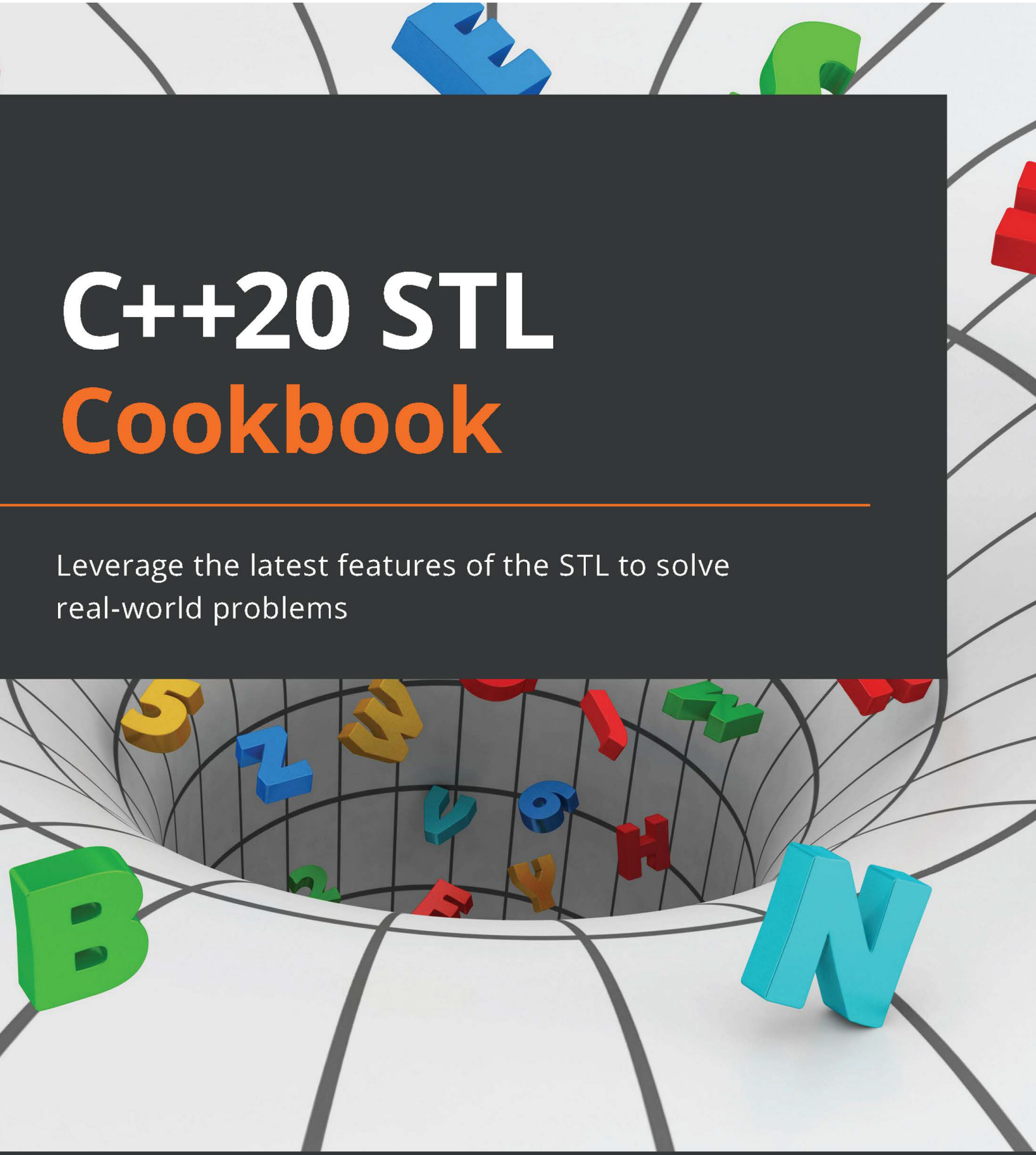
C++20 STL Cookbook

Leverage the latest features of the STL to solve real-world problems



Bill Weinman

C++20 STL Cookbook

Leverage the latest features of the STL to solve real-world problems

使用 STL 的新特性来解决实际问题

作者: Bill Weinman

译者：陈晓伟

本书概述

快速、高效和灵活是 C++ 编程语言的特点，时期应用于行业的各个领域来解决许多问题。C++20 将改变之前的编码方式，并带来了一系列支持应用快速部署的特性。这本书将展示，如何以最优的方式使用 STL。

本书将从 C++20 的新特性开始，帮助读者理解该语言的机制和库特性，并展示它们是如何工作的。与其他书籍不同，本书采用了一种特定于问题解决的方式。读者们将了解核心 STL 的概念，如容器、算法、工具类、Lambda 表达式、迭代器等，学习与实践相结合。本书使用 C++ STL 及其最新功能的参考指南，可用来探索函数式编程和 Lambda 表达式中的最细特性。

阅读完这本书后，将能够优雅地使用 STL 解决实际问题。

关键特性

- 了解 C++20 的新特性，并使用 STL 编写更好的代码
- 减少应用的开发时间，并进行更快的部署
- 启动和使用新版本中引入的、更精简的 STL 功能

作者简介

Bill Weinman 自其 1971 年 16 岁时拥有了他的第一台计算机以来，就一直在从事技术工作。自 20 世纪 70 年代初以来，一直用 C 和 C++ 编程，为包括 NASA、美国银行、施乐、IBM 和美国海军在内的主要客户编写系统和应用。他还是一名电子工程师，曾为旅行者 II 号宇宙飞船、SAE 的音频放大器和 Altec Lansing 的音响系统工作。

自 20 世纪 90 年代中期以来，Weinman 先生一直专注于写作和教学。他的书和课程涵盖了 HTML、SQL、CGI、Python，当然还有 C 和 C++。作为在线学习的早期贡献者，清晰、简洁的授课方式使他的课程在 lynda 和 LinkedIn learning 上很受欢迎。

可以关注 Bill 的网站:bw.org。

本书相关

- Github 地址：
  https://github.com/xiaoweiChen/CXX20-STL-Cookbook

前言

关于本书

本书介绍了使用 C++ STL(标准模板库) 的方法，以及 C++20 加入的新特性。

C++ 是一门强大的底层语言，其建立在 C 语言的基础上，具有类型安全、泛型编程和面向对象编程的语法扩展等特性。STL 提供了一组高级类、函数和算法，可使编程工作更容易、更有效、更不容易出错。

我认为五种语言和特性拼凑成了 C++，包括：1) C 语言，2) 神秘且功能强大的宏预处理器，3) 功能丰富的类/对象模型，4) 模板的泛型编程模型，5) 建立在 C++ 类和模板之上的 STL。

预备知识

本书假定读者对 C++ 有基本的了解，包括语法、结构、数据类型、类和对象、模板和 STL。

本书中例子假定已经包含某些头文件来使用库函数。示例通常不会列出所有必要的头文件，而更倾向于关注当前描述的技术。建议读者下载示例代码，其中包含完整的代码段。

可以从 GitHub 下载示例代码: https://github.com/PacktPublishing/CPP-20-STL-Cookbook。

本书使用 C++20

C++ 语言由国际标准化组织 (ISO) 以约三年为一个周期进行标准化，目前的标准称为 C++20(在此之前有 C++17、C++14 和 C++11)。C++20 于 2020 年 9 月获得批准。

C++20 为语言和 STL 添加了许多重要的特性。格式 (format)、模块 (module)、范围 (range) 等新特性，将对使用 STL 的方式产生重大影响。

还有一些变化。例如，想删除 vector 的匹配的元素，可以使用 erase-remove 方法:

```
auto it = std::remove(vec1.begin(), vec1.end(), value);
vec1.erase(it, vec1.end());
```

从 C++20 起，可以直接使用新的 std::erase 函数，并在一个函数调用中完成这些工作:

```
std::erase(vec1, value);
```

C++20 有许多细微的和实质性的改进。本书中，介绍了其中的大部分内容，特别是与 STL 相关的特性。

大括号初始化

本书中的方法经常使用大括号初始化，来代替更熟悉的复制初始化。

```
std::string name{ "Jimi Hendrix" }; // braced initialization
std::string name = "Jimi Hendrix"; // copy initialization
```

赋值操作符 (=) 作为赋值操作符和复制操作符，具有双重作用，并且为大家所熟悉的。

但赋值操作符有其缺点，它也是一个复制构造函数，所以数据类型可能会进行隐式窄向转换。这不仅效率低，并且可能导致意外的类型转换，会使程序变得很难调试。

大括号初始化使用列表初始化操作符 {}(C++11 引入)，可以避免这些问题。

同样值得注意的是，T{} 的特殊情况保证为“零初始化”。

```
int x; // uninitialized bad :(
int x = 0; // zero (copy constructed) good :)
int x{}; // zero (zero-initialized) best :D
```

空大括号 0 初始化为初始化新变量，提供了一种更简单快捷的方式。

隐藏 std:: 命名空间

本书中的练习将隐藏 std:: 命名空间，这主要是出于排版和可读性的考虑。我们都知道大多数 STL 标识符都在 std:: 命名空间中。所以这里会使用某种形式的 using 声明，以避免使用重复的前缀使示例变得混乱。例如，使用 cout 时，可以假设已经包含了一个 using 声明，就像这样:

```
using std::cout; // cout is now sans prefix
cout << "Hello, Jimi!\n";
```

示例代码中不会显示 using 声明，这可以使我们能更加专注于示例所要表达的内容。

但是，代码中导入整个 std:: 命名空间是一种糟糕的做法。应该避免这样使用:

```
using namespace std; // bad. don't do that.
cout << "Hello, Jimi!\n";
```

命名空间包含数千个标识符，没有理由让命名空间充满标识符。冲突的可能性还是有的，并且对于调试来说风险更高。当使用一个没有 std:: 前缀的名称时，首选的方法是一次导入一个命名空间。

为了进一步避免命名空间冲突，我经常为重用的类使用单独的命名空间。我倾向于使用 bw 作为我自己的命名空间，你也可以选择适合你的命名空间名称。

使用 using 进行类型别名

本书对类型别名使用的是 using，而不是 typedef。

STL 类和类型有时会很冗长。例如，模板化迭代器类可能长这样:

```
std::vector<std::pair<int,std::string>>::iterator
```

长类型名不仅很难输入，而且容易出错。

一种常见的方式是用 typedef 缩写长类型名:

```
typedef std::vector<std::pair<int,std::string>>::iterator
vecit_t
```

这为迭代器类型声明了一个别名。typedef 继承自 C 语言，其语法反映了这一点。

从 C++11 开始，using 关键字可以用来创建类型别名:

```
using vecit_t =
std::vector<std::pair<int,std::string>>::iterator;
```

通常，using 别名等同于 typedef。最重要的区别是 using 别名可以模板化:

```cpp
template<typename T>
using v = std::vector<T>;
v<int> x{};
```

本书更倾向于使用 using 来处理类型别名。

简化函数模板

从 C++20 开始，可以在没有模板文件的情况下指定简化函数模板。例如:

```cpp
void printc(const auto& c) {
  for (auto i : c) {
    std::cout << i << '\n';
  }
}
```

参数列表中的 auto 类型类似于匿名模板 typename:

```cpp
template<typename C>
void printc(const C& c) {
  for (auto i : c) {
    std::cout << i << '\n';
  }
}
```

虽然才在 C++20 标准中出现，但主流编译器早已支持简化函数模板了。本书将在许多示例中使用简化函数模板。

C++20 的 format()

C++20 前，我们可以选择使用传统的 printf() 或 STL cout 来格式化文本。虽然两者都有缺陷，但确实好用。从 C++20 开始，format() 函数的加入提供了文本格式化，灵感来自 Python3。

使用 STL 解决现实问题

本书中的方法使用 STL 为实际问题提供解决方案，仅依赖于 STL 和 C++ 标准库。这将使读者可以轻松地进行试验和学习，而无需安装和配置第三方依赖库。

现在，让我们与 STL 一起快乐的玩耍吧！

适读人群

本书是为中高级 C++ 开发者所著，这些开发者更多的想要了解 C++20 标准模板库的内容。当然，基本的编码知识和 C++ 概念必须要有，从而才能更愉快的阅读本书。

本书内容

第 1 章，C++20 的新特性，介绍了 C++20 中新 STL 的特性，熟悉新的语言功能。

第 2 章，STL 的泛型特性，讨论在新 C++ 标准中添加的 STL。

第 3 章，STL 容器，介绍了整个 STL 容器库。

第 4 章，兼容迭代器，展示了如何使用和创建 STL 兼容迭代器。

第 5 章，Lambda 表达式，将 STL 函数和算法与 Lambda 表达式一起使用。

第 6 章，STL 算法，提供了使用和创建 STL 兼容算法的方法。

第 7 章，字符串、流和格式化，描述了 STL 字符串和格式化工具类。

第 8 章，实用工具类，涵盖了日期和时间、智能指针、optional 类等 STL 实用工具。

第 9 章，并发和并行，描述了对并发的支持，包括线程、异步、原子类型等。

第 10 章，文件系统，介绍了 std::filesystem，以及如何将其与 C++20 的新特性结合起来使用。

第 11 章，一些的想法，提供了一些解决方案，包括 trie 类型，字符串分割等。展示了如何将 STL 用于实际问题的高级示例。

本书中使用的 GCC 编译器

除非另有说明，本书中的大多数示例都是使用 GCC 编译器 11.2 版本开发和测试的，11.2 版本是撰写本文时最新的稳定版本。

在撰写本书时，C++20 仍然很新，没有在任何编译器上完全实现。在 GCC(GNU)、MSVC(微软)和 Clang(苹果) 这三个主要编译器中，MSVC 编译器在实现新标准方面走得最快。不过，可能还是会遇到在 MSVC 或其他编译器上实现的特性，但 GCC 还没实现。这时，会明确说明使用的编译器。若某个特性还没有在可用的编译器上实现，会告诉读者“目前无法测试这个特性”。

代码可能已经在一个或多个这样的编译器上进行过测试	
GCC 11.2	Debian Linux 5.16.11
LLVM/Clang 13.1.6	macOS 12.13/Darwin 21.4
Microsoft C++ 19.32.31302	Windows 10

这里，强烈建议您安装 GCC 以遵循本书中的示例。GCC 是根据 GNU 通用公共许可证 (GPL) 免费提供的。获得最新版本 GCC 的最简单方法是安装 Debian Linux(也是 GPL) 操作系统，并使用 apt 进行安装 GCC。

若正在使用这本书的数字版本，建议自己输入代码或从本书的 GitHub 库访问代码 (下一节有链接)。这样做将避免与复制和粘贴代码相关的错误。

下载示例

可以从 GitHub 上下载这本书的示例代码文件https://github.com/PacktPublishing/CPP-20-STL-Cookbook。若代码有更新，会在 GitHub 库中更新。

目录

第 1 章　C++20 的新特性

本章主要介绍 C++20 添加的新特性，有些要等编译器支持。

C++20 中添加了很多内容，远远超出了本章的范围。下面这些，是我认为比较重要的。

- 格式化文本
- constexpr——使用编译时 vector 和字符串
- 安全地比较不同类型的整数
- 三向比较运算符 <=>——进行三种比较
- <version> 头文件——查找特性测试宏
- 概念 (concept) 和约束 (constraint)——创建更安全的模板
- 模块——避免重新编译模板库
- 范围容器中创建视图

本章旨在让读者熟悉 C++20 中的这些新特性，之后可以在自己的项目中使用。

1.1. 相关准备

本章所有的例子和源代码都可以在本书的 GitHub 库中找到，网址是https://github.com/PacktPublishing/CPP-20-STL-Cookbook/tree/main/chap01。

1.2. 格式化文本

若想格式化文本，可以使用传统的 printf 函数或 STL iostream 库，两者各有优缺点。

printf 函数继承自 C 语言，50 多年的发展，已让其很高效、灵活和方便。就是格式语法看起来有点晦涩，但习惯过后觉得还行。

```
printf("Hello, %s\n", c_string);
```

printf 的缺点是弱类型安全。printf() 函数 (及其相关函数)，使用 C 的可变参数模型将参数传递给格式化程序。在正常情况下非常高效，但参数类型与其对应的格式说明符不匹配时，可能会产生严重的问题。现代编译器会尽可能地进行类型检查，但编程模型本身存在缺陷，保护也只能到此为止。

STL 的 iostream 库以可读性和运行时性能为代价确保了类型安全。iostream 语法不常见，但很易懂。其重载按位左移操作符 («)，以允许产生格式化输出一连串的对象、操作数和格式化操纵符。

```
cout << "Hello, " << str << endl;
```

iostream 的缺点在于语法和实现方面的复杂性，构建格式化字符串可能会冗长而晦涩。许多格式操纵符在使用后必须重置，否则会产生难以调试的级联格式错误。该库本身庞大而复杂，导致代码比 printf 等效代码大得多，速度也慢很多。

这种令人沮丧的情况让 C++ 程序员别无选择，只能在两个有缺陷的方法中选择其中一个。

How to do it…

新格式库位于 <format> 头文件中。撰写本书时，其仅在 MSVC(Microsoft) 编译器中实现。当阅读本书时，应该在更多的编译器上实现了。如若不然，可以使用其参考实现进行理解 fmt.dev(j.bw.org/fmt)。

格式库基于 Python3 中的 str.format() 方法建模。格式字符串基本上与 Python 中的格式字符串相同，通常可以互换。来看一些简单的例子:

- format() 函数可以接受一个 string_view 格式的字符串和一个可变参数参数包，并返回一个字符串。其函数签名为:

```
template<typename... Args>
string format(string_view fmt, const Args&... args);
```

- format() 返回类型或值的字符串表示形式。例如:

```
string who{ "everyone" };
int ival{ 42 };
double pi{ std::numbers::pi };

format("Hello, {}!\n ", who); // Hello, everyone!
format("Integer: {}\n ", ival); // Integer: 42
format("π: {}\n", pi); // π: 3.141592653589793
```

格式化字符串使用大括号 {} 作为类型安全的占位符，可以将任何兼容类型的值转换为合理的字符串表示形式。

- 可以在格式字符串中包含多个占位符:

```
format("Hello {} {}", ival, who); // Hello 42
                  // everyone
```

- 可以指定替换值的顺序，这对本地化很有用:

```
format("Hello {1} {0}", ival, who); // Hello everyone 42
format("Hola {0} {1}", ival, who); // Hola 42 everyone
```

- 这也可以进行对齐，左 (<)、右 (>) 或中心 (^) 对齐，可以选择性使用填充字符:

```
format("{:.<10}", ival); // 42........
format("{:.>10}", ival); // ........42
format("{:.^10}", ival); // ....42....
```

- 可以设置十进制数值的精度:

```
format("π: {:.5}", pi); // π: 3.1416
```

这是一个丰富而完整的格式化方式，具有 iostream 的类型安全，以及 printf 的性能和简单性，达到了鱼和熊掌兼得的目的。

How it works…

格式库不包括 print() 函数，这是 C++23 计划的一部分。format() 函数本身返回一个字符串对象。若想打印字符串，需要使用 iostream 或 cstdio。:(

可以使用 iostream 输出字符串:

```
cout << format("Hello, {}", who) << "\n";
```

或者使用 cstdio:

```
puts(format("Hello, {}", who).c_str());
```

这两种方法都不理想，但是编写一个简单的 print() 函数并不难。在这个过程中，来了解一些格式库的工作方式。

下面是 print() 函数使用格式库的简单实现:

```
#include <format>
#include <string_view>
#include <cstdio>

template<typename... Args>
void print(const string_view fmt_str, Args&&... args) {
  auto fmt_args{ make_format_args(args...) };
  string outstr{ vformat(fmt_str, fmt_args) };
  fputs(outstr.c_str(), stdout);
}
```

使用与 format() 函数相同的参数。第一个参数是格式字符串的 string_view 对象，后面作为参数的可变参数包。

make_format_args() 函数的作用: 接受参数包并返回一个对象，该对象包含适合格式化的已擦除类型的值。然后，将该对象传递给 vformat()，vformat() 再返回适合打印的字符串。我们使用 fputs() 将值输出到控制台上 (这比 cout 高效得多)。

现在可以使用 print() 函数，来代替 cout << format() 的组合:

```
print("Hello, {}!\n", who);
print("π: {}\n", pi);
print("Hello {1} {0}\n", ival, who);
print("{:.^10}\n", ival);
print("{:.5}\n", pi);
```

输出为:

```
Hello, everyone!
π: 3.141592653589793
Hello everyone 42
....42....
3.1416
```

编译器支持 C++23 的 print() 时，使用 `std::print;` 就能完成所有工作，并且 print() 也会正常工作。

There's more…

能够格式化字符串和原语已经非常好了，但是要使格式库具有完整的功能，就需要对其进行定制，以便与自定义的类一起工作。

例如，这里有两个成员的简单结构体: 分子和分母。将其输出为分数:

```
struct Frac {
  long n;
  long d;
};

int main() {
  Frac f{ 5, 3 };
  print("Frac: {}\n", f);
}
```

编译时，会出现一系列错误，例如:“没有定义的转换运算符…”。让我们来解决它!

