalue`，而 `decltype` 将产生一个与 `f()` 返回类型完全匹配的右值引用类型。本质上，这种形式的 `decltype` 采用了表达式的主要特征——类型和值别——并在类型系统中，以一种能够完美转发返回值的方式对它们进行编码。

`decltype` 在 `auto` 推导不足以产生值的情景中也十分有用。例如，有一个未知迭代器类型的变量 `pos`，是某种未知的迭代器类型，我们希望创建一个变量 `element`，该 `element` 可以通过 `pos` 解引用来获取。

```
1 auto element = *pos;
```

然而，这始终都会对元素进行拷贝。如果写成

```
1 auto& element = *pos;
```

将始终接收到该元素的引用，但若迭代器的操作符 `*` 返回一个值，程序将会出错。

在 `auto` 的介绍性示例中使用后一种形式时，我们隐式地假定迭代器生成了对某些底层存储的引用。虽然通常适用于容器迭代器（除了 `vector<bool>` 以外的标准容器），但并非适用于所有迭代器。

为了解决这个问题，可以使用 `decltype` 来保留迭代器操作符 `*` 的值性或引用性：

```
1 decltype(*pos) element = *pos;
```

当迭代器提供的是引用时，就会产生一个引用类型；否则，就会进行值的复制。其主要缺陷是要求初始化式表达式写两次：一次在 `decltype` 中（在那里不求值），一次作为实际的初始化式。C++14 引入了 `decltype(auto)` 构造来解决这个问题，我们将在下文中讨论。

15.10.3 `decltype(auto)`

C++14 添加了一个结合了 `auto` 和 `decltype` 的特性：`decltype(auto)`。与 `auto` 类型说明符一样，它是占位符类型，变量、返回类型或模板参数的类型由关联表达式的类型（初始化式、返回值或模板参数）确定。但与 `auto` 不同的是，`auto` 使用模板参数推导规则来确定感兴趣的类型，实际的类型是通过直接对表达式应用 `decltype` 来确定。下面的例子说明了这一点：

```
1 int i = 42; // i has type int
2 int const& ref = i; // ref has type int const& and refers to i
3
4 auto x = ref; // x has type int and is a new independent object
5
6 decltype(auto) y = ref; // y has type int const& and also refers to i
```

`y` 的类型是通过对初始化式应用 `decltype` 获得的，这里是 `ref`，类型是 `int const&`。但自动类型推导规则产生的却是 `int` 类型。

另一个例子展示了索引 `std::vector`（生成左值）的区别：

```
1 std::vector<int> v = { 42 };
2 auto x = v[0]; // x denotes a new object of type int
3 decltype(auto) y = v[0]; // y is a reference (type int&)
```

这很好地解决了前面例子中冗余的问题：

```
1 decltype(*pos) element = *pos;
```

现在可以重写为

```
1 decltype(auto) element = *pos;
```

返回类型通常也很方便。看看下面的例子：

```
1 template<typename C> class Adapt
2 {
3     C container;
4     ...
5     decltype(auto) operator[] (std::size_t idx) {
6         return container[idx];
7     }
8 };
```

如果 `container[idx]` 产生一个左值，我们希望将这个左值传递给调用者（调用者可能希望获取地址或修改它）：这需要一个左值引用类型，这正是 `decltype(auto)` 解析的。若生成 `prvalue`，则引用类型将导致悬空引用，但 `decltype(auto)` 将针对这种情况生成对象类型（而不是引用类型）。

与 auto 不同， decltype(auto) 不允许修改其类型的说明符或声明符操作符：

```
1 decltype(auto)* p = (void*)nullptr; // invalid
2 int const N = 100;
3 decltype(auto) const NN = N*N; // invalid
```

初始化式中的圆括号可能很重要 (对于 decltype 构造很重要，如第 6.1 节所述)：

```
1 int x;
2 decltype(auto) z = x; // object of type int
3 decltype(auto) r = (x); // reference of type int&
```

圆括号会对 return 语句的有效性产生影响：

```
1 int g();
2 ...
3 decltype(auto) f() {
4     int r = g();
5     return (r); // run-time ERROR: returns reference to temporary
6 }
```

C++17 后， decltype(auto) 也可以用于可推导的非类型参数 (参见第 15.10.1 节)：

```
1 template<decltype(auto) Val> class S
2 {
3     ...
4 };
5 constexpr int c = 42;
6 extern int v = 42;
7 S<c> sc; // #1 produces S<42>
8 S<(v)> sv; // #2 produces S<(int&)v>
```

#1 行中， c 的圆括号的缺失导致可推导参数类型是 c 本身的类型 (int)。因为 c 是值为 42 的常数表达式，所以其等价于 S<42>。#2 行中，圆括号使 decltype(auto) 成为一个引用类型 int&，可以绑定到 int 类型的全局变量 v。通过声明，类模板依赖于对 v 的引用，而 v 值的变化可能会影响类 S 的行为 (详见第 11.4 节)。(S<v>，不带圆括号，则会产生错误，因为 decltype(v) 是 int 类型，因此需要 int 类型的常量参数。但是，v 不是一个 int 型常量值。)

注意，这两种情况的性质有些不同；因此，可以认为这样的非类型模板参数很可能会引起意外，也没有想到这会广泛使用。

最后，给出在函数模板中使用推导非类型参数的注解：

```
1 template<auto N> struct S {};
2 template<auto N> int f(S<N> p);
3 S<42> x;
4 int r = f(x);
```

因为形式为 X<…>(X 是类模板) 用于推导上下文，所以函数模板 f<>() 的参数 N 的类型是根据 S 的非类型参数类型推导出来的。然而，也有许多模式不能这样推导：

```
1 template<auto V> int f(decltype(V) p);
2 int r1 = deduce<42>(42); // OK
3 int r2 = deduce(42); // ERROR: decltype(V) is a nondeduced context
```

`decltype(V)` 是不可推导的上下文: `V` 没有唯一的值与参数 `42` 匹配 (例如, `decltype(7)` 产生与 `decltype(42)` 相同的类型)。因此, 必须显式指定非类型模板参数才能调用此函数。

15.10.4 auto 类型推导的特殊情况

`auto` 的简单推导规则也有一些特殊情况。第一个是当变量的初始化式为初始化器列表的情况, 函数调用的相应推导会失败, 因为无法从初始化列表的参数推导出模板类型参数:

```
1 template<typename T>
2 void deduceT (T);
3 ...
4 deduceT({ 2, 3, 4}); // ERROR
5 deduceT({ 1 }); // ERROR
```

但是, 如果函数有一个更具体的参数

```
1 template<typename T>
2 void deduceInitList(std::initializer_list<T>);
3 ...
4 deduceInitList({ 2, 3, 5, 7 }); // OK: T deduced as int
```

则推导成功。因此, 复制初始化 (使用 `=` 进行初始化) 具有初始化列表的 `auto` 变量可以写为更具体的参数:

```
1 auto primes = { 2, 3, 5, 7 }; // primes is std::initializer_list<int>
2 deduceT(primes); // T deduced as std::initializer_list<int>
```

C++17 前, 相应的 `auto` 变量的直接初始化 (不使用赋值操作符) 也可以这样处理, 但在 C++17 中做了改变, 可以更好地匹配大多数开发者的期望:

```
1 auto oops { 0, 8, 15 }; // ERROR in C++17
2 auto val { 2 }; // OK: val has type int in C++17
```

C++17 前, 两个初始化都有效, 初始化类型 `initializer_list<int>` 的 `oops` 和 `val`。

有趣的是, 为可推导占位符类型的函数, 返回带大括号的初始化列表无效:

```
1 auto subtleError() {
2     return { 1, 2, 3 }; // ERROR
3 }
```

这是因为函数作用域中的初始化列表指向底层数组对象 (具有列表中指定的元素值), 该对象在函数返回时过期。因此, 允许这种结构将出现悬垂引用。

共享相同的 `auto` 时, 另一种特殊情况发生在多个变量声明中, 如下所示:

```
1 auto first = container.begin(), last = container.end();
```

这种情况下, 对每个声明独立执行推导。换句话说, 第一个是新创建的模板类型参数 `T1`, 最后一个是新创建的模板类型参数 `T2`。只有当两个推论都成功, 并且 `T1` 和 `T2` 的推论是相同的类型时, 声明才是良好的。

这个例子没有将 * 放在 auto 处，这样会误导读者认为我们声明了两个指针。另一方面，当在一个声明中声明多个实体时，这些声明的不透明性也是不使用这种方式的一个理由。

```
1 char c;
2 auto *cp = &c, d = c; // OK
3 auto e = c, f = c+1; // ERROR: deduction mismatch char vs. int
```

使用共享的 auto 声明了两对变量。cp 和 d 的声明为 auto 推导出相同的 char 类型，因此这是有效的代码。然而，由于在计算 c+1 时将 char 和 int 提升为 int，e 和 f 的声明会推导出 char 和 int，而这种情况不一致会导致错误。

对于推导的返回类型的占位符，也可能出现类似的特殊情况。考虑下面的例子：

```
1 auto f(bool b) {
2     if (b) {
3         return 42.0; // deduces return type double
4     } else {
5         return 0; // ERROR: deduction conflict
6     }
7 }
```

每个 return 语句都会进行独立的推导，但若推导的类型不同，则程序无效。若返回表达式递归调用函数，则不能进行推导，程序无效（除非之前的推导已经确定了返回类型）。以下代码是无效的：

```
1 auto f(int n)
2 {
3     if (n > 1) {
4         return n*f(n-1); // ERROR: type of f(n-1) unknown
5     } else {
6         return 1;
7     }
8 }
```

但下面的等价代码是可以的：

```
1 auto f(int n)
2 {
3     if (n <= 1) {
4         return 1; // return type is deduced to be int
5     } else {
6         return n*f(n-1); // OK: type of f(n-1) is int and so is type of n*f(n-1)
7     }
8 }
```

推导的返回类型还有另一种特殊情况，在推导变量类型或非类型参数类型中，没有对应的返回类型：

```
1 auto f1() { } // OK: return type is void
2 auto f2() { return; } // OK: return type is void
```

f1() 和 f2() 都是有效的，并且返回类型为 void。但若返回类型模式不能匹配 void，这样是无效的：

```
1 auto* f3() {} // ERROR: auto* cannot deduce as void
```

任何使用推导返回类型的函数模板，都需要立即实例化模板，以确定返回类型。当涉及到 SFINAE(在第 8.4 节和第 15.7 节中描述) 时，这有一个令人惊讶的结果。看看下面的例子：

deduce/resulttypetmpl.cpp

```
1 template<typename T, typename U>
2 auto addA(T t, U u) -> decltype(t+u)
3 {
4     return t + u;
5 }
6
7 void addA(...);
8
9 template<typename T, typename U>
10 auto addB(T t, U u) -> decltype(auto)
11 {
12     return t + u;
13 }
14 void addB(...);
15 struct X {
16 };
17
18 using AddResultA = decltype(addA(X(), X())); // OK: AddResultA is void
19
20 using AddResultB = decltype(addB(X(), X())); // ERROR: instantiation of addB<X>
21 // is ill-formed
```

addB() 使用 decltype(auto)，而不是 decltype(t+u) 会导致重载解析期间的错误：addB() 模板的函数体必须完全实例化，以确定其返回类型。这个实例化并不在调用 addB() 的直接上下文中(参见第 15.7.1 节)，因此不属于 SFINAE，从而会导致一个直接的错误。因此，推导的返回类型不仅仅是复杂的显式返回类型的简写，而且应该小心地使用(理解它们不应该在依赖 SFINAE 属性的其他函数模板中进行调用)。

15.10.5 结构化绑定

C++17 增加了一个新特性，称为结构化绑定。

术语结构化绑定在该特性的原始提议中使用，最终也用于正式规范。简单地说，规范使用了术语分解声明。

用一个小例子就可以展示：

```
1 struct MaybeInt { bool valid; int value; };
```

```
2 MaybeInt g();
3 auto const& [b, N] = g(); // binds b and N to the members of the result of g()
```

对 g() 的调用会产生一个值 (是一个简单的 MaybeInt 类型的类聚合)，可以分解为“元素” (MaybeInt 的数据成员)。该调用产生的值就像括号中的标识符列表 [b, N] 中不同的变量名替换一样。如果该名称为 e，初始化将等价于：

```
1 auto const& e = g();
```

然后中括号中的每个标识符会绑定到 e 的对应元素上。因此，可以认为 [b, N] 就是 e 中标识符的每个名称 (下面会讨论绑定的细节)。

从语法上讲，结构化绑定必须始终具有 auto 类型，可选地通过 const 和/或 volatile 限定符和/或 & 和/或 && 声明符操作符扩展 (但不能使用 * 指针声明符或其他声明符构造)。后面是一个中括号列表，其中至少得有一个标识符 (类似 Lambdas 的“捕获”列表)，再后必须跟一个初始化式。

三种不同的类别可以初始化结构化绑定：

1. 第一种情况是简单类类型，其中所有非静态数据成员都是公共的 (如上面的示例所示)。要在这种情况下应用，所有非静态数据成员必须是公共的 (要么直接在类本身中，要么全部在相同的、明确的公共基类中；不得涉及匿名联合体)。带括号的标识符数量必须等于成员的数量，并且在结构化绑定的范围内使用这些标识符中的一个，就等于使用 e 所表示的对象的相应成员 (具有所有相关的属性；若对应的成员是位域，则不可能取其地址)。
2. 第二种情况对应于数组。下面是一个例子：

```
1 int main() {
2     double pt[3];
3     auto& [x, y, z] = pt;
4     x = 3.0; y = 4.0; z = 0.0;
5     plot(pt);
6 }
```

括号中的初始化式，只是未命名数组变量对应元素的简写。数组元素的数量必须等于括起来的初始化式的数量。

下面是另一个例子：

```
1 auto f() -> int(&)[2]; // f() returns reference to int array
2
3 auto [x, y] = f(); // #1
4 auto& [r, s] = f(); // #2
```

#1 比较特殊：对于这种情况，前面描述的实体 e 可以从下面形式进行推导：

```
1 auto e = f();
```

但这将推导出指向数组的指针，而不是在执行数组的结构化绑定时发生的事情。不过，e 会推导为一个数组类型的变量，对应于初始化式的类型。此后该数组从初始化式中逐个元素拷贝：对于内置数组来说这是个不太寻常的概念。

复制内置数组的另外两个位置是 Lambda 捕获和生成的复制构造函数。

最后，`x` 和 `y` 分别成为表达式 `e[0]` 和 `e[1]` 的别名。

#2 不涉及数组复制，并遵循 `auto` 的通常规则。假设的 `e` 声明如下：

```
1 auto& e = f();
```

这将产生对数组的引用，`x` 和 `y` 再次成为表达式 `e[0]` 和 `e[1]` 的别名（直接引用由调用 `f()` 产生的数组元素的左值）。

3. 最后，第三个选项允许类似于 `std::tuple` 的类使用 `get<i>()` 通过基于模板的协议分解元素。假设 `E` 是表达式 (`e`) 的类型，`e` 如上声明。因为 `E` 是表达式的类型，所以从来不是引用类型。如果表达式 `std::tuple_size<e>::value` 是一个整型常量表达式，必须等于括号内标识符的数量（并且协议生效，优先于数组的第一个选项，但不优先于数组的第二个选项）。用标识符 `n0`、`n1`、`n2` 等表示括号中的标识符。如果 `e` 具有名为 `get` 的成员，则行为就像将如下声明：

```
1 std::tuple_element<i, E>::type& ni = e.get<i>();
```

如果 `e` 推导为具有引用类型，或

```
1 std::tuple_element<i, E>::type&& ni = e.get<i>();
```

如果 `e` 没有成员 `get`，则相应的声明会变成：

```
1 std::tuple_element<i, E>::type& ni = get<i>(e);
```

或

```
1 std::tuple_element<i, E>::type&& ni = get<i>(e);
```

`get` 只会在关联的类和命名空间中查找（`get` 假设为一个模板，因此跟随的 `<` 是一个尖括号（而非小于号））。`std::tuple`、`std::pair` 和 `std::array` 模板都实现了这一接口，代码如下所示：

```
1 #include <tuple>
2
3 std::tuple<bool, int> bi{true, 42};
4 auto [b, i] = bi;
5 int r = i; // initializes r to 42
```

然而，添加 `std::tuple_size`、`std::tuple_element` 和函数模板或成员函数模板 `get<i>()` 并不困难，可以将此机制应用于类或枚举类型。例如：

```
1 #include <utility>
2
3 enum M {};
4
5 template<> class std::tuple_size<M> {
6     public:
7         static unsigned const value = 2; // map M to a pair of values
8     };
9 template<> class std::tuple_element<0, M> {
10    public:
11        using type = int; // the first value will have type int
12    };
13 template<> class std::tuple_element<1, M> {
```

```

14 public:
15     using type = double; // the second value will have type double
16 };
17
18 template<int> auto get(M);
19 template<> auto get<0>(M) { return 42; }
20 template<> auto get<1>(M) { return 7.0; }
21
22 auto [i, d] = M(); // as if: int&& i = 42; double&& d = 7.0;

```

注意，只需要包含 `<utility>` 头文件，就可以使用两个类似元组的访问助手函数 `std::tuple_size<>` 和 `std::tuple_element<>`。

另外，请注意上面的第三种情况（使用类似元组的协议）对括起来的初始化式执行实际的初始化，绑定是实际的引用变量；不仅仅是另一个表达式的别名（不像前两种情况使用简单的类型和数组）。因为引用初始化可能出错，可能会抛出异常，而这个异常目前无法避免。然而，C++ 标准化委员会还讨论了不将标识符与初始化的引用关联起来，让标识符以后的每次使用计算 `get<>()` 表达式的可能性。这允许结构化绑定用于在访问“第二个”值之前，测试“第一个”值的类型（例如 `std::optional`）。

15.10.6 泛型 Lambda

Lambda 已经迅速成为最受欢迎的 C++11 特性，部分原因是简化了 C++ 标准库和许多现代 C++ 库中函数结构的使用，这归功于其简洁的语法。但在模板本身内部，Lambda 可能会变得冗长，因为需要详细说明参数和结果类型。例如，一个从序列中查找第一个负数的函数模板：

```

1 template<typename Iter>
2 Iter findNegative(Iter first, Iter last)
3 {
4     return std::find_if(first, last,
5             [] (typename std::iterator_traits<Iter>::value_type
6                 value) {
7                 return value < 0;
8             });
9 }

```

这个函数模板中，Lambda（到目前为止）最复杂的部分是参数类型。C++14 引入了“泛型”Lambda 的概念，其中一个或多个参数类型使用 `auto` 来推导该类型：

```

1 template<typename Iter>
2 Iter findNegative(Iter first, Iter last)
3 {
4     return std::find_if(first, last,
5             [] (auto value) {
6                 return value < 0;
7             });
8 }

```

Lambda 参数 auto 的处理与使用初始化式的变量类型的 auto 处理相似: 同样由一个引入的模板类型参数 T 来取代。与变量的情况不同, 因为在创建 Lambda 时参数未知, 所以推导不会立即执行。Lambda 本身会变成泛型 (它还没有泛型的话), 并且所引入的模板类型参数添加到它的模板参数列表中。因此, 上面的 Lambda 可以用任何参数类型进行调用, 只要该参数类型支持结果可转换为 bool 的`<0`操作。例如, 这个 Lambda 可以用 int 或 float 值调用。

要理解泛型 Lambda 意味着什么, 首先考虑非泛型 Lambda 的实现模型。给定 Lambda:

```
1 [] (int i) {
2     return i < 0;
3 }
```

C++ 编译器将这个表达式转换为一个新的 Lambda 类实例。这个实例称为闭包或闭包对象, 类称为闭包类型。闭包类型有一个函数操作符, 因此闭包是一个函数对象。

Lambda 的这种转换模型实际上用于 C++ 语言的规范中, 这使得它既方便又准确地描述了语义。捕获的变量成为数据成员, 非捕获 Lambda 到函数指针的转换建模为类中的转换函数等。因为 Lambda 是函数对象, 所以无论何时定义了函数对象的规则, 其也适用于 Lambda。

对于这个 Lambda, 闭包类型如下所示 (简洁起见, 省略了转换函数为指向函数值的指针):

```
1 class SomeCompilerSpecificNameX
2 {
3     public:
4         SomeCompilerSpecificNameX(); // only callable by the compiler
5         bool operator() (int i) const
6     {
7         return i < 0;
8     }
9 }
```

如果检查类型类别中是否有 Lambda, `std::is_class` 将产生 true(参见第 D.2.1 节)。

Lambda 表达式从而导致该类的对象 (关闭类型)。例如:

```
1 foo(...,
2     [] (int i) {
3         return i < 0;
4     });
5 }
```

创建一个内部编译器特定类 `SomeCompilerSpecificNameX` 的对象 (闭包):

```
1 foo(...,
2     SomeCompilerSpecificNameX{}); // pass an object of the closure type
3 }
```

如果 Lambda 要捕获局部变量:

```
1 int x, y;
2 ...
3 [x,y] (int i) {
4     return i > x && i < y;
5 }
```

这些捕获将为相关类类型的初始化成员:

```
1 class SomeCompilerSpecificNameY {
2     private
3         int _x, _y;
4     public:
5         SomeCompilerSpecificNameY(int x, int y) // only callable by the compiler
6             : _x(x), _y(y) {
7         }
8         bool operator()(int i) const {
9             return i > _x && i < _y;
10        }
11    };
```

对于泛型 Lambda，函数调用操作符成为成员函数模板，因此简单泛型 Lambda 为

```
1 [] (auto i) {
2     return i < 0;
3 }
```

会转换成下面的类(同样忽略了函数转换，在泛型 Lambda 场景中是一个转换函数模板):

```
1 class SomeCompilerSpecificNameZ
2 {
3     public:
4         SomeCompilerSpecificNameZ(); // only callable by compiler
5         template<typename T>
6         auto operator()(T i) const
7         {
8             return i < 0;
9         }
10    };
```

成员函数模板在调用闭包时实例化，通常不是在 Lambda 表达式出现的地方。例如:

```
1 #include <iostream>
2
3 template<typename F, typename... Ts> void invoke(F f, Ts... ps)
4 {
5     f(ps...);
6 }
7
8 int main()
9 {
10     invoke([](auto x, auto y) {
11         std::cout << x+y << '\n'
12     },
13     21, 21);
14 }
```

这里，Lambda 表达式出现在 main() 中，并在那里创建了相关的闭包。然而，闭包的调用操作符并没有在此时实例化。invoke() 函数模板的实例化使用闭包类型作为第一个参数类型，int(21) 的类

型)作为第二个和第三个参数类型。调用 invoke 的实例化是用闭包副本调用的(仍然与原始 Lambda 关联的闭包), 实例化闭包的 operator() 模板可用来满足实例化调用 f(ps...)。

15.11. 别名模板

别名模板(参见第 2.8 节)在推导方面是“透明的”。只要一个别名模板带有一些模板参数出现, 别名定义(等号右边的类型)会替换为参数, 产生的结果为推导所用。例如, 模板参数推导对下面的三个调用都会成功:

deduce/aliastemplate.cpp

```
1 template<typename T, typename Cont>
2 class Stack;
3
4 template<typename T>
5 using DequeStack = Stack<T, std::deque<T>>;
6
7 template<typename T, typename Cont>
8 void f1(Stack<T, Cont>);
9
10 template<typename T>
11 void f2(DequeStack<T>);
12
13 template<typename T>
14 void f3(Stack<T, std::deque<T>>); // equivalent to f2
15
16 void test(DequeStack<int> intStack)
17 {
18     f1(intStack); // OK: T deduced to int, Cont deduced to std::deque<int>
19     f2(intStack); // OK: T deduced to int
20     f3(intStack); // OK: T deduced to int
21 }
```

第一次调用(对 f1())中, 在 intStack 类型中使用别名模板 DequeStack 对推导没有影响: 指定的类型 DequeStack<int> 可视为其替代类型 Stack<int, std::deque<int>>。第二次和第三次调用具有相同的涂掉行为, 因为 f2() 中的 DequeStack<T> 和 f3() 中的替代形式与 Stack<T, std::deque<T>> 相同。对模板参数推导的目标来说, 模板别名是透明的: 可以用来区分和简化代码, 但是对于推导如何进行却不会产生效果。

因为别名模板不能特化(参见第 16 章关于模板特化主题的详细信息), 所以这是可能的。假设以下代码可行:

```
1 template<typename T> using A = T;
2 template<> using A<int> = void; // ERROR, but suppose it were possible...
```

因为 A<int> 和 A<void> 都等于 void, 所以不能将 A<t> 与 void 类型匹配, 并得出结论 T 一定是 void。因为别名的每次使用都可以根据其定义进行扩展, 从而使别名可以进行透明地推导。

15.12. 类模板参数推导

C++17 引入了一种新的推导：从变量声明的初始化式或函数表示法类型转换中，指定的参数推导类类型的模板参数。例如：

```
1 template<typename T1, typename T2, typename T3 = T2>
2 class C
3 {
4     public:
5         // constructor for 0, 1, 2, or 3 arguments:
6         C (T1 x = T1{}, T2 y = T2{}, T3 z = T3{});
7         ...
8     };
9
10 C c1(22, 44.3, "hi"); // OK in C++17: T1 is int, T2 is double, T3 is char const*
11 C c2(22, 44.3); // OK in C++17: T1 is int, T2 and T3 are double
12 C c3("hi", "guy"); // OK in C++17: T1, T2, and T3 are char const*
13 C c4; // ERROR: T1 and T2 are undefined
14 C c5("hi"); // ERROR: T2 is undefined
```

注意，所有参数都必须由推导过程或由默认参数确定。不可能显式地指定一些参数去推断其他的参数。例如：

```
1 C<string> c10("hi", "my", 42); // ERROR: only T1 explicitly specified, T2 not deduced
2 C<> c11(22, 44.3, 42); // ERROR: neither T1 nor T2 explicitly specified
3 C<string, string> c12("hi", "my"); // OK: T1 and T2 are deduced, T3 has default
```

15.12.1 推导指引

考虑对 15.8.2 节中前面例子做的一个改动：

```
1 template<typename T>
2 class S {
3     private:
4     T a;
5     public:
6     S(T b) : a(b) {
7     }
8 };
9
10 template<typename T> S(T) -> S<T>; // deduction guide
11
12 S x{12}; // OK since C++17, same as: S<int> x{12};
13 S y(12); // OK since C++17, same as: S<int> y(12);
14 auto z = S{12}; // OK since C++17, same as: auto z = S<int>{12};
```

添加了一个名为推导指引的类模板构造。看起来有点像函数模板，但在语法上与函数模板有一些不同：

- 看起来像尾部返回类型的部分不能写成传统的返回类型，将指定的类型（示例中为 `S<T>`）称为引导类型。

- 没有前置的 auto 关键字来指示尾部的返回类型。
- 推导指引的“名称”必须是前面在同一作用域中，声明的类模板的非限定名称。
- 指引类型必须是一个模板标识，其模板名称对应于指引名称。
- 可以使用显式说明符进行声明。

声明 S x(12); 说明符 S 称为占位符类类型。

注意占位符类型和占位符类类型之间的区别，占位符类型是 auto 或 decltype(auto)，可以解析为任何类型，而占位符类类型是模板名称，只能解析为指定模板的实例的类类型。

当使用这样的占位符时，要声明的变量名必须立即跟在后面，然后必须跟一个初始化式。因此，以下内容无效：

```
1 S* p = &x; // ERROR: syntax not permitted
```

S x(12); 这一声明中，通过将与类 S 相关联的推导指引视为重载集，并尝试使用针对该重载集的初始化式进行重载解析，从而推导出变量的类型。此时，集合中只有一个指引，可成功地推导出 T 为 int，并且指引型为 S<int>。

与普通函数模板推导一样，也可以使用 SFINAE(若在引导类型中替换推导出的参数失败)。这个简单的例子中，情况并非如此。

因此，该引导类型为声明的类型。

注意，若一个类模板名称后面有多个声明符需要推导，那么每个声明符的初始化式必须生成相同的类型：

```
1 S s1(1), s2(2.0); // ERROR: deduces S both as S<int> and S<double>
```

这类似于推导 C++11 占位符类型 auto 时的约束。

在声明的推导指引和类 S 中声明的构造函数 S(T b) 之间有一个隐式连接。但这样的连接不是必需的，推导指引也可以用于聚合类模板：

```
1 template<typename T>
2 struct A
3 {
4     T val;
5 };
6
7 template<typename T> A(T) -> A<T>; // deduction guide
```

若没有推导指引，需要(甚至在 C++17 中)显式指定模板参数：

```
1 A<int> a1{42}; // OK
2 A<int> a2(42); // ERROR: not aggregate initialization
3 A<int> a3 = {42}; // OK
4 A a4 = 42; // ERROR: can't deduce type
```

但根据上面的指引，可以这样写：

```
1 A a4 = { 42 }; // OK
```

这类情况的微妙之处在于，初始化式必须是一个有效的聚合初始化式，必须使用带大括号的初始化列表。因此，以下方式非法：

```
1 A a5(42); // ERROR: not aggregate initialization
2 A a6 = 42; // ERROR: not aggregate initialization
```

15.12.2 隐式推导指引

通常，类模板中的每个构造函数都需要推导指引。这导致类模板参数推导的设计者为推导设计了一种隐式机制。这相当于为主类模板的每个构造函数和构造函数模板引入一个隐式推导指引，如下所示：

第 16 章介绍了以各种方式“特化”类模板的能力。此类特化不参与类模板参数推导。

- 隐式知道的模板参数列表包括类模板的模板参数，在构造函数模板的情况下，后面是构造函数模板的模板参数。构造函数模板的模板参数保留默认参数。
- 指引中的“类函数”参数从构造函数或构造函数模板中复制。
- 指引类型是模板的名称，参数是从类模板中获取的模板参数。

应用到原始的简单类模板上：

```
1 template<typename T>
2 class S {
3     private:
4     T a;
5     public:
6     S(T b) : a(b) {
7     }
8 };
```

模板参数列表是 typename T，类函数参数列表变成 (T b)，所以引导类型是 S<T>。因此，得到了一个与之前编写的用户声明指引等价的指引类型：该指引不需要达到我们预期的效果！只使用最初编写的简单类模板（没有推导指引），就可以有效地编写 S x(12)；预期的结果是 x 为 S<int>。

推导指引有一种模糊性。再次考虑简单类模板 S 和以下初始化：

```
1 S x{12}; // x has type S<int>
2 S y{x};
3 S z(x);
```

已经知道 x 的类型是 S<int>，但是 y 和 z 的类型应该是什么？直观产生的两种类型是 S<S<int>> 和 S<int>。委员会决定，两种情况下都应该是 S<int>，这有点争议。为什么会引起争议？考虑一个类似的 vector 类型的例子：

```
1 std::vector v{1, 2, 3}; // vector<int>, not surprising
2 std::vector w2{v, v}; // vector<vector<int>>
3 std::vector w1{v}; // vector<int>!
```

换句话说，一个和多个元素带大括号的初始化式的初始化推导不同。通常情况下，一个元素的结果是所希望的，但二者确实不同。然而，在泛型代码中，很容易忽略以下差别：

```
1 template<typename T, typename... Ts>
2 auto f(T p, Ts... ps) {
3     std::vector v{p, ps...}; // type depends on pack length
4     ...
5 }
```

若 T 推导为一个 vector 类型，那么 v 的类型将根据 ps 是空包或非空包而推导出不同的类型。

隐式模板指引本身的添加并非没有争议。反对者的主要论点是，该特性会自动向现有库添加接口。要理解这一点，再次考虑上面简单的类模板 S。自模板在 C++ 中引入以来，定义一直有效。但假设 S 的作者扩充了库，使得 S 的定义更加精细：

```
1 template<typename T>
2 struct ValueArg {
3     using Type = T;
4 };
5
6 template<typename T>
7 class S {
8     private:
9     T a;
10    public:
11     using ArgType = typename ValueArg<T>::Type;
12     S(ArgType b) : a(b) {
13     }
14 };
```

C++17 前，这样的转换（并不少见）不会影响现有的代码。然而，在 C++17 中禁用了隐式推导指引。为了了解这一点，编写一个对应于上述隐式推导指引构造过程产生的推导指引：模板参数列表和导向类型不变，但类函数参数现在是 ArgType，为 typename ValueArg<T>::Type：

```
1 template<typename> S(typename ValueArg<T>::Type) -> S<T>;
```

回想一下 15.2 节，像 ValueArg<T>::这样的名称限定符不能进行上下文的推导。因此，这种形式的推导指引无效，不能解析像 S x(12); 这样的声明。换句话说，库编写者执行这种转换很可能会破坏 C++17 的代码。

库作者该怎么做？建议仔细考虑每个构造函数是否希望在库的生命周期中，将其作为隐式推导指引的来源。如果不是，则用 typename ValueArg<X>::Type 替换类型为 X（可推导构造函数参数）的每个实例。但是，没有更简单的方法可以“退出”隐式推导指引。

15.12.3 其他要点

注入类名

考虑下面的例子：

```

1 template<typename T> struct X {
2     template<typename Iter> X(Iter b, Iter e);
3     template<typename Iter> auto f(Iter b, Iter e) {
4         return X(b, e); // What is this?
5     }
6 };

```

这段代码在 C++ 4 中是有效的: `X(b, e)` 中的 `X` 是注入类名, 在这个上下文中等价于 `X<T>`(参见第 13.2.3 节)。然而, 类模板参数推导的规则会使 `X` 等价于 `X<iter>`。

为了保持向后兼容性, 若模板的名称是注入类名, 则禁用类模板参数进行推导。

转发引用

考虑下面的例子:

```

1 template<typename T> struct Y {
2     Y(T const&);
3     Y(T&&);
4 };
5 void g(std::string s) {
6     Y y = s;
7 }

```

这里的目的是通过与复制构造函数关联的隐式推导指引, 推断 `T` 为 `std::string`。然而, 将隐式推导指引写成显式声明的指引则会令人吃惊:

```

1 template<typename T> Y(T const&) -> Y<T>; // #1
2 template<typename T> Y(T&&) -> Y<T>; // #2

```

回想一下第 15.6 节中, `T&&` 的行为在模板参数推导期间的特殊行为: 作为转发引用, 若对应的参数是左值, 会导致 `T` 推导为引用类型。上面的例子中, 推导过程中的参数是表达式 `s`, 是一个左值。隐式指引 #1 推导 `T` 是 `std::string`, 但要求将参数从 `std::string` 调整为 `std::string const`。然而, 指引 #2 通常会推断 `T` 是一个引用类型 `std::string&`, 并产生相同类型的参数(引用折叠规则), 这是一个更好的匹配, 不需要添加 `const` 来进行类型调整。

这个结果将相当令人惊讶, 并且可能会导致实例化错误(不允许引用类型的上下文中使用类模板参数), 或者更糟糕的是, 会静默产生行为错误的实例化(产生悬空引用)。

因此, C++ 标准化委员会决定在隐式推导指引执行推导时, 禁用 `T` 的特殊推导指引。若 `T` 最初是一个类模板参数(与构造函数模板参数相反; 对于这些, 特殊的推导规则仍然存在), 如上例推断 `T` 为 `std::string` 那样。

explicit 关键字

推导指引可以用关键字 `explicit` 声明。然后, 只在直接初始化的情况下使用:

```

1 template<typename T, typename U> struct Z {
2     Z(T const&);
3     Z(T&&);
4 };

```

```

5
6 template<typename T> Z(T const&) -> Z<T, T&>; // #1
7 template<typename T> explicit Z(T&&) -> Z<T, T>; // #2
8
9 Z z1 = 1; // only considers #1 ; same as: Z<int, int&> z1 = 1;
10 Z z2{2}; // prefers #2 ; same as: Z<int, int> z2{2};

```

注意，`z1` 的初始化是复制初始化。因为它是显式声明的，所以推导指引 #2 没用到。

复制构造和初始化列表

考虑下面的类模板：

```

1 template<typename ... Ts> struct Tuple {
2     Tuple(Ts...);
3     Tuple(Tuple<Ts...> const&);
4 };

```

为了理解隐式指引的作用，把它们写成显式声明：

```

1 template<typename... Ts> Tuple(Ts...) -> Tuple<Ts...>;
2 template<typename... Ts> Tuple(Tuple<Ts...> const&) -> Tuple<Ts...>;

```

现在来看一些例子：

```

1 auto x = Tuple{1,2};

```

这显然选择了第一个指引，也选择了第一个构造函数：因此 `x` 是一个 `Tuple<int, int>` 类型。继续看一些使用复制 `x` 的例子：

```

1 Tuple a = x;
2 Tuple b(x);

```

对于 `a` 和 `b`，两个指引都匹配。第一个指引选择类型 `Tuple<tuple<int, int>>`，而与复制构造函数相关联的指引生成 `Tuple<int, int>`。幸运的是，第二个指引匹配得更好，因此 `a` 和 `b` 都从 `x` 复制构建。

现在，考虑一些使用带大括号初始化列表的例子：

```

1 Tuple c{x, x};
2 Tuple d{x};

```

第一个示例 (`x`) 只能匹配第一个指引，因此产生 `Tuple<Tuple<int, int>, Tuple<int, int>>`。这表明，第二个示例应该推导 `d` 为 `Tuple<tuple<int>>` 类型。不过，其使用了复制的方式进行构造(首选第二个隐式指引)。这也发生在函数功能符类型的转换中：

```

1 auto e = Tuple{x};

```

这里，`e` 推导为一个 `Tuple<int, int>`，而不是 `Tuple<Tuple<int>>`。

指引仅为推导所用

推导指引不是函数模板：只用于演绎模板参数，不可“调用”。通过引用传递参数和通过值传递参数之间的差异，对于指引声明并不重要。例如：

```

1 template<typename T> struct X {
2     ...
3 };
4
5 template<typename T> struct Y {
6     Y(X<T> const&);
7     Y(X<T>&&);
8 };
9
10 template<typename T> Y(X<T>) -> Y<T>;

```

请注意推导指引与 `y` 的两个构造函数并不完全对应。这并不重要，因为该指引只用于推导。给定值 `xtt` 的类型为 `X<TT>`——lvalue 或 rvalue——将选择推导出的类型 `Y<TT>`。然后，初始化将对 `Y<TT>` 的构造函数执行重载解析，以决定调用哪一个 (这将取决于 `xtt` 是左值还是右值)。

15.13. 后记

函数模板的模板参数推导是最初 C++ 设计的一部分。事实上，由显式模板参数提供的替代方法，直到许多年后才成为 C++ 的一部分。

SFINAE 是本书第一版中介绍的术语，很快在整个 C++ 编程社区中变得流行起来。在 C++98 中，SFINAE 并不像现在这样强大：只应用于有限的类型操作集，并没有涵盖任意表达式或访问控制。随着越来越多的模板技术开始依赖 SFINAE(见第 19.4 节)，推广 SFINAE 条件的必要性变得明显。Steve Adamczyk 和 John Spicer 在 C++11 中 (通过 N2634) 进行了开发实现。尽管标准中的措辞变化相对较小，但在一些编译器中实现的工作量却大得不成比例。

`auto` 类型说明符和 `decltype` 构造最早在 C++03 引入，并成为 C++11 的一部分。他们的发展是由 Bjarne Stroustrup 和 Jaakko Järvi 牵头的 (参见他们的 N1607 关于 `auto` 类型说明符和 N2343 关于 `decltype`)。

Stroustrup 在最初的 C++ 实现 (称为 Cfront) 中已经考虑了 `auto` 语法。当这个特性在 C++11 中引入时，`auto` 作为存储说明符 (继承自 C 语言) 的原始含义保留了下来，并且消除歧义规则决定了关键字应该如何解释。当爱迪生设计团队的前端实现这个特性时，David Vandevoorde 发现这个规则可能会让 C++11 程序员 (N2337) 感到意外。研究了这个问题之后，标准化委员会通过论文 N2546(David Vandevoorde 和 Jens Maurer) 决定完全放弃 `auto` 的传统使用 (C++03 中使用关键字 `auto` 的地方，也可以忽略)。这是一个不寻常的先例，即从该语言中删除一个特性，而不先反对它，但后来证明这是一个正确的决定。

GNU 的 GCC 编译器接受了与 `decltype` 特性类似的扩展类型，开发者发现它在模板编程中很有用。但其是在 C 语言环境中开发的特性，并不完全适合 C++。因此，C++ 委员会既不能合并它，也不修改它，因为那样会破坏依赖于 GCC 行为的现有代码。这就是为什么 `decltype` 不拼写为 `typeof`。Jason Merrill 和其他人已经提出了强有力的论点，认为使用不同的操作符会更好，而不是使用现在 `decltype(x)` 和 `decltype((x))` 不同的方式，但他们没有足够的说服力来改变最终的规范。

C++17 中使用 `auto` 声明非类型模板参数的能力主要是由 Mike Spertus 在 James Touton、David Vandevoorde 和其他人一起开发的。该特性的更改在 P0127R2 中提出。尚不清楚是否有意使用

`decltype(auto)` 代替 `auto` 成为该语言的一部分 (显然，委员会未对此进行讨论，但这超出了规范的范畴)。

Mike Spertus 还推动了 C++17 中类模板参数推导的开发，Richard Smith 和 Faisal Vali 贡献了重要的技术思想 (包括推导指引的思想)。P0091R3 是会纳入了下一个语言标准的工作论文。

结构化绑定主要由 Herb Sutter 驱动，他与 Gabriel Dos Reis 和 Bjarne Stroustrup 一起撰写了 P0144R1，提出了该特性。委员会讨论期间进行了许多调整，包括使用括号来分隔分解标识符。Jens Maurer 将该提案转化为标准的最终规范 (P0217R3)。

第 16 章 模板的特化和重载

我们已经了解了 C++ 模板如何将泛型定义扩展为一系列相关的类、函数或变量。尽管这是一种强大的机制，但对于模板参数的特定替换，操作的泛化形式远非最佳。

支持泛型编程的其他流行编程语言中，C++ 有些独特，因为它具有丰富的特性，支持用更特殊方式替换泛型定义。本章中，将研究两种 C++ 语言机制，它们允许泛化的特殊形式：模板特化和函数模板重载。

16.1. 不完全契合的“泛型代码”

考虑下面的例子：

```
1 template<typename T>
2 class Array {
3     private:
4     T* data;
5     ...
6     public:
7     Array(Array<T> const&);
8     Array<T>& operator= (Array<T> const&);
9     void exchangeWith (Array<T>* b) {
10        T* tmp = data;
11        data = b->data;
12        b->data = tmp;
13    }
14    T& operator[] (std::size_t k) {
15        return data[k];
16    }
17    ...
18 };
19
20 template<typename T> inline
21 void exchange (T* a, T* b)
22 {
23     T tmp(*a);
24     *a = *b;
25     *b = tmp;
26 }
```

对于简单类型，`exchange()` 的泛型实现工作得很好。但对于具有昂贵复制操作的类型，泛型实现可能要比定制实现的开销大得多。示例中，泛型实现需要调用一次 `Array<T>` 的复制构造函数，并调用两次其复制赋值操作符。对于大型数据结构，通常涉及复制较大的内存。然而，`exchangeWith()` 可以通过交换内部数据指针来替换。

16.1.1 定制化

成员函数 `exchangeWith()` 提供了通用的 `exchange()` 函数的有效替代，但需要使用不同的函数在以下情况下不是很方便：

1. `Array` 类的用户必须记住多个接口，并且需要小心地使用。
2. 泛型算法通常不能区分各种可能性。例如：

```
1 template<typename T>
2 void genericAlgorithm(T* x, T* y)
3 {
4     ...
5     exchange(x, y); // How do we select the right algorithm?
6     ...
7 }
```

C++ 模板提供了自定义函数模板和类模板的方法。对于函数模板，这是通过重载机制实现的。例如，可以重载 `quickExchange()` 函数模板：

```
1 template<typename T>
2 void quickExchange(T* a, T* b) // #1
3 {
4     T tmp(*a);
5     *a = *b;
6     *b = tmp;
7 }
8
9 template<typename T>
10 void quickExchange(Array<T>* a, Array<T>* b) // #2
11 {
12     a->exchangeWith(b);
13 }
14
15 void demo(Array<int>* p1, Array<int>* p2)
16 {
17     int x=42, y=-7;
18     quickExchange(&x, &y); // uses #1
19     quickExchange(p1, p2); // uses #2
20 }
```

对 `quickExchange()` 的第一个调用有两个 `int*` 类型的参数，因此当用 `int` 替换 `T` 时，只有在第一个模板 (#1 声明) 上推导成功。相比之下，第二个调用可以与任意一个模板匹配：调用 `quickExchange(p1, p2)` 的函数是在第一个模板中用 `Array<int>` 替换 `T` 时获得，第二个模板中用 `int` 替换时也可获得。这两次替换的结果是，函数参数类型与第二次调用的参数类型完全匹配。可以得出这样的结论：调用有歧义，但是 C++ 认为第二个模板比第一个模板“更加特化”。在其他条件相同的情况下，重载解析更倾向于使用更特化的模板，因此选择了 #2 处的模板。

16.1.2 语义透明

上一节中所示的重载使用，用于实现实例化过程的定制化，但这种“透明”在很大程度上取决于实现细节。考虑 quickExchange() 的解决方案，虽然泛型算法和为 Array<T> 类型定制的算法最终都会交换指向的值，但操作的副作用不同。通过考虑一些比较 struct 对象交换和 Array<T> 交换的代码，可以说明这一点：

```
1 struct S {
2     int x;
3 } s1, s2;
4
5 void distinguish (Array<int> a1, Array<int> a2)
6 {
7     int* p = &a1[0];
8     int* q = &s1.x;
9     a1[0] = s1.x = 1;
10    a2[0] = s2.x = 2;
11    quickExchange(&a1, &a2); // *p == 1 after this (still)
12    quickExchange(&s1, &s2); // *q == 2 after this
13 }
```

调用 quickExchange() 之后，指向第一个数组的指针 p 变成了指向第二个数组的指针。即使在交换操作之后，指向非数组 s1 的指针仍然指向 s1：只交换所指向的值。前缀 quick 有助于使用快捷方式来实现所需的操作，但通用 exchange() 模板仍然可以对 Array<T> 进行优化：

```
1 template<typename T>
2 void exchange (Array<T>* a, Array<T>* b)
3 {
4     T* p = &(*a)[0];
5     T* q = &(*b)[0];
6     for (std::size_t k = a->size(); k-- != 0; ) {
7         exchange(p++, q++);
8     }
9 }
```

该版本与泛型代码相比的优点是不需要(可能)临时 Array<T>。exchange() 模板可递归调用，因此对于 Array<Array<char>> 这样的类型，也有良好的性能。更特化的模板版本通常不声明为内联，因为其做了相当多的工作，而原始的泛型实现是内联的，因为只进行了少量的操作。

16.2. 重载函数模板

上一节中，了解了名称相同的两个函数模板可以共存，即使以相同的参数类型实例化。下面是另一个例子：

deduce/funcoverload1.hpp

```
1 template<typename T>
2 int f(T)
```

```

3 {
4     return 1;
5 }
6
7 template<typename T>
8 int f(T*)
9 {
10    return 2;
11 }

```

第一个模板中用 `int*` 替换 `T` 时，得到的函数与在第二个模板中用 `int` 替换 `T` 得到的函数具有相同的参数(和返回)类型。不仅这些模板可以共存，它们的实例化也可以共存，即使具有相同的参数和返回类型。

下面演示了如何使用显式模板参数语法，调用这两个生成的函数(假设前面对模板进行了声明)：

deduce/funcoverload1.cpp

```

1 #include <iostream>
2 #include "funcoverload1.hpp"
3 int main()
4 {
5     std::cout << f<int*>((int*)nullptr); // calls f<T>(T)
6     std::cout << f<int>((int*)nullptr); // calls f<T>(T*)
7 }

```

程序输出：

12

为了说明，这里详细分析一下 `f<int*>((int*)nullptr)` 的调用。语法 `f<int*>()` 表示用 `int*` 替换模板 `f()` 的第一个模板参数。这里不止一个模板 `f()`，因此创建了一个重载集，其中包含从模板生成的两个函数：`f<int*>(int*)(从第一个模板生成)` 和 `f<int*>(int**)(从第二个模板生成)`。调用 `(int*)nullptr` 的参数类型为 `int*`，这匹配从第一个模板生成的函数。

对于第二次调用，创建的重载集包含 `f<int>(int)(从第一个模板生成)` 和 `f<int>(int*)(从第二个模板生成)`，因此只有第二个模板匹配。

16.2.1 签名

如果两个函数有不同的签名，可以在一个程序中共存。函数签名的定义如下所示：

这个定义与 C++ 标准中给出的定义不同，但其结果等价。

1. 函数的非限定名(或生成函数模板的名称)
2. 名称在类或命名空间作用域中，若该名称具有内部链接，则声明为该名称的翻译单元

3. 函数的 const、volatile 或 const volatile 限定符 (若具有这样限定符的成员函数)
4. 函数的 & 或 && 限定符 (若具有这样限定符的成员函数)
5. 函数参数的类型 (若函数是由函数模板生成, 则指的是替换前的模板参数)
6. 函数是由函数模板生成的, 则包括它的函数返回类型
7. 若函数是从函数模板中生成, 则包括模板参数和模板实参

下面的模板和实例化体可以在一个程序中共存:

```

1 template<typename T1, typename T2>
2 void f1(T1, T2);
3
4 template<typename T1, typename T2>
5 void f1(T2, T1);
6
7 template<typename T>
8 long f2(T);
9
10 template<typename T>
11 char f2(T);

```

当定义在相同的作用域中时, 实例化会产生重载歧义, 所以不能总一起使用。例如, 调用 f2(42) 对于上面声明的模板会产生歧义:

```

1 #include <iostream>
2 template<typename T1, typename T2>
3 void f1(T1, T2)
4 {
5     std::cout << "f1(T1, T2)\n";
6 }
7
8 template<typename T1, typename T2>
9 void f1(T2, T1)
10 {
11     std::cout << "f1(T2, T1)\n";
12 }
13
14 // fine so far
15 int main()
16 {
17     f1<char, char>('a', 'b'); // ERROR: ambiguous
18 }

```

这里, 函数

```
f1<T1 = char, T2 = char>(T1, T2)
```

可以与下面函数共存

```
f1<T1 = char, T2 = char>(T2, T1)
```

但是重载解析永远无法判断出哪一个更合适。若模板在不同的编译单元中出现, 这两个实例化实际上可以在同一个程序中共存(并且, 链接器不会抱怨重复的定义, 这是因为实例化的签名不同):

```

1 // translation unit 1:
2 #include <iostream>
3
4 template<typename T1, typename T2>
5 void f1(T1, T2)
6 {
7     std::cout << "f1(T1, T2)\n";
8 }
9
10 void g()
11 {
12     f1<char, char>('a', 'b');
13 }
14
15 // translation unit 2:
16 #include <iostream>
17
18 template<typename T1, typename T2>
19 void f1(T2, T1)
20 {
21     std::cout << "f1(T2, T1)\n";
22 }
23
24 extern void g(); // defined in translation unit 1
25
26 int main()
27 {
28     f1<char, char>('a', 'b');
29     g();
30 }

```

该程序有效，其输出如下：

```
f1(T2, T1)
f1(T1, T2)
```

16.2.2 重载函数模板的部分排序

重新考虑前面的例子：发现在替换了给定的模板参数列表 (`<int*>` 和 `<int>`) 后，重载解析最终调用了正确的函数：

```
std::cout << f<int*>((int*)nullptr); // calls f<T>(T)
std::cout << f<int>((int*)nullptr); // calls f<T>(T*)
```

因为模板参数推导发挥了作用，即使没有提供显式模板参数，函数也会选中。稍微修改一下前面例子中的 main():

deduce/funcoverload2.cpp

```
1 #include <iostream>
2
3 template<typename T>
4 int f(T)
5 {
6     return 1;
7 }
8
9 template<typename T>
10 int f(T*)
11 {
12     return 2;
13 }
14
15 int main()
16 {
17     std::cout << f(0); // calls f<T>(T)
18     std::cout << f(nullptr); // calls f<T>(T)
19     std::cout << f((int*)nullptr); // calls f<T>(T*)
20 }
```

考虑第一个调用，`f(0)`: 参数类型是 `int`，若用 `int` 替换 `T`，其与第一个模板的参数类型匹配。但第二个模板的参数类型始终是指针，在推导之后，会从第一个模板生成的实例是候选，重载解析在这没什么用。

`f(nullptr)`: 参数的类型是 `std::nullptr_t`，也只能匹配第一个模板。

第三个调用 (`f(int*)nullptr`) 更有趣：两个模板都成功推导了参数，产生了函数 `f<int*>(int*)` 和 `f<int>(int*)`。从重载解析的角度来看，使用 `int*` 参数调用这两个函数都是同样的，从而具有歧义（参见附录 C）。这里，重载解析标准起了作用：选择从更特化的模板生成的函数。因为，第二个模板更特化，所以输出为

112

16.2.3 正式的排序规则

上一个示例中，因为第一个模板可以接受任何参数类型，所以第二个模板似乎比第一个模板更特化，而第二个模板只接受指针类型。然而，其他例子并不一定如此直观。接下来，描述了确定参与重载集的函数模板，是否比另一个更特化的过程。这些是部分排序规则：有可能给定两个模板，其中一个比另一个更特化。若重载解析必须在两个这样的模板之间进行选择，则无法做出决定，并且程序包含一个模糊错误。

假设正在比较两个名称相同的函数模板，对于给定的函数调用似乎可行。重载解析的决策如下：

- 函数调用参数中没有使用的默认参数和省略号参数在后续将忽略。
- 然后，通过如下替换每个模板参数，构造出两个类型列表（对于转换函数模板，是返回类型）：
 1. 将每个模板类型参数替换为一个独特的虚拟类型。
 2. 将每个双重模板参数替换为一个独特的虚拟类模板。
 3. 将每个非类型模板参数替换为创建的相应类型的值。

（虚构出的类型、模板和值在这一上下文中都与任何其他的类型、模板或值不同，这些其他的类型、模板或值要么是开发者使用的，要么是编译器在其他上下文中合成。）

- 若第二个模板对第一个合成的参数类型列表中的模板参数推导，实现了精确匹配（反之不行），那么第一个模板比第二个模板更特化。相反，若第一个模板对第二个合成参数类型列表的模板参数推导成功实现精确匹配（反之不行），那么第二个模板比第一个模板更特化。

将其应用到上一个示例中的两个模板中，从而更直观。两个模板通过替换模板参数合成两个参数类型列表：(A1) 和 (A2*)（其中 A1 和 A2 是唯一的组成类型）。通过将 A2* 替换为 T，第一个模板对第二个参数类型列表的推导成功了，但没有办法使第二个模板的 T* 匹配第一个列表中的非指针类型 A1。因此得出结论，第二个模板比第一个更特化。

考虑一个涉及多个函数参数的例子：

```
1 template<typename T>
2 void t(T*, T const* = nullptr, ...);
3
4 template<typename T>
5 void t(T const*, T*, T* = nullptr);
6
7 void example(int* p)
8 {
9     t(p, p);
10 }
```

因为实际调用不使用第一个模板的省略号参数，而第二个模板的最后一个参数由其默认参数替代，所以这些参数在部分排序中忽略。注意没有使用第一个模板的默认参数；因此，相应的参数参与了排序。

参数类型的合成列表为 (A1*, A1 const*) 和 (A2 const*, A2*)。与第二个模板相比，(A1*, A1 const*) 的模板参数推导成功地将 T 替换为 A1 const，因为需要进行限定调整，所以无法进行精确匹配，使用类型为 (A1*, A1 const*) 的参数调用 $T < A1 \text{ const} > (A1 \text{ const}^*, A1 \text{ const}^*, A1 \text{ const}^* = 0)$ 。类似地，通过从参数类型列表 (A2 const*, A2*) 推导出第一个模板的参数，也找不到精确匹配。因此，两个模板之间没有排序关系，并且调用存在歧义。

正式的排序规则通常会直接选择函数模板。然而，有时会出现规则与直觉不符的例子。因此，将来可能会修改规则，从而适用于所有例子。

16.2.4 模板和非模板

函数模板可以用非模板函数重载。在其他条件相同的情况下，选择要调用的函数时，会首选非模板函数：

details/nontmpl1.cpp

```
1 #include <string>
2 #include <iostream>
3
4 template<typename T>
5 std::string f(T)
6 {
7     return "Template";
8 }
9
10 std::string f(int&)
11 {
12     return "Nontemplate";
13 }
14
15 int main()
16 {
17     int x = 7;
18     std::cout << f(x) << '\n' ; // prints: Nontemplate
19 }
```

输出为

```
Nontemplate
```

但当 `const` 限定符和引用限定符不同时，重载解析的优先级可能会改变。例如：

details/nontmpl2.cpp

```
1 #include <string>
2 #include <iostream>
3
4 template<typename T>
5 std::string f(T&)
6 {
7     return "Template";
8 }
9
10 std::string f(int const&)
11 {
12     return "Nontemplate";
13 }
```

```

14
15 int main()
16 {
17     int x = 7;
18     std::cout << f(x) << '\n' ; // prints: Template
19     int const c = 7;
20     std::cout << f(c) << '\n' ; // prints: Nontemplate
21 }

```

程序输出如下：

```

Template
Nontemplate

```

函数模板 $f <T>(T\&)$ 在传递非常量 int 时，匹配的更好。对于 int 来说，实例化的 $f <int\&>(int\&)$ 比 $f <int\&>(int\& const\&)$ 匹配得更好。因此，区别不仅仅在于是否是函数是模板，可以使用重载解析的一般性规则（见第 C.2 节）。只有对 int const 调用 $f()$ 时，两个签名才具有相同的 int const& 类型，因此首选非模板函数。

因此，将成员函数模板声明为

```

1 template<typename T>
2 std::string f(T const&)
3 {
4     return "Template";
5 }

```

当成员函数定义接受与复制或移动构造函数相同的参数时，很容易发生意外。例如：

details/tmp1constr.cpp

```

1 #include <string>
2 #include <iostream>
3
4 class C {
5     public:
6     C() = default;
7     C(C const&) {
8         std::cout << "copy constructor\n";
9     }
10    C(C&&) {
11        std::cout << "move constructor\n";
12    }
13    template<typename T>
14    C(T&&) {
15        std::cout << "template constructor\n";
16    }
17 };

```

```
18
19 int main()
20 {
21     C x;
22     C x2{x}; // prints: template constructor
23     C x3{std::move(x)}; // prints: move constructor
24     C const c;
25     C x4{c}; // prints: copy constructor
26     C x5{std::move(c)}; // prints: template constructor
27 }
```

程序输出如下：

```
template constructor
move constructor
copy constructor
template constructor
```

因此，成员函数模板比复制构造函数更适合复制 C。对于 std::move(c)，会产生类型 C 的 const&&(这是一种可能的类型，但通常没有有意义的语义)，成员函数模板也比移动构造函数更好。

当成员函数模板可能隐藏复制或移动构造函数时，通常必须部分禁用这些成员函数模板。这在第 6.4 节中有解释。

16.2.5 可变参数函数模板

可变参数函数模板(参见第 12.4 节)在部分排序过程中需要特殊处理，参数包的推导(在第 15.5 节中描述)会将单个参数与多个参数匹配。这种行为为函数模板排序引入了几种有趣的情况：

details/variadicoverload.cpp

```
1 #include <iostream>
2
3 template<typename T>
4 int f(T*)
5 {
6     return 1;
7 }
8
9 template<typename... Ts>
10 int f(Ts...)
11 {
12     return 2;
13 }
```

```

15 template<typename... Ts>
16 int f(Ts*...)
17 {
18     return 3;
19 }
20
21 int main()
22 {
23     std::cout << f(0, 0.0); // calls f<>(Ts...)
24     std::cout << f((int*)nullptr, (double*)nullptr); // calls f<>(Ts*...)
25     std::cout << f((int*)nullptr); // calls f<>(T*)
26 }

```

这个示例的输出是

231

第一个调用 `f(0, 0.0)` 中, 将考虑名为 `f` 的函数模板。对于第一个函数模板 `f(T*)`, 推导失败的原因有两个: 一是无法推导出模板参数 `T`, 二是这个非变参函数模板的函数实参比形参多。第二个函数模板 `f(Ts...)` 可变: 推导将函数参数包 `(Ts)` 的模式与两个参数的类型 (分别为 `int` 和 `double`) 进行比较, 将 `Ts` 推导为 `(int, double)`。对于第三个函数模板 `f(Ts*...)`, 将函数参数包 `Ts*` 的模式与每个参数类型进行比较。这时推导失败了 (不能推导 `Ts`), 只剩下第二个函数模板。所以, 这里不需要对函数模板排序。

第二个调用 `f((int*)nullptr, (double*)nullptr)`: 对于第一个函数模板, 推导失败, 因为函数实参比形参多, 但是对于第二个和第三个模板, 推导成功。

```
1 f<int*, double*>((int*)nullptr, (double*)nullptr) // for second template
```

和

```
1 f<int, double>((int*)nullptr, (double*)nullptr) // for third template
```

部分排序考虑第二个和第三个模板, 这两个都是可变参数模板: 当对可变参数模板应用第 16.2.3 节描述的正式排序规则时, 每个模板参数包将替换为单个组成的类型、类模板或值。第二个和第三个函数模板的合成参数类型分别是 `A1` 和 `A2*`, 其中 `A1` 和 `A2` 是合成类型。通过将参数包 `Ts` 替换为单元素序列 (`A2*`), 第二个模板针对第三个参数类型列表的推导成功。然而, 没有办法使第三个模板的参数包的模式 `Ts*` 与非指针类型 `A1` 匹配, 因此第三个函数模板 (接受指针参数) 比第二个函数模板 (接受任何参数) 更特化。

第三个调用, `f((int*)nullptr)`, 引入了一个新的问题: 三个函数模板都能成功推导, 需要对非变参模板和变参模板进行部分排序。这里, 我们比较了第一个和第三个函数模板。合成参数类型是 `A1*` 和 `A2*`, 其中 `A1` 和 `A2` 是合成类型。通过将 `A2` 替换为 `T`, 第一个模板对第三个合成参数列表的推导会成功。另外, 通过将单元素序列 (`A1`) 替换为参数包 `Ts`, 第三个模板对第一个合成参数列表的推导会成功。第一个模板和第三个模板之间的部分排序会产生歧义。然而, 有一个特殊规则禁止来自函数参数包的实参 (例如, 第三个模板的参数包 `Ts*…`) 匹配非参数包的形参 (第一个模板的形参

T^*)。因此，第一个模板对第三个合成参数列表的模板推导失败，并且第一个模板比第三个模板更特化。这个特殊规则认为非变参模板(具有固定数量的参数)比变参模板(具有可变数量的参数)更特化。

上面描述的规则同样适用于函数签名中的类型中的包扩展。可以将前面示例中每个函数模板的形参和实参包装到可变参数类模板 Tuple 中，从而得到一个不涉及函数参数包的示例：

details/tupleoverload.cpp

```
1 #include <iostream>
2
3 template<typename... Ts> class Tuple
4 {
5 };
6
7 template<typename T>
8 int f(Tuple<T*>)
9 {
10     return 1;
11 }
12
13 template<typename... Ts>
14 int f(Tuple<Ts...>)
15 {
16     return 2;
17 }
18
19 template<typename... Ts>
20 int f(Tuple<Ts*...>)
21 {
22     return 3;
23 }
24
25 int main()
26 {
27     std::cout << f(Tuple<int, double>()); // calls f<>(Tuple<Ts...>)
28     std::cout << f(Tuple<int*, double*>()); // calls f<>(Tuple<Ts*...>)
29     std::cout << f(Tuple<int*>()); // calls f<>(Tuple<T*>)
30 }
```

函数模板排序将 Tuple 的模板参数中的包扩展为函数参数包，会输出相同的结果：

16.3. 显式特化

重载函数模板的能力，结合部分排序规则来选择“最佳”匹配的函数模板，可以添加更多特化的模板，调优代码以提高效率，但类模板和变量模板不能重载。不过，可以选择另一种机制来支持类模板的定制：显式特化，一种全特化的语言特性。提供了一个模板的实现，可以完全替换模板参数：没有保留模板参数。类模板、函数模板和变量模板可以全特化。

别名模板是唯一不能通过全特化或偏特化进行特化的模板。为了使模板别名的模板参数推导透明化，这个限制是必要的。参见第 15.11 节。

在类定义体之外定义的类模板成员（即成员函数、嵌套类、静态数据成员和成员枚举类型）也可以这样做。

后面的一节中，将描述偏特化。这类似于全特化，不完全替换模板参数，而是在模板的替代实现中保留了一些参数。全特化和偏特化在源码中都是显式的，这就是为什么在讨论中避免使用显式特化这个术语。全特化和偏特化都不会引入全新的模板或模板实例，这些构造为已经在泛型（或非特化）模板中隐式声明实例提供了显式定义。这是一个相对重要的概念，也是与重载模板的关键性区别。

16.3.1 类模板全特化

通过三个标记的序列引入了全特化：`template`、`<` 和 `>`。

声明完整的函数模板特化也需要相同的前缀。C++ 早期设计不包括这个前缀，但是成员模板的添加需要额外的语法来消除复杂的特化歧义。

此外，类名后面跟着声明特化的模板参数：

```
1 template<typename T>
2 class S {
3     public:
4         void info() {
5             std::cout << "generic (S<T>::info())\n";
6         }
7     };
8
9 template<>
10 class S<void> {
11     public:
12         void msg() {
13             std::cout << "fully specialized (S<void>::msg())\n";
14         }
15 }
```

全特化的实现不需要与泛型定义相关：这允许拥有不同名称的成员函数（`info` 和 `msg`）。其与泛型定义的关联，仅决定类模板的名称。

指定的模板实参列表必须与模板形参列表对应，为模板类型参数指定非类型值无效。但对于带有默认模板实参的形参，模板实参可选：

```
1 template<typename T>
2 class Types {
3     public:
4         using I = int;
5     };
6
7 template<typename T, typename U = typename Types<T>::I>
8 class S; // #1
9 template<>
10 class S<void> { // #2
11     public:
12         void f();
13     };
14
15 template<> class S<char, char>; // #3
16
17 template<> class S<char, 0>; // ERROR: 0 cannot substitute U
18
19 int main()
20 {
21     S<int>* pi; // OK: uses #1, no definition needed
22     S<int> e1; // ERROR: uses #1, but no definition available
23     S<void>* pv; // OK: uses #2
24     S<void,int> sv; // OK: uses #2, definition available
25     S<void,char> e2; // ERROR: uses #1, but no definition available
26     S<char,char> e3; // ERROR: uses #3, but no definition available
27 }
28
29 template<>
30 class S<char, char> { // definition for #3
31 }
```

正如这个示例，全特化(和模板)的声明不一定是定义。当声明全特化时，对给定的模板参数集从不使用泛型定义。因此，若需要定义但没有提供，程序就会出错。对于类模板特化，有时“前置声明”类型很有用，可以构造相互依赖的类型。全特化声明与普通类声明相同(不是模板声明)，区别是语法和声明必须与模板声明相匹配。因为不是模板声明，所以完整类模板特化的成员可以使用普通的类外成员定义语法来定义(换句话说，不能指定 template<> 的前缀)：

```
1 template<typename T>
2 class S;
3 template<> class S<char*> {
4     public:
5         void print() const;
6     };
7
8 // the following definition cannot be preceded by template<>
9 void S<char*>::print() const
```

```
10 {
11     std::cout << "pointer to pointer to char\n";
12 }
```

更复杂的例子可能会强化这一概念:

```
1 template<typename T>
2 class Outside {
3     public:
4         template<typename U>
5             class Inside {
6         };
7     };
8
9 template<>
10 class Outside<void> {
11     // there is no special connection between the following nested class
12     // and the one defined in the generic template
13     template<typename U>
14         class Inside {
15             private:
16                 static int count;
17         };
18     };
19
20 // the following definition cannot be preceded by template<>
21 template<typename U>
22 int Outside<void>::Inside<U>::count = 1;
```

全特化是对某个泛型模板实例化的替代，在同一程序中不能同时存在一个模板的显式版本和生成版本。若在同一个文件中使用，编译器会报错:

```
1 template<typename T>
2 class Invalid {
3 };
4
5 Invalid<double> x1; // causes the instantiation of Invalid<double>
6
7 template<>
8 class Invalid<double>; // ERROR: Invalid<double> already instantiated
```

若使用发生在不同的翻译单元中，问题可能就不那么容易发现了。下面是一个错误的 C++ 举例，其由两个文件组成，并在许多实现上进行编译和链接:

```
1 // Translation unit 1:
2 template<typename T>
3 class Danger {
4     public:
5         enum { max = 10 };
6     };
7
```

```

8 char buffer[Danger<void>::max]; // uses generic value
9
10 extern void clear(char*);
11
12 int main()
13 {
14     clear(buffer);
15 }
16
17 // Translation unit 2:
18 template<typename T>
19 class Danger;
20
21 template<>
22 class Danger<void> {
23     public:
24     enum { max = 100 };
25 }
26
27 void clear(char* buf)
28 {
29     // mismatch in array bound:
30     for (int k = 0; k<Danger<void>::max; ++k) {
31         buf[k] = '\0';
32     }
33 }

```

这个示例只是为了说明问题，必须注意确保特化的声明对泛型模板的所有使用者可见，特化的声明通常应该跟在模板头文件的声明之后。当通用实现来自外部源时(这样对应的头文件就不会修改)，这并不一定实用，但创建包含通用模板和随后的特化声明的头文件可以这样做，以避免一些难以找到的错误。通常，最好避免使用来自外部源的模板特化，除非明确标记是为此目的而设计的。

16.3.2 函数模板全特化

(显式的)全函数模板特化背后的语法和原则与全类模板特化的语法和原则相似，但是重载和参数推导会发挥作用。

当特化模板可以通过参数推导(使用声明中提供的参数类型)和部分排序确定时，全特化声明可以省略显式的模板参数。例如：

```

1 template<typename T>
2 int f(T) // #1
3 {
4     return 1;
5 }
6
7 template<typename T>
8 int f(T*) // #2
9 {
10    return 2;

```

```

11 }
12
13 template<> int f(int) // OK: specialization of #1
14 {
15     return 3;
16 }
17
18 template<> int f(int* ) // OK: specialization of #2
19 {
20     return 4;
21 }

```

一个完整的函数模板特化只提供了一个替代定义，而没有提供替代声明。因此，签名（包括返回类型）必须完全匹配：

```

1 template<typename T> auto foo();
2
3 template<> int foo<int>() { return 42; } // ERROR
4
5 template<> auto foo<int>() { return 42; } // OK

```

完整函数模板特化不能包含默认参数值。然而，为特化模板指定默认参数仍然适用于显式特化：

```

1 template<typename T>
2 int f(T, T x = 42)
3 {
4     return x;
5 }
6
7 template<> int f(int, int = 35) // ERROR
8 {
9     return 0;
10 }

```

（因为全特化提供了替代定义，而不是替代声明。调用函数模板时，调用完全基于函数模板进行解析。）

全特化在许多方面与普通声明（更确切地说，是普通的声明）相似。特别是，这里没有声明模板，在程序中只出现了一个非内联全函数模板特化的定义。但要确保全特化的声明遵循模板规则，以避免使用从通用模板生成的函数。因此，模板 g() 和一个全特化的声明通常在两个文件中定义，如下所示：

- 接口文件包含主要模板和偏特化的定义，但只声明全特化版本：

```

1 #ifndef TEMPLATE_G_HPP
2 #define TEMPLATE_G_HPP
3
4 // template definition should appear in header file:
5 template<typename T>
6 int g(T, T x = 42)
7 {

```

```

8   return x;
9 }
10
11 // specialization declaration inhibits instantiations of the template;
12 // definition should not appear here to avoid multiple definition errors
13 template<> int g(int, int y);
14
15 #endif // TEMPLATE_G_HPP

```

- 对应的实现文件对全特化函数进行了定义:

```

1 #include "template_g.hpp"
2 template<> int g(int, int y)
3 {
4     return y/2;
5 }

```

特化也可以内联，其定义可以(应该)放在头文件中。

16.3.3 变量模板全特化

变量模板也可以全特化，语法很直观:

```

1 template<typename T> constexpr std::size_t SZ = sizeof(T);
2
3 template<> constexpr std::size_t SZ<void> = 0;

```

显然，特化可以提供与模板产生的初始化式不同的初始化式。有趣的是，变量模板特化并不要求其类型与特化模板匹配:

```

1 template<typename T> typename T::iterator null_iterator;
2
3 template<> BitIterator null_iterator<std::bitset<100>>;
4 // BitIterator doesn't match T::iterator, and that is fine

```

16.3.4 成员模板全特化

不仅是成员模板，普通的静态数据成员和类模板的成员函数也可以全特化。该语法要求每个外围类模板都使用 `template<>`。若要特化成员模板，必须添加 `template<>` 来表示特化。假设有以下声明:

```

1 template<typename T>
2 class Outer { // #1
3 public:
4     template<typename U>
5     class Inner { // #2
6         private:
7             static int count; // #3
8     };
9     static int code; // #4

```

```

10 void print() const { // #5
11     std::cout << "generic";
12 }
13 };
14
15 template<typename T>
16 int Outer<T>::code = 6; // #6
17
18 template<typename T> template<typename U>
19 int Outer<T>::Inner<U>::count = 7; // #7
20
21 template<>
22 class Outer<bool> { // #8
23 public:
24     template<typename U>
25     class Inner { // #9
26         private:
27             static int count; // #10
28     };
29     void print() const { // #11
30     }
31 };

```

普通成员代码在泛型外部模板 #1 的点 #4 和点 #5 的 print() 有单独的外围类模板，因此需要 template<> 来为特定的模板参数进行全特化：

```

1 template<>
2 int Outer<void>::code = 12;
3
4 template<>
5 void Outer<void>::print() const
6 {
7     std::cout << "Outer<void>";
8 }

```

对于类 Outer<void>，这些定义在点 #4 和点 #5 的泛型上使用，但是类 Outer<void> 的其他成员在点 #1 的模板中生成。声明之后，为 Outer<void> 提供的显式特化将不再有效。

与函数模板全特化一样，需要一种方法来声明类模板普通成员的特化，而不需要指定定义（避免多个定义）。虽然 C++ 中不允许对成员函数和普通类的静态数据成员进行非定义的类外声明，但在特化类模板成员时可以使用。前面的定义可以在这里声明

```

1 template<>
2 int Outer<void>::code;
3
4 template<>
5 void Outer<void>::print() const;

```

细心的读者可能会指出，Outer<void>::code 的全特化的非定义声明看起来与用默认构造函数初始化的定义完全相同，但这样的声明总是解释为非定义声明。对于只能使用默认构造函数初始化的类型中，静态数据成员的全特化必须求助于初始化列表语法：

```

1 class DefaultInitOnly {
2     public:
3     DefaultInitOnly() = default;
4     DefaultInitOnly(DefaultInitOnly const&) = delete;
5 };
6
7 template<typename T>
8 class Statics {
9     private:
10    static T sm;
11};

```

以下是声明:

```

1 template<>
2 DefaultInitOnly Statics<DefaultInitOnly>::sm;

```

下面是调用默认构造函数的定义:

```

1 template<>
2 DefaultInitOnly Statics<DefaultInitOnly>::sm{};

```

C++11 前, 不能这样。因此, 这种特化不能使用默认初始化式。通常, 会复制默认值的初始化式:

```

1 template<>
2 DefaultInitOnly Statics<DefaultInitOnly>::sm = DefaultInitOnly();

```

但对于我们的示例来说, 这是不行的, 因为复制构造函数删除了。但 C++17 引入了强制复制省略规则, 这使得这种替代方法有效, 因为不再涉及复制构造函数调用。

成员模板 Outer<T>::Inner 也可以为给定的模板参数特化, 而不影响 Outer<T> 特定实例化其他成员, 可以为其特化成员模板。同样, 因为是一个封闭的模板, 所以需要一个 template<>。这将产生如下代码

```

1 template<>
2 template<typename X>
3 class Outer<wchar_t>::Inner {
4     public:
5     static long count; // member type changed
6 };
7
8 template<>
9     template<typename X>
10    long Outer<wchar_t>::Inner<X>::count;

```

模板 Outer<T>::Inner 也可以全特化, 但只适用于 Outer<T> 的特定实例。现在需要两个 template<>: 一个为外围类, 另一个为全特化(内部)模板:

```

1 template<>
2     template<>
3         class Outer<char>::Inner<wchar_t> {

```

```

4   public:
5     enum { count = 1 };
6   };
7
8 // the following is not valid C++:
9 // template<> cannot follow a template parameter list
10 template<typename X>
11 template<> class Outer<X>::Inner<void>; // ERROR

```

将其与 Outer<bool> 的成员模板的特化进行对比。因为后者已经全特化了，不是封闭的模板，所以只需要一个 template<>:

```

1 template<>
2 class Outer<bool>::Inner<wchar_t> {
3   public:
4     enum { count = 2 };
5 }

```

16.4. 类模板偏特化

模板全特化很有用，但有时会希望为一组模板参数特化类模板或变量模板，而不是只特化一组特定的模板参数。例如，有一个链表的类模板:

```

1 template<typename T>
2 class List { // #1
3   public:
4   ...
5   void append(T const&);
6   inline std::size_t length() const;
7   ...
8 }

```

使用此模板的大型项目可以实例化多种类型成员。对于未内联展开的成员函数（例如：List<T>::append()），可能会导致代码的大幅度增长。从底层来看，List<int*>::append() 和 List<void*>::append() 的代码相同。我们希望指定所有指针 List 共享一个实现。虽然不能用 C++ 来表达，可以指定不同的模板定义实例化来实现：

```

1 template<typename T>
2 class List<T*> { // #2
3   private:
4     List<void*> impl;
5   ...
6   public:
7   ...
8   inline void append(T* p) {
9     impl.append(p);
10   }
11   inline std::size_t length() const {
12     return impl.length();
13 }

```

```
13 }
14 ...
15 };
```

此上下文中，位于 #1 的原始模板称为主模板，而后的定义称为偏特化 (因为使用此模板定义的模板参数仅部分指定)。描述偏特化的语法是模板参数列表声明 (`template<…>`) 和一组在类模板名称上显式指定的模板参数 (示例中是 `<T*>`) 的组合。

我们的代码有一个问题，因为 `List<void*>` 递归地包含了同一个 `List<void*>` 类型的成员。为了打破这个循环，可以在之前的偏特化前使用全特化：

```
1 template<>
2 class List<void*> { // #3
3 ...
4     void append (void* p);
5     inline std::size_t length() const;
6 ...
7 };
```

因为全特化比偏特化更匹配，所以可行。因此，指针 `List` 的所有成员函数都会转发 (通过容易内联的函数) 到 `List<void*>`。这是对抗代码膨胀 (C++ 模板经常被诟病) 的有效方法。

偏特化声明的形参和实参列表存在一些限制。其中一些如下：

1. 偏特化的实参必须在类型 (类型、非类型或模板) 上与主模板的相应形参匹配。
2. 偏特化的形参列表不能有默认实参，取而代之的是主类模板的默认实参。
3. 偏特化的非类型参数应该是非依赖值或普通非类型模板参数。不能是更复杂的依赖表达式，如 `2*N`(其中 `N` 是模板参数)。
4. 偏特化的模板实参列表不应与主模板的形参列表相同 (忽略重命名)
5. 若其中一个模板参数是包扩展，这个包必须放在模板参数列表的最后。

下面的例子说明了这些限制：

```
1 template<typename T, int I = 3>
2 class S; // primary template
3
4 template<typename T>
5 class S<int, T>; // ERROR: parameter kind mismatch
6
7 template<typename T = int>
8 class S<T, 10>; // ERROR: no default arguments
9
10 template<int I>
11 class S<int, I*2>; // ERROR: no nontype expressions
12
13 template<typename U, int K>
14 class S<U, K>; // ERROR: no significant difference from primary template
15
16 template<typename... Ts>
17 class Tuple;
```

```

19 template<typename Tail, typename... Ts>
20 class Tuple<Ts..., Tail>; // ERROR: pack expansion not at the end
21
22 template<typename Tail, typename... Ts>
23 class Tuple<Tuple<Ts...>, Tail>; // OK: pack expansion is at the end of a
24 // nested template argument list

```

每个偏特化都与主模板相关联。使用模板时，总是要查找主模板，但随后参数也要与相关特化的参数进行匹配（使用模板推导指引，如第 15 章所述），以确定选择哪个模板实现。就像函数模板参数推导一样，SFINAE 原则在这里也适用：当试图匹配偏特化时，若形成了无效构造，则放弃该特化，并检查另一个候选是否可用。若没有找到匹配的特化，则选择主模板。若找到多个匹配的特化，则选择最特化的那个；若没有最特化的，则会出现歧义性错误。

最后，类模板偏特化有可能比主模板拥有更多或更少的参数。考虑泛型模板 List，再次在点 #1 声明。已经讨论了如何优化指针列表的情况，但这里可能希望对某些指向成员的指针类型进行同样的优化。下面的代码实现了指向成员指针的指针：

```

1 // partial specialization for any pointer-to-void* member
2 template<typename C>
3 class List<void* C::*> { // #4
4     public:
5         using ElementType = void* C::*;
6         ...
7         void append(ElementType pm);
8         inline std::size_t length() const;
9         ...
10    };
11
12 // partial specialization for any pointer-to-member-pointer type except
13 // pointer-to-void* member, which is handled earlier
14 // (note that this partial specialization has two template parameters,
15 // whereas the primary template only has one parameter)
16 // this specialization makes use of the prior one to achieve the
17 // desired optimization
18 template<typename T, typename C>
19 class List<T* C::*> { // #5
20     private:
21         List<void* C::*> impl;
22         ...
23     public:
24         using ElementType = T* C::*;
25         ...
26         inline void append(ElementType pm) {
27             impl.append(static_cast<void* C::*>(pm));
28         }
29         inline std::size_t length() const {
30             return impl.length();
31         }
32         ...
33    };

```

除了对模板参数的数量外，在 #4 定义的实现是一个偏特化(对于简单的指针情况，是一个全特化)，所有其他定义都会使用 #5 点的声明转发。然而，#4 的特化比 #5 更特化；因此，不应出现歧义。

而且，显式编写的模板参数数量，甚至可能与主模板中的模板参数数量不同。这既可以在有默认模板参数的情况下发生，也可以使用可变参数模板：

```
1 template<typename... Elements>
2 class Tuple; // primary template
3
4 template<typename T1>
5 class Tuple<T>; // one-element tuple
6
7 template<typename T1, typename T2, typename... Rest>
8 class Tuple<T1, T2, Rest...>; // tuple with two or more elements
```

16.5. 变量模板偏特化

当变量模板添加到 C++11 标准草案中时，忽略了规范的几个方面，其中一些问题仍然没有解决。然而，实际的实现通常会对这些问题进行处理。

也许这些问题中最令人惊讶的是，标准提到了偏特化变量模板的能力，但它没有描述如何声明它们或它们的含义。因此，下面的内容是基于实践中的 C++ 实现(确实允许这样的偏特化)，而不是基于 C++ 标准。

其语法类似于全变量模板特化，不同的是使用实际的模板声明替换 `template<>`，并且变量模板名称后面的模板参数列表必须依赖于模板参数。例如：

```
1 template<typename T> constexpr std::size_t SZ = sizeof(T);
2
3 template<typename T> constexpr std::size_t SZ<T*> = sizeof(void*);
```

与变量模板的全特化一样，偏特化的类型不需要匹配主模板类型：

```
1 template<typename T> typename T::iterator null_iterator;
2
3 template<typename T, std::size_t N> T* null_iterator<T[N]> = null_ptr;
4 // T* doesn't match T::iterator, and that is fine
```

关于为变量模板偏特化指定的模板参数类型的规则与为类模板特化指定的规则相同。类似地，为给定的具体模板参数列表选择特化的规则也相同。

16.6. 后记

全特模板从一开始就是 C++ 模板机制的一部分。另一方面，函数模板重载和类模板偏特化出现要晚得多。HP aC++ 编译器是第一个实现函数模板重载的，EDG 的 C++ 前端是第一个实现类模板偏特化的。本章描述的部分排序原则最初是由 EDG 的 Steve Adamczyk 和 John Spicer 发明。

模板特化终止无限递归模板定义的能力(如第 16.4 节中给出的 `List<T*>` 示例)由来已久。然而, Erwin Unruh 可能是第一个注意到这可以进行模板元编程的人: 使用模板实例化机制在编译时执行计算。我们会在第 23 章专门讨论这个话题。

可能想知道为什么只有类模板和变量模板可以偏特化,主要是历史原因。其实,也可以为函数模板定义相同的机制(参见第 17 章)。重载函数模板的效果类似,但有细微的差别。使用时,只需要查找主模板。特化只在使用后考虑,以确定应该使用哪种实现。相反,必须通过查找将所有重载函数模板引入重载集,而且它们可能来自不同的命名空间或类。这无意中增加了重载模板名称的可能性。

相反,可以想象允许重载类模板和变量模板的形式:

```
1 // invalid overloading of class templates
2 template<typename T1, typename T2> class Pair;
3 template<int N1, int N2> class Pair;
```

然而,对这样的机制,似乎并没有迫切的需求。

第 17 章 未来的方向

从 1988 年的最初设计到 2003 年、2011 年、2014 年和 2017 年的各种标准化里程碑，C++ 模板在不断地发展。可以说，模板在某种程度上与 1998 年最初的标准之后增加的主要语言有关。

第一版列出了在第一个标准之后会看到的一些扩展，其中一些已经成为现实：

- 尖括号的修改:C++11 去掉了在两个尖括号之间插入空格的需要。
- 默认函数模板参数:C++11 允许函数模板有默认模板参数。
- `typedef` 模板:C++11 引入了类似的别名模板。
- 偏特化的模板参数列表不应与主模板的参数列表相同 (忽略重命名)
- `typeof` 操作符:C++11 引入了 `decltype` 操作符，其功能相同 (因为不能完全满足 C++ 开发者社区的需求，所以使用了不同的标记，避免与现有的扩展冲突)。
- 静态属性: 第一版预期编译器将直接支持类型特征的选择。这已经实现了，尽管接口是使用标准库来表示 (使用编译器扩展来实现多个特性)。
- 自定义实例化诊断: 新的关键字 `static_assert` 实现了第一版中的 `std::instantiation_error`。
- 参数列表:C++11 中的参数包。
- 布局控制:C++11 的 `alignof` 和 `alignas` 满足了第一版中描述的需求。此外，C++17 添加了 `std::variant` 模板支持联合结构。
- 初始化式推导:C++17 增加了类模板参数推导，解决了同样的问题。
- 函数表达式:C++11 的 Lambda 表达式提供了这种功能 (其语法与第一版中讨论的有所不同)。

第一版中提出的其他方向还没有进入现代 C++，但大多数仍在进行讨论，我们将它们的呈现在本书中。与此同时，也有其他想法的出现，这里会展示其中的一些。

17.1. 放宽的 `typename` 规则

第一版中，本节建议将来可能会对 `typename` 的使用规则进行两种放宽 (请参阅第 13.3.2 节)：以前不允许使用 `typename` 的地方，现在可以用 `typename` 了，编译器可以相对容易地推导出带有依赖限定符的限定名，所以不必为命名类型使用 `typename`。前者实现了 (C++11 中的 `typename` 冗余使用)，但后者没有。

最近，在对类型说明符的期望明确的各种类型，可以选用 `typename`：

- 命名空间和类作用域中的函数，以及成员函数声明的返回类型和参数类型。类似地，函数和成员函数模板，以及出现在作用域中的 Lambda 表达式。
- 变量、变量模板和静态数据成员声明的类型。同样，变量模板也类似。
- 别名类型或别名模板声明中等号表示后的类型。
- 模板的类型参数的默认参数。
- 出现在尖括号中的类型，紧跟在 `static_cast`、`const_cast`、`dynamic_cast` 或 `reinterpret_cast` 之后。
- `new` 表达式中命名的类型。

尽管这是一个相对特别的列表，但语言中的这种更改允许删除 `typename`，会使代码更紧凑和可读更强。

17.2. 广义非类型模板参数

对非类型模板参数的限制中，可能最让初学者和高级模板编写者感到惊讶的是不能提供字符串字面值作为模板参数。

看看下面的例子：

```
1 template<char const* msg>
2 class Diagnoser {
3     public:
4     void print();
5 };
6
7 int main()
8 {
9     Diagnoser<"Surprise!">().print();
10 }
```

然而，其中有一些问题。标准 C++ 中，当两个 Diagnostic 实例具有相同的参数时，其类型相同。参数是一个指针值，是一个地址。但出现在不同源位置的两个相同字符串，并不要求具有相同的地址。可能会尴尬的发现，`Diagnoser<"X">` 和 `Diagnoser<"X">` 实际上是两种不同且不兼容的类型！（请注意，“X”的类型是 `char const[2]`，但当作为模板参数传递时，会衰变为 `char const*`）

由于这些（以及相关的）情况，C++ 标准禁止将字符串字面值作为模板的参数。然而，一些实现确实将该工具作为扩展提供。通过在模板实例的内部表示中使用实际的字符串文字内容来实现这一点。尽管是可行的，但一些 C++ 语言评论者认为，可以用字符串字面值替换的非类型模板参数，声明方式应该与可以用地址替换的非类型模板参数不同。一种可能是在字符参数包中捕获字符串字面值，例如：

```
1 template<char... msg>
2 class Diagnoser {
3     public:
4     void print();
5 };
6
7 int main()
8 {
9     // instantiates Diagnoser<'S', 'u', 'r', 'p', 'r', 'i', 's', 'e', '!'>
10    Diagnoser<"Surprise!">().print();
11 }
```

还应该注意到另一个技术问题。考虑下面的模板声明，假设语言已经扩展为接受字符串字面值作为模板参数：

```
1 template<char const* str>
2 class Bracket {
3     public:
4     static char const* address();
5     static char const* bytes();
6 };
7
```

```

8 template<char const* str>
9 char const* Bracket<str>::address()
10 {
11     return str;
12 }
13
14 template<char const* str>
15 char const* Bracket<str>::bytes()
16 {
17     return str;
18 }

```

前面的代码中，除了名称之外，两个成员函数是相同的——这种情况很常见。想象一下，实现使用类似于宏展开的方式实例化 `Bracket<"X">`: 若两个成员函数在不同的翻译单元中实例化，可能返回不同的值。有趣的是，对目前提供此扩展的一些 C++ 编译器的测试表明，确实受到了这种行为的影响。

相关的问题是提供浮点字面量 (和简单的浮点常量表达式) 作为模板参数的能力。例如：

```

1 template<double Ratio>
2 class Converter {
3     public:
4         static double convert (double val) {
5             return val*Ratio;
6         }
7     };
8
9 using InchToMeter = Converter<0.0254>;

```

这也由一些 C++ 实现提供，并且不存在严重的技术挑战 (不像字符串字面量参数)。

C++11 引入了文字类类型的概念：这种类类型可以在编译时计算的常量值 (包括通过 `constexpr` 函数进行的复杂计算)。当这种类类型可用，就需要将其用于非类型模板参数。但出现了与上面描述的字符串字面量参数类似的问题。两个类类型值的“相等”不简单，因为通常是由 `operator==` 确定。这种相等性决定两个实例化是否等价，但在实践中，链接器必须通过比较损坏的名称来检查这种等价性。解决方法可能是选择将某些文字类标记为具有简单的相等标准，相当于对类的标量成员进行比较。只有具有这种简单的相等条件的类类型，才允许作为非类型模板参数类型。

17.3. 函数模板的偏特化

第 16 章中，讨论了类模板如何偏特化，函数模板则可以重载。这两种机制有些不同。

偏特化不会引入新模板，其为现有模板 (主模板) 的扩展。当查找类模板时，首先考虑主模板。若选择了主模板后，发现该模板的偏特化具有与实例化匹配的模板参数模式时，则实例化其定义，而非主模板的定义。(与全特化模板的工作方式完全相同)

相反，重载函数模板是完全独立的模板。在实例化模板时，将所有重载模板放在一起，重载解析尝试选择最适合的模板。乍一看，这似乎有充足的选择，但在实践中会有一些限制：

- 可以特化类的成员模板，而不需要更改该类的定义，但添加重载成员确实需要更改类的定义。

这不是一个好选择，因为可能不能这样做。此外，C++ 标准目前不允许向 std 命名空间添加新模板，但它允许对该命名空间模板进行特化。

- 要重载函数模板，其函数参数必须在某些重要方面有所不同。考虑一个函数模板 R convert(T const&)，其中 R 和 T 是模板参数。我们可能很想特化 R = void 的模板，但这不能使用重载来完成。
- 对于非重载函数的代码，在函数重载时不再有效。具体来说，给定两个函数模板 f(T) 和 g(T)(其中 T 是一个模板参数)，表达式 g(&f<int>) 只有在 f 没有重载时才有效(否则，无法确定 f 是哪个)。
- 友元声明指的是特定函数模板或特定函数模板的实例化。函数模板的重载版本不会自动拥有原始模板的权限。

列表共同构成了一个强有力的论据，支持函数模板的偏特化构造。

偏特化函数模板的一种自然语法是泛化类模板表示法：

```
1 template<typename T>
2 T const& max (T const&, T const&); // primary template
3
4 template<typename T>
5 T* const& max <T*>(T* const&, T* const&); // partial specialization
```

一些语言设计人员担心这种偏特化方法与函数模板重载的交互。例如：

```
1 template<typename T>
2 void add (T& x, int i); // a primary template
3
4 template<typename T1, typename T2>
5 void add (T1 a, T2 b); // another (overloaded) primary template
6
7 template<typename T>
8 void add<T*> (T* &, int); // Which primary template does this specialize?
```

然而，我们希望将这样的情况视为错误，不会对功能的使用产生重大影响。

这个扩展在 C++11 的标准化过程中简要地讨论过，但是最后却没有引起多少人的兴趣。不过，因为其巧妙地解决了一些常见的编程问题，这个话题偶尔还会出现。这个特性也许会在未来的 C++ 标准中采纳。

17.4. 命名模板参数

第 21.4 节描述了一种技术，其允许为特定参数提供非默认模板参数，而不必指定可以使用默认值的其他模板参数，但需要大量的工作才能得到相对简单的效果。因此，很自然的会想到使用一种语言机制来命名模板参数。

类似的扩展(有时称为关键字参数)是由 Roland Hartinger 在 C++ 标准化过程中提出的(参见 [StroustrupDnE] 第 6.5.1 节)。虽然在技术上合理，但该提议最终没有接受。我们不知道命名模板参数是否会成为 C++ 的一部分，但是这个话题会在委员会的讨论中经常出现。

为了完整起见，这里说一个已经讨论过的语法概念：

```

1 template<typename T,
2   typename Move = defaultMove<T>,
3   typename Copy = defaultCopy<T>,
4   typename Swap = defaultSwap<T>,
5   typename Init = defaultInit<T>,
6   typename Kill = defaultKill<T>>
7 class Mutator {
8   ...
9 };
10
11 void test(MatrixList ml)
12 {
13   mySort (ml, Mutator <Matrix, .Swap = matrixSwap>);
14 }

```

参数名称前的句点用来表示我们通过名称引用模板参数，这种语法类似于 1999 年 C 标准中引入的“指定初始化式”语法：

```

1 struct Rectangle { int top, left, width, height; };
2 struct Rectangle r = { .width = 10, .height = 10, .top = 0, .left = 0 };

