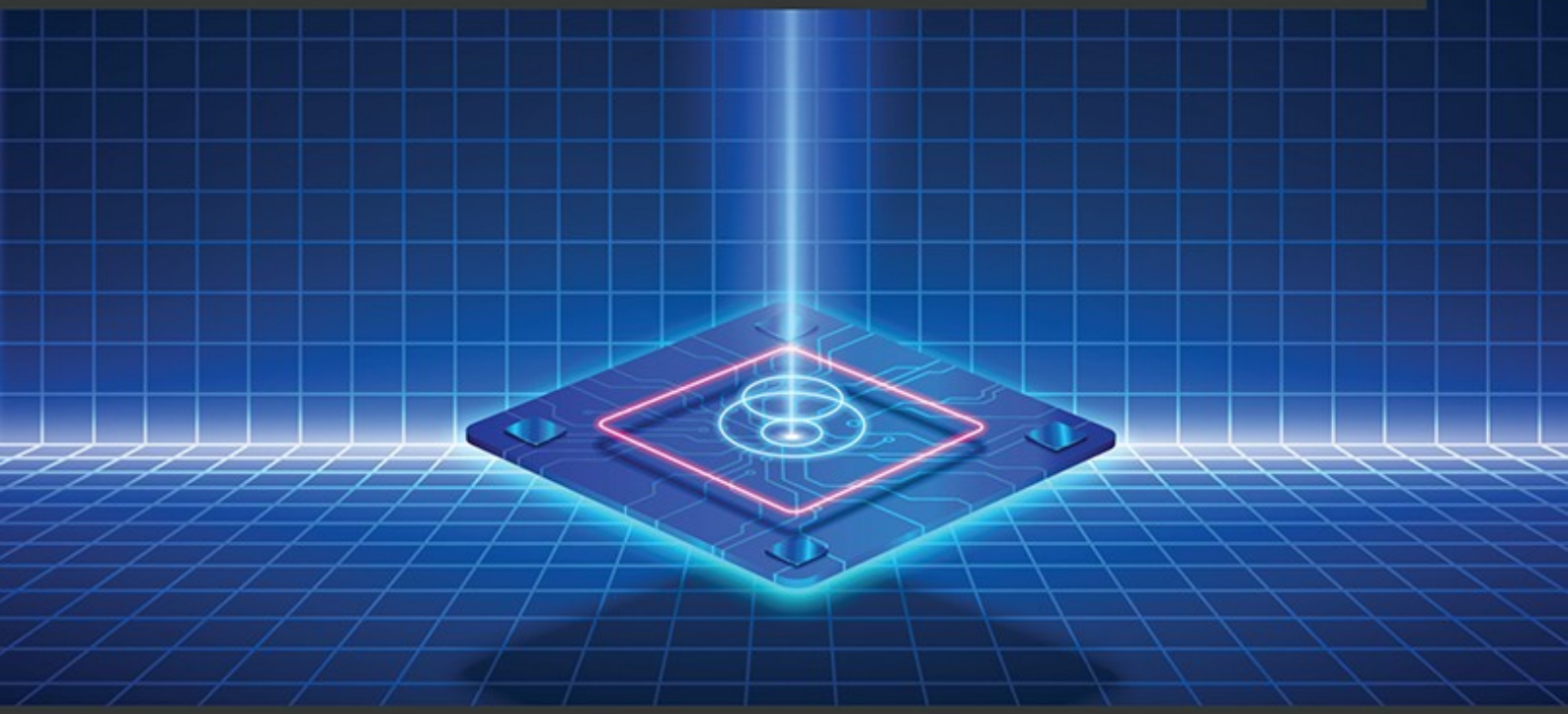
Template Metaprogramming with C++

Learn everything about C++ templates and unlock the power of template metaprogramming



Marius Bancila

Template Metaprogramming with C++

Learn everything about C++ templates and unlock the power of template metaprogramming

详细的了解 *C++* 模板，释放模板元编程的力量

作者: Marius Bancila

译者：陈晓伟

本书概述

了解元编程，可以创建在编译时进行计算的数据结构和函数。本书将带您了解如何使用模板避免编写重复的代码，以及其为什么会成为通用库 (如标准库或 Boost) 的关键 (通用库可用于很多程序中)。

本书将深入了解模板和元编程的基础知识，再练习编写复杂的模板，并探索高级概念，如模板递归、模板参数推导、转发引用、类型特征和条件编译。这个过程中，将了解如何编写可变参数模板，以及如何使用 C++20 约束和概念为模板参数进行限制。最后，可以使用 C++ 元编程模板来实现各种元编程模式。

本书的最后，将学习如何在日常编程中编写有效的模板和实现元编程。

关键特性

- 了解 C++20 的最新特性，并使用 STL 编写更好的代码
- 减少应用程序的开发时间，可以更快的进行部署
- 使用最新标准中引入的新而精简的 STL 功能

将会学到

- 类型模板的语法
- 特化和实例化如何工作
- 模板参数推断和转发引用
- 编写可变参数模板
- 熟悉类型特征和条件编译
- 用约束和概念限制 C++20 中的模板参数
- 实现 CRTP、Mixins 和标签分派等模式

作者简介

Marius Bancila 是一名软件工程师，为工业和金融部门开发解决方案方面，拥有近 20 年的经验。他是《现代 C++ 挑战》的作者和《学习 C# 编程》的合著者。并且是一名软件架构师，专注于 Microsoft 技术，主要使用 C++ 和 C# 开发桌面应用程序。热衷于分享其技术专长，自 2006 年以来，他一直是微软 C++ 和开发人员眼中的技术 MVP。Marius 住在罗马尼亚，并且活跃在各种网络社区。

审阅者简介

Aleksei Goriachikh 使用的 C++ 编程超过 8 年。Aleksei 于 2012 年从新西伯利亚州立大学 (Novosibirsk State University) 数学硕士学位毕业后，之后一直从事计算数学和优化、CAD 系统的几何核心设计，以及自动驾驶的多线程实现等领域的研究。Aleksei 目前的兴趣是流片前验证 (芯片相关)。

本书相关

- Github 地址：
  https://github.com/xiaoweiChen/Template-Metaprogramming-with-CPP

献给自己那“该死的”好奇心。

-*Marius Bancila*

前言

历经数十载，C++ 已经成为适用范围广泛的编程语言之一。它的成功不仅是因为其性能或易用性 (有人会反对)，而且是因为其的多功能性。C++ 是一种通用的、多范式的编程语言，混合了过程式、函数式和泛型编程。

泛型编程是一种编写代码的方式，例如：函数和类等实体为根据之后指定的类型编写。这些泛型实体仅在指参数指定为特定类型需要时才进行实例化，这些泛型实体在 C++ 中称为模板。

元编程是一种编程技术，使用模板 (和 C++ 中的 constexpr 函数) 可在编译时生成代码，然后与其余源代码合并，最终用于编译程序。元编程需要 (至少) 指定一个输入或输出的类型。

C++ 的模板可谓是声名狼藉，如同《C++ 核心指南》(Bjarne Stroustrup 和 Herb Sutter 维护的一份关于该做什么和不该做什么的文档) 中所描述的那样。不过，模板可用来实现泛型库，就如同 C++ 开发人员会一直使用的 C++ 标准库一样。无论是自己编写模板，还是使用他人编写的模板 (如标准容器或算法)，模板很可能是日常编码的一部分。

本书的目的是为让读者更加容易理解 C++ 模板 (从基本语法到 C++20 中的概念)，这是本书前两部分的重点内容。第三部分 (也是最后一部分) 将使用前面了解到的知识付诸实践，并使用模板进行元编程。

适读人群

这本书是为初学者到中级 C++ 开发人员 (想要学习和了解模板元编程的人员)，以及高级 C++ 开发人员 (希望了解与模板相关的新 C++20 特性和各种习语和模式)。阅读这本书之前，需要有基本的 C++ 编程经验。

本书内容

第 1 章，模板简介。通过几个简单的例子介绍 C++ 中模板元编程的概念，讨论为什么需要模板，以及模板的优缺点。

第 2 章，了解模板。探讨了 C++ 中所有形式的模板: 函数模板、类模板、变量模板和别名模板。每一个都会讨论其语法和工作方式等细节，这里还讨论了模板实例化和特化的关键概念。

第 3 章，可变参数模板。专注于可变参数模板，将详细讨论可变参数函数模板、可变参数类模板、可变参数别名模板、可变参数变量模板、参数包及其展开方式，以及如何编写可变参数模板的折叠表达式。

第 4 章，高级模板概念。汇集了一系列高级模板概念，如名称依赖和名称查找、模板参数推导、模板递归、完美转发、泛型和模板 Lambda。通过理解这些内容，读者将能够极大地扩展阅读或编写的模板的种类。

第 5 章，类型特征和条件编译。专注于讨论类型特征，将了解类型特征，标准库提供了哪些特征，以及如何使用它们解决不同的问题。

第 6 章，概念和约束。介绍了 C++20 的新机制，使用概念和约束定义模板参数的需求，可以了解指定约束的方法。此外，这里还概述了 C++20 标准概念库的内容。

第 7 章，模式和习语。探讨了一系列独立的高级主题，如何实现各种模式。我们将探讨静态多态性、类型擦除、标记分派和模式 (例如奇异迭代模板模式、表达式模板、混入和类型列表) 的概念。

*第 8 章，范围和算法。*容器、迭代器和算法是标准模板库的核心组件，将学习如何为其编写泛型容器和迭代器类型，以及通用算法。

*第 9 章，范围库。*探讨了 C++20 中新添加的范围库的关键特性，如范围、范围适配器和约束算法，可使我们能够编写更简单的代码来使用范围，还有如何编写自定义范围适配器。

附录是一个简短的结尾，对本书进行了总结。

习题解答包含了所有章节中所有习题的答案.

环境配置

阅读这本书时，需要具备基础的 C++ 编程知识。需要了解类、函数、操作符、函数重载、继承、虚函数等的语法和基础知识。若对模板不是很了解也没关系，因为本书将从零开始帮助读者了解模板。

本书中的所有代码示例都是跨平台的，可以使用任何编译器来构建和运行。但许多示例使用需要使用 C++11，但也有一些示例需要支持 C++17 或 C++20 兼容的编译器，建议您使用支持 C++20 的编译器版本，这样就可以运行本书所有示例。书中的示例都使用 MSVC 19.30 (Visual Studio 2022)、GCC 12.1/13 和 Clang 13/14 进行过测试。若机器上没有兼容 C++20 的编译器，可以在 Web 上试试。这里推荐几个平台:

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

本书将多次引用 C++ Insights 在线工具来分析编译器生成的代码。

若想了解编译器对不同 C++ 标准的支持程度，可以参考https://en.cppreference.com/w/cpp/compiler_support。

如果正在使用本书的数字版本，我们建议您自己输入代码或通过 GitHub 访问代码 (下一节提供链接)，将避免复制和粘贴代码。

扩展阅读

本书中，我们会多次提到 C++ 标准，其文件版权归国际标准化组织所有。官方的 C++ 标准文档可以从这里购买:https://www.iso.org/standard/79358.html。C++ 标准的多个草案，以及相应源码都可以在 GitHub 上的https://github.com/cplusplus/draft免费获得。并且，可以在https://isocpp.org/std/the-standard上找到关于 C++ 标准的更多信息。

cppreference 网站https://en.cppreference.com/，是一个很好的在线资源，提供了直接从 C++ 标准派生的 C++ 语言的详尽文档。本书多次引用了 cppreference 中的内容。cppreference 的内容基于 CC-BY-SA 协议，https://en.cppreference.com/w/Cppreference:Copyright/CC-BY-SA。

每一章的末尾的“扩展阅读”部分，包含了一份参考书目，相关条目中的具体内容，可用于加深对所介绍主题的理解。

下载例程

可以从 GitHub 上的https://github.com/PacktPublishing/Template-Metaprogramming-with-CPP下载这本书的示例代码文件。若代码有更新，将会体现在 GitHub 的库中。

我们还有其他的代码包，还有丰富的书籍和视频目录，都在https://github.com/PacktPublishing/。去看看吧!

下载彩图

我们还提供了一个 PDF 文件，其中有本书中使用的屏幕截图和图表的彩色图。可以在这里下载:https://packt.link/Un8j5。

目录

第一部分：核心概念

介绍模板并了解其优点，再了解编写函数模板、类模板、可变参模板和别名模板的语法。您将了解模板实例化和模板特化等概念，并学习如何使用可变数量的参数编写模板。

本部分包括以下章节:

- 第 *1* 章，模板简介
- 第 *2* 章，了解模板
- 第 *3* 章，可变参数模板

第 1 章 模板简介

作为 C++ 开发人员，应该熟悉**元编程**，也称为**模板**。模板/元编程是一种编程技术，其使用模板作为编译器生成代码的蓝图，并避免开发人员编写重复的代码。尽管通用库会大量使用模板，但 C++ 语言中模板的语法和内部工作方式可能令人望而生畏。C++ 语言的创始人 Bjarne Stroustrup 和 C++ 标准化委员会主席 Herb Sutter 编辑的《C++ 核心指南》是一套“该/不该做什么”的指南，并且这二位也认为模板非常可怕。

本书旨在让读者了解 C++ 语言的这一领域，并助您在模板元编程方面了解的更多。

在本章中，我们将讨论以下主题:

- 使用模板的动机
- 编写模板
- 理解模板
- 模板简史
- 模板的优缺点

了解如何使用模板的第一步，是了解其实际解决了什么问题。让我们从这里启程吧！

1.1. 使用模板的动机

每种语言特性都旨在帮助开发人员解决当前的问题或任务，模板的目的是为了避免编写只有细微差别的重复性代码。

这里，我们以一个经典的 max 函数为例。函数接受两个数值参数，并返回两个参数中最大的一个:

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

这个函数没毛病，但只适用于 int 类型的值 (或那些可转换为 int 的值)。若需要相同的函数，但参数类型是 double 怎么办呢？直接为 double 类型重载这个函数就好了 (创建一个具有相同名称，但参数数量或类型不同的函数):

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

然而，int 和 double 并不是唯一的数字类型。有 char, short, long, long 和相应的 unsigned 版本，unsigned char, unsigned short, unsigned long, unsigned long。还有浮点型和长双精度型，以及其他类型，如 int8_t, int16_t, int32_t 和 int64_t。可以比较其他类型，例如 bigint、Matrix、point2d 和任何支持大于运算符的自定义类型。通用库如何为这些类型提供诸如 max 这样的通用函数呢? 它可以重载所有内置类型和其他库类型的函数，但不能重载自定义类型。

使用不同参数重载函数的另一种方法是使用 void* 来传递不同类型的参数，但这很糟糕。下面的示例只是在没有模板的情况下作为一种可能的替代方案。这里，我们设计了一个排序函数，该函

数将对任何可能类型的元素数组运行快速排序，该数组提供严格的弱排序。快速排序算法的详细信息可以在网上查询，例如：https://en.wikipedia.org/wiki/Quicksort。

快速排序算法需要比较和交换两个元素，但由于事先不知道其数据类型，不能直接实现。解决方案是依赖回调，回调是作为参数传递的函数，可以在必要时调用:

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

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

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

  fswap(arr, i + 1, high);

  return i + 1;
}

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

为了调用快速排序函数，需要为传递给函数的每种类型的数组提供比较和交换函数的实现。以下是 int 类型的实现:

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

```

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

定义了所有这些后，可以编写如下代码对整数数组进行排序:

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

这些例子关注的是函数，同样的问题也适用于类。假设要编写一个类，可对具有可变大小的数值集合建模，并将元素连续存储在内存中。可以使用以下实现 (这里仅概述声明) 来存储整数:

```
struct int_vector
{
  int_vector();

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

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

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

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

目前这一切看起来都很好，但当需要存储类型为 double、std::string 或自定义类型的值时，将需要编写相同的代码，每次只更改元素的类型。应该没有人愿意干这种事情吧，这明显就是一项重复性极高的工作，而且当需要更改某些内容时 (例如添加新功能或修复错误)，就需要在多个地方进行相同的更改。

最后，当要定义变量时，可能会遇到类似的问题 (尽管不太常见)。考虑保存换行字符的变量的情况:

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

若需要相同的常量，但需要不同的编码，比如：宽字符串字面值、UTF-8 等，该怎么办？多个变量当然可以，有不同的名字，比如下面的例子:

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

模板是一种技术手段，其允许开发人员编写蓝图，使编译器能够为我们生成所有这些重复的代码。下一节中，我们将看到如何将前面的代码转换为 C++ 模板。

1.2. 编写模板

本节中，我们将从三个简单的示例开始编写 C++ 模板，每个示例对应前面给出的代码片段。

前面 max 函数的模板如下所示:

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

这里的类型名称 (例如 int 或 double) 被 T(代表类型) 取代。T 称为类型模板形参，并通过语法 template<typename T> 或 typename<class T> 引入。T 是一个参数，可以使用任意的名称。

此时，源码中的模板只是一个蓝图，编译器将根据其使用情况生成代码，并将为使用模板的每种类型实例化一个函数重载:

```
struct foo{};
int main()
{
  foo f1, f2;
  max(1, 2); // OK, compares ints
  max(1.0, 2.0); // OK, compares doubles
  max(f1, f2); // Error, operator> not overloaded for
               // foo
}
```

代码中，首先使用两个整数调用 max，operator> 可用于 int 类型。这将生成一个重载 int max(int const a, int const b)。其次，用两个 double 值调用 max，这也没关系，因为 operator> 适用于 double 值。因此，编译器将生成另一个重载，double max(double const a, double const b)。第三次调用 max 将生成一个编译器错误，因为 foo 类型没有重载 operator>。

这里不讨论太多细节，使用 max 函数的方式如下所示:

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

编译器能够推断推导参数的类型，因此写上去就是是多余的。但在某些情况下，编译器无法推导参数类型，所以需要使用以下语法显式指定类型。

第二个例子涉及前一节中的函数，是处理 void* 参数的 quicksort() 实现，可以很容易地改为模板版本:

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

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

  for (int j = low; j <= high - 1; j++)
  {
    if (arr[j] < pivot)
    {
      i++;
      swap(&arr[i], &arr[j]);
    }
  }

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

  return i + 1;
}

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

快速排序函数模板的使用与前面的非常类似，并且不需要将指针传递给回调函数了:

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

前一节中看到的第三个例子是 vector 类，其模板版本如下所示:

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

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

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

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

  T at(size_t const index) const;
  T operator[](size_t const index) const;

private:
  T* data_;
  size_t size_;
  size_t capacity_;
};
```

max 函数那种情况，变化很小。类的模板声明中，元素的 int 类型可以使用类型模板形参 T 取代。实现方式如下:

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

声明变量 v 时必须指定元素的类型。代码中，v 的元素是 int，否则编译器将无法推断它们的类型。C++17 中，有些情况下可以不声明类型，可将其称为**类模板参数推导** (将在第 4 章中讨论)。

第四个也是最后一个例子，只有类型不同的情况下声明几个变量。可以用一个模板替换所有这些变量，如下所示:

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

该模板的使用方式如下:

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

```
}
```

本节中的示例表明，无论模板表示函数、类还是变量，声明和使用模板的语法都是相同的。这将引导我们进入下一节，通过模板类型而来了解模板。

1.3. 理解模板

本章中，已经使用了模板，有四个不同的术语可以描述我们所编写的模板类型:

- **函数模板**是用于模板化函数的术语。例子就是 max 模板。
- **类模板**是用于模板化类的术语 (可以用 class、struct 或 union 关键字定义)。例子就是前一节中的 vector 类。
- **可变参模板**是用于模板化变量的术语，例如上一节中的 NewLine 模板。
- **别名模板**是用于模板化类型别名的术语。将在下一章看到的别名模板。

模板用一个或多个参数进行参数化 (迄今为止看到的示例中，只有一个参数)。这些参数被称为**模板参数**，可以分为三类:

- **类型模板参数**，例如 template<typename T>，其中参数表示使用模板时指定的类型。
- **非类型模板形参**，如 template<size_t N> 或 template<auto N>，其中每个参数必须有一个结构类型，包括整型、浮点型 (C++20)、指针类型、枚举类型、左值引用类型等。
- **双重模板参数**，如 template<typename K, typename V, template<typename> typename C>，其中参数类型为另一个模板。

模板可以通过提供替代实现进行特化。这些实现可以依赖于模板参数的特征。特化的目的是实现优化或减少代码膨胀，其有两种形式:

- **偏特化**: 仅为某些模板参数提供的替代实现。
- **(显式) 全特化**: 当提供了所有模板参数时，这是模板特化的实例。

编译器通过模板生成代码的过程称为**模板实例化**，将模板实参替换为模板定义中使用的模板形参实现。例如，在 vector<int> 的例子中，编译器在 T 出现的每一个地方都替换了 int 类型。

模板实例化可以有两种形式:

- **隐式实例化**: 当编译器由于在代码中使用模板而实例化模板时，就会发生这种情况，这只发生在那些正在使用的组合或参数上。例如，编译器遇到 vector<int> 和 vector<double>，将实例化 vector 类模板为类型 int 和 double。
- **显式实例化**: 显式地告诉编译器要创建模板的哪些实例化的方法，即使这些实例化没有显式地在代码中使用。例如，创建库文件时，因为未实例化的模板不会放入目标文件中。其还有助于减少编译时间和对象大小，我们将在后面的内容中看到这些方法。

本节中提到的所有术语和主题都将在本书的其他章节中详细介绍。本节旨在作为模板的参考指南，但还有许多其他与模板相关的术语会在适当的时候进行介绍。

1.4. 模板简史

模板元编程是泛型编程的 C++ 实现。这种范式在 20 世纪 70 年代首次出现，在 20 世纪 80 年代上半叶出现了第一批支持泛型的语言：Ada 和 Eiffel。David Musser 和 Alexander Stepanov 在 1989 年的一篇论文中定义了泛型编程:

> 泛型编程的核心思想是从具体、有效的算法中抽象出来，以获得与不同数据表示相结合的泛型算法，进而生成各种软件。

这就是编程范式的定义，算法根据稍后指定的类型定义，并根据使用进行实例化。

模板最初是 **C with Classes** 语言的一部分，并不是由 Bjarne Stroustrup 开发的。Stroustrup 描述 C++ 模板的第一篇论文出现在 1986 年，也就是《C++ 程序设计语言第一版》出版一年后。1990 年，在 ANSI 和 ISO C++ 标准化委员会成立之前，模板已经存在于 C++ 中了。

20 世纪 90 年代早期，Alexander Stepanov, David Musser 和 Meng Lee 尝试在 C++ 中实现各种泛型概念，这就是**标准模板库 (STL)** 的第一个实现。当 ANSI/ISO 委员会在 1994 年发现到这个库时，就很快的将其添加到规范中了。1998 年，STL 与 C++ 语言一起标准化，也就是 C++98 标准。

C++ 标准的新版本，统称为**现代 C++**，引入了对模板元编程的各种改进。下表简要列了一下:

版本	特性	描述
C++11	可变参模板	有可变数量的模板参数。
	模板别名	能够使用声明定义模板类型的别名。
	外部模板	告诉编译器不要实例化模板
	类型特征	新头文件 <type_traits> 包含标识对象类别和类型特征。
C++14	变量模板	支持定义变量或静态数据成员。
C++17	折叠表达式	用二进制运算符减少可变参数模板的参数包。
	模板参数 typename	typename 关键字可以用来代替模板参数中的 class。
	非类型模板参数 auto	关键字 auto 可以用于非类型的模板参数。
	类模板的参数推导	编译器从对象初始化的方式推导模板参数的类型。
C++20	模板 Lambda	Ladmbd 表达式可作为模板。
	字符串字面值作为模板参数	字符串字面量可以用作非类型模板参数，以及用户定义的字符串字面操作符的新形式。
	约束	明确模板参数的需求。
	概念	命名的约束集。

表 1.1

这些特性，以及模板元编程的其他方面，是本书的唯一主题，并将在接下来的章节中进行详细介绍。现在，来看看模板的优缺点。

1.5. 模板的优缺点

使用模板之前，了解模板的优缺点相当重要。

先看看优点:

- 避免编写重复的代码。
- 提供算法和类型的泛型库的创建，例如标准 C++ 库 (有时错误地称为 STL)，无论是何类型，都可以在许多应用程序中使用。
- 使用模板可以得到少而优的代码。例如，使用标准库中的算法可以帮助编写更少的代码，这些代码可能更容易理解和维护，并且可能更健壮 (这些算法的开发和测试投入了相当多的精力)。

再来看看缺点：

- 语法复杂，但只要进行过一些实践，这应该不会对模板的开发和使用造成影响。
- 与模板代码相关的编译器错误通常很长而且神秘，很难确定其原因。新版本的 C++ 编译器已经简化了这类错误 (仍然是一个重要的问题)，C++20 标准中的概念就可以看做为是一种尝试 (包括帮助为编译错误提供更好的诊断信息)。
- 增加了编译时间，因为其在头文件中实现。每当对模板进行更改时，包含该头文件的所有翻译单元都必须重新编译。
- 模板库是作为一个或多个头文件的集合，必须与使用其代码一起编译。
- 头文件中实现模板的另一个缺点是无法隐藏信息，所有人都可以在头文件阅读整个模板代码。标准库开发人员经常使用具有诸如 detail 或 details 等命名空间来包含标准库内部的代码，但这些代码是标准库的使用者不应该了解的。
- 由于编译器没有实例化未使用的代码，因此这部分代码可能更难验证。在编写单元测试时，必须确保良好的代码覆盖率，对于开发库来说尤是如此。

虽然缺点有点多，但使用模板并不是一件坏事或应该避免的事情。相反，模板是 C++ 语言的强大特性。模板并不总能被正确理解，有时还会误用或过度使用。然而，明智地使用模板具有毋庸置疑的优势。本书将提供理解模板一种更好的方式 (以及使用方式)。

1.6. 总结

本章介绍了 C++ 语言中有关模板的概念。

从学习使用模板来解决问题开始，通过函数模板、类模板和变量模板的简单示例了解了模板。介绍了模板的基本术语，并且将在接下来的章节中讨论更多。本章的最后，简要介绍了模板在 C++ 语言中的历史，并本章的最后讨论了使用模板的优缺点。所有这些话题将引导我们更好地理解接下来的章节。

下一章中，我们将探索 C++ 中模板的基础知识。

1.7. 习题

1. 为什么需要模板? 模板有什么优势?
2. 什么是函数和类模板?

3. 模板参数有多少种，都是什么?
4. 什么是偏特化和全特化?
5. 模板的主要缺点是什么?

1.8. 扩展阅读

- Generic Programming, David Musser, Alexander Stepanov, http://stepanovpapers.com/genprog.pdf
- A History of C++: 1979−1991, Bjarne Stroustrup, https://www.stroustrup.com/hopl2.pdf
- History of C++, https://en.cppreference.com/w/cpp/language/history
- Templates in C++ - Pros and Cons, Sergey Chepurin, https://www.codeproject.com/Articles/275063/Templates-in-CplusplusPros-and-Cons

第 2 章　了解模板

前一章中，了解了对模板。本章将详细探讨这些内容，并研究模板参数、实例化、特化、别名等。

- 如何定义函数模板，类模板，变量模板和别名模板
- 模板参数的类型
- 模板的实例化
- 模板的特化
- 使用 Lambda 和 Lambda 模板

本章结束时，读者们将了解 C++ 中模板的核心基础知识，能够理解大量的模板代码，并且能够自己编写模板。

在本章开始之前，我们将探索定义和使用函数模板的细节。

2.1. 定义函数模板

函数模板的定义方式与常规函数类似，只是函数声明之前是关键字 template，尖括号之间是模板参数列表。下面是一个简单的函数模板示例:

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

这个函数有两个参数 a 和 b，为相同的 T 类型。该类型列在模板参数列表中，通过关键字 typename 或 class(前者在本例中使用，本书中也使用) 引入。这个函数所做的只是将两个参数相加，并返回该操作的结果，该结果应该具有相同的 T 类型。

函数模板只是创建实际函数的蓝图，只存在于源码中。除非在源码中显式调用，否则函数模板不会出现在编译后的可执行文件中。当编译器遇到对函数模板的调用，并且能够将提供的实参及其类型与函数模板的形参匹配时，它将根据模板和用于调用它的实参生成一个实际的函数。为了理解这一点，来看一个例子:

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

这段代码中，使用两个 int 形参 42 和 21 调用 add 函数。编译器能够从所提供参数的类型推导出模板形参 T，因此无需显式地提供。下面两个调用也是可能的，而且与前面的调用相同:

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

这种使用方式，将使编译器将生成以下函数 (不同编译器生成的实际代码可能不同):

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

若将调用方式更改为以下形式，则显式地为模板形参 T 提供实参 short:

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

这种情况下，编译器将生成该函数的另一个实例化，使用 short 而不是 int。这个新的实例化会如下所示:

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

若两个参数的类型不明确，编译器将无法自动推断出它们的类型。下面的调用就是这种情况:

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

这个例子中，41.0 是一个 double 类型，而 21 是一个 int 类型。add 函数模板有两个相同类型的形参，因此编译器无法将其与提供的实参进行匹配，并将报错。为了避免这种情况，并且假设希望模板为 double 实例化，这时必须显式地指定类型:

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

只要两个实参具有相同的类型，并且加法运算符可用于实参的类型，就可以按照前面所示的方式调用函数模板 add。若加法操作符不可用，则编译器将无法生成实例化，即使模板参数已正确解析:

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

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

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

这种情况下，编译器将报错，即没有为类型为 foo 的参数找到二进制加法操作符。当然，对于不同的编译器，实际的消息是不同的，所有错误都是如此。为了能够对 foo 类型的参数调用 add，必须重载此类型的加法操作符。可能的实现如下所示:

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

目前为止，我们看到的所有例子都是只有一个模板形参。但当模板可以有任意数量的参数，甚至可以有可变数量的参数 (后一个主题将在第 3 章中讨论)。下面的函数是一个有两个类型模板形参的函数模板:

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

此函数接受两个指向范围和谓词的开始和结束的输入迭代器，并返回范围内与谓词匹配的元素数量。这个函数，至少在概念上，非常类似于标准库中 <algorithm> 头文件中的 std::count_if 函数，开发者应该始终倾向于使用标准算法，而不是自己手工实现。但对于本主题而言，这个函数是一个很好的示例，可以帮助理解模板的工作方式。

可以这样使用 count_if 函数:

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

同样，不需要显式地指定类型模板形参的实参 (输入迭代器的类型和一元谓词的类型)，因为编译器能够从调用中进行推断。

关于函数模板还有很多东西需要学习，本节仅提供了使用方面的介绍。现在来了解定义类模板的基础知识。

2.2. 定义类模板

类模板将 template 关键字和 template 参数列表放在类声明之前。下面的代码展示了 wrapper 类模板，其只有一个模板形参 (类型为 T)，用作数据成员、形参和函数返回类型的类型:

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

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

private:
```

```
  T value;
};
```

只要源码中没有使用类模板，编译器就不会生成代码。所以类模板必须实例化，并且其所有形参必须由用户显式地匹配，或由编译器隐式地匹配。实例化这个类模板的例子如下:

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

这段代码中的 a 和 e 的定义只在 C++17 及以后的版本中有效，这要归功于**类模板参数推导**特性。这个特性使我们可以使用类模板，而不指定任何模板参数，编译器能够进行推导即可 (在第 4 章中讨论)。在此之前，所有引用类模板的示例都将显式列出参数，如 wrapper<int> 或 wrapper<char const*>。

类模板可以在不定义的情况下声明，并在允许不完整类型的上下文中使用，例如函数的声明:

```
template <typename T>
class wrapper;

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

类模板必须在模板实例化发生的地方定义; 否则，编译器将报错。

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

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

int main()
{
  wrapper<int> a(42); // error, incomplete type
  use_wrapper(&a);
}

template <typename T>
class wrapper
{
  // template definition
};

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

声明 use_wrapper 函数时，只声明了类模板 wrapper，而没有定义。但在此上下文中允许使用不完整类型，因此可以使用 wrapper<T>。但在 main 函数中，正在实例化 wrapper 类模板的对象。因

为类模板的定义必须可用，所以这将生成编译器错误。要修复这个特定的示例，必须将 main 函数的定义移到末尾，在 wrapper 和 use_wrapper 定义之后。

本例中，类模板是使用 class 关键字定义的。C++ 中，使用 class 或 struct 关键字声明类之间几乎没什么区别:

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

可以使用 struct 关键字定义类模板，就像使用 class 关键字一样。对于用 struct 或 class 关键字定义的类模板，也可以观察到用 struct 或 class 关键字定义的类之间的差异。

类，不管是不是模板，都可以包含成员函数模板。下一节将讨论如何定义这些成员函数。

2.3. 定义成员函数模板

我们已经学习了函数模板和类模板。本节中，我们将学习如何在非模板类和类模板中定义成员函数模板。为了理解它们的区别，先来看个例子:

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

复合类是一个类模板，有一个名为 add 的成员函数，使用类型形参 T。这个类可以这样使用:

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

首先需要实例化组合类的一个对象，必须显式地指定类型形参 T 的实参，因为编译器不能自己推导出来 (没有上下文)。当调用 add 函数时，只提供参数。其类型 (由 T 类型模板形参表示，之前解析为 int) 是已知的，像 c.add<int>(42, 21) 这样的调用使编译器报错。add 函数不是一个函数模板，而是一个常规函数，它是复合类模板的成员。

下一个示例中，复合类略有变化。先来看看定义:

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

composition 是非模板类，但 add 函数是一个函数模板。要调用这个函数，必须执行以下操作:

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

为 T 类型模板形参显式指定 int 类型是多余的，编译器可以从调用的实参中自行推导出 int 类型。这里是为了更好地理解这两种实现之间的差异。

除了类模板的成员函数和类成员函数模板这两种情况外，还可以有类模板的成员函数模板。这种情况下，成员函数模板的模板形参必须与类模板的模板形参不同; 否则，编译器将报错。回到包装器类模板的示例，并对其进行如下修改:

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

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

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

这个实现还有一个成员，一个名为 as 的函数。这是一个函数模板，有一个名为 U 的类型模板形参。该函数用于将包装的值从类型 T 转换为类型 U，并将其返回给调用者:

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

模板形参的实参在实例化包装器类时指定 (double)——尽管在 C++17 中这是冗余的，并且在调用 as 函数 (int) 执行时也可以指定。

继续其他内容 (例如实例化、特化和其他形式的模板，包括变量和别名) 之前，花点时间了解更多关于模板参数的知识非常重要。

2.4. 模板参数

我们已经看到了带有一个或多个参数的模板的多个示例，其中参数表示实例化时提供的类型，这些类型可以由用户显式提供，也可以由编译器在可以推导时隐式提供。这些类型的参数称为类型模板参数，但模板也可以有非类型模板参数和双重模板参数。在接下来的部分中，我们将探讨这些问题。

2.4.1 类型模板参数

如前所述，这些参数表示模板实例化期间作为参数提供的类型，由 typename 或 class 关键字引入。使用这两个关键字没什么区别。类型模板参数可以有一个默认值，即类型。这与为函数形参指定默认值的方式相同:

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

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

类型模板参数的名称可以省略，这在转发声明时很有用:

```
template <typename>
class wrapper;

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

C++11 引入了可变参数模板，这些模板的参数数量可变。接受零个或多个参数的模板形参称为形参包，类型模板参数包具有以下形式:

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

可变参数模板将在第 3 章中讨论，这里不会详细讨论这类参数。

C++20 引入了概念和约束。约束指定了对模板参数的要求，一系列命名的约束称为概念。概念可以指定为类型模板参数，但语法有点不同。使用概念的名称 (若有的话，后面跟着尖括号中的模板参数列表) 来代替 typename 或 class 关键字。具有默认值和约束类型模板参数包的概念:

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

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

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

概念和约束将在第 6 章中讨论，将在那一章学习更多关于这类参数的知识。现在，来看看第二种模板参数，非类型模板参数。

2.4.2 非类型模板参数

模板参数并不总是必须表示类型，也可以是编译时表达式，例如常量、外部函数，或对象的地址或静态类成员的地址。编译时表达式提供的参数称为非类型模板参数，这类参数只能有结构类型。以下是结构类型的描述:

- 整数类型

- 浮点类型 (始于 C++20)
- 枚举类型
- 指针类型 (指向对象或函数)
- 指向成员类型的指针 (指向成员对象或成员函数)
- 左值引用类型 (指向对象或函数)
- 字面量类类型，需要满足以下要求:
  - 所有基类都是 public 且不可变的。
  - 所有非静态数据成员都是 public 且不可变的。
  - 所有基类和非静态数据成员的类型也是结构类型或其数组。

这些类型的 cv 限定形式也可以用于非类型模板参数。非类型模板参数可以以不同的方式指定，可能的形式如下所示:

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

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

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

这些例子中，非类型模板形参的类型是 int。第一个示例和第二个示例类似，只是第二个示例使用了默认值。第三个示例有很大不同，因为参数实际上是一个参数包，这将在下一章讨论。

为了更好地理解非类型模板参数，来看看下面的例子，这里有一个固定大小的数组类，称为 buffer:

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

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

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

这个 buffer 类包含 S 个 T 类型元素的内部数组，所以 S 需要是一个编译类型的值。这个类可以进行如下的实例化:

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

这两个定义相同，b1 和 b2 都是两个存储 10 个整数的 buffer。此外，其具有相同的类型，因为 2*5 和 10 是两个表达式，其计算值为相同的编译时值。可以用下面的语句进行检查:

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

现在不再是这样了，b3 对象的类型声明如下:

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

b3 是一个包含 15 个整数的 buffer，这与前面示例中包含 10 个整数的 buffer 类型不同。从概念上讲，编译器会生成以下代码:

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

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

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

这是主模板的代码，下面显示了一些特化:

```
template<>
class buffer<int, 10>
{
  int data_[10];
public:
  constexpr int * data() const;
  constexpr int & operator[](const size_t index);
  constexpr const int & operator[](
    const size_t index) const;
};

template<>
class buffer<int, 15>
{
  int data_[15];
public:
```

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

这个示例中，可以看到的特化的概念 (本章后续的内容中详细介绍)。目前，应该注意到这两种不同的 buffer 类型，可以用下面的语句验证 b1 和 b3 的类型是否相同:

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

实践中，结构类型 (如整数、浮点数或枚举类型) 的使用比其他类型更常见。理解其用法，并找到相应的例子可能更容易，也有使用指针或引用的场景。下面的例子中，将研究如何使用指向函数参数的指针。先来看看代码:

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

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

device 是一个基类，具有一个名为 output 的纯虚函数 (以及一个虚析构函数)。这是名为 smart_device 的类模板的基类，该类模板通过函数指针调用函数来实现 output 虚函数，此函数的指针会作为传递给类模板的非类型模板的实参。下面的示例展示了如何使用:

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

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

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

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

```
w2->output();
```

w1 和 w2 是两个 unique_ptr 对象，指向相同类型的对象，但事实并非如此。因为 smart_device<&say_hello_in_english> 和 smart_device<&say_hello_in_spanish> 是不同的类型，所以它们的函数指针实例化不同。这可以很容易确定:

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

另一方面，若用 std::unique_ptr<device> 替换 auto，那么 w1 和 w2 是指向基类 device 的智能指针，因此具有相同的类型:

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

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

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

虽然这个例子使用的是指向函数的指针，但也可以假设一个类似的例子，用于指向成员函数的指针。前面的例子可以转换为以下 (仍然使用相同的基类设备) 方式:

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

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

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

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

这些类的使用方式如下所示:

```
hello_command cmd;

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

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

C++17 中，引入了一种指定非类型模板参数的新形式，使用 auto(包括 auto* 和 auto&) 或 decltype(auto) 来代替类型名，编译器可以从作为实参提供的表达式中推断参数的类型。若派生的类型不允许用于非类型模板参数，编译器将生成错误的类型。让我们来看一个例子:

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

这个类模板可以这样用:

```
foo<42> f1;   // foo<int>
foo<42.0> f2; // foo<double> in C++20, error for older
              // versions
foo<"42"> f3; // error
```

第一个例子中，对于 f1，编译器将实参的类型推断为 int。第二个例子中，对于 f2，编译器将类型推断为 double，但需要支持 C++20。以前的 C++ 版本中，这一行会产生错误。C++20 之前，浮点类型不允许作为非类型模板参数。所以，最后一行会出错，因为“42”是一个字符串字面量，不能用作非类型模板参数。

然而，最后一个例子可以在 C++20 中通过将字面值字符串，包装在结构字面值类中来解决。这个类将字符串字面值的字符存储在一个固定长度的数组中:

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

前面的 foo 类模板也需要修改，需要显式使用 string_literal，而不是 auto:

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

C++20 中，foo<”42”> f; 的声明将在编译时通过。

auto 也可以与非类型模板参数包一起使用，类型为每个模板参数独立推导。模板参数的类型不需要相同:

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

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

本例中，编译器将模板参数的类型分别推断为 int、double、bool 和 char。

第三类也是最后一类模板参数是双重模板参数。

2.4.3 双重模板参数

尽管这个名字有点奇怪，指的是一类模板参数，这些参数本身就是模板。这些参数可以像类型模板参数一样指定，带或不带名称，带或不带默认值，以及作为带或不带名称的参数包。C++17 起，关键字 class 和 typename 都可以用来引入双重模板参数。此版本之前，只能使用 class 关键字。

为了展示双重模板参数的使用，先来看一下下面两个类模板:

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

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

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

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

simple_wrapper 类是一个简单的类模板，保存类型模板参数 T 的值。另一方面，fancy_wrapper 是一个更复杂的包装器实现，隐藏了包装的值，并公开了用于数据访问的成员函数。接下来，我们实现

一个名为 wrapped_pair 的类模板，包含两个包装类型的值。这可以是 simple_wrapper，fancy_wrapper，或任何类似的类型:

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

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

wrapping_pair 类模板有三个参数。前两个是类型模板形参，名为 T 和 U。第三个形参是模板模板参数为 W，有一个默认值，即 fancy_wrapper 类型。可以使用这个类模板，如下所示:

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

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

这个例子中，p1 是一个 wrapping_pair 对象包含两个值，一个 int 和一个 double，每个值都包装在一个 fancy_wrapper 对象中。这没有显式指定，其为双重模板参数的默认值。另一方面，p2 也是一个 wrapping_pair 对象，也包含一个 int 和一个 double，但是由一个 simple_wrapper 对象包装的，该对象在模板实例化中进行了显式指定。

这个例子中，我们看到了默认模板参数的使用。

2.4.4 默认模板参数

默认模板参数的指定与默认函数参数类似，在参数列表中的等号后面。以下规则适用于默认模板参数:

- 可以与任何类型的模板参数一起使用，参数包除外。
- 若为类模板、变量模板或类型别名的模板参数指定了默认值，则所有后续模板参数也必须具有默认值。若最后一个参数是模板参数包，则是例外。
- 若在函数模板中为模板参数指定了默认值，则后续模板参数也不限于具有默认值。
- 函数模板中，只有当参数包有默认实参或者其值可以由编译器从函数实参推导出来时，形参包才可以后面跟更多的类型参数。
- 不允许在友元类模板的声明中出现。
- 只有当友元函数模板的声明也有定义，并且在同一个翻译单元中没有其他函数声明时，才允许在友元函数模板的声明中使用。

- 不允许在函数模板或成员函数模板的显式特化的声明或定义中出现。

下面展示了使用默认模板参数的示例:

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

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

声明类模板时，带默认实参的模板参数不能后面跟着不带默认实参的形参，但此限制不适用于函数模板:

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

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

一个模板可以有多个声明 (但只有一个定义)，来自所有声明和定义的默认模板参数可以进行合并 (与合并默认函数参数的方式相同):

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

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

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

语义上等价于:

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

但这些具有不同默认模板参数的多个声明不能按任何顺序提供，前面提到的规则仍然适用。若类模板声明的第一个形参有默认实参，而后面的形参没有默认实参即是非法:

```
template <typename T = int, typename U>
struct foo; // error, U does not have a default argument

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

对默认模板实参的另一个限制是，同一个模板形参不能在同一个作用域中赋予多个默认值。因此，下面的例子将在编译时报错:

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

template <typename T = int> // error redefinition
                            // of default parameter
struct foo {};
```

当默认模板实参使用来自类名时，成员访问限制会在声明时进行检查，而不是在模板实例化时:

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

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

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

当 x 变量定义时，bar 类模板即会实例化，但是 foo::value_type 的类型的定义是 protected，因此不能在 foo 之外使用。结果是在 bar 类模板的声明处出现编译错误。

了解了这些内容后，我们先结束模板参数的话题。下一节中我们将探讨的是模板实例化，是根据模板定义和一组模板参数创建函数、类或变量的新定义。

2.5. 模板实例化

模板只是蓝图，编译器在遇到模板时，会根据模板创建实际代码。从模板声明中为函数、类或变量创建定义的行为称为**模板实例化**。这可以是**显式的** (告诉编译器何时应该生成定义时)，也可以是**隐式的** (编译器根据需要生成新定义时)。我们将在接下来的内容来了解这两种实例化方式。

2.5.1 隐式实例化

当编译器基于模板的使用生成定义，并且没有显式实例化时，就会触发隐式实例化。隐式实例化模板定义在与模板相同的命名空间中，不过编译器从模板创建定义的方式可能有所不同。先来看看这段代码:

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

```
6
7  int main()
8  {
9    foo<int> x;
10 }
```

有一个名为 foo 的类模板，有一个成员函数 f。main 函数中，定义了一个类型为 foo<int> 的变量，但没有使用其成员。因为采用这种方式使用 foo，所以编译器隐式地为 int 类型定义 foo 的特化。可以在 cppinsights.io 上使用在 Clang 运行，会看到下面的代码:

```
1  template<>
2  struct foo<int>
3  {
4    inline void f();
5  };
```

因为函数 f 在代码中没有调用，所以其只是声明而没有定义。若在 main 中添加调用 f 的代码，会发生如下的特化:

```
1  template<>
2  struct foo<int>
3  {
4    inline void f() { }
5  };
```

若在下面的实现中添加一个包含错误的函数 g，会在不同的编译器中看到不同的行为:

```
1  template <typename T>
2  struct foo
3  {
4    void f() {}
5    void g() {int a = "42";}
6  };
7
8  int main()
9  {
10   foo<int> x;
11   x.f();
12 }
```

g 的主体包含一个错误 (也可以使用 static_assert(false) 语句作为替代)。这段代码在 VC++ 中编译没有问题，但是在 Clang 和 GCC 中就会失败。因为 VC++ 忽略了模板中未使用的部分，前提是代码语法正确，但其他部分在模板实例化之前进行了语义验证。

对于函数模板，当用户代码在需要函数定义存在的上下文中引用函数时，就会发生隐式实例化。对于类模板，当用户代码在需要完整类型的上下文中引用模板时，或者当类型的完整性影响代码时，也会隐式实例化。此类上下文是构造此类类型的对象，但在声明指向类模板的指针时就是另外一种情况了。来看看下面的例子:

```
1  template <typename T>
2  struct foo
```

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

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

使用与前面示例相同的 foo 类模板，并声明了几个变量:p 是指向 foo<int> 的指针，x 是指向 foo<int> 的指针，q 是指向 foo<double> 的指针。由于声明了 x，此时编译器只需要实例化 foo<int>，考虑调用成员函数 f 和 g:

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

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

通过这些更改，编译器需要实例化以下内容:

- 当声明 x 变量时，实例化 foo<int>
- 当 x.f() 调用发生时，实例化 foo<int>::f()
- 当 q->g() 调用发生时，实例化 foo<double> 和 foo<double>::g()。

另外，当声明指针 p 时，编译器不需要实例化 foo<int>；当声明指针 q 时，也不需要实例化 foo<double>。然而，当涉及到指针转换时，编译器确实需要隐式实例化类模板特化:

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

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

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

函数 show 中，button<int>* 和 control<int>* 之间进行了转换，此时编译器必须实例化 button<int>。

当类模板包含静态成员时，编译器隐式实例化类模板，这些成员不会隐式实例化；只在编译器需要它们的定义时，才会隐式实例化。另外，类模板的每个特化都有自己的静态成员副本:

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

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

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

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

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

类模板 foo 有一个名为 data 的静态成员变量，在 foo 定义之后初始化。main 函数中，变量 a 的类型为 foo<int>，b 和 c 的类型为 foo<double>。最初，它们都将成员数据初始化为 0，但变量 b 和 c 共享相同的数据副本。因此，赋值 b.data = 42 后，a.data 仍然是 0，但 b.data 和 c.data 都是 42。

了解了隐式实例化如何工作之后，是时候来了解显式实例化了。

2.5.2 显式实例化

可以显式地告诉编译器实例化类模板或函数模板，这称为显式实例化。它有两种形式: **显式实例化定义**和**显式实例化声明**。

定义

显式实例化定义可以出现在程序中的任何地方，但不能出现在所引用的模板定义之后。显式模板实例化定义的语法采用以下形式:

- 类模板的语法如下:

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

- 函数模板的语法如下:

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

显式实例化定义用 template 关键字引入，但后面没有任何参数列表。对于类模板，class-key 可以是 class、struct 或 union 关键字中的一个。对于类模板和函数模板，带有给定参数列表的显式实例化定义只能在整个程序中出现一次。

我们将通过一些例子来理解这是如何工作的。下面是第一个例子:

```
namespace ns
{
  template <typename T>
  struct wrapper
  {
    T value;
  };
  template struct wrapper<int>; // [1]
}

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

int main() {}
```

wrapper<T> 是在 ns 命名空间中定义的类模板。代码中标记为 [1] 和 [2] 的语句都表示显式实例化定义，分别为 wrapper<int> 和 wrapper<double>。显式实例化定义只能出现在与其引用的模板 (如 [1]) 相同的命名空间中，或者必须是完全限定的 (如 [2])。可以为函数模板编写类似的显式模板定义:

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

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

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

int main() { }
```

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

若显式实例化定义与模板不在同一个命名空间中，则名称必须完全限定。using 语句的使用不会使名称在当前命名空间中可见:

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

using namespace ns;
```

```
template struct wrapper<double>; // error
```

本例中的最后一行在编译时会报错，因为 wrapper 是一个未知的名称，必须使用命名空间名称进行限定，如 ns::wrapper。

当类成员用于返回类型或参数类型时，显式实例化定义会忽略成员访问规则:

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

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

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

类 X<T>::bar 和函数 foo<T>::f() 都是 foo<T> 类的 private 部分，但它们可以在最后一行显示的显式实例化定义中使用。

了解了显式实例化定义是什么，以及它如何工作之后，现在的问题就是它什么时候有使用。为什么要告诉编译器从模板生成实例化? 答案是其有助于分发库、减少构建时间和可执行文件大小。若正在构建.lib 文件作为发布库，并且该库使用模板，则未实例化的模板定义不会放入库中，在每次使用库时用户代码的构建时间增加。通过强制在库中实例化模板，这些定义会放入目标文件和分发的.lib 文件中。因此，用户代码只需要链接到库文件中可用的函数即可，这就是 Microsoft MSVC CRT 库为所有流、区域设置和字符串类所做的工作。libstdc++ 库会对字符串类和其他类执行相同的操作。

模板实例化可能会出现的问题是，最终可能会得到多个定义，每个翻译单元一个定义。若包含模板的同一个头文件包含在多个翻译单元 (.cpp 文件) 中，并且使用了相同的模板实例化 (假设 wrapper<int> 来自前面的例子)，这些实例化的副本放在每个翻译单元中，这将导致目标文件大小的增加。这个问题可以在显式实例化声明的帮助下解决。

声明

显式实例化声明 (C++11 中提供) 可以告诉编译器模板实例化的定义在不同的翻译单元中，并且不应该生成新的定义。语法与显式实例化定义相同，除了关键字 extern 可用在声明前:

- 类模板的语法如下所示:

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

- 函数模板的语法如下所示:

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

若提供了显式的实例化声明，但在程序的翻译单元中都不存在实例化定义，则结果是编译器警告和链接器错误。该技术是在一个源文件中声明显式模板实例化，在其余的源文件中声明显式模板声明，这将减少编译时间和目标文件的大小。

来看看下面的例子:

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

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

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

template wrapper<int>; // [2]

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

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

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

// main.cpp
#include "wrapper.h"

int main()
{
  wrapper<int> a{ 0 };
}
```

这个例子中，可以看到:

- wrapper.h 头文件包含一个名为 wrapper<T> 的类模板。[1] 中，wrapper<int> 有一个显式的实例化声明，告诉编译器在编译包含此头文件的源文件 (翻译单元) 时，不要为该实例化生成定义。
- source1.cpp 文件包含 wrapper.h，并且 [2] 中包含 wrapper<int> 的显式实例化定义。这是整个程

序中这个实例化的唯一定义。

- 源文件 source2.cpp 和 main.cpp 都使用了 wrapper<int>，但是没有显式的实例化定义或声明。当头文件包含在每个文件中时，wrapper.h 的显式声明是可见的。

或者，可以从头文件中取出显式实例化声明，但随后必须将其添加到包含头文件的每个源文件中，而这很可能被遗忘。

进行显式模板声明时，类主体中定义的类成员函数总是视为 inline，因此会实例化，所以只能对在类体之外定义的成员函数使用 extern 关键字。

现在已经了解了什么是模板实例化，我们将继续讨论另一个重要的主题：模板特化。这个术语用于从模板实例化创建的定义，以处理一组特定的模板参数。

2.6. 模板特化

模板特化是从模板实例化创建的定义，进行特化的模板称为主模板。可以为给定的模板参数集提供显式的特化定义，从而覆盖编译器将生成的隐式代码。这种技术支持诸如类型特征和条件编译等特性，这些元编程概念将在第 5 章中探讨。

模板特化有两种形式: 显式 (全) 特化和偏特化。

2.6.1 显式特化

显式特化 (也称为全特化) 发生在，使用完整的模板参数集为模板实例化提供定义时。以下可以全特化的模板:

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

看一下下面的例子:

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

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

template <>
struct is_floating_point<double>
```

```
{
  constexpr static bool value = true;
};

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

is_float_point 是主模板，包含一个名为 value 的 constexpr 静态布尔数据成员，用 false 值初始化。然后，对这个主模板有三种完全的特化，分别用于 float、double 和 long double 类型。这些新定义改变了使用 true，而不是 false 初始化值的方式，所以可以使用这个模板编写如下代码:

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

第一行和最后一行打印 0，即 false; 其他行输出 1，即 true。这个例子演示了类型特征是如何工作的，标准库在 std 命名空间中包含一个名为 is_floating_point 的类模板，定义在 <type_traits> 标准头文件中。我们将在第 5 章中了解更多相关内容。

正如在本例中看到的，静态类成员可以全特化。每个特化版本都有自己的静态成员副本:

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

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

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

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

foo<T> 是一个类模板，只有一个静态成员，称为 value。对于主模板初始化为 0，对于 int 特化初始化为 42。声明变量 a、b 和 c 后，a.value 为 0,b.value 为 0，而 c.value 为 42。将值 100 赋给 a.value 后，b.value 也是 100，而 c.value 仍然是 42。

显式特化必须出现在主模板声明之后，不需要在显式特化之前对主模板进行定义:

```
template <typename T>
struct is_floating_point;

template <>
struct is_floating_point<float>
{
```

```
  constexpr static bool value = true;
};

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

模板特化也可以只声明而不定义，模板特化可以像其他不完整类型一样使用:

```
template <typename>
struct foo {}; // primary template

template <>
struct foo<int>; // explicit specialization declaration

foo<double> a; // OK
foo<int>* b; // OK
foo<int> c; // error, foo<int> incomplete type
```

foo<T> 是主模板，其中存在 int 类型的显式特化声明，可以使用 foo<double> 和 foo<int>*(支持声明指向部分类型的指针)。声明 c 变量时，完整的类型 foo<int> 是不可用的，因为缺少 int 的完整特化的定义，所以这会在编译时报错。

当特化函数模板时，若编译器可以从函数实参的类型推导出模板实参，那么该模板的参数可选填:

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

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

template<>
void func(foo<int>)
{
  std::cout << "int specialization\n";
}
```

func 函数模板的完整特化 int 的语法应该是 template<> func<int>(foo<int>)，而编译器能够从函数参数中推断出 T 所代表的实际类型，所以不必在定义特化时进行指定。

另外，函数模板和成员函数模板的声明或定义不允许包含默认函数参数。下面的例子中，编译器将报错:

```
template <typename T>
void func(T a)
{
```

```
  std::cout << "primary template\n";
}

template <>
void func(int a = 0) // error: default argument not allowed
{
  std::cout << "int specialization\n";
}
```

所有这些示例中，模板都只有一个模板参数。实际中，许多模板都有多个参数。显式特化要求定义指定完整的参数集:

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

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

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

func(1, 2); // int-int specialization
func(1, 2.0); // int-double specialization
func(1.0, 2.0); // primary template
```

了解了这些内容后，继续讨论偏特化，也就是显式 (全) 特化的泛化。

2.6.2 偏特化

偏特化发生在特化一个主模板但只指定一些模板参数时，所以偏特化同时具有模板形参列表 (跟在模板关键字后面) 和模板实参列表 (跟在模板名称后面)，只有类可以偏特化。

下面的例子用来说明这是如何工作的:

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

template <typename T>
```

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

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

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

这里有一个名为 collection 的主模板，有两个模板实参 (一个类型模板实参和一个非类型模板实参)，有三个偏特化:

- 非类型模板参数 S 的特化，其值为 10
- int 类型的特化
- 指针类型 T* 的特化

这些模板可以这样使用:

```
collection<char, 42> a; // primary template
collection<int, 42> b; // partial specialization <int, S>
collection<char, 10> c; // partial specialization <T, 10>
collection<int*, 20> d; // partial specialization <T*, S>
```

正如注释中所指定的，a 从主模板实例化，b 从 int 的偏特化 (collection<int, S>)，c 从 10 的偏特化 (collection<T, 10>)，d 从指针的偏特化 (collection<T*, S>)。有些组合是不可能的，因为类型模糊的原因，编译器无法选择使用哪个模板实例化。这里有几个例子:

```
collection<int, 10> e; // error: collection<T,10> or
                       // collection<int,S>
collection<char*, 10> f; // error: collection<T,10> or
                         // collection<T*,S>
```

第一种情况下，collection<T, 10> 和 collection<int, S> 偏特化都匹配类型 collection<int, 10>，而在第二种情况下，可以是 collection<T, 10> 或 collection<T*, S>。

当定义主模板的特化时，需要记住几点:

- 偏特化的模板参数列表中的参数不能有默认值。
- 模板参数列表表示模板参数列表中的顺序，该顺序仅针对偏特化。偏特化的模板参数列表不能与模板参数列表所表示的模板参数列表相同。

- 模板参数列表中，只能对非类型模板参数使用标识符，此上下文中不允许使用表达式:

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

当类模板具有偏特化版本时，编译器必须决定从哪里生成定义的最佳匹配，将模板特化的模板实参与主模板和偏特化的模板参数列表相匹配。根据匹配过程的结果，编译器执行以下操作:

- 若没有找到匹配项，则从主模板生成定义。
- 若发现单个偏特化，则从该特化生成定义。
- 若发现了多个偏特化，则从最特化的偏特化生成定义，但前提是其是唯一的；否则，编译器将报错 (如前所述)。若模板 A 接受了模板 B 接受的类型的子集，则认为模板 A 比模板 B 更特化，反之不然。

但偏特化不是通过名称查找找到的，只有在通过名称查找找到主模板时才会考虑使用。

为了理解偏特化有什么用，让我们看一个实际例子.