当格式系统遇到要转换的对象时，其会寻找具有相应类型的格式化程序对象的特化。标准的特化对于常见的对象，如字符串和数字等。Frac 类型的特化非常简单:

```
template<>
struct std::formatter<Frac>
{
  template<typename ParseContext>
  constexpr auto parse(ParseContext& ctx) {
    return ctx.begin();
  }
  template<typename FormatContext>
  auto format(const Frac& f, FormatContext& ctx) {
    return format_to(ctx.out(), "{0:d}/{1:d}",
    f.n, f.d);
  }
};
```

格式化特化，是具有两个简短模板函数的类:

- parse() 函数解析格式字符串，从冒号之后 (若没有冒号，则在开大括号之后) 直到但不包括结束大括号 (就是指定对象类型的部分)。其接受一个 ParseContext 对象，并返回一个迭代器。这里，可以只返回 begin() 迭代器。因为我们的类型不需要新语法，所以无需准备任何东西。
- format() 函数接受一个 Frac 对象和一个 FormatContext 对象，返回结束迭代器。format_to() 函数可使这变得很容易，其可以接受一个迭代器、一个格式字符串和一个参数包。本例中，参数包是 Frac 类的两个属性，分子和分母。
  需要做的就是提供一个简单的格式字符串“{0}/{1}”以及分子和分母的值 (0 和 1 表示参数的位置)。

现在有了 Frac 的特化，可以将对象传递给 print() 从而获得一个可读的结果:

```
int main() {
  Frac f{ 5, 3 };
  print("Frac: {}\n", f);
}
```

输出为：

```
Frac: 5/3
```

C++20 通过提供高效、方便的类型安全文本格式库，解决了一个长期存在的问题。

1.3. constexpr——使用编译时 vector 和字符串

C++20 允许在新的上下文中使用 constexpr，这些语句可以在编译时计算，从而提高了效率。

How to do it…

其中包括在 constexpr 上下文中使用 string 和 vector 对象的能力。所以，这些对象本身可能不声明为 constexpr，但可以在编译时上下文中使用:

```
constexpr auto use_string() {
  string str{"string"};
  return str.size();
}
```

也可以在 constexpr 上下文中使用算法:

```
constexpr auto use_vector() {
  vector<int> vec{ 1, 2, 3, 4, 5};
  return accumulate(begin(vec), end(vec), 0);
}
```

累加算法的结果可在编译时和 constexpr 的上下文中可用。

How it works…

constexpr 说明符声明了一个可以在编译时求值的变量或函数。C++20 前，这仅限于用文字值初始化的对象，或在有限约束条件下的函数。C++17 允许某种程度上的扩展，而 C++20 进行进一步的扩展。

C++20 开始，标准 string 和 vector 类具有限定的构造函数和析构函数，这是可在编译时使用的前提。所以，分配给 string 或 vector 对象的内存，也必须在编译时释放。

例如，constexpr 函数返回一个 vector，编译时不会出错:

```
constexpr auto use_vector() {
  vector<int> vec{ 1, 2, 3, 4, 5};
  return vec;
}
```

但是若试图在运行时环境中使用结果，会得到一个在常量求值期间分配内存的错误:

```
int main() {
  constexpr auto vec = use_vector();
  return vec[0];
}
```

因为在编译期间分配和释放了 vector 对象，所以该对象在运行时不可用。

另外，在运行时使用一些 vector 对象的适配 constexpr 的方法，比如 size():

```
int main() {
  constexpr auto value = use_vector().size();
  return value;
}
```

因为 size() 方法是 constexpr 限定的，所以表达式可以在编译时求值。

1.4. 安全地比较不同类型的整数

比较不同类型的整数，可能并不总是生成预期的结果。例如:

```
int x{ -3 };
unsigned y{ 7 };
if(x < y) puts("true");
else puts("false");
```

可能希望此代码输出 true，-3 通常小于 7，但代码也会输出 false。

问题是 x 是有符号的 y 是无符号的，标准化的行为是将有符号类型转换为无符号类型进行比较。这似乎违反直觉，不能可靠地将无符号值转换为相同大小的有符号值，因为有符号整数使用 2 的补数表示 (使用最高位作为符号)。给定相同大小的整数，最大有符号值是无符号值的一半。使用这个例子，若整数是 32 位，-3(有符号) 变成 FFFF FFFD(十六进制)，或者 4,294,967,293(无符号十进制)，所以不小于 7。

尝试比较有符号整数值和无符号整数值时，一些编译器可能会发出警告，但大多数编译器不会。

C++20 标准在 <utility> 头文件中包含了一组整数安全的比较函数。

How to do it…

新的整数比较函数可以在 <utility> 头文件找到。每个函数都有两个参数，分别对应于运算符的左边和右边。

```
#include <utility>
int main() {
  int x{ -3 };
  unsigned y{ 7 };
  if(cmp_less(x, y)) puts("true");
  else puts("false");
}
```

cmp_less() 函数给出了我们所期望的结果。-3 小于 7，程序现在输出 true。

<utility> 头文件提供了完整的整数比较函数。假设有 x 和 y 值，可以得到这些比较的结果:

```
cmp_equal(x, y) // x == y is false
cmp_not_equal(x, y) // x != y is true
cmp_less(x, y) // x < y is true
cmp_less_equal(x, y) // x <= y is true
cmp_greater(x, y) // x > y is false
cmp_greater_equal(x, y) // x >= y is false
```

How it works…

下面是 C++20 标准中 cmp_less() 函数的示例实现，可以了解它是如何工作的:

```
template< class T, class U >
constexpr bool cmp_less( T t, U u ) noexcept
{
  using UT = make_unsigned_t<T>;
  using UU = make_unsigned_t<U>;
  if constexpr (is_signed_v<T> == is_signed_v<U>)
    return t < u;
  else if constexpr (is_signed_v<T>)
    return t < 0 ? true : UT(t) < u;
  else
    return u < 0 ? false : t < UU(u);
}
```

UT 和 UU 别名声明为 make_unsigned_t，这是 C++17 引入的一种辅助类型。其允许有符号类型到无符号类型的安全转换。

函数首先测试两个参数是有符号的，还是无符号的。然后，返回一个简单的比较。

然后，测试两边是否有符号。若该带符号值小于零，则可以返回 true 或 false 而不执行比较。否则，将有符号值转换为无符号值进行比较。

相同的逻辑也适用于其他比较函数。

1.5. 三向比较运算符 <=>——进行三种比较

三向比较操作符 (<=>)，通常称为“宇宙飞船”操作符，因为它看起来像一个飞船，是 C++20 添加的新功能。您可能想知道，是因为现有的六个比较操作符有什么问题吗? 并不是，可以继续使用它们。宇宙飞船的目的是提供统一的比较操作符。

常见的双向比较操作符根据比较的结果返回两种状态之一，true 或 false。例如:

```
const int a = 7;
const int b = 42;
static_assert(a < b);
```

a < b 表达式使用小于比较操作符 (<) 来测试 a 是否小于 b。若条件满足，比较操作符返回 true，不满足则返回 false。这种情况下，返回 true，因为 7 小于 42。

这三种方式的比较方式有所不同，会返回三种状态之一。若操作数相等，三向操作符将返回一个等于 0 的值，若左操作数小于右操作数则返回负数，若左操作数大于右操作数则返回正数。

```
const int a = 7;
const int b = 42;
static_assert((a <=> b) < 0);
```

返回值不是整数，是 <compare> 头中的对象，与 0 进行比较。

若操作数为整型，则操作符从 <compare> 库返回 strong_ordered 对象。

```
strong_ordering::equal // operands are equal
strong_ordering::less // lhs is less than rhs
strong_ordering::greater // lhs is greater than rhs
```

若操作数为浮点类型，则操作符返回 partial_ordered 对象:

```
partial_ordering::equivalent // operands are equivelant
partial_ordering::less // lhs is less than rhs
partial_ordering::greater // lhs is greater than rhs
partial_ordering::unordered // if an operand is unordered
```

这些对象通常与 0 进行比较，从而区别于常规比较操作符 (例如，(a <=> b) < 0)。这使得三向比较的结果比常规比较更精确。

若这些看起来有点复杂，没关系。对于大多数应用程序，可能不会直接使用三向操作符，其主要作用在于为对象提供统一的比较运算符。

How to do it…

来看一个简单的类，封装了一个整数，并提供了比较运算符:

```
struct Num {
  int a;
  constexpr bool operator==(const Num& rhs) const
  { return a == rhs.a; }
  constexpr bool operator!=(const Num& rhs) const
  { return !(a == rhs.a); }
  constexpr bool operator<(const Num& rhs) const
  { return a < rhs.a; }
  constexpr bool operator>(const Num& rhs) const
  { return rhs.a < a; }
  constexpr bool operator<=(const Num& rhs) const
  { return !(rhs.a < a); }
  constexpr bool operator>=(const Num& rhs) const
  { return !(a < rhs.a); }
};
```

这样的比较操作符重载并不少见。若比较操作符并非两侧对象的友元函数，情况会变得更加复杂。

新的三向操作符，可以通过一次重载来完成:

```cpp
#include <compare>
struct Num {
  int a;
  constexpr Num(int a) : a{a} {}
  auto operator<=>(const Num&) const = default;
};
```

注意，需要为三向操作符返回类型包含 <compare> 头文件。可以声明一些变量，并通过比较进行测试:

```cpp
constexpr Num a{ 7 };
constexpr Num b{ 7 };
constexpr Num c{ 42 };

int main() {
  static_assert(a < c);
  static_assert(c > a);
  static_assert(a == b);
  static_assert(a <= b);
  static_assert(a <= c);
  static_assert(c >= a);
  static_assert(a != c);
  puts("done.");
}
```

对于每一次比较，编译器都会自动使用 <=> 操作符。

因为默认的 <=> 操作符已经是 constexpr 安全的，所以不需要在成员函数中这样声明。

How it works…

<=> 重载利用了 C++20 的一个新概念，即重写的表达式。重载解析期间，编译器根据一组规则重写表达式。例如，a < b，编译器会将其重写为 (a <=> b < 0)，以便其可与成员操作符可以一起工作。编译器将重写 <=> 操作符的每个相关比较表达式，其中不包含具体的操作符。

事实上，我们不再需要非成员函数来处理与左侧兼容类型的比较。编译器将合成一个与成员操作符一起工作的表达式。例如，42 > a，编译器将合成一个操作符颠倒的表达式 (a <=> 42 < 0)，以便与成员操作符一起工作。

> **Note**
>
> <=> 的优先级高于其他比较运算符，因此它总是先求值。所有比较运算符都从左到右计算。

There's more…

默认操作符可以很好地处理各种类，包括多个不同类型成员的类:

```cpp
struct Nums {
  int i;
  char c;
```

```
  float f;
  double d;
  auto operator<=>(const Nums&) const = default;
};
```

但若有一个更复杂的类型呢? 下面是一个简单 Frac 类:

```
struct Frac {
  long n;
  long d;
  constexpr Frac(int a, int b) : n{a}, d{b} {}
  constexpr double dbl() const {
    return static_cast<double>(n) /
    static_cast<double>(d);
  }
  constexpr auto operator<=>(const Frac& rhs) const {
    return dbl() <=> rhs.dbl();
  };
  constexpr auto operator==(const Frac& rhs) const {
    return dbl() <=> rhs.dbl() == 0;
  };
};
```

本例中，需要定义运算符 <=> 的重载，因为数据成员不是独立的标量。重载很简单，而且效果很好。

注意，还需要一个运算符 == 重载。因为表达式重写规则不会使用自定义操作符 <=> 重载重写 == 和!=，所以需要定义操作符 ==，从而编译器会根据需要重写!= 表达式。

现在可以定义一些对象:

```
constexpr Frac a(10,15); // compares equal with 2/3
constexpr Frac b(2,3);
constexpr Frac c(5,3);
```

可以用正常的比较运算符来测试，结果如预期的一样:

```
int main() {
  static_assert(a < c);
  static_assert(c > a);
  static_assert(a == b);
  static_assert(a <= b);
  static_assert(a <= c);
  static_assert(c >= a);
  static_assert(a != c);
}
```

三向操作符的强大之处在于，能够简化类中的比较重载。与单独重载每个操作符相比，其简单并且高效。

1.6. <version> 头文件——查找特性测试宏

只要添加了新特性，C++ 就会提供了某种形式的特性测试宏。C++20 起这个过程标准化了，所有库特性的测试宏，都会放到 <version> 头文件中。这将使测试代码中的新特性变得更加容易。

这是一个非常有用的功能，使用起来非常简单。

How to do it…

所有特性测试宏都以前缀__cpp_开头。库特性以__cpp_lib_开头。语言特性测试宏通常由编译器定义，库特性测试宏定义在新的 <version> 头文件中。可以像使用其他预处理器宏一样使用:

```
#include <version>
#ifdef __cpp_lib_three_way_comparison
# include <compare>
#else
# error Spaceship has not yet landed
#endif
```

某些情况下，可以使用__has_include 预处理器操作符 (C++17) 来测试是否包含了某个文件。

```
#if __has_include(<compare>)
# include <compare>
#else
# error Spaceship has not yet landed
#endif
```

因为它是一个预处理器指令，所以可以使用__has_include 来测试头文件是否存在。

How it works…

通常，可以通过使用 #ifdef 或 #if 定义测试非零值来使用特性测试宏。每个特性测试宏都有一个非零值，对应于标准委员会接受它的年份和月份。例如，__cpp_lib_three_way_comparison 宏的值为 2011907，意味着其在 2019 年 7 月采纳。

```
#include <version>
#ifdef __cpp_lib_three_way_comparison
cout << "value is " << __cpp_lib_three_way_comparison << "\n"
#endif
```

输出为：

```
$ ./working
value is 201907
```

宏的值可能在一些的情况下很有用，比如某个特性发生了变化，而程序会依赖于这些变化。通常，可以安全地忽略该值，只使用 #ifdef 测试非零即可。

一些网站维护功能测试宏的完整列表，这里推荐 cppreference(https://j.bw.org/cppfeature)。

1.7. 概念 (concept) 和约束 (constraint)——创建更安全的模板

模板对于编写适用于不同类型的代码非常有用。例如，此函数将适用于任何数字类型:

```
template <typename T>
T arg42(const T & arg) {
  return arg + 42;
}
```

当尝试用非数字类型调用它时，会发生什么呢?

```
const char * n = "7";
cout << "result is " << arg42(n) << "\n";
```

输出为:

```
Result is ion
```

这样编译和运行没有错误，但结果无法预测。该调用非常危险，很容易造成崩溃或成为漏洞。我更希望编译器生成一个错误消息，这样就可以提前修复代码。

有了概念后，就可以这样写:

```
template <typename T>
requires Numeric<T>
T arg42(const T & arg) {
  return arg + 42;
}
```

require 关键字是 C++20 的新特性，将约束应用于模板。Numeric 是一个只接受整数和浮点类型的概念的名称。现在，当用非数字参数编译这段代码时，就会得到编译错误:

```
error: 'arg42': no matching overloaded function found
error: 'arg42': the associated constraints are not satisfied
```

这样的错误消息比大多数编译器错误有用得多。

仔细看看如何在代码中使用概念和约束。

How to do it…

概念只是一个命名的约束。上面的 Numeric 概念是这样的:

```
#include <concepts>
template <typename T>
concept Numeric = integral<T> || floating_point<T>;
```

此概念需要类型 T，满足 std::integral 或 std::float_point 预定义概念。这些概念包含在 <concepts> 头文件中。

概念和约束可以用在类模板、函数模板或变量模板中。我们已经看到了一个约束函数模板，这里是一个简单的约束类模板示例:

```
template<typename T>
requires Numeric<T>
struct Num {
  T n;
  Num(T n) : n{n} {}
};
```

下面是一个简单的变量模板示例:

```
template<typename T>
requires floating_point<T>
T pi{3.14159265358979323851};
```

可以在任何模板上使用概念和约束。简单起见，我们将在这些示例中使用函数模板。

- 约束可以使用概念或类型特征来评估类型的特征。可以使用 <type_traits> 头文件中找到的任何类型特征，只要可以返回 bool 类型。
  例如:

```
template<typename T>
requires is_integral<T>::value // value is bool
constexpr double avg(vector<T> const& vec) {
  double sum{ accumulate(vec.begin(), vec.end(),
    0.0)
  };
  return sum / vec.size();
}
```

- require 关键字是 C++20 中新出现的，为模板参数引入了一个约束。本例中，约束表达式根据类型特征 is_integral 测试模板参数。
- 可以使用 <type_traits> 头文件中预定义的特性，或者自定义的特性，就像模板变量一样。为了在约束中使用，该变量必须返回 constexpr bool。例如:

```
template<typename T>
constexpr bool is_gt_byte{ sizeof(T) > 1 };
```

这定义了一个名为 is_gt_byte 的类型特征，该特性使用 sizeof 操作符来测试类型 T 是否大于 1 字节。

- 概念只是一组命名的约束。例如:

```
template<typename T>
concept Numeric = is_gt_byte<T> &&
  (integral<T> || floating_point<T>);
```

这定义了一个名为 Numeric 的概念，使用 is_gt_byte 约束，以及 <concepts> 头文件中的 floating_point 和 integral 概念。可以用它来约束模板，使其只接受大于 1 字节的数字类型。

```
template<Numeric T>
```

```
T arg42(const T & arg) {
  return arg + 42;
}
```

有读者会注意到，我在模板声明中应用了约束，而不是在 require 表达式中的单独一行中。有几种方法可以使用概念，让我们看看它是如何工作的。

How it works…

使用概念或约束有几种不同的方法:

- 可以用 require 关键字定义一个概念或约束:

```
template<typename T>
requires Numeric<T>
T arg42(const T & arg) {
  return arg + 42;
}
```

- 可以在模板声明中使用概念:

```
template<Numeric T>
T arg42(const T & arg) {
  return arg + 42;
}
```

- 可以在函数签名中使用 require 关键字:

```
template<typename T>
T arg42(const T & arg) requires Numeric<T> {
  return arg + 42;
}
```

- 可以在参数列表中使用概念来简化函数模板:

```
auto arg42(Numeric auto & arg) {
  return arg + 42;
}
```

如何进行选择，可能就只是偏好问题。

There's more…

该标准使用术语连接、分离和原子来描述可用于构造约束的表达式类型。我们来定义一下这些术语。

可以使用 && 和 || 操作符组合概念和约束。这些组合分别称为连接词和析取词，可以把它们看成逻辑的 AND 和 OR。

约束连接符是通过使用带有两个约束的 && 操作符形成的:

```
Template <typename T>
concept Integral_s = Integral<T> && is_signed<T>::value;
```

只有当 && 运算符的两边都满足时，连接符才满足。从左到右求值。连接的操作数可短路，若左边的约束不满足，右边的约束就不会求值。

约束析取通过使用 || 操作符和两个约束形成:

```
Template <typename T>
concept Numeric = integral<T> || floating_point<T>;
```

若 || 运算符的任意一边满足，则析取条件满足。从左到右求值。连接的操作数是短路的，若左边的约束得到满足，右边的约束就不会求值。

原子约束是返回 bool 类型的表达式，不能进一步分解。换句话说，不是一个连接或析取。

```
template<typename T>
concept is_gt_byte = sizeof(T) > 1;
```

也可以在原子约束中使用逻辑!(NOT) 操作符:

```
template<typename T>
concept is_byte = !is_gt_byte<T>;
```

果然，! 运算符将 bool 表达式的值颠倒到! 的右侧。

当然，可以将所有这些表达式类型组合成一个更大的表达式。可以在下面的例子中看到这些约束表达式的例子:

```
template<typename T>
concept Numeric = is_gt_byte<T> &&
(integral<T> || floating_point<T>);
```

来分析一下。

子表达式 (integral<T> || floating_point<T>) 是一个析取。子表达式 is_gt_byte<T> &&(…) 是一个连接。每个子表达式 integral<T>，float _point<T> 和 is_gt_byte<T>，都是原子表达式。

这些区别主要是为了描述的目的。虽然了解细节是很好的习惯，但在编写代码时，可将它们视为逻辑 ||，&& 和! 操作符。

概念和约束是 C++ 标准的一个很受欢迎的补充，非常期待在未来的项目中使用到它们。

1.8. 模块——避免重新编译模板库

头文件在 C 语言出现之初就存在了。最初，主要用于文本替换宏和翻译单元之间的外部符号链接。随着模板的引入，C++ 利用头文件来放置实现的代码。因为模板需要重新编译以适应相应的特化，所以多年来我们一直在这样使用头文件。随着 STL 多年的发展，这些头文件的体积也在不断增长。目前这种情况已经难以处理，并且可扩展性越来越差。

头文件通常包含比模板更多的内容，通常包含系统所需的配置宏和其他符号。随着头文件数量的增加，符号冲突的机率也在增加。考虑到使用更多的宏时，问题就更大了，它们不受命名空间的限制，也不受其他形式的类型安全限制。

C++20 中使用了模块解决了这个问题。

How to do it…

读者们可能习惯于创建这样的头文件:

```
#ifndef BW_MATH
#define BW_MATH
namespace bw {
  template<typename T>
  T add(T lhs, T rhs) {
    return lhs + rhs;
  }
}
#endif // BW_MATH
```

这个极简的例子说明了模块解决的几个问题。BW_MATH 符号用作包含守卫，唯一目的是防止头文件多次包含，但其符号贯穿整个翻译单元。当在源文件中包含这个头文件时，看起来像这样:

```
#include "bw-math.h"
#include <format>
#include <string>
#include <iostream>
```

现在 BW_MATH 符号可用于包含的其他头文件，以及其他头文件包含的头文件等。这就会增大冲突的机率，并且编译器不能检查这些冲突。他们是宏，在编译器有看到它们前，就已经使用预处理器翻译了。

现在打开头文件看看，即对模板函数进行了解:

```
template<typename T>
T add(T lhs, T rhs) {
  return lhs + rhs;
}
```

因为是模板，每次使用 add() 时，编译器需要进行特化。模板函数每次调用时，都需要解析和特化。这就是为什么模板实现要放在头文件中的原因，源代码必须在编译时可见。随着 STL 的发展和壮大，现在已经有许多大型模板类和函数，这就成为了一个可扩展性的问题。

模块解决了这些问题，以及其他问题。

作为一个模块，bw-math.h 变成了 bw-math.ixx(MSVC 的命名约定)，内容如下:

```
export module bw_math;
export template<typename T>
T add(T lhs, T rhs) {
  return lhs + rhs;
}
```

注意，导出的符号是模块名 bw_math 和函数名 add()。这可使命名空间保持干净。

这种用法也更简洁。在 module-test.cpp 中使用时，可以这样:

```
import bw_math;
import std.core;

int main() {
```

```
  double f = add(1.23, 4.56);
  int i = add(7, 42);
  string s = add<string>("one ", "two");

  cout <<
    "double: " << f << "\n" <<
    "int: " << i << "\n" <<
    "string: " << s << "\n";
}
```

import 声明用在可能使用 #include 预处理器指令的地方。会从模块中导入符号表进行链接。

示例的输出如下所示:

```
$ ./module-test
double: 5.79
int: 49
string: one two
```

模块版本的工作原理与头文件中相同，只是更干净、更高效。

> **Note**
>
> 编译后的模块包含单独的元数据文件 (module-name.ifc 是 MSVC 中的命名约定)，描述了模块接口，允许模块支持模板。元数据包含编译器创建模板特化所需的所有信息。