```

当然，引入命名模板参数意味着模板的模板参数名称现在是该模板的公共接口的一部分，不能随意更改。这可以通过更明确的、可选择的语法来解决，例如：

```

1 template<typename T,
2   Move: typename M = defaultMove<T>,
3   Copy: typename C = defaultCopy<T>,
4   Swap: typename S = defaultSwap<T>,
5   Init: typename I = defaultInit<T>,
6   Kill: typename K = defaultKill<T>>
7 class Mutator {
8   ...
9 };
10
11 void test(MatrixList ml)
12 {
13   mySort (ml, Mutator <Matrix, .Swap = matrixSwap>);
14 }

```

17.5. 重载类模板

完全可以想象类模板可以在其模板参数上重载。例如，创建一系列数组模板，其中包含动态和静态的数组：

```

1 template<typename T>
2 class Array {
3   // dynamically sized array
4   ...
5 };

```

```

6
7 template<typename T, unsigned Size>
8 class Array {
9   // fixed size array
10  ...
11 };
```

重载并不一定局限于模板参数的数量，参数类型也可以改变：

```

1 template<typename T1, typename T2>
2 class Pair {
3   // pair of fields
4   ...
5 };
```



```

7 template<int I1, int I2>
8 class Pair {
9   // pair of constant integer values
10  ...
11 };
```

虽然语言设计者已经非正式地讨论过了这个想法，但是还没有正式地提交给 C++ 标准委员会。

17.6. 中间包扩展的推导

只有当包展开发生或参数列表的末尾时，包展开的模板参数推导才有效。从列表中提取第一个元素相当简单：

```

1 template<typename... Types>
2 struct Front;
3
4 template<typename FrontT, typename... Types>
5 struct Front<FrontT, Types...> {
6   using Type = FrontT;
7 };
```

在第 16.4 节中描述的对偏特化的限制，不能简单地提取列表的最后一个元素：

```

1 template<typename... Types>
2 struct Back;
3
4 template<typename BackT, typename... Types>
5 struct Back<Types..., BackT> { // ERROR: pack expansion not at the end of
6   using Type = BackT; // template argument list
7 };
```

可变参数函数模板的模板参数推导也受到类似的限制。关于包展开和偏特化的模板参数推导的规则将放宽，允许包展开发生在模板参数列表的任何地方，这使得操作更加简单。此外，推论允许在同一个参数列表中进行多个包扩展（尽管可能性较小）：

```

1 template<typename... Types> class Tuple {
2 };
3
4 template<typename T, typename... Types>
5 struct Split;
6
7 template<typename T, typename... Before, typename... After>
8 struct Split<T, Before..., T, After...> {
9   using before = Tuple<Before...>;
10  using after = Tuple<After...>;
11 };

```

允许多个包扩展会带来额外的复杂性。例如，Split 是在 T 第一次出现时分开，在 T 最后一次出现时分开，还是在两者之间分开？推导过程的复杂性达到多少时，编译器才会放弃推导？

17.7. void 的规范化

编写模板时，规则性是一种优点：若单一构造可以覆盖所有情况，会使模板更简单。我们的程序有一点不规则，那就是类型。例如，考虑以下情况：

```

1 auto&& r = f(); // error if f() returns void

```

这适用于 f() 返回的所有类型，除了 void。同样的问题发生在使用 decltype(auto)：

```

1 decltype(auto) r = f(); // error if f()

```

void 并不是唯一的非规范类型：函数类型和引用类型在某些方面也经常出现异常行为。然而，void 会使模板变得复杂。例如，第 11.1.3 节如何使完美的 std::invoke() 包装器实现复杂化的示例。

可以将 void 定义为具有唯一值的普通值类型（如 std::nullptr_t 表示 nullptr）。为了向后兼容的目的，仍需要为函数声明保留一个特殊的情况：

```

1 void g(void); // same as void g();

```

在大多数方式中，void 会成为一个完整的值类型。可以声明 void 变量和引用：

```

1 void v = void{};
2 void&& rrsv = f();

```

更重要的是，许多模板不再需要专门处理 void。

17.8. 模板的类型检查

使用模板编程的复杂性，很大程度上来源于编译器无法在本地检查模板定义是否正确。相反，模板的大多数检查发生在模板实例化期间，此时模板定义上下文和模板实例化上下文交织在一起。这种不同上下文的混合使得很难进行分配。这是模板定义的错误，因为不正确地使用了模板参数？还是模板用户的错误，还是提供的模板参数不符合模板的要求？这个问题可以用一个简单的例子来说明，我们用一个常规编译器产生的错误信息进行了注释：

```

1 template<typename T>
2 T max(T a, T b)
3 {
4     return b < a ? a : b; // ERROR: "no match for operator <
5     // (operator types are 'x' and 'x')"
6 }
7
8 struct X {
9 };
10 bool operator> (X, X);
11
12 int main()
13 {
14     X a, b;
15     X m = max(a, b); // NOTE: "in instantiation of function template specialization
16                     // 'max<x>' requested here"
17 }
```

实际的错误(缺少合适的小于操作符)是在函数模板 `max()` 的定义中检测到的。这可能是真正的错误——也许 `max()` 应该使用大于操作符代替? 但编译器还提供了一个信息, 指出了 `max<X>` 实例化的位置, 这可能才是真正的错误——也许 `max()` 在文档中需要小于操作符? 无法回答这个问题的话, 会出现在第 9.4 节中提到的“错误小说”, 其中编译器提供了从初始化到检测到错误模板定义的整个模板实例化历史。然后, 开发者需要确定哪个模板定义(或者模板的原始使用)出现了错误。

模板类型检查背后的思想是, 在模板本身中描述模板的需求。这样, 编译器就可以在编译失败时判断是模板定义问题, 还是模板使用出了问题。解决这个问题的方法是使用概念将模板的需求描述为模板签名的一部分:

```

1 template<typename T> requires LessThanComparable<T>
2 T max(T a, T b)
3 {
4     return b < a ? a : b;
5 }
6
7 struct X { };
8 bool operator> (X, X);
9
10 int main()
11 {
12     X a, b;
13     X m = max(a, b); // ERROR: X does not meet the LessThanComparable requirement
14 }
```

通过描述模板参数 `T` 上的需求, 编译器能够确保函数模板 `max()` 只使用用户期望提供的 `T` 上的操作(`LessThanComparable` 描述了小于操作符的需求)。使用模板时, 编译器可以检查所提供的模板参数是否提供了 `max()` 函数模板正常工作所需的行为。通过分离类型检查问题, 编译器可以更容易地提供准确的诊断信息。

上面的例子中, `LessThanComparable` 称为概念: 表示编译器可以检查的类型上的约束(更一般的

情况下，是一组类型上的约束)。概念系统可以以不同的方式进行设计。

C++11 标准制定过程中，一个精心设计并实现的系统的概念足够强大，可以检查模板的实例化点和模板的定义。上面的例子中，前者意味着 `main()` 中的错误可以更早的发现，诊断出 `X` 不满足 `LessThanComparable` 的约束。后者在处理 `max()` 模板时，编译器检查是否使用了 `LessThanComparable` 概念不允许的操作(若违反了该约束，则会发出诊断信息)。出于各种实际的考虑，C++11 提案最终从语言规范中删除了概念(例如，仍然有许多小的规范问题，这些问题的解决会威胁到已经延迟发布的标准)。

C++11 最终发布后，委员会成员提出了一个新的提案(最初称为精简概念)并进行了开发。这个系统的目的不是基于模板所附带的约束来检查模板的正确性，它只关注实例化点。因此，若在我们的示例中使用大于操作符实现 `max()`，此时不会发出错误。但是，由于 `X` 不满足 `LessThanComparable` 的约束，在 `main()` 中仍然会发出错误。新版概念提案实现，并在所谓的概念 TS(TS 代表技术规范)中指定，称为概念的 C++ 扩展。

例如，2017 年初的概念 TS 版本的 N4641。

目前，该技术规范的基本已经集成到下一个标准的草案中(预计进入 C++20)。附录 E 涵盖了本书付印时草稿中规定的语言特性。

17.9. 反射元编程

编程上下文中，反射指的是以编程方式检查程序特性的能力(例如，回答诸如类型是整数吗？或类类型包含哪些非静态数据成员？)元编程是“对程序进行编程”的技巧，相当于以编程方式生成新代码。因此，反射式元编程是一种自动合成代码，使其适应程序的现有属性(通常是类型)的技术。

本书的第三部分中，我们将探讨模板如何实现一些简单形式的反射和元编程(某种意义上，模板实例化是元编程的一种形式，因为它会导致新代码的合成)。然而，当涉及到反射时，C++17 模板的能力是相当有限的(例如，不可能回答这个问题：一个类类型包含哪些非静态数据成员？)，元编程的选项不是很方便(特别是，语法变得笨拙，性能令人失望)。

认识到这一领域新特性的潜力，C++ 标准化委员会创建了一个研究小组(SG7)，以探索更强大的反射选项。组织的章程后来也扩展到元编程，以下是正在考虑的选项的一个例子：

```
1 template<typename T> void report(T p) {
2     constexpr {
3         std::meta::info infoT = reflexpr(T);
4         for (std::meta::info : std::meta::data_members(infoT)) {
5             -> {
6                 std::cout << (: std::meta::name(info) :)
7                 << ":" << p.(.info.) << '\n';
8             }
9         }
10    }
11    // code will be injected here
12 }
```

这段代码中有很多新内容。首先，`constexpr`…构造强制其中的语句在编译时进行求值，但若出现在模板中，则只在模板实例化时进行求值。其次，`reflexpr()` 操作符会生成一个不透明类型 `std::meta::info` 的表达式，这是一个句柄，用于反映关于其参数的信息（在本例中是 `T` 的类型）。标准元函数库允许查询此元信息。标准元函数之一是 `std::meta::data_members`，会生成一个 `std::meta::info` 序列，描述其操作数的直接非静态数据成员。所以上面的 `for` 循环实际上是对 `p` 的非静态数据成员的循环。

该系统元编程能力的核心是在各种范围内“注入”代码的能力。构造->…在启动 `constexpr` 求值的语句或声明之后，注入语句和/或声明。本例中，在 `constexpr`…构造。注入的代码片段可以包含特定的模式，这些模式可以使用计算值替换。这个例子中，`(:::)` 产生了一个字符串字面值（表达式 `std::meta::name(info)` 产生了一个类似字符串的对象，表示实体数据成员的非限定名，在这个例子中由 `info` 表示）。类似地，表达式 `(.info.)` 产生一个标识符，命名 `info` 所表示的实体。还提出了用于生成类型、模板参数列表等的其他模式。

这些就绪之后，实例化类型的函数模板 `report()`:

```
1 struct X {  
2     int x;  
3     std::string s;  
4 };
```

会产生类似于

```
1 template<> void report(X const& p) {  
2     std::cout << "x" << ":" << "p.x" << '\n';  
3     std::cout << "s" << ":" << "p.s" << '\n';  
4 }
```

该函数自动生成一个函数来输出类类型的非静态数据成员值。

这些类型的功能有许多应用程序。虽然类似的东西很可能最终会采纳到语言中，但还不清楚什么时候采用。在撰写本文时已经演示了一些实验性实现。（出版这本书之前，SG7 同意使用 `constexpr` 求值和类似 `std::meta::info` 的值类型来处理反射式元编程。然而，这里提出的注入机制没有达成一致意见，可能会采取另一种方式）

17.10. 包管理工具

参数包是在 C++11 中引入的，但处理它们通常需要递归模板实例化技术。回想一下 14.6 节中讨论的代码示例：

```
1 template<typename Head, typename... Remainder>  
2 void f(Head&& h, Remainder&&... r) {  
3     doSomething(h);  
4     if constexpr (sizeof...(r) != 0) {  
5         // handle the remainder recursively (perfectly forwarding the arguments):  
6         f(r...);  
7     }  
8 }
```

通过使用 C++17 编译时 if 语句的特性(请参阅第 8.5 节),这个示例变得更简单了,但仍然属于递归实例化技术,编译起来可能会很耗时。

委员会的几项提案试图在某种程度上简化这种状况。一个例子是引入了从包中挑选特定元素的符号。特别地,对于一个组合 P,有人建议用 P[N] 来表示该组合中的元素 N+1。同样,也有人建议将包装表示为“片”(例如,使用 P[b,e] 符号)。

研究这些建议时,可以清楚地看到它们与上面讨论的反射元编程概念有一定的相关性。目前还不清楚是否会将特定的包选择机制添加到该 C++ 中,或者是否会提供满足这一需求的元编程工具。

17.11. 模块

另一个即将到来的扩展——模块,只与模板有外围关系,但它仍然值得一提,因为模板库是最大受益者之一。

目前,库接口在头文件中指定,这些头文件以文本形式包含在翻译单元中。这种方法有几个缺点,令人讨厌的两个是(a)之前包含的代码可能会修改界面文本的含义(例如,通过宏),以及(b)每次重新处理文本迅速占据构建时间。

模块是一种特性,其允许库接口编译成编译器特定的格式,然后这些接口可以“导入”到翻译单元中,而不需要进行宏扩展,也不需要因为偶然出现的添加声明而修改代码的含义。此外,编译器可以安排只读取已编译模块文件中与外部代码相关的部分,从而大大加快编译过程。

模块定义可能是这样:

```
1 module MyLib;
2
3 void helper() {
4     ...
5 }
6
7 export inline void libFunc() {
8     ...
9     helper();
10    ...
11 }
```

这个模块导出了一个函数 libFunc(),可以在外部代码中使用,如下所示:

```
1 import MyLib;
2 int main() {
3     libFunc();
4 }
```

libFunc() 对外部代码可见,但函数 helper() 不是,即使编译的模块文件可能包含 helper() 的信息以启用内联。

向 C++ 添加模块的提议正在进行中,标准化委员会的目标是在 C++17 之后将其集成入语言。在开发这样一个提议时,需要考虑的问题是如何从头文件过渡到模块。已经有一些工具在某种程度上实现了这一点(例如,包含头文件而不使其内容成为模块的一部分的能力),以及其他仍在讨论中的功能(例如,从模块导出宏的能力)。

模块对于模板库特别有用，因为模板几乎总是在头文件中定义。即使包含一个像 `<vector>` 这样的基本标准头文件，也相当于处理数万行 C++ 代码（即使只有该头文件中少量的声明会使用）。其他流行库会将其代码量增加一个数量级。对于处理大型复杂代码库的 C++ 开发者来说，避免所有这些编译成本会让他们很感兴趣。

第三部分：模板和设计

程序通常使用所选编程语言提供的设计模式来构造。因为模板引入了全新的语言机制，所以不难发现其需要新的设计模式。我们将在这一部分探讨这些模式，C++ 标准库包含或使用了其中的几个。

模板与传统的语言不同，其允许我们参数化代码的类型和常量。当与 (1) 偏特化和 (2) 递归实例化相结合时，会带来惊人的效果。

我们不仅旨在列出各种有用的设计，还要传达启发此类设计的原则，以便创造新的技术。因此，下面的章节阐述了大量的设计技巧，包括：

- 多态性
- 具有特征的泛型编程
- 处理重载和继承
- 元编程
- 异构结构和算法
- 表达式模板

还提供了一些注释来帮助调试模板。

第 18 章 模板的多态性

多态性是将不同的行为与单个泛型表示法关联起来的能力。

多态性字面上指的是具有多种形式或形状的情况 (来自希腊语 polymorphos)。

多态性也是面向对象泛型编程的基础，C++ 中通过类继承和虚函数来支持多态。因为这些机制 (至少部分) 在运行时处理，所以这里讨论动态多态性。通常所说的多态性，指的就是动态多态性。然而，模板还允许将不同的特定行为与单个泛型表示关联起来，但这种关联通常在编译时进行处理，称之为静态多态性。本章中，我们回顾了多态的两种形式，并讨论哪种形式适合于哪种情况。

注意，介绍和讨论了一些设计问题之后，第 22 章将讨论一些处理多态性的方法。

18.1. 动态多态性

因为历史原因，C++ 一开始只通过结合使用继承和虚函数来支持多态。

严格地说，宏也可以认为是静态多态性的早期形式。但这里不对宏进行讨论，因为它们大多与其他语言机制不相关。

多态设计的艺术包括在相关对象类型中标识一组公共功能，并将它们声明为公共基类中的虚函数接口。

这种设计方法的代表是管理几何图形，并允许以某种方式 (例如，在屏幕上) 呈现的应用程序。这样的应用程序中，可以识别一个抽象基类 (ABC)GeoObj，声明了适用于几何对象的常见操作和属性。然后，特定几何对象的每个具体类都派生自 GeoObj(见图 18.1):

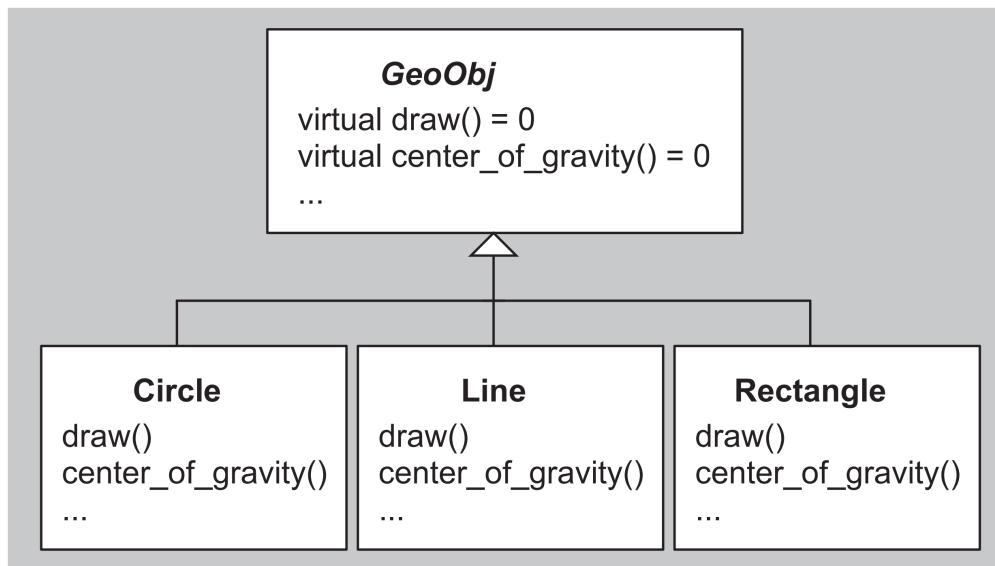


图 18.1. 通过继承实现的多态性

```

1 #include "coord.hpp"
2
3 // common abstract base class GeoObj for geometric objects
4 class GeoObj {
5     public:
6         // draw geometric object:
7         virtual void draw() const = 0;
8         // return center of gravity of geometric object:
9         virtual Coord center_of_gravity() const = 0;
10        ...
11        virtual ~GeoObj() = default;
12    };
13
14 // concrete geometric object class Circle
15 // - derived from GeoObj
16 class Circle : public GeoObj {
17     public:
18         virtual void draw() const override;
19         virtual Coord center_of_gravity() const override;
20         ...
21    };
22
23 // concrete geometric object class Line
24 // - derived from GeoObj
25 class Line : public GeoObj {
26     public:
27         virtual void draw() const override;
28         virtual Coord center_of_gravity() const override;
29         ...
30    };
31 ...

```

创建具体对象后，外部代码可以使用虚拟函数分派机制，通过引用或指向公共基类的指针来操作这些对象。通过指向基类子对象的指针或引用调用虚成员函数，是实际类型中的相应函数。

具体代码如下：

poly/dynopoly.hpp

```

1 #include "dynahier.hpp"
2 #include <vector>
3
4 // draw any GeoObj
5 void myDraw (GeoObj const& obj)
6 {
7     obj.draw(); // call draw() according to type of object
8 }
9
10 // compute distance of center of gravity between two GeoObjs
11 Coord distance (GeoObj const& x1, GeoObj const& x2)

```

```

12 {
13     Coord c = x1.center_of_gravity() - x2.center_of_gravity();
14     return c.abs(); // return coordinates as absolute values
15 }
16
17 // draw heterogeneous collection of GeoObjs
18 void drawElems (for (std::size_type i=0; i<elems.size(); ++i) {
21         elems[i]->draw(); // call draw() according to type of element
22     }
23 }
24
25 int main()
26 {
27     Line l;
28     Circle c, c1, c2;
29
30     myDraw(l); // myDraw(GeoObj&) => Line::draw()
31     myDraw(c); // myDraw(GeoObj&) => Circle::draw()
32
33     distance(c1,c2); // distance(GeoObj&,GeoObj&)
34     distance(l,c); // distance(GeoObj&,GeoObj&)
35
36     std::vector<GeoObj*> coll; // heterogeneous collection
37     coll.push_back(&l); // insert line
38     coll.push_back(&c); // insert circle
39     drawElems(coll); // draw different kinds of GeoObjs
40 }

```

关键的多态接口是函数 `draw()` 和 `center_of_gravity()`, 二者都是虚成员函数。示例演示了在函数 `mydraw()`、`distance()` 和 `drawElems()` 中的用法, 后面的函数使用基类型 `GeoObj` 中的实现。这种方法的结果是, 在编译时通常不知道必须调用哪个版本的 `draw()` 或 `center_of_gravity()`。但在运行时, 将访问虚函数调用的对象的完整动态类型, 以调用相应函数。

也就是说, 多态基类子对象的编码包括一些(大部分是隐藏的)支持运行时分配。

因此, 根据几何对象的实际类型, 会执行相应的操作: 如果对 `Line` 对象调用 `mydraw()`, 则表达式 `obj.draw()` 会调用 `Line::draw()`, 而对 `Circle` 对象调用 `Circle::draw()` 函数。类似地, 使用 `distance()`, 调用适合于参数对象的成员函数 `center_of_gravity()`。

这种动态多态的特点, 可能是处理异构对象集合的能力。`drawElems()` 可以很好的说明:

```

1 elems[i]->draw()

```

上述表达式结果调用不同的成员函数, 这取决于迭代元素的动态类型。

18.2. 静态多态性

模板还可以用来实现多态性，但不依赖基类中常见行为。相反，其共通性隐含在不同“图形”必须支持使用公共语法的操作（即，相关函数必须具有相同的名称）。具体类是彼此独立定义的（参见图 18.2）。当模板用具体的类实例化时，多态功能就启用了。

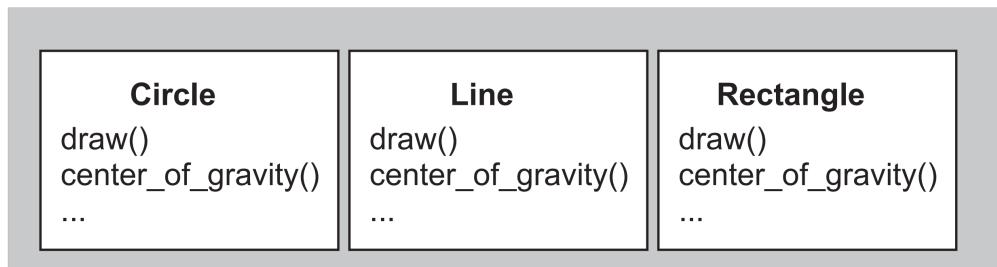


图 18.2. 通过模板实现的多态性

例如，前一节中的 myDraw() 函数：

```
1 void myDraw (GeoObj const& obj) // GeoObj is abstract base class
2 {
3     obj.draw();
4 }
```

可以改写为

```
1 template<typename GeoObj>
2 void myDraw (GeoObj const& obj) // GeoObj is template parameter
3 {
4     obj.draw();
5 }
```

比较 myDraw() 的两种实现，区别是将 GeoObj 规范为模板参数，而不是公共基类。然而，使用动态多态性，在运行时只有一个 myDraw() 函数，而对于模板有不同的函数，如 myDraw<Line>() 和 myDraw<Circle>()。

可以尝试使用静态多态性重新写前一节的示例。首先，有几个单独的几何类，而不是几何类的层次结构：

poly/staticchier.hpp

```
1 #include "coord.hpp"
2 // concrete geometric object class Circle
3 // - not derived from any class
4 class Circle {
5     public:
6         void draw() const;
7         Coord center_of_gravity() const;
8         ...
9     };
10 // concrete geometric object class Line
```

```

12 // - not derived from any class
13 class Line {
14     public:
15     void draw() const;
16     Coord center_of_gravity() const;
17     ...
18 };
19 ...

```

这些类的应用如下所示:

poly/statichier.cpp

```

1 #include "statichier.hpp"
2 #include <vector>
3
4 // draw any GeoObj
5 template<typename GeoObj>
6 void myDraw (GeoObj const& obj)
7 {
8     obj.draw(); // call draw() according to type of object
9 }
10
11 // compute distance of center of gravity between two GeoObjs
12 template<typename GeoObj1, typename GeoObj2>
13 Coord distance (GeoObj1 const& x1, GeoObj2 const& x2)
14 {
15     Coord c = x1.center_of_gravity() - x2.center_of_gravity();
16     return c.abs(); // return coordinates as absolute values
17 }
18
19 // draw homogeneous collection of GeoObjs
20 template<typename GeoObj>
21 void drawElems (std::vector<GeoObj> const& elems)
22 {
23     for (unsigned i=0; i<elems.size(); ++i) {
24         elems[i].draw(); // call draw() according to type of element
25     }
26 }
27
28 int main()
29 {
30     Line l;
31     Circle c, c1, c2;
32
33     myDraw(l); // myDraw<Line>(GeoObj&) => Line::draw()
34     myDraw(c); // myDraw<Circle>(GeoObj&) => Circle::draw()
35
36     distance(c1,c2); // distance<Circle,Circle>(GeoObj1&,GeoObj2&)
37     distance(l,c); // distance<Line,Circle>(GeoObj1&,GeoObj2&)

```

```

38
39 // std::vector<GeoObj*> coll; // ERROR: no heterogeneous collection possible
40 std::vector<Line> coll; // OK: homogeneous collection possible
41 coll.push_back(l); // insert line
42 drawElems(coll); // draw all lines
43 }

```

与 myDraw() 一样, GeoObj 不能再用作 distance() 的具体参数类型。相反, 这里提供了两个模板参数 GeoObj1 和 GeoObj2, 允许在距离计算中接受不同的几何对象类型组合:

```
1 distance(l, c); // distance<Line,Circle>(GeoObj1&,GeoObj2&)
```

然而, 异构集合不能够透明地处理。这就是静态多态性的限制: 所有类型必须在编译时确定, 可以为不同的几何对象类型引入不同的集合。不再要求集合仅限于指针, 因为指针仅在性能和类型安全方面具有优势。

18.3. 动态多态性与静态多态性

先对这两种形式的多态性进行分类和比较。

术语

动态和静态多态为不同的 C++ 编程习惯提供了支持:

有关多态术语的详细讨论, 请参见 [CzarneckiEiseneckerGenProg] 的第 6.5 到 6.7 节。

- 通过继承实现的多态是有界和动态的:
 - 有界意味着参与多态行为的类型的接口, 是由公共基类的设计预先确定的 (这个概念的其他术语描述是 invasive 和 intrusive)。
 - 动态意味着接口的绑定在运行时 (动态) 完成。
- 通过模板实现的多态性是无界和静态的:
 - 无界意味着参与多态行为的类型接口不是预先确定的 (该概念的其他术语描述是 noninvasive 和 nonintrusive)。
 - 静态意味着接口的绑定在编译时完成 (静态)。

严格地说, 在 C++ 语言中, 动态多态性和静态多态性是有界动态多态性和无界静态多态性的快捷方式。其他语言中, 也存在其他组合 (例如, Smalltalk 提供了无界的动态多态性)。在 C++ 的上下文中, 动态多态和静态多态这两个术语不会引起混淆。

优势与不足

C++ 中的动态多态性展示了以下优点:

- 可以优雅地处理异构集合。

- 可执行代码的体积更小(只需要一个多态函数，而必须生成不同的模板实例来处理不同的类型)。
- 代码可以完全编译；因此不需要发布实现源(发布模板库通常需要发布模板实现的源代码)。

C++ 中的静态多态有这些优点：

- 内置类型的集合很容易实现，接口通用性不需要通过公共基类来表示。
- 生成的代码可能更快(因为不需要指针，而且可以更频繁地内联非虚函数)。
- 若应用程序只执行了部分接口，仍然可以使用只提供部分接口的具体类型。

静态多态性通常认为比动态多态性更类型安全，因为所有绑定都在编译时检查。例如，模板实例化的容器中插入错误类型的对象没有危险。然而，需要指向公共基类的指针的容器中，这些指针有可能无意中指向不同类型的对象。

实践中，不同的语义假设隐藏在相同的接口后面时，模板实例化也会造成一些麻烦。当假定关联加法操作符的模板实例化了一个与该操作符无关的类型时，可能会发生意外。这种语义不匹配在基于继承的层次结构中很少发生，可能是因为更明确地指定了接口规范。

两种形式结合

当然，可以结合这两种形式的多态性。例如，可以从公共基类派生不同种类的几何对象，以便能够处理几何对象的异构集合。但仍然可以使用模板，为特定类型的几何对象编写代码。

继承和模板的结合将在第 21 章中进一步介绍，将看到如何将成员函数的虚态参数化，以及如何使用基于继承的奇异递归模板模式(CRTP)，为静态多态性提供灵活性。

18.4. 使用概念

反对模板静态多态的一个论点是，接口的绑定通过实例化相应的模板来完成，没有公共接口(类)用来编程。若所有实例化的代码都有效，那么模板就可以工作。若不是，可能会导致难以理解的错误消息，甚至导致有效但意外的行为。

由于这个原因，C++ 语言设计人员致力于为模板参数显式提供(和检查)接口的能力。这样的接口在 C++ 中称为概念，表示模板参数必须满足一组约束后，才能成功实例化模板。

尽管各路开发者这些年在这个领域做了许多工作，但是概念直到 C++17，才成为标准 C++ 的一部分。一些编译器提供了对这种特性的实验性支持，但这些概念可能会成为 C++17 之后标准的一部分。

例如，GCC 7 提供了-fconcepts 选项。

概念可以理解为静态多态的一种“接口”：

poly/conceptsreq.hpp

```

1 #include "coord.hpp"
2 template<typename T>
3 concept GeoObj = requires(T x) {
4     { x.draw() } -> void;
```

```

5   { x.center_of_gravity() } -> Coord;
6   ...
7 };

```

这里，使用关键字概念来定义概念 GeoObj，约束类型具有可调用成员 draw() 和 center_of_gravity()，并具有适当的结果类型。

现在，可以重写一些示例模板，以包含一个 require 子句，用 GeoObj 概念约束模板参数：

poly/conceptspoly.hpp

```

1 #include "conceptsreq.hpp"
2 #include <vector>
3
4 // draw any GeoObj
5 template<typename T>
6 requires GeoObj<T>
7 void myDraw (T const& obj)
8 {
9     obj.draw(); // call draw() according to type of object
10 }
11
12 // compute distance of center of gravity between two GeoObjs
13 template<typename T1, typename T2>
14 requires GeoObj<T1> && GeoObj<T2>
15 Coord distance (T1 const& x1, T2 const& x2)
16 {
17     Coord c = x1.center_of_gravity() - x2.center_of_gravity();
18     return c.abs(); // return coordinates as absolute values
19 }
20
21 // draw homogeneous collection of GeoObjs
22 template<typename T>
23 requires GeoObj<T>
24 void drawElems (std::vector<T> const& elems)
25 {
26     for (std::size_type i=0; i<elems.size(); ++i) {
27         elems[i].draw(); // call draw() according to type of element
28     }
29 }

```

对于能够参与(静态)多态行为的类型来说，这种方法属于非侵入：

```

1 // concrete geometric object class Circle
2 // - not derived from any class or implementing any interface
3 class Circle {
4     public:
5     void draw() const;
6     Coord center_of_gravity() const;
7     ...
8 };

```

这些类型仍然定义在没有特定基类或需求子句的情况下，并且仍然可以是基本类型，或源自独立框架的类型。

附录 E 包含了对 C++ 概念的更详细的讨论，这些概念将在下一个 C++ 标准中出现。

18.5. 设计模式的新形式

C++ 中静态多态性的可用性，带来了实现经典设计模式的新方法。以桥接模式为例，在许多 C++ 程序中扮演着重要的角色。使用桥接模式的一个目标是在接口的不同实现之间切换。

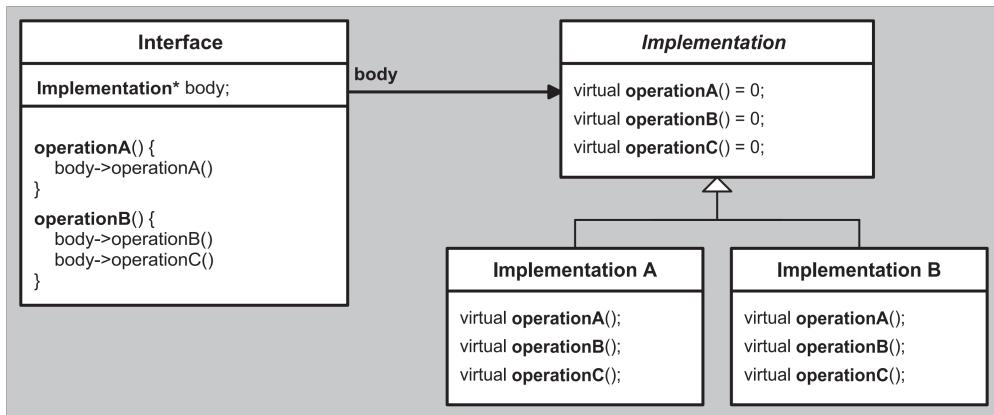


图 18.3. 使用继承实现的桥接模式

根据 [DesignPatternsGoF]，通过一个接口类来完成，嵌入一个指针来引用实际的实现，并通过这个指针委托所有调用 (参见图 18.3)。

但在编译时就知道实现的类型，那么可以利用模板的强大功能 (参见图 18.4)，带来更好的类型安全性 (部分原因是避免了指针转换) 和性能。

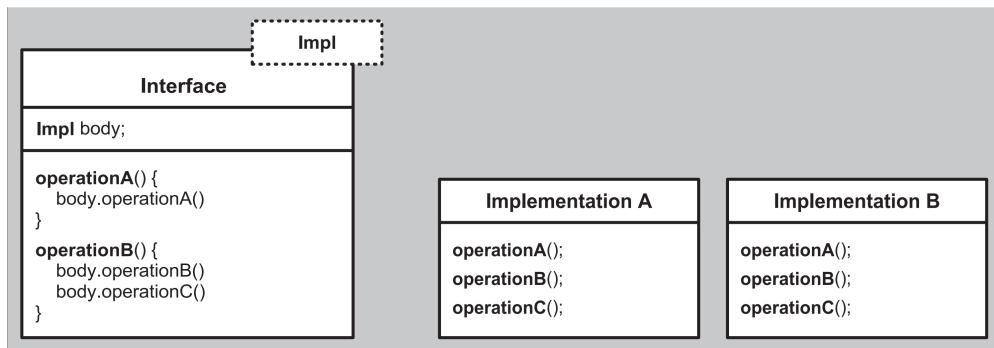


图 18.4. 使用模板实现的桥接模式

18.6. 泛型编程

静态多态性引出了泛型编程的概念。然而，泛型编程没有统一的定义 (就像面向对象编程没有统一的定义一样)。根据 [CzarneckiEiseneckerGenProg]，定义从使用通用参数编程到寻找最抽象的高效算法表示。书中总结道：

泛型编程是计算机科学的一个子学科，处理寻找高效算法、数据结构和其他软件概念的抽象表示，以及系统组织...。泛型编程侧重于表示一系列领域概念。(169 - 170 页)

C++ 上下文中，泛型编程有时定义为使用模板编程（而面向对象编程认为是使用虚函数编程），任何 C++ 模板都可以认为是泛型编程的一个实例。然而，专业人员经常认为泛型编程有其他成分：模板必须在一个框架中设计，以支持多种组合。

目前为止，该领域最重要的贡献是标准模板库 (Standard Template Library, STL)，后来并入到 C++ 标准库中。STL 是一个框架，为对象集合（称为容器）的许多线性数据结构提供了许多有用的操作（称为算法），算法和容器都是模板，但关键是算法不是容器的成员函数。这些算法以通用的方式编写，因此可以用于任何容器（以及元素的线性集合）。为此，STL 的设计者确定了一个抽象的迭代器概念，可以为任何类型的线性集合提供迭代器。本质上，容器操作的集合特定方面已经分解到迭代器的功能中了。

因此，可以实现这样的操作，比如：计算序列中的最大值，而不需要知道值如何存储在序列中的：

```
1 template<typename Iterator>
2 Iterator max_element (Iterator beg, // refers to start of collection
3                      Iterator end) // refers to end of collection
4 {
5     // use only certain Iterator operations to traverse all elements
6     // of the collection to find the element with the maximum value
7     // and return its position as Iterator
8     ...
9 }
```

与为每个线性容器提供所有有用的操作（如 `max_element()`）不同，容器只需要提供一个迭代器类型来遍历所包含的值序列，以及创建迭代器的相应成员函数：

```
1 namespace std {
2     template<typename T, ...>
3     class vector {
4         public:
5             using const_iterator = ...; // implementation-specific iterator
6             ... // type for constant vectors
7             const_iterator begin() const; // iterator for start of collection
8             const_iterator end() const; // iterator for end of collection
9             ...
10    };
11
12    template<typename T, ...>
13    class list {
14        public:
15            using const_iterator = ...; // implementation-specific iterator
16            ... // type for constant lists
17            const_iterator begin() const; // iterator for start of collection
18            const_iterator end() const; // iterator for end of collection
19            ...
20    };
21 }
```

现在，可以通过 `max_element()` 操作，以集合的开始和结束为参数来找到集合中的最大值(空集合的特殊处理省略了)：

poly/printmax.cpp

```
1 #include <vector>
2 #include <list>
3 #include <algorithm>
4 #include <iostream>
5 #include "MyClass.hpp"
6
7 template<typename T>
8 void printMax (T const& coll)
9 {
10    // compute position of maximum value
11    auto pos = std::max_element(coll.begin(), coll.end());
12
13    // print value of maximum element of coll (if any):
14    if (pos != coll.end()) {
15        std::cout << *pos << '\n';
16    }
17    else {
18        std::cout << "empty" << '\n';
19    }
20}
21
22 int main()
23 {
24    std::vector<MyClass> c1;
25    std::list<MyClass> c2;
26    ...
27    printMax(c1);
28    printMax(c2);
29}
```

通过用这些迭代器参数化其操作，STL 避免了操作定义数量的激增。开发者也不再为每个容器实现每个操作，而是只实现一次算法，每个容器就可以使用。通用的粘合剂是迭代器，因为迭代器有一个特定的接口，该接口由容器提供，并由算法使用。这个接口通常称为概念，表示模板必须满足的一组约束，以适应这个框架。此外，这个概念对其他操作和数据结构开放。

可能还记得在 18.4 节之前描述了一个概念语言特性(附录 E 中有更详细的描述)。语言特性与这里的概念完全对应，术语概念由 STL 的设计师首先引入，以形式化他们的工作。此后不久，开始尝试在模板中明确这些概念。

即将到来的语言特性将帮助我们指定和检查迭代器的要求(因为有不同的迭代器类别，如前向迭代器和双向迭代器，将涉及多个相应的概念；参见第 E.3.1 节)。如今的 C++ 中，这些概念大多隐含在泛型库(尤其是标准 C++ 库)的规范中。幸运的是，一些特性和技术(例如，`static_assert` 和 SFINAE)允许自动检查类型。

原则上，类似于 STL 的方法这样的功能可以通过动态多态性实现。但在实践中，因为与虚函数

调用机制相比，其用途有限，迭代器的概念过于轻量级。添加基于虚拟函数的接口层很可能会大大降低操作速度（甚至更慢）。

泛型编程之所以实用，是因为依赖于静态多态性，该多态性在编译时解析接口。另一方面，编译时解析接口的需求也需要相应的设计原则，这些原则与面向对象的设计原则不同。许多重要的通用设计原则将在本书的其余部分进行描述。此外，附录 E 通过描述对概念中的概念的直接语言支持，更深入地探讨了泛型编程。

18.7. 后记

容器类型是将模板引入 C++ 编程语言的主要动机。模板之前，多态层次结构是容器的一种流行方法。一个例子是国立卫生研究院类库 (NIHCL, National Institutes of Health Class Library)，在很大程度上转换了 Smalltalk 容器类的体系（见图 18.5）。

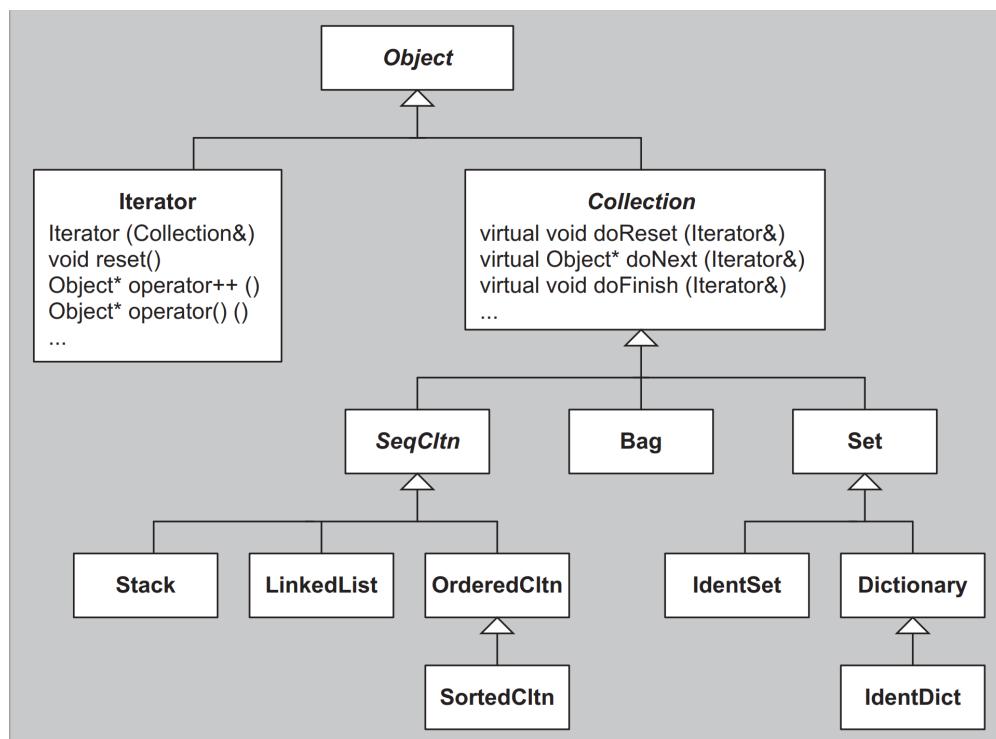


图 18.5. NIHCL 的体系

与 C++ 标准库类似，NIHCL 支持丰富的容器和迭代器。但实现遵循了 Smalltalk 的动态多态风格：迭代器使用抽象基类 Collection 来操作不同类型的集合：

```
1 Bag c1;
2 Set c2;
3 ...
4 Iterator i1(c1);
5 Iterator i2(c2);
6 ...
```

但就运行时间和内存用量而言，这种方法的代价很高。运行时间通常比使用 C++ 标准库的等效代码差几个数量级，因为大多数操作最终都需要一个虚调用（而在 C++ 标准库中，许多操作都是内

联的，迭代器和容器接口中不涉及虚函数)。此外，由于(与Smalltalk不同)接口是有边界的，内置类型必须封装在更大的多态类中(这样的封装器由NIHCL提供)，这反过来可能会导致存储需求的急剧增加。

即使在如今的模板时代，许多项目仍然在多态性的方法上做出次优选择。显然，动态多态性都是正确的选择，异构迭代就是一个例子。同样地，许多编程任务都可以使用模板有效地解决，同构容器就是一个例子。

静态多态可以很好地编写基本计算结构，而选择公共基类型的需要意味着动态多态库通常必须做出特定于领域的选择。因此，C++标准库的STL部分从未包含多态容器也不足为奇，但包含了一组使用静态多态的容器和迭代器(如第18.6节所示)。

中型和大型C++程序通常需要处理本章讨论的两种多态性。某些情况下，有必要将它们紧密地结合起来使用。根据我们的讨论，最优的设计选择很明确，花一些时间考虑长期的、潜在的回报是值得的。

第 19 章 特征的实现

模板能够对各种类型的类和函数进行参数化，引入尽可能多的模板参数以实现对类型或算法的定制。通过这种方式，“模板化”组件可以实例化，以满足外部代码的需求。然而，引入几十个模板参数来实现最大参数化其实没有必要。必须在外部代码中指定所有相应的参数也非常繁琐，而且模板参数会使组件与其外部代码之间的关系复杂化。

我们引入的大多数参数都有默认值，参数可以由几个主要参数决定，这些参数可以省略。也可以给其他参数提供依赖于主要参数的默认值，这可以满足大多数情况，但默认值偶尔需要重写(对于特殊应用程序)。然而，其他参数与主要参数无关：除了存在几乎符合要求的默认值外，其本身就是主要参数。

特征(或特征模板)是 C++ 编程工具，极大地促进了工业化模板设计中出现的那种参数管理。本章中，将证明这种模板是有用的，并演示各种技术，这些技术能够编写更健壮和强大的工具。

这里提供的大多数特性都可以在 C++ 标准库中，以某种形式使用。为了简单起见，我们给出简化的实现，其中省略了工业化实现(如标准库的那些)中的一些细节。出于这个原因，我们还使用了自己的命名方案，其很容易与标准特征对应。

19.1. 实例：累加一个序列

计算一系列值的和是一项相当常见的计算任务。然而，这个简单的问题为我们提供了一个很好的例子，可以引入策略类和特征解决各个层面的问题。

19.1.1 固化特征

首先，假设要计算的 sum 的值存储在一个数组中，并有一个指向要累计的第一个元素的指针和一个指向要累计的最后一个元素的下一位置的指针。因为本书专注于模板，所以希望编写一个适用于多种类型的模板。现在看来，似乎很简单：

为了简单起见，本节中的大多数示例都使用普通指针。显然，工业化的接口可能更偏向使用 C++ 标准库的迭代器(参见 [JosuttisStdLib])。

traits/accum1.hpp

```
1 #ifndef ACCUM_HPP
2 #define ACCUM_HPP
3
4 template<typename T>
5 T accum (T const* beg, T const* end)
6 {
7     T total{}; // assume this actually creates a zero value
8     while (beg != end) {
9         total += *beg;
10        ++beg;
11    }
12 }
```

```

12     return total;
13 }
14
15 #endif // ACCUM_HPP

```

这里唯一的决定是如何创建正确的 0 值类型来求和。我们使用值初始化 (使用 `{...}`)，在第 5.2 节中介绍。局部对象 `total` 要么由其默认构造函数初始化，要么由 0 初始化 (指针为 `nullptr`，布尔值为 `false`)。

这是我们的第一个特征模板，代码使用了 `accum()`:

traits/accum1.cpp

```

1 #include "accum1.hpp"
2 #include <iostream>
3 int main()
4 {
5     // create array of 5 integer values
6     int num[] = { 1, 2, 3, 4, 5 };
7
8     // print average value
9     std::cout << "the average value of the integer values is "
10    << accum(num, num+5) / 5
11    << '\n';
12
13    // create array of character values
14    char name[] = "templates";
15    int length = sizeof(name)-1;
16
17    // (try to) print average character value
18    std::cout << "the average value of the characters in \""
19    << name << "\" is "
20    << accum(name, name+length) / length
21    << '\n';
22 }

```

前半部分，使用 `accum()` 来计算五个整数值的和:

```

1 int num[] = { 1, 2, 3, 4, 5 };
2 ...
3 accum(num0, num+5)

```

然后，将所得和除以数组中值的数量，即可获得整数值的平均值。

后半部分尝试对单词“`templates`”中的所有字母执行相同的操作 (前提是 `a` 到 `z` 的字符在实际字符集中形成一个连续的序列，这对 ASCII 是成立的，但对 EBCDIC 不成立)。结果应该位于 `a` 的值和 `z` 的值之间。大多数平台上，这些值是由 ASCII 码决定的: `a` 编码为 97，`z` 编码为 122。因此，我们可以预期结果在 97 和 122 之间。但在我们的平台上，程序的输出为:

```
the average value of the integer values is 3
the average value of the characters in "templates" is -5
```

这里的问题是，我们的模板为 `char` 类型实例化，结果是对于相对较小的值的积累来说，其范围太小。显然，可以通过引入一个模板参数 `AccT` 来解决这个问题，该参数描述了变量 `total` 使用的类型（以及返回类型）。然而，这将给模板的所有用户带来负担：在每次调用模板时指定额外的类型。例子中，可能需要这样：

```
1 accum<int>(name, name+5)
```

这个约束并不过分，但可以避免。

使用额外参数的另一种方法是，在调用 `accum()` 的每个类型 `T` 与应该用于保存累积值的对应类型之间创建关联。这种关联可以认为是 `T` 类型的特征，因此计算总和的类型有时称为 `T` 的特征。事实证明，关联可以编码为模板的特化形式：

traits/accumtraits2.hpp

```
1 template<typename T>
2 struct AccumulationTraits;
3
4 template<>
5 struct AccumulationTraits<char> {
6     using AccT = int;
7 };
8
9 template<>
10 struct AccumulationTraits<short> {
11     using AccT = int;
12 };
13
14 template<>
15 struct AccumulationTraits<int> {
16     using AccT = long;
17 };
18
19 template<>
20 struct AccumulationTraits<unsigned int> {
21     using AccT = unsigned long;
22 };
23
24 template<>
25 struct AccumulationTraits<float> {
26     using AccT = double;
27 };
```

`AccumulationTraits` 模板称为特征模板，它保存了一个与其参数类型相同的特征（可以有不止一

个特征和参数)。我们选择不提供该模板的泛型定义，因为不知道累积类型是什么时，没有办法来选择累积类型。然而，可以认为 T 本身是这种类型的一个候选。

可以重写 accum() 模板：

C++11 中，必须声明类似 AccT 类型的返回类型。

traits/accum2.hpp

```
1 #ifndef ACCUM_HPP
2 #define ACCUM_HPP
3
4 #include "accumtraits2.hpp"
5 template<typename T>
6 auto accum (T const* beg, T const* end)
7 {
8     // return type is traits of the element type
9     using AccT = typename AccumulationTraits<T>::AccT;
10
11    AccT total{}; // assume this actually creates a zero value
12    while (beg != end) {
13        total += *beg;
14        ++beg;
15    }
16    return total;
17 }
18
19#endif // ACCUM_HPP
```

然后，示例程序的输出就会成为我们期望的结果：

```
the average value of the integer values is 3
the average value of the characters in "templates" is 108
```

我们添加了一个非常有用的机制来定制算法，但变化不是很显著。此外，若出现了与 accum() 一起使用的新类型，只需声明 AccumulationTraits 模板的显式特化，就可以将适当的 AccT 与其关联起来。这可以用于任何类型：基本类型、在其他库中声明的类型等。

19.1.2 值特征

已经了解了特征表示与给定的“主”类型相关的类型信息。本节中，将说明这些信息不需要局限于类型，常量和其他值类也可以与类型关联。

原来的 accum() 模板使用返回值的默认构造函数来初始化结果变量，希望其值是一个类似于 0 的值：

```
1 AccT total{}; // assume this actually creates a zero value
```

```
2 ...
3 return total;
```

显然，不能保证这能产生一个值来启动积累循环。类型 AccT 甚至可能没有默认构造函数。同样，特征也能起到帮助作用。例子中，可以在 AccumulationTraits 中添加一个新值特征：

traits/accumtraits3.hpp

```
1 template<typename T>
2 struct AccumulationTraits;
3
4 template<>
5 struct AccumulationTraits<char> {
6     using AccT = int;
7     static AccT const zero = 0;
8 };
9
10 template<>
11 struct AccumulationTraits<short> {
12     using AccT = int;
13     static AccT const zero = 0;
14 };
15
16 template<>
17 struct AccumulationTraits<int> {
18     using AccT = long;
19     static AccT const zero = 0;
20 };
21 ...
```

本例中，新特征提供了一个 0 元素作为常量，可以在编译时计算。因此，`accum()` 的表达式如下所示：

traits/accum3.hpp

```
1 #ifndef ACCUM_HPP
2 #define ACCUM_HPP
3
4 #include "accumtraits3.hpp"
5
6 template<typename T>
7 auto accum (T const* beg, T const* end)
8 {
9     // return type is traits of the element type
10    using AccT = typename AccumulationTraits<T>::AccT;
11
12    AccT total = AccumulationTraits<T>::zero; // init total by trait value
13    while (beg != end) {
14        total += *beg;
```

```

15     ++beg;
16 }
17 return total;
18 }
19
20 #endif // ACCUM_HPP

```

代码中，累加变量的初始化很简单：

```
1 Acct total = AccumulationTraits<T>::zero;
```

这种表达式的缺点是，C++ 只允许在其类中初始化具有整型或枚举类型的静态常量数据成员。

`constexpr` 静态数据成员更通用一些，允许使用浮点类型和其他字符类型：

```

1 template<>
2 struct AccumulationTraits<float> {
3     using Acct = float;
4     static constexpr float zero = 0.0f;
5 };

```

但 `const` 和 `constexpr` 都不允许以这种方式初始化非字符类型。例如，因为它通常在堆上分配，用户定义的 `BigInt` 类型可能不是字符类型；或者，只是因为所需的构造函数不是 `constexpr`。所以以下特化错误：

```

1 class BigInt {
2     BigInt(long long);
3     ...
4 };
5 ...
6 template<>
7 struct AccumulationTraits<BigInt> {
8     using Acct = BigInt;
9     static constexpr BigInt zero = BigInt{0}; // ERROR: not a literal type
10};

```

直接的替代方法是不在其类中定义值特征：

```

1 template<>
2 struct AccumulationTraits<BigInt> {
3     using Acct = BigInt;
4     static BigInt const zero; // declaration only
5 };

```

然后，在某个源文件中使用初始化式：

```
1 BigInt const AccumulationTraits<BigInt>::zero = BigInt{0};
```

尽管这样可以工作，但缺点是代码更冗长（必须在两个地方添加代码），而且效率更低，因为编译器通常不知道其他文件中的定义。

C++17 中，可以通过内联变量来解决：

```
1 template<>
2 struct AccumulationTraits<BigInt> {
3     using AccT = BigInt;
4     inline static BigInt const zero = BigInt{0}; // OK since C++17
5 };
```

C++17 前，一种可行的替代方法是对值特征使用内联成员函数，这些值特征并不总是产生整数值。同样，若该函数返回字符类型，则可以将其声明为 `constexpr`。

大多数现代 C++ 编译器都可以“看懂”简单内联函数的调用。此外，使用 `constexpr` 可以在表达式必须为常量的上下文中使用值特征(例如，在模板参数中)。

可以这样重写 `AccumulationTraits`:

traits/accumtraits4.hpp

```
1 template<typename T>
2 struct AccumulationTraits;
3
4 template<>
5 struct AccumulationTraits<char> {
6     using AccT = int;
7     static constexpr AccT zero() {
8         return 0;
9     }
10 } ;
11
12 template<>
13 struct AccumulationTraits<short> {
14     using AccT = int;
15     static constexpr AccT zero() {
16         return 0;
17     }
18 } ;
19
20 template<>
21 struct AccumulationTraits<int> {
22     using AccT = long;
23     static constexpr AccT zero() {
24         return 0;
25     }
26 } ;
27
28 template<>
29 struct AccumulationTraits<unsigned int> {
30     using AccT = unsigned long;
31     static constexpr AccT zero() {
32         return 0;
```

```

33     }
34 };
35
36 template<>
37 struct AccumulationTraits<float> {
38     using AccT = double;
39     static constexpr AccT zero() {
40         return 0;
41     }
42 };
43 ..

```

将这些特征扩展到我们的类型上:

traits/accumtraits4bigint.hpp

```

1 template<>
2 struct AccumulationTraits<BigInt> {
3     using AccT = BigInt;
4     static BigInt zero() {
5         return BigInt{0};
6     }
7 };

```

对于应用程序代码，区别是使用了函数调用语法(而不是对静态数据成员更简单的访问):

```

1 AccT total = AccumulationTraits<T>::zero(); // init total by trait function

```

显然，特征不仅仅是类型。可以作为一种机制，提供 `accum()` 所需调用元素类型的所有必要信息。这是特征概念的关键: 特征为泛型计算提供了配置具体元素(主要是类型)的途径。

19.1.3 参数化特征

前面章节 `accum()` 中是对特征的固化使用，当定义了解耦的特征，就不能在算法中替换。在某些情况下，这种重写是可行的。可能知道一组浮点值可以安全地求和为一个相同类型的变量，这样做可能提高效率。

可以通过为特征添加一个模板参数 AT 来解决这个问题，AT 的默认值由特征模板决定:

traits/accum5.hpp

```

1 #ifndef ACCUM_HPP
2 #define ACCUM_HPP
3
4 #include "accumtraits4.hpp"
5
6 template<typename T, typename AT = AccumulationTraits<T>>
7 auto accum (T const* beg, T const* end)
8 {
9     typename AT::AccT total = AT::zero();

```

```

10    while (beg != end) {
11        total += *beg;
12        ++beg;
13    }
14    return total;
15 }
16
17 #endif // ACCUM_HPP

```

通过这种方式，许多用户可以省略不必要的模板参数，但那些有更特殊需求的用户可以指定预置累积类型的替代方案。因为可以通过第一个参数推导出的每个类型配置适当的默认值，所以该模板的大多数用户永远不需要显式地提供第二个模板参数。

19.2. 特征、策略和策略类

我们已经把累加和求和搞定了，可以想象其他种类的积累，可以将给定值序列相乘；或者，将字符串连接起来。即使在序列中找到最大值，也可以表述为一个积累问题。所有这些替代方案中，唯一需要更改的 `accum()` 操作是 `total += *beg`。这个操作可以称为积累策略。

下面是如何在 `accum()` 函数模板中引入策略的示例：

traits/accum6.hpp

```

1 #ifndef ACCUM_HPP
2 #define ACCUM_HPP
3
4 #include "accumtraits4.hpp"
5 #include "sumpolicy1.hpp"
6
7 template<typename T,
8         typename Policy = SumPolicy,
9         typename Traits = AccumulationTraits<T>>
10 auto accum (T const* beg, T const* end)
11 {
12     using AccT = typename Traits::AccT;
13     AccT total = Traits::zero();
14     while (beg != end) {
15         Policy::accumulate(total, *beg);
16         ++beg;
17     }
18     return total;
19 }
20
21 #endif // ACCUM_HPP

```

`accum()` 的这个版本中，`SumPolicy` 是一个策略类，也就是说，这个类通过一个接口为一个算法实现一个或多个策略。

可以将其推广为策略参数，可以是类(如前所述)或指向函数的指针。

SumPolicy 可以这样写：

traits/sumpolicy1.hpp

```
1 #ifndef SUMPOLICY_HPP
2 #define SUMPOLICY_HPP
3 class SumPolicy {
4     public:
5         template<typename T1, typename T2>
6             static void accumulate (T1& total, T2 const& value) {
7                 total += value;
8             }
9         ;
10 #endif // SUMPOLICY_HPP
```

通过指定不同的策略来积累值，可以计算不同的东西。下面的程序，打算确定一些值的乘积：

traits/accum6.cpp

```
1 #include "accum6.hpp"
2 #include <iostream>
3
4 class MultPolicy {
5     public:
6         template<typename T1, typename T2>
7             static void accumulate (T1& total, T2 const& value) {
8                 total *= value;
9             }
10    ;
11 int main()
12 {
13     // create array of 5 integer values
14     int num[] = { 1, 2, 3, 4, 5 };
15
16     // print product of all values
17     std::cout << "the product of the integer values is "
18     << accum<int,MultPolicy>(num, num+5)
19     << '\n' ;
20 }
```

然而，这个程序的输出并不是我们想要的：

```
the product of the integer values is 0
```

这里的问题是由初值的选择引起的: 尽管 0 适用于求和, 但它不适用于乘法(零初值会导致累积乘法的结果为零)。这说明了不同的特征和政策可能会相互影响, 强调了设计模板的重要性。

可以了解到累积循环的初始化, 也是累积策略的一部分。这个策略可能使用特征 `zero()`, 也可能不使用。其他替代方案也不能忽略: 不是所有问题都必须用特性和策略来解决。例如, C++ 标准库的 `std::accumulate()` 函数将初始值作为第三个(函数调用)参数。

19.2.1 特征和策略: 有什么不同?

合理的例子可以支持这样一个事实, 即策略只是特征的特殊情况。相反, 也可以说, 特征就是策略的一种编码。

《新简明牛津英语词典》(参见 [NewShorterOED]) 是这样说的:

- trait n. …特征: 表征一事物的显著特征
- policy n. …作为有利或权宜之计而采取的行动方针

基于此, 我们倾向于将术语策略类的使用限制为对某种类型的操作进行编码的类, 这些操作与其他模板参数基本上是无关的。这与 Andrei Alexandrescu 在他的《现代 C++ 设计》一书中的观点一致(见 [AlexandrescuDesign] 第 8 页):

Alexandrescu 一直是策略界的主要声音, 他在此基础上开发了一套丰富的技术。

政策与特征有很多共同点, 但不同之处在于它们不太强调类型, 而更多地强调行为。

引入特征技术的 Nathan Myers 提出了更开放的定义(见 [MyersTraits]):

特征类: 用来代替模板参数的类。作为一个类, 聚合有用的数据和常量; 作为模板, 为解决所有“间接级”软件问题的提供了一条途径。

因此, 我们倾向于使用以下(稍微模糊的)定义:

- 特征表示模板参数的自然属性。
- 策略表示泛型函数和类型的可配置行为(通常带有一些常用的默认值)。

为了进一步阐述这两个概念之间的区别, 我们列出了以下关于特征的观察结果:

- 特性可以固化使用(例如, 不需要通过模板参数传递)。
- 特性参数通常有非常自然的默认值(很少重写, 或者不能重写)。
- 特性参数往往紧密地依赖于一个或多个主要参数。
- 特征主要组合类型和常量, 而不是成员函数。
- 特征倾向于在特征模板中进行收集。

对于策略类, 有以下观察:

- 若策略类没有作为模板参数传递, 那其作用不大。
- 策略参数不需要有默认值, 通常显式指定(尽管许多通用组件都配置了常用的默认策略)。
- 策略参数大多与模板的其他参数无关。
- 策略类通常组合成员函数。
- 策略可以在普通类或类模板中进行收集。

然而，这两个术语之间肯定有一条模糊的界线。例如，C++ 标准库的字符特征也定义了函数行为，如比较、移动和查找字符。通过替换这些特征，可以定义不区分大小写的字符串类（参见 [JosuttisStdLib] 中的 13.2.15 节），同时保持相同的字符类型。尽管称为特征，也有一些与策略相关的属性。

19.2.2 成员模板与双重模板参数

为了实现累积策略，选择将 SumPolicy 和 multipolicy 表示为具有成员模板的普通类。另一种方法是使用类模板设计策略类接口，然后将类模板用作模板参数（参见第 5.7 节和第 12.2.3 节）。可以将 SumPolicy 重写为模板：

traits/sumpolicy2.hpp

```
1 #ifndef SUMPOLICY_HPP
2 #define SUMPOLICY_HPP
3
4 template<typename T1, typename T2>
5 class SumPolicy {
6     public:
7         static void accumulate (T1& total, T2 const& value) {
8             total += value;
9         }
10    };
11
12 #endif // SUMPOLICY_HPP
```

Accum 的接口可以适应使用双重模板参数：

traits/accum7.hpp

```
1 #ifndef ACCUM_HPP
2 #define ACCUM_HPP
3
4 #include "accumtraits4.hpp"
5 #include "sumpolicy2.hpp"
6
7 template<typename T,
8          template<typename, typename> class Policy = SumPolicy,
9          typename Traits = AccumulationTraits<T>>
10 auto accum (T const* beg, T const* end)
11 {
12     using AccT = typename Traits::AccT;
13     AccT total = Traits::zero();
14     while (beg != end) {
15         Policy<AccT, T>::accumulate(total, *beg);
16         ++beg;
17     }
18     return total;
```

```
19 }
20
21 #endif // ACCUM_HPP
```

同样的变换也可以应用于特征参数(其他变体也有可能:与其显式地将 AccT 类型传递给策略类型,不如传递积累特征,并让策略从特征参数确定其结果的类型)。

通过双重模板参数访问策略类的主要优点是,使策略类更容易携带一些依赖于模板参数的类型的状态信息(即静态数据成员)。(第一种方法中,静态数据成员必须嵌入到成员类模板中。)

然而,双重模板参数方法的缺点是,现在必须将策略类改写为模板,并使用接口定义的一组精确的模板参数。这会使特征本身的表达式比简单的非模板类更冗长、更不自然。

19.2.3 组合多种策略和/或特征

特性和策略并没有完全消除多个模板参数,但确实将数量减少到可控的程度。那么,如何排列这些参数?

一个简单的策略,根据参数默认值选择的可能性的增加对参数进行排序。通常,特征参数遵循策略参数,因为后者在外部代码中更经常重写。(细心的读者可能在演示过程中注意到了这个策略)

如果增加大量的复杂性代码,那么存在一种替代方法,可以任意顺序指定非默认参数。有关详细信息,请参阅第 21.4 节。

19.2.4 通用迭代器的累加

结束对特性和策略的介绍之前,有必要看看 `accum()` 的另一个版本,它增加了处理通用迭代器(而不仅仅是指针)的功能,这是业界通用组件所期望的。这仍然允许用指针调用 `accum()`,因为 C++ 标准库提供了迭代器特性(特征到处都是!)。因此,可以这样定义 `accum()` 的初始版本(忽略后面的改进):

C++11 中,必须将返回类型声明为 VT。

traits/accum0.hpp

```
1 #ifndef ACCUM_HPP
2 #define ACCUM_HPP
3
4 #include <iostream>
5
6 template<typename Iter>
7 auto accum (Iter start, Iter end)
8 {
9     using VT = typename std::iterator_traits<Iter>::value_type;
10    VT total{}; // assume this actually creates a zero value
11    while (start != end) {
12        total += *start;
13        ++start;
14    }
}
```

```
15     return total;
16 }
17
18 #endif // ACCUM_HPP
```

std::iterator_traits 结构封装了迭代器的所有相关属性。因为指针的偏特化存在，所以这些特征可以与普通指针类型一起使用。以下是标准库的实现：

```
1 namespace std {
2     template<typename T>
3     struct iterator_traits<T*> {
4         using difference_type = ptrdiff_t;
5         using value_type = T;
6         using pointer = T*;
7         using reference = T&;
8         using iterator_category = random_access_iterator_tag ;
9     };
10 }
```

因为没有迭代器所指向值的累积类型，所以需要自行设计 AccumulationTraits。

19.3. 类型函数

初始特征示例说明，可以定义依赖于类型的行为。在 C/C++ 中，定义的函数可称为值函数：其接受一些值作为参数，并返回另一个值作为结果。通过模板，还可以定义类型函数：接受某种类型作为参数，并生成类型或常量作为结果的函数。

sizeof 是一个有用的内置类型函数，其返回一个常量，描述给定类型参数的大小（以字节为单位）。类模板也可以用作类型函数。类型函数的参数是模板参数，结果作为成员类型或成员常量提取。sizeof 操作符可以实现如下接口：

traits/sizeof.cpp

```
1 #include <cstddef>
2 #include <iostream>
3
4 template<typename T>
5 struct TypeSize {
6     static std::size_t const value = sizeof(T);
7 };
8
9 int main()
10 {
11     std::cout << "TypeSize<int>::value = "
12     << TypeSize<int>::value << '\n' ;
13 }
```

因为有内置的 sizeof 操作符可用，但 TypeSize<T> 是一个类型，因此可以作为类模板参数传递。或者，TypeSize 是一个模板，可以作为模板参数传递。

接下来的内容中，将开发更多的通用类型函数，并可以以这种方式作为特征类。

19.3.1 元素类型

假设有许多容器模板 (`std::vector<T>` 和 `std::list<T>`)，以及内置数组。需要一个类型函数，给定这样的容器类型，该类型函数生成元素类型。这可以通过使用偏特化来实现：

traits/elementtype.hpp

```
1 #include <vector>
2 #include <list>
3
4 template<typename T>
5 struct ElementT; // primary template
6
7 template<typename T>
8 struct ElementT<std::vector<T>> { // partial specialization for std::vector
9     using Type = T;
10 }
11
12 template<typename T>
13 struct ElementT<std::list<T>> { // partial specialization for std::list
14     using Type = T;
15 }
16
17 ...
18 template<typename T, std::size_t N>
19 struct ElementT<T[N]> { // partial specialization for arrays of known bounds
20     using Type = T;
21 }
22
23 template<typename T>
24 struct ElementT<T[]> { // partial specialization for arrays of unknown bounds
25     using Type = T;
26 }
27 ...
```

应该为所有可能的数组类型提供偏特化(详见第 5.4 节)。

可以这样使用类型函数：

traits/elementtype.cpp

```
1 #include "elementtype.hpp"
2 #include <vector>
3 #include <iostream>
4 #include <typeinfo>
5
6 template<typename T>
7 void printElementType (T const& c)
```

```

8 {
9   std::cout << "Container of "
10  << typeid(typename ElementT<T>::Type).name()
11  << " elements.\n";
12 }
13
14 int main()
15 {
16   std::vector<bool> s;
17   printElementType(s);
18   int arr[42];
19   printElementType(arr);
20 }
```

偏特化允许实现类型函数，而不需要容器知道类型。然而，类型函数是与适用类型一起设计的，并且可以简化实现。若容器类型定义了一个成员类型 `value_type`(就像标准容器所做的那样)，就可以这样：

```

1 template<typename C>
2 struct ElementT {
3   using Type = typename C::value_type;
4 };
```

这可以是默认实现，并且不排除没有定义合适的成员类型 `value_type` 的容器类型的特化。

尽管如此，还是建议为类模板类型参数提供成员类型定义，以便在泛型代码中更容易访问(就像标准容器模板那样)：

```

1 template<typename T1, typename T2, ...>
2 class X {
3   public:
4     using ... = T1;
5     using ... = T2;
6     ...
7 };
```

如何使用类型函数？其可以根据容器类型对模板进行参数化，而不需要元素类型和其他特征的参数。例如，

```

1 template<typename T, typename C>
2 T sumOfElements (C const& c);
```

需要 `sumOfElements<int>(list)` 这样的语法来显式指定元素类型，可以声明

```

1 template<typename C>
2 typename ElementT<C>::Type sumOfElements (C const& c);
```

其中元素类型由类型函数确定。

观察特征是如何作为现有类型扩展实现的，甚至可以为基本类型和封闭库的类型定义这些类型函数。

因为用于访问给定容器类型 C 的特征(通常，类中可以收集多个特征)，所以 `ElementT` 类型称为特征类。因此，特征类不局限于描述容器参数的特征，而是描述任何类型的“主要参数”。

方便起见，可以为类型函数创建别名模板。例如，可以引入

```
1 template<typename T>
2 using ElementType = typename ElementT<T>::Type;
```

这允许进一步简化上面的 sumOfElements 声明为

```
1 template<typename C>
2 ElementType<C> sumOfElements (C const& c);
```

19.3.2 特征变换

除了提供对主参数类型的特定方面的访问外，特征还可以对类型执行转换，例如添加或删除引用或 const 和 volatile 限定符。

删除引用

可以实现一个 RemoveReferenceT 特征，将引用类型转换为的底层对象或函数类型，只保留非引用类型：

traits/removerefERENCE.hpp

```
1 template<typename T>
2 struct RemoveReferenceT {
3     using Type = T;
4 };
5
6 template<typename T>
7 struct RemoveReferenceT<T&> {
8     using Type = T;
9 };
10
11 template<typename T>
12 struct RemoveReferenceT<T&&> {
13     using Type = T;
14 };
```

同样，别名模板可使使用方式更简单：

```
1 template<typename T>
2 using RemoveReference = typename RemoveReferenceT<T>::Type;
```

有时会产生引用类型的构造派生类型时，可以从类型中删除引用，例如在第 15.6 节中讨论的 T&& 类型的函数参数的特殊推导规则。

C++ 标准库提供了相应的类型特征 std::remove_reference<>，在第 D.4 节中描述。

添加引用

可以找一个已存在的类型，并从中创建一个左值或右值引用（以及别名模板）：

traits/addreference.hpp

```
1 template<typename T>
2 struct AddLValueReferenceT {
3     using Type = T&;
4 };
5
6 template<typename T>
7 using AddLValueReference = typename AddLValueReferenceT<T>::Type;
8
9 template<typename T>
10 struct AddRValueReferenceT {
11     using Type = T&&;
12 };
13
14 template<typename T>
15 using AddRValueReference = typename AddRValueReferenceT<T>::Type;
```

引用折叠规则(第15.6节)适用于此。调用 `AddLValueReference<int&&>` 会产生类型 `int&&`(因此不需要通过偏特化手动实现)。

若保留 `AddLValueReferenceT` 和 `AddRValueReferenceT`, 不引入其特化, 那么别名实际上可以简化为

```
1 template<typename T>
2 using AddLValueReferenceT = T&;
3
4 template<typename T>
5 using AddRValueReferenceT = T&&;
```

可以在不实例化类模板的情况下进行实例化(轻量级过程)。这是有风险的, 因为很可能想为特殊情况特化这些模板。如上所述, 不能使用 `void` 作为这些模板的模板参数。一些显式特化可以解决这个问题:

```
1 template<>
2 struct AddLValueReferenceT<void> {
3     using Type = void;
4 };
5
6 template<>
7 struct AddLValueReferenceT<void const> {
8     using Type = void const;
9 };
10
11 template<>
12 struct AddLValueReferenceT<void volatile> {
13     using Type = void volatile;
14 };
15
16 template<>
```

```
17 struct AddLValueReferenceT<void const volatile> {
18     using Type = void const volatile;
19 }
```

AddRValueReferenceT 也类似，必须根据类模板来制定别名模板，以确保采用特化(别名模板不能特化)模板。

C++ 标准库提供了相应的类型特征 std::add_lvalue_reference<> 和 std::add_rvalue_reference<>，在第 D.4 节中有描述。标准模板包括 void 类型的特化。

删除限定符

转换特征可以分解或引入任何类型的复合类型，而不仅仅是引用。如果存在 const 限定符，可以删除：

traits/removeconst.hpp

```
1 template<typename T>
2 struct RemoveConstT {
3     using Type = T;
4 }
5
6 template<typename T>
7 struct RemoveConstT<T const> {
8     using Type = T;
9 }
10
11 template<typename T>
12 using RemoveConst = typename RemoveConstT<T>::Type;
```

此外，特征转换可以组合，比如创建一个 RemoveCVT 特征，可以同时删除 const 和 volatile：

traits/removecv.hpp

```
1 #include "removeconst.hpp"
2 #include "removevolatile.hpp"
3
4 template<typename T>
5 struct RemoveCVT : RemoveConstT<typename RemoveVolatileT<T>::Type> {
6 }
7
8 template<typename T>
9 using RemoveCV = typename RemoveCVT<T>::Type;
```

关于 RemoveCVT 的定义，有两点需要注意。首先，同时使用 RemoveConstT 和 RemoveVolatileT，首先删除 volatile(若存在)，然后将结果类型传递给 RemoveConstT。

限定符删除的顺序没有语义上的影响：可以先删除 volatile，然后再删除 const。

其次，使用元函数转发从 RemoveConstT 继承 Type 成员，而不是声明自己与 RemoveConstT 特化中相同的 Type 成员。这里使用元函数转发只是为了减少 RemoveCVT 定义中的输入量，当没有为所有输入定义元函数时，元函数转发也很有用，这种技术将在第 19.4 节中进一步讨论。

别名模板 RemoveCV 可以简化为

```
1 template<typename T>
2 using RemoveCV = RemoveConst<RemoveVolatile<T>>;
```

同样，这只在 RemoveCVT 没有特化的情况下有效。与 AddLValueReference 和 AddRValueReference 不同，没有原因需要进行这样的特化。

C++ 标准库也提供了相应的类型特征 std::remove_volatile<>, std::remove_const<>, 和 std::remove_cv<>, 在第 D.4 节中有描述。

衰变

为了完善对特征转换的讨论，我们开发了一个特征，按值向参数传递时模拟类型转换。派生自 C 语言，参数会产生衰变(将数组类型转换为指针，将函数类型转换为指向函数的指针类型；请参阅第 7.4 节和第 11.1.1 节)，并删除所有 const、volatile 或引用限定符(因为解析函数调用时将忽略参数类型的类型限定符)。

这种按值传递的效果可以在下面的程序中看到，打印出编译器衰变指定类型后产生的实际参数类型：

traits/passbyvalue.cpp

```
1 #include <iostream>
2 #include <typeinfo>
3 #include <type_traits>
4
5 template<typename T>
6 void f(T)
7 { }
8
9 template<typename A>
10 void printParameterType(void (*)(A))
11 {
12     std::cout << "Parameter type: " << typeid(A).name() << '\n' ;
13     std::cout << "- is int: " << std::is_same<A,int>::value << '\n' ;
14     std::cout << "- is const: " << std::is_const<A>::value << '\n' ;
15     std::cout << "- is pointer: " << std::is_pointer<A>::value << '\n' ;
16 }
17
18 int main()
19 {
20     printParameterType(&f<int>);
21     printParameterType(&f<int const>);
22     printParameterType(&f<int[7]>);
23     printParameterType(&f<int(int)>);
24 }
```

程序的输出中，int 参数保持不变，但是 int const、int[7] 和 int(int) 形参分别衰变为 int、int* 和 int(*)(int)。

可以实现一个特征，产生相同的按值传递类型转换。为了与 C++ 标准库特征 std::decay 相匹配，我们把它命名为 DecayT。

使用术语衰变可能会有点混乱，因为在 C 中它只意味着从数组/函数类型到指针类型的转换，而在这里它还包括删除 const/volatile 限定符。

它的实现结合了上面描述的几种技术。

首先，定义了非数组、非函数的情况，可以简单地删除 const 和 volatile 限定符：

```
1 template<typename T>
2 struct DecayT : RemoveCVT<T> {
3 };
```

接下来，处理数组到指针的衰变，要求使用偏特化来识别数组类型（有或没有绑定）：

```
1 template<typename T>
2 struct DecayT<T[]> {
3     using Type = T*;
4 };
5
6 template<typename T, std::size_t N>
7 struct DecayT<T[N]> {
8     using Type = T*;
9 };
```

最后，处理函数到指针的衰变，必须匹配任何函数类型，而不管返回类型或参数类型的数量。为此，这里使用可变参数模板：

```
1 template<typename R, typename... Args>
2 struct DecayT<R(Args...)> {
3     using Type = R (*)(Args...);
4 };
5
6 template<typename R, typename... Args>
7 struct DecayT<R(Args..., ...)> {
8     using Type = R (*)(Args..., ...);
9 };
```

注意，第二个偏特化匹配任何使用 C 风格可变参数的函数类型。

严格地说，第二个省略号 (...) 之前的逗号可选，但这里为了清晰起见提供了逗号。由于省略可选，第一个偏特化中的函数类型在语法上不明确：可以解析为 R(Args, ...) (C 风格的可变参数) 或 R(Args... (参数包))。之所以选择第二种解释，是因为 Args 是未扩展的参数包。需要其他解释的情况下，可以显式地添加逗号。

主 DecayT 模板和它的四个偏特化一起实现参数类型衰变，如下例程序所示：

traits/decay.cpp

```
1 #include <iostream>
2 #include <typeinfo>
3 #include <type_traits>
4 #include "decay.hpp"
5
6 template<typename T>
7 void printDecayedType()
8 {
9     using A = typename DecayT<T>::Type;
10    std::cout << "Parameter type: " << typeid(A).name() << '\n' ;
11    std::cout << "- is int: " << std::is_same<A,int>::value << '\n' ;
12    std::cout << "- is const: " << std::is_const<A>::value << '\n' ;
13    std::cout << "- is pointer: " << std::is_pointer<A>::value << '\n' ;
14 }
15
16 int main()
17 {
18     printDecayedType<int>();
19     printDecayedType<int const>();
20     printDecayedType<int[7]>();
21     printDecayedType<int(int)>();
22 }
```

像往常一样，这里提供了一个别名模板：

```
1 template typename T>
2 using Decay = typename DecayT<T>::Type;
```

C++ 标准库还提供了相应的类型特征 `std::decay<>`，在第 D.4 节中描述。

19.3.3 谓词特征

我们已经研究和开发了单一类型的类型函数：给定一个类型，提供其他相关的类型或常量。但是，通常可以开发依赖多个参数的类型函数。这也导致了一种特殊形式的类型特征，类型谓词（产生布尔值的类型函数）。

IsSameT

IsSameT 特征会产生两种类型是否相等：

traits/issame0.hpp

```
1 template<typename T1, typename T2>
2 struct IsSameT {
3     static constexpr bool value = false;
4 };
```

```
5
6 template<typename T>
7 struct IsSameT<T, T> {
8     static constexpr bool value = true;
9 }
```

这里的主模板定义了两个不同的类型，通常作为模板参数传递。使得 value 成员为 false。使用偏特化，当传递的两个类型相同的特殊情况时，value 为 true。

下面的表达式检查传递的模板参数是否为整数：

```
1 if (IsSameT<T, int>::value) ...
```

对于产生常量值的特征，不能提供别名模板，但可以提供 constexpr 变量模板来扮演相同的角色：

```
1 template<typename T1, typename T2>
2 constexpr bool isSame = IsSameT<T1, T2>::value;
```

C++ 标准库提供了相应的类型特征 std::is_same<>，在第 D.3.3 节中详述。

true_type 和 false_type

可以通过为两种结果，提供不同的类型来改进 IsSameT 的定义。若声明一个类模板 BoolConstant，其中包含 TrueType 和 FalseType 两种可能的实例化：

traits/boolconstant.hpp

```
1 template<bool val>
2 struct BoolConstant {
3     using Type = BoolConstant<val>;
4     static constexpr bool value = val;
5 };
6 using TrueType = BoolConstant<true>;
7 using FalseType = BoolConstant<false>;
```

可以定义 IsSameT，根据这两种类型是否匹配，其派生自 TrueType 或 FalseType：

traits/issame.hpp

```
1 #include "boolconstant.hpp"
2
3 template<typename T1, typename T2>
4 struct IsSameT : FalseType
5 {
6 };
7
8 template<typename T>
9 struct IsSameT<T, T> : TrueType
10 {
11 };
```

结果类型为

```
1 IsSameT<T, int>
```

隐式转换为其基类 TrueType 或 FalseType，不仅提供了相应的值成员，而且可以在编译时，分配给不同的函数实现或类模板偏特化。例如：

traits/issame.cpp

```
1 #include "issame.hpp"
2 #include <iostream>
3
4 template<typename T>
5 void fooImpl(T, TrueType)
6 {
7     std::cout << "fooImpl(T,true) for int called\n";
8 }
9
10 template<typename T>
11 void fooImpl(T, FalseType)
12 {
13     std::cout << "fooImpl(T,false) for other type called\n";
14 }
15
16 template<typename T>
17 void foo(T t)
18 {
19     fooImpl(t, IsSameT<T, int>{}); // choose impl. depending on whether T is int
20 }
21
22 int main()
23 {
24     foo(42); // calls fooImpl(42, TrueType)
25     foo(7.7); // calls fooImpl(7.7, FalseType)
26 }
```

这种技术称为标签调度，将在第 20.2 节中介绍。

BoolConstant 的实现包含一个 Type 成员，其允许为 IsSameT 重新引入一个别名模板：

```
1 template<typename T>
2 using IsSame = typename IsSameT<T>::Type;
```

别名模板可以与变量模板 isSame 共存。

通常，产生布尔值的特征应该通过派生自 TrueType 和 FalseType 等类型来支持标记分配。为了泛型，应该只有一种类型表示真，一种类型表示假，而不是让每个泛型库为布尔常量定义自己的类型。

C++ 标准库在 `<type_traits>` 头文件中提供了相应的类型，因为 C++11: `std::true_type` 和 `std::false_type`。在 C++11 和 C++14 中，其定义如下：

```
1 namespace std {
```

```
2   using true_type = integral_constant<bool, true>;
3   using false_type = integral_constant<bool, false>;
4 }
```

C++17 后，其定义为

```
1 namespace std {
2   using true_type = bool_constant<true>;
3   using false_type = bool_constant<false>;
4 }
```

其中 `bool_constant` 在命名空间 `std` 中定义为

```
1 template<bool B>
2 using bool_constant = integral_constant<bool, B>;
```

请参阅第 D.1.1 节了解更多细节。

由于这个原因，本书后面章节将直接使用 `std::true_type` 和 `std::false_type`，特别是在定义类型谓词时。

19.3.4 结果类型特征

处理多种类型函数的另一个例子是结果类型特征，在编写操作符模板时非常有用。为了了解其用法，我们编写一个函数模板，可以添加两个数组容器：

```
1 template<typename T>
2 Array<T> operator+ (Array<T> const&, Array<T> const&);
```

因为语言允许在 `int` 值上添加一个 `char` 值，所以更倾向于混合类型的数组操作。然后，需要确定结果模板的返回类型

```
1 template<typename T1, typename T2>
2 Array<???> operator+ (Array<T1> const&, Array<T2> const&);
```

除了在 1.3 节中介绍的不同方法外，结果类型模板可以在前面的声明中填写问号，如下所示：

```
1 template<typename T1, typename T2>
2 Array<typename PlusResultT<T1, T2>::Type>
3 operator+ (Array<T1> const&, Array<T2> const&);
```

或，若假设一个别名模板

```
1 template<typename T1, typename T2>
2 Array<PlusResult<T1, T2>>
3 operator+ (Array<T1> const&, Array<T2> const&);
```

`PlusResultT` 特征通过加法操作符将两种（可能是不同的）类型的值相加，从而确定产生的类型：

traits/plus1.hpp

```
1 template<typename T1, typename T2>
2 struct PlusResultT {
```

```

3   using Type = decltype(T1() + T2());
4 }
5
6 template<typename T1, typename T2>
7 using PlusResult = typename PlusResultT<T1, T2>::Type;

```

这个特征模板使用 `decltype` 来计算表达式 $T1() + T2()$ 的类型，将确定结果类型的工作（包括处理提升规则和重载操作符）交给编译器。

然而，对于我们的例子来说，`decltype` 实际上保留了太多的信息（参见第 15.10.2 节关于 `decltype` 行为的描述）。`PlusResultT` 可能会产生引用类型，但我们的 `Array` 类模板不是为处理引用类型而设计的。更实际地说，重载加法操作符可能返回 `const` 类型的值：

```

1 class Integer { ... };
2 Integer const operator+ (Integer const&, Integer const&);

```

添加两个 `Array<Integer>` 值将得到一个 `Integer const` 数组，这很可能不是我们想要的。我们想要的是通过删除引用和限定符来转换结果类型，就像上一节那样：

```

1 template<typename T1, typename T2>
2 Array<RemoveCV<RemoveReference<PlusResult<T1, T2>>>>
3 operator+ (Array<T1> const&, Array<T2> const&);

```

这种特征嵌套在模板库中很常见，经常用于元编程上下文中。元编程将在第 23 章详细介绍。（别名模板对于像这样的多层次嵌套特别有用；否则，需要在每一层添加 `typename` 和 `::Type` 后缀。）

当将两个元素类型（可能是不同的）数组相加时，数组加法操作符将正确地计算结果类型。但 `PlusResultT` 对元素类型 `T1` 和 `T2` 添加了一个限制，因为表达式 $T1() + T2()$ 尝试对 `T1` 和 `T2` 类型的值进行初始化，这两种类型必须具有可访问的、非删除的默认构造函数（或者是非类类型）。`Array` 类本身可能不需要元素类型的值初始化，因此这是一个不必要的限制。

declval

通过使用生成给定类型 `T` 值的函数，为加法表达式生成值很容易，不需要构造函数。为此，C++ 标准提供 `std::declval`，在第 11.2.3 节中介绍。它在标准头文件 `<utility>` 中定义：

```

1 namespace std {
2     template<typename T>
3     add_rvalue_reference_t<T> declval() noexcept;
4 }

```

表达式 `declval<T>()` 生成 `T` 类型的值，不需要默认构造函数（或其他操作）。

这个函数模板故意没有定义，因其只用于 `decltype`、`sizeof` 或其他不需要定义的上下文中的操作。它还有另外两个有趣的属性：

- 对于可引用的类型，返回类型是该类型的右值引用，这使得 `declval` 可以用于不能从函数返回的类型，如抽象类类型（具有纯虚函数的类）或数组类型。否则，从 `T` 到 `T&&` 的转换对作为表达式使用的 `declval<T>()` 的行为没有影响：两者都是右值（若 `T` 是一个对象类型），而左值引用类型由于引用折叠规则而不变（第 15.6 节中描述）。

通过直接使用 `decltype` 可以发现返回类型 `T` 和 `T&&` 之间的区别。因为 `declval` 的使用限制，所以其没什么实际意义。

- `noexcept` 异常说明说明 `declval` 不会导致表达式认为抛出异常。当在 `noexcept` 操作符的上下文中使用 `declval` 时，就可以使用了（第 19.7.2 节）。

使用 `declval`，可以消除 `PlusResultT` 的值初始化要求：

traits/plus2.hpp

```
1 #include <utility>
2
3 template<typename T1, typename T2>
4 struct PlusResultT {
5     using Type = decltype(std::declval<T1>() + std::declval<T2>());
6 };
7
8 template<typename T1, typename T2>
9 using PlusResult = typename PlusResultT<T1, T2>::Type;
```

结果类型特征提供了一种确定特定操作的精确返回类型的方法，在描述函数模板的结果类型时很有用。

19.4. 基于 SFINAE 的特征

SFINAE 原则（替换失败不为过；参见第 8.4 节和第 15.7 节）将模板参数推导过程中无效类型和表达式形成过程中的错误（这会导致程序格式不正确）转化为推导失败，从而允许重载解析选择不同的候选项。虽然 SFINAE 最初的目的是避免函数模板重载的假失败，但它还支持显著的编译时技术，可以确定特定类型或表达式是否有效。这可以通过编写特征，来确定类型是否具有特定成员、是否支持特定操作或是否为类。

实现基于 SFINAE 特性的两种主要方法：SFINAE 排除函数重载和 SFINAE 排除偏特化。

19.4.1 去除函数重载

对基于 SFINAE 的特征的第一次尝试说明，使用带有函数重载的 SFINAE 来确定类型是否默认可构造的基本技术，这样就可以创建不需要初始化值的对象。对于给定的类型 `T`，像 `T()` 这样的表达式必须有效。

实现如下所示：

traits/isdefaultconstructible1.hpp

```
1 #include "issame.hpp"
2
3 template<typename T>
4 struct IsDefaultConstructibleT {
5     private:
```

```

6 // test() trying substitute call of a default constructor for T passed as U:
7 template<typename U, typename = decltype(U())>
8     static char test(void*);
9 // test() fallback:
10 template<typename>
11     static long test(...);
12 public:
13     static constexpr bool value
14         = IsSameT<decltype(test<T>(nullptr)), char>::value;
15 };

```

使用函数重载实现基于 SFINAE 的特征的通常方法是声明两个名为 test() 的重载函数模板，具有不同的返回类型：

```

1 template<...> static char test(void*);
2 template<...> static long test(...);

```

第一个重载设计为只在请求的检查成功时匹配（将在下面讨论如何实现这一点）。第二个重载是回退：

回退声明有时可以是普通成员函数声明，而不是成员函数模板。

它总是匹配调用，但是因为它匹配“带有省略号”（即可变参数），任何其他匹配都将首选（参见 C.2 节）。

“返回值”的值取决于选择了哪个 test() 成员的重载：

```

1 static constexpr bool value
2     = IsSameT<decltype(test<T>(nullptr)), char>::value;

```

如果选择了返回类型为 char 的第一个 test() 成员，value 将初始化为 IsSame<char,char>，为 true。否则，将初始化为 IsSame<long,char>，该值为 false。

现在，需要处理想要测试的特定属性。目标是使第一个 test() 重载，在检查条件适用时有效。本例中，想知道是否可以默认构造一个传递类型 T 的对象。为了实现这一点，将 T 传递为 U，并为 test() 的第一个声明提供第二个未命名（虚拟）模板参数，该实参初始化时且转换有效时，参数有效。使用的表达式只有在隐式或显式默认构造函数存在时，U() 才有效。表达式使用 decltype，使其成为初始化类型参数的有效表达式。不能推导第二个模板参数，因为没有传递相应的参数。不会提供显式的模板参数。因此会进行替换，如果替换失败，根据 SFINAE，test() 的声明将丢弃，以便匹配回退声明。

因此，可以这样使用该特征：

```

1 IsDefaultConstructibleT<int>::value // yields true
2
3 struct S {
4     S() = delete;
5 };
6 IsDefaultConstructibleT<S>::value // yields false

```

注意，不能在第一个 test() 中直接使用模板参数 T：

```

1 template<typename T>
2 struct IsDefaultConstructibleT {
3     private:
4         // ERROR: test() uses T directly:
5         template<typename, typename = decltype(T())>
6             static char test(void*);
7         // test() fallback:
8         template<typename>
9             static long test(...);
10    public:
11        static constexpr bool value
12            = IsSameT<decltype(test<T>(nullptr)), char>::value;
13    };

```

这样不行，对于任何 T 替换所有成员函数，因此对于不是默认可构造的类型，代码无法编译，而不忽略第一个 test() 重载。通过将类模板参数 T 传递给函数模板参数 U，仅为第二次 test() 重载创建了特定的 SFINAE。

基于 SFINAE 特征的替代实现策略

1998 年发布第一个 C++ 标准前，基于 SFINAE 的特性就已经可以实现了。

SFINAE 规则当时有更多的限制：当替换模板参数导致了一个不规范的类型构造（例如，T::X，其中 T 是 int），SFINAE 按预期工作，导致了一个无效的表达式（例如，`sizeof(f())`，其中 f() 返回 void），SFINAE 不起作用，并立即发出错误信息。

这种方法的关键在于声明两个重载的函数模板，返回不同的返回类型：

```

1 template<...> static char test(void*);
2 template<...> static long test(...);

```

然而，最初发布的技术使用返回类型的大小来确定选择哪个重载（因为无法使用 nullptr 和 constexpr，所以使用 0 和 enum）：

本书的第一版可能是这种技术的首次亮相。

```

1 enum { value = sizeof(test<...>(0)) == 1 };

```

某些平台上，`sizeof(char)==sizeof(long)` 可能为 true。例如，在数字信号处理器 (DSP) 或旧的 Cray 机器上，所有的整型基本类型可以具有相同的大小。根据定义，`sizeof(char)` 等于 1，在这些机器上，`sizeof(long)` 甚至 `sizeof(long long)` 也等于 1。

鉴于上述观察结果，希望确保 test() 函数的返回类型在所有平台上具有不同的大小。例如，定义之后

```

1 using Size1T = char;
2 using Size2T = struct { char a[2]; };

```

或

```
1 using Size1T = char(&)[1];
2 using Size2T = char(&)[2];
```

可以这样定义 test() 的重载:

```
1 template<...> static Size1T test(void*); // checking test()
2 template<...> static Size2T test(...); // fallback
```

这里，要么返回 Size1T，一个大小为 1 的单个字符，要么返回 (一个结构体) 一个包含两个字符的数组，该数组在所有平台上的大小至少为 2。

使用其中一种方法的代码仍然很常见。

注意，传递给 func() 的调用参数的类型并不重要。重要的是传递的参数与期望的类型匹配。例如，可以定义传递整数 42:

```
1 template<...> static Size1T test(int); // checking test()
2 template<...> static Size2T test(...); // fallback
3 ...
4 enum { value = sizeof(test<...>(42)) == 1 };
```

使基于 SFINAE 的特征成为特征

如第 19.3.3 节所述，返回布尔值的谓词特征应该返回一个派生自 std::true_type 或 std::false_type 的值。这样，也可以解决在某些平台上 sizeof(char) 与 sizeof(long) 相同的问题。

为此，需要 IsDefaultConstructibleT 的间接定义。特征本身应该从助手类的 Type 派生，助手类生成必要的基类。这里，可以简单地提供相应的基类作为 test() 重载的返回类型:

```
1 template<...> static std::true_type test(void*); // checking test()
2 template<...> static std::false_type test(...); // fallback
```

基类的 Type 成员可以声明为:

```
1 using Type = decltype(test<FROM>(nullptr));
```

并且，不再需要 IsSameT 特性。

因此，IsDefaultConstructibleT 的完整改进实现如下所示:

traits/isdefaultconstructible2.hpp

```
1 #include <type_traits>
2
3 template<typename T>
4 struct IsDefaultConstructibleHelper {
5     private:
6         // test() trying substitute call of a default constructor for T passed as U:
7         template<typename U, typename = decltype(U())>
8             static std::true_type test(void*);
9         // test() fallback:
10        template<typename>
```

```

11     static std::false_type test(...);
12
13     public:
14     using Type = decltype(test<T>(nullptr));
15
16 template<typename T>
17 struct IsDefaultConstructibleT : IsDefaultConstructibleHelper<T>::Type {
18 };

```

若第一个 test() 函数模板有效，是首选重载，因此成员 IsDefaultConstructibleHelper::Type 由它的返回类型 std::true_type 初始化。因此，IsDefaultConstructibleT<…> 派生于 std::true_type。

若第一个 test() 函数模板无效，会因为 SFINAE 而禁用，IsDefaultConstructibleHelper::Type 由 test() 回退的返回类型初始化，也就是 std::false_type。结果，IsDefaultConstructibleT<…> 派生自 std::false_type。

19.4.2 去除偏特化

实现基于 SFINAE 的特征的第二种方法使用偏特化，可以使用这个例子来找出 T 类型是否默认可构造：

traits/isdefaultconstructible3.hpp

```

1 #include "issame.hpp"
2 #include <type_traits> // defines true_type and false_type
3
4 // helper to ignore any number of template parameters:
5 template<typename...> using VoidT = void;
6
7 // primary template:
8 template<typename, typename = VoidT>>
9 struct IsDefaultConstructibleT : std::false_type
10 {
11 };
12
13 // partial specialization (may be SFINAE' d away):
14 template<typename T>
15 struct IsDefaultConstructibleT<T, VoidT<decltype(T())>> : std::true_type
16 {
17 };

```

与上面针对谓词特征的 IsDefaultConstructibleT 改进版本一样，定义了从 std::false_type 派生的一般情况，默认情况下，假定类型没有默认构造函数。

这里有趣的特性是第二个模板参数，默认为助手类的 VoidT 类型，可以使用编译时类型构造的偏特化。

本例中，只需要一个构造：

```
1 decltype(T())
```

再次检查 T 的默认构造函数是否有效。若对于特定的 T 构造无效，SFINAE 会丢弃整个偏特化，返回主模板。否则，偏特化有效并且首选。

C++17 中，C++ 标准库引入了一个类型特征 std::void_t<>，对应于这里引入的类型 VoidT。C++17 前，可以如上面那样自行定义，甚至可以在命名空间 std 中这样定义：

命名空间 std 中定义 void_t 在形式上是无效的：不允许用户在命名空间 std 添加声明。实践中，当前的编译器没有强制这样的限制，其行为也不会意外（标准指出这样做会导致“未定义行为”，可能出现任何情况）。

```
1 #include <type_traits>
2
3 #ifndef __cpp_lib_void_t
4 namespace std {
5     template<typename...> using void_t = void;
6 }
7 #endif
```

C++14 开始，C++ 标准化委员会建议编译器和标准库通过定义一致同意的特性宏，来表明它们实现了标准的哪些部分。这不是标准一致性的要求，但是实现者通常会遵循这些建议来帮助其用户进行分辨。

撰写本文时，发现 Microsoft Visual C++ 是个不幸的例外。

宏 __cpp_lib_void_t 是用来指示标准库实现 std::void_t 的宏，因此上面的代码以它为条件。

显然，这种定义类型特征的方法看起来比重载函数模板的第一种方法更简洁，但需要在模板参数声明中明确条件的能力。使用带有函数重载的类模板，使我们能够使用辅助函数或类型。

19.4.3 使用 Lambda

无论使用哪种技术，总是需要一些样板代码来定义特征：重载和调用两个 test() 成员函数或实现多个偏特化。接下来，将展示在 C++17 中如何通过在泛型 Lambda 中指定要检查的条件，来最小化这个样板代码。

首先，引入一个由两个嵌套的泛型 Lambda 表达式构造的工具：

traits/isvalid.hpp

```
1 #include <utility>
2
3 // helper: checking validity of f(args...) for F f and Args... args:
4 template<typename F, typename... Args,
5          typename = decltype(std::declval<F>()(std::declval<Args...>()))>
6 std::true_type isValidImpl(void* );
7
8 // fallback if helper SFINAE'd out:
9 template<typename F, typename... Args>
```

```

10 std::false_type isValidImpl(...);
11
12 // define a lambda that takes a lambda f and returns whether calling f with args is valid
13 inline constexpr
14 auto isValid = [] (auto f) {
15     return [] (auto&&... args) {
16         return decltype(isValidImpl<decltype(f)>,
17             decltype(args)&&...
18             >(nullptr)) {};
19    };
20};
21
22 // helper template to represent a type as a value
23 template<typename T>
24 struct TypeT {
25     using Type = T;
26};
27
28 // helper to wrap a type as a value
29 template<typename T>
30 constexpr auto type = TypeT<T>{};
31
32 // helper to unwrap a wrapped type in unevaluated contexts
33 template<typename T>
34 T valueT(TypeT<T>); // no definition needed

```

从isValid的定义开始:一个constexpr变量,其类型是Lambda闭包类型。声明必须使用占位符类型(代码中是auto),因为C++没有办法直接表示闭包类型。C++17前,Lambda表达式不能出现在常量表达式中,这就是为什么这段代码只在C++17中有效。因为isValid有一个闭包类型,所以可以调用。但是返回的本身是Lambda闭包类型的对象,其由内部的Lambda表达式产生。

深入研究内部Lambda表达式的细节之前,先了解一下isValid的用法:

```

1 constexpr auto isDefaultConstructible
2     = isValid([] (auto x) -> decltype((void) decltype(valueT(x)) ()) {
3     });

```

已经知道isDefaultConstructible有一个Lambda闭包类型,是一个检查类型是否为默认构造的函数对象(将在后面的内容中看到原因)。换句话说,isValid是一个特征工厂:根据参数生成特征检查对象的组件。

类型辅助变量模板允许将类型表示为值。通过这种方式获得的值x可以通过decltype(valueT(x))返回原始类型,

这对简单的辅助模板是一种基本技术,其位于Boost.Hana中!