本例中会创建一个函数，该函数将格式化数组的内容并将其输出到流中。格式化数组的内容应该类似于 [1,2,3,4,5]，但对于 char 元素的数组，元素不应该用逗号分隔，而应该显示为方括号内的字符串，例如 [demo]。为此，考虑使用 std::array 类。下面的实现使用元素间分隔符格式化数组的内容:

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

  return os;
}

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

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

pretty_print 是一个带有两个模板参数的函数模板，与 std::array 类的模板形参匹配。当以 arr 数组作为参数调用时，输出 [1,1,2,3,5,8,13,21]。当以 str 数组作为参数调用时，输出 [t,e,m,p,l,a,t,e]。然而，我们想在这种情况下打印 [template]。为此，需要另一个实现，使其专门用于 char 类型:

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

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

第二个实现中，pretty_print 是一个带有单个模板形参的函数模板，该形参是非类型模板参数，指示数组的大小。类型模板参数显式指定为 char，在 std::array<char, S> 中。这一次，调用带有 str 数组的 pretty_print 将 [template] 输出到控制台。

这里需要理解的是，其不是 pretty_print 函数模板，而是 std::array 类模板。函数模板不能特化，而是重载函数，因为 std::array<char, S> 是主类模板 std::array<T, S> 的特化。

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

2.7. 定义变量模板

变量模板是在 C++14 中引入，允许在命名空间范围内定义模板变量 (可以表示一组全局变量)，或者在类范围内定义模板变量 (表示静态数据成员)。

变量模板在命名空间范围内声明，如下所示。这是一个典型的例子，可以在文献中找到，这里用它来说明变量模板的好处:

```cpp
template<class T>
constexpr T PI = T(3.14159265358979323851L);
```

其语法类似于声明变量 (或数据成员)，但与声明模板的语法相结合。

随之而来的问题是变量模板有什么用处。为了回答这个问题，让我们些写个例子来说明。假设想要写一个函数模板，给定一个球体的半径，返回它的体积。球体体积公式为 $\frac{4\pi r^3}{3}$，可能的实现如下所示:

```cpp
constexpr double PI = 3.14159265358979323851L;

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

PI 定义为 double 类型的编译时常数。这将生成编译器警告，若使用 float，例如，类型模板参数 T:

```
float v1 = sphere_volume(42.0f); // warning
double v2 = sphere_volume(42.0); // OK
```

这个问题的一种解决方案是，使 PI 成为模板类的静态数据成员，其类型由类型模板参数决定:

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

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

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

尽管使用 PI<T>::value 并不理想，但也可行，若可以写成 PI<T> 就更好了。这正是本节开头显示的变量模板 PI 可以做的事情，下面是完整的解决方案:

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

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

下一个例子展示了变量模板的显式特化:

```
template<typename T>
constexpr T SEPARATOR = '\n';

template<>
constexpr wchar_t SEPARATOR<wchar_t> = L'\n';

template <typename T>
std::basic_ostream<T>& show_parts(
  std::basic_ostream<T>& s,
  std::basic_string_view<T> const& str)
{
  using size_type =
    typename std::basic_string_view<T>::size_type;
  size_type start = 0;
  size_type end;
```

```
  do
  {
    end = str.find(SEPARATOR<T>, start);
    s << '[' << str.substr(start, end - start) << ']'
    << SEPARATOR<T>;
    start = end+1;
  } while (end != std::string::npos);

  return s;
}

show_parts<char>(std::cout, "one\ntwo\nthree");
show_parts<wchar_t>(std::wcout, L"one line");
```

有一个名为 show_parts 的函数模板，会将输入字符串分割为用分隔符分隔的部分后处理。分隔符是在 (全局) 命名空间范围内定义的变量模板，并且为 wchar_t 类型显式特化。

如前所述，变量模板可以是类的成员，可表示静态数据成员，需要使用 static 关键字进行声明:

```
struct math_constants
{
  template<class T>
  static constexpr T PI = T(3.14159265358979323851L);
};

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

可以在类中声明一个变量模板，然后在类外提供它的定义。注意，变量模板必须声明为静态 const，而不是静态 constexpr，因为后者需要在类内初始化:

```
struct math_constants
{
  template<class T>
  static const T PI;
};

template<class T>
const T math_constants::PI = T(3.14159265358979323851L);
```

变量模板用于简化类型特征的使用。显式特化部分包含一个名 is_float _point 的类型特征示例。下面是其主模板:

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

有几个显式的特化，不在这里进行列出。然而，这种类型的特征可以这样使用:

```
std::cout << is_floating_point<float>::value << '\n';
```

使用 is_float _point<float>::value 相当麻烦，但是可以通过定义如下的变量模板来避免:

```
template <typename T>
inline constexpr bool is_floating_point_v =
  is_floating_point<T>::value;
```

is_floating_point_v 变量模板有助于编写更简单、更易于阅读的代码。下面是我喜欢的形式:

```
std::cout << is_floating_point_v<float> << '\n';
```

标准库为::value 定义了一系列以_v 结尾的变量模板，就像我们的例子一样 (例如 std::is_floating_point_v 或 std::is_same_v)。变量模板的实例化类似于函数模板和类模板。

这可以通过显式实例化或显式特化实现，也可以由编译器隐式实现。当在必须存在变量定义的上下文中使用变量模板时，或者需要该变量对表达式求常量时，编译器就会生成定义。

接下来，要介绍的主题为别名模板，其允许我们为类模板定义别名。

2.8. 定义别名模板

C++ 中别名是指先前定义的类型的名称，可以是内置类型，也可以是用户定义的类型。别名的主要目的是为具有较长名称的类型提供较短的名称，或者为某些类型提供语义上有意义的名称。这可以通过 typedef 声明来实现，也可以通过 using 声明来实现 (后者是 C++11 中引入的)。下面是使用 typedef 的几个例子:

```
typedef int index_t;
typedef std::vector<
      std::pair<int, std::string>> NameValueList;
typedef int (*fn_ptr)(int, char);

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

这个例子中，index_t 是 int 的别名，NameValueList 是 std::vector<std::pair<int, std::string>> 的别名，而 fn_ptr 是指向一个函数的指针类型的别名，该函数返回 int 类型，并具有 int 和 char 类型的两个参数。最后，foo::value_type 是类型模板 T 的别名。

自 C++11 起，这些类型别名可以在使用声明的帮助下创建:

```
using index_t = int;
using NameValueList =
  std::vector<std::pair<int, std::string>>;
using fn_ptr = int(*)(int, char);

template <typename T>
```

```
struct foo
{
  using value_type = T;
};
```

using 声明现在比 typedef 声明更受欢迎，因为更容易使用，也更容易阅读 (从左到右)。然而，与 typedef 相比，using 有一个重要的优势，即允许我们为模板创建别名。别名模板不是指类型而是指类型族的名称。记住，模板不是类、函数或变量，而是允许创建类型、函数或变量家族的蓝图。

为了理解别名模板的工作原理，看看以下示例:

```
template <typename T>
using customer_addresses_t =
  std::map<int, std::vector<T>>; // [1]

struct delivery_address_t {};
struct invoice_address_t {};

using customer_delivery_addresses_t =
  customer_addresses_t<delivery_address_t>; // [2]
using customer_invoice_addresses_t =
  customer_addresses_t<invoice_address_t>; // [3]
```

第 [1] 行上的声明引入别名模板 customer_addresses_t，其是 map 类型的别名，键类型为 int，值类型为 std::vector<T>。因为 std::vector<T> 不是一个类型，而是一个类型集，所以 customer_addresses_t<T> 定义了一个类型集。[2] 和 [3] 处的 using 声明从上述类型集中引入了两种类型别名，customer_delivery_addresses_t 和 customer_invoice_addresses_t。

别名模板可以像模板声明一样出现在命名空间或类范围内。另一方面，他们既不能全特化，也不能偏特化。然而，有一些方法可以跨越这一限制。解决方案是创建具有类型别名成员的类模板并特化该类，然后可以创建引用类型别名成员的别名模板。让我们通过一个示例来演示一下。

下面的代码虽然不是有效的 C++ 代码，但代表了我想要实现的最终目标 (若别名模板可能特化的话):

```
template <typename T, size_t S>
using list_t = std::vector<T>;

template <typename T>
using list_t<T, 1> = T;
```

list_t 是 std::vector<T> 的别名模板，前提是集合的大小大于 1。若只有一个元素，list_t 应该是类型模板参数 T 的别名:

```
template <typename T, size_t S>
struct list
{
  using type = std::vector<T>;
};

template <typename T>
struct list<T, 1>
```

```
{
  using type = T;
};

template <typename T, size_t S>
using list_t = typename list<T, S>::type;
```

list<T,S> 是一个类模板，其成员类型别名为 T。主模板中，是 std::vector<T> 的别名。偏特化列表 <T,1> 中，是 T 的别名,list_t 定义为 list<T, S>::type 的别名模板。下面的断言证明了这种机制的有效性:

```
static_assert(std::is_same_v<list_t<int, 1>, int>);
static_assert(std::is_same_v<list_t<int, 2>,
std::vector<int>>);
```

结束本章之前，还有一个主题需要讨论: 通用 Lambda 及其在 C++20 中进阶版本——Lambda 模板。

2.9. 通用 Lambda 和 Lambda 模板

Lambda 在形式上称为 Lambda 表达式，是对于定义函数对象的一种简化方法，通常包括传递给算法的谓词或比较函数。我们不会讨论 Lambda 表达式，但先来看看下面的例子:

```
int arr[] = { 1,6,3,8,4,2,9 };
std::sort(
  std::begin(arr), std::end(arr),
  [](int const a, int const b) {return a > b; });

int pivot = 5;
auto count = std::count_if(
  std::begin(arr), std::end(arr),
  [pivot](int const a) {return a > pivot; });
```

Lambda 表达式是语法糖，是定义匿名函数对象的一种简化方式。遇到 Lambda 表达式时，编译器会生成一个带有函数调用操作符的类。对于前面的例子，如下所示:

```
struct __lambda_1
{
  inline bool operator()(const int a, const int b) const
  {
    return a > b;
  }
};

struct __lambda_2
{
  __lambda_2(int & _pivot) : pivot{_pivot}
  {}

  inline bool operator()(const int a) const
```

```
  {
    return a > pivot;
  }
private:
  int pivot;
};
```

这里的名称很随意，每个编译器将生成不同的名称。另外，实现细节可能有所不同，这里看到的是编译器应该生成的最小值。注意，第一个 Lambda 和第二个 Lambda 的区别在于，后者包含通过值捕获的状态。

Lambda 表达式是在 C++11 中引入的，该标准的后续版本中进行了多次更新。值得注意的是，本章将讨论两个问题:

- 泛型 Lambda：在 C++14 中引入，其允许我们使用 auto，而不是显式地指定类型。这将生成的函数对象转换为，具有模板函数调用操作符的函数对象。
- 模板 Lambda：在 C++20 中引入，模板 Lambda 可以使用模板语法显式指定模板化的函数调用操作符。

为了理解它们之间的区别，以及泛型 Lambda 和模板 Lambda 的作用，来看看以下示例:

```
auto l1 = [](int a) {return a + a; }; // C++11, regular
                                      // lambda
auto l2 = [](auto a) {return a + a; }; // C++14, generic
                                       // lambda

auto l3 = []<typename T>(T a)
          { return a + a; }; // C++20, template lambda

auto v1 = l1(42); // OK
auto v2 = l1(42.0); // warning
auto v3 = l1(std::string{ "42" }); // error

auto v5 = l2(42); // OK
auto v6 = l2(42.0); // OK
auto v7 = l2(std::string{"42"}); // OK

auto v8 = l3(42); // OK
auto v9 = l3(42.0); // OK
auto v10 = l3(std::string{ "42" }); // OK
```

这里有三个不同的 Lambda 表达式: l1 是一个常规 Lambda, l2 是一个泛型 Lambda，因为至少有一个参数用 auto 定义，l3 是一个模板 Lambda，用模板语法定义，但没有使用 template 关键字。

可以用一个整数调用 l1，也可以使用 double 类型调用，但这一次编译器将产生一个关于可能丢失数据的警告。尝试用字符串参数调用会产生编译错误，因为 std::string 不能转换为 int。另一方面，l2 是泛型。编译器继续为它调用的所有参数类型实例化其特化版本，本例中是 int、double 和 std::string。下面的代码展示了生成的函数的外观 (至少在概念上):

```
struct __lambda_3
{
```

```
  template<typename T1>
  inline auto operator()(T1 a) const
  {
    return a + a;
  }

  template<>
  inline int operator()(int a) const
  {
    return a + a;
  }

  template<>
  inline double operator()(double a) const
  {
    return a + a;
  }

  template<>
  inline std::string operator()(std::string a) const
  {
    return std::operator+(a, a);
  }
};
```

可以在这里看到函数调用操作符的主模板，以及提到的三个特化。毫不奇怪，编译器将为第三个 Lambda 表达式 l3 生成相同的代码，这是一个模板 Lambda，仅在 C++20 中可用。由此产生的问题是，通用 Lambda 和 Lambda 模板有什么不同? 为了回答这个问题，稍微修改一下前面的例子:

```
auto l1 = [](int a, int b) {return a + b; };
auto l2 = [](auto a, auto b) {return a + b; };
auto l3 = []<typename T, typename U>(T a, U b)
          { return a + b; };

auto v1 = l1(42, 1); // OK
auto v2 = l1(42.0, 1.0); // warning
auto v3 = l1(std::string{ "42" }, '1'); // error

auto v4 = l2(42, 1); // OK
auto v5 = l2(42.0, 1); // OK
auto v6 = l2(std::string{ "42" }, '1'); // OK
auto v7 = l2(std::string{ "42" }, std::string{ "1" }); // OK

auto v8 = l3(42, 1); // OK
auto v9 = l3(42.0, 1); // OK
auto v10 = l3(std::string{ "42" }, '1'); // OK
auto v11 = l3(std::string{ "42" }, std::string{ "42" }); // OK
```

新 Lambda 表达式有两个参数，可以用两个整数或一个 int 和一个 double(尽管这同样会产生警告) 调用 l1，但不能用字符串和 char 调用，但可以用泛型 Lambda l2 和模板 Lambda l3 来完成所有

这些。从语义上看，编译器生成的代码对于 l2 和 l3 是相同的:

```
struct __lambda_4
{
  template<typename T1, typename T2>
  inline auto operator()(T1 a, T2 b) const
  {
    return a + b;
  }

  template<>
  inline int operator()(int a, int b) const
  {
    return a + b;
  }

  template<>
  inline double operator()(double a, int b) const
  {
    return a + static_cast<double>(b);
  }

  template<>
  inline std::string operator()(std::string a,
  char b) const
  {
    return std::operator+(a, b);
  }

  template<>
  inline std::string operator()(std::string a,
  std::string b) const
  {
    return std::operator+(a, b);
  }
};
```

我们看到了函数调用操作符的主模板，以及几个完全显式的特化: 两个 int 值，一个 double 和一个 int，一个字符串和一个 char，以及两个字符串对象。但若想将泛型 Lambda l2 的使用限制为相同类型的参数呢? 这是不可能的。编译器无法推断我们的意图，所以将为参数列表中出现的 auto，生成不同类型的模板参数。然而，C++20 的 Lambda 模板允许指定函数调用操作符的形式。来看看下面的例子:

```
auto l5 = []<typename T>(T a, T b) { return a + b; };

auto v1 = l5(42, 1); // OK
auto v2 = l5(42, 1.0); // error

auto v4 = l5(42.0, 1.0); // OK
auto v5 = l5(42, false); // error
```

```
auto v6 = l5(std::string{ "42" }, std::string{ "1" }); // OK

auto v6 = l5(std::string{ "42" }, '1'); // error
```

使用两个不同类型的参数调用 Lambda 模板是不可能的，即使可以隐式转换，例如从 int 转换为 double。编译器将会报错。调用模板 Lambda 时，不可能显式地提供模板参数，例如：l5<double>(42,1.0) 会产生一个编译错误。

decltype 类型说明符允许告知编译器从表达式推导出类型。C++14 中，可以在泛型 Lambda 中使用它，来说明前面泛型 Lambda 表达式中的第二个参数具有与第一个参数相同的类型。看起来如下所示:

```
auto l4 = [](auto a, decltype(a) b) {return a + b; };
```

所以第二个参数 b 的类型必须转换为第一个参数 a 的类型，并且可以进行以下方式的调用:

```
auto v1 = l4(42.0, 1); // OK
auto v2 = l4(42, 1.0); // warning
auto v3 = l4(std::string{ "42" }, '1'); // error
```

因为 int 可以隐式转换为 double，所以第一次调用编译时没有问题。第二个调用编译时发出警告，因为从 double 转换为 int 可能会导致数据丢失。但第三个调用会报错，因为 char 不能隐式转换为 std::string。l4 Lambda 比前面看到的泛型 Lambda l2 有所改进，但若参数类型不同，仍然不能帮助完全限制调用。这仅适用于前面所示的 Lambda 模板。

下一段代码中显示了 Lambda 模板的另一个示例。这个 Lambda 只有一个参数 std::array，但数组元素的类型和数组的大小指定为 Lambda 模板的模板参数:

```
auto l = []<typename T, size_t N>(
            std::array<T, N> const& arr)
{
  return std::accumulate(arr.begin(), arr.end(),
                         static_cast<T>(0));
};

auto v1 = l(1); // error
auto v2 = l(std::array<int, 3>{1, 2, 3}); // OK
```

试图用 std::array 对象以外的对象调用这个 Lambda 会产生编译器错误。编译器生成的函数对象可能如下所示:

```
struct __lambda_5
{
  template<typename T, size_t N>
  inline auto operator()(
    const std::array<T, N> & arr) const
  {
    return std::accumulate(arr.begin(), arr.end(),
                  static_cast<T>(0));
  }
```

```
10
11    template<>
12    inline int operator()(
13      const std::array<int, 3> & arr) const
14    {
15      return std::accumulate(arr.begin(), arr.end(),
16                  static_cast<int>(0));
17    }
18  };
```

泛型 Lambda 相对于常规 Lambda 的好处是递归 Lambda。Lambda 是匿名的，因此不能直接递归调用。必须定义一个 std::function 对象，将 Lambda 表达式赋给它，并在捕获列表中通过引用捕获它。下面是一个递归 Lambda 的例子，用来计算数字的阶乘:

```
1  std::function<int(int)> factorial;
2  factorial = [&factorial](int const n) {
3    if (n < 2) return 1;
4    else return n * factorial(n - 1);
5  };
6
7  factorial(5);
```

这可以通过使用泛型 Lambdas 来简化，不需要 std::function 及其捕获。递归泛型 Lambda 的实现如下所示:

```
1  auto factorial = [](auto f, int const n) {
2    if (n < 2) return 1;
3    else return n * f(f, n - 1);
4  };
5
6  factorial(factorial, 5);
```

若理解起来很难，编译器生成的代码应该能帮助搞清楚这是如何进行工作的:

```
1  struct __lambda_6
2  {
3    template<class T1>
4    inline auto operator()(T1 f, const int n) const
5    {
6      if(n < 2) return 1;
7      else return n * f(f, n - 1);
8    }
9
10   template<>
11   inline int operator()(__lambda_6 f, const int n) const
12   {
13     if(n < 2) return 1;
14     else return n * f.operator()(__lambda_6(f), n - 1);
15   }
16 };
17
```

```
__lambda_6 factorial = __lambda_6{};
factorial(factorial, 5);
```

泛型 Lambda 是一个带有模板函数调用操作符的函数对象。用 auto 指定的第一个参数可以是任何东西，包括 Lambda 本身。因此，编译器将为生成的类型，提供完全显式特化的调用操作符。

当需要将函数对象作为参数传递给其他函数时，Lambda 表达式可以避免编写显式代码。相反，编译器可以生成代码。泛型 Lambda，在 C++14 中引入，避免为不同类型编写相同的 Lambda。C++20 的 Lambda 模板允许在模板语法和语义下，指定生成调用操作符的形式。

2.10. 总结

本章介绍了 C++ 模板的核心特性，了解了如何定义类模板、函数模板、变量模板和别名模板。学习了模板参数之后，详细研究了模板实例化和模板特化。还了解了通用 Lambda 和 Lambda 模板，以及与常规 Lambda 相比，有哪些优点。阅读完本章，现在应该熟悉了模板的基础知识，并且应该可以看懂大部分模板代码，并可以自己编写模板。

下一章中，将讨论另一个重要的主题，即具有可变数量参数的模板，即可变参数模板。

2.11. 习题

1. 什么类型可以用于非类型模板参数?
2. 默认模板参数在哪里不允许使用?
3. 什么是显式实例化声明，在语法上与显式实例化定义有何不同?
4. 什么是别名模板?
5. 什么是模板 Lambda?

2.12. 扩展阅读

- C++ Template: A Quick UpToDate Look(C++11/14/17/20), http://www.vishalchovatiya.com/c-template-a-quick-uptodate-look/
- Templates aliases for C++, https://www.stroustrup.com/templatealiases.pdf
- Lambdas: From C++11 to C++20, Part 2, https://www.cppstories.com/2019/03/lambdas-story-part2/

第 3 章 可变参数模板

可变参数模板是具有可变数量参数的模板，是 C++11 中引入的特性。继承自 C 语言的特性，将泛型代码与参数数量可变的函数结合在一起。尽管语法和细节看起来很麻烦，但可变参数模板可编写带有可变数量参数的函数模板或带有可变数量数据成员的类模板，这在以前的编译时计算和类型安全方面是不可能的。

本章中，将了解以下主题:

- 可变参数模板的需求
- 可变参数函数模板
- 参数包
- 可变参数类模板
- 折叠表达式
- 可变参数别名模板
- 可变参数变量模板

本章结束时，读者们将很好地理解如何编写可变参数模板。

3.1. 可变参数模板的需求

最著名的 C/C++ 函数之一是 printf，将格式化输出写入标准输出流。实际上，I/O 库中有一系列用于写入格式化输出的函数，其中还包括 fprintf(写入文件流)、sprint 和 snprintf(写入字符缓冲区)。这些函数很相似，接受一个定义输出格式的字符串和可变数量的参数。C++ 语言提供了一种方法，用可变数量的参数来编写自己的函数。下面是一个例子，函数可以接受一个或多个参数，并返回最小值:

```
#include<stdarg.h>

int min(int count, ...)
{
  va_list args;
  va_start(args, count);

  int val = va_arg(args, int);
  for (int i = 1; i < count; i++)
  {
    int n = va_arg(args, int);
    if (n < val)
      val = n;
  }

  va_end(args);

  return val;
}

```

```
int main()
{
  std::cout << "min(42, 7)=" << min(2, 42, 7) << '\n';
  std::cout << "min(1,5,3,-4,9)=" <<
          min(5, 1, 5, 3, -4,
             9) << '\n';
}
```

这个实现特定于 int 类型，也可以编写一个类似的函数，即函数模板:

```
template <typename T>
T min(int count, ...)
{
  va_list args;
  va_start(args, count);

  T val = va_arg(args, T);
  for (int i = 1; i < count; i++)
  {
    T n = va_arg(args, T);
    if (n < val)
    val = n;
  }

  va_end(args);

  return val;
}

int main()
{
  std::cout << "min(42.0, 7.5)="
          << min<double>(2, 42.0, 7.5) << '\n';
  std::cout << "min(1,5,3,-4,9)="
              << min<int>(5, 1, 5, 3, -4, 9) << '\n';
}
```

这样的代码，无论是否通用，都有几个缺点:

- 需要使用几个宏:va_list(提供对其他宏的信息)、va_start(启动参数的迭代)、va_arg(提供下一个参数) 和 va_end(停止参数的迭代)。
- 即使传递给函数的参数的数量和类型在编译时已知，求值部分也发生在运行时。
- 以这种方式实现的可变函数类型不安全。va_macros 执行低内存操作，类型转换在运行时的 va_arg 中完成，会导致运行时异常。
- 可变参函数需要以某种方式指定变量参数的数量。min 函数的实现中，第一个参数表示参数的数量。类似 printf 的函数接受一个格式化字符串，从中确定预期参数的数量。例如，printf 函数计算并忽略额外的参数 (若提供的参数多于格式化字符串中指定的数量)，但若提供的参数较少，则为未定义行为。

此外，C++11 之前，只有函数是可变参的，有些类也可以从拥有可变参数数量的数据成员中受益。典型的例子是 tuple 类和 variant 类，前者表示固定大小的异构值集合，后者是类型安全的联合体。

可变参数模板有助于解决所有这些问题，在编译时求值类型安全，不需要宏，不需要显式指定参数的数量，可以编写可变参数函数模板和可变参数类模板。此外，也有可变参数变量模板和可变参数别名模板。

下一节，我们开始研究可变参数函数模板。

3.2. 可变参数函数模板

可变参数函数模板是具有可变数量参数的模板函数，用省略号 (…) 的用法来指定一组参数，这些参数可以有不同的语法。

为了理解可变参数函数模板的基本原理，我们重写一下之前 min 函数:

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

template <typename T, typename... Args>
T min(T a, Args... args)
{
  return min(a, min(args...));
}

int main()
{
  std::cout << "min(42.0, 7.5)=" << min(42.0, 7.5)
            << '\n';
  std::cout << "min(1,5,3,-4,9)=" << min(1, 5, 3, -4, 9)
            << '\n';
}
```

这里有两个 min 函数的重载。第一个是带有两个形参的函数模板，返回两个形参中最小的一个。第二个是带有可变数量参数的函数模板，通过扩展形参包递归地调用自身。尽管可变函数模板实现看起来像是使用了某种编译时递归机制 (带有两个参数的重载作为结束)，但只依赖于重载函数，从模板和提供的参数集实例化。

可变参数变量函数模板的实现中，省略号 (…) 在三个不同的地方使用，具有不同的含义:

- 要在模板参数列表中指定一组参数，如 typename…Args，这称为模板参数包。可以为类型模板、非类型模板和双重模板参数定义的模板参数包。
- 函数参数列表中指定一组参数，如 Args…args，这称为函数参数包。
- 函数体中展开包，如在 args…中，可在 min(args…) 中看到，这称为参数包展开。这种展开的结果是一个由零个或多个值 (或表达式) 组成的逗号分隔列表。

从 min(1,5,3,-4,9) 开始，编译器实例化了一组带有 5,4,3 和 2 个参数的重载函数。理论上，其拥有以下一组重载函数:

```
int min(int a, int b)
{
  return a < b ? a : b;
}

int min(int a, int b, int c)
{
  return min(a, min(b, c));
}

int min(int a, int b, int c, int d)
{
  return min(a, min(b, min(c, d)));
}

int min(int a, int b, int c, int d, int e)
{
  return min(a, min(b, min(c, min(d, e))));
}
```

min(1,5,3,-4,9) 会扩展为 min(1, min(5, min(3, min(- 4,9))))。这可能会引发关于可变参数模板性能的问题。实际中，编译器会进行大量的优化，例如：尽可能多地内联。当启用优化时，将不会有实际函数的调用。可以使用在线资源，例如 Compiler Explorer(https://godbolt.org/))，以查看不同编译器使用不同选项生成的代码 (例如优化设置)。考虑下面的代码片段 (min 是前面所示实现的可变参数函数模板):

```
int main()
{
  std::cout << min(1, 5, 3, -4, 9);
}
```

GCC 11.2 中使用-O 标志进行编译以进行优化，会生成以下汇编代码:

```
sub rsp, 8
mov esi, -4
mov edi, OFFSET FLAT:_ZSt4cout
call std::basic_ostream<char, std::char_traits<char>>
        ::operator<<(int)
mov eax, 0
add rsp, 8
ret
```

这里，不需特别了解汇编。对 min(1,5,3,-4,9) 的计算在编译时完成，结果-4 直接加载到 ESI 寄存器中。没有运行时调用，也没有计算，一切在编译时皆为已知。当然，情况并非总是如此。

下面的代码段显示了对 min 函数模板的调用，不能在编译时求值，因为参数只有在运行时才明了:

```
int main()
{
  int a, b, c, d, e;
  std::cin >> a >> b >> c >> d >> e;
  std::cout << min(a, b, c, d, e);
}
```

这一次，生成的汇编代码如下所示 (只显示了调用 min 函数的代码):

```
mov esi, DWORD PTR [rsp+12]
mov eax, DWORD PTR [rsp+16]
cmp esi, eax
cmovg esi, eax
mov eax, DWORD PTR [rsp+20]
cmp esi, eax
cmovg esi, eax
mov eax, DWORD PTR [rsp+24]
cmp esi, eax
cmovg esi, eax
mov eax, DWORD PTR [rsp+28]
cmp esi, eax
cmovg esi, eax
mov edi, OFFSET FLAT:_ZSt4cout
call std::basic_ostream<char, std::char_traits<char>>
        ::operator<<(int)
```

编译器已经内联了所有对 min 重载的调用。只有一系列的指令用于将值加载到寄存器中，比较寄存器值，并根据比较结果进行跳转，但是没有函数调用。

当禁用优化时，确实会发生函数调用。可以通过使用特定于编译器的宏来跟踪在调用 min 函数期间发生的这些调用。GCC 和 Clang 提供了一个名为__PRETTY_FUNCTION__的宏，包含函数的签名及其名称。类似地，Visual C++ 提供了一个名为__FUNCSIG__的宏，做了同样的事情。这些可以在函数体中使用，以打印其名称和签名:

```
template <typename T>
T min(T a, T b)
{
#if defined(__clang__) || defined(__GNUC__) || defined(__GNUG__)
  std::cout << __PRETTY_FUNCTION__ << "\n";
#elif defined(_MSC_VER)
  std::cout << __FUNCSIG__ << "\n";
#endif
  return a < b ? a : b;
}

template <typename T, typename... Args>
T min(T a, Args... args)
{
#if defined(__clang__) || defined(__GNUC__) || defined(__GNUG__)
  std::cout << __PRETTY_FUNCTION__ << "\n";
```

```
#elif defined(_MSC_VER)
  std::cout << __FUNCSIG__ << "\n";
#endif
  return min(a, min(args...));
}

int main()
{
  min(1, 5, 3, -4, 9);
}
```

当用 Clang 编译这个程序时，执行的结果如下所示:

```
T min(T, Args...) [T = int, Args = <int, int, int, int>]
T min(T, Args...) [T = int, Args = <int, int, int>]
T min(T, Args...) [T = int, Args = <int, int>]
T min(T, T) [T = int]
T min(T, T) [T = int]
T min(T, T) [T = int]
T min(T, T) [T = int]
```

当用 Visual C++ 编译时，输出如下所示:

```
int __cdecl min<int,int,int,int,int>(int,int,int,int,int)
int __cdecl min<int,int,int,int>(int,int,int,int)
int __cdecl min<int,int,int>(int,int,int)
int __cdecl min<int>(int,int)
int __cdecl min<int>(int,int)
int __cdecl min<int>(int,int)
int __cdecl min<int>(int,int)
```

尽管 Clang/GCC 和 VC++ 之间签名的格式有很大的不同，但也有相同的显示: 首先，调用一个带有五个参数的重载函数；然后是一个带有四个参数的重载函数；再然后是一个带有三个参数的重载函数；最后，有四个带有两个参数的重载函数调用 (这标志着扩展的结束)。

理解参数包的扩展是理解可变参数模板的关键，我们将在下一节中详细探讨这个主题。

3.3. 参数包

模板或函数形参包可以接受零个、一个或多个参数。该标准没有指定参数数量的上限，但编译器可能会进行设定。该标准所做的是为这些限制推荐最小值，但不要求遵守这些限定值。这些限制如下:

- 对于函数形参包，参数的最大数量取决于函数调用的参数的限制，建议至少为 256 个。
- 对于模板形参包，参数的最大数量取决于模板形参的限制，建议至少为 1024 个。

参数包中的参数数量，可以在编译时使用 sizeof…操作符检索。该操作符返回 std::size_t 类型的 constexpr 值。

第一个示例中，sizeof…操作符用于借助 constexpr if 语句实现可变参数函数模板 sum 的递归模式的结束。若形参包中的参数数量为零 (函数只有一个参数)，则处理的是最后一个参数，因此只返

回值。否则，将第一个参数与其余参数进行加和。实现如下所示:

```
template <typename T, typename... Args>
T sum(T a, Args... args)
{
  if constexpr (sizeof...(args) == 0)
    return a;
  else
    return a + sum(args...);
}
```

这在语义上是等价的，比下面的可变参数函数模板的实现更加简洁:

```
template <typename T>
T sum(T a)
{
  return a;
}

template <typename T, typename... Args>
T sum(T a, Args... args)
{
  return a + sum(args...);
}
```

注意 sizeof…(args)(函数形参包) 和 sizeof…(Args)(模板形参包) 返回相同的值。另一方面，sizeof…(args) 和 sizeof(args)…不是一回事。前者是用于参数包 args 的 sizeof 操作符，后者是 sizeof 操作符上参数包 args 的扩展:

```
template<typename... Ts>
constexpr auto get_type_sizes()
{
  return std::array<std::size_t,
  sizeof...(Ts)>{sizeof(Ts)...};
}

auto sizes = get_type_sizes<short, int, long, long long>();
```

这段代码中，sizeof…(Ts) 在编译时计算为 4，而 sizeof(Ts)…展开为以下以逗号分隔的参数包:sizeof(short)、sizeof(int)、sizeof(long)、sizeof(long long)。所以，前面的函数模板 get_type_sizes 相当于下面的函数模板，有四个模板形参:

```
template<typename T1, typename T2,
         typename T3, typename T4>
constexpr auto get_type_sizes()
{
  return std::array<std::size_t, 4> {
    sizeof(T1), sizeof(T2), sizeof(T3), sizeof(T4)
  };
}
```

通常，形参包是函数或模板的尾形参。但若编译器可以推断实参，则一个形参包后面可以跟其他形参包，其中包括更多形参包:

```
template <typename... Ts, typename... Us>
constexpr auto multipacks(Ts... args1, Us... args2)
{
  std::cout << sizeof...(args1) << ','
            << sizeof...(args2) << '\n';
}
```

这个函数应该取两组可能类型不同的元素，并对它们做一些处理:

```
multipacks<int>(1, 2, 3, 4, 5, 6);
                // 1,5
multipacks<int, int, int>(1, 2, 3, 4, 5, 6);
                // 3,3
multipacks<int, int, int, int>(1, 2, 3, 4, 5, 6);
                // 4,2
multipacks<int, int, int, int, int, int>(1, 2, 3, 4, 5, 6);
                // 6,0
```

对于第一次调用，args1 包在函数调用时指定 (如 multipacks<int> 一样) 并包含 1，而 args2 则从函数参数推断为 2,3,4,5,6。类似地，对于第二次调用，两个包将拥有相同数量的参数，更准确地说是 1,2,3 和 3,4,6。对于最后一次调用，第一个包包含所有元素，第二个包为空。这些示例中，元素都是 int 类型。但在下面的例子中，这两个包中就会有不同类型的元素了:

```
multipacks<int, int>(1, 2, 4.0, 5.0, 6.0); // 2,3
multipacks<int, int, int>(1, 2, 3, 4.0, 5.0, 6.0); // 3,3
```

对于第一次调用，args1 包将包含整数 1、2，而 args2 包将推导为包含 double 值 4.0、5.0、6.0。类似地，对于第二次调用，args1 包将是 1,2,3，而 args2 包将包含 4.0,5.0,6.0。

若稍微改变一下函数模板的多重包，要求这些包具有相同的大小，那么只有前面显示的调用仍然可用。如下例所示:

```
template <typename... Ts, typename... Us>
constexpr auto multipacks(Ts... args1, Us... args2)
{
  static_assert(
  sizeof...(args1) == sizeof...(args2),
  "Packs must be of equal sizes.");
}

multipacks<int>(1, 2, 3, 4, 5, 6); // error
multipacks<int, int, int>(1, 2, 3, 4, 5, 6); // OK
multipacks<int, int, int, int>(1, 2, 3, 4, 5, 6); // error
multipacks<int, int, int, int, int, int>(1, 2, 3, 4, 5, 6); // error

multipacks<int, int>(1, 2, 4.0, 5.0, 6.0); // error
multipacks<int, int, int>(1, 2, 3, 4.0, 5.0, 6.0); // OK
```

这里，只有第 2 个和第 6 个调用是有效的。这两种情况下，两个推导的包各有三个元素。其他情况下，就像前面的例子一样，包的大小不同，static_assert 将在编译时生成错误。

多参数包并不特定于可变参数函数模板，也可以用于偏特化中的可变类模板，前提是编译器可以推断模板参数。为了说明，我们考虑用一对函数指针表示类模板的情况，实现应该允许存储指向任何函数的指针。所以，我们定义了一个主模板，称为 func_pair，以及带有四个模板参数的偏特化:

- 第一个函数的返回类型的类型模板参数
- 第一个函数的参数类型的模板参数包
- 第二个函数的返回类型的第二个类型模板参数
- 第二个模板形参包，用于第二个函数的形参类型

func_pair 类模板如下所示:

```
template<typename, typename>
struct func_pair;

template<typename R1, typename... A1,
         typename R2, typename... A2>
struct func_pair<R1(A1...), R2(A2...)>
{
  std::function<R1(A1...)> f;
  std::function<R2(A2...)> g;
};
```

为了演示这个类模板的使用，考虑以下两个函数:

```
bool twice_as(int a, int b)
{
  return a >= b*2;
}

double sum_and_div(int a, int b, double c)
{
  return (a + b) / c;
}
```

可以实例化 func_pair 类模板，并使用它来使用这两个函数:

```
func_pair<bool(int, int), double(int, int, double)> funcs{
  twice_as, sum_and_div };

funcs.f(42, 12);
funcs.g(42, 12, 10.0);
```

参数包可以在不同的上下文中展开，这将成为下一节的主题。

3.3.1 展开参数包

参数包可以出现在不同的上下文中，扩展形式可能取决于这一背景。前面列出了这些可能的上下文和示例:

- 模板参数 (parameter) 列表: 为模板指定参数时:

```
template <typename... T>
struct outer
{
  template <T... args>
  struct inner {};
};

outer<int, double, char[5]> a;
```

- 模板参数 (argument) 列表: 为模板指定参数时:

```
template <typename... T>
struct tag {};

template <typename T, typename U, typename ... Args>
void tagger()
{
  tag<T, U, Args...> t1;
  tag<T, Args..., U> t2;
  tag<Args..., T, U> t3;
  tag<U, T, Args...> t4;
}
```

- 函数参数 (parameter) 列表: 为函数模板指定参数时:

```
template <typename... Args>
void make_it(Args... args)
{ }
make_it(42);
make_it(42, 'a');
```

- 函数参数 (argument) 列表: 展开包出现在函数调用的圆括号内时，省略号左侧表达式或大括号初始化列表就是展开模式:

```
template <typename T>
T step_it(T value)
{
  return value+1;
}

template <typename... T>
int sum(T... args)
{
  return (... + args);
}

template <typename... T>
void do_sums(T... args)
{
  auto s1 = sum(args...);
```

```
  // sum(1, 2, 3, 4)

  auto s2 = sum(42, args...);
  // sum(42, 1, 2, 3, 4)

  auto s3 = sum(step_it(args)...);
  // sum(step_it(1), step_it(2),... step_it(4))
}

do_sums(1, 2, 3, 4);
```

- 圆括号初始化式: 展开包出现在直接初始化式、函数样式强制转换、成员初始化式、new 表达式和其他类似上下文的圆括号内时，规则与函数实参列表的上下文相同:

```
template <typename... T>
struct sum_wrapper
{
  sum_wrapper(T... args)
  {
    value = (... + args);
  }

  std::common_type_t<T...> value;
};

template <typename... T>
void parenthesized(T... args)
{
  std::array<std::common_type_t<T...>,
             sizeof...(T)> arr {args...};
  // std::array<int, 4> {1, 2, 3, 4}

  sum_wrapper sw1(args...);
  // value = 1 + 2 + 3 + 4

  sum_wrapper sw2(++args...);
  // value = 2 + 3 + 4 + 5
}
parenthesized(1, 2, 3, 4);
```

- 大括号初始化式: 使用大括号符号进行初始化时:

```
template <typename... T>
void brace_enclosed(T... args)
{
  int arr1[sizeof...(args) + 1] = {args..., 0};
  // arr1: {1,2,3,4,0}

  int arr2[sizeof...(args)] = { step_it(args)... };
  // arr2: {2,3,4,5}
}
```

```
10
11 brace_enclosed(1, 2, 3, 4);
```

- 基类说明符和成员初始化程序列表: 包展开可以在类声明中指定基类的列表。此外，也可能出现在成员初始化器列表中，因为调用基类的构造函数可能需要这样做:

```
1  struct A {};
2  struct B {};
3  struct C {};
4
5  template<typename... Bases>
6  struct X : public Bases...
7  {
8    X(Bases const & ... args) : Bases(args)...
9    { }
10 };
11
12 A a;
13 B b;
14 C c;
15 X x(a, b, c);
```

- 使用声明: 从一组基类派生的上下文中，能够将基类中的名称引入派生类的定义中也是有用的，所以包展开也可能出现在 using 声明中:

```
1  struct A
2  {
3    void execute() { std::cout << "A::execute\n"; }
4  };
5
6  struct B
7  {
8    void execute() { std::cout << "B::execute\n"; }
9  };
10
11 struct C
12 {
13   void execute() { std::cout << "C::execute\n"; }
14 };
15
16 template<typename... Bases>
17 struct X : public Bases...
18 {
19   X(Bases const & ... args) : Bases(args)...
20   {}
21
22   using Bases::execute...;
23 };
24
25 A a;
26 B b;
```

```
C c;
X x(a, b, c);

x.A::execute();
x.B::execute();
x.C::execute();
```

- Lambda 捕获:Lambda 表达式的捕获子句可能包含包展开:

```
template <typename... T>
void captures(T... args)
{
  auto l = [args...]{
            return sum(step_it(args)...); };
  auto s = l();
}

captures(1, 2, 3, 4);
```

- 折叠表达式:

```
template <typename... T>
int sum(T... args)
{
  return (... + args);
}
```

- sizeof…操作符: 本节前面已经展示了示例。这里再来一个例子:

```
template <typename... T>
auto make_array(T... args)
{
  return std::array<std::common_type_t<T...>,
                    sizeof...(T)> {args...};
};

auto arr = make_array(1, 2, 3, 4);
```

- 对齐说明符: 对齐说明符中的包展开与将多个对齐说明符应用于同一声明具有相同的效果。参数包可以是类型包，也可以是非类型包。下面列出了这两种情况的示例:

```
template <typename... T>
struct alignment1
{
  alignas(T...) char a;
};

template <int... args>
struct alignment2
{
  alignas(args...) char a;
};
```

```
12
13 alignment1<int, double> al1;
14 alignment2<1, 4, 8> al2;
```

- 属性列表: 目前还没有编译器支持。

现在，已经了解了更多关于参数包及其扩展的知识，可以继续探索可变参数类模板。

3.4. 可变参数类模板

类模板也可以有数量可变的模板参数，这是构建标准库中某些类型类别 (如 tuple 和 variant) 的关键。本节中，将了解如何为 tuple 类编写一个简单的实现。tuple 是一种表示固定大小的异构值集合的类型。

实现可变函数模板时，使用了带有两个重载的递归模式，一个用于一般情况，另一个用于结束递归。除了为此目的需要使用特化外，可变参数类模板也必须采用相同的方法。下面是一个 tuple 类的最小实现:

```
1  template <typename T, typename... Ts>
2  struct tuple
3  {
4    tuple(T const& t, Ts const &... ts)
5    : value(t), rest(ts...)
6    {
7    }
8
9    constexpr int size() const { return 1 + rest.size(); }
10
11   T value;
12   tuple<Ts...> rest;
13 };
14
15 template <typename T>
16 struct tuple<T>
17 {
18   tuple(const T& t)
19   : value(t)
20   {
21   }
22
23   constexpr int size() const { return 1; }
24
25   T value;
26 };
```

第一个类是主模板，有两个模板参数: 一个类型模板和一个参数包。所以，至少必须为实例化该模板指定一种类型。主模板元组有两个成员变量:value，类型为 T, rest 的类型为 tuple<Ts…>，这是模板其余参数的扩展。说明包含 N 个元素的元组将包含第一个元素和另一个 tuple; 这第二个 tuple，依次包含第二个元素和另一个 tuple; 这第三个嵌套 tuple 包含其余内容。这种模式一直持续下去，直

到得到只有一个元素的 tuple，这是由偏特化 tuple<T> 定义。与主模板不同，此特化不聚合另一个 tuple 对象。

可以使用这个简单的实现来编写如下代码:

```
tuple<int> one(42);
tuple<int, double> two(42, 42.0);
tuple<int, double, char> three(42, 42.0, 'a');

std::cout << one.value << '\n';
std::cout << two.value << ','
          << two.rest.value << '\n';
std::cout << three.value << ','
          << three.rest.value << ','
          << three.rest.rest.value << '\n';
```

虽然可行，但需要通过 rest 成员访问元素，例如 three.rest.rest.value，还是很繁琐的。而且 tuple 的元素越多，用这种方式写代码就越难。因此，我们希望使用一些辅助函数来简化 tuple 元素的访问:

```
std::cout << get<0>(one) << '\n';
std::cout << get<0>(two) << ','
          << get<1>(two) << '\n';
std::cout << get<0>(three) << ','
          << get<1>(three) << ','
          << get<2>(three) << '\n';
```

get<N> 是一个可变参函数模板，接受一个 tuple 作为参数，并返回对元组中第 N 个索引处元素的引用。其原型可能如下所示:

```
template <size_t N, typename... Ts>
typename nth_type<N, Ts...>::value_type & get(tuple<Ts...>& t);
```

模板参数是元组类型的索引和形参包，但其实现需要一些帮助器类型。首先，需要了解 tuple 中第 N 个下标处元素的类型。可以在以下 nth_type 可变类型模板的帮助下进行检索:

```
template <size_t N, typename T, typename... Ts>
struct nth_type : nth_type<N - 1, Ts...>
{
  static_assert(N < sizeof...(Ts) + 1,
                "index out of bounds");
};

template <typename T, typename... Ts>
struct nth_type<0, T, Ts...>
{
  using value_type = T;
};
```

同样，有一个主模板，使用递归继承和索引 0 的特化。特化为第一个类型模板定义了一个名为 value_type 的别名 (是模板参数列表的头)，此类型仅用作确定 tuple 元素类型的机制，需要另一个可变参数类模板来检索值:

```
template <size_t N>
struct getter
{
  template <typename... Ts>
  static typename nth_type<N, Ts...>::value_type&
  get(tuple<Ts...>& t)
  {
    return getter<N - 1>::get(t.rest);
  }
};

template <>
struct getter<0>
{
  template <typename T, typename... Ts>
  static T& get(tuple<T, Ts...>& t)
  {
    return t.value;
  }
};
```

可以看到相同的递归模式，有一个主模板和一个显式特化。类模板称为 getter，只有一个模板参数，这是非类型模板参数，这个参数表示要访问的 tuple 元素的索引。这个类模板有一个名为 get 的静态成员函数，是一个可变参数函数模板。主模板中的实现以 tuple 的其余成员作为参数调用 get 函数，并且显式特化的实现会返回对元组成员值的引用。

定义了这些之后，现在可以为辅助可变参数函数模板 get 提供一个实现。这个实现依赖于 getter 类模板，并调用其 get 可变参数函数模板:

```
template <size_t N, typename... Ts>
typename nth_type<N, Ts...>::value_type &
get(tuple<Ts...>& t)
{
  return getter<N>::get(t);
}
```

若这个例子有点复杂，那么对其一步步地分析可能会有助于读者们更好地理解其工作原理。先从以下片段开始:

```
tuple<int, double, char> three(42, 42.0, 'a');
get<2>(three);
```

使用 cppinsights.io Web 工具来检查，这个代码段中发生的模板实例化。首先要查看的是类模板元组，有一个主模板和几个特化:

```
template <typename T, typename... Ts>
struct tuple
{
  tuple(T const& t, Ts const &... ts)
    : value(t), rest(ts...)
```

```
  { }

  constexpr int size() const { return 1 + rest.size(); }

  T value;
  tuple<Ts...> rest;
};

template<> struct tuple<int, double, char>
{
  inline tuple(const int & t,
               const double & __ts1, const char & __ts2)
  : value{t}, rest{tuple<double, char>(__ts1, __ts2)}
  {}

  inline constexpr int size() const;

  int value;
  tuple<double, char> rest;
};

template<> struct tuple<double, char>
{
  inline tuple(const double & t, const char & __ts1)
  : value{t}, rest{tuple<char>(__ts1)}
  {}

  inline constexpr int size() const;

  double value;
  tuple<char> rest;
};

template<> struct tuple<char>
{
  inline tuple(const char & t)
  : value{t}
  {}

  inline constexpr int size() const;

  char value;
};

template<typename T>
struct tuple<T>
{
  inline tuple(const T & t) : value{t}
  { }
```

```
55
56    inline constexpr int size() const
57    { return 1; }
58
59    T value;
60  };
```

tuple<int, double, char> 结构包含一个 int 和一个 tuple<double, char>；tuple<double, char> 包含一个 double 和一个 tuple<char>；tuple<char> 又包含一个 char 值，最后一个类表示元组的递归定义的结束。可以用以下图形表示:

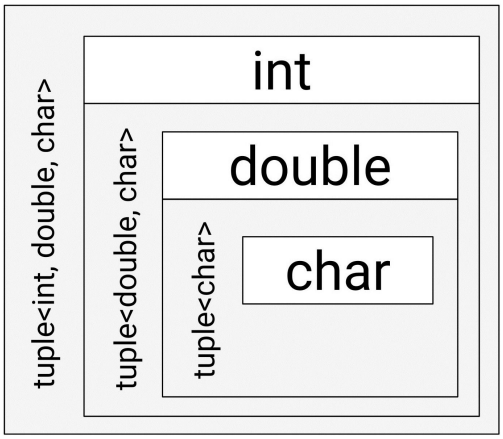


图 3.1 - 元组的示例

接下来，是 nth_type 类模板，一个主模板和几个特化:

```
1  template <size_t N, typename T, typename... Ts>
2  struct nth_type : nth_type<N - 1, Ts...>
3  {
4    static_assert(N < sizeof...(Ts) + 1,
5                  "index out of bounds");
6  };
7
8  template<>
9  struct nth_type<2, int, double, char> :
10   public nth_type<1, double, char>
11 { };
12
13 template<>
14 struct nth_type<1, double, char> : public nth_type<0, char>
15 { };
16
17 template<>
18 struct nth_type<0, char>
19 {
20   using value_type = char;
21 };
22
23 template<typename T, typename ... Ts>
```

```
struct nth_type<0, T, Ts...>
{
  using value_type = T;
};
```

nth_type<2, int, double, char> 特化派生自 nth_type<1, double, char>，而 nth_type<1, double, char> 又派生自 nth_type<0, char>，nth_type<0, char> 是层次结构中的最后一个基类 (递归层次结构的末尾)。

nth_type 结构体用作 getter 辅助类模板中的返回类型，实例化如下所示:

```
template <size_t N>
struct getter
{
  template <typename... Ts>
  static typename nth_type<N, Ts...>::value_type&
  get(tuple<Ts...>& t)
  {
    return getter<N - 1>::get(t.rest);
  }
};

template<>
struct getter<2>
{
  template<>
  static inline typename
  nth_type<2UL, int, double, char>::value_type &
  get<int, double, char>(tuple<int, double, char> & t)
  {
    return getter<1>::get(t.rest);
  }
};

template<>
struct getter<1>
{
  template<>
  static inline typename nth_type<1UL, double,
                                  char>::value_type &
  get<double, char>(tuple<double, char> & t)
  {
    return getter<0>::get(t.rest);
  }
};
template<>
struct getter<0>
{
  template<typename T, typename ... Ts>
  static inline T & get(tuple<T, Ts...> & t)
  {
    return t.value;
```

```
  }

  template<>
  static inline char & get<char>(tuple<char> & t)
  {
    return t.value;
  }
};
```

最后，检索 tuple 元素值的 get 函数模板定义:

```
template <size_t N, typename... Ts>
typename nth_type<N, Ts...>::value_type &
get(tuple<Ts...>& t)
{
  return getter<N>::get(t);
}

template<>
typename nth_type<2UL, int, double, char>::value_type &
get<2, int, double, char>(tuple<int, double, char> & t)
{
  return getter<2>::get(t);
}
```

若有更多对 get 函数的调用，get 的特化就会更多。例如，对于 get<1>(3)，将添加以下特化:

```
template<>
typename nth_type<1UL, int, double, char>::value_type &
get<1, int, double, char>(tuple<int, double, char> & t)
{
  return getter<1>::get(t);
}
```

这个例子演示了如何使用用于一般情况的主模板，和一个用于可变参数递归的结束情况的特化来实现可变参数类模板。

这里使用关键字 typename 作为 nth_type<N, Ts...>::value_type 类型的前缀，是一个依赖类型。C++20 中，这个不再必要，这个主题将在第 4 章中详细讨论。

由于实现可变参数模板通常是冗长且繁琐的，C++17 标准添加了折叠表达式来进行简化。我们将在下一节中来探讨折叠表达式。

3.5. 折叠表达式

折叠表达式是一种包含参数包的表达式，将参数包中的元素折叠 (或减少) 到二进制运算符上。为了了解其工作原理，来看几个例子。之前，实现了一个名为 sum 的变量函数模板，返回所有提供参数的和。方便起见，在这里再次展示其实现:

```
template <typename T>
T sum(T a)
```

```
{
  return a;
}

template <typename T, typename... Args>
T sum(T a, Args... args)
{
  return a + sum(args...);
}
```

使用折叠表达式，需要两次重载的实现可以简化为以下形式:

```
template <typename... T>
int sum(T... args)
{
  return (... + args);
}
```

不再需要重载函数了。表达 (⋯+ args) 表示折叠表达式，计算后变为 ((((arg0 + arg1) + arg2) +⋯)+ argN)。括号是折叠表达式的一部分，可以使用这个新实现，就像使用最初的实现一样:

```
int main()
{
  std::cout << sum(1) << '\n';
  std::cout << sum(1,2) << '\n';
  std::cout << sum(1,2,3,4,5) << '\n';
}
```

折叠方式有四种:

折叠方式	语法	展开方式
一元右折叠	(pack op ...)	(arg1 op (... op (argN-1 op argN)))
一元左折叠	(... op pack)	(((arg1 op arg2) op ...) op argN)
二元右折叠	(pack op ... op init)	(arg1 op (... op (argN-1 op (argN op init))))
二元左折叠	(init op ... op pack)	((((init op arg1) op arg2) op ...) op argN)

表 3.1

该表中，使用了以下名词:

- pack 是一个包含未展开形参包的表达式，arg1、arg2、argN-1 和 argN 是这个包中包含的参数。
- op 下面是一个二元操作符: +, -, *, /, %, ^, &, |, =, <, >, <<, >>, +=, -=, *=, /=, %=, ^=, &=, |=, <<=, >>=, ==, !=, <=, >=, &&, ||, ,(逗号表达式), .*, ->*.
- init 不包含未展开参数包的表达式。

一元折叠中，若参数包不包含任何元素，则只允许使用某些操作符。下表列出了这些值，以及空参数包的值:

操作符	空参数包的值
&& (逻辑 AND)	true
|| (逻辑 OR)	false
, (逗号操作符)	void()

表 3.2

一元折叠和二元折叠的不同之处在于初始化值的使用，初始化值只适用于二元折叠。二元折叠将二元操作符重复两次 (必须是同一个操作符)。通过包含初始化值，可以将可变函数模板和从使用一元右折叠表达式转换为使用二元右折叠表达式。这里有一个例子:

```
template <typename... T>
int sum_from_zero(T... args)
{
  return (0 + ... + args);
}
```

貌似 sum 和 sum_from_zero 的函数模板没有区别，但事实并非如此。考虑以下情况:

```
int s1 = sum(); // error
int s2 = sum_from_zero(); // OK
```

调用不带参数的 sum 将产生编译器错误，一元折叠表达式 (加法运算符) 必须有非空展开。然而，二进制折叠表达式没有这个问题，因此调用 sum_from_zero 而不带参数可以工作，函数将返回 0。

这两个使用 sum 和 sum_from_zero 的例子中，参数包 args 直接出现在折叠表达式中。其可以是表达式的一部分，只要没有完全展开:

```
template <typename... T>
void printl(T... args)
{
  (..., (std::cout << args)) << '\n';
}

template <typename... T>
void printr(T... args)
{
  ((std::cout << args), ...) << '\n';
}
```

参数包 args 是 (std::cout << args) 表达式的一部分，这不是一个折叠表达式。折叠表达式是 ((std::cout << args)，…)。这实际是逗号运算符上的一元左折叠。printl 和 printr 函数可以像下面这样使用:

```
printl('d', 'o', 'g'); // dog
printr('d', 'o', 'g'); // dog
```

这两种情况下，输出到控制台的文本都是 dog。这是因为一元左折叠扩展为 (((std::cout << 'd'), std::cout << 'o'), « std::cout << 'g')，一元右折叠扩展为 (std::cout << 'd', (std::cout << 'o', (std::cout

<< 'g'))，因为由逗号分隔的一对表达式从左到右求值，所以这两种计算方式相同 (内置逗号操作符就是这样)。对于重载逗号操作符的类型，其行为取决于如何重载该操作符。然而，重载逗号操作符的情况非常少 (例如简化多维数组索引)，如 Boost 库。赋值和 SOCI 会重载逗号运算符，但通常情况下，最好不要去重载逗号操作符。

来看一下在折叠表达式中的表达式中使用参数包的另一个示例。下面的可变参数函数模板，会将多个值插入到 std::vector 的末尾:

```
template<typename T, typename... Args>
void push_back_many(std::vector<T>& v, Args&&... args)
{
  (v.push_back(args), ...);
}

push_back_many(v, 1, 2, 3, 4, 5); // v = {1, 2, 3, 4, 5}
```

参数包 args 与 v.push_back(args) 表达式一起使用，该表达式折叠在逗号操作符上，一元左折叠表达式为 (v.push_back(args)，…)。

与使用递归实现可变参数模板相比，折叠表达式有几个优点:

- 代码更少、更简单。
- 更快的编译时间，更少的模板实例化。
- 可能更快的代码，因为多个函数调用替换为单个表达式。不过，这一点有待验证，至少在启用优化时不是。我们已经看到编译器会通过删除这些函数来优化代码。

现在已经了解了如何创建可变参数函数模板、可变参数类模板，以及如何使用折叠表达式，接下来将讨论其他类型的可变参数模板: 别名模板和变量模板。

3.6. 可变参数别名模板

所有可以模板化的东西也可以变参数化，别名模板是一组类型的别名 (另一个名称)。可变别名模板是具有可变数量模板参数的类型族的名称。编写别名模板会相当简单的，先来看一个例子:

```
template <typename T, typename... Args>
struct foo
{
};

template <typename... Args>
using int_foo = foo<int, Args...>;
```

类模板 foo 是可变参数的，并且至少接受一个类型模板参数。int_foo 只是一个名称，用于从 foo 类型实例化的类型，并且 int 作为第一个类型模板参数的模板:

```
foo<double, char, int> f1;
foo<int, char, double> f2;
int_foo<char, double> f3;
static_assert(std::is_same_v<decltype(f2), decltype(f3)>);
```

这段代码中，f1 和 f2 和 f3 是不同 foo 类型的实例，都是从 foo 的不同模板参数集实例化的。然而，f2 和 f3 是同一类型的实例，所以 int_foo<char, double> 是 foo<int, char, double> 类型的别名。

前面介绍了一个类似的例子，尽管有点复杂。标准库包含一个名为 std::integer_sequence 的类模板，表示一个编译时的整数序列，以及一堆别名模板，从而帮助创建各种类型的此类整数序列。虽然这是一个简化的代码段，在概念上的实现如下所示:

```
template<typename T, T... Ints>
struct integer_sequence
{};

template<std::size_t... Ints>
using index_sequence = integer_sequence<std::size_t,
                        Ints...>;

template<typename T, std::size_t N, T... Is>
struct make_integer_sequence :
  make_integer_sequence<T, N - 1, N - 1, Is...>
{};

template<typename T, T... Is>
struct make_integer_sequence<T, 0, Is...> :
  integer_sequence<T, Is...>
{};

template<std::size_t N>
using make_index_sequence = make_integer_sequence<std::size_t,
                            N>;

template<typename... T>
using index_sequence_for =
  make_index_sequence<sizeof...(T)>;
```

这里有三个别名模板:

- index_sequence, 为 size_t 类型创建 integer_sequence; 这是一个可变别名模板。
- index_sequence_for, 从参数包中创建 integer_sequence; 这也是一个可变别名模板。
- make_index_sequence, 为 size_t 类型创建了 integer_sequence，值为 0,1,2，…，N-1。与前面的模板不同，这不是可变参数模板的别名。

本章要讨论的最后一个主题是可变变量模板。

3.7. 可变参数变量模板

如前所述，变量模板也有可变参数。但变量不能递归定义，也不能像类模板那样特化。折叠表达式简化了从可变数量的参数生成表达式的过程，对于创建可变参数的变量模板非常方便。

下面的例子中，定义了一个名为 Sum 的可变变量模板，在编译时初始化为所有作为非类型模板参数提供的整数的和:

```cpp
template <int... R>
constexpr int Sum = (... + R);
int main()
{
  std::cout << Sum<1> << '\n';
  std::cout << Sum<1,2> << '\n';
  std::cout << Sum<1,2,3,4,5> << '\n';
}
```

这类似于用折叠表达式写的和函数，要添加的数字作为函数参数提供。这里，作为模板参数提供给变量 template。区别主要是句法上的; 启用优化后，最终结果在生成的汇编代码和性能方面可能相同。

可变参数变量模板与所有其他类型的模板遵循相同的模式，尽管不像其他模板那么常用。结束这个主题后，目前已经完成了对 C++ 中可变参数模板的了解。

3.8. 总结

本章中，探索了一个重要的模板类别，可变参数模板，这是具有可变数量模板参数的模板。我们可以创建可变函数模板、类模板、变量模板和别名模板。创建可变参数函数模板和可变参数类模板的技术不同，但会引起一种编译时递归。对于后者，通过模板特化完成；而对于前者，通过函数重载完成。折叠表达式有助于将可变数量的参数扩展为单个表达式，避免使用函数重载的需要，并支持创建一些类别的可变变量模板。

下一章中，我们将研究一系列更高级的特性，这些特性将巩固对于模板的了解。

3.9. 习题

1. 什么是可变参数模板，这种模板有什么用?
2. 什么是参数包?
3. 什么是可以展开参数包的上下文?
4. 什么是折叠表达式?
5. 使用折叠表达式有什么好处?