How it works…

导入和导出声明是模块实现的核心，再来看看 bw-math.ixx:

```
export module bw_math;
export template<typename T>
T add(T lhs, T rhs) {
  return lhs + rhs;
}
```

请注意这两个导出声明。第一个函数使用 export module bw_math 导出模块本身，这将翻译单元声明为一个模块。每个模块文件的顶部，以及其他语句之前必须有一个模块声明。第二个导出使函数 add() 对模块使用者可用。

若模块需要 #include 指令，或者其他全局代码段，需要一个简单的模块声明:

```
module;
#define SOME_MACRO 42
#include <stdlib.h>
export module bw_math;
...
```

module; 声明在文件顶部的一行中单独引入了一个全局模块，只有预处理器指令可以出现在全局模块片段中。之后必须立即声明一个标准模块 (export module bw_math;) 和其余的模块内容。来看看它是如何工作的:

- 导出声明使一个符号对模块使用者可见，即导入模块的代码，符号默认为 private。

```
export int a{7}; // visible to consumer
int b{42}; // not visible
```

- 可以导出一个代码块，像这样:

```
export {
  int a() { return 7; }; // visible
  int b() { return 42; }; // also visible
}
```

- 导出命名空间:

```
export namespace bw { // all of the bw namespace is
  visible
  template<typename T>
  T add(T lhs, T rhs) { // visible as bw::add()
    return lhs + rhs;
  }
}
```

- 或者，可以从命名空间中导出单独的符号:

```
namespace bw { // all of the bw namespace is visible
  export template<typename T>
  T add(T lhs, T rhs) { // visible as bw::add()
    return lhs + rhs;
  }
}
```

- import 声明在调用代码中导入一个模块:

```
import bw_math;
int main() {
  double f = bw::add(1.23, 4.56);
  int i = bw::add(7, 42);
  string s = bw::add<string>("one ", "two");
}
```

- 甚至可以导入一个模块，并将其导出传递给调用代码:

```
export module bw_math;
export import std.core;
```

export 关键字必须在 import 关键字之前。

std.core 模块现在可以在调用代码中使用:

```
import bw_math;
using std::cout, std::string, std::format;
```

```
3
4  int main() {
5    double f = bw::add(1.23, 4.56);
6    int i = bw::add(7, 42);
7    string s = bw::add<string>("one ", "two");
8
9    cout <<
10     format("double {} \n", f) <<
11     format("int {} \n", i) <<
12     format("string {} \n", s);
13 }
```

模块是头文件的简化、直接的替代品，很多人都期待着模块的广泛使用。这样就可以大大减少了对头文件的依赖。

Note

撰写本文时，模块的唯一完整实现是在 MSVC 的预览版中。对于其他编译器，模块扩展名 (.ixx) 可能不同。此外，使用合并的 std.core 模块 (MSVC 版本中将 STL 作为模块实现的一部分)，其他编译器可能不使用这个约定。在完全兼容的实现发布时，可能会发生变化。

示例文件中，包含了基于格式的 print() 函数的模块版本，这适用于 MSVC 的当前预览版本。当其他系统支持模块规范，当前的代码可能需要一些修改才能在其他系统上工作。

1.9. 范围容器中创建视图

范围库是 C++20 中添加的重要特性，可为过滤和处理容器提供了一种新的范例。范围为更有效和可读的代码提供了清晰和直观的构建块。

先来定义几个术语:

- “范围”是一个可以迭代的对象的集合，支持 begin() 和 end() 迭代器的结构都是范围。这包括大多数 STL 容器。
- “视图”是转换另一个基础范围的范围。视图是惰性的，只在范围迭代时操作。视图从底层范围返回数据，不拥有任何数据。视图的运行时间复杂度是 O(1)。
- 视图适配器是一个对象，可接受一个范围，并返回一个视图对象。视图适配器可以使用 | 操作符连接到其他视图适配器。

Note

<ranges> 中定义了 std::ranges 和 std::ranges::view 命名空间。这貌似有些复杂，标准包含了 std::ranges::view 的别名，即 std::view。我还是觉得那很麻烦。对于这个示例，将使用以下别名，我觉得这种方式更优雅:

```
1 namespace ranges = std::ranges; // save the fingers!
2 namespace views = std::ranges::views;
```

这适用于本节中的代码。

How to do it…

范围和视图类在 <ranges> 头文件中，看看如何使用它们:

- 将视图 (View) 应用于范围 (Range)，如下所示:

```
const vector<int> nums{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
auto result = ranges::take_view(nums, 5);
for (auto v: result) cout << v << " ";
```

输出为:

```
1 2 3 4 5
```

ranges::take_view(range, n)：返回前 n 个元素的视图。

也可以使用视图适配器版本的 take_view():

```
auto result = nums | views::take(5);
for (auto v: result) cout << v << " ";
```

输出为:

```
1 2 3 4 5
```

视图适配器位于 std::ranges::views 命名空间中。视图适配器从 | 操作符的左侧获取范围操作数，很像 << 操作符的 iostreams 用法。| 操作赋会从左向右求值。

- 因为视图适配器可迭代，所以也有资格作为范围。这使得它们可以连续使用:

```
const vector<int> nums{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
auto result = nums | views::take(5) | views::reverse;
```

输出为:

```
5 4 3 2 1
```

- filter() 视图使用了一个谓词函数:

```
auto result = nums |
  views::filter([](int i){ return 0 == i % 2; });
```

输出为:

```
2 4 6 8 10
```

- transform() 视图使用了一个转换函数:

```
auto result = nums |
  views::transform([](int i){ return i * i; });
```

输出为:

```
1 4 9 16 25 36 49 64 81 100
```

- 视图和适配器可适用于任何范围:

```
cosnt vector<string>
words{ "one", "two", "three", "four", "five" };
auto result = words | views::reverse;
```

输出为:

```
five four three two one
```

- 范围库还包括一些工厂函数。iota 工厂将生成一系列递增的值:

```
auto rnums = views::iota(1, 10);
```

输出为:

```
1 2 3 4 5 6 7 8 9
```

- iota(value, bound) 函数的作用是: 生成一个从 value 开始，到 bound 之前结束的序列。若省略了 bound，序列则为无穷大:

```
auto rnums = views::iota(1) | views::take(200);
```

输出为:

```
1 2 3 4 5 6 7 8 9 10 11 12 [···] 196 197 198 199 200
```

范围、视图和视图适配器灵活好用。让我们更深入地了解一下。

How it works···

为了满足范围的基本要求，对象必须至少有两个迭代器 begin() 和 end()，其中 end() 迭代器是一个哨兵，用于确定 Range 的端点。大多数 STL 容器都符合范围的要求，包括 string、vector、array、map 等。不过，容器适配器除外，如 stack 和 queue，因为它们没有起始迭代器和结束迭代器。

视图是一个对象，操作一个范围并返回一个修改后的范围。视图为惰性操作的，不包含自己的数据。不保留底层数据的副本，只是根据需要返回底层元素的迭代器。来看个代码段:

```
vector<int> vi { 0, 1, 2, 3, 4, 5 };
ranges::take_view tv{vi, 2};
for(int i : tv) {
  cout << i << " ";
}
cout << "\n";
```

输出为:

```
0 1
```

本例中，take_view 对象接受两个参数，一个范围 (在本例中是 vector<int> 对象) 和一个计数，结果是一个包含 vector 中第一个 count 对象的视图。在 for 循环的迭代过程求值时，take_view 对象会根据需要返回指向 vector 对象元素的迭代器。

在此过程中不修改 vector 对象。

范围命名空间中的许多视图在视图命名空间中都有相应的范围适配器，这些适配器可以使用按位或 (|) 操作符作为管道:

```
vector<int> vi { 0, 1, 2, 3, 4, 5 };
auto tview = vi | views::take(2);
for(int i : tview) {
  cout << i << " ";
}
cout << "\n";
```

输出为:

```
0 1
```

如预期的那样，| 操作符从左到右求值。因为范围适配器的结果是另一个范围，所以这些适配器表达式可以链接起来:

```
vector<int> vi { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
auto tview = vi | views::reverse | views::take(5);
for(int i : tview) {
  cout << i << " ";
}
cout << "\n";
```

输出为:

```
9 8 7 6 5
```

标准库包括一个过滤视图，用于定义简单的过滤器:

```
vector<int> vi { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
auto even = [](long i) { return 0 == i % 2; };
auto tview = vi | views::filter(even);
```

输出为:

```
0 2 4 6 8
```

还包括一个 transform 视图，与 transform 函数一起用于转换结果:

```
vector<int> vi { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
auto even = [](int i) { return 0 == i % 2; };
auto x2 = [](auto i) { return i * 2; };
auto tview = vi | views::filter(even) | views::transform(x2);
```

输出为:

```
0 4 8 12 16
```

库中有相当多有用的视图和视图适配器。可访问 (https://j.bw.org/ranges) 获取完整的列表。

There's more…

从 C++20 开始，<algorithm> 头文件中的大多数算法都会基于范围。这些版本在 <algorithm> 头文件中，但是在 std::ranges 命名空间中，这将它们与传统算法区别开来。

所以，无需再调用带有两个迭代器的算法:

```
sort(v.begin(), v.end());
```

现在可以用范围来调用:

```
ranges::sort(v);
```

这当然更方便，但它真正意义在哪里呢?

回想一下，要对 vector 的一部分排序时的情况。可以用老方法来做:

```
sort(v.begin() + 5, v.end());
```

这将对 vector 的前 5 个元素进行排序。范围版本中，可以使用视图来跳过前 5 个元素:

```
ranges::sort(views::drop(v, 5));
```

甚至可以组合视图:

```
ranges::sort(views::drop(views::reverse(v), 5));
```

也可以使用范围适配器作为 ranges::sort 的参数:

```
ranges::sort(v | views::reverse | views::drop(5));
```

用传统的排序算法和 vector 迭代器来完成:

```
ranges::sort(v.rbegin() + 5, v.rend());
```

虽然这更简单，也很好理解，但我觉得范围适配器版本的会更直观。

可以在 cppreference(https://j.bw.org/algoranges) 上找到限制使用范围的完整算法列表。

这个示例中，我们只触及了范围和视图的冰山一角。这个特性是许多不同团队十多年来工作的结晶，我希望它能从根本上改变我们使用 STL 容器的方式。

第 2 章　STL 的泛型特性

本章介绍了 STL 的特性和技术。这些大多是近几年引入的新功能，还没有广泛使用，可以简化代码量，并增加可读性。

- span 类——使 C 语言数组更安全
- 结构化绑定
- if 和 switch 语句中初始化变量
- 模板参数推导
- if constexpr——简化编译时决策

2.1. 相关准备

本章所有的例子和源代码都可以在本书的 GitHub 库中找到，网址是https://github.com/PacktPublishing/CPP-20-STL-Cookbook/tree/main/chap02。

2.2. span 类——使 C 语言数组更安全

对于 C++20，std::span 类是一个包装器，可在连续的对象序列上创建视图。span 没有属于自己的数据，其引用底层结构中的数据。可以把它看作 C 数组的 string_view，底层结构可以是 C 数组、vector 或 STL array。

How to do it…

可以使用兼容的连续存储结构创建 span，最常见的的是 C 数组。例如，将一个 C 数组传递给函数，该数组将降级为指针，并且函数没有办法直接了解数组的长度:

```
void parray(int * a); // loses size information
```

若用 span 形参定义函数，可以传递一个 C 数组，其将会升级为 span。下面是一个模板函数，其接受一个 span 参数，并以元素和字节为单位输出:

```
template<typename T>
void pspan(span<T> s) {
  cout << format("number of elements: {}\n", s.size());
  cout << format("size of span: {}\n", s.size_bytes());
  for(auto e : s) cout << format("{} ", e);
  cout << "\n";
}
```

可以传递一个 C 数组给这个函数，会自动提升为 span:

```
int main() {
  int carray[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
  pspan<int>(carray);
}
```

输出为:

```
number of elements: 10
number of bytes: 40
1 2 3 4 5 6 7 8 9 10
```

span 的目的是封装原始数据，并以最小的开销保证安全性和实用性。

How it works…

span 类本身不拥有数据，数据属于底层数据结构。span 本质上是底层数据的视图，并提供了一些有用的成员函数。

 头文件中定义的 span 类看起来像这样:

```
template<typename T, size_t Extent = std::dynamic_extent>
class span {
  T * data;
  size_t count;
  public:
  ...
};
```

Extent 参数是一个 constexpr size_t 类型的常量，在编译时计算。其要么是底层数据中的元素数量，要么是 std::dynamic_extent 常量，这表明其大小可变。这允许 span 使用底层结构 (如 vector)，但大小并不总是相同。

所有成员函数都是 constexpr 和 const 限定的。成员函数包括:

公共成员函数	返回值
T& front()	第一个元素
T& back()	最后一个元素
T& operator[]	索引位置的元素
T* data()	指向序列头部的指针
iterator begin()	指向第一个元素的迭代器
iterator end()	指向最后一个元素后面的迭代器
iterator rbegin()	指向第一个元素的反向迭代器
iterator rend()	指向最后一个元素后面的反向迭代器
size_t size()	序列中元素的数量
size_t size_bytes()	序列所占字节数
bool empty()	空为真
span<T>first<count>() span<T>first(count)	由序列的第一个计数元素组成的子 span

span<T>last<count>() span<T>last(count)	返回最后一个 count 元素的子 span
span<T>subspan(offset, count)	返回以 offset 偏移对头进行偏移后的 count 个元素组成的子 span

重要的 Note

span 类只是一个简单的包装器，不执行边界检查。若尝试访问 n 个元素中的元素 n+1，结果就是未定义的，所以最好不要这样做。

2.3. 结构化绑定

结构化绑定可以很容易地将值解压缩到单独的变量中，从而提高代码的可读性。

使用结构化绑定，可以直接将成员值分配给变量:

```
things_pair<int,int> { 47, 9 };
auto [this, that] = things_pair;
cout << format("{} {}\n", this, that);
```

输出为:

```
47 9
```

How to do it…

- 结构化绑定使用 pair、tuple、array 和 struct。C++20 起，还会包括位域。下面的例子使用了 C 数组:

```
int nums[] { 1, 2, 3, 4, 5 };
auto [ a, b, c, d, e ] = nums;
cout << format("{} {} {} {} {}\n", a, b, c, d, e);
```

输出为:

```
1 2 3 4 5
```

因为结构化绑定使用自动类型推断，所以类型必须是 auto。各个变量的名称都在方括号内，[a, b, c, d, e]。

这个例子中，int 型 c 数组 nums 包含五个值。使用结构化绑定将这五个值赋给变量 (a、b、c、d 和 e)。

- 这也适用于 STL 数组对象:

```
array<int,5> nums { 1, 2, 3, 4, 5 };
auto [ a, b, c, d, e ] = nums;
cout << format("{} {} {} {} {}\n", a, b, c, d, e);
```

输出为:

```
1 2 3 4 5
```

- 或者将它与 tuple 一起使用:

```
tuple<int, double, string> nums{ 1, 2.7, "three" };
auto [ a, b, c ] = nums;
cout << format("{} {} {}\n", a, b, c);
```

输出为:

```
1 2.7 three
```

- 将它与结构体一起使用时，将按照定义的顺序接受变量:

```
struct Things { int i{}; double d{}; string s{}; };
Things nums{ 1, 2.7, "three" };
auto [ a, b, c ] = nums;
cout << format("{} {} {}\n", a, b, c);
```

输出为:

```
1 2.7 three
```

- 可以使用带有结构化绑定的引用，可以修改绑定容器中的值，同时避免数据复制:

```
array<int,5> nums { 1, 2, 3, 4, 5 };
auto& [ a, b, c, d, e ] = nums;
cout << format("{} {}\n", nums[2], c);
c = 47;
cout << format("{} {}\n", nums[2], c);
```

输出为:

```
3 3
47 47
```

因为变量绑定为引用，所以可以给 c 赋一个值，这也会改变数组中的值 (nums[2])。

- 可以声明数组 const 来避免值的改变:

```
const array<int,5> nums { 1, 2, 3, 4, 5 };
auto& [ a, b, c, d, e ] = nums;
c = 47; // this is now an error
```

或者可以声明为 const 绑定来达到同样的效果，同时允许在其他地方更改数组，从而避免复制数据:

```
array<int,5> nums { 1, 2, 3, 4, 5 };
const auto& [ a, b, c, d, e ] = nums;
c = 47; // this is also an error
```

How it works…

结构化绑定使用 auto 类型推断将结构解压缩到变量中。其独立地确定每个值的类型，并为每个变量分配相应的类型。

- 因为结构化绑定使用 auto 类型推断，所以不能为绑定指定类型，必须使用 auto。若使用一个类型进行绑定，会得到相应的错误信息:

```
array<int,5> nums { 1, 2, 3, 4, 5 };
int [ a, b, c, d, e ] = nums;
```

输出为:

```
error: structured binding declaration cannot have type 'int'
note: type must be cv-qualified 'auto' or reference to
cv-qualified 'auto'
```

当我尝试使用 int 与结构化绑定一起使用，上面就是来自 GCC 的报错。

- 对于函数的返回类型，通常使用结构化绑定:

```
struct div_result {
  long quo;
  long rem;
};

div_result int_div(const long & num, const long & denom)
{
  struct div_result r{};
  r.quo = num / denom;
  r.rem = num % denom;
  return r;
}

int main() {
  auto [quo, rem] = int_div(47, 5);
  cout << format("quotient: {}, remainder {}\n",
```

```
    quo, rem);
}
```

输出为:

```
quotient: 9, remainder 2
```

- 因为 map 容器类为每个元素返回一个 pair，所以使用结构化绑定来检索键/值对就很方便:

```
map<string, uint64_t> inhabitants {
  { "humans", 7000000000 },
  { "pokemon", 17863376 },
  { "klingons", 24246291 },
  { "cats", 1086881528 }
};

// I like commas
string make_commas(const uint64_t num) {
  string s{ std::to_string(num) };
  for(int l = s.length() - 3; l > 0; l -= 3) {
    s.insert(l, ",");
  }
  return s;
}

int main() {
  for(const auto & [creature, pop] : inhabitants) {
    cout << format("there are {} {}\n",
    make_commas(pop), creature);
  }
}
```

输出为:

```
there are 1,086,881,528 cats
there are 7,000,000,000 humans
there are 24,246,291 klingons
there are 17,863,376 pokemon
```

使用结构化绑定来解包结构可以使代码更清晰、更容易维护。

2.4. if 和 switch 语句中初始化变量

C++17 起，if 和 switch 具有初始化语法，就像 C99 开始的 for 循环一样。可以在限制条件中，确定变量的使用范围。

How to do it…

读者们可能习惯于这样的代码:

```
const string artist{ "Jimi Hendrix" };
size_t pos{ artist.find("Jimi") };
if(pos != string::npos) {
  cout << "found\n";
} else {
  cout << "not found\n";
}
```

这使得变量 pos 暴露在条件语句的作用域之外，需要对其进行管理，否则它可能与使用相同符号的其他变量发生冲突。

现在，可以把初始化表达式放在 if 条件中:

```
if(size_t pos{ artist.find("Jimi") }; pos != string::npos) {
  cout << "found\n";
} else {
  cout << "not found\n";
}
```

pos 变量的作用域限制在条件变量的作用域内。

How it works…

初始化表达式可以在 if 或 switch 语句中使用，以下是例子。

- 使用带有初始化表达式的 if 语句:

```
if(auto var{ init_value }; condition) {
  // var is visible
} else {
  // var is visible
}
// var is NOT visible
```

在初始化表达式中定义的变量，整个 if 语句的范围内都可见，包括 else 子句。出了 if 语句的作用域，该变量将不再可见，并且会调用相关的析构函数。

- 使用初始化表达式的 switch:

```
switch(auto var{ init_value }; var) {
  case 1: ...
  case 2: ...
  case 3: ...
  ...
  Default: ...
}
// var is NOT visible
```

初始化表达式中定义的变量，在整个 switch 语句的作用域中可见，包括所有 case 子句和 default 子句 (有的话)。出了 switch 语句的作用域，变量将不再可见，并且会调用相关的析构函数。

There's more…

一个有趣的用例是限制锁定互斥锁的 lock_guard 的作用域。使用初始化表达式，会让代码变得更简单:

```
if (lock_guard<mutex> lg{ my_mutex }; condition) {
  // interesting things happen here
}
```

lock_guard 在构造函数中锁定互斥量，在析构函数中解锁互斥量。过去，必须删除它或将整个 if 语句括在一个额外的大括号块中。现在，当 lock_guard 超出 if 语句的作用域时，将自动销毁。

另一个用例可能是使用输出参数的遗留接口，就像下面示例，来自于 SQLite:

```
if(
  sqlite3_stmt** stmt,
  auto rc = sqlite3_prepare_v2(db, sql, -1, &_stmt,
  nullptr);
  !rc) {
  // do SQL things
} else { // handle the error
  // use the error code
  return 0;
}
```

这里，可以将句柄和错误代码本地化到 if 语句的范围内。否则，需要全局化地管理这些对象。

使用初始化表达式将有助于保持代码紧凑、整洁、紧凑、易阅读。重构和管理代码也会变得更加容易。

2.5. 模板参数推导

当模板函数或类模板构造函数 (C++17 起) 的实参类型足够清楚，无需使用模板实参，编译器就能理解时，就会进行模板实参推导。这个功能有一定的规则，但主要规则是很直观的。

How to do it…

通常，当使用具有明确兼容参数的模板时，模板参数推断会自动发生。让我们看一些例子。

- 函数模板中，参数推断通常是这样:

```
template<typename T>
const char * f(const T a) {
  return typeid(T).name();
}
int main() {
  cout << format("T is {}\n", f(47));
  cout << format("T is {}\n", f(47L));
  cout << format("T is {}\n", f(47.0));
  cout << format("T is {}\n", f("47"));
  cout << format("T is {}\n", f("47"s));
}
```

输出为:

```
T is int
T is long
T is double
T is char const *
T is class std::basic_string<char...
```

因为类型很容易识别，所以没有理由在函数调用中指定 f<int>(47) 这样的模板形参。编译器可以从实参中推导出 <int> 类型。

> **Note**
>
> 上面的输出显示了有意义的类型名，大多数编译器都会使用简写，比如 i 表示 int, PKc 表示 const char *，等等。

- 这同样适用于多个模板参数:

```
template<typename T1, typename T2>
string f(const T1 a, const T2 b) {
  return format("{} {}", typeid(T1).name(),
  typeid(T2).name());
}

int main() {
  cout << format("T1 T2: {}\n", f(47, 47L));
  cout << format("T1 T2: {}\n", f(47L, 47.0));
  cout << format("T1 T2: {}\n", f(47.0, "47"));
}
```