这正是在上面传递给isValid的Lambda函数中所做的。若提取的类型不能默认构造,那么decltype(valueT(x))()就无效,要么会得到一个编译错误,再要么相关的声明会进行SFINAE(由于isValid定义的细节,我们将实现后一种效果)。

```
1 isDefaultConstructible can be used as follows:  
2 isDefaultConstructible(type<int>) // true (int is default-constructible)  
3 isDefaultConstructible(type<int&>) // false (references are not default-constructible)
```

要查看所有部分是如何工作的，考虑 isValid 中的内部 Lambda 表达式的变化，isValid 的参数 f 绑定到 isDefaultConstructible 定义中指定的泛型 Lambda 参数。通过在 isValid 的定义中执行替换，从而得到了等价代码：

这段代码在 C++ 中是无效的，因为出于编译技术的原因，Lambda 表达式不能直接出现在 decltype 操作数中，但其含义明确。

```
1 constexpr auto isDefaultConstructible  
2 = [] (auto&&... args) {  
3     return decltype(  
4         isValidImpl<  
5             decltype([] (auto x)  
6                 -> decltype((void) decltype(valueT(x))()),  
7                 decltype(args)&&...  
8                 >(nullptr));  
9     };
```

回头看看上面 isvaliddmpl() 的第一个声明，就会注意到包含了默认模板参数列表

```
1 decltype(std::declval<F>() (std::declval<Args&&>() ...))>
```

试图调用第一个模板参数类型的值，该参数是 isDefaultConstructible 定义中 Lambda 的闭包类型，并将参数类型的值 (decltype(args)&&...) 传递给 isDefaultConstructible。由于 Lambda 中只有一个参数 x，args 的扩展只能有一个参数；上面的 static_assert 示例中，该参数的类型为 TypeT<int> 或 TypeT<int&>。在 TypeT<int&> 情况下，decltype(valueT(x)) 是 int& 这使得 decltype(valueT(x))() 无效，因此在 isValidImpl() 的第一个声明中对默认模板参数的替换失败，并通过 SFINAE 去除。所以只剩下第二个声明（否则将是一个较小的匹配），其产生一个 false_type 值。也就是说，isDefaultConstructible 在传递 type<int&> 时产生 false_type。若传递了 type<int>，替换不会失败，并且选择 isValidImpl() 的第一个声明，产生一个 true_type 类型的值。要使 SFINAE 工作，替换必须在要替换模板的上下文中产生。要替换的模板是 sValidImpl 的第一个声明和传递给 isValid 的泛型 Lambda 的调用操作符。因此，要测试的构造必须出现在 Lambda 的返回类型中，而不是其主体中！

isDefaultConstructible 特征与以前的特征实现有一点不同，其需要函数式调用，而不是指定模板参数。可以说是一种可读性更强的表示法，使用之前的方式可以得到：

```
1 template<typename T>  
2 using IsDefaultConstructibleT  
3 = decltype(isDefaultConstructible(std::declval<T>()));
```

由于这是一个传统的模板声明，只能出现在命名空间作用域中，而 isDefaultConstructible 的定义可以在块作用域中引入。

这种技术似乎并不引人注目，因为实现中涉及的表达式和使用风格都比以前的技术更复杂。然而，当 `isValid` 就位并理解，许多特征就可以通过一个声明实现。例如，测试是否访问名为 `first` 的成员，可以相当干净（见第 19.6.4 节的完整示例）：

```
1 constexpr auto hasFirst
2   = isValid([](auto x) -> decltype((void)valueT(x).first) {
3     });
```

19.4.4 SFINAE 友好的特征

类型特征能够回应特定的查询，而不会导致程序病态。基于 SFINAE 的特征通过 SFINAE 捕捉来解决这个问题，将这些错误会转化为负面结果。

然而，目前出现的一些特征（如第 19.3.4 节中描述的 `PlusResultT` 特征）在出现错误时表现不佳。回想一下该节中对 `PlusResultT` 的定义：

traits/plus2.hpp

```
1 #include <utility>
2
3 template<typename T1, typename T2>
4 struct PlusResultT {
5   using Type = decltype(std::declval<T1>() + std::declval<T2>());
6 };
7
8 template<typename T1, typename T2>
9 using PlusResult = typename PlusResultT<T1, T2>::Type;
```

这个定义中，加法用于 SFINAE 不保护的上下文中。因此，若程序试图对没有合适加法操作符的类型计算 `PlusResultT`，对 `PlusResultT` 本身的计算将导致程序格式不正确，如下尝试声明不相关类型 A 和 B 的添加数组的返回类型：

简单起见，返回值只使用 `PlusResultT<T1,T2>::Type`。实践中，还应该使用 `RemoveReferenceT<>` 和 `RemoveCVT<>` 来计算返回类型，以避免返回引用。

```
1 template<typename T>
2 class Array {
3   ...
4 };
5
6 // declare + for arrays of different element types:
7 template<typename T1, typename T2>
8 Array<typename PlusResultT<T1, T2>::Type>
9 operator+ (Array<T1> const&, Array<T2> const&);
```

若没有为数组元素定义相应的加法操作符，这里使用 `PlusResultT<>` 将出现错误。

```
1 class A {
```

```

2 } ;
3 class B {
4 } ;
5 void addAB(Array<A> arrayA, Array<B> arrayB) {
6     auto sum = arrayA + arrayB; // ERROR: fails in instantiation of PlusResultT<A, B>
7     ...
8 }

```

实际的问题不在于这种失败发生在明显格式不正确的代码中(没有办法将 A 数组添加到 B 数组中), 而是发生在对加法操作无的模板参数推导期间, PlusResultT<A,B> 的实例化。

这产生了一个后果: 即使添加 A 和 B 数组时, 也添加了特定的重载, 程序也可能无法编译。因为 C++ 没有指定若另一个重载会更好, 函数模板中的类型是否可以实例化:

```

1 // declare generic + for arrays of different element types:
2 template<typename T1, typename T2>
3 Array<typename PlusResultT<T1, T2>::Type>
4 operator+ (Array<T1> const&, Array<T2> const&);
5
6 // overload + for concrete types:
7 Array<A> operator+ (Array<A> const& arrayA, Array<B> const& arrayB);
8
9 void addAB (Array<A> const& arrayA, Array<B> const& arrayB) {
10     auto sum = arrayA + arrayB; // ERROR?: depends on whether the compiler
11     ... // instantiates PlusResultT<A,B>
12 }

```

若编译器不需要对加法操作符的第一个(模板)声明执行推导和替换, 就能确定加法操作符的第二个声明是更好的匹配。

然而, 在推导和替换函数模板候选时, 类模板定义实例化期间发生的事情都不是函数模板替换的一部分, SFINAE 不能避免的形成无效类型或表达式。不是仅丢弃函数模板候选, 而是立即发出一个错误, 因为我们试图对 PlusResultT<> 中的 A 和 B 类型的两个元素使用加法操作符:

```

1 template<typename T1, typename T2>
2 struct PlusResultT {
3     using Type = decltype(std::declval<T1>() + std::declval<T2>());
4 };

```

为了解决这个问题, 必须使 PlusResultT 对 SFINAE 友好, 通过一个合适的定义使它更有弹性, 即使 decltype 表达式是病态的。

按照上一节描述的 PlusResultT 的例子, 定义了一个 HasPlusT 特性, 其可以检测给定类型是否有合适的加法操作:

traits/hasplus.hpp

```

1 #include <utility> // for declval
2 #include <type_traits> // for true_type, false_type, and void_t
3
4 // primary template:

```

```

5 template<typename, typename, typename = std::void_t<>>
6 struct HasPlusT : std::false_type
7 {
8 };
9
10 // partial specialization (may be SFINAE'ed away):
11 template<typename T1, typename T2>
12 struct HasPlusT<T1, T2, std::void_t<decltype(std::declval<T1>()
13 + std::declval<T2>())>>
14 : std::true_type
15 {
16 };

```

若产生结果, PlusResultT 可以使用现有的实现。否则, PlusResultT 需要一个安全的默认值。对于一组模板参数没有任何意义的结果特征, 最好的默认值是不提供成员 Type。这样, 若特征在 SFINAE 中使用——上面数组加法操作符模板的返回类型——缺失的成员 Type 将导致模板参数推导失败, 这正是数组加法操作符模板所期望的行为。

下面的 PlusResultT 实现了这种行为:

traits/plus3.hpp

```

1 #include "hasplus.hpp"
2
3 template<typename T1, typename T2, bool = HasPlusT<T1, T2>::value>
4 struct PlusResultT { // primary template, used when HasPlusT yields true
5     using Type = decltype(std::declval<T1>() + std::declval<T2>());
6 };
7
8 template<typename T1, typename T2>
9 struct PlusResultT<T1, T2, false> { // partial specialization, used otherwise
10 };

```

这个版本的 PlusResultT 中, 添加了一个模板参数, 该参数带有一个默认参数, 用于确定前两个参数, 是否像上面的 HasPlusT 特征所确定的那样支持加法。然后, 偏特化 PlusResultT 获取参数的 `false`, 并且偏特化定义没有成员, 从而避免了所提到的问题。对于支持加法的情况, 默认参数的计算结果为 `true`, 并使用现有的 Type 成员定义选择主模板。因此, 我们践行了约定, 即 PlusResultT 只在加法操作定义良好的情况下, 才提供结果类型。(注意, 添加的参数永远不应该有显式的模板参数。)

再次考虑添加 `Array<A>` 和 `Array` 的情况。在最新的 PlusResultT 模板实现中, PlusResultT<A,B> 的实例化不会有 Type 成员, 因为 A 和 B 的值不可求和。因此, 数组加法操作符模板的结果类型无效, SFINAE 将从考虑中剔除函数模板。因此, 将选择特定于 `Array<A>` 和 `Array` 的重载加法操作符。

作为一个通用的设计原则, 若给出合理的模板参数作为输入, 特征模板应该不会在实例化时失败。而一般的方法通常会进行两次检查:

- 减法操作是否有效

- 检查计算结果

已经在 PlusResultT 中看到了，通过调用 HasPlusT $\langle\rangle$ 来确定在 PlusResultImpl $\langle\rangle$ 中加法操作符的调用是否有效。

将这一原则应用到 19.3.1 节介绍的 ElementT：从容器类型生成元素类型。同样，因为答案依赖于(容器)类型，其有一个成员类型 value_type，所以只有当容器类型有这样一个 value_type 成员时，主模板才尝试定义成员类型：

```

1 template<typename C, bool = HasMemberT<value_type<C>::value>
2 struct ElementT {
3     using Type = typename C::value_type;
4 };
5
6 template<typename C>
7 struct ElementT<C, false> {
8 };

```

使特征对 SFINAE 友好的第三个例子在第 19.7.2 节，其中 IsNothrowMoveConstructibleT 首先检查移动构造函数是否存在，然后检查其是否使用 noexcept 声明。

19.5. IsConvertibleT

由于细节原因，基于 SFINAE 特征的一般方法在实践中可能会变得复杂。可以通过定义特征来说明这一点，该特征可以确定类型是否可以转换——若期望是某个基类或其派生类。IsConvertibleT 特征决定是否可以将传递的第一个类型转换为传递的第二个类型：

traits/isconvertible.hpp

```

1 #include <type_traits> // for true_type and false_type
2 #include <utility> // for declval
3
4 template<typename FROM, typename TO>
5 struct IsConvertibleHelper {
6     private:
7         // test() trying to call the helper aux(TO) for a FROM passed as F:
8         static void aux(TO);
9         template<typename F, typename,
10                 typename = decltype(aux(std::declval<F>()))>
11         static std::true_type test(void*);
12         // test() fallback:
13         template<typename, typename>
14         static std::false_type test(...);
15     public:
16         using Type = decltype(test<FROM>(nullptr));
17     };
18
19 template<typename FROM, typename TO>
20 struct IsConvertibleT : IsConvertibleHelper<FROM, TO>::Type {

```

```

21 } ;
22
23 template<typename FROM, typename TO>
24 using IsConvertible = typename IsConvertibleT<FROM, TO>::Type;
25
26 template<typename FROM, typename TO>
27 constexpr bool isConvertible = IsConvertibleT<FROM, TO>::value;

```

使用函数重载的方法，如第 19.4.1 节。在辅助类内部，声明了两个名为 test() 的重载函数模板，其具有不同的返回类型，并为结果特征的基类声明了一个 Type 成员：

```

1 template<...> static std::true_type test(void*);
2 template<...> static std::false_type test(...);
3 ...
4 using Type = decltype(test<FROM>(nullptr));
5 ...
6 template<typename FROM, typename TO>
7 struct IsConvertibleT : IsConvertibleHelper<FROM, TO>::Type {
8 };

```

与往常一样，第一个 test() 重载设计在检查成功时匹配，而第二个重载会回退。因此，目标是使第一个 test() 重载在当类型 FROM 转换为类型 TO 时有效。为了实现这一点，再次给出 test 的第一个声明，一个虚拟的(未命名的)模板参数初始化使用了构造，当转换有效时该构造才有效。这个模板参数不能推导，也不会提供显式的模板参数。因此，参数将进行替换，若替换失败，将丢弃 test() 的声明。

下面的操作无效：

```

1 static void aux(TO);
2 template<typename = decltype(aux(std::declval<FROM>()))>
3   static char test(void*);

```

当解析成员函数模时，FROM 和 TO 是确定的，因此一对转换无效的类型(例如，double* 和 int*)将在调用 test() 前立即触发错误(因此在 SFINAE 之外)。

因此，引入 F 作为一个特定的成员函数模板参数

```

1 static void aux(TO);
2 template<typename F, typename = decltype(aux(std::declval<F>()))>
3   static char test(void*);

```

并在调用 test() 时显式地提供 FROM 类型作为模板参数，该参数出现在 value 的初始化中：

```

1 static constexpr bool value
2   = isSame<decltype(test<FROM>(nullptr)), char>;

```

不使用构造函数，如何使用 std::declval 来生成一个值？若该值可转换为 TO，则对 aux() 的调用有效，并且 test() 的声明匹配。否则，SFINAE 将失败，并且只匹配回退声明。

因此，可以这样使用该特征：

```

1 IsConvertibleT<int, int>::value // yields true
2 IsConvertibleT<int, std::string>::value // yields false

```

```
3 IsConvertibleT<char const*, std::string>::value // yields true
4 IsConvertibleT<std::string, char const*>::value // yields false
```

处理特殊情况

IsConvertibleT 还未正确处理的三种情况：

1. 向数组类型的转换应该生成 false，但 aux() 声明中的 TO 类型参数会衰变为指针类型，因此对某些 FROM 类型为 true。
2. 函数类型的转换应该生成 false，但就像数组一样，实现只将它们视为衰变类型。
3. 转换为 (const/volatile 限定的)void 类型应该生成 true。因为参数类型不能是 void 类型 (而且 aux() 是用这样参数声明的)，所以上面的实现没有成功实例化 TO 是 void 类型的情况。

对于所有这些情况，需要额外的偏特化。为每一种可能的 const 和 volatile 限定符组合添加这样的特化其实很难。不过，可以向辅助类模板添加额外的模板参数，如下所示：

```
1 template<typename FROM, typename TO, bool = IsVoidT<TO>::value
2     || IsArrayT<TO>::value
3     || IsFunctionT<TO>::value>
4 struct IsConvertibleHelper {
5     using Type = std::integral_constant<bool,
6         IsVoidT<TO>::value
7         && IsVoidT<FROM>::value>;
8 };
9
10 template<typename FROM, typename TO>
11 struct IsConvertibleHelper<FROM, TO, false> {
12     ... // previous implementation of IsConvertibleHelper here
13 };
```

该模板参数，对所有这些特殊情况使用主辅助特征的实现。若转换为数组或函数 (因为 IsVoidT<TO> 是假的)，或者 FROM 是 void 而 TO 不是，则生成 false_type；对于两个 void 类型，将生成 true_type。其他情况都为第三个参数产生 false 参数，因此采用偏特化，这与已经讨论过的实现相对应。

请参阅 19.8.2 节和 19.8.3 节来讨论如何实现 IsArrayT。

C++ 标准库提供了相应的类型特征 `std::is_convertible<>`，在第 D.3.3 节中有描述。

19.6. 成员检查

基于 SFINAE 的特征的另一个尝试涉及创建特征 (或者更确切地说，一组特征)，可以确定给定的类型 T 是否具有给定名称 X 的成员 (类型成员还是非类型成员)。

19.6.1 检查成员类型

首先定义一个特征，可以确定给定类型 T 是否有成员类型 `size_type`：

traits/hassizetype.hpp

```

1 #include <type_traits> // defines true_type and false_type
2
3 // helper to ignore any number of template parameters:
4 template<typename...> using VoidT = void;
5
6 // primary template:
7 template<typename, typename = VoidT<>>
8 struct HasSizeTypeT : std::false_type
9 {
10 };
11
12 // partial specialization (may be SFINAE' d away):
13 template<typename T>
14 struct HasSizeTypeT<T, VoidT<typename T::size_type>> : std::true_type
15 {
16 };

```

使用 SFINAE 的方法剔除了偏特化。

与通常的谓词特征一样，定义从 `std::false_type` 派生出一般情况，因为默认情况下类型没有成员 `size_type`。

本例中，只需要一个构造：

```

1 typename T::size_type

```

当类型 `T` 具有成员类型 `size_type` 时，该构造有效，这是我们试图确定的。若对于特定的 `T`，构造无效（类型 `T` 没有成员类型 `size_type`），SFINAE 会丢弃偏特化，会返回到主模板。否则，偏特化有效，并且是首选。

可以这样使用这个特征：

```

1 std::cout << HasSizeTypeT<int>::value; // false
2 struct CX {
3     using size_type = std::size_t;
4 };
5 std::cout << HasSizeTypeT<CX>::value; // true

```

若成员类型 `size_type` 私有，`HasSizeTypeT` 会产生 `false`，因为特征模板没有对参数类型的访问权限，因此 `typename T::size_type` 无效（触发 SFINAE）。换句话说，特征会测试是否有可访问的成员类型 `size_type`。

处理引用类型

作为开发者，我们会考虑“临界”情况可能出现的问题。对于像 `HasSizeTypeT` 这样的特征模板，引用类型可能会产生一些问题。例如，下面的代码工作很好：

```

1 struct CXR {
2     using size_type = char&; // Note: type size_type is a reference type
3 };
4 std::cout << HasSizeTypeT<CXR>::value; // OK: prints true

```

以下代码则会失败:

```
1 std::cout << HasSizeTypeT<CX&>::value; // OOPS: prints false
2 std::cout << HasSizeTypeT<CXR&>::value; // OOPS: prints false
```

这就很神奇。引用类型本身没有成员，但每当使用引用时，得到的表达式都有底层类型，因此考虑底层类型可能会更好。可以通过在 HasSizeTypeT 的偏特化中使用之前的 RemoveReference 特征来实现:

```
1 template<typename T>
2 struct HasSizeTypeT<T, VoidT<RemoveReference<T>::size_type>>
3 : std::true_type {
4 };
```

注入类名

用来检测成员类型的特征也会为注入类名生成一个真实值(参见第 13.2.3 节)。例如:

```
1 struct size_type {
2 };
3
4 struct Sizeable : size_type {
5 };
6
7 static_assert(HasSizeTypeT<Sizeable>::value,
8 "Compiler bug: Injected class name missing");
```

因为 size_type 引入自己的名称作为成员类型，并且继承该名称，所以后一个静态断言成功。若没有成功，这就是编译器的一个缺陷。

19.6.2 检查成员类型

定义像 HasSizeTypeT 这样的特征会引发一个问题，即如何将特征参数化，以便能够检查成员类型名。

因为没有语言机制来描述“这种”名称，所以目前只能通过宏来实现。

撰写本文时，C++ 标准化委员会正在探索以程序可以探索的方式“反映”各种程序实体(如类类型及其成员)的方法。参见第 17.9 节。

在不使用宏的情况下，可以使用泛型 Lambda，如第 19.6.4 节所示。

宏的例子:

traits/hastype.hpp

```
1 #include <type_traits> // for true_type, false_type, and void_t
2 #define DEFINE_HAS_TYPE(MemType) \
3 template<typename, typename = std::void_t<>> \
4 struct HasTypeT_##MemType \
```

```

5 : std::false_type { } ; \
6 template<typename T> \
7 struct HasTypeT_##MemType<T, std::void_t<typename T::MemType>> \
8 : std::true_type { } // ; intentionally skipped
9

```

每次使用 `DEFINE_HAS_TYPE(MemberType)` 都会定义一个新的 `HasTypeT_MemberType` 特征。可以使用它来检测类型是否有 `value_type` 或 `char_type` 成员类型:

traits/hastype.cpp

```

1 #include "hastype.hpp"
2 #include <iostream>
3 #include <vector>
4
5 DEFINE_HAS_TYPE(value_type);
6 DEFINE_HAS_TYPE(char_type);
7
8 int main()
9 {
10     std::cout << "int::value_type: "
11     << HasTypeT_value_type<int>::value << '\n' ;
12     std::cout << "std::vector<int>::value_type: "
13     << HasTypeT_value_type<std::vector<int>>::value << '\n' ;
14     std::cout << "std::iostream::value_type: "
15     << HasTypeT_value_type<std::iostream>::value << '\n' ;
16     std::cout << "std::iostream::char_type: "
17     << HasTypeT_char_type<std::iostream>::value << '\n' ;
18 }

```

19.6.3 检查非类型成员

可以修改特征来检查数据成员和(单个)成员函数:

traits/hasmember.hpp

```

1 #include <type_traits> // for true_type, false_type, and void_t
2
3 #define DEFINE_HAS_MEMBER(Member) \
4 template<typename, typename = std::void_t<>> \
5 struct HasMemberT_##Member \
6 : std::false_type { } ; \
7 template<typename T> \
8 struct HasMemberT_##Member<T, std::void_t<decltype(&T::Member)>> \
9 : std::true_type { } // ; intentionally skipped

```

当 `&T::Member` 无效时, 使用 SFINAE 禁用偏特化。要使该结构有效, 以下条件必须为 `true`:

- 成员必须明确地标识 T 的成员 (例如, 不能是重载的成员函数名, 或同名的多个继承成员的名称),
- 成员必须可访问,
- 成员必须是非类型、非枚举成员 (否则前缀 & 将无效),
- 若 T::Member 是静态数据成员, 其类型不能提供使 &T::Member 无效的操作符 &(例如, 使该操作符不可访问)。

可以这样使用模板:

traits/hasmember.cpp

```

1 #include "hasmember.hpp"
2
3 #include <iostream>
4 #include <vector>
5 #include <utility>
6
7 DEFINE_HAS_MEMBER(size);
8 DEFINE_HAS_MEMBER(first);
9
10 int main()
11 {
12     std::cout << "int::size: "
13         << HasMemberT_size<int>::value << '\n' ;
14     std::cout << "std::vector<int>::size: "
15         << HasMemberT_size<std::vector<int>>::value << '\n' ;
16     std::cout << "std::pair<int,int>::first: "
17         << HasMemberT_first<std::pair<int,int>>::value << '\n' ;
18 }
```

修改偏特化, 以排除 &T::Member 不是指向成员类型指针, 这并不困难(相当于排除静态数据成员)。类似地, 可以排除或要求使用指针成员函数, 将特征限制为数据成员或成员函数。

检查成员函数

注意, HasMember 特征只检查是否存在一个具有相应名称的成员。若存在两个成员, 特征也会失败。若检查重载的成员函数, 就可能发生这种情况。例如:

```

1 DEFINE_HAS_MEMBER(begin);
2 std::cout << HasMemberT_begin<std::vector<int>>::value; // false
```

然而, SFINAE 原则可以防止在函数模板声明中同时, 创建无效类型和表达式, 允许上面的重载扩展到测试表达式是否定义良好。

检查是否可以以特定的方式调用感兴趣的函数, 并且即使函数重载也能成功调用。与第 19.5 节中的 IsConvertibleT 特征一样, 技巧在于构造表达式来检查是否可以在 decltype 表达式中调用 begin() 来获取函数模板参数的默认值:

traits/hasbegin.hpp

```

1 #include <utility> // for declval
2 #include <type_traits> // for true_type, false_type, and void_t
3
4 // primary template:
5 template<typename, typename = std::void_t<>>
6 struct HasBeginT : std::false_type {
7 };
8
9 // partial specialization (may be SFINAE'ed away):
10 template<typename T>
11 struct HasBeginT<T, std::void_t<decltype(std::declval<T>().begin())>>
12 : std::true_type {
13 };

```

这里，我们使用

```
1 decltype(std::declval<T>().begin())
```

测试给定一个 T 类型的值/对象 (使用 std::declval 来避免使用构造函数)，调用成员 begin() 是否有效。

除了 decltype(调用表达式) 不需要一个非引用、非 void 的返回类型作为完整的返回类型，这与其他表达式不同。而使用 decltype(std::declval<T>().begin(), 0)，因为返回的值不再是 decltype 操作数的结果，所以要求调用的返回类型完整。

检查其他表达式

可以将上述技术用于其他类型的表达式，甚至组合多个表达式。可以测试给定类型 T1 和 T2，是否为这些类型的值定义了小于操作符：

traits/hasless.hpp

```

1 include <utility> // for declval
2 #include <type_traits> // for true_type, false_type, and void_t
3
4 // primary template:
5 template<typename, typename, typename = std::void_t<>>
6 struct HasLessT : std::false_type {
7 };
8
9
10 // partial specialization (may be SFINAE'ed away):
11 template<typename T1, typename T2>
12 struct HasLessT<T1, T2, std::void_t<decltype(std::declval<T1>()
13 < std::declval<T2>())>>
14 : std::true_type {
15 };

```

```
16 };
```

这里的挑战在于为条件检查定义一个有效的表达式，并使用 decltype 将其放置在 SFINAE 中，若表达式无效，则会回退到主模板：

```
1 decltype(std::declval<T1>() < std::declval<T2>())
```

以这种方式检测有效表达式的特征相当健壮：当表达式定义良好时，才会为 true，当小于操作符不明确、删除或不可访问时，会正确地返回 false。

C++11 扩展 SFINAE 覆盖无效表达式前，检测特定表达式有效性的技术集中于为测试函数引入新的重载（例如，小于操作符），该重载具有过于宽松的签名和一个大小怪异的返回类型，以作为一个备用案例。但这种方法容易产生歧义，并会由于访问控制导致错误。

可以这样使用这个特征：

```
1 HasLessT<int, char>::value // yields true
2 HasLessT<std::string, std::string>::value // yields true
3 HasLessT<std::string, int>::value // yields false
4 HasLessT<std::string, char*>::value // yields true
5 HasLessT<std::complex<double>, std::complex<double>>::value // yields false
```

可以使用这个特征要求模板参数 T 支持小于操作符：

```
1 template<typename T>
2 class C
3 {
4     static_assert(HasLessT<T>::value,
5                 "Class C requires comparable elements");
6     ...
7 };
```

由于 std::void_t 的性质，可以在一个特征中组合多个约束：

traits/hasvarious.hpp

```
1 #include <utility> // for declval
2 #include <type_traits> // for true_type, false_type, and void_t
3
4 // primary template:
5 template<typename, typename = std::void_t<>>
6 struct HasVariousT : std::false_type
7 {
8 };
9
10 // partial specialization (may be SFINAE' d away):
11 template<typename T>
12 struct HasVariousT<T, std::void_t<decltype(std::declval<T>().begin())>,
13             typename T::difference_type,
14             typename T::iterator>>
```

```
15 : std::true_type  
16 {  
17 };
```

检测特定语法有效性的特性非常强大，允许模板根据特定操作的存在或不存在来定制行为。这些特征将再次作为 SFINAE 友好特征的一部分（第 19.4.4 节）和协助基于类型属性的重载（第 20 章）。

19.6.4 使用泛型 Lambda 检查成员

第 19.4.3 节中介绍的 `isValid` Lambda 提供了一种更紧凑的技术来检查成员的特征，有助于避免使用宏来处理成员名称。

下面的例子说明了如何定义特征，检查数据成员或类型成员（如 `first` 或 `size_type`）是否存在，或者是否为两个不同类型的对象定义了小于操作符：

traits/isvalid1.cpp

```
1 #include "isvalid.hpp"  
2 #include<iostream>  
3 #include<string>  
4 #include<utility>  
5  
6 int main()  
7 {  
8     using namespace std;  
9     cout << boolalpha;  
10  
11    // define to check for data member first:  
12    constexpr auto hasFirst  
13        = isValid([](auto x) -> decltype((void)valueT(x).first) {  
14            });  
15  
16    cout << "hasFirst: " << hasFirst(type<pair<int,int>>) << '\n' ; // true  
17  
18    // define to check for member type size_type:  
19    constexpr auto hasSizeType  
20        = isValid([](auto x) -> typename decltype(valueT(x))::size_type {  
21            });  
22  
23    struct CX {  
24        using size_type = std::size_t;  
25    };  
26    cout << "hasSizeType: " << hasSizeType(type<CX>) << '\n' ; // true  
27  
28    if constexpr(!hasSizeType(type<int>)) {  
29        cout << "int has no size_type\n";  
30        ...  
31    }  
32  
33    // define to check for <:
```

```

34 constexpr auto hasLess
35   = isValid([](auto x, auto y) -> decltype(valueT(x) < valueT(y)) {
36     });
37
38 cout << hasLess(42, type<char>) << '\n' ; // yields true
39 cout << hasLess(type<string>, type<string>) << '\n' ; // yields true
40 cout << hasLess(type<string>, type<int>) << '\n' ; // yields false
41 cout << hasLess(type<string>, "hello") << '\n' ; // yields true
42 }

```

因为不能从引用中访问类型成员，所以 hasSizeType 使用 std::decay 从传递的 x 中移除了引用。若跳过该步骤，因为使用了 isValidImpl<>() 的第二次重载，所以特征将总生成 false。

为了能够使用通用泛型语法，将类型作为模板参数，可以再次定义辅助程序：

traits/isvalid2.cpp

```

1 #include "isValid.hpp"
2 #include<iostream>
3 #include<string>
4 #include<utility>
5
6 constexpr auto hasFirst
7   = isValid([](auto&& x) -> decltype((void)&x.first) {
8     });
9
10 template<typename T>
11 using HasFirstT = decltype(hasFirst(std::declval<T>()));
12 constexpr auto hasSizeType
13   = isValid([](auto&& x)
14     -> typename std::decay_t<decltype(x)>::size_type {
15     });
16
17 template<typename T>
18 using HasSizeTypeT = decltype(hasSizeType(std::declval<T>()));
19 constexpr auto hasLess
20   = isValid([](auto&& x, auto&& y) -> decltype(x < y) {
21     });
22
23 template<typename T1, typename T2>
24 using HasLessT = decltype(hasLess(std::declval<T1>(), std::declval<T2>()));
25
26 int main()
27 {
28   using namespace std;
29
30   cout << "first: " << HasFirstT<pair<int,int>>::value << '\n' ; // true
31
32   struct CX {
33     using size_type = std::size_t;

```

```

34    } ;
35
36    cout << "size_type: " << HasSizeTypeT<CX>::value << '\n' ; // true
37    cout << "size_type: " << HasSizeTypeT<int>::value << '\n' ; // false
38
39    cout << HasLessT<int, char>::value << '\n' ; // true
40    cout << HasLessT<string, string>::value << '\n' ; // true
41    cout << HasLessT<string, int>::value << '\n' ; // false
42    cout << HasLessT<string, char*>::value << '\n' ; // true
43 }

```

现在

```

1 template<typename T>
2 using HasFirstT = decltype(hasFirst(std::declval<T>()));

```

允许调用

```
1 HasFirstT<std::pair<int,int>>::value
```

来获取 pair 的两个整型数，其计算方法如上所述。

19.7. 其他方法

最后，介绍并讨论一些定义特征的其他方法。

19.7.1 If-Then-Else

上一节中，根据另一种类型特征 HasPlusT 的结果，PlusResultT 特征的最终定义是一个完全不同的实现。可以用一个特殊的类型模板 IfThenElse 来表达这个 if-else 行为，其接受一个布尔型非类型模板参数，可以从两个类型参数中选择一个：

traits/ifthenelse.hpp

```

1 #ifndef IFTHENELSE_HPP
2 #define IFTHENELSE_HPP
3
4 // primary template: yield the second argument by default and rely on
5 // a partial specialization to yield the third argument
6 // if COND is false
7 template<bool COND, typename TrueType, typename FalseType>
8 struct IfThenElseT {
9     using Type = TrueType;
10 };
11
12 // partial specialization: false yields third argument
13 template<typename TrueType, typename FalseType>
14 struct IfThenElseT<false, TrueType, FalseType> {
15     using Type = FalseType;
16 };

```

```

17
18 template<bool COND, typename TrueType, typename FalseType>
19 using IfThenElse = typename IfThenElseT<COND, TrueType, FalseType>::Type;
20 #endif // IFTHENELSE_HPP

```

下面的示例，通过定义一个类型函数来确定给定值的最低整数类型，来演示此模板的应用：

traits/smallestint.hpp

```

1 #include <limits>
2 #include "ifthenelse.hpp"
3 template<auto N>
4 struct SmallestIntT {
5     using Type =
6         typename IfThenElseT<N <= std::numeric_limits<char>::max(), char,
7         typename IfThenElseT<N <= std::numeric_limits<short>::max(), short,
8         typename IfThenElseT<N <= std::numeric_limits<int>::max(), int,
9         typename IfThenElseT<N <= std::numeric_limits<long>::max(), long,
10        typename IfThenElseT<N <= std::numeric_limits<long long>::max(),
11        long long, // then
12        void // fallback
13    >::Type
14    >::Type
15    >::Type
16    >::Type
17    >::Type;
18 }

```

，与普通 C++ 的 if-then-else 不同，“then” 和“else” 分支的模板参数在选择前求值，因此两个分支不可能包含格式错误的代码，或者格式错误。考虑一个特征，为给定的有符号类型生成相应的无符号类型。有一个标准特征 `std::make_unsigned`，可以执行这个转换，但是要求传递的类型必须是有符号的整型，并且不能是 `bool` 类型；否则，将导致未定义行为（参见第 D.4 节）。可以实现特征来产生相应的无符号类型，否则直接传递的类型（若给出了不恰当的类型，就可以避免了未定义行为）可能是一个好主意。简单的实现不起任何作用：

```

1 // ERROR: undefined behavior if T is bool or no integral type:
2 template<typename T>
3 struct UnsignedT {
4     using Type = IfThenElse<std::is_integral<T>::value
5         && !std::is_same<T,bool>::value,
6         typename std::make_unsigned<T>::type,
7         T>;
8 }

```

`UnsignedT<bool>` 的实例化仍具有未定义行为，因为编译器仍然会试图对这个类型进行转换

```

1 typename std::make_unsigned<T>::type

```

为了解决这个问题，需要添加间接层，以便将 `IfThenElse` 参数本身作为封装结果的类型函数使用：

```

1 // yield T when using member Type:
2 template<typename T>
3 struct IdentityT {
4     using Type = T;
5 };
6
7 // to make unsigned after IfThenElse was evaluated:
8 template<typename T>
9 struct MakeUnsignedT {
10     using Type = typename std::make_unsigned<T>::type;
11 };
12
13 template<typename T>
14 struct UnsignedT {
15     using Type = typename IfThenElse<std::is_integral<T>::value
16             && !std::is_same<T,bool>::value,
17             MakeUnsignedT<T>,
18             IdentityT<T>
19             >::Type;
20 };

```

UnsignedT 这个定义中，IfThenElse 的类型参数都是类型函数本身实例。但在 IfThenElse 选择一个类型函数前，类型函数不会进行实际计算。不过，IfThenElse 选择类型函数实例 (MakeUnsignedT 或 IdentityT)。然后，::Type 计算选定的 Type 函数实例以生成 Type。

这完全依赖于 IfThenElse 构造中，未选择的包装器类型从未完全实例化。特别地，以下代码无效：

```

1 template<typename T>
2 struct UnsignedT {
3     using Type = typename IfThenElse<std::is_integral<T>::value
4             && !std::is_same<T,bool>::value,
5             MakeUnsignedT<T>::Type,
6             T
7             >::Type;
8 };

```

必须在以后为 MakeUnsignedT<T> 添加::Type，还需要 IdentityT 助手在 else 分支中为 T 添加::Type。

所以，不能这样使用

```

1 template<typename T>
2     using Identity = typename IdentityT<T>::Type;

```

可以声明这样一个别名模板，可能在其他地方有用，但是不能在 IfThenElse 的定义中使用。因为 Identity<T> 会立即导致 IdentityT<T> 的完整实例化，从而会检测其 Type 成员。

IfThenElseT 模板在 C++ 标准库中可用，格式为 std::conditional<>(参见第 D.5 节)。UnsignedT 特征将有如下定义：

```

1 template<typename T>
2 struct UnsignedT {
3     using Type
4     = typename std::conditional_t<std::is_integral<T>::value
5             && !std::is_same<T, bool>::value,
6             MakeUnsignedT<T>,
7             IdentityT<T>
8         >::Type;
9 }

```

19.7.2 检测非抛出操作

确定某个特定操作是否会引发异常有时有用。例如，移动构造函数应该标记为 noexcept，表明在任何情况下都不会抛出异常。然而，特定类的移动构造函数是否抛出异常，通常取决于其成员和基类的移动构造函数是否抛出异常。考虑一个简单的类模板 Pair 的移动构造函数：

```

1 template<typename T1, typename T2>
2 class Pair {
3     T1 first;
4     T2 second;
5     public:
6     Pair(Pair&& other)
7         : first(std::forward<T1>(other.first)),
8         second(std::forward<T2>(other.second)) {
9     }
10 }

```

当 T1 或 T2 的移动操作可以抛出异常时，Pair 的移动构造函数可以抛出异常。给定一个特征 IsNothrowMoveConstructibleT，可以通过在 Pair 的移动构造函数中使用计算出来的 noexcept 异常规范来表示这个属性：

```

1 Pair(Pair&& other) noexcept(IsNothrowMoveConstructibleT<T1>::value &&
2     IsNothrowMoveConstructibleT<T2>::value)
3     : first(std::forward<T1>(other.first)),
4     second(std::forward<T2>(other.second))
5 { }

```

剩下的工作就是实现 IsNothrowMoveConstructibleT 特征。可以使用 noexcept 操作符直接实现此特征，确定给定表达式是否不抛出异常：

traits/isnothrowmoveconstructible1.hpp

```

1 #include <utility> // for declval
2 #include <type_traits> // for bool_constant
3
4 template<typename T>
5 struct IsNothrowMoveConstructibleT
6 : std::bool_constant<noexcept(T(std::declval<T>()))>
7 {

```

```
8 };
```

这里使用了操作符版本的 noexcept，用于确定表达式不抛出异常。因为结果是一个布尔值，可以直接传递它来定义基类 std::bool_constant<>，可用于定义 std::true_type 和 std::false_type(参见 19.3.3 节)。

C++11 和 C++14 中，必须将基类指定为 std::integral_constant<bool, …>，而非 std::bool_constant<…>。

因为不是 SFINAE 友好的(参见 19.4.4 节)，所以这个实现应该改进：若特征实例化的类型没有可用的移动或复制构造函数——使得表达式 T(std::declval<T&&>()) 无效——整个程序呈现病态：

```
1 class E {
2     public:
3         E(E&&) = delete;
4     };
5 ...
6 std::cout << IsNothrowMoveConstructibleT<E>::value; // compile-time ERROR
```

类型特征应该产生 false 值，而不是终止编译。

必须在计算结果的表达式求值之前检查其是否有效，要先弄清楚移动构造是否有效，然后再检查是否有例外。因此，修改了特征的第一个版本，添加了一个默认为 void 的模板参数参和一个使用 std::void_t 的局部参数，该参数只有当移动构造有效时才有效：

traits/isnothrowmoveconstructible2.hpp

```
1 #include <utility> // for declval
2 #include <type_traits> // for true_type, false_type, and bool_constant<>
3
4 // primary template:
5 template<typename T, typename = std::void_t<>>
6 struct IsNothrowMoveConstructibleT : std::false_type
7 {
8 };
9
10 // partial specialization (may be SFINAE' d away):
11 template<typename T>
12 struct IsNothrowMoveConstructibleT
13     <T, std::void_t<decltype(T(std::declval<T>()))>>
14 : std::bool_constant<noexcept(T(std::declval<T>()))>
15 {
16 };
```

若替换偏特化中的 std::void_t<…> 有效，特化版本匹配，基类说明符中的 noexcept(…) 表达式可以安全地求值。否则，非实例化偏特化就会丢弃，而会实例化主模板(产生 std::false_type 结果)。

若不能直接调用移动构造函数，就无法检查是否会抛出异常。移动构造函数是 public，且不删除(不够的)，还要求对应的类型不是抽象类(对抽象类的引用或指针都可以)。所以，类型特征命名

为 IsNothrowMoveConstructible，而不是 HasNothrowMoveConstructor。对于其他情况，需要编译器的支持。

C++ 标准库提供了相应的类型特征 std::is_move_constructible<>，在第 D.3.2 节中描述。

19.7.3 特征的便利性

对于类型特征的常见抱怨是它们非常冗长，因为类型特征的每次使用通常需要尾随::type，并在上下文中需要前置 typename 关键字，两者都是样板。当组合多个类型特征时，这可能会导致一些尴尬的格式化，比如在运行的数组加法操作符的例子中，若正确地实现它，确保没有返回常量或引用类型：

```
1 template<typename T1, typename T2>
2 Array<
3     typename RemoveCVT<
4         typename RemoveReferenceT<
5             typename PlusResultT<T1, T2>::Type
6         >::Type
7     >::Type
8 >
9 operator+ (Array<T1> const&, Array<T2> const&);
```

通过使用别名模板和变量模板，可以更方便地使用特征，分别生成类型或值。但这些快捷方式有时可能不可用的，这时必须使用原始的类模板。我们已经在 MemberPointerToIntT 示例中讨论过这样的情况，但接下来将进行更通用的讨论。

别名模板和特征

别名模板提供了一种减少代码冗长的方法。可以直接使用别名模板，而不是将类型特征表示为具有类型成员 type 的类模板。例如，以下三个别名模板包装了上面使用的类型特征：

```
1 template<typename T>
2 using RemoveCV = typename RemoveCVT<T>::Type;
3
4 template<typename T>
5 using RemoveReference = typename RemoveReferenceT<T>::Type;
6
7 template<typename T1, typename T2>
8 using PlusResult = typename PlusResultT<T1, T2>::Type;
```

有了这些别名模板，可以将加法操作符简化为

```
1 template<typename T1, typename T2>
2 Array<RemoveCV<RemoveReference<PlusResult<T1, T2>>>>
3 operator+ (Array<T1> const&, Array<T2> const&);
```

第二个版本显然更短，使特征的组成更明显。这样的改进使别名模板更适合于类型特征的某些使用。

然而，对类型特征使用别名模板也有缺点：

1. 别名模板不能特化(如第 16.3 节所述),由于许多写入特征的技术都依赖特化,别名模板无论如何都需要重定向到类模板。
2. 有些特征是由用户特化的,例如描述特定加法操作是否可交换的特征,当大多数使用涉及别名模板时,特化类模板可能会令人困惑。
3. 别名模板的使用总是会实例化该类型(例如,底层的类模板特化),这使得避免实例化对给定类型没有意义的特征变得更加困难(如 19.7.1 节所讨论的)。

最后一点的另一种表达方式是,别名模板不能与元函数转发一起使用(参见 19.3.2 节)。

为类型特征使用别名模板既有积极的一面,也有消极的一面,所以建议像本节中,以及在 C++ 标准库中那样使用它们:为两个类模板提供特定的命名约定(我们选择了 T 后缀和 type 类型成员),为别名模板提供略有不同的命名约定(我们去掉了 T 后缀),并根据底层类模板定义每个别名模板。通过这种方式,可以通过别名模板提供更清晰代码,但为了更高级的使用,也可以返回到类模板。

由于历史原因,C++ 标准库有不同的约定。类型特征类模板产生类型中的类型,并且没有特定的后缀(很多是在 C++11 中引入的)。因为没有后缀的名称已经标准化(参见 D.1 节),相应的别名模板(直接产生类型)在 C++14 中引入,并赋予了_t 后缀。

可变参模板和特征

返回值的特征需要添加::value(或类似的成员选择)来产生特征的结果。constexpr 变量模板(在第 5.6 节中介绍)提供了一种减少这种冗长的方法。

下面的变量模板包装了第 19.3.3 节定义的 IsSameT 特征和第 19.5 节定义的 IsConvertibleT 特征:

```

1 template<typename T1, typename T2>
2 constexpr bool IsSame = IsSameT<T1, T2>::value;
3
4 template<typename FROM, typename TO>
5 constexpr bool IsConvertible = IsConvertibleT<FROM, TO>::value;

```

可以将代码简化为

```

1 if (IsSame<T, int> || IsConvertible<T, char>) ...

```

从而替代原始方式

```

1 if (IsSameT<T, int>::value || IsConvertibleT<T, char>::value) ...

```

由于历史原因,C++ 标准库有不同的约定。产生值结果的特征类模板没有特定的后缀,它们中的许多在 C++11 标准中引入。直接产生结果值的相应变量模板在 C++17 中使用了_v 后缀(参见 D.1 节)。

C++ 标准化委员会受到长期存在的传统约束,即所有标准名称都由小写字符和可选的下划线分隔。也就是说,像 isSame 或 isSame 这样的名称不太可能加入标准中(使用这种拼写风格的概念除外)。

19.8. 类型分类

有时，能够提前知道模板参数是内置类型、指针类型、类类型。下面的部分中，将开发一组类型特征，允许确定给定类型的各种属性。因此，可以编写特定于某些类型的代码：

```
1 if (IsClass<T>::value) {  
2     ...  
3 }
```

或者，C++17(见第 8.5 节)起使用编译时(见第 19.7.3 节)：

```
1 if constexpr (IsClass<T>) {  
2     ...  
3 }
```

或者，通过偏特化：

```
1 template<typename T, bool = IsClass<T>>  
2 class C { // primary template for the general case  
3     ...  
4 };  
5  
6 template<typename T>  
7 class C<T, true> { // partial specialization for class types  
8     ...  
9 };
```

此外，像 `IsPointerT<T>::value` 这样的表达式将是布尔常量，是有效的非类型模板参数。从而允许构建更复杂、更强大的模板，这些模板的行为特化于类型参数的属性。

C++ 标准库定义了几个类似的特征，以确定类型的主要类型和复合类型类别。

“主要”和“复合”类型类别的使用不应与“基本”和“复合”类型混淆。该标准描述了基本类型(如 `int` 或 `std::nullptr_t`)和复合类型(如指针类型和类类型)。这与复合类型类别(如算术)不同，复合类型类别是主要类型类别(如浮点)的组合。

参见第 D.2.1 节和第 D.2.2 节的详细信息。

19.8.1 基本类型

首先，开发一个模板来确定一个类型是否是基本类型。默认情况下，假设一个类型不是基本类型，我们特化了基础模板：

traits/isfunda.hpp

```
1 #include <cstddef> // for nullptr_t  
2 #include <type_traits> // for true_type, false_type, and  
3     bool_constant<>  
4 // primary template: in general T is not a fundamental type  
5 template<typename T>
```

```

6 struct IsFundaT : std::false_type {
7 };
8
9 // macro to specialize for fundamental types
10 #define MK_FUNDA_TYPE(T) \
11 template<> struct IsFundaT<T> : std::true_type { \
12 } ;
13
14 MK_FUNDA_TYPE(void)
15 MK_FUNDA_TYPE(bool)
16 MK_FUNDA_TYPE(char)
17 MK_FUNDA_TYPE(signed char)
18 MK_FUNDA_TYPE(unsigned char)
19 MK_FUNDA_TYPE(wchar_t)
20 MK_FUNDA_TYPE(char16_t)
21 MK_FUNDA_TYPE(char32_t)
22
23 MK_FUNDA_TYPE(signed short)
24 MK_FUNDA_TYPE(unsigned short)
25 MK_FUNDA_TYPE(signed int)
26 MK_FUNDA_TYPE(unsigned int)
27 MK_FUNDA_TYPE(signed long)
28 MK_FUNDA_TYPE(unsigned long)
29 MK_FUNDA_TYPE(signed long long)
30 MK_FUNDA_TYPE(unsigned long long)
31
32 MK_FUNDA_TYPE(float)
33 MK_FUNDA_TYPE(double)
34 MK_FUNDA_TYPE(long double)
35
36 MK_FUNDA_TYPE(std::nullptr_t)
37
38 #undef MK_FUNDA_TYPE

```

主模板定义一般情况，IsFundaT<T>::value 将计算为 false：

```

1 template<typename T>
2 struct IsFundaT : std::false_type {
3     static constexpr bool value = false;
4 }

```

对于每个基本类型，都定义了一个特化，以便 IsFundaT<T>::value 为 true。我们定义了一个宏，可以扩展为方便起见所需的代码。例如：

```

1 MK_FUNDA_TYPE(bool)

```

展开为：

```

1 template<> struct IsFundaT<bool> : std::true_type {
2     static constexpr bool value = true;
3 }

```

下面的程序演示了该模板的用法:

traits/isfundatest.cpp

```
1 #include "isfunda.hpp"
2 #include <iostream>
3
4 template<typename T>
5 void test (T const&)
6 {
7     if (IsFundaT<T>::value) {
8         std::cout << "T is a fundamental type" << '\n' ;
9     }
10    else {
11        std::cout << "T is not a fundamental type" << '\n' ;
12    }
13 }
14
15 int main()
16 {
17     test(7);
18     test("hello");
19 }
```

具有以下输出:

```
T is a fundamental type
T is not a fundamental type
```

同样的方式，可以定义类型函数 IsIntegralT 和 IsFloatingT 来识别这些类型中哪些是整型标量类型，哪些是浮点标量类型。

C++ 标准库使用了比只检查类型是否是基本类型更细粒度的方法。首先，定义了主要类型类别，其中每个类型精确匹配一个类型类别（参见第 D.2.1 节），然后是复合类型类别，如 std::is_integral 或 std::is_fundamental（参见第 D.2.2 节）。

19.8.2 复合类型

由其他类型构造而成的类型。简单复合类型包括指针类型、左值和右值引用类型、成员指针类型和数组类型，由一个或两个基础类型构造而成。类类型和函数类型也是复合类型，但组合可能涉及任意数量的类型（用于参数或成员）。枚举类型在此分类中也可视为非简单复合类型，尽管它们不是多种基础类型的复合类型。可以使用偏特化对简单的复合类型进行分类。

指针

可以从简单指针的类型分类开始：

traits/ispointer.hpp

```
1 template<typename T>
2 struct IsPointerT : std::false_type { // primary template: by default not a pointer
3 } ;
4
5 template<typename T>
6 struct IsPointerT<T*> : std::true_type { // partial specialization for pointers
7     using BaseT = T; // type pointing to
8 } ;
```

主模板是一个非指针类型的案例，提供 value 常量 false，通过基类 std::false_type，表明该类型不是指针。偏特化可以捕获指针 (T*)，并将 value 置为 true，来指示所提供的类型是指针。此外，还提供了一个类型成员 BaseT，该成员描述指针所指向的类型。注意，该类型成员仅在原始类型为指针时可用，使其成为 SFINAE 友好型的类型特征(参见第 19.4.4 节)。

C++ 标准库提供了相应的特征 std::is_pointer<>，但不提供指针所指向类型的成员。在第 D.2.1 节中有描述。

引用

类似地，可以确定左值引用类型：

traits/islvaluerefrence.hpp

```
1 template<typename T>
2 struct IsLValueReferenceT : std::false_type { // by default no lvalue reference
3 } ;
4
5 template<typename T>
6 struct IsLValueReferenceT<T&> : std::true_type { // unless T is lvalue references
7     using BaseT = T; // type referring to
8 } ;
```

和右值引用类型：

traits/isrvaluerefrence.hpp

```
1 template<typename T>
2 struct IsRValueReferenceT : std::false_type { // by default no rvalue reference
3 } ;
4
5 template<typename T>
6 struct IsRValueReferenceT<T&&> : std::true_type { // unless T is rvalue reference
7     using BaseT = T; // type referring to
8 } ;
```

可以组合成一个 IsReferenceT<> 特征：

traits/isreference.hpp

```
1 #include "islvaluereference.hpp"
2 #include "isrvaluereference.hpp"
3 #include "ifthenelse.hpp"
4
5 template<typename T>
6 class IsReferenceT
7 : public IfThenElseT<IsLValueReferenceT<T>::value,
8     IsLValueReferenceT<T>,
9     IsRValueReferenceT<T>
10 >::Type {
11 };
```

使用 `IfThenElseT`(19.7.1 节) 来选择 `IsLValueReference<T>` 或 `IsRValueReference<T>` 作为基类, 使用元函数转发(19.3.2 节讨论)。如果 `T` 是左值引用, 则继承 `IsLValueReference<T>` 以获得适当的值和 `BaseT` 成员。否则, 继承 `IsRValueReference<T>`, 确定类型是否是右值引用(并在两种情况下提供适当的成员)。

C++ 标准库提供了相应的特征 `std::is_lvalue_reference<>` 和 `std::is_rvalue_reference<>`, 在第 D.2.1 节中描述; 以及 `std::is_reference<>`, 在第 D.2.2 节中描述。同样, 这些特征不会为引用所引用的类型提供成员。

数组

当使用特征来确定数组时, 可能会惊讶于偏特化比主模板需要更多的模板参数:

traits/isarray.hpp

```
1 #include <cstddef>
2
3 template<typename T>
4 struct IsArrayT : std::false_type { // primary template: not an array
5 };
6
7 template<typename T, std::size_t N>
8 struct IsArrayT<T[N]> : std::true_type { // partial specialization for arrays
9     using BaseT = T;
10    static constexpr std::size_t size = N;
11 };
12
13 template<typename T>
14 struct IsArrayT<T[]> : std::true_type { // partial specialization for unbound arrays
15     using BaseT = T;
16     static constexpr std::size_t size = 0;
17 };
```

多个成员提供了关于分类数组的信息: 它们的基类型和大小(0 表示未知大小)。

C++ 标准库提供了相应的特征 `std::is_array` 来检查类型是否为数组，这在第 D.2.1 节中有描述。此外，诸如 `std::rank` 和 `std::extent` 这样的特征可以查询它们的维数和特定维的大小（参见第 D.3.1 节）。

指向成员的指针

成员指针可以使用相同的技术处理：

traits/ispointertomember.hpp

```
1 template<typename T>
2 struct IsPointerToMemberT : std::false_type { // by default no pointer-to-member
3 };
4
5 template<typename T, typename C>
6 struct IsPointerToMemberT<T C::*> : std::true_type { // partial specialization
7     using MemberT = T;
8     using ClassT = C;
9 };
```

这里，附加成员提供了成员的类型和产生该成员类的类型。

C++ 标准库提供了更具体的特征，`std::is_member_object_pointer` 和 `std::is_member_function_pointer`，在第 D.2.1 节中描述，以及 `std::is_member_pointer`，在第 D.2.2 节中描述。

19.8.3 函数类型

函数类型很有趣，除了结果类型之外，还有任意数量的参数。因此，匹配函数类型的偏特化中，使用一个参数包来捕获所有的参数类型：

traits/isfunction.hpp

```
1 #include "../typelist/typelist.hpp"
2
3 template<typename T>
4 struct IsFunctionT : std::false_type { // primary template: no function
5 };
6
7 template<typename R, typename... Params>
8 struct IsFunctionT<R (Params...)> : std::true_type { // functions
9     using Type = R;
10    using ParamsT = Typelist<Params...>;
11    static constexpr bool variadic = false;
12 };
13
14 template<typename R, typename... Params>
15 struct IsFunctionT<R (Params..., ...)> : std::true_type { // variadic functions
16     using Type = R;
17 }
```

```
17  using ParamsT = Typelist<Params...>;
18  static constexpr bool variadic = true;
19 }
```

函数类型的每个部分都是公开的: type 提供结果类型, 而所有参数都作为 ParamsT 在一个类型列表中捕获(类型列表将在第 24 章中介绍), variadic 表示该函数类型是否使用了 C 风格的 varargs。

但 IsFunctionT 不能处理所有的函数类型, 因为函数类型可以有 const 和 volatile 限定符, 以及左值(&)和右值(&&)引用限定符(在第 C.2.1 节中描述)。C++17 后, 没有任何限定符的情况除外:

```
1 using MyFuncType = void (int&) const;
```

这种函数类型只能用于非静态成员函数, 但仍然是函数类型。此外, 标记为 const 的函数类型实际上不是 const,

当函数类型标记为 const 时, 指向隐式参数 this 所指向的对象上的限定符, 而 const 类型上的 const 指向的是实际类型的对象。

因此, RemoveConst 不能从函数类型中删除 const。要识别具有限定符的函数类型, 需要引入大量的偏特化, 包括限定符的每一种组合(包括带有和不带 C 风格可变参数的组合)。这里, 只展示其中的 5 个偏特化要求:

最新的数字是 48。

```
1 template<typename R, typename... Params>
2 struct IsFunctionT<R (Params...) const> : std::true_type {
3     using Type = R;
4     using ParamsT = Typelist<Params...>;
5     static constexpr bool variadic = false;
6 };
7
8 template<typename R, typename... Params>
9 struct IsFunctionT<R (Params..., ...) volatile> : std::true_type {
10    using Type = R;
11    using ParamsT = Typelist<Params...>;
12    static constexpr bool variadic = true;
13 };
14
15 template<typename R, typename... Params>
16 struct IsFunctionT<R (Params..., ...) const volatile> : std::true_type {
17    using Type = R;
18    using ParamsT = Typelist<Params...>;
19    static constexpr bool variadic = true;
20 };
21
22 template<typename R, typename... Params>
23 struct IsFunctionT<R (Params..., ...) &> : std::true_type {
24    using Type = R;
```

```

25 using Paramst = Typelist<Params...>;
26 static constexpr bool variadic = true;
27 };
28
29 template<typename R, typename... Params>
30 struct IsFunctionT<R (Params..., ...) const&> : std::true_type {
31     using Type = R;
32     using Paramst = Typelist<Params...>;
33     static constexpr bool variadic = true;
34 };
35 ...

```

所有这些就绪后，可以对除类类型和枚举类型外的所有类型进行分类。我们将在下面的部分中处理这些情况。

C++ 标准库提供特征 `std::is_function<>`，将在第 D.2.1 节中描述。

19.8.4 类类型

与其他复合类型不同，我们没有专门匹配类类型的偏特化模式。也不可能像基本类型那样枚举所有类类型。不过，需要使用间接方法来标识类类型，方法是提出对所有类类型（而不是其他类型）有效的某种类型或表达式。对于这种类型或表达式，可以应用于第 19.4 节中的 SFINAE。

类类型最方便利用的属性是，只有类类型可以用作成员指针类型的基础。在 `X Y::*` 形式的类型构造中，`Y` 只能是类类型。`IsClassT<>` 利用了这个属性（将类型 `X` 选定为 `int`）：

traits/isclass.hpp

```

1 #include <type_traits>
2
3 template<typename T, typename = std::void_t<>>
4 struct IsClassT : std::false_type { // primary template: by default no class
5 };
6
7 template<typename T>
8 struct IsClassT<T, std::void_t<int T::*>> // classes can have pointer-to-member
9 : std::true_type {
10 };

```

C++ 语言指定 Lambda 表达式的类型是“唯一、未命名的非联合类类型”。因此，检查 Lambda 表达式是否为类类型对象时，结果为 `true`：

```

1 auto l = []{};
2 static_assert<IsClassT<decltype(l)>::value, "">; // succeeds

```

表达式 `int T::*` 对于联合类型也是有效的（根据 C++ 标准，其也是类类型）。

C++ 标准库提供了特征 `std::is_class<>` 和 `std::is_union<>`，在第 D.2.1 节中有描述。因为目前无法使用标准的核心语言技术来区分类和结构体与联合体类型，所以这些特征需要编译器支持。

大多数编译器支持像`_is_union`这样的内在操作符，以帮助标准库实现各种特征模板。甚至对于一些可以使用本章中的技术实现的特性也是如此，这些特性可以提高编译性能。

19.8.5 枚举类型

还没有特征分类的类型是枚举类型。枚举类型的测试可以通过编写基于 SFINAE 的特征直接执行，该特征检查会转换为整型类型（比如`int`），并显式排除基本类型、类类型、引用类型、指针类型和指向成员的指针类型，所有这些类型都可以转换为整型类型，但不是枚举类型。

本书的第一版就是这样描述枚举类型检测的。但它检查了到整型的隐式转换，这对于 C++98 标准来说足够了。语言中引入作用域枚举类型（没有这种隐式转换）使枚举类型的检测变得更加复杂。

不过，可以通过“不属于其他类别的类型都必须是枚举类型”的方式来实现：

traits/isenum.hpp

```
1 template<typename T>
2 struct IsEnumT {
3     static constexpr bool value = !IsFundamentalT<T>::value &&
4         !IsPointerT<T>::value &&
5         !IsReferenceT<T>::value &&
6         !IsArrayT<T>::value &&
7         !IsPointerToMemberT<T>::value &&
8         !IsFunctionT<T>::value &&
9         !IsClassT<T>::value;
10    };
```

C++ 标准库提供特征`std::is_enum<>`，在第 D.2.1 节中描述。为了提高编译性能，编译器会直接提供对这种特征的支持。

19.9. 策略特征

特征模板示例已经用于确定模板参数的属性：表示什么类型，应用于该类型值的操作符的结果类型等。这样的特征称为属性特征。

一些特征定义了某些类型应该如何对待，称之为策略特征。这让人想起了前面讨论的策略类的概念（特征和策略之间的区别并不完全清晰），但是策略特征往往是与模板参数相关联的唯一属性（而策略类通常独立于其他模板参数）。

虽然属性特征通常可以作为类型函数实现，但策略特征通常将策略封装在成员函数中。让我们来看一个类型函数，定义了传递只读参数的策略。

19.9.1 只读参数类型

C 和 C++ 中，函数调用参数默认按值传递。调用方计算的参数值可复制到被调用方控制的位置。对于大型结构来说，这样做的代价会很高，而且对于这样的结构来说，可以通过指向 `const` 的引用（或 C 中指向 `const` 的指针）传递参数。对于较小的结构，从性能角度来看，最好的机制取决于所编写代码的架构。大多数情况下，这并不是那么重要，但有时即使是很小的结构也必须谨慎处理。

对于模板，事情变得更加微妙：事先不知道替换的模板参数类型有多大。此外，这个决定不仅取决于大小：小的结构可能会附带一个昂贵的复制构造函数，仍然可以通过引用到 `const` 来验证传递只读参数。

这个问题可以通过一个策略特征模板处理，该模板是一个类型函数：该函数将预期的参数类型 T 映射到最优参数类型 T 或 T `const&`。主模板可以对不超过两个指针的类型使用按值传递，对其他类型使用引用 `const` 传递：

```
1 template<typename T>
2 struct RParam {
3     using Type = typename IfThenElse<sizeof(T)<=2*sizeof(void*),
4                           T,
5                           T const&>::Type;
6 };
```

另一方面，对于 `sizeof` 返回小值的容器类型，可能会涉及昂贵的复制构造函数，因此可能需要许多特化和偏特化：

```
1 template<typename T>
2 struct RParam<Array<T>> {
3     using Type = Array<T> const&;
4 };
```

因为这类类型在 C++ 中很常见，所以只使用简单的复制和移动构造函数，作为值类型来标记小类型可能会更安全

若对复制或移动构造函数的调用，可以用底层字节的简单复制来替换，则将其称 trivial。

然后有选择地添加其他类类型 (`std::is_trivially_copy_constructible` 和 `std::is_trivially_move_constructible` 类型特征是 C++ 标准库的一部分)。

traits/rparam.hpp

```
1 #ifndef RPARAM_HPP
2 #define RPARAM_HPP
3
4 #include "ifthenelse.hpp"
5 #include <type_traits>
6
7 template<typename T>
8 struct RParam {
9     using Type
```

```

10 = IfThenElse<(sizeof(T) <= 2*sizeof(void*)
11     && std::is_trivially_copy_constructible<T>::value
12     && std::is_trivially_move_constructible<T>::value),
13     T,
14     T const&>;
15 };
16
17 #endif // RPARAM_HPP

```

无论采用哪种方式，策略现在都可以集中在特征模板定义中，外部可以很好地利用它。假设有两个类，其中一个类指定通过值调用更适合只读参数：

traits/rparamcls.hpp

```

1 #include "rparam.hpp"
2 #include <iostream>
3
4 class MyClass1 {
5 public:
6     MyClass1 () {
7 }
8     MyClass1 (MyClass1 const&) {
9         std::cout << "MyClass1 copy constructor called\n";
10    }
11 };
12
13 class MyClass2 {
14 public:
15     MyClass2 () {
16 }
17     MyClass2 (MyClass2 const&) {
18         std::cout << "MyClass2 copy constructor called\n";
19 }
20 };
21
22 // pass MyClass2 objects with RParam<> by value
23 template<>
24 class RParam<MyClass2> {
25 public:
26     using Type = MyClass2;
27 };

```

可以声明使用 `RParam<>` 作为只读参数的函数，并调用这些函数：

traits/rparamI.cpp

```

1 #include "rparam.hpp"
2 #include "rparamcls.hpp"
3
4 // function that allows parameter passing by value or by reference

```

```

5 template<typename T1, typename T2>
6 void foo (typename RParam<T1>::Type p1,
7 typename RParam<T2>::Type p2)
8 {
9     ...
10 }
11
12 int main()
13 {
14     MyClass1 mc1;
15     MyClass2 mc2;
16     foo<MyClass1,MyClass2>(mc1,mc2);
17 }
```

使用 RParam 有一些明显的缺点。首先，函数声明明显更混乱。其次，因为模板参数只出现在函数参数的限定符中，所以不能用参数推导调用 foo()。因此，调用站点必须明确指定模板参数。

这个选项的一个笨拙的解决方法是使用内联包装器函数模板，提供完美的转发功能(第 15.6.3 节)，但其假定编译器会省略内联函数：

traits/rparam2.cpp

```

1 #include "rparam.hpp"
2 #include "rparamcls.hpp"
3
4 // function that allows parameter passing by value or by reference
5 template<typename T1, typename T2>
6 void foo_core (typename RParam<T1>::Type p1,
7     typename RParam<T2>::Type p2)
8 {
9     ...
10 }
11
12 // wrapper to avoid explicit template parameter passing
13 template<typename T1, typename T2>
14 void foo (T1 && p1, T2 && p2)
15 {
16     foo_core<T1,T2>(std::forward<T1>(p1),std::forward<T2>(p2));
17 }
18
19 int main()
20 {
21     MyClass1 mc1;
22     MyClass2 mc2;
23     foo(mc1,mc2); // same as foo_core<MyClass1,MyClass2>(mc1,mc2)
24 }
```

19.10. 标准库

C++11 中，类型特征成为 C++ 标准库的一部分，或多或少包含了本章讨论的所有类型函数和类型特征。但对于其中的一些，比如操作检测特征和 `std::is_union`，没有已知的语言解决方案，但编译器需要为这些特性提供了内在的支持。此外，因为有缩短编译时间的解决方案，所以编译器也开始支持特征了。

因此若需要类型特征，建议在可用的情况下使用 C++ 标准库中的类型特征。它们在附录 D 中都有详细的描述。

请注意（如前所述），有些特征具有诡异行为（至少对于菜鸟开发者来说）。除了在的 11.2.1 节和 D.1.2 节中给出的一般性提示外，也要考虑在附录 D 中提供的具体描述。

C++ 标准库还定义了一些策略和属性特征：

- 类模板 `std::char_traits` 可以作为字符串和 I/O 流类的策略特征参数。
- 为了使算法易于所使用的标准迭代器类型，提供了 `std::iterator_traits` 属性特性模板（并在标准库接口中使用）。
- 模板 `std::numeric_limits` 可以用作属性特征模板。
- 最后，标准容器类型的内存分配使用策略特征类处理。自 C++98 以来，模板 `std::allocator` 作为标准组件提供。C++11 中，添加了模板 `std::allocator_traits` 以能够改变分配器的策略/行为（可在经典行为和作用域分配器之间切换；后者无法在 C++11 之前的框架中使用）。

19.11. 后记

Nathan Myers 是第一个将特征参数的概念形式化的人。他最初把它们提交给 C++ 标准化委员会，作为定义在标准库组件（例如输入和输出流）中应该如何处理字符类型的工具。当时，他称它们为“包模板”，并指出它们包含特征。然而，一些 C++ 委员会成员并不喜欢“包”这个词，取而代之的是使用名称特征。从那时起，这个术语逐渐广泛使用。

外部代码通常根本不处理特征：默认的特征类满足最常见的需求，而且因为是默认的模板参数，根本不需要出现在外部源码中。这就可以为默认特征模板使用较长的描述性名称。当外部代码确实通过提供自定义特征参数来调整模板行为时，为产生的特化声明一个适合自定义行为的类型别名，是一个很好的方式。特征类可以赋予一个很长的描述性名称，而不会浪费太多的资源。

特征可以作为反射的一种形式，程序检查自己的高级属性（比如类型结构）。像 `IsClassT` 和 `PlusResultT` 这样的特征，以及许多其他检查程序中类型的类型特征，实现了一种编译时反射的形式，是元编程的强大盟友（参见第 23 章和第 17.9 节）。

将类型属性存储为模板特化成员的想法，至少可以追溯到 20 世纪 90 年代中期。在类型分类模板的早期应用中，有 SGI（当时称为硅图形）发布的 STL 实现中的 `_type_traits` 程序。SGI 模板意在表示其模板参数的一些属性（是否是一个普通的旧数据类型（POD）或析构函数是简单）。然后，使用这些信息对给定类型的某些 STL 算法进行优化。SGI 解决方案的有趣特性是，一些 SGI 编译器能够识别 `_type_traits` 特化，并提供关于不能使用标准技术派生的参数的信息（`_type_traits` 模板的通用实现使用起来尽管不是最优的，但是安全的）。

Boost 提供了一套相当完整的类型分类模板（参见 [BoostTypeTraits]），构成了 2011 C++ 标准库中 `<type_traits>` 头文件的基础。这些特征中的许多特性都可以通过本章描述的技术实现，但其他的

(用于检测 POD 的 `std::is_pod`) 需要编译器的支持，就像 SGI 编译器提供的 `_type_traits` 特化一样。

第一次标准化工作期间描述类型推演和替代规则时，就注意到 SFINAE 原则用于类型分类的目的。然而，这没有正式的文档记录，后来花费了大量的精力创建本章中描述的一些技术。本书的第一版是这种技术最早的来源之一，介绍了术语 SFINAE。这一领域的另一位著名的早期贡献者是 Andrei Alexandrescu，他普及了 `sizeof` 操作符来确定重载解析的结果。使这种技术变得流行，以至于 2011 年的标准将 SFINAE 的范围从简单的类型错误，扩展为函数模板的错误(参见 [SpicerSFINAE])。这个扩展结合了 `decltype`、右值引用和可变参数模板的添加，极大地扩展了在特征中测试特定属性的能力。

使用像 `isValid` 这样的泛型 Lambda 来提取 SFINAE 条件，是 Louis Dionne 在 2015 年引入的一种技术，Boost.Hana 使用了这种技术(参见 [boostana]，一个适合在编译时对类型和值进行计算的元编程库)。

策略类是由许多开发者开发的。Andrei Alexandrescu 使策略类这个术语流行起来，他的书《现代 C++ 设计》比我们的简短介绍(参见 [AlexandrescuDesign])更加详细地介绍了策略类。

第 20 章 类型属性上的重载

函数重载允许在多个函数中使用相同的函数名，需要通过参数类型区分这些函数即可：

```
1 void f(int);  
2 void f(char const*);
```

使用函数模板，可以重载类型模式，如 T 的指针或 Array<T>：

```
1 template<typename T> void f(T*);  
2 template<typename T> void f(Array<T>);
```

鉴于类型特征的存在（第 19 章），希望使用基于模板参数的属性重载函数模板。例如：

```
1 template<typename Number> void f(Number); // only for numbers  
2 template<typename Container> void f(Container); // only for containers
```

但 C++ 目前没有提供直接的方法来表示基于类型属性的重载，上面的两个 f 函数模板实际上是同一个函数模板的声明，因为在比较两个函数模板时，模板参数的名称会忽略。

幸运的是，有许多技术可以用来模拟基于类型属性的函数模板的重载。本章将讨论这些技术，以及使用这种重载的动机。

20.1. 算法特化

函数模板重载的一个常见动机是，基于所涉及的类型知识提供算法的更特化版本。使用简单的 swap() 来交换两个值：

```
1 template<typename T>  
2 void swap(T& x, T& y)  
3 {  
4     T tmp(x);  
5     x = y;  
6     y = tmp;  
7 }
```

这个实现涉及三个复制操作。对于某些类型，可以提供更有效的 swap() 操作，例如 Array<T> 将其数据存储为指向数组内容和长度的指针：

```
1 template<typename T>  
2 void swap(Array<T>& x, Array<T>& y)  
3 {  
4     swap(x.ptr, y.ptr);  
5     swap(x.len, y.len);  
6 }
```

swap() 的两个实现都将交换两个 Array<T> 对象的内容。但后一种实现更高效，因为它使用了 Array<T> 的属性（特别是 ptr 和 len 及其各种成员），而这些属性对于其他类型不可用。

swap() 的一个更好的选择是使用 std::move() 来避免在主模板中进行复制，这里提出的替代方案适用范围更广。

因此，后一个函数模板（在概念上）比前一个更特化，前一个函数模板接受的类型子集执行相同的操作。基于函数模板的部分排序规则，第二个函数模板也更加特化了（参见 16.2.2 节），因此编译器将选择更特化（因此更高效）的函数模板（例如，`Array<T>` 参数），而当更特化的版本不适用时，会回退到更通用（可能更低效）的算法。

为通用算法引入更特化的设计和优化方法称为算法特化。更特化的版本应用于泛型算法的有效输入子集，根据特定的类型或类型的属性可以识别这个子集，其通常比泛型算法的一般实现更有效。

对于实现算法特化至关重要，当有更特化的变量适用时，调用者无需了解这些变量的存在就会自动选择。`swap()` 的示例中，这是通过使用通用函数模板（第一个 `swap()`）重载（概念上）更特化的函数模板（第二个 `swap()`）来实现的，并确保更特化的函数模板根据 C++ 的部分排序规则也更特化。

并不是所有概念上更特化的算法版本，都可以直接转换为提供正确的部分排序行为的函数模板。对于下一个示例，`advanceIter()` 函数（类似于 C++ 标准库中的 `std::advance()`），将迭代器 `x` 向前移动 `n` 步。这个通用算法可以操作输入迭代器：

```
1 template<typename InputIterator, typename Distance>
2 void advanceIter(InputIterator& x, Distance n)
3 {
4     while (n > 0) { // linear time
5         ++x;
6         --n;
7     }
8 }
```

对于提供随机访问操作的特定迭代器类，可以提供更高效的实现：

```
1 template<typename RandomAccessIterator, typename Distance>
2 void advanceIter(RandomAccessIterator& x, Distance n) {
3     x += n; // constant time
4 }
```

定义这两个函数模板将导致编译器报错，因为仅根据模板参数名称，而不同的函数模板不可重载。本章的其余部分将讨论模拟重载这些函数模板所需的技术。

20.2. 标签调度

算法特化的方法是通过唯一“标记”来识别不同的实现，为了处理 `advanceIter()` 的问题，可以使用标准库的迭代器类别标签类型（定义如下）来识别 `advanceIter()` 算法的两个实现变体：

```
1 template<typename Iterator, typename Distance>
2 void advanceIterImpl(Iterator& x, Distance n, std::input_iterator_tag)
3 {
4     while (n > 0) { // linear time
5         ++x;
6         --n;
7     }
8 }
9
10 template<typename Iterator, typename Distance>
```

```
11 void advanceIterImpl(Iterator& x, Distance n,
12     std::random_access_iterator_tag) {
13     x += n; // constant time
14 }
```

advanceIter() 函数模板只是简单地转发参数和相应的标签:

```
1 template<typename Iterator, typename Distance>
2 void advanceIter(Iterator& x, Distance n)
3 {
4     advanceIterImpl(x, n,
5         typename
6             std::iterator_traits<Iterator>::iterator_category());
7 }
```

特征类 std::iterator_traits 通过其成员类型 iterator_category 为迭代器提供了一个类别。迭代器类别是前面提到的_tag 类型之一，其指定了迭代器类型的类型。C++ 标准库中，可用的标签定义如下，使用继承来反映标签描述类别是从另一个标签派生而来的：

类别反映了概念，概念的继承称为精炼。在附录 E 中详细介绍了概念和精炼。

```
1 namespace std {
2     struct input_iterator_tag { };
3     struct output_iterator_tag { };
4     struct forward_iterator_tag : public input_iterator_tag { };
5     struct bidirectional_iterator_tag : public forward_iterator_tag { };
6     struct random_access_iterator_tag : public bidirectional_iterator_tag { };
7 }
```

利用标签调度的关键在于标签之间的关系。advanceIterImpl() 的两个变体标记为 std::input_iterator_tag 和 std::random_access_iterator_tag，并且 std::random_access_iterator_tag 继承自 std::input_iterator_tag，所以使用随机访问迭代器调用 advanceIterImpl() 时，普通函数都会优先使用更特化的算法版本（使用 std::random_access_iterator_tag）。标签调度依赖于从单一的、主要的函数模板委派到一组_impl 变体，对这些变体进行标记，这样普通的函数重载将选择适用于给定模板参数最特化的算法。

当算法使用的属性具有体系结构，以及提供这些标记值的现有特征集时，标签调度工作得很好。当算法特化依赖于特定的类型属性时，就不那么方便了，比如类型 T 是否具有普通的复制赋值操作符。因此，我们需要更强大的技术。

20.3. 启用/禁用函数模板

算法特化包括提供根据模板参数的属性选择的不同函数模板，函数模板的部分排序（16.2.2 节）和重载解析（附录 C）都不足以表达更高级的算法特化形式。

C++ 标准库为此提供了 std::enable_if（第 6.3 节）。本节讨论如何通过引入相应的别名模板来进行实现一个辅助类，将其称为 EnableIf（以避免名称冲突）。

与 `std::enable_if` 一样, `EnableIf` 别名模板可以用于在特定条件下启用 (或禁用) 特定函数模板。`advanceIter()` 算法的随机访问版本可以这样实现:

```
1 template<typename Iterator>
2 constexpr bool IsRandomAccessIterator =
3     IsConvertible<
4         typename std::iterator_traits<Iterator>::iterator_category,
5         std::random_access_iterator_tag>;
6
7 template<typename Iterator, typename Distance>
8 EnableIf<IsRandomAccessIterator<Iterator>>
9 advanceIter(Iterator& x, Distance n) {
10     x += n; // constant time
11 }
```

`EnableIf` 特化用于随机访问迭代器时, 启用 `advanceIter()` 的相应版本。`EnableIf` 的两个参数是一个布尔条件, 指示是否应该启用该模板, 以及当条件为 `true` 时, 由 `EnableIf` 展开生成的类型。上面的例子中, 使用类型特征 `IsConvertible`(第 19.5 节和第 19.7.3 节中介绍) 作为条件, 定义了类型特征 `IsRandomAccessIterator`。当 `Iterator` 的具体类型可用作随机访问迭代器时, 才会考虑 `advanceIter()` 实现的这个版本(即其与一个可转换为 `std::random_access_iterator_tag` 的标记相关联)。

`EnableIf` 有一个相当简单的实现:

typeoverload/enableif.hpp

```
1 template<bool, typename T = void>
2 struct EnableIfT {
3 };
4
5 template<typename T>
6 struct EnableIfT<true, T> {
7     using Type = T;
8 };
9
10 template<bool Cond, typename T = void>
11 using EnableIf = typename EnableIfT<Cond, T>::Type;
```

`EnableIf` 展开为一个类型, 因此实现为一个别名模板。我们希望对其实现使用偏特化(参见第 16 章), 但是别名模板不能偏特化。这里, 可以引入辅助类模板 `EnableIfT`, 其可以完成相应的实际工作, 并使用别名模板 `EnableIf` 从辅助模板中简单地选择结果类型。当条件为 `true` 时, `EnableIfT<...>::Type` 只计算第二个模板参数 `T`。当条件为 `false` 时, `EnableIf` 不会产生一个有效的类型, 因为 `EnableIfT` 的主类模板没有 `Type` 成员。这将是一个错误, 但在 SFINAE(替换失败不是过, 在第 15.7 节中描述) 上下文中——函数模板的返回类型——会导致模板参数推导失败, 从而忽略函数模板。

`EnableIf` 也可以放在默认的模板参数中, 这比放在结果类型中更有优势。参见第 20.3.2 节关于 `EnableIf` 位置的讨论。

对于 `advanceIter()`, 使用 `EnableIf` 意味着, 当 `Iterator` 参数是随机访问迭代器时, 函数模板可用

(并且返回类型为 void), 当 Iterator 参数不是随机访问迭代器时, 函数模板将从对象中移除。可以把 EnableIf 看作是一种“保护”模板, 当模板参数不满足模板实现要求的时, 才实例化的方法。因为这个 advanceIter() 只能用随机访问迭代器实例化, 所以相应的操作只能在随机访问迭代器上使用。虽然以这种方式使用 EnableIf 并不绝对可靠——用户可以断言某个类型是随机访问迭代器, 而不需要提供必要的操作——但它可以帮助及早的诊断错误。

现在已经建立了显式地“激活”所应用类型的更特化的模板, 这还不够: 还必须“停用”不太特化的模板, 因为编译器无法对这两个模板进行“排序”, 若两个版本都适用, 就会出现歧义错误。不过, 实现这一点并不困难: 只需要在不太特化的模板上使用相同的 EnableIf 模式, 对条件表达式求反即可。这样做可以确保激活 Iterator 类型两个模板中的一个。因此, advanceIter() 的非随机访问迭代器版本如下所示:

```
1 template<typename Iterator, typename Distance>
2 EnableIf<!IsRandomAccessIterator<Iterator>>
3 advanceIter(Iterator& x, Distance n)
4 {
5     while (n > 0) { // linear time
6         ++x;
7         --n;
8     }
9 }
```

20.3.1 多个特化版本

前面的模式可以推广到需要两个以上不同实现: 为每个实现提供 EnableIf 构造, 其条件对一组具体模板参数互斥。这些条件通常会利用各种特征表达的属性。

例如, 引入 advanceIter() 算法的第三种实现: 这次希望通过指定负距离来允许“后退”移动。

算法特化只用于在计算时间或资源使用方面提供效率增益。然而, 一些算法的特化也提供了更多的功能, 例如(在本例中)按顺序向后移动的能力。

这对于输入迭代器无效, 但对于随机访问迭代器有效。标准库还包含双向迭代器的概念, 其允许向后移动, 而不需要随机访问。实现这种情况需要复杂一点的逻辑: 每个函数模板必须使用 EnableIf 和一个条件, 该条件与表示算法不同实现的所有其他函数模板的条件互斥。这将需要以下一组条件:

- 随机访问迭代器: 随机访问情况(常量耗时, 向前或向后)
- 双向迭代器, 而不是随机访问: 双向情况(线性耗时, 向前或向后)
- 输入迭代器且非双向: 一般情况(线性耗时, 向前)

下面一组函数模板实现了这一点:

typeoverload/advance2.hpp

```
1 #include <iterator>
2
3 // implementation for random access iterators:
```

```

4 template<typename Iterator, typename Distance>
5 EnableIf<IsRandomAccessIterator<Iterator>>
6 advanceIter(Iterator& x, Distance n) {
7     x += n; // constant time
8 }
9
10 template<typename Iterator>
11 constexpr bool IsBidirectionalIterator =
12     IsConvertible<
13         typename std::iterator_traits<Iterator>::iterator_category,
14         std::bidirectional_iterator_tag>;
15
16 // implementation for bidirectional iterators:
17 template<typename Iterator, typename Distance>
18 EnableIf<IsBidirectionalIterator<Iterator> &&
19     !IsRandomAccessIterator<Iterator>>
20 advanceIter(Iterator& x, Distance n) {
21     if (n > 0) {
22         for ( ; n > 0; ++x, --n) { // linear time
23     }
24     } else {
25         for ( ; n < 0; --x, ++n) { // linear time
26     }
27     }
28 }
29
30 // implementation for all other iterators:
31 template<typename Iterator, typename Distance>
32 EnableIf<!IsBidirectionalIterator<Iterator>>
33 advanceIter(Iterator& x, Distance n) {
34     if (n < 0) {
35         throw "advanceIter(): invalid iterator category for negative n";
36     }
37     while (n > 0) { // linear time
38         ++x;
39         --n;
40     }
41 }

```

通过使每个函数模板的 `EnableIf` 条件与其他每个函数模板的 `EnableIf` 条件互斥，可以确保对于给定的参数集，最多有一个函数模板能成功推导模板参数。

示例说明了将 `EnableIf` 用于算法特化的缺点之一：每次引入算法的新实现时，都需要重新访问所有算法实现的条件，以确保所有实现都互斥。而使用标签调度（20.2 节）引入双向迭代器变体，只需要使用标记 `std::bidirectional_iterator_tag` 添加一个新的 `advanceIterImpl()` 重载即可。

标签调度和 `EnableIf` 这两种技术很有用：标签调度支持基于分层标签的简单调度，而 `EnableIf` 支持基于由类型特征确定的属性集的高级调度。

20.3.2 使用 EnableIfGo

`EnableIf` 用于函数模板的返回类型。但这种方法不适用于构造函数模板或转换函数模板，因为没有指定的返回类型。

虽然转换函数模板确实有返回类型，但转换到该类型的模板参数必须可推导(参见第15章)，这样转换函数模板才能正常工作。

此外，`EnableIf` 的使用会使返回类型很难阅读，所以可以将 `EnableIf` 嵌入到默认的模板参数中：

typeoverload/container1.hpp

```
1 #include <iterator>
2 #include "enableif.hpp"
3 #include "isconvertible.hpp"
4
5 template<typename Iterator>
6 constexpr bool IsInputIterator =
7     IsConvertible<
8         typename std::iterator_traits<Iterator>::iterator_category,
9         std::input_iterator_tag>;
10
11 template<typename T>
12 class Container {
13 public:
14     // construct from an input iterator sequence:
15     template<typename Iterator,
16             typename = EnableIf<IsInputIterator<Iterator>>>
17     Container(Iterator first, Iterator last);
18
19     // convert to a container so long as the value types are convertible:
20     template<typename U, typename = EnableIf<IsConvertible<T, U>>>
21     operator Container<U>() const;
22 };
```

若添加另一个重载 (更高效的随机访问迭代器 Container 构造函数版本), 将会导致错误:

```
1 // construct from an input iterator sequence:  
2 template<typename Iteration,  
3     typename = EnableIf<IsInputIterator<Iteration> &&  
4             !IsRandomAccessIterator<Iteration>>>  
5 Container(Iteration first, Iteration last);  
6  
7 template<typename Iterator,  
8     typename = EnableIf<IsRandomAccessIterator<Iteration>>>  
9 Container(Iteration first, Iteration last); // ERROR: redeclaration  
10 // of constructor template
```

问题是，除了默认模板参数外，两个构造函数模板完全相同，但是在确定两个模板是否等价时不会考虑默认模板参数。

可以通过添加另一个默认的模板参数来解决这个问题，这样两个构造函数模板的模板参数数量就不同了：

```
1 // construct from an input iterator sequence:
2 template<typename Iterator,
3     typename = EnableIf<IsInputIterator<Iterator> &&
4         !IsRandomAccessIterator<Iterator>>>
5 Container(Iterator first, Iterator last);
6
7 template<typename Iterator,
8     typename = EnableIf<IsRandomAccessIterator<Iterator>>,
9     typename = int> // extra dummy parameter to enable both constructors
10 Container(Iterator first, Iterator last); // OK now
```

20.3.3 编译时 if

C++17 的 `constexpr if` 特性（请参阅第 8.5 节）可以替代 `EnableIf`。C++17 中，可以这样重写 `advanceIter()`：

typeoverload/advance3.hpp

```
1 template<typename Iterator, typename Distance>
2 void advanceIter(Iterator& x, Distance n) {
3     if constexpr(IsRandomAccessIterator<Iterator>) {
4         // implementation for random access iterators:
5         x += n; // constant time
6     }
7     else if constexpr(IsBidirectionalIterator<Iterator>) {
8         // implementation for bidirectional iterators:
9         if (n > 0) {
10             for ( ; n > 0; ++x, --n) { // linear time for positive n
11                 }
12         } else {
13             for ( ; n < 0; --x, ++n) { // linear time for negative n
14                 }
15         }
16     }
17     else {
18         // implementation for all other iterators that are at least input iterators:
19         if (n < 0) {
20             throw "advanceIter(): invalid iterator category for negative n";
21         }
22         while (n > 0) { // linear time for positive n only
23             ++x;
24             --n;
25         }
26     }
27 }
```

```
26 }  
27 }
```