3.10. 扩展阅读

- C++ Fold Expressions 101, Jonathan Boccara, https://www.fluentcpp.com/2021/03/12/cpp-fold-expressions/
- Fold Expressions in C++ 17, Vaibhav, https://mainfunda.com/foldexpressions-in-cpp17/
- Nifty Fold Expression Tricks, Jonathan Müller, https://www.foonathan.net/2020/05/fold-tricks/

第二部分：模板进阶特性

本部分将探索各种高级特性，包括名称绑定和依赖名称、模板递归、模板参数推导和转发引用。将了解帮助查询类型信息，以及使用各种语言特性执行条件编译的类型特征。此外，还将了解如何使用 C++20 的概念和约束来指定模板参数的要求，并探索标准概念库的内容。

本部分包括以下章节:

- 第 *4* 章，高级模板概念
- 第 *5* 章，类型特征和条件编译
- 第 *6* 章，概念和约束

第 4 章 高级模板概念

前面的章节中，了解了 C++ 模板的核心基础知识。现在，阅读过之前章节的读者应该能够编写简单的模板了。不过，关于模板还有更多的细节，本章专门讨论这些更高级的主题。

我们接下来要讨论以下主题:

- 名称绑定和依赖名称
- 模板递归
- 模板参数推导
- 转发引用和完美转发
- decltype 说明符和 std::declval 类型操作符
- 模板间的“友情”

完成本章后，将对这些高级模板概念有更深入的了解，并能够理解和编写更复杂的模板代码。

4.1. 名称绑定和依赖名称

“名称绑定”指的是查找模板中使用的每个名称的声明的过程。模板中使用两种名称: 依赖名称和非依赖名称。前者是依赖模板参数的类型或值的名称，可以是类型参数、非类型形参或模板参数。不依赖于模板参数的名称称为非依赖名称。依赖名称和非依赖名称的查找方式不同:

- 依赖名称，在模板实例化时执行。
- 非依赖名称，则在模板定义时执行。

首先，来看看非依赖名称，位于模板定义的前面。先来看一下下面的例子:

```
template <typename T>
struct parser; // [1] template declaration
void handle(double value) // [2] handle(double) definition
{
  std::cout << "processing a double: " << value << '\n';
}

template <typename T>
struct parser // [3] template definition
{
  void parse()
  {
    handle(42); // [4] non-dependent name
  }
};

void handle(int value) // [5] handle(int) definition
{
  std::cout << "processing an int: " << value << '\n';
}

int main()
```

```
{
  parser<int> p; // [6] template instantiation
  p.parse();
}
```

注释中有几个参考点。[1] 声明了一个名为 parser 的类模板，[2] 定义一个名为 handle 的函数，该函数以 double 作为参数。类模板的定义在 [3]。该类包含一个名为 run 的方法，该方法调用一个名为 handle 的函数，其参数值为 42，位于 [4]。

handle 是一个非依赖名称，因为它不依赖于任何模板参数，所以此处执行名称查找和绑定。handle 必须是 [3] 已知的函数，[2] 上定义的函数是唯一匹配的。定义类模板之后 ([5])，就有了函数 handle 的重载定义，该函数 handle 以整数作为参数。这是 handle(42) 更好的匹配候选，但它是在执行名称绑定之后出现的，因此将被忽略。main 函数中 ([6])，有 int 类型的解析器类模板的实例化。调用 run 函数时，会将”processing adouble: 42” 输出至控制台。

下一个示例就来介绍依赖名称的概念:

```
template <typename T>
struct handler // [1] template definition
{
  void handle(T value)
  {
    std::cout << "handler<T>: " << value << '\n';
  }
};

template <typename T>
struct parser // [2] template definition
{
  void parse(T arg)
  {
    arg.handle(42); // [3] dependent name
  }
};

template <>
struct handler<int> // [4] template specialization
{
  void handle(int value)
  {
    std::cout << "handler<int>: " << value << '\n';
  }
};

int main()
{
  handler<int> h; // [5] template instantiation
  parser<handler<int>> p; // [6] template instantiation
  p.parse(h);
}
```

这个示例与前面的示例略有不同。parser 类模板非常相似，但 handle 函数已经成为另一个类模板的成员。

注释 [1]，有一个名为 handler 的类模板的定义。包含一个名为 handle 的公共方法，该方法接受 T 类型的参数，并将其值输出到控制台。接下来，[2] 有 parser 类模板的定义。这与前一个类似，除了 [3] 在其参数上调用一个名为 handle 的方法。因为实参的类型是模板形参 T，其使 handle 成为依赖名称。依赖名称在模板实例化时查找，因此句柄此时没有绑定。[4] 有 int 类型的处理程序类模板的模板特化，这是与依赖名称更好的匹配。因此，当模板实例化发生在 [6] 时，handler<int>::handle 会绑定到 [3]，使用的依赖名称的名称。运行此程序，控制台将会输出”handler<int>: 42”。

现在已经了解了名称绑定是如何发生的，接下来来了解它与模板实例化之间的关系。

4.1.1 两阶段的名称查找

上一节的关键内容是，名称查找对于依赖名称 (依赖于模板参数的名称) 和非依赖名称 (不依赖于模板参数的名称，加上模板名称和当前模板实例化中定义的名称) 不同。当编译器遍历模板定义时，需要判断名称是依赖的还是非依赖的，高阶名称查找依赖于这种分类，并且发生在模板定义点 (对于非依赖名称) 或模板实例化点 (对于依赖名称)。因此，模板的实例化会分为两个阶段:

- 第一个阶段发生在定义时，检查模板语法并将名称分类为依赖或非依赖。
- 第二个阶段发生在实例化时，此时模板实参替换为模板参数。依赖名称的绑定这时发生。

这个分为两步的过程称为**两阶段名称查找**，来看一个例子:

```
template <typename T>
struct base_parser
{
  void init()
  {
    std::cout << "init\n";
  }
};

template <typename T>
struct parser : base_parser<T>
{
  void parse()
  {
    init(); // error: identifier not found
    std::cout << "parse\n";
  }
};

int main()
{
  parser<int> p;
  p.parse();
}
```

代码中有两个类模板:base_parser 和 parser，前者包含名为 init 的公共方法，后者派生自 base_parser，并包含一个名为 parse 的方法。parse 成员函数调用了一个名为 init 的函数，目的是在这里调用的是基类方法 init。然而，编译器将报错，因为它无法找到 init。发生这种情况的原因是 init 是一个不依赖的名称 (不依赖于模板参数)。因此，必须在定义解析器模板时就知道。尽管 base_parser<T>::init 存在，但编译器不能假定它是我们想要的，因为主模板 base_parser 可以稍后进行特化，而 init 可以定义为其他东西 (例如：类型、变量或另一个函数)。因此，名称查找不会发生在基类中，而只发生在其外围作用域中，并且 parser 中没有名为 init 的函数。

这个问题可以通过将 init 设置为依赖名称来解决。这可以通过添加前缀 this-> 或 base_parser<T>:: 来实现。将 init 转换为依赖名称，其名称绑定将从模板定义点移动到模板实例化点。下面的代码段中，是通过 this 指针来解决问题的:

```
template <typename T>
struct parser : base_parser<T>
{
  void parse()
  {
    this->init(); // OK
    std::cout << "parse\n";
  }
};
```

继续这个例子，在定义 parser 类模板之后，int 类型的 base_parser 的特化可用。如下所示:

```
template <>
struct base_parser<int>
{
  void init()
  {
    std::cout << "specialized init\n";
  }
};
```

此外，看看 parser 类模板的如下用法:

```
int main()
{
  parser<int> p1;
  p1.parse();
  parser<double> p2;
  p2.parse();
}
```

运行这个程序时，下面的文本将输出到控制台:

```
specialized init
parse
init
parse
```

出现这种行为的原因是 p1 是 parser<int> 的实例，并且其基类 base_parser<int> 实现了 init 函数，并将特化的 init 打印到控制台。另一方面，p2 是 parser<double> 的实例。由于 double 类型的 base_parser 的特化不可用，因此将调用主模板中的 init 函数，并且只将 init 输出到控制台。

下一个主题就是如何使用依赖名称 (即类型)。

4.1.2 依赖类型的名称

目前的例子中，依赖名称是函数或成员函数。但在某些情况下，依赖名称是类型:

```
template <typename T>
struct base_parser
{
  using value_type = T;
};

template <typename T>
struct parser : base_parser<T>
{
  void parse()
  {
    value_type v{}; // [1] error
    // or
    base_parser<T>::value_type v{}; // [2] error
    std::cout << "parse\n";
  }
};
```

这个代码段中，base_parser 是类模板，为 T 定义了名为 value_type 的类型别名。parser 类模板派生自 base_parser，需要在其 parser 方法中使用这种类型。然而，value_type 和 base_parser<T>::value_type 都不起作用，编译器会报错。value_type 无效，因为它是一个不依赖的名称，因此不会在基类中进行查找，只能在外围作用域中查找。base_parser<T>::value_type 也不能工作，因为编译器不能假设这是一个实际类型。base_parser 的特化可能紧随其后，所以 value_type 不一定是一种类型。

为了解决这个问题，需要告诉编译器这个名称指向的类型。否则，编译器默认会假定它不是类型。这可通过 typename 关键字在定义点完成，如下所示:

```
template <typename T>
struct parser : base_parser<T>
{
  void parse()
  {
    typename base_parser<T>::value_type v{}; // [3] OK
    std::cout << "parse\n";
  }
};
```

实际上，这条规则有两个例外:

- 指定基类时
- 初始化类成员时

来看看这两个例外:

```
struct dictionary_traits
{
  using key_type = int;
  using map_type = std::map<key_type, std::string>;
  static constexpr int identity = 1;
};

template <typename T>
struct dictionary : T::map_type // [1]
{
  int start_key { T::identity }; // [2]
  typename T::key_type next_key; // [3]
};

int main()
{
  dictionary<dictionary_traits> d;
}
```

dictionay_traits 是一个类，用作 dictionary 类 template 的模板参数。这个类派生于 T::map_type(参见第 [1] 行)，但不需要使用 typename 关键字。dictionary 类定义了一个名为 start_key 的成员，是一个 int 型，初始化值为 T::identity(参见第 [2] 行)。同样，这里不需要 typename 关键字。然而，若想定义类型 T::key_type 的另一个成员 (见 [3] 行)，需要使用 typename。

C++20 中，对使用 typename 的要求已经放宽了，从而更容易使用类型名。编译器现在能够推断出我们在许多上下文中引用的是类型名。例如，在 [3] 行上那样定义成员变量时，不再需要使用 typename 关键字作为前缀了。

C++20 中，typename 在以下情形中是隐式的 (可以由编译器推导):

- 使用声明时
- 数据成员声明中
- 函数参数的声明或定义中
- 尾部返回类型中
- 模板类型参数的默认类型中
- static_cast、const_cast、reinterpret_cast 或 dynamic_cast 语句的 type-id 中

以下代码段举例说明了其中一些情况:

```
template <typename T>
struct dictionary : T::map_type
{
  int start_key{ T::identity };
  T::key_type next_key; // [1]

  using value_type = T::map_type::mapped_type; // [2]

  void add(T::key_type const&, value_type const&) {} // [3]
};
```

这个代码段中，[1]、[2] 和 [3] 标记的所有行中，C++20 之前，需要 typename 关键字来指示类型名称 (例如 T::key_type 或 T::map_type::mapped_type)。当使用 C++20 编译时，这就不再需要了。

> **Note**
>
> 第 2 章中，我们已经看到关键字 typename 和 class 可以用来引入类型模板参数，而且是可互换的。这里的关键字 typename 虽然具有类似的使用方式，但不能用 class 替换。

不仅类型可以是依赖的名称，其他模板也可以。

4.1.3 依赖模板的名称

某些情况下，依赖名称是模板，例如函数模板或类模板。但编译器的默认行为是将依赖项名称解释为非类型，这会导致使用小于比较操作符时出现错误:

```
template <typename T>
struct base_parser
{
  template <typename U>
  void init()
  {
    std::cout << "init\n";
  }
};

template <typename T>
struct parser : base_parser<T>
{
  void parse()
  {
    // base_parser<T>::init<int>(); // [1] error
    base_parser<T>::template init<int>(); // [2] OK
    std::cout << "parse\n";
  }
};
```

这类似于前面的代码，base_parser 中的 init 函数也是一个模板。尝试使用 base_parser<T>::init<int>()，如 [1] 所示，将导会致编译器报错，所以必须使用 template 关键字告诉编译器依赖名称是模板，如 [2] 所示。

template 关键字只能跟随作用域解析操作符 (::)、通过指针 (->) 进行成员访问和成员访问 (.)。正确用法的例子为 X::template foo<T>()，this->template foo<T>() 和 obj.template foo()。

依赖名称不一定是函数模板，也可以是一个类模板:

```
template <typename T>
struct base_parser
{
  template <typename U>
  struct token {};
```

```
};

template <typename T>
struct parser : base_parser<T>
{
  void parse()
  {
    using token_type =
      base_parser<T>::template token<int>; // [1]
    token_type t1{};

    typename base_parser<T>::template token<int> t2{};
                          // [2]
    std::cout << "parse\n";
  }
};
```

token 类是 base_parser 类模板的内部类模板，既可以在 [1] 行中使用，定义了类型别名 (然后用于实例化对象)；也可以在 [2] 行中使用，直接用于声明变量。typename 关键字在 [1] 中是不必要的，其中 using 声明表示正在处理类型，因为编译器会假定其为非类型名称，所以在 [2] 中是必需的。

观察当前模板实例化的某些上下文中，并不需要使用 typename 和 template 关键字。

4.1.4 实例化

类模板定义的上下文中，可以避免使用 typename 和 template 关键字来消除依赖名称的歧义。在类模板定义的上下文中，编译器能够推导出一些依赖名称 (例如嵌套类的名称) 来引用当前实例化，所以一些错误可以在定义时 (而不是实例化时) 就可以找出来。

根据 C++ 标准 §13.8.2.1-依赖类型，可以引用当前实例化名称的完整列表如下所示:

内容	名称
类模板定义	嵌套类 类模板的成员 嵌套类的成员 注入模板的类名 注入的嵌套类的类名
主类模板定义 或 定义主类模板的成员	类模板的名称，后面跟着主模板 的模板实参列表，其中每个实参都 等效于其对应的形参
嵌套类或类模板的定义	用作当前实例化成员的嵌套类的名称
偏特化的定义 或 偏特化成员的定义	类模板的名称，后面跟着偏特化 的模板参数列表，其中每个参数 等效于其相应的形参

表 4.1

以下是将名称作为当前实例化的部分规则:

- 当前实例化或其非依赖基类中找到的非限定名称 (不在作用域解析操作符::) 的右侧
- 限定名 (出现在范围解析操作符的右侧::)，若其限定符 (出现在范围解析操作符左侧的部分) 命名当前实例化，并且在当前实例化或其非依赖基类中找到
- 类成员访问表达式中使用的名称，其中对象表达式是当前实例化，且名称在当前实例化或其非依赖基类中找到

Note

若基类是依赖类型 (依赖于模板形参) 并且不在当前实例化中，那么基类就是依赖类。否则，基类为非依赖类。

这些规则听起来可能有点难以理解，以下几个例子可能有助于对其进行理解:

```
template <typename T>
struct parser
{
  parser* p1; // parser is the CI
  parser<T>* p2; // parser<T> is the CI
  ::parser<T>* p3; // ::parser<T> is the CI
  parser<T*> p4; // parser<T*> is not the CI

  struct token
  {
    token* t1; // token is the CI
    parser<T>::token* t2; // parser<T>::token is the CI
    typename parser<T*>::token* t3;
    // parser<T*>::token is not the CI
  };
};

template <typename T>
struct parser<T*>
{
  parser<T*>* p1; // parser<T*> is the CI
  parser<T>* p2; // parser<T> is not the CI
};
```

主模板 parser 中，名称 parser、parser<T> 和::parser<T> 都指向当前实例化，但 parser<T*> 没有。token 类是主模板 parser 的嵌套类。该类的范围内，token 和 parser<T>::token 都表示当前实例化，但对于 parser<T*>::token 则不是这样。该代码段还包含指针类型 T* 的主模板的偏特化。在这个偏特化的上下文中，parser<T*> 是当前实例化，而 parser<T> 不是。

依赖名称对于模板编程很重要。本节的关键内容是将名称分为依赖名称 (依赖于模板参数的名称) 和非依赖名称 (不依赖于模板参数的名称)。名称绑定发生在非依赖类型的定义点和依赖类型的实例化点。某些情况下，需要关键字 typename 和 template 来消除名称使用的歧义，并告诉编译器名称指的是类型或模板。然而，在类模板定义的上下文中，编译器能够找出一些依赖名称指向当前实例化，这使它能够更快地找到错误。

下一节中，我们将把注意力转移到模板递归上。

4.2. 模板递归

第 3 章中，我们讨论了可变参数模板，其是用一种看起来像递归的机制实现的。实际上，是重载函数和类模板的特化。可以创建递归模板，为了演示其是如何工作的，我们将研究如何实现阶乘函数的编译时版本。这通常会以递归的方式实现，可能的实现如下所示:

```
constexpr unsigned int factorial(unsigned int const n)
{
  return n > 1 ? n * factorial(n - 1) : 1;
}
```

这应该很容易理解: 通过递归地调用函数和递减的参数，返回函数参数与返回值相乘的结果，若参数为 0 或 1，则返回值 1。参数的类型 (以及返回值) 是 unsigned int，以避免使用负整数。

为了在编译时计算阶乘函数的值，需要定义一个类模板，其中包含一个持有函数值的数据成员:

```
template <unsigned int N>
struct factorial
{
  static constexpr unsigned int value =
    N * factorial<N - 1>::value;
};

template <>
struct factorial<0>
{
  static constexpr unsigned int value = 1;
};

int main()
{
  std::cout << factorial<4>::value << '\n';
}
```

第一个定义是主模板，有一个非类型模板参数，表示需要计算其阶乘的值。该类包含一个名为 value 的静态 constexpr 数据成员，初始化的结果是参数 N 与阶乘类模板的值相乘，参数自减后实例化。递归需要一个初始值，这是由 (非类型模板参数的) 第一个参数 (0 号) 的显式特化提供的，成员值初始化为 1。

main 函数中遇到实例化 factorial<4>::value 时，编译器生成从 factorial<4> 到 factorial<0> 的所有递归实例化:

```
template<>
struct factorial<4>
{
  inline static constexpr const unsigned int value =
    4U * factorial<3>::value;
};
```

```
7
8  template<>
9  struct factorial<3>
10 {
11   inline static constexpr const unsigned int value =
12     3U * factorial<2>::value;
13 };
14
15 template<>
16 struct factorial<2>
17 {
18   inline static constexpr const unsigned int value =
19     2U * factorial<1>::value;
20 };
21
22 template<>
23 struct factorial<1>
24 {
25   inline static constexpr const unsigned int value =
26     1U * factorial<0>::value;
27 };
28
29 template<>
30 struct factorial<0>
31 {
32   inline static constexpr const unsigned int value = 1;
33 };
```

这些实例化中，编译器能够计算出数据成员阶乘 <N>::value 的值。当启用优化时，甚至不会生成此代码，但生成的常量将直接用于生成的汇编代码中。

阶乘类模板的实现相对简单，类模板基本上只是静态数据成员值的包装器。实际上，可以通过变量模板来避免这种情况:

```
1  template <unsigned int N>
2  inline constexpr unsigned int factorial = N * factorial<N - 1>;
3
4  template <>
5  inline constexpr unsigned int factorial<0> = 1;
6
7  int main()
8  {
9    std::cout << factorial<4> << '\n';
10 }
```

阶乘类模板和阶乘变量模板的实现之间有很多相似之处。对于变量模板，我们取出了数据成员值，并将其称为阶乘。另一方面，因为不需要访问 factorial<4>::value 中数据成员的值，这也会容易使用。

编译时计算阶乘还有第三种方法: 使用函数模板。可能的实现如下所示:

```cpp
template <unsigned int n>
constexpr unsigned int factorial()
{
  return n * factorial<n - 1>();
}

template<> constexpr unsigned int factorial<1>() {
                        return 1; }
template<> constexpr unsigned int factorial<0>() {
                        return 1; }

int main()
{
  std::cout << factorial<4>() << '\n';
}
```

可以看到主模板递归地调用 factorial 函数模板，并且对值 1 和 0 有两个完全的特化，都返回 1。

这三种不同的方法中哪一种是最好的可能不太好说，但阶乘模板的递归实例化的复杂性保持不变，这取决于模板的性质。下面的代码段增加了复杂性:

```cpp
template <typename T>
struct wrapper {};

template <int N>
struct manyfold_wrapper
{
  using value_type =
    wrapper<
      typename manyfold_wrapper<N - 1>::value_type>;
};

template <>
struct manyfold_wrapper<0>
{
  using value_type = unsigned int;
};

int main()
{
  std::cout <<
    typeid(manyfold_wrapper<0>::value_type).name() << '\n';
  std::cout <<
    typeid(manyfold_wrapper<1>::value_type).name() << '\n';
  std::cout <<
    typeid(manyfold_wrapper<2>::value_type).name() << '\n';
  std::cout <<
    typeid(manyfold_wrapper<3>::value_type).name() << '\n';
}
```

本例中有两个类模板。第一个称为 wrapper，有一个空实现 (并不重要)，但它表示某种类型的包

装器类 (或者更准确地说，某种类型的值)。第二个模板称为 manyfold_wrapper，表示一个包装器对一个类型的包装器进行多次包装。包装数量没有上限，但是下限有。0 值的全特化为 unsigned int 类型定义了一个名为 value_type 的成员类型。因此，manyfold_wrapper<1> 为 wrapper<unsigned int> 定义了一个名为 value_type 的成员类型，manyfold_wrapper<2> 为 wrapper<wrapper<unsigned int» 定义了一个名为 value_type 的成员类型，以此类推。在执行 main 函数时，会将输出以下内容到控制台:

```
unsigned int
struct wrapper<unsigned int>
struct wrapper<struct wrapper<unsigned int> >
struct wrapper<struct wrapper<struct wrapper<unsigned int> > >
```

C++ 标准没有为递归嵌套的模板实例化指定限制，但推荐限制为 1024 个。这只是一个建议，而不是要求，所以不同的编译器实现了不同的限制。VC++ 16.11 编译器的限制设置为 500，GCC 12 为 900，Clang 13 为 1024。超过此限制时将使编译器报错。以下是一些例子:

VC++:

```
fatal error C1202: recursive type or function dependency
context too complex
```

GCC:

```
fatal error: template instantiation depth exceeds maximum of
900 (use '-ftemplate-depth=' to increase the maximum)
```

Clang:

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

对于 GCC 和 Clang，可以使用编译器选项-ftemplate-depth=N 可以增加嵌套模板实例化的最大值，但 Visual C++ 编译器没有这种选项。

递归模板在编译时，可以以递归的方式解决一些问题。使用递归函数模板、变量模板，还是类模板，取决于具体问题或开发偏好。但深度模板递归的深度是有限制的，需要明智地使用模板递归。

本章要讨论的下一个高级主题是模板参数推导，包括函数和类。

4.3. 函数模板的参数推导

编译器有时可以从函数调用的上下文推导出模板参数，从而可以不用显式地指定模板参数类型。模板参数推断的规则更复杂，我们将在本节中探讨这个主题。

从一个简单的例子开始:

```
template <typename T>
void process(T arg)
{
  std::cout << "process " << arg << '\n';
}

int main()
{
  process(42); // [1] T is int
  process<int>(42); // [2] T is int, redundant
  process<short>(42); // [3] T is short
}
```

这段代码中，process 是一个具有单一类型模板形参的函数模板。process(42) 和 process<int>(42) 等价，在第一种情况下，编译器能够从传递给函数的参数值推断类型模板形参 T 的类型为 int。

当编译器试图推断模板参数时，会将模板形参的类型与用于调用函数的实参的类型的匹配。有一些规则可以控制这种匹配。编译器可以匹配以下内容:

- 形式为 T, T const, T volatile 的类型 (包括 cv-类型和 non-类型):

```
struct account_t
{
  int number;
};

template <typename T>
void process01(T) { std::cout << "T\n"; }

template <typename T>
void process02(T const) { std::cout << "T const\n"; }

template <typename T>
void process03(T volatile) { std::cout << "T volatile\n";
}

int main()
{
  account_t ac{ 42 };
  process01(ac); // T
  process02(ac); // T const
  process03(ac); // T volatile
}
```

- 指针 (T*)、左值引用 (T&) 和右值引用 (T&&):

```
template <typename T>
void process04(T*) { std::cout << "T*\n"; }

template <typename T>
```

```
void process04(T&) { std::cout << "T&\n"; }

template <typename T>
void process05(T&&) { std::cout << "T&&\n"; }

int main()
{
  account_t ac{ 42 };
  process04(&ac); // T*
  process04(ac); // T&
  process05(ac); // T&&
}
```

- 数组，如 T[5] 或 C[5][n]，其中 C 是类类型，n 是非类型模板参数:

```
template <typename T>
void process06(T[5]) { std::cout << "T[5]\n"; }

template <size_t n>
void process07(account_t[5][n])
{ std::cout << "C[5][n]\n"; }

int main()
{
  account_t arr1[5] {};
  process06(arr1); // T[5]

  account_t ac{ 42 };
  process06(&ac); // T[5]

  account_t arr2[5][3];
  process07(arr2); // C[5][n]
}
```

- 函数指针，形式为 T(*)()，C(*)(T) 和 T(*)(U)，其中 C 是类类型，T 和 U 是类型模板参数:

```
template<typename T>
void process08(T(*)()) { std::cout << "T (*)()\n"; }

template<typename T>
void process08(account_t(*)(T))
{ std::cout << "C (*) (T)\n"; }

template<typename T, typename U>
void process08(T(*)(U)) { std::cout << "T (*)(U)\n"; }

int main()
{
  account_t (*pf1)() = nullptr;
  account_t (*pf2)(int) = nullptr;
  double (*pf3)(int) = nullptr;
```

```
16
17    process08(pf1); // T (*)()
18    process08(pf2); // C (*)(T)
19    process08(pf3); // T (*)(U)
20  }
```

- 具有以下形式之一的成员函数的指针，T (C::*)()，T (C::*)(U)，T (U::*)()，T (U::*)(V)，C (T::*)()，C (T::*)(U) 和 D (C::*)(T)，其中 C 和 D 是类类型，T, U 和 V 是类型模板参数:

```
1   struct account_t
2   {
3     int number;
4     int get_number() { return number; }
5     int from_string(std::string text) {
6       return std::atoi(text.c_str()); }
7   };
8
9   struct transaction_t
10  {
11    double amount;
12  };
13
14  struct balance_report_t {};
15  struct balance_t
16  {
17    account_t account;
18    double amount;
19
20    account_t get_account() { return account; }
21    int get_account_number() { return account.number; }
22    bool can_withdraw(double const value)
23      {return amount >= value; };
24    transaction_t withdraw(double const value) {
25      amount -= value; return transaction_t{ -value }; }
26    balance_report_t make_report(int const type)
27    {return {}; }
28  };
29
30  template<typename T>
31  void process09(T(account_t::*)())
32  { std::cout << "T (C::*)()\n"; }
33
34  template<typename T, typename U>
35  void process09(T(account_t::*)(U))
36  { std::cout << "T (C::*)(U)\n"; }
37
38  template<typename T, typename U>
39  void process09(T(U::*)())
40  { std::cout << "T (U::*)()\n"; }
41
```

```
template<typename T, typename U, typename V>
void process09(T(U::*)(V))
{ std::cout << "T (U::*)(V)\n"; }

template<typename T>
void process09(account_t(T::*)())
{ std::cout << "C (T::*)()\n"; }

template<typename T, typename U>
void process09(transaction_t(T::*)(U))
{ std::cout << "C (T::*)(U)\n"; }

template<typename T>
void process09(balance_report_t(balance_t::*)(T))
{ std::cout << "D (C::*)(T)\n"; }

int main()
{
  int (account_t::* pfm1)() = &account_t::get_number;
  int (account_t::* pfm2)(std::string) =
    &account_t::from_string;
  int (balance_t::* pfm3)() =
    &balance_t::get_account_number;
  bool (balance_t::* pfm4)(double) =
    &balance_t::can_withdraw;
  account_t (balance_t::* pfm5)() =
    &balance_t::get_account;
  transaction_t(balance_t::* pfm6)(double) =
    &balance_t::withdraw;
  balance_report_t(balance_t::* pfm7)(int) =
    &balance_t::make_report;

  process09(pfm1); // T (C::*)()
  process09(pfm2); // T (C::*)(U)
  process09(pfm3); // T (U::*)()
  process09(pfm4); // T (U::*)(V)
  process09(pfm5); // C (T::*)()
  process09(pfm6); // C (T::*)(U)
  process09(pfm7); // D (C::*)(T)
}
```

- 数据成员的指针，如 T C::*、C T::* 和 T U::*，其中 C 是类类型，T 和 U 是类型模板形参:

```
template<typename T>
void process10(T account_t::*)
{ std::cout << "T C::*\n"; }

template<typename T>
void process10(account_t T::*)
{ std::cout << "C T::*\n"; }
```

```

template<typename T, typename U>
void process10(T U::*) { std::cout << "T U::*\n"; }

int main()
{
  process10(&account_t::number); // T C::*
  process10(&balance_t::account); // C T::*
  process10(&balance_t::amount); // T U::*
}
```

- 模板中参数列表包含至少一个类型模板形参; 一般形式为 C<T>，其中 C 是类类型，T 是类型模板形参:

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

template<typename T>
void process11(wrapper<T>) { std::cout << "C<T>\n"; }

int main()
{
  wrapper<double> wd{ 42.0 };
  process11(wd); // C<T>
}
```

- 模板中参数列表包含至少一个非类型模板实参; 一般形式为 C<i>，其中 C 是类类型，i 是非类型模板参数:

```
template <size_t i>
struct int_array
{
  int data[i];
};

template<size_t i>
void process12(int_array<i>) { std::cout << "C<i>\n"; }

int main()
{
  int_array<5> ia{};
  process12(ia); // C<i>
}
```

- 双重模板参数，其实参列表包含至少一个类型模板形参; 一般形式为 TT<T>，其中 TT 是模板模板形参，T 是类型模板:

```
template<template<typename> class TT, typename T>
```

```
void process13(TT<T>) { std::cout << "TT<T>\n"; }

int main()
{
  wrapper<double> wd{ 42.0 };
  process13(wd); // TT<U>
}
```

- 双重模板参数，其实参列表包含至少一个非类型模板实参; 一般形式为 TT<i>，其中 TT 是模板模板形参，i 是非类型模板实参:

```
template<template<size_t> typename TT, size_t i>
void process14(TT<i>) { std::cout << "TT<i>\n"; }
int main()
{
  int_array<5> ia{};
  process14(ia); // TT<i>
}
```

- 双重模板参数，参数列表中没有依赖于模板形参的模板实参; 形式为 TT<C>，其中 TT 是模板模板形参，C 是类类型:

```
template<template<typename> typename TT>
void process15(TT<account_t>) { std::cout << "TT<C>\n"; }

int main()
{
  wrapper<account_t> wa{ {42} };
  process15(wa); // TT<C>
}
```

尽管编译器能够推导出许多类型的模板参数，但能做的事也不多:

- 编译器不能从非类型模板实参的类型，推断出类型模板实参的类型。下面的例子中，process 是一个有两个模板形参的函数模板: 一个类型模板为 T，一个类型为 T 的非类型模板 i，调用带有五个 double 数组的函数不允许编译器确定 T 的类型，即使这是指定数组大小的值的类型:

```
template <typename T, T i>
void process(double arr[i])
{
  using index_type = T;
  std::cout << "processing " << i
            << " doubles" << '\n';

  std::cout << "index type is "
              << typeid(T).name() << '\n';
}

int main()
{
  double arr[5]{};
```

```
  process(arr); // error
  process<int, 5>(arr); // OK
}
```

- 编译器不能根据默认值的类型确定模板参数的类型。前面的代码中用函数模板过程举例说明了这一点，该过程只有一个类型模板形参，但有两个函数形参，都是 T 类型，并且都有默认值。

  process() 调用 (没有实参) 失败，因为编译器不能从函数形参的默认值推断类型模板形参 T 的类型。process<int>() 没问题，因为 template 参数显式提供。process(6) 也没问题，因为第一个函数形参的类型可以从提供的实参中推导出来，所以类型模板参数也可以推导出来:

```
template <typename T>
void process(T a = 0, T b = 42)
{
  std::cout << a << "," << b << '\n';
}

int main()
{
  process(); // [1] error
  process<int>(); // [2] OK
  process(10); // [3] OK
}
```

- 尽管编译器可以从函数指针或成员函数指针推导出函数模板实参，但这种功能有几个限制: 不能从指向函数模板的指针推导出实参，也不能从指向函数的指针推导出实参，该函数的重载集包含多个匹配所需类型的重载函数。

  前面的代码中，函数模板调用接受一个指向函数的指针，该函数有两个参数，第一个是类型模板形参 T，第二个是 int 型，并返回 void。这个函数模板不能传递指向 alpha([1]) 的指针，这是一个函数模板，不能传递给 beta([2])，并且有多个可以匹配类型 T 的重载。可以用指向 gamma 的指针调用 ([3])，编译器将正确地推断出第二个重载的类型:

```
template <typename T>
void invoke(void(*pfun)(T, int))
{
  pfun(T{}, 42);
}

template <typename T>
void alpha(T, int)
{ std::cout << "alpha(T,int)" << '\n'; }

void beta(int, int)
{ std::cout << "beta(int,int)" << '\n'; }

void beta(short, int)
{ std::cout << "beta(short,int)" << '\n'; }

```

```cpp
void gamma(short, int, long long)
{ std::cout << "gamma(short,int,long long)" << '\n'; }

void gamma(double, int)
{ std::cout << "gamma(double,int)" << '\n'; }

int main()
{
  invoke(&alpha); // [1] error
  invoke(&beta); // [2] error
  invoke(&gamma); // [3] OK
}
```

- 编译器的另一个限制是对数组的主维进行参数推断，这不是函数形参类型的一部分。这种限制的例外情况是维度引用引用或指针类型。下面的代码段演示了这些限制:
  - [1] 处调用 process1() 会生成一个错误，因为它引用数组的主维度，所以编译器无法推断非类型模板参数 Size 的值。
  - [2] 的点上调用 process2() 是正确的，因为非类型模板参数 Size 指的是数组的第二维度。
  - 另一方面，process3()(在 [3]) 和 process4()([4]) 都没问题，因为函数参数要么是一个引用，要么是指向一维数组的指针:

```cpp
template <size_t Size>
void process1(int a[Size])
{ std::cout << "process(int[Size])" << '\n'; };

template <size_t Size>
void process2(int a[5][Size])
{ std::cout << "process(int[5][Size])" << '\n'; };

template <size_t Size>
void process3(int(&a)[Size])
{ std::cout << "process(int[Size]&)" << '\n'; };

template <size_t Size>
void process4(int(*a)[Size])
{ std::cout << "process(int[Size]*)" << '\n'; };

int main()
{
  int arr1[10];
  int arr2[5][10];
  process1(arr1); // [1] error
  process2(arr2); // [2] OK
  process3(arr1); // [3] OK
  process4(&arr1); // [4] OK
}
```

- 若在函数模板形参列表中的表达式中使用了非类型模板实参，则编译器无法推断其值。

下面的代码中，ncube 是一个类模板，其非类型模板参数 N 表示多个维度。函数模板 process 也有一个非类型的模板形参 N，用在与其单个形参类型相同的模板形参列表中的表达式中，所以编译器不能从函数参数的类型推断出 N 的值 (如 [1] 所示)，这必须显式指定 (如 [2] 所示):

```
template <size_t N>
struct ncube
{
  static constexpr size_t dimensions = N;
};

template <size_t N>
void process(ncube<N - 1> cube)
{
  std::cout << cube.dimensions << '\n';
}

int main()
{
  ncube<5> cube;
  process(cube); // [1] error
  process<6>(cube); // [2] OK
}
```

本节中讨论的模板参数推导的所有规则也适用于可变参数函数模板，但目前讨论的所有内容都是在函数模板上下文中进行的。模板参数推断也适用于类模板，我们将在下一节探讨这个主题。

4.4. 类模板的参数推导

C++17 之前，模板参数推断只适用于函数，而不适用于类。所以当一个类模板实例化时，必须提供所有的模板参数。下面的代码段展示了几个例子:

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

std::pair<int, double> p{ 42, 42.0 };
std::vector<int> v{ 1,2,3,4,5 };
wrapper<int> w{ 42 };
```

通过为函数模板利用模板实参推断，一些标准类型具有辅助函数，这些函数可以创建类型的实例，而不需要显式地指定模板实参。例如：std::make_pair 用于 std::pair, std::make_unique 用于 std::unique_ptr。这些辅助函数模板会使用 auto 关键字，避免了为类模板指定模板实参。这里有一个例子:

```
auto p = std::make_pair(42, 42.0);
```

尽管并非所有标准类模板都有这样一个用于创建实例的辅助函数，但编写自己的帮助函数并不难。下面的代码中，可以看到一个 make_vector 函数模板用于创建 std::vector<T> 实例，还有一个 make_wrapper 函数模板用于创建 wrapper<T> 实例:

```
template <typename T, typename... Ts,
typename Allocator = std::allocator<T>>
auto make_vector(T&& first, Ts&&... args)
{
  return std::vector<std::decay_t<T>, Allocator> {
    std::forward<T>(first),
    std::forward<Ts>(args)...
  };
}

template <typename T>
constexpr wrapper<T> make_wrapper(T&& data)
{
  return wrapper{ data };
}

auto v = make_vector(1, 2, 3, 4, 5);
auto w = make_wrapper(42);
```

C++17 标准通过为类模板提供模板参数推导，简化了类模板的使用。因此使用 C++17，本节中的第一段代码可以进行如下的简化:

```
std::pair p{ 42, 42.0 }; // std::pair<int, double>
std::vector v{ 1,2,3,4,5 }; // std::vector<int>
wrapper w{ 42 }; // wrapper<int>
```

本例中，因为编译器能够从初始化式的类型推断出模板参数，编译器从变量的初始化表达式中推导出。当然，编译器也能够从 new 表达式和函数样式的强制转换表达式中推导出模板参数。下面举几个例子:

```
template <typename T>
struct point_t
{
  point_t(T vx, T vy) : x(vx), y(vy) {}
  private:
  T x;
  T y;
};

auto p = new point_t(1, 2); // [1] point<int>
              // new expression

std::mutex mt;
auto l = std::lock_guard(mt); // [2]
// std::lock_guard<std::mutex>
// function-style cast expression
```

类模板的模板参数推导方式与函数模板不同，当在变量声明或函数样式强制转换中遇到类模板的名称时，编译器将构建一组推导指南。

有虚构函数代表一个虚构的类的构造函数签名模板类型，还可以提供推导参考，这些参考线将添加到编译器生成的参考列表中。若在虚构函数模板的构造集上重载解析失败 (因为这些函数表示构造函数，返回类型不是匹配过程的一部分)，则程序格式错误并报错。否则，所选函数模板特化的返回类型将变成推导出类模板的特化。

为了更好地理解这一点，来看看推导指南实际的样子。下面的代码中，可以看到编译器为 std::pair 类生成的一些指南。实际的列表更长，这里只列出了一些:

```
template <typename T1, typename T2>
std::pair<T1, T2> F();

template <typename T1, typename T2>
std::pair<T1, T2> F(T1 const& x, T2 const& y);

template <typename T1, typename T2, typename U1,
typename U2>
std::pair<T1, T2> F(U1&& x, U2&& y);
```

这组隐式推导指南由类模板的构造函数生成。包括默认构造函数、复制构造函数、移动构造函数和所有转换构造函数，参数以精确的顺序复制。若构造函数是显式的，推导指南也是显式的。但若类模板没有用户定义的构造函数，则为假设的默认构造函数创建推导指南。总会为假设的复制构造函数创建推导指南。

用户定义的演绎指南可以在源代码中提供，其语法类似于带有尾部返回类型，但没有 auto 关键字的函数。推导指南可以是函数也可以是函数模板，但这些必须在与类模板相同的命名空间中提供。若要为 std::pair 类添加用户定义的推导指南，则必须在 std 命名空间中完成。下面是一个例子:

```
namespace std
{
  template <typename T1, typename T2>
  pair(T1&& v1, T2&& v2) -> pair<T1, T2>;
}
```

目前，推导指南都针对的是函数模板，但它们不一定是函数模板，也可以是正则函数:

```
std::pair p1{1, "one"}; // std::pair<int, const char*>
std::pair p2{"two", 2}; // std::pair<const char*, int>
std::pair p3{"3", "three"};
// std::pair<const char*, const char*>
```

使用编译器退化的 std::pair 类推导指南，推导出的类型为 std::pair<int, const char*> (p1)，std::pair<const char*，int> (p2)，std::pair<const char*，const char*> (p3)。换句话说，编译器在使用字面值字符串的地方推导出的类型是 const char*(正如预料的那样)。可以通过提供几个用户定义的推导指南，告诉编译器推导 std::string，而非 const char*:

```
namespace std
{
  template <typename T>
```

```
  pair(T&&, char const*) -> pair<T, std::string>;

  template <typename T>
  pair(char const*, T&&) -> pair<std::string, T>;

  pair(char const*, char const*) ->
  pair<std::string, std::string>;
}
```

前两个是函数模板，但第三个是常规函数。有了这些指南，从前面的例子中为 p1、p2 和 p3 推导出的类型分别是 std::pair<int, std::string>，std::pair<std::string, int> 和 std::pair<std::string, std::string>。

再看一个用户定义指南的例子，这次是一个用户定义的类。考虑以下对范围建模的类模板:

```
template <typename T>
struct range_t
{
  template <typename Iter>
  range_t(Iter first, Iter last)
  {
    std::copy(first, last, std::back_inserter(data));
  }
  private:
  std::vector<T> data;
};
```

这个实现没有太多内容，但已经足够了。假设想从一个整数数组中构造一个 range 对象:

```
int arr[] = { 1,2,3,4,5 };
range_t r(std::begin(arr), std::end(arr));
```

运行这段代码将报错。不同的编译器会生成不同的错误消息，也许 Clang 提供的错误消息最能描述问题:

```
error: no viable constructor or deduction guide for deduction
of template arguments of 'range_t'
range_t r(std::begin(arr), std::end(arr));
       ^
note: candidate template ignored: couldn't infer template
argument 'T'
range_t(Iter first, Iter last)
         ^
note: candidate function template not viable: requires 1
argument, but 2 were provided
struct range_t
```

然而，不管实际的错误消息是什么，其含义都是一样的:range_t 的模板参数推导失败。为了使推导工作，需要提供自定义的推导指南:

```
template <typename Iter>
range_t(Iter first, Iter last) ->
range_t<
typename std::iterator_traits<Iter>::value_type>;
```

该推导指南指示的是，当遇到对带有两个迭代器实参的构造函数的调用时，模板形参 T 的值应该推导为迭代器特征的值类型。迭代器特征将在第 5 章中讨论，有了这个功能，前面的代码段运行起来就没有问题了，编译器会按预期的那样推导出 r 变量的类型为 range_t<int>。

本节开始时，提供了以下示例，其中 w 的类型推断为 wrapper<int>:

```
wrapper w{ 42 }; // wrapper<int>
```

C++17 中，若没有用户定义的推导指南，这实际上不正确。因为是 wrapper<T> 是一个聚合类型，并且类模板参数推导在 C++17 中从聚合初始化中不起作用。所以为了使上一行代码工作，需要提供如下的推导指南:

```
template <typename T>
wrapper(T) -> wrapper<T>;
```

C++20 中取消了对这种用户定义的推导指南的要求，标准提供了对聚合类型的支持 (只要任何依赖基类都没有虚函数或虚基类，并且变量可从一个非空的初始化器列表中进行初始化)。

只有在没有提供模板参数的情况下，类模板参数推导才有效。因此，下面的 p1 和 p2 声明都是有效的，并且发生类模板参数推导。对于 p2，推导出的类型是 std::pair<int, std::string>(假设用户定义的指南可用)。然而，p3 和 p4 的声明会产生错误，因为类模板参数推导没有发生。而模板参数列表是存在的 (<> 和 <int>)，但不包含所有必需的参数:

```
std::pair<int, std::string> p1{ 1, "one" }; // OK
std::pair p2{ 2, "two" }; // OK
std::pair<> p3{ 3, "three" }; // error
std::pair<int> p4{ 4, "four" }; // error
```

类模板参数推断可能并不总是产生预期的结果:

```
std::vector v1{ 42 };
std::vector v2{ v1, v1 };
std::vector v3{ v1 };
```

v1 的推导类型是 std::vector<int>，v2 的推导类型是 std::vector<std::vector<int>>。但编译器应该为 v3 类型推断出什么? 有两个选项:std::vector<std::vector<int>> 和 std::vector<int>。若期望是前者，那么将失望地发现编译器实际上推导出后者。这是因为推导依赖于参数的数量和类型。

当参数数量大于 1 时，将使用接受初始化列表的构造函数。对于 v2 变量，即 std::initializer_list<std::vector<int>>。当参数的数量为 1 时，则考虑参数的类型。若参数的类型是 std::vector 的 (特化)-考虑到这种显式情况-则使用复制构造函数，推导的类型是实参的声明类型。这就是变量 v3 的情况，其中推导出的类型是 std::vector<int>。否则，使用接受初始化列表 (包含单个元素) 的构造函数，例如变量 v1，其推导出的类型为 std::vector<int>。在 cppinsights.io 可视化工具的帮助下，可以显示生成的代码 (对于前面的代码段)。为了简单起见，删除了 allocator 参数:

```cpp
std::vector<int> v1 =
std::vector<int>{std::initializer_list<int>{42}};
std::vector<vector<int>> v2 =
std::vector<vector<int>>{
  std::initializer_list<std::vector<int>>{
    std::vector<int>(v1),
      std::vector<int>(v1)
    }
  };

std::vector<int> v3 = std::vector<int>{v1};
```

类模板参数推导是 C++17 的一个特性，C++20 改进了对聚合类型的处理。当编译器能够推导出不必要的模板参数时，器有助于避免编写不必要的显式模板参数，即使在某些情况下，编译器可能需要用户定义的推导指南才能进行推导。这还避免了创建工厂函数的需要，例如 std::make_pair 或 std::make_tuple，这是在类模板可用之前从模板参数推导中受益的一种方式。

关于模板参数推导的内容比我们到目前为止讨论的内容要多。函数模板参数推断有一种特殊情况，称为转发引用。这个主题将在下面的章节中进行讨论。

4.5. 转发引用

C++11 中添加到该语言的最重要的特性之一是移动语义，通过避免不必要的复制来帮助提高性能。移动语义由 C++11 的另一个特性支持，称为右值引用。讨论这些之前，值得一提的是，在 C++ 中有两种值:

- 左值是指向内存位置的值，可以用 & 操作符获取它们的地址。左值可以出现在赋值表达式的左边和右边。
- 右值是非左值的值，它们的定义互斥。右值不指向内存位置，并且不能使用 & 操作符获取它们的地址。右值是字面量和临时对象，只能出现在赋值表达式的右侧。

> **Note**
>
> C++11 中，还有一些其他的值类别，glvalue, prvalue 和 xvalue。在这里讨论它们会让我们的注意力过于分散，可以在https://en.cppreference.com/w/cpp/language/value_category上阅读更多关于它们的信息。

引用是已经存在的对象或函数的别名。就像我们有两种值一样，C++11 中有两种引用:

- 用 & 表示的左值引用，例如 &x，是对左值的引用。
- 用 && 表示的右值引用，例如在 &&x 中，是对右值的引用。

来看一些例子，可以更好地理解这些概念:

```cpp
struct foo
{
  int data;
};
```

```
5
6  void f(foo& v)
7  { std::cout << "f(foo&)\n"; }
8
9  void g(foo& v)
10 { std::cout << "g(foo&)\n"; }
11
12 void g(foo&& v)
13 { std::cout << "g(foo&&)\n"; }
14
15 void h(foo&& v)
16 { std::cout << "h(foo&&)\n"; }
17
18 foo x = { 42 }; // x is lvalue
19 foo& rx = x; // rx is lvalue
```

这里有三个函数:f，接受一个左值引用 (即 int&);g，有两个重载，一个用于左值引用，另一个用于右值引用 (即 int&&);h，取一个右值引用。还有两个变量，x 和 rx。x 是一个左值，类型为 foo，可以用 &x 看到它的地址。左值也是 rx，它是一个左值引用，类型是 foo&。现在，来看看如何调用 f、g 和 h 函数:

```
1 f(x); // f(foo&)
2 f(rx); // f(foo&)
3 f(foo{42}); // error: a non-const reference
4             // may only be bound to an lvalue
```

因为 x 和 rx 都是左值，所以将它们传递给 f 没问题，因为这个函数接受一个左值引用。然而，foo{42} 是一个临时对象，并且因为它不存在于调用 f 的上下文之外，所以其是一个右值，将它传递给 f 将导致编译器错误，因为函数的形参是 foo& 类型的，非常量引用只能绑定到左值。若函数 f 的签名更改为 f(int const &v)，就没问题了。接下来，讨论 g 函数:

```
1 g(x); // g(foo&)
2 g(rx); // g(foo&)
3 g(foo{ 42 }); // g(foo&&)
```

前面的代码段中，使用 x 或 rx 调用 g 将解析为第一个重载，该重载接受左值引用。但使用 foo{42}(foo{42} 是一个临时对象，因此是一个右值) 将解析为第二个重载，将接受一个右值引用。看当我们想对 h 函数进行相同的调用时会发生什么:

```
1 h(x); // error, cannot bind an lvalue to an rvalue ref
2 h(rx); // error
3 h(foo{ 42 }); // h(foo&&)
4 h(std::move(x)); // h(foo&&)
```

该函数可以接受一个右值引用，但将 x 或 rx 传递给它会导致编译器错误，因为左值不能绑定到右值引用。表达式 foo{42} 是一个右值，可以作为参数传递。若把其语义从左值改为右值 (可在 std::move 的帮助下完成)，也可以把左值 x 传递给函数 h。std::move 并不真正的移动，只是做了一种从左值到右值的转换而已。

理解将右值传递给函数有两个目的: 要么对象是临时的，函数可以对它做任何事情，要么函数应该拥有接收对象。这就是移动构造函数和移动赋值操作符的目的，其实很少会看到其他函数接受右值引用。在最后一个例子的函数 h 中，参数 v 是一个左值，但绑定为右值。变量 x 存在于对 h 之外，将其传递给 std::move 可转换为右值。在调用 h 返回，其仍然作为左值存在，但需要假设函数 h 对它做了一些事情，它的状态并不确定。

因此，右值引用的目的是启用移动语义。还有另一个功能，那就是实现完美转发。为了理解这一点，来看看以下对函数 g 和 h 的修改:

```
void g(foo& v) { std::cout << "g(foo&)\n"; }
void g(foo&& v) { std::cout << "g(foo&&)\n"; }

void h(foo& v) { g(v); }
void h(foo&& v) { g(v); }
```

这个代码片段中，g 的实现与前面看到的相同。h 也有两个重载，一个接受左值引用并调用 g，另一个接受右值引用并调用 g。换句话说，函数 h 只是将参数转发给 g。现在，看看以下调用方式:

```
foo x{ 42 };
h(x); // g(foo&)
h(foo{ 42 }); // g(foo&)
```

由此，可以期望调用 h(x) 将使 g 的重载使用左值引用，而对 h(foo{42}) 的调用将使 g 的重载的使用右值引用。但事实上，它们都会调用 g 的第一个重载，因此将 g(foo&) 打印到控制台。当理解了引用的工作原理，解释就很简单了: 在上下文中 h(foo&& v)，参数 v 实际上是一个左值 (有名字，可以取它的地址)，所以用它调用 g 会调用接受左值引用的重载。为了使其如预期那样工作，需要改变 h 函数的实现:

```
void h(foo& v) { g(std::forward<foo&>(v)); }
void h(foo&& v) { g(std::forward<foo&&>(v)); }
```

std::forward 是一个允许正确转发值的函数。函数的作用为:

- 若参数是一个左值引用，函数的行为就像调用 std::move(将语义从左值更改为右值)。
- 若实参是右值引用，则什么也不做。

目前为止，我们讨论的所有内容都与模板无关，而模板才是本书的主题，而函数模板也可以用于接收左值和右值引用，了解在非模板场景中的工作原理非常重要。在模板中，右值引用的工作方式略有不同，有时右值引用，但实际上有时是左值引用。

表现出这种行为的引用称为转发引用，也称为**通用引用**。在 C++11 之后不久，由 Scott Meyers 创造的一个术语，当时标准中还没有这种类型的引用术语。因为通用引用这个术语不能很好地描述它们的语义，为了解决这个问题，C++ 标准委员会在 C++14 中将这些引用称为**转发引用**。为了忠实于标准术语，我们在本书中称其为转发引用。

开始讨论转发引用之前，我们考虑以下重载函数模板和类模板:

```
template <typename T>
void f(T&& arg) // forwarding reference
{ std::cout << "f(T&&)\n"; }

```

```cpp
template <typename T>
void f(T const&& arg) // rvalue reference
{ std::cout << "f(T const&&)\n"; }

template <typename T>
void f(std::vector<T>&& arg) // rvalue reference
{ std::cout << "f(vector<T>&&)\n"; }

template <typename T>
struct S
{
  void f(T&& arg) // rvalue reference
  { std::cout << "S.f(T&&)\n"; }
};
```

可以像下面这样调用这些函数:

```cpp
int x = 42;
f(x); // [1] f(T&&)
f(42); // [2] f(T&&)

int const cx = 100;
f(cx); // [3] f(T&&)
f(std::move(cx)); // [4] f(T const&&)

std::vector<int> v{ 42 };
f(v); // [5] f(T&&)
f(std::vector<int>{42});// [6] f(vector<T>&&)

S<int> s;
s.f(x); // [7] error
s.f(42); // [8] S.f(T&&)
```

这段代码中，可以注意到:

- [1] 和 [2] 处用左值或右值使用 f 时，将解析为第一个重载：f(T&&)。
- [3] 处调用左值为常数的 f 也会解析为第一个重载，但 [4] 处调用右值为常数的 f 会解析为第二个重载 f(T const&&)，这个匹配度更高。
- [5] 处用左值 std::vector 对象调用 f 会解析为第一个重载，[6] 处用右值 std::vector 对象调用 f 会解析为第三个重载，f(vector<T>&&)。
- [7] 处用左值调用 S::f 是错误的，因为左值不能绑定到右值引用，[8] 处用右值调用是正确的。

本例中的所有 f 函数重载都采用右值引用。第一次重载中的 && 并不一定是右值引用。若传递的是右值，则表示右值引用; 若传递的是左值，则表示左值引用。这样的引用称为转发引用，但转发引用只出现在模板形参右值引用的上下文中，必须是 T&& 形式。T const&& 或 std::vector<T>&& 不是转发引用，而是正常的右值引用。类似地，类模板 S 的 f 函数成员中的 T&& 也是右值引用，因为 f 不是模板，而是类模板的非模板成员函数，所以转发引用的规则在这里不适用。

转发引用是函数模板参数推断的一种特殊情况，这是之前在本章中讨论过的主题。其目的是实

现模板的完美转发，这是 C++11 的新特性，引用折叠实现的。在展示是如何解决完美转发问题之前，让我们先看看个例子。

C++11 之前，不可能将一个引用引用于另一个引用。现在 C++11 中，对于 typedef 和模板来说，这是可能的:

```
using lrefint = int&;
using rrefint = int&&;
int x = 42;
lrefint& r1 = x; // type of r1 is int&
lrefint&& r2 = x; // type of r2 is int&
rrefint& r3 = x; // type of r3 is int&
rrefint&& r4 = 1; // type of r4 is int&&
```

规则非常简单: 对右值引用的右值引用折叠为右值引用，其他组合都折算为左值引用。可以用表格形式表示如下:

类型	引用类型	变量类型
T&	T&	T&
T&	T&&	T&
T&&	T&	T&
T&&	T&&	T&&

表 4.2

下表所示的其他组合都不涉及引用折叠规则，只有当这两种类型都是引用时才适用:

类型	引用类型	变量类型
T	T	T
T	T&	T&
T	T&&	T&&
T&	T	T&
T&&	T	T&&

表 4.3

转发引用不仅适用于模板，也适用于自动推导规则。当使用 auto&& 时，其意味着转发引用。这同样不适用于其他任何方式，比如 auto const&& 这样的 cv 限定方式。下面是一些例子:

```
int x = 42;
auto&& rx = x; // [1] int&
auto&& rc = 42; // [2] int&&
auto const&& rcx = x; // [3] error

std::vector<int> v{ 42 };
auto&& rv = v[0]; // [4] int&
```

前两个例子中，rx 和 rc 都是转发引用，分别绑定到左值和右值。然而，rcx 是一个右值引用，因为 auto const&& 并不表示转发引用，所以尝试将其绑定为左值是错误的。类似地，rv 是一个转发引用，并绑定为左值。

如前所述，转发引用的目的是实现完美转发。我们已经在前面的非模板上下文中看到了完美转发的概念，其工作方式与模板类似。为了演示这一点，这里将函数 h 重新定义为模板函数:

```
void g(foo& v) { std::cout << "g(foo&)\n"; }
void g(foo&& v) { std::cout << "g(foo&&)\n"; }

template <typename T> void h(T& v) { g(v); }
template <typename T> void h(T&& v) { g(v); }

foo x{ 42 };
h(x); // g(foo&)
h(foo{ 42 }); // g(foo&)
```

g 重载的实现是相同的，但 h 重载现在是函数模板。用一个左值和一个右值调用 h 实际上解析为对 g 的相同调用，第一个重载取一个左值。因为在函数 h 的上下文中，所以 v 是一个左值，将它传递给 g 将调用取左值的重载。

这个问题的解决方案与讨论模板之前已经看到的解决方案相同，这里有一个区别: 不再需要两个重载，而是需要一个转发引用:

```
template <typename T>
void h(T&& v)
{
  g(std::forward<T>(v));
}
```

这个实现使用 std::forward 将左值作为左值传递，将右值作为右值传递，同样适用于可变函数模板。下面是 std::make_unique 函数的概念实现，其创建了一个 std::unique_ptr 对象:

```
  void g(foo& v) { std::cout << "g(foo&)\n"; }
  void g(foo&& v) { std::cout << "g(foo&&)\n"; }

  template <typename T> void h(T& v) { g(v); }
  template <typename T> void h(T&& v) { g(v); }

  foo x{ 42 };
  h(x); // g(foo&)
  h(foo{ 42 }); // g(foo&)
```

总结一下本节，请记住转发引用 (也称为通用引用) 基本上是针对函数模板参数的特殊推导规则。基于引用折叠规则工作，目的是实现完全转发。这是通过保留值语义将引用传递给另一个函数: 右值应作为右值传递，左值应作为左值传递。

本章中我们将讨论的下一个主题是 decltype 说明符。

4.6. decltype 说明符

此说明符在 C++11 中引入，用于返回表达式的类型，通常在模板中与 auto 说明符一起使用。可以用于声明依赖于模板参数的函数模板的返回类型，或者包装另一个函数，并返回执行包装函数结果的返回类型。

decltype 说明符不限制在模板代码中使用。可以用于不同的表达式，并且根据表达式产生不同的结果。规则如下:

1. 若表达式是标识符或类成员访问，则结果是由表达式命名的实体类型。若实体不存在，或者是具有重载集的函数 (存在多个同名函数)，编译器将报错。
2. 若表达式是函数调用或重载操作符函数，则结果为函数的返回类型。若重载的操作符括在括号中，则忽略这些操作符。
3. 若表达式是左值，则结果类型是对表达式类型的左值引用。
4. 若表达式为其他类型，则结果类型为表达式的类型。

为了更好地理解这些规则，来看一组示例。考虑在 decltype 表达式中使用的以下函数和变量:

```
int f() { return 42; }
int g() { return 0; }
int g(int a) { return a; }

struct wrapper
{
  int val;
  int get() const { return val; }
};

int a = 42;
int& ra = a;
const double d = 42.99;
long arr[10];
long l = 0;
char* p = nullptr;
char c = 'x';
wrapper w1{ 1 };
wrapper* w2 = new wrapper{ 2 };
```

下面的代码显示了 decltype 说明符的多种用法。适用于每种情况的规则，以及推导出的类型，都在注释中的每行指定:

```
decltype(a) e1; // R1, int
decltype(ra) e2 = a; // R1, int&
decltype(f) e3; // R1, int()
decltype(f()) e4; // R2, int
decltype(g) e5; // R1, error
decltype(g(1)) e6; // R2, int
decltype(&f) e7 = nullptr; // R4, int(*)()
decltype(d) e8 = 1; // R1, const double
decltype(arr) e9; // R1, long[10]
```

```
decltype(arr[1]) e10 = l; // R3, long&
decltype(w1.val) e11; // R1, int
decltype(w1.get()) e12; // R1, int
decltype(w2->val) e13; // R1, int
decltype(w2->get()) e14; // R1, int
decltype(42) e15 = 1; // R4, int
decltype(1 + 2) e16; // R4, int
decltype(a + 1) e17; // R4, int
decltype(a = 0) e18 = a; // R3, int&
decltype(p) e19 = nullptr; // R1, char*
decltype(*p) e20 = c; // R3, char&
decltype(p[0]) e21 = c; // R3, char&
```

我们不详细说明所有这些声明。根据指定的规则，其中大多数都相对容易遵循。不过，为了说明一些推导出的类型，有几点注意事项值得考虑:

- decltype(f) 只使用重载集命名函数，因此适用规则 1。decltype(g) 也命名一个函数，但它有一个重载集。因此，应用规则 1，编译器会报错。
- decltype(f()) 和 decltype(g(1)) 都对表达式使用函数调用，因此适用规则 2，即使 g 有重载集，声明也是正确的。
- decltype(&f) 使用函数 f 的地址，因此应用规则 4，生成 int(*)()。
- decltype(1+2) 和 decltype(a+1) 使用重载加法操作符返回右值，因此适用规则 4，结果是 int。然而，decltype(a = 1) 使用返回左值的赋值操作符，因此应用规则 3，生成左值引用 int&。

decltype 说明符定义了一个未求值的上下文。所以与该说明符一起使用的表达式不会进行计算，因为该说明符只查询其操作数的属性。可以在下面的代码片段中看到这一点，其中赋值 a=1 与 decltype 说明符一起使用来声明变量 e，但在声明之后，a 是其初始化时的值:

```
int a = 42;
decltype(a = 1) e = a;
std::cout << a << '\n'; // prints 42
```