输出为:

```
T1 T2: int long
T1 T2: long double
T1 T2: double char const *
```

这里编译器同时推导出了 T1 和 T2 的类型。

- 注意，类型必须与模板兼容。例如，不能从字面量获取类型:

```
template<typename T>
const char * f(const T& a) {
  return typeid(T).name();
}
int main() {
  int x{47};
  f(47); // this will not compile
  f(x); // but this will
```

```
}
```

- C++17 起，可以对类使用模板参数推导:

```
pair p(47, 47.0); // deduces to pair<int, double>
tuple t(9, 17, 2.5); // deduces to tuple<int, int, double>
```

这消除了对 std::make_pair() 和 std::make_tuple() 的需求，现在可以直接初始化这些类，无需显式的模板参数。std::make_* 工厂函数则保持向后兼容性可用。

How it works…

我们定义一个类，这样就可以了解它是如何工作的:

```
template<typename T1, typename T2, typename T3>
class Thing {
  T1 v1{};
  T2 v2{};
  T3 v3{};
  public:
  explicit Thing(T1 p1, T2 p2, T3 p3)
  : v1{p1}, v2{p2}, v3{p3} {}
  string print() {
    return format("{}, {}, {}\n",
      typeid(v1).name(),
      typeid(v2).name(),
      typeid(v3).name()
    );
  }
};
```

这是一个具有三种类型和三个相应数据成员的模板类。有一个 print() 函数，该函数返回一个带有三个类型名称的格式化字符串。

若没有模板参数推导，则需要实例化一个这种类型的对象:

```
Things<int, double, string> thing1{1, 47.0, "three" }
```

现在可以这样做:

```
Things thing1{1, 47.0, "three" }
```

既简单又不易出错。

当在 thing1 对象上使用 print() 函数时，得到了这样的结果:

```
cout << thing1.print();
```

输出为:

```
int, double, char const *
```

STL 包含了一些这样的辅助函数，比如 make_pair() 和 make_tuple() 等。这些代码现在已经过时了，但是为了与旧代码兼容，会保留这些代码。

There's more…

考虑带有参数包的构造函数:

```
template <typename T>
class Sum {
  T v{};
public:
  template <typename... Ts>
  Sum(Ts&& ... values) : v{ (values + ...) } {}
  const T& value() const { return v; }
};
```

注意构造函数中的折叠表达式 (values + …)。这是 C++17 的特性，可将操作符应用于一个参数包的所有成员。本例中，将 v 初始化为参数包的和。

该类的构造函数接受任意数量的形参，其中每个形参可以是不同的类。可以这样调用:

```
Sum s1 { 1u, 2.0, 3, 4.0f }; // unsigned, double, int,
                // float
Sum s2 { "abc"s, "def" }; // std::sring, c-string
```

当然，这无法编译。模板实参推导不能为所有这些不同的形参找到一个公共类型，编译器将会报出一个错误消息:

```
cannot deduce template arguments for 'Sum'
```

可以用模板推导指南来解决这个问题。推导指南是一种辅助模式，用于协助编译器进行复杂的推导。下面是构造函数的指南:

```
template <typename... Ts>
Sum(Ts&& ... ts) -> Sum<std::common_type_t<Ts...>>;
```

这告诉编译器使用 std::common_type_t 的特征，试图为包中的所有参数找到一个公共类型。现在，参数推导工作了，可以看到确切的类型:

```
Sum s1 { 1u, 2.0, 3, 4.0f }; // unsigned, double, int,
                // float
Sum s2 { "abc"s, "def" }; // std::sring, c-string

auto v1 = s1.value();
auto v2 = s2.value();
cout << format("s1 is {} {}, s2 is {} {}",
    typeid(v1).name(), v1, typeid(v2).name(), v2);
```

输出为:

```
s1 is double 10, s2 is class std::string abcdef
```

2.6. if constexpr——简化编译时决策

if constexpr(condition) 语句用于根据编译时条件执行代码的地方，条件可以是 bool 类型的 constexpr 表达式。

How to do it…

试想，有一个模板函数，需要根据模板形参的类型进行不同的操作。

```
template<typename T>
auto value_of(const T v) {
  if constexpr (std::is_pointer_v<T>) {
    return *v; // dereference the pointer
  } else {
    return v; // return the value
  }
}

int main() {
  int x{47};
  int* y{&x};
  cout << format("value is {}\n", value_of(x)); // value
  cout << format("value is {}\n", value_of(y)); // pointer
  return 0;
}
```

输出为:

```
value is 47
value is 47
```

模板形参 T 的类型在编译时可用，constexpr if 语句可以让代码轻松区分指针和值。

How it works…

constexpr if 语句的工作方式与普通 if 语句类似，只不过其是在编译时求值的。运行时代码将不包含来自 constexpr if 语句的任何分支。考虑上面的分支语句:

```
if constexpr (std::is_pointer_v<T>) {
  return *v; // dereference the pointer
} else {
  return v; // return the value
}
```

is_pointer_v<T> 会测试模板参数，该参数在运行时不可用。当模板形参 <T> 可用时，constexpr 关键字会告诉编译器这个 if 语句需要在编译时求值。

这将使元编程更加容易，if constexpr 语句可在 C++17 及更高版本中使用。

第 3 章 STL 容器

本章中，将关注 STL 中的容器类，容器是一个包含其他对象或元素集合的对象。STL 提供了一套完整的容器类型，它们是 STL 的基础。

3.1. STL 容器类型的概述

STL 提供了一套全面的容器类型，包括顺序容器、关联容器和容器适配器。以下是简要的概述:

顺序容器

顺序容器提供了一个接口，其中元素按顺序排列。虽然可以按顺序使用元素，但其中一些容器使用连续存储，而其他容器则不使用。STL 包含以下顺序容器:

- array 是固定大小的序列，在连续存储器中保存特定数量的元素，分配之后就不能改变大小。这是最简单和访问速度最快的连续存储容器。
- vector 就像数组，可以缩小和扩大。其元素是连续存储的，因此改变大小可能涉及分配内存和移动数据的开销。vector 可以保留额外的空间来降低操作成本。在 vector 容器后面以外的任何位置插入和删除元素将触发元素的重新排列，以保持内存的连续存储。
- list 是双链表结构，允许在常量 (O(1)) 时间内插入和删除元素。遍历列表的时间为线性 O(n)。单链表的变体是 forward_list，只能向前迭代。forward_list 会使用更少的空间，并且比双链表更高效，但缺少一些功能。
- deque(通常发音为 deck) 是双端队列，是连续的容器，可以在两端展开或收缩。deque 允许随机访问它的元素，很像 vector，但不保证存储的连续性。

关联容器

关联容器将一个键与每个元素关联起来。元素是通过键来引用的，而不是其在容器中的位置。STL 关联容器包括以下容器:

- set 是一个关联容器，每个元素也是自己的键，元素通常按某种二叉树方式排序。set 中的元素不可变，不能修改，但是可以插入和移除。set 中的元素是唯一的，不允许重复。set 可以根据排序操作符按顺序进行迭代。
- multiset 就像一个具有非唯一键的集合，允许重复。
- unordered_set 就像一个不按顺序迭代的集合。元素不按特定顺序排序，而是根据哈希值进行组织，以便快速访问。
- unordered_multiset 类似于 unordered_set，允许重复。
- map 是键-值对的关联容器，其中每个键都映射到特定的值 (或有效负载)。键和值的类型可能不同；键是唯一的，但值不是。map 根据其排序操作符，按键的顺序进行迭代。
- multimap 类似于具有非唯一键的映射，允许重复键。
- unordered_map 就像一个没有按顺序迭代的 map。
- unordered_multimap 类似于 unordered_map，允许重复。

容器适配器

容器适配器是封装底层容器的类，容器类提供了一组特定的成员函数来访问底层容器元素。STL 提供了以下容器适配器:

- stack 提供了后进先出 (LIFO) 接口，该接口中只能从容器的一端添加和提取元素。底层容器可以是 vector、deque 或 list 中的一种。若没有指定底层容器，默认为 deque。
- queue 提供了先进先出 (FIFO) 接口，其中元素可以在容器的一端添加，并从另一端提取。底层容器可以是 deque 或 list 容器之一。若没有指定底层容器，默认为 deque。
- priority_queue 按照严格的弱顺序将最大的值元素保持在顶部，以对数时间插入和提取为代价，提供了对最大值元素的常数时间查找。底层容器可以是 vector 或 deque 中的一个。若没有指定底层容器，默认为 vector。

我们将讨论以下主题:

- 使用擦除函数从容器中删除项
- 常数时间内从未排序的向量中删除项
- 安全地访问 vector 元素
- 保持 vector 元素的顺序
- 高效地将元素插入到 map 中
- 高效地修改 map 项的键值
- 自定义键值的 unordered_map
- 使用 set 对输入进行排序和筛选
- 简单的 RPN 计算器与 deque
- 使用 map 的词频计数器
- 找出含有相应长句的 vector
- 使用 multimap 制作待办事项列表

3.2. 相关准备

本章所有的例子和源代码都可以在本书的 GitHub 库中找到，网址是https://github.com/PacktPublishing/CPP-20-STL-Cookbook/tree/main/chap03。

3.3. 使用擦除函数从容器中删除项

C++20 前，erase-remove 通常用于从 STL 容器中删除元素。这操作有点麻烦，通常使用这样的函数来完成:

```
template<typename Tc, typename Tv>
void remove_value(Tc & c, const Tv v) {
  auto remove_it = std::remove(c.begin(), c.end(), v);
  c.erase(remove_it, c.end());
}
```

std::remove() 函数在 <algorithms> 头文件中声明。remove() 搜索指定的值，并将元素从容器的末尾向前移动来删除它，所以并不会改变容器的大小。它返回一个超出移位范围末端的迭代器，然后调用容器的 erase() 函数删除剩余的元素。

有了新的擦除功能，这个两步过程可以简化为一步:

```
std::erase(c, 5); // same as remove_value() function
```

这个函数与上面的 remove_value() 函数功能相同。

还有一个版本使用了谓词函数。例如，要从数值容器中删除所有偶数值:

```
std::erase_if(c, [](auto x) { return x % 2 == 0; });
```

How to do it…

擦除函数有两种形式。第一种形式叫做 erase()，有两个参数，一个容器和一个值:

```
erase(container, value);
```

容器可以是顺序容器 (vector, list, forward_list, deque)，数组除外，数组不能改变大小。

第二种形式称为 erase_if()，接受一个容器和一个谓词函数:

```
erase_if(container, predicate);
```

这种形式适用于使用 erase() 的容器，也适用于关联容器、set、map 及其多键和无序的版本。

函数 erase() 和 erase_if() 作为非成员函数定义在相应容器的头文件中。

来看一些例子:

- 首先，定义一个简单的函数来打印顺序容器的大小和元素:

```
void printc(auto & r) {
  cout << format("size({}) ", r.size());
  for( auto & e : r ) cout << format("{} ", e);
  cout << "\n";
}
```

printc() 函数使用 C++20 的 format() 函数为 cout 格式化字符串。

- 下面是一个包含 10 个整数元素的 vector，用 printc() 函数打印出来:

```
vector v{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
printc(v);
```

输出为:

```
size: 10: 0 1 2 3 4 5 6 7 8 9
```

可以使用 erase() 删除所有值为 5 的元素:

```
erase(v, 5);
printc(v);
```

输出为:

```
size: 9: 0 1 2 3 4 6 7 8 9
```

std::erase() 函数的 vector 版本定义在 <vector> 头文件中。在 erase() 调用之后，删除值为 5 的元素，vector 中有 9 个元素。

- 这也适用于列表容器:

```
list l{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
printc(l);
erase(l, 5);
printc(l);
```

输出为:

```
size: 10: 0 1 2 3 4 5 6 7 8 9
size: 9: 0 1 2 3 4 6 7 8 9
```

std::erase() 函数的列表版本定义在 <list> 头文件中。erase() 之后，删除值为 5 的元素，list 有 9 个元素。

- erase_if() 可以使用一个简单的谓词函数，删除所有偶数元素:

```
vector v{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
printc(v);
erase_if(v, [](auto x) { return x % 2 == 0; });
printc(v);
```

输出为:

```
size: 10: 0 1 2 3 4 5 6 7 8 9
size: 5: 1 3 5 7 9
```

- erase_if() 函数也适用于关联容器，比如 map:

```
void print_assoc(auto& r) {
  cout << format("size: {}: ", r.size());
  for( auto& [k, v] : r ) cout << format("{}:{} ",
    k, v);
  cout << "\n";
}

int main() {
  map<int, string> m{ {1, "uno"}, {2, "dos"},
    {3, "tres"}, {4, "quatro"}, {5, "cinco"} };
  print_assoc(m);
  erase_if(m,
```

```
    [](auto& p) { auto& [k, v] = p;
    return k % 2 == 0; }
  );
  print_assoc(m);
}
```

输出为:

```
size: 5: 1:uno 2:dos 3:tres 4:quatro 5:cinco
size: 3: 1:uno 3:tres 5:cinco
```

因为 map 的每个元素都是成对返回的，所以需要不同的函数来打印。print_assoc() 函数的作用: 在 for 循环中使用结构化绑定来解包对元素。还可以在 erase_if() 的谓词函数中使用结构化绑定，来隔离用于过滤偶数元素的键值。

How it works…

erase() 和 erase_if() 函数只是一次性执行 erase-remove 的包装器:

```
template<typename Tc, typename Tv>
void remove_value(Tc & c, const Tv v) {
  auto remove_it = std::remove(c.begin(), c.end(), v);
  c.erase(remove_it, c.end());
}
```

若考虑一个简单的 int vector，称为 vec，具有以下值:

```
vector vec{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

可以将 vec 可视化为一行的 int 值表:

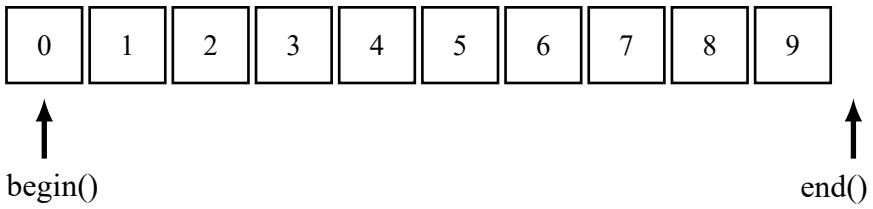


图 3.1 begin() 和 end() 迭代器

begin() 迭代器指向第一个元素，end() 迭代器指向最后一个元素。此配置是所有 STL 顺序容器的标准。

使用 remove(c.begin()，c.end()，5) 时，算法从 begin() 迭代器开始搜索匹配的元素。对于找到的每个匹配元素，将下一个元素移到它的位置。然后，继续搜索和移动，直到到达 end() 迭代器。结果是一个容器，其中所有剩余的元素都在最开始的部分，没有删除的元素，并按照它们原来的顺序。end() 迭代器不变，其余元素未定义。可以这样可视化操作:

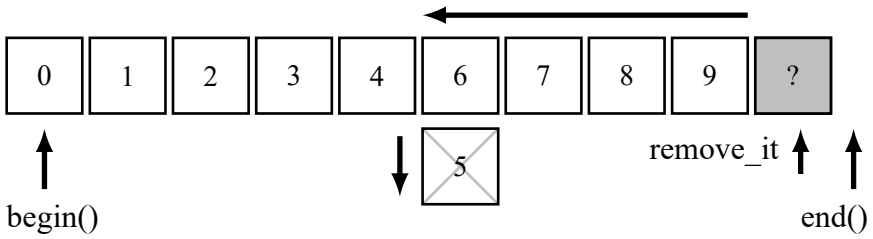


图 3.2 移除一个元素

remove() 函数的作用是: 返回一个迭代器 (remove_it)，指向移位的元素之后的第一个元素。end() 迭代器保持在 remove() 操作之前的状态。为了进一步说明，若要使用 remove_if() 删除所有偶数元素，结果如下所示:

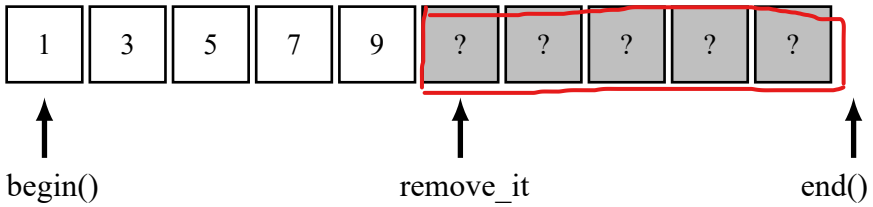


图 3.3 移除偶数元素后

本例中，剩下的就是 5 个奇数元素，后面跟着 5 个未定义值的元素。

然后，调用容器的 erase() 函数来擦除剩余的元素:

```
c.erase(remove_it, c.end());
```

容器的 erase() 函数使用 remove_it 和 end() 迭代器调用，可以删除所有未定义的元素。

erase() 和 erase_if() 函数同时调用 remove() 函数和容器的 erase() 函数，以便一步执行 erase-remove。

3.4. 常数时间内从未排序的向量中删除项

使用擦除函数 (或 erase-remove) 从 vector 中间删除项需要 O(n)(线性) 时间。因为元素必须从向量的末尾移动，以填补删除项之间的空白。若 vector 中项目的顺序不重要，就可以优化这个过程，使其花费 O(1)(常数) 时间。

How to do it…

这个方法利用了这样一个事实，即从 vector 的末尾删除一个元素是快速和简单的。

- 先从定义函数来打印 vector 开始:

```
void printc(auto & r) {
  cout << format("size({}) ", r.size());
  for( auto & e : r ) cout << format("{} ", e);
```

```
  cout << '\n';
}
```

- main() 函数中，定义了一个 int 类型的 vector，并使用 printc() 将其打印出来:

```
int main() {
  vector v{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  printc(v);
}
```

输出为:

```
size(10) 0 1 2 3 4 5 6 7 8 9
```

- 现在编写一个函数，从 vector 中删除一个元素:

```
template<typename T>
void quick_delete(T& v, size_t idx) {
  if (idx < v.size()) {
    v[idx] = move(v.back());
    v.pop_back();
  }
}
```

quick_delete() 函数有两个参数，一个 vector v 和一个索引 idx。首先，检查索引是否在边界之内。然后，从 <algorithms> 头文件中调用 move() 函数将 vector 的最后一个元素移动到索引的位置。最后，调用 v.pop_back() 函数从后面缩短 vector。

- 还有一个版本的 quick_delete()，用于迭代器而非索引。

```
template<typename T>
void quick_delete(T& v, typename T::iterator it) {
  if (it < v.end()) {
    *it = move(v.back());
    v.pop_back();
  }
}
```

- 现在可以在 main() 函数中使用:

```
int main() {
  vector v{ 12, 196, 47, 38, 19 };
  printc(v);
  auto it = std::ranges::find(v, 47);
  quick_delete(v, it);
  printc(v);
  quick_delete(v, 1);
  printc(v);
}
```

输出如下所示:

```
size(5) 12 196 47 38 19
size(4) 12 196 19 38
size(3) 12 38 19
```

对 quick_delete() 的第一次调用使用 std::ranges::find() 算法中的迭代器。这将从 vector 中删除值 47，vector(19) 后面的值取代了它的位置。第二次调用 quick_delete() 使用索引 (1) 从 vector(196) 中删除第二个元素。同样，vector 后面的值会取代其位置。

How it works…

quick_delete() 函数使用一个简单的技巧快速有效地从 vector 中删除元素。vector 后面的元素会移动 (而不是复制) 到要删除的元素的位置。删除的元素在进程中丢弃。

然后，pop_back() 函数将 vector 从末尾开始缩短一个元素，即删除 vector 后面的元素开销很小。pop_back() 函数的操作复杂度不变，因为它只更改 end() 迭代器。

这个图显示了 quick_delete() 操作前后 vector 的状态:

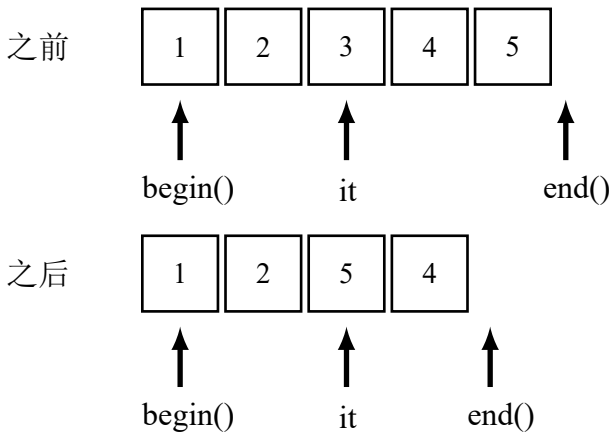


图 3.4 quick_delete() 之前和之后

quick_remove() 操作只是将元素从 vector 的后面移动到迭代器 (it) 的位置，然后将 vector 缩短一个元素。使用 std::move() 而不是赋值来移动元素很重要，移动操作比复制赋值快得多，特别是对于大型对象。

若不需要元素有序，那么这种方式就非常有效了。可以在常数 (O(1)) 时间内完成，并不涉及到其他元素。

3.5. 安全地访问 vector 元素

vector 是 STL 中使用最广泛的容器之一，并且可以使用 [] 操作符来访问 vector 中的元素:

```
vector v{ 19, 71, 47, 192, 4004 };
auto & i = v[2];
```

vector 类还提供了一个成员函数，与 [] 操作符效果相同:

```
auto & i = v.at(2);
```

结果一样，但有一个重要的区别。at() 函数执行边界检查，而 [] 操作符不检查，[] 操作符为了保持与原始 C 数组的兼容性。

How to do it…

有两种方法访问 vector 中带有索引的元素。at() 成员函数执行边界检查，而 [] 操作符不检查。

- 下面是一个简单的 main() 函数，其初始化一个 vector 并访问一个元素:

```
int main() {
  vector v{ 19, 71, 47, 192, 4004 };
  auto & i = v[2];
  cout << format("element is {}\n", i);
}
```

输出为:

```
element is 47
```

这里，使用 [] 操作符直接访问 vector 中的第三个元素。与 C++ 中的大多数顺序对象一样，索引从 0 开始，因此第三个元素是 2。

- 这个 vector 有 5 个元素，从 0 到 4。若访问 5 号元素，将超出 vector 的边界:

```
vector v{ 19, 71, 47, 192, 4004 };
auto & i = v[5];
cout << format("element is {}\n", i);
```

输出为:

```
element is 0
```

这个结果极具欺骗性。这是一个常见的错误，因为人类倾向于从 1 开始计数，而不是 0。但是不能保证 vector 范围外的元素的值。

- 更糟糕的是，[] 操作符会无声地允许对超出 vector 结尾的位置进行写入:

```
vector v{ 19, 71, 47, 192, 4004 };
v[5] = 2001;
auto & i = v[5];
cout << format("element is {}\n", i);
```

输出为:

```
element is 2001
```

现在已经写入内存，编译器会默许这样的行为，不会出现任何错误信息或崩溃。但是不要被骗了——这是极其危险的代码，在未来的某个时候会出现问题。越界内存访问是安全漏洞的主要原因之一。

- 解决方案是尽可能使用 at() 成员函数，而不是 [] 操作符:

```
vector v{ 19, 71, 47, 192, 4004 };
auto & i = v.at(5);
cout << format("element is {}\n", i);
```

现在出现了一个运行时异常:

```
terminate called after throwing an instance of 'std::out_
of_range'
  what(): vector::_M_range_check: __n (which is 5) >=
this->size() (which is 5)
Aborted
```

代码编译时没有错误，但是 at() 函数检查容器的边界，并尝试访问这些边界之外的内存时，就抛出运行时异常。这是来自 GCC 编译器编译的代码的异常消息，在不同的环境中，信息也会不同。

How it works…

[] 操作符和 at() 成员函数做同样的工作，可根据容器元素的索引位置直接访问容器元素。[] 操作符不进行边界检查，因此在一些频繁迭代的应用程序中，会快一点。

不过，at() 函数应该是首选。虽然边界检查可能需要几个 CPU 周期，但这是一种成本很低的保险。对于大多数应用来说，这样做物有所值。

vector 类通常用作直接访问容器，而 array 和 deque 容器也同时支持 [] 操作符和 at() 成员函数。

There's more…

某些应用程序中，可能不希望应用程序在遇到出界条件时崩溃。这种情况下，可以捕获异常:

```
int main() {
  vector v{ 19, 71, 47, 192, 4004 };
  try {
    v.at(5) = 2001;
  } catch (const std::out_of_range & e) {
    std::cout <<
    format("Ouch!\n{}\n", e.what());
  }
  cout << format("end element is {}\n", v.back());
}
```

输出为:

```
Ouch!
vector::_M_range_check: __n (which is 5) >= this->size() (which is 5)
end element is 4004
```

try 块捕获 catch 子句中指定的异常，当前的异常是 std::out_of_range。e.what() 函数返回一个 C 字串，其中包含来自 STL 库的错误消息。当然，每个库都会有不同的消息，这也适用于 array 和 deque 容器。

3.6. 保持 vector 元素的顺序

vector 是一个顺序容器，会按照插入元素的顺序保存元素。并不对元素进行排序，也不以任何方式改变其顺序。其他容器，如 set 和 map，会将元素排序，但这些容器不可随机访问。不过，只需要稍加处理，vector 也可以保持有序。

How to do it…

这个示例是创建一个简单的函数 insert_sorted()，将一个元素插入到 vector 中的正确位置，以保持 vector 元素的顺序。

- 先从 string vector 类型的别名开始:

```
using Vstr = std::vector<std::string>;
```

这里我喜欢使用类型别名，因为 vector 的确切细节并不重要。

- 然后，可以定义几个辅助函数:

```
// print a vector
void printv(const auto& v) {
  for(const auto& e : v) {
    cout << format("{} ", e);
  }
  cout << "\n";
}

// is it sorted?
void psorted(const Vstr& v) {
  if(std::ranges::is_sorted(v)) cout<< "sorted: ";
  else cout << "unsorted: ";
  printv(v);
}
```

printv() 函数非常简单，其将 vector 的元素打印在一行上。

psorted() 函数使用 is_sorted() 算法的范围版本，确定 vector 是否已排序。然后，使用 printv() 来输出 vector。

- 可以在 main() 函数中初始化一个 Vstr vector:

```
int main() {
  Vstr v{
    "Miles",
    "Hendrix",
    "Beatles",
    "Zappa",
    "Shostakovich"
  };
  psorted(v);
}
```

输出为:

```
unsorted: Miles Hendrix Beatles Zappa Shostakovich
```

这里，我们有一个 Vstr vector 与一些有趣的音乐家的名字，但并没有进行排序。

- 使用 sort() 算法的范围版本对 vector 排序。

```
std::ranges::sort(v);
psorted(v);
```

输出为:

```
sorted: Beatles Hendrix Miles Shostakovich Zappa
```

- 我们希望能够将相应项插入到 vector 中，以便进行排序。insert_sorted() 函数做了以下工作:

```
void insert_sorted(Vstr& v, const string& s) {
  const auto pos{ std::ranges::lower_bound(v, s) };
  v.insert(pos, s);
}
```

insert_sorted() 函数使用 lower_bound() 算法的范围版本来获取 insert() 函数的迭代器，该迭代器保持 vector 的排序。

- 可以使用 insert_sorted() 函数，将更多的音乐家名字插入到 vector 中:

```
insert_sorted(v, "Ella");
insert_sorted(v, "Stones");
```

输出为:

```
sorted: Beatles Ella Hendrix Miles Shostakovich Stones Zappa
```

How it works…

insert_sorted() 函数用于将元素插入到已排序的 vector 中，同时保持其顺序:

```
void insert_sorted(Vstr& v, const string& s) {
  const auto pos{ std::ranges::lower_bound(v, s) };
  v.insert(pos, s);
}
```

lower_bound() 算法查找不小于实参的第一个元素。然后，使用 lower_bound() 返回的迭代器在正确的位置插入一个元素。

本例中，使用了 lower_bound() 的范围版本，其他版本也是可以使用的。

There's more…

insert_sorted() 函数可以通过模板变得更加通用。此版本将用于其他容器类型，如 set、deque 和 list。

```
template<typename C, typename E>
void insert_sorted(C& c, const E& e) {
  const auto pos{ std::ranges::lower_bound(c, e) };
  c.insert(pos, e);
}
```

std::sort() 算法 (及其衍生算法) 需要支持随机访问的容器，并不是所有的 STL 容器都满足这个要求。值得注意的是，std::list 不支持随机访问。

3.7. 高效地将元素插入到 map 中

map 类是保存键-值对的关联容器，其中键在容器内必须唯一。

填充 map 容器的方法有很多种，可以这样定义的 map:

```
map<string, string> m;
```

使用 [] 操作符添加一个元素:

```
m["Miles"] = "Trumpet"
```

使用 insert() 成员函数:

```
m.insert(pair<string,string>("Hendrix", "Guitar"));
```

或者，使用 emplace() 成员函数:

```
m.emplace("Krupa", "Drums");
```

我倾向于使用 emplace() 函数，可以完全转发来为容器放置 (在适当的位置创建) 新元素。参数直接转发给元素构造函数，快速、高效且易于阅读。

尽管 emplace() 肯定是对其他选项的改进，但其问题在于，即使在不需要对象时，也会构造对象。这包括调用构造函数、分配内存、移动数据，然后丢弃临时对象。

为了解决这个问题，C++17 提供了新的 try_emplace() 函数，该函数只在需要时构造值对象，这对于大型对象尤为重要。

Note

map 的每个元素都是一个键值对，其中元素命名为 first 和 second，但在 map 中是键和值。我倾向于将值对象视为有效负载。要搜索一个现有的键，try_emplace() 函数必须构造键对象，但不需要构造有效负载对象，除非需要插入到 map 中。

How to do it…

新的 try_emplace() 函数避免了构造有效负载对象的开销，这在键值碰撞的情况下效率很高，特是在大有效载荷的情况下。

- 首先，创建一个有效负载类。出于演示目的，该类有一个简单的 std::string 有效负载，并在构造时显示一条消息:

```
struct BigThing {
  string v_;
  BigThing(const char * v) : v_(v) {
    cout << format("BigThing constructed {}\n", v_);
  }
};
using Mymap = map<string, BigThing>;
```

这个 BigThing 类只有一个成员函数——构造函数，在构造对象时显示一条消息。我们将使用它来跟踪 BigThing 对象的构造频率。实际中，这个类会更大，使用更多的资源。

每个 map 元素将由 pair 对象组成，一个 std::string 用于键，一个 BigThing 对象用于负载。这里，Mymap 只是一个别名，只是为了让我们能够更专注于功能。

- 再创建一个 printm() 函数来打印 map 的内容:

```
void printm(Mymap& m) {
  for(auto& [k, v] : m) {
    cout << format("[{}:{}] ", k, v.v_);
  }
  cout << "\n";
}
```

使用 C++20 的 format() 函数打印 map，就可以在插入元素时观察它们了。

- main() 函数中，创建了 map 对象并插入了一些元素:

```
int main() {
  Mymap m;
  m.emplace("Miles", "Trumpet");
  m.emplace("Hendrix", "Guitar");
  m.emplace("Krupa", "Drums");
  m.emplace("Zappa", "Guitar");
  m.emplace("Liszt", "Piano");
  printm(m);
}
```

输出为:

```
BigThing constructed Trumpet
BigThing constructed Guitar
BigThing constructed Drums
BigThing constructed Guitar
BigThing constructed Piano
[Hendrix:Guitar] [Krupa:Drums] [Liszt:Piano]
[Miles:Trumpet] [Zappa:Guitar]
```

输出显示了每个有效负载对象的构造，然后是 printm() 函数的输出。

- 使用 emplace() 函数将元素添加到 map 中，每个有效负载元素只构造一次。这里，也可以使用 try_emplace() 函数，结果相同:

```
Mymap m;
m.try_emplace("Miles", "Trumpet");
m.try_emplace("Hendrix", "Guitar");
m.try_emplace("Krupa", "Drums");
m.try_emplace("Zappa", "Guitar");
m.try_emplace("Liszt", "Piano");
printm(m);
```

输出为:

```
BigThing constructed Trumpet
BigThing constructed Guitar
BigThing constructed Drums
BigThing constructed Guitar
BigThing constructed Piano
[Hendrix:Guitar] [Krupa:Drums] [Liszt:Piano]
[Miles:Trumpet] [Zappa:Guitar]
```

- emplace() 和 try_emplace() 之间的区别显示在尝试插入具有重复键的新元素时:

```
cout << "emplace(Hendrix)\n";
m.emplace("Hendrix", "Singer");
cout << "try_emplace(Zappa)\n";
m.try_emplace("Zappa", "Composer");
printm(m);
```

输出为:

```
emplace(Hendrix)
BigThing constructed Singer
try_emplace(Zappa)
[Hendrix:Guitar] [Krupa:Drums] [Liszt:Piano]
[Miles:Trumpet] [Zappa:Guitar]
```

emplace() 函数尝试添加一个具有重复键的元素（“Hendrix”）。但失败了，但仍然构造了有效负载对象（“Singer”）。try_emplace() 函数还尝试添加一个具有重复键的元素 (”Zappa”)。也失败了，没有构造有效负载对象。

这个例子演示了 emplace() 和 try_emplace() 之间的区别。

How it works…

try_emplace() 函数签名与 emplace() 函数签名相似，因此对代码的修改应该很容易。下面是 try_emplace() 函数签名:

```
pair<iterator, bool> try_emplace( const Key& k,
Args&&... args );
```

乍一看，这与 emplace() 签名不同:

```
pair<iterator,bool> emplace( Args&&... args );
```

区别在于 try_emplace() 为键参数使用了一个单独的形参，这允许在构造时隔离。从函数上讲，若正在使用模板参数推导，则可以使用 try_emplace() 替换:

```
m.emplace("Miles", "Trumpet");
m.try_emplace("Miles", "Trumpet");
```

try_emplace() 的返回值与 emplace() 的返回值相同，是一个表示迭代器和 bool 的 pair:

```
const char * key{"Zappa"};
const char * payload{"Composer"};
if(auto [it, success] = m.try_emplace(key, payload);
!success) {
  cout << "update\n";
  it->second = payload;
}
printm(m);
```

输出为:

```
update
BigThing constructed Composer
[Hendrix:Guitar] [Krupa:Drums] [Liszt:Piano] [Miles:Trumpet]
[Zappa:Composer]
```

这里使用了结构化绑定 (auto [it, success] =) 和 if 初始化语句来测试返回值，并有条件地更新有效负载。注意，这仍然构造了有效负载对象。

try_emplace() 函数也适用于 unordered_map:

```
using Mymap = unordered_map<string, BigThing>;
```

try_emplace() 优点，只在准备将有效负载对象存储到 map 中时构造有效负载对象。实际中，可在运行时节省大量资源，所以应该首选 try_emplace()，而非 emplace()。

3.8. 高效地修改 map 项的键值

map 是存储键值对的关联容器，容器是按键排序的。键必须是唯一的，并且是 const 限定的，所以不能更改。

例如，若填充一个 map 并试图更改键，在编译时会得到一个错误:

```
map<int, string> mymap {
  {1, "foo"}, {2, "bar"}, {3, "baz"}
};
auto it = mymap.begin();
it->first = 47;
```

输出为:

```
error: assignment of read-only member ...
5 | it->first = 47;
  |  ~~~~~~~~~~^~~~
```

若需要重新排序 map 容器，可以通过使用 extract() 方法交换键来实现。C++17 中，extract() 是 map 类，及其派生类中的成员函数。

它允许从序列中提取 map 元素，而不涉及有效负载。当提取出来时，键就不再是 const 限定的，并且可以修改。

How to do it…

本例中，我们将定义一个表示比赛中的选手的 map。在比赛过程中的某个时刻，顺序发生了变化，需要修改 map 的键值。

- 先从定义 map 类型的别名开始:

```
using Racermap = map<unsigned int, string>;
```

- 编写一个函数来打印 map:

```
void printm(const Racermap &m)
{
  cout << "Rank:\n";
  for (const auto& [rank, racer] : m) {
    cout << format("{}:{}\n", rank, racer);
  }
}
```

可以随时将 map 传递给这个函数，以打印出参赛者的当前排名。

- main() 函数中，定义了一个具有赛车初始状态的 map:

```
int main() {
  Racermap racers {
    {1, "Mario"}, {2, "Luigi"}, {3, "Bowser"},
    {4, "Peach"}, {5, "Donkey Kong Jr"}
  };
  printm(racers);
  node_swap(racers, 3, 5);
  printm(racers);
}
```

键是 int 型，表示赛车的级别，值是赛车手名字的字符串。

然后，使用 printm() 来打印当前排名。node_swap() 将交换两个赛车手的键，然后再次输出。

- 在某个时刻，一名选手落后了，而另一名选手则趁机提升了排名。node_swap() 函数将交换两个赛车手的排名:

```
template<typename M, typename K>
bool node_swap(M & m, K k1, K k2) {
  auto node1{ m.extract(k1) };
  auto node2{ m.extract(k2) };
  if(node1.empty() || node2.empty()) {
    return false;
  }
  swap(node1.key(), node2.key());
  m.insert(move(node1));
  m.insert(move(node2));
  return true;
}
```

这个函数使用 map.extract() 方法从映射中提取指定的元素。这些提取出来的元素称为节点。节点是一个从 C++17 出现的新概念，可以在不涉及元素本身的情况下从 map 类型结构中提取元素。解除节点链接，返回节点句柄。提取后，节点句柄通过节点的 key() 函数提供对键的可写访问。然后，可以交换键并插入到 map 中，无需复制或操作有效负载。

- 当我们运行这段代码时，得到了节点交换前后 map 的输出:

```
Rank:
1:Mario
2:Luigi
3:Bowser
4:Peach
5:Donkey Kong Jr
Rank:
1:Mario
2:Luigi
3:Donkey Kong Jr
4:Peach
5:Bowser
```

这都是通过 extract() 方法和新的 node_handle 类实现的。让我们仔细了解一下它是如何工作的。

How it works…

该技术使用 extract() 函数，该函数返回一个 node_handle 对象，node_handle 是节点的句柄，由关联元素及其相关结构组成。extract 函数在将节点保留在原处的同时将其解除关联，并返回一个 node_handle 对象。这样做的效果是从关联容器中删除节点，而不涉及数据本身。node_handle 允许访问已解除关联的节点。

node_handle 有一个成员函数 key()，返回一个对节点键的可写引用。这就可以在键与容器解关联时，对键值进行修改。

There's more…

使用 extract() 和 node_handle 时，有几个重点:

- 若没有找到键，extract() 函数返回一个空节点句柄。可以用 empty() 函数测试节点句柄是否为空:

```
auto node{ mapthing.extract(key) };
if(node.empty()) {
  // node handle is empty
}
```

- extract() 有两个重载:

```
node_type extract(const key_type& x);
node_type extract(const_iterator position);
```

我们使用了第一种形式，通过传递键，使用迭代器。

- 不能使用字面量引用，因此像 extract(1) 这样的调用会因段错误而使程序崩溃。

- 键插入映射时必须保持唯一。

  例如，若将一个键更改为映射中已经存在的值:

```
auto node_x{ racers.extract(racers.begin()) };
node_x.key() = 5; // 5 is Donkey Kong Jr
auto status = racers.insert(move(node_x));
if(!status.inserted) {
  cout << format("insert failed, dup key: {}",
    status.position->second);
  exit(1);
}
```

  插入失败，会得到错误信息:

```
insert failed, dup key: Donkey Kong Jr
```

  本例中，将 begin() 迭代器传递给 extract()。然后，为键值分配了一个已经在使用的值 (5, Donkey Kong Jr)。插入失败和结果 status.inserted 为 false。status.position 是指向已找到键的迭代器。在 if() 中，我使用 format() 来打印找到的键值。

3.9. 自定义键值的 unordered_map

对于有序 map，键的类型必须是可排序的，必须至少支持小于比较操作符。假设希望使用具有不可排序的自定义类型的关联容器。例如，一个向量 (0,1) 既不大于也不小于 (1,0)，只是指向不同的方向。这种情况下，可以使用 unordered_map 类型。

How to do it…

对于这个示例，我们将创建一个 unordered_map 对象，该对象使用 x/y 坐标作为键。为此，并且需要一些功能支持。

- 首先，将为坐标定义为一个结构体:

```
struct Coord {
  int x{};
  int y{};
};
```

  有两个元素，x 和 y，作为坐标。

- map 将使用 Coord 结构作为键，并使用 int 作为值:

```
using Coordmap = unordered_map<Coord, int>;
```

  为了方便使用 map，我们使用了 using 别名。

- 要使用 Coord 结构体作为键，需要几个重载。这些是使用 unordered_map 所必需的。首先，定义一个相等比较运算符:

```
bool operator==(const Coord& lhs, const Coord& rhs) {
  return lhs.x == rhs.x && lhs.y == rhs.y;
}
```

这是一个很简单的函数，用来比较 x 元素和 y 元素。

- 还需要一个特化的 std::hash 类，这就可以使用键检索 map 元素了:

```
namespace std {
  template<>
  struct hash<Coord> {
    size_t operator()(const Coord& c) const {
      return static_cast<size_t>(c.x)
      + static_cast<size_t>(c.y);
    }
  };
}
```

这为 std::unordered_map 类使用的默认哈希类提供了特化，必须在 std 命名空间中。

- 还需要一个 print 函数来打印 Coordmap 对象:

```
void print_Coordmap(const Coordmap& m) {
  for (const auto& [key, value] : m) {
    cout << format("{{ ({}, {}): {} }} ",
    key.x, key.y, value);
  }
  cout << '\n';
}
```

这使用 C++20 的 format() 函数来打印 x/y 键和值。注意使用双大括号 {{和}} 来打印单个大括号。

- 现在有了所有的辅助函数，就可以编写 main() 函数了。

```
int main() {
  Coordmap m {
    { {0, 0}, 1 },
    { {0, 1}, 2 },
    { {2, 1}, 3 }
  };
  print_Coordmap(m);
}
```

输出为:

```
{ (2, 1): 3 } { (0, 1): 2 } { (0, 0): 1 }
```

此时，已经定义了一个 Coordmap 对象，可接受键的 Coord 对象，并将它们映射到值。

- 也可以基于 Coord 键访问单个成员:

```
Coord k{ 0, 1 };
cout << format("{{ ({}, {}): {} }}\n", k.x, k.y,
m.at(k));
```

输出为:

```
{ (0, 1): 2 }
```

这里，定义了一个名为 k 的 Coord 对象，并使用 at() 函数从 unordered_map 中检索值。

How it works…

unordered_map 类依赖于哈希类从键中查找元素，通常会这样实例化对象:

```
std::unordered_map<key_type, value_type> my_map;
```

因为没有创建哈希类，所以其使用了默认的哈希类。unordered_map 类的完整模板类型定义如下:

```
template<
  class Key,
  class T,
  class Hash = std::hash<Key>,
  class KeyEqual = std::equal_to<Key>,
  class Allocator = std::allocator< std::pair<const Key,
    T> >
> class unordered_map;
```

该模板为 Hash、KeyEqual 和 Allocator 提供了默认值。在示例中，为默认的 std::hash 类提供了特化。

STL 包含 std::hash 对大多数标准类型的特化，如 string、int 等。为了与我们的类一起工作，其需要进行特化。

可以向模板形参传递一个函数:

```
std::unordered_map<coord, value_type, my_hash_type> my_map;
```

但在我看来，特化方式会更通用。

3.10. 使用 set 对输入进行排序和筛选

set 容器是关联容器，其中每个元素都是一个单独的值作为键。set 中的元素按序排部，不允许重复。

set 比一般的容器 (如 vector 和 map) 有更少和更具体的用途。set 的一个常见用途是从一组值中筛选重复项。

How to do it…

这个示例中，我们将从标准输入中读取单词，并过滤掉重复的单词。

- 从定义 istream 迭代器的别名开始。我们将使用它从命令行获取输入。