这就清楚多了！更特化的代码路径(用于随机访问迭代器)会支持它们的类型实例化。因此，只要操作使用 `if constexpr` 保护，那代码中包含并非所有迭代器中都有的操作(例如 `+=`)就是安全的。

然而，其也有缺点。当通用组件中的差异完全可以在函数模板体中表达时，才可能使用 `constexpr if`。下面的情况下，仍然需要 `EnableIf`：

- 涉及到不同的“接口”
- 需要不同的类定义
- 对于某些模板参数列表，不存在有效的实例化。

可以用下面的模式来处理最后一种情况：

```
1 template<typename T>  
2 void f(T p) {  
3     if constexpr (condition<T>::value) {  
4         // do something here...  
5     }  
6     else {  
7         // not a T for which f() makes sense:  
8         static_assert(condition<T>::value, "can't call f() for such a T");  
9     }  
10 }
```

不过这样做并不可取，因为其不适用于 SFINAE：函数 `f<T>()` 没有从候选列表中删除，因此可能会抑制另一个重载解析的结果。替代方案中，使用 `EnableIf f<T>()` 将删除 `EnableIf<…>` 失败情况的替换。

20.3.4 概念

目前为止所介绍的技术表现的都还好，但是有点笨拙，可能会使用大量的编译器资源，并且在出现错误的情况下，可能会出现不清晰的诊断信息。因此，许多泛型库的作者都期望有一种能够更直接地达到同样效果的语言特性。出于这个原因，概念的特性可能会添加到语言中；请参阅第 6.5 节、第 18.4 节和附录 E。

例如，期望重载的 Container 构造函数如下所示：

typeoverload/container4.hpp

```
1 template<typename T>  
2 class Container {  
3     public:  
4         // construct from an input iterator sequence:  
5         template<typename Iterator>  
6         requires IsInputIterator<Iterator>  
7         Container(Iterator first, Iterator last);  
8  
9         // construct from a random access iterator sequence:
```

```

10 template<typename Iterator>
11 requires IsRandomAccessIterator<Iterator>
12 Container(Iterator first, Iterator last);
13
14 // convert to a container so long as the value types are convertible:
15 template<typename U>
16 requires IsConvertible<T, U>
17 operator Container<U>() const;
18 };
```

require 子句 (在 E.1 节中讨论) 描述了模板的需求。若没有满足要求，该模板就不认为是候选模板。因此，其是 EnableIf 思想的更直接的表达，语言本身也支持这种表达。

与 EnableIf 相比，requires 子句有更多的优点。约束包含 (在第 E.3.1 节) 提供了模板之间的排序，只在 require 子句中有所不同，从而消除了标签调度的需要。此外，可以将 require 子句附加到非模板上。仅当类型 T 可使用小于操作符进行比较时，才提供 sort() 成员函数：

```

1 template<typename T>
2 class Container {
3     public:
4     ...
5     requires HasLess<T>
6     void sort() {
7         ...
8     }
9 };
```

20.4. 类的特化

类模板偏特化可用于为特定模板参数提供类模板的特化实现，就像我们为函数模板使用重载一样。与重载函数模板一样，可以根据模板参数的属性区分偏特化。考虑使用键值类型作为模板参数的泛型 Dictionary 类模板。只要键类型只提供相等操作符，就可以实现一个简单 (但效率低) 的 Dictionary：

```

1 template<typename Key, typename Value>
2 class Dictionary
3 {
4     private:
5     vector<pair<Key const, Value>> data;
6     public:
7     // subscripted access to the data:
8     value& operator[](Key const& key)
9     {
10         // search for the element with this key:
11         for (auto& element : data) {
12             if (element.first == key) {
13                 return element.second;
14             }
15         }
16     }
17 }
```

```

16 // there is no element with this key; add one
17 data.push_back(pair<Key const, Value>(key, Value()));
18 return data.back().second;
19 }
20 ...
21 };

```

若键类型支持小于操作符，则可以基于标准库的 map 容器提供更有效的实现。类似地，若键类型支持哈希操作，可以使用标准库的 unordered_map。

20.4.1 启用/禁用类模板

启用/禁用类模板的不同实现的方法是使用启用/禁用类模板的偏特化。要在类模板偏特化中使用 EnableIf，首先向 Dictionary 引入一个未命名的默认模板参数：

```

1 template<typename Key, typename Value, typename = void>
2 class Dictionary
3 {
4     ... // vector implementation as above
5 };

```

新的模板参数作为 EnableIf 的锚，可以嵌入到 Dictionary 的 map 版本的偏特化模板参数列表中：

```

1 template<typename Key, typename Value>
2 class Dictionary<Key, Value,
3                 EnableIf<HasLess<Key>>>
4 {
5     private:
6     map<Key, Value> data;
7     public:
8     value& operator[](Key const& key) {
9         return data[key];
10    }
11    ...
12 };

```

与重载函数模板不同，这里不需要在主模板上禁用条件，因为偏特化优先于主模板。当使用哈希操作添加另一个键实现时，需要确保偏特化条件互斥：

```

1 template<typename Key, typename Value, typename = void>
2 class Dictionary
3 {
4     ... // vector implementation as above
5 };
6
7 template<typename Key, typename Value>
8 class Dictionary<Key, Value,
9                 EnableIf<HasLess<Key> && !HasHash<Key>>>
10 {
11     ... // map implementation as above
12 };

```

```

13
14 template<typename Key, typename Value>
15 class Dictionary<Key, Value,
16 EnableIf<HasHash<Key>>>
17 {
18     private:
19     unordered_map<Key, Value> data;
20     public:
21     value& operator[](Key const& key) {
22         return data[key];
23     }
24     ...
25 };

```

20.4.2 类模板的标签调度

标签调度也可以用于在类模板偏特化之间进行选择。我们定义了一个函数对象类型 Advance<Iterator>, 类似于前面章节中使用的 advanceIter() 算法, 将迭代器向前推进若干步。我们提供了通用实现(用于输入迭代器), 以及双向和随机访问迭代器的特化实现, 依赖于一个辅助特征 BestMatchInSet(如下所述) 来为迭代器的类别标签选择最佳匹配:

```

1 // primary template (intentionally undefined):
2 template<typename Iterator,
3     typename Tag =
4     BestMatchInSet<
5         typename std::iterator_traits<Iterator>
6             ::iterator_category,
7             std::input_iterator_tag,
8             std::bidirectional_iterator_tag,
9             std::random_access_iterator_tag>>
10 class Advance;
11
12 // general, linear-time implementation for input iterators:
13 template<typename Iterator>
14 class Advance<Iterator, std::input_iterator_tag>
15 {
16     public:
17     using DifferenceType =
18     typename std::iterator_traits<Iterator>::difference_type;
19     void operator() (Iterator& x, DifferenceType n) const
20     {
21         while (n > 0) {
22             ++x;
23             --n;
24         }
25     }
26 };
27
28 // bidirectional, linear-time algorithm for bidirectional iterators:

```

```

29 template<typename Iterator>
30 class Advance<Iterator, std::bidirectional_iterator_tag>
31 {
32     public:
33     using DifferenceType =
34         typename std::iterator_traits<Iterator>::difference_type;
35
36     void operator() (Iterator& x, DifferenceType n) const
37     {
38         if (n > 0) {
39             while (n > 0) {
40                 ++x;
41                 --n;
42             }
43         } else {
44             while (n < 0) {
45                 --x;
46                 ++n;
47             }
48         }
49     }
50 };
51
52 // bidirectional, constant-time algorithm for random access iterators:
53 template<typename Iterator>
54 class Advance<Iterator, std::random_access_iterator_tag>
55 {
56     public:
57     using DifferenceType =
58         typename std::iterator_traits<Iterator>::difference_type;
59
60     void operator() (Iterator& x, DifferenceType n) const
61     {
62         x += n;
63     }
64 }
```

表达式与函数模板的标签调度非常相似，问题在于编写特征 BestMatchInSet，该特征要确定给定迭代器的(输入、双向和随机访问迭代器标签)最匹配的标签。这个特征说明，给定迭代器的 category 标签的值，将选择下列哪个重载，并报告其参数类型：

```

1 void f(std::input_iterator_tag);
2 void f(std::bidirectional_iterator_tag);
3 void f(std::random_access_iterator_tag);
```

模拟重载解析最简单的方法是使用重载解析：

```

1 // construct a set of match() overloads for the types in Types...
2 template<typename... Types>
3 struct MatchOverloads;
```

```

5 // basis case: nothing matched:
6 template<>
7 struct MatchOverloads<> {
8     static void match(...);
9 };
10
11 // recursive case: introduce a new match() overload:
12 template<typename T1, typename... Rest>
13 struct MatchOverloads<T1, Rest...> : public MatchOverloads<Rest...> {
14     static T1 match(T1); // introduce overload for T1
15     using MatchOverloads<Rest...>::match; // collect overloads from bases
16 };
17
18 // find the best match for T in Types...:
19 template<typename T, typename... Types>
20 struct BestMatchInSetT {
21     using Type = decltype(MatchOverloads<Types...>::match(declval<T>()));
22 };
23
24 template<typename T, typename... Types>
25 using BestMatchInSet = typename BestMatchInSetT<T, Types...>::Type;

```

MatchOverloads 模板使用递归继承来为 Types 输入集中的每个类型声明一个 match() 函数。递归 match 重载偏特化的每次实例化，都会为列表中的下一个类型引入一个新的 match() 函数。然后，使用 using 声明引入在其基类中定义的 match() 函数，该函数处理列表中的其余类型。当处理递归时，结果是一组对应于给定类型的 match() 重载，每个重载返回其参数类型。BestMatchInSetT 模板然后将一个 T 对象传递给这组重载 match() 函数，并生成所选(最佳)match() 函数的返回类型。

C++17 中，可以在基类列表和 using 声明中使用包扩展来消除递归(第 4.4.5 节)。将在第 26.4 节演示这种技术。

如果两个函数都不匹配，则返回 void(使用省略号来捕获参数) 表示失败。

失败的情况下，不提供结果会更好一些，使其成为 SFINAE 友好特征(参见第 19.4.4 节)。此外，健壮的实现将返回类型包装在类似 Identity 的东西中，因为有些类型(如数组和函数类型)可以是参数类型，但不能是返回类型。为了简洁和可读性，省略了这些改进。

总而言之，BestMatchInSetT 将函数重载结果转换为特征，并使标签调度在类模板偏特化间的选择，变得相对容易。

20.5. 实例化安全的模板

EnableIf 本质是只在模板参数满足某些特定条件时，启用特化或偏特化模板。advanceIter() 算法有效的形式是检查迭代器参数类别是否可转换为 std::random_access_iterator_tag，从而算法可用于各种随机访问迭代器。

若把这个概念发挥到极致，模板对其模板参数执行的每个操作都编码为 `EnableIf` 条件的一部分，会怎么样呢？模板实例化永远不会失败，因为不提供所需操作的模板参数推导将失败（通过 `EnableIf`），而不是允许实例化继续进行。我们将这种模板称为“实例化安全”模板，并在这里介绍这种模板的实现。

我们从一个非常基本的模板 `min()` 开始，计算两个值的最小值。通常会这样实现一个模板：

```
1 template<typename T>
2 T const& min(T const& x, T const& y)
3 {
4     if (y < x) {
5         return y;
6     }
7     return x;
8 }
```

该模板要求类型 `T` 具有能够使用小于操作符比较两个 `T` 值（特别是两个 `T const` 左值），然后隐式地将比较结果转换为 `bool`，以便在 `if` 语句中使用。检查小于操作符，并计算结果类型的特征类似于在第 19.4.4 节讨论的 SFINAE 友好的 `PlusResultT` 特征，但为了方便起见，在这里先展示一下 `LessResultT` 特征：

typeoverload/lessresult.hpp

```
1 #include <utility> // for declval()
2 #include <type_traits> // for true_type and false_type
3
4 template<typename T1, typename T2>
5 class HasLess {
6     template<typename T> struct Identity;
7     template<typename U1, typename U2> static std::true_type
8         test(Identity<decltype(std::declval<U1>() < std::declval<U2>())>*);
9     template<typename U1, typename U2> static std::false_type
10        test(...);
11
12 public:
13     static constexpr bool value = decltype(test<T1, T2>(nullptr))::value;
14 };
15
16 template<typename T1, typename T2, bool HasLess>
17 class LessResultImpl {
18     public:
19         using Type = decltype(std::declval<T1>() < std::declval<T2>());
20 };
21
22 template<typename T1, typename T2>
23 class LessResultImpl<T1, T2, false> {
24 };
25
26 template<typename T1, typename T2>
27 class LessResultT
28 : public LessResultImpl<T1, T2, HasLess<T1, T2>::value> {
```

```
28 } ;
29
30 template<typename T1, typename T2>
31 using LessResult = typename LessResultT<T1, T2>::Type;
```

这个特征可以和 IsConvertible 特征组合在一起，使 min() 实例化安全：

typeoverload/min2.hpp

```
1 #include "isconvertible.hpp"
2 #include "lessresult.hpp"
3
4 template<typename T>
5 EnableIf<IsConvertible<LessResult<T const&, T const&>, bool>,
6     T const&>
7 min(T const& x, T const& y)
8 {
9     if (y < x) {
10         return y;
11     }
12     return x;
13 }
```

尝试使用不同类型的小于操作符 (或完全忽略操作符) 来调用这个 min() 函数：

typeoverload/min.cpp

```
1 #include "min.hpp"
2
3 struct X1 { };
4 bool operator< (X1 const&, X1 const&) { return true; }
5
6 struct X2 { };
7 bool operator<(X2, X2) { return true; }
8
9 struct X3 { };
10 bool operator<(X3&, X3&) { return true; }
11
12 struct X4 { };
13 struct BoolConvertible {
14     operator bool() const { return true; } // implicit conversion to bool
15 };
16
17 struct X5 { };
18 BoolConvertible operator< (X5 const&, X5 const&)
19 {
20     return BoolConvertible();
21 }
22
23 struct NotBoolConvertible { // no conversion to bool
```

```

24 } ;
25
26 struct X6 { };
27 NotBoolConvertible operator< (X6 const&, X6 const&)
28 {
29     return NotBoolConvertible();
30 }
31
32 struct BoolLike {
33     explicit operator bool() const { return true; } // explicit conversion to bool
34 };
35 struct X7 { };
36 BoolLike operator< (X7 const&, X7 const&) { return BoolLike(); }
37
38 int main()
39 {
40     min(X1(), X1()); // X1 can be passed to min()
41     min(X2(), X2()); // X2 can be passed to min()
42     min(X3(), X3()); // ERROR: X3 cannot be passed to min()
43     min(X4(), X4()); // ERROR: X4 cannot be passed to min()
44     min(X5(), X5()); // X5 can be passed to min()
45     min(X6(), X6()); // ERROR: X6 cannot be passed to min()
46     min(X7(), X7()); // UNEXPECTED ERROR: X7 cannot be passed to min()
47 }

```

编译时虽然对 X3、X4、X6 和 X7 的 4 个不同 min() 调用存在错误，但这些错误并不来自 min() 的主体。相反，而是抱怨没有合适的 min() 函数，因为 SFINAE 已经取消了匹配的选项。Clang 会输出以下诊断信息：

```

min.cpp:41:3: error: no matching function for call to 'min'
min(X3(), X3()); // ERROR: X3 cannot be passed to min
^~~
./min.hpp:8:1: note: candidate template ignored: substitution failure
[with T = X3]: no type named 'Type' in
'LessResultT<const X3 &, const X3 &>'
min(T const& x, T const& y)

```

因此，EnableIf 只允许实例化那些满足模板要求的模板参数 (X1, X2, 和 X5)，所以从 min() 函数体中不会得到错误。此外，若有一些可能适用于这些类型的 min() 的其他重载，重载解析可以选择其中一个。

示例中的最后一个类型 X7，演示了实现实例化安全模板的一些微妙之处。特别是，若将 X7 传递给非实例化安全的 min()，则实例化成功。因为 BoolLike 不能隐式转换为 bool，所以实例化安全的 min() 不能使用。这里的区别特别微妙：显式转换为 bool 可以在某些上下文中隐式使用，包括控制流语句的布尔条件 (if、while、for 和 do)、内置的!、&& 和 || 操作符以及三元操作符。这些上下文

中，值可以转换为 bool。

C++11 引入了到 bool 的上下文转换概念，以及显式转换操作符。它们取代了“安全 bool”的习惯用法 ([KarlssonSafeBool])，这通常涉及到(隐式)用户定义的数据指针转换。使用指向数据的指针的原因是可以将其视为 bool 值，而不需要其他多余的转换，例如 bool 作为算术操作的一部分可以提升为 int。BoolConvertible() + 5(不幸的是)是定义良好的代码。

然而，我们坚持使用一般的、隐式的 bool 转换，结果导致实例化安全模板过度约束；指定的需求(在 EnableIf 中)比实际需求(模板需要正确实例化的东西)更强。另一方面，若完全忘记了转换到 bool 的要求，min() 模板就会约束不足，会允许一些可能导致实例化失败的模板参数参与实例化(例如 X6)。

为了修复实例化安全的 min()，需要特征来确定类型 T 是否在上下文中可转换为 bool。控制流语句对定义特征没有帮助，因为语句不能在 SFINAE 中出现(逻辑操作也不能)，所以可以为任意类型重载逻辑操作。三元操作符是一个表达式，并且没有重载，因此可以使用它来测试类型是否在上下文条件下可以转换为 bool 类型：

typeoverload/iscontextualbool.hpp

```
1 #include <utility> // for declval()
2 #include <type_traits> // for true_type and false_type
3
4 template<typename T>
5 class IsContextualBoolT {
6     private:
7         template<typename T> struct Identity;
8         template<typename U> static std::true_type
9             test(Identity<decltype(declval<U>() ? 0 : 1)>*);
10        template<typename U> static std::false_type
11            test(...);
12    public:
13        static constexpr bool value = decltype(test<T>(nullptr))::value;
14    };
15
16 template<typename T>
17 constexpr bool IsContextualBool = IsContextualBoolT<T>::value;
```

有了这个新特征，可以在 EnableIf 中提供一个实例化安全的 min()，其包含正确的请求集：

typeoverload/min3.hpp

```
1 #include "iscontextualbool.hpp"
2 #include "lessresult.hpp"
3
4 template<typename T>
5 EnableIf<IsContextualBool<LessResult<T const&, T const&>>,
6         T const&>
7 min(T const& x, T const& y)
```

```
8 {
9   if (y < x) {
10     return y;
11   }
12   return x;
13 }
```

这里用于使 `min()` 实例化安全的技术，可以扩展为描述非简单模板的需求，方法是将各种需求检查组合到描述某些类型类的特征中，比如前向迭代器，并在 `EnableIf` 中组合这些特征。这样做有两个好处，一是可以更好地重载行为，二是可以消除编译器在嵌套模板实例化中深入输出错误时，产生新的错误。另一方面，提供的错误消息往往缺乏关于哪个特定操作失败的特性。此外，正如用 `min()` 所展示的那样，准确地确定和编码模板的需求，可能是一项令人生畏的任务。我们将在 28.2 节中探讨利用这些特性进行调试的方法。

20.6. 标准库

C++ 标准库为输入、输出、前向、双向和随机访问迭代器标签提供了迭代器标签，在本文中已经使用了这些标签。这些迭代器标签是标准迭代器特征 (`std::iterator_traits`) 和放置在迭代器上的要求的一部分，所以可以安全地用于标签调度。

C++11 标准库 `std::enable_if` 类模板提供了与 `EnableIfT` 类模板相同的功能。唯一的区别是标准使用了名为 `type` 的小写成员类型，而不是大写类型。

C++ 标准库的很多地方都使用了特化算法。`std::advance()` 和 `std::distance()` 都有几个基于其迭代器参数类别的版本。尽管最近一些已经采用 `std::enable_if` 来实现这种特化算法，但大多数标准库实现倾向于使用标签调度。许多 C++ 标准库实现，也在内部使用这些技术来实现各种标准算法的特化算法。`std::copy()` 可以实现 `std::memcpy()` 或 `std::memmove()`，当迭代器指向连续的内存并且它们的值类型有普通的复制赋值操作符时。可以对 `std::fill()` 进行优化，以实现 `std::memset()`，并且当已知类型具有普通析构函数时，各种算法可以避免调用析构函数。这些特化算法不像 `std::advance()` 或 `std::distance()` 那样由 C++ 标准强制规定，但是实现者出于效率的原因会有选择的进行提供。

正如在第 8.4 节中介绍的那样，C++ 标准库还在其要求中使用 `std::enable_if<>` 或类似的基于 SFINAE 的技术。`std::vector` 有一个构造函数模板，允许从迭代器序列构建 `vector`:

```
1 template<typename InputIterator>
2 vector(InputIterator first, InputIterator second,
3         allocator_type const& alloc = allocator_type());
```

要求“若构造函数调用的，是不符合输入迭代器条件的 `InputIterator` 类型，那么构造函数将不参与重载解析”（参见 [C++11]23.2.3 第 14 段）。这个措辞非常模糊，可以使用当时最有效的技术来实现它，在将它添加到标准中时，应该会使用 `std::enable_if<>` 实现。

20.7. 后记

标签调度在 C++ 中存在很久了，其用于 STL 的原始实现中（参见 [StepanovLeeSTL]），并且经常与特征一起使用。SFINAE 和 `EnableIf` 比较新：本书的第一版（参见 [VandevoordeJosuttisTemplates1st]）介绍了术语 SFINAE，并用于检测成员类型的存在。

“enable if” 技术和术语最早是由 Jaakko Jarvi、Jeremiah Willcock、Howard Hinnant 和 Andrew Lumsdaine 在 [OverloadingProperties] 中提出的，描述了 EnableIf 模板，如何用 EnableIf(和 DisableIf) 对实现函数重载，以及如何用类模板偏特化来使用 EnableIf。从那时起，EnableIf 和类似的技术在高级模板库(包括 C++ 标准库)的实现中无处不在。此外，这些技术的流行激发了 C++11 扩展 SFINAE 的行为(参见 15.7 节)。Peter Dimov 是第一个注意到函数模板的默认模板参数(另一个 C++11 特性)使得可以在构造函数模板中使用 EnableIf，而无需引入另一个函数参数。

概念语言特性(附录 E 中描述)有望在 C++17 后的 C++ 标准中出现，其可使许多涉及 EnableIf 的技术过时。与此同时，C++17 的 `constexpr if` 语句(参见第 8.5 节和第 20.3.3 节)也在逐渐减少它们在现代模板库中的存在感。

第 21 章 模板和继承

可能没有理由认为模板和继承会以友善的方式进行交互。若有什么区别的话，可以在第 13 章了解到，从依赖基类的派生迫使我们小心地处理非限定名称。然而，一些有趣的技术结合了这两个特性，包括奇异递归模板模式 (CRTP) 和混合类。本章中，将介绍其中的一些。

21.1. 空基类优化 (EBCO)

C++ 类通常是“空的”，内部表示在运行时不需要内存位。对于包含类型成员、非虚函数成员和静态数据成员的类，通常就是这样。另一方面，非静态数据成员、虚函数和虚基类在运行时确实需要一些内存。

即使是空类，其大小也是非零的。若想验证这一点，可以试试下面的代码：

inherit/empty.cpp

```
1 #include <iostream>
2
3 class EmptyClass {
4 };
5
6 int main()
7 {
8     std::cout << "sizeof(EmptyClass): " << sizeof(EmptyClass) << '\n';
9 }
```

对于多数平台，这个程序将输出 1 作为 EmptyClass 的大小。一些系统对类类型有更严格的对齐要求，可能会打印另一个小整数（通常为 4）。

21.1.1 结构布置原则

C++ 的设计者有各种理由来避免零大小的类。一个由零大小类组成的数组的大小应该也为零，但是指针算法的属性将不再适用。假设 ZeroSizedT 是一个零大小的类型：

```
1 ZeroSizedT z[10];
2 ...
3 &z[i] - &z[j] // compute distance between pointers/addresses
```

前一个示例中的差异是通过两个地址之间的字节数，除以它所指向的类型的大小得到的，但当该大小为零时，这显然不令人满意。

然而，即使在 C++ 中没有零大小的类型，C++ 标准指定一个空类作为基类时，不需要为它分配空间，只要不分配到与另一个相同类型对象或子对象相同地址中即可。我们通过一些示例来阐明这个空基类优化 (EBCO) 在实践中的含义：

inherit/ebo1.cpp

```
1 #include <iostream>
```

```

2
3 class Empty {
4     using Int = int; // type alias members don't make a class nonempty
5 };
6
7 class EmptyToo : public Empty {
8 };
9
10 class EmptyThree : public EmptyToo {
11 };
12
13 int main()
14 {
15     std::cout << "sizeof(Empty): " << sizeof(Empty) << '\n';
16     std::cout << "sizeof(EmptyToo): " << sizeof(EmptyToo) << '\n';
17     std::cout << "sizeof(EmptyThree): " << sizeof(EmptyThree) << '\n';
18 }

```

若编译器实现了 EBCO，将为每个类打印相同的大小，但这些类的大小都不是 0(参见图 21.1)。类 EmptyToo 中，类 Empty 没有赋予空间，优化过的空基类(没有其他基类)的空类也是空的。这解释了，为什么类 EmptyThree 也可以具有与类 Empty 相同的大小。若编译器没有实现 EBCO，将输出不同的大小(参见图 21.2)。

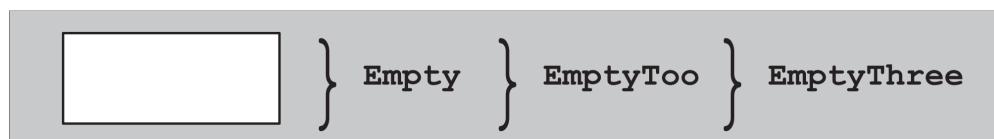


图 21.1. EmptyThree 的结构，由实现 EBCO 的编译器

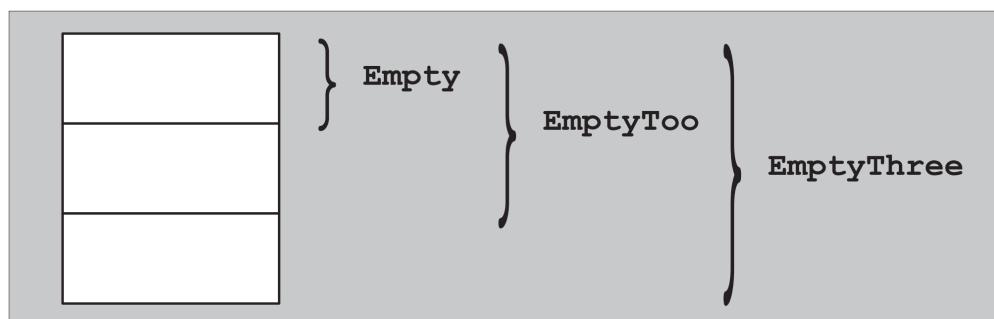


图 21.2. EmptyThree 的结构，由未实现 EBCO 的编译器

考虑遇到 EBCO 约束的情况：

inherit/ebco2.cpp

```

1 #include <iostream>
2
3 class Empty {
4     using Int = int; // type alias members don't make a class nonempty
5 };

```

```

6
7 class EmptyToo : public Empty {
8 };
9
10 class NonEmpty : public Empty, public EmptyToo {
11 };
12
13 int main()
14 {
15     std::cout << "sizeof(Empty): " << sizeof(Empty) << '\n';
16     std::cout << "sizeof(EmptyToo): " << sizeof(EmptyToo) << '\n';
17     std::cout << "sizeof(NonEmpty): " << sizeof(NonEmpty) << '\n';
18 }

```

NonEmpty 类不是一个空类！没有任何成员，基类也没有。但 NonEmpty 的基类 Empty 和 EmptyToo 不能分配到相同地址，这将导致 EmptyToo 的基类 Empty 与 NonEmpty 的基类 Empty 位于相同的地址。相同类型的两个子对象将以相同的偏移量结束，而这是 C++ 的对象布局规则不允许的。可以想象，Empty 基子对象中的一个放置在偏移量“0 字节”处，另一个放置在偏移量“1 字节”处，但是完整的 NonEmpty 对象仍然不能有 1 字节的大小，因为在两个 NonEmpty 对象的数组中，所以第一个元素的 Empty 子对象不能与第二个元素的 Empty 子对象在同一地址（参见图 21.3）。

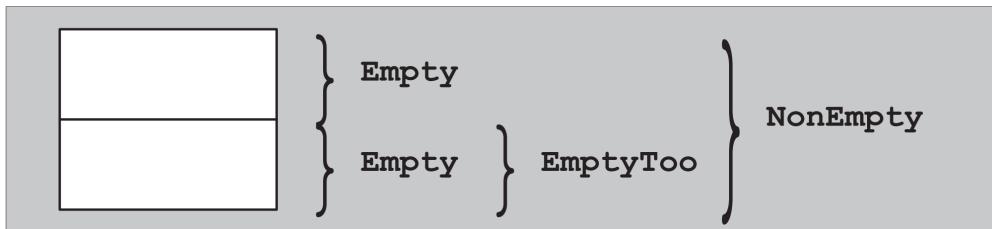


图 21.3. 实现 EBCO 的编译器的非空结构

对 EBCO 进行约束的基本原理源于这样一个事实，即能够比较两个指针是否指向同一个对象。指针在内部表示为地址，必须确保两个不同的地址（即指针值）对应两个不同的对象。

这个限制看起来可能不重要，但在实践中，经常会遇到这种情况，因为许多类倾向于使用一小组空类定义一些常见类型别名的空类继承。当此类中的两个子对象同时用于同一完整对象时，就抑制了优化。

即使有这个约束，EBCO 仍然是模板库的重要优化，因为许多技术依赖于基类引入，只是为了引入新类型别名或提供额外的功能，而不添加新数据。

21.1.2 基类的成员

EBCO 对于数据成员没有同等功能，因为（其他方面）会在成员指针的表示方面产生一些问题。因此，有时需要将成员变量实现为（私有）基类，这很有挑战性。

这个问题在模板上下文中最有趣，模板参数经常使用空类类型替换，但不能依赖于此规则。若对模板类型参数一无所知，就不能使用 EBCO。考虑一下下面这个例子：

```

1 template<typename T1, typename T2>
2 class MyClass {

```

```
3 private:
4 T1 a;
5 T2 b;
6 ...
7 };
```

完全有可能使用空类类型替代一个或两个模板参数。若是这样，那么 `MyClass<T1,T2>` 的表示可能是次优的，并且可能为每个 `MyClass<T1,T2>` 的实例浪费一个字符的内存。

可以通过将模板参数设置为基类来避免这种情况发生：

```
1 template<typename T1, typename T2>
2 class MyClass : private T1, private T2 {
3 };
```

这种简单的替代方法也存在一些问题：

- 当使用非类类型或联合类型替代 `T1` 或 `T2` 时，这种方式无效。
- 当两个参数替换为相同类型时，也不起作用 (可以通过添加另一层继承来解决)。
- 类可以是 `final` 类，在这种情况下，试图对它进行继承将导致编译错误。

即使圆满地解决了这些问题，另一个严重的问题仍然存在：添加基类会从根本上修改类的接口。对于 `MyClass` 类，这似乎不太重要，因为要影响的接口很少，但在后面会看到，从模板参数继承可以影响成员函数是否为虚函数。显然，这种使用 EBCO 的方法存在很多麻烦。

当模板参数只能由类类型替换，且类模板的另一个成员可用时，可以设计一种更实用的工具。主要思想是使用 EBCO，将可能为空的类型参数与其他成员“合并”。

```
1 template<typename CustomClass>
2 class Optimizable {
3   private:
4     CustomClass info; // might be empty
5     void* storage;
6   ...
7 };
```

模板实现者会使用以下语句：

```
1 template<typename CustomClass>
2 class Optimizable {
3   private:
4     BaseMemberPair<CustomClass, void*> info_and_storage;
5   ...
6 };
```

即使没有看到模板 `BaseMemberPair` 的实现，也会使 `Optimizable` 的实现更加冗长。然而，各种模板库实现者都认为，性能的提高 (对于他们库的客户端来说) 证明了增加复杂性的合理性。我们将在第 25.1.1 节中讨论元组时，进一步探讨这种习惯性用法。

`BaseMemberPair` 的实现可以很紧凑：

inherit/basememberpair.hpp

```

1 #ifndef BASE_MEMBER_PAIR_HPP
2 #define BASE_MEMBER_PAIR_HPP
3
4 template<typename Base, typename Member>
5 class BaseMemberPair : private Base {
6     private:
7         Member mem;
8
9     public:
10        // constructor
11        BaseMemberPair (Base const & b, Member const & m)
12            : Base(b), mem(m) {
13        }
14        // access base class data via base()
15        Base const& base() const {
16            return static_cast<Base const&>(*this);
17        }
18        Base& base() {
19            return static_cast<Base&>(*this);
20        }
21        // access member data via member()
22        Member const& member() const {
23            return this->mem;
24        }
25        Member& member() {
26            return this->mem;
27        }
28    };
29
30 #endif // BASE_MEMBER_PAIR_HPP

```

实现需要使用成员函数 `base()` 和 `member()` 来访问封装 (可能经过存储优化) 的数据成员。

21.2. 奇异递归模板模式 (CRTP)

另一个模式是奇异递归模板模式 (CRTP)。这个命名奇怪的模式指的是一种通用的技术类，包括将一个派生类作为模板参数传递给自己的基类。这种模式最简单的 C++ 代码如下所示：

```

1 template<typename Derived>
2 class CuriousBase {
3     ...
4 };
5
6 class Curious : public CuriousBase<Curious> {
7     ...
8 };

```

CRTP 的第一个概要展示了一个非依赖性基类：类 `Curious` 不是模板，因此不受依赖基类的一些名称可见性问题的影响。然而，这并不是 CRTP 的内在特征。完全可以使用以下备选方式：

```

1 template<typename Derived>
2 class CuriousBase {
3 ...
4 };
5
6 template<typename T>
7 class CuriousTemplate : public CuriousBase<CuriousTemplate<T>> {
8 ...
9 };

```

通过模板参数将派生类向下传递给基类，基类可以自定义派生类的行为，而不需要使用虚函数。这使得 CRTP 可以分解出只能是成员函数(例如，构造函数、析构函数和下标操作符)或依赖于派生类标识的实现。

CRTD 的简单应用包括跟踪创建了多少特定类类型的对象，这很容易通过在每个构造函数中递增一个整型静态数据成员，并在析构函数中递减来实现。但必须在每个类中提供这样的代码就很烦，并且通过单个(非 CRTD)基类实现此功能会混淆不同派生类的对象计数。不过，可以这样编写模板：

inherit/objectcounter.hpp

```

1 #include <cstddef>
2 template<typename CountedType>
3 class ObjectCounter {
4 private:
5     inline static std::size_t count = 0; // number of existing objects
6
7 protected:
8
9     // default constructor
10    ObjectCounter() {
11        ++count;
12    }
13
14    // copy constructor
15    ObjectCounter (ObjectCounter<CountedType> const&) {
16        ++count;
17    }
18
19    // move constructor
20    ObjectCounter (ObjectCounter<CountedType> &&) {
21        ++count;
22    }
23
24    // destructor
25    ~ObjectCounter() {
26        --count;
27    }
28
29 public:

```

```

30 // return number of existing objects:
31     static std::size_t live() {
32         return count;
33     }
34 };

```

使用内联是为了能够在类结构内部，定义和初始化 count 成员。C++17 前，必须在类模板外定义：

```

1 template<typename CountedType>
2 class ObjectCounter {
3     private:
4         static std::size_t count; // number of existing objects
5     ...
6 };
7
8 // initialize counter with zero:
9 template<typename CountedType>
10 std::size_t ObjectCounter<CountedType>::count = 0;

```

若想计算某个类类型的活(即未销毁)对象的数量，从 ObjectCounter 模板派生类就足够了。可以这样定义和使用用于计数的字符串类：

inherit/countertest.cpp

```

1 #include "objectcounter.hpp"
2 #include <iostream>
3
4 template<typename CharT>
5 class MyString : public ObjectCounter<MyString<CharT>> {
6     ...
7 };
8
9 int main()
10 {
11     MyString<char> s1, s2;
12     MyString<wchar_t> ws;
13     std::cout << "num of MyString<char>: "
14             << MyString<char>::live() << '\n';
15     std::cout << "num of MyString<wchar_t>: "
16             << ws.live() << '\n';
17 }

```

21.2.1 Barton-Nackman 技巧(友元工厂)

1994 年，John J. Barton 和 Lee R. Nackman 提出了一种模板技术，他们称之为限制模板展开(参见 [BartonNackman])。这种技术的部分动机是由于函数模板的重载在当时非常受限

为了了解函数模板重载在现代 C++ 中的工作原理，有必要阅读第 16.2 节。

并且在大多数编译器中无法使用命名空间。

假设一个类模板 Array，希望为其定义相等 operator==。一种可能是将操作符声明为类模板的成员，但这不是很好的实践方式，因为第一个参数（绑定到 this 指针）要遵守与第二个参数不同的转换规则。因为 operator== 参数对称，所以最好将其声明为命名空间作用域的函数。其实现方法大致如下所示：

```
1 template<typename T>
2 class Array {
3     public:
4     ...
5 };
6
7 template<typename T>
8 bool operator==(Array<T> const& a, Array<T> const& b)
9 {
10     ...
11 }
```

但若函数模板不能重载，就会出现一个问题：不能在该作用域中声明其他 operator== 模板，而其他类模板很可能需要这样的模板。Barton 和 Nackman 将类中的操作符定义为友元函数，解决了这个问题：

```
1 template<typename T>
2 class Array {
3     static bool areEqual(Array<T> const& a, Array<T> const& b);
4
5     public:
6     ...
7     friend bool operator==(Array<T> const& a, Array<T> const& b) {
8         return areEqual(a, b);
9     }
10 }
```

假设这个版本的 Array 实例化为 float 类型，友元操作符函数声明为该实例化的结果，但该函数本身不是函数模板的实例化，其只是普通的非模板函数，作为实例化过程的副作用注入到全局作用域中。因为是非模板函数，所以即使在向语言中添加函数模板重载之前，也可以使用 operator== 的其他声明重载。Barton 和 Nackman 将此称为限制模板展开，因为它避免使用适用于所有类型 T（换句话说，不受限制的展开）的模板 operator==(T, T)。

因为

```
1 operator==(Array<T> const&, Array<T> const&)
```

是在类定义内部定义的，其隐式地认为是一个内联函数，因此我们决定将实现委托给一个不需要内联的静态成员函数 areEqual。

自 1994 年以来，友元函数的名称查找已经发生了变化，因此 Barton-Nackman 技巧在标准 C++ 中并不那么有用。在它发明的时候，友元声明在类模板的外围作用域中可见，这个模板是通过一个

名为友元名称注入的过程进行实例化。标准 C++ 通过依赖于参数的查找来查找友元函数声明（具体细节请参阅第 13.2.2 节），从而函数调用的至少一个参数，必须已含友元函数作为关联类的类。若参数类的类型不相关，可以转换为包含友元的类，则无法找到友元函数。例如：

inherit/wrapper.cpp

```
1 class S {
2 };
3
4 template<typename T>
5 class Wrapper {
6     private:
7     T object;
8     public:
9     Wrapper(T obj) : object(obj) { // implicit conversion from T to Wrapper<T>
10    }
11    friend void foo(Wrapper<T> const&) {
12    }
13 };
14
15 int main()
16 {
17     S s;
18     Wrapper<S> w(s);
19     foo(w); // OK: Wrapper<S> is a class associated with w
20     foo(s); // ERROR: Wrapper<S> is not associated with s
21 }
```

这里，`foo(w)` 是有效的，因为函数 `foo()` 是在 `Wrapper<S>` 中声明的友元，它是一个与参数 `w` 相关联的类。

S 也是一个与 w 相关的类，因为它是 w 类型的模板参数。ADL 的具体规则在第 13.2.1 节中讨论。

在调用 `foo(s)` 中，因为定义它的类 `Wrapper<S>` 与类型 `S` 的参数 `s` 不相关，所以函数 `foo(Wrapper<S> const&)` 的友元声明不可见。即使存在有效的从类型 `s` 到类型 `Wrapper<S>` 的隐式转换（通过 `Wrapper<S>` 的构造函数），因为候选函数 `foo()` 在第一个地方没有找到，所以这种转换从未进行考虑。在 Barton 和 Nackman 发明他们的技巧时，友元名称注入将使友元函数 `foo()` 可见，并且调用 `foo(s)` 会成功。

现代 C++ 中，在类模板中定义友元函数比简单地定义普通函数模板的优势是语法：友元函数定义可以访问其外围类的私有和受保护成员，并且不需要重述外围类模板的所有模板参数。但友元函数定义在与 CRPT 结合使用时可能会更好，如下面一节描述的操作符实现所示。

21.2.2 实现操作符

实现提供重载操作符的类时，通常会为许多不同(但相关)的操作符提供重载。实现相等操作符($=$)的类，可能还会实现不等操作符(\neq)，而实现小于操作符($<$)的类，也可能实现其他关系操作符($>$, \leq , \geq)。这些操作符中只需要定义一个，而其他操作符可以使用这个操作符定义。类X的不等操作符可能定义为相等操作符：

```
1 bool operator!= (X const& x1, X const& x2) {
2     return !(x1 == x2);
3 }
```

考虑到有大量具有类似 \neq 定义的类型，很容易将其推广到模板中：

```
1 template<typename T>
2 bool operator!= (T const& x1, T const& x2) {
3     return !(x1 == x2);
4 }
```

C++标准库中也包含类似的定义，作为`<utility>`头文件的一部分。当确定它们在命名空间`std`中可用时会导致问题时，这些定义(对于 \neq 、 $>$ 、 \leq 和 \geq)在标准化期间降级到命名空间`std::rel_ops`中。实际上，让这些定义可见会使类型看起来都有一个 \neq 操作符(可能无法实例化)，并且该操作符对其两个参数总是精确匹配。虽然第一个问题可以通过使用SFINAE技术(请参阅第19.4节)来克服，这样 \neq 定义将只对具有适当的 $=$ 操作符的类型进行实例化，但第二个问题仍然存在：上面的 \neq 定义将优于用户提供的定义，这些定义需要进行派生到基的转换，这就很诡异了。

基于CRTP的这些操作符模板的另一种表达方式允许类选择通用操作符定义，提供了增加代码重用的优势，而没有过度通用操作符的副作用：

inherit/equalitycomparable.cpp

```
1 template<typename Derived>
2 class EqualityComparable
3 {
4     public:
5         friend bool operator!= (Derived const& x1, Derived const& x2) {
6             return !(x1 == x2);
7         }
8     };
9
10 class X : public EqualityComparable<X>
11 {
12     public:
13         friend bool operator== (X const& x1, X const& x2) {
14             // implement logic for comparing two objects of type X
15         }
16     };
17
18 int main()
19 {
20     X x1, x2;
```

```
21 if (x1 != x2) { }
22 }
```

这里，结合了 CRTP 和 Barton-Nackman 技巧!EqualityComparable<> 使用 CRTP 提供一个操作符!根据派生类对 operator== 的定义，为其指定派生类。实际上，通过友元函数定义 (Barton-Nackman 技巧) 提供了该定义，将两个参数提供给 operator!= 转换的等价行为。

当将行为分解到基类中同时，保留最终派生类的标识时，CRTP 可能很有用。除了 Barton-Nackman 技巧之外，CRTP 还可以基于一些规范操作符提供一些操作符的一般定义。这些特性使 Barton-Nackman 技巧的 CRTP，成为 C++ 模板库作者最喜欢的技术之一。

21.2.3 门面模式

使用 CRTP 和 Barton-Nackman 技巧来定义一些操作符是一种方便的快捷方式。可以进一步采用这种思想，这样 CRTP 基类就可以用 CRTP 派生类公开的一个小得多 (但更容易实现) 的接口来定义类的大部分或所有公共接口。这种模式称为门面模式，可以定义需要满足某些现有接口数字类型、迭代器、容器等要求的新类型。

为了说明门面模式，我们将为迭代器实现一个门面模式，这将简化编写 (符合标准库要求的) 迭代器的过程。迭代器类型 (特别是随机访问迭代器) 所需的接口相当大。下面是类模板 IteratorFacade 的框架，演示了迭代器接口的需求：

inherit/iteratorfacadeskel.hpp

```
1 template<typename Derived, typename Value, typename Category,
2         typename Reference = Value&, typename Distance = std::ptrdiff_t>
3 class IteratorFacade
4 {
5     public:
6     using value_type = typename std::remove_const<Value>::type;
7     using reference = Reference;
8     using pointer = Value*;
9     using difference_type = Distance;
10    using iterator_category = Category;
11
12    // input iterator interface:
13    reference operator *() const { ... }
14    pointer operator ->() const { ... }
15    Derived& operator ++() { ... }
16    Derived operator ++(int) { ... }
17    friend bool operator==(IteratorFacade const& lhs,
18                           IteratorFacade const& rhs) { ... }
19    ...
20
21    // bidirectional iterator interface:
22    Derived& operator --() { ... }
23    Derived operator --(int) { ... }
24
25    // random access iterator interface:
```

```

26 reference operator [](difference_type n) const { ... }
27 Derived& operator +=(difference_type n) { ... }
28 ...
29 friend difference_type operator -(IteratorFacade const& lhs,
30 IteratorFacade const& rhs) { ... }
31 friend bool operator <(IteratorFacade const& lhs,
32 IteratorFacade const& rhs) { ... }
33 ...
34 };

```

简洁起见，省略了一些声明，即使实现每个新迭代器列出的所有声明，也相当繁琐。幸运的是，该接口可以总结为几个核心操作：

- 对于所有迭代器：
 - dereference(): 访问迭代器指向的值 (通常通过操作符 * 和-> 使用)。
 - increment(): 移动迭代器指向序列中的下一项。
 - equals(): 确定两个迭代器是否指向序列中的同一项。
- 双向迭代器：
 - decrement(): 移动迭代器以指向列表中的前一项。
- 随机访问迭代器：
 - advance(): 将迭代器向前 (或向后) 移动 n 步。
 - measureDistance(): 确定序列中从一个迭代器到另一个迭代器的步数。

门面是调整一个只实现那些核心操作的类型，以提供完整的迭代器接口。IteratorFacade 的实现主要涉及将迭代器语法映射到最小的接口。下面的例子中，使用成员函数 asDerived() 来访问 CRTP 派生类：

```

1 Derived& asDerived() { return *static_cast<Derived*>(this); }
2 Derived const& asDerived() const {
3   return *static_cast<Derived const*>(this);
4 }

```

根据该定义，门面的大部分实现都很简单。

为了简化表示，忽略代理迭代器的存在，因为代理迭代器的解引用操作不会返回真正的引用。迭代器外观的完整实现 (如 [BoostIterator] 中的实现) 将调整操作符的结果类型-> 和操作符 [] 以考虑代理。

这里，只举例说明输入迭代器的一些要求的定义，其他迭代器的情况也相同。

```

1 reference operator*() const {
2   return asDerived().derefrence();
3 }
4 Derived& operator++() {
5   asDerived().increment();

```

```

6   return asDerived();
7 }
8 Derived operator++(int) {
9   Derived result(asDerived());
10  asDerived().increment();
11  return result;
12 }
13 friend bool operator==(IteratorFacade const& lhs,
14 IteratorFacade const& rhs) {
15   return lhs.asDerived().equals(rhs.asDerived());
16 }

```

定义链表迭代器

通过 IteratorFacade 的定义，可以将迭代器定义为简单的链表类。假设这样定义链表中的节点：

inherit/listnode.hpp

```

1 template<typename T>
2 class ListNode
3 {
4 public:
5   T value;
6   ListNode*>* next = nullptr;
7   ~ListNode() { delete next; }
8 };

```

使用 IteratorFacade，可以以一种简单的方式定义链表迭代器：

inherit/listnodeiterator0.hpp

```

1 template<typename T>
2 class ListNodeIterator
3 : public IteratorFacade<ListNodeIterator<T>, T,
4 std::forward_iterator_tag>
5 {
6   ListNode*>* current = nullptr;
7 public:
8   T& dereference() const {
9     return current->value;
10 }
11 void increment() {
12   current = current->next;
13 }
14 bool equals(ListNodeIterator const& other) const {
15   return current == other.current;
16 }
17 ListNodeIterator(ListNode*>* current = nullptr) : current(current) { }
18 };

```

ListNodeIterator 提供了充当前向迭代器所需的所有正确操作符和嵌套类型，并且只需要很少的代码来实现。定义更复杂的迭代器(例如，随机访问迭代器)只需要少量的工作。

隐藏接口

ListNodeIterator 实现的缺点是，需要作为公共接口公开操作 dereference()、advance() 和 equals()。为了消除这个需求，可以重写 IteratorFacade，通过单独的访问类 IteratorFacadeAccess，在派生的 CRTP 类上执行所有操作：

inherit/iteratorfacadeaccessskel.hpp

```
1 // 'friend' this class to allow IteratorFacade access to core iterator operations:
2 class IteratorFacadeAccess
3 {
4     // only IteratorFacade can use these definitions
5     template<typename Derived, typename Value, typename Category,
6         typename Reference, typename Distance>
7     friend class IteratorFacade;
8
9     // required of all iterators:
10    template<typename Reference, typename Iterator>
11    static Reference dereference(Iterator const& i) {
12        return i.dereference();
13    }
14    ...
15    // required of bidirectional iterators:
16    template<typename Iterator>
17    static void decrement(Iterator& i) {
18        return i.decrement();
19    }
20
21    // required of random-access iterators:
22    template<typename Iterator, typename Distance>
23    static void advance(Iterator& i, Distance n) {
24        return i.advance(n);
25    }
26    ...
27};
```

该类为每个核心迭代器操作提供静态成员函数，调用所提供的迭代器对应的(非静态)成员函数。所有的静态成员函数都是私有的，只授予 IteratorFacade 本身访问权。因此，ListNodeIterator 可以将 IteratorFacadeAccess 设为友元，并对门面所需的接口保持私有：

```
1 friend class IteratorFacadeAccess;
```

迭代器适配器

IteratorFacade 可以很容易地构建一个迭代器适配器，可接受一个现有的迭代器，并公开一个新的迭代器，以提供底层序列的转换视图。例如，可能有一个 Person 值的容器：

inherit/person.hpp

```
1 struct Person {
2     std::string firstName;
3     std::string lastName;
4
5     friend std::ostream& operator<<(std::ostream& strm, Person const& p) {
6         return strm << p.lastName << ", " << p.firstName;
7     }
8 };
```

但不需要遍历容器中的所有 Person 值，而只希望看到名字。本节中，将开发一个名为 ProjectionIterator 的迭代器适配器，会将底层(基)迭代器的值“投射”到一些指向数据的成员，例如 Person::firstName。

ProjectionIterator 是根据基本迭代器(iterator)和将由迭代器(T)公开值类型定义的迭代器：

inherit/projectioniteratorskel.hpp

```
1 template<typename Iterator, typename T>
2 class ProjectionIterator
3 : public IteratorFacade<
4     ProjectionIterator<Iterator, T>,
5     T,
6     typename std::iterator_traits<Iterator>::iterator_category,
7     T&,
8     typename std::iterator_traits<Iterator>::difference_type>
9 {
10     using Base = typename std::iterator_traits<Iterator>::value_type;
11     using Distance =
12         typename std::iterator_traits<Iterator>::difference_type;
13
14     Iterator iter;
15     T Base::* member;
16
17     friend class IteratorFacadeAccess;
18     ... // implement core iterator operations for IteratorFacade
19 public:
20     ProjectionIterator(Iterator iter, T Base::* member)
21         : iter(iter), member(member) { }
22 };
23
24 template<typename Iterator, typename Base, typename T>
25 auto project(Iterator iter, T Base::* member) {
26     return ProjectionIterator<Iterator, T>(iter, member);
27 }
```

每个投射迭代器存储两个值:iter 和 member，前者是底层序列(Base 值)的迭代器，后者是指向数据的成员，描述投射哪个成员。可以考虑为 IteratorFacade 的基类提供模板参数：第一个是

ProjectionIterator(用于启用 CRTP), 第二个 (T) 和第四个 (T&) 参数是投射迭代器的值和引用类型, 将其定义为一个由 T 值组成的序列。

为了简化示例, 假定底层迭代器返回的是引用, 而不是代理。

第三和第五个参数仅传递底层迭代器的类别和不同类型。当迭代器是输入迭代器时, 投射迭代器将是输入迭代器; 当 iterator 是双向迭代器时, 投射迭代器将是双向迭代器, 以此类推。project() 函数可以方便地构建投射迭代器。

唯一缺少的部分是 IteratorFacade 核心需求的实现。最有趣的是 dereference(), 它对底层迭代器进行解引用, 然后通过指向数据的指针成员进行投射:

```
1 T& dereference() const {
2     return (*iter).*member;
3 }
```

剩下的操作由底层迭代器实现:

```
1 void increment() {
2     ++iter;
3 }
4 bool equals(ProjectionIterator const& other) const {
5     return iter == other.iter;
6 }
7 void decrement() {
8     --iter;
9 }
```

简单起见, 这里省略了随机访问迭代器的定义, 其类似于下面的定义。

就是这样! 使用投射迭代器, 可以打印出 vector 中 Person 对象的名字:

inherit/projectioniterator.cpp

```
1 #include <vector>
2 #include <algorithm>
3 #include <iterator>
4
5 int main()
6 {
7     std::vector<Person> authors = { {"David", "Vandevoorde"}, 
8         {"Nicolai", "Josuttis"}, 
9         {"Douglas", "Gregor"} };
10    std::copy(project(authors.begin(), &Person::firstName),
11              project(authors.end(), &Person::firstName),
12              std::ostream_iterator<std::string>(std::cout, "\n"));
13 }
```

这个程序会输出:

David
Nicolai
Douglas

门面模式对于创建符合某些特定接口的新类型特别有用。新类型只需要向门面公开少量的核心操作 (门面迭代器的核心操作在 3 到 6 个之间)，门面负责使用 CRTP 和 Barton-Nackman 技巧的组合，并提供完整而正确的公共接口即可。

21.3. 混合类

简单的 Polygon 类，由一系列 Point 组成：

```
1 class Point
2 {
3     public:
4         double x, y;
5         Point() : x(0.0), y(0.0) { }
6         Point(double x, double y) : x(x), y(y) { }
7     };
8
9 class Polygon
10 {
11     private:
12         std::vector<Point> points;
13     public:
14         ... // public operations
15 };
```

若用户可以扩展与每个 Point 关联的信息，包括应用程序特定的数据，比如：每个 Point 的颜色，或者可能将标签与每个 Point 相关联。可以根据 Point 的类型参数化 Polygon，使这个扩展成为可能：

```
1 template<typename P>
2 class Polygon
3 {
4     private:
5         std::vector<P> points;
6     public:
7         ... // public operations
8 };
```

用户可以使用继承创建自己的 Point 数据类型，提供与 Point 相同的接口，但包含其他应用程序特定的数据：

```
1 class LabeledPoint : public Point
2 {
3     public:
4         std::string label;
```

```
5     LabeledPoint() : Point(), label("") { }
6     LabeledPoint(double x, double y) : Point(x, y), label("") { }
7 }
```

这种实现有其缺点。首先，要求向用户公开 Point 类型，以便用户可以从它派生。此外，LabeledPoint 的作者需要小心地提供与 Point 完全相同的接口（例如，继承或提供与 Point 相同的所有构造函数），否则 LabeledPoint 将无法与 Polygon 一起工作。若 Point 从一个版本的 Polygon 模板更改为另一个，这个约束就会有问题：添加新的 Point 构造函数可能需要更新每个派生类。

混合类提供了另一种方法来定制类型的行为，而不继承。混合类本质上颠倒了继承的方向，因为新类作为类模板的基类“混合”到继承层次中，而不是作为新的派生类创建。这种方法允许引入新的数据成员和其他操作，而不需要复制任何接口。

支持混合的类模板通常会接受任意数量的类，并从中派生：

```
1 template<typename... Mixins>
2 class Point : public Mixins...
3 {
4     public:
5     double x, y;
6     Point() : Mixins()..., x(0.0), y(0.0) { }
7     Point(double x, double y) : Mixins()..., x(x), y(y) { }
8 }
```

现在，可以“混合”一个包含标签的基类来生成一个 LabeledPoint：

```
1 class Label
2 {
3     public:
4     std::string label;
5     Label() : label("") { }
6 };
7
8 using LabeledPoint = Point<Label>;
```

或者混合几个基类：

```
1 class Color
2 {
3     public:
4     unsigned char red = 0, green = 0, blue = 0;
5 };
6
7 using MyPoint = Point<Label, Color>;
```

有了这个基于混合的 Point，在不改变接口的情况下，可以很容易地向 Point 引入其他信息，所以 Polygon 很容易使用和扩展。用户只需要将特化的 Point 隐式转换到自定义混合类（上面的 Label 或 Color），就可以访问该数据或接口。此外，Point 类可以隐藏，并混合提供给 Polygon 类模板本身：

```
1 template<typename... Mixins>
2 class Polygon
3 {
```

```
4 private:
5     std::vector<Point<Mixins...>> points;
6 public:
7     ... // public operations
8 };
```

模板需要进行少量定制的情况下，混合类很有用——比如适用用户指定的数据存储内部对象——而不需要库公开和记录这些内部数据类型及其接口。

21.3.1 奇怪的混合类

通过将混合类与第 21.2 节描述 CRTP 相结合，混合类可以更强大。每个混合实际上都是一个类模板，将与派生类的类型一起提供，从而可以对派生类进行定制。一个 CRTP-混合版本的 Point 可以这样：

```
1 template<template<typename>... Mixins>
2 class Point : public Mixins<Point>...
3 {
4     public:
5     double x, y;
6     Point() : Mixins<Point>(..., x(0.0), y(0.0)) { }
7     Point(double x, double y) : Mixins<Point>(..., x(x), y(y)) { }
8 };
```

公式需要对要混合的每个类做更多工作，因此像 Label 和 Color 这样的类要成为类模板。但混合类可以根据混合到的派生类特定实例，调整其行为。可以将前面讨论的 ObjectCounter 模板混合到 Point 中，以计算 Polygon 创建的点的数量，并将该混合类与其他类混合在一起。

21.3.2 参数化的虚拟性

混合类还允许间接参数化派生类的其他属性，例如成员函数的虚函数。一个简单的例子展示了这种奇特的技巧：

inherit/virtual.cpp

```
1 #include <iostream>
2
3 class NotVirtual {
4 };
5
6 class Virtual {
7     public:
8     virtual void foo() {
9     }
10 };
11
12 template<typename... Mixins>
13 class Base : public Mixins... {
```

```

14 public:
15 // the virtuality of foo() depends on its declaration
16 // (if any) in the base classes Mixins...
17 void foo() {
18     std::cout << "Base::foo()" << '\n';
19 }
20 };
21
22 template<typename... Mixins>
23 class Derived : public Base<Mixins...> {
24 public:
25     void foo() {
26         std::cout << "Derived::foo()" << '\n';
27     }
28 };
29
30 int main()
31 {
32     Base<NotVirtual>* p1 = new Derived<NotVirtual>;
33     p1->foo(); // calls Base::foo()
34
35     Base<Virtual>* p2 = new Derived<Virtual>;
36     p2->foo(); // calls Derived::foo()
37 }

```

这种技术可以提供一种方式来设计类模板，该模板既可用于实例化具体类，也可用于使用继承进行扩展。然而，在一些成员函数上使用虚拟性来获得一个类还不够，这个类可以作为更好的基类来实现更特化的功能。这种开发方法需要更基本的设计决策，因此设计两个不同的工具（类或类模板架构）通常比将全部集成到一个模板结构中更为实用。

21.4. 命名模板参数

模板技术有时会导致类模板具有许多不同的模板类型参数，这些参数中的许多都有合理的默认值。定义此类类模板的方法如下：

```

1 template<typename Policy1 = DefaultPolicy1,
2         typename Policy2 = DefaultPolicy2,
3         typename Policy3 = DefaultPolicy3,
4         typename Policy4 = DefaultPolicy4>
5 class BreadSlicer {
6     ...
7 };

```

这样的模板通常可以使用 `BreadSlicer<>` 语法与默认模板参数值一起使用，若必须指定非默认参数，那么前面的参数也必须指定（即使可能有默认值）。

显然，若能够使用类似于 `BreadSlicer<Policy3 = Custom>` 的构造，而不是使用 `BreadSlicer<DefaultPolicy1, DefaultPolicy2, Custom>` 的构造，将会更有吸引力。接下来，我们开发了一种技术来实现这一点。

注意，类似的函数调用参数的语言扩展在 C++ 标准化过程中提出过（但被拒绝）（详见第 17.4 节）。

技术包括将默认类型值放在基类中，并通过派生进行重写。通过辅助类提供类型参数，而不是直接指定类型参数，可以使用 `BreadSlicer<Policy3_is<Custom>>`。因为每个模板参数可以描述任何策略，默认值相同。换句话说，每个模板参数都等价：

```
1 template<typename PolicySetter1 = DefaultPolicyArgs,
2         typename PolicySetter2 = DefaultPolicyArgs,
3         typename PolicySetter3 = DefaultPolicyArgs,
4         typename PolicySetter4 = DefaultPolicyArgs>
5 class BreadSlicer {
6     using Policies = PolicySelector<PolicySetter1, PolicySetter2,
7                           PolicySetter3, PolicySetter4>;
8     // use Policies::P1, Policies::P2, ... to refer to the various policies
9     ...
10 };
```

剩下的挑战是编写 `PolicySelector` 模板，必须将不同的模板参数合并为一个类型，该类型用指定的非默认值覆盖默认类型别名成员。这种合并可以通过继承实现：

```
1 // PolicySelector<A,B,C,D> creates A,B,C,D as base classes
2 // Discriminator<> allows having even the same base class more than once
3 template<typename Base, int D>
4 class Discriminator : public Base {
5 };
6
7 template<typename Setter1, typename Setter2,
8          typename Setter3, typename Setter4>
9 class PolicySelector : public Discriminator<Setter1,1>,
10                      public Discriminator<Setter2,2>,
11                      public Discriminator<Setter3,3>,
12                      public Discriminator<Setter4,4> {
13 };
```

注意中间 `Discriminator` 模板的使用，需要允许各种 `Setter` 是相同的类型。（同一类型不能有多个直接基类。另一方面，间接基类可以具有与其他基类相同的类型。）

如前所述，可以在基类中收集默认值：

```
1 // name default policies as P1, P2, P3, P4
2 class DefaultPolicies {
3     public:
4         using P1 = DefaultPolicy1;
5         using P2 = DefaultPolicy2;
6         using P3 = DefaultPolicy3;
7         using P4 = DefaultPolicy4;
8 };
```

若多次继承这个基类，则必须小心避免歧义。因此，需要确保基类是虚继承的：

```
1 // class to define a use of the default policy values
2 // avoids ambiguities if we derive from DefaultPolicies more than once
3 class DefaultPolicyArgs : virtual public DefaultPolicies {
4 };
```

最后，还需要一些模板来覆盖默认策略值：

```
1 template<typename Policy>
2 class Policy1_is : virtual public DefaultPolicies {
3     public:
4         using P1 = Policy; // overriding type alias
5     };
6
7 template<typename Policy>
8 class Policy2_is : virtual public DefaultPolicies {
9     public:
10        using P2 = Policy; // overriding type alias
11    };
12
13 template<typename Policy>
14 class Policy3_is : virtual public DefaultPolicies {
15     public:
16        using P3 = Policy; // overriding type alias
17    };
18
19 template<typename Policy>
20 class Policy4_is : virtual public DefaultPolicies {
21     public:
22        using P4 = Policy; // overriding type alias
23    };
```

有了这些，预期目标就实现了。下面例化一个 BreadSlicer<>：

```
1 BreadSlicer<Policy3_is<CustomPolicy>> bc;
```

BreadSlicer<>，类型 Policies 定义为

```
1 PolicySelector<Policy3_is<CustomPolicy>,
2     DefaultPolicyArgs,
3     DefaultPolicyArgs,
4     DefaultPolicyArgs>
```

在 Discriminator<> 类模板的帮助下，这会产生一个架构，其中所有模板参数都是基类(参见图 21.4)。重点是这些基类

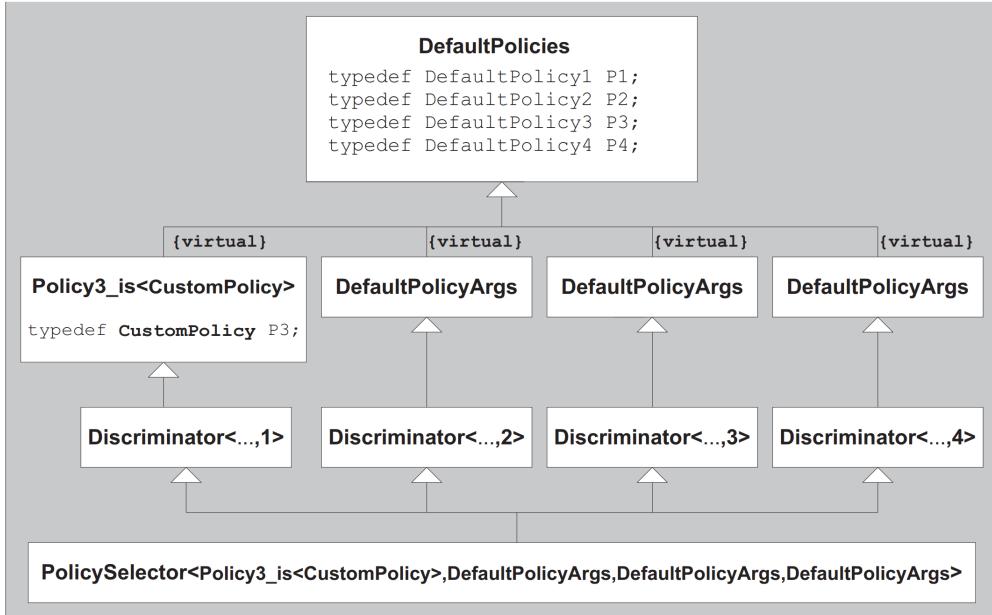


图 21.4. BreadSlicer<...>::Policies 具有架构的结果类型

都有相同的虚基类 DefaultPolicies，定义了 P1、P2、P3 和 P4 的默认类型，但 P3 在一个派生类中重新定义——即在 Policy3_is<> 中定义。根据控制规则，这个定义隐藏了基类的定义，因此没有歧义。

可以在第一个 C++ 标准的 10.2/6 节（参见 [C++98]）中找到控制规则，并在 [EllisStroustrupARM] 的 10.1.1 节中对它进行了讨论。

在 BreadSlicer 模板中，可以使用诸如 Policies::P3 这样的限定名称来引用这四个策略。例如：

```

1 template<...>
2 class BreadSlicer {
3 ...
4 public:
5     void print () {
6         Policies::P3::doPrint();
7     }
8 ...
9 };

```

在 inherit/namedtmpl.cpp 中可以找到整个示例。

我们开发了用于四个模板类型参数的技术，显然可以扩展到任何合理数量的此类参数，我们从未实例化包含虚基类的辅助类对象。因此，虚基类并不会导致性能或内存消耗的问题。

21.5. 后记

Bill Gibbons 是将 EBCO 引入 C++ 编程语言的主要发起人。Nathan Myers 使它流行起来，并提出了一个类似于 BaseMemberPair 的模板，从而可以更好地利用它。Boost 库包含相当复杂的模板，

称为 `compressed_pair`, 其解决了我们在本章中 `MyClass` 模板的一些问题。`boost::compressed_pair` 也可以用来代替 `BaseMemberPair`。

CRTP 至少从 1991 年开始使用。然而, James Coplien 是第一个将它们正式描述为模式的人(参见 [CoplienCRTP])。从那时起, RTP 的许多应用已经公开发表, 参数化继承有时错误地等同于 RTP。RTP 根本不要求对派生进行参数化, 而且许多形式的参数化继承不符合 RTP。因为 Barton 和 Nackman 技巧经常将 RTP 与友元名注入结合使用(后者是 Barton-Nackman 技巧的重要组成部分), 所以 RTP 有时会与 Barton-Nackman 技巧相混淆(参见第 21.2.1 节)。使用 RTP 和 Barton-Nackman 技巧来提供操作符实现, 与 Boost.Operators 库 ([BoostOperators]) 使用相同的实现方法, 提供了一组通用的操作符定义。类似地, 我们对迭代器门面的处理与 Boost.Iterator 库 ([BoostIterator]) 类似, 它为派生类型提供了丰富的、符合标准库的迭代器接口, 派生类型提供了一些核心迭代器操作(相等、解引用、移动), 还解决了涉及代理迭代器的棘手问题(为了保证示例的简洁, 我们在这里没有解决这个问题)。我们的 `ObjectCounter` 示例与 Scott Meyers 在 [MeyersCounting] 中使用的技术完全相同。

至少从 1986 年 ([MoonFlavors]) 开始, 混合的概念就出现在面向对象编程中, 作为一种将小块功能引入 OO 类的方式。在第一个 C++ 标准发布后不久, 在 C++ 中使用混合类模板就开始流行起来, 两篇论文 ([SmaragdakisBatoryMixins] 和 [EiseneckerBlinnCzarnecki]) 描述了目前常用的混合类方法。从那时起, 混合类成为 C++ 库设计中的一种流行技术。

命名模板参数用于简化 Boost 库中的某些类模板。Boost 使用元编程创建具有与 `PolicySelector` 类似属性的类型(但不使用虚继承), 这里介绍的更简单的替代方案是由 Vandevoorde 编写。

第 22 章 桥接静态和动态多态性

第 18 章描述了 C++ 中静态多态(通过模板)和动态多态(通过继承和虚函数)。这两种多态为编写程序提供了强大的抽象能力，但也有各自的缺点：静态多态提供了与非多态代码相同的性能，可以在运行时使用的类型集在编译时需要已知。另一方面，通过继承的动态多态性允许多态函数在编译时处理不确定的类型，但是因为类型必须从公共基类继承，所以灵活性较低。

本章介绍了如何在 C++ 中搭建静态和动态多态性之间的桥梁，在 18.3 节中讨论了每个模型中优点：更小的可执行代码大小和(几乎)完全编译的动态多态性的特性，以及静态多态性的接口灵活性，允许内置类型无缝衔接的工作。作为示例，我们将构建标准库 `function<>` 模板的简化版本。

22.1. 函数对象、指针和 `std::function<>`

函数对象对于为模板提供可定制的行为，下面的函数模板枚举从 0 到某个值的整数值，并将每个值提供给给定的函数对象 `f`：

bridge/forupto1.cpp

```
1 #include <vector>
2 #include <iostream>
3
4 template<typename F>
5 void forUpTo(int n, F f)
6 {
7     for (int i = 0; i != n; ++i)
8     {
9         f(i); // call passed function f for i
10    }
11 }
12
13 void printInt(int i)
14 {
15     std::cout << i << ' ';
16 }
17
18 int main()
19 {
20     std::vector<int> values;
21
22     // insert values from 0 to 4:
23     forUpTo(5,
24             [&values](int i) {
25                 values.push_back(i);
26             });
27
28     // print elements:
29     forUpTo(5,
30             printInt); // prints 0 1 2 3 4
```

```
31     std::cout << '\n' ;
32 }
```

`forUpTo()` 函数模板可以与函数对象一起使用，包括 Lambda、函数指针、函数操作符或转换为函数指针/引用的类，每次使用 `forUpTo()` 都可能产生不同的实例化。示例函数模板非常小，但若模板很大，这些实例化可能会增加代码量。

限制代码量增加的方法，是将函数模板转换为不需要实例化的非模板。可以尝试使用函数指针：

bridge/forupto2.hpp

```
1 void forUpTo(int n, void (*f)(int))
2 {
3     for (int i = 0; i != n; ++i)
4     {
5         f(i); // call passed function f for i
6     }
7 }
```

虽然这个实现在传递 `printInt()` 时正常工作，但在传递 Lambda 时将产生错误：

```
1 forUpTo(5,
2 printInt); // OK: prints 0 1 2 3 4
3
4 forUpTo(5,
5     [&values](int i) { // ERROR: lambda not convertible to a function pointer
6         values.push_back(i);
7     });
8 }
```

标准库的类模板 `std::function<>` 允许使用 `forUpTo()` 的替代表达式：

bridge/forupto3.hpp

```
1 #include <functional>
2 void forUpTo(int n, std::function<void(int)> f)
3 {
4     for (int i = 0; i != n; ++i)
5     {
6         f(i); // call passed function f for i
7     }
8 }
```

`std::function<>` 的模板参数是一个函数类型，描述了函数对象将接收的参数类型和应该产生的返回类型，就像函数指针描述参数和结果类型一样。

`forUpTo()` 提供了静态多态的一些方面——使用合适的函数操作符处理一组无界类型的能力，包括函数指针、Lambda 和类——而其本身仍然是非模板函数，只有一个实现。可以使用称为类型擦除的技术来实现，这种技术弥合了静态和动态多态之间的差异。

22.2. 广义函数指针

`std::function<>` 类型实际上是 C++ 函数指针的泛化形式，提供了相同的基本操作：

- 可以用于调用函数，而调用者不需要知道函数的信息。
- 可以复制、移动和赋值。
- 可以从另一个函数（带有兼容的签名）初始化或赋值。
- 有一个“空”状态，表示没有函数已绑定。

与 C++ 函数指针不同的是，`std::function<>` 还可以使用合适的函数操作符存储 Lambda 或其他函数对象，所有这些操作符都可能具有不同的类型。

本节的剩余部分，将构建自己的通用函数指针类模板——`FunctionPtr`，需要提供这些核心操作和功能，并用来替代 `std::function`：

bridge/forupto4.cpp

```
1 #include "functionptr.hpp"
2 #include <vector>
3 #include <iostream>
4
5 void forUpTo(int n, FunctionPtr<void(int)> f)
6 {
7     for (int i = 0; i != n; ++i)
8     {
9         f(i); // call passed function f for i
10    }
11 }
12
13 void printInt(int i)
14 {
15     std::cout << i << ' ';
16 }
17
18 int main()
19 {
20     std::vector<int> values;
21
22     // insert values from 0 to 4:
23     forUpTo(5,
24             [&values](int i) {
25                 values.push_back(i);
26             });
27
28     // print elements:
29     forUpTo(5,
30             printInt); // prints 0 1 2 3 4
31     std::cout << '\n';
32 }
```

FunctionPtr 的接口相当简单，提供函数对象的构造、复制、移动、销毁、初始化和赋值，以及底层函数对象的调用。该接口最有趣的部分是，如何在类模板偏特化中进行描述，其将模板参数（一个函数类型）分解为其组件（结果类型和参数类型）：

bridge/functionptr.hpp

```
1 // primary template:
2 template<typename Signature>
3 class FunctionPtr;
4
5 // partial specialization:
6 template<typename R, typename... Args>
7 class FunctionPtr<R(Args...)>
8 {
9     private:
10    FunctorBridge<R, Args...>* bridge;
11
12    public:
13    // constructors:
14    FunctionPtr() : bridge(nullptr) {
15    }
16    FunctionPtr(FunctionPtr const& other); // see functionptr-cpinv.hpp
17    FunctionPtr(FunctionPtr& other)
18        : FunctionPtr(static_cast<FunctionPtr const&>(other)) {
19    }
20    FunctionPtr(FunctionPtr&& other) : bridge(other.bridge) {
21        other.bridge = nullptr;
22    }
23    // construction from arbitrary function objects:
24    template<typename F> FunctionPtr(F&& f); // see functionptr-init.hpp
25
26    // assignment operators:
27    FunctionPtr& operator= (FunctionPtr const& other) {
28        FunctionPtr tmp(other);
29        swap(*this, tmp);
30        return *this;
31    }
32    FunctionPtr& operator= (FunctionPtr&& other) {
33        delete bridge;
34        bridge = other.bridge;
35        other.bridge = nullptr;
36        return *this;
37    }
38    // construction and assignment from arbitrary function objects:
39    template<typename F> FunctionPtr& operator= (F&& f) {
40        FunctionPtr tmp(std::forward<F>(f));
41        swap(*this, tmp);
42        return *this;
43    }
```

```

44 // destructor:
45 ~FunctionPtr() {
46     delete bridge;
47 }
48
49 friend void swap(FunctionPtr& fp1, FunctionPtr& fp2) {
50     std::swap(fp1.bridge, fp2.bridge);
51 }
52 explicit operator bool() const {
53     return bridge != nullptr;
54 }
55
56 // invocation:
57 R operator()(Args... args) const; // see functionptr-cpinv.hpp
58 };

```

该实现包含非静态成员变量 `bridge`, 负责存储和操作存储的函数对象。这个指针的所有权与 `FunctionPtr` 对象绑定, 因此大多数实现仅去管理这个指针。未实现的函数包含实现中的部分, 将在下面的小节中进行描述。

22.3. 桥接接口

`FunctorBridge` 类模板负责底层函数对象的所有权和操作, 实现为一个抽象基类, 是 `FunctionPtr` 动态多态性的基础:

bridge/functorbridge.hpp

```

1 template<typename R, typename... Args>
2 class FunctorBridge
3 {
4     public:
5     virtual ~FunctorBridge() {
6     }
7     virtual FunctorBridge* clone() const = 0;
8     virtual R invoke(Args... args) const = 0;
9 };

```

`FunctorBridge` 提供了通过虚函数操作存储函数对象所需的基本操作: 析构函数、执行复制的 `clone()` 操作和调用。操作调用基础函数对象。不要忘记将 `clone()` 和 `invoke()` 定义为 `const` 成员函数。

将 `invoke()` 设为 `const`, 为的是避免通过 `const` 函数 ptr 对象调用非 `const` 操作符 () 重载, 这与开发者的预期不符。

使用这些虚函数, 可以实现 `FunctionPtr` 的复制构造函数和函数调用操作符:

bridge/functionptr-cpinv.hpp

```

1 template<typename R, typename... Args>
2 FunctionPtr<R(Args...)>::FunctionPtr(FunctionPtr const& other)
3 : bridge(nullptr)
4 {
5     if (other.bridge) {
6         bridge = other.bridge->clone();
7     }
8 }
9
10 template<typename R, typename... Args>
11 R FunctionPtr<R(Args...)>::operator()(Args... args) const
12 {
13     return bridge->invoke(std::forward<Args>(args)...);
14 }

```

22.4. 类型擦除

FunctorBridge 的每个实例都是抽象类，因此其派生类负责提供虚函数的实现。为了支持潜在函数对象的完整范围 (无界集合)，需要无界的派生类。通过对派生类存储的函数对象类型，进行参数化来实现：

bridge/specifcfunctorbridge.hpp

```

1 template<typename Functor, typename R, typename... Args>
2 class SpecificFunctorBridge : public FunctorBridge<R, Args...> {
3     Functor functor;
4
5     public:
6         template<typename FunctorFwd>
7             SpecificFunctorBridge(FunctorFwd&& functor)
8                 : functor(std::forward<FunctorFwd>(functor)) {
9
10         virtual SpecificFunctorBridge* clone() const override {
11             return new SpecificFunctorBridge(functor);
12         }
13         virtual R invoke(Args... args) const override {
14             return functor(std::forward<Args>(args)...);
15         }
16     };

```

每个 SpecificFunctorBridge 实例存储一个函数对象的副本 (类型是 Functor)，可以调用、复制或销毁 (析构函数中隐式进行)。当一个 FunctionPtr 初始化为新的函数对象时，特定的 functorbridge 实例将创建，完成 FunctionPtr 示例：

bridge/functionptr-init.hpp

```

1 template<typename R, typename... Args>
2 template<typename F>

```

```

3 FunctionPtr<R(Args...)>::FunctionPtr(F&& f)
4 : bridge(nullptr)
5 {
6     using Functor = std::decay_t<F>;
7     using Bridge = SpecificFunctorBridge<Functor, R, Args...>;
8     bridge = new Bridge(std::forward<F>(f));
9 }

```

虽然 FunctionPtr 构造函数在函数对象类型 F 上模板化的，但该类型只有特定的 SpecificFunctorBridge 特化才知晓（由 Bridge 类型别名描述）。当为新 Bridge 实例分配数据成员 bridge，因为类型从 Bridge * 到 FunctorBridge<R, Args...> *，所以关于特定类型 F 的信息将丢失。

虽然类型可以通过 `dynamic_cast` 查询，FunctionPtr 类使 bridge 指针私有化，所以 FunctionPtr 的外部代码不能访问类型本身。

这种类型信息的丢失解释了，为什么使用术语类型擦除，来描述静态和动态多态之间的桥接技术。

该实现的特点是使用 `std::decay`（请参阅第 D.4 节）来产生 Functor 类型，这使得推导类型 F 适合存储，例如：通过将对函数类型的引用转换成函数指针类型，并移除 `const`、`volatile` 和引用类型的限定。

22.5. 可选桥接

FunctionPtr 模板可以直接替代函数指针，但还不支持函数指针提供的一个操作：测试两个 FunctionPtr 对象是否会调用同一个函数。添加这样的操作需要用等号操作更新 FunctorBridge：

```

1 virtual bool equals(FunctorBridge const* fb) const = 0;

```

以及 SpecificFunctorBridge 中的实现，当具有相同类型时比较存储的函数对象：

```

1 virtual bool equals(FunctorBridge<R, Args...> const* fb) const override {
2     if (auto specFb = dynamic_cast<SpecificFunctorBridge const*>(fb)) {
3         return functor == specFb->functor;
4     }
5     // functors with different types are never equal:
6     return false;
7 }

```

最后，为 FunctionPtr 实现 `operator==` 它首先检查空函数，然后委托给 FunctorBridge：

```

1 friend bool
2 operator==(FunctionPtr const& f1, FunctionPtr const& f2) {
3     if (!f1 || !f2) {
4         return !f1 && !f2;
5     }
6     return f1.bridge->equals(f2.bridge);
7 }
8 friend bool

```

```

9 operator!=(FunctionPtr const& f1, FunctionPtr const& f2) {
10    return !(f1 == f2);
11 }

```

这个实现正确，但它有一个缺点：若给 FunctionPtr 赋值或初始化函数对象没有合适的 operator==(例如，包含 Lambda)，程序将无法编译。因为 FunctionPtrsd 还没有使用 operator==，而且许多其他类模板（如 std::vector）可以用没有 operator== 的类型实例化，只要不使用 operator== 就好。

operator== 的问题由类型擦除导致，因为当对 FunctionPtr 赋值或初始化，就会丢失函数对象的类型。因此需要在赋值或初始化完成前，捕获关于该类型所需的所有信息。这些信息包括对函数对象的 operator== 的调用，所以不能确定什么时候需要使用该操作符。

机制上，调用 operator== 的代码会实例化，因为类模板的所有虚函数（本例中是 SpecificFunctorBridge）通常在类模板本身实例化时实例化。

这里可以使用基于 SFINAE 的特征（第 19.4 节中讨论），在调用 operator== 之前，使用一个复杂的特征查询是否可用：

bridge/isequalitycomparable.hpp

```

1 #include <utility> // for declval()
2 #include <type_traits> // for true_type and false_type
3
4 template<typename T>
5 class IsEqualityComparable
6 {
7     private:
8         // test convertibility of == and != to bool:
9         static void* conv(bool); // to check convertibility to bool
10        template<typename U>
11        static std::true_type test decltype(conv(std::declval<U const&>()) ==
12                                     std::declval<U const&>()),
13                                     decltype(conv(!std::declval<U const&>() ==
14                                     std::declval<U const&>()))
15        );
16
17     // fallback:
18     template<typename U>
19     static std::false_type test(...);
20
21     public:
22     static constexpr bool value = decltype(test<T>(nullptr,
23                                               nullptr))::value;
24 };

```

IsEqualityComparable 特征应用了表达式测试特征的形式，如第 19.4.1 节所介绍的：两个 test() 重载，其中一个包含用 decltype 包装的测试表达式，另一个通过省略号接受任意参数。第一个 test() 函数尝试使用 == 比较两个 T 类型的 const 对象，并确保结果可以隐式转换为 bool（对于第一个参数）

并传递给逻辑否定操作符 `operator!`, 结果可转换为 `bool`。如果两个操作都定义良好，则参数类型都为 `void*`。

使用 `IsEqualityComparable` 特征，可以构造 `TryEquals` 类模板，对给定类型调用 `==`(当该类型可用时)，或者在没有合适的 `==` 存在时抛出异常：

bridge/tryequals.hpp

```
1 #include <exception>
2 #include "isequalitycomparable.hpp"
3
4 template<typename T,
5         bool EqComparable = IsEqualityComparable<T>::value>
6 struct TryEquals
7 {
8     static bool equals(T const& x1, T const& x2) {
9         return x1 == x2;
10    }
11 };
12
13 class NotEqualityComparable : public std::exception
14 {
15 };
16
17 template<typename T>
18 struct TryEquals<T, false>
19 {
20     static bool equals(T const& x1, T const& x2) {
21         throw NotEqualityComparable();
22     }
23 };
```

最后，通过在 `SpecificFunctorBridge` 的实现中使用 `TryEquals`，当存储的函数对象类型匹配且函数对象支持 `==` 操作时，就能够在 `FunctionPtr` 中提供 `==` 操作：

```
1 virtual bool equals(FunctorBridge<R, Args...> const* fb) const override {
2     if (auto specFb = dynamic_cast<SpecificFunctorBridge const*>(fb)) {
3         return TryEquals<Functor>::equals(functor, specFb->functor);
4     }
5     // functors with different types are never equal:
6     return false;
7 }
```

22.6. 性能考量

类型擦除提供了静态多态和动态多态的一些优点，使用类型擦除生成的代码的性能更接近于动态多态，因为两者都通过虚函数使用动态分配。因此，静态多态的一些传统优势，比如：编译器内联调用的能力，可能会失去。这种性能损失是否明显取决于具体应用，但通常可以通过调用函数中

的工作量，相对于虚函数调用的耗时来判断：若两者接近（使用 FunctionPtr 简单地将两个整数相加），类型擦除的执行速度可能比静态多态版本慢得多。另一方面，若函数调用执行大量的工作——查询数据库、对容器进行排序或更新用户接口——则类型擦除的开销可能无法评估。

22.7. 后记

Kevlin Henney 在 C++ 中推广了类型擦除，引入了 any 类型 [HenneyValuedConversions]，该类型后来成为 Boost 库 [BoostAny]，并成为 C++17 标准的一部分。在 Boost.Function[BoostFunction] 中对该技术进行了一些改进，其应用了各种性能和代码量优化，最终成为 std::function<>。然而，每一个早期库只处理一组操作：any 是一个简单的值类型，只有一个复制和强制转换操作；函数添加了调用。

之后的工作，比如 Boost.TypeErasure 库 [BoostTypeErasure] 和 Adobe 的 Poly 库 [AdobePoly]，使用模板元编程技术，允许用户形成具有特定功能列表的类型擦除值。例如，下面的类型（使用 Boost.TypeErasure 库）处理复制构造、类类型操作和打印输出流：

```
1 using AnyPrintable = any<mpl::vector<copy_constructible<>,
2                         typeid_<>,
3                         ostreamable<>
4                         >>;
```

第 23 章 元编程

元编程由“程序编程”构成，也就是我们编写编程系统执行的代码，以生成实现真正想要的功能的新代码。通常，术语元编程意味着一种自反属性：元编程组件是程序的一部分，其可生成一段代码（即，程序的另一段或不同的代码）。

为什么使用元编程？与大多数其他编程技术一样，目标是用更少的工作实现更多的功能，其中的工作可以通过代码大小、维护成本等来衡量。元编程的特点是在转换时产生一些用户定义的计算。潜在的动机通常是性能（转换时计算的东西通常可以优化掉）或接口简单（元程序通常比它扩展的要短）或两者兼有。

元编程通常依赖于特征和类型函数的概念，如第 19 章所述。因此，建议在研究这一章之前先熟悉一下第 19 章。

23.1. 现代 C++ 的元编程

C++ 元编程技术随着时间的推移而发展（本章末尾的附注回顾了这一领域的里程碑），我们讨论并分类现代 C++ 中常用的各种元编程方法。

23.1.1 值元编程

本书的第一版中，我们限制在最初的 C++ 标准中引入的特性上（1998 年发布，2003 年进行了很小的修改）。那个世界里，编写简单的编译时（“元”）计算是一个挑战，我们在本章中花了大量的时间来研究这个问题；一个相当高级的示例，在编译时使用递归模板实例化计算整数值的平方根。正如 8.2 节中介绍的，在 C++11 中，尤其是 C++14，通过引入 `constexpr` 函数，降低了挑战的难度。

C++11 的 `constexpr` 功能足以解决许多常见的挑战，但编程模型并不总是令人满意的（例如，没有循环语句，所以迭代计算必须利用递归函数调用；参见 23.2 节）。C++14 启用了循环语句和其他构造方式。

C++14 后，计算平方根的编译时函数很容易写成这样：

meta/sqrtconstexpr.hpp

```
1 template<typename T>
2 constexpr T sqrt(T x)
3 {
4     // handle cases where x and its square root are equal as a special case to simplify
5     // the iteration criterion for larger x:
6     if (x <= 1) {
7         return x;
8     }
9     // repeatedly determine in which half of a [lo, hi] interval the square root of x is located,
10    // until the interval is reduced to just one value:
11    T lo = 0, hi = x;
12    for (;;) {
```

```

13     auto mid = (hi+lo)/2, midSquared = mid*mid;
14     if (lo+1 >= hi || midSquared == x) {
15         // mid must be the square root:
16         return mid;
17     }
18     // continue with the higher/lower half-interval:
19     if (midSquared < x) {
20         lo = mid;
21     }
22     else {
23         hi = mid;
24     }
25 }
26 }
```

该算法通过对已知包含 x 平方根的区间反复二分来寻找答案(根号 0 和 1 为特殊情况, 以保持收敛准则简单)。这个 `sqrt()` 函数可以在编译或运行时求值:

```

1 static_assert(sqrt(25) == 5, ""); // OK (evaluated at compile time)
2 static_assert(sqrt(40) == 6, ""); // OK (evaluated at compile time)
3
4 std::array<int, sqrt(40)+1> arr; // declares array of 7 elements (compile time)
5 long long l = 53478;
6 std::cout << sqrt(l) << '\n'; // prints 231 (evaluated at run time)
```

这个函数的实现现在运行时可能不是最高效的(运行时利用机器的特性通常会有很好的性能回报), 但因为执行的是编译时计算, 所以绝对效率不如可移植性重要。在这个平方根示例中没有高级的“魔术模板”, 只有函数模板的常见模板参数推导。这里的代码是“普通的 C++”, 读起来并不困难。

上面所做的是值元编程(即对编译时值的计算进行编程), 但还有两种元编程可以用现代 C++ 执行(比如 C++14 和 C++17): 类型元编程和混合元编程。

23.1.2 类型元编程

第 19 章讨论某些特征模板时, 已经遇到了一种类型计算的形式, 其以一个类型作为输入, 并由此产生一个新类型。`RemoveReferenceT` 类模板计算引用类型的基础类型, 但在第 19 章中开发的示例只计算了相当基本的类型操作。通过依赖递归模板实例化(基于模板的元编程的支柱), 可以执行相当复杂的类型计算。

看看下面的例子:

meta/removeallextents.hpp

```

1 // primary template: in general we yield the given type:
2 template<typename T>
3 struct RemoveAllExtentsT {
4     using Type = T;
5 };
```

```

7 // partial specializations for array types (with and without bounds):
8 template<typename T, std::size_t SZ>
9 struct RemoveAllExtentsT<T[SZ]> {
10     using Type = typename RemoveAllExtentsT<T>::Type;
11 };
12 template<typename T>
13 struct RemoveAllExtentsT<T[]> {
14     using Type = typename RemoveAllExtentsT<T>::Type;
15 };
16
17 template<typename T>
18 using RemoveAllExtents = typename RemoveAllExtentsT<T>::Type;

```

RemoveAllExtents 是一个类型元函数(即生成结果类型的计算设备), 将从类型中移除任意数量的顶层“数组层”, 可以这样使用它:

```

1 RemoveAllExtents<int[]> // yields int
2 RemoveAllExtents<int[5][10]> // yields int
3 RemoveAllExtents<int[]> // yields int
4 RemoveAllExtents<int(*)[5]> // yields int(*)[5]

```

C++ 标准库提供了相应的类型特征 `std::remove_all_extents`。详见第 D.4 节。

元函数通过让匹配顶层数组情况的偏特化, 递归地“调用”元函数本身来执行其任务。

若能够使用的值都是标量值, 那么使用值进行计算将非常有限。不过, 编程语言都有一个值容器构造, 可以增强该语言的能力(而且大多数语言都有各种容器类型, 如数组/vector、哈希表等)。类型元编程也是如此: 添加“类型容器”构造大大增加了规程的适用性。现代 C++ 包含了支持开发这种容器的机制。第 24 章开发了一个 `Typelist<>` 类模板, 就是这样的类型容器。

23.1.3 混合元编程

通过值和类型元编程, 可以在编译时计算值和类型。但我们最终对运行时效果感兴趣, 所以在需要类型和常量的地方在运行时代码中使用元编程。然而, 元编程可以做的不止这些: 可以在编译时以编程方式汇编代码位, 并具有运行时效果。我们称之为混合元编程。

为了说明这一原理, 先从一个简单的例子开始: 计算两个 `std::array` 值的点积。回想一下, `std::array` 是固定长度的容器模板, 声明如下:

```

1 namespace std {
2     template<typename T, size_t N> struct array;
3 }

```

其中 `N` 是数组中类型为 `T` 的元素个数。给定两个相同数组类型的对象, 其点积可以这样计算:

```

1 template<typename T, std::size_t N>
2 auto dotProduct(std::array<T, N> const& x, std::array<T, N> const& y)
3 {
4     T result{};

```

```

5   for (std::size_t k = 0; k < N; ++k) {
6       result += x[k] * y[k];
7   }
8   return result;
9 }
```

直接编译 for 循环将产生分支指令，与直线执行相比，在某些机器上可能会造成一些开销

```

1 result += x[0] * y[0];
2 result += x[1] * y[1];
3 result += x[2] * y[2];
4 result += x[3] * y[3];
5 ...
```

现代编译器会将循环优化为对目标平台最有效的形式。为了便于讨论，可以以一种避免循环的方式重写 dotProduct() 实现：

这称为循环展开。我们通常不建议在可移植代码中显式展开循环，因为决定最佳展开策略的细节高度依赖于目标平台和循环体；编译器通常会更好地考虑这些因素。

```

1 template<typename T, std::size_t N>
2 struct DotProductT {
3     static inline T result(T* a, T* b) {
4         return *a * *b + DotProduct<T, N-1>::result(a+1, b+1);
5     }
6 };
7
8 // partial specialization as end criteria
9 template<typename T>
10 struct DotProductT<T, 0> {
11     static inline T result(T*, T*) {
12         return T{};
13     }
14 };
15
16 template<typename T, std::size_t N>
17 auto dotProduct(std::array<T, N> const& x,
18                 std::array<T, N> const& y)
19 {
20     return DotProductT<T, N>::result(x.begin(), y.begin());
21 }
```