关于模板实例化的规则有一个例外。当与 decltype 说明符一起使用的表达式包含模板时，需要在编译时计算表达式之前实例化模板:

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

decltype(wrapper<double>::data) e1; // double

int a = 42;
decltype(wrapper<char>::data, a) e2; // int&
```

e1 的类型是 double，并且实例化了 wrapper<double> 来推导这个类型。另一方面，e2 的类型是 int&(a 是左值)。即使该类型仅从变量 a 推导出来 (使用了逗号操作符)，wrapper<char> 依旧会在这里进行实例化。

上述规则并不是唯一用于确定类型的规则，还有几个用于数据成员访问的方法。具体如下:

- decltype 表达式中使用的对象的 const 或 volatile 说明符不构成推导的类型。
- 对象或指针表达式是左值还是右值并不影响推导的类型。
- 若数据成员访问表达式括号括起来，例如 decltype((expression))，则前两条规则不适用。对象的 const 或 volatile 限定符确实会影响推导的类型，包括对象的值。

下面的代码片段演示了前两条规则:

```
struct foo
{
  int a = 0;
  volatile int b = 0;
  const int c = 42;
};

foo f;
foo const cf;
volatile foo* pf = &f;

decltype(f.a) e1 = 0; // int
decltype(f.b) e2 = 0; // int volatile
decltype(f.c) e3 = 0; // int const

decltype(cf.a) e4 = 0; // int
decltype(cf.b) e5 = 0; // int volatile
decltype(cf.c) e6 = 0; // int const

decltype(pf->a) e7 = 0; // int
decltype(pf->b) e8 = 0; // int volatile
decltype(pf->c) e9 = 0; // int const

decltype(foo{}.a) e10 = 0; // int
decltype(foo{}.b) e11 = 0; // int volatile
decltype(foo{}.c) e12 = 0; // int const
```

右边的注释提到了每种情况的推导类型。当表达式加圆括号时，这两个规则会颠倒过来。来看看下面的代码:

```
foo f;
foo const cf;
volatile foo* pf = &f;

int x = 1;
int volatile y = 2;
int const z = 3;

decltype((f.a)) e1 = x; // int&
decltype((f.b)) e2 = y; // int volatile&
decltype((f.c)) e3 = z; // int const&

```

```
decltype((cf.a)) e4 = x; // int const&
decltype((cf.b)) e5 = y; // int const volatile&
decltype((cf.c)) e6 = z; // int const&

decltype((pf->a)) e7 = x; // int volatile&
decltype((pf->b)) e8 = y; // int volatile&
decltype((pf->c)) e9 = z; // int const volatile&

decltype((foo{}.a)) e10 = 0; // int&&
decltype((foo{}.b)) e11 = 0; // int volatile&&
decltype((foo{}.c)) e12 = 0; // int const&&
```

decltype 用于声明变量 e1 到 e9 的所有表达式都是左值，推导出的类型是一个左值引用。用于声明变量 e10、e11 和 e12 的表达式是右值，推导的类型是一个右值引用。此外，cf 是一个常量对象，foo::a 的类型是 int，结果类型是 const int&。类似地，foo::b 的类型为 volatile int，结果类型是 const volatile int&。这些只是本段代码中的几个示例，但其他示例遵循相同的推导规则。

因为 decltype 是一个类型说明符，所以多余的 const 和 volatile 限定符以及引用说明符将忽略:

```
int a = 0;
int& ra = a;
int const c = 42;
int volatile d = 99;

decltype(ra)& e1 = a; // int&
decltype(c) const e2 = 1; // int const
decltype(d) volatile e3 = 1; // int volatile
```

现在，已经了解了 decltype 说明符的工作方式。其真正目的是在模板中使用 decltype，其中函数的返回值取决于模板参数，并且在实例化之前是未知的。为了理解这个场景，先从下面的函数模板示例开始，该函数模板返回两个值中的最小值:

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

可以这样使用:

```
auto m1 = minimum(1, 5); // OK
auto m2 = minimum(18.49, 9.99);// OK
auto m3 = minimum(1, 9.99);
                    // error, arguments of different type
```

因为提供的参数是相同类型的，所以前两次调用都是正确的。第三次调用将产生编译器错误，因为参数具有不同的类型，所以需要将整数值强制转换为 double 类型。当然，还有另一种选择: 可以编写一个函数模板，接受两个可能不同类型的参数，并返回这两个参数中的最小值。如下所示:

```
template <typename T, typename U>
??? minimum(T&& a, U&& b)
```

```
{
  return a < b ? a : b;
}
```

问题是，这个函数的返回类型是什么呢? 根据所使用的标准版本，可以以不同的方式实现。

C++11 中，可以使用带有尾随返回类型的 auto 说明符，使用 decltype 说明符从表达式推导返回类型:

```
template <typename T, typename U>
auto minimum(T&& a, U&& b) -> decltype(a < b ? a : b)
{
  return a < b ? a : b;
}
```

若使用 C++14 或标准的更新版本，则可以简化此语法。后面的返回类型将不再需要:

```
template <typename T, typename U>
decltype(auto) minimum(T&& a, U&& b)
{
  return a < b ? a : b;
}
```

还可以进一步简化，简单地使用 auto 作为返回类型:

```
template <typename T, typename U>
auto minimum(T&& a, U&& b)
{
  return a < b ? a : b;
}
```

虽然 decltype(auto) 和 auto 在本例中具有相同的效果，但情况并非总是如此。看看下面的例子，有一个函数返回一个引用，而另一个函数完美地转发了相关参数:

```
template <typename T>
T const& func(T const& ref)
{
  return ref;
}

template <typename T>
auto func_caller(T&& ref)
{
return func(std::forward<T>(ref));
}

int a = 42;
decltype(func(a)) r1 = func(a); // int const&
decltype(func_caller(a)) r2 = func_caller(a); // int
```

函数 func 返回一个引用，而 func_caller 应该完全转发到这个函数。使用 auto 作为返回类型，在前面的代码片段中推断为 int(参见变量 r2)。为了完美地转发返回类型，必须对其使用 decltype(auto):

```
template <typename T>
decltype(auto) func_caller(T&& ref)
{
  return func(std::forward<T>(ref));
}

int a = 42;
decltype(func(a)) r1 = func(a); // int const&
decltype(func_caller(a)) r2 = func_caller(a); // int const&
```

这一次，结果和预期的一样，并且这个代码片段中的 r1 和 r2 的类型都是 int const&。

正如在本节中看到的，decltype 是一个类型说明符，用于推断表达式的类型。可以在不同的上下文中使用，但其目的是让模板确定函数的返回类型，并确保完美转发。与 decltype 一起出现的另一个特性是 std::declval，我们将在下一节中对其进行讨论。

4.7. std::declval 类型操作符

<utility> 中文件中，std::declval 是一个工具类型的操作函数，与 std::move 和 std::forward 等函数属于同一类别，其功能非常简单: 将右值引用添加到类型模板参数中。这个函数的声明如下所示:

```
template<class T>
typename std::add_rvalue_reference<T>::type declval() noexcept;
```

这个函数没有定义，因此不能直接调用，只能在未求值的上下文中使用——decltype、sizeof、typeid 和 noexcept。这些是仅在编译时执行的上下文，在运行时不会进行计算。std::declval 的目的是帮助对没有默认构造函数或有默认构造函数，但由于是 private 或 protected 而不能访问的类型，进行依赖类型求值。

为了理解这是如何工作的，来看一个类模板，它将两个不同类型的值组合在一起，我们希望为这两个类型的值使用加号操作符的结果，创建一个类型别名。如何定义这样的类型别名? 来看看下面的代码:

```
template <typename T, typename U>
struct composition
{
  using result_type = decltype(???);
};
```

可以使用 decltype 说明符，但需要提供一个表达式。不能是 decltype(T + U)，因为它们是类型，而不是值。可以调用默认构造函数，因此可以使用表达式 decltype(T{} + U{})。这可以很好地用于内置类型，如 int 和 double，如下面的代码所示:

```
static_assert(
  std::is_same_v<double,
    composition<int, double>::result_type>);
```

也适用于具有 (可访问的) 默认构造函数的类型，但不能用于没有默认构造函数的类型。以下的 wrapper 就是这样一个例子:

```
struct wrapper
{
  wrapper(int const v) : value(v){}
  int value;

  friend wrapper operator+(int const a, wrapper const& w)
  {
    return wrapper(a + w.value);
  }

  friend wrapper operator+(wrapper const& w, int const a)
  {
    return wrapper(a + w.value);
  }
};

// error, no appropriate default constructor available
static_assert(
  std::is_same_v<wrapper,
    composition<int,wrapper>::result_type>);
```

解决方案是使用 std::declval()，类模板组合的实现将进行如下修改:

```
template <typename T, typename U>
struct composition
{
  using result_type = decltype(std::declval<T>() +
                  std::declval<U>());
};
```

修改之后，前面显示的两个静态断言都可以编译，没有任何错误。这个函数避免需要使用特定的值来确定表达式的类型，其 ui 生成一个类型为 T 的值，而不涉及默认构造函数。返回右值引用的原因是，能够处理函数不能返回的类型，例如：数组和抽象类型。

前面 wrapper 类的定义包含两个友元操作符。当涉及到模板时，“友情”有一定的特殊性。我们将在下一节讨论这个问题。

4.8. 模板间的“友情”

定义类时，可以使用 protected 和 private 访问说明符限制对其成员数据和成员函数的访问。若成员是 private，则只能在类中访问它。若成员是 protected，则可以从具有 public 访问权限或 protected 访问权限的派生类访问。但类可以通过 friend 关键字将其 private 成员或 protected 成员的访问权授予其他函数或类，这些授予特殊访问权限的函数或类称为友元函数或友元类。先来看一个简单的例子:

```
struct wrapper
{
  wrapper(int const v) :value(v) {}
```

```
private:
  int value;

  friend void print(wrapper const & w);
};

void print(wrapper const& w)
{ std::cout << w.value << '\n'; }

wrapper w{ 42 };
print(w);
```

wrapper 类有一个名为 value 的 private 数据成员。有一个名为 print 的独立函数，接受类型 wrapper 的参数，并将包装的值输出到控制台。为了能够访问它，将该函数声明为 wrapper 类的友元。

我们将不关注友情对非模板的作用。读者需要熟悉该特性，才能继续在模板上下文中对其进行讨论。当涉及到模板时，事情变得有点复杂。我们将通过几个例子来研究这个问题:

```
struct wrapper
{
  wrapper(int const v) :value(v) {}
private:
  int value;

  template <typename T>
  friend void print(wrapper const&);

  template <typename T>
  friend struct printer;
};

template <typename T>
void print(wrapper const& w)
{ std::cout << w.value << '\n'; }

template <typename T>
struct printer
{
  void operator()(wrapper const& w)
  { std::cout << w.value << '\n'; }
};

wrapper w{ 42 };
print<int>(w);
print<char>(w);
printer<int>()(w);
printer<double>()(w);
```

print 函数现在是一个函数模板，有一个类型模板参数，但实际上并没有使用。可能有点奇怪，但这是一个有效的代码，需要通过指定模板参数来使用。但不管模板参数如何，print 的模板实例化

都可以访问 wrapper 类的私有成员。注意用于将其声明为 wrapper 类的友元的语法: 使用模板语法。这同样适用于类模板 printer，可声明为 wrapper 类的友元，任何模板实例化，无论模板参数如何，都可以访问其私有部分。

若想限制对这些模板的某些实例的访问呢? 比如只有 int 类型的特化? 然后，可以将这些特殊化声明为友元:

```
struct wrapper;

template <typename T>
void print(wrapper const& w);

template <typename T>
struct printer;

struct wrapper
{
  wrapper(int const v) :value(v) {}
private:
  int value;

  friend void print<int>(wrapper const&);
  friend struct printer<int>;
};

template <typename T>
void print(wrapper const& w)
{ std::cout << w.value << '\n'; /* error */ }

template <>
void print<int>(wrapper const& w)
{ std::cout << w.value << '\n'; }

template <typename T>
struct printer
{
  void operator()(wrapper const& w)
  { std::cout << w.value << '\n'; /* error*/ }
};

template <>
struct printer<int>
{
  void operator()(wrapper const& w)
  { std::cout << w.value << '\n'; }
};

wrapper w{ 43 };
print<int>(w);
print<char>(w);
```

```
printer<int>()(w);
printer<double>()(w);
```

代码段中，wrapper 类与前面相同。对于 print 函数模板和 printer 类模板，都有一个主模板和 int 类型的全特化。只有 int 实例化声明为 wrapper 类的友元。所以其他的特化，想试图访问主模板中 wrapper 类的私有部分时，会产生编译器错误。

这些例子中，将友情授予私有部分的类是非模板类，但类模板也可以声明友元。先从类模板和非模板函数的情况开始:

```
template <typename T>
struct wrapper
{
  wrapper(T const v) :value(v) {}
private:
  T value;

  friend void print(wrapper<int> const&);
};

void print(wrapper<int> const& w)
{ std::cout << w.value << '\n'; }

void print(wrapper<char> const& w)
{ std::cout << w.value << '\n'; /* error */ }
```

此实现中，wrapper 类模板声明以 wrapper<int> 作为参数的 print 重载作为友元。这个重载函数中，可以访问私有数据成员，但不能在任何其他重载中访问。当友元函数或类是模板，而只想用一个特化来访问私有部分时，也会发生类似的情况:

```
template <typename T>
struct printer;

template <typename T>
struct wrapper
{
  wrapper(T const v) :value(v) {}
private:
  T value;

  friend void print<int>(wrapper<int> const&);
  friend struct printer<int>;
};

template <typename T>
void print(wrapper<T> const& w)
{ std::cout << w.value << '\n'; /* error */ }

template<>
void print(wrapper<int> const& w)
```

```
{ std::cout << w.value << '\n'; }

template <typename T>
struct printer
{
  void operator()(wrapper<T> const& w)
  { std::cout << w.value << '\n'; /* error */ }
};

template <>
struct printer<int>
{
  void operator()(wrapper<int> const& w)
  { std::cout << w.value << '\n'; }
};
```

wrapper 类模板的这种实现，使 print 函数模板和 printer 类模板的 int 特化成为友元。试图访问主模板 (或任何其他特化) 中的私有数据成员值时，编译器将报错。

若是 wrapper 类模板允许友元访问 print 函数模板或 printer 类模板的实例化，代码可以这样写:

```
template <typename T>
struct printer;

template <typename T>
struct wrapper
{
  wrapper(T const v) :value(v) {}
private:
  T value;

  template <typename U>
  friend void print(wrapper<U> const&);

  template <typename U>
  friend struct printer;
};

template <typename T>
void print(wrapper<T> const& w)
{ std::cout << w.value << '\n'; }

template <typename T>
struct printer
{
  void operator()(wrapper<T> const& w)
  { std::cout << w.value << '\n'; }
};
```

注意声明友元时，语法是 template<typename U>，而不是 template<typename T>。模板参数的名称 U 可以是除 T 名称之外的名称，这将掩盖 wrapper 类模板的模板参数名称，这是一个错误。使用

这种语法，print 或 printer 的特化都可以访问 wrapper 类模板特化的私有成员。若希望只有满足包装器类 template 参数的友元特化才可以访问私有部分，那么必须使用以下语法:

```
template <typename T>
struct wrapper
{
  wrapper(T const v) :value(v) {}
private:
  T value;

  friend void print<T>(wrapper<T> const&);
  friend struct printer<T>;
};
```

这类似于之前只授予 int 特化访问权限时看到的情况，只不过现在适用于匹配 T 的特化。

除了这些情况外，类模板还可以将友元访问权授予类型模板参数:

```
template <typename T>
struct connection
{
  connection(std::string const& host, int const port)
    :ConnectionString(host + ":" + std::to_string(port))
  {}
private:
  std::string ConnectionString;
  friend T;
};

struct executor
{
  void run()
  {
    connection<executor> c("localhost", 1234);
    std::cout << c.ConnectionString << '\n';
  }
};
```

connection 类模板有一个名为 ConnectionString 的私有数据成员，类型模板参数 T 是类的友元。executor 类使用实例化 connection<executor>，所以 executor 类型是模板参数，并受益于与 connection 类的友元权限，它可以访问私有数据成员 ConnectionString。

从所有这些示例中可以看出，与模板之间的友谊与非模板实体之间的友情略有不同。朋友可以访问类的所有非公共成员，所以交朋友需要小心。另一方面，若需要将访问权限授予一些私有成员，而不是全部成员，则可以借助“客户-律师”模式。此模式允许控制对类的私有部分的访问粒度。可以在这里了解关于这个模式的信息: https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Friendship_and_the_Attorney-Client。

4.9. 总结

本章中，我们学习了一系列高级主题。从名称绑定和依赖名称开始，并学习了如何使用 typename 和模板关键字来告诉编译器引用的是哪种依赖名称。然后，学习了递归模板，以及如何使用不同的方法实现递归函数的编译时版本。

还学习了函数模板和类模板的参数推导，以及如何在用户定义的推导指南的帮助下帮助编译器完成推导。本章涉及的一个重要主题是转发引用，以及如何帮助我们实现完美转发。本章的最后，学习了 decltype 类型说明符、std::declvalue 类型工具，以及在类模板间的友情。

下一章中，我们将开始利用目前为止积累的关于模板的知识来进行模板元编程，基本上就是编写在编译时运行的代码。

4.10. 习题

1. 什么时候执行名称查找?
2. 什么是推导指南?
3. 什么是转发引用?
4. decltype 是用来做什么的?
5. std::declval 是用来做什么的?

4.11. 扩展阅读

- Dependent name lookup for C++ templates –Eli Bendersky, https://eli.thegreenplace.net/2012/02/06/dependent-name-lookup-for-ctemplates
- Universal References in C++11 –Scott Meyers, https://isocpp.org/blog/2012/11/universal-references-in-c11-scott-meyers
- C++ Rvalue References Explained –Thomas Becker, http://thbecker.net/articles/rvalue_references/section_01.html
- Universal vs Forwarding References in C++ –Petr Zemek, https://blog.petrzemek.net/2016/09/17/universal-vs-forwardingreferences-in-cpp/

第 5 章　类型特征和条件编译

类型特征是一种元编程技术，能够在编译时检查类型的属性或执行类型的转换。类型特征本身就是模板，可以将其看作为元类型。了解类型的性质、支持的操作，及其各种属性等信息，是模板代码进行条件编译的关键。在编写模板库时，也非常有用。

本章中，将了解以下内容:

- 定义类型特征
- 了解 SFINAE
- 使用 enable_if 类型特性启用 SFINAE
- constexpr if
- 探索标准的类型特征
- 实际使用类型特征的例子

本章结束时，读者们将很好地理解类型特征是什么，如何使用，以及 C++ 标准库中有哪些类型特征可用。

5.1. 定义类型特征

简而言之，类型特征是包含常量值的小型类模板，该常量值就是所询问的关于类型问题的答案。此类问题的例子是: 该类型是浮点类型吗? 构建提供此类类型信息的类型特征依赖于模板特化: 定义一个主模板，以及一个或多个特化。

来看看如何构建类型特征，在编译时确定类型是否为浮点类型:

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

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

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

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

有两件事需要注意:

- 定义了一个主模板，以及几个完整的特化，每个类型对应一个浮点类型。
- 主模板有一个用 false 值初始化的静态 const 布尔成员，全特化将该成员的值设置为 true。

is_float_point<T> 是一种类型特征，告诉我们一种类型是否是浮点类型。可以这样使用:

```
int main()
{
  static_assert(is_floating_point<float>::value);
  static_assert(is_floating_point<double>::value);
  static_assert(is_floating_point<long double>::value);
  static_assert(!is_floating_point<int>::value);
  static_assert(!is_floating_point<bool>::value);
}
```

这证明我们构建了正确的类型特征，但这并不是一个实际的用例场景。为了使这个类型特征真正有用，需要在编译时使用它来处理信息。

假设要构建一个函数，需要对浮点值做一些事情。有多种浮点类型，例如：float、double 和 long double。为了避免编写多个实现，将其构建为模板函数。可以将其他类型作为模板参数传递，所以需要一种方法来避免这种情况。一个简单的解决方案是使用前面看到的 static_assert()，若用户提供的值不是浮点数，就会报错:

```
template <typename T>
void process_real_number(T const value)
{
  static_assert(is_floating_point<T>::value);
  std::cout << "processing a real number: " << value
        << '\n';
}
int main()
{
  process_real_number(42.0);
  process_real_number(42); // error:
  // static assertion failed
}
```

这是一个非常简单的示例，演示了使用类型特征进行条件编译。除了使用 static_assert()，还有其他方法。现在，来看看第二个例子。

假设有定义写入输出流的操作的类，这基本上是序列化的一种形式。但有些方法使用重载操作符 << 进行支持，其他方法则使用名为 write 的成员函数:

```
struct widget
{
  int id;
  std::string name;

  std::ostream& write(std::ostream& os) const
```

```
  {
    os << id << ',' << name << '\n';
    return os;
  }
};

struct gadget
{
  int id;
  std::string name;

  friend std::ostream& operator <<(std::ostream& os,
                                   gadget const& o);
};

std::ostream& operator <<(std::ostream& os,
gadget const& o)
{
  os << o.id << ',' << o.name << '\n';
  return os;
}
```

本例中，widget 类包含一个成员函数 write。但对于 gadget 类，流操作符 << 出于同样的目的进行重载。可以使用这些类编写以下代码:

```
widget w{ 1, "one" };
w.write(std::cout);

gadget g{ 2, "two" };
std::cout << g;
```

然而，我们的目标是定义一个函数模板，并且能够以相同的方式对待它们。换句话说，不使用 write 或 << 操作符:

```
serialize(std::cout, w);
serialize(std::cout, g);
```

这就带来了一些问题。首先，这样的函数模板是什么样的? 其次，如何知道类型是否提供了写入方法或 << 操作符重载? 第二个问题的答案是类型特征。可以构建一个类型特征，帮助我们在编译时回答后一个问题。所以需要这样的类型特征:

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

template <>
struct uses_write<widget>
{
  static constexpr bool value = true;
```

```
};
```

这与之前定义的类型特征非常相似，uses_write 说明一个类型是否定义了写成员函数。主模板将名为 value 的数据成员设置为 false，但 widget 类的全特化将其设置为 true。为了避免 uses_write<T>::value 的冗长语法，还可以定义一个变量模板，将语法简化为 uses_write_v<T> 的形式。这个变量模板如下所示:

```
template <typename T>
inline constexpr bool uses_write_v = uses_write<T>::value;
```

为了使例子更简单，假设不提供 write 成员函数的类型会重载输出流操作符。实践中，情况并非如此，为了简单起见，我们将基于此假设进行构建。

定义为序列化所有类提供统一 API 的函数模板 serialize 的下一步是定义更多的类模板，它们将遵循相同的方式——提供一种形式的序列化的主模板和提供另一种形式的全特化:

```
template <bool>
struct serializer
{
  template <typename T>
  static void serialize(std::ostream& os, T const& value)
  {
    os << value;
  }
};

template<>
struct serializer<true>
{
  template <typename T>
  static void serialize(std::ostream& os, T const& value)
  {
    value.write(os);
  }
};
```

serializer 类模板有一个模板参数，是非类型模板参数，也是一个匿名模板参数 (不会在实现中使用)。这个类模板包含一个成员函数，实际上是具有单一类型模板参数的成员函数模板，这个参数定义了要序列化的值的类型，主模板使用 << 操作符将值输出到提供的流。另一方面，serializer 类模板的全特化使用成员函数 write 来完成同样的工作。这里，我们全特化了序列化器类模板，而不是 serialize 成员函数模板。

现在只剩下实现所需的独立函数 serialize 了，其实现将基于 serializer<T>::serialize 函数:

```
template <typename T>
void serialize(std::ostream& os, T const& value)
{
  serializer<uses_write_v<T>>::serialize(os, value);
}
```

函数模板的签名与 serializer 类模板中的 serialize 成员函数的签名相同，主模板和全特化之间的选择可使用变量模板 uses_write_v 完成，其提供了一种方便的方式来访问 uses_write 类型特征的数据成员。

这些示例中，我们已经了解了如何实现类型特征，并使用它们在编译时提供的信息来对类型施加限制，或者在一个或另一个实现之间进行选择。类似的目的还有另一种称为 SFINAE 的元编程技术，我们将在下面的小节中对其进行介绍。

5.2. 了解 SFINAE

编写模板时，有时需要限制模板参数。例如，应该适用于任何数字类型，整型和浮点型，但不适用于其他类型。或者可以有一个类模板，只接受普通类型的参数。

还有一些可能有重载函数模板的情况，每个函数模板只适用于某些类型。例如，一种重载适用于整型，另一种重载适用于浮点类型。实现这一目标可以用不同的方法，我们将在本章和下一章中进行探讨。

然而，类型特征以某种方式存在于这些类型中。本章将讨论的第一个特性是 SFINAE。另一种优于 SFINAE 的方法由概念表示，将在下一章中讨论。

SFINAE 表示**替换失败不是错误**。当编译器遇到函数模板时，会替换实参以实例化模板。若此时发生错误，则只将其视为推导失败。函数会从重载集中移除，而不是引起错误。只有在重载集中没有匹配项时，才会产生错误。

没有具体的例子的情况下，很难真正理解 SFINAE。这里，将通过几个例子来了解一下这个概念。

每个标准容器，比如 std::vector、std::array 和 std::map，不仅有迭代器可以访问容器的元素，还可以修改容器 (在迭代器指向的元素之后插入元素)。因此，这些容器具有成员函数，用于返回容器的第一个元素和最后一个元素的迭代器，这些函数就是 begin 和 end。

还有其他函数，如 cbegin 和 cend, rbegin 和 rend，以及 crbegin 和 crend，不过这些函数超出了本章的主题范畴。C++11 中，也有独立函数 std:begin 和 std::end，可以完成同样的事情。这些函数不仅适用于标准容器，也适用于数组。这样做的一个好处是为数组启用基于范围的 for 循环。问题是如何实现这个非成员函数来同时使用容器和数组? 当然，需要函数模板的两次重载。一个可能的实现如下所示:

```
template <typename T>
auto begin(T& c) { return c.begin(); } // [1]

template <typename T, size_t N>
T* begin(T(&arr)[N]) {return arr; } // [2]
```

第一个重载调用成员函数 begin 并返回值，这种重载仅限于具有成员函数 begin 的类型; 否则，编译器将报错。第二个重载只返回一个指向数组第一个元素的指针。这仅限于数组类型，任何其他操作都会产生编译器错误。可以这样使用这些重载:

```
std::array<int, 5> arr1{ 1,2,3,4,5 };
std::cout << *begin(arr1) << '\n'; // [3] prints 1

```

```
int arr2[]{ 5,4,3,2,1 };
std::cout << *begin(arr2) << '\n'; // [4] prints 5
```

编译这段代码时，因为 SFINAE 编译过程不会出现错误，甚至不会出现警告。解析 begin(arr1) 调用时，将 std::array<int, 5> 替换为第一个重载 ([1]) 成功，但替换第二个重载 ([2]) 失败。编译器不会发出错误，而是忽略它，因此它用单个实例化构建一个重载集，所以可以成功地为调用找到匹配。类似地，解析 begin(arr2) 调用时，对第一个重载替换 int[5] 失败并忽略，但对第二个重载替换成功并添加到重载集，最终为调用找到一个良好的匹配。因此，两个调用都可以成功进行。若两个重载中有一个不存在，begin(arr1) 或 begin(arr2) 将无法匹配函数模板，并且编译器会报错。

SFINAE 只适用于函数的直接上下文。直接上下文基本上是模板声明 (包括模板参数列表、函数返回类型和函数参数列表)，不适用于函数体。看看下面的例子:

```
template <typename T>
void increment(T& val) { val++; }

int a = 42;
increment(a); // OK

std::string s{ "42" };
increment(s); // error
```

在增量函数模板的直接上下文中，对类型 T 没有限制。函数体中，参数 val 用后修正操作符 ++ 加 1。所以，将 T 替换为没有实现后修复操作符 ++ 的类型都是失败的。因为这个失败是一个错误，所以编译器不会忽略。

C++ 标准 (许可证使用链接:http://creativecommons.org/licenses/by-sa/3.0/) 定义了 SFINAE 错误的错误列表 (在 §13.10.2，模板参数推导，C++20 标准中)。以下行为为 SFINAE 错误:

- 创建一个 void 数组、一个引用数组、一个函数数组、一个长度为负的数组、一个长度为 0 的数组和一个非整型长度的数组
- 在作用域解析操作符:: 左侧使用不是类或枚举的类型 (例如：T::value_type 中，T 是一个数字类型)
- 创建指向引用的指针
- 创建对 void 的引用
- 创建指向 T 成员的指针，其中 T 不是类类型
- 类型不包含该成员时，使用类型的成员
- 使用类型的成员，其中类型是必需的，但该成员不是类型
- 使用需要模板，但成员不是模板类型的成员
- 需要非类型，但成员不是非类型的情况下使用类型的成员
- 创建具有 void 类型参数的函数类型
- 创建返回数组类型或其他函数类型的函数类型
- 模板参数表达式或函数声明中使用的表达式中进行了无效转换
- 为非类型模板形参提供无效类型
- 实例化包含不同长度的多个包的扩展

列表中的最后一个错误是在 C++11 中与可变参数模板一起引入的，其他的是在 C++11 之前定义的。我们不会继续逐一举例，说明所有这些错误，但需要再看几个例子。第一个问题涉及尝试创建一个长度为 0 的数组。假设想要有两个函数模板重载，一个处理偶数长度的数组，另一个处理奇数长度的数组。解决方法如下:

```
template <typename T, size_t N>
void handle(T(&arr)[N], char(*)[N % 2 == 0] = 0)
{
  std::cout << "handle even array\n";
}

template <typename T, size_t N>
void handle(T(&arr)[N], char(*)[N % 2 == 1] = 0)
{
  std::cout << "handle odd array\n";
}

int arr1[]{ 1,2,3,4,5 };
handle(arr1);

int arr2[]{ 1,2,3,4 };
handle(arr2);
```

模板参数和第一个函数参数类似于我们看到的数组的 begin，这些句柄重载有第二个匿名参数，其默认值为 0。该参数的类型是一个指向 char 类型数组的指针，数组长度由表达式 N%2==0 和 N%2==1 指定。对于每一个可能的数组，这两个中的一个为真，另一个为假。第二个参数是 char(*)[1] 或 char(*)[0]，后者是 SFINAE 错误 (试图创建一个长度为 0 的数组)，因此才能够调用其他重载，而不会产生编译器错误。

本节的最后一个示例将展示 SFINAE 尝试使用一个不存在的类的成员:

```
template <typename T>
struct foo
{
  using foo_type = T;
};

template <typename T>
struct bar
{
  using bar_type = T;
};

struct int_foo : foo<int> {};
struct int_bar : bar<int> {}
```

这里有两个类，foo 的成员类型是 foo_type，而 bar 的成员类型是 bar_type。还有派生自这两个类。目标是编写两个函数模板，一个处理类的 foo 层次结构，另一个处理类的 bar 层次结构。一个可能的实现如下所示:

```
template <typename T>
decltype(typename T::foo_type(), void()) handle(T const& v)
{
  std::cout << "handle a foo\n";
}

template <typename T>
decltype(typename T::bar_type(), void()) handle(T const& v)
{
  std::cout << "handle a bar\n";
}
```

两个重载都有一个模板参数和一个类型为 T const& 的函数参数，也返回相同的类型，并且该类型为 void。表达式 decltype(typename T::foo_type(), void()) 可能需要思考一下才能更好地理解。我们在第 4 章中讨论了 decltype，这是一个类型说明符，用于推断表达式的类型。我们使用逗号操作符，因此对第一个参数求值，但随后丢弃，因此 decltype 将从 void() 进行类型推导，并且推导出的类型为 void。然而，参数 typename T::foo_type() 和 typename T::bar_type() 确实使用了内部类型，而且这只存在于 foo 或 bar 中。这就是使用 SFINAE 的地方，如下所示:

```
int_foo fi;
int_bar bi;
int x = 0;
handle(fi); // OK
handle(bi); // OK
handle(x); // error
```

调用带有 int_foo 值的句柄将匹配第一个重载，而第二个因为替换失败而丢弃。类似地，调用带有 int_bar 值的句柄将匹配第二个重载，而第一个因为替换失败而丢弃。然而，使用 int 类型调用句柄将导致两个重载的替换失败，因此用于替换 int 类型的最终重载集将为空，从而调用没有匹配项，所以会产生编译错误。

SFINAE 并不是实现条件编译的最佳方式。现代 C++ 中，最好能与 enable_if 的类型特征一起使用。这就是我们接下来要讨论的问题。

5.3. 使用 enable_if 类型特性启用 SFINAE

C++ 标准库中，类型库是其子库之一。这个库定义了 std::size_t、std::nullptr_t 和 std::byte 等类型，运行时类型标识支持 std::type_info 等类型，以及类型特征的集合。类型特征有两类:

- 类型特征能够在编译时查询类型的属性。
- 能够在编译时执行类型转换的类型特征 (添加或删除 const 限定符，或从类型中添加或删除指针或引用)，这些类型特征也称为元功能。

来自第二类的一个类型特征是 std::enable_if，这用于启用 SFINAE 并从函数的重载集中删除候选。一个可能的实现如下所示:

```
template<bool B, typename T = void>
struct enable_if {};
```

```
template<typename T>
struct enable_if<true, T> { using type = T; };
```

有一个主模板，有两个模板参数，一个布尔型非类型模板和一个默认参数为 void 的类型参数。主模板是一个空类，非类型模板参数值也有偏特化。这定义了一个称为 type 的成员类型，它是模板参数 T 的别名模板。

enable_if 元函数用于布尔表达式，这个布尔表达式求值为 true 时，定义了一个名为 type 的成员类型。若布尔表达式为 false，则不定义此成员类型。来看看它是如何工作的。

还记得本章开头理解和定义类型特征一节中的例子吗? 例子中，有一些类提供了一个写入方法，将内容写入输出流，以及为了同样的目的重载了操作符 << 的类。在那一节中，我们定义了一个名为 uses_write 的类型特征，并编写了一个 serialize 函数模板，该模板允许以统一的方式序列化这两种类型的对象 (widget 和 gadget)。然而，实现相当复杂。使用 enable_if，可以以一种简单的方式实现该函数。一个可能的实现如下所示:

```
template <typename T,
      typename std::enable_if<
      uses_write_v<T>>::type* = nullptr>
void serialize(std::ostream& os, T const& value)
{
  value.write(os);
}

template <typename T,
      typename std::enable_if<
      !uses_write_v<T>>::type*=nullptr>
void serialize(std::ostream& os, T const& value)
{
  os << value;
}
```

这个实现中有两个重载函数模板，有两个模板参数。第一个参数是类型模板参数，称为 T。第二个参数是指针类型的匿名非类型模板参数，其默认值是 nullptr。只有当 uses_write_v 变量的值为 true 时，才使用 enable_if 定义成为 type 的成员。因此，对于具有成员函数 write 的类，第一次重载替换成功，但第二次重载替换失败，因为 typename * = nullptr 不是有效参数。对于操作符 << 重载的类，情况则相反。

enable_if 元函数可以在以下几种情况下使用:

- 定义具有默认参数的模板参数
- 定义具有默认参数的函数参数
- 指定函数的返回类型

出于这个原因，所以在前面提供 serialize 重载实现只是一种可能。类似的使用 enable_if 来定义一个带有默认参数的函数参数，如下所示:

```
template <typename T>
void serialize(
```

```
  std::ostream& os, T const& value,
  typename std::enable_if<
        uses_write_v<T>>::type* = nullptr)
{
  value.write(os);
}

template <typename T>
void serialize(
  std::ostream& os, T const& value,
  typename std::enable_if<
        !uses_write_v<T>>::type* = nullptr)
{
  os << value;
}
```

这里，我们把参数从模板参数列表移到了函数参数列表。没有其他变化，用法相同，如下所示:

```
widget w{ 1, "one" };
gadget g{ 2, "two" };

serialize(std::cout, w);
serialize(std::cout, g);
```

第三种选择是使用 enable_if 来包装函数的返回类型，实现略有不同 (默认参数对于返回类型没有意义):

```
template <typename T>
typename std::enable_if<uses_write_v<T>>::type serialize(
  std::ostream& os, T const& value)
{
  value.write(os);
}

template <typename T>
typename std::enable_if<!uses_write_v<T>>::type serialize(
  std::ostream& os, T const& value)
{
  os << value;
}
```

这个实现中，若 uses_write_v<T> 为 true，则定义返回类型。否则，替换失败，SFINAE。

尽管在所有这些示例中，enable_if 类型特征都用于在函数模板的重载解析期间启用 SFINAE，但此类型特征也可用于限制类模板的实例化。下面的例子中，有一个叫做 integral_wrapper 的类，只进行了整型类型实例化，还有一个叫做 float_wrapper 的类，只被浮点型类型实例化:

```
template <
  typename T,
  typename=typenamestd::enable_if_t<
            std::is_integral_v<T>>>
struct integral_wrapper
```

```
{
  T value;
};

template <
  typename T,
  typename=typename std::enable_if_t<
              std::is_floating_point_v<T>>>
struct floating_wrapper
{
  T value;
};
```

这两个类模板都有两个类型模板参数。第一个称为 T，但第二个匿名，有一个默认参数。根据布尔表达式的值，这个参数的值是否在 enable_if 类型特征的帮助下定义。

这个实现中，可以看到:

- 别名模板 std::enable_if_t，这是访问 std::enable_if<B, T>::type 成员类型的方法。其定义如下:

```
template <bool B, typename T = void>
using enable_if_t = typename enable_if<B,T>::type;
```

- 两个变量模板 std::is_integral_v 和 std::is_floating_point_v 是访问数据成员的方法，std::is_integral<T>::value 和 std::is_floating_point<T>::value。std::is_integral 和 std::is_float_point 类是标准类型特征，分别检查类型是整型还是浮点型。

前面展示的两个 wrapper 类模板可以按如下方式使用:

```
integral_wrapper w1{ 42 }; // OK
integral_wrapper w2{ 42.0 }; // error
integral_wrapper w3{ "42" }; // error

floating_wrapper w4{ 42 }; // error
floating_wrapper w5{ 42.0 }; // OK
floating_wrapper w6{ "42" }; // error
```

其中只有两个实例化可以工作：w1，integral_wrapper 是用 int 类型实例化的；w5，float_wrapper 是用 double 类型实例化的。所有其他选项都会让编译器报错。

需要指出的是，此代码示例仅适用于 C++20 中提供的 integral_wrapper 和 float_wrapper 的定义。对于标准的以前版本，因为编译器无法推导模板参数，即使是 w1 和 w5 的定义也会产生编译器错误。为了使其工作，必须更改类模板以包括一个构造函数，如下所示:

```
template <
  typename T,
  typename=typenamestd::enable_if_t<
              std::is_integral_v<T>>>
struct integral_wrapper
{
  T value;

```

```
  integral_wrapper(T v) : value(v) {}
};

template <
  typename T,
  typename=typename std::enable_if_t<
             std::is_floating_point_v<T>>>
struct floating_wrapper
{
  T value;

  floating_wrapper(T v) : value(v) {}
};
```

虽然 enable_if 有助于通过更简单、更可读的代码实现 SFINAE，但它仍然复杂。幸运的是，C++17 中有一个更好的选择——constexpr if。接下来让我们来探索一下这个替代方案。

5.4. constexpr if

C++17 的 constexpr if 特性使得 SFINAE 更加简单，是 if 语句的编译时版本，有助于用更简单的版本替换复杂的模板代码。先看一下 serialize 函数的 C++17 实现，可以统一序列化 widget 和 gadget:

```
template <typename T>
void serialize(std::ostream& os, T const& value)
{
  if constexpr (uses_write_v<T>)
    value.write(os);
  else
    os << value;
}
```

其语法是 if constexpr(condition)，条件必须是编译时表达式，求值表达式不执行短路逻辑。若表达式具有形式 a && b 或 a || b，那么 a 和 b 都必须是定义良好的。

constexpr if 能够在编译时根据表达式的值丢弃分支。例子中，当 uses_write_v 为 true 时，else 分支将丢弃，而第一个分支的主体将保留，最终得到了 widget 和 gadget 类的特化:

```
template<>
void serialize<widget>(std::ostream & os,
                       widget const & value)
{
  if constexpr(true)
  {
    value.write(os);
  }
}

template<>
void serialize<gadget>(std::ostream & os,
                       gadget const & value)
```

```
{
  if constexpr(false)
  {
  }
  else
  {
    os << value;
  }
}
```

当然，编译器可能会进一步简化这段代码，这些特化可能会像下面这样:

```
template<>
void serialize<widget>(std::ostream & os,
                       widget const & value)
{
  value.write(os);
}

template<>
void serialize<gadget>(std::ostream & os,
              gadget const & value)
{
  os << value;
}
```

最终结果与使用 SFINAE 和 enable_if 实现的结果相同，但这里编写的实际代码更简单，更容易理解。

constexpr if 是一个很好的简化代码的工具，在第 3 章中看到过，当时实现了一个名为 sum 的函数:

```
template <typename T, typename... Args>
T sum(T a, Args... args)
{
  if constexpr (sizeof...(args) == 0)
    return a;
  else
    return a + sum(args...);
}
```

本例中，constexpr if 协助我们避免出现两个重载，一个用于一般情况，另一个用于结束递归。本书中已经介绍了另一个例子，这个例子中，constexpr if 可以简化实现，是第 4 章中的阶乘函数模板。该函数的实现如下所示:

```
template <unsigned int n>
constexpr unsigned int factorial()
{
  return n * factorial<n - 1>();
}

```

```
template<>
constexpr unsigned int factorial<1>() { return 1; }

template<>
constexpr unsigned int factorial<0>() { return 1; }
```

使用 constexpr if，可以用一个模板替换所有这些，并让编译器负责提供正确的特化。C++17 版本的函数实现如下所示:

```
template <unsigned int n>
constexpr unsigned int factorial()
{
  if constexpr (n > 1)
    return n * factorial<n - 1>();
  else
    return 1;
}
```

constexpr if 在很多情况下都很有用。本节给出的最后一个示例是一个名为 are_equal 的函数模板，其决定提供的两个参数是否相等。通常，会认为使用 operator== 就足以确定两个值是否相等。大多数情况下是正确的，除了浮点数。因为浮点数可以存储而没有精度损失 (像 1,1.25,1.5 这样的数字，并且小数部分都可以表示为 2 的逆幂级数)，所以在比较浮点数时需要特别注意。通常，这是通过确保两个浮点值之间的差值小于某个阈值来解决的，所以函数的实现可能如下所示:

```
template <typename T>
bool are_equal(T const& a, T const& b)
{
  if constexpr (std::is_floating_point_v<T>)
    return std::abs(a - b) < 0.001;
  else
    return a == b;
}
```

当 T 类型是浮点类型时，将两个数字之差的绝对值与所选阈值进行比较。否则，退回到 operator==。这使得不仅可以对算术类型使用此函数，还可以对其他重载了相等操作符的类型使用此函数。

```
are_equal(1, 1); // OK
are_equal(1.999998, 1.999997); // OK
are_equal(std::string{ "1" }, std::string{ "1" }); // OK
are_equal(widget{ 1, "one" }, widget{ 1, "two" }); // error
```

可以使用参数类型为 int、double 和 std::string 调用 are_equal 函数模板，但尝试对 widget 类型的值执行相同的操作将使编译器报错，因为 == 操作符没有在此类型中进行重载。

本章中，我们已经了解了什么是类型特征，以及执行条件编译的不同方法。还看到了标准库中可用的一些类型特征。本章的第二部分，我们将探讨标准中关于类型特征的内容。

5.5. 探索标准类型特征

标准库提供了一系列类型特征，用于查询类型的属性以及对类型执行转换。这些类型特征可以在 <type_traits> 头文件中作为类型支持库使用。类型特征有以下几种:

- 查询类型类别 (主或复合类型)
- 查询类型属性
- 查询支持的操作
- 查询类型关系
- 修改 cv 说明符、引用、指针或符号
- 各种转换

尽管研究每一种类型的特征超出了本书的范畴，但将探讨这些类别，看看它们包含什么。下面的小节中，将列出构成这些类别的类型特征 (或大部分类型特征)。这些列表以及关于每种类型特征的详细信息可以在 C++ 标准中找到 (参见本章末尾的扩展阅读，以获得标准草案版本的链接) 或 cppreference.com 网站https://en.cppreference.com/w/cpp/header/type_traits(许可链接:http://creativecommons.org/licenses/by-sa/3.0/)。

5.5.1 查询类型类别

之前已经使用了几个类型特征，例如 std::is_integral，std::is_floating_point，以及 std::is_arithmetic。这些用于查询主类型和复合类型类别的一些标准类型特征。下表列出了所有这些类型的特征:

名称	描述
is_void	类型是否为 void 类型。
is_null_pointer	一个类型是否为 std::nullptr_t 类型。
is_integral	类型是否为整型，包括有符号、无符号和 cv 限定变量。整型类型为: • bool, char, char8_t(C++20), char16_t, char32_t, wchar_t, short, int, long 和 long long • 任何扩展整数类型
is_floating_point	类型是否为浮点类型，包括 cv 限定变量。T 可能的类型有 float，double 和 long double.
is_array	类型是否为数组类型。
is_enum	类型是否为枚举类型。
is_union	类型是否为联合类型。
is_class	类型是否是类类型，而不是联合类型。
is_function	类型是否为函数类型。除了 Lambda，重载调用操作符的类，指向函数的指针，而 std::function 类型除外。

名称	描述
is_pointer	类型是指向对象的指针、指向函数的指针还是 cv 限定变量。 这不包括指向成员对象的指针或指向成员函数的指针。
is_member_pointer	类型是指向非静态成员对象的指针， 还是指向非静态成员函数的指针。
is_member_object_pointer	类型是否为非静态成员对象指针。
is_member_function_pointer	类型是否为非静态成员函数指针。
is_lvalue_reference	类型是否为左值引用类型。
is_rvalue_reference	类型是否为右值引用类型。
is_reference	类型是否为引用类型，可以是左值引用类型， 也可以是右值引用类型。
is_fundamental	类型是否为基本类型。基本类型为算术类型， void 类型和 std::nullptr_t 类型。.
is_scalar	类型是标量类型还是标量类型的 cv-限定符版本。标量类型包括: • 算术类型 • 指针类型 • 指向成员类型的指针 • 枚举类型 • std::nullptr_t
is_object	类型是否为 cv-qualifier 版本的对象类型。 对象类型不是函数类型、引用类型或 void 类型。
is_compound	类型是复合类型还是复合类型的 cv 变体。 复合类型不是基本类型，它们是 • 数组 • 函数 • 类 • 联合 • 对象指针和函数指针 • 成员对象指针和成员函数指针 • 引用 • 枚举

表 5.1

这些类型特征在 C++11 中可用。从 C++17 开始，每个变量都有一个变量模板来简化对布尔成员 value 的访问。对于名称为 is_abc 的类型特征，则存在名为 is_abc_v 的变量模板。对于所有具有名为 value 的布尔成员的类型特征都是如此，这些变量的定义很简单。下面的代码展示了 is_arithmentc_v 变量模板的定义:

```cpp
template< class T >
inline constexpr bool is_arithmetic_v =
  is_arithmetic<T>::value;
```

下面是使用这些类型特征的例子:

```cpp
template <typename T>
std::string as_string(T value)
{
  if constexpr (std::is_null_pointer_v<T>)
    return "null";
  else if constexpr (std::is_arithmetic_v<T>)
    return std::to_string(value);
  else
    static_assert(always_false<T>);
}

std::cout << as_string(nullptr) << '\n'; // prints null
std::cout << as_string(true) << '\n'; // prints 1
std::cout << as_string('a') << '\n'; // prints a
std::cout << as_string(42) << '\n'; // prints 42
std::cout << as_string(42.0) << '\n'; // prints 42.000000
std::cout << as_string("42") << '\n'; // error
```

函数模板 as_string 返回一个包含 pass 值作为参数的字符串。它只适用于算术类型，并且适用于 nullptr_t，将为其返回值“null”。

聪明的读者一定注意到了 static_assert(always_false<T>)，并想知道这个 always_false<T> 表达式到底是什么。其是一个 bool 类型的变量模板，计算结果为 false。其定义简单如下:

```cpp
template<class T>
constexpr bool always_false = std::false_type::value;
```

static_assert(false) 会使程序格式不正确。原因是它的条件不依赖于模板参数，而是求值为 false。若模板中不能为 constexpr if 语句的子语句生成有效的特化，则程序是格式错误的 (不需要诊断)。为了避免这种情况，static_assert 的条件必须依赖于模板参数。对于 static_assert(always_false<T>)，编译器在模板实例化之前，不知道这将计算为 true 还是 false。

我们探索的下一类类型特征，是查询类型的属性。

5.5.2 查询类型属性

能够查询类型属性的类型特征如下所示:

名称	C++ 版本	描述
is_const	C++11	是否为 const 限定 (const 或 const volatile)。
is_volatile	C++11	是否为 volatile 限定 (volatile 或 const volatile)。

名称	C++ 版本	描述
is_trivial	C++11	是普通类型，还是 cv 限定变量。以下是简单类型: • 标量类型或标量类型的数组 • 具有简单的默认构造函数 或此类数组的简单的可复制类。
is_trivially_copyable	C++11	是否可复制。以下是可复制的类型: • 标量类型或标量类型的数组 • 可复制的类或此类的数组
is_standard_layout	C++11	是标准布局类型，还是 cv 限定类型 • 标量类型或标量类型的数组 • 标准布局类或此类的数组
is_empty	C++11	是否为空类型。空类型是一种类类型 (不是联合体)， 是否为空类型。空类型是一种类类型 (不是联合体)， 没有虚函数，没有虚基类，也没有非空基类。
is_polymorphic	C++11	是否为多态类型。多态类型至少继承 自有一个虚函数的类型 (不是联合体)。
is_abstract	C++11	是否为抽象类型。抽象类型至少继承 自有一个虚纯函数的类型 (不是联合体)。
is_final	C++14	是否是使用 final 说明符声明的类型。
is_aggregate	C++17	是否为聚合类型。
is_signed	C++11	是浮点类型，还是有符号整型。
is_unsigned	C++11	类型是无符号整型，还是 bool 类型。
is_bounded_array	C++20	是否为已知边界的数组类型 (比如 int[5])
is_unbounded_array	C++20	是否为未知边界的数组类型 (比如 int[]).
is_scoped_enum	C++23	是否为范围枚举类型。
has_unique_object_representation	C++17	是否可复制，并且该类型的两个具有相同值的对象 也具有相同表现的形式。

表 5.2

其中大多数可能很容易理解，但有两个乍一看似乎是相同的，is_trivial 和_trivial_copyable。对于标量类型或标量类型数组，两者都成立，也适用于可复制的类或此类数组，但 is_trivial 仅适用于具有普通默认构造函数的可复制类。

根据 C++20 标准 §11.4.4.1，若默认构造函数不是用户提供的，那么默认构造函数是普通的，类没有虚成员函数，没有虚基类，没有具有默认初始化式的非静态成员，其每个直接基类都有一个普通的默认构造函数，类的每个非静态成员也都有一个普通的默认构造函数。为了更好地理解这一点，来看看下面的例子:

```cpp
struct foo
{
  int a;
};

struct bar
{
  int a = 0;
};

struct tar
{
  int a = 0;
  tar() : a(0) {}
};

std::cout << std::is_trivial_v<foo> << '\n'; // true
std::cout << std::is_trivial_v<bar> << '\n'; // false
std::cout << std::is_trivial_v<tar> << '\n'; // false

std::cout << std::is_trivially_copyable_v<foo>
          << '\n'; // true
std::cout << std::is_trivially_copyable_v<bar>
          << '\n'; // true
std::cout << std::is_trivially_copyable_v<tar>
          << '\n'; // true
```

本例中，有三个类似的类。这三个变量 foo、bar 和 tar 都可复制，只有 foo 类是一个普通类，因为它有一个普通的默认构造函数。bar 类有一个带有默认初始化式的非静态成员，tar 类有一个用户定义的构造函数，这使得它们不普通。

除了复制能力之外，还可以在其他类型特征的帮助下查询支持的操作。

5.5.3 查询支持的操作

下面的类型特征可以查询类型支持的操作:

名称	描述
is_constructible is_trivially_constructible is_nothrow_constructible	是否有可以接受特定参数的构造函数。
is_default_constructible is_trivially_default_constructible is_nothrow_default_constructible	是否有默认构造函数。

名称	描述
is_copy_constructible is_tribially_copy_constructible is_nothrow_copy_constructible	是否具有复制构造函数
is_move_constructible is_trivially_move_constructible is_nothrow_move_constructible	是否具有移动构造函数。
is_assignable is_trivially_assignable is_nothrow_assignable	是否具有特定参数的赋值操作符。
is_copy_assignable is_trivially_copy_assignable is_nothrow_copy_assignable	是否为复制赋值运算符。
is_move_assignable is_trivially_move_assigneable is_nothrow_move_assignable	是否有移动赋值操作符。
is_destructible is_trivially_destructible is_nothrow_destructible	是否具有析构函数。
has_virtual_destructor	是否具有虚析构函数。
is_swappable_with is_swappable is_nothrow_swappable_with is_nothrow_swappable	是否可以交换相同类型的对象或不同类型的对象。

表 5.3

除了最后一个是在 C++17 引入，其他都在 C++11 引入。每种类型特征都有多个版本，包括用于检查普通操作或使用 noexcept 说明符声明为无异常抛出操作的版本。

现在来看看类型特征，以及如何查询类型之间的关系。

5.5.4 查询类型的关系

这里，可以找到几个类型特征，可以查询类型之间的关系。这些类型特征如下所示:

名称	C++ 版本	描述
is_same	C++11	两个类型是否相同，包括可能的 cv 限定符。
is_base_of	C++11	一个类型是否派生自另一个类型。
is_convertible is_nothrow_convertible	C++11 C++14	一种类型是否可以转换为另一种类型。
is_invocable is_invocable_r is_nothrow_invocable is_nothrow_invocable_r	C++17	是否可以调用一个类型与指定参数类型。
is_layout_compatible	C++20	检查两种类型是否具有兼容的布局。若两个类是相同类型 (忽略 cv 限定符)，或者其公共初始序列包含所有非静态数据成员和位字段，或者是具有相同底层类型的枚举，则为布局兼容。
is_pointer_ inconvertible_base _of	C++20	是否为另一类型的指针不可转换基类。

表 5.4

这里，使用最多的可能是 std::is_same。这种类型特征在判断两种类型是否相同时非常有用，不过这种类型的特征需要考虑 const 和 volatile 限定符，所以 int 和 int const 不是同一类型。

可以使用这个类型特征来扩展前面所示的 as_string 函数的实现，若用 true 或 false 参数调用，则会输出 1 或 0，而不是 true/false。可以为 bool 类型添加一个显式检查，并返回一个包含这两个值之一的字符串:

```
template <typename T>
std::string as_string(T value)
{
  if constexpr (std::is_null_pointer_v<T>)
    return "null";
  else if constexpr (std::is_same_v<T, bool>)
    return value ? "true" : "false";
  else if constexpr (std::is_arithmetic_v<T>)
    return std::to_string(value);
  else
    static_assert(always_false<T>);
}

std::cout << as_string(true) << '\n'; // prints true
std::cout << as_string(false) << '\n'; // prints false
```

目前看到的所有类型特征，都用于查询关于类型的某种信息。下一节中，将看到对类型执行进行修改的类型特征。

5.5.5 修改 cv 限定符、引用、指针或符号

类型上执行转换的类型特征也称为元函数。这些类型特征提供了一个称为 type 的成员类型 (typedef)，表示转换后的类型。这些类型特征包括:

名称	描述
add_cv add_const add_volatile	将 const、volatile 或两者都添加到类型中。
remove_cv remove_const remove_volatile	从类型中删除 const、volatile 或同时删标识符。
add_lvalue_reference add_rvalue_reference	向类型添加左值或右值引用。
remove_reference	从类型中删除引用 (左值或右值)。
remove_cvref	从类型中删除 const 和 volatile 限定符以及左值或右值引用。融合了 remove_cv 和 remove_reference。
add_pointer	添加指向类型的指针。
remove_pointer	从类型中移除指针。
make_signed make_unsigned	创建有符号或无符号的整型 (bool 类型除外) 或枚举类型。支持的整数类型有 short, int, long, long long, char, wchar_t, char8_t, char16_t 和 char32_t。
remove_extent remove_all_extents	从数组类型中移除一个范围或所有范围。

表 5.5

除了 remove_cvref 是在 C++20 中添加的，本表中列出的所有其他类型特征在 C++11 中可用。这些并不是标准库中的所有元函数，更多的将在下一节中看到。

5.5.6 各种转换

除了前面列出的元函数之外，还有其他执行类型转换的类型特征。其中常用的如下表所示:

名称	C++ 版本	描述
enable_if	C++11	启用从重载解析中删除函数重载或模板特化。
conditional	C++11	通过基于编译时布尔条件进行选择， 将 type 成员类型定义为两种可能类型之一。
decay	C++11	在类型上应用转换 (数组类型为数组到指针， 引用类型为左值到右值，函数类型为指针)， 删除 const 和 volatile 限定符，并使用结果成员 类型定义类型作为其自己的成员类型定义类型。
common_type	C++11	从一组类型中确定公共类型。
common_reference	C++20	从一组类型中确定公共引用类型。
underlaying_type	C++11	确定枚举类型的基础类型。
void_t	C++17	将类型序列映射到 void 类型的类型别名。
type_identity	C++20	提供成员 typedef 类型作为类型参数 T 的别名。

列表中，已经讨论了 enable_if，还有一些需要举例说明的类型特征。先来看看 std::decay，考虑一下 as_string 函数的另一种实现方式:

```
template <typename T>
std::string as_string(T&& value)
{
  if constexpr (std::is_null_pointer_v<T>)
    return "null";
  else if constexpr (std::is_same_v<T, bool>)
    return value ? "true" : "false";
  else if constexpr (std::is_arithmetic_v<T>)
    return std::to_string(value);
  else
    static_assert(always_false<T>);
}
```

其中的变化是将参数传递给函数的方式。不是按值传递，而是按右值引用传递，这是一个转发引用。可以通过传递右值 (例如字面量) 进行调用，但传递左值会触发编译器错误:

```
std::cout << as_string(true) << '\n'; // OK
std::cout << as_string(42) << '\n'; // OK

bool f = true;
std::cout << as_string(f) << '\n'; // error

int n = 42;
std::cout << as_string(n) << '\n'; // error
```

最后两个调用将导致 static_assert 失败，实际的类型模板参数是 bool& 和 int&。因此 std::is_same<bool, bool&> 将用 false 初始化值 member，std::is_arithmetic<int&> 也会做同样的事情。为了求值这些类型，需要忽略引用以及 const 和 volatile 限定符。这里使用的类型特征是 std::decay，其会执行几个转换。其执行的概念如下所示:

```cpp
template <typename T>
struct decay
{
private:
  using U = typename std::remove_reference_t<T>;
public:
  using type = typename std::conditional_t<
    std::is_array_v<U>,
    typename std::remove_extent_t<U>*,
    typename std::conditional_t<
      std::is_function<U>::value,
      typename std::add_pointer_t<U>,
      typename std::remove_cv_t<U>
    >
  >;
};
```

可以看到 std::decay 是在其他元函数的帮助下实现的，包括 std::conditional，是基于编译时表达式在一种类型或另一种类型之间进行选择的关键。这种类型特征会多次使用，若需要根据多个条件进行选择，那么可以这样做。

在 std::decay 的帮助下，可以修改 as_string 函数、剥离引用和 cv-限定符:

```cpp
template <typename T>
std::string as_string(T&& value)
{
  using value_type = std::decay_t<T>;

  if constexpr (std::is_null_pointer_v<value_type>)
    return "null";
  else if constexpr (std::is_same_v<value_type, bool>)
    return value ? "true" : "false";
  else if constexpr (std::is_arithmetic_v<value_type>)
    return std::to_string(value);
  else
    static_assert(always_false<T>);
}
```

通过修改实现，可使消除前面对 as_string 的编译错误。

std::decay 的实现中，多次使用了 std::conditional。这是一个相当好用的元函数，可以简化许多实现。第 2 章中，我们看到了一个例子，构建了一个名为 list_t 的列表类型。有一个名为 type 的成员别名模板，若列表的大小为 1，则该模板类型为 T，若列表的大小大于 1，则该模板类型为 std::vector<T>。再来回顾一下:

```cpp
template <typename T, size_t S>
struct list
{
  using type = std::vector<T>;
};
```

```
6
7  template <typename T>
8  struct list<T, 1>
9  {
10   using type = T;
11 };
12
13 template <typename T, size_t S>
14 using list_t = typename list<T, S>::type;
```

在 std::conditional 的帮助下，这个实现可以进行简化:

```
1  template <typename T, size_t S>
2  using list_t =
3    typename std::conditional<S ==
4           1, T, std::vector<T>>::type;
```

没有必要依赖类模板特化来定义这样的列表类型，整个解决方案可以简化为定义一个别名模板。可以用一些 static_assert 来验证它是否如预期的那样工作:

```
1  static_assert(std::is_same_v<list_t<int, 1>, int>);
2  static_assert(std::is_same_v<list_t<int, 2>,
3                std::vector<int>>);
```