这个新实现将工作委托给一个类模板 DotProductT，使我们能够使用递归模板实例化和类模板偏特化来结束递归。DotProductT 的每次实例化，会产生点积和数组其他组件的点积。对于类型 `std::array<T, N>`，因此将有 `N` 个主模板实例和一个终止偏特化的实例。为了提高效率，编译器必须每次调用内联静态成员函数 `result()`。即使启用了中等级别的编译器优化，这也是正确的。

我们在这里显式地指定 `inline` 关键字，是因为一些编译器（尤其是 Clang）将此作为提示，以尝试更努力地进行内联调用。从语言的角度来看，这些函数隐式内联，因为它们在其外围类的体中定义。

这段代码的主要特点是，混合了确定代码整体结构的编译时计算（通过递归模板实例化实现）和确定特定运行时效果的运行时计算（调用 `result()`）。

前面提到过，由于“类型容器”的可用性，类型元编程得到了极大的增强。混合元编程中，固定长度的数组类型很有用。尽管如此，混合元编程真正的“英雄容器”是元组。元组是一个值序列，每个值都有一个可选择的类型。C++ 标准库包含一个支持该概念的 `std::tuple` 类模板。例如，

```
1 std::tuple<int, std::string, bool> tVal{42, "Answer", true};
```

定义了一个变量 `tVal`，它聚合了三个类型的值：`int`、`std::string` 和 `bool`（按照特定的顺序）。由于类元组容器在现代 C++ 编程中的重要性，我们将在第 25 章中详细开发一个容器。上面的 `tVal` 类型非常类似于一个简单的结构类型：

```
1 struct MyTriple {
2     int v1;
3     std::string v2;
4     bool v3;
5 };
```

鉴于在 `std::array` 和 `std::tuple` 中，有数组类型和（简单）结构类型的灵活对应，很自然地想知道简单联合类型的对应是否也对混合计算有用：答案是肯定的。C++ 标准库在 C++17 中为此引入了 `std::variant` 模板，我们在第 26 章开发了一个类似的组件。

因为与 `struct` 一样，`std::tuple` 和 `std::variant` 都是异构类型，因此使用此类类型的混合元编程有时也称为异构元编程。

23.1.4 单元类型的混合元编程

另一个展示混合计算能力的例子是库，能够计算不同单元类型的值的结果。值计算在运行时执行，但结果元计算在编译时确定。

用一个高度简化的例子来说明，用它们占主单位的比例（分数）来表示单位。时间的主要单位是秒，则毫秒用比例 $1/1000$ 表示，分钟用比例 $60/1$ 表示。关键在于定义比率类型，其中每个值都有自己的类型：

meta/ratio.hpp

```
1 template<unsigned N, unsigned D = 1>
2 struct Ratio {
3     static constexpr unsigned num = N; // numerator
4     static constexpr unsigned den = D; // denominator
5     using Type = Ratio<num, den>;
6 };
```

现在可以定义编译时计算，比如添加两个单元：

meta/ratioadd.hpp

```
1 // implementation of adding two ratios:
2 template<typename R1, typename R2>
3 struct RatioAddImpl
4 {
5     private:
6     static constexpr unsigned den = R1::den * R2::den;
7     static constexpr unsigned num = R1::num * R2::den + R2::num * R1::den;
8     public:
9     typedef Ratio<num, den> Type;
10 };
11
12 // using declaration for convenient usage:
13 template<typename R1, typename R2>
14 using RatioAdd = typename RatioAddImpl<R1, R2>::Type;
```

在编译时计算两个比率的总和:

```
1 using R1 = Ratio<1,1000>;
2 using R2 = Ratio<2,3>;
3 using RS = RatioAdd<R1,R2>; // RS has type Ratio<2003,2000>
4 std::cout << RS::num << ' / ' << RS::den << '\n' ; // prints 2003/3000
5
6 using RA = RatioAdd<Ratio<2,3>,Ratio<5,7>>; // RA has type Ratio<29,21>
7 std::cout << RA::num << ' / ' << RA::den << '\n' ; // prints 29/21
```

现在可以为 duration 定义一个类模板，参数化为任意值类型和一个单位类型，该单位类型是 Ratio<> 的实例:

meta/duration.hpp

```
1 // duration type for values of type T with unit type U:
2 template<typename T, typename U = Ratio<1>>
3 class Duration {
4     public:
5     using ValueType = T;
6     using UnitType = typename U::Type;
7     private:
8     ValueType val;
9     public:
10    constexpr Duration(ValueType v = 0)
11        : val(v) {
12    }
13    constexpr ValueType value() const {
14        return val;
15    }
16};
```

有趣的是定义了一个加法操作符来加和两个 duration:

meta/durationadd.hpp

```
1 // adding two durations where unit type might differ:  
2 template<typename T1, typename U1, typename T2, typename U2>  
3 auto constexpr operator+(Duration<T1, U1> const& lhs,  
4             Duration<T2, U2> const& rhs)  
5 {  
6     // resulting type is a unit with 1 a nominator and  
7     // the resulting denominator of adding both unit type fractions  
8     using VT = Ratio<1,RatioAdd<U1,U2>::den>;  
9     // resulting value is the sum of both values  
10    // converted to the resulting unit type:  
11    auto val = lhs.value() * VT::den / U1::den * U1::num +  
12        rhs.value() * VT::den / U2::den * U2::num;  
13    return Duration<decltype(val), VT>(val);  
14 }
```

允许参数有不同的单位类型，U1 和 U2。使用这些单位类型来计算结果持续时间，从而得到一个对应的单位分数(分子为1的分数)的单位类型。有了所有这些，可以编译以下代码：

```
1 int x = 42;  
2 int y = 77;  
3  
4 auto a = Duration<int, Ratio<1,1000>>(x); // x milliseconds  
5 auto b = Duration<int, Ratio<2,3>>(y); // y 2/3 seconds  
6 auto c = a + b; // computes resulting unit type 1/3000 seconds  
7     // and generates run-time code for c = a*3 + b*2000
```

关键的“混合”效果是，对于 **sum c**，编译器在编译时确定结果单元类型 **Ratio**<1,3000>，并生成代码在运行时计算结果值，该值会根据结果单元类型进行调整。

因为值类型是模板参数，所以可以将 Duration 类用于 int 以外的值类型，甚至可以使用异构值类型(只要定义了这些类型的值如何相加):

```
1 auto d = Duration<double, Ratio<1,3>>(7.5); // 7.5 1/3 seconds  
2 auto e = Duration<int, Ratio<1>>(4); // 4 seconds  
3  
4 auto f = d + e; // computes resulting unit type 1/3 seconds  
5     // and generates code for f = d + e*3
```

此外，若值在编译时已知，编译器甚至可以在编译时执行值计算，因为持续时间的 **operator+** 是 **constexpr**。

C++ 标准库类模板 std::chrono 使用了这种方法，并进行了一些改进，例如使用预定义的单元(例如，std::chrono::milliseconds)，支持字面时间段(例如，10ms)，以及如何处理溢出。

23.2. 反射元编程的维数

前面描述了基于 `constexpr` 计算的值元编程和基于递归模板实例化的类型元编程，两项在现代 C++ 中都有使用，涉及不同的方法来驱动计算。值元编程也可以通过递归模板实例化来驱动，C++11 中引入 `constexpr` 函数之前，这是首选的机制。下面的代码使用递归实例化计算整数的平方根：

bridge/sqrt1.hpp

```
1 // primary template to compute sqrt(N)
2 template<int N, int LO=1, int HI=N>
3 struct Sqrt {
4     // compute the midpoint, rounded up
5     static constexpr auto mid = (LO+HI+1)/2;
6     // search a not too large value in a halved interval
7     static constexpr auto value = (N<mid*mid) ? Sqrt<N,LO,mid-1>::value
8     : Sqrt<N,mid,HI>::value;
9 };
10
11 // partial specialization for the case when LO equals HI
12 template<int N, int M>
13 struct Sqrt<N,M,M> {
14     static constexpr auto value = M;
15 };
```

这个元程序使用了与第 23.1.1 节中整数平方根 `constexpr` 函数使用相同的算法，连续将已知包含平方根的区间减半。元函数的输入是非类型模板参数，跟踪区间边界的“局部变量”也改为非类型模板参数。显然，这是一种远不如 `constexpr` 函数友好的方法，但我们仍会分析这段代码，以检查它如何使用编译器的资源。

我们可以看到元编程的计算引擎可能有很多选择，这并不是可以考虑的唯一方面。相反，我们倾向于选择 C++ 元编程解决方案必须在以下三个方面做出权衡：

- 计算
- 反射
- 生成

反射是一种以编程方式检查程序特性的能力。生成是指为程序生成代码的能力。

已经看到了两个计算选项：递归实例化和 `constexpr` 计算，我们在类型特征中找到了部分解决方案（参见第 19.6.1 节）。尽管可用的特性支持相当多的高级模板技术，但远不能涵盖语言中反射功能所需的功能。给定一个类类型，应用程序希望以编程的方式探索该类的成员。当前的特性基于模板实例化，C++ 可以提供语言工具或“固有”库组件

C++ 标准库中提供的一些特性已经依赖于编译器的某些合作（通过非标准的“固有”操作符）。参见 19.10 节。

生成在编译时包含反射信息的类模板实例，这种方法适合基于递归模板实例化的计算。但类模板实例占用大量编译器存储空间，而这些存储空间直到编译结束时才会释放（否则会导致编译时间

大大增加)。另一种选择是引入一种新的标准类型来表示反射信息，该选择有望与“计算”维度的 `constexpr` 评估选项配对。17.9 节讨论了这个选项 (C++ 标准化委员会正在积极调研)。

第 17.9 节还展示了实现强大代码生成的未来方法。在现有 C++ 语言中创建一个灵活的、通用的、开发者友好的代码生成机制，仍然是一个挑战。实例化模板是一种“代码生成”机制，编译器在扩展对内联小函数的调用方面已经足够可靠，这种机制可以用作代码生成的载体。这些观察结果正是上面 `DotProductT` 示例的基础，结合更强大的反射功能，现有技术已经可以完全实现元编程。

23.3. 递归实例化的代价

看下 23.2 节中介绍的 `Sqrt<>` 模板，主模板是一般的递归计算，使用模板参数 `N`(用于计算平方根的值)和其他两个可选参数调用。这些可选参数表示结果可以拥有的最小值和最大值。若模板只调用一个参数，并且我们已知平方根至少是 1，所以结果最多是值本身。

然后递归使用二分搜索技术(通常称为二分法)。模板内部，计算值是在 `LO` 和 `HI` 之间的范围的前半部分，还是后半部分，这种情况使用三元操作符来区分。若二分的中间值大于 `N`，我们继续上半部分的搜索。若二分的中间值小于等于 `N`，对第二部分使用相同的模板。当 `LO` 和 `HI` 具有相同的值 `M`(即最终值)时，偏特化结束递归。

模板实例化并不简单：即使是相对普通的类模板，也可以为每个实例分配超过 1 千字节的存储空间，并且在编译完成之前不能回收这些存储空间。因此，可以来研究一下使用 `Sqrt` 模板的编程细节：

meta/sqrt1.cpp

```
1 #include <iostream>
2 #include "sqrt1.hpp"
3
4 int main()
5 {
6     std::cout << "Sqrt<16>::value = " << Sqrt<16>::value << '\n' ;
7     std::cout << "Sqrt<25>::value = " << Sqrt<25>::value << '\n' ;
8     std::cout << "Sqrt<42>::value = " << Sqrt<42>::value << '\n' ;
9     std::cout << "Sqrt<1>::value = " << Sqrt<1>::value << '\n' ;
10 }
```

表达式

```
1 Sqrt<16>::value
```

可扩展为

```
1 Sqrt<16,1,16>::value
```

模板内部，元程序计算 `Sqrt<16,1,16>::value`:

```
1 mid = (1+16+1)/2
2     = 9
3 value = (16<9*9) ? Sqrt<16,1,8>::value
4                 : Sqrt<16,9,16>::value
```

```

5     = (16<81) ? Sqrt<16,1,8>::value
6       : Sqrt<16,9,16>::value
7     = Sqrt<16,1,8>::value

```

计算结果为 `Sqrt<16,1,8>::value`, 展开如下:

```

1 mid = (1+8+1)/2
2   = 5
3 value = (16<5*5) ? Sqrt<16,1,4>::value
4       : Sqrt<16,5,8>::value
5     = (16<25) ? Sqrt<16,1,4>::value
6       : Sqrt<16,5,8>::value
7     = Sqrt<16,1,4>::value

```

`Sqrt<16,1,4>::value` 分解如下:

```

1 mid = (1+4+1)/2
2   = 3
3 value = (16<3*3) ? Sqrt<16,1,2>::value
4       : Sqrt<16,3,4>::value
5     = (16<9) ? Sqrt<16,1,2>::value
6       : Sqrt<16,3,4>::value
7     = Sqrt<16,3,4>::value

```

最后, `Sqrt<16,3,4>::value` 的结果如下:

```

1 mid = (3+4+1)/2
2   = 4
3 value = (16<4*4) ? Sqrt<16,3,3>::value
4       : Sqrt<16,4,4>::value
5     = (16<16) ? Sqrt<16,3,3>::value
6       : Sqrt<16,4,4>::value
7     = Sqrt<16,4,4>::value

```

因为匹配捕获相等的上下界的显式特化, 所以 `Sqrt<16,4,4>::value` 结束递归进程。最终的结果是

```

1 value = 4

```

23.3.1 跟踪所有实例化

上面的分析计算 16 的平方根, 当编译器计算表达式时

```

1 (16<=8*8) ? Sqrt<16,1,8>::value
2       : Sqrt<16,9,16>::value

```

不仅实例化正分支中的模板, 还实例化负分支中的模板 (`Sqrt<16,9,16>`)。此外, 由于代码试图使用`::`操作符访问结果类类型的成员, 该类类型内的所有成员也会实例化。这意味着 `Sqrt<16,9,16>` 的完全实例化会导致 `Sqrt<16,9,12>` 和 `Sqrt<16,13,16>` 的完全实例化。当详细检查整个过程时, 会发现生成了几十个实例, 总数是 N 值的两倍。

有一些技术可以减少实例化数量的激增。为了介绍这样的技术, 这里重写 `Sqrt` 元程序:

meta/sqrt2.hpp

```
1 #include "ifthenelse.hpp"
2
3 // primary template for main recursive step
4 template<int N, int LO=1, int HI=N>
5 struct Sqrt {
6     // compute the midpoint, rounded up
7     static constexpr auto mid = (LO+HI+1)/2;
8
9     // search a not too large value in a halved interval
10    using SubT = IfThenElse<(N<mid*mid),
11        Sqrt<N, LO, mid-1>,
12        Sqrt<N, mid, HI>>;
13    static constexpr auto value = SubT::value;
14 };
15
16 // partial specialization for end of recursion criterion
17 template<int N, int S>
18 struct Sqrt<N, S, S> {
19     static constexpr auto value = S;
20 };
```

关键变化是 IfThenElse 模板的使用，该模板 19.7.1 节中介绍过。IfThenElse 模板是一个基于给定布尔常数，在两种类型之间进行选择的工具。若该常量为真，则第一个类型被类型别名为 type；否则，Type 代表第二种类型。并且，为类模板实例定义类型别名，不会导致 C++ 编译器实例化该实例体。因此，当代码如下时

```
1 using SubT = IfThenElse<(N<mid*mid),
2             Sqrt<N, LO, mid-1>,
3             Sqrt<N, mid, HI>>;
```

Sqrt<N,LO,mid-1> 和 Sqrt<N,mid,HI> 都没有完全实例化。查找 SubT::value 时，这两种类型中的一种最终成为 SubT 的同义词会进行完全实例化。与我们的第一种方法相反，这种策略导致了与 $\log_2(N)$ 成比例的大量实例化：当 N 变得相当大时，元编程的成本降低了。

23.4. 计算完整性

Sqrt<> 示例演示了模板元程序可以有：

- 状态变量：模板参数
- 循环构造：通过递归
- 执行路径选择：通过使用条件表达式或特化
- 整数运算

若不限制递归实例化的数量和允许状态变量的数量，这足以进行任何计算，但使用模板可能不方便这样做。此外，由于模板实例化需要大量的编译器资源，大量的递归实例化会降低编译器的处

理速度，甚至耗尽可用的资源。C++ 标准建议（但不是强制）至少允许 1024 级递归实例化，这对于大多数（但肯定不是所有）模板元编程任务来说已经足够了。

实践中，模板元程序应该有节制地使用。作为实现方便模板的工具，其不可替代。特别是，有时可以隐藏在模板内部，以便从关键算法实现中挤出更多的性能。

23.5. 递归实例化与递归模板参数

考虑以下递归模板：

```
1 template<typename T, typename U>
2 struct Doublify {
3 };
4
5 template<int N>
6 struct Trouble {
7     using LongType = Doublify<typename Trouble<N-1>::LongType,
8         typename Trouble<N-1>::LongType>;
9 };
10
11 template<>
12 struct Trouble<0> {
13     using LongType = double;
14 };
15
16 Trouble<10>::LongType ouch;
```

使用 `Trouble<10>::LongType` 不仅触发了 `Trouble<9>`, `Trouble<8>`, ..., `Trouble<0>` 的递归实例化，但其复杂性在 `Doublify` 的实例化处。表 23.1 说明了扩展增长的速度。

类型别名	基本类型
<code>Trouble<0>::LongType</code>	<code>double</code>
<code>Trouble<1>::LongType</code>	<code>Doublify<double,double></code>
<code>Trouble<2>::LongType</code>	<code>Doublify<Doublify<double,double>, Doublify<double,double>></code>
<code>Trouble<3>::LongType</code>	<code>Doublify<Doublify<Doublify<double,double>, Doublify<double,double>>, <Doublify<double,double>, Doublify<double,double>>></code>

表 23.1. `Trouble<N>::LongType` 实例化的增长

从表 23.1 可以看出，表达式 `Trouble<N>::LongType` 的类型描述的复杂性会随着 `N` 的增加呈指数增长。通常，这种情况对 C++ 编译器的压力，甚至比不涉及递归模板实参的递归实例化更大。这里的问题是，编译器为类型保留了损坏名称。这个混乱的名称以某种方式编码了模板特化，早期的

C++ 实现使用的编码大致与模板标识的长度成比例。这些编译器为 `Trouble<10>::LongType` 使用了超过 10,000 个字符的内存。

较新的 C++ 实现考虑到，嵌套模板标识在现代 C++ 程序中相当常见，并使用巧妙的压缩技术来大幅减少名称编码的增长（例如，`Trouble<10>::LongType` 需要几百个字符）。因为没有为模板实例生成低层代码，这些编译器还会避免在不需要的情况下生成混乱的名称。尽管如此，在其他条件都相同的情况下，以模板参数不需要递归嵌套的方式组织递归实例化可能是更好的方法。

23.6. 枚举值与静态常量

早期的 C++，枚举值是在类声明中创建“真正的常量”（称为常量表达式）作为命名成员的唯一机制，可以定义一个 Pow3 元程序来计算 3 的幂：

meta/pow3enum.hpp

```
1 // primary template to compute 3 to the Nth
2 template<int N>
3 struct Pow3 {
4     enum { value = 3 * Pow3<N-1>::value };
5 }
6
7 // full specialization to end the recursion
8 template<>
9 struct Pow3<0> {
10     enum { value = 1 };
11 }
```

C++98 的标准化引入了类内静态常量初始化器的概念，因此 Pow3 元程序可以这样写：

meta/pow3const.hpp

```
1 // primary template to compute 3 to the Nth
2 template<int N>
3 struct Pow3 {
4     static int const value = 3 * Pow3<N-1>::value;
5 }
6
7 // full specialization to end the recursion
8 template<>
9 struct Pow3<0> {
10     static int const value = 1;
11 }
```

这个版本有一个缺点：静态常量成员是左值（参见附录 B）

```
1 void foo(int const&);
```

将元程序的结果进行传递：

```
1 foo(Pow3<7>::value);
```

编译器必须传递 Pow3<7>::value 的地址，这迫使编译器实例化和分配静态成员的定义，计算不再局限于纯粹的“编译时”效应。

枚举值不是左值(没有地址)，当通过引用传递时，没有使用静态内存。就像将计算值作为文字传递一样。因此，本书的第一版在这类应用程序中更倾向于使用枚举常量。

然而，C++11 引入了 `constexpr` 静态数据成员，而且这些成员不限于整型。它们没有解决上面提出的地址问题，尽管有这个缺点，不过目前是生成元程序结果的常用方法。其优点是具有正确的类型(与手动 `enum` 类型相反)，并且当使用 `auto` 类型说明符声明静态成员时，可以推导出该类型。C++17 添加了内联静态数据成员，这确实解决了上面提出的地址问题，并且可以与 `constexpr` 一起使用。

23.7. 后记

最早记录元程序的例子是 Erwin Unruh，当时他在 C++ 标准化委员会中作为西门子的代表。他注意到模板实例化过程的计算完整性，并通过开发第一个元程序证明了他的观点。他使用 Metaware 编译器并诱导它发出包含连续素数的错误消息。以下是在 1994 年 C++ 委员会会议上流传的代码(经过修改，现在可以在符合标准的编译器上编译)：

meta/unruh.cpp

```
1 // prime number computation
2 // (modified from original from 1994 by Erwin Unruh)
3
4 template<int p, int i>
5 struct is_prime {
6     enum { pri = (p==2) || ((p%i) && is_prime<(i>2?p:0),i-1>::pri) };
7 }
8
9 template<>
10 struct is_prime<0,0> {
11     enum {pri=1};
12 }
13
14 template<>
15 struct is_prime<0,1> {
16     enum {pri=1};
17 }
18
19 template<int i>
20 struct D {
21     D(void*);
22 }
23
24 template<int i>
25 struct CondNull {
26     static int const value = i;
27 }
```

```

28 template<>
29 struct CondNull<0> {
30     static void* value;
31 };
32 void* CondNull<0>::value = 0;
33
34 template<int i>
35 struct Prime_print { // primary template for loop to print prime numbers
36     Prime_print<i-1> a;
37     enum { pri = is_prime<i,i-1>::pri };
38     void f() {
39         D<i> d = CondNull<pri ? 1 : 0>::value; // 1 is an error, 0 is fine
40         a.f();
41     }
42 };
43
44 template<>
45 struct Prime_print<1> { // full specialization to end the loop
46     enum {pri=0};
47     void f() {
48         D<1> d = 0;
49     }
50 };
51
52 #ifndef LAST
53 #define LAST 18
54 #endif
55
56 int main()
57 {
58     Prime_print<LAST> a;
59     a.f();
60 }

```

若编译这个程序，当在 `Prime_print::f()` 中，`d` 的初始化失败时，编译器将打印错误消息。当初始值为 1 时就会发生这种情况，因为只有 `void*` 的构造函数，只有 0 有到 `void*` 的有效转换。在某个编译器上，我们得到(其他几个消息中)以下错误信息：

```

unruh.cpp:39:14: error: no viable conversion from 'const int' to 'D<17>'
unruh.cpp:39:14: error: no viable conversion from 'const int' to 'D<13>'
unruh.cpp:39:14: error: no viable conversion from 'const int' to 'D<11>'
unruh.cpp:39:14: error: no viable conversion from 'const int' to 'D<7>'
unruh.cpp:39:14: error: no viable conversion from 'const int' to 'D<5>'
unruh.cpp:39:14: error: no viable conversion from 'const int' to 'D<3>'
unruh.cpp:39:14: error: no viable conversion from 'const int' to 'D<2>'

```

由于编译器中的错误处理不同，有些编译器可能会在打印第一个错误消息后停止。

C++ 模板元编程作为一种编程工具的概念，是由 Todd Veldhuizen 在他使用 C++ 模板元程序的论文 (参见 [VeldhuizenMeta95]) 中首次发表 (并在某种程度上形式化) 的。Todd 在 Blitz++(C++ 的数字数组库，参见 [Blitz++]) 方面的工作还对元编程 (以及表达式模板技术，在第 27 章中介绍) 进行了许多改进和扩展。

本书的第一版和 Andrei Alexandrescu 的《现代 C++ 设计》(参见 [AlexandrescuDesign]) 通过对至今仍在使用的基本技术进行编码，促进了利用基于模板的元编程的 C++ 库的爆发 (Boost 项目 (参见 [Boost]) 控制了这次爆发的场面)。早期的这些技术引入了 MPL(元编程库)，为类型元编程定义了一致的框架，这种框架也通过 Abrahams 和 Gurtovoy 的书《C++ 模板元编程》(参见 [BoostMPL]) 而流行起来。

Louis Dionne 在使元编程在语法上更容易访问方面取得了其他重要进展，特别是通过他的 Boost.Hana 库 (参见 [boosthana])。Louis 和 Andrew Sutton、Herb Sutter、David Vandevoorde 等人现在正在标准化委员会中努力，为语言使用元编程提供一流的支持。这项工作的一个重要基础是探索哪些程序属性应该通过反射可用;Matúš Chochlík, Axel Naumann 和 David Sankel 是该领域的主要贡献者。

John J. Barton 和 Lee R. Nackman 在 [BartonNackman] 中说明了在执行计算时如何跟踪维度单位。SIunits 库是一个更全面的库，用于处理 Walter Brown 开发的物理单元。标准库中的 std::chrono 组件 (我们使用它作为第 23.1.4 节的灵感来源) 只处理时间和日期，由 Howard Hinnant 贡献。

第 24 章 类型列表

编程通常需要使用各种数据结构，元编程也不例外。对于类型元编程，主要数据结构是类型列表，是一个包含类型的列表。模板元程序可以对这些类型列表进行操作，最终生成可执行程序的一部分。本章中，将讨论如何使用类型列表。本章大多数涉及到类型列表的操作都使用模板元编程，所以建议先了解一下第 23 章的元编程。

24.1. 解析

类型列表是表示列表的类型，可以由模板元程序操作。提供了与列表相关联的操作：迭代列表中的元素（类型）、添加元素或删除元素。类型列表与大多数运行时数据结构（如 std::list）不同，其不允许修改。向 std::list 中添加一个元素会改变列表本身，而这个改变会让程序中有权访问该列表的操作观察到。另一方面，向类型列表添加一个元素不会改变原始的打字员：向现有的类型列表员添加一个元素会创建一个新的类型列表，而非对原始类型列表的修改。熟悉函数式编程语言（如 Scheme、ML 和 Haskell）的读者可能会感觉到，使用 C++ 中的类型列表和使用这些语言中的列表非常相似。

类型列表通常实现为一个类模板特化，该特化在其模板参数中编码类型列表的内容（即它所包含的类型及其顺序）。类型列表的直接实现对参数包中的元素进行编码：

typelist/typelist.hpp

```
1 template<typename... Elements>
2 class Typelist
3 {
4 };
```

类型列表的元素可以直接作为模板参数。空的类型列表表示为 $\langle \rangle$ ，只包含 int 的类型列表表示为 $\langle \text{int} \rangle$ ，依此类推。下面是一个包含所有带符号整型的类型列表：

```
1 using SignedIntegralTypes =
2     Typelist<signed char, short, int, long, long long>;
```

操作这个类型列表通常需要将类型列表拆分成几个部分，通常是将列表中的第一个元素（头）与列表中的其余元素（尾）分开。从 Front 元函数从类型表中提取第一个元素：

typelist/typelistfront.hpp

```
1 template<typename List>
2 class FrontT;
3
4 template<typename Head, typename... Tail>
5 class FrontT<Typelist<Head, Tail...>>
6 {
7     public:
8     using Type = Head;
9 };
```

```
11 template<typename List>
12 using Front = typename FrontT<List>::Type;
```

因此, FrontT<SignedIntegralTypes>::Type(简化地写成 Front<SignedIntegralTypes>) 将生成 signed char。类似地, PopFront 元函数从列表中删除第一个元素, 实现将列表元素分为头和尾, 然后从尾中的元素形成一个新的类型列表特化。

typelist/typelistpopfront.hpp

```
1 template<typename List>
2 class PopFrontT;
3
4 template<typename Head, typename... Tail>
5 class PopFrontT<Typelist<Head, Tail...>> {
6     public:
7         using Type = Typelist<Tail...>;
8     };
9
10 template<typename List>
11 using PopFront = typename PopFrontT<List>::Type;
```

PopFront<SignedIntegralTypes> 会产生类型列表:

```
1 Typelist<short, int, long, long long>
```

还可以将所有元素捕获到模板参数包中, 然后创建一个包含所有这些元素的新类型列表特化, 从而将元素插入到类型列表的前面:

typelist/typelistpushfront.hpp

```
1 template<typename List, typename NewElement>
2 class PushFrontT;
3
4 template<typename... Elements, typename NewElement>
5 class PushFrontT<Typelist<Elements..., NewElement>> {
6     public:
7         using Type = Typelist<NewElement, Elements...>;
8     };
9
10 template<typename List, typename NewElement>
11 using PushFront = typename PushFrontT<List, NewElement>::Type;
```

如我们所期望,

```
1 PushFront<SignedIntegralTypes, bool>
```

生成了:

```
1 Typelist<bool, signed char, short, int, long, long long>
```

24.2. 算法

可以组合基本的类型列表，操作 Front、PopFront 和 PushFront 可以创建更有趣的类型列表操作。可以对 PopFront 的结果应用 PushFront 来替换类型列表中的第一个元素：

```
1 using Type = PushFront<PopFront<SignedIntegralTypes>, bool>;
2 // equivalent to Typelist<bool, short, int, long, long long>
```

更进一步，可以实现算法——搜索、转换、反转——作为操作在类型列表上的模板元函数。

24.2.1 索引

类型列表上最基本的操作是从列表中提取特定的元素。第 24.1 节说明了如何实现提取第一个元素的操作。这里，我们推广这个操作来提取第 n 个元素，要在给定的类型列表的索引 2 处提取类型：

```
1 using TL = NthElement<Typelist<short, int, long>, 2>;
```

TL 是一个别名。NthElement 操作是通过一个递归元程序实现的，遍历类型列表，直到找到所请求的元素：

typelist/nthelement.hpp

```
1 // recursive case:
2 template<typename List, unsigned N>
3 class NthElementT : public NthElementT<PopFront<List>, N-1>
4 {
5 };
6
7 // basis case:
8 template<typename List>
9 class NthElementT<List, 0> : public FrontT<List>
10 {
11 };
12
13 template<typename List, unsigned N>
14 using NthElement = typename NthElementT<List, N>::Type;
```

首先，考虑由 N 为 0 的偏特化的情况，特化通过在列表前面提供元素来终止递归。通过从 FrontT<List> 公开继承来实现，(间接地) 提供了 Type 类型别名，既是该列表的前端，也是 NthElement 元函数的结果，并使用元函数转发 (在 19.3.2 节中讨论)。

递归情况 (也是模板的主要定义) 遍历了类型列表。由于偏特化保证 $N > 0$ ，递归会从列表中删除前面的元素，并在剩余的列表元素中查找 $(N-1)^{st}$ 元素：

```
1 NthElementT<Typelist<short, int, long>, 2>
```

继承自

```
1 NthElementT<Typelist<int, long>, 1>
```

再继承自

```
1 NthElementT<Typelist<long>, 0>
```

这里，FrontT<typelist<long>> 继承自通过嵌套类型 Type 的结果类型。

24.2.2 寻找最佳匹配

许多类型列表算法在列表中搜索数据，希望找到类型列表中最大的类型（为列表中的类型分配足够的存储空间）。这也可以通过递归模板元程序来完成：

typelist/largesttype.hpp

```
1 template<typename List>
2 class LargestTypeT;
3
4 // recursive case:
5 template<typename List>
6 class LargestTypeT
7 {
8     private:
9         using First = Front<List>;
10    using Rest = typename LargestTypeT<PopFront<List>>::Type;
11    public:
12        using Type = IfThenElse<(sizeof(First) >= sizeof(Rest)), First, Rest>;
13    };
14
15 // basis case:
16 template<>
17 class LargestTypeT<Typelist<>>
18 {
19     public:
20         using Type = char;
21    };
22
23 template<typename List>
24 using LargestType = typename LargestTypeT<List>::Type;
```

LargestType 算法将返回类型列表中第一个最大的类型，给定类型列表 Typelist<bool, int, long, short>，该算法将返回与 long 大小相同的第一个类型，可能是 int 或 long，这取决于执行代码的平台。

某些平台上，bool 甚至与 long 大小相同！

LargestTypeT 的主要模板作为算法的递归用例会使用多次，从而采用了常见的第一/其他的惯用法，其有三个步骤。第一步中，只基于第一个元素（本例中是列表的前端元素）计算部分结果，并将其放在 First 中。接下来，递归计算列表中其余元素的结果，并将结果放在 Rest 中。在递归的第一步中，对于 Typelist<bool, int, long, short>，First 是 bool，而 Rest 是将算法应用到 Typelist<int, long,

`short>` 的结果。最后，第三步结合 `First` 和 `Rest` 得到结果。`IfThenElse` 选择列表中第一个元素中较大的 (`First`) 或目前为止最好的候选元素 (`Rest`)，并返回最大的那个。

类型列表可以包含 `sizeof`，但这个操作有不适用的类型，例如 `void`。这种情况下，编译器在试图计算类型表的最大类型时，产生一个错误。

`>=` 会倾向于使用前面的元素，也就是第一个最大的类型。

当列表为空时，递归终止。默认情况下，使用 `char` 作为初始化算法的哨兵类型，因为每种类型都和 `char` 一样大。

基本用例显式地提到了空类型列表。因为它排除了使用其他形式的类型列表，我们将在后面的部分 (包括第 24.3 节、第 24.5 节和第 25 章) 中返回这些类型列表。为了解决这个问题，引入了 `IsEmpty` 元函数来确定给定的类型列表是否为空：

typelist/typelistisempty.hpp

```
1 template<typename List>
2 class IsEmpty
3 {
4     public:
5     static constexpr bool value = false;
6 };
7
8 template<>
9 class IsEmpty<TypeList<>> {
10     public:
11     static constexpr bool value = true;
12 };
```

使用 `IsEmpty`，可以实现 `LargestType`。这样对实现 `Front`、`PopFront` 和 `IsEmpty` 的类型列表都有效：

typelist/genericlargesttype.hpp

```
1 template<typename List, bool Empty = IsEmpty<List>::value>
2 class LargestTypeT;
3
4 // recursive case:
5 template<typename List>
6 class LargestTypeT<List, false>
7 {
8     private:
9     using Contender = Front<List>;
10    using Best = typename LargestTypeT<PopFront<List>>::Type;
11
12    public:
13    using Type = IfThenElse<(sizeof(Contender) >= sizeof(Best)), 
14        Contender, Best>;
15 };
```

```

15
16 // basis case:
17 template<typename List>
18 class LargestTypeT<List, true>
19 {
20     public:
21     using Type = char;
22 };
23
24 template<typename List>
25 using LargestType = typename LargestTypeT<List>::Type;

```

LargestTypeT 的第二个默认模板参数 Empty 检查列表是否为空。若不空，递归情况 (将此参数固定为 false) 将继续查找列表。否则，基本情况 (将参数固定为 true) 终止递归并提供初始结果 (char)。

24.2.3 添加类型

PushFront 操作允许向类型列表的头部添加一个新类型，从而生成一个新的类型列表。假設想在列表的末尾添加一个新类型，对于类型列表模板，这个操作只需要对第 24.1 节中的 PushFront 实现做一些修改，就可以完成 PushBack 操作：

typelist/typelistpushback.hpp

```

1 template<typename List, typename NewElement>
2 class PushBackT;
3
4 template<typename... Elements, typename NewElement>
5 class PushBackT<Typelist<Elements...>, NewElement>
6 {
7     public:
8     using Type = Typelist<Elements..., NewElement>;
9 };
10
11 template<typename List, typename NewElement>
12 using PushBack = typename PushBackT<List, NewElement>::Type;

```

与 LargestType 算法一样，可以为 PushBack 实现一个通用算法，只使用基本操作 Front、PushFront、PopFront 和 IsEmpty：

要试用这个版本的算法，需要删除 Typelist 的 PushBack 的偏特化，否则要使用它来代替通用版本。

typelist/genericpushback.hpp

```

1 template<typename List, typename NewElement, bool = IsEmpty<List>::value>
2 class PushBackRect;
3

```

```

4 // recursive case:
5 template<typename List, typename NewElement>
6 class PushBackRecT<List, NewElement, false>
7 {
8     using Head = Front<List>;
9     using Tail = PopFront<List>;
10    using NewTail = typename PushBackRecT<Tail, NewElement>::Type;
11
12    public:
13        using Type = PushFront<Head, NewTail>;
14    };
15
16 // basis case:
17 template<typename List, typename NewElement>
18 class PushBackRecT<List, NewElement, true>
19 {
20     public:
21        using Type = PushFront<List, NewElement>;
22    };
23
24 // generic push-back operation:
25 template<typename List, typename NewElement>
26 class PushBackT : public PushBackRecT<List, NewElement> { };
27
28 template<typename List, typename NewElement>
29 using PushBack = typename PushBackT<List, NewElement>::Type;

```

PushBackRecT 模板管理递归，使用 PushFront 将 NewElement 添加到空列表中，因为 PushFront 相当于空列表中的 PushBack。递归的情况要有趣得多：将列表分成第一个元素 (Head) 和包含其余元素的类型列表 (Tail)，递归地将新元素附加到 Tail，以生成 NewTail。然后再次使用 PushFront 将 Head 添加到列表 NewTail 的前面，从而形成最终的列表。

使用一个简单的例子展开递归：

```
1 PushBackRecT<Typelist<short, int>, long>
```

最外面的步骤中，Head 是 short，Tail 是 Typelist<int>。继续递归

```
1 PushBackRecT<Typelist<int>, long>
```

其中 Head 是 int，Tail 是 Typelist<>。

再次递归计算

```
1 PushBackRecT<Typelist<>, long>
```

触发基本情况，并返回 PushFrontTypelist<>, long>，其计算结果为 Typelist<long>。然后展开递归，将前一个 Head 推到列表的前面：

```
1 PushFront<int, Typelist<long>>
```

这会产生 Typelist<int, long>。递归再次展开，将最外层的 Head(short) 推到这个列表上：

```
1 PushFront<short, Typelist<int, long>>
```

就产生了最终的结果:

```
1 Typelist<short, int, long>
```

通用的 PushBackRecT 实现适用于任何类型的类型列表。与本节中的其他算法一样，需要线性数量的模板实例来计算，因为对于长度为 N 的类型列表，将有 $N + 1$ 个 PushBackRecT 和 PushFrontT 的实例，以及 N 个 FrontT 和 PopFrontT 的实例。计算模板实例化的数量，可以粗略估计编译特定元程序所需的时间，因为模板实例化本身对于编译器来说是一个相当复杂的过程。

对于大型模板元程序来说，编译时间可能是一个问题，可以减少由这些算法执行模板实例化的数量。

Abrahams 和 Gurtovoy ([AbrahamsGurtovoyMeta]) 对模板元程序的编译时间进行了更深入的讨论，包括许多减少编译时间的技术。我们只是涉及了其表面。

PushBack 的第一个实现 (Typelist 上使用了偏特化) 只需要固定数量的模板实例化，这使得它(在编译时) 比通用版本要高效得多。因为描述为 PushBackT 的偏特化，所以在对 Typelist 实例执行 PushBack 时，将自动选择这个高效的实现，将算法特化的概念(如第 20.1 节所述) 引入模板元程序。本节中讨论的许多技术都可以应用于模板元程序，以减少算法模板实例化的数量。

24.2.4 反转

类型列表的元素之间有一定顺序时，在应用某些算法时，可以反转类型列表中元素的顺序。第 24.1 节中介绍的 SignedIntegralTypes 类型列表是按照递增的整数秩排序的。但可以使用元函数实现 Reverse 算法，将这个列表反向生成类型列表 Typelist<long long, long, int, short, signed char>:

typelist/typelistreverse.hpp

```
1 template<typename List, bool Empty = IsEmpty<List>::value>
2 class ReverseT;
3
4 template<typename List>
5 using Reverse = typename ReverseT<List>::Type;
6
7 // recursive case:
8 template<typename List>
9 class ReverseT<List, false>
10 : public PushBackT<Reverse<PopFront<List>>, Front<List>> { };
11
12 // basis case:
13 template<typename List>
14 class ReverseT<List, true>
15 {
16     public:
17         using Type = List;
```

18 } ;

此元函数递归情况是空类型表上的恒等函数。递归情况将列表分解为第一个元素和列表中的其余元素。若给定类型列表 `Typelist<short, int, long>`, 递归步骤将第一个元素 (`short`) 与其余元素 (`Typelist<int, long>`) 分离。然后递归地反转剩余的元素列表 (生成 `Typelist<long, int>`), 最后使用 `PushBackT` 将第一个元素追加到反转列表 (生成 `Typelist<long, int, short>`)。

`Reverse` 算法可以为类型列表实现 `PopBackT` 操作, 可从类型列表中删除最后一个元素:

typelist/typelistpopback.hpp

```
1 template<typename List>
2 class PopBackT {
3     public:
4         using Type = Reverse<PopFront<Reverse<List>>>;
5     };
6
7 template<typename List>
8 using PopBack = typename PopBackT<List>::Type;
```

该算法反转列表, 从反转列表中删除第一个元素 (使用 `PopFront`), 然后再次反转结果列表。

24.2.5 修改

以前的类型列表算法允许我们从类型列表器中提取元素、在列表中搜索、构造新列表和反向列表。但我们还没有对类型列表中的元素执行任何操作, 比如: 以某种方式“转换”类型列表中的所有类型,

函数式语言社区中, 这种操作通常称为 `map`(映射), 我们使用变换这个术语是为了更好地与 C++ 标准库自己的算法名称保持一致。

例如: 使用 `AddConst` 元函数将每个类型转换为符合常量条件的类型:

typelist/addconst.hpp

```
1 template<typename T>
2 struct AddConstT
3 {
4     using Type = T const;
5 };
6
7 template<typename T>
8 using AddConst = typename AddConstT<T>::Type;
```

为此, 我们将实现一个 `Transform` 算法, 该算法接受一个类型列表和一个元函数, 并生成另一个类型列表, 其中包含将元函数应用到每种类型的结果。

```
1 Transform<SignedIntegralTypes, AddConstT>
```

将是一个包含有符号 char const、short const、int const、long const 和 long long const 的类型列表。元函数通过双重模板参数提供，将输入类型映射到输出类型。Transform 算法本身，如期望一样，是一个递归算法：

typelist/transform.hpp

```
1 template<typename List, template<typename T> class MetaFun,
2 bool Empty = IsEmpty<List>::value>
3 class TransformT;
4
5 // recursive case:
6 template<typename List, template<typename T> class MetaFun>
7 class TransformT<List, MetaFun, false>
8 : public PushFrontT<typename TransformT<PopFront<List>, MetaFun>::Type,
9 typename MetaFun<Front<List>>::Type>
10 {
11 };
12
13 // basis case:
14 template<typename List, template<typename T> class MetaFun>
15 class TransformT<List, MetaFun, true>
16 {
17     public:
18     using Type = List;
19 }
20
21 template<typename List, template<typename T> class MetaFun>
22 using Transform = typename TransformT<List, MetaFun>::Type;
```

递归虽然语法繁琐，但不难。转换的结果是转换列表中的第一个元素 (到 PushFront 的第二个参数) 的结果，并将其添加到递归转换列表中的其余元素 (到 PushFront 的第一个参数) 生成的列表头部。

参见第 24.4 节，该节展示了如何开发更有效的 Transform 实现。

24.2.6 累加

变换是一种对序列的每个元素进行变换的算法，经常与累加一起使用，后者将序列的所有元素组合成单个结果值。

函数式语言社区中，这种操作通常称为归约，使用术语“累加”是为了更好地与 C++ 标准库的算法名称保持一致。

Accumulate 算法使用一个类型列表 T，其中包含元素 T₁、T₂、…、T_N(初始类型 I) 和元函数 F(接受两种类型并返回一种类型)。它返回 F(F(F(:F(I;T₁);T₂);…;T_{N-1});T_N)，其中在累加的第 i 步 F 应用到前面 i-1 步的结果和 T_i。

根据类型列表、F 的选择和初始类型的不同，可以使用 Accumulate 生成许多不同的结果，F 选

择两种类型中最大的，那么 Accumulate 的行为将类似于 LargestType 算法。另一方面，若 F 接受一个列表和一个类型，并将类型推到类型列表的后面，Accumulate 的行为将类似于 Reverse 算法。

Accumulate 的实现遵循标准的递归元程序分解：

typelist/accumulate.hpp

```
1 template<typename List,
2     template<typename X, typename Y> class F,
3     typename I,
4     bool = IsEmpty<List>::value>
5 class AccumulateT;
6
7 // recursive case:
8 template<typename List,
9     template<typename X, typename Y> class F,
10    typename I>
11 class AccumulateT<List, F, I, false>
12 : public AccumulateT<PopFront<List>, F,
13     typename F<I, Front<List>>::Type>
14 {
15 };
16
17 // basis case:
18 template<typename List,
19     template<typename X, typename Y> class F,
20     typename I>
21 class AccumulateT<List, F, I, true>
22 {
23     public:
24     using Type = I;
25 };
26
27 template<typename List,
28     template<typename X, typename Y> class F,
29     typename I>
30 using Accumulate = typename AccumulateT<List, F, I>::Type;
```

初始类型 I 也用作累加器，捕获当前结果。因此，在到达列表末尾时返回此结果。

这也确保了累加空列表的结果将是初始值。

递归情况下，算法将 F 应用到前面的结果 (I) 和列表的前面，将应用 F 的结果作为初始类型传递给列表的其余部分的累加。

有了 Accumulate，可以使用 PushFrontT 作为元函数 F，并使用空的类型列表 (Typelist<T>) 作为初始类型 I，生成一个反转的类型列表：

```
1 using Result = Accumulate<SignedIntegralTypes, PushFrontT, Typelist<>>;
2     // produces Typelist<long long, long, int, short, signed char>
```

实现基于累加器的 LargestType 版本，因为需要产生一个返回两种类型中较大的类型的元函数，所以 LargestTypeAcc 需要多做一些工作：

typelist/largesttypeacc0.hpp

```
1 template<typename T, typename U>
2 class LargerTypeT
3 : public IfThenElseT<sizeof(T) >= sizeof(U), T, U>
4 {
5 };
6
7 template<typename Typelist>
8 class LargestTypeAccT
9 : public AccumulateT<PopFront<Typelist>, LargerTypeT,
10 Front<Typelist>>
11 {
12 };
13
14 template<typename Typelist>
15 using LargestTypeAcc = typename LargestTypeAccT<Typelist>::Type;
```

因为提供了类型列表的第一个元素作为初始类型，所以 LargestType 的这种形式需要一个非空的类型列表。可以显式地处理空列表的情况，要么返回一些哨点类型 (char 或 void)，要么使算法对 SFINAE 友好，如 19.4.4 节所述：

typelist/largesttypeacc.hpp

```
1 template<typename T, typename U>
2 class LargerTypeT
3 : public IfThenElseT<sizeof(T) >= sizeof(U), T, U>
4 {
5 };
6
7 template<typename Typelist, bool = IsEmpty<Typelist>::value>
8 class LargestTypeAccT;
9
10 template<typename Typelist>
11 class LargestTypeAccT<Typelist, false>
12 : public AccumulateT<PopFront<Typelist>, LargerTypeT,
13 Front<Typelist>>
14 {
15 };
16
17 template<typename Typelist>
18 class LargestTypeAccT<Typelist, true>
19 {
20 };
21
22 template<typename Typelist>
```

```
23 using LargestTypeAcc = typename LargestTypeAccT<Typelist>::Type;
```

因为 Accumulate 允许表示许多不同的操作，所以其是一种功能强大的类型列表算法，可以认为是类型列表操作的基本算法。

24.2.7 插入排序

对于最后一个的类型列表算法，这里将实现插入排序。与其他算法一样，递归步骤将列表分成第一个元素(头)和其余元素(尾)。然后，对尾部进行排序(递归地)，并将头部插入已排序列表中的正确位置。该算法的对外表示为类型列表算法：

typelist/insertionsort.hpp

```
1 template<typename List,
2     template<typename T, typename U> class Compare,
3     bool = IsEmpty<List>::value>
4 class InsertionSortT;
5
6 template<typename List,
7     template<typename T, typename U> class Compare>
8 using InsertionSort = typename InsertionSortT<List, Compare>::Type;
9
10 // recursive case (insert first element into sorted list):
11 template<typename List,
12     template<typename T, typename U> class Compare>
13 class InsertionSortT<List, Compare, false>
14 : public InsertSortedT<InsertionSort<PopFront<List>, Compare>,
15     Front<List>, Compare>
16 {
17 };
18
19 // basis case (an empty list is sorted):
20 template<typename List,
21     template<typename T, typename U> class Compare>
22 class InsertionSortT<List, Compare, true>
23 {
24     public:
25     using Type = List;
26 };
```

Compare 参数是用于对类型列表中的元素进行排序的比较。其接受两种类型，并通过其 value 成员计算为布尔值。基本情况是输入一个空的类型列表，处理也很简单。

插入排序的核心是 InsertSortedT 元函数，已经排序的列表的第一点插入一个值，将保持列表的顺序：

typelist/insertsorted.hpp

```
1 #include "identity.hpp"
```

```

2 template<typename List, typename Element,
3     template<typename T, typename U> class Compare,
4     bool = IsEmpty<List>::value>
5 class InsertSortedT;
6
7 // recursive case:
8 template<typename List, typename Element,
9     template<typename T, typename U> class Compare>
10 class InsertSortedT<List, Element, Compare, false>
11 {
12     // compute the tail of the resulting list:
13     using NewTail =
14         typename IfThenElse<Compare<Element, Front<List>>::value,
15             IdentityT<List>,
16             InsertSortedT<PopFront<List>, Element, Compare>
17             >::Type;
18
19     // compute the head of the resulting list:
20     using NewHead = IfThenElse<Compare<Element, Front<List>>::value,
21         Element,
22         Front<List>>;
23
24     public:
25     using Type = PushFront<NewTail, NewHead>;
26 }
27
28 // basis case:
29 template<typename List, typename Element,
30     template<typename T, typename U> class Compare>
31 class InsertSortedT<List, Element, Compare, true>
32 : public PushFrontT<List, Element>
33 {
34 }
35
36 template<typename List, typename Element,
37     template<typename T, typename U> class Compare>
using InsertSorted = typename InsertSortedT<List, Element, Compare>::Type;

```

因为单元素列表总是排序的，所以也没啥难度。递归情况的不同取决于要插入的元素应该位于列表的开头还是列表的后面。若插入的元素位于列表中的第一个元素之前（该元素已经排序），则结果是该元素添加到带有 PushFront 的列表中。否则，要将列表分为头和尾，递归将该元素插入尾，然后将头添加到插入尾的元素的前面。

该实现包括一个编译时优化，以避免实例化不使用的类型，该技术在第 19.7.1 节中讨论过。下面的实现在技术上是正确的：

```

1 template<typename List, typename Element,
2     template<typename T, typename U> class Compare>
3 class InsertSortedT<List, Element, Compare, false>
4 : public IfThenElseT<Compare<Element, Front<List>>::value,
5     PushFront<List, Element>,

```

```

6     PushFront<InsertSorted<PopFront<List>,
7         Element, Compare>,
8     Front<List>>>
9 {
10 };

```

因为它计算 IfThenElseT 的两个分支中的模板参数，即使只使用一个分支，这种递归情况的表述都非常低效。例子中，then 分支中的 PushFront 通常是相当廉价，但 else 分支中的递归 InsertSorted 则不然。

我们的优化实现中，第一个 IfThenElse 计算结果列表的尾部 NewTail。IfThenElse 的第二个和第三个参数都是为该分支计算结果的元函数。第二个参数 (“then” 分支) 使用 IdentityT(见第 19.7.1 节) 生成未修改的 List。第三个参数 (“else” 分支) 使用 InsertSortedT 计算在排序列表中插入的元素。使用时，只实例化 IdentityT 或 InsertSortedT 中的一个，因此执行的工作量很小(最坏的情况下是 PopFront)。第二个 IfThenElse 会计算结果列表的头，因为假设两个分支都十分廉价，所以会立即对分支进行评估。最终的列表是由计算出来的 NewHead 和 NewTail 构造。该方式具有一个理想的特性，即向排序列表中插入一个元素所需的实例化数量与其在结果列表中的位置成正比。这表现为插入排序的一个更高级别的属性，即对已经排序的列表进行排序的实例化的数量，与列表的长度成线性关系。(若已排序列表的排列顺序和预期顺序相反的话，所需要的实例化数目和列表长度的平方成正比)

下面的程序演示了如何使用插入排序，来根据类型的大小对类型列表进行排序。比较操作使用 sizeof 操作符比较结果：

typelist/insertionsorttest.hpp

```

1 template<typename T, typename U>
2 struct SmallerThanT {
3     static constexpr bool value = sizeof(T) < sizeof(U);
4 };
5
6 void testInsertionSort()
7 {
8     using Types = Typelist<int, char, short, double>;
9     using ST = InsertionSort<Types, SmallerThanT>;
10    std::cout << std::is_same<ST, Typelist<char, short, int, double>>::value
11        << '\n' ;
12 }

```

24.3. 非类型

类型列表提供了使用一组丰富的算法和操作描述，以及一系列操作类型的能力。其在处理编译时可以处理一组值，例如：多维数组的边界或另一个类型列表的索引。

有几种方法可以生成编译时值的类型列表。一种简单的方法包括定义一个 CTValue 类模板(编译时值命名)，在一个类型列表中表示特定类型的值：

标准库定义了 std::integral_constant 模板，它是 CTValue 的加强版本。

typelist/ctvalue.hpp

```
1 template<typename T, T Value>
2 struct CTValue
3 {
4     static constexpr T value = Value;
5 }
```

使用 CTValue 模板，可以表示一个包含前几个素数的整数值的类型列表：

```
1 using Primes = Typelist<CTValue<int, 2>, CTValue<int, 3>,
2                 CTValue<int, 5>, CTValue<int, 7>,
3                 CTValue<int, 11>>;
```

通过这种表示，可以对值的类型列表执行数值计算，例如：计算这些质数的乘积。

首先，MultiplyT 模板接受两个相同类型的编译时值，并生成一个相同类型的新的编译时值，将输入值相乘：

typelist/multiply.hpp

```
1 template<typename T, typename U>
2 struct MultiplyT;
3
4 template<typename T, T Value1, T Value2>
5 struct MultiplyT<CTValue<T, Value1>, CTValue<T, Value2>> {
6     public:
7         using Type = CTValue<T, Value1 * Value2>;
8     };
9
10 template<typename T, typename U>
11 using Multiply = typename MultiplyT<T, U>::Type;
```

然后，通过使用 MultiplyT，下面的表达式得到所有质数的乘积：

```
1 Accumulate<Primes, MultiplyT, CTValue<int, 1>>::value
```

不过，类型列表和 CTValue 的这种用法相当繁琐，特别是对于所有值都相同类型的情况。可以通过引入一个别名模板 CTTypelist 进行优化。提供了一个同类的值列表，描述为 CTValues 的类型列表：

typelist/cttypelist.hpp

```
1 template<typename T, T... Values>
2 using CTTypelist = Typelist<CTValue<T, Values>...>;
```

现在可以使用 CTTypelist 编写一个等效的(但更简洁的)质数定义：

```
1 using Primes = CTTypelist<int, 2, 3, 5, 7, 11>;
```

这种方法的唯一缺点是别名模板只是别名，因此错误消息可能最终会打印 CTValueTypes 的底层类型列表。为了解决这个问题，可以创建一个全新的类型列表类 Valuelist，可以存储值：

typelist/valuelist.hpp

```
1 template<typename T, T... Values>
2 struct Valuelist {
3 };
4
5 template<typename T, T... Values>
6 struct IsEmpty<Valuelist<T, Values...>> {
7     static constexpr bool value = sizeof...(Values) == 0;
8 };
9
10 template<typename T, T Head, T... Tail>
11 struct FrontT<Valuelist<T, Head, Tail...>> {
12     using Type = CTValue<T, Head>;
13     static constexpr T value = Head;
14 };
15
16 template<typename T, T Head, T... Tail>
17 struct PopFrontT<Valuelist<T, Head, Tail...>> {
18     using Type = Valuelist<T, Tail...>;
19 };
20
21 template<typename T, T... Values, T New>
22 struct PushFrontT<Valuelist<T, Values...>, CTValue<T, New>> {
23     using Type = Valuelist<T, New, Values...>;
24 };
25
26 template<typename T, T... Values, T New>
27 struct PushBackT<Valuelist<T, Values...>, CTValue<T, New>> {
28     using Type = Valuelist<T, Values..., New>;
29 };
```

通过提供 IsEmpty, FrontT, PopFrontT 和 PushFrontT，已经使 Valuelist 成为好用的类型列表，并且可以在本章定义的算法中使用。PushBackT 作为一种算法特化提供，以减少编译时此操作的成本。例如，Valuelist 可以与之前定义的 InsertionSort 算法一起使用：

typelist/valuelisttest.hpp

```
1 template<typename T, typename U>
2 struct GreaterThanT;
3
4 template<typename T, T First, T Second>
5 struct GreaterThanT<CTValue<T, First>, CTValue<T, Second>> {
```

```

6   static constexpr bool value = First > Second;
7 }
8 void valuelisttest()
9 {
10    using Integers = Valuelist<int, 6, 2, 4, 9, 5, 2, 1, 7>;
11
12    using SortedIntegers = InsertionSort<Integers, GreaterThanT>;
13
14    static_assert(std::is_same_v<SortedIntegers,
15                  Valuelist<int, 9, 7, 6, 5, 4, 2, 2, 1>>,
16                  "insertion sort failed");
17 }

```

可以通过使用字面符操作符来初始化 CTValue,

```
1 auto a = 42_c; // initializes a as CTValue<int, 42>
```

参阅第 25.6 节了解详细信息。

24.3.1 可推导的非类型参数

C++17 中, CTValue 可以通过使用单个可推导的非类型形参 (用 auto) 进行改进:

typelist/ctvalue17.hpp

```

1 template<auto Value>
2 struct CTValue
3 {
4     static constexpr auto value = Value;
5 };

```

这就省去了每次使用 CTValue 时指定类型的需要:

```

1 using Primes = Typelist<CTValue<2>, CTValue<3>, CTValue<5>,
2                               CTValue<7>, CTValue<11>>;

```

C++17 的值列表也可以这样, 但结果不一定更好。如 15.10.1 节所述, 带有推导类型的非类型参数包允许每个参数的类型不同:

```

1 template<auto... Values>
2 class Valuelist { };
3
4 int x;
5 using MyValueList = Valuelist<1, 'a', true, &x>;

```

虽然这样的异构值列表可能有用, 但与前面要求所有元素都具有相同类型的值列表不同。虽然可以要求所有元素都具有相同的类型 (在 15.10.1 节中也讨论了这一点), 但空的 Valuelist<> 肯定没有已知的元素类型。

24.4. 包扩展优化算法

包扩展(在第 12.4.1 节中有详细描述)是一种机制, 可将类型列表迭代的工作转移给编译器。因为需要对列表中的每个元素应用相同的操作(进行包扩展), 这里可以使用在第 24.2.5 节中开发的 Transform 算法。这为类型列表的 Transform 提供了一个特化算法(通过偏特化):

typelist/variadictransform.hpp

```
1 template<typename... Elements, template<typename T> class MetaFun>
2 class TransformT<Typelist<Elements...>, MetaFun, false>
3 {
4     public:
5     using Type = Typelist<typename MetaFun<Elements>::Type...>;
6 }
```

这个实现将类型列表元素捕获到一个参数包 `elements` 中, 使用模式 `typename MetaFun<Elements>::Type` 进行包扩展, 将元功能应用到 `Elements` 中的每个类型, 并形成一个类型列表。因为它不需要递归, 所以这个实现更简单, 并且以一种相当直接的方式使用语言特性。因为只需要实例化 `Transform` 模板的一个实例, 所以需要更少的模板实例化。该算法仍然需要线性数量的 `MetaFun` 实例化, 这些实例化是算法的基础。

其他算法间接受益于使用包扩展, 在第 24.2.4 节中描述的反向算法需要线性数量的 `PushBack` 实例化。在那一节中描述的类型列表 `PushBack` 的包扩展形式(需要单个实例化), `Reverse` 是线性的。然而, 描述的 `Reverse` 更一般的递归实现本身在实例化数量上是线性的, 使 `Reverse` 的复杂度与数量成平方关系(2 次方)!

选择给定索引列表中的元素以生成新的类型列表时, 可以使用 Pack 扩展。`Select` 元函数接受一个类型列表和一个包含该类型列表索引的值列表, 然后生成一个包含由值列表指定元素的新类型列表:

typelist/select.hpp

```
1 template<typename Types, typename Indices>
2 class SelectT;
3
4 template<typename Types, unsigned... Indices>
5 class SelectT<Types, Valuelist<unsigned, Indices...>>
6 {
7     public:
8     using Type = Typelist<NthElement<Types, Indices>...>;
9 }
10
11 template<typename Types, typename Indices>
12 using Select = typename SelectT<Types, Indices>::Type;
```

索引在参数包 `Indices` 中捕获, 该参数包扩展以生成索引到给定类型列表的一系列 `NthElement` 类型, 并在新参数列表中捕获结果。下面的例子说明了如何使用 `Select` 来反转输入列表:

```
1 using SignedIntegralTypes =
```

```

2   Typelist<signed char, short, int, long, long long>;
3
4   using ReversedSignedIntegralTypes =
5     Select<SignedIntegralTypes, Valuelist<unsigned, 4, 3, 2, 1, 0>>;
6   // produces Typelist<long long, long, int, short, signed char>

```

包含另一个列表索引的非类型类型列表，称为索引列表(或索引序列)，允许简化或消除递归计算。索引列表在 25.3.4 节有详细的描述。

24.5. Cons 风格

引入可变参数模板前，可以使用模仿 LISP 的 cons 单元的递归数据结构来制定类型列表。每个 Cons 单元格包含一个值(列表头部)和一个嵌套列表，后者可以是另一个 Cons 单元格，也可以是空列表 nil。这个概念可以直接用 C++ 表示：

typelist/cons.hpp

```

1 class Nil { };
2
3 template<typename HeadT, typename TailT = Nil>
4 class Cons {
5   public:
6     using Head = HeadT;
7     using Tail = TailT;
8 }

```

空的类型列表写为 Nil，而包含 int 的单元素列表写为 Cons<int, Nil>，或者更简单的 Cons<int>。较长的列表则需要嵌套：

```

1 using TwoShort = Cons<short, Cons<unsigned short>>;

```

可以通过深度递归嵌套来构造任意长度的类型列表，手工编写这样长的列表可能会有些蠢：

```

1 using SignedIntegralTypes = Cons<signed char, Cons<short, Cons<int,
2   Cons<long, Cons<long long, Nil>>>>;

```

提取 Cons 样式列表中的第一个元素直接指向列表的头：

typelist/consfront.hpp

```

1 template<typename List>
2 class FrontT {
3   public:
4     using Type = typename List::Head;
5   };
6
7 template<typename List>
8 using Front = typename FrontT<List>::Type;

```

前面添加一个元素将对现有列表进行限制:

typelist/conspushfront.hpp

```
1 template<typename List, typename Element>
2 class PushFrontT {
3     public:
4         using Type = Cons<Element, List>;
5     };
6
7 template<typename List, typename Element>
8 using PushFront = typename PushFrontT<List, Element>::Type;
```

最后, 从递归类型列表中删除第一个元素将提取列表的尾部:

typelist/conspopfront.hpp

```
1 template<typename List>
2 class PopFrontT {
3     public:
4         using Type = typename List::Tail;
5     };
6
7 template<typename List>
8 using PopFront = typename PopFrontT<List>::Type;
```

Nil 的 IsEmpty 特化完成了核心类型列表操作集:

typelist/consisempty.hpp

```
1 template<typename List>
2 struct IsEmpty {
3     static constexpr bool value = false;
4 };
5
6 template<>
7 struct IsEmpty<Nil> {
8     static constexpr bool value = true;
9 };
```

有了这些类型列表操作, 就可以使用在 24.2.7 节中定义的 InsertionSort 算法, 这次使用 Cons 风格的列表:

typelist/conslisttest.hpp

```
1 template<typename T, typename U>
2 struct SmallerThanT {
3     static constexpr bool value = sizeof(T) < sizeof(U);
4 };
```

```
6 void conslisttest()
7 {
8     using ConsList = Cons<int, Cons<char, Cons<short, Cons<double>>>>;
9     using SortedTypes = InsertionSort<ConsList, SmallerThanT>;
10    using Expected = Cons<char, Cons<short, Cons<int, Cons<double>>>>;
11    std::cout << std::is_same<SortedTypes, Expected>::value << '\n' ;
12 }
```

正如在插入排序中看到的，Cons 风格的类型列表可以表达适用于可变类型列表上的所有算法。但这也有一些缺点，所以我们更倾向于可变类型的版本：首先，嵌套使得长 Cons 风格类型列表在源代码很难书写，并且编译器与之相关的诊断信息很难读懂。第二，一些算法（包括 PushBack 和 Transform）可以专门用于可变类型列表，以提供更有效的实现（通过实例化的数量来衡量）。最后，在类型列表中使用可变参数模板与在异构容器中使用可变参数模板相同，这已在第 25 章和第 26 章中讨论过了。

24.6. 后记

类型列表似乎在 1998 年第一个 C++ 标准发布后不久就出现了。Krysztof Czarnecki 和 Ulrich Eisenecker 在 [CzarneckiEiseneckerGenProg] 中引入了受 LISP 启发的 Cons 风格的整型常量列表，不过他们并没有考虑过泛型类型列表。

Alexandrescu 在他极具影响力的著作《现代 C++ 设计》([AlexandrescuDesign]) 中使类型列表流行起来。最重要的是，Alexandrescu 展示了类型列表的更多功能，可以用模板元编程和类型列表解决有趣的设计问题，并能使 C++ 开发者可以使用这些技术。

Abrahams 和 Gurtovoy 在 [AbrahamsGurtovoyMeta] 中为元编程提供了所需的结构，描述了从 C++ 标准库中提取的对类型列表、类型列表算法和相关组件的抽象：序列、迭代器、算法和（元）函数。库 Boost.MPL([BoostMPL]) 可用来操作类型列表。

第 25 章 元组

本书中，我们经常使用同构容器和类数组类型来说明模板的强大功能。这种同构结构扩展了 C/C++ 数组的概念，在大多数应用程序中普遍存在。C++(和 C) 也有非同构的容器设施：类(或结构体)，本章探讨元组类似于类和结构的方式聚合数据，包含 int、double 和 std::string 的元组类似于包含 int、double 和 std::string 成员的结构体，只是元组的元素是按位置引用的(如 0、1、2)，而不是通过名称引用。位置接口和从类型列表轻松构造元组的能力，使元组比结构体更适合与模板元编程技术一起使用。

元组的另一种方式是在可执行程序中显示类型列表，Typelist<int, double, std::string> 描述了可以在编译时操作的包含 int, double 和 std::string 的类型序列，Tuple<int, double, std::string> 描述了可以在运行时操作的 int, double 和 std::string 的存储。例如，下面的程序创建这样一个元组的实例：

```
1 template<typename... Types>
2 class Tuple {
3     ... // implementation discussed below
4 };
5
6 Tuple<int, double, std::string> t(17, 3.14, "Hello, World!");
```

通常使用模板元编程和类型列表来生成可用于存储数据的元组，即使在上面的例子中选择了 int、double 和 std::string 作为元素类型，也可以用元程序创建元组中存储的类型集。

本章，我们将探索 Tuple 类模板的实现和操作，是 std::tuple 类模板的简化版本。

25.1. 类型设计

25.1.1 存储

元组包含模板参数列表中每个类型的存储空间，可以通过函数模板 get 来访问，用作元组 t 的 get<i>(t)。例如，在前面的例子中，t 上的 get<0>(t) 将返回对 int 17 的引用，而 get<1>(t) 将返回对 double 3.14 的引用。

元组存储的递归公式基于这样一种思想：包含 $N(>0)$ 个元素的元组，既可以存储为单个元素(列表的第一个元素或头元素)，也可以存储为包含 $N - 1$ 个元素的元组(列表的尾部)，对于空元组有单独的处理。因此，三元组 tuple<int, double, std::string> 可以存储为 int 类型，而 tuple<double, std::string> 可以存储为三元组，包含两个元素的元组可以存储为 double 类型和 tuple<std::string> 类型，其本身可以存储为 std::string 类型和 Tuple<> 类型。这与通用版本的类型列表算法中使用的递归分解相同，递归元组存储的实现过程也类似：

typelist/tuple0.hpp

```
1 template<typename... Types>
2 class Tuple;
3
4 // recursive case:
5 template<typename Head, typename... Tail>
6 class Tuple<Head, Tail...>
```

```

7  {
8      private:
9      Head head;
10     Tuple<Tail...> tail;
11 
12     public:
13     // constructors:
14     Tuple() {
15     }
16     Tuple(Head const& head, Tuple<Tail...> const& tail)
17         : head(head), tail(tail) {
18     }
19     ...
20 
21     Head& getHead() { return head; }
22     Head const& getHead() const { return head; }
23     Tuple<Tail...>& getTail() { return tail; }
24     Tuple<Tail...> const& getTail() const { return tail; }
25 }
26 
27 // basis case:
28 template<>
29 class Tuple<> {
30     // no storage required
31 };

```

递归时，每个 Tuple 实例包含存储列表中第一个元素的数据成员头部，以及存储列表中剩余元素的数据成员尾部。一般空元组，没有相关的存储。

get 函数模板遍历这个递归结构来提取相应的元素：

get() 的完整实现还应该处理非常量和右值引用元组。

typelist/tupleget.hpp

```

1 // recursive case:
2 template<unsigned N>
3 struct TupleGet {
4     template<typename Head, typename... Tail>
5     static auto apply(Tuple<Head, Tail...> const& t) {
6         return TupleGet<N-1>::apply(t.getTail());
7     }
8 };
9 
10 // basis case:
11 template<>
12 struct TupleGet<0> {
13     template<typename Head, typename... Tail>
14     static Head const& apply(Tuple<Head, Tail...> const& t) {
15         return t.getHead();
16     }
17 };

```

```

16     }
17 }
18
19 template<unsigned N, typename... Types>
20 auto get(Tuple<Types...> const& t) {
21     return TupleGet<N>::apply(t);
22 }
```

函数模板 `get` 只是对 `TupleGet` 的静态成员函数使用的一个简单包装，有效地解决了函数模板缺乏偏特化的问题（在第 17.3 节中讨论），我们使用它特化 `N` 的值。在递归情况下 (`N > 0`)，静态成员函数 `apply()` 可取当前元组的尾部，并递减 `N` 以继续寻找元组中稍后请求的元素。一般情况会 (`N = 0`) 返回当前元组的头部。

25.1.2 构造

除了目前定义的构造函数之外：

```

1 Tuple() {
2 }
3
4 Tuple(Head const& head, Tuple<Tail...> const& tail)
5 : head(head), tail(tail) {
6 }
```

要使元组有效，需要能够从一组独立的值（每个元素一个值）和另一个元组中进行构造。复制构造从一组独立的值传递第一个值来初始化头部元素（通过基类），然后将剩余的值传递给基类，表示尾部：

```

1 Tuple(Head const& head, Tail const&... tail)
2 : head(head), tail(tail...) {
3 }
```

初始化元组：

```
1 Tuple<int, double, std::string> t(17, 3.14, "Hello, World!");
```

这并不是最通用的接口：用户可能希望使用移动构造来初始化部分（但可能不是全部）元素，或者使用不同类型的值构造。因此，应该使用完美转发（第 15.6.3 节）来初始化元组：

```

1 template<typename VHead, typename... VTail>
2 Tuple(VHead&& vhead, VTail&&... vtail)
3 : head(std::forward<VHead>(vhead)),
4     tail(std::forward<VTail>(vtail))...
5 }
```

接下来，实现使用一个元组构造另一个元组：

```

1 template<typename VHead, typename... VTail>
2 Tuple(Tuple<VHead, VTail...> const& other)
3 : head(other.getHead()), tail(other.getTail()) { }
```

然而，这个构造函数还不足以允许元组转换：给定上面的元组 `t`，尝试创建另一个具有兼容类型的元组将会失败：

```
1 // ERROR: no conversion from Tuple<int, double, string> to long
2 Tuple<long int, long double, std::string> t2(t);
```

这里的问题是，用于从一组独立值进行初始化的构造函数模板，比接受元组的构造函数模板更匹配。为了解决这个问题，当尾部没有预期的长度时，必须使用 `std::enable_if<>`(参见第 6.3 节和第 20.3 节) 来禁用两个成员函数模板：

```
1 template<typename VHead, typename... VTail,
2         typename = std::enable_if_t<sizeof...(VTail)==sizeof...(Tail)>>
3     Tuple(VHead&& vhead, VTail&&... vtail)
4     : head(std::forward<VHead>(vhead)),
5     tail(std::forward<VTail>(vtail)) { }
6
7 template<typename VHead, typename... VTail,
8         typename = std::enable_if_t<sizeof...(VTail)==sizeof...(Tail)>>
9     Tuple(Tuple<VHead, VTail...> const& other)
10    : head(other.getHead()), tail(other.getTail()) { }
```

可以在 `tuple/tuple.hpp` 中找到所有构造函数的声明。

`makeTuple()` 函数模板使用推导来确定返回元组的元素类型，使得给定的元素集创建元组变得相对容易许多：

typelist/maketuple.hpp

```
1 template<typename... Types>
2 auto makeTuple(Types&&... elems)
3 {
4     return Tuple<std::decay_t<Types>...>(std::forward<Types>(elems)...);
5 }
```

再次使用完美转发与 `std::decay<>` 特征相结合，将字面值和其他原始数组转换成指针，并移除 `const` 和引用限定。

```
1 makeTuple(17, 3.14, "Hello, World!")
```

初始化

```
1 Tuple<int, double, char const*>
```

25.2. 基础操作

25.2.1 比较

元组是包含其他值的结构类型。要比较两个元组，比较元素就足够了，可以实现 `operator==` 的定义来比较两个定义的元素：

typelist/tupleeq.hpp

```

1 // basis case:
2 bool operator==(Tuple<> const&, Tuple<> const&)
3 {
4     // empty tuples are always equivalent
5     return true;
6 }
7
8 // basis case:
9 bool operator==(Tuple<> const&, Tuple<> const&)
10 {
11     // empty tuples are always equivalent
12     return true;
13 }

```

与许多关于类型列表和元组的算法一样，元素比较先访问头部元素，然后递归访问尾部元素。操作符!=、<、>、<= 和 >= 的顺序类似。

25.2.2 输出

本章将创建新的元组类型，因此能够在执行程序中看到这些元组是很有用的。以下操作符 << 可以打印任何可打印的元组元素类型：

typelist/tupleio.hpp

```

1 #include <iostream>
2
3 void printTuple(std::ostream& strm, Tuple<> const&, bool isFirst = true)
4 {
5     strm << ( isFirst ? ' (' : ') ' );
6 }
7
8 template<typename Head, typename... Tail>
9 void printTuple(std::ostream& strm, Tuple<Head, Tail...> const& t,
10      bool isFirst = true)
11 {
12     strm << ( isFirst ? "(" : ", " );
13     strm << t.getHead();
14     printTuple(strm, t.getTail(), false);
15 }
16
17 template<typename... Types>
18 std::ostream& operator<<(std::ostream& strm, Tuple<Types...> const& t)
19 {
20     printTuple(strm, t);
21     return strm;
22 }

```

现在，创建和显示元组就很容易：

```
1 std::cout << makeTuple(1, 2.5, std::string("hello")) << '\n' ;
```

输出为

```
(1, 2.5, hello)
```

25.3. 算法

元组是一个容器，提供了访问和修改其每个元素的能力 (通过 `get`)，以及创建新的元组 (直接或使用 `makeTuple()`)，并将元组分解为头部和尾部 (`getHead()` 和 `getTail()`)。这些基本构建块足以构建一套元组算法，例如从元组中添加或删除元素，重新排序元组中的元素，或选择元组中元素的某些子集。

元组算法特别有趣，因为同时需要编译时和运行时计算。与第 24 章的类型列表算法相同，将算法应用到元组可能会得到完全不同类型的元组，这需要编译时计算。对 `Tuple<int, double, string>` 进行反转会产生 `Tuple<string, double, int>`，就像同构容器的算法 (例如，`std::vector` 上的 `std::reverse()`) 一样，元组算法实际上需要在运行时执行代码，所以需要注意生成代码的效率。

25.3.1 元组作为类型列表

若忽略 `Tuple` 模板的实际运行时组件，会发现其结构与第 24 章中开发的类型列表模板完全相同：可以接受任意数量的模板类型参数。通过一些局部特化，可以将 `Tuple` 变成一个功能齐全的类型列表：

tuples/tupletypelist.hpp

```
1 // determine whether the tuple is empty:
2 template<>
3 struct IsEmpty<Tuple<>> {
4     static constexpr bool value = true;
5 };
6
7 // extract front element:
8 template<typename Head, typename... Tail>
9 class FrontT<Tuple<Head, Tail...>> {
10     public:
11     using Type = Head;
12 };
13
14 // remove front element:
15 template<typename Head, typename... Tail>
16 class PopFrontT<Tuple<Head, Tail...>> {
17     public:
18     using Type = Tuple<Tail...>;
19 };
```

```

20
21 // add element to the front:
22 template<typename... Types, typename Element>
23 class PushFrontT<Tuple<Types...>, Element> {
24     public:
25     using Type = Tuple<Element, Types...>;
26 };
27
28 // add element to the back:
29 template<typename... Types, typename Element>
30 class PushBackT<Tuple<Types...>, Element> {
31     public:
32     using Type = Tuple<Types..., Element>;
33 };

```

第 24 章中开发的所有类型列表算法，在 Tuple 和类型列表中也能工作，可以轻松地处理元组的类型：

```

1 Tuple<int, double, std::string> t1(17, 3.14, "Hello, World!");
2 using T2 = PopFront<PushBack<decltype(t1), bool>>;
3 T2 t2(get<1>(t1), get<2>(t1), true);
4 std::cout << t2;

```

输出为

```
(3.14, Hello, World!, 1)
```

应用于元组类型的类型列表算法，可用来确定元组算法的结果类型。

25.3.2 添加和删除元素

对于元组的值，在开头或结尾添加元素的能力对于构建更高级的算法很重要。与类型列表一样，在元组前面插入比在后面插入容易，所以先从 pushFront 开始：

typelist/pushfront.hpp

```

1 template<typename... Types, typename V>
2 PushFront<Tuple<Types...>, V>
3 pushFront(Tuple<Types...> const& tuple, V const& value)
4 {
5     return PushFront<Tuple<Types...>, V>(value, tuple);
6 }

```

现有元组的前面添加一个新元素（称为值）要求以值为头，以现有元组作为尾。产生的元组类型是 Tuple<V, Types...>。但我们选择使用类型列表算法 PushFront 来演示元组算法的编译时和运行时间的耦合关系：编译时 PushFront 计算需要构造的类型，以产生运行时的值。

在现有元组的末尾添加新元素更为复杂，因为需要对元组进行递归遍历，并在遍历过程中构建修改后的元组。`pushBack()` 实现的结构遵循第 24.2.3 节中类型列表 `pushBack()` 的递归公式：

typelist/pushback.hpp

```
1 // basis case
2 template<typename V>
3 Tuple<V> pushBack(Tuple<> const&, V const& value)
4 {
5     return Tuple<V>(value);
6 }
7
8 // recursive case
9 template<typename Head, typename... Tail, typename V>
10 Tuple<Head, Tail..., V>
11 pushBack(Tuple<Head, Tail...> const& tuple, V const& value)
12 {
13     return Tuple<Head, Tail..., V>(tuple.getHead(),
14         pushBack(tuple.getTail(), value));
15 }
```

基本情况，通过生成只包含该值的元组，将值追加到零长度的元组。在递归的情况下，需要用列表头部的当前元素 (`tuple.gethead()`)，并将新元素添加到列表尾部的结果 (递归 `pushBack` 调用)，组成一个新的元组。虽然将构造的类型表示为 `Tuple<Head, Tail..., V>`，这相当于在编译时使用 `PushBack<Tuple<Head, Tail...>, V>`。

另外，`popFront()` 很容易实现：

typelist/popfront.hpp

```
1 template<typename... Types>
2 PopFront<Tuple<Types...>>
3 popFront(Tuple<Types...> const& tuple)
4 {
5     return tuple.getTail();
6 }
```

现在可以对 25.3.1 节中的示例进行编程：

```
1 Tuple<int, double, std::string> t1(17, 3.14, "Hello, World!");
2 auto t2 = popFront(pushBack(t1, true));
3 std::cout << std::boolalpha << t2 << '\n' ;
```

输出为

```
(3.14, Hello, World!, true)
```

25.3.3 反转

元组元素可以用另一种递归元组算法进行反转，该算法的结构遵循第 24.2.4 节的类型列表反转：

typelist/reverse.hpp

```
1 // basis case
2 Tuple<> reverse(Tuple<> const& t)
3 {
4     return t;
5 }
6
7 // recursive case
8 template<typename Head, typename... Tail>
9 Reverse<Tuple<Head, Tail...>> reverse(Tuple<Head, Tail...> const& t)
10 {
11     return pushBack(reverse(t.getTail()), t.getHead());
12 }
```

基本情况很简单，而递归情况则反转列表的尾部，并将当前头部元素追加到反转的列表中。

```
1 reverse(makeTuple(1, 2.5, std::string("hello")))
```

将产生一个 `Tuple<string, double, int>`，其值分别为 `string("hello")`，`2.5` 和 `1`。

和类型列表一样，可以通过调用 `popFront()` 来提供 `popBack()` 来反转列表，使用第 24.2.4 节中的 `popBack`：

typelist/popback.hpp

```
1 template<typename... Types>
2 PopBack<Tuple<Types...>>
3 popBack(Tuple<Types...> const& tuple)
4 {
5     return reverse(popFront(reverse(tuple)));
6 }
```

25.3.4 索引列表

上一节中，元组反转递归公式是正确，但在运行时效率很低。为了了解这个问题，我们引入了简单的类，其会计算复制自己的次数：

C++17 前，不支持内联静态成员，必须在一个翻译单元中初始化类结构外的 `numCopies`。

typelist/copycounter.hpp

```
1 template<int N>
```

```
2 struct CopyCounter
3 {
4     inline static unsigned numCopies = 0;
5     CopyCounter() {
6     }
7     CopyCounter(CopyCounter const&) {
8         ++numCopies;
9     }
10};
```

创建并反转一个 CopyCounter 实例元组:

typelist/copycountertest.hpp

```
1 void copycountertest()
2 {
3     Tuple<CopyCounter<0>, CopyCounter<1>, CopyCounter<2>,
4         CopyCounter<3>, CopyCounter<4>> copies;
5     auto reversed = reverse(copies);
6     std::cout << "0: " << CopyCounter<0>::numCopies << " copies\n";
7     std::cout << "1: " << CopyCounter<1>::numCopies << " copies\n";
8     std::cout << "2: " << CopyCounter<2>::numCopies << " copies\n";
9     std::cout << "3: " << CopyCounter<3>::numCopies << " copies\n";
10    std::cout << "4: " << CopyCounter<4>::numCopies << " copies\n";
11}
```

程序将输出:

```
0: 5 copies
1: 8 copies
2: 9 copies
3: 8 copies
4: 5 copies
```

这么多次复制! 元组反向的理想实现中, 每个元素只会复制一次就够了, 从源元组直接复制到结果元组中的正确位置。我们可以通过引用(包括使用中间参数类型的引用)来实现这一目标, 但会使实现变得相当复杂。

为了在元组反向操作中消除多余的副本, 考虑如何对已知长度的单个元组(如本例中为 5 个元素)实现一次性元组反向操作。可以简单地使用 `makeTuple()` 和 `get()`:

```
1 auto reversed = makeTuple(get<4>(copies), get<3>(copies),
2                             get<2>(copies), get<1>(copies),
3                             get<0>(copies));
```

这个程序产生了期望的输出, 每个元组元素只有一个副本:

```
0: 1 copies
1: 1 copies
2: 1 copies
3: 1 copies
4: 1 copies
```

索引列表 (也称为索引序列; 参见第 24.4 节) 通过将元组索引集 (本例中为 4、3、2、1、0) 捕获到参数包中来推广，并允许通过包展开产生 get 调用序列。计算索引 (可以是复杂的模板元程序) 与索引列表的应用程序可以分离开来。在索引列表中，运行时效率很重要。标准类型 std::integer_sequence(C++14 中引入) 用于表示索引列表。

25.3.5 使用索引列表进行反转

要使用索引列表对元组进行反转，首先需要索引列表，其是一个类型列表，包含的值可以用作类型列表或异构数据结构的索引 (参见第 24.4 节)。对于索引列表，将使用 24.3 节中开发的 Valuelist 类型。与上面的元组反转示例对应的索引列表是

```
1 Valuelist<unsigned, 4, 3, 2, 1, 0>
```

要如何生成这个索引列表？一种方法是使用简单的模板元程序 MakeIndexList 生成一个从 0 到 N - 1(包括) 的索引列表，其中 N 是一个元组的长度：

C++14 提供了一个类似的模板 make_index_sequence，生成了 std::size_t 类型的索引列表，以及更通用的 make_integer_sequence，其允许选择特定的类型。

typelist/makeindexlist.hpp

```
1 // recursive case
2 template<unsigned N, typename Result = Valuelist<unsigned>>
3 struct MakeIndexListT
4 : MakeIndexListT<N-1, PushFront<Result, CTValue<unsigned, N-1>>>
5 {
6 };
7
8 // basis case
9 template<typename Result>
10 struct MakeIndexListT<0, Result>
11 {
12     using Type = Result;
13 };
14
15 template<unsigned N>
16 using MakeIndexList = typename MakeIndexListT<N>::Type;
```

可以将此操作与类型列表 Reverse 组合，以生成相应的索引列表：

```
1 using MyIndexList = Reverse<MakeIndexList<5>>;
2 // equivalent to Valuelist<unsigned, 4, 3, 2, 1, 0>
```

要执行反转，需要将索引列表中的索引捕获到非类型参数包中。这通过将索引集元组的 reverse() 算法实现分成两部分来处理的：

typelist/indexlistreverse.hpp

```
1 template<typename... Elements, unsigned... Indices>
2 auto reverseImpl(Tuple<Elements...> const& t,
3 Valuelist<unsigned, Indices...>)
4 {
5     return makeTuple(get<Indices>(t)...);
6 }
7
8 template<typename... Elements>
9 auto reverse(Tuple<Elements...> const& t)
10 {
11     return reverseImpl(t,
12     Reverse<MakeIndexList<sizeof...(Elements)>>());
13 }
```

在 C++11 中，返回类型必须声明为

```
1 -> decltype(makeTuple(get<Indices>(t)...))
```

和

```
1 -> decltype(reverseImpl(t, Reverse<MakeIndexList<sizeof...(Elements)>>()))
```

reverseImpl() 函数模板将 Valuelist 参数中的索引捕获到参数包 indices 中，然后返回调用 makeTuple() 的结果，其参数通过使用捕获的索引集（对元组调用 get() 获得）。

reverse() 算法本身仅形成相应的索引集，并将其提供给 reverseImpl 算法。索引作为模板元程序进行操作，不会产生运行时代码。唯一的运行时代码会在 reverseImpl 中生成，使用 makeTuple() 构造产生元组，因此元组元素只复制一次。

25.3.6 打乱和选择

上一节中用于形成反向元组的 reverseImpl() 函数模板，实际上不包含与 reverse() 操作相关的代码。相反，它只是从现有元组中选择一组特定的索引，并使用它们来组成一个新的元组。Reverse() 提供了一组反向索引，但许多算法可以建立在这个核心元组的 select() 算法基础上：

C++11 中，返回类型必须声明为 `->decltype(makeTuple(get<indices>(t)))`。

typelist/select.hpp

```

1 template<typename... Elements, unsigned... Indices>
2 auto select(Tuple<Elements...> const& t,
3             Valuelist<unsigned, Indices...>)
4 {
5     return makeTuple(get<Indices>(t)...);
6 }

```

元组 `splat` 操作是建立在 `select()` 上的一个简单算法，其接受元组中的单个元素，将其复制并创建另一个元组，该元组具有该元素的一定数量的副本：

```

1 Tuple<int, double, std::string> t1(42, 7.7, "hello");
2 auto a = splat<1, 4>(t);
3 std::cout << a << '\n';

```

会生成一个 `Tuple<double, double, double, double>`，其中每个值都是 `get<1>(t)` 的副本，所以会输出

```
1 (7.7, 7.7, 7.7, 7.7)
```

给定元程序来生成“复制”索引集，该索引集由值 I 的 N 个副本组成，`splat()` 是 `select()` 的直接应用：

C++11 中，`splat()` 的返回类型必须声明为`->decltype(return-expression)`。

typelist/splat.hpp

```

1 template<unsigned I, unsigned N, typename IndexList = Valuelist<unsigned>>
2 class ReplicatedIndexList;
3
4 template<unsigned I, unsigned N, unsigned... Indices>
5 class ReplicatedIndexListT<I, N, Valuelist<unsigned, Indices...>>
6 : public ReplicatedIndexListT<I, N-1,
7           Valuelist<unsigned, Indices..., I>> {
8 }
9
10 template<unsigned I, unsigned... Indices>
11 class ReplicatedIndexListT<I, 0, Valuelist<unsigned, Indices...>> {
12 public:
13     using Type = Valuelist<unsigned, Indices...>;
14 }
15
16 template<unsigned I, unsigned N>
17 using ReplicatedIndexList = typename ReplicatedIndexListT<I, N>::Type;
18
19 template<unsigned I, unsigned N, typename... Elements>
20 auto splat(Tuple<Elements...> const& t)
21 {
22     return select(t, ReplicatedIndexList<I, N>());

```

23 }

即使是复杂的元组算法，也可以通过索引列表上的模板元程序和 `select()` 来实现，可以使用第 24.2.7 节中开发的插入排序，根据元素类型大小对元组进行排序。给定这样一个 `sort()` 函数，其能接受一个比较元组元素类型的模板元函数进行比较操作，这里可以按类型大小对元组元素进行排序：

typelist/tuplesorttest.hpp

```

1 #include <complex>
2
3 template<typename T, typename U>
4 class SmallerThanT
5 {
6     public:
7         static constexpr bool value = sizeof(T) < sizeof(U);
8     };
9
10 void testTupleSort()
11 {
12     auto t1 = makeTuple(17LL, std::complex<double>(42, 77), 'c', 42, 7.7);
13     std::cout << t1 << '\n';
14     auto t2 = sort<SmallerThanT>(t1); // t2 is Tuple<int, long, std::string>
15     std::cout << "sorted by size: " << t2 << '\n';
16 }
```

输出可能如下所示：

```

1 (17, (42, 77), c, 42, 7.7)
2 sorted by size: (c, 42, 7.7, 17, (42, 77))
```

结果的顺序取决于特定于平台。例如，`double` 的大小可能比 `long long` 的大小更小、相同或更大

`sort()` 实现包括使用 `InsertionSort` 和元组 `select()`：

C++11 中，`sort()` 的返回类型必须声明为`->decltype(return-expression)`。

tuples/tuplesort.hpp

```

1 // metafunction wrapper that compares the elements in a tuple:
2 template<typename List, template<typename T, typename U> class F>
3 class MetafunOfNthElementT {
4     public:
5         template<typename T, typename U> class Apply;
6         template<unsigned N, unsigned M>
7         class Apply<CTValue<unsigned, M>, CTValue<unsigned, N>>
8             : public F<NthElement<List, M>, NthElement<List, N>> { };
```

```

9 } ;
10
11 // sort a tuple based on comparing the element types:
12 template<template<typename T, typename U> class Compare,
13 typename... Elements>
14 auto sort(Tuple<Elements...> const& t)
15 {
16     return select(t,
17         InsertionSort<MakeIndexList<sizeof... (Elements)>,
18         MetafunOfNthElementT<
19             Tuple<Elements...>,
20             Compare>::template Apply>());
21 }

```

仔细查看 InsertionSort 的使用: 要排序的实际类型列表是类型列表中的索引列表, 并使用 MakeIndexList<> 构造, 插入排序的结果是元组中的一组索引, 然后将其提供给 select()。但因为 InsertionSort 对索引进行操作, 所以比较操作是比较两个索引。当考虑 std::vector 的某种索引时, 这个更容易理解, 如下(非元编程)示例:

tuples/indexsort.hpp

```

1 #include <vector>
2 #include <algorithm>
3 #include <string>
4
5 int main()
6 {
7     std::vector<std::string> strings = {"banana", "apple", "cherry"};
8     std::vector<unsigned> indices = { 0, 1, 2 };
9     std::sort(indices.begin(), indices.end(),
10        [&strings] (unsigned i, unsigned j) {
11            return strings[i] < strings[j];
12        });
13 }

```

索引包含 vector 字符串的索引, sort() 操作对实际的索引进行排序, 因此作为比较操作提供的 Lambda 接受两个无符号值(而不是字符串值)。但 Lambda 函数体可以使用无符号值作为字符串 vector 的索引, 因此排序实际上是根据字符串内容进行的。排序的最后, 索引提供字符串的索引, 并根据字符串中的值排序。

我们对元组 sort() 的 InsertionSort 使用了相同的方法。适配器模板 MetafunOfNthElementT 提供了模板元函数(其嵌套的 Apply), 该函数接受两个索引(CTValue 特化), 并使用 NthElement 从类型列表参数中提取相应的元素。成员模板 Apply “捕获”了提供给外围模板(MetafunOfNthElementT)的类型列表, 与 Lambda 从其外围作用域捕获字符串 vector 的方式相同。然后, Apply 将提取的元素类型转发给底层元函数 F, 完成调整。

排序的所有计算都在编译时执行, 直接形成结果元组, 在运行时没有多余的值复制。

25.4. 扩展

元组用于将一组相关值存储到单个值中，而不管这些值是什么类型，或有多少个相关值。某些情况下，可能需要解包这样的元组，例如：将其元素作为单独的参数传递给函数。举个简单的例子，可以取一个元组，并将其元素传递给变量 print() 操作，如第 12.4 节所述：

```
1 Tuple<std::string, char const*, int, char> t("Pi", "is roughly",
2 3, '\n');
3 print(t...); // ERROR: cannot expand a tuple; it isn't a parameter pack
```

如示例中所述，因为不是参数包，所以解包元组的尝试不会成功。可以使用索引列表实现相同的方法。下面的函数模板 apply() 接受一个函数和一个元组，然后用未打包的元组元素调用该函数：

tuples/apply.hpp

```
1 template<typename F, typename... Elements, unsigned... Indices>
2 auto applyImpl(F f, Tuple<Elements...> const& t,
3                 Valuelist<unsigned, Indices...>)
4             ->decltype(f(get<Indices>(t...)))
5 {
6     return f(get<Indices>(t...));
7 }
8
9 template<typename F, typename... Elements,
10         unsigned N = sizeof...(Elements)>
11 auto apply(F f, Tuple<Elements...> const& t)
12     ->decltype(applyImpl(f, t, MakeIndexList<N>()))
13 {
14     return applyImpl(f, t, MakeIndexList<N>());
15 }
```

applyImpl() 函数模板接受给定的索引列表，并将元组中的元素展开为其函数对象参数 f 的参数列表。面向用户的 apply() 只负责构造初始索引列表，其允许将一个元组扩展为 print() 的参数：

```
1 Tuple<std::string, char const*, int, char> t("Pi", "is roughly",
2 3, '\n');
3 apply(print, t); // OK: prints Pi is roughly 3
```

C++17 提供了一个类似的函数，可用于类元组类型。

25.5. 优化

元组是基本的异构容器，有必要考虑如何在运行时（存储、执行时）和编译时（模板实例化数量）优化元组的使用。本节讨论对 Tuple 实现的一些特定优化。

25.5.1 元组和 EBCO

元组存储公式使用了更多的存储空间。一个问题是尾部成员最终将是一个空元组，每个非空元组都以一个空元组结束，而且数据成员必须始终有至少一个字节的存储空间。

为了提高元组的存储效率，可以通过继承尾元组而不是使其成为成员来应用 21.1 节中讨论的空基类优化 (EBCO):

tuples/tuplestorage1.hpp

```
1 // recursive case:  
2 template<typename Head, typename... Tail>  
3 class Tuple<Head, Tail...> : private Tuple<Tail...>  
4 {  
5     private:  
6     Head head;  
7     public:  
8     Head& getHead() { return head; }  
9     Head const& getHead() const { return head; }  
10    Tuple<Tail...>& getTail() { return *this; }  
11    Tuple<Tail...> const& getTail() const { return *this; }  
12};
```

这与我们在第 21.1.2 节中使用的 BaseMemberPair 方法相同。但有一个副作用，即会在构造函数中颠倒元组元素初始化的顺序。以前，因为头部成员位于尾部成员之前，所以会先初始化头部成员。这种新的元组存储方式中，尾部位于基类中，因此将在成员头部之前初始化。

此更改的另一个影响是，因为基类通常存储在成员之前，所以元组的元素最终将以相反的顺序存储。

这个问题可以通过将头部成员嵌入到基类列表中，位于尾部之前的基类来解决。直接实现将引入一个 TupleElt 模板，用于封装每个元素类型，以便 Tuple 可以继承:

tuples/tuplestorage2.hpp

```
1 template<typename... Types>  
2 class Tuple;  
3  
4 template<typename T>  
5 class TupleElt  
6 {  
7     T value;  
8  
9     public:  
10    TupleElt() = default;  
11    template<typename U>  
12    TupleElt(U&& other) : value(std::forward<U>(other)) {}  
13  
14    T& get() { return value; }  
15    T const& get() const { return value; }  
16};  
17  
18 // recursive case:
```

```

19 template<typename Head, typename... Tail>
20 class Tuple<Head, Tail...>
21 : private TupleElt<Head>, private Tuple<Tail...>
22 {
23     public:
24     Head& getHead() {
25         // potentially ambiguous
26         return static_cast<TupleElt<Head> *>(this)->get();
27     }
28     Head const& getHead() const {
29         // potentially ambiguous
30         return static_cast<TupleElt<Head> const*>(this)->get();
31     }
32     Tuple<Tail...>& getTail() { return *this; }
33     Tuple<Tail...> const& getTail() const { return *this; }
34 };
35
36 // basis case:
37 template<>
38 class Tuple<> {
39     // no storage required
40 };

```

虽然这种方法解决了初始化排序问题，但也引入了一个新的（更糟糕的）问题：不能再从具有两个相同类型元素的元组中提取元素，例如 `tuple<int, int>`，从元组到该类型的元组的派生到基的转换（例如 `TupleElt<int>`）将有歧义。

为了打破这种模糊性，需要确保每个 `TupleElt` 基类在给定的 `Tuple` 中唯一。一种方法是在其元组中编码该值的“高度”，即尾元组的长度。元组中最后一个元素的高度为 0，倒数第二个元素的高度为 1，依此类推：

更直观的做法是简单地使用元组元素的索引，而不是高度。因为给定的元组既可能作为独立元组出现，也可能作为另一个元组的尾部出现，所以这些信息在元组中并不容易获得。然而，给定的元组确实知道自己的尾部有多少元素。

tuples/tupleelt1.hpp

```

1 template<unsigned Height, typename T>
2 class TupleElt {
3     T value;
4     public:
5     TupleElt() = default;
6
7     template<typename U>
8     TupleElt(U&& other) : value(std::forward<U>(other)) { }
9
10    T& get() { return value; }
11    T const& get() const { return value; }

```

```
12 };
```

通过这个解决方案，可以产生一个 Tuple，其使用 EBCO，同时保持初始化顺序，并支持同一类型的多个元素：

tuples/tuplestorage3.hpp

```
1 template<typename... Types>
2 class Tuple;
3
4 // recursive case:
5 template<typename Head, typename... Tail>
6 class Tuple<Head, Tail...>
7 : private TupleElt<sizeof...(Tail), Head>, private Tuple<Tail...>
8 {
9     using HeadElt = TupleElt<sizeof...(Tail), Head>;
10    public:
11        Head& getHead() {
12            return static_cast<HeadElt*>(this)->get();
13        }
14        Head const& getHead() const {
15            return static_cast<HeadElt const*>(this)->get();
16        }
17        Tuple<Tail...>& getTail() { return *this; }
18        Tuple<Tail...> const& getTail() const { return *this; }
19    };
20
21 // basis case:
22 template<>
23 class Tuple<> {
24     // no storage required
25 };
```

有了这个实现，再看看下面的程序：

tuples/compressedtuple1.cpp

```
1 #include <algorithm>
2 #include "tupleelt1.hpp"
3 #include "tuplestorage3.hpp"
4 #include <iostream>
5
6 struct A {
7     A() {
8         std::cout << "A()" << '\n';
9     }
10};
11
12 struct B {
13     B() {
```

```

14     std::cout << "B()" << '\n' ;
15 }
16 };
17
18 int main()
19 {
20     Tuple<A, char, A, char, B> t1;
21     std::cout << sizeof(t1) << " bytes" << '\n' ;
22 }
```

输出为

```

1 A()
2 A()
3 B()
4 5 bytes
```

EBCO 删除了一个字节 (对于空元组 `Tuple<>`)。但 A 和 B 都是空类，这意味着在 `Tuple` 中有更多的机会使用 EBCO。`TupleElt` 可以进行扩展，在安全的情况下继承元素类型，不需要更改 `Tuple`:

tuples/tupleelt2.hpp

```

1 #include <type_traits>
2
3 template<unsigned Height, typename T,
4          bool = std::is_class<T>::value && !std::is_final<T>::value>
5 class TupleElt;
6
7 template<unsigned Height, typename T>
8 class TupleElt<Height, T, false>
9 {
10     T value;
11
12     public:
13     TupleElt() = default;
14     template<typename U>
15         TupleElt(U&& other) : value(std::forward<U>(other)) { }
16
17     T& get() { return value; }
18     T const& get() const { return value; }
19 };
20
21 template<unsigned Height, typename T>
22 class TupleElt<Height, T, true> : private T
23 {
24     public:
25     TupleElt() = default;
26     template<typename U>
27         TupleElt(U&& other) : T(std::forward<U>(other)) { }
```

```

29 T& get() { return *this; }
30 T const& get() const { return *this; }
31 };

```

当 TupleElt 是一个非 final 类时，会私有地继承该类，以允许 EBCO 可以用于存储值。有了这个修改，程序现在的输出为

```

1 A()
2 A()
3 B()
4 2 bytes

```

25.5.2 常量时间的 get()

使用元组时，get() 操作非常常见，但它的递归实现需要线性数量的模板实例化，这会影响编译时间。在上一节介绍的 EBCO 优化已经实现了更有效的 get 实现，我们将在这里进行描述。

关键的理解是模板参数推导（第 15 章），当形参（基类类型）与实参（派生类类型）匹配时，需要推导基类的模板实参。若可以计算想要提取的元素的高度 H，就可以靠从 Tuple 特化到 TupleElt<H, T>（其中 T 需要进行推导）的转换来提取，而无需遍历所有索引：

tuples/constantget.hpp

```

1 template<unsigned H, typename T>
2 T& getHeight(TupleElt<H, T>& te)
3 {
4     return te.get();
5 }
6
7 template<typename... Types>
8 class Tuple;
9
10 template<unsigned I, typename... Elements>
11 auto get(Tuple<Elements...>& t)
12     -> decltype(getHeight<sizeof...(Elements)-I-1>(t))
13 {
14     return getHeight<sizeof...(Elements)-I-1>(t);
15 }

```

因为 get<I>(t) 接收所需元素的索引 I（从元组的开始计数），而元组的实际存储是根据高度 H（从元组的结束计数）计算的，所以可以通过 I 计算 H。调用 getHeight() 的模板参数推导执行实际的搜索：高度 H 是固定的，因为可以在调用中显式地提供，所以只有一个 TupleElt 基类匹配，从而可以推导出类型 T。注意，必须将 getHeight() 声明为 Tuple 的友元，从而允许转换为私有基类。例如：

```

1 // inside the recursive case for class template Tuple:
2 template<unsigned I, typename... Elements>
3 friend auto get(Tuple<Elements...>& t)
4     -> decltype(getHeight<sizeof...(Elements)-I-1>(t));

```

因为已经将复杂的索引匹配工作转移到编译器模板推导那里，所以这里的实现方式只需要常数数量的模板实例化。

25.6. 下标

原则上，可以定义 `operator[]` 来访问元组的元素。然而，与 `std::vector` 不同，元组的元素类型可以有不同，因此元组的 `operator[]` 必须是一个模板，其结果类型根据元素的索引不同而不同。这要求每个索引具有不同的类型，因此可以使用索引的类型来确定元素类型。

在第 24.3 节中介绍的类模板 `CTValue`，可以在类型中编码数字索引，可以用其将 `operator[]` 定义为 `Tuple` 的成员：

```
1 template<typename T, T Index>
2 auto& operator[](CTValue<T, Index>) {
3     return get<Index>(*this);
4 }
```

这里，在 `CTValue` 参数的类型中传递索引值来进行相应的 `get<>()` 调用。

可以这样使用这个类：

```
1 auto t = makeTuple(0, '1', 2.2f, std::string("hello"));
2 auto a = t[CTValue<unsigned, 2>{}];
3 auto b = t[CTValue<unsigned, 3>{}];
```

`a` 和 `b` 将由 `Tuple t` 中的第三和第四个值的类型和值初始化。

为了更方便地使用常量索引，可以使用 `constexpr` 实现字面操作符，从后缀为 `_c` 的文字直接计算编译时的数字：

tuples/literals.hpp

```
1 #include "ctvalue.hpp"
2 #include <cassert>
3 #include <cstddef>
4 // convert single char to corresponding int value at compile time:
5 constexpr int toInt(char c) {
6     // hexadecimal letters:
7     if (c >= 'A' && c <= 'F') {
8         return static_cast<int>(c) - static_cast<int>('A') + 10;
9     }
10    if (c >= 'a' && c <= 'f') {
11        return static_cast<int>(c) - static_cast<int>('a') + 10;
12    }
13    // other (disable '.' for floating-point literals):
14    assert(c >= '0' && c <= '9');
15    return static_cast<int>(c) - static_cast<int>('0');
16 }
17
18 // parse array of chars to corresponding int value at compile time:
19 template<std::size_t N>
```

```

20 constexpr int parseInt(char const (&arr) [N]) {
21     int base = 10; // to handle base (default: decimal)
22     int offset = 0; // to skip prefixes like 0x
23     if (N > 2 && arr[0] == '0') {
24         switch (arr[1]) {
25             case 'x': // prefix 0x or 0X, so hexadecimal
26             case 'X':
27                 base = 16;
28                 offset = 2;
29                 break;
30             case 'b': // prefix 0b or 0B (since C++14), so binary
31             case 'B':
32                 base = 2;
33                 offset = 2;
34                 break;
35             default: // prefix 0, so octal
36                 base = 8;
37                 offset = 1;
38                 break;
39         }
40     }
41     // iterate over all digits and compute resulting value:
42     int value = 0;
43     int multiplier = 1;
44     for (std::size_t i = 0; i < N - offset; ++i) {
45         if (arr[N-1-i] != '\'' ) { // ignore separating single quotes (e.g. in 1' 000)
46             value += toInt(arr[N-1-i]) * multiplier;
47             multiplier *= base;
48         }
49     }
50     return value;
51 }
52
53 // literal operator: parse integral literals with suffix _c as sequence of chars:
54 template<char... cs>
55 constexpr auto operator"" _c() {
56     return CTValue<int, parseInt<sizeof... (cs)>({cs...})>{};
57 }
```

对于数字字面值，可以使用字面值操作符来推导出字面值的每个字符作为模板参数（详细信息请参阅第 15.5.1 节）。可以将字符传递给 constexpr 辅助函数 `parseInt()`，该函数在编译时计算字符串的值，并将其生成为 `CTValue`。例如：

- `42_c` 生成 `CTValue<int,42>`
- `0x815_c` 生成 `CTValue<int,2069>`
- `0b1111'1111_c` 生成 `CTValue<int,255>`

C++14 开始，就支持二进制字面值的前缀 `0b` 和分隔数字的单引号字符。

解析器不处理浮点字面值，断言会导致编译时错误，这是一个不能在编译时上下文中使用的运行时特性。

现在，就可以这样使用元组了：

```
1 auto t = makeTuple(0, '1', 2.2f, std::string("hello"));
2 auto c = t[2_c];
3 auto d = t[3_c];
```

Boost.Hana(参见 [boosthana]，适合对类型和值进行计算的元编程库) 就使用了这种方法。

25.7. 后记

元组构造模板的实现，是对模板应用的一个实例。Boost.Tuple 库 [BoostTuple] 是 C++ 中最流行的一种元组的实现方式，并最终发展成 C++11 中的 std::tuple。

C++11 之前，许多元组的实现都是基于递归对结构的思想，本书的第一版 [VandevoordeJosuttisTemplates1st] 通过“递归组合”阐明了这种方法。Andrei Alexandrescu 在 [AlexandrescuDesign] 中提出了一个有趣的替代方案，他用类型列表的概念(如第 24 章所讨论的)作为元组的基础，将元组中的类型列表和字段列表清晰地分离开来。

C++11 引入了可变参数模板，其中参数包可以捕获元组的类型列表，消除了递归对的需要。包扩展和索引列表的概念 [GregorJarviPowellVariadicTemplates] 使递归模板实例化为更为简单、更有效的模板实例，从而使元组的使用门槛更低。索引列表对元组和类型列表算法的性能具有关键性影响，以至于编译器有一个内部别名模板，如 __make_integer_seq<S, T, N>，会扩展为 S<T, 0, 1, ..., N>，不需要额外的模板实例化，从而让 std::make_index_sequence 和 make_integer_sequence 使用起来更简单。

Tuple 是使用最广泛的异构容器，但它不唯一。Boost.Fusion 库 [BoostFusion] 为通用容器提供了异构对应，如异构 list、deque、set 和 map。提供了一个为异构集合编写算法的框架，使用与 C++ 标准库本身相同的抽象类型和术语(例如，迭代器、序列和容器)。

Boost.Hana[BoostHana] 采纳了 Boost 中出现的许多想法。MPL Boost.MPL[BoostMPL] 和 Boost.Fusion，在 C++11 实现之前就设计和实现了，并且用 C++11(和 C++14) 新的语言特性重新进行了设计，从而产生了一个优雅的库，其为异构计算提供了强大的和可组合的组件。

第 26 章 可辨识联合

前一章中的元组将一些类型列表的值聚合为单个类值，使它们具有与简单结构体相同的功能。这自然就会想知道，联合的对应类型是什么：其将包含单个值，但该值将具有从一些可能类型中选择的类型。例如，一个数据库字段可能包含一个整数、浮点值、字符串或二进制对象，但在相应的时间内，其只能表示这些类型中的一种。

本章中，我们开发了一个类模板 Variant，可以动态存储给定的一组值类型中的一个值，类似于 C++17 标准库的 std::variant<>。Variant 是可辨识联合，从而其值的类型是动态的，并提供了比 C++ 联合更好的类型安全性。Variant 本身是一个可变参数模板，可接受动态值可能具有的类型列表。

```
1 Variant<int, double, string> field;
```

可以存储整型、双精度或字符串，但只能存储其中一个值。

类型列表在声明 Variant 时是固定的，所以 Variant 是封闭的可辨识联合。开放的可辨识联合允许在创建时不知道有哪些类型的值存储在联合中。第 22 章中讨论的 FunctionPtr 类可以看作是一种开放可识别联合。

下面的程序演示了 Variant 的行为：

variant/variant.cpp

```
1 #include "variant.hpp"
2 #include <iostream>
3 #include <string>
4
5 int main()
6 {
7     Variant<int, double, std::string> field(17);
8     if (field.is<int>()) {
9         std::cout << "Field stores the integer "
10        << field.get<int>() << '\n';
11    }
12    field = 42; // assign value of same type
13    field = "hello"; // assign value of different type
14    std::cout << "Field now stores the string '"
15        << field.get<std::string>() << "'\n";
16 }
```

输出：

```
Field stores the integer 17
Field now stores the string "hello"
```

可以赋值给 Variant 任何类型的值，可以使用成员函数 `is<T>()` 来测试 Variant 当前是否包含类型为 T 的值，然后使用成员函数 `get<T>()` 获取存储值。

26.1. 存储

我们的 Variant 类型的主要设计是为了管理动态值的存储，也就是当前存储在 Variant 中的值。不同的类型可能需要考虑不同的大小和对齐方式。此外，该变体还需要存储一个辨别器，来指示哪些可能的类型是动态值的类型。一种简单的（尽管低效）存储机制直接使用元组（参见第 25 章）：

variant/variantstorageastuple.hpp

```
1 template<typename... Types>
2 class Variant {
3     public:
4         Tuple<Types...> storage;
5         unsigned char discriminator;
6     };

```

标识符充当元组的动态索引。只有静态索引等于当前 `discriminator` 值的元组元素才有效。所以当 `discriminator` 为 0 时，`get<0>(storage)` 提供了对动态值的访问；当 `discriminator` 为 1 时，`get<1>(storage)` 提供对动态值的访问，以此类推。

可以构建核心操作 `is<T>()`，并在元组上使用 `get<T>()`。这样做非常低效，因为即使一次只有一个动态值，Variant 本身现在需要的存储等于所有可能值类型的大小之和。

这种方法还有许多其他问题，比如要求 Types 中的所有类型都有默认构造函数。

更好的方法是将每个可能类型的存储重叠，可以通过递归地解开 Variant 的头和尾来实现，但这里使用的是联合，不是类：

variant/variantstorageasunion.hpp

```
1 template<typename... Types>
2 union VariantStorage;
3
4 template<typename Head, typename... Tail>
5 union VariantStorage<Head, Tail...> {
6     Head head;
7     VariantStorage<Tail...> tail;
8 };
9
10 template<>
11 union VariantStorage<> {
12 };

```

联合保证有足够的大小和对齐方式，以允许在相应时间内存储 Types 中的类型。但联合本身就很难处理，因为实现 Variant 将使用继承，这是联合不允许的。

不过，我们改变了 Variant 存储的底层表示：足够大的字符数组，以容纳任何类型，并对任何类型具有适当的对齐方式，我们将其用作存储活动值的缓冲区。VariantStorage 类模板实现了这个缓冲区和一个辨别器：

variant/variantstorage.hpp

```
1 #include <new> // for std::launder()
2
3 template<typename... Types>
4 class VariantStorage {
5     using LargestT = LargestType<Typelist<Types...>>;
6     alignas(Types...) unsigned char buffer[sizeof(LargestT)];
7     unsigned char discriminator = 0;
8
9     public:
10    unsigned char getDiscriminator() const { return discriminator; }
11    void setDiscriminator(unsigned char d) { discriminator = d; }
12
13    void* getRawBuffer() { return buffer; }
14    const void* getRawBuffer() const { return buffer; }
15
16    template<typename T>
17        T* getBufferAs() { return std::launder(reinterpret_cast<T*>(buffer)); }
18    template<typename T>
19        T const* getBufferAs() const {
20            return std::launder(reinterpret_cast<T const*>(buffer));
21        }
21};
```

使用在第 24.2.2 节中开发的 LargestType 元程序来计算缓冲区的大小，确保对于任何类型都足够大。类似地，alignas 包扩展确保缓冲区具有适合任何值类型的对齐方式。

可以使用模板元程序来计算最大对齐数（而不是使用对齐的包扩展）。两种方法的结果都是相同的，可将上面的公式将对齐计算工作移到了编译时。

缓冲区本质上是联合的物理底层表示。可以使用 getBuffer() 访问指向缓冲区的指针，并通过显式类型转换、new(创建新值) 和显式销毁(销毁创建的值) 来操作存储。若不熟悉 getBufferAs() 中使用的 std::launder()，那也没关系，现在只要知道它不修改地返回参数就足够了；我们将在讨论 Variant 模板的赋值操作符时解释其作用（参见第 26.4.3 节）。

26.2. 设计

既然有了变量存储问题的解决方案，就可以设计 Variant 类型了。与 Tuple 类型一样，可以使用继承来为 Types 列表提供每个类型的行为。与 Tuple 不同的是，这些基类不会存储。每个基类都使用 21.2 节中讨论的奇异递归模板模式 (CRTP)，通过派生最多的类型访问共享变量。

下面定义的类模板 VariantChoice 提供了在变量的活动值为 (或将为)T 类型时，对缓冲区进行操作所需的操作：

variant/variantchoice.hpp

```
1 #include "findindexof.hpp"
2 template<typename T, typename... Types>
3 class VariantChoice {
4     using Derived = Variant<Types...>;
5     Derived& getDerived() { return *static_cast<Derived*>(this); }
6     Derived const& getDerived() const {
7         return *static_cast<Derived const*>(this);
8     }
9 protected:
10    // compute the discriminator to be used for this type
11    constexpr static unsigned Discriminator =
12        FindIndexOfT<Typelist<Types...>, T>::value + 1;
13 public:
14     VariantChoice() {}
15     VariantChoice(T const& value); // see variantchoiceinit.hpp
16     VariantChoice(T& value); // see variantchoiceinit.hpp
17     bool destroy(); // see variantchoicedestroy.hpp
18     Derived& operator=(T const& value); // see variantchoiceassign.hpp
19     Derived& operator=(T& value); // see variantchoiceassign.hpp
20 };
```

模板参数包 `Types` 将包含变体中的所有类型，其可以形成 `Derived` 类型 (用于 CRTP)，从而提供向下转换操作 `getDerived()`。类型的第二个有趣用法是，在类型列表中查找特定类型 `T` 的位置，可以通过元函数 `FindIndexOfT` 完成：

variant/findindexof.hpp

```
1 template<typename List, typename T, unsigned N = 0,
2         bool Empty = IsEmpty<List>::value>
3 struct FindIndexOfT;
4
5 // recursive case:
6 template<typename List, typename T, unsigned N>
7 struct FindIndexOfT<List, T, N, false>
8 : public IfThenElse<std::is_same<Front<List>, T>::value,
9     std::integral_constant<unsigned, N>,
10    FindIndexOfT<PopFront<List>, T, N+1>>
11 {
12 };
13
14 // basis case:
15 template<typename List, typename T, unsigned N>
16 struct FindIndexOfT<List, T, N, true>
17 {
18 };
```

该指标值用于计算 `T` 对应的辨别器的值；稍后我们将返回特定的判别值。

Variant 的框架如下，表明了 Variant、VariantStorage 和 VariantChoice 之间的关系：

variant/variant-skel.hpp

```
1 template<typename... Types>
2 class Variant
3 : private VariantStorage<Types...>,
4 private VariantChoice<Types, Types...>...
5 {
6     template<typename T, typename... OtherTypes>
7         friend class VariantChoice; // enable CRTP
8     ...
9 };
```

每个 Variant 都有一个单独的、共享的 VariantStorage 基类。

基类是私有的，不是公共接口的一部分。友元模板需要 VariantChoice 中的 asDerived() 函数，才能向下转换到 Variant。

还有一些 VariantChoice 基类，是由以下嵌套的包扩展产生的（参见第 12.4.4 节）：

```
1 VariantChoice<Types, Types...>...
```

实例中，有两个展开：外层展开，通过展开对类型的第一个引用，为类型中的每个类型 T 生成 VariantChoice 基类；内部展开，展开了第二次出现的 Types，另外还将 Types 中的所有类型传递给每个 VariantChoice 基类。

```
1 Variant<int, double, std::string>
```

会产生下面一组 VariantChoice 基类：

通过类型 T 来区分给定 Variant 的 VariantChoice 基类的效果是，可以防止类型重复。 Variant<double, int, double> 将产生编译时错误，指出类不能直接继承同一基类（在本例中， VariantChoice<double, double, int, double> 会出现两个编译错误）。

```
1 VariantChoice<int, int, double, std::string>,
2 VariantChoice<double, int, double, std::string>,
3 VariantChoice<std::string, int, double, std::string>
```

这三个基类的辨别器值将分别为 1、2 和 3。当变量存储的 discriminator 成员与特定 VariantChoice 基类的 discriminator 匹配时，该基类负责管理动态值。

标识值 0 是为 Variant 不包含值的情况保留，这是一种奇怪的状态，只有在赋值期间，抛出异常时才能观察到。对 Variant 的讨论中，将小心处理为 0 的辨别值（并在适当的时候进行设置），我们将这种情况的讨论留到第 26.4.3 节。

Variant 的完整定义如下所示，下面几节将描述 Variant 每个成员的实现。

variant/variant.hpp

```

1 template<typename... Types>
2 class Variant
3 : private VariantStorage<Types...>,
4   private VariantChoice<Types, Types...>...
5 {
6   template<typename T, typename... OtherTypes>
7     friend class VariantChoice;
8
9   public:
10   template<typename T> bool is() const; // see variantis.hpp
11   template<typename T> T& get() &; // see variantget.hpp
12   template<typename T> T const& get() const&; // see variantget.hpp
13   template<typename T> T&& get() &&; // see variantget.hpp
14
15   // see variantvisit.hpp:
16   template<typename R = ComputedResultType, typename Visitor>
17     VisitResult<R, Visitor, Types&...> visit(Visitor&& vis) &;
18   template<typename R = ComputedResultType, typename Visitor>
19     VisitResult<R, Visitor, Types const&...> visit(Visitor&& vis) const&;
20   template<typename R = ComputedResultType, typename Visitor>
21     VisitResult<R, Visitor, Types&&...> visit(Visitor&& vis) &&;
22
23   using VariantChoice<Types, Types...>::VariantChoice...;
24   Variant(); // see variantdefaultctor.hpp
25   Variant(Variant const& source); // see variantcopyctor.hpp
26   Variant(Variant&& source); // see variantmovector.hpp
27   template<typename... SourceTypes>
28     Variant(Variant<SourceTypes...> const& source); // variantcopyctortmpl.hpp
29   template<typename... SourceTypes>
30     Variant(Variant<SourceTypes...>&& source);
31
32   using VariantChoice<Types, Types...>::operator=...;
33   Variant& operator= (Variant const& source); // see variantcopyassign.hpp
34   Variant& operator= (Variant&& source);
35   template<typename... SourceTypes>
36     Variant& operator= (Variant<SourceTypes...> const& source);
37   template<typename... SourceTypes>
38     Variant& operator= (Variant<SourceTypes...>&& source);
39
40   bool empty() const;
41
42   ~Variant() { destroy(); }
43   void destroy(); // see variantdestroy.hpp
44 };

```

26.3. 值的查询与提取

对于 Variant 类型最基本的查询是，咨询动态值的类型是否是特定的类型 T，并在其类型已知时访问动态值。下面定义的 is() 成员函数，可以确定 Variant 当前是否存储 T 类型的值：

variant/variantis.hpp

```
1 template<typename... Types>
2 template<typename T>
3 bool Variant<Types...>::is() const
4 {
5     return this->getDiscriminator() ==
6     VariantChoice<T, Types...>::Discriminator;
7 }
```

给定变量 v, v.is<int>() 将确定 v 的动态值是否为 int 类型。检查很简单，将变量存储中的 discriminator 与对应 VariantChoice 基类的 Discriminator 值进行比较。

若正在寻找的类型 (T) 在列表中没有找到，VariantChoice 基类将无法进行实例化，因为 FindIndexOfT 将不包含值成员，从而导致 is<T>() 中的 (故意的) 编译失败。这可以防止用户在请求不可能存储在变体中的类型时，报错进行提示。

get() 成员函数提取对存储值的引用。必须提供要提取的类型 (例如，v.get<int>())，并且只有当变量的动态值确实为该类型时才有效：

variant/variantget.hpp

```
1 #include <exception>
2
3 class EmptyVariant : public std::exception {
4 };
5
6 template<typename... Types>
7 template<typename T>
8 T& Variant<Types...>::get() & {
9     if (empty()) {
10        throw EmptyVariant();
11    }
12
13    assert(is<T>());
14    return *this->template getBufferAs<T>();
15 }
```

当 Variant 不存储值 (标识值是 0) 时，get() 会抛出 EmptyVariant 异常。由于异常，discriminator 可以是 0，在第 26.4.3 节中描述。从错误类型的 Variant 获取值的尝试，会通过断言进行检查。

26.4. 元素初始化、赋值和销毁

当动态值的类型为 T 时，每个 VariantChoice 基类负责处理初始化、赋值和销毁。本节通过填充 VariantChoice 类模板的详细信息，来实现这些操作。

26.4.1 初始化

从变量所存储的类型之一的值开始初始化变量，用双精度值初始化 Variant<int, double, string>，可以通过 VariantChoice 的构造函数完成，它接受 T 类型的值：

variant/variantchoiceinit.hpp

```
1 #include <utility> // for std::move()
2
3 template<typename T, typename... Types>
4 VariantChoice<T, Types...>::VariantChoice(T const& value) {
5     // place value in buffer and set type discriminator:
6     new(getDerived().getRawBuffer()) T(value);
7     getDerived().setDiscriminator(Discriminator);
8 }
9
10 template<typename T, typename... Types>
11 VariantChoice<T, Types...>::VariantChoice(T&& value) {
12     // place moved value in buffer and set type discriminator:
13     new(getDerived().getRawBuffer()) T(std::move(value));
14     getDerived().setDiscriminator(Discriminator);
15 }
```

构造函数都使用 CRTP 操作 getDerived() 访问共享缓冲区，然后执行 new，用类型 T 的新值初始化存储。第一个构造函数复制构造传入值，而第二个构造函数移动构造传入值。

这里的构造使用，阻止了在 Variant 设计中使用引用类型。这个限制可以通过在类中包装引用来自解决，比如 std::reference_wrapper。

构造函数设置辨别值来指示变量存储的(动态)类型。

最终目标是能够用任何类型的值初始化变量，甚至是隐式转换。

```
1 Variant<int, double, string> v("hello"); // implicitly converted to string
```

为了实现这一点，通过引入 using 声明，将 VariantChoice 构造函数继承为 Variant 本身

在 using 声明(第 4.4.5 节)中介绍了包扩展的使用。C++17 前，继承这些构造函数需要递归继承模式，类似于第 25 章中展示的 Tuple 公式。

```
1 using VariantChoice<Types, Types...>::VariantChoice...;
```

这个 using 声明产生了 Variant 构造函数，从 Types 中的每个类型 T 复制或移动。对于 Variant<int, double, string>，构造函数实际上是：

```
1 Variant(int const&);
2 Variant(int&&);
3 Variant(double const&);
4 Variant(double&&);
5 Variant(string const&);
6 Variant(string&&);
```

26.4.2 销毁

初始化 Variant 时，将在其缓冲区中构造一个值。destroy 操作为该值的销毁过程：

variant/variantchoicedestroy.hpp

```
1 template<typename T, typename... Types>
2 bool VariantChoice<T, Types...>::destroy() {
3     if (getDerived().getDiscriminator() == Discriminator) {
4         // if type matches, call placement delete:
5         getDerived().template getBufferAs<T>() ->~T();
6         return true;
7     }
8     return false;
9 }
```

当辨别器匹配时，通过使用-> T() 调用适当的析构函数，可以显式地销毁缓冲区中的内容。 VariantChoice::destroy() 操作只有在辨别器匹配时才有用。但我们通常希望销毁存储在 Variant 中的值，而不考虑当前的类型。因此，Variant::destroy() 在基类中调用 VariantChoice::destroy() 的操作：

variant/variantdestroy.hpp

```
1 template<typename... Types>
2 void Variant<Types...>::destroy() {
3     // call destroy() on each VariantChoice base class; at most one will succeed:
4     bool results[] = {
5         VariantChoice<Types, Types...>::destroy() ...
6     };
7     // indicate that the variant does not store a value
8     this->setDiscriminator(0);
9 }
```

结果初始化式中的包展开确保对每个 VariantChoice 基类调用了 destroy。这些调用中最多有一个会成功 (匹配辨别器的那个)，而 Variant 为空。通过将辨别器的值设置为 0 来表示空状态。

数组结果本身只是提供了一个使用初始化列表的上下文；它的实际值会忽略。C++17 中，可以使用折叠表达式 (在第 12.4.6 节中讨论过) 来消除这个变量：

variant/variantdestroy17.hpp

```

1 template<typename... Types>
2 void Variant<Types...>::destroy()
3 {
4     // call destroy() on each VariantChoice base class; at most one will succeed:
5     (VariantChoice<Types, Types...>::destroy(), ...);
6
7     // indicate that the variant does not store a value
8     this->setDiscriminator(0);
9 }

```

26.4.3 赋值

赋值构建在初始化和销毁的基础上，如赋值操作符所示：

variant/variantchoiceassign.hpp

```

1 template<typename T, typename... Types>
2 auto VariantChoice<T, Types...>::operator=(T const& value) -> Derived& {
3     if (getDerived().getDiscriminator() == Discriminator) {
4         // assign new value of same type:
5         *getDerived().template getBufferAs<T>() = value;
6     }
7     else {
8         // assign new value of different type:
9         getDerived().destroy(); // try destroy() for all types
10        new(getDerived().getRawBuffer()) T(value); // place new value
11        getDerived().setDiscriminator(Discriminator);
12    }
13    return getDerived();
14 }
15
16 template<typename T, typename... Types>
17 auto VariantChoice<T, Types...>::operator=(T&& value) -> Derived& {
18     if (getDerived().getDiscriminator() == Discriminator) {
19         // assign new value of same type:
20         *getDerived().template getBufferAs<T>() = std::move(value);
21     }
22     else {
23         // assign new value of different type:
24         getDerived().destroy(); // try destroy() for all types
25         new(getDerived().getRawBuffer()) T(std::move(value)); // place new value
26         getDerived().setDiscriminator(Discriminator);
27     }
28     return getDerived();
29 }

```

与从一种存储值类型进行初始化一样，每个 VariantChoice 提供一个赋值操作符，该操作符将其存储的值类型复制（或移动）到变量的存储中。这些赋值操作符由 Variant 通过以下 using 声明继承：

```
1 using VariantChoice<Types, Types...>::operator=...;
```

赋值操作符的实现有两条路径。若该变体已经存储了给定类型 T 的值(由辨别器匹配),那么赋值操作符将根据需要,直接将类型 T 的值复制赋值或移动赋值到缓冲区中,辨别器不变。

若 Variant 没有存储 T 类型的值,赋值需要两个步骤:使用 Variant:: Destroy() 销毁当前值,然后使用 new 初始化 T 类型的新值,并设置辨别器。

使用 new 的两步赋值有三个常见问题,我们必须考虑:

- 自赋值
- 异常
- std::launder()

自赋值

由于像下面这样的表达式,变量 v 可以自赋值:

```
1 v = v.get<T>()
```

上面实现的两步过程中,源值将在复制之前销毁,这可能会导致内存损坏。不过,自赋值意味着辨别器匹配,所以这样的代码将调用 T 的赋值操作符。

异常

若现有值的销毁完成,但 new 初始化引发异常,变量的状态是什么?我们的实现中,Variant::destroy() 将标识符的值重置为 0。非异常情况下,将在初始化完成后适当地设置辨别器。在初始化新值期间发生异常时,标识符保持 0,表示该变量不存储值。我们的设计中,这是产生无值 Variant 的唯一方法。

下面的程序演示了如何通过复制构造函数,抛出的类型的值来生成无值 Variant:

variant/variantexception.cpp

```
1 include "variant.hpp"
2 #include <exception>
3 #include <iostream>
4 #include <string>
5
6 class CopiedNonCopyable : public std::exception
7 {
8 };
9
10 class NonCopyable
11 {
12 public:
13     NonCopyable() {
14     }
15
16     NonCopyable(NonCopyable const&) {
```

```

17     throw CopiedNonCopyable();
18 }
19 NonCopyable(NonCopyable&&) = default;
20
21 NonCopyable& operator= (NonCopyable const&) {
22     throw CopiedNonCopyable();
23 }
24
25 NonCopyable& operator= (NonCopyable&&) = default;
26 };
27
28 int main()
29 {
30     Variant<int, NonCopyable> v(17);
31     try {
32         NonCopyable nc;
33         v = nc;
34     }
35     catch (CopiedNonCopyable) {
36         std::cout << "Copy assignment of NonCopyable failed." << '\n' ;
37         if (!v.is<int>() && !v.is<NonCopyable>()) {
38             std::cout << "Variant has no value." << '\n' ;
39         }
40     }
41 }
```

输出为:

```

Copy assignment of NonCopyable failed.
Variant has no value.
```

对没有值的 Variant 访问，无论是通过 get() 还是通过下一节描述的访问者机制，都会抛出 EmptyVariant 异常，并允许程序从这种异常情况中恢复。empty() 成员函数检查变量可以检查，Variant 实例是否处于空状态：

variant/variantempty.hpp

```

1 template<typename... Types>
2 bool Variant<Types...>::empty() const {
3     return this->getDiscriminator() == 0;
4 }
```

两步赋值的第三个问题很微妙，C++ 标准化委员会直到 C++17 标准化结束时才意识到这个问题。下面先简要地解释一下。

std::launder()

C++ 编译器的目标通常是生成高性能的代码，而提高生成代码性能的主要机制可能是避免重复地将数据从内存复制到寄存器。为了做好这一点，编译器必须做出一些假设，其中一个假设是某些类型的数据在其生命周期内不可变。这包括 `const` 数据、引用 (可以初始化，但之后不能修改) 和存储在多态对象中的一些数据存储，这些对象用于分派虚函数、定位虚基类、处理 `typeid` 和 `static_cast` 操作符。

上面两步赋值过程的问题是，偷偷地结束一个对象的生命周期，并以编译器可能无法识别的方式在相同的地方开始另一个对象的生命周期。编译器可能会假设，`Variant` 对象的前一个状态获得的值仍然有效。而实际上，带有 `new` 初始化会使其失效。若不解决，将在获得良好性能而编译时，使用带有不可变数据成员的 `Variant` 类型可能偶尔会产生无效的结果。这样的错误通常很难查 (一部分原因是它们很少发生，部另一分原因是它们在源代码中不可见)。

C++17 后，这个问题的解决方案是通过 `std::laundry()` 访问新对象的地址，只返回实参，但这会导致编译器识别出结果地址指向的对象可能与编译器传递给 `std::laundry()` 的实参不同。但 `std::laundry()` 只修复它返回的地址，而不是传递给 `std::laundry()` 的参数，因为编译器用表达式来表示，而非实际地址 (因为其在运行时才存在)。因此，在构造 `new` 的初始值之后，必须确保接下来的每个访问都进行数据“清洗”，这就是总是“清洗”指向 `Variant` 缓冲区指针的原因。有一些方法可以做得更好 (添加一个指针成员，该成员引用缓冲区，并在每次赋值后通过 `new` 获得“清洗”地址)，但这会以难以维护的方式使代码复杂化。我们的方法只要通过 `getBufferAs()` 成员独占地访问缓冲区，简单又正确。

`std::laundry()` 的并不完全令人满意：很微妙，很难察觉 (直到本书快要出版之前才注意到)，而且很难缓解相关的问题 (`std::laundry()` 不太容易使用)。因此，委员会的几位成员需要在这方面做更多的工作，以找到一个更令人满意的解决办法。请参阅 [JosuttisLaunder] 以获得该问题的更详细描述。

26.5. 访问

`is()` 和 `get()` 成员函数可以检查动态值是否为特定类型，并访问该类型的值。但检查变量中的所有可能类型很快就会变成冗长的 `if` 语句链。下面打印了名为 `v` 的 `Variant<int, double, string>` 的值：

```
1 if (v.is<int>()) {
2     std::cout << v.get<int>();
3 }
4 else if (v.is<double>()) {
5     std::cout << v.get<double>();
6 }
7 else {
8     std::cout << v.get<string>();
9 }
```

要一般化输出存储在变量中的值，需要递归实例化函数模板和辅助函数。例如：

`variant/printrec.cpp`

```
1 #include "variant.hpp"
2 #include <iostream>
3
```

```

4 template<typename V, typename Head, typename... Tail>
5 void printImpl(V const& v)
6 {
7     if (v.template is<Head>()) {
8         std::cout << v.template get<Head>();
9     }
10    else if constexpr (sizeof... (Tail) > 0) {
11        printImpl<V, Tail...>(v);
12    }
13 }
14
15 template<typename... Types>
16 void print(Variant<Types...> const& v)
17 {
18     printImpl<Variant<Types...>, Types...>(v);
19 }
20
21 int main() {
22     Variant<int, short, float, double> v(1.5);
23     print(v);
24 }
```

对于一个简单的操作来说，这代码体积已经相当大了。为了简化，通过使用 visit() 操作扩展 Variant 来解决这个问题。外部传入访问函数对象，该对象的函数操作符的调用将使用动态值。因为动态值可以是 Variant 的任何类型，所以这个函数操作符很可能是重载的，或者本身就是一个函数模板。泛型 Lambda 提供了一个模板函数操作符，可以表示 Variant v 的打印操作：

```

1 v.visit([](auto const& value) {
2     std::cout << value;
3 });
```

泛型 Lambda 大致等价于下面的函数对象，对于还不支持泛型 Lambda 的编译器也有用：

```

1 class VariantPrinter {
2     public:
3     template<typename T>
4     void operator() (T const& value) const
5     {
6         std::cout << value;
7     }
8 };
```

visit() 操作的核心类似于递归打印操作：遍历变量的类型，检查动态值是否具有给定的类型（使用 is<T>()），然后在找到合适的类型时进行打印：

variant/variantvisitimpl.hpp

```

1 template<typename R, typename V, typename Visitor,
2     typename Head, typename... Tail>
3 R variantVisitImpl(V&& variant, Visitor&& vis, Typelist<Head, Tail...>) {
```

```

4   if (variant.template is<Head>()) {
5     return static_cast<R>(
6       std::forward<Visitor>(vis) (
7         std::forward<V>(variant).template get<Head>()));
8   }
9   else_if constexpr (sizeof...(Tail) > 0) {
10    return variantVisitImpl<R>(std::forward<V>(variant),
11      std::forward<Visitor>(vis),
12      Typelist<Tail...>());
13  }
14  else {
15    throw EmptyVariant();
16  }
17 }
```

`variantVisitImpl()` 是一个带有许多模板参数的非成员函数模板。模板参数 R 描述了访问操作的结果类型，在稍后返回。V 是 Variant 类型，访问器是 Visitor 的类型。Head 和 Tail 用于分解 Variant 中的类型，从而影响递归。

第一个 if 执行 (运行时) 检查，以确定给定变量的活动值是否为 Head 类型：若是，则通过 `get()` 从变量中提取值，并传递给 visitor，终止递归。第二个 if 在需要考虑更多元素时执行递归。若没有匹配的类型，则该 Variant 不包含值，

这个案例在第 26.4.3 节有详细的讨论

这种情况下，实现可以抛出 `EmptyVariant` 异常。

除了 `VisitResult` 提供的结果类型计算 (在下一节中讨论)，`visit()` 的实现也很简单：

variant/variantvisit.hpp

```

1 template<typename... Types>
2   template<typename R, typename Visitor>
3 VisitResult<R, Visitor, Types&...>
4 Variant<Types...>::visit(Visitor&& vis) & {
5   using Result = VisitResult<R, Visitor, Types&...>;
6   return variantVisitImpl<Result>(*this, std::forward<Visitor>(vis),
7     Typelist<Types...>());
8 }
9
10 template<typename... Types>
11 template<typename R, typename Visitor>
12 VisitResult<R, Visitor, Types const&...>
13 Variant<Types...>::visit(Visitor&& vis) const & {
14   using Result = VisitResult<R, Visitor, Types const &...>;
15   return variantVisitImpl<Result>(*this, std::forward<Visitor>(vis),
16     Typelist<Types...>());
17 }
18
19 template<typename... Types>
```

```

20 template<typename R, typename Visitor>
21 VisitResult<R, Visitor, Types&&...>
22 Variant<Types...>::visit(Visitor&& vis) && {
23     using Result = VisitResult<R, Visitor, Types&&...>;
24     return variantVisitImpl<Result>(std::move(*this),
25             std::forward<Visitor>(vis),
26             TypeList<Types...>());
27 }

```

实现直接委托给 variantVisitImpl，传递 Variant 本身，转发访问器，并提供完整的类型列表。这三种实现之间唯一的区别是，是否将 Variant 本身作为 Variant&、Variant const& 或 Variant&& 进行传递。

26.5.1 结果类型

visit() 的结果类型仍然是个谜。给定的访问器可能具有不同的函数操作符重载，以产生不同的结果类型，结果类型依赖于其形参类型的模板函数操作符，或者某种组合。看看下面的泛型 Lambda:

```

1 [] (auto const& value) {
2     return value + 1;
3 }

```

这个 Lambda 的结果类型取决于输入类型: 给定 int 型，则将产生 int 型值；给定 double 型，将产生 double 型值。若这个泛型 Lambda 传递给了 Variant<int, double> 的 visit() 操作，结果类型应该是什么？

这里没有正确的答案，因此 visit() 操作允许提供结果类型。在另一个 Variant<int, double> 中捕获结果，可以将结果类型指定为 visit() 作为第一个模板参数:

```

1 v.visit<Variant<int, double>>([] (auto const& value) {
2     return value + 1;
3 });

```

当没有通用解决方案时，指定结果类型就很重要。但要求在所有情况下指定结果类型可能会让代码显得非常冗长，visit() 使用默认模板参数和简单元程序组合了这两个选项。回顾 visit() 的声明:

```

1 template<typename R = ComputedResultType, typename Visitor>
2 VisitResult<R, Visitor, Types&&...> visit(Visitor&& vis) &;

```

上面的例子中显式地指定了模板参数 R，有一个默认参数，因此不需要总是指定。该默认参数是一个不完整的哨兵类型 ComputedResultType:

```

1 class ComputedResultType;

```

要计算它的结果类型，visit 将所有的模板参数传递给 VisitResult，这是一个别名模板，提供了对新类型特征 VisitResultT 的访问:

variant/variantvisitresult.hpp

```

1 // an explicitly-provided visitor result type:

```

```

2 template<typename R, typename Visitor, typename... ElementTypes>
3 class VisitResultT
4 {
5     public:
6     using Type = R;
7 };
8
9 template<typename R, typename Visitor, typename... ElementTypes>
10 using VisitResult =
11 typename VisitResultT<R, Visitor, ElementTypes...>::Type;

```

VisitResultT 的主要定义了处理 R 参数显式指定的情况，因此 Type 定义为 R。当 R 接收到默认参数 ComputedResultType 时，启用偏特化：

```

1 template<typename Visitor, typename... ElementTypes>
2 class VisitResultT<ComputedResultType, Visitor, ElementTypes...>
3 {
4     ...
5 }

```

这种偏特化负责为常见情况计算适当的结果类型，这是下一节的主题。

26.5.2 常见的结果类型

当调用为每个变量的元素类型产生不同类型的访问器时，如何将这些类型组合成一个单独的结果类型用于 visit()？有一些简单的情况——若访问器为每个元素类型返回相同的类型，应该是 visit() 的结果类型。

C++ 已经有了合理结果类型的概念，在第 1.3.3 节中介绍过：三元表达式 $b ? x : y$ 的类型是 x 和 y 类型之间的公共类型。若 x 有 int 类型，y 有 double 类型，公共类型是 double，因为 int 可以提升为 double。也可以在类型特征中捕捉到共同类型的概念：

variant/commonType.hpp

```

1 using std::declval;
2
3 template<typename T, typename U>
4 class CommonTypeT
5 {
6     public:
7     using Type = decltype(true? declval<T>() : declval<U>());
8 };
9
10 template<typename T, typename U>
11 using CommonType = typename CommonTypeT<T, U>::Type;

```

公共类型的概念扩展到一组类型：公共类型是集合中的所有类型，都可以提升到的类型。对于访问器，我们希望计算访问器在使用变量中的每个类型调用时，将产生的结果的通用类型：

variant/variantvisitresultcommon.hpp

```
1 #include "accumulate.hpp"
2 #include "common_type.hpp"
3
4 // the result type produced when calling a visitor with a value of type T:
5 template<typename Visitor, typename T>
6 using VisitElementResult = decltype(declval<Visitor>() (declval<T>()));
7
8 // the common result type for a visitor called with each of the given element types:
9 template<typename Visitor, typename... ElementTypes>
10 class VisitResultT<ComputedResultType, Visitor, ElementTypes...>
11 {
12     using ResultTypes =
13         Typelist<VisitElementResult<Visitor, ElementTypes>...>;
14
15     public:
16     using Type =
17         Accumulate<PopFront<ResultTypes>, CommonTypeT, Front<ResultTypes>>;
```

VisitResult 的计算分两个阶段进行。首先，VisitElementResult 计算调用访问器的值为 T 类型时，产生的结果类型。这个元函数应用于每个给定的元素类型，以确定访问器可以产生的所有结果类型，并在类型列表 ResultTypes 中进行捕获。

接下来，计算使用 24.2.6 节中描述的 Accumulate 算法，将公共类型计算应用于结果类型的类型列表。其初始值 (Accumulate 的第三个参数) 是第一个结果类型，通过 CommonTypeT 与 ResultTypes 类型列表剩余部分的连续值组合在一起。最终结果是通用类型，访问器的所有结果类型都可以转换为通用类型，若结果类型不兼容，则会出现错误。

C++11 后，标准库提供了一个相应的类型特征，`std::common_type<>`，有效地结合了 CommonTypeT 和 Accumulate，使用这种方法生成任意数量传递类型的通用类型 (参见 D.5 节)。通过使用 `std::common_type<>`，VisitResultT 的实现可以更简单：

variant/variantvisitresultstd.hpp

```
1 template<typename Visitor, typename... ElementTypes>
2 class VisitResultT<ComputedResultType, Visitor, ElementTypes...>
3 {
4     public:
5     using Type =
6         std::common_type_t<VisitElementResult<Visitor, ElementTypes>...>;
7 }
```

下面的示例程序输出了通过传入泛型 Lambda 来生成的类型，该泛型 Lambda 将其得到的值加 1：

variant/visit.cpp

```
1 #include "variant.hpp"
```

```

2 #include <iostream>
3 #include <typeinfo>
4
5 int main()
6 {
7     Variant<int, short, double, float> v(1.5);
8     auto result = v.visit([](auto const& value) {
9         return value + 1;
10    });
11    std::cout << typeid(result).name() << '\n';
12 }

```

因为结果是所有结果类型都可以转换的类型，所以这个程序的输出将是 double 的 type_info 名。

26.6. 变量初始化赋值

Variant 可以通过多种方式进行初始化和赋值，包括默认构造、复制和移动构造以及复制和移动赋值。本节详细介绍了这些 Variant 操作。

默认构造

Variant 是否应该提供默认构造函数？若不这样做，因为总是需要一个初始值（即使初始值在编程上没有意义），所以 Variant 的使用可能会变得困难。若提供了默认构造函数，那么语义应该是什么？

一种可能的语义是默认初始化没有存储值，由辨别器 0 表示。空 Variant 通常没有用（不能访问或找到任何值来提取），并且将此作为默认初始化行为会将空 Variant 的异常状态（在第 26.4.3 节中描述）提升为普通状态。

或者，默认构造函数可以构造某种类型的值。对于我们的 Variant，遵循 C++17 的 std::variant<> 语义，并默认构造类型列表中的第一个类型的值：

variant/variantdefaultctor.hpp

```

1 template<typename... Types>
2 Variant<Types...>::Variant() {
3     *this = Front<TypeList<Types...>>();
4 }

```

这种方法简单可预测，并避免在大多数使用中引入空 Variant。这个行为可以在下面的程序中看到：

variant/variantdefaultctor.cpp

```

1 #include "variant.hpp"
2 #include <iostream>
3
4 int main()
5 {

```

```

6 Variant<int, double> v;
7 if (v.is<int>()) {
8     std::cout << "Default-constructed v stores the int "
9     << v.get<int>() << '\n';
10 }
11 Variant<double, int> v2;
12 if (v2.is<double>()) {
13     std::cout << "Default-constructed v2 stores the double "
14     << v2.get<double>() << '\n';
15 }
16 }
```

输出为

```

Default-constructed v stores the int 0
Default-constructed v2 stores the double 0
```

复制/移动构造

复制和移动构造更有趣。要复制源变量，需要确定当前存储的是哪种类型，将该值复制构造到缓冲区中，并设置辨别器。visit() 可以解码源 Variant 的动态值，而从 VariantChoice 继承的复制赋值操作符将复制构造一个值到缓冲区：

尽管在 Lambda 表达式中使用了赋值操作符 (=)，但赋值操作符在 VariantChoice 中的实际实现执行的是复制构造，因为该变量最初无存储值。

variant/variantcopyctor.hpp

```

1 template<typename... Types>
2 Variant<Types...>::Variant(Variant const& source) {
3     if (!source.empty()) {
4         source.visit([&] (auto const& value) {
5             *this = value;
6         });
7     }
8 }
```

移动构造函数与此类似，不同之处在于访问源变量时使用 std::move，并根据源值进行移动赋值：

variant/variantcopyctor.hpp

```

1 template<typename... Types>
2 Variant<Types...>::Variant(Variant&& source) {
3     if (!source.empty()) {
```

```

4     std::move(source).visit([&] (auto&& value) {
5         *this = std::move(value);
6     });
7 }
8 }
```

基于访问器实现的一个有趣的方面是，其也适用于复制和移动操作的模板参数。模板复制构造函数可以这样定义：

variant/variantcopyctorimpl.hpp

```

1 template<typename... Types>
2 template<typename... SourceTypes>
3 Variant<Types...>::Variant(Variant<SourceTypes...> const& source) {
4     if (!source.empty()) {
5         source.visit([&] (auto const& value) {
6             *this = value;
7         });
8     }
9 }
```

这段代码访问了 source，所以对 *this 的赋值将在每个 SourceTypes 中发生。此赋值的重载解析将为每个 SourceTypes 找到最合适的目标类型，并根据需要进行隐式转换。下面的例子说明了不同 Variant 类型的构造和赋值：

variant/variantpromote.hpp

```

1 #include "variant.hpp"
2 #include <iostream>
3 #include <string>
4
5 int main()
6 {
7     Variant<short, float, char const*> v1((short)123);
8
9     Variant<int, std::string, double> v2(v1);
10
11    std::cout << "v2 contains the integer " << v2.get<int>() << '\n';
12
13    v1 = 3.14f;
14    Variant<double, int, std::string> v3(std::move(v1));
15    std::cout << "v3 contains the double " << v3.get<double>() << '\n';
16
17    v1 = "hello";
18    Variant<double, int, std::string> v4(std::move(v1));
19    std::cout << "v4 contains the string " << v4.get<std::string>() << '\n';
20 }
```

从 v1 构造或赋值到 v2 或 v3 涉及整型提升 (short 到 int)、浮点提升 (float 到 double) 和用户定义的转换 (char const* 到 std::string)。

输出如下所示:

```
v2 contains the integer 123  
v3 contains the double 3.14  
v4 contains the string hello
```

赋值

Variant 赋值操作符类似于上面的复制和移动构造函数。这里，只演示复制赋值操作符:

variant/variantcopyassign.hpp

```
1 template<typename... Types>  
2 Variant<Types...>& Variant<Types...>::operator= (Variant const& source) {  
3     if (!source.empty()) {  
4         source.visit([&] (auto const& value) {  
5             *this = value;  
6         });  
7     }  
8     else {  
9         destroy();  
10    }  
11    return *this;  
12 }
```

这里唯一有趣的是在 **else** 分支中: 当 source 不包含值 (由辨别器 0 表示) 时, 我们销毁目标 Variant 的值, 隐式地将其辨别器设置为 0。

26.7. 后记

Andrei Alexandrescu 在一系列文章 [alexandrescu_edunions] 中详细介绍了可辨别联合。我们对 Variant 的处理使用了相同的技术, 比如使用对齐缓冲区, 就地存储和访问来提取值。一些差异是由于基础语言造成的:Andrei 使用的是 C++98, 因此不能使用可变参数模板或继承构造函数。Andrei 还花了相当多的时间来计算对齐, C++11 直接引入了对齐, 使得这项工作变得很简单。设计差异在于对辨别器的处理: 虽然我们选择使用整型辨别器来指示, 当前存储在变体中的是哪种类型, 但 Andrei 使用了一种“静态虚函数表”的方法, 使用函数指针来构造、复制、查询和销毁底层元素类型。有趣的是, 这种静态虚函数表方法作为开放可辨别联合的优化技术影响更大, 比如在第 22.2 节中开发的 FunctionPtr 模板, 它是 std::function 实现的一种常见优化, 以消除虚函数的使用。Boost 的 any 类型 ([BoostAny]) 是另一种开放可辨别联合类型。C++17 的标准库中, 引入了 std::any。

后来, Boost 库 ([Boost]) 引入了几种可辨别联合类型, 包括一种变体类型 ([BoostVariant]), 它影响了本章中开发的联合类型。Boost.Variant([BoostVariant]) 的设计文档, 包含了关于变量赋值异常安全

的问题(称为“永不为空的约定”)和各种不完全令人满意的解决方案讨论记录。当标准库在 C++17 引入 std::variant 时,放弃了永不空的约定:通过允许 std::variant 状态可以变成 valueless_by_exception,从而消除了为备份分配堆存储的需要,而 std::variant 状态会赋值给它抛出的新值,我们用空变量对这一行为进行了建模。

与我们的 Variant 模板不同, std::variant 允许多个相同的模板参数(例如, std::variant<int, int>)。在 Variant 中启用该功能需要在设计方面进行大量修改,包括添加一个方法来消除 VariantChoice 基类的歧义,以及在 26.2 节中描述的嵌套包扩展的替代方法。

本章描述的 visit() 操作变体在结构上与 Andrei Alexandrescu 在 [AlexandrescuAdHocVisitor] 中描述的临时访问器模式相同。Alexandrescu 的特别访问器旨在简化针对一组已知派生类(描述为类型列表)检查某个公共基类指针的过程。该实现使用 dynamic_cast 来针对类型列表中的每个派生类测试指针,当发现匹配时,使用派生类指针调用访问器。

第 27 章 表达式模板

本章中，我们将探索一种叫做表达式模板的模板编程技术，最初是为了支持数字数组类而发明的。

数字数组类支持对整个数组对象进行数字操作，可以将两个数组相加，结果包含的元素是参数数组中相应值的和。类似地，整个数组可以乘以一个标量，这意味着数组的每个元素都要扩展。自然，保留内置标量类型所熟悉的操作符表示法是可行的：

```
1 Array<double> x(1000), y(1000);
2 ...
3 x = 1.2*x + x*y;
```

对于专注于数据分析的开发者来说，在运行代码的平台上高效地计算这些表达式是至关重要的。使用本例中的运算符表示法来实现这一点，并不是一项简单的任务，但是表达式模板可以帮助我们。

表达式模板让人想起模板元编程，因为表达式模板有时依赖于深度嵌套的模板实例化，这与模板元程序中遇到的递归实例化相同。这两种技术最初都是为了支持高性能数组操作而开发的（请参阅第 23.1.3 节中使用模板展开循环的示例）。当然，技术是互补的。元编程对于固定大小的小型数组很方便，而表达式模板对于运行时大小为中型到大型数组的操作非常有效。

27.1. 临时变量和分割循环

为了激发表达式模板的灵感，先从一种简单的方法开始，来实现支持数字数组操作的模板。基本数组模板可能如下所示（SArray 代表简单数组）：

exprtmpl/sarray1.hpp

```
1 #include <cstddef>
2 #include <cassert>
3
4 template<typename T>
5 class SArray {
6 public:
7     // create array with initial size
8     explicit SArray (std::size_t s)
9     : storage(new T[s]), storage_size(s) {
10         init();
11     }
12
13     // copy constructor
14     SArray (SArray<T> const& orig)
15     : storage(new T[orig.size()]), storage_size(orig.size()) {
16         copy(orig);
17     }
18
19     // destructor: free memory
20     ~SArray() {
```

```

21     delete[] storage;
22 }
23
24 // assignment operator
25 SArray<T>& operator= (SArray<T> const& orig) {
26     if (&orig!=this) {
27         copy(orig);
28     }
29     return *this;
30 }
31
32 // return size
33 std::size_t size() const {
34     return storage_size;
35 }
36
37 // index operator for constants and variables
38 T const& operator[] (std::size_t idx) const {
39     return storage[idx];
40 }
41 T& operator[] (std::size_t idx) {
42     return storage[idx];
43 }
44
45 protected:
46 // init values with default constructor
47 void init() {
48     for (std::size_t idx = 0; idx<size(); ++idx) {
49         storage[idx] = T();
50     }
51 }
52
53 // copy values of another array
54 void copy (SArray<T> const& orig) {
55     assert(size()==orig.size());
56     for (std::size_t idx = 0; idx<size(); ++idx) {
57         storage[idx] = orig.storage[idx];
58     }
59 }
60
61 private:
62 T* storage; // storage of the elements
63 std::size_t storage_size; // number of elements
64 };

```

数字运算符的代码如下：

exprtmpl/sarrayopsI.hpp

```

1 // addition of two SArrays

```

```

2 template<typename T>
3 SArray<T> operator+ (SArray<T> const& a, SArray<T> const& b)
4 {
5     assert(a.size()==b.size());
6     SArray<T> result(a.size());
7     for (std::size_t k = 0; k<a.size(); ++k) {
8         result[k] = a[k]+b[k];
9     }
10    return result;
11 }
12
13 // multiplication of two SArrays
14 template<typename T>
15 SArray<T> operator* (SArray<T> const& a, SArray<T> const& b)
16 {
17     assert(a.size()==b.size());
18     SArray<T> result(a.size());
19     for (std::size_t k = 0; k<a.size(); ++k) {
20         result[k] = a[k]*b[k];
21     }
22     return result;
23 }
24
25 // multiplication of scalar and SArray
26 template<typename T>
27 SArray<T> operator* (T const& s, SArray<T> const& a)
28 {
29     SArray<T> result(a.size());
30     for (std::size_t k = 0; k<a.size(); ++k) {
31         result[k] = s*a[k];
32     }
33     return result;
34 }
35
36 // multiplication of SArray and scalar
37 // addition of scalar and SArray
38 // addition of SArray and scalar
39 ...

```

这些操作符和其他操作符的许多其他版本都可以这样写，这些足够我们的示例表达式使用：

exprtmpl/sarray1.cpp

```

1 #include "sarray1.hpp"
2 #include "sarrayops1.hpp"
3
4 int main()
{
6     SArray<double> x(1000), y(1000);
7     ...

```

```
8 x = 1.2*x + x*y;
9 }
```

由于两个原因，这个实现非常低效：

- 操作符的每个应用程序 (赋值除外) 至少创建一个临时数组 (即，假设编译器执行所有允许的临时复制消除操作，则在本例中至少创建三个大小为 1,000 的临时数组)。
- 操作符的每个应用程序都需要对参数数组和结果数组进行遍历 (假设只生成三个临时 SArray 对象，在示例中将读取大约 6000 个 double，并写入大约 4000 个 double)。

具体的是一个使用临时变量操作的循环序列：

```
1 what happens concretely is a sequence of loops that operates with temporaries:
2 tmp1 = 1.2*x; // loop of 1,000 operations
3     // plus creation and destruction of tmp1
4 tmp2 = x*y; // loop of 1,000 operations
5     // plus creation and destruction of tmp2
6 tmp3 = tmp1+tmp2; // loop of 1,000 operations
7     // plus creation and destruction of tmp3
8 x = tmp3; // 1,000 read operations and 1,000 write operations
```

除非使用特殊的快速分配器，否则创建不需要的临时变量通常占用小数组操作所需的时间。因为没有存储它们的储空间，所以对于真正的大型临时数组完全不可接受 (具有挑战性的数值模拟通常试图使用所有可用的内存来获得更真实的结果。若内存用来保存不需要的临时文件，模拟的质量就会受到影响)。

数字数组库的早期实现也面临这个问题，并鼓励用户使用计算赋值 (如 +=、 *= 等)。这些赋值的优点是参数和目的地都是由调用方提供的，因此不需要临时变量。可以这样添加 SArray 成员：

exprtmpl/sarrayops2.hpp

```
1 // additive assignment of SArray
2 template<typename T>
3 SArray<T>& SArray<T>::operator+= (SArray<T> const& b)
4 {
5     assert(size()==orig.size());
6     for (std::size_t k = 0; k<size(); ++k) {
7         (*this) [k] += b[k];
8     }
9     return *this;
10 }
11
12 // multiplicative assignment of SArray
13 template<typename T>
14 SArray<T>& SArray<T>::operator*= (SArray<T> const& b)
15 {
16     assert(size()==orig.size());
17     for (std::size_t k = 0; k<size(); ++k) {
18         (*this) [k] *= b[k];
19     }
}
```

```

20     return *this;
21 }
22
23 // multiplicative assignment of scalar
24 template<typename T>
25 SArray<T>& SArray<T>::operator*= (T const& s)
26 {
27     for (std::size_t k = 0; k < size(); ++k) {
28         (*this)[k] *= s;
29     }
30     return *this;
31 }
```

使用这样的运算符，计算示例可以重写为

exprtmp/sarray2.cpp

```

1 #include "sarray2.hpp"
2 #include "sarrayops1.hpp"
3 #include "sarrayops2.hpp"
4
5 int main()
6 {
7     SArray<double> x(1000), y(1000);
8     ...
9     // process  $x = 1.2*x + x*y$ 
10    SArray<double> tmp(x);
11    tmp *= y;
12    x *= 1.2;
13    x += tmp;
14 }
```

显然，计算赋值的技术仍有不足：

- 符号很笨拙。
- 不需要的临时 tmp。
- 循环拆分为多个操作，总共需要从内存中读取大约 6000 个 double 元素，并将 4000 个 double 元素写入内存。

我们真正想要的是一个“理想循环”，可以处理每个索引的整个表达式：

```

1 int main()
2 {
3     SArray<double> x(1000), y(1000);
4     ...
5     for (int idx = 0; idx < x.size(); ++idx) {
6         x[idx] = 1.2*x[idx] + x[idx]*y[idx];
7     }
8 }
```

现在不需要临时数组，每次迭代只需要两次内存读取 ($x[idx]$ 和 $y[idx]$) 和一次内存写入 ($x[k]$)。因此，手动循环只需要大约 2000 次内存读取和 1000 次内存写入。

考虑到在现代高性能计算机架构中，内存带宽是这类数组操作速度的限制因素，这里展示的简单运算符重载方法的性能比手动编码循环慢一到两个数量级也不足为奇。但我们希望获得手动编码循环的性能，而不需要手工编写这些循环，也不需要使用笨拙的表示法，但这样做既麻烦又容易出错。

27.2. 模板参数中的编码表达式

解决这个问题的关键是，在整个表达式之前（示例中，调用赋值操作符之前）不要尝试计算表达式。在求值之前，必须记录哪些操作应用于哪些对象。操作在编译时确定，因此可以对模板参数进行编码。

我们的示例表达式，

```
1 1.2*x + x*y;
```

$1.2*x$ 的结果不是一个新数组，而是一个表示 x 的每个值乘以 1.2 的对象。类似地， $x*y$ 必须得到 x 中的每个元素乘以 y 中每个对应的元素。当需要得到结果数组的值时，进行存储以备后续的计算。

这里设计一个具体的实现，实现对表达式进行求值

```
1 1.2*x + x*y;
```

转换成以下类型的对象：

```
1 A_Add<A_Mult<A_Scalar<double>, Array<double>>,
2   A_Mult<Array<double>, Array<double>>>
```

我们将新的基本数组类模板与类模板 A_Scalar 、 A_Add 和 A_Mult 组合在一起。表达式对应的语法树中，可以找到一个前缀表示（参见图 27.1）。这个嵌套的模板标识表示所涉及的操作，以及操作应该应用到的对象的类型。 A_Scalar 稍后会出现，但其只是数组表达式中标量的占位符。

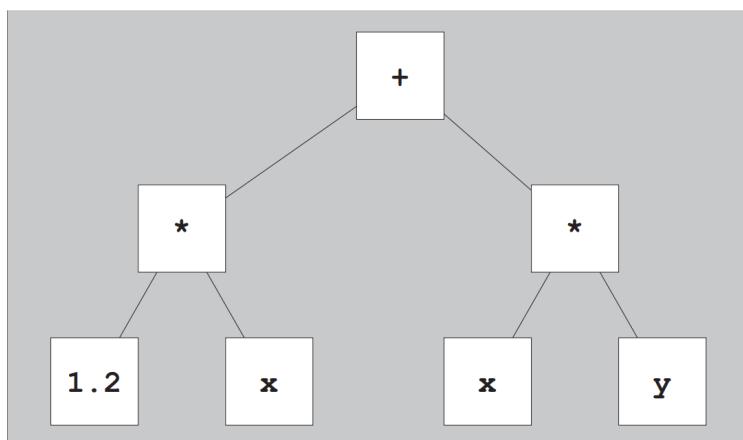


图 27.1. 表达式 $1.2*x+x*y$ 的树型表示

27.2.1 表达式模板的操作数

为了完成表达式的表示，必须在每个 A_Add 和 A_Mult 对象中存储对参数的引用，并记录 A_Scalar 对象的值（或对其的引用）。以下是相应操作数的定义：

exprtmpl/exprops1.hpp

```
1 #include <cstddef>
2 #include <cassert>
3
4 // include helper class traits template to select whether to refer to an
5 // expression template node either by value or by reference
6 #include "exprops1a.hpp"
7
8 // class for objects that represent the addition of two operands
9 template<typename T, typename OP1, typename OP2>
10 class A_Add {
11     private:
12         typename A_Traits<OP1>::ExprRef op1; // first operand
13         typename A_Traits<OP2>::ExprRef op2; // second operand
14
15     public:
16         // constructor initializes references to operands
17         A_Add (OP1 const& a, OP2 const& b)
18             : op1(a), op2(b) {
19         }
20
21         // compute sum when value requested
22         T operator[] (std::size_t idx) const {
23             return op1[idx] + op2[idx];
24         }
25
26         // size is maximum size
27         std::size_t size() const {
28             assert (op1.size()==0 || op2.size()==0
29             || op1.size()==op2.size());
30             return op1.size()!=0 ? op1.size() : op2.size();
31         }
32     };
33
34 // class for objects that represent the multiplication of two operands
35 template<typename T, typename OP1, typename OP2>
36 class A_Mult {
37     private:
38         typename A_Traits<OP1>::ExprRef op1; // first operand
39         typename A_Traits<OP2>::ExprRef op2; // second operand
40
41     public:
42         // constructor initializes references to operands
43         A_Mult (OP1 const& a, OP2 const& b)
```

```

44 : op1(a), op2(b) {
45 }
46
47 // compute product when value requested
48 T operator[] (std::size_t idx) const {
49     return op1[idx] * op2[idx];
50 }
51
52 // size is maximum size
53 std::size_t size() const {
54     assert (op1.size()==0 || op2.size()==0
55     || op1.size()==op2.size());
56     return op1.size()!=0 ? op1.size() : op2.size();
57 }
58 };

```

这里添加了下标和大小查询操作，这些操作可以计算数组元素的大小和值，这些操作由在根部给定对象上的节点子树表示。

对于只涉及数组的操作，结果的大小是两个操作数的大小。对于同时涉及数组和标量的操作，结果的大小是数组操作数的大小。为了区分数组操作数和标量操作数，定义标量的大小为 0。因此，`A_Scalar` 模板的定义如下：

exprtmpl/exprscalar.cpp

```

1 // class for objects that represent scalars:
2 template<typename T>
3 class A_Scalar {
4     private:
5     T const& s; // value of the scalar
6
7     public:
8     // constructor initializes value
9     constexpr A_Scalar (T const& v)
10    : s(v) {
11    }
12
13    // for index operations, the scalar is the value of each element
14    constexpr T const& operator[] (std::size_t) const {
15        return s;
16    }
17
18    // scalars have zero as size
19    constexpr std::size_t size() const {
20        return 0;
21    };
22 };

```

(我们已经声明了构造函数和成员函数 `constexpr`，因此这个类可以在编译时使用。然而，对于我们的目的来说，这不是必需的。)

注意，标量还提供索引操作符。表达式内部，其表示一个数组，每个索引都具有相同的标量值。操作符类可以使用辅助类 A_Traits 来定义操作数的成员：

```
1 typename A_Traits<OP1>::ExprRef op1; // first operand
2 typename A_Traits<OP2>::ExprRef op2; // second op
```

通常，可以将它们声明为引用，因为大多数临时节点都绑定在顶层表达式中，因此直到整个表达式的计算结束为止都存在。唯一的例外是 A_Scalar 节点，其绑定在操作符函数中，可能直到整个表达式求值结束才会存在。为了避免成员引用不再存在的标量，必须按值复制 A_Scalar 操作数。

换句话说，我们需要这样的成员

- 一般的常量引用：

```
1 OP1 const& op1; // refer to first operand by reference
2 OP2 const& op2; // refer to second operand by reference
```

- 作为标量是普通值：

```
1 OP1 op1; // refer to first operand by value
2 OP2 op2; // refer to second operand by value
```

这是特性类的完美应用。特性类定义了一个类型，通常是常量引用，但对于标量是普通值：

exprtmpl/exprops1a.hpp

```
1 // helper traits class to select how to refer to an expression template node
2 // - in general by reference
3 // - for scalars by value
4
5 template<typename T> class A_Scalar;
6
7 // primary template
8 template<typename T>
9 class A_Traits {
10   public:
11     using ExprRef = T const&; // type to refer to is constant reference
12 };
13
14 // partial specialization for scalars
15 template<typename T>
16 class A_Traits<A_Scalar<T>> {
17   public:
18     using ExprRef = A_Scalar<T>; // type to refer to is ordinary value
19 };
```

因为 A_Scalar 对象引用顶层表达式中的标量，所以这些标量可以使用引用类型，从而 A_Scalar<T>::s 是引用成员变量。

27.2.2 数组类型

有了使用轻量级表达式模板表达式的能力，现在必须创建一个 `Array` 类型，控制实际的存储，并且了解表达式模板。对于工程目的来说，保持具有存储功能的真实数组的接口，与产生数组的表达式的表示的接口最好尽可能相似。可以声明 `Array` 模板：

```
1 template<typename T, typename Rep = SArray<T>>
2 class Array;
```

若 `Array` 是一个存储阵列, `Rep` 类型可以是 `SArray`,

这里重用以前开发的 SArray 就很方便，但在工业库中，因为其不会使用 SArray 的所有特性，所以特殊用途的实现可能更合适。

或者可以是嵌套的模板标识，如 `A_Add` 或 `A_Mult`，其构建一个表达式。无论哪种方式，这里都在处理 `Array` 实例化，这简化了后面的处理操作。即使是 `Array` 模板的定义，尽管有些成员不能使用 `A_Mult` 代替 `Rep` 这样的类型实例化，但也不需要特化来区分这两种情况。

这是定义。该功能主要局限于 SArray 模板所提供的功能，在理解了代码意图后，添加该功能并不难：

exprtmpl/exprarray.hpp

```
1 #include <cstddef>
2 #include <cassert>
3 #include "sarray1.hpp"
4
5 template<typename T, typename Rep = SArray<T>>
6 class Array {
7     private:
8         Rep expr_rep; // (access to) the data of the array
9
10    public:
11        // create array with initial size
12        explicit Array (std::size_t s)
13            : expr_rep(s) {
14        }
15
16        // create array from possible representation
17        Array (Rep const& rb)
18            : expr_rep(rb) {
19        }
20
21        // assignment operator for same type
22        Array& operator= (Array const& b) {
23            assert(size()==b.size());
24            for (std::size_t idx = 0; idx<b.size(); ++idx) {
25                expr_rep[idx] = b[idx];
26            }
27        }
28    }
```

```

27     return *this;
28 }
29
30 // assignment operator for arrays of different type
31 template<typename T2, typename Rep2>
32 Array& operator=(Array<T2, Rep2> const& b) {
33     assert(size()==b.size());
34     for (std::size_t idx = 0; idx<b.size(); ++idx) {
35         expr_rep[idx] = b[idx];
36     }
37     return *this;
38 }
39
40 // size is size of represented data
41 std::size_t size() const {
42     return expr_rep.size();
43 }
44
45 // index operator for constants and variables
46 decltype(auto) operator[](std::size_t idx) const {
47     assert(idx<size());
48     return expr_rep[idx];
49 }
50 T& operator[](std::size_t idx) {
51     assert(idx<size());
52     return expr_rep[idx];
53 }
54
55 // return what the array currently represents
56 Rep const& rep() const {
57     return expr_rep;
58 }
59 Rep& rep() {
60     return expr_rep;
61 }
62 };

```

许多操作可以转发至底层的 Rep 对象。在复制另一个数组时，必须考虑到另一个数组实际上是在表达式模板上构建的可能。因此，可以根据底层表示参数化的复制操作。

下标操作符需要更多的讨论，该操作符的 const 版本使用的是推导的返回类型，而不是传统的 T const&。若 Rep 代表是 A_Mult 或 A_Add，其下标操作符需要返回一个临时值(即 prvalue)，不能通过引用返回 (decltype(auto)) 将为 prvalue 情况推导出非引用类型)。另一方面，若 Rep 是 SArray<T>，则底层下标操作符产生一个 const 左值，并且推导出的返回类型将是匹配的 const 引用。

27.2.3 运算符

除了运算符本身之外，我们已经具备了为数值数组模板创建高效数值运算符的大部分机制。这些操作符只组装表达式模板对象——并不实际计算结果数组。

对于每个普通的二元运算符，必须实现三个版本：数组-数组、数组-标量和标量-数组。为了能够计算初始值，我们需要以下操作符：

exprtmpl/exprops2.hpp

```
1 // addition of two Arrays:  
2 template<typename T, typename R1, typename R2>  
3 Array<T,A_Add<T,R1,R2>>  
4 operator+ (Array<T,R1> const& a, Array<T,R2> const& b) {  
5     return Array<T,A_Add<T,R1,R2>>  
6         (A_Add<T,R1,R2>(a.rep(),b.rep()));  
7 }  
8  
9 // multiplication of two Arrays:  
10 template<typename T, typename R1, typename R2>  
11 Array<T, A_Mult<T,R1,R2>>  
12 operator* (Array<T,R1> const& a, Array<T,R2> const& b) {  
13     return Array<T,A_Mult<T,R1,R2>>  
14         (A_Mult<T,R1,R2>(a.rep(), b.rep()));  
15 }  
16  
17 // multiplication of scalar and Array:  
18 template<typename T, typename R2>  
19 Array<T, A_Mult<T,A_Scalar<T>,R2>>  
20 operator* (T const& s, Array<T,R2> const& b) {  
21     return Array<T,A_Mult<T,A_Scalar<T>,R2>>  
22         (A_Mult<T,A_Scalar<T>,R2>(A_Scalar<T>(s), b.rep()));  
23 }  
24  
25 // multiplication of Array and scalar, addition of scalar and Array  
26 // addition of Array and scalar:  
27 ...
```

这些操作符的声明有些繁琐（从这些示例可以看出），但这些函数实际上没有做太多工作。两个数组的 plus 操作符首先创建一个 `A_Add` 对象，该对象表示操作符和操作数

```
1 A_Add<T,R1,R2>(a.rep(),b.rep())
```

并将该对象包装在 `Array` 对象中，以便可以将结果用作其他表示数组数据的对象：

```
1 return Array<T,A_Add<T,R1,R2>> (...);
```

对于标量乘法，可以使用 `A_Scalar` 模板来创建 `A_Mult` 对象

```
1 A_Mult<T,A_Scalar<T>,R2>(A_Scalar<T>(s), b.rep())
```

然后再次包装：

```
1 return Array<T,A_Mult<T,A_Scalar<T>,R2>> (...);
```

其他非成员二元操作符非常类似，可以使用宏来实现大多数操作符。另一个（较小的）宏可用于非成员一元操作符。

27.2.4 审查

第一次发现表达式模板思想时，各种声明和定义之间的交互可能会令人生畏。因此，对示例代码所发生的事情进行自上而下的审查，有助于理解。这里将要分析的代码如下所示(可以在 meta/exprmain.cpp 中找到):

```
1 int main()
2 {
3     Array<double> x(1000), y(1000);
4     ...
5     x = 1.2*x + x*y;
6 }
```

因为在 x 和 y 的定义中省略了 Rep 参数，所以其设置为默认值，即 SArray<double>。因此，x 和 y 是具有“真实”存储的数组，而不仅仅是记录操作。

解析表达式时

```
1 1.2*x + x*y
```

编译器首先应用最左边的 * 操作，这是一个标量数组操作符。因此，重载解析选择了操作符 * 的标量数组形式：

```
1 template<typename T, typename R2>
2 Array<T, A_Mult<T,A_Scalar<T>,R2>>
3 operator*(T const& s, Array<T,R2> const& b) {
4     return Array<T,A_Mult<T,A_Scalar<T>,R2>>
5         (A_Mult<T,A_Scalar<T>,R2>(A_Scalar<T>(s), b.rep()));
6 }
```

操作数类型为 double 和 Array<double, SArray<double>>。因此，结果的类型为

```
1 Array<double, A_Mult<double, A_Scalar<double>, SArray<double>>>
```

结果值构造为 A_Scalar<double> 对象的引用，该对象由 double 值 1.2 和对象 x 的 SArray<double> 表示构成。

接下来，计算第二个乘法：它是一个数组-数组操作 x*y。这次使用了另一个操作符 *：

```
1 template<typename T, typename R1, typename R2>
2 Array<T, A_Mult<T,R1,R2>>
3 operator*(Array<T,R1> const& a, Array<T,R2> const& b) {
4     return Array<T,A_Mult<T,R1,R2>>
5         (A_Mult<T,R1,R2>(a.rep(), b.rep()));
6 }
```

操作数类型都是 Array<double, SArray<double>>，因此结果类型为

```
1 Array<double, A_Mult<double, SArray<double>, SArray<double>>>
```

这次包装的 A_Mult 对象引用了两个 SArray<double> 表示：一个是 x，一个是 y。

最后，计算加法操作。这又是一个数组-数组操作，操作数类型是刚才推导的结果类型，使用数组加法操作符：

```

1 template<typename T, typename R1, typename R2>
2 Array<T,A_Add<T,R1,R2>>
3 operator+ (Array<T,R1> const& a, Array<T,R2> const& b) {
4     return Array<T,A_Add<T,R1,R2>>
5         (A_Add<T,R1,R2>(a.rep(),b.rep()));
6 }

```

T 可替换为 double, R1 可替换为

```

1 A_Mult<double, A_Scalar<double>, SArray<double>>

```

R2 可替换为

```

1 A_Mult<double, SArray<double>, SArray<double>>

```

赋值标记右侧的表达式类型为

```

1 Array<double,
2   A_Add<double,
3   A_Mult<double, A_Scalar<double>, SArray<double>>,
4   A_Mult<double, SArray<double>, SArray<double>>>

```

该类型匹配数组模板的赋值操作符模板:

```

1 template<typename T, typename Rep = SArray<T>>
2 class Array {
3     public:
4     ...
5     // assignment operator for arrays of different type
6     template<typename T2, typename Rep2>
7     Array& operator= (Array<T2, Rep2> const& b) {
8         assert(size() == b.size());
9         for (std::size_t idx = 0; idx < b.size(); ++idx) {
10             expr_rep[idx] = b[idx];
11         }
12         return *this;
13     }
14     ...
15 };

```

赋值操作符通过将下标操作符应用于右侧的表示(其类型为), 来计算目标 x 的每个元素

```

1 A_Add<double,
2   A_Mult<double, A_Scalar<double>, SArray<double>>,
3   A_Mult<double, SArray<double>, SArray<double>>>

```

仔细观察这个下标操作符可以发现, 对于给定的下标 idx, 可进行计算

```

1 (1.2*x[idx]) + (x[idx]*y[idx])

```

这正是我们想要的。

27.2.5 表达式模板赋值

我们示例的 A_Mult 和 A_Add 表达式模板上，不可能构建一个 Rep 参数数组的实例化写操作。(事实上， $a+b=c$ 毫无意义)但完全可以编写其他表达式模板，对其结果赋值，对整数值数组进行索引将对应于子集选择。换句话说，表达式

```
1 x[y] = 2*x[y];
```

应该等价于

```
1 for (std::size_t idx = 0; idx<y.size(); ++idx) {  
2     x[y[idx]] = 2*x[y[idx]];  
3 }
```

启用此功能意味着构建在表达式模板上，数组的行为类似于左值(可写)。这类操作的表达式模板组件与 A_Mult 没有本质上的区别，只是提供了下标操作符的 const 版本和非 const 版本，可以返回左值(引用)：

exprtmpl/exprops3.hpp

```
1 template<typename T, typename A1, typename A2>  
2 class A_Subscript {  
3     public:  
4         // constructor initializes references to operands  
5         A_Subscript (A1 const& a, A2 const& b)  
6         : a1(a), a2(b) {}  
7  
8         // process subscription when value requested  
9         decltype(auto) operator[] (std::size_t idx) const {  
10             return a1[a2[idx]];  
11         }  
12         T& operator[] (std::size_t idx) {  
13             return a1[a2[idx]];  
14         }  
15  
16         // size is size of inner array  
17         std::size_t size() const {  
18             return a2.size();  
19         }  
20         private:  
21         A1 const& a1; // reference to first operand  
22         A2 const& a2; // reference to second operand  
23     };
```

同样， decltype(auto) 在处理数组下标时很有用，无论底层表示产生的是值，还是左值。

前面建议的具有子集语义的扩展下标操作符，需要向 Array 模板添加额外的下标操作符。其中一个操作符可以这样定义(可能还需要对应的 const 版本)：

exprtmpl/exprops4.hpp

```

1 template<typename T, typename R>
2 template<typename T2, typename R2>
3 Array<T, A_Subscript<T, R, R2>>
4 Array<T, R>::operator[] (Array<T2, R2> const& b) {
5     return Array<T, A_Subscript<T, R, R2>>
6         (A_Subscript<T, R, R2>(*this, b));
7 }

```

27.3. 表达式模板的性能与约束

为了证明表达式模板思想的复杂性，我们已经在数组操作上提高了性能。在跟踪表达式模板时，会发现许多小型内联函数相互调用，并且在调用堆栈上分配了许多小型表达式模板对象。优化器必须执行完整的内联和消除小对象，以产生与手动编码循环一样有效的代码。本书的第一版中，我们说过很少有编译器能够实现这样的优化。但从那时起，情况有了很大的改善，部分原因是因为该技术很受欢迎。

表达式模板技术不能解决涉及数组数值操作的所有问题。不适用于这种形式的矩阵-向量乘法
 $x = A * x;$

其中 x 是一个大小为 n 的列向量， a 是一个 $n \times n$ 的矩阵。问题是必须使用一个临时元素，因为结果中的每个元素都依赖于原始 x 中的每个元素。但表达式模板循环会更新 x 的第一个元素，然后使用新计算的元素来计算第二个元素，这是错误的。另一个稍有不同的表达

$x = A * y;$

另一方面，若 x 和 y 不是彼此的别名，则不需要临时变量，从而解决方案必须在运行时知道操作数的关系。这里建议创建一个表示表达式运行时的树型结构，而不是用表达式模板的类型对树型结构进行编码。这种方法是由 Robert Davies 的 NewMat 库首创的（见 [NewMat]）。早在表达式模板开发出来之前，这种方式就已经很出名了。

27.4. 后记

表达式模板由 Todd Veldhuizen 和 David Vandevoorde（Todd 创造了这个词）独立开发，当时成员模板还不是 C++ 编程语言的一部分（而且在当时看来，似乎永远不会添加到 C++ 中）。这在实现赋值操作符时，出现了一些问题：无法为表达式模板对其进行参数化。解决这一问题的技术是在表达式模板中引入一个到复制类的转换操作符，该复制类用表达式模板参数化，但继承了一个只在元素类型中参数化的基类。然后，这个基类提供了赋值操作符，从而可以引用的（虚）copy_to 接口。

下面是这个机制的一个概述（以及本章中使用的模板名称）：

```

1 template<typename T>
2 class CopierInterface {
3     public:
4         virtual void copy_to(Array<T, SArray<T>>&) const;
5     };
6
7 template<typename T, typename X>
8 class Copier : public CopierInterface<T> {

```

```

9  public:
10 Copier(X const& x) : expr(x) {
11 }
12 virtual void copy_to(Array<T, SArray<T>>&) const {
13     // implementation of assignment loop
14     ...
15 }
16 private:
17 X const& expr;
18 };
19
20 template<typename T, typename Rep = SArray<T>>
21 class Array {
22 public:
23     // delegated assignment operator
24     Array<T, Rep>& operator=(CopierInterface<T> const& b) {
25         b.copy_to(rep);
26     };
27     ...
28 };
29
30 template<typename T, typename A1, typename A2>
31 class A_mult {
32 public:
33     operator Copier<T, A_Mult<T, A1, A2>>();
34     ...
35 };

```

这给表达式模板增加了另一个层次的复杂性和运行时成本，但产生的性能优势在当时还是令人印象深刻。

C++ 标准库包含了一个类模板 valarray，可以证明本章开发的 Array 模板所使用的技术正确。valarray 的前身设计出来是为了让面向科学计算市场的编译器能够识别数组类型，并使用高度优化的内部代码进行操作，编译器在某种意义上已经“理解”了这些类型。然而，这种情况从未发生（部分原因是所涉及的市场相对较小，部分原因是随着 valarray 成为模板，问题变得越来越复杂）。在表达式模板技术发现后的一段时间，我们中的一个 (Vandevoorde) 向 C++ 委员会提交了一份提案，将 valarray 转变为开发的 Array 模板（受现有 valarray 功能的启发，有很多花哨的功能），该提议首次记录了 Rep 参数的概念。在此之前，实际存储的数组和表达式模板伪数组是不同的模板。当外部代码引入接受数组的函数 foo() 时——例如，

```
1 double foo(Array<double> const&);
```

调用 `foo(1.2*x)` 强制将表达式模板转换为具有实际存储空间的数组，即使应用于该参数的操作不需要临时变量。若表达式模板内嵌在 Rep 参数中，则可以声明

```
1 template<typename Rep>
2 double foo(Array<double, Rep> const&);
```

除非确实需要，否则不会发生转换。

`valarray` 提案出现在 C++ 标准化过程的后期，实际上重写了标准中所有关于 `valarray` 的内容。结果被拒绝了，取而代之的是对现有内容进行了一些调整，允许基于表达式模板的实现，但使用这种方式仍然比这里讨论的要麻烦得多。撰写本文时，还不知道存在这样的实现，标准 `valarray` 在执行其设计的操作时效率非常低。

最后，本章介绍的许多开创性技术，以及后来称为 STL 的技术，

标准模板库 (STL) 彻底改变了 C++ 的世界，后来成为 C++ 标准库的一部分 (参见 [JosuttisStdLib])。

最初都是在同一个编译器上实现 (Borland 的 C++ 编译器)

Jaakko 在开发核心语言特性方面发挥了重要作用。

这可能是第一个使模板编程在 C++ 编程社区中广泛使用的编译器。

表达式模板最初主要应用于对类数组类型的操作。几年后，发现了新的应用场景。其中最具开创性的是 Jaakko Järvi 和 Gary Powell 的 Boost.Lambda 库 (参见 [LambdaLib])，在 Lambda 表达式成为核心语言特性之前提供了一个可用的 Lambda 表达式工具，以及 Eric Niebler 的 Boost.Proto 库，是一个元程序表达式模板库，目标是在 C++ 中创建嵌入式领域特定语言。其他 Boost 库，比如 Boost.Fusion 和 Boost.Hana 中也使用了高级的表达式模板。

第 28 章 调试模板

调试模板时，会遇到两类挑战。对于模板编写者来说，有一类挑战无疑是一个问题：如何确保所编写的模板对满足条件的模板参数都能起作用？另一类问题几乎完全相反：当模板的行为与文档中描述的不一致时，模板的用户如何找出不满足哪些模板参数要求？

深入讨论这些问题之前，考虑一下可能施加在模板参数上的各种约束。本章中，我们主要处理导致编译错误的约束，称其为语法约束。语法约束包括特定类型构造函数的存在，特定函数调用的无歧义性等。另一种约束称之为语义约束，要验证这些约束条件非常困难。通常，这样做甚至可能不实际，可能要求在模板类型参数上定义一个小于操作符（这是一种语法约束），但通常也会要求操作符实际上在其域上定义某种排序（这是一种语义约束）。

术语概念通常用来表示模板库中需要的一组约束，C++ 标准库依赖于随机访问迭代器和默认可构造函数等概念。有了这个术语，调试模板代码包含了大量确定在模板实现，及其使用中如何违反概念的工作。本章深入探讨了设计和调试技术，这些技术可以让模板的作者和用户更容易地使用模板。

28.1. 浅式实例化

当模板错误发生时，问题通常在实例化后发现，从而会有冗长的错误消息，就像在第 9.4 节中那样。

无疑在写代码时会遇到一些错误消息，会使最初的示例看起来很乏味！

为了说明这一点，请考虑以下的手写代码：

```
1 template<typename T>
2 void clear (T& p)
3 {
4     *p = 0; // assumes T is a pointer-like type
5 }
6
7 template<typename T>
8 void core (T& p)
9 {
10    clear(p);
11 }
12
13 template<typename T>
14 void middle (typename T::Index p)
15 {
16    core(p);
17 }
18
19 template<typename T>
20 void shell (T const& env)
21 {
```

```
22 typename T::Index i;
23 middle<T>(i);
24 }
```

这个例子阐明了软件开发的分层: 像 shell() 这样的高级函数模板依赖于像 middle() 这样的组件, 而这些组件本身会使用像 core() 这样的功能。当实例化 shell() 时, 下面的层也需要实例化。这个例子中, 有一个问题: core() 实例化为 int 类型 (在 middle() 中使用 Client::Index), 并试图错误的解引用该类型。

该错误仅在实例化时可检测到。例如:

```
1 class Client
2 {
3     public:
4     using Index = int;
5 };
6
7 int main()
8 {
9     Client mainClient;
10    shell(mainClient);
11 }
```

好的通用诊断包括导致所有级别的跟踪, 但是获得这么多信息, 也会让我们感觉手足无措。

在 [StroustrupDnE] 中可以找到围绕这个问题核心思想的讨论, Bjarne Stroustrup 确定了两类方法来更早地确定模板参数是否满足一组约束: 通过语言扩展或更早的参数使用。在第 17.8 节和附录 E 中介绍了前一种选择, 后一种选择包括在浅层实例化中强制错误。这通过插入未使用的代码来实现, 若代码使用的模板参数不满足更深层模板的要求, 就会触发错误。

前面的例子中, 可以在 shell() 中添加代码, 尝试对 T::Index 类型的值解引用。例如:

```
1 template<typename T>
2 void ignore(T const&)
3 { }
4
5 template<typename T>
6 void shell (T const& env)
7 {
8     class ShallowChecks
9     {
10         void deref(typename T::Index ptr) {
11             ignore(*ptr);
12         }
13     };
14     typename T::Index i;
15     middle(i);
16 }
```

若 T 是不能解引用 T::Index 的类型, 则会在局部类 ShallowChecks 上出现编译错误。因为没有使用局部类, 所以添加的代码不会影响 shell() 函数的运行时间, 但许多编译器会警告说没有使用