举例说明每个标准类型特征的使用超出了本书的范畴。本章的下一节提供了更复杂的例子，需要使用几个标准的类型特征。

5.6. 实际使用类型特征的例子

本章的前一节中，已经探讨了标准库提供的各种类型特征。为每一种类型特征找到例子的确很困难，也没有必要。但有必要展示一些可以使用多种类型特征，来解决问题的示例。

5.6.1 实现一个复制算法

第一个示例问题是 std::copy 标准算法 (<algorithm>) 的可能实现。我们接下来将看到的不是实际的实现，而是一个可帮助我们更多地了解类型特征使用的实现。该算法的声明如下:

```
1  template <typename InputIt, typename OutputIt>
2  constexpr OutputIt copy(InputIt first, InputIt last,
3              OutputIt d_first);
```

这个函数只有在 C++20 中是 constexpr，但是可以在这里进行讨论。其所做的是将范围 [first, last) 中的所有元素复制到另一个以 d_first 开头的范围内。还有一个重载接受执行策略，还有一个版本 std::copy_if，复制与谓词匹配的所有元素，但这些对于我们的示例并不重要。这个函数的简单实现如下所示:

```
1  template <typename InputIt, typename OutputIt>
2  constexpr OutputIt copy(InputIt first, InputIt last,
3              OutputIt d_first)
```

```
{
  while (first != last)
  {
    *d_first++ = *first++;
  }
  return d_first;
}
```

但在某些情况下，这种实现可以通过简单地复制内存来优化。为了达到这个目的，必须满足一些条件:

- 两种迭代器类型 InputIt 和 OutputIt 都必须是指针。
- 两个模板参数 InputIt 和 OutputIt 必须指向相同的类型 (忽略 cv-限定符)。
- InputIt 所指向的类型，具有普通的复制赋值操作符。

可以用以下标准类型特征来检查这些条件:

- std::is_same(和 std::is_same_v 变量) 来检查两种类型是否相同。
- std::is_pointer(和 std::is_pointer_v 变量) 来检查一个类型是否是指针类型。
- std::is_trivially_copy_assignable(和 std::is_trivially_copy_assignable_v 变量) 检查类型是否具有普通的复制赋值操作符。
- std::remove_cv(和 std::remove_cv_t 别名模板) 从类型中删除 cv-限定符。

来看看如何实现。首先，需要有一个带有泛型实现的主模板，然后有一个带有优化实现的指针类型特化。可以使用带有成员函数模板的类模板来实现，如下所示:

```
namespace detail
{
  template <bool b>
  struct copy_fn
  {
    template<typename InputIt, typename OutputIt>
    constexpr static OutputIt copy(InputIt first,
                    InputIt last,
                    OutputIt d_first)
    {
      while (first != last)
      {
        *d_first++ = *first++;
      }
      return d_first;
    }
  };

  template <>
  struct copy_fn<true>
  {
    template<typename InputIt, typename OutputIt>
    constexpr static OutputIt* copy(
      InputIt* first, InputIt* last,
```

```
      OutputIt* d_first)
    {
      std::memmove(d_first, first,
            (last - first) * sizeof(InputIt));
      return d_first + (last - first);
    }
  };
}
```

为了在源和目标之间复制内存，这里使用 std::memmove，即使对象重叠，也会复制数据。这些实现是在一个名为 detail 的命名空间中，由复制函数轮流使用进行实现的细节，而不是直接由用户使用。该通用复制算法的实现方式如下:

```
template<typename InputIt, typename OutputIt>
constexpr OutputIt copy(InputIt first, InputIt last,
OutputIt d_first)
{
  using input_type = std::remove_cv_t<
    typename std::iterator_traits<InputIt>::value_type>;
  using output_type = std::remove_cv_t<
    typename std::iterator_traits<OutputIt>::value_type>;

  constexpr bool opt =
    std::is_same_v<input_type, output_type> &&
    std::is_pointer_v<InputIt> &&
    std::is_pointer_v<OutputIt> &&
    std::is_trivially_copy_assignable_v<input_type>;

  return detail::copy_fn<opt>::copy(first, last, d_first);
}
```

可以看到选择哪个特化的决定，是基于使用前面提到的类型特征确定的 constexpr 布尔值。下面的代码段显示了使用这个复制函数的例子:

```
std::vector<int> v1{ 1, 2, 3, 4, 5 };
std::vector<int> v2(5);

// calls the generic implementation
copy(std::begin(v1), std::end(v1), std::begin(v2));

int a1[5] = { 1,2,3,4,5 };
int a2[5];

// calls the optimized implementation
copy(a1, a1 + 5, a2);
```

这不是在标准库实现中找到的泛型算法副本的真正定义，标准库实现是进一步优化的。但这是一个很好的示例，可以演示如何将类型特征用于实际问题。

简单起见，我在全局命名空间中定义了复制函数。这是一个不好的做法。一般来说，代码 (尤其是库中的代码) 是按命名空间分组的。在本书附带的 GitHub 源代码中，会发现这个函数定义在一

个名为 n520 的命名空间中 (这只是一个名称，与主题无关)。当调用我们定义的复制函数时，实际上需要使用完全限定名 (包括命名空间的名称)，如下所示:

```
n520::copy(std::begin(v1), std::end(v1), std::begin(v2));
```

若没有这个条件，参数依赖查找 (ADL) 就会开始运作。因为传递的参数可以在 std 命名空间中找到，所以这将调用 std::copy 函数。可以在https://en.cppreference.com/w/cpp/language/adl上阅读更多关于 ADL 的信息。

接下来，看看下一个例子。

5.6.2 构建一个同构可变参的函数模板

对于第二个示例，我们希望构建一个可变参数函数模板，该模板只能接受相同类型的参数，或者可以隐式转换为公共类型的参数。从下面的框架定义开始:

```
template<typename... Ts>
void process(Ts&&... ts) {}
```

这样做的问题是，下面所有的函数调用都可以工作 (这个函数的函数体是空的，不会因为执行某些类型上不可用的操作而出现错误):

```
process(1, 2, 3);
process(1, 2.0, '3');
process(1, 2.0, "3");
```

第一个例子中，传递了三个 int 值。第二个例子中，传递了一个 int 型、一个 double 型和一个 char 型的值;int 和 char 都可以隐式转换为 double 类型，所以这是正确的。然而，第三个例子中，传递了一个 int 型、一个 double 型和一个 char 型 const* 的值，最后一个类型不能隐式转换为 int 型或 double 型。所以，最后一个调用应该会触发编译器错误，但实际上并没有。

为此，需要确保当函数参数的公共类型不可用时，编译器将生成一个错误。可以使用 static_assert 或 std::enable_if 和 SFINAE，但确实需要弄清楚是否存在一种常见类型。这在 std::common_type 类型特征的帮助下，可以进行判断。

std::common_type 是一个元函数，它在所有类型参数中定义公共类型，所有类型都可以隐式转换为该类型，所以 std::common_type<int, double, char>::type 将为 double 类型的别名。使用这个类型特征，可以构建另一个类型特征，从而说明共同类型是否存在。一个可能的实现如下所示:

```
template <typename, typename... Ts>
struct has_common_type : std::false_type {};
template <typename... Ts>
struct has_common_type<
    std::void_t<std::common_type_t<Ts...>>,
    Ts...>
  : std::true_type {};

template <typename... Ts>
constexpr bool has_common_type_v =
  sizeof...(Ts) < 2 ||
  has_common_type<void, Ts...>::value;
```

可以看到我们将实现基于其他几个类型特征。有 std::false_type 和 std::true_type，分别是 std::bool_constant<false> 和 std::bool_constant<true> 的类型别名。std::bool_constant 类在 C++17 中可用，反之，它是 bool 类型的 std::integral_constant 类特化的别名模板。最后一个类模板包装了一个指定类型的静态常量，其的概念实现如下所示 (也提供了一些操作):

```
template<class T, T v>
struct integral_constant
{
  static constexpr T value = v;
  using value_type = T;
};
```

这有助于简化需要定义布尔编译时值的类型特征的定义。

has_common_type 类的实现中使用的第三种类型的特征是 std::void_t，此类型特征定义了可变数量的类型与 void 类型之间的映射。可以使用它来构建公共类型 (若存在的话) 和 void 类型之间的映射。可使我们能够利用 SFINAE 对 has_common_type 类模板进行特化。

最后，定义了名为 has_common_type_v 的变量模板，以简化 has_common_type 特征的使用。

所有这些都可以用来修改流程函数模板的定义，以确保它只允许公共类型的参数。一个可能的实现如下所示:

```
template<typename... Ts,
     typename = std::enable_if_t<
            has_common_type_v<Ts...>>>
void process(Ts&&... ts)
{ }
```

因此，process(1, 2.0, ”3”) 这样的调用将产生编译器错误，因为对于这组参数没有重载的 process 函数。

如前所述，有不同的方法来使用 has_common_type 特征来实现定义的目标。其中之一是使用 std::enable_if，也可以使用 static_assert，而使用概念应该是更好的方式，我们将在下一章中看到相关的内容。

5.7. 总结

本章探讨了类型特征的概念，类型特征是定义关于类型的元信息或类型转换操作的类型。首先研究了类型特征是如何实现的，以及如何帮助我们解决问题。接下来，学习了 SFINAE，替换失败不是错误。这是一种使我们能够为模板参数提供约束的技术。

然后，看到在 C++17 中使用 enable_if 和 constexpr if，可以更好地实现这个目的。本章的第二部分中，研究了标准库中可用的类型特征，并演示了如何使用其中一些类型特征。本章的最后，几个真实的例子中，使用了多种类型特征来解决了特定的问题。

下一章中，我们将通过学习 C++20 的概念和约束来继续约束模板的参数。

5.8. 习题

1. 什么是类型特征?

2. 什么是 SFINAE?
3. 什么是 constexpr if?
4. std::is_same 可用来做什么?
5. std::conditional 可用来做什么?

5.9. 扩展阅读

- C++ Type traits, John Maddock and Steve Cleary, https://cs.brown.edu/~jwicks/boost/libs/type_traits/cxx_type_traits.htm
- N4861 Post-Prague 2020 C++ working draft, https://github.com/cplusplus/draft/releases/tag/n4861
- What is ADL?, Arthur O'Dwyer, https://quuxplusone.github.io/blog/2019/04/26/what-is-adl/

第 6 章 概念和约束

C++20 标准对包含概念和约束的模板元编程提供了一系列重大改进。约束是定义模板参数需求的一种现代方式，概念是一组命名的约束。概念为编写模板的传统方式提供了几个好处，主要是提高了代码的可读性、更好的诊断和减少编译时间。

本章中，将了解到以下内容:

- 概念的需求
- 定义概念
- 探索 requires 表达式
- 组合约束
- 模板中约束的顺序
- 约束非模板成员函数
- 约束类模板
- 约束变量模板和模板别名
- 更多指定约束的方法
- 使用概念来约束 auto 参数
- 探索标准概念库

本章结束时，将对 C++20 的概念有一个很好的理解，并对标准库提供的概念有所了解。

现在，我们将讨论是什么催生了概念的发展，以及概念的好处。

6.1. 概念的需求

概念可以让代码有更好的可读性和错误消息。了解如何使用概念之前，先来回顾一下之前的例子:

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

这个函数模板接受两个参数，并返回它们的和。实际上，返回的不是和，而是对两个参数应用加号运算符后的结果。用户定义的类型可以重载此操作符并执行某些特定的操作。术语“和”只有在讨论数学类型时才有意义，例如整型、浮点类型、std::complex 类型、矩阵类型、向量类型等。

对于字符串类型，加号操作符可以表示连接。对于大多数类型来说，重载根本没有意义。因此，只看函数的声明，而不检查它的主体，不能真正说这个函数可以接受什么作为输入，以及这个函数是做什么的。可以这样调用这个函数:

```
add(42, 1); // [1]
add(42.0, 1.0); // [2]
add("42"s, "1"s); // [3]
add("42", "1"); // [4] error: cannot add two pointers
```

前三个调用都很好。第一个调用添加两个整数，第二个调用添加两个 double 值，第三个调用连接两个 std::string 对象。但第四次调用将产生编译器错误，因为 const char * 替换了 T 类型模板参数，并且指针类型没有重载加号操作符。

这个 add 函数模板的目的是只允许传递算术类型的值，即整型和浮点型。C++20 之前，可以通过几种方式实现这一点。一种方法是使用 std::enable_if 和 SFINAE，就像在前一章看到的那样:

```
template <typename T,
  typename = typename std::enable_if_t
    <std::is_arithmetic_v<T>>>
T add(T const a, T const b)
{
  return a + b;
}
```

首先，可读性下降了。第二，类模板参数难以阅读，需要对模板有很好的了解才能理解，但 [3] 和 [4] 行上的调用都产生了编译器错误。不同的编译器会产生不同的错误消息。下面是三个主要编译器的代码:

- VC++ 17:

```
error C2672: 'add': no matching overloaded function found
error C2783: 'T add(const T,const T)': could not deduce
template argument for '<unnamed-symbol>'
```

- GCC 12:

```
prog.cc: In function 'int main()':
prog.cc:15:8: error: no matching function for call
to 'add(std::__cxx11::basic_string<char>, std::__
cxx11::basic_string<char>)'
15 |      add("42"s, "1"s);
   |      ~~~^~~~~~~~~~~~~
prog.cc:6:6: note: candidate: 'template<class T, class> T
add(T, T)'
6 | T add(T const a, T const b)
  |   ^~~
prog.cc:6:6: note: template argument deduction/
substitution failed:
In file included from /opt/wandbox/gcc-head/include/
c++/12.0.0/bits/move.h:57,
                 from /opt/wandbox/gcc-head/include/
c++/12.0.0/bits/nested_exception.h:40,
```

```
                 from /opt/wandbox/gcc-head/include/
c++/12.0.0/exception:154,
                 from /opt/wandbox/gcc-head/include/
c++/12.0.0/ios:39,
                 from /opt/wandbox/gcc-head/include/
c++/12.0.0/ostream:38,
                 from /opt/wandbox/gcc-head/include/
c++/12.0.0/iostream:39,
                 from prog.cc:1:
/opt/wandbox/gcc-head/include/c++/12.0.0/type_traits: In
substitution of 'template<bool _Cond, class _Tp> using
enable_if_t = typename std::enable_if::type [with bool _
Cond = false; _Tp = void]':
prog.cc:5:14: required from here
/opt/wandbox/gcc-head/include/c++/12.0.0/type_traits:2603:11:
error: no type named 'type' in 'struct std::enable_if<false,
void>'
2603 | using enable_if_t = typename enable_if<_Cond, _Tp>::type;
     |          ^~~~~~~~~~~
```

- Clang 13:

```
prog.cc:15:5: error: no matching function for call to
'add'
   add("42"s, "1"s);
   ^~~
prog.cc:6:6: note: candidate template ignored:
requirement 'std::is_arithmetic_v<std::string>' was not
satisfied [with T = std::string]
  T add(T const a, T const b)
    ^
```

GCC 中的错误消息非常冗长，VC++ 没有说明匹配模板参数失败的原因是什么。可以说，Clang 在提供可理解的错误消息方面做得更好。

C++20 之前，定义此函数限制的另一种方法是使用 static_assert:

```
template <typename T>
T add(T const a, T const b)
{
  static_assert(std::is_arithmetic_v<T>,
```

```
5           "Arithmetic type required");
6   return a + b;
7 }
```

这个实现中，回到了最初的问题，即仅通过查看函数的声明，不知道它会接受什么样的参数，没有任何限制。另一方面，错误信息如下所示:

- VC++ 17:

```
error C2338: Arithmetic type required
main.cpp(157): message : see reference to function
template instantiation 'T add<std::string>(const T,const
T)' being compiled
    with
    [
       T=std::string
    ]
```

- GCC 12:

```
prog.cc: In instantiation of 'T add(T, T) [with T =
std::__cxx11::basic_string<char>]':
prog.cc:15:8: required from here
prog.cc:7:24: error: static assertion failed: Arithmetic
type required
   7 | static_assert(std::is_arithmetic_v<T>,
"Arithmetic type required");
     |                    ~~~~~^~~~~~~~~~~~~~~~~~
prog.cc:7:24: note: 'std::is_arithmetic_v<std::__
cxx11::basic_string<char> >' evaluates to false
```

- Clang 13:

```
prog.cc:7:5: error: static_assert failed due to
requirement 'std::is_arithmetic_v<std::string>'
"Arithmetic type required"
    static_assert(std::is_arithmetic_v<T>, "Arithmetic
type required");
    ^ ~~~~~~~~~~~~~~~~~~~~~~~
prog.cc:15:5: note: in instantiation of function template
```

```
specialization 'add<std::string>' requested here
   add("42"s, "1"s);
   ^
```

不管编译器是什么，使用 static_assert 会出现类似的错误消息。

C++20 中，可以通过使用约束来改进这两个方面 (可读性和错误消息)。可以通过新关键字 requires 引入，如下所示:

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

requires 关键字引入了 requires 子句，定义了模板参数的约束。实际上，有两种可供选择的语法: 一种是 requires 子句跟在模板参数列表后面，如前所述; 另一种是 requires 子句跟在函数声明后面，如下所示:

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

这两种语法可以根据个人喜好进行选择，但这两种情况，可读性都比 C++20 之前的实现好得多。通过阅读声明，就知道 T 类型模板参数必须是算术类型。此外，函数只是两个数字的相加，不需要看定义就能知道。当调用带有无效参数的函数时，错误消息为:

- VC++ 17:

```
error C2672: 'add': no matching overloaded function found
error C7602: 'add': the associated constraints are not
satisfied
```

- GCC 12:

```
prog.cc: In function 'int main()':
prog.cc:15:8: error: no matching function for call
to 'add(std::__cxx11::basic_string<char>, std::__cxx11::
basic_string<char>)'
  15 |   add("42"s, "1"s);
```

```
    |    ~~~^~~~~~~~~~~~~
prog.cc:6:6: note: candidate: 'template<class T>
requires is_arithmetic_v<T> T add(T, T)'
  6 |   T add(T const a, T const b)
    |     ^~~
prog.cc:6:6: note: template argument deduction/substitution
 failed:
prog.cc:6:6: note: constraints not satisfied
prog.cc: In substitution of 'template<class
T> requires is_arithmetic_v<T> T add(T, T) [with T =
std::__cxx11::basic_string<char>]':
prog.cc:15:8: required from here
prog.cc:6:6: required by the constraints of
'template<class T> requires is_arithmetic_v<T> T add(T,
T)'
prog.cc:5:15: note: the expression 'is_arithmetic_v<T>
[with T = std::__cxx11::basic_string<char, std::char_
traits<char>, std::allocator<char> >]' evaluated to 'false'
   5 | requires std::is_arithmetic_v<T>
     |          ~~~~~^~~~~~~~~~~~~~~~~~
```

- Clang 13:

```
prog.cc:15:5: error: no matching function for call to
'add'
add("42"s, "1"s);
^~~
prog.cc:6:6: note: candidate template ignored:
constraints not satisfied [with T = std::string]
T add(T const a, T const b)
^
prog.cc:5:10: note: because 'std::is_arithmetic_
v<std::string>' evaluated to false
requires std::is_arithmetic_v<T>
^
```

错误消息遵循相同的模式:GCC 太冗长，VC++ 缺少基本信息 (未满足的约束)，而 Clang 更简洁，更好地定位错误的原因。总的来说，错误消息看起来有了改进，但仍有改进空间。

约束是在编译时求值为 true 或 false 的谓词。前一个例子中使用的表达式 std::is_arithmetic_v<T>，只是使用了一个标准类型特征 (前一章中看到过)，这些不同类型的表达式都可以在约束中使用。

下一节中，我们将讨论如何定义和使用命名约束。

6.2. 定义概念

约束是在使用它们的地方定义的匿名谓词。许多约束是通用的，可以在多个地方使用。以下是类似于 add 函数的函数示例，这个函数执行算术乘法:

```
template <typename T>
requires std::is_arithmetic_v<T>
T mul(T const a, T const b)
{
  return a * b;
}
```

与 add 函数相同的 requires 子句也出现在这里。为了避免这种重复的代码，可以定义一个可以在多个地方重用的名称约束，命名约束也称为概念。概念用新关键字 concept 和模板语法定义。这里有一个例子:

```
template<typename T>
concept arithmetic = std::is_arithmetic_v<T>;
```

即使赋值为布尔值，概念名也不应该包含动词。它们表示需求，并用作模板参数的属性或限定符，所以应该使用 arithmetic、copyable、serializable、container 等名称，而不是 is_arithmetic、is_copyable、is_serializable 和 is_container。前面定义的 arithmetic 概念可以这样使用:

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

template <arithmetic T>
T mul(T const a, T const b) { return a * b; }
```

这里直接使用了这个概念，而不是 typename 关键字。它用 arithmetic 来限定 T 类型，只有满足此要求的类型才能用作模板参数。相同的算术概念可以用不同的语法定义，如下所示:

```
template<typename T>
concept arithmetic = requires { std::is_arithmetic_v<T>; };
```

这使用了 requires 表达式，其使用大括号分支 {}，而 requires 子句没有。requires 表达式可以包含不同类型的需求序列: 简单需求、类型需求、复合需求和嵌套需求。这里看到的是一个简单的需求。为了定义这个特定的概念，这种语法更加复杂，但具有相同的最终效果。但在某些情况下，确实需要复杂的要求。

考虑这样一种情况，想要定义一个模板时，只接受容器类型作为参数。在概念出现之前，这个问题可以通过类型特征和 SFINAE 或 static_assert 来解决，就像在本章开头看到的那样。不过，容器类型实际上不容易定义，可以基于标准容器的一些属性来实现:

- 其成员类型有 value_type、size_type、allocator_type、iterator 和 const_iterator。

- 具有成员函数 size，该函数返回容器中元素的数量。
- 具有成员函数 begin/end 和 cbegin/cend，返回指向容器中第一个和倒数一个元素的迭代器和常量迭代器。

根据第 5 章的知识，可以这样定义 is_container 类型特征:

```
template <typename T, typename U = void>
struct is_container : std::false_type {};

template <typename T>
struct is_container<T,
  std::void_t<typename T::value_type,
        typename T::size_type,
        typename T::allocator_type,
        typename T::iterator,
        typename T::const_iterator,
        decltype(std::declval<T>().size()),
        decltype(std::declval<T>().begin()),
        decltype(std::declval<T>().end()),
        decltype(std::declval<T>().cbegin()),
        decltype(std::declval<T>().cend())>>
  : std::true_type{};

template <typename T, typename U = void>
constexpr bool is_container_v = is_container<T, U>::value;
```

可以在 static_assert 的帮助下验证类型特征是否正确地标识了容器类型:

```
struct foo {};

static_assert(!is_container_v<foo>);
static_assert(is_container_v<std::vector<foo>>);
```

概念使得编写这样的模板约束更加容易，可以使用概念语法和 requires 表达式来定义以下内容:

```
template <typename T>
concept container = requires(T t)
{
  typename T::value_type;
  typename T::size_type;
  typename T::allocator_type;
  typename T::iterator;
  typename T::const_iterator;
  t.size();
  t.begin();
  t.end();
  t.cbegin();
  t.cend();
};
```

这个定义更短，可读性更强。既使用简单需求，如 t.size()，也使用类型需求，如 typename T::value_type。可以用前面看到的方式来约束模板参数，但也可以与 static_assert 一起使用 (约束计算为编译时布尔值):

```
struct foo{};

static_assert(!container<foo>);
static_assert(container<std::vector<foo>>);

template <container C>
void process(C&& c) {}
```

下一节中，将深入探讨可以在 requires 表达式中使用的各种需求。

6.3. 探索 requires 表达式

requires 表达式可能是一个复杂表达式，如前面的容器概念示例中所示。requires 表达式的形式与函数语法非常相似:

```
requires (parameter-list) { requirement-seq }
```

参数列表是一个以逗号分隔的参数列表。与函数声明的唯一区别是不允许使用默认值，但此列表中指定的参数不具有存储空间、链接或生命周期。编译器不为它们分配任何内存，只用于定义需求。其有一个作用域，那就是 requires 表达式的右大括号。

requirements-seq 是一系列需求，每个这样的需求都必须以分号结束。这里，有四种类型的需求:

- 简单需求
- 类型需求
- 复合需求
- 嵌套需求

这些需求的参考如下:

- 模板范围内的参数
- requires 表达式的参数列表中可以引入局部参数
- 封闭上下文中可见的其他声明

下面的小节中，将探讨所有提到的需求类型。

6.3.1 简单需求

简单需求是一个不求值，只检查正确性的表达式。表达式必须是有效的，以便对需求求值为 true。因为它定义了嵌套的需求，所以表达式不能以 requires 关键字开头。

前面定义算术和容器概念时，已经看到了简单语句的示例。让我们再看几个:

```
template<typename T>
concept arithmetic = requires
{
  std::is_arithmetic_v<T>;
```

```
};

template <typename T>
concept addable = requires(T a, T b)
{
  a + b;
};

template <typename T>
concept logger = requires(T t)
{
  t.error("just");
  t.warning("a");
  t.info("demo");
};
```

第一个概念 arithmetic，和之前定义的一样。std::is_arithmetic_v<T> 表达式是一个简单的需求。当参数列表为空时，可以省略，只检查 T 类型模板参数是否为算术类型。

addable 和 logger 概念都有一个参数列表，因为正在检查 T 类型值的操作。表达式 a + b 是一个简单的要求，因为编译器只检查加号运算符是否为 T 类型重载。最后一个例子中，确保 T 类型有三个成员函数，分别是 error、warning 和 info，可以接受一个 const char* 类型的参数，或可以从 const char* 构造的其他类型的参数。但作为参数传递的实际值并不重要，因为这些调用从未执行过，只检查其正确性。

简要地阐述最后一个例子，并看看以下代码:

```
template <logger T>
void log_error(T& logger)
{}

struct console_logger
{
  void error(std::string_view text){}
  void warning(std::string_view text) {}
  void info(std::string_view text) {}
};

struct stream_logger
{
  void error(std::string_view text, bool = false) {}
  void warning(std::string_view text, bool = false) {}
  void info(std::string_view text, bool) {}
};
```

log_error 函数模板需要一个类型满足记录器要求的参数。这里有两个类，分别为 console_logger 和 stream_logger。第一个满足 logger 的要求，但第二个不满足。这是因为 info 函数不能用 const char* 类型的单个参数调用。该函数还需要第二个布尔参数。前两个方法，error 和 warning，为第二个参数定义了默认值，所以可以通过 t.error(”just”) 和 warning(”a”) 进行调用。

然而，由于第三个成员函数，stream_logger 不是一个满足预期要求的 log 类，因此不能与 log_error 函数一起使用。console_logger 和 stream_logger 的使用示例如下:

```
console_logger cl;
log_error(cl); // OK

stream_logger sl;
log_error(sl); // error
```

下一节中，将讨论第二类需求：类型需求。

6.3.2 类型需求

类型需求是通过关键字 typename，后面跟着类型名称引入的。定义容器约束时，已经看到了几个例子。类型名必须有效，需求才为真。类型需求可用于以下几个目的:

- 验证嵌套类型是否存在 (例如 typename T::value_type;)
- 验证类模板特化是否命名了类型
- 验证别名模板特化是否命名了类型

看几个例子来学习如何使用类型需求。第一个例子中，检查一个类型是否包含内部类型，key_type 和 value_type:

```
template <typename T>
concept KVP = requires
{
  typename T::key_type;
  typename T::value_type;
};

template <typename T, typename V>
struct key_value_pair
{
  using key_type = T;
  using value_type = V;

  key_type key;
  value_type value;
};

static_assert(KVP<key_value_pair<int, std::string>>);
static_assert(!KVP<std::pair<int, std::string>>);
```

类型 key_value_pair<int, std::string> 满足这些类型要求，但 std::pair<int, std::string> 不满足。pair 类型确实有内部类型，但称为 first_type 和 second_type。

第二个例子中，检查类模板特化是否命名了类型。类模板是 container，特化是 container<T>:

```
template <typename T>
requires std::is_arithmetic_v<T>
struct container
```

```
{ /* ... */ };

template <typename T>
concept containerizeable = requires {
  typename container<T>;
};

static_assert(containerizeable<int>);
static_assert(!containerizeable<std::string>);
```

这段代码中，container 是一个类模板，只能针对算术类型特化，例如 int、long、float 或 double。因此，特化 container<int> 存在，但 container<std::string> 不存在。containerizeable 概念指定了类型 T 定义有效的 container 特化的需求。因此，containerizeable<int> 为 true，但 containerizeable<std::string> 为 false。

现在已经理解了简单的需求和类型需求，是时候探索更复杂的需求类别了。

6.3.3 复合需求

简单需求允许验证表达式是否有效，但有时需要验证表达式的某些属性，而不仅仅是其是否有效。这可以包括表达式是否对结果类型抛出异常或要求 (例如函数的返回类型)。一般形式如下:

```
{ expression } noexcept -> type_constraint;
```

noexcept 规范和 type_constraint(前导为->) 都是可选的。替换过程和约束的检查如下所示:

1. 替换表达式中的模板参数。
2. 若指定了 noexcept，则表达式不能抛出异常; 否则，需求为 false。
3. 若存在类型约束，则模板参数也会替换为 type_constraint，并且 decltype((expression)) 必须满足 type_constraint 所施加的条件; 否则，需求为 false。

这里讨论几个示例，来学习如何使用复合需求。第一个例子中，检查函数是否标为 noexcept:

```
template <typename T>
void f(T) noexcept {}

template <typename T>
void g(T) {}

template <typename F, typename ... T>
concept NonThrowing = requires(F && func, T ... t)
{
  {func(t...)} noexcept;
};

template <typename F, typename ... T>
  requires NonThrowing<F, T...>
void invoke(F&& func, T... t)
{
  func(t...);
```

```
}
```

这里，有两个函数模板:f 声明为 noexcept;，所以不会抛出异常，而 g 可能会抛出异常。NonThrowing 概念强制要求类型为 F 的可变函数不能抛出异常。因此，在以下两个调用中，只有第一个是有效的，而第二个将产生编译错误:

```
invoke(f<int>, 42);
invoke(g<int>, 42); // error
```

Clang 生成的错误消息如下表所示:

```
prog.cc:28:7: error: no matching function for call to 'invoke'
     invoke(g<int>, 42);
      ^~~~~~
prog.cc:18:9: note: candidate template ignored: constraints not
satisfied [with F = void (&)(int), T = <int>]
   void invoke(F&& func, T... t)
        ^
prog.cc:17:16: note: because 'NonThrowing<void (&)(int), int>'
evaluated to false
     requires NonThrowing<F, T...>
              ^
prog.cc:13:20: note: because 'func(t)' may throw an exception
     {func(t...)} noexcept;
                  ^
```

因为 g<int> 可能抛出异常，导致 nonthrows <F, T⋯> 计算为 false，所以 invoke(g<int>,42) 无效。

对于第二个示例，我们将定义一个概念，为 timer 类提供需求。它要求存在一个名为 start 的函数，可以在没有参数的情况下调用它，并且它返回 void。还要求存在第二个名为 stop 的函数，可以在没有任何参数的情况下调用，并且返回一个可以转换为 long long 的值。概念定义如下:

```
template <typename T>
concept timer = requires(T t)
{
  {t.start()} -> std::same_as<void>;
  {t.stop()} -> std::convertible_to<long long>;
};
```

注意，类型约束不能是任何编译时布尔表达式，而是实际的类型需求，所以使用其他概念来指定返回类型。std::same_as 和 std::convertible_to 都是在标准库 <concepts> 头文件中可用的概念。现在，考虑以下计时器的类实现:

```
struct timerA
{
  void start() {}
```

```
  long long stop() { return 0; }
};

struct timerB
{
  void start() {}
  int stop() { return 0; }
};

struct timerC
{
  void start() {}
  void stop() {}
  long long getTicks() { return 0; }
};

static_assert(timer<timerA>);
static_assert(timer<timerB>);
static_assert(!timer<timerC>);
```

本例中，timerA 满足 timer 概念，包含两个必需的方法: 返回 void 的 start 方法和返回 long long 的 stop 方法。类似地，timerB 也满足 timer 概念，具有相同的方法，即使 stop 返回 int。然而，int 类型可隐式转换为 long long 类型，所以需要满足类型要求。最后，timerC 也有相同的方法，但返回 void，所以停止返回类型的类型要求不满足，因此，timer 概念的约束条件不满足。

最后一类需要研究的需求是嵌套需求。

6.3.4 嵌套需求

最后一类需求是嵌套需求。嵌套需求是通过 requires 关键字引入的 (简单需求不使用 requires 关键字引入)，其形式如下:

```
requires constraint-expression;
```

表达式必须由替换的参数来满足，将模板参数替换为约束表达式，只是为了检查表达式是否满足。

下面的例子中，要定义一个函数，对可变数量的参数执行加法。但我们想提一些条件:

- 有多个参数。
- 所有的参数都有相同的类型。
- 表达式 arg1 + arg2 +···+ argn 是有效的。

这里，我们定义了一个名为 HomogenousRange 的概念，如下所示:

```
template<typename T, typename... Ts>
inline constexpr bool are_same_v =
  std::conjunction_v<std::is_same<T, Ts>...>;

template <typename ... T>
concept HomogenousRange = requires(T... t)
```

```
{
  (... + t);
  requires are_same_v<T...>;
  requires sizeof...(T) > 1;
};
```

这个概念包含一个简单的需求和两个嵌套的需求。一个嵌套需求使用 are_same_v 变量模板，其值由一个或多个类型特征 (std::is_same) 连接决定的，另一个是编译时布尔表达式 size···(T) > 1。

使用这个概念，可以这样定义 add 可变函数模板:

```
template <typename ... T>
requires HomogenousRange<T...>
auto add(T&&... t)
{
  return (... + t);
}

add(1, 2); // OK
add(1, 2.0); // error, types not the same
add(1); // error, size not greater than 1
```

前面举例的第一个调用是正确的，因为有两个参数，而且都是 int 类型。第二次调用产生错误，因为实参的类型不同 (int 和 double)。类似地，第三个调用也会产生错误，因为只提供了一个参数。

HomogenousRange 概念也可以在几个 static_assert 的辅助下测试，如下所示:

```
static_assert(HomogenousRange<int, int>);
static_assert(!HomogenousRange<int>);
static_assert(!HomogenousRange<int, double>);
```

我们已经介绍了可用于定义约束的所有需求表达式的类别，约束也可以组合，这就是我们接下来要讨论的内容。

6.4. 组合约束

我们已经看到了多个约束模板参数的例子，但目前为止，我们使用的都是单一约束。不过，可以使用 && 和 || 操作符组合约束。使用 && 操作符组合两个约束称为合取，使用 || 操作符组合两个约束称为析取。

要使一个合取为真，两个约束都必须为真。与逻辑与操作类似，两个约束从左到右求值，若左边的约束为假，则不对右边的约束求值。来看一个例子:

```
template <typename T>
requires std::is_integral_v<T> && std::is_signed_v<T>
T decrement(T value)
{
  return value--;
}
```

这段代码中，有一个函数模板，返回接收到的参数的递减值，但只接受带符号的整数值。这是通过连接两个约束来指定的，std::is_integral_v<T> && std::is_signed_v<T>。使用不同的方法定义连接可以获得相同的结果:

```
template <typename T>
concept Integral = std::is_integral_v<T>;

template <typename T>
concept Signed = std::is_signed_v<T>;

template <typename T>
concept SignedIntegral = Integral<T> && Signed<T>;

template <SignedIngeral T>
T decrement(T value)
{
  return value--;
}
```

可以看到这里定义了三个概念: 一个约束整型，一个约束有符号类型，一个约束整型和有符号类型。

析取的工作原理与此类似。要使析取为真，至少有一个约束条件必须为真。若左边的约束为真，则不对右边的约束求值。再看一个例子，若还记得本章第一节的 add 函数模板，我们用 std::is_arithmetic 类型特征来约束它。还可以使用 std::is_integral 和 std::is_float_point 得到相同的结果:

```
template <typename T>
requires std::is_integral_v<T> || std::is_floating_point_v<T>
T add(T a, T b)
{
  return a + b;
}
```

表达式 std::is_integral_v<T> || std::is_float_point_v<T> 定义了两个原子约束的析取。稍后我们将更详细地讨论这种约束。目前，原子约束是 bool 类型的表达式，不能分解为更小的部分。类似地，对于我们之前所做的，也可以建立一个析取的概念并使用:

```
template <typename T>
concept Integral = std::is_integral_v<T>;

template <typename T>
concept FloatingPoint = std::is_floating_point_v<T>;

template <typename T>
concept Number = Integral<T> || FloatingPoint<T>;

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

如前所述，合取和析取是短路的。这对检查程序的正确性有重要的意义。考虑形式为 A<T> && B<T> 的连接，则首先检查并求值 A<T>；若为假，则不再检查第二个约束 B<T>。

类似地，对于 A<T> || B<T> 的析取，在检查 A<T> 后，若计算结果为真，则不会检查第二个约束 B<T>。若希望检查两个合取的格式是否良好，要么在之后通过其布尔值进行确定，要么不使用 && 和 || 操作符。只有当 && 和 || 分别嵌套在括号中或作为 && 或 || 的操作数出现时，才会形成合取或析取。否则，这些操作符将视为逻辑操作符。这里，用例子来解释一下:

```
template <typename T>
requires A<T> || B<T>
void f() {}

template <typename T>
requires (A<T> || B<T>)
void f() {}

template <typename T>
requires A<T> && (!A<T> || B<T>)
void f() {}
```

所有这些示例中，|| 定义了一个析取。然而，当在强制转换表达式或逻辑 NOT 中使用时，&& 和 || 定义了一个逻辑表达式:

```
template <typename T>
requires (!(A<T> || B<T>))
void f() {}

template <typename T>
requires (static_cast<bool>(A<T> || B<T>))
void f() {}
```

这种情况下，首先检查整个表达式的正确性，然后确定其布尔值。后面的例子中，两个表达式!(A<T> || B<T>) 和 static_cast<bool>(A<T> || B<T>) 都需要包装在另一组括号中，因为 requires 子句的表达式不能以! 标记或进行转换。

合取和析取符不能用于约束模板参数包，但有一种变通方法可以实现它。考虑 add 函数模板的可变参数实现，要求所有参数必须是整型。可以尝试以以下形式编写约束:

```
template <typename ... T>
requires std::is_integral_v<T> && ...
auto add(T ... args)
{
  return (args + ...);
}
```

这将产生编译器错误，因为在此上下文中不允许使用省略号。为了避免这个错误，可以把表达式用括号括起来:

```
  template <typename ... T>
  requires std::is_integral_v<T> && ...
```

```
  auto add(T ... args)
  {
    return (args + ...);
  }
```

表达式 (std::is_integral_v<T> &&…) 现在是一个折叠表达式，不是一个合取，这样就得到了一个原子约束。编译器将首先检查整个表达式的正确性，然后确定其布尔值。要构建一个合取，首先需要定义一个概念:

```
template <typename T>
concept Integral = std::is_integral_v<T>;
```

接下来我们需要做的是更改 requires 子句，以便使用新定义的概念，而不是布尔变量 std::is_integral_v<T>:

```
template <typename ... T>
requires (Integral<T> && ...)
auto add(T ... args)
{
  return (args + ...);
}
```

看起来变化不大，由于使用了概念，验证正确性和确定布尔值将分别针对每个模板参数进行。若某个类型的约束不满足，其余部分将短路，验证将停止。

在本节前面，我两次使用了原子约束这个术语。有人会问，什么是原子约束? 它是 bool 类型的表达式，不能进一步分解。原子约束是在约束规范化过程中形成的，编译器将约束分解为原子约束的合取和析取。其工作原理如下:

- 表达式 E1 && E2 可分解为 E1 和 E2 的合取。
- 表达式 E1 || E2 可分解为 E1 和 E2 的析取。
- 概念 C<A1, A2，…An> 在将所有模板参数替换为其原子约束后，其定义被替换。

原子约束用于确定约束的部分排序，这些约束又决定函数模板和类模板特化部分的排序，以及重载解析中非模板函数的下一个候选。

6.5. 模板中约束的顺序

当编译器遇到函数调用或类模板实例化时，就需要确定哪种重载 (对于函数) 或特化 (对于类) 是最佳匹配。函数可以用不同的类型约束重载，类模板也可以用不同的类型约束特化。为了决定哪个是最佳匹配，编译器需要找出哪个是最匹配约束的，在替换所有模板参数后的同时，计算结果为 true。为了弄清楚这一点，需要进行约束归一化。这是将约束表达式转换为原子约束合取和析取的过程，如前一节末尾所述。

若 A 包含 B，则一个原子约束 A 包含另一个原子约束 B。一个约束声明 D1 的约束包含另一个声明 D2 的约束，则该约束声明 D1 至少与 D2 一样具有约束。若 D1 至少和 D2 一样受约束，反之则不成立，所以 D1 比 D2 的约束更强。编译器会选择约束更强的重载作为最佳匹配。

为了理解约束如何影响重载解析，这里讨论几个示例。首先，从以下两个重载开始:

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

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

第一个重载是非模板函数，接受两个 int 参数并返回它们的和。第二个是我们在本章中已经看到的模板实现。

现在，来看下如下的调用:

```
add(1.0, 2.0); // [1]
add(1, 2); // [2]
```

第一个调用 ([1]) 接受两个 double 值，因此只有模板重载匹配，其 double 类型的实例化将调用。add 函数的第二次调用 ([2]) 接受两个整数参数，两个重载都可能匹配。编译器将选择最具体的一个，即非模板重载。

若两个重载都是模板，但其中一个是受约束的，该怎么办?

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

template <typename T>
requires std::is_integral_v<T>
T add(T a, T b)
{
  return a + b;
}
```

第一个重载是前面看到的函数模板。第二个具有相同的实现，指定了模板参数的要求，该参数仅限于整型。若考虑前面代码片段中的相同两个调用，对于 [1] 处具有两个 double 值的调用，只有第一个重载匹配良好。对于 [2] 处的调用，有两个整数值，两个重载都很好的匹配。然而，第二个重载的约束更强 (与第一个没有约束的重载相比，只有一个约束)，因此编译器将选择这个重载。

下一个示例中，这两个重载都有约束。第一次重载要求模板实参的大小为 4，第二次重载要求模板实参必须是整型:

```
template <typename T>
requires (sizeof(T) == 4)
T add(T a, T b)
{
  return a + b;
```

```
}

template <typename T>
requires std::is_integral_v<T>
T add(T a, T b)
{
  return a + b;
}
```

考虑以下对这个重载函数模板的调用:

```
add((short)1, (short)2); // [1]
add(1, 2); // [2]
```

[1] 的调用使用 short 类型的参数。这是一个大小为 2 的整型，只有第二个重载匹配。但 [2] 处的调用使用 int 类型的参数，这是一个大小为 4 的整型所以，这两个重载都是很好的匹配。这是一种模棱两可的情况，编译器无法在两者之间进行选择，所以会产生编译错误。

但若稍微改变这两个重载，会发生什么呢?

```
template <typename T>
requires std::is_integral_v<T>
T add(T a, T b)
{
  return a + b;
}

template <typename T>
requires std::is_integral_v<T> && (sizeof(T) == 4)
T add(T a, T b)
{
  return a + b;
}
```

这两种重载都要求模板参数必须是整型，但第二次重载还要求整型的大小必须是 4 字节。对于第二个重载，使用两个原子约束的结合。我们将讨论同样的两次调用，使用 short 参数和 int 参数。

对于 [1] 上的调用，传递两个 short 值，只有第一个重载是良好的匹配，因此将调用这个重载。对于接受两个 int 参数的 [2] 调用，两个重载都是匹配的。第二种情况更受约束，所以编译器无法决定哪个匹配更好，并将发出一个模糊调用的错误。这可能会让开发者感到惊讶，因为在开始时，我说过将从重载集中选择最受约束的重载。因为我们使用类型特征来约束这两个函数，所以其在我们的示例中它不起作用。若使用概念，行为就不一样了:

```
template <typename T>
concept Integral = std::is_integral_v<T>;

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

```
}

template <typename T>
requires Integral<T> && (sizeof(T) == 4)
T add(T a, T b)
{
  return a + b;
}
```

现在，不再有歧义，编译器将从重载集中选择第二个重载作为最佳匹配。这说明编译器优先处理概念，使用概念使用约束有不同的方法，但前面的定义只是用概念替换了类型特征，所以可以说是演示这种行为的更好选择，而不是下一个实现:

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

template <Integral T>
requires (sizeof(T) == 4)
T add(T a, T b)
{
  return a + b;
}
```

本章中讨论的所有示例都涉及到约束函数模板，但可以约束非模板成员函数，以及类模板和类模板特化。我们将在下一节中讨论这些。

6.6. 约束非模板成员函数

作为类模板成员的非模板函数，可以用相应的方式进行约束，模板类只能为满足某些需求的类型定义成员函数。下面的例子中，operator== 是受约束的:

```
template <typename T>
struct wrapper
{
  T value;

  bool operator==(std::string_view str)
  requires std::is_convertible_v<T, std::string_view>
  {
    return value == str;
  }
};
```

wrapper 类保存一个 T 类型的值，并仅为可转换为 std::string_view 的类型定义 operator== 成员:

```
wrapper<int> a{ 42 };
wrapper<char const*> b{ "42" };
```

```

if(a == 42) {} // error
if(b == "42") {} // OK
```

这里有 wrapper 类的两个实例化，一个用于 int，一个用于 char const*。尝试将 a 对象与字面值 42 进行比较会产生编译器错误，因为没为这种类型定义 operator==。但将 b 对象与字符串字面值”42” 进行比较是可能的，因为相等操作符是为可以隐式转换为 std::string_view 的类型定义的，char const* 就是这样一种类型。

约束非模板成员很有用，相较于强制成员为模板并使用 SFINAE，这是一种更清晰的解决方案。为了更好地进行理解，看看如下 wrapper 类的实现:

```
template <typename T>
struct wrapper
{
  T value;

  wrapper(T const & v) :value(v) {}
};
```

这个类模板可以进行如下的实例化:

```
wrapper<int> a = 42; //OK

wrapper<std::unique_ptr<int>> p =
  std::make_unique<int>(42); //error
```

第一行编译成功，但第二行会产生编译器错误。不同的编译器会发出不同的消息，但错误的核心是调用隐式删除的 std::unique_ptr 复制构造函数。

我们要做的是限制包装器的复制构造来自 T 类型的对象，以便它只适用于可复制构造的 T 类型。C++20 之前可用的方法是将复制构造函数转换为模板，并使用 SFINAE:

```
template <typename T>
struct wrapper
{
  T value;
  template <typename U,
    typename = std::enable_if_t<
      std::is_copy_constructible_v<U> &&
      std::is_convertible_v<U, T>>>
  wrapper(U const& v) :value(v) {}
};
```

当试图从 std::unique_ptr<int> 值初始化 wrapper<std::unique_ptr<int>> 时，也会收获一个错误，但错误是不同的。例如，下面是 Clang 生成的错误消息:

```
prog.cc:19:35: error: no viable conversion from 'typename __
unique_if<int>::__unique_single' (aka 'unique_ptr<int>') to
  'wrapper<std::unique_ptr<int>>'
```

```
wrapper<std::unique_ptr<int>> p = std::make_
unique<int>(42); // error
                                ^ ~~~~~~~~~~~~~~~~~~~~~~~~~
prog.cc:6:8: note: candidate constructor (the implicit copy
constructor) not viable: no known conversion from 'typename
__unique_if<int>::__unique_single' (aka 'unique_ptr<int>') to
'const wrapper<std::unique_ptr<int>> &' for 1st argument
struct wrapper
       ^
prog.cc:6:8: note: candidate constructor (the implicit move
constructor) not viable: no known conversion from 'typename
__unique_if<int>::__unique_single' (aka 'unique_ptr<int>') to
'wrapper<std::unique_ptr<int>> &&' for 1st argument
struct wrapper
       ^
prog.cc:13:9: note: candidate template ignored: requirement
'std::is_copy_constructible_v<std::unique_ptr<int,
std::default_delete<int>>>' was not satisfied [with U =
std::unique_ptr<int>]
      wrapper(U const& v) :value(v) {}
      ^
```

帮助理解问题原因的最重要的信息是最后一条，说 U 替换为 std::unique_ptr<int> 的要求不满足布尔条件。C++20 中，可以更好地实现对 T 模板参数的相同限制。这一次，我们可以使用约束，复制构造函数不再需要是模板。C++20 中的实现如下所示:

```
template <typename T>
struct wrapper
{
  T value;
  wrapper(T const& v)
    requires std::is_copy_constructible_v<T>
    :value(v)
  {}
};
```

不仅代码更少，不需要复杂的 SFINAE 机制，而且更简单，更容易理解，还可能生成更好的错误消息。Clang 的情况下，前面的错误信息会替换为以下内容:

```
prog.cc:9:5: note: candidate constructor not viable:
constraints not satisfied
```

```
    wrapper(T const& v)
    ^

prog.cc:10:18: note: because 'std::is_copy_constructible_
v<std::unique_ptr<int> >' evaluated to false
      requires std::is_copy_constructible_v<T>
```

值得一提的是，不仅属于类成员的非模板函数可以受到约束，独立函数也可以。非模板函数的用例很少见，可以用其他简单的解决方案来实现，比如 constexpr if:

```
void handle(int v)
{ /* do something */ }

void handle(long v)
  requires (sizeof(long) > sizeof(int))
{ /* do something else */ }
```

这个代码片段中，我们有两个 handle 函数的重载。第一个重载采用 int，第二个重载采用 long。这些重载函数体并不重要，但当 long 的大小与 int 的大小不同时，就应该做不同的事情。标准规定 int 的大小至少是 16 位，尽管在大多数平台上它是 32 位，long 长度至少为 32 位。但有一些平台，比如 LP64，其中 int 是 32 位，long 是 64 位。这些平台上，两个重载都应该是可用的。在所有其他平台上，两种类型具有相同的大小，只有第一个重载可用。可以用前面所示的形式定义，在 C++17 中使用 constexpr 也可以实现同样的功能:

```
void handle(long v)
{
  if constexpr (sizeof(long) > sizeof(int))
  {
    /* do something else */
  }
  else
  {
    /* do something */
  }
}
```

下一节中，我们将学习如何使用约束来定义类模板的模板参数的限制。

6.7. 约束类模板

类模板和类模板特化也可以像函数模板一样受到约束。首先，考虑 wrapper 类模板，但这一次要求它只适用于整型的模板参数。这可以在 C++20 中简单地进行指定:

```
template <std::integral T>
struct wrapper
{
```

```
  T value;
};

wrapper<int> a{ 42 }; // OK
wrapper<double> b{ 42.0 }; // error
```

为 int 实例化模板是可以的，但对 double 无效，因为其不是整型。

同样用 requires 子句和类模板特化指定的需求也可以约束。为了展示这一点，考虑这样一个场景: 希望特化包装器类模板，但只针对大小为 4 字节的类型。可以这样实现:

```
template <std::integral T>
struct wrapper
{
  T value;
};

template <std::integral T>
requires (sizeof(T) == 4)
struct wrapper<T>
{
  union
  {
    T value;
    struct
    {
      uint8_t byte4;
      uint8_t byte3;
      uint8_t byte2;
      uint8_t byte1;
    };
  };
};
```

如何使用这个类模板，如下所示:

```
wrapper<short> a{ 42 };
std::cout << a.value << '\n';

wrapper<int> b{ 0x11223344 };
std::cout << std::hex << b.value << '\n';
std::cout << std::hex << (int)b.byte1 << '\n';
std::cout << std::hex << (int)b.byte2 << '\n';
std::cout << std::hex << (int)b.byte3 << '\n';
std::cout << std::hex << (int)b.byte4 << '\n';
```

对象 a 是 wrapper<short>;，所以使用主模板。另一方面，对象 b 是 wrapper<int> 的实例。由于 int 的大小为 4 字节 (在大多数平台上)，因此使用了特化，我们可以通过 byte1、byte2、byte3 和 byte4 成员访问包装值的各个类型。

最后，我们将讨论如何约束变量模板和模板别名。

6.8. 约束变量模板和模板别名

C++ 中，除了函数模板和类模板外，还有变量模板和别名模板，这些都需要定义约束。目前讨论的约束模板参数的相同规则也适用于这两个参数。

定义 PI 常量以显示变量模板的工作方式是一个典型的例子，这是一个简单的定义:

```
template <typename T>
constexpr T PI = T(3.14159265358979323851);
```

然而，这只对浮点类型有意义 (也可能是其他类型，比如十进制，在 C++ 中还不存在)，所以这个定义应该限制为浮点类型:

```
template <std::floating_point T>
constexpr T PI = T(3.14159265358979323851);

std::cout << PI<double> << '\n'; // OK
std::cout << PI<int> << '\n'; // error
```

使用 PI<double> 是正确的，但 PI<int> 会产生编译器错误，可以以简单易读的方式进行约束。

最后一类模板——别名模板，也可以受到约束。下面的代码片段中，可以看到这样一个例子:

```
template <std::integral T>
using integral_vector = std::vector<T>;
```

当 T 是整型时，integral_vector 模板是 std::vector<T> 的别名。使用以下替代声明也可以达到同样的效果，不过声明更长了:

```
template <typename T>
requires std::integral<T>
using integral_vector = std::vector<T>;
```

可以像下面这样使用这个 integral_vector 别名模板:

```
integral_vector<int> v1 { 1,2,3 }; // OK
integral_vector<double> v2 {1.0, 2.0, 3.0}; // error
```

定义 v1 对象没问题，因为 int 是整型。不过，定义 v2 会产生编译器错误，因为 double 不是整型。

若注意了本节中的例子，会注意到其没有使用我们之前在本章中使用的类型特征 (以及相关的变量模板)，而是使用了几个概念:std::integral 和 std::float_point。这些定义在 <concepts> 头文件中，可避免使用基于 C++11(或更新的) 类型特征重复定义相同的概念。后面，我们将简要介绍标准概念库的内容。在此之前，先来看看在 C++20 中还可以使用哪些方法来定义约束。

6.9. 更多指定约束的方法

本章中，已经讨论了 requires 子句和 requires 表达式。虽然两者都是用新的 requires 关键字引入的，但它们是不同的东西:

- requires 子句决定函数是否参与重载解析，这取决于编译时布尔表达式的值。

- requires 表达式确定一个或多个表达式的集合是否格式良好，而不会对程序的行为产生任何副作用。requires 表达式是一个布尔表达式，可以与 requires 子句一起使用。

再来看一个例子:

```
template <typename T>
concept addable = requires(T a, T b) { a + b; };
                        // [1] requires expression
template <typename T>
requires addable<T> // [2] requires clause
auto add(T a, T b)
{
  return a + b;
}
```

[1] 以 requires 关键字开头的构造是一个 requires 表达式。验证表达式 a + b 对于任何 T 类型都是格式良好的。另一方面，[2] 的构造是一个 requires 子句。若布尔表达式 addable<T> 的值为 true，则该函数参与重载解析; 否则，不会。

虽然 requires 子句应该使用概念，但也可以使用 requires 表达式。基本上，概念定义中可以放在 = 标记右侧的东西，都可以与 requires 子句一起使用:

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

虽然这是完全合法的代码，但是否是使用约束的好方法却有有争议。我建议避免创建以 requires requires 开头的构造，因为其可读性较差，可能会造成混乱。此外，命名概念可以在其他地方使用，而若需要用于多个函数，则必须复制带有 requires 表达式的 requires 子句。

现在已经了解了如何使用约束和概念以多种方式约束模板实参，让我们看看如何简化函数模板语法，并约束模板实参。

6.10. 使用概念来约束 auto 参数

第 2 章中，讨论了 C++14 中引入的泛型 Lambda，以及 C++20 中引入的 Lambda 模板。对至少一个参数使用 auto 说明符的 Lambda 称为泛型 Lambda。编译器生成的函数对象，将具有模板化的调用操作符。这里有一个例子:

```
auto lsum = [](auto a, auto b) {return a + b; };
```

C++20 标准将此特性推广到所有函数，可以在函数参数列表中使用 auto。这具有将函数转换为模板函数的效果:

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

这是一个函数，接受两个参数并返回和 (或者更准确地说，是对两个值应用 operator+ 的结果)。使用 auto 作为函数参数的函数称为缩写函数模板，是函数模板的简写语法。上述函数的等效模板如下所示:

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

可以像调用模板函数一样调用这个函数，编译器将通过用实际类型替换模板参数的方式，生成适当的实例化。考虑以下的调用方式:

```
add(4, 2); // returns 6
add(4.0, 2); // returns 6.0
```

可以用 cppinsights.io 网站检查编译器生成的代码，以添加基于这两个调用的缩写函数模板。生成以下特化:

```
template<>
int add<int, int>(int a, int b)
{
  return a + b;
}

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

由于缩写函数模板只不过是具有简化语法的常规函数模板，这样的函数可以由用户显式特化:

```
template<>
auto add(char const* a, char const* b)
{
  return std::string(a) + std::string(b);
}
```

这是 char const* 类型的全特化。这种特化能够调用 add(”4”，”2”)，尽管结果是一个 std::string 值。

这类简化的函数模板称为无约束函数模板，模板参数没有限制，但可以用概念为它们的参数提供约束。使用概念的简短函数模板称为约束函数模板。下面，会看到一个 add 函数约束整型的例子:

```
auto add(std::integral auto a, std::integral auto b)
{
  return a + b;
}
```

若再次考虑前面看到的相同调用，第一个将会成功，但第二个将会产生编译器错误，因为没有重载接受 double 和 int 值:

```
add(4, 2); // OK
add(4.2, 0); // error
```

约束 auto 也可以用于可变参数缩写函数模板。示例如下所示:

```
auto add(std::integral auto ... args)
{
  return (args + ...);
}
```

最后，受约束的 auto 也可以用于通用的 Lambda。若希望本节开头所示的泛型 Lambda 仅用于整型，可以按以下方式对其进行约束:

```
auto lsum = [](std::integral auto a, std::integral auto b)
{
  return a + b;
};
```

随着本节的结束，我们已经了解了 C++20 中与概念和约束相关的语言特性。接下来要讨论的是标准库提供的概念集。

6.11. 探索标准概念库

标准库提供了一组基本概念，可用于定义对函数模板、类模板、变量模板和别名模板的模板实参的需求，正如在本章中看到的那样。C++20 中的标准概念分布在多个头文件和命名空间中。我们将在本节中介绍其中一些，可以在https://en.cppreference.com/上找到所有的概念。

主要的概念集可以在 <concepts> 头文件和 std 命名空间中找到。这些概念中的大多数等价于一个或多个现有的类型特征。对于其中一些，它们的实现是定义良好的; 另一些，则是不明确的。可分为四类: 核心语言概念、比较概念、对象概念和可调用概念。这组概念包含以下内容 (但不仅如此):

概念	描述
same_as	类型 T 与另一个类型 U 相同。
derived_from	类型 D 从另一个类型 B 派生。
convertible_to	类型 T 可以隐式转换为另一类型 U。
common_reference_with	类型 T 和 U 具有共同引用类型。
common_with	类型 T 和 U 具有共同类型，这两种类型都可以转换为共同类型。
integral	类型 T 是整型。
signed_integral	类型 T 是有符号整型。
unsigned_integral	类型 T 是无符号整型。
floating_point	类型 T 是浮点类型。
assignable_from	U 类型的表达式可以赋值给 T 类型的左值表达式。
swappable	两个相同类型 T 的值可以交换。
swappable_with	类型 T 的值可以与类型 U 的值相匹配。
destructible	可以安全地销毁类型 T 的值 (析构函数不抛出异常)。
constructible_from	可以用给定的参数类型集构造类型为 T 的对象。
default_initializable	类型 T 的对象可以是默认可构造 (值初始化 T()，从空初始化列表 T{} 直接列表初始化，或默认初始化，如 T t;)。
move_constructible	可以用移动语义构造类型 T 的对象。
copy_constructible	类型 T 的对象可以复制构造和移动构造。
moveable	可以移动和交换类型为 T 的对象。
copyable	可以复制、移动和交换类型为 T 的对象。
regular	类型 T 需要同时满足 semiregular 和 equality_comparable 两个概念。
semiregular	可以复制、移动、交换和默认构造类型为 T 的对象。
equality_comparable	类型 T 的比较运算符 == 反映相等，当两个值相等时为 true。 类似的，=! 表示了不相等。
predicate	可调用类型 T 是布尔谓词。

表 6.1

其中一些概念是使用类型特征定义的，一些是其他概念或概念与类型特征的组合，还有一些具有部分未指定的实现。下面是一些例子:

```
template < class T >
concept integral = std::is_integral_v<T>;

template < class T >
concept signed_integral = std::integral<T> &&
                          std::is_signed_v<T>;

template <class T>
concept regular = std::semiregular<T> &&
                  std::equality_comparable<T>;
```

C++20 还引入了一个新的基于概念的迭代器系统，并在 <iterator> 头文件中定义了一组概念。下表列出了其中一些概念:

概念	描述
indirectly_readable	可以通过应用 *operator 读取类型值。
indirectly_writable	迭代器类型所引用的对象可以写入。
input_iterator	类型是输入迭代器 (支持读取、前缀递增和后缀递增)。
output_iterator	类型是输出迭代器 (支持写入、前缀递增和后缀递增)。
forward_iterator	作为 input_iterator 的类型同时也是前向迭代器 (支持相等比较和多次传递)
bidirectional_iterator	作为 forward_iterator 的类型同时，也是双向迭代器 (支持向后移动)
randon_access_iterator	作为 bidirectional_iterator 类型的同时，也是随机迭代器 (支持在常数时间内下标和前进)。
contiguous_iterator	作为 random_access_iterator 类型同时，也是连续迭代器 (元素存储在连续的内存位置) 的要求。

表 6.2

下面是 C++ 标准中 random_access_iterator 概念的定义:

```
template<typename I>
concept random_access_iterator =
  std::bidirectional_iterator<I> &&
  std::derived_from</*ITER_CONCEPT*/<I>,
            std::random_access_iterator_tag> &&
  std::totally_ordered<I> &&
  std::sized_sentinel_for<I, I> &&
  requires(I i,
       const I j,
       const std::iter_difference_t<I> n)
  {
    { i += n } -> std::same_as<I&>;
    { j + n } -> std::same_as<I>;
    { n + j } -> std::same_as<I>;
    { i -= n } -> std::same_as<I&>;
    { j - n } -> std::same_as<I>;
    { j[n] } -> std::same_as<std::iter_reference_t<I>>;
  };
```

这里使用了几个概念 (其中一些没有在这里列出)，以及一个 requires 表达式来确保某些表达式是格式良好的。

此外，<iterator> 头文件中，有一组旨在简化通用算法约束的概念。下表列出了其中一些概念:

概念	描述
indriectly_movable	值可以从 indirectly_readable 类型移动到 indirectly_readable 类型。
indirectly_copyable	值可以从 indirectly_readable 类型复制到 indirectly_copyable 类型。
mergeable	通过复制元素将排序序列合并为输出序列的算法。
sortable	将序列修改为有序序列的算法。
permutable	对元素重新排序的算法。

表 6.3

C++20 中包含的几个主要特性之一 (以及概念、模块和协程) 是范围。范围库定义了一系列类和函数，用于简化使用范围操作。其中有一组概念，在 <ranges> 头文件和 std::ranges 命名空间中定义。其中的一些概念:

概念	描述
range	类型 R 为范围，需要提供了一个开始迭代器和一个结束哨兵。
sized_range	类型 R 是在常数时间内已知大小的范围。
view	类型 T 是视图，需要提供了恒定时间的复制、移动和赋值操作。
input_range	类型 R 是一个范围，其迭代器类型需要满足 input_iterator 概念。
output_range	类型 R 是一个范围，其迭代器类型需要满足 output_iterator 概念。
forward_range	类型 R 是一个范围，其迭代器类型需要满足 forward_iterator 概念。
bidirectional_range	类型 R 是一个范围，其迭代器类型需要满足 bidirectional_iterator 概念。
random_access_range	类型 R 是一个范围，其迭代器类型需要满足 random_access_iterator 概念。
contiguous_range	类型 R 是一个范围，其迭代器类型需要满足 contiguous_iterator 概念。

表 6.4

以下是其中一些概念的定义:

```
template< class T >
concept range = requires( T& t ) {
  ranges::begin(t);
  ranges::end (t);
};

template< class T >
concept sized_range = ranges::range<T> &&
  requires(T& t) {
    ranges::size(t);
  };

template< class T >
concept input_range = ranges::range<T> &&
  std::input_iterator<ranges::iterator_t<T>>;
```

如前所述，这里列出的概念要比这里列出的多得多，未来可能还会增加。本节不打算作为标准概念的完整参考，而是作为它们的介绍。感兴趣的读者可以在https://en.cppreference.com/官方 C++ 参考文档了解更多关于这些概念的信息。至于范围库，我们将在第 8 章中了解到更多，并探索标准库提供了什么。

6.12. 总结

C++20 标准为语言和标准库引入了一些新的主要特性。其中之一是概念，这是本章的主题。概念是一个命名约束，可用于定义函数模板、类模板、变量模板和别名模板的模板参数需求。

本章中，详细探讨了如何使用约束和概念，以及其是如何工作的。我们已经了解了 requires 子句 (确定模板是否参与重载解析) 和 requires 表达式 (指定表达式格式良好的要求)。已经看到了用于指定约束的各种语法，还了解了为函数模板提供简化语法的缩写函数模板。本章的最后，探讨了标准库中可用的基本概念。

下一章中，我们将把注意力转移到应用到使用目前为止积累的知识，来实现各种基于模板的模式和习惯用法上。

6.13. 习题

1. 什么是约束，什么是概念?
2. 什么是 requires 子句和 requires 表达式?
3. requires 表达式的类别是什么?
4. 约束如何影响重载解析中模板的顺序?
5. 缩写函数模板是什么?