ShallowChecks(成员也是如此)。可以使用 ignore() 模板等技巧来抑制此类警告，但也会增加代码的复杂性。

概念检查

显然，示例中的代码开发可能会变得与实现模板的实际功能代码一样复杂。为了控制这种复杂性，可以尝试在某种类型的库中收集各种代码片段。这样的库可以包含宏，当模板参数替换违反该特定参数的概念时，这些宏可以扩展为触发适当错误的代码。这类库中最流行的是 Concept Check 库，是 Boost 发行版的一部分(参见 [BCCL])。但这种技术的可移植性不是特别好(不同编译器诊断错误的方式不同)，有时还会掩盖在更高级别上无法捕获错误的问题。

当 C++ 中有了概念(参见附录 E)，就有了其他的方法来支持需求和预期行为的定义。

28.2. 静态断言

assert() 宏通常在 C++ 代码中用于检查程序执行过程中，是否存在某些特定条件。若断言失败，程序将停止运行，以便开发者修复问题。

C++11 中引入的 static_assert 关键字具有相同的目的，但会在编译时进行求值：若条件(必须是常量表达式)求值为 false，编译器将发出错误消息。错误消息将包括一个字符串(它是 static_assert 本身的一部分)，告诉开发者哪里出错了。下面的静态断言确保我们在一个带有 64 位指针的平台上编译：

```
1 static_assert(sizeof(void*) * CHAR_BIT == 64, "Not a 64-bit platform");
```

当模板参数不满足模板的约束时，静态断言可用于提供有用错误消息。使用第 19.4 节描述的技术，可以创建一个类型特征来确定类型是否可解引用：

debugging/hasderef.hpp

```
1 #include <utility> // for declval()
2 #include <type_traits> // for true_type and false_type
3
4 template<typename T>
5 class HasDereference {
6     private:
7         template<typename U> struct Identity;
8         template<typename U> static std::true_type
9             test(Identity<decltype(*std::declval<U>())>*);
10        template<typename U> static std::false_type
11            test(...);
12    public:
13        static constexpr bool value = decltype(test<T>(nullptr))::value;
14 }
```

可以在 shell() 中引入一个静态断言，若上一节中的 shell() 模板实例化时使用了不可解引用的类型，该断言会提供更好的诊断信息：

```
1 template<typename T>
```

```

2 void shell (T const& env)
3 {
4     static_assert(HasDereference<T>::value, "T is not dereferenceable");
5     typename T::Index i;
6     middle(i);
7 }

```

通过这些更改，编译器会产生简单扼要的诊断信息，从而表明类型 T 不可解引用。

静态断言可以使错误消息更短、更简单，从而极大地改善使用模板库时的用户体验。

也可以将其应用于类模板，并使用附录 D 中的类型特征：

```

1 template<typename T>
2 class C {
3     static_assert(HasDereference<T>::value, "T is not dereferenceable");
4     static_assert(std::is_default_constructible<T>::value,
5                  "T is not default constructible");
6     ...
7 };

```

28.3. 原型

编写模板时，很难确保模板定义能够针对满足该模板约束的模板参数进行编译。考虑一个简单的 find() 算法，在数组中查找值，以及其文档约束：

```

1 // T must be EqualityComparable, meaning:
2 // two objects of type T can be compared with == and the result converted to bool
3 template<typename T>
4 int find(T const* array, int n, T const& value);

```

可以想象这个函数模板的简单实现：

```

1 template<typename T>
2 int find(T const* array, int n, T const& value) {
3     int i = 0;
4     while(i != n && array[i] != value)
5         ++i;
6     return i;
7 }

```

这个模板定义有两个问题，当某些在技术上满足模板要求，但行为与模板作者预期略有不同的模板参数时，这两个问题将表现为编译错误。我们将使用原型的概念，根据 find() 模板指定的需求来测试实现对模板参数的使用。

原型是用户定义的类，可以用作模板参数来测试模板定义是否遵守其对相应模板参数施加的约束。原型是定制的，以满足模板的需求，而不提供任何无关操作。若将原型作为模板参数的模板定义实例化成功，那就知道模板定义不会尝试使用模板没有明确要求的其他操作。

下面是一个原型，用于满足 find() 算法文档中描述的 EqualityComparable 概念需求：

```

1 class EqualityComparableArchetype

```

```

2 {
3 };
4
5 class ConvertibleToBoolArchetype
6 {
7     public:
8     operator bool() const;
9 };
10
11 ConvertibleToBoolArchetype
12 operator==(EqualityComparableArchetype const&,
13     EqualityComparableArchetype const&);

```

`EqualityComparableArchetype` 没有成员函数或数据，可以提供的唯一操作是重载 `operator==`，以满足 `find()` 的相等性要求。`operator==` 本身相当小，仅返回另一个原型 `ConvertibleToBoolArchetype`，其唯一定义的操作是用户定义的到 `bool` 的转换。

`EqualityComparableArchetype` 显然满足了 `find()` 模板的要求，可以通过使用 `EqualityComparableArchetype` 实例化 `find()` 来检查 `find()` 的实现是否在约定末端有效：

```

1 template int find(EqualityComparableArchetype const*,
2 EqualityComparableArchetype const&);

```

`find<equalitycomparablearchetype>` 的实例化将失败，表明我们已经发现了第一个问题：`EqualityComparable` 描述只需要 `operator==`，但 `find()` 的实现依赖于将 `T` 对象与 `!=` 进行比较。实现可以处理大多数用户定义的类型，可以将 `==` 和 `!=` 作为一对实现，但实际上这并不正确。原型旨在在模板库开发的早期发现这类问题。

改变 `find()` 的实现，使用相等来解决第一个问题，`find()` 模板将成功地（使用原型）编译：

程序将编译，但不会链接，因为从未定义重载 `operator==`。这是典型的原型，其通常只是作为编译时检查的辅助工具。

```

1 template<typename T>
2 int find(T const* array, int n, T const& value) {
3     int i = 0;
4     while(i != n && !(array[i] == value))
5         ++i;
6     return i;
7 }

```

使用原型展示 `find()` 中的第二个问题需要更多的技巧。`ind()` 的新定义现在应用了 `operator!` 直接指向 `operator==` 的结果。原型中，这依赖于用户定义的到 `bool` 的转换和内置的逻辑求反 `operator!`。`ConvertibleToBoolArchetype` 的实现会毒害 `operator!`，使其不能恰当地使用：

```

1 class ConvertibleToBoolArchetype
2 {
3     public:
4     operator bool() const;

```

```
5     bool operator!() = delete; // logical negation was not explicitly required  
6 };
```

可以使用删除函数扩展这个原型

删除函数是作为普通函数参与重载解析的函数。但若它们通过重载解析选择，编译器则会产生一个错误。

来修改操作符 `&&` 和 `||`，以发现其他模板定义中的问题。通常，模板实现者希望为模板库中确定的概念开发原型，然后使用这些原型根据其声明的需求测试每个模板的定义。

28.4. 跟踪

我们已经讨论了在编译或链接包含模板的程序时出现的 bug。在成功构建之后，最具挑战性的任务是确保程序在运行时的行为正确。因为模板所表示的泛型代码行为只依赖于该模板的外部使用（当然比普通的类和函数要多得多），所以模板有时会使这项任务变得困难。跟踪程序是一种软件工具，可以通过在开发早期检测模板定义中的问题来减轻这方面的工作。

跟踪程序是一个用户定义的类，可以用作要测试的模板的参数。跟踪程序也是一个原型，其只是为了满足模板的需求。跟踪程序应该生成对其调用操作的跟踪，可以通过实验验证算法的效率，以及操作的顺序。

下面是一个可以用来测试排序算法跟踪器的例子：

C++17 前，必须在翻译单元中在类声明之外初始化静态成员。

debugging/tracer.hpp

```
1 #include <iostream>  
2 class SortTracer {  
3     private:  
4         int value; // integer value to be sorted  
5         int generation; // generation of this tracer  
6         inline static long n_created = 0; // number of constructor calls  
7         inline static long n_destroyed = 0; // number of destructor calls  
8         inline static long n_assigned = 0; // number of assignments  
9         inline static long n_compared = 0; // number of comparisons  
10        inline static long n_max_live = 0; // maximum of existing objects  
11  
12        // recompute maximum of existing objects  
13        static void update_max_live() {  
14            if (n_created - n_destroyed > n_max_live) {  
15                n_max_live = n_created - n_destroyed;  
16            }  
17        }  
18  
19    public:
```

```

20     static long creations() {
21         return n_created;
22     }
23     static long destructions() {
24         return n_destroyed;
25     }
26     static long assignments() {
27         return n_assigned;
28     }
29     static long comparisons() {
30         return n_compared;
31     }
32     static long max_live() {
33         return n_max_live;
34     }
35
36     public:
37     // constructor
38     SortTracer (int v = 0) : value(v), generation(1) {
39         ++n_created;
40         update_max_live();
41         std::cerr << "SortTracer #" << n_created
42             << ", created generation " << generation
43             << " (total: " << n_created - n_destroyed
44             << ") \n";
45     }
46
47     // copy constructor
48     SortTracer (SortTracer const& b)
49     : value(b.value), generation(b.generation+1) {
50         ++n_created;
51         update_max_live();
52         std::cerr << "SortTracer #" << n_created
53             << ", copied as generation " << generation
54             << " (total: " << n_created - n_destroyed
55             << ") \n";
56     }
57
58     // destructor
59     ~SortTracer() {
60         ++n_destroyed;
61         update_max_live();
62         std::cerr << "SortTracer generation " << generation
63             << " destroyed (total: "
64             << n_created - n_destroyed << ") \n";
65     }
66
67     // assignment
68     SortTracer& operator= (SortTracer const& b) {

```

```

69     ++n_assigned;
70     std::cerr << "SortTracer assignment #" << n_assigned
71         << " (generation " << generation
72         << " = " << b.generation
73         << ") \n";
74     value = b.value;
75     return *this;
76 }
77
78 // comparison
79 friend bool operator < (SortTracer const& a,
80                         SortTracer const& b) {
81     ++n_compared;
82     std::cerr << "SortTracer comparison #" << n_compared
83         << " (generation " << a.generation
84         << " < " << b.generation
85         << ") \n";
86     return a.value < b.value;
87 }
88
89 int val() const {
90     return value;
91 }
92 };

```

除了用于排序的值，跟踪器还提供了几个成员来跟踪实际的排序：对于每个对象，生成跟踪的依据是从原始对象中删除了多少个复制操作。也就是说，原始操作 `generation == 1`，操作的直接副本为 `generation == 2`，副本的副本为 `generation == 3`，依此类推。其他静态成员会跟踪创建(构造函数调用)、销毁、赋值比较和最大值操作。

这个特殊的跟踪器允许跟踪实体创建和销毁的模式，以及给定模板执行的赋值和比较。下面的测试程序演示了 C++ 标准库的 `std::sort()` 算法：

debugging/tracertest.cpp

```

1 #include <iostream>
2 #include <algorithm>
3 #include "tracer.hpp"
4
5 int main()
6 {
7     // prepare sample input:
8     SortTracer input[] = { 7, 3, 5, 6, 4, 2, 0, 1, 9, 8 };
9
10    // print initial values:
11    for (int i=0; i<10; ++i) {
12        std::cerr << input[i].val() << ' ';
13    }
14    std::cerr << '\n';
15

```

```

16 // remember initial conditions:
17 long created_at_start = SortTracer::creations();
18 long max_live_at_start = SortTracer::max_live();
19 long assigned_at_start = SortTracer::assignments();
20 long compared_at_start = SortTracer::comparisons();

21
22 // execute algorithm:
23 std::cerr << "---[ Start std::sort() ]-----\n";
24 std::sort(&input[0], &input[9]+1);
25 std::cerr << "---[ End std::sort() ]-----\n";
26
27 // verify result:
28 for (int i=0; i<10; ++i) {
29     std::cerr << input[i].val() << ' ';
30 }
31 std::cerr << "\n\n";
32
33 // final report:
34 std::cerr << "std::sort() of 10 SortTracer's"
35     << " was performed by:\n"
36     << SortTracer::creations() - created_at_start
37     << " temporary tracers\n"
38     << "up to "
39     << SortTracer::max_live()
40     << " tracers at the same time ("
41     << max_live_at_start << " before)\n"
42     << SortTracer::assignments() - assigned_at_start
43     << " assignments\n"
44     << SortTracer::comparisons() - compared_at_start
45     << " comparisons\n\n";
46 }
```

运行这个程序会产生相当多的输出，但是可以从最终输出中得出很多结论。对于 `std::sort()` 函数的实现，有如下输出：

```

std::sort() of 10 SortTracer's was performed by:
9 temporary tracers
up to 11 tracers at the same time (10 before)
33 assignments
27 comparisons
```

例如，尽管在排序时在程序中创建了 9 个临时跟踪器，但最多有两个跟踪器存在。

因此，跟踪器扮演了两个角色：证明标准 `sort()` 算法不需要跟踪器的更多功能（不需要操作符 `==` 和 `>`），并且让我们了解算法的成本，这并不能说明排序模板的正确性。

28.5. 动态分析

跟踪器相对简单和有效，我们仅针对特定的输入数据和相关功能的特定行为跟踪模板的执行。比较操作符必须满足哪些条件才能使排序算法有意义(或正确)，但在示例中只测试了一个与小于操作符(整数)行为完全相同的比较操作符。

跟踪程序的扩展在一些圈子中称为 oracle(或运行时分析 oracle)，是连接到推理引擎的跟踪器——一个程序，可以记录关于断言和推断某些结论的原因。

某些情况下，Oracle 允许动态地验证模板算法，而无需完全指定替代模板参数 (oracle 就是参数) 或输入数据(当推理引擎遇到问题时，可能会请求某种输入假设)。然而，可以用这种方式分析算法的复杂性还好(因为推理引擎的限制)，但工作量相当大。出于这些原因，我们不深入研究 oracle 的发展史，有兴趣的读者可以查看后记中提到的出版物(和其中的参考资料)。

28.6. 后记

在 Jeremy Siek 的概念检查库(参见 [BCCL]) 中可以找到通过在高级模板中添加虚设代码来改进 C++ 编译器诊断的相当系统的尝试，是 Boost 库的一部分(参见 [Boost])。

Robert Klarer 和 John Maddock 提出了 static_assert 特性来帮助开发者在编译时检查条件，后来成为 C++11 的特性之一。在此之前，通常表示为一个库或宏，使用的技术类似于第 28.1 节中描述的那些技术。Boost.StaticAssert 库就是这样一种实现。

MELAS 系统为 C++ 标准库的某些部分提供了 oracle，允许对其一些算法进行验证。这个系统在 [MusserWangDynaVeri] 中进行了讨论。

其中一位作者 David Musser 也是 C++ 标准库开发的关键人物。除此之外，他还设计并实现了第一个关联容器。

附录 A：定义原则

定义规则称为 ODR，是 C++ 程序结构的基石。ODR 最常见的（很容易记住）应用：所有文件中只定义一次非内联函数或对象，每个翻译单元中最多定义一次类、内联函数和内联变量，确保对同一实体的所有定义相同。

问题在于细节，当与模板实例化结合使用时，这些细节可能会令人生畏。本附录旨在为感兴趣的读者提供 ODR 的全面概述。在正文中还指出了相关的问题。

A.1 翻译单元

实践中，通过用“代码”填充文件来编写 C++ 程序，文件在 ODR 上下文中并不是特别重要，重要的是翻译单位。本质上，翻译单元是将预处理器应用于提供给编译器的文件的结果，预处理器删除没有通过条件编译指令 (#if, #ifdef 和友元) 选择的代码段，删除注释，插入 #include 的文件（递归），并展开宏。

就 ODR 而言，以下两个文件

```
1 // header.hpp:  
2 #ifdef DO_DEBUG  
3 #define debug(x) std::cout << x << '\n'  
4 #else  
5 #define debug(x)  
6 #endif  
7  
8 void debugInit();  
9  
10 // myprog.cpp:  
11 #include "header.hpp"  
12 int main()  
13 {  
14     debugInit();  
15     debug("main()");  
16 }
```

等价于以下单个文件：

```
1 // myprog.cpp:  
2 void debugInit();  
3 int main()  
4 {  
5     debugInit();  
6 }
```

跨翻译单元的连接，是通过在两个翻译单元中具有相应的外部链接声明（例如，两个全局函数 debugInit() 的声明）来建立的。

翻译单元的概念比“预处理文件”更抽象一些。若将一个预处理文件两次提供给编译器以形成一个程序，编译器将把两个不同的翻译单元带入程序（然而，这样做没什么意义）。

A.2 声明和定义

术语声明和定义在常见的“开发者对话”中经常交替使用。在 ODR 的范围内，这些词的含义很重要。

交换关于 C 和 C++ 的观点时，仔细处理术语是一个好习惯。我们在整本书中都是这样做的。

声明是一种 C++ 构造，(通常)

有些结构 (如 `static_assert`) 不引入名称，但在语法上视为声明。

程序中引入或重新引入一个名称。声明也可以是定义，这取决于它引入了哪个，以及如何引入：

- **命名空间和命名空间别名**: 命名空间及其别名的声明也需要定义，尽管术语定义在此上下文中并不常见，因为命名空间的成员列表可以在以后进行“扩展”(例如，与类和枚举类型不同)。
- **类、类模板、函数、函数模板、成员函数和成员函数模板**: 当声明包含与名称相关联的大括号正文时，声明就是定义。该规则包括联合、操作符、成员操作符、静态成员函数、构造函数和析构函数，以及这些东西的模板版本的显式特化(即任何类实体和函数实体)。
- **枚举**: 当声明包含用大括号括起来的枚举数列表时，该声明是定义。
- **局部变量和非静态数据成员**: 这些可以视为定义(尽管区别很少)，函数定义中的函数参数声明本身就是一个定义，因为它表示一个局部变量，但函数声明中的函数参数不是定义。
- **全局变量**: 若声明之前没有使用关键字 `extern`，或者有初始化式，那么全局变量的声明也是该变量的定义。否则，就不是定义。
- **静态数据成员**: 当声明出现在成员的类或类模板外部，或者在类或类模板中内联或 `constexpr` 声明时，声明就是定义。
- **显式和偏特化**: 若 `template<>` 或 `template<…>` 本身是一个定义，只是静态数据成员或静态数据成员模板的显式特化，只有在包含初始化式时才是定义。

其他声明不是定义。这包括类型别名(带有 `typedef` 或 `using`)、`using` 声明、`using` 指令、模板参数声明、显式实例化指令、`static_assert` 声明等。

A.3 定义原则的细节

本附录的介绍中所述，ODR 有很多细节。我们根据规则的范围，逐个介绍相应约束。

A.3.1 程序的约束

每个程序最多只能有以下的一个定义：

- 非内联函数和非内联成员函数(包括函数模板的全特化)
- 非内联变量(本质上是在命名空间作用域或全局作用域中声明的变量，没有静态说明符)
- 非内联静态数据成员

例如，由以下两个翻译单元组成的 C++ 程序无效：

```
1 // translation unit 1:  
2 int counter;  
3  
4 // translation unit 2:  
5 int counter; // ERROR: defined twice (ODR violation)
```

此规则不适用于具有内部链接的实体（本质上是在全局作用域或命名空间作用域中，使用静态说明符声明的实体），即使两个这样的实体具有相同的名称，也会认为是不同的。同样，在匿名命名空间中声明的实体，若出现在不同的翻译单元中，则认为是不同的；C++11 及以后的版本中，这些实体默认也具有内部链接，但在 C++11 前，默认具有外部链接。以下两个翻译单元可以组合成一个有效的 C++ 程序：

```
1 // translation unit 1:  
2 static int counter = 2; // unrelated to other translation units  
3  
4 namespace {  
5     void unique() // unrelated to other translation units  
6     {}  
7 }  
8  
9 // translation unit 2:  
10 static int counter = 0; // unrelated to other translation units  
11 namespace {  
12     void unique() // unrelated to other translation units  
13     {  
14         ++counter;  
15     }  
16 }  
17  
18 int main()  
19 {  
20     unique();  
21 }
```

此外，若在 `constexpr if` 语句的丢弃分支之外的上下文中使用，则程序中必须只有前面提到的一项（该特性仅在 C++17 中可用；参见第 14.6 节）。在上下文中使用的术语具有确切的含义，表明在程序的某个地方存在对实体的某种引用，所以在直接的代码生成中需要实体。

各种优化技术可能会对实体进行删除，但该语言没有假设这种优化。

此引用可以是对变量值的访问、对函数的调用或此类实体的地址。这个引用可以在源文件中显式引用，也可以隐式引用。`new` 表达式可以隐式地创建对 `delete` 操作符的调用，以处理当构造函数抛出异常要求清理未使用（但已分配）内存时的情况。另一个例子包括复制构造函数，即使最终优化掉了，也必须定义（除非语言要求优化，C++17 中经常出现这种情况）。虚函数也是隐式使用（支持虚函数调用的内部结构使用），除非是纯虚函数。还有其他几种隐式用法，但为了简明起见，这里暂时忽略它们。

有些引用不构成使用: 出现在未求值操作数中的引用 (sizeof 或 decltype 操作符的操作数)。typeid 操作符 (参见第 9.1.1 节) 的操作数只在某些情况下不计算, 若引用作为 typeid 操作符的一部分出现就不使用, 除非 typeid 操作符的参数最终指定了一个多态对象 (可能继承了虚函数的对象)。例如, 以下单文件程序:

```
1 #include <typeinfo>
2
3 class Decider {
4     #if defined(DYNAMIC)
5     virtual ~Decider() {
6     }
7     #endif
8 };
9
10 extern Decider d;
11
12 int main()
13 {
14     char const* name = typeid(d).name();
15     return (int)sizeof(d);
16 }
```

当预处理器符号 DYNAMIC 未定义时, 这是一个有效的程序。变量 d 没有定义, 但是 sizeof(d) 中对 d 的引用并不构成使用, 而 typeid(d) 中的引用, 只有在 d 是多态类型对象时才构成使用 (在运行时之前, 不总确定多态类型 id 操作的结果)。

根据 C++ 标准, 本节描述的约束不需要从 C++ 实现进行诊断, 这些符号就是链接器报告为重复或缺失的定义。

A.3.2 翻译单元的限制

翻译单元中, 实体都不能定义超过一次。所以下面的例子无效:

```
1 inline void f() {}
2 inline void f() {} // ERROR: duplicate definition
```

这也是在头文件中使用守卫包围代码的主要原因之一:

```
1 // guarddemo.hpp:
2 #ifndef GUARDDMO_HPP
3 #define GUARDDMO_HPP
4 ...
5 #endif // GUARDDMO_HPP
```

这样的保护确保头文件第二次 #include 时, 丢弃内容, 从而避免其类、内联实体、模板等的重复定义。

ODR 还指定必须在某些情况下定义某些实体。这可能是类类型、内联函数和内联变量的情况。接下来, 我们将回顾详细的规则。

类类型 X(包括结构体和联合体) 必须在翻译单元中定义, 才能在该翻译单元中进行下列类型的使用:

- 创建类型为 X 的对象 (例如，通过变量声明或通过 new 表达式)。当创建本身包含 X 类型对象的对象时，可以间接创建。
- 类型 X 的数据成员声明。
- 对 X 类型的对象应用 sizeof 或 typeid 操作符。
- 显式或隐式地访问类型 X 的成员。
- 使用类型的转换将表达式转换为或从类型 X 转换，或者使用隐式强制转换、static_cast 或 dynamic_cast 换将表达式转换为指向 X 的指针或引用 (void* 除外)。
- 将值赋给 X 类型的对象。
- 定义或调用带有参数或返回类型为 X 的函数，但声明这样的函数并不需要定义其类型。

类型规则也适用于从类模板生成的类型 X，需要在定义类型 X 的情况下定义相应的模板。这些情况会创建实例化点或 POI(参见第 14.3.2 节)。

内联函数必须使用每个翻译单元中定义 (这些翻译单元中调用它们或获取它们的地址)，但与类类型不同，它们的定义可以遵循以下使用要点：

```

1 inline int notSoFast();
2
3 int main()
4 {
5     notSoFast();
6 }
7
8 inline int notSoFast()
9 { }
```

虽然这是有效的 C++ 代码，但基于旧技术的编译器实际上并不会“内联”调用尚未看到主体的函数；因此，预期的效果可能无法达到。

与类模板一样，使用由参数化函数声明 (函数或成员函数模板，或类模板的成员函数) 生成的函数会创建一个实例化点。但与类模板不同，相应的定义可以出现在实例化点之后。

本小节中解释的 ODR 的各个方面通常很容易通过 C++ 编译器验证；因此，C++ 标准要求编译器在违反这些规则时发出某种诊断。异常是缺少参数化函数的定义，这种情况通常没有诊断信息。

A.3.3 交叉翻译单元的等价约束

多个翻译单元中定义某些类型可能会带来新的错误：多个定义不匹配。但传统的编译器每次只处理一个翻译单元，所以很难检测到这些错误。因此，C++ 标准并不要求检测或诊断多个定义之间的差异，但若违反交叉转换单元的约束，C++ 标准将其限定为未定义行为，从而任何合理或不合理的事情都可能发生。通常，这种未诊断的错误可能导致程序崩溃或错误的结果，但也可能导致其他更直接的损害 (例如，文件损坏)。

gcc 编译器的第 1 版实际上没有做到了这一点，只是某些情况下启动了“肉鸽游戏”。

交叉翻译单元约束指定，实体在两个不同的地方定义时，这两个地方必须包含完全相同的标记序列 (预处理之后保留的关键字、操作符、标识符等)。此外，这些标记必须在它们各自的上下文中

的表示相同(例如, 标识符可能需要引用相同的变量)。

考虑下面的例子:

```
1 // translation unit 1:  
2 static int counter = 0;  
3 inline void increaseCounter()  
4 {  
5     ++counter;  
6 }  
7  
8 int main()  
9 { }  
10  
11 // translation unit 2:  
12 static int counter = 0;  
13 inline void increaseCounter()  
14 {  
15     ++counter;  
16 }
```

这个例子是错误的, 因为即使内联函数 `incrementcounter()` 的标记序列在两个翻译单元中看起来相同, 从而包含一个指向两个不同实体的标记计数器。不过, 因为名为 `counter` 的两个变量具有内部链接(静态说明符), 所以尽管名称相同, 但不相关。即使没有实际使用内联函数, 这也是一个错误。

#included 头文件中可以放置在多个翻译单元中的实体定义, 无论何时需要定义, 都要包含这些定义, 确保在所有情况下标记序列都相同。

条件编译指令在不同的翻译单元中会有不同的计算结果, 所以小心使用这些指令。其他更不常见差异也可能存在。

使用这种方法, 两个相同的标记引用不同事物的情况将变得相当罕见。发生时, 产生的错误往往是隐秘的, 难以查找。

交叉翻译单元约束不仅可以在多个地方定义的实体, 还适用于声明中的默认参数。换句话说, 下面的程序有未定义行为:

```
1 // translation unit 1:  
2 void unused(int = 3);  
3 int main()  
4 { }  
5  
6 // translation unit 2:  
7 void unused(int = 4);
```

这里, 标记流的等价有时会涉及微妙的隐式影响。下面的例子是从 C++ 标准中提取出来的(稍微修改了一下):

```
1 // translation unit 1:  
2 class X {  
3     public:4 }
```

```

4   X(int, int);
5   X(int, int, int);
6 };
7
8 X::X(int, int = 0)
9 { }
10
11 class D {
12   X x = 0;
13 };
14
15 D d1; // X(int, int) called by D()
16
17 // translation unit 2:
18 class X {
19 public:
20   X(int, int);
21   X(int, int, int);
22 };
23
24 X::X(int, int = 0, int = 0)
25 { }
26
27 class D : public X {
28   X x = 0;
29 };
30
31 D d2; // X(int, int, int) called by D()

```

本例中，出现问题是因为类 D 隐式生成的默认构造函数在两个翻译单元中不同。一个调用带两个参数的 X 构造函数，另一个调用带三个参数的 X 构造函数。如果有任何区别的话，就是将默认参数限制在程序中的一个位置(可能的话，这个位置应该在头文件中)的动机。幸运的是，在类外定义上放置默认参数是一种少见的做法。

对于相同的标记必须引用相同的实体规则，当然也有例外。如果相同的标记引用了具有相同值的不相关常量，并且没有使用结果表达式的地址(甚至通过将引用绑定到产生该常量的变量，也没有隐式地使用)，那么这些标记等价。该异常允许的程序结构如下所示：

```

1 // header.hpp:
2 #ifndef HEADER_HPP
3 #define HEADER_HPP
4
5 int const length = 10;
6
7 class MiniBuffer {
8   char buf[length];
9   ...
10 };
11
12 #endif // HEADER_HPP

```

原则上，当头文件包含在两个不同的翻译单元中时，将创建两个名为 `length` 的不同常量。因为在此上下文中，`const` 意味着 `static`。然而，这种常量变量通常用于定义编译时常量值，而不是运行时的特定存储位置。因此，若不强制存在这样一个存储位置（通过引用变量的地址），两个常量有相同的值就够了。

最后，关于模板的说明。模板中的名称绑定分为两个阶段，非依赖名称在模板定义点绑定。对于这些规则，等价规则的处理类似于其他非模板定义。对于在实例化点绑定的名称，必须在该点应用等价规则，绑定必须等价。

附录 B：值类别

表达式是 C++ 语言的基石，提供了表达计算的主要机制。每个表达式都有一个类型，该类型描述其计算产生的值的静态类型。表达式 7 是 int 类型，表达式 $5 + 2$ 也是。若 x 是 int 类型的变量，则表达式 x 是 int 类型。每个表达式也有一个值类别，描述值如何形成，以及如何影响表达式的行为。

B.1 传统的左值和右值

历史上，只有两种值类别：左值和右值。左值是指存储在内存或机器寄存器中实际值的表达式，例如表达式 x，其中 x 是变量名。这些表达式可以修改，从而允许更新存储的值。若 x 是一个 int 类型的变量，下面的赋值会将 x 的值替换为 7：

```
1 x = 7;
```

术语左值来自于这些表达式在赋值中可能扮演的角色：字母“l”代表“左手边”，因为（在 C 中）只有左值可能出现在赋值的左手边。相反，右值（“r”代表“右手边”）只能出现在赋值表达式的右手边。

1989 年 C 语言标准化后，情况发生了变化：当 int const 仍然是存储在内存中的值时，不能出现在赋值操作的左侧：

```
1 int const x; // x is a nonmodifiable lvalue
2 x = 7; // ERROR: modifiable lvalue required on the left
```

C++ 进一步改变了这一点：类的右值可以出现在赋值的左侧。这样的赋值实际上是对类的适当赋值操作符的函数调用，而不是对标量类型的“简单”赋值，因此遵循（单独的）成员函数调用规则。

由于所有这些变化，术语左值现在有时称为本地化值。引用变量的表达式并不是唯一一种左值表达式。另一类左值表达式包括指针解引用操作（例如，`*p`）和类对象成员的表达式（例如，`p->data`）。解引用操作指向存储在指针引用地址的值。即使调用返回用 & 声明的“传统”左值引用类型的函数也是左值。例如（详见第 B.4 节）：

```
1 std::vector<int> v;
2 v.front() // yields an lvalue because the return type is an lvalue reference
```

字符串字面值也是（不可修改的）左值。

右值是纯粹的数学值（如 7 或字符'a'），不一定有相关的存储；它们的存在就为了计算，但当使用它们时就不能再引用。除了字符串字面量（例如，7，'a'，true，nullptr）之外的字面值都是右值，就像许多内置算术计算（例如， $x + 5$ 对于整数类型的 x）和调用按值返回结果的函数一样，所有的临时变量都是右值（不过，这不适用于引用它们的命名引用）。

B.1.1 左值到右值的转换

由于右值的短暂性，只能出现在赋值语句（“简单”）的右侧：赋值语句 $7 = 8$ 是没有意义的，因为不允许重新定义数学上的 7。另一方面，左值似乎没有同样的限制：当 x 和 y 是兼容类型的变量时，可以计算赋值 $x = y$ ，即使表达式 x 和 y 都是左值。

赋值 $x = y$ 可行，因为右边的表达式 y 进行了隐式转换，称为左值到右值转换。顾名思义，左值可以转换为右值，并通过从与左值相关联的存储或寄存器中读取该左值，来生成相同类型的右值。因此，这种转换完成了两件事：第一，确保左值可以在任何需要右值的地方使用（作为赋值的右手边或在数学表达式中，如 $x + y$ ）。第二，确定了在程序中（优化之前）编译器发出“load”指令从内存中读取值的位置。

B.2 C++11 的值类别

当 C++11 中引入右值引用以支持移动语义时，将表达式划分为左值和右值的方法已不足以描述 C++11 的所有语言行为。因此，C++ 标准化委员会基于三个核心类别和两个复合类别，重新设计了价值类别系统（参见图 B.1）。核心类别是：左值、prvalue（“纯右值”）和 xvalue（“过期值”）。复合类别是：glvalue（“广义左值”，是左值和 xvalue 的并集）和 rvalue（xvalue 和 prvalue 的并集）。

所有表达式仍然是左值或右值，但右值类别现在会进一步细分。

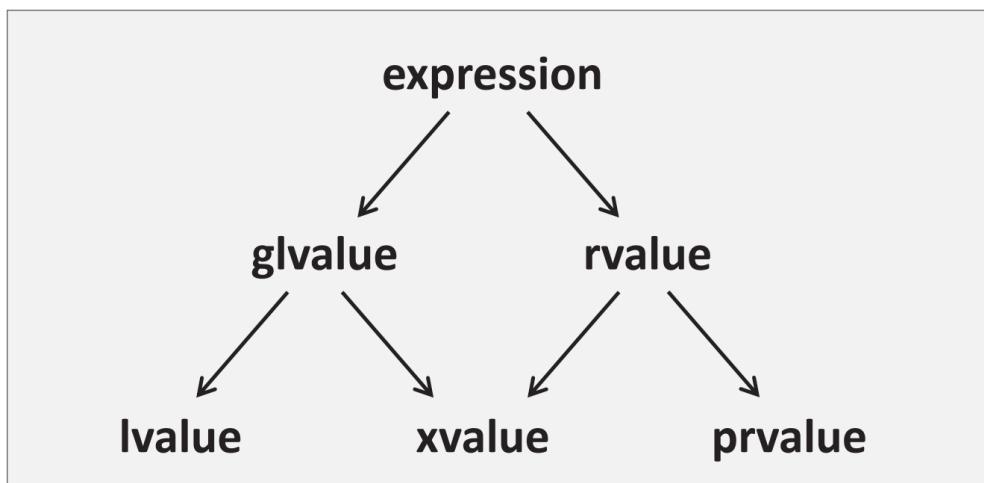


图 B.1. 自 C++11 后的值类别

C++11 的分类在 C++17 中仍然有效，但 C++17 的分类重新表述如下：

- glvalue 是一个表达式，其计算值决定了对象、位域或函数（即具有存储空间的实体）的标识。
- prvalue 是一个表达式，其求值是初始化一个对象或位域，或计算操作符的操作数的值。
- xvalue 是 glvalue，指定一个对象或位域，其资源可以重用（通常是因为其即将“过期”—xvalue 中的“x”最初来自“过期值”（eXpiring value））。
- 左值为不是 xvalue 的 glvalue。
- 右值为 prvalue 或 xvalue 的表达式。

C++17 中（某种程度上，C++11 和 C++14 也是如此），glvalue 和 prvalue 的区别比传统的左值和右值的区别更为基础。

虽然这描述了在 C++17 中引入的特性，这些描述也适用于 C++11 和 C++14（之前的描述是相同的，但很难推断）。

除位域外，glvalue 生成带有地址的实体。该地址可以是更大的封闭对象子对象的地址。在基类子对象的情况下，glvalue（表达式）类型称为其静态类型，而基类所属的派生对象类型，称为 glvalue 的动态类型。若 glvalue 不生成基类子对象，则其静态和动态类型相同（即表达式的类型）。

左值的例子有:

- 指定变量或函数的表达式
- 内置一元乘法运算符(“指针间接”)的应用
- 只是字符串字面量的表达式
- 调用返回类型为左值引用的函数

prvalue 的例子有:

- 由非字符串字面量或用户定义字面量的字面量组成的表达式

用户定义的字面量可以生成左值或右值，这取决于相关字面量操作符的返回类型。

- 内置的一元 & 操作符的应用(例如，取表达式的地址)
- 内置算术运算符的应用
- 对返回类型不是引用类型函数的调用
- Lambda 表达式

xvalue 的例子有:

- 调用返回类型为对象类型右值引用的函数(std::move())
- 转换为对对象类型的右值引用

对函数类型的右值引用会产生左值，而不是 xvalue。

需要强调的是，glvalue、prvalue、xvalue 等都是表达式，而不是值

这些术语属于用词不当。

或实体。例如，变量不是左值，尽管表示该变量的表达式是左值:

```
1 int x = 3; // x here is a variable, not an lvalue. 3 is a prvalue initializing
2     // the variable x.
3 int y = x; // x here is an lvalue. The evaluation of that lvalue expression does not
4     // produce the value 3, but a designation of an object containing the value 3.
5     // That lvalue is then then converted to a prvalue, which is what initializes y.
```

B.2.1 临时实现

前面提到过，左值通常要进行到右值的转换

使用 C++11 的值分类，短语 glvalue 到 prvalue 的转换会更准确，但传统术语仍然更为常见。

因为值是初始化对象(或为大多数内置操作符提供操作数)的表达式类型。

C++17 中，这种转换有一个对偶形式，称为临时实现(也可以称为“glvalue 到 prvalue 的转换”):只要期望 glvalue(包括 xvalue) 出现的 prvalue 有效，就会创建一个临时对象，并使用 prvalue 初始化(回想一下，prvalue 主要是“初始化值”)，然后用指定临时的 xvalue 替换 prvalue。例如:

本例中的 f() 有一个引用参数，所以需要一个 glvalue 参数。表达式 3 是一个 prvalue，因此“临时实现”规则开始发挥作用。表达式 3 “转换”为一个 xvalue，该 xvalue 指定一个用值 3 初始化的临时对象。

在以下情况下，临时对象会实现，并用 prvalue 进行初始化：

- prvalue 绑定到引用 (上面调用 f(3))。
- 访问类 prvalue 的成员。
- 数组 prvalue 的下标。
- 数组 prvalue 转换为指向其第一个元素的指针 (数组衰变)。
- prvalue 出现在一个带括号的初始化列表中，对于某些类型 X，其初始化 std::initializer_list<X> 类型的对象。
- 将 sizeof 或 typeid 操作符应用于 prvalue。
- prvalue 是“expr;”形式的语句中的顶层表达式，或者将表达式转换为 void。

C++17 中，由 prvalue 初始化的对象由上下文决定，因此临时对象只在真正需要时才创建。C++17 前，prvalue(特别是类)总是隐含一个临时变量。这些临时变量的副本以后可以有选择地省略，但是编译器仍然需要强制执行复制操作的大多数语义约束(需要复制构造函数可调用)。下面的例子展示了 C++17 修改规则的结果：

```
1 class N {  
2     public:  
3     N();  
4     N(N const&) = delete; // this class is neither copyable ...  
5     N(N&&) = delete; // ... nor movable  
6 };  
7  
8 N make_N() {  
9     return N{}; // Always creates a conceptual temporary prior to C++17.  
10 }           // In C++17, no temporary is created at this point.  
11  
12 auto n = make_N(); // ERROR prior to C++17 because the prvalue needs a  
13     // conceptual copy. OK since C++17, because n is  
14     // initialized directly from the prvalue.
```

C++17 前，prvalue N{} 生成了一个类型为 N 的临时变量，允许编译器删除该临时变量的复制和移动构造(实际上它们总是这样做的)。这意味着调用 make_N() 的临时结果可以直接在 N 的存储中构造，不需要进行复制或移动操作。但 C++17 前的编译器仍需要检查，是否可以进行复制或移动操作，在本例中这是不可能的，因为 N 的复制构造函数删除了(并且没有生成移动构造函数)。因此，C++11 和 C++14 编译器必须对这个例子产生错误信息。

C++17 中，prvalue N 本身不会产生一个临时变量，会初始化了一个由上下文决定的对象：这个对象是 N 表示的对象。不考虑复制或移动操作(这不是优化，而是语言保证)，因此代码是有效的 C++17 代码。

我们以一个展示各种值类别情况的例子作为结束：

```
1 class X {  
2 };
```

```

3
4 X v;
5 X const c;
6
7 void f(X const&); // accepts an expression of any value category
8 void f(X&&); // accepts prvalues and xvalues only but is a better match
9     // for those than the previous declaration
10
11 f(v); // passes a modifiable lvalue to the first f()
12 f(c); // passes a nonmodifiable lvalue to the first f()
13 f(X()); // passes a prvalue (since C++17 materialized as xvalue) to the 2nd f()
14 f(std::move(v)); // passes an xvalue to the second f()

```

B.3 使用 decltype 检查值类别

使用关键字 `decltype`(在 C++11 中引入), 可以检查 C++ 表达式的值类别。对于任意表达式 `x`, `decltype((x))`(注意双括号) 会产生:

- `x` 是类型, 则为 prvalue
- `x` 是左值引用, 则为 lvalue
- `x` 是右值引用, 则为 xvalue

`decltype((x))` 中需要双括号, 以避免在表达式 `x` 确实命名了实体的情况下, 产生命名实体的声明类型(其他情况下, 括号不起作用)。若表达式 `x` 只是将一个变量命名为 `v`, 那么不带圆括号的构造就变成了 `decltype(v)`, 其生成变量 `v` 的类型, 而不是反映引用该变量的表达式 `x` 的值类别的类型。

因此, 对任意表达式 `e` 使用类型特征, 可以使用如下方法检查其值的类别:

```

1 if constexpr (std::is_lvalue_reference<decltype((e))>::value) {
2     std::cout << "expression is lvalue\n";
3 }
4 else if constexpr (std::is_rvalue_reference<decltype((e))>::value) {
5     std::cout << "expression is xvalue\n";
6 }
7 else {
8     std::cout << "expression is prvalue\n";
9 }

```

详见第 15.10.2 节。

B.4 引用类别

C++ 中的引用类型(如 `int&`)以两种重要的方式与值类别交互。第一个是引用可能会限制, 可以绑定到的表达式的值类别。`int&` 类型的非 `const` 左值引用, 只能用 `int` 类型的左值表达式初始化。类似地, `int` 类型的右值引用, 只能用 `int` 类型的右值表达式初始化。

值类别与引用交互的第二种方式是与函数的返回类型交互, 其中使用引用类型作为返回类型会影响对该函数调用的值类别。特别是:

- 对返回类型为左值引用的函数，将生成左值。
- 若函数的返回类型，是对象类型的右值引用，调用该函数会产生 xvalue(对函数类型的右值引用会产生左值)。
- 调用返回非引用类型的函数会产生 prvalue。

下面的示例中，将演示引用类型和值类别之间的交互。

```
1 int& lvalue();
2 int&& xvalue();
3 int prvalue();
```

给定表达式的值类别和类型可以通过 decltype 来确定。如第 15.10.2 节所述，使用引用类型来描述表达式何时为左值或 xvalue:

```
1 std::is_same_v<decltype(lvalue()), int&> // yields true because result is lvalue
2 std::is_same_v<decltype(xvalue()), int&&> // yields true because result is xvalue
3 std::is_same_v<decltype(prvalue()), int> // yields true because result is prvalue
```

因此，可以进行以下的调用:

```
1 int& lref1 = lvalue(); // OK: lvalue reference can bind to an lvalue
2 int& lref3 = prvalue(); // ERROR: lvalue reference cannot bind to a prvalue
3 int& lref2 = xvalue(); // ERROR: lvalue reference cannot bind to an xvalue
4
5 int&& rref1 = lvalue(); // ERROR: rvalue reference cannot bind to an lvalue
6 int&& rref2 = prvalue(); // OK: rvalue reference can bind to a prvalue
7 int&& rref3 = xvalue(); // OK: rvalue reference can bind to an xrvalue
```

附录 C：重载解析

重载解析是为给定调用表达式选择要调用函数的过程。看看下面这个简单的例子：

```
1 void display_num(int); // #1
2 void display_num(double); // #2
3
4 int main()
{
    display_num(399); // #1 matches better than #2
    display_num(3.99); // #2 matches better than #1
}
```

例子中，函数名 `display_num()` 重载，当在调用中使用这个名称时，C++ 编译器必须使用更多信息来区分不同的候选名称；大多数情况下，这些信息是调用参数的类型。我们的例子中，当函数调用一个整型参数时调用整型版本，当函数提供一个浮点参数时调用双精度浮点版本就很简单。尝试对这种简单选择建模的流程就是重载解析。

指导重载解析规则背后的思想很简单，但在 C++ 标准化过程中，细节变得相当复杂。这种复杂性主要是由支持各种实际示例愿望所驱使，这些示例似乎具有“明显最佳匹配”，但试图将这种直觉形式化时，各种微妙的问题就出现了。

本附录中，我们将对重载解析规则进行较为详细的介绍。这个过程非常复杂，我们不能涵盖该主题的每个部分。

C.1 何时应用重载解析

重载解析只是函数调用完整处理的一部分，并不是每个函数调用的一部分。首先，通过函数指针和通过指向成员函数的指针进行的调用不受重载解析的影响，因为要调用的函数完全（在运行时）由指针决定。其次，类函数宏不能重载，因此不需要重载解析。

在高层次上，对命名函数的调用可以按以下方式处理：

- 查找名称以形成初始重载集。
- 若有必要，这个集合会以各种方式进行调整（发生模板参数推导和替换，这会导致丢弃一些函数模板候选）。
- 完全不匹配的候选（考虑隐式转换和默认参数之后）将从重载集中删除。这就产生了一组匹配的候选函数。
- 执行重载解析以找到最佳候选。如果有，则选中；否则，调用具有歧义。
- 选中的候选需要检查，若是一个已删除函数（即用 `=delete` 定义的函数）或一个不可访问的私有成员函数，则编译器会发出错误信息。

每个步骤都有自己的微妙之处，但是重载解析无疑是复杂的。幸运的是，一些简单的原则可以解释大多数情况。接下来我们将研究这些原则。

C.2 简化的重载解析

重载解析通过比较调用的每个参数与候选函数对应参数的匹配程度，来对可行候选函数进行排序。为了使一个候选函数比另一个更好，更好的候选不能有参数比另一个候选的相应参数匹配的差。下面的例子说明了这一点：

```
1 void combine(int, double);
2 void combine(long, int);
3
4 int main()
5 {
6     combine(1, 2); // ambiguous!
7 }
```

本例中，对 `combine()` 的调用具有歧义，因为第一个候选参数最匹配第一个参数 (`int` 类型的 1 字面值)，而第二个候选参数最匹配第二个参数。可以说 `int` 在某种意义上比 `double` 更接近 `long`(支持选择第二个候选项)，但是 C++ 并没有定义包含多个调用参数的匹配度指标。

有了这第一原则，就需要指定给定参数与可行候选参数的对应参数的匹配程度。作为第一次近似的候选，可以将可能的匹配按如下顺序排列(从最好到最差)：

1. 完美匹配。参数具有表达式的类型，或者具有指向表达式类型的引用类型(可能添加了 `const` 和/或 `volatile` 限定符)。
2. 匹配需要微调。将数组变量衰变为指向其第一个元素的指针，或将 `const` 相加以匹配 `int**` 类型的参数和 `const* const* int` 类型的参数。
3. 类型升级匹配。类型升级是一种隐式转换，包括将小整型(如 `bool`、`char`、`short`，有时还包括枚举)转换为 `int`、`unsigned int`、`long` 或 `unsigned long`，以及将 `float` 转换为 `double`。
4. 只匹配标准转换。任何类型的标准转换(如 `int` 到 `float`)或从派生类到其公共、明确的基类的转换，但不包括对转换操作符或转换构造函数的隐式调用。
5. 匹配用户定义的转换。这允许任何类型的隐式转换。
6. 匹配省略号(`...`)，省略号参数几乎可以匹配任何类型，但有一个例外：具有重要复制构造函数的类型可能是有效的，也可能不是有效的(实现可以进行允许或禁止)。

下面的例子展示了其中的一些匹配方式：

```
1 int f1(int); // #1
2 int f1(double); // #2
3 f1(4); // calls #1 : perfect match (#2 requires a standard conversion)
4
5 int f2(int); // #3
6 int f2(char); // #4
7 f2(true); // calls #3 : match with promotion
// (#4 requires stronger standard conversion)
8
9
10 class X {
11     public:
12     X(int);
13 };
14 int f3(X); // #5
```

```

15 int f3(...); // #6
16 f3(7); // calls #5 : match with user-defined conversion
17 // (#6 requires a match with ellipsis)

```

重载解析发生在模板参数推导后，而且推导不考虑所有这些类型的转换。例如：

```

1 template<typename T>
2 class MyString {
3     public:
4         MyString(T const*); // converting constructor
5         ...
6     };
7
8 template<typename T>
9 MyString<T> truncate(MyString<T> const&, int);
10
11 int main()
12 {
13     MyString<char> str1, str2;
14     str1 = truncate<char>("Hello World", 5); // OK
15     str2 = truncate("Hello World", 5); // ERROR
16 }

```

模板参数推导期间，不会考虑通过转换构造函数提供的隐式转换。`str2` 的赋值没有找到可行函数 `truncate()`，因此不执行重载解析。

模板参数推导的上下文中，若对应的参数是左值，则对模板参数的右值引用可以推断为左值引用类型(在引用折叠之后)。若该参数是右值，则可以推断为右值引用类型(参见 15.6 节)。例如：

```

1 template<typename T> void strange(T&&, T&&);
2 template<typename T> void bizarre(T&&, double&&);

3
4 int main()
5 {
6     strange(1.2, 3.4); // OK: with T deduced to double
7     double val = 1.2;
8     strange(val, val); // OK: with T deduced to double&
9     strange(val, 3.4); // ERROR: conflicting deductions
10    bizarre(val, val); // ERROR: lvalue val doesn't match double&&
11 }

```

前面的原则只是一种近似方式，但覆盖了许多情况。有相当多的常见情况，这些规则没有充分解释。我们将继续讨论这些规则的最重要改进。

C.2.1 成员函数的隐含参数

对非静态成员函数的调用具有一个隐藏参数，成员函数的定义中可访问该参数为 `*this`。对于类 `MyClass` 的成员函数，隐藏参数的类型通常为 `MyClass&`(用于非 `const` 成员函数) 或 `MyClass const&`(用于 `const` 成员函数)。

若成员函数是 volatile 类型,也可以是 MyClass volatile& 类型,或者 MyClass const volatile& 类型,但这种情况非常罕见。

考虑到其具有指针类型,如果让这个等于现在的 *this 就更好了。但这是 C++ 早期版本的一部分,在引用类型成为语言的一部分之前,并且当添加引用类型时,太多的代码已经依赖于 this 指针了。

隐藏的 *this 参数和显式参数一样参与重载解析,但偶尔也会发生意外。下面的例子显示了一个类似字符串的类,其不能按预期工作(这种代码实际存在):

```
1 #include <cstddef>
2
3 class BadString {
4     public:
5     BadString(char const*);
6     ...
7
8     // character access through subscripting:
9     char& operator[] (std::size_t); // #1
10    char const& operator[] (std::size_t) const;
11
12    // implicit conversion to null-terminated byte string:
13    operator char* () ; // #2
14    operator char const* () ;
15    ...
16 };
17
18 int main()
19 {
20     BadString str("correkt");
21     str[5] = 'c' ; // possibly an overload resolution ambiguity!
22 }
```

首先,表达式 str[5] 似乎没有任何问题。在 #1 处的下标操作符似乎是完美的匹配。但并不完美,因为参数 5 是 int 类型,而操作符需要无符号整型(size_t 和 std::size_t 通常有 unsigned int 类型或 unsigned long 类型,但从来没有 int 类型)。尽管如此,一个简单的标准整数转换使 #1 很容易实现。还有另一种候选方法:内置下标操作符。若将隐式转换操作符应用于 str(隐式成员函数参数),就会获得指针类型,现在内置下标操作符也会应用。这个内置操作符接受一个ptrdiff_t 类型的参数,该参数在许多平台上等价于 int,因此参数 5 完美匹配。因此,尽管内置下标操作符对隐含参数匹配的很差(通过用户定义的转换),但比在 #1 处定义的操作符对实际下标的匹配更好!因此,可能存在歧义。

这种歧义只存在于 size_t 是 unsigned int 的平台上。在它是 unsigned long 平台上,类型ptrdiff_t 是 long 的类型别名,并且不存在歧义,因为内置的下标操作符也需要对下标表达式进行转换。

为了可移植地解决这类问题，可以用一个 `ptrdiff_t` 参数声明 `operator[]`，或者可以用显式转换来替换隐式到 `char*` 的类型转换（这是推荐做法）。

一组候选成员可能同时包含静态成员和非静态成员。当比较静态成员和非静态成员时，将忽略隐式参数的匹配度（只有非静态成员具有隐式的 `*this` 参数）。

默认情况下，非静态成员函数有一个隐含的 `*this` 参数，是左值引用类型，但 C++11 引入了语法使其成为右值引用类型。例如：

```
1 struct S {  
2     void f1(); // implicit *this parameter is an lvalue reference (see below)  
3     void f2() &&; // implicit *this parameter is an rvalue reference  
4     void f3() &; // implicit *this parameter is an lvalue reference  
5 };
```

从这个例子中可以看出，不仅可以使隐式参数成为右值引用（带 `&&` 后缀），还可以确认左值引用情况（带 `&` 后缀）。有趣的是，指定 `&` 后缀并不完全等同于引用它：旧的特殊情况允许右值绑定到对非 `const` 类型的左值引用，当该引用是传统的隐式 `*this` 参数时，但若显式请求左值引用处理，这种特殊情况（有些危险）不再适用。因此，根据上面 `S` 的定义，有如下的结果：

```
1 int main()  
2 {  
3     S().f1(); // OK: old rule allows rvalue S() to match implied  
4             // lvalue reference type S& of *this  
5     S().f2(); // OK: rvalue S() matches rvalue reference type  
6             // of *this  
7     S().f3(); // ERROR: rvalue S() cannot match explicit lvalue  
8             // reference type of *this  
9 }
```

C.2.2 改善完美匹配

对于 `X` 类型的参数，有四种常见的参数类型可以构成完美匹配：`X`、`X&`、`X const&` 和 `X&&(X const&&` 也是完美匹配，但很少使用）。但是，在两种引用上重载函数是很常见的。C++11 之前，这意味着：

```
1 void report(int&); // #1  
2 void report(int const&); // #2  
3  
4 int main()  
5 {  
6     for (int k = 0; k<10; ++k) {  
7         report(k); // calls #1  
8     }  
9     report(42); // calls #2  
10 }
```

没有其他的 `const` 的版本更适合用于左值，而只有带有 `const` 的版本才能匹配右值。

随着 C++11 中右值引用的引入，需要区分两个完全匹配的另一种常见情况如下所示：

```

1 struct Value {
2     ...
3 };
4 void pass(Value const&); // #1
5 void pass(Value&&); // #2
6
7 void g(X&& x)
8 {
9     pass(x); // calls #1 , because x is an lvalue
10    pass(X()); // calls #2 , because X() is an rvalue (in fact, prvalue)
11    pass(std::move(x)); // calls #2 , because std::move(x) is an rvalue (in fact, xvalue)
12 }

```

这次，采用右值引用的版本认为是右值的更好匹配，但不能匹配左值。

这也适用于成员函数调用的隐式参数：

```

1 class Wonder {
2     public:
3     void tick(); // #1
4     void tick() const; // #2
5     void tack() const; // #3
6 };
7
8 void run(Wonder& device)
9 {
10    device.tick(); // calls #1
11    device.tack(); // calls #3 , because there is no non-const version
12        // of Wonder::tack()
13 }

```

最后，下面代码对前面示例的修改说明，不管使用和不使用引用，两个完美匹配都会产生歧义：

```

1 void report(int); // #1
2 void report(int&); // #2
3 void report(int const&); // #3
4
5 int main()
6 {
7     for (int k = 0; k<10; ++k) {
8         report(k); // ambiguous: #1 and #2 match equally well
9     }
10    report(42); // ambiguous: #1 and #3 match equally well
11 }

```

C.3 重载的细节

上一节介绍了在 C++ 编程中遇到的大多数重载情况。但对于这些规则，还有更多的规则和例外——本书并不是转为 C++ 函数重载所著，因此不会一一介绍。但我们在里会讨论其中一些规则，

一方面是因为它们比其他规则应用得更多，另一方面是为了让大家了解其中的水有多深。

C.3.1 选择非模板或更特化的临时模板

当重载解析的所有方面都相同时，非模板函数优先于模板的实例（该实例是从泛型模板定义生成，还是作为显式特化提供，都无关紧要）。例如：

```
1 template<typename T> int f(T); // #1
2 void f(int); // #2
3
4 int main()
{
    return f(7); // ERROR: selects #2 , which doesn't return a value
5 }
```

这个例子还清楚地说明了重载解析通常不涉及所选函数的返回类型。

但当重载解析的其他方面略有不同时（具有不同的 `const` 和引用限定符），首先应用重载解析的一般规则。当成员函数定义为接受与复制或移动构造函数相同的参数时，这种方式就很容易导致不可预期的行为。参见 16.2.4 节。

如果要在两个模板之间进行选择，首选最特化的模板（前提是其中一个实际上比另一个更特化）。关于这个概念的详细解释，请参阅 16.2.2 节。这种区别的特殊情况发生在两个模板仅在添加末尾参数包时：没有包的模板更特化，因此若与调用相匹配将首先选择。4.1.2 节讨论了这种情况的一个例子。

C.3.2 转换序列

隐式转换通常可以是转换的序列。考虑下面的代码示例：

```
1 class Base {
2     public:
3         operator short() const;
4 };
5
6 class Derived : public Base {
7 };
8
9 void count(int);
10
11 void process(Derived const& object)
12 {
13     count(object); // matches with user-defined conversion
14 }
```

调用 `count(object)` 有效，是因为 `object` 可以隐式转换为 `int`。这种转换需要几步：

1. 从派生 `const` 对象到基本 `const` 对象的转换（这是 glvalue 转换，保留了对象的身份）
2. 将生成的 `Base const` 对象转换为 `short` 类型的用户定义转换
3. `short` 升为 `int`

这是最常见的转换序列: 标准转换(在本例中是派生到基类的转换), 然后是用户定义的转换, 然后是另一个标准转换。虽然在转换序列中最只能有一个用户定义的转换, 但也可能只有标准转换。

重载解析的一个重要原则是, 作为另一个转换序列的子序列的转换序列要优于后一个序列。若有其他候选函数

```
1 void count(short);
```

因为不需要转换序列中的第三步(类型提升), 所以调用 `count(object)` 会将其作为首选。

C.3.3 指针转换

指针和指向成员的指针进行各种特殊的标准转换, 包括

- 转换为 `bool` 类型
- 将指针类型转换为 `void*`
- 派生类的指针转换为基类指针
- 成员指针转换为派生类指针

尽管所有这些都可以“只与标准转换匹配”, 但排名并不相同。

首先, 到 `bool` 类型的转换(无论是从普通指针还是从指向成员的指针), 认为比其他类型的标准转换都要糟糕。例如:

```
1 void check(void*); // #1
2 void check(bool); // #2
3
4 void rearrange (Matrix* m)
5 {
6     check(m); // calls #1
7     ...
8 }
```

常规指针转换的类别中, `void*` 类型的转换认为比从派生类指针到基类指针的转换更糟糕。此外, 若存在到与继承相关不同的转换, 则首选到最派生类的转换。下面是另一个例子:

```
1 class Interface {
2     ...
3 };
4
5 class CommonProcesses : public Interface {
6     ...
7 };
8
9 class Machine : public CommonProcesses {
10    ...
11};
12
13 char* serialize(Interface*); // #1
14 char* serialize(CommonProcesses*); // #2
15
16 void dump (Machine* machine)
```

```
17 {
18     char* buffer = serialize(machine); // calls #2
19     ...
20 }
```

从 Machine* 到 CommonProcesses* 的转换优于到 Interface* 的转换，后者非常直观。

类似的规则也适用于成员指针：相关的指针成员类型的两种转换之间，优先选择继承图中“最接近的基”（即派生最少的基）。

C.3.4 初始化式列表

初始化列表参数（初始化列表参数用大括号传递）可以转换为几种不同类型的参数：initializer_lists、带有 initializer_list 构造函数的类类型、初始化列表元素可以作为构造函数（单独）参数的类类型，或者聚合类类型，其成员可以由初始化列表的元素初始化。下面的程序阐明了这些情况：

overload/initlist.cpp

```
1 #include <initializer_list>
2 #include <string>
3 #include <vector>
4 #include <complex>
5 #include <iostream>
6
7 void f(std::initializer_list<int>) {
8     std::cout << "#1\n";
9 }
10
11 void f(std::initializer_list<std::string>) {
12     std::cout << "#2\n";
13 }
14
15 void g(std::vector<int> const& vec) {
16     std::cout << "#3\n";
17 }
18
19 void h(std::complex<double> const& cmplx) {
20     std::cout << "#4\n";
21 }
22
23 struct Point {
24     int x, y;
25 };
26 void i(Point const& pt) {
27     std::cout << "#5\n";
28 }
29
30 int main()
```

```
31 {
32     f({1, 2, 3}); // prints #1
33     f({"hello", "initializer", "list"}); // prints #2
34     g({1, 1, 2, 3, 5}); // prints #3
35     h({1.5, 2.5}); // prints #4
36     i({1, 2}); // prints #5
37 }
```

在对 f() 的前两次调用中，初始化式列表参数转换为 std::initializer_list 值，这将初始化式列表中的每个元素转换为 std::initializer_list 的元素类型。第一个调用中，所有元素都已经是 int 类型，因此不需要进行额外的转换。第二次调用中，通过调用 string(char const*) 构造函数，初始化式列表中的每个字符串字面值都转换为 std::string。第三个调用 (g()) 使用 std::vector(std::initializer_list<int>) 构造函数执行用户定义的转换。下一个调用 std::complex(double, double) 构造函数，就好像编写了 std::complex<double>(1.5, 2.5)。最后一个调用执行聚合初始化，从初始化式列表中的元素初始化 Point 类实例的成员，而不调用 Point 的构造函数。

聚合初始化只适用于 C++ 中的聚合类型，即数组或简单的类 C，这些类没有用户提供的构造函数，没有 private 或 protected 的非静态数据成员，没有基类，也没有虚函数。C++14 前，也不能有默认的成员初始化式。C++17 后，可以使用公共基类。

对于初始化式列表，有几种重载情况。当将初始化式列表转换为 initializer_list 时，如上例的前两次调用，整个转换排名与初始化式列表中给定元素到 initializer_list 元素类型的最差转换相同（即 initializer_list<T> 中的 T）。这可能会导致一些意外出现，比如下面的例子：

overload/initlist.cpp

```
1 #include <initializer_list>
2 #include <iostream>
3
4 void ovl(std::initializer_list<char>) { // #1
5     std::cout << "#1\n";
6 }
7
8 void ovl(std::initializer_list<int>) { // #2
9     std::cout << "#2\n";
10 }
11
12 int main()
13 {
14     ovl({'h', 'e', 'l', 'l', 'o', '\0'}); // prints #1
15     ovl({'h', 'e', 'l', 'l', 'o', 0}); // prints #2
16 }
```

对 ovl() 的第一次调用中，初始化式列表的每个元素都是一个字符。对于第一个 ovl() 函数，这些元素根本不需要转换。对于第二个 ovl() 函数，这些元素需要提升为 int。因为完美匹配比类型升级要好，所以对 ovl() 的第一次调用的是 #1。

对 `ovl()` 的第二次调用中，前五个元素是 `char` 类型，而最后一个元素是 `int` 类型。对于第一个 `ovl()` 函数，`char` 元素完美匹配，但是 `int` 需要标准转换，因此整个转换为标准转换。对于第二个 `ovl()` 函数，`char` 元素需要升级为 `int`，而末尾的 `int` 元素完全匹配。第二个 `ovl()` 函数的整体转换排名为提升，这使得它比第一个 `ovl()` 函数更为合适（尽管只有单个元素的转换更好）。

当使用初始化式列表初始化类对象时，就像在我们的原始示例中调用 `g()` 和 `h()` 一样，重载解析分为两个阶段进行：

1. 第一阶段只考虑初始化列表构造函数，对于某些类型 `T`，其唯一非默认参数是 `std::initializer_list<T>` 类型的构造函数（删除顶层引用和 `const/volatile` 限定符之后）。
2. 若没有找到这样的可行构造函数，第二阶段将考虑其他构造函数。

该规则有一个例外：若初始化列表为空，并且类有一个默认构造函数，则跳过第一阶段，以便调用默认构造函数。

该规则的效果是，任何初始化式列表构造函数，比非初始化式列表构造函数更匹配：

overload/initlistctor.cpp

```
1 #include <initializer_list>
2 #include <string>
3 #include <iostream>
4
5 template<typename T>
6 struct Array {
7     Array(std::initializer_list<T>) {
8         std::cout << "#1\n";
9     }
10    Array(unsigned n, T const&) {
11        std::cout << "#2\n";
12    }
13 };
14
15 void arr1(Array<int>) {
16 }
17
18 void arr2(Array<std::string>) {
19 }
20
21 int main()
22 {
23     arr1({1, 2, 3, 4, 5}); // prints #1
24     arr1({1, 2}); // prints #1
25     arr1({10u, 5}); // prints #1
26     arr2({"hello", "initializer", "list"}); // prints #1
27     arr2({10, "hello"}); // prints #2
28 }
```

第二个构造函数接受一个 `unsigned` 和一个 `T const&` 参数，从初始化列表初始化 `Array<int>` 对象时不会调用，因为其初始化列表构造函数总是比非初始化列表构造函数更匹配。但对于

`Array<string>`, 当初始化列表构造函数不可用时, 就会调用非初始化列表构造函数, 就像对 `arr2()` 的第二次调用一样。

C.3.5 函子和代理函数

查找函数名以创建初始重载集之后, 可以各种方式调整该集合。当调用表达式引用类类型对象而不是函数时, 会出现一种有趣的情况, 重载集会增加。

第一个方式很简单: 将成员的函数操作符(函数调用操作符)添加到集合中。带有此类操作符的对象通常称为函子或函数对象(参见第 11.1 节)。

当类对象包含指向函数类型指针(或指向函数类型引用)的隐式转换操作符时, 重载集就会出现不那么明显的增加。

转换操作符也必须适用于以下情况, 例如: `const` 对象不考虑非 `const` 操作符。

这种情况下, 将向重载集添加一个虚拟(或代理)函数。除了具有与该转换函数的目标类型中, 参数类型对应的参数外, 这个代理函数候选函数认为具有转换函数指定的类型的参数。下面的例子清楚地说明了这一点:

```
1 using FuncType = void (double, int);
2
3 class IndirectFunctor {
4     public:
5     ...
6     void operator()(double, double) const;
7     operator FuncType*() const;
8 };
9
10 void activate(IndirectFunctor const& funcObj)
11 {
12     funcObj(3, 5); // ERROR: ambiguous
13 }
```

调用 `funcObj(3, 5)` 视为带有三个参数的调用:`funcObj`、`3` 和 `5`。候选函数包括成员 `operator()`(视为具有 `IndirectFunctor const&`、`double` 和 `double` 参数类型) 和参数类型为 `FuncType*`、`double` 和 `int` 的代理函数。代理函数与隐含参数的匹配较差(因为它需要用户定义的转换), 但与最后一个参数的匹配较好; 因此, 不能使用这俩候选。因此, 该调用具有歧义。

代理函数是 C++ 中最晦涩的角落, 在实践中很少出现(幸运的是)。

C.3.6 其他情况的重载

已经讨论了重载在确定调用表达式中, 应该调用哪个函数的上下文中。但在其他一些情况下, 必须做出类似的选择。

第一个上下文发生在需要函数地址的时候。考虑下面的例子:

```
1 int numElems(Matrix const&); // #1
2 int numElems(Vector const&); // #2
```

```
3 ...
4 int (*funcPtr) (Vector const&) = numElems; // selects #2
```

这里，`numElems` 指的是一个重载集，但我们希望知道其中一个函数的地址。然后重载解析尝试将所需的函数类型(本例中是 `funcPtr` 的类型)与可用的候选函数匹配。

另一个需要重载解析的上下文是初始化，但这是一个和微妙的主题，超出了附录所能涵盖的范围。一个简单的例子说明了重载解析的特殊情况：

```
1 #include <string>
2 class BigNum {
3     public:
4     BigNum(long n); // #1
5     BigNum(double n); // #2
6     BigNum(std:::string const&); // #3
7     ...
8     operator double() ; // #4
9     operator long() ; // #5
10    ...
11 };
12
13 void initDemo()
14 {
15     BigNum bn1(100103); // selects #1
16     BigNum bn2("7057103224.095764"); // selects #3
17     int in = bn1; // selects #5
18 }
```

本例中，需要重载解析来选择适当的构造函数或转换操作符。具体来说，`bn1` 的初始化调用第一个构造函数，`bn2` 的初始化调用第三个构造函数，`in()` 的初始化调用 `operator long()`。大多数情况下，重载规则会产生直观的结果。但这些规则的细节相当复杂，一些(少量)应用程序会依赖于 C++ 语言这一领域中的技术。

附录 D：标准类型的使用

C++ 标准库主要由模板组成，其中许多模板依赖于本书中介绍和讨论的技术。因为标准库定义了几个模板来用通用代码实现库，所以一些技术“标准化”了。这些类型实用工具（类型特征和其他帮助工具）在本章中列出并解释。

一些类型特征需要编译器的支持，而另一些类型特征可以使用现有语言特性进行实现（在第 19 章讨论了其中一些）。

D.1 使用类型特征

使用类型特征时，必须包含头文件 `<type_traits>`：

```
1 #include <type_traits>
```

然后，该用法取决于特征是否产生类型或值：

- 对于产生类型的特征，使用如下方式访问该类型：

```
1 typename std::trait<...>::type
2 std::trait_t<...> // since C++14
```

- 对于产生值的特征，可以通过如下方式访问该值：

```
1 std::trait<...>::value
2 std::trait<...>() // implicit conversion to its type
3 std::trait_v<...> // since C++17
```

例如：

utils/traits1.cpp

```
1 #include <type_traits>
2 #include <iostream>
3
4 int main()
5 {
6     int i = 42;
7     std::add_const<int>::type c = i; // c is int const
8     std::add_const_t<int> c14 = i; // since C++14
9     static_assert(std::is_const<decltype(c)>::value, "c should be const");
10
11    std::cout << std::boolalpha;
12    std::cout << std::is_same<decltype(c), int const>::value // true
13        << '\n';
14    std::cout << std::is_same_v<decltype(c), int const> // since C++17
15        << '\n';
16    if (std::is_same<decltype(c), int const>{}) { // implicit conversion to bool
17        std::cout << "same\n";
18    }
```

参见第 2.8 节，了解 traits 的`_t`版本的定义方式。关于特征的`_v`版本的定义方式，请参见 5.6 节。

D.1.1 std::integral_constant 和 std::bool_constant

所有产生值的标准类型特征都派生自辅助类模板实例 `std::integral_constant`:

```

1 namespace std {
2     template<typename T, T val>
3     struct integral_constant {
4         static constexpr T value = val; // value of the trait
5         using value_type = T; // type of the value
6         using type = integral_constant<T, val>;
7         constexpr operator value_type() const noexcept {
8             return value;
9         }
10        constexpr value_type operator() () const noexcept { // since C++14
11            return value;
12        }
13    };
14 }
```

就是说:

- 可以使用 `value_type` 成员来查询结果的类型。由于产生值的许多特征是谓词，`value_type` 通常 是 `bool`。
- 特征类型的对象具有隐式类型转换，转换为类型特征产生值的类型。
- C++14(以及以后的版本) 中，类型特征的对象也可为函数对象(函子)，其中“函数调用”会产生值。
- 类型成员只生成底层 `integral_constant` 实例。

若特征产生布尔值，也可以这样

```

1 namespace std {
2     template<bool B>
3     using bool_constant = integral_constant<bool, B>; // since C++17
4     using true_type = bool_constant<true>;
5     using false_type = bool_constant<false>;
6 }
```

C++17 前，该标准不包括别名模板 `bool_constant<>`。`std::true_type` 和 `std::false_type` 在 C++11 和 C++14 中确实存在，但分别指定为 `integral_constant<bool,true>` 和 `integral_constant<bool,false>`。

以便这些布尔特征在特定属性适用时继承 `std::true_type`，否则继承 `std::false_type`，所以它们对应的值成员等于 `true` 和 `false`。为结果值设置不同的类型 `true` 和 `false`，可以基于类型特征的结果进行标签调度(参见第 19.3.3 节和第 20.2 节)。

例如：

utils/traits2.cpp

```
1 #include <type_traits>
2 #include <iostream>
3
4 int main()
5 {
6     using namespace std;
7     cout << boolalpha;
8
9     using MyType = int;
10    cout << is_const<MyType>::value << '\n' ; // prints false
11
12    using VT = is_const<MyType>::value_type; // bool
13    using T = is_const<MyType>::type; // integral_constant<bool, false>
14    cout << is_same<VT,bool>::value << '\n' ; // prints true
15    cout << is_same<T, integral_constant<bool, false>>::value
16        << '\n' ; // prints true
17    cout << is_same<T, bool_constant<false>>::value
18        << '\n' ; // prints true (not valid
19        // prior to C++17)
20
21    auto ic = is_const<MyType>(); // object of trait type
22    cout << is_same<decltype(ic), is_const<int>>::value << '\n' ; // true
23    cout << ic() << '\n' ; // function call (prints false)
24
25    static constexpr auto mytypeIsConst = is_const<MyType>{};
26    if constexpr(mytypeIsConst) { // compile-time check since C++17 => false
27        ... // discarded statement
28    }
29    static_assert(!std::is_const<MyType>{}, "MyType should not be const");
30 }
```

各种元编程上下文中，具有不同的非布尔 `integral_constant` 特化类型。参见第 24.3 节对类似类型 `CTValue` 的讨论，以及第 25.6 节对元组元素访问的使用。

D.1.2 使用特征

使用性格特征时需要注意以下几点：

- 类型特征应用于类型，但 `decltype` 允许测试表达式、变量和函数的属性，`decltype` 只在变量或函数的命名没有附加括号的情况下才会产生类型；对于任何其他表达式，生成的类型也反映表达式的类型类别。例如：

```
1 void foo (std::string&& s)
2 {
3     // check the type of s:
4     std::is_lvalue_reference<decltype(s)>::value // false
```

```
5 std::is_rvalue_reference<decltype(s)>::value // true, as declared
6 // check the value category of s used as expression:
7 std::is_lvalue_reference<decltype((s))>::value // true, s used as lvalue
8 std::is_rvalue_reference<decltype((s))>::value // false
9 }
```

详见第 15.10.2 节。

- 对于菜鸟开发者来说，有些特征可能具有反直觉的行为。参见第 11.2.1 节中的示例。
- 有些特征是有要求或前提条件，违反这些前提条件会导致未定义行为。

C++ 标准委员会考虑了 C++17 的提议，要求违反类型特征的前提条件会导致编译时错误。然而，因为一些类型特征目前的需求比严格的必要条件更强（比如总是需要完整的类型），所以这个改变推迟了。

参见第 11.2.1 节中的一些示例。

- 许多特征需要完整的类型（参见第 10.3.1 节）。为了能够在不完整类型中使用，有时可以引入模板来延迟计算（详见第 11.5 节）。
- 有时逻辑操作符 `&&`、`||` 和 `!` 不能用于定义基于其他类型特征的新类型特征。此外，处理可能失败的特征可能会有问题，至少会有一些缺陷。出于这个原因，我们提供了一些特殊的特性，能够在逻辑上组合布尔特性。参见第 D.6 部分了解详细信息。
- 尽管标准别名模板（以 `_t` 或 `_v` 结尾），但也有缺点，并在某些元编程上下文中不可用。详见第 19.7.3 节。

D.2 主要类型和复合类型

我们从测试主要类型和复合类型类别的标准特征开始（参见图 D.1）。

感谢 Howard Hinnant 提供了这种类型层次结构 <http://howardhinnant.github.io/TypeHierarchy.pdf>

通常，每个类型都只属于一个主要类型类别（图 D.1 中的白色元素）。复合类型类别需要将主要类型合并到更高级的概念中。

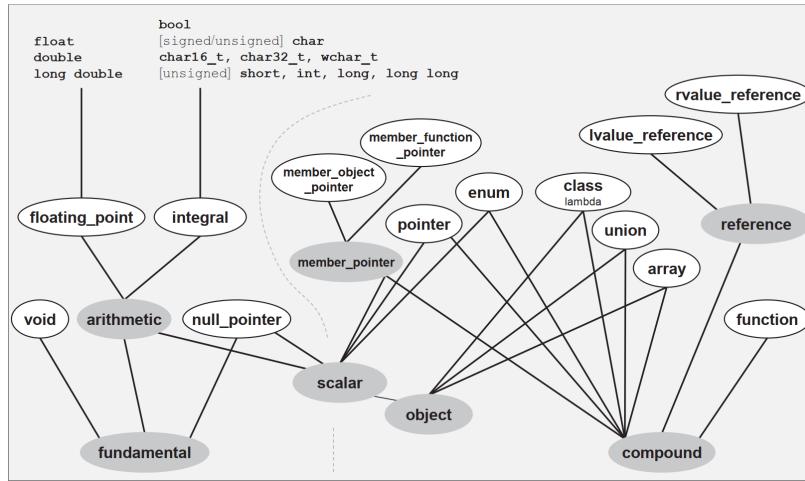


图 D.1. 主要和复合类型类别

D.2.1 测试主要类型的类别

本节测试给定类型的主要类型类别。对于给定类型，只有一个主类型类别具有计算结果为 true 的静态值成员。

C++14 前，唯一的例外是 `std::nullptr_t`，对于所有的主类型类别实用程序都产生了 false，因为 `is_null_pointer` 不是 C++11 的一部分。

结果与类型是否用 `const` 和/或 `volatile(cv)` 限定无关。

注意，对于类型 `std::size_t` 和 `std::ptrdiff_t`，当 `_integral` 时结果为真。对于类型 `std::max_align_t`，这些主要类型类别中哪一个是 true 是实现的细节（可能是整型、浮点型或类）。语言指定 Lambda 表达式是类（参见第 15.10.6 节），将 `is_class` 应用于该类型可得到 true。

特征	作用
<code>is_void<T></code>	<code>void</code> 类型
<code>is_integral<T></code>	整数类型（包括 <code>bool</code> , <code>char</code> , <code>char16_t</code> , <code>char32_t</code> , <code>wchar_t</code> ）
<code>is_floating_point<T></code>	浮点类型（ <code>float</code> , <code>double</code> , <code>long double</code> ）
<code>is_array<T></code>	普通的数组类型（非 <code>std::array</code> 类型）
<code>is_pointer<T></code>	指针类型（包括函数指针，但不包括指向非静态成员的指针）
<code>is_null_pointer<T></code>	<code>nullptr</code> 的类型（C++14）
<code>is_member_object_pointer<T></code>	指向非静态数据成员的指针
<code>is_member_function_pointer<T></code>	指向非静态成员函数的指针
<code>is_lvalue_reference<T></code>	左值引用
<code>is_rvalue_reference<T></code>	右值引用
<code>is_enum<T></code>	枚举类型
<code>is_class<T></code>	<code>class/struct</code> 或 <code>Lambda</code> 类型，但非联合类型
<code>is_union<T></code>	联合类型
<code>is_function<T></code>	函数类型

表 D.1. 检查主要类型类别的特征

Lambda 表达式的类型是一个类(参见第 15.10.6 节), 将 `is_class` 应用于该类型将得到 `true`。

`std::is_void<T>::value`

- 如果类型 T 是 (cv 限定的)void, 则生成 true。
- 例如:

```

1 is_void_v<void> // yields true
2 is_void_v<void const> // yields true
3 is_void_v<int> // yields false
4 void f();
5 is_void_v<decltype(f)> // yields false (f has function type)
6 is_void_v<decltype(f())> // yields true (return type of f() is void)

```

`std::is_integral<T>::value`

- 如果类型 T 是以下 (cv 限定的) 类型之一, 则生成 true:

- `bool`
- 字符类型 (`char`, `signed char`, `unsigned char`, `char16_t`, `char32_t` 或 `wchar_t`)
- 一个整数类型 (`short`、`int`、`long` 或 `long long` 的有符号或无符号; 这包括 `std::size_t` 和 `std::ptrdiff_t`)

`std::is_floating_point<T>::value`

- 若类型 T 是 (cv 限定的)`float`、`double` 或 `long double` 则为 true

`std::is_array<T>::value`

- 若类型 T 是一个 (cv 限定的) 数组类型, 则生成 true。
- 由语言规则声明为数组 (带或不带长度) 的参数可以是指针类型。
- `std::array` 不是数组类型, 而是类。
- 例如:

```

1 is_array_v<int[]> // yields true
2 is_array_v<int[5]> // yields true
3 is_array_v<int*> // yields false
4
5 void foo(int a[], int b[5], int* c)
6 {
7     is_array_v<decltype(a)> // yields false (a has type int*)
8     is_array_v<decltype(b)> // yields false (b has type int*)
9     is_array_v<decltype(c)> // yields false (c has type int*)
10 }

```

- 参见第 19.8.2 节了解实现细节。

`std::is_pointer<T>::value`

- 若类型 T 是一个 (cv 限定的) 指针, 则生成 true。包括:
 - 指向静态/全局 (成员) 函数的指针

- 参数声明为数组(带/不带长度)或函数类型

不包括:

- 指向成员类型的指针(例如, `&X::m` 的类型, 其中 X 是类类型, m 是非静态成员函数或非静态数据成员)
- `nullptr` 的类型是 `std::nullptr_t`

- 例如:

```

1 is_pointer_v<int> // yields false
2 is_pointer_v<int*> // yields true
3 is_pointer_v<int* const> // yields true
4 is_pointer_v<int*&> // yields false
5 is_pointer_v<decltype(nullptr)> // yields false
6
7 int* foo(int a[5], void(f)())
8 {
9     is_pointer_v<decltype(a)> // yields true (a has type int*)
10    is_pointer_v<decltype(f)> // yields true (f has type void(*)())
11    is_pointer_v<decltype(foo)> // yields false
12    is_pointer_v<decltype(&foo)> // yields true
13    is_pointer_v<decltype(foo(a,f))> // yields true (for return type int*)
14 }
```

- 参见第 19.8.2 节了解实现细节。

`std::is_null_pointer<T>::value`

- 若类型 T 是 (cv 限定的)`std::nullptr_t`, 则是 `nullptr` 的类型, 则生成 true。
- 例如:

```

1 is_null_pointer_v<decltype(nullptr)> // yields true
2 void* p = nullptr;
3
4 is_null_pointer_v<decltype(p)> // yields false (p has not type std::nullptr_t)
```

- 从 C++14 开始支持。

`std::is_member_object_pointer<T>::value`

`std::is_member_function_pointer<T>::value`

- 若类型 T 是一个 (cv 限定的) 指向成员的指针类型 (`int X::*` 或对于某些类 X 的 `int (X::*)()`), 则返回 true。

`std::is_lvalue_reference<T>::value`

`std::is_rvalue_reference<T>::value`

- 若类型 T 分别是 (cv 限定的) 左值或右值引用类型, 则生成 true。
- 例如:

```

1 is_lvalue_reference_v<int> // yields false
2 is_lvalue_reference_v<int&> // yields true
```

```
3 is_lvalue_reference_v<int&&> // yields false
4 is_lvalue_reference_v<void> // yields false
5 is_rvalue_reference_v<int> // yields false
6 is_rvalue_reference_v<int&> // yields false
7 is_rvalue_reference_v<int&&> // yields true
8 is_rvalue_reference_v<void> // yields false
```

- 参见第 19.8.2 节了解实现细节。

std::is_enum<T>::value

- 若类型 T 是 (cv 限定的) 枚举类型，则生成 true。这既适用于作用域枚举类型，也适用于非作用域枚举类型。
- 参见第 19.8.5 节了解实现细节。

std::is_class<T>::value

- 若类型 T 是用 class 或 struct 声明的 (cv 限定的) 类，包括从实例化类模板中生成的这种类型，则生成真。语言保证 Lambda 表达式的类型是类 (参见 15.10.6 节)。
- 对联合、作用域枚举类型 (尽管用 enum 类声明)、std::nullptr_t 和任何其他类型会产生 false。
- 例如：

```
1 is_class_v<int> // yields false
2 is_class_v<std::string> // yields true
3 is_class_v<std::string const> // yields true
4 is_class_v<std::string&> // yields false
5 auto l1 = []{};
6 is_class_v<decltype(l1)> // yields true (a lambda is a class object)
```

- 参见第 19.8.4 节了解实现细节。

std::is_union<T>::value

- 若类型 T 是一个 (cv 限定的) 联合，包括一个由联合模板的类模板生成的联合，则生成 true。

std::is_function<T>::value

- 若类型 T 是一个 (cv 限定的) 函数类型，则生成 true。对函数指针类型、Lambda 表达式的类型和其他类型产生 false。
- 根据语言规则声明为函数类型的参数，实际上可以是指针类型。
- 例如：

```
1 void foo(void(f)())
2 {
3     is_function_v<decltype(f)> // yields false (f has type void(*)())
4     is_function_v<decltype(foo)> // yields true
5     is_function_v<decltype(&foo)> // yields false
6     is_function_v<decltype(foo(f))> // yields false (for return type)
7 }
```

- 参见第 19.8.3 节了解实现细节。

D.2.2 复合类型类别的测试

以下类型可以确定类型是否属于更一般类型类别，该类型类别是一些主要类型类别的联合。复合类型类别不构成严格的划分：一个类型可以属于多个复合类型类别（例如，指针类型既是标量类型又是复合类型）。同样，`cv` 限定符 (`const` 和 `volatile`) 在对类型进行分类时无关。

`std::is_reference<T>::value`

- 若类型 `T` 是引用类型，则生成 `true`。
- 等价于：

```
1 is_lvalue_reference_v<T> || is_rvalue_reference_v<T>
```

- 参见第 19.8.2 节了解实现细节。

特性	作用
<code>is_reference<T></code>	左值或右值引用
<code>is_member_pointer<T></code>	指向非静态成员的指针
<code>is_arithmetic<T></code>	整型（包括 <code>bool</code> 和字符型）或浮点型
<code>is_fundamental<T></code>	<code>void</code> , 整型（包括 <code>bool</code> 和字符型），浮点，或 <code>std::nullptr_t</code>
<code>is_scalar<T></code>	整型（包括 <code>bool</code> 和字符型）、浮点型、枚举型、指针型、成员指针型和 <code>std::nullptr_t</code> 型
<code>is_object<T></code>	除 <code>void</code> 、函数或引用外的类型
<code>is_compound<T></code>	与 <code>is_fundamental</code> 相反 <code><T></code> : 数组、枚举、联合、类、函数、引用、指针或成员指针

表 D.2. 检查复合类型类别的特征

`std::is_member_pointer <T>::value`

- 若类型 `T` 是任何指向成员的类型，则生成 `true`。
- 等价于：

```
1 is_member_object_pointer_v<T> || is_member_function_pointer_v<T>
```

`std::is_arithmetic<T>::value`

- 若类型 `T` 是算术类型 (`bool`、字符类型、整数类型或浮点类型)，则生成 `true`。
- 等价于：

```
1 is_integral_v<T> || is_floating_point_v<T>
```

`std::is_fundamental<T>::value`

- 若类型 `T` 是基本类型（算术类型或 `void` 或 `std::nullptr_t`），则生成 `true`。
- 等价于：

```
1 is_arithmetic_v<T> || is_void_v<T> || is_null_pointer_v<T>
```

- 等价于:

```
1 !is_compound_v<T>
```

- 参见第 448 页 19.8.1 节中的 IsFundamental 的实现细节。

std::is_scalar<T>::value

- 若类型 T 是“标量”类型，则生成 true。
- 等价于:

```
1 is_arithmetic_v<T> || is_enum_v<T> || is_pointer_v<T>
2 || is_member_pointer_v<T> || is_null_pointer_v<T>
```

std::is_object<T>::value

- 若类型 T 描述了对象的类型，则生成 true。
- 等价于:

```
1 is_scalar_v<T> || is_array_v<T> || is_class_v<T> || is_union_v<T>
```

- 等价于:

```
1 ! (is_function_v<T> || is_reference_v<T> || is_void_v<T>)
```

std::is_compound<T>::value

- 若类型 T 是由其他类型组合而成的类型，则生成 true。
- 等价于:

```
1 !is_fundamental_v<T>
```

- 等价于:

```
1 is_enum_v<T> || is_array_v<T> || is_class_v<T> || is_union_v<T>
2 || is_reference_v<T> || is_pointer_v<T> || is_member_pointer_v<T>
3 || is_function_v<T>
```

D.3 类型属性和操作

下一组特征测试单类型的其他属性，以及可能适用于某些操作（例如值交换）。

D.3.1 其他类型的属性

std::is_signed<T>::value

- 若 T 是有符号算术类型（即包含负数表示的算术类型；这包括 (signed) int, float 等类型）。
- 对于 bool 类型，会产生 false。
- 对于 char 类型，其结果由具体实现定义。
- 对于所有非算术类型（包括枚举类型），is_signed 会生成 false。

std::is_unsigned<T>::value

- 若 T 是无符号算术类型 (即不包含负数表示的算术类型; 这包括 unsigned int 和 bool 类型)。
- 对于 char 类型，其结果由具体实现定义。
- 对于所有非算术类型 (包括枚举类型)，`is_unsigned` 会生成 false。

`std::is_const<T>::value`

- 若类型是 const 的，则生成 true。
- const 指针具有 const 类型，而非 const 指针或指向 const 类型的引用则不具有 const 类型。例如：

```
1 is_const<int* const>::value // true
2 is_const<int const*>::value // false
3 is_const<int const&>::value // false
```

- 若元素类型是 const 的，则该将数组定义为 const 的。

C++11 发布后，核心问题 1059 的解决方案澄清了这一点。

例如：

```
1 is_const<int[3]>::value // false
2 is_const<int const[3]>::value // true
3 is_const<int[]>::value // false
4 is_const<int const[]>::value // true
```

特征	作用
is_signed<T>	有符号算术类型
is_unsigned<T>	无符号算术类型
is_const<T>	常量(不可变)限定
is_volatile<T>	可变限定
is_aggregate<T>	聚合类型(C++17)
is_trivial<T>	标量、普通类或这些类型的数组
is_trivially_copyable<T>	标量、可简单复制的类或这些类型的数组
is_standard_layout<T>	标量、标准布局类或这些类型的数组
is_pod<T>	普通旧数据类型(memcpy()用于复制对象的类型)
is_literal_type<T>	类型的标量、引用、类或数组(C++17起弃用)
is_empty<T>	没有成员、虚成员函数或虚基类的类
is_polymorphic<T>	具有(派生)虚成员函数的类
is_abstract<T>	抽象类(至少一个纯虚函数)
is_final<T>	最后一个类(不允许派生的类, C++14)
has_virtual_destructor<T>	具有虚析构函数的类
has_unique_object_representations<T>	两个具有相同值的对象, 在内存中具有相同的表示形式(C++17)
alignment_of<T>	相当于alignof(T)
rank<T>	数组类型的维数(或0)
extent<T,I=0>	维度I(或0)的范围
underlying_type<T>	枚举类型的基础类型

表 D.3. 特征测试简单类型的属性

is_invocable<T,Args...>	用作可调用的Args...(C++17)
is_nothrow_invocable<T,Args...>	用作可调用的Args...不抛出异常(C++17)
is_invocable_r<RT,T,Args...>	用作可调用的Args...返回RT(C++17)
is_nothrow_invocable_r<RT,T,Args...>	用作可调用的Args...返回RT而不抛出异常(C++17)
invoke_result<T,Args...>	结果类型用作可调用的Args...(C++17)
result_of<F,ArgTypes>	使用参数类型ArgTypes调用F的结果类型(C++17起弃用)

表 D.3. 特征测试简单类型的属性(续)

std::is_volatile<T>::value

- 若类型是变量限定，则生成true。
- 可变指针具有可变限定的类型，而非可变指针或对可变类型的引用不是可变限定的。例如：

```

1 is_volatile<int* volatile>::value // true
2 is_volatile<int volatile*>::value // false
3 is_volatile<int volatile&>::value // false

```

- 若元素类型是可变限定的，语言将数组定义为可变限定的。

C++11 发布后，核心问题 1059 的解决方案澄清了这一点。

例如：

```
1 is_volatile<int[3]>::value // false
2 is_volatile<int volatile[3]>::value // true
3 is_volatile<int[]>::value // false
4 is_volatile<int volatile[]>::value // true
```

`std::is_aggregate<T>::value`

- 若 T 是聚合类型(数组或没有用户定义的类型、显式的或继承的构造函数的 class/struct/union, 没有私有的或受保护的非静态数据成员, 没有虚函数, 没有虚拟的、私有的或受保护的基类), 则生成 true。

聚合的基类和/或数据成员不必聚合。C++14 前，聚合类类型不能有默认的成员初始化式。C++17 前，聚合不能有公共基类。

- 帮助确定是否需要列表初始化。例如：

```
1 template<typename Coll, typename... T>
2 void insert(Coll& coll, T&... val)
3 {
4     if constexpr(!std::is_aggregate_v<typename Coll::value_type>) {
5         coll.emplace_back(std::forward<T>(val)...); // invalid for aggregates
6     }
7     else {
8         coll.emplace_back(typename Coll::value_type{std::forward<T>(val)...});
9     }
10 }
```

- 要求给定的类型要么完整(见第 10.3.1 节)，要么是 (cv 限定的)void。
- C++17 后可用。

`std::is_trivial<T>::value`

- 若类型是一个“简单”类型，则返回 true：

- 标量类型(整型、浮点型、枚举型、指针型；参见 `is_scalar()`)
 - 简单类类型(没有虚函数、虚基类、(间接) 用户定义的默认构造函数、复制/移动构造函数、复制/移动赋值操作符或析构函数、非静态数据成员没有初始化式、可变成员和非普通成员的类)
 - 简单类型的数组
 - 以及这些类型的 cv 限定类型
- 若 `is_trivially_copyable_v<T>` 会产生 true，并且存在一个简单的默认构造函数。

- 要求给定的类型要么完整(见第 10.3.1 节), 要么是 (cv 限定的)void。

std::is_trivially_copyable<T>::value

- 若类型是“简单可复制”类型, 则返回 true:

- 标量类型(整型、浮点型、枚举型、指针型; 参见 `is_scalar<>`)
 - 简单类类型(没有虚函数、虚基类、(间接) 用户定义的默认构造函数、复制/移动构造函数、复制/移动赋值操作符或析构函数、非静态数据成员没有初始化式、可变成员和非普通成员的类)
 - 这种类型的数组
 - 以及这些类型的 cv 限定类型
- 产生与 `is_trivial_v<T>` 相同的结果, 但可以为没有普通默认构造函数的类类型产生 true。
 - 与 `is_standard_layout<>` 相比, 不允许可变成员, 允许引用。成员可以有不同的访问权限, 成员可以分布在不同的(基)类中。
 - 要求给定的类型要么完整(见第 10.3.1 节), 要么是 (cv 限定的)void。

std::is_standard_layout<T>::value

- 若该类型具有标准布局, 则生成 true。这样可以更容易地与其他语言交换该类型的值。

- 标量类型(整型、浮点型、枚举型、指针型; 参见 `is_scalar<>`)
 - 标准布局类类型(没有虚函数、没有虚基类、没有非静态引用成员, 所有非静态成员都在具有相同访问权限的相同(基)类中, 所有成员也是标准布局类型)
 - 这种类型的数组
 - 以及这些类型的 cv 限定类型
- 与 `is_trivial<>` 相比, 允许可变成员, 不允许引用, 成员可能有相同的访问权限, 成员可能不分布在不同的(基)类中。
 - 要求给定的类型(对于数组, 基本类型)要么完整(见 10.3.1 节), 要么是 (cv 限定的)void。

std::is_pod<T>::value

- 若 T 是普通的旧数据类型(POD), 则生成 true。
- 这种类型的对象可以通过复制底层存储来复制(例如, 使用 `memcpy()`)。
- 等价于:

```
| is_trivial_t<T> && is_standard_layout_v<T>
```