6.14. 扩展阅读

- C++20 Concepts - A Quick Introduction, Bartlomiej Filipek, https://www.cppstories.com/2021/concepts-intro/
- How C++20 Concepts can simplify your code, Andreas Fertig, https://andreasfertig.blog/2020/07/how-cpp20-concepts-cansimplify-your-code/
- What are C++20 concepts and constraints? How to use them?, Sorush Khajepor, https://iamsorush.com/posts/concepts-cpp/
- Requires-clause, Andrzej Krzemieński, https://akrzemi1.wordpress.com/2020/03/26/requires-clause/
- Ordering by constraints, Andrzej Krzemieński, https://akrzemi1.wordpress.com/2020/05/07/ordering-by-constraints/

第三部分：模板的应用

本部分中，将实践到目前为止积累的模板知识。将了解静态多态性和模式，例如奇异递归模板模式和混入 **(Mixins)**，以及类型擦除、标记分派、表达式模板和类型列表。还将了解标准容器、迭代器和算法的设计，并学习实现自己的容器、迭代器和算法。我们将探索 **C++20** 范围库及其范围和约束算法，将学习如何编写自己的范围适配器。

本部分由以下章节组成:

- 第 *7* 章，模式和习语
- 第 *8* 章，范围和算法
- 第 *9* 章，范围库

第 7 章 模式和习语

本书前面的部分旨在了解关于模板的一切，从基础到最高级的特性，包括 C++20 的最新概念和约束。现在，是时候将这些知识用于工作，并可以开始学习各种元编程技术了。

本章中，我们将讨论以下主题:

- 动态与静态多态性
- 奇异递归模板模式 (CRTP)
- 混入 (Mixins)
- 类型擦除
- 标记分派
- 表达式模板
- 类型列表

本章结束时，将很好地理解各种多路编程技术，这些技术将帮助读者们解决各种实际问题。

7.1. 动态与静态多态性

学习面向对象编程时，需要了解其基本原则，即抽象、封装、继承和多态性。C++ 是一种支持面向对象编程的多范式编程语言。不过关于面向对象编程原理的更广泛的讨论超出了本章和本书的范畴，但可以讨论与多态相关的一些方面。

那么，什么是多态性? 这个词来源于希腊语，意思是“多种形式”。在编程中，它是一种将不同类型的对象视为相同类型的能力。C++ 标准实际定义了一个多态类，如下所示 (见 C++20 标准，第 11.7.2 段，虚函数):

> 声明或继承虚函数的类称为多态类。

还基于此定义定义了多态对象，如下所示 (参见 C++20 标准，第 6.7.2 段，对象模型):

> 一些对象是多态的 *(11.7.2);* 该实现生成与每个此类对象相关的信息，从而可以在程序执行期间确定该对象的类型。

然而，这实际上指的是所谓的动态多态 (或后期绑定)，但还有另一种形式的多态，称为静态多态 (或早期绑定)。动态多态性在运行时借助接口和虚函数产生，而静态多态性则在编译时借助重载函数和模板产生。这在 Bjarne Stroustrup 的 C++ 语言术语表中有所描述 (参见https://www.stroustrup.com/glossary.html):

> 多态性——为不同类型的实体提供单一接口。虚函数通过基类提供的接口提供动态 *(运行时)* 多态性。重载函数和模板提供了静态 *(编译时)* 多态性。

来看一个动态多态性的例子。以下是代表游戏中不同单位的类的结构，这些单元可能会攻击其他单元，因此有一个基类，带有一个称为 attack 的纯虚函数，还有几个派生类实现了特定的单元，这些单元覆盖了这个虚函数，做不同的事情 (为了简单起见，这里只将消息输出到控制台)。如下所示:

```
struct game_unit
{
  virtual void attack() = 0;
};

struct knight : game_unit
{
  void attack() override
  { std::cout << "draw sword\n"; }
};

struct mage : game_unit
{
  void attack() override
  { std::cout << "spell magic curse\n"; }
};
```

基于这种类的层次结构 (根据标准称为多态类)，可以编写如下所示的函数 fight。这接受一个指向基本 game_unit 类型对象的指针序列，并调用 attack 成员函数。下面是其实现:

```
void fight(std::vector<game_unit*> const & units)
{
  for (auto unit : units)
  {
    unit->attack();
  }
}
```

这个函数不需要知道每个对象的实际类型，由于动态多态性，可以像处理相同 (基本) 类型一样处理。下面是一个用例:

```
knight k;
mage m;
fight({&k, &m});
```

但现在，假设可以结合一个法师和一个骑士，并创造一个新的单位，一个拥有特殊能力的骑士法师。能够编写如下代码:

```
knight_mage km = k + m;
km.attack();
```

这不是现成的，但是语言支持重载操作符，可以对用户定义的类型这样做。为了使上一行成为可能，需要以下内容:

```
struct knight_mage : game_unit
{
  void attack() override
  { std::cout << "draw magic sword\n"; }
};

```

```
knight_mage operator+(knight const& k, mage const& m)
{
  return knight_mage{};
}
```

这些只是一些简单的代码片段，没有任何复杂的代码。但是把一个 knight 和一个 mage 加在一起来创建一个 knight_mage 的能力就像把两个整数加在一起一样，或者一个 double 和一个 int，或者两个 std::string 对象。这是因为加法运算符有很多重载 (内置类型和用户定义类型都有)，编译器会根据操作数选择适当的重载，所以这个算子可以有很多种形式。对于所有可以重载的操作符都是如此，加法运算符只是一个典型的例子。这是多态性的编译时版本，称为静态多态性。

运算符并不是唯一可以重载的函数，函数都可以重载。来再看一个例子:

```
struct attack { int value; };
struct defense { int value; };

void increment(attack& a) { a.value++; }
void increment(defense& d) { d.value++; }
```

这段代码中，increment 函数对 attack 和 defense 类型都进行了重载，所以可以编写如下代码:

```
attack a{ 42 };
defense d{ 50 };

increment(a);
increment(d);
```

可以用一个函数模板替换这两个增量重载。更改是最小的，如下所示:

```
template <typename T>
void increment(T& t) { t.value++; }
```

前面的代码继续工作，但有一个显著的区别: 前面的示例中，有两个重载，一个用于 attack，一个用于 defense，因此可以使用这些类型的对象调用函数，但不能使用其他类型的对象。后者有一个模板，可为类型 T 定义了一组重载函数，该类型 T 有一个名为 value 的数据成员，其类型支持后增量操作符。我们可以为这样的函数模板定义约束，这是我们在本书前两章中看到的内容，但关键是重载函数和模板是 C++ 中实现静态多态的一种机制。

动态多态性会导致性能损失，为了知道要调用什么函数，编译器需要构建一个指向虚函数的指针表 (若存在虚继承，还需要构建一个指向虚基类的指针表)，所以当以多态方式调用虚函数时，存在某种程度的间接调用。此外，因为编译器不能优化它们，所以虚拟函数的细节不会提供给编译器。

当这些可以作为性能问题进行验证时，可以提出这样的问题: 能在编译时获得动态多态性的好处吗? 答案是肯定的，有一种方法可以实现: 奇异递归模板模式。

7.2. 奇异递归模板模式 (CRTP)

这个模式有一个相当奇怪的名字: 奇异递归模板模式，简称 CRTP。之所以称为" 奇异"，是因为奇怪和不直观。1995 年，James Coplien 在 c++ Report 的一个专栏中首次描述了这个模式 (并创造了它的名字)。这种模式如下:

- 定义 (静态) 接口的基类模板。
- 派生类本身就是基类模板的模板参数。
- 基类的成员函数调用其类型模板参数 (即派生类) 的成员函数。

来看看模式实现在实际中是什么样子的。我们将把前面带有游戏单位的例子转换成使用 CRTP 的版本，模式实现如下所示:

```
template <typename T>
struct game_unit
{
  void attack()
  {
    static_cast<T*>(this)->do_attack();
  }
};

struct knight : game_unit<knight>
{
  void do_attack()
  { std::cout << "draw sword\n"; }
};

struct mage : game_unit<mage>
{
  void do_attack()
  { std::cout << "spell magic curse\n"; }
};
```

game_unit 类现在是一个模板类，但包含相同的成员函数 attack。在内部，它将 this 指针上转换为 T*，然后调用一个名为 do_attack 的成员函数。knight 和 mage 类派生自 game_unit 类，并将自己作为类型模板参数 T 进行传递。它们都提供了一个名为 do_attack 的成员函数。

注意，基类模板中的成员函数和派生类中调用的成员函数具有不同的名称。否则，若具有相同的名称，因为它们不再是虚函数了，派生类成员函数将隐藏基类成员，、。

fight 函数 (获取游戏单位集合并调用攻击函数) 也需要改变。需要作为函数模板实现，实现如下所示:

```
template <typename T>
void fight(std::vector<game_unit<T>*> const & units)
{
  for (auto unit : units)
  {
    unit->attack();
  }
}
```

使用此函数与以前略有不同:

```
knight k;
mage m;
fight<knight>({ &k });
```

```
fight<mage>({ &m });
```

我们已经将运行时多态性移到了编译时，所以 fight 函数不能多态地处理 knight 和 mage 的对象。相反，我们得到了两个不同的重载，一个可以处理 knight 对象，一个可以处理 mage 对象，这就是静态多态。

尽管这个模式看起来并不复杂，但此需要问问自己: 这个模式实际上有什么用处? 可以使用 CRT 解决的问题，包括:

- 限制类型可以实例化的次数
- 增加公共功能并避免代码重复
- 实现复合设计模式

下面的小节中，我们将逐一研究这些问题，并了解如何使用 CRTP 解决。

7.2.1 限制实例化对象的次数

假设在创造骑士和法师的游戏中，需要一些道具在有限的实例中可用。例如，有一种特殊的剑叫做 Excalibur，它应该只有一个实例。另一方面，有一本魔法咒语书，但在游戏中一次不能超过三个实例。怎么解决这个问题? 显然，剑的问题可以用单例模式解决。但当我们需要把这个数限制到一个更高的值，但仍然是有限的时候，该怎么办呢? 单例模式不会有什么帮助 (除非将其转换为“多例”)，但 CRTP 可以。

首先，从基类模板开始。这个类模板所做的事就是记录实例化的次数。计数器是静态数据成员，在构造函数中自增，在析构函数中自减。当该计数超过定义的限制时，将触发异常:

```
template <typename T, size_t N>
struct limited_instances
{
  static std::atomic<size_t> count;
  limited_instances()
  {
    if (count >= N)
      throw std::logic_error{ "Too many instances" };
    ++count;
  }
  ~limited_instances() { --count; }
};

template <typename T, size_t N>
std::atomic<size_t> limited_instances<T, N>::count = 0;
```

模板的第二部分包括定义派生类。针对上述问题，具体实现如下:

```
struct excalibur : limited_instances<excalibur, 1>
{};

struct book_of_magic : limited_instances<book_of_magic, 3>
{};
```

我们可以实例化 excalibur 一次。当第二次尝试做同样的事情时 (当第一个实例仍然存在时)，将抛出异常:

```
excalibur e1;
try
{
  excalibur e2;
}
catch (std::exception& e)
{
  std::cout << e.what() << '\n';
}
```

类似地，可以实例化 book_of_magic 三次，第四次尝试这样做时将抛出异常:

```
book_of_magic b1;
book_of_magic b2;
book_of_magic b3;
try
{
  book_of_magic b4;
}
catch (std::exception& e)
{
  std::cout << e.what() << '\n';
}
```

接下来，来看一个更常见的场景，向类型添加通用功能。

7.2.2 增加功能

奇异递归模板模式可以解决的另一种情况是，通过基类中仅依赖于派生类成员的泛型函数为派生类提供公共功能。我们通过一个例子来理解这个用例。

假设一些游戏单位拥有 step_forth 和 step_back 等成员函数，它们将向前或向后移动一个位置。这些类看起来如下所示:

```
struct knight
{
  void step_forth();
  void step_back();
};

struct mage
{
  void step_forth();
  void step_back();
};
```

然而，这可能是一种要求，即所有可以来回移动一步的东西都能够前进或后退任意数量的步骤。但这个功能可以基于 step_forward 和 step_back 函数来实现，这将有助于避免在每个游戏单元类中

出现重复的代码，所以这个问题的 CRTP 实现如下所示:

```
template <typename T>
struct movable_unit
{
  void advance(size_t steps)
  {
    while (steps--)
      static_cast<T*>(this)->step_forth();
  }

  void retreat(size_t steps)
  {
    while (steps--)
      static_cast<T*>(this)->step_back();
  }
};

struct knight : movable_unit<knight>
{
  void step_forth()
  { std::cout << "knight moves forward\n"; }

  void step_back()
  { std::cout << "knight moves back\n"; }
};

struct mage : movable_unit<mage>
{
  void step_forth()
  { std::cout << "mage moves forward\n"; }

  void step_back()
  { std::cout << "mage moves back\n"; }
};
```

可以通过调用基类 advance 和 retreat 成员函数来推进和后退单位，如下所示:

```
knight k;
k.advance(3);
k.retreat(2);

mage m;
m.advance(5);
m.retreat(3);
```

可以认为，使用非成员函数模板也可以实现相同的结果。为了便于讨论，这种解决方案的实现如下所示:

```
struct knight
{
```

```
  void step_forth()
  { std::cout << "knight moves forward\n"; }

  void step_back()
  { std::cout << "knight moves back\n"; }
};

struct mage
{
  void step_forth()
  { std::cout << "mage moves forward\n"; }

  void step_back()
  { std::cout << "mage moves back\n"; }
};

template <typename T>
void advance(T& t, size_t steps)
{
  while (steps--) t.step_forth();
}

template <typename T>
void retreat(T& t, size_t steps)
{
  while (steps--) t.step_back();
}
```

使用端的代码需要很小的修改:

```
knight k;
advance(k, 3);
retreat(k, 2);

mage m;
advance(m, 5);
retreat(m, 3);
```

这两者之间的选择可能取决于问题的性质和偏好。而 CRTP 有一个优点，那就是很好地描述了派生类的接口 (比如例子中的 knight 和 mage)。对于非成员函数，并不一定知道这个功能，可能来自需要包含的头文件。使用 CRTP 时，类接口对于使用者来说是可见的。

针对在这里讨论的最后一个场景，来看看 CRTP 如何帮助实现复合设计模式。

7.2.3 实现复合设计模式

著名的《设计模式: 可重用面向对象软件的元素》中，四人组 (Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides) 描述了一种称为复合的结构模式，其能够将对象组合成更大的结构，并统一对待单个对象和组合。当希望表示对象的部分-整体层次结构，并且希望忽略单个对象和单个对象的组合之间的差异时，可以使用此模式。

为了将这种模式付诸实践，再次考虑游戏场景。英雄有特殊的能力，可以做不同的行动，其中之一是与另一个英雄结盟。这可以很容易地进行如下建模:

```
struct hero
{
  hero(std::string_view n) : name(n) {}
  void ally_with(hero& u)
  {
    connections.insert(&u);
    u.connections.insert(this);
  }
private:
  std::string name;
  std::set<hero*> connections;
  friend std::ostream& operator<<(std::ostream& os,
                  hero const& obj);
};

std::ostream& operator<<(std::ostream& os,
hero const& obj)
{
  for (hero* u : obj.connections)
    os << obj.name << " --> [" << u->name << "]" << '\n';

  return os;
}
```

这些英雄由 hero 类来表示，这个英雄类包含一个名字，一个到其他 hero 对象的连接列表，以及一个成员函数 ally_with，它定义了两个英雄之间的联盟。可以这样使用:

```
hero k1("Arthur");
hero k2("Sir Lancelot");
hero k3("Sir Gawain");

k1.ally_with(k2);
k2.ally_with(k3);

std::cout << k1 << '\n';
std::cout << k2 << '\n';
std::cout << k3 << '\n';
```

运行代码段的输出:

```
Arthur --> [Sir Lancelot]

Sir Lancelot --> [Arthur]
Sir Lancelot --> [Sir Gawain]
```

```
Sir Gawain --> [Sir Lancelot]
```

目前为止，一切都很简单。但要求是英雄们可以聚集在一起组成政党。一个英雄可以与一个团体结盟，一个团体可以与一个英雄结盟，也可以与整个团体结盟。突然间，需要的功能激增:

```
struct hero_party;

struct hero
{
  void ally_with(hero& u);
  void ally_with(hero_party& p);
};

struct hero_party : std::vector<hero>
{
  void ally_with(hero& u);
  void ally_with(hero_party& p);
};
```

这就是复合设计模式可以统一对待英雄和团队，并避免不必要的代码重复的地方。通常有不同的实现方法，但其中一种方法是使用重复出现的模板模式。实现需要定义公共接口的基类。我们的例子中，这将是一个类模板，只有一个名为 ally_with 的成员函数:

```
template <typename T>
struct base_unit
{
  template <typename U>
  void ally_with(U& other);
};
```

我们把 hero 类定义为 base_unit<hero> 的派生类。这一次，hero 类不再实现自身的 ally_with。但它提供了 begin 和 end 方法，用于模拟容器的行为:

```
struct hero : base_unit<hero>
{
  hero(std::string_view n) : name(n) {}

  hero* begin() { return this; }
  hero* end() { return this + 1; }

private:
  std::string name;
  std::set<hero*> connections;

  template <typename U>
  friend struct base_unit;

  template <typename U>
  friend std::ostream& operator<<(std::ostream& os,
```

```
                    base_unit<U>& object);
};
```

为一组英雄建模的类叫做 hero_party，源于 std::vector<hero>(用来定义英雄对象的容器) 和 base_unit<hero_party>。这就是为什么 hero 类有 begin 和 end 函数来对英雄对象执行迭代操作，就像对 hero_party 对象所做的那样:

```
struct hero_party : std::vector<hero>,
          base_unit<hero_party>
{};
```

需要实现基类的 ally_with 成员函数，代码如下所示。其所做的是遍历当前对象的所有子对象，并将它们与所提供参数的所有子对象连接起来:

```
template <typename T>
template <typename U>
void base_unit<T>::ally_with(U& other)
{
  for (hero& from : *static_cast<T*>(this))
  {
    for (hero& to : other)
    {
      from.connections.insert(&to);
      to.connections.insert(&from);
    }
  }
}
```

hero 类将基类 base_unit 声明为友元，以便访问 connections 成员。其还将操作符 << 声明为友元，以便该函数可以访问 connections 和 name 私有成员。有关模板及其朋友的更多信息，请参阅第 4 章的相关章节。输出流操作符的实现如下所示:

```
template <typename T>
std::ostream& operator<<(std::ostream& os,
base_unit<T>& object)
{
  for (hero& obj : *static_cast<T*>(&object))
  {
    for (hero* n : obj.connections)
      os << obj.name << " --> [" << n->name << "]"
        << '\n';
  }
  return os;
}
```

定义了所有这些之后，可以编写如下代码:

```
hero k1("Arthur");
hero k2("Sir Lancelot");

hero_party p1;
```

```
p1.emplace_back("Bors");

hero_party p2;
p2.emplace_back("Cador");
p2.emplace_back("Constantine");

k1.ally_with(k2);
k1.ally_with(p1);

p1.ally_with(k2);
p1.ally_with(p2);

std::cout << k1 << '\n';
std::cout << k2 << '\n';
std::cout << p1 << '\n';
std::cout << p2 << '\n';
```

这里，可以让一个 hero 与另一个 hero 和一个 hero_party 结盟，也可以让一个 hero_party 与一个 hero 或另一个 hero_party 结盟。这就是我们的目标，并且能够在不复制 hero 和 hero_party 之间的代码的情况下做到。执行上一段代码的输出如下所示:

```
Arthur --> [Sir Lancelot]
Arthur --> [Bors]

Sir Lancelot --> [Arthur]
Sir Lancelot --> [Bors]

Bors --> [Arthur]
Bors --> [Sir Lancelot]
Bors --> [Cador]
Bors --> [Constantine]

Cador --> [Bors]
Constantine --> [Bors]
```

了解了 CRTP 如何帮助实现不同的目标之后，再来看看 CRTP 在 C++ 标准库中的使用。

7.2.4 标准库中的 CRTP

标准库包含一个名为 std::enabled_shared_from_this 的辅助类型 (<memory> 头文件中)，允许由 std::shared_ptr 管理的对象以安全的方式生成更多 std::shared_ptr 实例。std::enabled_shared_from_this 类是 CRTP 模式中的基类。前面的描述可能很抽象，所以我们通过例子来理解它。

假设有一个叫做 building 的类，正在以以下方式创建 std::shared_ptr 对象:

```
struct building {};

building* b = new building();
std::shared_ptr<building> p1{ b }; // [1]
std::shared_ptr<building> p2{ b }; // [2] bad
```

我们有一个原始指针 [1]，实例化了一个 std::shared_ptr 对象来管理其生命周期。但在 [2] 处，可为同一个指针实例化了第二个 std::shared_ptr 对象。当这两个智能指针对彼此一无所知，因此一旦超出作用域，它们都会删除堆上分配的构建对象。删除已经删除的对象是未定义的行为，可能会导致程序崩溃。

std::enable_shared_from_this 类以安全的方式从现有的 shared_ptr 对象创建更多的 shared_ptr 对象。首先，需要实现 CRTP 模式:

```
struct building : std::enable_shared_from_this<building>
{
};
```

有了这个新的实现，可以调用成员函数 shared_from_this 来从一个对象创建更多的 std::shared_ptr 实例，这些实例都引用了该对象的同一个实例:

```
building* b = new building();
std::shared_ptr<building> p1{ b }; // [1]
std::shared_ptr<building> p2{
  b->shared_from_this()}; // [2] OK
```

std::enable_shared_from_this 的接口实现如下:

```
template <typename T>
class enable_shared_from_this
{
  public:
  std::shared_ptr<T> shared_from_this();
  std::shared_ptr<T const> shared_from_this() const;
  std::weak_ptr<T> weak_from_this() noexcept;
  std::weak_ptr<T const> weak_from_this() const noexcept;
  enable_shared_from_this<T>& operator=(
    const enable_shared_from_this<T> &obj ) noexcept;
};
```

前面的例子展示了 enable_shared_from_this 是如何工作的，但并不有助于理解什么时候使用它合适。因此，让我们修改示例，展示一个实际的示例。

试想现有的建筑可以升级，这是一个需要一些时间和几个步骤的过程。这个任务，以及游戏中的其他任务，都是由一个指定的实体执行的，我们称之为 executor。最简单的形式中，这个 executor 类有一个名为 execute 的公共成员函数，该函数接受一个函数对象，并在不同的线程上执行它。下面是一种简单的实现:

```
struct executor
{
  void execute(std::function<void(void)> const& task)
```

```
  {
    threads.push_back(std::thread([task]() {
      using namespace std::chrono_literals;
      std::this_thread::sleep_for(250ms);
      task();
    }));
  }

  ~executor()
  {
    for (auto& t : threads)
    t.join();
  }
private:
  std::vector<std::thread> threads;
};
```

building 类有一个指向 executor 的指针，该指针从调用端传递过来。其还有一个名为 upgrade 的成员函数，用于启动执行过程。然而，实际的升级发生在一个不同的、私有的、名为 do_upgrade 的函数中，这是从传递给 executor 的 execute 成员函数的 Lambda 表达式中调用的。所有这些都显示在下面的代码中:

```
struct building
{
  building() { std::cout << "building created\n"; }
  ~building() { std::cout << "building destroyed\n"; }

  void upgrade()
  {
    if (exec)
    {
      exec->execute([self = this]() {
        self->do_upgrade();
      });
    }
  }
  void set_executor(executor* e) { exec = e; }
private:
  void do_upgrade()
  {
    std::cout << "upgrading\n";
    operational = false;

    using namespace std::chrono_literals;
    std::this_thread::sleep_for(1000ms);

    operational = true;
    std::cout << "building is functional\n";
  }

```

```
  bool operational = false;
  executor* exec = nullptr;
};
```

调用端代码相对简单: 创建一个 executor，创建一个 shared_ptr 管理的 building，设置 executor 引用，并进行升级:

```
int main()
{
  executor e;
  std::shared_ptr<building> b =
    std::make_shared<building>();
  b->set_executor(&e);
  b->upgrade();

  std::cout << "main finished\n";
}
```

若运行这个程序，会得到以下输出:

```
building created
main finished
building destroyed
upgrading
building is functional
```

在升级过程开始之前，建筑就摧毁了。这会导致了未定义的行为，尽管这个程序没有崩溃，但实际的程序肯定会崩溃。

这种行为的罪魁祸首是升级代码中的这一行:

```
exec->execute([self = this]() {
  self->do_upgrade();
});
```

正在创建一个 Lambda 表达式来捕获 this 指针，该指针稍后在所指向的对象销毁后使用。为了避免这种情况，需要创建并捕获一个 shared_ptr 对象，安全的方法是借助 std::enable_shared_from_this 类。有两个改变，第一个是从 std::enable_shared_from_this 类中派生 building 类:

```
struct building : std::enable_shared_from_this<building>
{
  /* ··· */
};
```

第二个变化是要在 Lambda 捕获中调用 shared_from_this:

```
exec->execute([self = shared_from_this()]() {
  self->do_upgrade();
});
```

这对代码的两个小更改，但效果非常显著。Lambda 表达式在单独的线程上执行之前，building 对象不再销毁 (因为现在有一个的共享指针，引用与主函数中创建的共享指针相同的对象)。结果，得到了我们期望的输出 (没有对调用端代码进行更改):

```
building created
main finished
upgrading
building is functional
building destroyed
```

你可以争辩说，主函数完成后，不应该关心发生了什么。注意，这只是一个演示程序。在实践中，这会发生在其他一些函数中，并且在该函数返回后，程序会继续运行很长时间。

至此，我们结束了关于奇怪的重复模板模式的讨论。接下来，我们将研究混入 (Mixins) 技术，它经常与 CRTP 模式混合使用。

7.3. 混入 (Mixins)

Mixins 是一种小型类，用于向其他类添加功能。若读过关于 Mixins 的文章，会发现在 C++ 中使用奇异递归模板模式来实现 Mxins，但这是一个不正确的说法。CRTP 有助于实现与 Mixins 类似的目标，但它们是不同的技术。Mixins 的关键在于，向类添加功能，而不是作为类的基类。而作为基类，是 CRTP 模式的关键。相反，Mixins 应该从它们添加功能的类继承，属于逆 CRTP。

还记得前面关于骑士和法师的例子吗? 可以使用 step_forth 和 step_back 成员函数来回移动。骑士和法师的职业是从 movable_unit 类模板派生出来的，该模板添加了 advance 和 retreat 的功能，这使得单位可以向前或向后移动几步。同样的例子可以以相反的顺序使用 Mixins 来实现。以下是具体实现:

```
struct knight
{
  void step_forth()
  {
    std::cout << "knight moves forward\n";
  }
  void step_back()
  {
    std::cout << "knight moves back\n";
  }
};
struct mage
{
  void step_forth()
  {
    std::cout << "mage moves forward\n";
  }

```

```
  void step_back()
  {
    std::cout << "mage moves back\n";
  }
};

template <typename T>
struct movable_unit : T
{
  void advance(size_t steps)
  {
    while (steps--)
      T::step_forth();
  }

  void retreat(size_t steps)
  {
    while (steps--)
      T::step_back();
  }
};
```

knight 和 mage 现在是没有基类的职业，都提供了 step_forth 和 step_back 成员函数，就像之前实现的 CRTP 模式。现在，movable_unit 类模板是从这些类中派生出来的，并定义了 advance 和 retreat 函数，并在循环中调用了 step_forth 和 step_back。可以这样使用:

```
movable_unit<knight> k;
k.advance(3);
k.retreat(2);

movable_unit<mage> m;
m.advance(5);
m.retreat(3);
```

这与使用 CRTP 模式非常相似，除了创建的实例是 movable_unit<knight> 和 movable_unit<mage>，而不是 knight 和 mage。两种模式的比较如下图所示 (左边是 CRTP，右边是 Mixins):

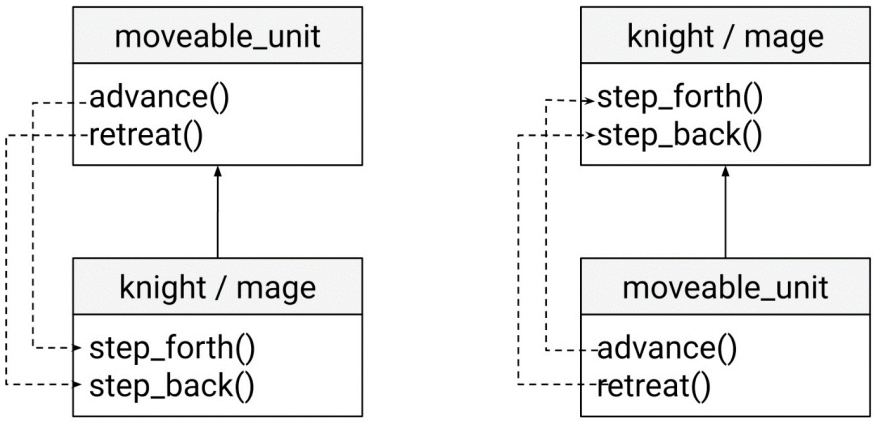


图 7.1:CRTP 和 Mixins 模式的比较

可以结合使用 Mixins 实现的静态多态，与使用接口和虚函数实现的动态多态。这里将通过一个关于战斗的游戏单位来说明这一点。我们之前在讨论 CRTP 时举过一个例子，其中 knight 和 mage 职业都有一个叫做 attack 的成员函数。

假设想要定义多种攻击风格。例如，每个游戏单位可以使用侵略性或中等攻击风格。所以这意味着四种组合: 侵略性和中等骑士，侵略性和中等法师。另一方面，骑士和法师都可能是喜欢独自战斗的孤独战士，或者是总是与其他单位一起战斗的团队成员。

我们可以有单独的好斗骑士和温和骑士，也可以有团队合作的好斗骑士和温和骑士，这同样适用于法师。组合的数量增长了很多，Mixins 是一个很好的方式来提供这种额外的功能，而不扩大 knight 和 mage 的职业。最后，我们希望能够在运行时多态地处理所有这些。现在，来看看怎么做。

首先，可以定义好斗和温和的战斗风格。简单实现如下所示:

```
struct aggressive_style
{
  void fight()
  {
    std::cout << "attack! attack attack!\n";
  }
};

struct moderate_style
{
  void fight()
  {
    std::cout << "attack then defend\n";
  }
};
```

接下来，将 Mixins 定义为能够独自或在团队中战斗的需求。这些类是模板，由模板参数派生:

```
template <typename T>
struct lone_warrior : T
{
  void fight()
  {
    std::cout << "fighting alone.";
    T::fight();
  }
};
template <typename T>
struct team_warrior : T
{
  void fight()
  {
    std::cout << "fighting with a team.";
    T::fight();
  }
};
```

最后，需要定义 knight 和 mage 职业，这些本身就是战斗风格的 Mixins。为了能够在运行时多态地进行处理，我们从 game_unit 类中派生出它们，这个类包含一个叫做 attack 的纯虚方法:

```
struct game_unit
{
  virtual void attack() = 0;
  virtual ~game_unit() = default;
};

template <typename T>
struct knight : T, game_unit
{
  void attack()
  {
    std::cout << "draw sword.";
    T::fight();
  }
};

template <typename T>
struct mage : T, game_unit
{
  void attack()
  {
    std::cout << "spell magic curse.";
    T::fight();
  }
};
```

knight 和 mage 的 attack 成员函数使用了 T::fight 方法，aggresive_style 和 moderate_style 类，以及 lone_warrior 和 team_warrior 混合类都提供了这样的成员函数。所以，可以进行以下组合:

```
std::vector<std::unique_ptr<game_unit>> units;

units.emplace_back(new knight<aggressive_style>());
units.emplace_back(new knight<moderate_style>());
units.emplace_back(new mage<aggressive_style>());
units.emplace_back(new mage<moderate_style>());
units.emplace_back(
  new knight<lone_warrior<aggressive_style>>());
units.emplace_back(
  new knight<lone_warrior<moderate_style>>());
units.emplace_back(
  new knight<team_warrior<aggressive_style>>());
units.emplace_back(
  new knight<team_warrior<moderate_style>>());
units.emplace_back(
  new mage<lone_warrior<aggressive_style>>());
units.emplace_back(
```

```
  new mage<lone_warrior<moderate_style>>());
units.emplace_back(
  new mage<team_warrior<aggressive_style>>());
units.emplace_back(
  new mage<team_warrior<moderate_style>>());

for (auto& u : units)
  u->attack();
```

这里总共定义了 12 种组合，而这一切只需要 6 个类就能实现。这显示了 Mixins 如何添加功能，同时将代码的复杂性保持在较低的水平。若运行代码，会得到以下输出:

```
draw sword.attack! attack attack!
draw sword.attack then defend
spell magic curse.attack! attack attack!
spell magic curse.attack then defend
draw sword.fighting alone.attack! attack attack!
draw sword.fighting alone.attack then defend
draw sword.fighting with a team.attack! attack attack!
draw sword.fighting with a team.attack then defend
spell magic curse.fighting alone.attack! attack attack!
spell magic curse.fighting alone.attack then defend
spell magic curse.fighting with a team.attack! attack attack!
spell magic curse.fighting with a team.attack then defend
```

这里看到了两种模式，CRTP 和 Mixins，都旨在向其他类添加额外的 (公共) 功能。尽管看起来很相似，但它们的结构不同，不应该混淆。利用不相关类型的通用功能的另一种技术称为类型擦除，我们将在下一节进行讨论。

7.4. 类型擦除

类型擦除描述了一种模式，其中删除类型信息，允许以通用方式处理不一定相关的类型。这不是 C++ 语言特有的东西，这个概念也存在于其他语言 (如 Python 和 Java)。有不同形式的类型擦除，如多态性和使用 void 指针 (C 语言的使用方式，需要避免)，但真正的类型擦除是通过模板实现的。讨论这个问题之前，先简要地看一下其他问题。

类型擦除的最基本形式是使用 void 指针。这是典型的 C 语言方式，尽管在 C++ 中可以，但绝不推荐这样做。这样做不是类型安全的，因此很容易出错。但为了便于讨论，来看看这种方法。

再次假设我们有 knight 和 mage 类型，他们都有攻击功能 (行为)，我们希望以一种共同的方式来对待他们，以展示这种行为:

```
struct knight
{
  void attack() { std::cout << "draw sword\n"; }
```

```
};

struct mage
{
  void attack() { std::cout << "spell magic curse\n"; }
};
```

类 C 语言的实现中，可以为每一种类型都创建一个函数，对该类型的对象使用 void*，将其转换为预期的指针类型，然后调用 attack 成员函数:

```
void fight_knight(void* k)
{
  reinterpret_cast<knight*>(k)->attack();
}

void fight_mage(void* m)
{
  reinterpret_cast<mage*>(m)->attack();
}
```

它们有相似的特征，唯一不同的是名字。可以定义一个函数指针，然后将一个对象 (或者更准确地说，一个指向对象的指针) 与一个指向处理它的正确函数的指针关联起来。以下是具体实现:

```
using fight_fn = void(*)(void*);
void fight(
  std::vector<std::pair<void*, fight_fn>> const& units)
{
  for (auto& u : units)
  {
    u.second(u.first);
  }
}
```

最后一个代码片段中没有关于类型的信息，所有这些都已使用 void 指针擦除。fight 函数可以按如下方式调用:

```
knight k;
mage m;

std::vector<std::pair<void*, fight_fn>> units {
  {&k, &fight_knight},
  {&m, &fight_mage},
};

fight(units);
```

C++ 的角度来看，这可能看起来很奇怪。这个示例中，我将 C 与 C++ 类相结合，希望不会在生产环境中看到这样的代码片段。若将一个 mage 传递给 fight_knight 函数或者相反，就会因为一个简单的输入错误而出错。然而，这是可能的，并且是类型擦除的一种形式。C++ 中一个明显的替代解决方案是通过继承使用多态性。

这是在本章开头看到的第一个解决方案：

```
struct game_unit
{
  virtual void attack() = 0;
};

struct knight : game_unit
{
  void attack() override
  { std::cout << "draw sword\n"; }
};

struct mage : game_unit
{
  void attack() override
  { std::cout << "spell magic curse\n"; }
};

void fight(std::vector<game_unit*> const & units)
{
  for (auto unit : units)
    unit->attack();
}
```

fight 功能可以同时处理 knight 和 mage 的对象，不知道传递给它地址的实际对象 (在 vector 内)，因此可以说类型并没有完全删除。knight 和 mage 都是 game_unit，而 fight 函数处理任何 game_unit。对于这个函数要处理的另一种类型，需要派生自 game_unit 纯抽象类。

有时候这是不可能的。在类似的事情中处理不相关的类型 (鸭子类型的过程)，但不能改变这些类型。例如，我们并不拥有源代码。这个问题的解决方案，就是使用模板进行真正的类型擦除。

看到这个模式是什么样子之前，来一步一步理解这个模式的发展，从不相关的 knight 和 mage 开始，以及不能修改他们的实现作为前提。但我们可以编写包装器，为公共功能 (行为) 提供统一的接口:

```
struct knight
{
  void attack() { std::cout << "draw sword\n"; }
};

struct mage
{
  void attack() { std::cout << "spell magic curse\n"; }
};

struct game_unit
{
  virtual void attack() = 0;
  virtual ~game_unit() = default;
};
```

```
struct knight_unit : game_unit
{
  knight_unit(knight& u) : k(u) {}
  void attack() override { k.attack(); }

private:
  knight& k;
};

struct mage_unit : game_unit
{
  mage_unit(mage& u) : m(u) {}
  void attack() override { m.attack(); }

private:
  mage& m;
};

void fight(std::vector<game_unit*> const & units)
{
  for (auto u : units)
  u->attack();
}
```

我们不需要像在 knight 和 mage 中那样调用 game_unit 中的 attack 成员函数，其名称可以随意，纯粹是基于模仿原始行为名称。fight 函数接受一个指向 game_unit 的指针集合，因此能够同时处理 knight 和 mage 对象:

```
knight k;
mage m;

knight_unit ku{ k };
mage_unit mu{ m };

std::vector<game_unit*> v{ &ku, &mu };
fight(v);
```

这个解决方案的问题是有很多重复的代码，knight_unit 和 mage_unit 基本上是一样的。当其他类需要类似地处理时，这种重复会增加更多。代码复制的解决方案是使用模板。我们用下面的职业模板替换 knight_unit 和 mage_unit:

```
template <typename T>
struct game_unit_wrapper : public game_unit
{
  game_unit_wrapper(T& unit) : t(unit) {}

  void attack() override { t.attack(); }
private:
  T& t;
```

```
};
```

这个类在源代码中只有一个副本，但是编译器会使用实例化多个特化。除某些类型限制外，类型信息都会擦除——T 类型必须有一个名为 attack 的成员函数，该函数不接受参数。注意，fight 函数根本没有改变。调用端代码需要稍作修改:

```
knight k;
mage m;

game_unit_wrapper ku{ k };
game_unit_wrapper mu{ m };

std::vector<game_unit*> v{ &ku, &mu };
fight(v);
```

这将我们引向类型擦除模式的形式，将抽象基类和包装器类模板放在另一个类中:

```
struct game
{
  struct game_unit
  {
    virtual void attack() = 0;
    virtual ~game_unit() = default;
  };

  template <typename T>
  struct game_unit_wrapper : public game_unit
  {
    game_unit_wrapper(T& unit) : t(unit) {}

    void attack() override { t.attack(); }
  private:
    T& t;
  };

  template <typename T>
  void addUnit(T& unit)
  {
    units.push_back(
    std::make_unique<game_unit_wrapper<T>>(unit));
  }

  void fight()
  {
    for (auto& u : units)
      u->attack();
  }
private:
  std::vector<std::unique_ptr<game_unit>> units;
};
```

game 类包含 game_unit 对象的集合，并有一个向 game_unit(具有 attack 成员函数) 添加新包装器的方法。还有一个成员函数 fight，用于调用常见的行为。这次的调用端的代码如下所示:

```
knight k;
mage m;

game g;
g.addUnit(k);
g.addUnit(m);

g.fight();
```

类型擦除模式中，抽象基类称为概念，继承的包装器称为模型。若要以既定的方式实现类型擦除模式，可以进行如下实现:

```
struct unit
{
  template <typename T>
  unit(T&& obj) :
    unit_(std::make_shared<unit_model<T>>(
        std::forward<T>(obj)))
  {}

  void attack()
  {
    unit_->attack();
  }

  struct unit_concept
  {
    virtual void attack() = 0;
    virtual ~unit_concept() = default;
  };

  template <typename T>
  struct unit_model : public unit_concept
  {
    unit_model(T& unit) : t(unit) {}

    void attack() override { t.attack(); }
  private:
    T& t;
  };

private:
  std::shared_ptr<unit_concept> unit_;
};

void fight(std::vector<unit>& units)
{
```

```
  for (auto& u : units)
    u.attack();
}
```

这段代码片中，game_unit 重命名为 unit_concept，game_unit_wrapper 重命名为 unit_model，除了名字没有其他变化。它们是名为 unit 的新类的成员，该类存储一个指针，指向实现 unit_concept 的对象; 这可以是 unit_model<knight> 或 unit_model<mage>。unit 类有一个模板构造函数，能够从 knight 和 mage 对象中创建模型对象。

其还有一个公共成员函数 attack(同样，它可以有任何名称)。另一方面，fight 函数处理 unit 对象，并调用它们的 fight 成员函数。调用端代码可能如下所示:

```
knight k;
mage m;

std::vector<unit> v{ unit(k), unit(m) };

fight(v);
```

若想知道这个模式在实际代码中现在哪里使用，标准库中就有两个例子:

- std::function: 这是一个通用的多态函数包装器，能够存储、复制和调用可调用的东西，例如函数、Lambda 表达式、绑定表达式、函数对象、指向成员函数的指针和指向数据成员的指针。下面是一个使用 std::function 的例子:

```
class async_bool
{
  std::function<bool()> check;
public:
  async_bool() = delete;
  async_bool(std::function<bool()> checkIt)
    : check(checkIt)
  { }

  async_bool(bool val)
    : check([val]() {return val; })
  { }

  operator bool() const { return check(); }
};

async_bool b1{ false };
async_bool b2{ true };
async_bool b3{ []() { std::cout << "Y/N? ";
            char c; std::cin >> c;
            return c == 'Y' || c == 'y'; } };

if (b1) { std::cout << "b1 is true\n"; }
if (b2) { std::cout << "b2 is true\n"; }
if (b3) { std::cout << "b3 is true\n"; }
```

- std::any: 这是一个将容器表示为可复制构造类型值的类。下面的代码中使用了一个例子:

```
std::any u;

u = knight{};
if (u.has_value())
  std::any_cast<knight>(u).attack();

u = mage{};
if (u.has_value())
  std::any_cast<mage>(u).attack();
```

类型擦除是一种习语，将面向对象编程的继承与模板结合起来，以创建可以存储任何类型的包装器。本节中，我们了解了模式的表现形式和工作方式，以及该模式的一些实际实现。

下一节中，我们将讨论标记分派。

7.5. 标记分派

标记分派是一种技术，能够在编译时选择一个或另一个函数重载，是 std::enable_if 和 SFINAE 的替代方案，易于理解和使用。术语标记描述了一个没有成员 (数据) 或函数 (行为) 的空类。这样的类仅用于定义函数的形参 (通常是最后一个)，以决定是否在编译时选择它，这取决于提供的参数。

标准库包含一个名为 std::advance 的实用函数:

```
template<typename InputIt, typename Distance>
void advance(InputIt& it, Distance n);
```

C++17 中，这也是 constexpr(稍后会详细介绍)。该函数将给定的迭代器增加 n 个元素。但迭代器有几种类型 (输入、输出、向前、双向和随机)，所以这样的操作可以以不同的方式计算:

- 对于输入迭代器，可以调用 operator++ n 次。
- 对于双向迭代器，可以调用 operator++ n 次 (若 n 是一个正数) 或运算符-n 次 (若 n 是一个负数)。
- 对于随机访问迭代器，它可以使用 operator+= 直接增加 n 个元素。

所以，可以有三种不同的实现，可以在编译时选择与所调用迭代器的类别最匹配的实现。一种解决方案是标记分派，首先要做的是定义标记，标记是空类。因此，对应于这 5 种迭代器类型的标记可进行如下定义:

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag : input_iterator_tag {};
struct bidirectional_iterator_tag :
  forward_iterator_tag {};
struct random_access_iterator_tag :
  bidirectional_iterator_tag {};
```

这正是 C++ 标准库中 std 名称空间中定义它们的方式，这些标记用于为 std::advance 的每次重载定义一个附加参数:

```
namespace std
{
  namespace details
  {
    template <typename Iter, typename Distance>
    void advance(Iter& it, Distance n,
    std::random_access_iterator_tag)
    {
      it += n;
    }

    template <typename Iter, typename Distance>
    void advance(Iter& it, Distance n,
           std::bidirectional_iterator_tag)
    {
      if (n > 0)
      {
        while (n--) ++it;
      }
      else
      {
        while (n++) --it;
      }
    }

    template <typename Iter, typename Distance>
    void advance(Iter& it, Distance n,
             std::input_iterator_tag)
    {
      while (n--)
      {
        ++it;
      }
    }
  }
}
```

这些重载定义在 std 命名空间的单独 (内部) 命名空间中，这样标准命名空间就不会被不必要的定义所污染。这里可以看到，每个重载都有三个参数: 对迭代器的引用、要递增 (或递减) 的元素数量和标记。

最后要做的是提供一个用于直接使用的高级函数的定义。这个函数没有第三个形参，但是通过确定调用迭代器的类别来调用这些重载中的一个。其实现可能如下所示:

```
namespace std
{
  template <typename Iter, typename Distance>
  void advance(Iter& it, Distance n)
  {
    details::advance(it, n,
```

```
      typename std::iterator_traits<Iter>::
               iterator_category{});
  }
}
```

std::iterator_traits 类为迭代器类型定义了一种接口，包含几个成员类型，其中一个是 iterator_category。这将解析为前面定义的一个迭代器标记，例如 std::input_iterator_tag 用于输入迭代器，std::random_access_iterator_tag 用于随机迭代器。基于所提供的迭代器的类别，实例化这些标记类，在编译时从 details 命名空间确定适当的重载选择。就可以像下面这样使用 std::advance 函数了:

```
std::vector<int> v{ 1,2,3,4,5 };
auto sv = std::begin(v);
std::advance(sv, 2);

std::list<int> l{ 1,2,3,4,5 };
auto sl = std::begin(l);
std::advance(sl, 2);
```

std::vector 迭代器的类别类型是随机访问。另一方面，std::list 的迭代器类别类型是双向的。但可以利用标记调度技术使用依赖于不同优化实现的单个函数。

7.5.1 标签分派的替代方案

C++17 之前，标记分派的唯一替代方案是 SFINAE 和 enable_if。这是一种相当传统的技术，在现代 C++ 中有更好的替代方案——constexpr if 和概念。

constexpr if

C++11 引入了 constexpr 值的概念，其是编译时已知的值，也是可以在编译时求值的函数 (若所有输入都是编译时的值)。C++14、C++17 和 C++20 中，许多标准库函数或标准库类的成员函数更改为 constexpr。其中之一是 std::advance，它在 C++17 中的实现基于 constexpr if 特性，也在 C++17 中添加的。

下面是 C++17 中的可能实现:

```
template<typename It, typename Distance>
constexpr void advance(It& it, Distance n)
{
  using category =
  typename std::iterator_traits<It>::iterator_category;
  static_assert(std::is_base_of_v<std::input_iterator_tag,
  category>);
  auto dist =
  typename std::iterator_traits<It>::difference_type(n);
  if constexpr (std::is_base_of_v<
  std::random_access_iterator_tag,
  category>)
  {
    it += dist;
  }
```

```
  else
  {
    while (dist > 0)
    {
      --dist;
      ++it;
    }
    if constexpr (std::is_base_of_v<
            std::bidirectional_iterator_tag,
            category>)
    {
      while (dist < 0)
      {
        ++dist;
        --it;
      }
    }
  }
}
```

虽然这个实现仍然使用前面的迭代器标记，但不再用于调用不同的重载函数，而是用于确定一些编译时表达式的值。std::is_base_of 类型特性 (通过 std::is_base_of_v 变量模板) 用于在编译时确定迭代器类别的类型。

这个实现有几个优点:

- 具有算法的单一实现 (std 命名空间中)
- 不需要多个重载，实现细节定义在单独的命名空间中

调用端代码不受影响，所以标准库实现者能够用基于 constexpr if 的新版本替换基于标记分派的旧版本，而不会影响调用 std::advance 的代码。

然而，在 C++20 中有一个更好的选择。接下来让我们来探索一下。

概念

前一章专门介绍 C++20 中引入的约束和概念，不仅看到了这些特性是如何工作的，还看到了标准库在几个头文件中定义的一些概念，如 <concepts> 和 <iterator>。其中一些概念指定类型是某个迭代器类别。例如，std::input_iterator 指定类型为输入迭代器。类似地，还定义了以下概念:std::output_iterator、std::forward_iterator、std::bidirectional_iterator、std::random_access_iterator 和 std::contiguous_iterator(最后一个表示迭代器是随机访问迭代器，指的是连续存储在内存中的元素)。

std::input_iterator 概念的定义如下:

```
template<class I>
  concept input_iterator =
    std::input_or_output_iterator<I> &&
    std::indirectly_readable<I> &&
    requires { typename /*ITER_CONCEPT*/<I>; } &&
    std::derived_from</*ITER_CONCEPT*/<I>,
              std::input_iterator_tag>;
```

值得注意的是，这个概念是一组约束，用于验证以下内容:

- 迭代器是可解引用的 (支持 *i) 和可递增的 (支持 ++i 和 i++)
- 迭代器类别派生自 std::input_iterator_tag。

类别检查是在约束内执行的，所以这些概念仍然基于迭代器标记，但技术上与标记调度有很大不同。因此，在 C++20 中可以有 std::advance 算法的另一个实现，如下所示:

```
template <std::random_access_iterator Iter, class Distance>
void advance(Iter& it, Distance n)
{
  it += n;
}

template <std::bidirectional_iterator Iter, class Distance>
void advance(Iter& it, Distance n)
{
  if (n > 0)
  {
    while (n--) ++it;
  }
  else
  {
    while (n++) --it;
  }
}

template <std::input_iterator Iter, class Distance>
void advance(Iter& it, Distance n)
{
  while (n--)
  {
    ++it;
  }
}
```

有几点需要注意:

- advanced 函数还有三种不同的重载。
- 这些重载定义在 std 命名空间中，不需要单独的命名空间来隐藏实现细节。

尽管再次显式地编写了几个重载，但这种解决方案可以说比基于 constexpr if 的解决方案更容易阅读和理解，因为代码很好地分离到不同的单元 (函数) 中，因此更容易理解。

标记分派是在编译时进行重载选择的一项重要技术。但若使用 C++17 或 C++20，也有更好的选择。若编译器支持概念，出于前面提到的原因，应该选择这种替代方案。

下一节讨论的模式是表达式模板。

7.6. 表达式模板

表达式模板是一种元编程技术，支持在编译时对计算进行惰性计算。这有助于避免在运行时发生低效操作，但这并不是免费的，因为表达式模板需要更多的代码，并且阅读或理解起来可能很麻烦，其经常用于线性代数库的实现。

了解表达式模板之前，先了解它们可以解决什么问题。假设想要对矩阵进行一些操作，实现了基本的操作，加法、减法和乘法 (两个矩阵或一个标量和一个矩阵)。可以有以下表达式:

```
auto r1 = m1 + m2;
auto r2 = m1 + m2 + m3;
auto r3 = m1 * m2 + m3 * m4;
auto r4 = m1 + 5 * m2;
```

m1, m2, m3 和 m4 是矩阵; 类似地，r1, r2, r3 和 r4 是由右边的操作得到的矩阵。第一个操作没有任何问题:m1 和 m2 相加，结果赋值给 r1。第二个操作是不同的，因为有三个矩阵相加。所以首先操作 m1 和 m2，并创建一个临时对象，然后将其添加到 m3，并将结果分配给 r2。

对于第三个操作，有两个临时值: 一个是 m1 和 m2 相乘的结果，一个是 m3 和 m4 相乘的结果; 然后这两个相加结果被赋值给 r3。最后一个操作类似于第二个操作，所以一个临时对象是由标量 5 和矩阵 m2 相乘产生的，然后将这个临时对象添加到 m1，并将结果分配给 r4。

操作越复杂，生成的临时项就越多。当对象很大时，这会影响性能。表达式模板通过将计算建模为编译时表达式来帮助避免这种情况。整个数学表达式 (如 m1 + 5 * m2) 在计算赋值时变成一个单独的表达式模板，不需要任何临时对象。

为了演示，我们将使用 vector 而不是矩阵构建一些示例，因为 vector 是更简单的数据结构，而且练习的重点不是数据的表示，而是表达式模板的创建。下面的代码中，可以看到一个 vector 类的最小实现，提供了几个操作:

- 从初始化列表或表示大小的值构造实例 (不初始化值)
- 检索向 vector 的元素数
- 使用下标操作符 ([]) 访问元素

代码如下所示:

```
template<typename T>
struct vector
{
  vector(std::size_t const n) : data_(n) {}

  vector(std::initializer_list<T>&& l) : data_(l) {}

  std::size_t size() const noexcept
  {
    return data_.size();
  }

  T const & operator[](const std::size_t i) const
  {
    return data_[i];
```

```
  }

  T& operator[](const std::size_t i)
  {
    return data_[i];
  }

private:
  std::vector<T> data_;
};
```

这看起来非常类似于 std::vector 标准容器，其在内部会使用这个容器来保存数据，但这与我们要解决的问题无关。这里，使用的是 vector，而不是矩阵，因为用几行代码来表示它更容易。有了这个类，就可以定义必要的操作: 两个向量之间，以及标量与向量之间的加法和乘法:

```
template<typename T, typename U>
auto operator+ (vector<T> const & a, vector<U> const & b)
{
  using result_type = decltype(std::declval<T>() +
  std::declval<U>());
  vector<result_type> result(a.size());
  for (std::size_t i = 0; i < a.size(); ++i)
  {
    result[i] = a[i] + b[i];
  }
  return result;
}

template<typename T, typename U>
auto operator* (vector<T> const & a, vector<U> const & b)
{
  using result_type = decltype(std::declval<T>() +
  std::declval<U>());
  vector<result_type> result(a.size());
  for (std::size_t i = 0; i < a.size(); ++i)
  {
    result[i] = a[i] * b[i];
  }
  return result;
}

template<typename T, typename S>
auto operator* (S const& s, vector<T> const& v)
{
  using result_type = decltype(std::declval<T>() +
  std::declval<S>());
  vector<result_type> result(v.size());
  for (std::size_t i = 0; i < v.size(); ++i)
  {
    result[i] = s * v[i];
```

```
  }
  return result;
}
```

这些实现相对简单，在这一点上不应该构成理解问题。+ 和 * 运算符都取两个可能类型不同的 vector，例如 vector<int> 和 vector<double>，并返回一个包含结果类型元素的向量，是由使用 std::declval 将模板类型 T 和 U 的两个值相加的结果决定的。对于标量和向量相乘，也可以使用类似的实现。有了这些操作符，可以编写如下代码:

```
vector<int> v1{ 1,2,3 };
vector<int> v2{ 4,5,6 };
double a{ 1.5 };

vector<double> v3 = v1 + a * v2; // {7.0, 9.5, 12.0}
vector<int> v4 = v1 * v2 + v1 + v2; // {9, 17, 27}
```

如前所述，这将在计算 v3 时创建一个临时对象，在计算 v4 时创建两个临时对象。下面的图表举例说明了这一点。第一个是第一个计算，v3 = v1 + a * v2:

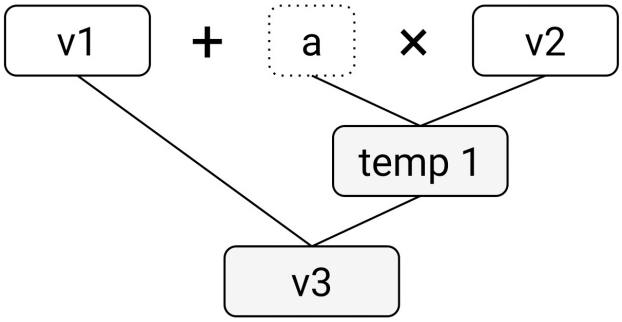


图 7.2: 第一个表达式

下面显示的第二个图，给出了第二个表达式，v4 = v1 * v2 + v1 + v2 计算的表示:

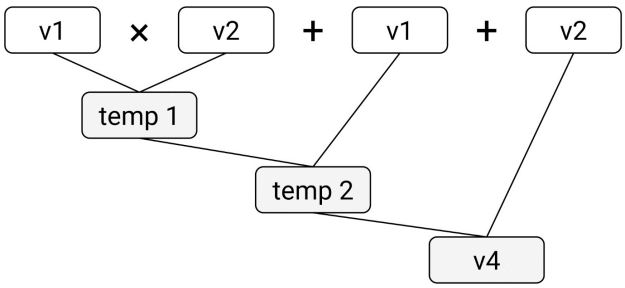


图 7.3: 第二个表达式

为了避免这些临时变量，可以使用表达式模板模式重写 vector 类的实现:

- 定义类模板来表示两个对象之间的表达式 (例如两个向量相加或相乘的表达式)。
- 修改 vector 类并为其内部数据参数化容器，默认情况下，容器内部数据是 std::vector，但也可以是表达式模板。
- 更改重载的 + 和 * 操作符的实现。

从向量实现开始:

```cpp
template<typename T, typename C = std::vector<T>>
struct vector
{
  vector() = default;

  vector(std::size_t const n) : data_(n) {}

  vector(std::initializer_list<T>&& l) : data_(l) {}


  vector(C const & other) : data_(other) {}
  template<typename U, typename X>
  vector(vector<U, X> const& other) : data_(other.size())
  {
    for (std::size_t i = 0; i < other.size(); ++i)
    data_[i] = static_cast<T>(other[i]);
  }

  template<typename U, typename X>
  vector& operator=(vector<U, X> const & other)
  {
    data_.resize(other.size());
    for (std::size_t i = 0; i < other.size(); ++i)
      data_[i] = static_cast<T>(other[i]);
    return *this;
  }

  std::size_t size() const noexcept
  {
    return data_.size();
  }

  T operator[](const std::size_t i) const
  {
    return data_[i];
  }

  T& operator[](const std::size_t i)
  {
    return data_[i];
  }

  C& data() noexcept { return data_; }

  C const & data() const noexcept { return data_; }

private:
  C data_;
};
```

除了初始实现中可用的操作外，还定义了以下操作:

- 默认构造函数
- 容器的转换构造函数
- 复制包含可能不同类型元素向量的复制构造函数
- 来自包含可能不同类型元素向量的复制赋值操作符
- 成员函数数据，提供对存储数据的底层容器的访问

表达式模板是一个简单的类模板，其存储两个操作数，并提供了一种执行操作求值的方法。本例中，需要实现两个向量相加、两个向量相乘，以及标量与向量相乘的表达式。来看看两个向量相加的表达式模板的实现:

```
template<typename L, typename R>
struct vector_add
{
  vector_add(L const & a, R const & b) : lhv(a), rhv(b) {}

  auto operator[](std::size_t const i) const
  {
    return lhv[i] + rhv[i];
  }

  std::size_t size() const noexcept
  {
    return lhv.size();
  }

private:
  L const & lhv;
  R const & rhv;
};
```

该类存储对两个向量的常量引用 (或者，任何重载下标操作符，并提供 size 成员函数的类型)。表达式的求值发生在重载下标操作符中，而不是整个向量，只添加指定索引处的元素。

注意，此实现不处理不同大小的向量 (可以将其作为练习进行更改)。但应该很容易理解这种方法的惰性性质，因为加法操作只在调用下标操作符时发生。

两个操作的乘法表达式模板，以类似的方式实现:

```
template<typename L, typename R>
struct vector_mul
{
  vector_mul(L const& a, R const& b) : lhv(a), rhv(b) {}

  auto operator[](std::size_t const i) const
  {
    return lhv[i] * rhv[i];
  }
```

```
  std::size_t size() const noexcept
  {
    return lhv.size();
  }

private:
  L const & lhv;
  R const & rhv;
};

template<typename S, typename R>
struct vector_scalar_mul
{
  vector_scalar_mul(S const& s, R const& b) :
    scalar(s), rhv(b)
  {}

  auto operator[](std::size_t const i) const
  {
    return scalar * rhv[i];
  }

  std::size_t size() const noexcept
  {
    return rhv.size();
  }

private:
  S const & scalar;
  R const & rhv;
};
```

更改的最后一部分是重载的 + 和 * 操作符的定义:

```
template<typename T, typename L, typename U, typename R>
auto operator+(vector<T, L> const & a,
vector<U, R> const & b)
{
  using result_type = decltype(std::declval<T>() +
                  std::declval<U>());
  return vector<result_type, vector_add<L, R>>(
    vector_add<L, R>(a.data(), b.data()));
}

template<typename T, typename L, typename U, typename R>
auto operator*(vector<T, L> const & a,
vector<U, R> const & b)
{
  using result_type = decltype(std::declval<T>() +
```

```
                std::declval<U>());
  return vector<result_type, vector_mul<L, R>>(
    vector_mul<L, R>(a.data(), b.data()));
}

template<typename T, typename S, typename E>
auto operator*(S const& a, vector<T, E> const& v)
{
  using result_type = decltype(std::declval<T>() +
                std::declval<S>());
  return vector<result_type, vector_scalar_mul<S, E>>(
    vector_scalar_mul<S, E>(a, v.data()));
}
```

尽管在实现此模式时代码更加复杂，但调用端代码不需要更改。这段代码没有任何修改，但以一种懒惰的方式实现。每个元素的求值，由 vector 类的复制构造函数和复制赋值操作符中的下标操作符触发。

若这个模式看起来很麻烦，那么还有更好的选择: 范围库。

7.6.1 使用范围库作为表达式模板的替代方案

C++20 的主要特性之一是范围库，其是容器的泛化——允许迭代其数据 (元素) 的类。范围库的一个关键元素是视图，是其他范围的非所有权包装器，通过某些操作转换底层范围。

此外，其是惰性求值的，构造、复制或销毁的时间不取决于底层范围的大小。惰性求值 (即在元素被请求时应用转换，而不是在视图创建时) 是该库的一个关键特性。然而，这正是表达式模板所支持的，所以表达式模板的许多用法可以用范围代替。

C++ 的范围库是基于 Eric Niebler 创建的 range-v3 库。这个库可以在https://github.com/ericniebler/range-v3/上获取。使用 range-v3，可以编写以下代码来执行操作 v1 + a * v2:

```
namespace rv = ranges::views;
std::vector<int> v1{ 1, 2, 3 };
std::vector<int> v2{ 4, 5, 6 };
double a { 1.5 };

auto sv2 = v2 |
       rv::transform([&a](int val) {return a * val; });
auto v3 = rv::zip_with(std::plus<>{}, v1, sv2);
```

vector 类不需要自定义实现，这只适用于 std::vector 容器，也不需要重载任何操作符。代码应该很容易理解，若对范围库比较熟悉的话。首先，创建一个视图，通过将每个元素与标量相乘来转换 v2 向量的元素。然后，创建第二个视图，将加号运算符应用于 v1 范围的元素和前一个操作生成的视图。

但这段代码不能使用标准库在 C++20 中编写，因为 zip_with 视图没有包含在 C++20 中。但这个视图将在 C++23 中以 zip_view 的名称提供。所以在 C++23 中，可以这样实现:

```
namespace rv = std::ranges::views;
std::vector<int> v1{ 1, 2, 3 };
```

```
std::vector<int> v2{ 4, 5, 6 };
double a { 1.5 };

auto sv2 = v2 |
           rv::transform([&a](int val) {return a * val; });
auto v3 = rv::zip_wiew(std::plus<>{}, v1, sv2);
```

为了结束对表达式模板模式的讨论，应该记住以下要点: 该模式旨在为代价高昂的操作提供惰性求值，而这样做的代价是必须编写更多的代码 (也可以说是更麻烦) 和增加编译时间 (因为沉重的模板代码将对此产生影响)。然而，从 C++20 开始，这个模式的替代可由范围库完成。

本章的下一节也是最后一节，我们将讨论类型列表。

7.7. 类型列表

类型列表是一种编译时构造，能够管理类型序列。类型列表在某种程度上类似于元组，但不存储数据。类型列表只携带类型信息，并在编译时用于实现不同的元编程算法、类型开关或设计模式 (如抽象工厂或访问者)。

重要的 Note

虽然 type list 和 typelist 两个名称都在使用，但大多数时候，会在 C++ 书籍和文章中看到术语 typelist。因此，这将是我们在本书中使用的术语名称 typelist。

类型列表是由 Andrei Alexandrescu 在他的书《现代 C++ 设计》中普及开来的，这本书在 C++11(以及可变参数模板) 发布的 10 年前出版。Alexandrescu 定义了一个类型列表:

```
template <class T, class U>
struct Typelist
{
  typedef T Head;
  typedef U Tail;
};
```

他的实现中，一个类型表由头部 (一个类型) 和尾部 (另一个类型表) 组成。为了在类型列表上执行各种操作 (稍后将讨论)，还需要一个类型来表示类型列表的末尾。这可以是一个简单的空类型，Alexandrescu 的定义如下:

```
class null_typelist {};
```

有了这两个结构，就可以用以下方式定义类型列表:

```
typedef Typelist<int,
         Typelist<double, null_typelist>> MyList;
```

可变参数模板简化了类型列表的实现，如下所示:

```
template <typename ... Ts>
struct typelist {};

using MyList = typelist<int, double>;
```

类型列表操作的实现 (访问给定索引处的类型，从列表中添加或删除类型等) 根据所选方法的不同而有很大差异。本书中，只考虑可变参数模板的版本。这种方法的优点是在不同层次上都很简单: 类型列表的定义更短，不需要类型来表示列表的末尾，定义类型列表别名也更短，更容易阅读。

目前，也许许多由类型列表解决的问题，也可以使用可变参数模板来解决。这里有一个例子: 试想一个可变元函数 (一个执行类型转换的类型特征)，对类型模板参数进行一些转换 (比如添加 const 限定符)。此元函数定义了一个表示输入类型的成员类型和一个表示转换类型的成员类型。若这样定义，是行不通的:

```
template <typename ... Ts>
struct transformer
{
  using input_types = Ts...;
  using output_types = std::add_const_t<Ts>...;
};
```

因为在此上下文中不可能展开参数包，所以这段代码会产生编译器错误。解决方案是使用类型列表:

```
template <typename ... Ts>
struct transformer
{
  using input_types = typelist<Ts...>;
  using output_types = typelist<std::add_const_t<Ts>...>;
};

static_assert(
  std::is_same_v<
    transformer<int, double>::output_types,
    typelist<int const, double const>>);
```

代码变化很小，但产生了预期的结果。虽然这是一个需要类型列表的好例子，但它不是使用类型列表的例子。下面我们将看一个这样的例子。

7.7.1 使用类型列表

讨论如何在类型列表上实现操作之前，有必要研究一个更复杂的示例。这将使各位读者更加了解类型列表的可能用法，也可以在网上搜索更多用法。

回到 game_unit 的例子。简单起见，只考虑以下类:

```
struct game_unit
{
  int attack;
  int defense;
};
```

game_unit 有两个数据成员，代表进攻和防守的指数 (或级别)。我们希望借助一些函子对这些成员进行操作更改。下面的代码中显示了两个这样的函数:

```
struct upgrade_defense
```

```
{
  void operator()(game_unit& u)
  {
    u.defense = static_cast<int>(u.defense * 1.2);
  }
};
struct upgrade_attack
{
  void operator()(game_unit& u)
  {
    u.attack += 2;
  }
};
```

第一种增加防御指数 20%，第二种增加攻击指数两个单位。尽管这是一个用于演示的小示例，也可以想象可以在一些定义良好的组合中应用类似的函子的更大种类。例子中，我们想在 game_unit 对象上应用这两个函子，希望有这样一个函数:

```
void upgrade_unit(game_unit& unit)
{
  using upgrade_types =
    typelist<upgrade_defense, upgrade_attack>;
  apply_functors<upgrade_types>{}(unit);
}
```

upgrade_unit 函数接受一个 game_unit 对象，并对其应用 upgrade_defense 和 upgrade_attack 函数子。为此，使用另一个名为 apply_functors 的辅助函子，这是一个只有一个 template 参数的类模板，模板参数是一个类型列表。apply_functors 函子的可能实现如下所示:

```
template <typename TL>
struct apply_functors
{
private:
  template <size_t I>
  static void apply(game_unit& unit)
  {
    using F = at_t<I, TL>;
    std::invoke(F{}, unit);
  }

  template <size_t... I>
  static void apply_all(game_unit& unit,
  {
    (apply<I>(unit), ...);
  }

public:
  void operator()(game_unit& unit) const
  {
    apply_all(unit,
```

```
    std::make_index_sequence<length_v<TL>>{});
  }
};
```

这个类模板有一个重载调用操作符和两个私有辅助函数:

- apply, 将类型列表的 I 索引中的函子应用到 game_unit 对象上。
- apply_all, 通过在包扩展中使用 apply 函数，将类型列表中的所有函子应用到 game_unit 对象。

可以像下面这样使用 upgrade_unit 函数:

```
game_unit u{ 100, 50 };
std::cout << std::format("{},{}\n", u.attack, u.defense);
// prints 100,50

upgrade_unit(u);
std::cout << std::format("{},{}\n", u.attack, u.defense);
// prints 102,60
```

若注意过 apply_functors 类模板的实现，就会注意到 at_t 别名模板和 length_v 变量模板的使用，这两个模板我们还没有定义。

7.7.2 实现对类型列表的操作

类型列表是一种在编译时只携带有 value 信息的类型，类型列表可充当其他类型的容器。使用类型列表时，需要执行各种操作，例如计算列表中的类型，在给定索引处访问类型，在列表的开头或结尾添加类型，或者反向操作，从列表的开头或结尾删除类型等。仔细想想，这些都是容器中的典型操作，比如 vector 容器。所以在本节中，我们将讨论如何实现以下操作:

- size: 列表的长度
- front: 检索列表中的第一个类型
- back: 检索列表中的最后一个类型
- at: 检索列表中指定索引处的类型
- push_back: 将新类型添加到列表末尾
- push_front: 将一个新类型添加到列表开头
- pop_back: 删除列表末尾的类型
- pop_front: 删除列表开头的类型

类型列表是编译时构造，是一个不可改变的实体。因此，添加或删除类型的操作不会修改类型列表，而是创建一个新的类型列表。首先，从最简单的操作开始，即检索类型列表的长度。

为了避免与 size_t 类型的命名混淆，将此操作称为 length_t，而不是 size_t。可以这样定义:

```
namespace detail
{
  template <typename TL>
  struct length;

  template <template <typename...> typename TL,
        typename... Ts>
```

```
  struct length<TL<Ts...>>
  {
    using type =
    std::integral_constant<std::size_t, sizeof...(Ts)>;
  };
}

template <typename TL>
using length_t = typename detail::length<TL>::type;

template <typename TL>
constexpr std::size_t length_v = length_t<TL>::value;
```

detail 命名空间中，有一个名为 length 的类模板，一个类型列表有一个主模板 (没有定义) 和一个特化。这个特化定义了一个名为 type 的成员类型，是一个 std::integral_constant，其值类型为 std::size_t，表示形参包 Ts 中的参数数量。此外，还有一个别名模板 length_h，是 length 类模板中名为 type 的成员的别名。最后，有一个名为 length_v 的变量模板，是由 std::integral_constant 成员的值初始化，该成员也称为 value。

可以通过 static_assert 来验证这个实现的正确性:

```
static_assert(
  length_t<typelist<int, double, char>>::value == 3);
static_assert(length_v<typelist<int, double, char>> == 3);
static_assert(length_v<typelist<int, double>> == 2);
static_assert(length_v<typelist<int>> == 1);
```

这里使用的方法将用于定义所有其他操作。接下来，看看如何访问列表中的 front 类型:

```
struct empty_type {};
namespace detail
{
  template <typename TL>
  struct front_type;

  template <template <typename...> typename TL,
        typename T, typename... Ts>
  struct front_type<TL<T, Ts...>>
  {
    using type = T;
  };

  template <template <typename...> typename TL>
  struct front_type<TL<>>
  {
    using type = empty_type;
  };
}

template <typename TL>
using front_t = typename detail::front_type<TL>::type;
```

detail 命名空间中，有一个名为 front_type 的类模板。同样，我们声明了一个主模板，但没有定义。但有两个特化: 一个用于至少包含一种类型的类型列表，另一个用于空类型列表。前一种情况下，type 成员是类型列表中的第一个类型的别名。后一种情况下，没有类型，所以 type 成员别名为 empty_type 类型。这是一个空类，唯一的作用是充当不返回类型的操作的返回类型。可以进行如下的验证:

```
static_assert(
  std::is_same_v<front_t<typelist<>>, empty_type>);
static_assert(
  std::is_same_v<front_t<typelist<int>>, int>);
static_assert(
  std::is_same_v<front_t<typelist<int, double, char>>,
            int>);
```

若想访问 back 类型的操作的实现与此类似，应该不会失望。它是这个样子:

```
namespace detail
{
  template <typename TL>
  struct back_type;

  template <template <typename...> typename TL,
        typename T, typename... Ts>
  struct back_type<TL<T, Ts...>>
  {
    using type = back_type<TL<Ts...>>::type;
  };

  template <template <typename...> typename TL,
            typename T>
  struct back_type<TL<T>>
  {
    using type = T;
  };

  template <template <typename...> typename TL>
  struct back_type<TL<>>
  {
    using type = empty_type;
  };
}

template <typename TL>
using back_t = typename detail::back_type<TL>::type;
```

实现的区别是 back_type 类模板有三种特化，并且涉及到递归。这三种特化分别针对空类型列表、具有单一类型的类型列表，以及具有两个或多个类型的类型列表。最后一个 (实际上是前一个代码示例中的第一个) 是在其类型成员的定义中使用模板递归。为了确保我们以正确的方式实现操

作，可以进行如下验证:

```
static_assert(
  std::is_same_v<back_t<typelist<>>, empty_type>);
static_assert(
  std::is_same_v<back_t<typelist<int>>, int>);
static_assert(
  std::is_same_v<back_t<typelist<int, double, char>>,
           char>);
```

除了访问类型列表中的第一个和最后一个类型外，还对在给定索引处访问类型感兴趣，但这个操作的实现并不是那么简单:

```
namespace detail
{
  template <std::size_t I, std::size_t N, typename TL>
  struct at_type;

  template <std::size_t I, std::size_t N,
        template <typename...> typename TL,
        typename T, typename... Ts>
  struct at_type<I, N, TL<T, Ts...>>
  {
    using type =
    std::conditional_t<
    I == N,
    T,
    typename at_type<I, N + 1, TL<Ts...>>::type>;
  };

  template <std::size_t I, std::size_t N>
  struct at_type<I, N, typelist<>>
  {
    using type = empty_type;
  };
}

template <std::size_t I, typename TL>
using at_t = typename detail::at_type<I, 0, TL>::type;
```

at_t 别名模板有两个模板参数: 一个索引和一个类型列表。at_t 模板是来自 detail 命名空间的 at_type 类模板的成员类型的别名。主模板有三个模板参数: 一个索引表示要检索的类型的位置 (I)，另一个索引表示列表中类型迭代的当前位置 (N)，以及一个类型列表 (TL)。

这个主模板有两种特化: 一种特化用于至少包含一种类型的类型列表，另一种特化用于空类型列表。后一种情况下，成员类型是 empty_type 类型的别名。前一种情况下，成员类型是借助 std::conditional_t 元函数定义的。当 I == N 时将其成员类型定义为第一个类型 (T)，当此条件为 false 时将其成员类型定义为第二个类型 (typename at_type<I, N + 1, TL<Ts⋯>>::type)。这里，再次使用模板递归，在每次迭代中增加第二个索引的值。下面是 static_assert 的验证实现:

```
static_assert(
  std::is_same_v<at_t<0, typelist<>>, empty_type>);
static_assert(
  std::is_same_v<at_t<0, typelist<int>>, int>);
static_assert(
  std::is_same_v<at_t<0, typelist<int, char>>, int>);

static_assert(
  std::is_same_v<at_t<1, typelist<>>, empty_type>);
static_assert(
  std::is_same_v<at_t<1, typelist<int>>, empty_type>);
static_assert(
  std::is_same_v<at_t<1, typelist<int, char>>, char>);

static_assert(
  std::is_same_v<at_t<2, typelist<>>, empty_type>);
static_assert(
  std::is_same_v<at_t<2, typelist<int>>, empty_type>);
static_assert(
  std::is_same_v<at_t<2, typelist<int, char>>,
            empty_type>);
```

要实现的下一类操作是在类型列表的开头和结尾添加类型，称之为 push_back_t 和 push_front_t，其定义如下:

```
namespace detail
{
  template <typename TL, typename T>
  struct push_back_type;

  template <template <typename...> typename TL,
        typename T, typename... Ts>
  struct push_back_type<TL<Ts...>, T>
  {
    using type = TL<Ts..., T>;
  };

  template <typename TL, typename T>
  struct push_front_type;

  template <template <typename...> typename TL,
        typename T, typename... Ts>
  struct push_front_type<TL<Ts...>, T>
  {
    using type = TL<T, Ts...>;
  };
}

template <typename TL, typename T>
using push_back_t =
```

```
  typename detail::push_back_type<TL, T>::type;

template <typename TL, typename T>
using push_front_t =
  typename detail::push_front_type<TL, T>::type;
```

根据在前面的操作中看到的内容，这些操作应该很容易理解。相反的操作，当从类型列表中删除第一个或最后一个类型时，则更加复杂。第一个，pop_front_t:

```
namespace detail
{
  template <typename TL>
  struct pop_front_type;

  template <template <typename...> typename TL,
            typename T, typename... Ts>
  struct pop_front_type<TL<T, Ts...>>
  {
    using type = TL<Ts...>;
  };

  template <template <typename...> typename TL>
  struct pop_front_type<TL<>>
  {
    using type = TL<>;
  };
}

template <typename TL>
using pop_front_t =
  typename detail::pop_front_type<TL>::type;
```

我们有主模板 pop_front_type 和两个特化: 第一个特化用于至少有一种类型的类型列表，第二个特化用于空类型列表。后者将成员 type 定义为空列表，前者将成员 type 定义为尾部由类型列表参数组成的类型列表。

最后一个操作，删除类型列表中的最后一个类型，称为 pop_back_t:

```
namespace detail
{
  template <std::ptrdiff_t N, typename R, typename TL>
  struct pop_back_type;

  template <std::ptrdiff_t N, typename... Ts,
        typename U, typename... Us>
  struct pop_back_type<N, typelist<Ts...>,
              typelist<U, Us...>>
  {
    using type =
      typename pop_back_type<N - 1,
                  typelist<Ts..., U>,
```

```
                    typelist<Us...>>::type;
  };

  template <typename... Ts, typename... Us>
  struct pop_back_type<0, typelist<Ts...>,
              typelist<Us...>>
  {
    using type = typelist<Ts...>;
  };

  template <typename... Ts, typename U, typename... Us>
  struct pop_back_type<0, typelist<Ts...>,
              typelist<U, Us...>>
  {
    using type = typelist<Ts...>;
  };

  template <>
  struct pop_back_type<-1, typelist<>, typelist<>>
  {
    using type = typelist<>;
  };
}

template <typename TL>
using pop_back_t = typename detail::pop_back_type<
  static_cast<std::ptrdiff_t>(length_v<TL>)-1,
        typelist<>, TL>::type;
```

为了实现这个操作，需要从一个类型列表开始，然后逐元素递归地构造另一个类型列表，直到得到输入类型列表中的最后一个类型，这个类型应该省略。为此，我们使用计数器，表示迭代了该类型列表的次数。

这是在类型列表的大小减 1 时开始的，当达到 0 时需要停止。由于这个原因，pop_back_type 类模板有四个特化，一个用于在类型列表中进行迭代时的一般情况，两个用于计数器达到 0 时的情况，一个用于计数器达到-1 时的情况。这是初始类型列表为空时的情况 (length_t<TL>-1 将计算为-1)。下面是如何使用 pop_back_t 的演示:

```
static_assert(std::is_same_v<pop_back_t<typelist<>>,
                typelist<>>);
static_assert(std::is_same_v<pop_back_t<typelist<double>>,
                typelist<>>);
static_assert(
  std::is_same_v<pop_back_t<typelist<double, char>>,
                 typelist<double>>);
static_assert(
  std::is_same_v<pop_back_t<typelist<double, char, int>>,
                 typelist<double, char>>);
```

定义了这些后，我们提供了一系列使用类型列表所必需的操作。length_t 和 at_t 操作在前面的例子中使用，在 game_unit 对象上执行函数子。

希望本节对对类型列表的介绍，对读者们有所帮助，使您不仅能够理解它们是如何实现的，而且还能够理解如何使用它们。

7.8. 总结

本章致力于学习各种元编程技术。我们首先了解动态多态和静态多态之间的区别，然后研究了用于实现静态多态的奇异迭代模板模式。

混入 (Mixins) 是另一种与 CRTP 目的相似的模式——向类中添加功能，但与 CRTP 不同的是，不需要修改它们。我们学习的第三种技术是类型擦除，允许对不相关的相似类型进行泛型处理。第二部分中，学习了标记分派——允许在编译时的重载和表达式模板之间进行选择——支持在编译时对计算进行惰性求值，以避免在运行时发生低效操作。最后，探讨了类型列表，并学习了如何使用它们，以及如何使用它们实现操作。

下一章中，我们将讨论标准模板库、容器、迭代器和算法核心。

7.9. 习题

1. CRTP 可以解决哪些问题?
2. 什么是 Mixins? 其目的是什么?
3. 什么是类型擦除?
4. 什么是标记分派，它的替代方案是什么?
5. 表达式模板是什么? 可以在哪里使用它们?

7.10. 扩展阅读

- Design Patterns: Elements of Reusable Object-Oriented Software –Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, p. 163, Addison-Wesley
- Modern C++ Design: Generic Programming and Design Patterns Applied –Andrei Alexandrescu, Addison-Wesley Professional
- Mixin-Based Programming in C++ –Yannis Smaragdakis, Don Batory, https://yanniss.github.io/practical-fmtd.pdf
- Curiously Recurring Template Patterns –James Coplien, http://sites.google.com/a/gertrudandcope.com/info/Publications/InheritedTemplate.pdf
- Mixin Classes: The Yang of the CRTP –Jonathan Boccara, https://www.fluentcpp.com/2017/12/12/mixin-classes-yang-crtp/
- What the Curiously Recurring Template Pattern can bring to your code –Jonathan Boccara, https://www.fluentcpp.com/2017/05/16/what-the-crtpbrings-to-code/
- Combining Static and Dynamic Polymorphism with C++ Mixin classes – Michael Afanasiev, https://michael-afanasiev.github.io/2016/08/03/Combining-Static-an

d-Dynamic-Polymorphism-with-C++Template-Mixins.html

- Why C++ is not just an Object-Oriented Programming Language – Bjarne Stroustrup, https://www.stroustrup.com/oopsla.pdf
- enable_shared_from_this - overview, examples, and internals – Hitesh Kumar, https://www.nextptr.com/tutorial/ta1414193955/enable_shared_from_this-overview-examples-and-internals
- Tag dispatch versus concept overloading –Arthur O'Dwyer, https://quuxplusone.github.io/blog/2021/06/07/tag-dispatch-andconcept-overloading/
- C++ Expression Templates: An Introduction to the Principles of Expression Templates – Klaus Kreft and Angelika Langer, http://www.angelikalanger.com/Articles/Cuj/ExpressionTemplates/ExpressionTemplates.htm
- We don't need no stinking expression templates, https://gieseanw.wordpress.com/2019/10/20/we-dont-need-no-stinkingexpression-templates/
- Generic Programming: Typelists and Applications –Andrei Alexandrescu, https://www.drdobbs.com/generic-programmingtypelists-andapplica/184403813
- Of type lists and type switches –Bastian Rieck,https://bastian.rieck.me/blog/posts/2015/type_lists_and_switches/

第 8 章 范围和算法

读到这里，您已经了解了 C++ 中模板的所有语法和机制，以及 C++20 标准。为您提供了编写从简单表单到复杂表单的模板所需的知识。模板是编写泛型库的关键，即使自己没有编写这样的库，仍然会使用一个或多个库。事实上，每天用 C++ 写的代码都使用模板。主要原因是作为一个现代 C++ 开发者，都会使用标准库，标准库就是一个基于模板的库。

然而，标准库是许多库的集合，例如容器库、迭代器库、算法库、数值库、输入/输出库、文件系统库、正则表达式库、线程支持库、工具库等。总的来说，这是一个很大的库，这个主题至少可以写成一本书。但值得探索库的一些关键部分，从而帮助读者们更好地理解正在或可能经常使用的一些概念和类型。

讨论这个主题会占用大量的篇幅，我将把讨论分为两个部分。

本章中，我们将讨论以下主题:

- 容器、迭代器和算法的设计
- 自定义容器和迭代器
- 自定义通用算法

本章结束时，可以了解标准模板库的三个主要核心，即容器、迭代器和算法。

本章开始时，我们将概述标准库在这方面所提供的功能。

8.1. 容器、迭代器和算法的设计

容器是表示元素集合的类型。这些集合可以基于各种数据结构实现，每种结构都具有不同的语义: 列表、队列、树等等。标准库提供了三类容器:

- 序列容器:vector、deque、list、array 和 forward_list
- 关联容器:set、map、multiset 和 multimap
- 无序关联容器:unordered_set、unordered_map、unordered_multiset 和 unordered_multimap

除此之外，还有为序列容器提供不同接口的容器适配器。这个类别包括堆 stack、queue 和 priority_queue 类。最后，还有一个名为 span 的类，表示连续对象序列上的非所属视图。

将这些容器作为模板的基本原理，已经在第 1 章模板介绍中介绍过。因为不希望为需要存储在容器中，每种不同类型的元素一次又一次地编写相同的实现。标准库中最常用的容器如下:

- vector: 一个连续存储在内存中的可变大小的元素集合。若没有定义特殊需求，是默认选择的容器。内部存储器根据需要自动扩展或收缩以容纳存储的元素。vector 分配的内存比需要的多，因此需要扩展的风险很低。扩展是一项开销很大的操作，因为需要分配新的内存，需要将当前存储的内容复制到新存储中，最后需要丢弃之前的存储。因为元素连续地存储在内存中，所以索引可以在常数时间内随机访问。
- array: 连续存储在内存中的固定大小的元素集合，size 必须是编译时常量表达式。数组类的语义与保存 C 风格数组 (T[n]) 的结构相同。就像 vector 类型一样，数组类的元素可以在常数时间内随机访问。
- map: 将值关联到唯一键的集合。键用比较函数排序，map 类通常实现为红黑树。搜索、插入或删除元素的操作具有对数 (log(n)) 复杂度。

- set: 唯一键的集合。键是容器中存储的实际值，map 类中没有键值对。就像 map 类一样，set 通常也使用红黑树实现，在搜索、插入和删除元素时具有对数 (log(n)) 复杂度。

不管它们的类型是什么，标准容器有一些共同点:

- 几种常见成员类型
- 存储管理的分配器 (除了 std::array 类)
- 几个公共成员函数 (其中一些在一个或另一个容器中缺失)
- 迭代器可访问存储的数据

以下成员类型由所有标准容器定义:

```
using value_type = /* ... */;
using size_type = std::size_t;
using difference_type = std::ptrdiff_t;
using reference = value_type&;
using const_reference = value_type const&;
using pointer = /* ... */;
using const_pointer = /* ... */;
using iterator = /* ... */;
using const_iterator = /* ... */;
```

这些别名的实际类型可能因容器而异。例如，对于 std::vector, value_type 是模板参数 T，但是对于 std::map, value_type 是 std::pair<const Key, T> 类型。这些成员类型的目的是进行泛型编程。

除了 std::array 类 (表示编译时已知大小的数组)，其他所有容器都是动态分配内存的，可以通过分配器对象来控制的。其类型指定为类型模板形参，但若没有指定，则所有容器默认为 std::allocator。这个标准分配器使用全局的 new 和 delete 操作符来分配和释放内存。标准容器的所有构造函数 (包括复制和移动构造函数) 允许指定分配器。

标准容器中还定义了一些常用的成员函数。下面是一些例子:

- size, 返回元素的数量 (std::forward_list 没有)。
- empty, 检查容器是否为空。
- clear, 清除容器的内容 (std::array，std::stack，std::queue 和 std::priority_queue 中没有)。
- swap, 交换容器对象的内容。
- begin 和 end 方法，返回到容器的开始和结束的迭代器 (std::stack, std::queue 和 std::priority_queue 中没有，尽管这些不是容器，而是容器适配器)。

最后提到了迭代器。这些类型抽象了容器中访问元素的细节，提供了一种统一的方法来标识和遍历容器中的元素。这很重要，因为标准库的关键部分是由通用算法表示的。有超过一百种这样的算法，从序列操作 (如 count、count_if、find 和 for_each) 到修改操作 (如复制、填充、转换、旋转和反向) 到分区和排序 (分区、排序、nth_element) 等。迭代器是确保它们一般性工作的关键，若每个容器都有不同的方式访问其元素，那么编写泛型算法几乎是不可能的。

这里考虑将元素从一个容器复制到另一个容器的简单操作。例如，有一个 std::vector 对象，希望将其元素复制到 std::list 对象:

```
std::vector<int> v {1, 2, 3};
std::list<int> l;
```

```
3
4 for (std::size_t i = 0; i < v.size(); ++i)
5   l.push_back(v[i]);
```

若想从 std::list 复制到 std::set，或者从 std::set 复制到 std::array 呢? 每种情况都需要不同类型的代码。通用算法使能够以统一的方式完成:

```
1  std::vector<int> v{ 1, 2, 3 };
2
3  // copy vector to vector
4  std::vector<int> vc(v.size());
5  std::copy(v.begin(), v.end(), vc.begin());
6
7  // copy vector to list
8  std::list<int> l;
9  std::copy(v.begin(), v.end(), std::back_inserter(l));
10
11 // copy list to set
12 std::set<int> s;
13 std::copy(l.begin(), l.end(), std::inserter(s, s.begin()));
```

这里有一个 std::vector 对象，将其内容复制到另一个 std::vector 对象，也复制到 std::list 对象。因此，std::list 对象的内容随后复制到 std::set 对象。对于所有情况，都可以使用 std::copy 算法。该算法有几个参数: 两个迭代器定义源的开始和结束，一个迭代器定义目标的开始。该算法通过输出迭代器每次从输入范围复制一个元素到元素指针，然后对输出迭代器进行递增。可以实现的方式如下所示:

```
1  template<typename InputIt, class OutputIt>
2  OutputIt copy(InputIt first, InputIt last,
3          OutputIt d_first)
4  {
5    for (; first != last; (void)++first, (void)++d_first)
6    {
7      *d_first = *first;
8    }
9    return d_first;
10 }
```

重要的 Note

该算法在第 5 章中讨论过，当时我们研究了如何利用类型特征优化其实现。

考虑到前面的例子，有些情况下，目标容器的内容还没有分配以便进行复制。这是复制到列表和复制到集合的情况。类似迭代器的类型 std::back_insert_iterator 和 std:insert_iterator，通过 std::back_inserter 和 std::inserter 辅助函数间接使用，用于将元素插入容器中。std::back_insert_iterator 类使用 push_back 函数，std::insert_iterator 类使用 insert 函数。

C++ 中有六种迭代器类型:

- 输入迭代器

- 输出迭代器
- 前向迭代器
- 双向迭代器
- 随机迭代器
- 连续迭代器

连续迭代器是在 C++17 中添加的。所有操作符都可以使用前缀或后缀增量操作符进行递增。下表显示了每个类别定义的操作:

类别	属性	表达式
输入	单次增加	i++ ++i
	相等/不相等的比较	i == j i != j
	可以解引用 (作为右值)	*i i->m
前向	多次增加	i = j, *j++, *i
双向	可减	–i i–
随机	算术运算符 + 和-	i + n n + i i - n n - i
	不等式比较 (使用迭代器)	i <j i >j i <= j i >= j
	复合赋值	i += n i -= n
	偏移解引用操作符	i[n]
连续	逻辑上相邻的元素，内存物理上相邻。	
输出	单次增加	i++ ++i
	可以解引用 (作为左值)	*i = v *i++ = v

表 8.1

除了输出类别外，每个类别都有相关的内容。所以前向迭代器是输入迭代器，双向迭代器是前向迭代器，随机迭代器是双向迭代器。最后，连续迭代器是随机访问迭代器。前五种类型的迭代器，也可以同时是输出迭代器。这样的迭代器称为可变迭代器。否则，称为常量迭代器。

C++20 标准增加了对概念和概念库的支持。这个标准库为每个迭代器类别定义了标准概念。它们之间的相关性如下表所示:

迭代器类型	概念
输入迭代器	std::input_iterator
前向迭代器	std::output_iterator
双向迭代器	std::bidirectional_iterator
随机迭代器	std::random_access_iterator
连续迭代器	std::contiguous_iterator
输出迭代器	std::output_iterator

表 8.2

所有容器都有以下成员:

- begin: 返回一个指向容器开头的迭代器。
- end: 返回一个指向容器末尾的迭代器。
- cbegin: 返回一个指向容器开头的常量迭代器。
- cend: 返回一个指向容器末端的常量迭代器。

一些容器也有返回反向迭代器的成员:

- rbegin: 返回一个指向反向容器开头的反向迭代器。
- rend: 返回一个指向反向容器末尾的反向迭代器。
- rcbegin: 返回一个指向反向容器开头的常量反向迭代器。
- rcend: 返回一个常量反向迭代器，指向反向容器的末尾。

要使用容器和迭代器，需要理解两点:

- 容器的末尾不是容器的最后一个元素，而是在最后一个元素之后的元素。
- 反向迭代器以相反的顺序提供对元素的访问，指向容器第一个元素的反向迭代器实际上是非反向容器的最后一个元素。

为了更好地理解这两点，来看看下面的例子:

```
std::vector<int> v{ 1,2,3,4,5 };

// prints 1 2 3 4 5
std::copy(v.begin(), v.end(),
      std::ostream_iterator<int>(std::cout, " "));

// prints 5 4 3 2 1
std::copy(v.rbegin(), v.rend(),
      std::ostream_iterator<int>(std::cout, " "));
```

对 std::copy 的第一个调用以给定的顺序打印容器的元素。另一方面，对 std::copy 的第二次调用以相反的顺序打印元素。

对 std::copy 的第一个调用以给定的顺序打印容器的元素。另一方面，对 std::copy 的第二次调用以相反的顺序打印元素。

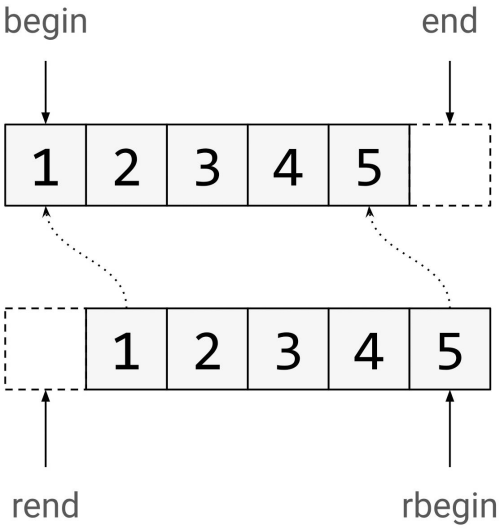


图 8.1

两个迭代器 (begin 和 end，即最后一个元素后面的那个) 分隔的元素序列 (不管它们在内存中存储的是什么样的数据结构) 称为范围。这个术语在 C++ 标准 (尤其是算法) 和文献中被广泛使用，也是 C++20 中范围库的名称，该库将在第 9 章中讨论。

除了标准容器的 begin/end 成员函数集合外，还有同名的独立函数。其等价性如下表所示:

成员函数	独立函数
c.begin() c.cbegin()	std::begin(c) std::cbegin(c)
c.end() c.cend()	std::end(c) std::cend(c)
c.rbegin() c.crbegin()	std::rbegin(c) std::crbegin(c)
c.rend() c.crend()	std::rend(c) std::crend(c)

表 8.3

尽管这些独立函数在使用标准容器时没有带来太多好处，但因为所有这些独立函数对于静态数组都是重载的，其有助于我们编写既可以处理标准容器，又可以处理类 C 数组的泛型代码。这里有一个例子:

```
std::vector<int> v{ 1,2,3,4,5 };
std::copy(std::begin(v), std::end(v),
      std::ostream_iterator<int>(std::cout, " "));

int a[] = { 1,2,3,4,5 };
std::copy(std::begin(a), std::end(a),
```

```
      std::ostream_iterator<int>(std::cout, " "));
```

若没有这些函数，将要写成 std::copy(a, a + 5，…)，也许这些函数的好处是，使我们能够使用基于范围的 for 循环的数组:

```
std::vector<int> v{ 1,2,3,4,5 };
for (auto const& e : v)
  std::cout << e << ' ';

int a[] = { 1,2,3,4,5 };
for (auto const& e : a)
  std::cout << e << ' ';
```

本书的目的不是教你如何使用每个容器或许多标准算法，但学习如何创建容器、迭代器和算法应该是有帮助的。这就是我们接下来要做的。

8.2. 自定义容器和迭代器

理解容器和迭代器如何工作的最好方法是，通过创建自己的容器和迭代器来亲身体验。为了避免实现标准库中已经存在的东西，将考虑一些不同的东西——更准确地说，是循环缓冲区。这是一个容器，当它满时，将覆盖现有的元素。可以想象这样一个容器的不同工作方式; 因此，首先定义它的需求是很重要的。

- 容器应该有一个在编译时已知的固定容量，所以不存在运行时内存管理。
- 容量是容器可以存储的元素数量，大小是容器实际包含的元素数量。当大小等于容量时，容器满了。
- 容器满时，添加一个新元素将覆盖容器中最旧的元素。
- 添加新元素总是在最后完成，删除现有元素总是在开头 (容器中最旧的元素) 完成。
- 可以通过下标操作符和迭代器对容器的元素进行随机访问。

基于这些需求，可以想到以下实现细节:

- 元素可以存储在数组中，这可以是 std::array。
- 需要两个变量 (head 和 tail) 来存储容器的第一个和最后一个元素的索引。因为容器的循环性质，开始和结束会随着时间的推移而变化。
- 第三个变量将存储容器中的元素数量。没有的话，将无法从头和尾索引的值区分容器是空，还是只有一个元素。

> **重要的 Note**
>
> 这里展示的实现仅用于教学目的，并不是用于生产的解决方案。有经验的读者会发现实现的不同方面可以优化。但这里的目的是学习如何编写容器，而不是如何优化实现。

下图显示了这样一个循环缓冲区的可视化表示，容量为八个不同状态的元素:

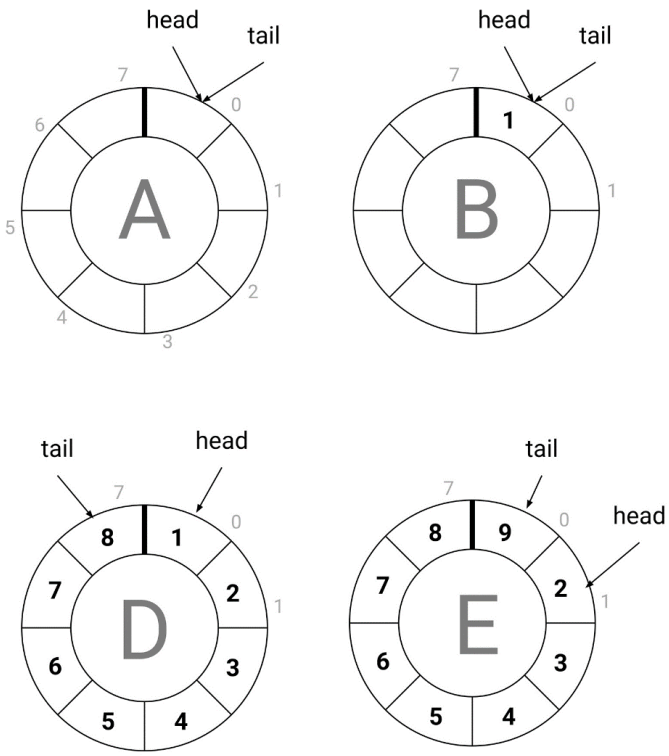


图 8.2

从这张图中可以看到:

- 图 A: 这是一个空缓冲区。容量为 8，大小为 0，头部和尾部都指向索引 0。
- 图 B: 缓冲区包含一个元素。容量仍然是 8，大小是 1，头部和尾部都指向索引 0。
- 图 C: 缓冲区包含两个元素。大小为 2，头部包含索引 0，尾部包含索引 1。
- 图 D: 缓冲区已满。大小为 8，等于容量，头部包含索引 0，尾部包含索引 7。
- 图 E: 缓冲区仍然是满的，但是添加了一个额外的元素，触发了缓冲区中最旧元素的覆盖。大小为 8，头部包含索引 1，尾部包含索引 0。

现在已经了解了循环缓冲区的语义，可以开始编写实现了，先从容器类开始。

8.2.1 实现循环缓冲区容器

容器类的代码太长，所以我们将它分成多个片段。首先是:

```
template <typename T, std::size_t N>
  requires(N > 0)
class circular_buffer_iterator;

template <typename T, std::size_t N>
  requires(N > 0)
class circular_buffer
{
  // ...
};
```

这里有两件事: 一个是类模板 circular_buffer_iterator 的前向声明，另一个是类模板 circular_buffer。两者都有相同的模板实参，一个类型模板形参 T，表示元素的类型，一个非类型模板形参，表示缓冲区的容量。使用约束来确保提供的容量值总是正的。若不支持 C++20，可以用 static_assert 替换这个约束，或者 enable_if 来执行相同的限制。下面的代码段是 circular_buffer 类的一部分。

首先，有一系列成员类型定义，为与 circular_buffer 类模板相关的不同类型提供别名。这些将在类的实现中使用:

```
  public:
  using value_type = T;
  using size_type = std::size_t;
  using difference_type = std::ptrdiff_t;
  using reference = value_type&;
  using const_reference = value_type const&;
  using pointer = value_type*;
  using const_pointer = value_type const*;
  using iterator = circular_buffer_iterator<T, N>;
  using const_iterator =
  circular_buffer_iterator<T const, N>;
```

其次，有存储缓冲区状态的数据成员，元素存储在 std::array 对象中。head、tail 和 size 都存储在 size_type 数据类型的变量中。这些成员都是私有的:

```
private:
  std::array<value_type, N> data_;
  size_type head_ = 0;
  size_type tail_ = 0;
  size_type size_ = 0;
```

第三，有实现前面描述的功能的成员函数。以下所有成员都是公开的。首先是构造函数:

```
constexpr circular_buffer() = default;
constexpr circular_buffer(value_type const (&values)[N]) :
  size_(N), tail_(N-1)
{
  std::copy(std::begin(values), std::end(values),
  data_.begin());
}
constexpr circular_buffer(const_reference v):
  size_(N), tail_(N-1)
{
  std::fill(data_.begin(), data_.end(), v);
}
```

这里定义了三个构造函数 (可以考虑其他的构造函数)，初始化空缓冲区的默认构造函数 (也是默认构造函数)，大小为 N 的类 C 数组的构造函数，通过复制数组元素初始化完整缓冲区，最后是接受单个值的构造函数，通过将该值复制到缓冲区的每个元素来初始化完整缓冲区。这些构造函数允许以以下方式创建循环缓冲区:

```
circular_buffer<int, 1> b1; // {}
circular_buffer<int, 3> b2({ 1, 2, 3 }); // {1, 2, 3}
circular_buffer<int, 3> b3(42); // {42, 42, 42}
```

接下来，定义了几个成员函数来描述循环缓冲区状态:

```
constexpr size_type size() const noexcept
{ return size_; }

constexpr size_type capacity() const noexcept
{ return N; }

constexpr bool empty() const noexcept
{ return size_ == 0; }

constexpr bool full() const noexcept
{ return size_ == N; }

constexpr void clear() noexcept
{ size_ = 0; head_ = 0; tail_ = 0; };
```

size 函数返回缓冲区中的元素数量，capacity 函数返回缓冲区中可以容纳的元素数量，empty 函数检查缓冲区中是否没有元素 (与 size() == 0 相同)，full 函数检查缓冲区是否已满 (与 size() == N 相同)。还有一个名为 clear 的函数将循环缓冲区置于空状态，此函数不销毁任何元素 (不释放内存或调用析构函数)，而只是重置定义缓冲区状态的值。

需要访问缓冲区中的元素，为此定义了以下函数:

```
constexpr reference operator[](size_type const pos)
{
  return data_[(head_ + pos) % N];
}

constexpr const_reference operator[](size_type const pos) const
{
  return data_[(head_ + pos) % N];
}

constexpr reference at(size_type const pos)
{
  if (pos < size_)
    return data_[(head_ + pos) % N];

  throw std::out_of_range("Index is out of range");
}

constexpr const_reference at(size_type const pos) const
{
  if (pos < size_)
    return data_[(head_ + pos) % N];

```

```
  throw std::out_of_range("Index is out of range");
}

constexpr reference front()
{
  if (size_ > 0) return data_[head_];
  throw std::logic_error("Buffer is empty");
}

constexpr const_reference front() const
{
  if (size_ > 0) return data_[head_];
  throw std::logic_error("Buffer is empty");
}

constexpr reference back()
{
  if (size_ > 0) return data_[tail_];
  throw std::logic_error("Buffer is empty");
}

constexpr const_reference back() const
{
  if (size_ > 0) return data_[tail_];
  throw std::logic_error("Buffer is empty");
}
```

每个成员都有一个 const 重载，用于缓冲区的常量实例。常量成员返回一个常量引用，非 const 成员返回一个正常的引用:

- [] 操作符，返回由其索引指定的元素的引用，而不检查索引的值
- at 方法的工作原理与下标操作符类似，但检查索引是否小于大小，若小于则抛出异常
- front 方法，返回对第一个元素的引用；若缓冲区为空，则抛出异常
- back 方法，返回对最后一个元素的引用；若缓冲区为空，则抛出异常

可以使用成员函数来访问元素，也需要成员来向缓冲区中添加和删除元素。添加新元素总是发生在最后，因此称为 push_back。删除现有元素总是发生在最开始 (最旧的元素)，因此称为 pop_front。让我们先来看看前者:

```
constexpr void push_back(T const& value)
{
  if (empty())
  {
    data_[tail_] = value;
    size_++;
  }
  else if (!full())
  {
    data_[++tail_] = value;
    size_++;
```

```
  }
  else
  {
    head_ = (head_ + 1) % N;
    tail_ = (tail_ + 1) % N;
    data_[tail_] = value;
  }
}
```

这基于已定义的需求和图 8.2 中的可视化表示:

- 若缓冲区为空，则将该值复制到 tail_索引所指向的元素，并增加其大小。
- 若缓冲区既不空也不满，执行同样的操作，也增加了 tail_索引的值。
- 若缓冲区已满，则同时增加 head_和 tail_，然后将值复制到 tail_索引所指向的元素。

该函数将值参数复制到缓冲区元素，也可以针对推送到缓冲区后不再需要的临时对象或对象进行优化。因此，提供了接受右值引用的重载。这会将值移动到缓冲区，避免不必要的复制。这个重载显示在下面的代码中:

```
constexpr void push_back(T&& value)
{
  if (empty())
  {
    data_[tail_] = value;
    size_++;
  }
  else if (!full())
  {
    data_[++tail_] = std::move(value);
    size_++;
  }
  else
  {
  head_ = (head_ + 1) % N;
  tail_ = (tail_ + 1) % N;
  data_[tail_] = std::move(value);
  }
}
```

类似的方法用于实现 pop_back 函数来从缓冲区中删除元素:

```
constexpr T pop_front()
{
  if (empty()) throw std::logic_error("Buffer is empty");

  size_type index = head_;

  head_ = (head_ + 1) % N;
  size_--;

  return data_[index];
```

```
}
```

若缓冲区为空，此函数将抛出异常。否则，将 head_索引的值加 1，并返回 head_之前位置的元素的值。下图直观地描述了这一点:

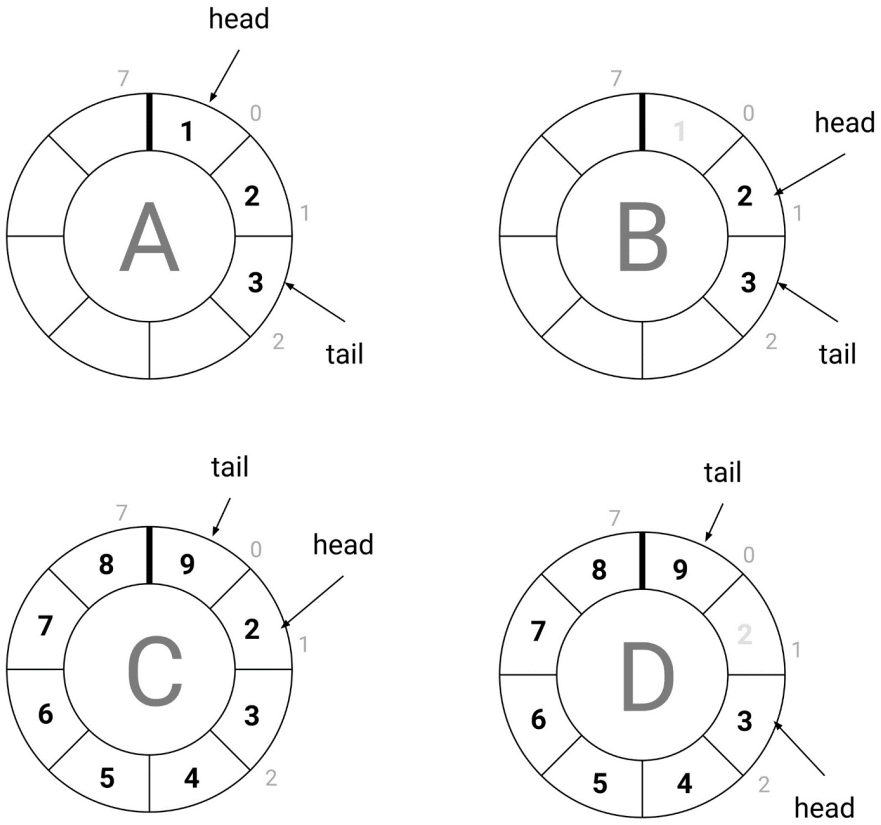


图 8.3

可以看到:

- 图 A: 缓冲区有 3 个元素 (1、2 和 3)，头在索引 0，尾在索引 2。
- 图 B: 前面删除了一个索引为 0 的元素。头部现在是指数 1，尾部仍然是指数 2。缓冲区现在有两个元素。
- 图 C: 缓冲区有 8 个元素，这是其最大容量，一个元素已经覆盖。头部在索引 1，尾部在索引 0。
- 图 D: 前面删除了一个元素，即索引 1。头部现在在索引 2，尾部仍然在索引 0。缓冲区现在有 7 个元素。

下面的代码展示了同时使用 push_back 和 pop_front 的例子:

```
circular_buffer<int, 4> b({ 1, 2, 3, 4 });
assert(b.size() == 4);

b.push_back(5);
b.push_back(6);
b.pop_front();

assert(b.size() == 3);
assert(b[0] == 4);
assert(b[1] == 5);
```

```
assert(b[2] == 6);
```

最后，成员函数 begin 和 end 返回指向缓冲区中第一个和倒数第一个元素的迭代器:

```
iterator begin()
{
  return iterator(*this, 0);
}

iterator end()
{
  return iterator(*this, size_);
}

const_iterator begin() const
{
  return const_iterator(*this, 0);
}

const_iterator end() const
{
  return const_iterator(*this, size_);
}
```

为了理解这些，需要了解迭代器类实际上是如何实现的。我们将在下一节中对此进行探讨。

8.2.2 为循环缓冲区容器实现迭代器类型

上一节开始使用 circular_buffer 容器时声明了迭代器类模板，但也需要定义它的实现。然而，还有一件事必须做: 为了使迭代器类能够访问容器的私有成员，需要声明为友元:

```
private:
  friend circular_buffer_iterator<T, N>;
```

来看看 circular_buffer_iterator 类，与容器类有相似之处。这包括模板参数、约束和成员类型集(其中一些与 circular_buffer 中的成员类型相同):

```
template <typename T, std::size_t N>
requires(N > 0)
class circular_buffer_iterator
{
public:
  using self_type = circular_buffer_iterator<T, N>;
  using value_type = T;
  using reference = value_type&;
  using const_reference = value_type const &;
  using pointer = value_type*;
  using const_pointer = value_type const*;
  using iterator_category =
    std::random_access_iterator_tag;
  using size_type = std::size_t;
```

```
  using difference_type = std::ptrdiff_t;
public:
  /* definitions */

private:
  std::reference_wrapper<circular_buffer<T, N>> buffer_;
  size_type index_ = 0;
};
```

circular_buffer_iterator 类有一个对循环缓冲区的引用，以及其所指向的缓冲区中元素的索引。circular_buffer<T, N> 的引用包装在 std::reference_wrapper 对象中。可以通过提供这两个参数显式地创建这样的迭代器，构造函数如下所示:

```
explicit circular_buffer_iterator(
  circular_buffer<T, N>& buffer,
  size_type const index):
  buffer_(buffer), index_(index)
{ }
```

若现在回顾 circular_buffer 的 begin 和 end 成员函数的定义，可以看到第一个参数是 *this，第二个参数是 0 (begin 迭代器)，size_(end 迭代器)。第二个值是迭代器指向的元素头部的偏移量。因此，0 是第一个元素，而 size_是缓冲区中最后一个元素。

这里，需要随机访问缓冲区的元素，所以迭代器类别是随机迭代器。成员类型 iterator_category 是 std::random_access_iterator_tag 的别名。因此，需要提供这样一个迭代器所支持的所有操作。本章的前一节中，讨论了迭代器类别，以及每个类别所需的操作。接下来我们将逐一进行实现。

从输入迭代器开始:

```
self_type& operator++()
{
  if(index_ >= buffer_.get().size())
    throw std::out_of_range("Iterator cannot be
                incremented past the end of the range");

  index_++;
  return *this;
}

self_type operator++(int)
{
  self_type temp = *this;
  ++*this;
  return temp;
}

bool operator==(self_type const& other) const
{
  return compatible(other) && index_ == other.index_;
}

```

```
bool operator!=(self_type const& other) const
{
  return !(*this == other);
}

const_reference operator*() const
{
  if (buffer_.get().empty() || !in_bounds())
    throw std::logic_error("Cannot dereferentiate the
  iterator");
  return buffer_.get().data_[
    (buffer_.get().head_ + index_) %
     buffer_.get().capacity()];
}

const_reference operator->() const
{
  if (buffer_.get().empty() || !in_bounds())
    throw std::logic_error("Cannot dereferentiate the
                iterator");

  return buffer_.get().data_[
    (buffer_.get().head_ + index_) %
     buffer_.get().capacity()];
}
```

这里实现了递增 (前缀和后缀)、检查相等/不相等，以及解引用。* 和-> 操作符在无法解引用元素时抛出异常。这种情况发生在缓冲区为空时，或者索引不在边界内 (在 head_和 tail_之间)。使用了两个辅助函数 (都是私有的)，分别称为 compatible 和 is_bounds:

```
bool compatible(self_type const& other) const
{
  return buffer_.get().data_.data() ==
       other.buffer_.get().data_.data();
}

bool in_bounds() const
{
  return
    !buffer_.get().empty() &&
    (buffer_.get().head_ + index_) %
     buffer_.get().capacity() <= buffer_.get().tail_;
}
```

前向迭代器是输入迭代器，同时也是输出迭代器。之前看到的那些用于输入迭代器的迭代器，因为对可解引用的前向迭代器执行操作会使其迭代器值解引用，所以前向迭代器可以用于多遍算法。若 a 和 b 是两个前向迭代器并且相等，要么不可解引用；否则，其迭代器值 *a 和 *b 指向同一个对象。反之亦然，若 *a 和 *b 相等，那么 a 和 b 也相等。这对于我们的实现来说是正确的。

前向迭代器的另一个要求是是可交换的。若 a 和 b 是两个前向迭代器，那么 swap(a, b) 应该是

一个有效的操作。回到使用 std::reference_wrapper 对象来保存对循环_buffer<T, N> 的引用。引用是不可交换的，所以 circular_buffer_iterator 不可交换。然而，std::reference_wrapper 可交换，这也使得我们的迭代器类型可交换。可以使用 static_assert 进行验证:

```
static_assert(
  std::is_swappable_v<circular_buffer_iterator<int, 10>>);
```

重要的 Note

std::reference_wrapper 的另一种替代方法是使用指向 circular_buffer 类的原始指针，因为指针可以赋值，因此是可交换的。使用哪一种就看开发者的开发风格和个人喜好。本例中，我倾向于避免使用原始指针的解决方案。

为了满足双向迭代器类别的要求，需要支持递减。下一个代码段中，可以看到前缀和后缀递减操作符的实现:

```
self_type& operator--()
{
  if(index_ <= 0)
    throw std::out_of_range("Iterator cannot be
      decremented before the beginning of the range");

  index_--;
  return *this;
}

self_type operator--(int)
{
  self_type temp = *this;
  --*this;
  return temp;
}
```

最后，因为需要实现随机迭代器。首先要实现的需求是算术 (+ 和-) 和复合 (+= 和-=) 操作:

```
self_type operator+(difference_type offset) const
{
  self_type temp = *this;
  return temp += offset;
}

self_type operator-(difference_type offset) const
{
  self_type temp = *this;
  return temp -= offset;
}

difference_type operator-(self_type const& other) const
{
  return index_ - other.index_;
```

```
}

self_type& operator +=(difference_type const offset)
{
  difference_type next =
    (index_ + next) % buffer_.get().capacity();
  if (next >= buffer_.get().size())
    throw std::out_of_range("Iterator cannot be
                incremented past the bounds of the range");

  index_ = next;
  return *this;
}

self_type& operator -=(difference_type const offset)
{
  return *this += -offset;
}
```

随机访问迭代器必须支持与其他操作的不相等比较，需要重载 <、<=、> 和 >= 操作符，但 <=、> 和 >= 操作符可以基于 < 操作符实现。其定义如下所示:

```
bool operator<(self_type const& other) const
{
  return index_ < other.index_;
}

bool operator>(self_type const& other) const
{
  return other < *this;
}

bool operator<=(self_type const& other) const
{
  return !(other < *this);
}

bool operator>=(self_type const& other) const
{
  return !(*this < other);
}
```

最后，但同样重要的是，需要使用下标操作符 ([]) 提供对元素的访问。可能的实现如下所示:

```
value_type& operator[](difference_type const offset)
{
  return *((*this + offset));
}

value_type const & operator[](difference_type const offset)
const
```

```
{
  return *((*this + offset));
}
```

至此，我们已经完成了循环缓冲区迭代器类型的实现。若在理解这两个类的大量代码片段时遇到困难，可以在本书的 GitHub 存储库中找到完整实现。下面是使用迭代器类型的简单示例:

```
circular_buffer<int, 3> b({1, 2, 3});
std::vector<int> v;
for (auto it = b.begin(); it != b.end(); ++it)
{
  v.push_back(*it);
}
```

这段代码实际上可以通过基于范围的 for 循环来简化。这里，我们不直接使用迭代器，但编译器生成的代码可以。因此，下面的代码段相当于前面的代码段:

```
circular_buffer<int, 3> b({ 1, 2, 3 });
std::vector<int> v;
for (auto const e : b)
{
  v.push_back(e);
}
```

但这里为 circular_buffer_iterator 提供的实现，不能使以下代码段编译通过:

```
circular_buffer<int, 3> b({ 1,2,3 });
*b.begin() = 0;

assert(b.front() == 0);
```

这要求我们能够通过迭代器写入元素，但我们的实现不满足输出迭代器类别的要求。这要求表达式如 *it = v，或 *it++ = v 是有效的。为此，需要提供返回非 const 引用类型的 * 和-> 操作符的非 const 重载方式:

```
reference operator*()
{
  if (buffer_.get().empty() || !in_bounds())
    throw std::logic_error("Cannot dereferentiate the
                iterator");

  return buffer_.get().data_[
    (buffer_.get().head_ + index_) %
     buffer_.get().capacity()];
}

reference operator->()
{
  if (buffer_.get().empty() || !in_bounds())
    throw std::logic_error("Cannot dereferentiate the
                iterator");
```

```
17
18    return buffer_.get().data_[
19      (buffer_.get().head_ + index_) %
20       buffer_.get().capacity()];
21  }
```

GitHub 库中可以找到更多带迭代器和不带迭代器的 circular_buffer 类的示例。接下来，我们将把注意力集中在实现一个适用于任何范围的通用算法上，包括在这里定义的 circular_buffer 容器。

8.3. 自定义通用算法

本章的第一节中，了解了为什么使用迭代器抽象对容器元素的访问是构建通用算法的关键。但练习编写这样的算法是有用的，有助于更好地理解迭代器的使用。所以在本节中，我们将编写一个通用算法。

标准库具有许多这样的算法，但其中缺失的就有 zip 算法。实际上，不同的人对 zip 的理解是不同的。有些人认为，zip 是获取两个或多个输入范围，并用插入输入范围中的元素创建一个新范围。如下图所示:

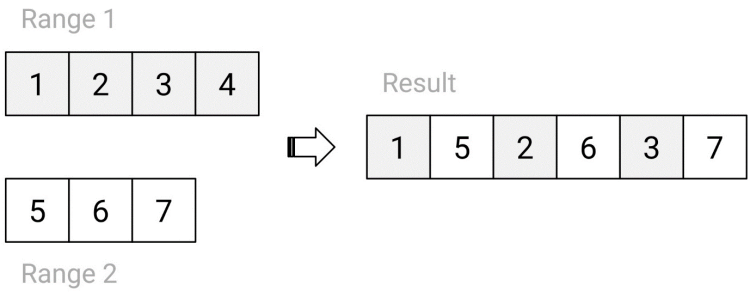


图 8.4

另一些人认为，zip 是获取两个或多个输入范围并创建一个新范围，其中元素是由输入范围的元素组成的元组。如下图所示:

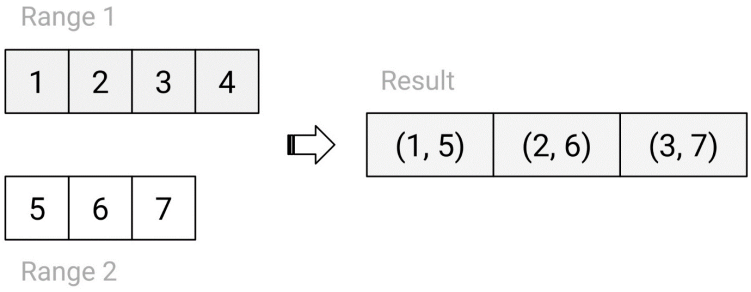


图 8.5

本节中，我们将实现第一个算法。为了避免混淆，将其称为 flatzip。以下是其要求:

- 该算法接受两个输入范围并写入一个输出范围。
- 算法以迭代器作为参数。对第一个和最后一个输入迭代器定义了每个输入范围的边界，输出迭代器定义了将写入元素的输出范围的开始位置。

- 两个输入范围应该包含相同类型的元素。输出范围必须具有相同类型的元素，或者输入类型可隐式转换为的类型。
- 若两个输入范围的大小不同，当两个输入范围中最小的一个被处理时，算法就会停止 (如前图所示)。
- 返回值是复制倒数第一元素的输出迭代器。

所描述的算法的可能实现如下所示:

```
template <typename InputIt1, typename InputIt2,
      typename OutputIt>
OutputIt flatzip(
  InputIt1 first1, InputIt1 last1,
  InputIt2 first2, InputIt2 last2,
  OutputIt dest)
{
  auto it1 = first1;
  auto it2 = first2;

  while (it1 != last1 && it2 != last2)
  {
    *dest++ = *it1++;
    *dest++ = *it2++;
  }

  return dest;
}
```

实现非常简单。这里所做的只是同时遍历两个输入范围，并交替地从它们复制元素到目标范围。当到达最小范围的终点时，两个输入范围上的迭代停止。可以这样使用算法:

```
// one range is empty
std::vector<int> v1 {1,2,3};
std::vector<int> v2;
std::vector<int> v3;

flatzip(v1.begin(), v1.end(), v2.begin(), v2.end(),
    std::back_inserter(v3));
assert(v3.empty());

// neither range is empty
std::vector<int> v1 {1, 2, 3};
std::vector<int> v2 {4, 5};
std::vector<int> v3;

flatzip(v1.begin(), v1.end(), v2.begin(), v2.end(),
    std::back_inserter(v3));
assert(v3 == std::vector<int>({ 1, 4, 2, 5 }));
```

这些例子对输入和输出范围都使用 std::vector，但 flatzip 算法对容器一无所知。容器的元素可以在迭代器的帮助下访问，所以只要迭代器满足指定的要求，就可以使用任何容器。这包括我们之

前编写的 circular_buffer 容器，因为 circular_buffer_container 同时满足输入和输出迭代器类别的要求。还可以编写如下代码:

```
circular_buffer<int, 4> a({1, 2, 3, 4});
circular_buffer<int, 3> b({5, 6, 7});
circular_buffer<int, 8> c(0);

flatzip(a.begin(), a.end(), b.begin(), b.end(), c.begin());

std::vector<int> v;
for (auto e : c)
  v.push_back(e);
assert(v == std::vector<int>({ 1, 5, 2, 6, 3, 7, 0, 0 }));
```

这里有两个输入循环缓冲区:a 有四个元素，b 有三个元素。目标循环缓冲区的容量为 8 个元素，全部初始化为 0。应用 flatzip 算法之后，目标循环缓冲区的六个元素将使用 a 和 b 缓冲区中的值写入。结果是循环缓冲区将包含元素 1,5,2,6,3,7,0,0。

8.4. 总结

本章讨论如何使用模板构建通用库。虽然我们无法详细讨论这些主题，但已经探讨了 C++ 标准库中的容器、迭代器和算法的设计，这些是标准库的核心。本章的大部分内容都用于理解如何编写与标准容器类似的容器，以及提供对其元素的访问的迭代器类。为此，我们实现了一个表示循环缓冲区的类，这是一种固定大小的数据结构，当容器满了，元素就会进行覆盖。最后，实现了一个通用算法，从两个范围压缩元素。这适用于任何容器，包括循环缓冲区容器。

如本章所述，范围是一个抽象的概念。这种情况在 C++20 中发生了改变，通过新的范围库引入了更具体的范围概念。这就是我们将在本书最后一章讨论的内容。

8.5. 习题

1. 标准库中的序列容器是什么?
2. 标准容器中定义的公共成员函数是什么?
3. 什么是迭代器，存在多少个类别?
4. 随机迭代器支持哪些操作?
5. 范围访问函数是什么?

第 9 章 范围库

前一章致力于理解标准库的三个主要核心: 容器、迭代器和算法。我们使用了抽象的范围概念，来表示由两个迭代器分隔的元素序列。C++20 标准提供了一个范围库，使范围变得更容易使用，其主要由两部分组成: 一方面，定义非归属范围和自适应范围的类型；另一方面，处理这些范围类型的算法，这些算法不需要迭代器来定义元素范围。

最后一章中，将讨论以下主题:

- 从抽象范围到范围库
- 范围和视图
- 约束算法
- 编写范围适配器

本章结束时，读者们将对范围库的内容有一个很好的理解，您将能够编写自己的范围适配器。

让我们开始吧！

9.1. 从抽象范围到范围库

前一章中，已经多次使用范围这个术语。范围是一个元素序列的抽象，由两个迭代器分隔 (一个指向序列的第一个元素，一个指向最后一个元素)。容器如 std::vector、std::list 和 std::map 都是范围抽象的具体实现，其拥有元素的所有权，并且使用各种数据结构 (如数组、链表或树) 实现。标准算法是通用的，可以对 std::vector、std::list 或 std::map 的内部实现一无所知，在迭代器的帮助下可以进行范围抽象，而这有一个缺点: 需要从容器中检索开始和结束迭代器:

```
// sorts a vector
std::vector<int> v{ 1, 5, 3, 2, 4 };
std::sort(v.begin(), v.end());

// counts even numbers in an array
std::array<int, 5> a{ 1, 5, 3, 2, 4 };
auto even = std::count_if(
  a.begin(), a.end(),
  [](int const n) {return n % 2 == 0; });
```

极少数情况下，只需要处理容器元素的一部分。大多数情况下，需要一遍遍地写 v.begin() 和 v.end()，包括对 cbegin()/cend()、rbegin()/rend() 或独立函数 std::begin()/std::end() 的调用。理想情况下，我们希望能够编写如下代码:

```
// sorts a vector
std::vector<int> v{ 1, 5, 3, 2, 4 };
sort(v);

// counts even numbers in an array
std::array<int, 5> a{ 1, 5, 3, 2, 4 };
auto even = std::count_if(
  a,
  [](int const n) {return n % 2 == 0; });
```

另一方面，我们经常需要组合操作。即使使用标准算法，也会涉及许多操作和过于冗长的代码。考虑下面的例子: 给定一个整数序列，希望将除前两个以外的所有偶数的平方按它们的值降序 (而不是在序列中的位置) 输出到控制台。有多种方法可以解决这个问题，以下是一个可能的解决方案:

```
std::vector<int> v{ 1, 5, 3, 2, 8, 7, 6, 4 };

// copy only the even elements
std::vector<int> temp;
std::copy_if(v.begin(), v.end(),
      std::back_inserter(temp),
      [](int const n) {return n % 2 == 0; });

// sort the sequence
std::sort(temp.begin(), temp.end(),
    [](int const a, int const b) {return a > b; });

// remove the first two
temp.erase(temp.begin() + temp.size() - 2, temp.end());

// transform the elements
std::transform(temp.begin(), temp.end(),
        temp.begin(),
        [](int const n) {return n * n; });

// print each element
std::for_each(temp.begin(), temp.end(),
        [](int const n) {std::cout << n << '\n'; });
```

熟悉标准算法的开发者可以轻松地阅读这段代码，但仍然有很多东西需要写，还需要一个临时容器和重复的开始/结束调用。因此，我想大多数人会更容易理解以下版本的代码，并且喜欢这样进行书写:

```
std::vector<int> v{ 1, 5, 3, 2, 8, 7, 6, 4 };
sort(v);
auto r = v
    | filter([](int const n) {return n % 2 == 0; })
    | drop(2)
    | reverse
    | transform([](int const n) {return n * n; });

for_each(r, [](int const n) {std::cout << n << '\n'; });
```

这是 C++20 标准在范围库的帮助下提供的功能。这有两个主要组成部分:

- 视图或范围适配器，表示非拥有的可迭代序列。使我们能够更容易地组合操作，例如在最后一个示例中。
- 约束算法，能够操作具体的范围 (标准容器或范围)，不是使用一对迭代器分隔的抽象范围 (尽管这也是可能的)。

下一节中，我们将探讨范围库的这两种功能，先从范围开始。

9.2. 范围和视图

术语范围是指，由开始迭代器和结束迭代器限定的元素序列的抽象，所以范围表示元素的可迭代序列。这样的序列可以用几种方式定义:

- 一个开始迭代器和一个结束哨兵，这样的序列从开始迭代到结束。哨兵是表示序列结束的对象。可以具有与迭代器类型相同的类型，也可以具有不同的类型。
- 一个开始对象和一个大小 (元素的数量)，表示一个所谓的计数序列。这样的序列从一开始就要迭代 N 次 (其中 N 表示大小)。
- 有一个开始和一个谓词，表示所谓的有条件终止序列。这样的序列从开始迭代，直到谓词返回 false 为止。
- 只有一个开始值，表示所谓的无界序列。这样的序列可以无限迭代。

所有这些类型的可迭代序列都是范围。因为范围是一种抽象，C++20 标准库定义了一系列概念来描述范围类型的需求，可以在 <ranges> 头文件和 std::ranges 命名空间中使用。下表列出了范围概念的列表:

名称	描述
range	通过提供开始迭代器和结束哨兵，定义类型 R 为范围的需求。 迭代器和哨兵可以是不同的类型。
bgorrowed_range	定义 R 类型的要求，以便函数可以按值接受该类型的对象 并返回从该对象获得的迭代器，而不会出现悬空的危险。
sized_range	将类型 R 的要求定义为在常数时间内知道其大小的范围。
common_range	将类型 R 的要求定义为迭代器类型和哨兵类型相同的范围。
view	定义类型 R 的需求，该类型 R 是一个具有固定时间 复制、移动和赋值操作的范围。
viewable_range	定义可转换为视图的范围类型 R 的需求。
input_range	要求范围类型具有满足 input_iterator 概念的迭代器类型。
output_range	要求范围类型具有满足 output_iterator 概念的迭代器类型。
forward_range	要求范围类型具有满足 forward_iterator 概念的迭代器类型。
bidirectional_range	要求范围类型具有满足 bidirectional_iterator 概念的迭代器类型。
random_access_range	要求范围类型具有满足 random_access_iterator 概念的迭代器类型。
contiguous_range	要求范围类型具有满足 contiguous_iterator 概念的迭代器类型。

表 9.1

标准库为容器和数组定义了一组访问函数。这包括 std::begin 和 std::end 代替成员函数 begin 和 end, std::size 代替成员函数 size 等，这些称为范围访问函数。类似地，范围库定义了一组范围访问函数。其为范围设计的，可以在 <ranges> 和 <iterator> 头文件和 std::ranges 命名空间中使用。下表列出了它们:

Rnage 的访问范围	容器/数组的等效范围访问	描述
begin/end cbegin/cend	begin/end cbegin/cend	返回一个迭代器和一个常量迭代器，分别指向范围的开始/结束。
rbegin/rend crbegin/crend	rbegin/rend crbegin/crend	分别返回一个指向范围开始/结束的反向迭代器和常量反向迭代器。
size/ssize	size/ssize	将范围的大小返回为整数或带符号的整数值。
empty	empty	返回一个布尔值，指示范围是否为空。
data/cdata	data	分别返回指向连续范围和只读连续范围开头的指针。

表 9.2

下面的代码段演示了其中一些函数的使用:

```
std::vector<int> v{ 8, 5, 3, 2, 4, 7, 6, 1 };
auto r = std::views::iota(1, 10);

std::cout << "size(v)=" << std::ranges::size(v) << '\n';
std::cout << "size(r)=" << std::ranges::size(r) << '\n';

std::cout << "empty(v)=" << std::ranges::empty(v) << '\n';
std::cout << "empty(r)=" << std::ranges::empty(r) << '\n';

std::cout << "first(v)=" << *std::ranges::begin(v) << '\n';
std::cout << "first(r)=" << *std::ranges::begin(r) << '\n';

std::cout << "rbegin(v)=" << *std::ranges::rbegin(v)
      << '\n';
std::cout << "rbegin(r)=" << *std::ranges::rbegin(r)
      << '\n';

std::cout << "data(v)=" << *std::ranges::data(v) << '\n';
```

这段代码中，使用了一个名为 std::views::iota 的类型。正如命名空间所示，这是一个视图。视图是带有限制的范围，视图是具有非所有语义的轻量级对象，以一种不需要复制或改变序列的方式呈现底层元素序列 (范围) 的视图。关键特征是惰性求值。所以不管它们应用了什么转换，只在请求 (迭代) 元素时执行，而不是在创建时执行。

C++20 中提供了一系列视图，C++23 中也包含了新视图。视图可以在 <ranges> 头文件和 std::ranges 命名空间的形式中使用，std::ranges::abc_view，例如 std::ranges::iota_view。为了使用方便，在 std::views 命名空间中，还存在一个形式为 std::views::abc 的变量模板，例如 std::views::iota。这就是我们在前面的例子中看到的。下面是两个使用 iota 的等效示例:

```
// using the iota_view type
for (auto i : std::ranges::iota_view(1, 10))
  std::cout << i << '\n';
```

```
4
5  // using the iota variable template
6  for (auto i : std::views::iota(1, 10))
7    std::cout << i << '\n';
```

iota 视图是工厂视图的一部分。这些工厂视图是新生成范围的视图，范围库中有以下工厂视图:

类型	变量	描述
ranges::empty_view	ranges::views::empty	生成不包含 T 类型元素的视图。
ranges::single_view	ranges::views::single	生成具有单个 T 类型元素的视图。
ranges::iota_view	ranges::views::iota	生成一个连续元素序列的视图，从开始值到结束值 (有界视图) 或无限 (无界视图)。
ranges::basic_iostream_view	ranges::views::istream	通过反复应用操作符 >> 生成一个元素序列的视图。

表 9.3

为什么 empty_view 和 single_view 有用，答案应该不难找到。模板代码中，可以处理空范围或只有一个元素的范围是有效输入的范围。一个函数模板不用多次重载来处理这些特殊情况；反之，可以接受一个 empty_view 或 single_view 范围。下面的代码使用工厂视图的几个示例:

```
1  constexpr std::ranges::empty_view<int> ev;
2  static_assert(std::ranges::empty(ev));
3  static_assert(std::ranges::size(ev) == 0);
4  static_assert(std::ranges::data(ev) == nullptr);
5
6  constexpr std::ranges::single_view<int> sv{42};
7  static_assert(!std::ranges::empty(sv));
8  static_assert(std::ranges::size(sv) == 1);
9  static_assert(*std::ranges::data(sv) == 42);
```

对于 iota_view，已经看过一些有界视图的例子。下面的代码段再次展示了一个例子，不仅使用了 iota 生成的有界视图，还使用了同样由 iota 生成的无界视图:

```
1  auto v1 = std::ranges::views::iota(1, 10);
2  std::ranges::for_each(
3    v1,
4    [](int const n) {std::cout << n << '\n'; });
5
6  auto v2 = std::ranges::views::iota(1) |
7       std::ranges::views::take(9);
8  std::ranges::for_each(
9    v2,
10   [](int const n) {std::cout << n << '\n'; });
```

最后一个示例中，使用了另一个名为 take_view 的视图。这将生成另一个视图 (我们的示例中是 iota 生成的无界视图) 的前 N 个元素 (我们的示例中是 9) 的视图。首先，举一个使用第四个视图工厂 basic_iostream_view 的例子。假设在文本中有一个商品价格列表，用空格分隔，需要输出这些价格的总和。有不同的解决方法，但这里给出了一个可能的解决方案:

```
auto text = "19.99 7.50 49.19 20 12.34";
auto stream = std::istringstream{ text };
std::vector<double> prices;

double price; // 高亮显示
while (stream >> price) // 高亮显示
{ // 高亮显示
  prices.push_back(price); // 高亮显示
} // 高亮显示

auto total = std::accumulate(prices.begin(), prices.end(),
                0.0);
std::cout << std::format("total: {}\n", total);
```

高亮显示的部分可以用以下两行代码替换，使用 basic_iostream_view，或者更准确地说，使用 istream_view 别名模板:

```
for (double const price :
    std::ranges::istream_view<double>(stream))
{
  prices.push_back(price);
}
```

istream_view 范围工厂所做的是在 istringstream 对象上重复应用操作符 >>，并在每次应用时产生一个值。不能指定分隔符，只适用于空格。若喜欢使用标准算法，可以使用 range::for_each 约束算法来产生相同的结果:

```
std::ranges::for_each(
  std::ranges::istream_view<double>(stream),
  [&prices](double const price) {
    prices.push_back(price); });
```

目前为止，本章给出的示例包括 filter, take, drop 和 reverse 等视图。这些只是 C++20 中可用的一些标准视图，C++23 中会添加了更多。下表列出了整个标准视图集:

类型 (ranges 命名空间中)	变量 (ranges::view 命名空间中)	C++ 版本	描述
fileter_view	fileter	C++20	表示范围适配器的类型，该范围适配器提供下级范围的视图，该范围仅包括满足谓词的元素。
transform_ view	transform	C++20	一种类型，表示提供底层范围视图的范围适配器，并将转换应用到范围的每个元素。
split_view	split	C++20	表示范围适配器的类型，该类型提供通过在指定分隔符上分割范围而产生的范围序列的视图。该范围不能是输入范围，并且视图的惰性语义不会被观察到。
lazy_split_ view	lazy_split	C++20	与 split_view 相同，还适用于输入范围，并观察范围的惰性机制。
reverse_view	reverse	C++20	一种类型，表示范围适配器，该适配器以相反的顺序提供基础范围元素的视图。
keys_view	keys	C++20	表示范围适配器的类型，提供从底层视图的类元组值 (std::pari 和 std::tuple) 的第一个元素投影的视图。
values_view	values	C++20	表示范围适配器的类型，提供从底层视图的类元组值 (std::pair 和 std::tuple) 的第二个元素投影出来的视图。
elements_view	elements	C++20	表示范围适配器的类型，该范围适配器提供从非视图类元组值的第 n 个元素投影的视图。
zip_view	zip	C++23	一种表示范围适配器的类型，提供一个由一个或多个底层视图构建的视图，将每个视图的第 n 个元素投影到一个元组中。
zip_ transform_ view	zip_ transform	C++23	一种类型，表示一个范围适配器，该适配器提供一个由一个或多个底层视图和一个可调用对象构建的视图，其元素是通过将可调用对象应用于每个底层视图的第 n 个元素来计算的。
adjacent_ view	adjacent	C++23	一种表示范围适配器的类型，该范围适配器提供类似元组值的视图，通过获取底层视图的 N 个连续元素进行投影。
adjacent_ transform_ view	adjacent_ transform	C++23	表示范围适配器的类型，该类型通过将可调用对象应用到底层视图的 N 个连续元素来提供投影值的视图。

表 9.4

除了上表中列出的视图 (范围适配器) 之外，还有一些在某些特定场景中可能有用的视图。为了

完整起见，下表将列出了这些参数:

类型 (ranges 命名空间中)	变量 (ranges::view 命名空间中)	C++ 版本	描述
	all	C++20	一个对象，创建一个包含范围参数的所有元素的视图。
	all_t	C++20	可以安全转换为视图的范围的视图类型的别名模板。
	counted	C++20	一个对象，创建一个包含 N 个范围元素的视图，从给定迭代器所表示的元素开始。
ref_view		C++20	将引用封装到另一个范围的视图类型。
owning_view		C++20	存储给定范围的视图类型，拥有存储范围和移动语义的唯一所有权。
common_view	common	C++20	一种对迭代器和哨兵类型对采用不同类型的视图的类型，对迭代器和哨兵类型使用相同类型的视图。

表 9.5

我们已经列举了所有标准范围适配器，来看一下使用适配器的更多示例。

9.2.1 更多的例子

本节之前，我们看到了以下示例 (使用显式命名空间):

```
namespace rv = std::ranges::views;
std::ranges::sort(v);
auto r = v
    | rv::filter([](int const n) {return n % 2 == 0; })
    | rv::drop(2)
    | rv::reverse
    | rv::transform([](int const n) {return n * n; });
```

这实际上是以下内容的短而易读版本:

```
std::ranges::sort(v);auto r =
  rv::transform(
    rv::reverse(
      rv::drop(
        rv::filter(
          v,
          [](int const n) {return n % 2 == 0; }),
          2)),
    [](int const n) {return n * n; });
```

第一个版本是可能的，因为管道操作符 (|) 可重载，以更易于阅读的形式简化视图的组合。有些范围适配器接受一个参数，有些可能接受多个参数。适用规则如下:

- 若范围适配器 a 有一个参数，一个视图 V，那么 a(V) 和 v| a 是等价的。这样的范围适配器是 reverse_view:

```
std::vector<int> v{ 1, 5, 3, 2, 8, 7, 6, 4 };
namespace rv = std::ranges::views;
auto r1 = rv::reverse(v);
auto r2 = v | rv::reverse;
```

- 若一个范围适配器 a 有多个参数，一个视图 V 和 args…，那么 a (V, args…)，a (args…)(V) 和 V| a (args…) 是等价的。这样的范围适配器是 take_view:

```
std::vector<int> v{ 1, 5, 3, 2, 8, 7, 6, 4 };
namespace rv = std::ranges::views;
auto r1 = rv::take(v, 2);
auto r2 = rv::take(2)(v);
auto r3 = v | rv::take(2);
```

已经看过滤器、变换、反转和删除的例子。下面，我们会通过一系列例子来演示表 8.7 中的视图的使用。以下所有示例中，我们将把 rv 视为 std::ranges::views 命名空间的别名:

- 输出序列中最后两个奇数，顺序相反:

```
std::vector<int> v{ 1, 5, 3, 2, 4, 7, 6, 8 };
for (auto i : v |
  rv::reverse |
  rv::filter([](int const n) {return n % 2 == 1; }) |
  rv::take(2))
{
  std::cout << i << '\n'; // prints 7 and 3
}
```

- 从不包括第一个连续奇数的范围中打印小于 10 的连续数的子序列:

```
std::vector<int> v{ 1, 5, 3, 2, 4, 7, 16, 8 };
for (auto i : v |
  rv::take_while([](int const n){return n < 10; }) |
  rv::drop_while([](int const n){return n % 2 == 1; })
) {
  std::cout << i << '\n'; // prints 2 4 7
}
```

- 分别输出元组序列中的第一个元素、第二个元素和第三个元素:

```
std::vector<std::tuple<int,double,std::string>> v =
{
  {1, 1.1, "one"},
  {2, 2.2, "two"},
  {3, 3.3, "three"}
};
```

```

for (auto i : v | rv::keys)
  std::cout << i << '\n'; // prints 1 2 3

for (auto i : v | rv::values)
  std::cout << i << '\n'; // prints 1.1 2.2 3.3

for (auto i : v | rv::elements<2>)
  std::cout << i << '\n'; // prints one two three
```

- 输出一个由整数 vector 中的所有元素:

```
std::vector<std::vector<int>> v {
  {1,2,3}, {4}, {5, 6}
};

for (int const i : v | rv::join)
  std::cout << i << ' '; // prints 1 2 3 4 5 6
```

- 输出由整数 vector 中的所有元素，但在每个 vector 的元素之间插入一个 0。范围适配器 join_with 是 C++23 的新功能，编译器可能还不支持:

```
std::vector<std::vector<int>> v{
  {1,2,3}, {4}, {5, 6}
};
for(int const i : v | rv::join_with(0))
  std::cout << i << ' '; // print 1 2 3 0 4 0 5 6
```

- 输出句子中的单个单词，其中分隔符为空格:

```
std::string text{ "this is a demo!" };
constexpr std::string_view delim{ " " };
for (auto const word : text | rv::split(delim))
{
  std::cout << std::string_view(word.begin(),
                 word.end())
       << '\n';
}
```

- 从一个整数数组的元素和一个双精度 vector 创建元组视图:

```
std::array<int, 4> a {1, 2, 3, 4};
std::vector<double> v {10.0, 20.0, 30.0};

auto z = rv::zip(a, v)
// { {1, 10.0}, {2, 20.0}, {3, 30.0} }
```

- 创建包含整数数组的相乘元素和双精度 vector 的视图:

```
std::array<int, 4> a {1, 2, 3, 4};
std::vector<double> v {10.0, 20.0, 30.0};

auto z = rv::zip_transform(
```

```
  std::multiplies<double>(), a, v)
// { {1, 10.0}, {2, 20.0}, {3, 30.0} }
```

- 输出整数序列的相邻元素对:

```
std::vector<int> v {1, 2, 3, 4};
for (auto i : v | rv::adjacent<2>)
{
  // prints: (1, 2) (2, 3) (3, 4)
  std::cout << std::format("({},{})",
               i.first, i.second)";
}
```

- 输出从一个整数序列中每三个连续值相乘得到的值:

```
std::vector<int> v {1, 2, 3, 4, 5};
for (auto i : v | rv::adjacent_transform<3>(
  std::multiplies()))
{
  std::cout << i << ' '; // prints: 3 24 60
}
```

这些示例希望能够帮助理解每个可用视图的可能用例，可以在本书附带的源代码中找到更多示例，也可以在扩展阅读部分找到更多示例。下一节中，我们将讨论范围库的另一部分，约束算法。

9.3. 约束算法

标准库提供了一百多种通用算法，它们有一个共同点: 在迭代器的帮助下处理抽象范围。可以接受迭代器作为参数，有时返回迭代器。这使得重复使用标准容器或数组非常麻烦。这里有一个例子:

```
auto l_odd = [](int const n) {return n % 2 == 1; };

std::vector<int> v{ 1, 1, 2, 3, 5, 8, 13 };
std::vector<int> o;
auto e1 = std::copy_if(v.begin(), v.end(),
            std::back_inserter(o),
            l_odd);

int arr[] = { 1, 1, 2, 3, 5, 8, 13 };
auto e2 = std::copy_if(std::begin(arr), std::end(arr),
            std::back_inserter(o),
            l_odd);
```

这段代码中，有一个 vector v 和一个 array arr，我们将这两个 vector 中的奇数元素复制到第二个 vector o 中。为此，使用 std::copy_if 算法，其接受开始和结束输入迭代器 (定义输入范围)、第二个范围的输出迭代器 (复制的元素将插入其中) 和一个一元谓词 (在本例中为 lambda 表达式)，并返回的是指向最后一个复制元素之后的目标范围的迭代器。

若查看 std::copy_if 算法的声明，会发现有以下两个重载:

```
template <typename InputIt, typename OutputIt,
      typename UnaryPredicate>
constexpr OutputIt copy_if(InputIt first, InputIt last,
                OutputIt d_first,
                UnaryPredicate pred);

template <typename ExecutionPolicy,
      typename ForwardIt1, typename ForwardIt2,
      typename UnaryPredicate>
ForwardIt2 copy_if(ExecutionPolicy&& policy,
           ForwardIt1 first, ForwardIt1 last,
           ForwardIt2 d_first,
           UnaryPredicate pred);
```

第一个重载就是这里使用和描述的重载，第二个重载是在 C++17 中引入的。允许指定执行策略，如并行或串行。这基本上支持标准算法的并行执行。但这与本章的主题无关，我们将不再深入探讨。

大多数标准算法在 std::ranges 命名空间中都有一个新的约束版本。这些算法存在于 <algorithm>，<numeric> 和 <memory> 头文件中，并且具有以下特征:

- 与现有算法的名称相同。
- 具有重载，允许指定一个范围，可以使用开始迭代器和结束哨兵，也可以作为单个范围参数。
- 修改了返回类型，提供关于执行的更多信息。
- 支持应用于已处理元素的投影。投影可以是一个可调用的实体，可以是指向成员的指针、Lambda 表达式或函数指针。在算法逻辑使用范围元素之前，将这样的投影应用到范围元素。

下面是如何声明 std::ranges::copy_if 算法的重载:

```
template <std::input_iterator I,
      std::sentinel_for<I> S,
      std::weakly_incrementable O,
      class Proj = std::identity,
      std::indirect_unary_predicate<
        std::projected<I, Proj>> Pred>
requires std::indirectly_copyable<I, O>
constexpr copy_if_result<I, O> copy_if(I first, S last,
                     O result,
                       Pred pred,
                     Proj proj = {} );

template <ranges::input_range R,
      std::weakly_incrementable O,
      class Proj = std::identity,
      std::indirect_unary_predicate<
      std::projected<ranges::iterator_t<R>, Proj>> Pred>
requires std::indirectly_copyable<ranges::iterator_t<R>, O>
constexpr copy_if_result<ranges::borrowed_iterator_t<R>, O>
      copy_if(R&& r,
```

```
          O result,
          Pred pred,
          Proj proj = {});
```

若这些看起来比较难读，因为参数太多、约束和更长的类型名。然而，好的方面是它们使代码更容易编写。下面是之前的代码片段，使用 std::ranges::copy_if:

```
std::vector<int> v{ 1, 1, 2, 3, 5, 8, 13 };
std::vector<int> o;
auto e1 = std::ranges::copy_if(v, std::back_inserter(o),
                l_odd);

int arr[] = { 1, 1, 2, 3, 5, 8, 13 };
auto e2 = std::ranges::copy_if(arr, std::back_inserter(o),
                l_odd);

auto r = std::ranges::views::iota(1, 10);
auto e3 = std::ranges::copy_if(r, std::back_inserter(o),
                l_odd);
```

这些例子展示了两件事: 如何从 std::vector 对象和数组复制元素，以及如何从视图 (范围适配器) 复制元素。这里没有显示的是投影，这一点在前面简要提到过。

投影是一个可调用的实体，基本上是一个功能适配器。它影响谓词，提供了一种执行函数组合的方法，并且没有提供改变算法的方法。假设有以下类型:

```
struct Item
{
  int id;
  std::string name;
  double price;
};
```

为了便于解释，再考虑以下元素序列:

```
std::vector<Item> items{
  {1, "pen", 5.49},
  {2, "ruler", 3.99},
  {3, "pensil case", 12.50}
};
```

投影允许对谓词执行组合。例如，想要将所有名称以字母 p 开头的项复制到第二个 vector 中。可以这样:

```
std::vector<Item> copies;
std::ranges::copy_if(
  items,
  std::back_inserter(copies),
  [](Item const& i) {return i.name[0] == 'p'; });
```

然而，也可以写成如下等价的方式:

```
std::vector<Item> copies;
std::ranges::copy_if(
  items,
  std::back_inserter(copies),
  [](std::string const& name) {return name[0] == 'p'; },
  &Item::name);
```

本例中，投影是指向成员的指针表达式 &Item::name，在执行谓词 (Lambda 表达式) 之前应用于每个 Item 元素。当有可重用的函数对象或 Lambda 表达式，而不想再写一个来传递不同类型的参数时，这会很有用。

以这种方式，项目不能用于将范围从一种类型转换为另一种类型。例如，不能将 Item 名称从 std::vector<Item> 复制到 std::vector<std::string>。这需要使用 std::ranges::transform 范围适配器，如下所示:

```
std::vector<std::string> names;
std::ranges::copy_if(
  items | rv::transform(&Item::name),
  std::back_inserter(names),
  [](std::string const& name) {return name[0] == 'p'; });
```

有许多受约束的算法，但不会在这里列出它们。有兴趣的读者可以直接在标准中或在https://en.cppreference.com/w/cpp/algorithm/ranges页面上找到它们。

本章的最后一个主题是编写自定义范围适配器。

9.4. 编写范围适配器

标准库包含一系列范围适配器，可用于解决许多不同的任务。新版本的标准中增加了更多的内容。但某些情况下，可能希望创建自己的范围适配器，以便与范围库中的其他适配器一起使用。这实际上不是一项微不足道的任务，所以在本章的最后一节中，我们将探讨编写这样一个范围适配器所需要遵循的步骤。

为此，考虑一个范围适配器，接受范围的第 n 个元素，而跳过其他元素。我们将这个适配器称为 step_view。我们可以使用它来编写如下代码:

```
for (auto i : std::views::iota(1, 10) | views::step(1))
  std::cout << i << '\n';

for (auto i : std::views::iota(1, 10) | views::step(2))
  std::cout << i << '\n';

for (auto i : std::views::iota(1, 10) | views::step(3))
  std::cout << i << '\n';

for (auto i : std::views::iota(1, 10) | views::step(2) |
       std::views::take(3))
  std::cout << i << '\n';
```

第一个循环将打印从 1 到 9 的所有数字。第二个循环将输出所有的奇数，1、3、5、7、9。第三个循环将输出 1,4,7。最后，第四个循环将输出 1,3,5。

为了实现这个，我们需要以下实现:

- 定义范围适配器的类模板
- 推导指南，用于帮助演绎范围适配器的类模板参数
- 为范围适配器定义迭代器类型的类模板
- 为范围适配器定义哨点类型的类模板
- 实现所需的重载管道操作符 (|) 和辅助函子
- 编译时常量全局对象，以简化范围适配器的使用

一个个地来了解如何定义它们，从哨兵类开始。哨点是对过尾迭代器的抽象，允许检查迭代是否到达了范围的末端。哨兵可以使结束迭代器的类型与范围迭代器的类型不同，哨兵不能解除引用或增加。下面是其定义:

```
template <typename R>
struct step_iterator;

template <typename R>
struct step_sentinel
{
  using base = std::ranges::iterator_t<R>;
  using size_type = std::ranges::range_difference_t<R>;

  step_sentinel() = default;

  constexpr step_sentinel(base end) : end_{ end } {}
  constexpr bool is_at_end(step_iterator<R> it) const;

private:
  base end_;
};

// definition of the step_iterator type

template <typename R>
constexpr bool step_sentinel<R>::is_at_end(
  step_iterator<R> it) const
{
  return end_ == it.value();
}
```

哨兵由迭代器构造，包含一个名为 is_at_end 的成员函数，该函数检查存储的范围迭代器是否等于存储在 step_iterator 对象中的范围迭代器。这个类型 step_iterator 是一个类模板，定义了范围适配器的迭代器类型，称之为 step_view。下面是该迭代器类型的实现:

```
template <typename R>
struct step_iterator : std::ranges::iterator_t<R>
{
```

```
  using base
    = std::ranges::iterator_t<R>;
  using value_type
    = typename std::ranges::range_value_t<R>;
  using reference_type
    = typename std::ranges::range_reference_t<R>;

  constexpr step_iterator(
    base start, base end,
    std::ranges::range_difference_t<R> step) :
    pos_{ start }, end_{ end }, step_{ step }
  {
  }

  constexpr step_iterator operator++(int)
  {
    auto ret = *this;
    pos_ = std::ranges::next(pos_, step_, end_);
    return ret;
  }

  constexpr step_iterator& operator++()
  {
    pos_ = std::ranges::next(pos_, step_, end_);
    return *this;
  }

  constexpr reference_type operator*() const
  {
    return *pos_;
  }
  constexpr bool operator==(step_sentinel<R> s) const
  {
    return s.is_at_end(*this);
  }

  constexpr base const value() const { return pos_; }

private:
  base pos_;
  base end_;
  std::ranges::range_difference_t<R> step_;
};
```

这个类型必须有几个成员:

- 别名模板 base，表示基础范围迭代器的类型。
- 别名模板称为 value_type，表示底层范围的元素类型。
- 重载操作符 ++ 和 *。

- 重载操作符 == 将该对象与哨兵进行比较。

++ 操作符的实现使用 std::ranges::next 约束算法对迭代器增加 N 个位置，但不超过范围的结束。

为了将 step_iterator 和 step_sentinel 对用于 step_view 范围适配器，必须确保这个对实际上是格式良好的。为此，必须确保 step_iterator 类型是一个输入迭代器，并且 step_sentinel 类型确实是 step_iterator 类型的哨兵类型。这可以在 static_assert 帮助下完成:

```
namespace details
{
  using test_range_t =
    std::ranges::views::all_t<std::vector<int>>;
  static_assert(
    std::input_iterator<step_iterator<test_range_t>>);
  static_assert(
    std::sentinel_for<step_sentinel<test_range_t>,
    step_iterator<test_range_t>>);
}
```

step_iterator 类型用于 step_view 范围适配器的实现:

```
template<std::ranges::view R>
struct step_view :
  public std::ranges::view_interface<step_view<R>>
{
private:
  R base_;
  std::ranges::range_difference_t<R> step_;

public:
  step_view() = default;

  constexpr step_view(
    R base,
    std::ranges::range_difference_t<R> step)
      : base_(std::move(base))
      , step_(step)
  {
  }

  constexpr R base() const&
    requires std::copy_constructible<R>
  { return base_; }
  constexpr R base()&& { return std::move(base_); }

  constexpr std::ranges::range_difference_t<R> const&
  increment() const
  { return step_; }

  constexpr auto begin()
  {
    return step_iterator<R const>(
```

```
      std::ranges::begin(base_),
      std::ranges::end(base_), step_);
  }

  constexpr auto begin() const
  requires std::ranges::range<R const>
  {
    return step_iterator<R const>(
      std::ranges::begin(base_),
      std::ranges::end(base_), step_);
  }

  constexpr auto end()
  {
    return step_sentinel<R const>{
      std::ranges::end(base_) };
  }

  constexpr auto end() const
  requires std::ranges::range<R const>
  {
    return step_sentinel<R const>{
      std::ranges::end(base_) };
  }

  constexpr auto size() const
  requires std::ranges::sized_range<R const>
  {
    auto d = std::ranges::size(base_);
    return step_ == 1 ? d :
      static_cast<int>((d + 1)/step_);
  }

  constexpr auto size()
  requires std::ranges::sized_range<R>
  {
    auto d = std::ranges::size(base_);
    return step_ == 1 ? d :
      static_cast<int>((d + 1)/step_);
  }
};
```

定义范围适配器时，必须遵循一个模式。这种模式表现为以下几个方面:

- 类模板必须有符合 std::ranges::view 概念的模板参数。
- 类模板应该派生自 std::ranges:view_interface，其本身接受一个模板参数，应该是范围适配器类。这基本上是第 7 章中的 CRTP 实现。
- 类必须有默认构造函数。
- 类必须具有返回基础范围的基成员函数。

- 该类必须有 begin 成员函数，该函数返回指向范围内第一个元素的迭代器。
- 该类必须有一个结束成员函数，该函数返回一个指向范围中倒数第一元素的迭代器或一个哨兵。
- 对于满足 std::ranges::size_range 概念要求的范围，该类还必须包含名为 size 的成员函数，该函数返回范围内的元素数量。

为了能够对 step_view 类使用类模板参数推导，应该定义一个用户定义的推导指南。这在第 4 中讨论过，指南应该如下所示:

```
template<class R>
step_view(R&& base,
      std::ranges::range_difference_t<R> step)
  -> step_view<std::ranges::views::all_t<R>>;
```

为了能够使用管道迭代器 (|) 将此范围适配器与其他范围适配器组合在一起，必须重载此操作符。但需要一些辅助函数对象，如下所示:

```
namespace details
{
  struct step_view_fn_closure
  {
    std::size_t step_;
    constexpr step_view_fn_closure(std::size_t step)
      : step_(step)
    {
    }

    template <std::ranges::range R>
    constexpr auto operator()(R&& r) const
    {
      return step_view(std::forward<R>(r), step_);
    }
  };
  template <std::ranges::range R>
  constexpr auto operator | (R&& r,
                step_view_fn_closure&& a)
  {
    return std::forward<step_view_fn_closure>(a)(
      std::forward<R>(r));
  }
}
```

step_view_fn_closure 类是一个函数对象，存储了一个表示每个迭代器要跳过的元素数量的值。其重载调用操作符以一个范围作为参数，并返回从该范围创建的 step_view 对象和要跳转的步数的值。

最后，希望能够以与标准库中可用的方式类似的方式编写代码，标准库为每个存在的范围适配器在 std::views 命名空间中提供了一个编译时全局对象。例如，可以使用 std::views::transform，而不是 std::ranges::transform_view。类似地，我们希望有一个对象 views::step，而不是 step_view(某个命

名空间中)。为此，还需要另一个函数对象:

```
namespace details
{
  struct step_view_fn
  {
    template<std::ranges::range R>
    constexpr auto operator () (R&& r,
                    std::size_t step) const
    {
      return step_view(std::forward<R>(r), step);
    }

    constexpr auto operator () (std::size_t step) const
    {
      return step_view_fn_closure(step);
    }
  };
}

namespace views
{
  inline constexpr details::step_view_fn step;
}
```

step_view_fn 类型是一个函数对象，对调用操作符有两个重载: 一个接受一个范围和一个整数，并返回一个 step_view 对象，另一个接受一个整数并返回这个值的闭包。更准确地说，是前面 step_view_fn_closure 的实例。

实现了这些之后，就可以成功地运行本节开头所示的代码。我们已经完成了一个简单的范围适配器的实现，希望这能让您了解写入范围适配器需要什么。从细节上看，范围库非常复杂。本章中，已经了解了一些关于库内容的基础知识，如何简化代码，以及如何使用自定义特性扩展它。若想利用其他资源了解更多，这些知识应该只能算是一个起点。

9.5. 总结

本书的最后一章中，我们探索了 C++20 的范围库。我们从抽象的范围概念到新的范围库。了解了这个库的内容，以及如何帮助我们编写更简单的代码。我们主要讨论了范围适配器，但也讨论了约束算法。本章的最后，学习了如何编写可与标准适配器组合使用的自定义范围适配器。

9.6. 习题

1. 什么是范围?
2. 什么是范围库中的视图?
3. 什么是约束算法?
4. 什么是哨兵?
5. 如何检查哨兵类型是否与迭代器类型相对应?

9.7. 扩展阅读

- A beginner’s guide to C++ Ranges and Views, Hannes Hauswedell, https://hannes.hauswedell.net/post/2019/11/30/range_intro/
- Tutorial: Writing your first view from scratch (C++20/P0789), Hannes Hauswedell, https://hannes.hauswedell.net/post/2018/04/11/view1/
- C++20 Range Adaptors and Range Factories, Barry Revzin, https://brevzin.github.io/c++/2021/02/28/ranges-reference/
- Implementing a better views::split, Barry Revzin, https://brevzin.github.io/c++/2020/07/06/split-view/
- Projections are Function Adaptors, Barry Revzin, https://brevzin.github.io/c++/2022/02/13/projections-function-adaptors/
- Tutorial: C++20’s Iterator Sentinels, Jonathan Müller, https://www.foonathan.net/2020/03/iterator-sentinel/
- Standard Ranges, Eric Niebrel, https://ericniebler.com/2018/12/05/standard-ranges/
- Zip, Tim Song, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2321r2.html
- From range projections to projected ranges, Oleksandr Koval, https://oleksandrkvl.github.io/2021/10/11/projected-ranges.html
- Item 30 - Create custom composable views, Wesley Shillingford, https://cppuniverse.com/EverydayCpp20/RangesCustomViews
- A custom C++20 range view, Marius Bancila, https://mariusbancila.ro/blog/2020/06/06/a-custom-cpp20-range-view/
- New C++23 Range Adaptors, Marius Bancila, https://mariusbancila.ro/blog/2022/03/16/new-cpp23-range-adaptors/
- C++ Code Samples Before and After Ranges, Marius Bancila, https://mariusbancila.ro/blog/2019/01/20/cpp-code-samples-beforeand-after-ranges/

结束语

这里是本书的结尾。模板并不是 C++ 编程中最简单的部分，人们通常觉得它很难或可怕。然而，模板在 C++ 代码中大量使用，无论编写什么样的代码，很可能每天都要使用模板。我们从学习模板是什么，以及我们为什么需要它们开始这本书。然后，学习了如何定义函数模板、类模板、变量模板和别名模板。

学习了模板参数、特化和实例化。第 3 章中，学习了具有可变参数数量的模板，这些模板称为可变参数模板。第 4 章专门介绍更高级的模板概念，如名称绑定、递归、参数推导和转发引用。

然后，第 5 章学习了类型特征、SFINAE 和 constexpr if 的使用，并探索了标准库中可用的类型特征集合。第 6 章介绍 C++20 标准中的概念和约束，学习了如何以不同的方式为模板参数指定约束，以及如何定义概念，以及与它们相关的一切。还探讨了标准库中可用概念的集合。

本书的最后一部分中，我们重点介绍了如何将模板用于实际目的。首先，第 7 章探讨了一系列模式和习惯用法，例如 CRTP、混入、类型擦除、标记分派、表达式模板和类型列表。然后，第 8 章学习了容器、迭代器和算法，这些是标准模板库的核心，并编写了一些属于我们自己的容器和迭代器。最后，第 9 章专门介绍了 C++20 范围库，学习了范围、范围适配器，以及约束算法。

到此为止，已经完成了使用 C++ 模板学习元编程的旅程，但这个学习过程并没有到此结束。一本书只能为你提供学习元编程所必需的信息，以一种易于理解和理解的方式组织起来。不过，只读一本书而不实践，会使学到的东西无用。各位读者现在的任务，就是把你从这本书中学到的知识运用到工作中、在学校里或在家里。因为只有通过练习，才能真正掌握 C++ 语言和使用模板的元编程，以及其他任何其他技能。

为实现成为高产的 C++ 模板的目标，这本书会自证其价值。在撰写本书的时候，我也尝试在简单和有意义之间寻求平衡，这样读者们就能更容易地学习和了解一些较为困难的知识点，但愿我做到了。

感谢你阅读这本书，并祝各位在实践中好运！

参考答案

第 1 章，模板简介

习题 1

为什么需要模板? 模板有什么优势?

参考答案

使用模板有几个好处: 避免编写重复的代码，促进了泛型库的创建，它们可以帮助我们编写少而优的代码。

习题 2

什么是函数和类模板?

参考答案

作为模板的函数称为函数模板。类似地，作为模板的类称为类模板。

习题 3

模板参数有多少种，都是什么?

参考答案

有三种模板参数: 类型模板参数、非类型模板参数和模板模板参数。

习题 4

什么是偏特化和全特?

参考答案

特化是一种为模板提供替代实现的技术，称为主模板。偏特化是仅为某些模板参数提供的替代实现。当为所有模板形参提供参数时，完特化是另一种实现。

习题 5

模板的主要缺点是什么?

参考答案

使用模板的主要缺点包括: 复杂的语法，冗长的编译器错误，难以阅读和理解，以及会增加编译时间。

第 2 章，了解模板

习题 1

什么类型可以用于非类型模板参数?

参考答案

非类型模板参数只能有结构类型。结构类型包括整型、浮点型 (截至 C++20)、枚举类型、指针类型 (指向对象或函数)、指向成员类型的指针 (指向成员对象或成员函数)、左值引用类型 (指向对象或函数)，以及满足以下要求的文字类类型: 所有基类都是公共的和不可变的，所有非静态数据成员都是公共的和不可变的，所有基类和非静态数据成员的类型也是结构类型或其数组。这些类型的 const 和 volatile 限定版本也允许。

习题 2

默认模板参数在哪里不允许使用?

参考答案

默认模板实参不能用于形参包、友元类模板的声明以，及函数模板或成员函数模板的显式特化声明或定义中。

习题 3

什么是显式实例化声明，在语法上与显式实例化定义有何不同?

参考答案

显式实例化声明是告诉编译器模板实例化的定义在不同的翻译单元中，并且不应该生成新的定义的一种方式。其语法与显式实例化定义相同，只是在声明前使用了 extern 关键字。

习题 4

什么是别名模板?

参考答案

别名模板是一个名称，与引用另一种类型的类型别名不同，它引用一个模板。换句话说，引用一组类型。使用 using 声明引入别名模板，不能用 typedef 声明引入。

习题 5

什么是模板 Lambda?

参考答案

模板 Lambda 是泛型 Lambda 的进阶形式，在 C++20 中引入。允许使用模板语法显式地指定编译器，为 Lambda 表达式生成函数对象模板化函数调用操作符的形式。

第 3 章，可变参数模板

习题 1

什么是可变参数模板，这种模板有什么用?

参考答案

可变参数模板是具有可变数量参数的模板，不仅允许编写参数数量可变的函数，还允许我们编写类模板、变量模板和别名模板。与其他方法 (如使用 va_宏) 不同，它们是类型安全的，不需要宏，也不需要显式地指定参数的数量。

习题 2

什么是参数包?

参考答案

有两种参数包: 模板参数包和函数参数包。前者是接受零个、一个或多个模板实参的模板形参。后者是接受零个、一个或多个函数参数的函数参数。

习题 3

什么是可以展开参数包的上下文?

参考答案

参数包可以在多种上下文中展开，如下所示: 模板形参列表、模板实参列表、函数形参列表、圆括号括起来的初始化式、大括号括起来的初始化式、基形说明符和成员初始化式列表、折叠表达式、using 声明、Lambda 捕获、sizeof…操作符、对齐说明符和属性列表。

习题 4

什么是折叠表达式?

参考答案

折叠表达式是一种包含参数包的表达式，将参数包中的元素折叠 (或减少) 到二进制运算符上。

习题 5

使用折叠表达式有什么好处?

参考答案

折叠表达式的好处包括编写的代码更少、更简单，模板实例化更少，具有更快的编译时间和潜在的更快的代码，因为多个函数只使用一个表达式。

第 4 章，高级模板概念

习题 1

什么时候执行名称查找?

参考答案

模板实例化依赖名称 (依赖模板形参的类型或值) 时执行名称查找，在模板定义非依赖名称 (不依赖模板形参的名称) 时执行名称查找。

习题 2

什么是推导指南?

参考答案

推导指南是一种告诉编译器如何执行类模板参数推导的机制，是表示虚构类类型的构造函数签名的虚构函数模板。若重载解析在构造的虚拟函数模板集上失败，则程序是格式错误的，并生成一个错误。否则，所选函数模板特化的返回类型，将成为推导类模板的特化。

习题 3

什么是转发引用?

参考答案

转发引用 (也称为通用引用) 是模板中的一种引用，若右值作为参数传递，则表现为右值引用; 若左值作为参数传递，则表现为左值引用。转发引用必须具有 T&& 形式，例如 template <typename T> void f(T&&)。像 T const && 或 std::vector<T>&& 这样的格式并不表示转发引用，而是正常的右值引用。

习题 4

decltype 是用来做什么的?

参考答案

decltype 说明符是一个类型说明符，返回表达式的类型，通常在模板中与 auto 说明符一起使用，以声明依赖于模板参数的函数模板的返回类型，或者包装另一个函数并返回执行包装函数中函数的返回类型。

习题 5

std::declval 是用来做什么的?

参考答案

std::declval 是 <utility> 头文件中的工具函数模板，可将右值引用添加到其类型模板参数中。期只能用于未求值的上下文中 (仅编译时上下文中，在运行时不求值)，目的是帮助对没有默认构造函数，或由于 private 或 protected 而不能访问的类型进行依赖类型求值。

第 5 章，类型特征和条件编译

习题 1

什么是类型特征?

参考答案

类型特征是 (小) 型类模板，能够查询类型的属性或执行类型的转换。

习题 2

什么是 SFINAE?

参考答案

SFINAE 是**替换失败不是错误**的首字母缩写。这是模板替换的规则，其工作原理如下: 当编译器遇到使用函数模板时，会替换实参以实例化模板; 若此时发生错误，则不将其视为格式错误的代码，只将其视为推导失败，相应函数从重载集中删除，而不是引起错误。只有在特定函数调用的重载集中没有匹配项时，才会产生错误。

习题 3

什么是 constexpr if?

参考答案

constexpr if 是 if 语句的编译时版本，其语法是 if constexpr(condition)，从 C++17 开始可用。允许在编译时，根据编译时表达式的值丢弃分支。

习题 4

std::is_same 可用来做什么?

参考答案

std::is_same 是一个类型特征，用于检查两个类型是否相同，包括对 const 和 volatile 限定符的检查，对于具有不同限定符的两种类型 (例如 int 和 int const)，将生成 false。

习题 5

std::conditional 可用来做什么?

参考答案

std::conditional 是一个元函数，可根据编译时常数选择一种类型或另一种类型。

第 6 章，概念和约束

习题 1

什么是约束，什么是概念?

参考答案

约束是强加在模板参数上的要求。概念是一个或多个约束的命名集。

习题 2

什么是 requires 子句和 requires 表达式?

参考答案

requires 子句是一个构造，允许在模板参数或函数声明上指定约束。这个构造由 requires 关键字和一个编译时布尔表达式组成。require 子句影响函数的行为，只有当布尔表达式为 true 时，才包括它进行重载解析。另一方面，requires 表达式有 requires(参数列表) 表达式，其中 parameters-list 可选，其目的是验证某些表达式是否格式良好，而不会产生任何副作用或影响函数的行为。requires 表达式可以与 requires 子句一起使用。不过，推荐首选命名概念，主要是从可读性的角度考虑。

习题 3

requires 表达式的类别是什么?

参考答案

需求表达式有四类: 简单需求、类型需求、复合需求和嵌套需求。

习题 4

约束如何影响重载解析中模板的顺序?

参考答案

函数的约束影响重载解析他们的顺序。当多个重载与参数集匹配时，将选择约束更强的重载。但请记住，类型特征 (或布尔表达式) 和概念的约束在语义上不相同。

习题 5

缩写函数模板是什么?

参考答案

缩写函数模板是 C++20 引入的一个新特性，为函数模板提供了简化的语法。自动说明符可用于定义函数形参，模板语法可跳过。编译器会自动从缩写的函数模板生成一个函数模板。可以使用概念约束这些函数，因此可以对模板参数施加约束。

第 7 章，模式和习语

习题 1

CRTP 可以解决哪些问题?

参考答案

奇异迭代模板模式 (CRTP) 通常用于解决，诸如向类型添加公共功能和避免代码重复、限制类型可以实例化的次数或实现复合设计模式等问题。

习题 2

什么是 Mixins? 其目的是什么?

参考答案

Mixins 是一些小类型，通过它们补充的类继承来为其他类添加功能。这与 CRTP 模式相反。

习题 3

什么是类型擦除?

参考答案

类型擦除是用于描述从类型中删除信息的模式的术语，从而使不相关的类型能够以通用的方式进行处理。虽然类型擦除的形式可以通过 void 指针或多态性实现，但真正的类型擦除模式是通过模板实现的 (C++ 中)。

习题 4

什么是标记分派，它的替代方案是什么?

参考答案

标记分派是一种技术，使我们能够在编译时选择一个或另一个函数重载。尽管标记调度本身是 std::enable_if 和 SFINAE 的替代方案，但它也有替代方案。在 C++17 中是 constexpr if，在 C++20 中是概念。

习题 5

表达式模板是什么? 可以在哪里使用它们?

参考答案

表达式模板是一种元编程技术，支持在编译时对计算进行惰性计算。这种技术的好处是，可避免在运行时执行低效的操作，代价是代码更加复杂，难以理解。表达式模板通常用于实现线性代数库。

第 8 章，范围和算法

习题 1

标准库中的序列容器是什么?

参考答案

C++ 标准库中的序列容器有 std::vector、std::deque、std::list、std::array 和 std::forward_list。

习题 2

标准容器中定义的公共成员函数是什么?

参考答案

标准库中为大多数容器定义的成员函数是 size(std::forward_list 中不存在)、empty、clear(std::array、std::stack、std::queue 和 std::priority_queue 中不存在)、swap、begin 和 end。

习题 3

什么是迭代器，存在多少个类别?

参考答案

迭代器是一种抽象，能够以通用的方式访问容器的元素，而不必知道每个容器的实现细节。迭代器是编写通用算法的关键。C++ 中有六类迭代器: 输入、向前、双向、随机访问、连续 (至 C++17) 和输出。

习题 4

随机迭代器支持哪些操作?

参考答案

随机迭代器必须支持以下操作 (除了输入迭代器、正向迭代器和双向迭代器所需的操作之外):+ 和-算术运算符、不等式比较 (与其他迭代器)、复合赋值和偏移解引用操作符。

习题 5

范围访问函数是什么?

参考答案

范围访问函数是非成员函数，提供统一的方式来访问容器、数组和 std::initializer_list 类的数据或属性。这些函数包括 std::size/std::ssize、std::empty、std::data、std::begin 和 std::end。

第 9 章，范围库

习题 1

什么是范围?

参考答案

范围是元素序列的抽象，用开始迭代器和结束迭代器定义。开始迭代器指向序列中的第一个元素，结束迭代器指向序列的最后一个元素。

习题 2

什么是范围库中的视图?

参考答案

C++ 范围库中的视图，也称为范围适配器，是实现一种算法的对象，该算法以一个或多个范围作为输入。可能还有其他参数，并返回一个经过调整的范围。视图可惰性求值，所以可以直到其元素迭代时，才进行适配。

习题 3

什么是约束算法?

参考答案

约束算法是现有标准库算法的实现，在 C++20 范围库中，其之所以“受约束”，是因为它们的模板参数使用 C++20 的概念进行约束。这些算法中，值范围接受单个范围参数，而不是需要 begin-end 迭代器对进行指定，但也存在接受迭代器-哨兵对的重载。

习题 4

什么是哨兵?

参考答案

哨点是对结束迭代器的抽象。这使得结束迭代器的类型可能与范围迭代器的类型不同，哨兵不能解除引用或增加。当对范围尽头的测试依赖于某些可变 (动态) 条件，并且直到某些事情发生 (条件变为 false) 时，才知道处于范围末尾时，哨兵是有用的。

习题 5

如何检查哨兵类型是否与迭代器类型相对应?

参考答案

可以使用 <iterator> 头文件中的 std::sentinel_for 概念来检查哨兵类型是否可以与迭代器类型一起使用。