- 生成 false 的情况:

- 没有简单的默认构造函数、复制/移动构造函数、复制/移动赋值或析构函数的类
- 具有虚成员或虚基类的类
- 具有可变成员或引用成员的类
- 不同(基)类中具有成员或具有不同访问权限的类
- Lambda 表达式的类型(称为闭包类型)

- 函数
- void
- 由这些类型组成的类型

- 要求给定的类型要么完整(见 10.3.1 节), 要么是 (cv 限定的)void。

std::is_literal_type<T>::value

- 若给定的类型是 constexpr 函数的有效返回类型, 则生成 true(需要注意的是, 该类型不包含非简单销毁的类型)。
- 若 T 是字面类型, 则返回 true:
 - 标量类型(整型、浮点型、枚举型、指针型; 参见 `is_scalar()`)
 - 引用
 - 每个(基)类中至少有一个 constexpr 构造函数, 该构造函数不是复制/移动构造函数, 在(基)类或成员中都没有用户定义或虚拟析构函数, 并且对非静态数据成员的每次初始化都是常量表达式
 - 这种类型的数组
- 要求给定的类型要么完整(见第 10.3.1 节), 要么是 (cv 限定的)void。
- 这个特征自 C++17 以来已弃用, 因为“它太弱了, 不能在泛型代码中有意义地使用。我们需要的是能够了解特定构造是否会产生恒定的初始化。”

std::is_empty<T>::value

- 若 T 是类类型而不是联合类型(其对象不包含数据), 则生成 true。
- 若 T 定义为带有的类或结构, 则生成 true
 - 除了长度为 0 的位域外, 没有非静态数据成员
 - 没有虚成员函数
 - 没有虚基类
 - 没有非空基类
- 若是类/结构体, 则要求给定的类型完整(见 10.3.1 节)(不完整的联合也可以)。

std::is_polymorphic<T>::value

- 若是类/结构体, 则要求给定的类型完整(见 10.3.1 节)(不完整的联合也可以)。
- 要求给定的类型要么完整(见第 10.3.1 节), 要么既不是类也不是结构体。

std::is_abstract<T>::value

- 要求给定的类型要么完整(见第 10.3.1 节), 要么既不是类, 也不是结构体。
- 若是类/结构体, 则要求给定的类型完整(见 10.3.1 节)(不完整的联合也可以)。

std::is_final<T>::value

- 若 T 是 final 类类型(因为声明为 final, 该类或联合类不能作为基类), 则生成 true。
- 对于所有非类/联合类型, 如 int, 会返回 false(这与类似于可派生不同)。

- 要求给定的类型 T 要么完整 (见第 10.3.1 节), 要么既不是类/结构体, 也不是联合。
- C++14 后可用。

std::has_virtual_destructor<T>::value

- 若类型 T 具有虚析构函数, 则生成 true。
- 若是类/结构体, 则要求给定的类型完整 (见第 10.3.1 节)(不完整的联合也可以)。

std::has_unique_object_representations<T>::value

- 若任意两个 T 类型的对象在内存中具有相同的对象表示, 则生成 true。也就是, 始终使用相同的字节值序列表示两个相同的值。
- 具有此属性的对象, 可以通过哈希关联的字节序列产生可靠的哈希值 (没有参与对象值的某些位可能在不同情况下不同, 这没有风险)。
- 要求给定的类型是简单可复制的 (见第 D.3.1 节), 并且要么是完整的 (见第 10.3.1 节), 要么是 (cv 限定的)void 或未知边界的数组。
- C++17 后可用。

std::alignment_of<T>::value

- 生成类型为 T 的对象的对齐值 std::size_t(对于数组, 为元素类型; 对于引用, 为引用类型)。
- 等价于: alignof(T)
- 这个特性是在 C++11 中 alignof(…) 之前引入。因为特征可以作为类类型传递, 对元编程很有用。
- 要求 alignof(T) 是一个有效的表达式。
- 使用 aligned_union<> 来获得多种类型的公共对齐 (参见第 D.5 节)。

std::rank<T>::value

- 生成类型为 T 的数组的维数 std::size_t。
- 其他类型的结果为 0。
- 指针没有相关的维度, 数组类型中未指定的边界指定维度。(用数组类型声明的函数参数没有实际的数组类型, 而且 std::array 也不是数组类型。参见 D.2.1 节。)

例如:

```

1 int a2[5][7];
2 rank_v<decltype(a2)>; // yields 2
3 rank_v<int*>; // yields 0 (no array)
4 extern int p1[];
5 rank_v<decltype(p1)>; // yields 1

```

std::extent<T>::value

std::extent<T, IDX>::value

- 生成类型为 T 的数组的第一个或第 idx 维的大小为 std::size_t。
- 若 T 不是数组, 维度不存在, 或者维度未知, 则生成 0。
- 参见第 19.8.2 节了解实现细节。

```

1 int a2[5][7];
2 extent_v<decltype(a2)>; // yields 5
3 extent_v<decltype(a2),0>; // yields 5
4 extent_v<decltype(a2),1>; // yields 7
5 extent_v<decltype(a2),2>; // yields 0
6 extent_v<int*>; // yields 0
7 extern int p1[];
8 extent_v<decltype(p1)>; // yields 0

```

std::underlying_type<T>::type

- 生成枚举类型 T 的基本类型。
- 要求给定的类型是完整的(见 10.3.1 节)枚举类型。对于所有其他类型，具有未定义行为。

std::is_invocable<T, Args...>::value

std::is_nothrow_invocable<T, Args...>::value

- 若 T 可用作 Args... 的可调用对象，则生成 true(保证不抛出异常)。
- 可以使用这些特征来测试是否可以调用或 std::invoke() 给定 Args... 的可调用 T(关于可调用对象和 std::invoke() 的详细信息请参见第 11.1 节。)
- 要求所有给定的类型都是完整的(见第 10.3.1 节)或 (cv 限定的)void 或一个边界未知的数组。
- 例如：

```

1 struct C {
2     bool operator() (int) const {
3         return true;
4     }
5 };
6 std::is_invocable<C>::value // false
7 std::is_invocable<C,int>::value // true
8 std::is_invocable<int*>::value // false
9 std::is_invocable<int(*)()>::value // true

```

- C++17 之后可用。

C++17 标准化过程的后期，`is_invocable` 改名为 `is_callable`。

std::is_invocable_r<RET_T, T, Args...>::value

std::is_nothrow_invocable_r<RET_T, T, Args...>::value

- 若可以使用 T 作为 Args... 的可调用对象，结果为 true(保证不抛出异常)，返回一个可转换为 RET_T 类型的值。
- 可以使用这些特征来测试是否可以调用或 std::invoke() 传递给 Args... 的可调用 T，并使用返回值 RET_T(关于可调用对象和 std::invoke() 的详细信息请参见第 11.1 节)。
- 要求传入的所有类型都完整(见第 10.3.1 节)或 (cv 限定的)void 或未知边界的数组。
- 例如：

```

1 struct C {
2     bool operator() (int) const {
3         return true;
4     }
5 };
6 std::is_invocable_r<bool,C,int>::value // true
7 std::is_invocable_r<int,C,long>::value // true
8 std::is_invocable_r<void,C,int>::value // true
9 std::is_invocable_r<char*,C,int>::value // false
10 std::is_invocable_r<long,int(*)(int)>::value // false
11 std::is_invocable_r<long,int(*)(int),int>::value // true
12 std::is_invocable_r<long,int(*)(int),double>::value // true

```

- C++17 后可用。

`std::invoke_result<T, Args...>::value`

`std::result_of<T, Args...>::value`

- 生成为 Args... 调用可调用 T 的返回类型
- 语法略有不同:

- 对于 `invoke_result<>`, 必须同时传递可调用对象的类型和参数的类型作为参数。

- 对于 `result_of<>`, 必须传递一个使用相应类型的“函数声明”。

- 若不能调用, 则没有定义类型成员, 因此使用它是一个错误 (可能会在函数模板的声明中 SFINAE 输出使用它的函数模板; 参见第 8.4 节)。
- 可以使用这些特征来获取 Args... 在调用或 `std::invoke()` 给定的可调用 T 时, 获得的返回类型 (关于可调用对象和 `std::invoke()` 的详细信息请参见第 11.1 节。)
- 要求所有给定的类型要么完整(见第 10.3.1 节), (cv 限定的)void, 要么是边界未知的数组类型。
- `invoke_result<>` 自 C++17 后可用, 并取代了 C++17 弃用的 `result_of<>`, 因为 `invoke_result<>` 提供了一些改进, 更容易的语法和接受 T 的抽象类型。
- 例如:

```

1 std::string foo(int);
2
3 using R0 = typename std::result_of<decltype(&foo) (int)>::type; // C++11
4 using R1 = std::result_of_t<decltype(&foo) (int)>; // C++14
5 using R2 = std::invoke_result_t<decltype(foo), int>; // C++17
6
7 struct ABC {
8     virtual ~ABC() = 0;
9     void operator() (int) const {
10    }
11 };
12
13 using T1 = typename std::result_of<ABC(int)>::type; // ERROR: ABC is abstract
14 using T2 = typename std::invoke_result<ABC, int>::type; // OK since C++17

```

请参阅第 11.1.3 节以获得完整的示例。

D.3.2 测试具体操作

特性	作用
<code>is_constructible<T,Args... ></code>	可以用类型 Args 初始化类型 T
<code>is_trivially_constructible<T,Args... ></code>	可以用 Args 类型简单初始化类型 T
<code>is_nothrow_constructible<T,Args... ></code>	可以初始化类型 T 与类型 Args，并且操作不能抛出异常
<code>is_default_constructible<T ></code>	可以不带参数初始化 T
<code>is_trivially_default_constructible<T ></code>	可以不带参数地简单初始化 T
<code>is_nothrow_default_constructible<T ></code>	可以不带参数地初始化 T，操作不能抛出异常
<code>is_copy_constructible<T ></code>	可复制 T
<code>is_trivially_copy_constructible<T ></code>	可简单的复制 T
<code>is_nothrow_copy_constructible<T ></code>	是否可以复制一个 T，而操作不能抛出异常
<code>is_move_constructible<T ></code>	可以移动 T
<code>is_trivially_move_constructible<T ></code>	可以简单移动 T
<code>is_nothrow_move_constructible<T ></code>	可以移动一个 T，而那个操作不能抛出异常
<code>is_assignable<T,T2 ></code>	将 T2 赋值给 T
<code>is_trivially_assignable<T,T2 ></code>	简单地将 T2 赋值给 T
<code>is_nothrow_assignable<T,T2 ></code>	可以将类型 T2 赋值给类型 T，而该操作不抛出异常
<code>is_copyAssignable<T ></code>	复制赋值给 T
<code>is_trivially_copyAssignable<T ></code>	简单的复制赋值给 T
<code>is_nothrow_copyAssignable<T ></code>	复制赋值操作不能抛出异常
<code>is_moveAssignable<T ></code>	移动赋值给 T
<code>is_trivially_moveAssignable<T ></code>	简单的移动赋值给 T
<code>is_nothrow_moveAssignable<T ></code>	移动赋值 T，不能抛出异常
<code>is_destructible<T ></code>	可销毁 T
<code>is_trivially_destructible<T ></code>	可简单的销毁 T
<code>is_nothrow_destructible<T ></code>	销毁 T，而不抛出异常
<code>is_swappable<T ></code>	可以对该类型使用 <code>swap()</code> (C++ 17)
<code>is_nothrow_swappable<T ></code>	可以对该类型调用 <code>swap()</code> ，而不能抛出异常 (C++17)
<code>is_swappable_with<T,T2 ></code>	可以为这两种类型调用 <code>swap()</code> 与特定的值类别 (C++17)
<code>is_nothrow_swappable_with<T,T2 ></code>	可以为这两种类型调用 <code>swap()</code> 与特定的值类别，该操作不能抛出异常 (C++17)

表 D.4. 检查特定操作的特征

表 D.4 列出了允许检查某些特定操作的类型特征。`is_trivially_…`，检查对象、成员或基类调用的所有(子)操作是否简单(既不是用户定义的，也不是虚的)。`is_nothrow_…`，检查调用的操作是否保证不抛出异常，所有 `is_…_constructible` 检查对应的 `is_…_destructible`。例如：

utils/isconstructible.cpp

```
1 #include <iostream>
2
3 class C {
4 public:
5     C() { // default constructor has no noexcept
6     }
7     virtual ~C() = default; // makes C nontrivial
8 };
9
10 int main()
11 {
12     using namespace std;
13     cout << is_constructible_v<C> << '\n' ; // true
14     cout << is_trivially_constructible_v<C> << '\n' ; // false
15     cout << is_nothrow_constructible_v<C> << '\n' ; // false
16     cout << is_copy_constructible_v<C> << '\n' ; // true
17     cout << is_trivially_copy_constructible_v<C> << '\n' ; // true
18     cout << is_nothrow_copy_constructible_v<C> << '\n' ; // true
19     cout << is_destructible_v<C> << '\n' ; // true
20     cout << is_trivially_destructible_v<C> << '\n' ; // false
21     cout << is_nothrow_destructible_v<C> << '\n' ; // true
22 }
```

因虚构造函数的定义，所有操作都不再简单。因为我们定义了一个默认构造函数，没有使用 noexcept，所以可能会抛出异常。默认情况下，所有其他操作都不会抛出异常。

std::is_constructible<T, Args...>::value
std::is_trivially_constructible<T, Args...>::value
std::is_nothrow_constructible<T, Args...>::value

- 若 T 类型的对象可以用 Args 给出的参数类型初始化，则生成 true。(不使用非简单操作或保证不抛出异常)。也就是说，以下代码有效：

```
1 T t(std::declval<Args>() ...);
```

关于 std::declval 的效果，请参见第 11.2.3 节

- true 意味着对象可以销毁 (is_destructible_v<T>, is_trivially_destructible_v<T>, 或 is_nothrow_destructible_v<T> 的结果为 true)。
- 要求所有给定的类型要么完整(见第 10.3.1 节)，要么是 (cv 限定的)void，要么是边界未知的数组。
- 例如：

```
1 is_constructible_v<int> // true
2 is_constructible_v<int, int> // true
3 is_constructible_v<long, int> // true
```

```

4 is_constructible_v<int,void*> // false
5 is_constructible_v<void*,int> // false
6 is_constructible_v<char const*,std::string> // false
7 is_constructible_v<std::string,char const*> // true
8 is_constructible_v<std::string,char const*,int,int> // true

```

- 注意，`is_convertible` 对源类型和目标类型有不同的顺序。

`std::is_default_constructible<T>::value`

`std::is_trivially_default_constructible<T>::value`

`std::is_nothrow_default_constructible<T>::value`

- 若 `T` 类型的对象可以初始化，而不需要初始化参数(不需要使用非简单操作或保证不抛出异常)，则生成 `true`。
- 与 `is_constructible_v<T>`, `is_trivially_constructible_v<T>` 或 `is_nothrow_constructible_v<T>` 相同。
- `true` 意味着可以销毁对象(例如，`is_destructible_v<T>`, `is_trivially_destructible_v<T>` 或 `is_nothrow_destructible_v<T>` 生成 `true`)。
- 要求给定的类型要么完整(见第 10.3.1 节), (cv 限定的)void, 要么是一个边界未知的数组。

`std::is_copy_constructible<T>::value`

`std::is_trivially_copy_constructible<T>::value`

`std::is_nothrow_copy_constructible<T>::value`

- 若 `T` 类型的对象可以通过复制 `T` 类型的另一个值来创建(无需使用非简单操作或保证不抛出异常)，则生成 `true`。
- 若 `T` 不是可引用类型，则生成 `false`(可以是(cv 限定的)void, 也可以是 const、volatile、& 和/或 `&&` 限定的函数类型)。
- 假设 `T` 是一个可引用的类型，就像 `is_constructible<T,T const&>::value`, `is_trivially_constructible<T,T const&>::value` 或 `is_nothrow_constructible<T,T const&>::value` 一样。
- 要找出 `T` 类型的对象是否可以从类型 `T` 的右值复制构造，可使用 `is_constructible<T,T&&>`。
- `true` 意味着可以销毁对象(例如，`is_destructible_v<T>`, `is_trivially_destructible_v<T>` 或 `is_nothrow_destructible_v<T>` 生成 `true`)。
- 要求给定的类型要么完整(见第 10.3.1 节), (cv 限定的)void, 要么是一个边界未知的数组。
- 例如：

```

1 is_copy_constructible_v<int> // yields true
2 is_copy_constructible_v<void> // yields false
3 is_copy_constructible_v<std::unique_ptr<int>> // yields false
4 is_copy_constructible_v<std::string> // yields true
5 is_copy_constructible_v<std::string&> // yields true
6 is_copy_constructible_v<std::string&&> // yields false
7 // in contrast to:
8 is_constructible_v<std::string,std::string> // yields true
9 is_constructible_v<std::string&,std::string&> // yields true
10 is_constructible_v<std::string&&,std::string&&> // yields true

```

```
std::is_move_constructible<T>::value
std::is_trivially_move_constructible<T>::value
std::is_nothrow_move_constructible<T>::value
```

- 若 T 类型的对象可以由 T 类型的右值创建 (无需使用非简单操作或保证不抛出异常), 则生成 true。
- 若 T 不是可引用类型 ((cv 限定的)void 或使用 const、volatile、& 和/或 && 限定的函数类型), 则生成 false。
- 假设 T 是一个可引用的类型, 就像 is_constructible<T,T&&>::value, is_trivially_constructible<T,T&&>::value, 或者 is_nothrow_constructible<T,T&&>::value 一样。
- true 意味着对象可以销毁 (即, is_destructible_v<T>, is_trivially_destructible_v<T>, 或者 is_nothrow_destructible_v<T> 生成 true)。
- 若不能为 T 类型的对象直接调用移动构造函数, 就没有办法检查它是否可以抛出异常。构造函数为 public, 而不删除不行; 还要求对应的类型不是抽象类 (对抽象类的引用或指针可以正常工作)。
- 参见 19.7.2 节了解实现细节。
- 例如:

```
1 is_move_constructible_v<int> // yields true
2 is_move_constructible_v<void> // yields false
3 is_move_constructible_v<std::unique_ptr<int>> // yields true
4 is_move_constructible_v<std::string> // yields true
5 is_move_constructible_v<std::string&> // yields true
6 is_move_constructible_v<std::string&&> // yields true
7 // in contrast to:
8 is_constructible_v<std::string, std::string> // yields true
9 is_constructible_v<std::string&, std::string&> // yields true
10 is_constructible_v<std::string&&, std::string&&> // yields true
```

```
std::is_assignable<TO, FROM>::value
std::is_trivially_assignable<TO, FROM>::value
std::is_nothrow_assignable<TO, FROM>::value
```

- 若可以将 FROM 类型的对象赋值给 TO 类型的对象 (无需使用重要操作或保证不抛出异常), 则生成 true。
- 要求给定的类型要么完整 (见第 10.3.1 节), (cv 限定的)void, 要么是边界未知的数组。
- 作为第一类型的非引用非类类型 is_assignable_v<> 总是会产生 false, 因为这样的类型会产生 prvalues。也就是说, 42 = 77; 不是有效的。对于类类型, 只要给定适当的赋值操作符, 就可以对右值赋值 (这是由于旧的规则, 可以对类类型的右值调用非 const 成员函数)。
- is_convertible 对源类型和目标类型有不同的顺序。
- 例如:

```
1 is_assignable_v<int, int> // yields false
2 is_assignable_v<int&, int> // yields true
```

```
3 is_assignable_v<int&&, int> // yields false
4 is_assignable_v<int&, int&> // yields true
5 is_assignable_v<int&&, int&> // yields false
6 is_assignable_v<int&, long&> // yields true
7 is_assignable_v<int&, void*> // yields false
8 is_assignable_v<void*, int> // yields false
9 is_assignable_v<void*, int&> // yields false
10 is_assignable_v<std::string, std::string> // yields true
11 is_assignable_v<std::string&, std::string&> // yields true
12 is_assignable_v<std::string&&, std::string&> // yields true
```

std::is_copyAssignable<T>::value
std::isTriviallyCopyAssignable<T>::value
std::isNothrowCopyAssignable<T>::value

- 若 T 类型的值可以 (复制-) 赋值给 T 类型的对象 (无需使用非简单操作或保证不抛出异常), 则生成 true。
- 若 T 不是可引用类型 ((cv 限定的)void 或使用 const、volatile、& 和/或 && 限定的函数类型), 则生成 false。
- 假设 T 是一个可引用的类型, 就和 isAssignable<T&, T const&>::value, isTriviallyAssignable<T&, T const&>::value 或 isNothrowAssignable<T&, T const&>::value 一样。
- 要找出是否可以将类型 T 的右值复制赋值给另一个类型 T 的右值, 要使用 isAssignable<T&&, T&&>。
- void、内置数组类型和带有删除复制赋值操作符的类, 不能复制赋值。
- 要求给定的类型要么完整 (见第 10.3.1 节), (cv 限定的)void, 要么是一个边界未知的数组。
- 例如:

```
1 is_copyAssignable_v<int> // yields true
2 is_copyAssignable_v<int&> // yields true
3 is_copyAssignable_v<int&&> // yields true
4 is_copyAssignable_v<void> // yields false
5 is_copyAssignable_v<void*> // yields true
6 is_copyAssignable_v<char[]> // yields false
7 is_copyAssignable_v<std::string> // yields true
8 is_copyAssignable_v<std::unique_ptr<int>> // yields false
```

std::isMoveAssignable<T>::value
std::isTriviallyMoveAssignable<T>::value
std::isNothrowMoveAssignable<T>::value

- 若 T 类型的右值可以移动赋值给 T 类型的对象 (无需使用非简单操作或保证不抛出异常), 则生成 true。
- 若 T 不是可引用类型 ((cv 限定的)void 或使用 const、volatile、& 和/或 && 限定的函数类型), 则生成 false。

- 假设 T 是一个可引用的类型，就像 `is_assignable<T&, T&&>::value`, `is_trivially_assignable<T&, T&&>::value` 或 `is_nothrow_assignable<T&, T&&>::value` 一样。
- `void`、内置数组类型和带有删除移动赋值操作符的类不能进行移动赋值。
- 要求给定的类型要么完整(见第 10.3.1 节)，要么是 (cv 限定的)`void` 或一个边界未知的数组。
- 例如：

```

1 is_moveAssignable_v<int> // yields true
2 is_moveAssignable_v<int&> // yields true
3 is_moveAssignable_v<int&&> // yields true
4 is_moveAssignable_v<void> // yields false
5 is_moveAssignable_v<void*> // yields true
6 is_moveAssignable_v<char[]> // yields false
7 is_moveAssignable_v<std::string> // yields true
8 is_moveAssignable_v<std::unique_ptr<int>> // yields true

```

`std::is_swappable_with<T1, T2>::value`

`std::is_nothrow_swappable_with<T1, T2>::value`

- 若类型 T1 的表达式可以与类型 T2 的表达式交换，则生成 `true`，除非引用类型只确定表达式的值类别(保证不抛出异常)。
- 要求给定的类型要么完整(见第 10.3.1 节)，要么是 (cv 限定的)`void` 或边界未知的数组。
- 作为第一或第二种类型的非引用、非类类型 `is_swappable_with_v<>` 总是会产生 `false`，因为这样的类型会产生 prvalue。也就是说，`swap(42,77)` 无效。
- 例如：

```

1 is_swappable_with_v<int, int> // yields false
2 is_swappable_with_v<int&, int> // yields false
3 is_swappable_with_v<int&&, int> // yields false
4 is_swappable_with_v<int&, int&> // yields true
5 is_swappable_with_v<int&&, int&&> // yields false
6 is_swappable_with_v<int&, long&> // yields false
7 is_swappable_with_v<int&, void*> // yields false
8 is_swappable_with_v<void*, int> // yields false
9 is_swappable_with_v<void*, int&> // yields false
10 is_swappable_with_v<std::string, std::string> // yields false
11 is_swappable_with_v<std::string&, std::string&> // yields true
12 is_swappable_with_v<std::string&&, std::string&&> // yields false

```

- C++17 后可用。

`std::is_swappable<T>::value`

`std::is_nothrow_swappable<T>::value`

- 若 T 类型的左值可以交换(保证不抛出异常)，则生成 `true`。
- 假设 T 是一个可引用的类型。就像 `is_swappable_with<T&, T&&>::value` 或 `is_nothrow_swappable_with<T&, T&&>::value` 一样。
- 若 T 不是可引用类型 ((cv 限定的)`void` 或使用 `const`、`volatile`、`&` 和/或 `&&` 限定的函数类型)，则生成 `false`。

- 要找出一个 T 的右值是否可以与另一个 T 的右值交换,请使用 `is_swappable_with<T&&, T&&>`。
- 要求给定的类型是一个完整的类型(第 10.3.1 节)、(cv 限定的)void 或一个边界未知的数组。
- 例如:

```

1 is_swappable_v<int> // yields true
2 is_swappable_v<int&> // yields true
3 is_swappable_v<int&&> // yields true
4 is_swappable_v<std::string&&> // yields true
5 is_swappable_v<void> // yields false
6 is_swappable_v<void*> // yields true
7 is_swappable_v<char[]> // yields false
8 is_swappable_v<std::unique_ptr<int>> // yields true

```

- C++17 后可用。

D.3.3 类型之间的关系

表 D.5 列出了允许测试类型之间特定关系的类型特征, 包括检查为类类型提供了哪些构造函数和赋值操作符。

特性	作用
<code>is_same<T1,T2 ></code>	T1 和 T2 是相同的类型(包括 const/volatile 限定符)
<code>is_base_of<T,D ></code>	类型 T 是类型 D 的基类
<code>is_convertible<T,T2 ></code>	类型 T 可以转换为类型 T2

表 D.5. 测试类型关系的特征

`std::is_same<T1, T2>::value`

- 若 T1 和 T2 命名相同的类型, 包括 cv 限定符(const 和 volatile), 则生成 true。
- 若一个类型是另一个类型的别名, 则生成 true。
- 若两个对象由相同类型的对象初始化, 则生成 true。
- 对于与两个不同的 Lambda 表达式关联的(闭包)类型, 即使定义了相同的行为, 也会产生 false。
- 例如:

```

1 auto a = nullptr;
2 auto b = nullptr;
3 is_same_v<decltype(a), decltype(b)> // yields true
4
5 using A = int;
6 is_same_v<A, int> // yields true
7
8 auto x = [] (int) {};
9 auto y = x;
10 auto z = [] (int) {};
11 is_same_v<decltype(x), decltype(y)> // yields true
12 is_same_v<decltype(x), decltype(z)> // yields false

```

- 参见第 19.3.3 节了解实现细节。

`std::is_base_of<B, D>::value`

- 若 B 是 D 的基类或 B 与 D 是同一个类，则为 true。
- 类型是 cv 限定的、私有的还是受保护的继承并不重要，D 有多个 B 的基类，或者 D 通过多个继承(通过虚拟继承)将 B 作为基类。
- 若至少有一个类型是联合类型，则生成 false。
- 要求类型 D 要么是完整的(见第 10.3.1 节)，与 B 具有相同的类型(忽略任何 const/volatile 限定)，要么既不是结构体也不是类。
- 例如：

```

1 class B {
2 };
3 class D1 : B {
4 };
5 class D2 : B {
6 };
7 class DD : private D1, private D2 {
8 };
9 is_base_of_v<B, D1> // yields true
10 is_base_of_v<B, DD> // yields true
11 is_base_of_v<B const, DD> // yields true
12 is_base_of_v<B, DD const> // yields true
13 is_base_of_v<B, B const> // yields true
14 is_base_of_v<B&, DD&> // yields false (no class type)
15 is_base_of_v<B[3], DD[3]> // yields false (no class type)
16 is_base_of_v<int, int> // yields false (no class type)

```

`std::is_convertible<FROM, TO>::value`

- 若 FROM 类型的表达式可转换为 TO 类型，则生成 true。因此，以下代码有效：

```

1 TO test() {
2     return std::declval<FROM>();
3 }

```

关于 `std::declval` 的效果，请参见第 11.2.3 节。

- 类型 FROM 之上的引用仅用于确定转换表达式的值类别；基础类型就是源表达式的类型。
- `is_constructible` 并不总等价于 `is_convertible`。例如：

```

1 class C {
2 public:
3     explicit C(C const&); // no implicit copy constructor
4     ...
5 };
6 is_constructible_v<C, C> // yields true
7 is_convertible_v<C, C> // yields false

```

- 要求给定的类型要么完整(见第 10.3.1 节), 要么是 (cv 限定的)void 或是边界未知的数组。
- `s_constructible`(参见 D.3.2 节) 和 `is_assignable`(参见 D.3.2 节) 对于源和目标类型有不同的顺序。
- 参见第 19.5 节了解实现细节。

D.4 类型结构

表 D.6 中列出的特征可以从其他类型构建类型。

特征	作用
<code>remove_const<T></code>	对应无 <code>const</code> 的类型
<code>remove_volatile<T></code>	对应无 <code>volatile</code> 的类型
<code>remove_cv<T></code>	对应无 <code>const</code> 和 <code>volatile</code> 的类型
<code>add_const<T></code>	对应有 <code>const</code> 的类型
<code>add_volatile<T></code>	对应有 <code>volatile</code> 的类型
<code>add_cv<T></code>	对应有 <code>const volatile</code> 的类型
<code>make_signed<T></code>	对应的有符号非引用类型
<code>make_unsigned<T></code>	对应的无符号非引用类型
<code>remove_reference<T></code>	对应的非引用类型
<code>add_lvalue_reference<T></code>	对应的左值引用类型(右值变成左值)
<code>add_rvalue_reference<T></code>	对应的右值引用类型(左值保持不变)
<code>remove_pointer<T></code>	指针的引用类型(其他类型相同)
<code>add_pointer<T></code>	指向对应非引用类型的指针类型
<code>remove_extent<T></code>	数组的元素类型(其他类型相同)
<code>remove_all_extents<T></code>	多维数组的元素类型(其他类型相同)
<code>decay<T></code>	转换为对应的“值”类型

表 D.6. 类型构造的特性

```
std::remove_const<T>::type
std::remove_volatile<T>::type
std::remove_cv<T>::type
```

- 生成不带 `const` 或/和 `volatile` 的 `T` 类型。
- `const` 指针是 `const` 限定类型, 而非 `const` 指针或指向 `const` 类型的引用则不是 `const` 限定类型。
例如:

```
1 remove_cv_t<int> // yields int
2 remove_const_t<int const> // yields int
3 remove_cv_t<int const volatile> // yields int
4 remove_const_t<int const&> // yields int const& (only refers to int const)
```

类型构造特征的顺序很重要:

由于这个原因，C++17 后的下一个标准可能会提供 remove_refcv 特性。

```
1 remove_const_t<remove_reference_t<int const&>> // yields int
2 remove_reference_t<remove_const_t<int const&>> // yields int const
```

相反，我们可能更喜欢使用 std::decay<>，但它会将数组和函数类型转换成相应的指针类型(见第 D.4 节)：

```
1 decay_t<int const&> // yields int
```

- 参见 19.3.2 节了解实现细节。

std::add_const<T>::type

std::add_volatile<T>::type

std::add_cv<T>::type

- 生成在带有 const 或/和 volatile 限定符的 T 类型。
- 将这些特征应用于引用类型或函数类型无效。例如：

```
1 add_cv_t<int> // yields int const volatile
2 add_cv_t<int const> // yields int const volatile
3 add_cv_t<int const volatile> // yields int const volatile
4 add_const_t<int> // yields int const
5 add_const_t<int const> // yields int const
6 add_const_t<int&> // yields int&
```

std::make_signed<T>::type

std::make_unsigned<T>::type

- 生成对应的有符号/无符号 T 类型。
- T 是枚举类型，或除 bool 以外的整型 (cv 限定)。所有其他类型都会导致未定义行为(参见第 19.7.1 节，讨论如何避免这种未定义行为)。
- 将这些特征应用于引用类型或函数类型无效，而非 const 指针或对 const 类型的引用不由 const 限定。例如：

```
1 make_unsigned_t<char> // yields unsigned char
2 make_unsigned_t<int> // yields unsigned int
3 make_unsigned_t<int const&> // undefined behavior
```

std::remove_reference<T>::type

- 生成引用类型 T 引用的类型(若 T 不是引用类型，则生成 T 本身)。
- 例如：

```
1 remove_reference_t<int> // yields int
2 remove_reference_t<int const> // yields int const
3 remove_reference_t<int const&> // yields int const
4 remove_reference_t<int&&> // yields int
```

- 引用类型本身不是 `const` 类型。因此，类型构造特征的顺序很重要：

由于这个原因，C++17 后的下一个标准可能会提供 `remove_refcv` 特性。

```
1 remove_const_t<remove_reference_t<int const&>> // yields int
2 remove_reference_t<remove_const_t<int const&>> // yields int const
```

相反，我们可能更喜欢使用 `std::decay`，但会将数组和函数类型转换成相应的指针类型（见第 D.4 节）：

```
1 decay_t<int const&> // yields int
```

- 参见 19.3.2 节了解实现细节。

`std::add_lvalue_reference`<T>::type

`std::add_rvalue_reference`<T>::type

- 若 T 是可引用类型，则生成对 T 的左值或右值引用。
- 若 T 不是可引用的 (cv 限定的)void 或用 `const`、`volatile`、`&` 和/或 `&&` 限定的函数类型，则生成 T。
- 若 T 已经是一个引用类型，特性使用引用折叠规则（参见第 15.6.1 节）：只有当使用 `add_rvalue_reference` 并且 T 是一个右值引用时，结果是一个右值引用。
- 例如：

```
1 add_lvalue_reference_t<int> // yields int&
2 add_rvalue_reference_t<int> // yields int&&
3 add_rvalue_reference_t<int const> // yields int const&&
4 add_lvalue_reference_t<int const&> // yields int const&
5 add_rvalue_reference_t<int const&> // yields int const& (reference collapsing rules)
6 add_rvalue_reference_t<remove_reference_t<int const&>> // yields int&&
7 add_lvalue_reference_t<void> // yields void
8 add_rvalue_reference_t<void> // yields void
```

- 参见第 19.3.2 节了解实现细节。

`std::remove_pointer`<T>::type

- 生成指针类型 T 所指向的类型（若不是指针类型，则生成 T 本身）。
- 例如：

```
1 remove_pointer_t<int> // yields int
2 remove_pointer_t<int const*> // yields int const
3 remove_pointer_t<int const* const* const*> // yields int const* const
```

`std::add_pointer`<T>::type

- 生成指向 T 的指针的类型，或在引用类型 T 的情况下，生成指向 T 基础类型的指针的类型。
- 若没有这种类型，则生成 T（适用于 cv 限定的函数类型）。
- 例如：

```

1 add_pointer_t<void> // yields void*
2 add_pointer_t<int const* const> // yields int const* const*
3 add_pointer_t<int&> // yields int
4 add_pointer_t<int[3]> // yields int(*)[3]
5 add_pointer_t<void(&)(int)> // yields void(*)(int)
6 add_pointer_t<void(int)> // yields void(*)(int)
7 add_pointer_t<void(int) const> // yields void(int) const (no change)

```

std::remove_extent<T>::type

std::remove_all_extents<T>::type

- 给定数组类型, `remove_extent`生成直接元素类型(本身可以是数组类型), 而 `remove_all_extents`剥离所有“数组层”来产生底层元素类型(因此不再是数组类型)。若 T 不是数组类型, 则生成 T 本身。
- 指针没有维度信息, 数组类型中未指定的边界指定维度(用数组类型声明的函数参数没有实际的数组类型, 而且 `std::array` 也不是数组类型。参见第 D.2.1 节)。
- 例如:

```

1 remove_extent_t<int> // yields int
2 remove_extent_t<int[10]> // yields int
3 remove_extent_t<int[5][10]> // yields int[10]
4 remove_extent_t<int[][10]> // yields int[10]
5 remove_extent_t<int*> // yields int*
6 remove_all_extents_t<int> // yields int
7 remove_all_extents_t<int[10]> // yields int
8 remove_all_extents_t<int[5][10]> // yields int
9 remove_all_extents_t<int[][10]> // yields int
10 remove_all_extents_t<int(*)[5]> // yields int(*)[5]

```

- 参见第 23.1.2 节了解实现细节。

std::decay<T>::type

- 生成衰变的 T。
- 对于 T 类型执行以下转换:
 - 首先, 使用 `remove_reference`(参见 D.4)。
 - 若结果是数组类型, 则生成指向直接元素类型的指针(参见第 7.1 节)。
 - 若结果是一个函数类型, 则生成该函数类型的 `add_pointer` 产生的类型(参见第 11.1.1 节)。
 - 生成的结果不包含 `const/volatile` 限定符。
- 初始化 `auto` 类型的对象时, 通过参数的值传递或类型转换 `decay<>` 模型。
- `decay<>` 对于处理可能用引用类型替代, 但用于确定另一个函数的返回类型或参数类型的模板参数特别有用。关于讨论和使用 `std::decay<>()`(历史上有用后者来实现 `std::make_pair<>()`) 的例子, 请参阅第 1.3.2 节和第 7.6 节。
- 例如:

```

1 decay_t<int const&> // yields int
2 decay_t<int const[4]> // yields int const*
3 void foo();
4 decay_t<decltype(foo)> // yields void(*)()

```

- 参见 19.3.2 节了解实现细节。

D.5 其他特性

表 D.7 列出了所有剩余的类型特征，可以查询特殊属性或提供更复杂的类型转换。

特性	作用
enable_if<B,T=void>	当 bool B 为 true 时才返回类型 T
conditional<B,T,F>	若 bool B 为真，则返回类型 T，否则返回类型 F
common_type<T1,...>	所有传递类型的通用类型
aligned_storage<Len>	默认 Len 字节对齐的类型
aligned_storage<Len,Align>	根据 size_t Align 的除数，对齐 Len 字节的类型
aligned_union<Len,Types...>	Len 字节对齐的 Types...联合类型

表 D.7. 剩余类型特征

std::enable_if<cond>::type

std::enable_if<cond, T>::type

- 若 cond 为 true，则在其成员类型中返回 void 或 T。否则，不定义成员类型。
- 因为当 cond 为 false 时，没有定义类型成员，因此该特性可以通常给定的条件，禁用或使用 SFINAE 退出函数模板。
- 请参阅第 6.3 节了解详细信息和第一个例子。有关使用参数包的另一个例子，请参阅第 D.6 节。
- 有关 std::enable_if 如何实现的详细信息，请参见第 20.3 节。

std::conditional<cond, T, F>::type

- 若 cond 为 true，则为 T，否则为 F。
- 这是 19.7.1 节介绍的特征 IfThenElseT 的标准版本。
- 与普通的 C++ if-then-else 语句不同，then 和 else 分支的模板参数都是在选择之前进行求值的，因此两个分支都不包含错误的代码，或者程序可能是格式错误的。可能需要添加一个间接级别，以避免 then 和 else 分支中的表达式在未使用该分支时进行计算。第 19.7.1 节演示了特征 IfThenElseT 的这个功能，其具有相同的行为。
- 参见第 11.5 节中的示例。
- 请参阅 19.7.1 节，了解 std::conditional 的实现。

std::common_type<T...>::type

- 生成给定 T1, T2, ..., Tn 类型的“共同类型”。

- 普通类型的计算比本附录中讨论的要复杂一些。粗略地说，当第二个和第三个操作数是 U 和 V 类型时（引用类型仅用于确定两个操作数的值类别），U 和 V 两种类型是条件三元操作符产生的类型；若该类型无效，则没有共同类型。`decay_t` 应用于此结果。这个计算可以通过特化重写为 `std::common_type<U, V>`（C++ 标准库中，偏特化存在于持续时间和时间点）。
- 若没有给出类型或没有公共类型存在，则没有定义类型成员，因此会出错（可能会导致 SFINAE 输出使用它的函数模板）。
- 若给出单类型，结果是对该类型为 `decay_t`。
- 对于两个以上的类型，`common_type` 递归地用公共类型替换前两个类型 T1 和 T2。若该进程失败，则没有通用类型。
- 处理普通类型时，传递的类型是衰变类型，因此特性总是产生衰变类型（参见 D.4 节）。
- 参见第 1.3.3 节对该特性应用的讨论和示例。
- 这个特性的主模板通常是通过以下方式实现的（这里只使用两个参数）：

```

1 template<typename T1, typename T2>
2 struct common_type<T1, T2> {
3     using type = std::decay_t<decltype(true ? std::declval<T1>()
4         : std::declval<T2>());}
5 };

```

`std::aligned_union<MIN_SZ, T...>::type`

- 生成一个普通的旧数据类型（POD）可用作未初始化存储，大小至少为 `MIN_SZ`，适合保存给定的类型 `T1, T2, ...Tn`。
- 此外，还可以生成一个静态成员 `alignment_value`，其值表示所有给定类型中需要严格对齐于该值，对于结果类型来说等价于
 - `std::alignment_of<type>::value`（详见 D.3.）
 - `alignof(type)`
- 要求至少提供一种类型。
- 例如：

```

1 using POD_T = std::aligned_union_t<0, char,
2                         std::pair<std::string, std::string>>;
3 std::cout << sizeof(POD_T) << '\n';
4 std::cout << std::aligned_union<0, char,
5                         std::pair<std::string, std::string>
6                         ::alignment_value;
7 << '\n';

```

使用 `aligned_union` 来获取对齐值，而不是使用 `aligned_union_t` 获取类型。

`std::aligned_storage<MAX_TYPE_SZ>::type`

`std::aligned_storage<MAX_TYPE_SZ, DEF_ALIGN>::type`

- 生成一个普通的旧数据类型（POD）可用作未初始化的存储，其大小可容纳所有可能的类型，其大小最大为 `MAX_TYPE_SZ`，并需要考虑到默认对齐或传递 `DEF_ALIGN` 的对齐情况。

- 要求 MAX_TYPE_SZ 大于 0，平台至少要有一种对齐值为 DEF_ALIGN 的类型。
- 例如：

```
1 using POD_T = std::aligned_storage_t<5>;
```

D.6 组合类型特性

大多数上下文中，可以使用逻辑操作符组合多个类型特征谓词。模板元编程的上下文中，这并不够：

- 必须处理可能失败的特征(不完整的类型)。
- 组合类型特征定义。

为此提供了类型特征 std::conjunction<>, std::disjunction<> 和 std::negation<>。

这些辅助函数会短路布尔值的计算(分别在 && 和 || 的第一个 false 后中止计算，或第一个 true 后中止计算)。例如，若使用不完整类型：

```
1 struct X {
2     X(int); // converts from int
3 };
4 struct Y; // incomplete type
```

因为 is_constructible 会导致(不完整类型的)未定义行为(尽管有些编译器接受这段代码)，所以以下代码可能无法编译：

```
1 // undefined behavior:
2 static_assert(std::is_constructible<X, int>{},
3               || std::is_constructible<Y, int>{},
4               "can't init X or Y from int");
```

因为 is_constructible<X, int> 已经产生了 true，以下语句保证可以编译：

```
1 // OK:
2 static_assert(std::disjunction<std::is_constructible<X, int>,
3               std::is_constructible<Y, int>{},
4               "can't init X or Y from int");
```

另一个是通过逻辑组合现有类型特征，来定义新类型特征的一种简单方法，可以定义一个特性来检查一个类型是否“不是指针”(既不是指针，也不是成员指针，也不是空指针)：

```
1 template<typename T>
2 struct isNoPtrT : std::negation<std::disjunction<std::is_null_pointer<T>,
3                               std::is_member_pointer<T>,
4                               std::is_pointer<T>>>
5 {
6 };
```

因为结合了相应的特征类，所以这里不能使用逻辑操作符。根据这个定义，以下方式可用：

```
1 std::cout << isNoPtrT<void*>::value << '\n' ; // false
2 std::cout << isNoPtrT<std::string>::value << '\n' ; // true
```

```

3 auto np = nullptr;
4 std::cout << isNoPtrT<decltype(np)>::value << '\n' ; // false

```

并配以相应的变量模板:

```

1 template<typename T>
2 constexpr bool isNoPtr = isNoPtrT<T>::value;

```

可以这样写:

```

1 std::cout << isNoPtr<void*> << '\n' ; // false
2 std::cout << isNoPtr<int> << '\n' ; // true

```

最后一个例子，下面的函数模板只有在所有模板参数既不是类，也不是联合时才启用:

```

1 template<typename... Ts>
2 std::enable_if_t<std::conjunction_v<std::negation<std::is_class<Ts>>...,
3                 std::negation<std::is_union<Ts>>...>
4 >>
5 print(Ts...)
6 {
7     ...
8 }

```

省略号放在 `std::negation` 后面，以便用于参数包的每个元素。

特性	作用
<code>conjunction<B...></code>	逻辑和布尔特征 B…(C++17)
<code>disjunction<B...></code>	逻辑或布尔特征 B…(C++17)
<code>negation</code>	逻辑非布尔特征 B(C++17)

表 D.8. 组合其他类型特征

`std::conjunction<B...>::value`

`std::disjunction<B...>::value`

- 传入的特性 B... 至少有一个或全部为布尔特征时，生成 true
- 逻辑上分别对传入的特征使用操作符 `&&` 或 `||`。
- 这两个特性都会短路(第一个 `false` 或 `true` 之后中止计算)。
- C++17 后可用

`std::negation::value`

- 传入的 B 是布尔特征则生成 `false`。
- 对传入的特性使用逻辑非运算符。
- C++17 后可用

D.7 其他应用方式

C++ 标准库提供了一些其他实用工具，这些工具对于编写可移植的泛型代码非常有用。

特性	作用
<code>declval<T>()</code>	生成一个不构造类型的“对象”(右值引用)
<code>addressof(r)</code>	生成对象或函数的地址

表 D.9. 用于元编程的其他工具

std::declval<T>()

- 头文件 `<utility>` 中定义。
- 生成类型的“对象”或函数，而不使用构造函数或初始化。
- 若 T 为 `void`，则返回类型为 `void`。
 可以用于处理未求值表达式中的对象或函数类型。
- 其简单定义如下：

```
1 template<typename T>
2 add_rvalue_reference_t<T> declval() noexcept;
```

因此：

- 若 T 是普通类型或右值引用，则生成 `T&&`。
 - 若 T 是一个左值引用，它会产生一个 `T&`。
 - 若 T 是 `void`，它就产生 `void`。
- 参阅第 19.3.4 节和第 11.2.3 节，以及第 D.5 节中 `common_type<>` 类型特性，可以看看它的示例。

std::addressof(r)

- 头文件 `<memory>` 中定义。
- 即使为对象或函数的类型重载了操作符 `&`，也会产生对象或函数 r 的地址。
- 详见第 11.2.2 节。

附录 E：概念

多年来，C++ 语言设计人员一直在探索如何约束模板的参数。例如，我们的原型 `max()` 模板中，希望预先声明，对于不能使用小于操作符进行比较的类型，不应该调用。其他模板可能希望使用有效的“迭代器”类型(对于该术语的一些正式定义)或有效的“算术”类型(可能比内置算术类型集更广泛的概念)进行实例化。

概念是一个或多个模板参数的命名约束集。开发 C++11 标准的过程中，为概念设计了一个丰富的系统。但将该特性集成到语言规范中的话，需要太多的委员会资源，概念最终从 C++11 中删除了。过了一段时间，就有一种不同的特性设计提出，概念似乎将以某种形式进入语言。就在这本书将要印刷之际，标准化委员会投票决定将新设计的概念集成到 C++20 的草案中。这里，我们将介绍这种新设计的概念。

本书的主要章节中，已经提出并展示了一些概念的应用：

- 第 6.5 节说明了如何使用需求和概念来启用构造函数，只有当模板参数可转换为字符串时(以避免意外地将构造函数用作复制构造函数)。
- 第 18.4 节展示了如何使用概念，来指定和要求用于表示几何对象类型的约束。

E.1 使用概念

首先研究一下如何在用户代码中使用概念(即，定义模板的代码不必定义应用于模板参数的概念)。

处理需求

下面是我们的双参数 `max()` 模板，其有一个约束条件：

```
1 template<typename T> requires LessThanComparable<T>
2 T max(T a, T b) {
3     return b < a ? a : b;
4 }
```

增加了一条 `requires` 子句

```
1 requires LessThanComparable<T>
```

假设之前已经进行了声明——通过头文件包含——`LessThanComparable` 的概念。

这样的概念是布尔谓词(即产生 `bool` 类型值的表达式)，其计算结果为常量表达式。因为约束在编译时计算，因此就生成的代码而言，不会产生多余的开销：这个受约束的模板生成的代码，与之前讨论的不受约束的版本性能一样。

当尝试使用该模板时，其不会实例化，直到对 `requires` 子句进行了计算，并生成了 `true`。若产生的是 `false`，则可能会产生一个错误，解释需求的哪一部分失败了(或者，可能会选择一个匹配的重载模板，但是没有满足需求)。

`require` 子句不必用概念来表示(这样做是很好的实践，会产生更好的诊断信息)：可以使用布尔常量表达式。如 6.5 节所讨论的，下面的代码确保模板构造函数不能用作复制构造函数：

```

1 class Person
2 {
3     private:
4     std::string name;
5     public:
6     template<typename STR>
7     requires std::is_convertible_v<STR, std::string>
8     explicit Person(STR&& n)
9     : name(std::forward<STR>(n)) {
10    std::cout << "TMPL-CONSTR for '" << name << "' \n";
11 }
12 ...
13 };

```

因为特别的布尔表达式 (在这种情况下使用类型特征), 所以这里不使用命名概念 (参见第 E.2 节) 可能更合理

```
1 std::is_convertible_v<STR, std::string>
```

用于修复可能使用的模板构造函数, 而不是复制构造函数的问题。如何组织概念和约束的细节仍然是 C++ 社区探索的领域, 并且可能随着时间的推移而演变, 但似乎有一个共识, 即概念应该反映代码的含义, 而不是它是否可以编译。

处理多种需求

上例中, 只有一个 `requires`, 但是有多个 `requires` 的情况并不少见。可以想象一个描述元素值序列的 Sequence 概念 (与标准中相同的概念相匹配) 和一个模板 `find()`, 给定一个序列和一个值, 返回一个指向该序列中第一个出现的值的迭代器 (如果有的话)。该模板可以定义如下:

```

1 template<typename Seq>
2     requires Sequence<Seq> &&
3         EqualityComparable<typename Seq::value_type>
4     typename Seq::iterator find(Seq const& seq,
5         typename Seq::value_type const& val)
6 {
7     return std::find(seq.begin(), seq.end(), val);
8 }

```

对该模板的调用都将首先依次检查每个需求, 并且只有当所有需求产生 `true` 时, 才能为调用选择模板, 并实例化模板 (若重载解析不会因为其他原因丢弃模板)。

也可以使用 `||` 来表达“替代”需求。很少需要这样做, 在 `require` 子句中过度使用 `||` 操作符可能会占用编译资源 (使编译速度明显变慢)。在某些情况下, 这会非常方便。例如:

```

1 template<typename T>
2     requires Integral<T> || 
3         FloatingPoint<T>
4 T power(T b, T p);

```

单需求也可以涉及多个模板参数, 而单概念可以表达多个模板参数上的谓词。例如:

```
1 template<typename T, typename U>
2     requires SomeConcept<T, U>
3     auto f(T x, U y) -> decltype(x+y)
```

因此，概念可以在类型参数之间建立关系。

单个需求的快捷方式

为了减少 `requires` 子句的符号开销，当约束只涉及一个参数时，可以使用一种语法快捷方式。可以通过使用上面 `max()` 模板声明的简写：

```
1 template<LessThanComparable T>
2 T max(T a, T b) {
3     return b < a ? a : b;
4 }
```

功能上等价于前面的 `max()` 定义。然而，当重新声明一个受约束的模板时，必须使用与原始声明的形式相同（从这个意义上说，它们只在功能上等价）。

可以在 `find()` 模板中对这两个需求之一使用相同的简写方式：

```
1 template<Sequence Seq>
2     requires EqualityComparable<typename Seq::value_type>
3     typename Seq::iterator find(Seq const& seq,
4         typename Seq::value_type const& val)
5     {
6         return std::find(seq.begin(), seq.end(), val);
7     }
```

这等价于前面为序列类型定义的 `find()` 模板。

E.2 定义概念

概念很像 `bool` 类型的 `constexpr` 变量模板，但类型没有显式指定：

```
1 template<typename T> concept LessThanComparable = ... ;
```

这里的“`...`”可以用一个表达式来代替，该表达式使用各种特征来确定，类型 `T` 是否确实可以使用小于操作符进行比较。但是概念提供了一个工具可简化这个任务：`requires` 表达式（与上面描述的 `requires` 子句不同）。以下是这个概念的完整定义：

```
1 template<typename T>
2 concept LessThanComparable = requires(T x, T y) {
3     { x < y } -> bool;
4 }
```

请注意 `require` 表达式如何包含一个可选参数列表：这些参数永远不会用参数替换，可以认为是一组“哑变量”，可用来在 `require` 表达式体中表达需求。短语表达了这样的需求

```
1 { x < y } -> bool;
```

这种语法意味着 (a) 表达式 $x < y$ 必须在 SFINAE 意义上有效, (b) 表达式的结果必须可转换为 `bool` 类型。这种形式的短语中, 关键字 `noexcept` 可以插入到`->`标记之前, 以表示大括号中的表达式不会抛出异常(即, 应用于该表达式的 `noexcept(...)` 为 `true`)。若不需要这样的约束, 短语的隐式转换部分(即`->`类型)可以省略, 若只需要检查表达式的有效性, 则可以删除大括号, 这样短语就可以简化为表达式。

```
1 template<typename T>
2 concept Swappable = requires(T x, T y) {
3     swap(x, y);
4 }
```

`requires` 表达式还可以表达对关联类型的需求。考虑前面假设的序列概念: 除了要求 `seq.begin()` 等表达式有效性外, 还需要相应的序列迭代器类型。可以表示为:

```
1 template<typename Seq>
2 concept Sequence = requires(Seq seq) {
3     typename Seq::iterator;
4     { seq.begin() } -> Seq::iterator;
5     ...
6 }
```

`typename type;` 表示类型存在的需求(这称为类型需求)。本例中, 必须存在的类型是概念模板参数的成员, 但不一定总是这样, 可以要求存在一个 `IteratorFor<Seq>` 类型, 通过 `require` 短语实现

```
1 ...
2 typename IteratorFor<Seq>;
3 ...
```

上面的 `Sequence` 概念定义展示了, 通过逐个列出短语来组合短语。还有第三类需求短语, 其只包含调用另一个概念。假设有一个迭代器的概念, 希望序列概念不仅要求 `Seq::iterator` 是一种类型, 而且要求该类型满足 `iterator` 概念的约束条件。表达式如下:

```
1 template<typename Seq>
2 concept Sequence = requires(Seq seq) {
3     typename Seq::iterator;
4     requires Iterator<typename Seq::iterator>;
5     { seq.begin() } -> Seq::iterator;
6     ...
7 }
```

也就是说, 可以在 `requires` 表达式中添加子句(这种短语视为嵌套需求)。

E.3 重载约束

假设已经定义了概念 `IntegerLike<T>` 和 `StringLike<T>`, 并且决定编写模板来打印出这两个概念类型的值。可以这样做:

```
1 template<IntegerLike T> void print(T val); // #1
2 template<StringLike T> void print(T val); // #2
```

若没有不同的约束，这两个声明将声明相同的模板。但约束是模板签名的一部分，为了在重载解析期间区分模板。若发现两个模板都是可行的候选模板，但只有模板 #1 满足其约束，那么重载会选择满足条件的模板。假设 int 满足 IntegerLike, std::string 满足 StringLike，但反之不行：

```
1 int main()
2 {
3     printf(1); // selects template #1
4     printf("1"s); // selects template #2
5 }
```

可以想象一个类似字符串的类型，支持类似整数的计算。若”6”_NS 和”7”_NS 是该类型的两个字面值，将这些字面值相乘将产生与”42”_NS 相同的值。这样的类型可能同时满足 IntegerLike 和 StringLike，因此，像 print(”42”_NS) 这样的调用将有歧义。

E.3.1 约束类型

第一次讨论重载函数模板时，所涉及的约束通常互斥。在使用 IntegerLike 和 StringLike 的示例中，可以设想同时满足这两个概念的类型，但希望这种情况比较罕见，以便重载打印模板仍然可用。

然而，有些概念从来不相互排斥，而是“包容”。例子是标准库的迭代器类别：输入迭代器、前向迭代器、双向迭代器、随机访问迭代器，以及 C++17 中的连续迭代器。

连续迭代器是 C++17 中引入的随机访问迭代器改进版。若更改了标签，因为依赖于 std::random_access_iterator_tag 的现有算法将不再选择，所以没有为其添加 std::continuous_iterator_tag。

假设这里有一个 ForwardIterator 的定义：

```
1 template<typename T>
2 concept ForwardIterator = ...;
```

“更精细的”概念 BidirectionalIterator 可以这样定义：

```
1 template<typename T>
2 concept BidirectionalIterator =
3     ForwardIterator<T> &&
4     requires (T it) {
5         { --it } -> T&;
6     };
```

在前向迭代器已经提供的功能基础上，添加了前缀减法操作符的功能。

考虑 std::advance() 算法（称之为 advanceIter()），重载了使用受限模板的前向和双向迭代器：

```
1 template<ForwardIterator T, typename D>
2 void advanceIter(T& it, D n)
3 {
4     assert(n >= 0);
5     for (; n != 0; --n) { ++it; }
6 }
7
```

```

8 template<BidirectionalIterator T, typename D>
9 void advanceIter(T& it, D n)
10 {
11     if (n > 0) {
12         for (; n != 0; --n) { ++it; }
13     } else if (n < 0) {
14         for (; n != 0; ++n) { --it; }
15     }
16 }

```

当使用普通的前向迭代器(即非双向迭代器)调用 `advanceIter()` 时, 只有第一个模板的约束条件满足, 重载解析很简单: 选择第一个模板, 双向迭代器将满足两个模板的约束条件。这种情况下, 当重载解析在其他方面并不偏好某个候选时, 将偏好其约束包含其他候选约束的候选, 反之不成立。包容的确切定义超出了这个介绍性附录的范畴, 但只要知道约束 `C2<Ts...>` 定义为要求约束 `C1<Ts...>` 和其他约束(即 `&&`), 则前者包含后者, 即可。

用于标准化的规范比这更强大。其将约束分解为“原子组件”的集合(包括 `require` 表达式的一部分), 并分析这些集合, 其中一个是否是另一个的严格子集。

示例中, `BidirectionalIterator<T>` 包含 `ForwardIterator<T>`, 因此使用双向迭代器调用时, 首选第二个 `advanceIter()` 模板。

E.3.2 约束和标签调度

第 20.2 节中, 讨论了使用标记调度重载 `advanceIter()` 算法的问题。该方法以一种相当优雅的方式集成到受限模板中, 输入迭代器和前向迭代器不能通过语法接口加以区分。所以, 可以使用标签来定义其中之一:

```

1 template<typename T>
2 concept ForwardIterator =
3 InputIterator<T> &&
4 requires {
5     typename std::iterator_traits<T>::iterator_category;
6     is_convertible_v<std::iterator_traits<T>::iterator_category,
7                     std::forward_iterator_tag>;
8 }

```

这样, `ForwardIterator<T>` 包含了 `InputIterator<T>`, 现在可以重载两个迭代器类别的约束模板。

E.4 实用技巧

尽管 C++ 的概念已经研究了很多年, 并且实验性的实现已经出现了十多年, 但广泛的使用才刚刚开始。我们希望本书的未来版本能够提供关于如何设计受约束模板库的实用指南, 我们先给出了三个观察结果。

E.4.1 测试概念

概念是布尔谓词，有效的常量表达式。给定一个概念 C 和一些类型 T1, T2, ... 模型的概念，可以静态地断言观察：

```
1 static_assert(C<T1, T2, ...>, "Model failure");
```

设计概念时，建议也设计以这种方式测试其简单类型。包括挑战概念界限的类型，则需要回答如下问题：

- 接口和/或算法需要复制和/或移动建模类型的对象吗？
- 哪些转换是可以接受的？需要哪些转换？
- 模板假定的基本操作集唯一么？例如，可以使用 *= 或 * 和 = 操作吗？

这里，了解概念的原型（参见 28.3 节）也很有用。

E.4.2 概念的粒度

随着概念成为 C++ 语言的一部分，就可以对“概念库”进行构建了，就像我们在这些特性可用时构建类库和模板库一样。与其他库一样，我们也很希望以各种方式对概念进行分层。这里，简要地讨论了迭代器类别的例子，假设可以在这些类别之外构建“范围类别”，或者在这些类别之上构建“序列概念”等。

另一方面，会试图在“基本语法”概念的基础上构建所有这些概念。例如：

```
1 template<typename T, typename U>
2 concept Addable =
3     requires (T x, U y) {
4         x + y;
5     }
```

不建议这样做，因为这是一个没有明确语义的概念。当 T 和 U 都是 std::string 或者当一个类型是指针而另一个是整型，当然还有算术类型时，概念条件就满足了。在这三种情况下，可添加的概念有一些不同的含义（分别是连接、迭代器位移和算术加法）。因此，引入这样的概念将导致库接口模糊，并可能引起歧义。

相反，概念似乎最适合用于建模问题领域中出现的语义概念。以一种纪律性的方式来做这件事，会改善库的整体设计，因为将给使用者带来一致和明确的接口。当标准模板库（STL）添加到 C++ 标准库中时，情况就是如此。尽管它没有使用基于语言的“概念”，但在设计时在很大程度上考虑了概念的思想（如迭代器和迭代器层次结构），其余的都已成为历史了。

E.4.3 二进制兼容性

资深 C++ 开发者知道，当某些实体（特别是函数和成员函数）会在编译为低层机器码时，相关联的名称会将声明名称与实体类型和作用范围相结合。这个名称，通常称为实体的重组名称，是实际为链接器提供实体的引用（例如，来自其他对象文件）的名称。例如，定义为的函数的重组名称

```
1 namespace X {
2     void f() {}
3 }
```

使用 EvItanium C++ ABI [ItaniumABI] 时是 `_ZN1X1f`, 其中的字母 X 和 f 分别来自命名空间名和函数名。

混乱的名称不能在程序中“冲突”, 若两个函数可能在一个程序中共存, 那必须具有不同的、混乱的名称。反之, 约束必须在函数名中编码(因为模板特化除了约束和函数体之外, 其他方面都是相同的, 可以出现在不同的翻译单元中)。考虑以下两个翻译单元:

```
1 #include <iostream>
2
3 template<typename T>
4 concept HasPlus = requires (T x, T y) {
5     x+y;
6 };
7
8 template<typename T> int f(T p) requires HasPlus<T> {
9     std::cout << "TU1\n";
10 }
11
12 void g();
13
14 int main() {
15     f(1);
16     g();
17 }
```

和

```
1 #include <iostream>
2
3 template<typename T>
4 concept HasMult = requires (T x, T y) {
5     x*y;
6 };
7
8 template<typename T> int f(T p) requires HasMult<T> {
9     std::cout << "TU2\n";
10 }
11
12 template int f(int);
13
14 void g() {
15     f(2);
16 }
```

程序必须输出

TU1

TU2

这意味着 `f()` 的两个定义必须以不同的方式处理。

GCC 7.1 中对概念的实现实现在这方面有缺陷。

文献表

这个参考书目列出了本书中提到、采用或引用的资源。如今，许多编程的进步都在线上论坛上。因此，除了更传统的书籍和文章之外，发现相当多的 Web 站点也就不足为奇了。我们并不认为这里的清单很全面，不过这些资源的确是与 C++ 模板主题相关的。

Web 站点通常比书籍和文章更加不稳定。这里列出的互联网链接将来可能无效。因此，在以下网站提供了这本书的实际链接列表 (希望这个网站是稳定的):

<http://www.templbook.com>

列出书籍、文章和 Web 站点之前，先介绍新闻组提供的更具互动性的资源。

论坛

本书的第一版中，我们将 Usenet 组 (前万维网在线论坛的大集合) 作为关于 C++ 编程语言讨论的来源。从那时起，这些群体大多已经消失，但许多其他的在线编程社区已经兴起，其中有几个是为 C++ 开发者服务的。我们在这里列出了一些最受欢迎的:

- 关于 C 和 C++(各种语言) 的参考信息的 Cppreference “wiki” (即，集体编辑)。

<http://www.cppreference.com>

- Stackoverflow 是一个广泛的开发者社区，特别涵盖了 C++ 和 C++ 模板。

<https://stackoverflow.com/questions/tagged/c%2b%2b>

<https://stackoverflow.com/questions/tagged/c%2b%2b%20templates>

- Quora 类似于 Stackoverflow，但不限于技术讨论。

<https://www.quora.com/topic/C%2B%2B-programming-language>

- 标准 C++ 基金会是一个非营利组织，由 C++ 标准化委员会的一些杰出成员 (尽管这两个组织是独立的) 运营，以支持 C++ 编程社区。有助于资助标准化委员会的某些方面的会议，以及 CppCon(一个关于 C++ 的主要年度会议)(如果喜欢，强烈推荐读者参加其中)。还包括一个在线论坛目录 (以“谷歌组”的形式托管)，这些论坛涵盖了各种 C++ 主题。

<https://isocpp.org/forums>

<https://cppcon.org>

- C 和 C++ 用户协会 (ACCU) 是一个位于英国的组织，面向“对开发和提高编程技能感兴趣的人”。每年会举办一次编程会议，特别关注于 C++。

<https://www.accu.org>

图书和网站

[AbrahamsGurtovoyMeta]

David Abrahams 和 Aleksey Gurtovoy

C++ 模板元编程——Boost 和 Beyond 的概念、工具和技术

Addison-Wesley, Boston, MA, 2005

[AlexandrescuDesign]

Andrei Alexandrescu

现代 C++ 设计-通用编程和设计模式的应用

Addison-Wesley, Boston, MA, 2001

[AlexandrescuDiscriminatedUnions]

Andrei Alexandrescu

可辨别联合 (第 I、II、III 部分)

C/C++ Users Journal, April/June/August, 2002

[AlexandrescuAdHocVisitor]

Andrei Alexandrescu

泛型编程: 类型列表和应用程序

Dr. Dobb's Journal, February, 2002

[AusternSTL]

Matthew H. Austern

泛型编程和 STL——使用和扩展 C++ 标准模板库

Addison-Wesley, Boston, MA, 1999

[BartonNackman]

John J. Barton and Lee R. Nackman

科学与工程 C++ ——高级技术和实例介绍

Addison-Wesley, Boston, MA, 1994

[BCCL]

Jeremy Siek

Boost 概念检查库

http://www.boost.org/libs/concept_check/concept_check.htm

[Blitz++]

Todd Veldhuizen

Blitz++: 面向对象的科学计算

<http://blitz.sourceforge.net/>

[Boost]

免费的 Boost 库, C++ 库

<http://www.boost.org>

[BoostAny]

Kevlin Henney

Boost Any 库

<http://www.boost.org/libs/any>

[BoostFusion]

Joel de Guzman, Dan Marsden, and Tobias Schwinger

Boost Fusion 库

<http://boost.org/libs/fusion>

[BoostHana]

Louis Dionne

Boost Hana 元编程库

<http://boostorg.github.io/hana>

[BoostIterator]

David Abrahams, Jeremy Siek, Thomas Witt

Boost 迭代器

<http://www.boost.org/libs/iterator>

[BoostMPL]

Aleksey Gurtovoy 和 David Abrahams

Boost MPL

<http://www.boost.org/libs/mpl>

[BoostOperators]

David Abrahams

Boost 操作符

<http://www.boost.org/libs/utility/operators.htm>

[BoostTuple]

Jaakko Järvi

Boost 元组库

<http://boost.org/libs/tuple>

[BoostOptional]

Fernando Luis Cacciola Carballal

Boost Optional 库

<http://www.boost.org/libs/optional>

[BoostSmartPtr]

智能指针库

http://www.boost.org/libs/smart_ptr

[BoostTypeTraits]

类型特性库

http://www.boost.org/libs/type_traits

[BoostVariant]

Eric Friedman 和 Itay Maman

Boost Variant 库

<http://www.boost.org/libs/variant>

[BrownSIunits]

Walter E. Brown

单元计算 SI 库简介

<http://lss.fnal.gov/archive/1998/conf/Conf-98-328.pdf>

[C++98]

ISO

C++ 编程语言的标准

ISO/IEC, Document Number 14882-1998, 1998

[C++03]

ISO

C++ 编程语言的标准

ISO/IEC, Document Number 14882-2003, 2003

[C++11]

ISO

C++ 编程语言的标准

ISO/IEC, Document Number 14882-2011, 2011

[C++14]

ISO

C++ 编程语言的标准

ISO/IEC, Document Number 14882-2014, 2014

[C++17]

ISO

C++ 编程语言的标准

ISO/IEC, Document Number 14882-2017, 2017

[CacciolaKrzemienski2013]

Fernando Luis Cacciola Carballal 和 Andrzej Krzemieński

添加实用程序类来表示可选对象的建议

<http://wg21.link/n3527>

[CargillExceptionSafety]

Tom Cargill

异常处理: 错误的安全感

C++ Report, November-December 1994

[CoplienCRTP]

James O. Coplien

奇异递归模板模式

C++ Report, February 1995

[CoreIssue1395]

C++ 标准核心问题 1395 期

<http://wg21.link/cwg1395>

[CzarneckiEiseneckerGenProg]

Krzysztof Czarnecki 和 Ulrich W. Eisenecker

生成编程——方法、工具和应用

Addison-Wesley, Boston, MA, 2000

[DesignPatternsGoF]

Erich Gamma, Richard Helm, Ralph Johnson, 和 John Vlissides

设计模式——可重用面向对象软件

Addison-Wesley, Boston, MA, 1995

[DosReisMarcusAliasTemplates]

Gabrial Dos Reis and Mat Marcus

C++ 中添加模板别名的建议

<http://wg21.link/n1449>

[EDG]

Edison Design Group
编译器前端的 OEM 市场
<http://www.edg.com>

[EiseneckerBlinnCzarnecki]

Ulrich W. Eisenecker, Frank Blinn, 和 Krzysztof Czarnecki
基于 Mixin 的 C++ 编程
Dr. Dobbs Journal, January, 2001

[EllisStroustrupARM]

Margaret A. Ellis 和 Bjarne Stroustrup
The Annotated C++ Reference Manual (ARM)
Addison-Wesley, Boston, MA, 1990

[GregorJarviPowellVariadicTemplates]

Douglas Gregor, Jaakko Järvi, 和 Gary Powell
可变参数模板
<http://wg21.link/n2080>

[HenneyValuedConversions]

Kevlin Henney
值的转换
C++ Report 12(7), July-August 2000

[OverloadingProperties]

Jaakko Järvi, Jeremiah Willcock, Howard Hinnant, 和 Andrew Lumsdaine
基于类型属性的函数重载
C/C++ Users Journal 12 (6), June, 2003

[ItaniumABI]

Itanium C++ ABI
<http://itanium-cxx-abi.github.io/cxx-abi/>

[JosuttisLaunder]

Nicolai Josuttis
On launder()
<https://wg21.link/p0532r0>

[JosuttisStdLib]

Nicolai M. Josuttis
C++ 标准库——教程和参考 (第 2 版)
Addison-Wesley, Boston, MA, 2012

[KarlssonSafeBool]

Bjorn Karlsson
安全 Bool 习语
C++ Source, July, 2004

[KoenigMooAcc]

Andrew Koenig 和 Barbara E. Moo
加速 C++ —— 实例编程
Addison-Wesley, Boston, MA, 2000

[LambdaLib]

Jaakko Järvi and Gary Powell
LL, The Lambda Library
<http://www.boost.org/libs/lambda>

[LibIssue181]

C++ 库问题 181
<http://wg21.link/lwg181>

[LippmanObjMod]

Stanley B. Lippman
C++ 对象模型内部
Addison-Wesley, Boston, MA, 1996

[MeyersCounting]

Scott Meyers
C++ 中的计数对象
C/C++ Users Journal, April 1998

[MeyersEffective]

Scott Meyers
高效 C++ —— 50 种改进程序和设计的具体方法 (第二版)
Addison-Wesley, Boston, MA, 1998

[MeyersMoreEffective]

Scott Meyers

更高效的 C++ —— 35 种改进程序和设计的新方法

Addison-Wesley, Boston, MA, 1996

[MoonFlavors]

David A. Moon

使用 Flavor 进行面向对象编程

Conference proceedings on Object-oriented programming systems, languages and applications, 1986

[MTL]

Andrew Lumsdaine 和 Jeremy Siek

MTL, 矩阵模板库

<http://www.osl.iu.edu/research/mtl>

[MusserWangDynaVeri]

D. R. Musser 和 C. Wang

C++ 泛型算法的动态验证

IEEE Transactions on Software Engineering, Vol. 23, No. 5, May 1997

[MyersTraits]

Nathan C. Myers

特性: 全新的和有用的模板技术

<http://www.cantrip.org/traits.html>

[NewMat]

Robert Davies

NewMat10, C++ 中的一个矩阵库

http://www.robertnz.net/nm_intro.htm

[NewShorterOED]

Leslie Brown (Ed.)

新版简明牛津英语词典 (第四版)

Oxford University Press, Oxford, 1993

[POOMA]

POOMA: 一个用于并行科学计算的高性能 C++ 工具包

<http://www.nongnu.org/freepooma/>

[SmaragdakisBatoryMixins]

Yannis Smaragdakis 和 Don S. Batory

基于 Mixin 的 C++ 编程

第二届生成与构件软件国际研讨会论文集

Engineering, October, 2000

[SpicerSFINAE]

John Spicer

Solving the SFINAE Problem for Expressions

<http://wg21.link/n2634>

[StepanovLeeSTL]

Alexander Stepanov 和 Meng Lee

标准模板库——惠普实验室技术报告 95-11(R.1)

November 14, 1995

[StepanovNotes]

Alexander Stepanov

编程笔记

<http://stepanovpapers.com/notes.pdf>

[StroustrupC++PL]

Bjarne Stroustrup

C++ 程序设计语言 (特别版)

Addison-Wesley, Boston, MA, 2000

[StroustrupDnE]

Bjarne Stroustrup

C++ 的设计与发展

Addison-Wesley, Boston, MA, 1994

[StroustrupGlossary]

Bjarne Stroustrup

Bjarne Stroustrup 的 C++ 术语表

<http://www.stroustrup.com/glossary.html>

[SutterExceptional]

Herb Sutter

特殊的 C++ ——47 个工程难题，编程问题和解决方案
Addison-Wesley, Boston, MA, 2000

[SutterMoreExceptional]

Herb Sutter

更特殊的 C++ ——40 个新的工程难题，编程问题和解决方案
Addison-Wesley, Boston, MA, 2001

[UnruhPrimeOrig]

Erwin Unruh

原始元程序的质数计算

<http://www.erwin-unruh.de/primorig.html>

[VandevoordeJosuttisTemplates1st]

David Vandevoorde and Nicolai M. Josuttis

C++ 模板: 完整指南

Addison-Wesley, Boston, MA, 2003

[VandevoordeSolutions]

David Vandevoorde

C++ 的解决方案

Addison-Wesley, Boston, MA, 1998

[VeldhuizenMeta95]

Todd Veldhuizen

使用 C++ 模板进行元程序

C++ Report, May 1995

术语表

这个术语表是本书中使用的术语汇总。参见 [StroustrupGlossary] 了解 C++ 程序员使用的全面、通用的术语表。

抽象类 (abstract class)

不可能创建具体对象 (实例) 的类。抽象类可以用于在单一类型中收集不同类的公共属性，或者定义多态接口。因为抽象类可作为基类，所以缩写 ABC 有时用来表示抽象基类。

ADL

参数相关查找的缩写。ADL 是一个进程，在命名空间和类中查找函数 (或操作符) 名称，这些命名空间和类以某种方式与出现该函数 (或操作符名称) 的函数调用的参数相关联。由于历史原因，有时称为扩展 Koenig 查找，或是 Koenig 查找 (后者也用于仅应用于操作符的 ADL)。

别名模板 (alias template)

表示一组类型别名的构造，指定了一个模式，通过用特定的实体替换模板参数，可以从该模式生成实际的类型别名。别名模板可以是类成员。

尖括号黑客 (angle bracket hack)

一种 C++ 特性，要求编译器接受两个连续的右尖括号字符作为两个结束尖括号，尖括号黑客会使 `vector<list<int>>` 与 `vector<list<int>>` 以相同地方式处理。其称为 (词法) 黑客，因为它不适合 C++ 的正式规范 (尤其是语法)，也不适合典型编译器的架构。另一个类似的黑客处理形成临时有向图 (见有向图)。

尖括号 (angle brackets)

字符 < 和 > 用作分隔符，而不是用作小于和大于操作符。

ANSI

美国国家标准协会 (American National Standard Institute) 的首字母缩写，该协会是一家私人非营利组织，负责协调各种标准规范的制定工作。参见 INCITS。

参数 (argument)

替代编程实体的参数的值 (广义上)。在函数 `abs(-3)` 调用中，参数是 -3。一些编程社区中，参数称为实际参数 (而声明参数称为形式参数)。参见 模板参数。

参数依赖查询 (argument-dependent lookup)

参见 ADL。

类 (class)

对一类对象的描述,类为该类型的对象定义了一组特征。这包括数据(属性、数据成员)以及操作(方法、成员函数)。C++中,类是具有成员(也可以是函数)的结构,并且受到访问限制,使用关键字class或struct声明。

类模板(class template)

表示类集的构造。其指定了一个模式,通过用特定的类型替换模板参数,可以从该模式生成实际的类。类模板有时称为参数化类。

类类型(class type)

用class、struct或union声明的C++类型。

集合类(collection class)

用于管理一组对象的类。在C++中,集合类也称为容器。

编译器(compiler)

将翻译单元中的源代码转换为目标代码的程序或库组件(带有符号注释的机器代码,允许链接器跨翻译单元解析引用)。

完整的类型(complete type)

已定义的类、包含完整元素和已知大小的数组、具有已定义底层类型的枚举类型,以及除void(可选使用const和/或volatile)之外的基本数据类型。

概念(concept)

可应用于一个或多个模板参数的命名约束集。请参阅附录E。

常量表达式(constant-expression)

编译器可以在编译时计算表达式的值。有时称它为真常量,以避免与常量表达式(没有连字符)混淆。后者包含常量表达式,但编译器在编译时不一定会计算这些表达式。

常量成员函数(const member function)

可以为常量和临时对象调用的成员函数,通常不修改*this对象的成员。

容器(container)

参见集合类(collection class)。

转换函数(conversion function)

特殊的成员函数,定义了如何隐式(或显式)的将对象转换为另一类型的对象,使用函数操作符的形式声明。

转换操作符 (conversion operator)

转换函数的同义词。后者是标准术语，但前者也常用。

CPP 文件 (CPP file)

存放变量和非内联函数定义的文件。程序的大多数可执行 (与声明性相反) 代码通常放在 CPP 文件中，之所以命名为 CPP 文件，是因为通常以.CPP 后缀命名。但由于历史原因，后缀也可以是.C、.c、.cc 或.cxx。参见头文件和翻译单元。

奇异递归模板模式 (CRTP)

奇异递归模板模式的缩写。这指的是一种代码模式，其中类 X 派生自具有 X 作为模板参数的基类。

奇异递归模板模式 (curiously recurring template pattern)

参见 CRTP。

衰变 (decay)

数组或函数到指针的隐式转换。字符串字面值”Hello” 具有 char 类型 const[6]，但在许多 C++ 中，会隐式转换为 char 类型 const* 指针 (指向字符串的第一个字符)。

声明 (declaration)

C++ 作用域中引入或重新引入名称的 C++ 构造。参见定义 (definition)。

推导 (deduction)

使用模板隐式确定模板参数的过程，完整的术语是模板参数推导。

定义 (definition)

一种形式的声明，使已声明的实体为人所知。若是变量，则强制为已声明实体保留存储空间。对于类类型和函数定义，这相当于包含大括号括起来的主体的声明。对于外部变量声明，要么声明没有 extern 关键字，要么声明有初始化式。

依赖基类 (dependent base class)

依赖于模板参数的基类。必须特别注意访问从属基类的成员。另请参见两阶段查找。

依赖名称 (dependent name)

其含义依赖于模板参数的名称。当 A 或 T 是模板参数时， $A<T>::x$ 是一个依赖名称。若函数调用中的实参类型依赖于模板形参，那么函数调用中的函数名也依赖于模板形参。例如， $f((T*)0)$ 中的 f 是依赖于 T 是模板参数的。但是，模板参数的名称不是依赖，请参见两阶段查找。

有向图 (digraph)

C++ 代码中相当于另一个字符的两个连续字符的组合。有向图的目的是用缺少某些字符的键盘输入 C++ 源代码。虽然很少使用，但当左尖括号后面跟着一个范围解析操作符 (::) 而没有必要的中间空格时，有时会意外地形成有向图 <:。C++11 引入了一个词法黑客，在这种情况下禁用有向图解释。

空基类优化 (EBCO)

由大多数现代编译器执行的优化，其中“空”基类子对象不占用存储空间。

空基类优化 (empty base class optimization)

参见 EBCO。

显式实例化指令 (explicit instantiation directive)

一个 C++ 构造，其唯一目的是创建一个实例化点 (POI)。

显式特化 (explicit specialization)

为替换模板声明或定义替代定义的构造，原始(通用)模板称为主模板。若替代定义仍然依赖于一个或多个模板参数，则称为偏特化。否则，就是全特化。

表达式模板 (expression template)

用来表示表达式一部分的类模板，模板本身表示一种特定的操作。模板参数表示操作所应用的操作数类型。

转发引用 (forwarding reference)

T&& 的右值引用的两个术语之一，其中 T 是一个可推导的模板参数，适用不同于普通右值引用的特殊规则(参见第 6.1 节)。这个术语由 C++17 引入，因为引用的主要用途是转发对象，所以作为通用引用的替代，但它不会自动转发。这个术语并不描述它具体是什么，而是用于做什么。

友元名称注入 (friend name injection)

当函数名的唯一声明是友元声明时，使函数名可见的过程。

全特化 (full specialization)

参见显示特化 (explicit specialization)。

函数对象 (function object)

可以使用函数调用语法调用的对象。C++ 中，这些指针是指向函数的指针、带有重载函数操作符的类(参见 functor)，以及带有转换函数的类，这些转换函数产生指向函数的指针或指向函数的引用。

函数模板 (function template)

表示函数族的构造，其指定了一个模式，通过使用特定的参数替换模板参数，可以从该模式生成实际的函数。函数模板是模板，不是函数。函数模板有时称为参数化函数。

函子/仿函数 (functor)

具有重载函数操作符的类类型对象，可以使用函数调用的方式调用它。包括 Lambda 表达式的闭包类型。

广义可本地化值 (glvalue)

为存储值(广义可本地化值)生成位置的一类表达式，glvalue 可以是左值或 xvalue。参见值类别章节和 B.2 节。

头文件 (header file)

通过 #include 指令成为翻译单元的一部分的文件。这些文件通常包含从多个翻译单元引用的变量和函数的声明，以及类型、内联函数、模板、常量和宏的定义。通常以.hpp、.h、.H 或.hxx 等后缀命名，也称为包含文件。参见 CPP 文件和翻译单元。

国际信息技术标准委员会 (INCITS)

国际信息技术标准委员会 (InterNational Committee for Information Technology Standards) 是 ANSI 认可的美国标准开发组织 (原名 X3) 的缩写。名为 J16 的小组委员会是 C++ 标准化背后的驱动力，它与国际标准化组织 (ISO) 合作密切。

包含文件 (include file)

参见头文件 (header file)。

不完整类型 (incomplete type)

声明但未定义的类、元素类型不完整或大小未知的数组、未定义底层类型的枚举类型或 void(可选使用 const 和/或 volatile)。

间接调用 (indirect call)

在调用实际发生之前(在运行时)，调用的函数是未知的。

初始化式 (initializer)

指定如何初始化对象的构造。例如，初始化式为 = 1.0 和 (0.0,1.0)

```
std::complex<float> z1 = 1.0, z2(0.0, 1.0);
```

初始化列表 (initializer list)

用大括号括起来的以逗号分隔的表达式列表，用于初始化对象和引用。初始化列表通常用于初始化变量，也用于(例如) 初始化构造函数定义中的成员和基类。初始化可以直接进行，也可以通过 std::initializer_list 对象进行。

类名称注入 (injected class name)

类的称在它自己的定义范围内可见。对于类模板，如果模板名后面没有模板参数列表，则模板名在模板的作用域内作为类名。

实例 (instance)

实例一词在 C++ 编程中有两层含义。源自面向对象术语的含义是类的实例：是类的实现的对象，C++ 中 `std::cout` 是 `std::ostream` 类的一个实例。另一种含义是模板实例：通过用特定值替换所有模板参数获得的类、函数或成员函数。所以，实例也称为特化，尽管后一个术语经常误认为是显式特化。

实例化 (instantiation)

在模板定义中替换模板参数，以创建具体实体（函数、类、变量或别名）。若只替换模板的声明而不替换模板的定义，则有时会使用术语“部分模板实例化”。参见替换。创建类的实例（对象）的另一种含义在本书中没有提及（参见 `instance`）。

国际标准化组织 (ISO)

国际标准化组织的全球首字母缩写，名为 WG21 的 ISO 工作组是标准化和开发 C++ 的幕后推手。

迭代器 (iterator)

了解如何遍历元素序列的对象，这些元素属于一个集合（参见集合类）。

可链接的对象 (linkable entity)

以下任何一种：函数或成员函数、全局变量或静态数据成员，包括从模板生成的，并对连接器可见。

连接器 (linker)

一种程序或操作系统服务，将已编译的翻译单元链接在一起，并跨翻译单元对可链接的实体进行引用和解析。

左值 (lvalue)

一种表达式的类别，为假定不可移动的存储值（即非 `xvalue` 的 `glvalues`）产生一个地址。典型的例子是表示命名对象（变量或成员）和字符串字面符的表达式。参见的值类别章节和 B.1 节。

成员类模板 (member class template)

一个表示成员类族的类型，在另一个类或类模板定义中声明的类模板。并有自己的一组模板参数（不像类模板的成员类）。

成员函数模板 (member function template)

表示一组成员函数的构造，有自己的一组模板参数形参(与类模板的成员函数不同)。非常类似于函数模板，但当替换所有模板参数时，结果是一个成员函数(而不是普通函数)。成员函数模板不能为虚模板。

成员模板 (**member template**)

成员类模板、成员函数模板或静态数据成员模板。

现代 C++(**Modern C++**)

本书中指的是 C++11 或更高版本(即 C++11、C++14 或 C++17) 的标准化语言。

不相关名称 (**nondependent name**)

不依赖于模板参数的名称。参见依赖名称和两阶段查找。

单一定义规则 (**ODR**)

单一定义规则的首字母缩略词，该规则对 C++ 程序中出现的定义施加了一些限制。详细信息请参见第 10.4 节和附录 A。

单一定义规则 (**one-definition rule**)

参见单一定义规则 (ODR)。

重载解析 (**overload resolution**)

当存在多个候选函数(通常都有相同的名称)时，描述了选择调用哪个函数的过程。参见附录 C。

参数 (**parameter**)

占位符实体，某些时候会使用实际“值”(参数)替换。对于宏参数和模板参数，替换发生在编译时。对于函数调用参数，发生在运行时。在一些编程社区中，参数会称为形式参数(而实参称为实际参数)。参见参数和模板参数。

参数化类 (**parameterized class**)

类模板或嵌套在类模板中的类。因为在指定模板参数之前，都不对应于唯一的类，所以这两个类都是参数化的。

参数化函数 (**parameterized function**)

一个函数或成员函数模板或类模板的成员函数。因为在指定模板实参之前，并不对应于唯一的函数(或成员函数)，所以所有函数都是参数化的。

偏特化 (**partial specialization**)

为模板的某些替换声明或定义替代定义的构造，原始(通用)模板称为主模板，另一种定义仍然依赖于模板参数，这个构造只存在于类模板中。另请参见显式特化。

普通的旧数据 (POD)

“普通旧数据 (类型)”的缩写。POD 类型可以在没有某些 C++ 特性 (如虚成员函数、访问关键字等) 的情况下定义，每个普通的 C 结构体都是一个 POD。

实例化点 (POI)

实例化点的缩写。POI 是源代码中的一个位置，模板 (或模板的成员) 在这里通过用模板参数替换模板参数，从而在概念上展开。实践中，这种扩展并不需要在每个 POI 中都发生。另请参见显式实例化指令。

实例化点 (point of instantiation)

参见实例化点 (POI)。

策略类 (policy class)

类或类模板，其成员描述泛型组件的可配置行为。策略通常作为模板参数传递，排序模板可能有一个排序策略，策略类也称为策略模板或策略。参见特性模板。

多态性 (polymorphism)

操作 (由其名称确定) 应用于不同类型对象的能力。C++ 中，传统的面向对象的多态性概念 (也称为运行时或动态多态性) 是通过在派生类中重写的虚函数实现的。另外，C++ 模板也支持静态多态性。

预编译头文件 (precompiled header)

编译器可以快速加载的一种经过处理的源代码形式。预编译头文件的源代码必须是翻译单元的第一部分 (不能从翻译单元中间的某个地方开始)，预编译头文件对应许多头文件。使用预编译头文件可以大大减少构建用 C++ 编写的大型应用程序所需的时间。

主模板 (primary template)

非偏特化的模板。

纯右值 (prvalue)

执行初始化的一类表达式。预值可以假定为指定纯数值，如 1 或 true 和临时值 (特别是由值返回的值)。C++11 前的右值在，C++11 中都是纯右值。参见值类别和 B.2 节。

限定名 (qualified name)

包含作用域限定符 (::) 的名称。

引用计数 (reference counting)

一种资源管理策略，记录有多少实体引用了特定的资源。当计数减到 0 时，可以回收资源。

右值 (rvalue)

不是左值的一类表达式。右值可以是 prvalue(例如临时值) 或 xvalue(例如，用 std::move() 标记的左值)。C++11 之前的右值，C++11 中为纯右值。参见 674 页的值类别和 B.2 节。

替换失败不为过 (SFINAE)

替换失败不为过的缩写。当以无效的方式替换模板参数时，一种静默丢弃模板的机制，而不是触发编译错误。弱替换成功，重载集中的其他模板就有机会选中。

源文件 (source file)

头文件或 CPP 文件。

特化 (specialization)

用实际值替换模板参数的结果。特化可以通过实例化创建，也可以通过显式特化创建。这个术语有时会错误地等同于显式特化。参见实例。

静态数据成员模板 (static data member template)

作为类或类模板成员的变量模板。

替换 (substitution)

将模板实体中的模板参数替换为实际类型、值或模板的过程。替换的程度取决于上下文，在重载解析期间，只执行建立候选函数类型的最小替换，若该替换导致无效构造，则应用 SFINAE 规则。参见实例化。

模板 (template)

表示一系列类型、函数、成员函数或变量的方式，指定了一个模式，通过使用特定的实体替换模板参数，可以从该模式生成实际的类型、函数、成员函数或变量。本书中，这个术语不包括函数、类、静态数据成员和仅作为类模板成员而参数化的类型别名。请参见别名模板、变量模板、类模板、参数化类、函数模板和参数化函数。

模板参数 (template argument)

替换模板参数的“值”。这个值通常是一个类型，某些常量值和模板也可以是有效的模板参数。参见参数。

模板参数推导 (template argument deduction)

参见推导 (deduction)。

模板标识 (template-id)

模板名后跟尖括号中指定的模板参数的组合 (例如，std::list<int>)。

模板参数 (template parameter)

模板中的通用占位符。最常见的模板参数是类型参数。非类型参数表示某一类型的常量值，模板模板参数表示类型模板。参见参数。

模板化实体 (templated entity)

模板中定义或创建的模板或实体。后者包括类模板的普通成员函数或模板中出现的 Lambda 闭包类型。

特征模板 (traits template)

一种类模板，其成员描述模板参数的特征。通常，特征模板的目的是避免模板参数过多。请参见策略类。

翻译单元 (translation unit)

包含所有头文件和标准库头文件的 CPP 文件，使用 #include 指令，减去条件编译指令 (如 #if) 排除的程序文本，也可以认为是对 CPP 文件进行预处理的结果。参见 CPP 文件和头文件。

真常量 (true constant)

编译器可以在编译时计算其值的表达式。参见常数表达式。

元组 (tuple)

C 结构体概念的泛化，成员可以通过编号访问。

两阶段查找 (two-phase lookup)

用于模板中名称的名称查找机制。这两个阶段是 (1) 处理模板定义，(2) 为特定的模板参数实例化模板。非依赖名称只在第一阶段查找，在该阶段不考虑非依赖基类。具有作用域限定符 (::) 的相关名称只在第二阶段查找。在两个阶段中都可以查找没有作用域限定符的相关名称，但在第二个阶段中，只执行与参数相关的查找。

类型别名 (type alias)

类型的另一种名称，使用 typedef 声明、别名声明或别名模板的实例化引入。

类型模板 (type template)

类模板、成员类模板或别名模板。

通用引用 (universal reference)

T&& 右值引用的两个术语之一，其中 T 是一个可推导的模板参数。适用不同于普通右值引用的特殊规则 (参见第 6.1 节)。这个术语是由 Scott Meyers 创造的，作为左值引用和右值引用的共同术语。因为“通用”太通用了，所以在 C++17 中引入了转发引用。

用户定义的转换 (user-defined conversion)

由开发者定义的类型转换，可以是用一个参数或转换函数调用的构造函数。除非构造函数或转换函数使用关键字 `explicit` 声明，否则可以隐式地进行类型转换。

值类别 (value category)

表达式的分类。传统的值类别左值和右值是从 C 继承而来的。C++11 引入了替代类别:`glvalue`(广义左值)，其求值标识存储的对象，以及 `prvalue`(纯右值)，其求值初始化对象。其他类别将 `glvalue` 细分为 `lvalue`(可本地化值) 和 `xvalue`(过期值)。在 C++11 中，右值作为 `xvalue` 和 `prvalue` 的一般类别(C++11 之前，右值就是 C++11 中的 `prvalue`)。详见附录 B。

变量模板 (variable template)

表示一组变量或静态数据成员的构造，指定了一种模式，通过使用特定的实体替换模板参数，可以从该模式生成实际变量和静态数据成员。

空白 (whitespace)

C++ 中，这是在源代码中分隔标记(标识符、文字、符号等)的空格。除了传统的空格、新行和制表字符外，还包括注释。其他空白字符(例如，页面提要控制字符)有时也是有效的。

过期值 (xvalue)

为存储的对象生成位置的一类表达式，该对象可以假定不再需要。一个典型的例子是用 `std::move()` 标记的左值。参见值类别和 B.2 节。