```
using input_it = istream_iterator<string>;
```

- main() 函数中，将为单词定义一个 set：

```
int main() {
  set<string> words;
```

set 定义为一组字符串元素。

- 定义了用于 inserter() 函数的迭代器:

```
input_it it{ cin };
input_it end{};
```

结束迭代器使用其默认构造函数初始化，这称为流结束迭代器。当输入结束时，这个迭代器将与等价于 cin 迭代器。

- inserter() 函数用于将元素插入到 set 容器中:

```
copy(it, end, inserter(words, words.end()));
```

使用 std::copy() 可以从输入流中复制单词。

- 现在可以打印 set 来查看结果:

```
for(const string & w : words) {
  cout << format("{} ", w);
}
cout << '\n';
```

- 通过将一堆单词作为输入来运行程序:

```
$ echo "a a a b c this that this foo foo foo" | ./
set-words
a b c foo that this
```

该集合消除了重复项，并保留了插入的单词的排序列表。

How it works…

set 容器是这示例的核心，其只保存唯一的元素。当插入重复元素时，插入将失败。所以，最终将得到了一个由每个唯一元素组成的有序列表。

但这并不是这个示例唯一有趣的地方。

istream_iterator 是一个从流中读取对象的输入迭代器，可以像这样实例化输入迭代器:

```
istream_iterator<string> it{ cin };
```

现在有了一个来自 cin 流的 string 类型的输入迭代器。每次解引用此迭代器时，都会从输入流中返回一个单词。

我们还实例化了另一个 istream_iterator:

```
istream_iterator<string> end{};
```

这将调用默认构造函数，提供了一个特殊的流结束迭代器。当输入迭代器到达流的末尾时，等于流的结束迭代器。这对于结束循环很方便，比如 copy() 算法创建的循环。

copy() 算法接受三个迭代器，要复制的范围的开始和结束，以及一个目标迭代器:

```
copy(it, end, inserter(words, words.end()));
```

inserter() 函数的作用是: 接受一个容器和插入点的迭代器，并返回容器及其元素的适当类型的 insert_iterator。

copy() 和 inserter() 的组合使得将元素从流复制到 set 容器变得更容易。

3.11. 简单的 RPN 计算器与 deque

RPN(逆波兰表达式) 计算器是一种基于堆栈的计算器，使用后缀符号，其中操作符紧跟在操作数之后。通常用于打印计算器，特别是 HP 12C，有史以来最受欢迎的电子计算器。

熟悉了其操作模式后，许多人更喜欢 RPN 计算器 (自从惠普 12C 和 16C 在上世纪 80 年代初首次推出以来，我一直在使用)。例如，使用传统的代数符号，要将 1 和 2 相加，可以输入 1 + 2。使用 RPN，可以输入 1 2 +。操作符跟在操作数后面。

使用代数计算器，需要按下等号键来表示需要一个结果。对于 RPN 计算器，这是不必要的，因为操作人员立即进行处理，具有双重目的。另一方面，RPN 计算器通常需要按回车键将操作数推入堆栈。

我们可以使用基于堆栈的数据结构轻松实现 RPN 计算器。例如，实现一个具有四个位置堆栈的 RPN 计算器:

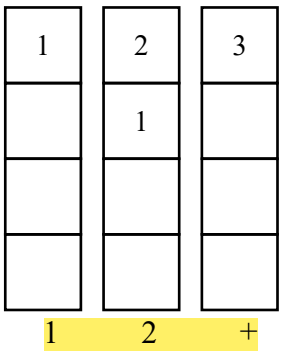


图 3.5 RPN 的加法操作

每个操作数在输入时压入堆栈。当输入操作符时，操作数弹出，操作，结果压回堆栈。然后，该结果可用于下一个操作。例如，考虑 (3+2)×3 的情况:

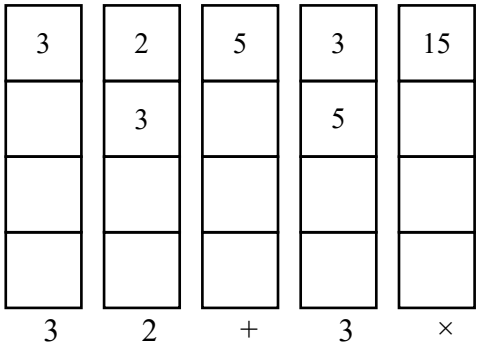


图 3.6 RPN 的堆栈操作

RPN 的优点是可以将操作数留在堆栈上以供将来计算，从而减少了对单独内存寄存器的需求。考虑 (9×6)+(2×3) 的情况:

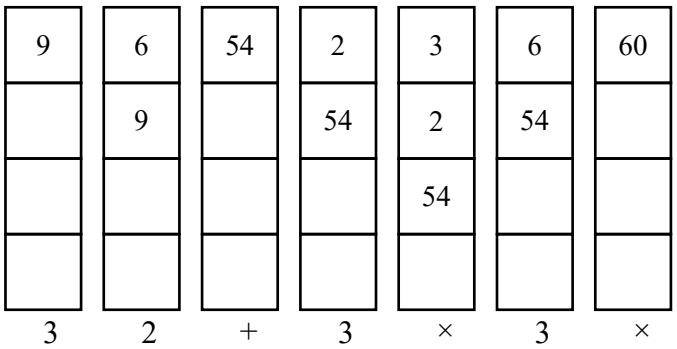


图 3.7 RPN 的多栈操作

注意，首先执行括号内的操作，然后对中间结果执行最后的操作。这可能一开始看起来比较复杂，习惯后就好了。

现在，使用 deque 容器构建一个简单的 RPN 计算器。

How to do it…

对于这个实现，我们将为堆栈使用 deque 容器。为什么不使用 stack 容器呢?stack 类是一个容器适配器，使用另一个容器 (通常是 deque) 进行存储。就我们的目标而言，stack 与 deque 相比并没有什么区别。deque 允许我们遍历和显示 RPN 堆栈，就像纸带计算器一样。

- 我们将把 RPN 计算器封装在一个类中，封装提供了安全性、可重用性、可扩展性和干净的接口。我们将类命名为 RPN:

```
class RPN {
  deque<double> deq_{};
  constexpr static double zero_{0.0};
  constexpr static double inf_
    { std::numeric_limits<double>::infinity() };
... // public and private members go here
};
```

deque 数据存储名为 deq_，位于类的 private 区域。这是我们存储 RPN 堆栈的地方。

zero_常量在整个类中都有使用，既作为返回值，也作为比较操作数。常量 inf_用于除零错误。这些常量会声明为 constexpr static，因此不会在每个实例中占用空间。

命名私有数据成员时，我喜欢在后面加下划线，以提醒我它们是私有的。

- 我们不需要显式构造函数或析构函数，因为 deque 类管理自己的资源。所以，公共接口只包含三个功能:

```
public:
  // process an operand/operator
  double op(const string & s) {
    if(is_numeric(s)) {
      double v{stod(s, nullptr)};
      deq_.push_front(v);
      return v;
    }
    else return optor(s);
  }
  // empty the stack
  void clear() {
    deq_.clear();
  }
  // print the stack
  string get_stack_string() const {
    string s{};
    for(auto v : deq_) {
      s += format("{} ", v);
    }
    return s;
  }
```

double op() 函数是 RPN 类的主要入口点，接受一个字符串，包含数字或操作符。若是数字，则转换为双精度数并压入堆栈。若是一个操作符，则调用 optor() 来执行该操作。这是这个类的主要逻辑。

void clear() 函数只是在 deque 上调用 clear() 来清空堆栈。

最后，string get_stack_string() 函数以字符串形式返回堆栈的内容。

- private 部分中，有为公共接口工作进行支持的工具函数。pop_get2() 函数从堆栈中弹出两个操作数，并将它们作为一对返回，这里使用 this 作为操作符的操作数:

```
pair<double, double> pop_get2() {
  if(deq_.size() < 2) return {zero_, zero_};
  double v1{deq_.front()};
  deq_.pop_front();
  double v2{deq_.front()};
  deq_.pop_front();
  return {v2, v1};
}
```

- is_numeric() 函数的作用是: 检查字符串是否完全是数字，也接受小数点字符。

```
bool is_numeric(const string& s) {
  for(const char c : s) {
    if(c != '.' && !std::isdigit(c)) return
    false;
  }
  return true;
}
```

- optor() 函数执行操作符，使用 map 容器将操作符映射到相应的 lambda 函数。

```
double optor(const string& op) {
  map<string, double (*)(double, double)> opmap {
    {"+", [](double l, double r){ return l + r; }},
    {"-", [](double l, double r){ return l - r; }},
    {"*", [](double l, double r){ return l * r; }},
    {"/", [](double l, double r){ return l / r; }},
    {"^", [](double l, double r)
      { return pow(l, r); }},
    {"%", [](double l, double r)
      { return fmod(l, r); }}
  };
  if(opmap.find(op) == m.end()) return zero_;
  auto [l, r] = pop_get2();
  // don't divide by zero
  if(op == "/" && r == zero_) deq_.push_front(inf_);
  else deq_.push_front(opmap.at(op)(l, r));
  return deq_.front();
}
```

带有 lambda 函数的 map 容器可以快速简便地创建跳转表。

使用 map 中的 find() 函数来测试是否有一个有效的操作符。

对除零进行测试之后，取消对 map 的引用，并调用操作符。

操作的结果会压入堆栈并返回。

- 这些都是 RPN 类的函数成员，可以在 main() 函数中使用:

```
int main() {
  RPN rpn;

  for(string o{}; cin >> o; ) {
    rpn.op(o);
    auto stack_str{rpn.get_stack_string()};
    cout << format("{}: {}\n", o, stack_str);
  }
}
```

我们将通过从命令行将字符串输送到程序中来进行测试，使用 for 循环从 cin 流中获取每个单词，并将其传递给 rpn.op()。我喜欢这里的 for 循环，因为其很容易包含 o 变量的作用域。然后，在每个命令行项后使用 get_stack_string() 函数打印堆栈。

- 可以通过输入这样的表达式来运行程序:

```
$ echo "9 6 * 2 3 * +" | ./rpn
9: 9
6: 6 9
*: 54
2: 2 54
3: 3 2 54
*: 6 54
+: 60
```

这看起来像很多编码，但实际上很简单。加上注释，RPN 类的代码不到 70 行。完整的 rpn.cpp 源代码在 GitHub 的库中。

How it works…

RPN 类首先确定每个输入块的性质。若是一个数字，则压入堆栈。若是操作符，则从堆栈顶部取出两个操作数，进行操作，并将结果推回堆栈。若是不识别的输入，就忽略它。

deque 类是一个双端队列。为了将其用作堆栈，我们选择一个端点，并从同一端点进行 push 和 pop。

若确定一个输入是数字，可以将其转换为 double，并使用 push_front() 将它推到 deque 的前面。

```
if(is_numeric(s)) {
  double v{stod(s, nullptr)};
  deq_.push_front(v);
  return v;
}
```

当需要使用堆栈中的值时，可以将它们从 deque 的前面弹出。使用 front() 获取值，然后 pop_front() 将其从堆栈中弹出。

```
pair<double, double> pop_get2() {
  if(deq_.size() < 2) return {zero_, zero_};
  double v1{deq_.front()};
  deq_.pop_front();
  double v2{deq_.front()};
  deq_.pop_front();
  return {v2, v1};
}
```

将操作符放入 map 中，可使得检查操作符是否有效和执行操作变得更加容易。

```
map<string, double (*)(double, double)> opmap {
  {"+", [](double l, double r){ return l + r; }},
  {"-", [](double l, double r){ return l - r; }},
  {"*", [](double l, double r){ return l * r; }},
```

```
  {"/", [](double l, double r){ return l / r; }},
  {"^", [](double l, double r){ return pow(l, r); }},
  {"%", [](double l, double r){ return fmod(l, r); }}
};
```

可以使用 find() 函数来测试操作符的有效性:

```
if(opmap.find(op) == opmap.end()) return zero_;
```

可以通过使用 at() 函数对 map 进行解引用来调用该操作符:

```
opmap.at(op)(l, r)
```

这里调用运算符 lambda，并在一条语句中将结果推入 deque:

```
deq_.push_front(opmap.at(op)(l, r));
```

There's more…

这个示例中，我们使用 cin 流向 RPN 计算器提供操作。使用 STL 容器同样可以做到相同的效果。

```
int main() {
  RPN rpn;
  vector<string> opv{ "9", "6", "*", "2", "3", "*", "+"
  };
  for(auto o : opv) {
    rpn.op(o);
    auto stack_str{rpn.get_stack_string()};
    cout << format("{}: {}\n", o, stack_str);
  }
}
```

输出为:

```
9: 9
6: 6 9
*: 54
2: 2 54
3: 3 2 54
*: 6 54
+: 60
```

通过将 RPN 计算器放在接口清晰的类中，我们创建了一个可以在许多不同上下文中使用的灵活工具。

3.12. 使用 map 的词频计数器

map 容器是一个关联容器，由按键值对组织的元素组成。键用于查找，并且必须是唯一的。在这个示例中，我们将利用 map 容器的唯一键值来计算文本文件中每个单词的出现次数。

How to do it…

这个任务有几个部分可以分开解决:

1. 要从一个文件中获取文本，使用 cin 流。
2. 需要把单词和标点符号，以及其他非单词的内容分开，所以需要使用正则表达式库。
3. 需要计算每个单词的出现频率，为此使用 map 容器。
4. 最后，需要对结果进行排序，首先按频率排序，然后按频率内的单词的字母顺序排序，将对 vector 容器进行排序。

- 为了方便起见，先从别名开始:

```
namespace ranges = std::ranges;
namespace regex_constants = std::regex_constants;
```

对于 std:: 空间中的命名空间，我喜欢使用短别名。在范围 (ranges) 命名空间中，会复用现有算法的名称。

- 将正则表达式存储在一个常量中。我不喜欢使全局名称空间混乱，因为这会导致冲突。我倾向于使用基于我名字首字母的命名空间，比如:

```
namespace bw {
  constexpr const char * re{"(\\w+)"};
}
```

稍后使用 bw::re 很容易获取它。

- 在 main() 的顶部，定义了数据结构:

```
int main() {
  map<string, int> wordmap{};
  vector<pair<string, int>> wordvec{};
  regex word_re(bw::re);
  size_t total_words{};
```

我们的主 map 叫做 wordmap。我们有一个名为 wordvec 的 vector，我们将对其进行排序。最后，是正则表达式类，word_re。

- for 循环是大部分工作发生的地方，从 cin 流中读取文本，应用正则表达式，并将单词存储在 map 中:

```
for(string s{}; cin >> s; ) {
  auto words_begin{
    sregex_iterator(s.begin(), s.end(), word_re) };
  auto words_end{ sregex_iterator() };
  for(auto r_it{words_begin}; r_it != words_end;
  ++r_it) {
    smatch match{ *r_it };
```

```
    auto word_str{match.str()};
    ranges::transform(word_str, word_str.begin(),
      [](unsigned char c){ return tolower(c); });
    auto [map_it, result] =
      wordmap.try_emplace(word_str, 0);
    auto & [w, count] = *map_it;
    ++total_words;
    ++count;
  }
}
```

我喜欢 for 循环，因为它可以控制变量 s 的作用域。

首先为正则表达式结果定义迭代器。这使我们能够区分多个单词，即使周围只有标点符号。for(r_it…) 循环返回 cin 字符串中的单个单词。

smatch 类型是正则表达式字符串匹配类的特化，给出了正则表达式中的下一个单词。

然后，使用转换算法使单词小写——这样就可以不考虑大小写而计算单词。(例如，“The”和“the”是同一个词。)

接下来，使用 try_emplace() 将单词添加到 map 中。若已经有了，就不会替换。

最后，使用 ++count 增加 map 中单词的计数。

- 现在，map 上有了单词和它们的频率计数，目前按字母顺序排列。不过，我们希望按词频降序排列。为此，会将其放在一个 vector 中，并对这个 vector 进行排序:

```
auto unique_words = wordmap.size();
wordvec.reserve(unique_words);
ranges::move(wordmap, back_inserter(wordvec));
ranges::sort(wordvec, [](const auto& a, const
auto& b) {
  if(a.second != b.second)
  return (a.second > b.second);
  return (a.first < b.first);
});
cout << format("unique word count: {}\n",
  total_words);
cout << format("unique word count: {}\n",
  unique_words);
```

Wordvec 是一个 vector，每个元素包含有单词和频率计数。

使用 ranges::move() 算法来填充 vector，然后使用 ranges::sort() 算法对 vector 进行排序。注意，谓词 Lambda 函数首先按计数排序 (降序)，然后按单词排序 (升序)。

- 最后，打印结果:

```
  for(int limit{20}; auto& [w, count] : wordvec) {
    cout << format("{}: {}\n", count, w);
    if(--limit == 0) break;
  }
}
```

目前，设置了只打印前 20 个条目的限制。读者们可以注释掉 if(-limit == 0) break;，从而可以打印整个列表。

- 示例文件中，包含了埃德加·艾伦·坡的《乌鸦》的文本文件，我们可以用它来测试程序:

```
$ ./word-count < the-raven.txt
total word count: 1098
unique word count: 439
56: the
38: and
32: i
24: my
21: of
17: that
17: this
15: a
14: door
11: chamber
11: is
11: nevermore
10: bird
10: on
10: raven
9: me
8: at
8: from
8: in
8: lenore
```

这首诗共有 1098 个单词，其中出现了 439 个单词。

How it works…

这个示例的核心是使用一个 map 对象来计数重复的单词。

我们使用 cin 流从标准输入中读取文本。默认情况下，cin 在读入字符串对象时会跳过空格。通过将一个字符串对象放在 >> 操作符的右边 (cin >> s)，可以得到用空格分隔的文本块。对于许多目的来说，这是足够好的定义，但我们需要语言学上的单词。为此，需要使用正则表达式。

regex 类提供了正则表达式语法的选择，它默认为 ECMA 语法。在 ECMA 语法中，正则表达式“(\w+)”是“([a- za -z0-9_]+)”的简写。这将选择包含这些字符的单词。

正则表达式本身就是一种语言。要了解更多关于正则表达式的知识，我推荐 Jeffrey Friedl 的《Mastering regular expressions》这本书。

当从正则表达式引擎获取每个单词时，可以使用 map 对象的 try_emplace() 方法有条件地将单词添加到单词 map 中。若单词不在 map 中，则将其添加为 0。若单词已经在 map 中，则计数不会受到影响。我们在后面的循环中增加计数，所以其内容总是正确的。

在用文件中的所有单词填充 map 后，我们使用 ranges::move() 算法将其传输到一个 vector 中，move() 算法使这种传输快速而有效。然后，可以使用 ranges::sort() 对 vector 进行排序。用于排序的谓词 lambda 函数包括对对两边的比较，因此最终得到一个按字数 (降序) 和单词排序的结果。

3.13. 找出含有相应长句的 vector

对于作者来说，确保使用了不同的句子长度，或者确保句子不太长对相关工作会很有帮助。现在，来构建一个评估文本句子长度的工具。在使用 STL 时，选择适当的容器是关键。若需要一些有序的东西，通常最好使用关联容器，比如 map 或 multimap。这种情况下，因为需要自定义排序，所以 vector 可能更合适。

vector 容器通常是 STL 容器中最灵活的。每当另一种容器类型看起来合适，但缺少一个重要功能时，vector 通常是有效的解决方案。所以，需要自定义排序时，就很适合使用 vector 完成。

这个示例使用了 vector 的 vector。内部 vector 存储句子中的单词，外部 vector 存储内部 vector，这在保留所有相关数据的同时提供了很大的灵活性。

How to do it…

程序需要读入单词，找到句子的结尾，存储和排序句子，然后打印出结果。

- 先写一个小函数来判断什么时候到达了句尾:

```
bool is_eos(const string_view & str) {
  constexpr const char * end_punct{ ".!?" };
  for(auto c : str) {
    if(strchr(end_punct, c) != nullptr) return
    true;
  }
  return false;
}
```

is_eos() 函数使用 string_view，因为它很有效，而且不需要其他东西。然后，使用 strchr() 库函数检查单词是否包含句尾标点字符 (".!?")。这是英语中结束句子最有可能的三个字符。

- main() 函数中，我们首先定义 vector 的 vector:

```
vector<vector<string>> vv_sentences{vector<string>{}};
```

这定义了一个名为 vv_sentence 的元素 vector，类型为 vector<string>。vv_sentence 对象初始化为第一个句子的空 vector。

这将创建一个包含其他 vector 的 vector，内部 vector 将每个包含一个词的句子。

- 现在可以处理单词流了:

```
for(string s{}; cin >> s; ) {
  vv_sentences.back().emplace_back(s);
  if(is_eos(s)) {
```

```
    vv_sentences.emplace_back(vector<string>{});
  }
}
```

for 循环每次从输入流返回一个单词。vv_sentence 对象上的 back() 方法用于访问单词的当前 vector，并且使用 emplace_back() 添加当前单词。然后，使用 is_eos() 来查看这是否是句子的结尾。所以，可以添加一个空的 vector 到 vv_sentence 的下一个句子中。

- 我们总是在 vv_sentence 的每个句尾字符之后添加一个新的空 vector，所以需要在结尾得到一个空的句子 vector。在这里进行检查，并在必要时将其删除:

```
  // delete back if empty
  if(vv_sentences.back().empty())
    vv_sentences.pop_back();
```

- 现在可以根据句子的大小对 vv_sentence vector 进行排序:

```
  sort(vv_sentences, [](const auto& l,
    const auto& r) {
      return l.size() > r.size();
    });
```

这就是为什么 vector 对这个项目如此方便。使用 ranges::sort() 算法和一个简单的谓词，会使按大小降序排序非常快速和容易。

- 现在可以打印结果了:

```
  constexpr int WLIMIT{10};
  for(auto& v : vv_sentences) {
    size_t size = v.size();
    size_t limit{WLIMIT};
    cout << format("{}: ", size);
    for(auto& s : v) {
      cout << format("{} ", s);
      if(--limit == 0) {
        if(size > WLIMIT) cout << "...";
        break;
      }
    }
    cout << '\n';
  }
  cout << '\n';
}
```

外环和内环分别对应于外 vector 和内 vector。简单地遍历这些 vector，并使用 format("{}: "，size) 打印出内部 vector 的大小，然后使用 format("{} "，s) 打印出每个单词。若不想完整地打印非常长的句子，可以定义了 10 个单词的限制，若有更多，则打印省略号。

- 输出看起来像这样:

```
$ ./sentences < sentences.txt
27: It can be useful for a writer to make sure ...
19: Whenever another container type seems appropriate,
but is missing one ...
18: If you need something ordered, it's often best to use
...
17: The inner vector stores the words of a sentence, and
...
16: In this case, however, since we need a descending
sort, ...
16: In this case, where we need our output sorted in ...
15: As you'll see, this affords a lot of flexibility
while ...
12: Let's build a tool that evaluates a text file for ...
11: The vector is generally the most flexible of the STL
...
9: Choosing the appropriate container key when using the
STL.
7: This recipe uses a vector of vectors.
```

How it works…

使用 C 标准库中的 strchr() 函数查找标点符号非常简单，所有的 C 及其标准库都包含在 C++ 语言的定义中，可以在适当的时候使用。

```
bool is_eos(const string_view & str) {
  constexpr const char * end_punct{ ".!?" };
  for(auto c : str) {
    if(strchr(end_punct, c) != nullptr) return true;
  }
  return false;
}
```

若单词中间有标点符号，这个函数将无法正确地分开句子。这可能发生在某些形式的诗歌或格式糟糕的文本文件中。我曾见过用 std::string 迭代器和正则表达式来实现这一点，但对于我们来说，快速和简单就够了。

使用 cin 逐字读取文本文件:

```
for(string s{}; cin >> s; ) {
  ...
}
```

这避免了一次性将一个大文件读入内存的开销。vector 已经很大，包含文件的所有单词，没有必要将整个文本文件也保存在内存中。在文件过大的情况下，有必要找到另一种策略，或使用数据库。

“vector 的 vector”乍一看可能很复杂，但它并不比使用两个独立的 vector 复杂。

```
vector<vector<string>> vv_sentences{vector<string>{}};
```

这声明了一个外部 vector，内部元素类型为 vector<string>。外层 vector 命名为 vv_sentence。内 vector 是匿名的，不需要命名。这里的定义用一个空 vector<string> 对象初始化 vv_sentence 对象。

当前的内部 vector，可以作为 vv_sentence.back() 使用:

```
vv_sentences.back().emplace_back(s);
```

当完成一个内部 vector 量时，可以简单地创建一个新 vector:

```
vv_sentences.emplace_back(vector<string>{});
```

这将创建一个新的匿名 vector<string> 对象，并将其放置在 vv_sentence 对象的后面。

3.14. 使用 multimap 制作待办事项列表

有序任务列表 (或待办事项列表) 是一种常见的计算应用程序，是一个与优先级相关的任务列表，按反向数字顺序排序。

读者可能倾向于使用 priority_queue，因为它已经按照优先级 (反向数字) 顺序排序了。priority_queue 的缺点是没有迭代器，因此若不从队列中推入或弹出项，就很难对其进行操作。

对于本节的示例，我们将为有序列表使用一个 multimap。multimap 关联容器保持项目的顺序，并且可以使用反向迭代器以适当的排序顺序对其进行访问。

How to do it…

这是一个简短而简单的示例，用于初始化 multimap 并以相反的顺序打印。

- 从 multimap 的类型别名开始:

```
using todomap = multimap<int, string>;
```

todomap 是一个具有 int 键和字符串的 multimap。

- 这里有一个小函数以倒序输出 todomap:

```
void rprint(todomap& todo) {
  for(auto it = todo.rbegin(); it != todo.rend();
  ++it) {
    cout << format("{}: {}\n", it->first,
    it->second);
  }
  cout << '\n';
}
```

可以使用反向迭代器来输出 todomap。

- main() 函数很简单:

```
int main()
{
  todomap todo {
    {1, "wash dishes"},
    {0, "watch teevee"},
    {2, "do homework"},
    {0, "read comics"}
  };
  rprint(todo);
}
```

我们用任务初始化 todomap。任务没有任何特定的顺序，但在键中确实有优先级。rprint() 函数可以按优先级顺序打印它们。

- 输出如下所示:

```
$ ./todo
2: do homework
1: wash dishes
0: read comics
0: watch teevee
```

待办事项列表会按优先级顺序打印出来。

How it works…

这是一个简短而简单的示例，使用 multimap 容器来保存优先级列表中的项。

需要一些说明的仅是 rprint() 函数:

```
void rprint(todomap& todo) {
  for(auto it = todo.rbegin(); it != todo.rend(); ++it) {
    cout << format("{}: {}\n", it->first, it->second);
  }
  cout << '\n';
}
```

注意反向迭代器 rbegin() 和 rend()。不可能改变 multimap 的排序顺序，但是它提供了反向迭代器。这使得 multimap 的行为完全符合我们对优先级列表的要求。

第 4 章 兼容迭代器

迭代器是 STL 中的一个基本概念。迭代器使用 C 指针的语义实现，使用相同的自增、自减和解引用操作符。大多数 C/C++ 程序员都熟悉指针的用法，诸如 std::sort 和 std::transform 等算法，可以在基本内存缓冲区和 STL 容器上工作。

4.1. 迭代器

STL 使用迭代器来导航其容器类的元素，大多数容器都包含 begin() 和 end() 迭代器，通常实现为返回迭代器对象的成员函数。begin() 迭代器指向容器的初始元素，end() 迭代器指向最终元素之后的元素:

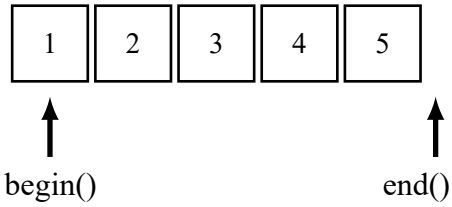


图 4.1 begin() 和 end() 迭代器

end() 迭代器可以作为长度不确定的容器的哨兵。我们将在本章中看到一些例子。

大多数 STL 容器都定义了自己特定的迭代器类型。例如，对于 int 类型的 vector:

```
std::vector<int> v;
```

迭代器类型定义为:

```
std::vector<int>::iterator v_it;
```

可以看到这是多么复杂。若有一个 string 的 vector

```
std::vector<std::vector<int, std::string>> v;
```

其迭代器类型为:

```
std::vector<std::vector<int, std::string>>::iterator v_it;
```

幸运的是，C++11 提供了 auto 类型推断和自动类型。通过使用 auto，很少需要使用完整的迭代器类型定义。例如，在 for 循环中需要迭代器，可以使用 auto 类型:

```
for(auto v_it = v.begin(); v_it != v.end(); ++v_it) {
  cout << *v_it << '\n';
}
```

注意使用解引用操作符从迭代器中访问元素，这和解引用指针的语法一样:

```
const int a[]{ 1, 2, 3, 4, 5 };
size_t count{ sizeof(a) / sizeof(int) };
for(const int* p = a; count > 0; ++p, --count) {
```

```
  cout << *p << '\n';
}
```

可以使用一个基于范围的 for 循环和一个原生数组:

```
const int a[]{ 1, 2, 3, 4, 5 };
for(auto e : a) {
  cout << e << '\n';
}
```

或者使用 STL 容器:

```
std::vector<int> v{ 1, 2, 3, 4, 5 };
for(auto e : v) {
  cout << e << '\n';
}
```

基于范围的 for 循环只是带迭代器的 for 循环的简写:

```
{
  auto begin_it{ std::begin(container) };
  auto end_it{ std::end(container) };
  for ( ; begin_it != end_it; ++begin_it) {
    auto e{ *begin_it };
    cout << e << '\n';
  }
}
```

因为迭代器使用与基元指针相同的语法，所以基于范围的 for 循环对这两种容器的处理都一样。

注意，基于范围的 for 循环调用 std::begin() 和 std::end()，而不是直接调用 begin() 和 end() 成员函数。函数调用成员函数来获取迭代器。为什么不直接调用成员函数呢? 非成员函数也设计用于原始数组。这就是为什么 for 循环适用于数组:

```
const int arr[]{ 1, 2, 3, 4, 5 };
for(auto e : arr) {
  cout << format("{} ", e);
}
```

输出为:

```
1 2 3 4 5
```

通常，我倾向于使用成员函数 begin() 和 end()，因为它们更显式。其他人更喜欢使用 std:: 的非成员函数，因为它们更通用。萝卜白菜，各有所爱; 不过，我建议读者们选择一种风格并坚持下去。

类别

C++20 之前，迭代器根据其功能分为以下几类:

<table>
<tr><th colspan="5">迭代器类别</th><th>迭代器功能</th></tr>
<tr><td rowspan="5">连续
迭代器</td><td rowspan="4">随机
访问迭代器</td><td rowspan="3">双向
迭代器</td><td rowspan="2">前向
迭代器</td><td>输入迭代器</td><td>• 读取
• 递增一次</td></tr>
<tr><td></td><td>• 递增多次</td></tr>
<tr><td colspan="2"></td><td>• 可递减</td></tr>
<tr><td colspan="3"></td><td>• 随机访问</td></tr>
<tr><td colspan="4"></td><td>• 连续存储 (比如数组)</td></tr>
<tr><td colspan="6">当上面任何一个迭代器也可以写入时，也称为可变迭代器。</td></tr>
<tr><td colspan="5">输出迭代器</td><td>• 可写入
• 递增一次</td></tr>
</table>

这些类别是分层级的，功能较强的迭代器继承功能较弱的迭代器的功能，输入迭代器只能读取和递增一次。前向迭代器具有输入迭代器的功能，并且可以多次递增。双向迭代器具有这些功能，还可以自减。

输出迭代器可以写入和递增一次。若其他迭代器也可以写入，则是可变迭代器。

概念

C++20 的概念和约束是新加入的特性。概念只是一个命名约束，将参数的类型限制在模板函数或类中，并帮助编译器选择适当的特化。

C++20 起，STL 用概念而不是类别来定义迭代器。这些概念都在 std:: 命名空间中。

概念	描述
indirectly_readable	迭代器可以由解引用操作符读取。 这包括指针、智能指针和输入迭代器。
indirectly_writeable	迭代器的对象引用是可写的。
weakly_incrementable	这个值可以用 ++ 递增，但不能保持相等。 例如，a==b，但 ++a 可能不等于 ++b。
incrementable	可以用 ++ 增加，并且保持相等。
input_or_output_ iterator	迭代器可以递增和解引用。 每个迭代器都必须满足这个概念。
sentinel_for	哨兵迭代器用于查找大小不确定的对象的结束， 例如输入流。
sized_sentinel_for	哨兵迭代器可以与另一个迭代器和-操作符一起使用， 以确定它在常数时间内的距离。
input_iterator	可读且可加的迭代器。
output_iterator	可写入且可递增的迭代器。
forward_iterator	这将使 input_iterator 具有可递增性。

bidirectional_ iterator	它通过使用–操作符自减添加能力 preserves 来修改 forward_iterator，使其保持相等。
random_access_ iterator	通过添加对 +、+=、-、-= 和 [] 操作符 的支持对 bidirectional_iterator 进行修改。
contiguous_iterator	修改 random_access_iterator 以表示连续的存储。

可以使用这些概念来约束模板的参数:

```
template<typename T>
requires std::random_access_iterator<typename T::iterator>
void printc(const T & c) {
  for(auto e : c) {
    cout << format("{} ", e);
  }
  cout << '\n';
  cout << format("element 0: {}\n", c[0]);
}
```

函数需要一个 random_access_iterator。若用非随机访问容器的列表使用时，编译器会报错:

```
int main()
{
  list<int> c{ 1, 2, 3, 4, 5 };
  printc(c);
}
```

list 迭代器类型不支持 random_access_iterator 概念。所以，编译器又报出了一个错误:

```
error: no matching function for call to 'printc(std::__
cxx11::list<int>&)'
  27 | printc(c);
     | ~~~~~~^~~
note: candidate: 'template<class T> requires random_access_
iterator<typename T::iterator> void printc(const T&)'
  16 | void printc(const T & c) {
     |      ^~~~~~
note: template argument deduction/substitution failed:
note: constraints not satisfied
```

这是 GCC 的错误输出。不同编译器的错误看起来可能不同。

若用一个 vector 来调用，其是一个随机访问容器:

```
int main()
{
  vector<int> c{ 1, 2, 3, 4, 5 };
  printc(c);
```

```
}
```

现在编译和运行都挺好:

```
$ ./working
1 2 3 4 5
element 0: 1
```

虽然针对不同类型的功能 (和概念) 有不同类型的迭代器，但其复杂性是为了支持易用性。

通过对迭代器的介绍，现在让我们继续本章中的主菜：

- 创建可迭代范围
- 使迭代器与 STL 迭代器特性兼容
- 使用迭代器适配器填充 STL 容器
- 创建一个迭代器生成器
- 反向迭代器适配器的反向迭代
- 用哨兵迭代未知长度的对象
- 构建 zip 迭代器适配器
- 创建随机访问迭代器

4.2. 相关准备

本章所有的例子和源代码都可以在本书的 GitHub 库中找到，网址是https://github.com/PacktPublishing/CPP-20-STL-Cookbook/tree/main/chap04。

4.3. 创建可迭代范围

本节示例描述了一个简单的类，其生成一个可迭代范围，适合与基于范围的 for 循环一起使用。其思想是创建一个序列生成器，从开始值迭代到结束值。

为了完成这个任务，我们需要一个迭代器类，以及对象接口类。

How to do it…

这个示例有两个主要部分: 主接口、Seq 和迭代器类。

- 首先，定义 Seq 类，只需要实现 begin() 和 end() 成员函数:

```
template<typename T>
class Seq {
  T start_{};
  T end_{};
public:
  Seq(T start, T end) : start_{start}, end_{end} {}
  iterator<T> begin() const {
    return iterator{start_};
```

```
  }
  iterator<T> end() const { return iterator{end_}; }
};
```

构造函数设置 start_和 end_变量，用于构造 begin() 和 end() 迭代器。成员函数 begin() 和 end() 返回迭代器对象。

- 迭代器类通常定义在容器类的公共部分中，称为成员类或嵌套类。我们将在 Seq 构造函数后面插入它:

```
public:
  Seq(T start, T end) : start_{ start }, end_{ end } {}
  class iterator {
    T value_{};
    public:
    explicit iterator(T position = 0)
    : value_{position} {}
    T operator*() const { return value_; }
    iterator& operator++() {
      ++value_;
      return *this;
    }
    bool operator!=(const iterator& other) const {
      return value_ != other.value_;
    }
  };
```

迭代器类通常命名为 iterator，其类型为 Seq<type>::iterator。

迭代器构造函数是限定显式的，以避免隐式转换。

value_变量由迭代器维护，可以解引用指针返回一个值。

支持基于范围的 for 循环的最低要求是解引用操作符、前自增操作符和不等比较操作符。

- 现在可以写一个 main() 函数来测试序列生成器:

```
int main()
{
  Seq<int> r{ 100, 110 };
  for (auto v : r) {
    cout << format("{} ", v);
  }
  cout << '\n';
}
```

这将构造一个 Seq 对象，并打印其序列。

输出如下所示:

```
$ ./seq
100 101 102 103 104 105 106 107 108 109
```

How it works…

这个示例的重点是制作一个序列生成器，与基于范围的 for 循环一起工作。我们首先考虑基于范围的 for 循环的等效代码:

```
{
  auto begin_it{ std::begin(container) };
  auto end_it{ std::end(container) };
  for ( ; begin_it != end_it; ++begin_it) {
    auto v{ *begin_it };
    cout << v << '\n';
  }
}
```

从这段等价的代码中，可以推出使用 for 循环的要求:

- 具有 begin() 和 end() 迭代器
- 迭代器支持不相等的比较操作符
- 迭代器支持前缀自增运算符
- 迭代器支持解引用操作符

主 Seq 类接口只有三个公共成员函数: 构造函数、begin() 和 end() 迭代器:

```
Seq(T start, T end) : start_{ start }, end_{ end } {}
iterator begin() const { return iterator{start_}; }
iterator end() const { return iterator{end_}; }
```

Seq::iterator 类的实现携带了实际的负载:

```
class iterator {
  T value_{};
```

这是常见的配置，有效负载只能通过迭代器访问。

这里需要实现三个操作符:

```
T operator*() const { return value_; }
iterator& operator++() {
  ++value_;
  return *this;
}
bool operator!=(const iterator& other) const {
  return value_ != other.value_;
}
```

就是基于范围的 for 循环所需的:

```
Seq<int> r{ 100, 110 };
for (auto v : r) {
  cout << format("{} ", v);
}
```

There's more…

通常，会将迭代器定义为容器的成员类，但不是必需的。这允许迭代器类型从属于容器类型:

```
Seq<int>::iterator it = r.begin();
```

因为 auto 类型，C++11 之后具体类型就不那么重要了，但它仍然是最佳实践。

4.4. 使迭代器与 STL 迭代器特性兼容

许多 STL 算法要求迭代器符合某些特征, 但这些要求在编译器、系统和 C++ 版本之间并不一致。

出于学习的目的，我们将使用之前的例子来说明这个问题。

main() 中，添加对 minmax_element() 算法的使用:

```
Seq<int> r{ 100, 110 };
auto [min_it, max_it] = minmax_element(r.begin(), r.end());
cout << format("{} - {}\n", *min_it, *max_it);
```

这将无法通过编译。错误消息也是十分模糊，但仔细观察，会发现迭代器不满足与此算法兼容的要求。

好吧，让我们来解决这个问题。

How to do it…

首先，需要对迭代器做一些简单的处理，使其与算法兼容。我们的迭代器需要满足前向迭代器的最低要求，先从这里开始:

- 目前已经有了前向迭代器所需的其他操作符，唯一缺少的是相等比较运算符。可以使用 operator==() 重载将其添加到迭代器中:

```
bool operator==(const iterator& other) const {
  return value_ == other.value_;
}
```

有趣的是，这会使代码在某些系统上编译和运行，但在 Clang 上不行，Clang 会给出这样的错误消息:

```
No type named 'value_type' in 'std::iterator_
traits<Seq<int>::iterator>'
```

说明需要在迭代器中设置特性。

- iterator_traits 类在迭代器类中查找一组类型定义 (使用别名实现):

```
public:
  using iterator_concept = std::forward_iterator_tag;
  using iterator_category = std::forward_iterator_tag;
  using value_type = std::remove_cv_t<T>;
```

```
  using difference_type = std::ptrdiff_t;
  using pointer = const T*;
  using reference = const T&;
```

我倾向于把它们放在迭代器类 public 的顶部，会更容易看到。

现在就有了一个完全符合前向迭代器的类，代码可以在我当前拥有的编译器上运行。

How it works…

using 语句是用于定义迭代器可以执行哪些功能的特性，来看一下:

```
using iterator_concept = std::forward_iterator_tag;
using iterator_category = std::forward_iterator_tag;
```

前两个是 category 和 concept，都设置为 forward_iterator_tag。该值表示迭代器符合前向迭代器规范。

有些代码不查看这些值，而是查找设置和功能:

```
using value_type = std::remove_cv_t<T>;
using difference_type = std::ptrdiff_t;
using pointer = const T*;
using reference = const T&;
```

value_type 是 std::remove_cv_t<T> 的别名，这是值的类型，可以删除 const 限定符。

difference_type 是 std::ptrdiff_t 的别名，作为指针地址差异的特殊类型。

指针和引用别名分别设置为，指针和引用的 const 限定版本。

定义这些类型别名是大多数迭代器的基本要求。

There's more…

值得注意的是，定义这些特征允许在迭代器中使用概念受限的模板。例如:

```
template<typename T>
requires std::forward_iterator<typename T::iterator>
void printc(const T & c) {
  for(auto v : c) {
    cout << format("{} ", v);
  }
  cout << '\n';
}
```

这个输出序列的函数受到 forward_iterator 概念的限制。若类不符合条件，就不会编译。

还可以使用 ranges:: 版本的算法:

```
auto [min_it, max_it] = ranges::minmax_element(r);
```

这使得迭代器使用起来更加方便。

可以用静态断言的方式来测试 forward_range 的兼容性:

```
static_assert(ranges::forward_range<Seq<int>>);
```

4.5. 使用迭代器适配器填充 STL 容器

迭代器本质上是一种抽象，有一个特定的接口，并以特定的方式使用。除此之外，其代码可以用于其他目的。迭代器适配器是一个看起来像迭代器，但需要做其他事情的类。

STL 附带了各种迭代器适配器，通常与算法库一起使用。STL 迭代器适配器通常分为三类:

- 插入迭代器或插入器用于在容器中插入元素。
- 流迭代器读取和写入流。
- 反向迭代器反转迭代器的方向。

How to do it…

在本节示例中，有一些 STL 迭代器适配器的例子:

- 一个简单的函数，打印容器的内容:

```
void printc(const auto & v, const string_view s = "") {
  if(s.size()) cout << format("{}: ", s);
  for(auto e : v) cout << format("{} ", e);
  cout << '\n';
}
```

printc() 函数可以轻松地查看算法的结果，其包含一个可选的 string_view 参数用于描述。

- main() 函数中，将定义两个 deque 容器。使用的是 deque 容器，所以可以在两端进行插入:

```
int main() {
  deque<int> d1{ 1, 2, 3, 4, 5 };
  deque<int> d2(d1.size());
  copy(d1.begin(), d1.end(), d2.begin());
  printc(d1);
  printc(d2, "d2 after copy");
}
```

输出为:

```
1 2 3 4 5
d2 after copy: 1 2 3 4 5
```

我们为相同数量的元素定义了带有 5 个 int 值的 deque d1 和带有空格的 d2。copy() 算法不会分配空间，因此 d2 必须为元素留有空间。

copy() 算法接受三个迭代器:begin 和 end 迭代器表示要复制的元素范围，begin 迭代器表示目标范围，不检查迭代器以确保它们有效。(尝试不分配空间的 vector，会出现段错误)

我们在两个容器上调用 printc() 来显示结果。

- copy() 算法对此并不总可行。有时想复制并添加元素到容器的末尾，若有一个算法可以为每个元素调用 push_back() 就太好了。这就是迭代器适配器有用的地方。让我们在 main() 的末尾添加一些代码:

```
copy(d1.begin(), d1.end(), back_inserter(d2));
printc(d2, "d2 after back_inserter");
```

输出为:

```
d2 after back_inserter: 1 2 3 4 5 1 2 3 4 5
```

back_inserter() 是一个插入迭代器适配器，为分配给它的每个项调用 push_back()，可以在需要输出迭代器的地方使用。

- 还有一个 front_inserter() 适配器，用于在容器的前端插入元素:

```
deque<int> d3{ 47, 73, 114, 138, 54 };
copy(d3.begin(), d3.end(), front_inserter(d2));
printc(d2, "d2 after front_inserter");
```

输出为:

```
d2 after front_inserter: 54 138 114 73 47 1 2 3 4 5 1 2 3 4 5
```

front_inserter() 适配器使用容器的 push_front() 方法在前端插入元素。因为每个元素都插入到前一个元素之前，所以目标容器中的元素是反向的。

- 若想在中间插入，可以使用 inserter() 适配器:

```
auto it2{ d2.begin() + 2};
copy(d1.begin(), d1.end(), inserter(d2, it2));
printc(d2, "d2 after middle insert");
```

输出为:

```
d2 after middle insert: 54 138 1 2 3 4 5 114 73 47 ...
```

inserter() 适配器接受插入起始点的迭代器。

- 流迭代器可以方便地读写 iostream 对象，这是 ostream_iterator():

```
cout << "ostream_iterator: ";
copy(d1.begin(), d1.end(), ostream_iterator<int>(cout));
cout << '\n';
```

输出为:

```
ostream_iterator: 12345
```

- 下面是 istream_iterator():

```
vector<string> vs{};
copy(istream_iterator<string>(cin),
  istream_iterator<string>(),
  back_inserter(vs));
printc(vs, "vs2");
```

输出为:

```
$ ./working < five-words.txt
vs2: this is not a haiku
```

若没有流迭代器，istream_iterator() 适配器将默认返回一个结束迭代器。

- 反向适配器包含在大多数容器中，如函数成员 rbegin() 和 rend():

```
for(auto it = d1.rbegin(); it != d1.rend(); ++it) {
  cout << format("{} ", *it);
}
cout << '\n';
```

输出为:

```
5 4 3 2 1
```

How it works…

迭代器适配器通过包装现有容器来工作。当调用一个适配器时，比如用容器对象的 back_inserter():

```
copy(d1.begin(), d1.end(), back_inserter(d2));
```

适配器返回一个模拟迭代器的对象，在本例中是 std::back_insert_iterator 对象，在每次赋值给迭代器时调用容器对象上的 push_back() 方法。可以使用适配器代替迭代器，同时执行其他任务。

istream_adapter() 也需要一个哨兵，哨兵表示长度不确定的迭代器的结束。当从一个流中读取时，并不知道流中有多少对象。当流到达结束时，哨兵将与迭代器进行相等比较，标志流的结束。istream_adapter() 在不带参数的情况下调用时会创建一个哨兵:

```
auto it = istream_adapter<string>(cin);
auto it_end = istream_adapter<string>(); // creates sentinel
```

这允许测试流的结束，就像测试其他容器一样:

```
for(auto it = istream_iterator<string>(cin);
    it != istream_iterator<string>();
    ++it) {
  cout << format("{} ", *it);
}
```

```
cout << '\n';
```

输出为:

```
$ ./working < five-words.txt
this is not a haiku
```

4.6. 创建一个迭代器生成器

生成器是生成自己的值序列的迭代器，不使用容器。它动态地创建值，根据需要一次返回一个值。并且，C++ 生成器可以独立运行。

本节示例中，我们将为斐波那契数列构建一个生成器。这是一个数列，其中每个数字都是数列中前两个数字的和，从 0 和 1 开始:

$$F(n) = \begin{cases} 0, n = 0 \\ 1, n = 1 \\ F(n-1) + F(n-2), n > 1 \end{cases}$$

斐波那契数列的前十个值，不包括零，分别是:1,1,2,3,5,8,13,21,34,55。这与自然界的黄金比例非常接近。

How to do it…

斐波那契数列通常是用递归循环创建的。生成器中的递归可能会很困难，而且需要大量资源，因此只保存序列中的前两个值并将它们相加，这样更有效率更高。

- 首先，定义一个函数来打印序列:

```
void printc(const auto & v, const string_view s = "") {
  if(s.size()) cout << format("{}: ", s);
  for(auto e : v) cout << format("{} ", e);
  cout << '\n';
}
```

之前使用过这个 printc() 函数，可打印一个可迭代范围，以及一个描述字符串。

- 我们的类从一个类型别名和一些对象变量开始，所有这些都在 private 部分中定义。

```
class fib_generator {
  using fib_t = unsigned long;
  fib_t stop_{};
  fib_t count_ { 0 };
  fib_t a_ { 0 };
  fib_t b_ { 1 };
```

stop_变量将在后面用作哨兵，设置为要生成的值的数量。count_用于跟踪生成了多少个值。a_和 b_是前两个序列值，用于计算下一个值。

- private 部分，有一个简单的函数来计算斐波那契数列中的下一个值。

```
  constexpr void do_fib() {
    const fib_t old_b = b_;
    b_ += a_;
    a_ = old_b;
  }
```

- public 部分，有一个简单的构造函数，有一个默认值:

```
public:
  explicit fib_generator(fib_t stop = 0) : stop_{ stop
  } {}
```

该构造函数不带参数，用于创建哨兵。stop 参数初始化 stop_变量以表示要生成多少值。

- 其余的公共函数是前向迭代器所期望的操作符重载:

```
  fib_t operator*() const { return b_; }
  constexpr fib_generator& operator++() {
    do_fib();
    ++count_;
    return *this;
  }
  fib_generator operator++(int) {
    auto temp{ *this };
    ++*this;
    return temp;
  }
  bool operator!=(const fib_generator &o) const {
    return count_ != o.count_;
  }
  bool operator==(const fib_generator&o) const {
    return count_ == o.count_;
  }
  const fib_generator& begin() const { return *this; }
  const fib_generator end() const {
    auto sentinel = fib_generator();
    sentinel.count_ = stop_;
    return sentinel;
  }
  fib_t size() { return stop_; }
};
```

还有一个简单的 size() 函数，若需要为复制操作初始化一个目标容器，这个函数会很有用。

- 现在可以在 main 函数中使用生成器

```
printc():
  int main() {
    printc(fib_generator(10));
  }
```

这将创建一个匿名的 fib_generator 对象传递给 printc() 函数。

- 用前 10 个斐波那契数 (不包括零) 可得到如下输出:

```
1 1 2 3 5 8 13 21 34 55
```

How it works…

fib_generator 类作为前向迭代器运行:

```
fib_generator {
public:
  fib_t operator*() const;
  constexpr fib_generator& operator++();
  fib_generator operator++(int);
  bool operator!=(const fib_generator &o) const;
  bool operator==(const fib_generator&o) const;
  const fib_generator& begin() const;
  const fib_generator end() const;
};
```

就基于范围的 for 循环而言，这是一个迭代器 (看起来像一个迭代器)。

该值在 do_fib() 函数中计算:

```
constexpr void do_fib() {
  const fib_t old_b = b_;
  b_ += a_;
  a_ = old_b;
}
```

这只是简单地添加了 b_ += a_，将结果存储在 b_中，将旧的 b_存储在 a_中，为下一次迭代保存必要的值。

解引用操作符返回 b_的值，是序列中的下一个值:

```
fib_t operator*() const { return b_; }
```

end() 函数创建了一个对象，其中 count_变量等于 stop_变量，从而创建了一个哨兵:

```
const fib_generator end() const {
  auto sentinel = fib_generator();
  sentinel.count_ = stop_;
  return sentinel;
}
```

现在，相等比较操作符可以很容易地检测序列的结束:

```
bool operator==(const fib_generator&o) const {
  return count_ == o.count_;
}
```

There's more…

若想让生成器与算法库一起工作，需要提供特性别名。这些放在 public 的顶部:

```
public:
  using iterator_concept = std::forward_iterator_tag;
  using iterator_category = std::forward_iterator_tag;
  using value_type = std::remove_cv_t<fib_t>;
  using difference_type = std::ptrdiff_t;
  using pointer = const fib_t*;
  using reference = const fib_t&;
```

现在可以使用生成器和算法:

```
fib_generator fib(10);
auto x = ranges::views::transform(fib,
  [](unsigned long x){ return x * x; });
printc(x, "squared:");
```

使用 transform() 算法的 ranges::views 版本对每个值进行平方。生成的对象可以用于可以使用迭代器的地方。

可以从 printc() 中获得以下输出:

```
squared:: 1 1 4 9 25 64 169 441 1156 3025
```

4.7. 反向迭代器适配器的反向迭代

反向迭代器适配器是反转迭代器类方向的抽象，需要是一个双向迭代器。

How to do it…

STL 中的大多数双向容器都包含一个反向迭代器适配器。其他容器则没有，如原始 C 数组。来看一些例子:

- 从本章一直使用的 printc() 函数开始:

```
void printc(const auto & c, const string_view s = "") {
  if(s.size()) cout << format("{}: ", s);
  for(auto e : c) cout << format("{} ", e);
  cout << '\n';
}
```

使用一个基于范围的 for 循环来打印容器的元素。

- 基于范围的 for 循环，甚至适用于没有迭代器类的 C 数组。因此，printc() 函数也适用于 C 数组:

```
int main() {
  int array[]{ 1, 2, 3, 4, 5 };
```

```
  printc(array, "c-array");
}
```

会得到这样的输出:

```
c-array: 1 2 3 4 5
```

- 可以使用 begin() 和 end() 迭代器适配器为 C 数组创建普通的前向迭代器:

```
auto it = std::begin(array);
auto end_it = std::end(array);
while (it != end_it) {
  cout << format("{} ", *it++);
}
```

for 循环的输出为:

```
1 2 3 4 5
```

- 或者使用 rbegin() 和 rend() 反向迭代器适配器为 C 数组创建反向迭代器:

```
auto it = std::rbegin(array);
auto end_it = std::rend(array);
while (it != end_it) {
  cout << format("{} ", *it++);
}
```

现在输出反过来了:

```
5 4 3 2 1
```

- 甚至可以创建另一个 printr()，进行反向打印:

```
void printr(const auto & c, const string_view s = "") {
  if(s.size()) cout << format("{}: ", s);
  auto rbegin = std::rbegin(c);
  auto rend = std::rend(c);
  for(auto it = rbegin; it != rend; ++it) {
    cout << format("{} ", *it);
  }
  cout << '\n';
}
```

当用 C 数组使用时:

```
printr(array, "rev c-array");
```

会得到这样的输出:

```
rev c-array: 5 4 3 2 1
```

- 当然，这也适用于双向 STL 容器:

```
vector<int> v{ 1, 2, 3, 4, 5 };
printc(v, "vector");
printr(v, "rev vector");
```

输出为:

```
vector: 1 2 3 4 5
rev vector: 5 4 3 2 1
```

How it works…

普通的迭代器类有一个 begin() 迭代器指向第一个元素，还有一个 end() 迭代器指向最后一个元素之后:

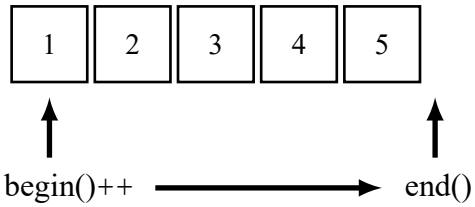


图 4.3 前向迭代器

通过使用自加操作符递增 begin() 迭代器来迭代容器，直到 end() 迭代器的值。

反向迭代器适配器拦截迭代器接口，并将其反转。使 begin() 迭代器指向最后一个元素，end() 迭代器指向第一个元素之前。++ 和–操作符也是颠倒的:

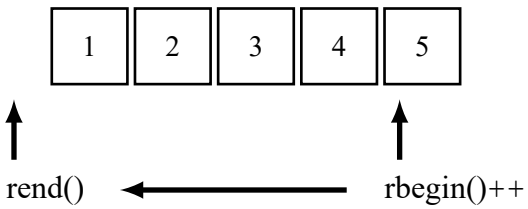


图 4.4 反向迭代器适配器

反向迭代器中，++ 操作符递减，--操作符递增。

大多数双向 STL 容器已经包含了一个反向迭代器适配器，可以通过成员函数 rbegin() 和 rend() 访问:

```
vector<int> v;
it = v.rbegin();
it_end = v.rend();
```

这些迭代器将反向操作，可以用于多种场景。

4.8. 用哨兵迭代未知长度的对象

有些对象没有特定的长度。要想知道长度，需要遍历它们的所有元素。例如，在本章的其他地方，我们已经看到了一个没有特定长度的生成器，更常见的例子是 C 字串。

C-string 是由字符组成的 C 数组，以空值“\0”作为结束。

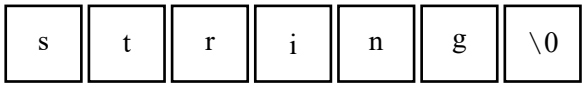


图 4.5 带有空结束符的 C-string

即使没有意识到，我们也一直在使用 C-string。C/C++ 中的字面值字符串都是 C-string:

```
std::string s = "string";
```

这里，STL 字符串 s 初始化为一个字面值字符串。字面值字符串是一个 C 字串。若看一下十六进制中的单个字符，就会看到结束符:

```
for (char c : "string") {
  std::cout << format("{:02x} ", c);
}
```

单词“string”有六个字母。循环的输出显示了数组中的七个元素:

```
73 74 72 69 6e 67 00
```

第 7 个是空——结束符。

循环看到的是包含 7 个值的 C 字符数组，是一个字符串的事实对循环是不可见的抽象。若想让循环像对待字符串一样对待它，需要一个迭代器和一个哨兵。

哨兵是一个对象，标志着一个长度不确定的迭代器的结束。当迭代器到达数据的末尾时，哨兵将与迭代器进行相等比较。

为了了解这是如何工作的，先为 C-string 构建一个迭代器!

How to do it…

要使用带有 C-string 哨兵，需要构建一个自定义迭代器。不需要很复杂，只需要使用基于范围的 for 循环的基本要素即可。

- 先从几个定义开始:

```
using sentinel_t = const char;
constexpr sentinel_t nullchar = '\0';
```

sentinel_t 是 const char 的别名，我们的哨兵就用这个类型。

我们还为空字符结束符定义了常量 nullchar。

- 现在来定义迭代器类型:

```
class cstr_it {
  const char *s{};
public:
  explicit cstr_it(const char *str) : s{str} {}
  char operator*() const { return *s; }
  cstr_it& operator++() {
    ++s;
    return *this;
  }
  bool operator!=(sentinel_t) const {
    return s != nullptr && *s != nullchar;
  }
  cstr_it begin() const { return *this; }
  sentinel_t end() const { return nullchar; }
};
```

这是基于范围的 for 循环所必需的最小值，end() 函数返回一个 nullchar，操作符!=() 重载与 nullchar 进行比较。这就是哨兵所需要的。

- 现在，可以定义一个函数，使用哨兵来打印 C-string:

```
void print_cstr(const char * s) {
  cout << format("{}: ", s);
  for (char c : cstr_it(s)) {
    std::cout << format("{:02x} ", c);
  }
  std::cout << '\n';
}
```

这个函数中，首先打印字符串。然后，使用 format() 函数将每个字符打印为十六进制值。

- 现在，可以在 main() 中调用 print_cstr():

```
int main() {
  const char carray[]{"array"};
  print_cstr(carray);
  const char * cstr{"c-string"};
  print_cstr(cstr);
}
```

输出如下所示:

```
array: 61 72 72 61 79
c-string: 63 2d 73 74 72 69 6e 67
```

注意，这里没有多余的字符，也没有空终止符。因为哨兵告诉 for 循环，在看到 nullchar 时停止。

How it works…

迭代器类的哨兵非常简单，可以通过在 end() 函数中返回空结束符，将其作为哨兵使用:

```
sentinel_t end() const { return nullchar; }
```

然后，可以用不等于比较操作符进行测试:

```
bool operator!=(sentinel_t) const {
  return s != nullptr && *s != nullchar;
}
```

注意，参数只是一个类型 (sentinel_t)。参数类型是函数签名必需的，但我们不需要值，要做的就是将当前迭代器与哨兵进行比较。

当类型或类没有预定的比较终点时，就会使用这种方法。

4.9. 构建 zip 迭代器适配器

许多脚本语言都具有一个将两个序列压缩在一起的函数，zip 操作就可以接受两个输入序列，并为两个输入中的每个位置返回一对值:

考虑两个序列的情况——可以是容器、迭代器或初始化列表:

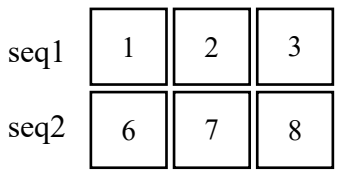


图 4.6 需要压缩的容器

我们想把它们压缩在一起，用前两个序列中的元素对组成一个新序列:

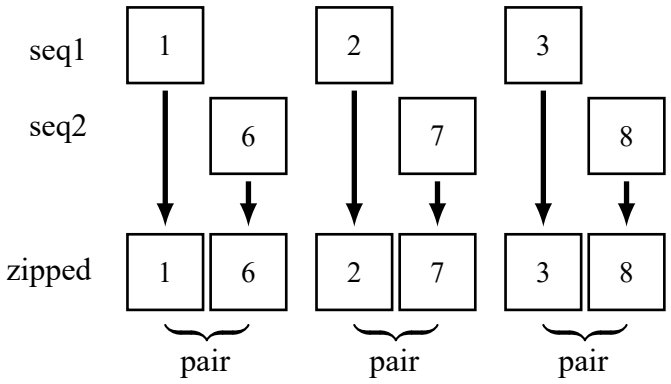


图 4.7 Zip 操作

示例中，将使用迭代器适配器来完成这项任务。

How to do it…

这个示例中，将构建一个 zip 迭代器适配器，接受两个相同类型的容器，并将值压缩到 std::pair 对象中:

- main() 函数中，可以用两个 vector 来调用适配器:

```
int main()
{
  vector<std::string> vec_a {"Bob", "John", "Joni"};
  vector<std::string> vec_b {"Dylan", "Williams",
    "Mitchell"};
  cout << "zipped: ";
  for(auto [a, b] : zip_iterator(vec_a, vec_b)) {
    cout << format("[{}, {}] ", a, b);
  }
  cout << '\n';
}
```

可以使用 zip_iterator 来代替单独的 vector 迭代器。

期望输出如下所示:

```
zipped: [Bob, Dylan] [John, Williams] [Joni, Mitchell]
```

- 迭代器适配器在一个名为 zip_iterator 的类中。为了方便起见，我们将从一些类型别名开始:

```
template<typename T>
class zip_iterator {
  using val_t = typename T::value_type;
  using ret_t = std::pair<val_t, val_t>;
  using it_t = typename T::iterator;
```

- 迭代器中不存储数据，只存储目标容器的 begin() 和 end() 迭代器的副本:

```
it_t ita_{};
```

```
it_t itb_{};
// for begin() and end() objects
it_t ita_begin_{};
it_t itb_begin_{};
it_t ita_end_{};
it_t itb_end_{};
```

ita_和 itb_是目标容器的迭代器，其他四个迭代器用于为 zip_iterator 适配器生成 begin() 和 end() 迭代器。

- 还有一个私有的构造函数:

```
// private constructor for begin() and end() objects
zip_iterator(it_t ita, it_t itb) : ita_{ita}, itb_{itb}
{}
```

稍后将用于构造专门用于 begin() 和 end() 迭代器的适配器对象。

- public 部分，从迭代器特性类型定义开始:

```
public:
using iterator_concept =
  std::forward_iterator_tag;
using iterator_category =
  std::forward_iterator_tag;
using value_type = std::pair<val_t, val_t>;
using difference_type = long int;
using pointer = const val_t*;
using reference = const val_t&;
```

- 构造函数设置所有私有迭代器变量:

```
zip_iterator(T& a, T& b) :
  ita_{a.begin()},
  itb_{b.begin()},
  ita_begin_{ita_},
  itb_begin_{itb_},
  ita_end_{a.end()},
  itb_end_{b.end()}
{}
```

- 定义了用于前向迭代器的最小操作符重载:

```
zip_iterator& operator++() {
  ++ita_;
  ++itb_;
  return *this;
}
bool operator==(const zip_iterator& o) const {
  return ita_ == o.ita_ || itb_ == o.itb_;
}
bool operator!=(const zip_iterator& o) const {
  return !operator==(o);
```

```
}
ret_t operator*() const {
  return { *ita_, *itb_ };
}
```

- 最后，begin() 和 end() 函数返回各自的迭代器:

```
zip_iterator begin() const
  { return zip_iterator(ita_begin_, itb_begin_); }
zip_iterator end() const
  { return zip_iterator(ita_end_, itb_end_); }
```

存储的迭代器和私有构造函数使这些函数变得简单。

- 现在展开 main() 函数进行测试:

```
int main()
{
  vector<std::string> vec_a {"Bob", "John", "Joni"};
  vector<std::string> vec_b {"Dylan", "Williams",
    "Mitchell"};

  cout << "vec_a: ";
  for(auto e : vec_a) cout << format("{} ", e);
  cout << '\n';

  cout << "vec_b: ";
  for(auto e : vec_b) cout << format("{} ", e);
  cout << '\n';

  cout << "zipped: ";
  for(auto [a, b] : zip_iterator(vec_a, vec_b)) {
    cout << format("[{}, {}] ", a, b);
  }
  cout << '\n';
}
```

- 这给了我们期望的输出:

```
vec_a: Bob John Joni
vec_b: Dylan Williams Mitchell
zipped: [Bob, Dylan] [John, Williams] [Joni, Mitchell]
```

How it works…

压缩迭代器适配器是迭代器抽象可以多么灵活的一个例子，可以获取两个容器的迭代器，并在一个聚合迭代器中使用。

zip_iterator 类的主构造函数接受两个容器对象。为了便于讨论，我们将这些对象称为目标对象。

```
zip_iterator(T& a, T& b) :
  ita_{a.begin()},
  itb_{b.begin()},
  ita_begin_{ita_},
  itb_begin_{itb_},
  ita_end_{a.end()},
  itb_end_{b.end()}
{}
```

构造函数从目标 begin() 迭代器初始化 ita_和 itb_变量，这些将用于导航目标对象。目标 begin() 和 end() 迭代器也会保存以供后续使用。

这些变量在 private 部分定义:

```
it_t ita_{};
it_t itb_{};
// for begin() and end() objects
it_t ita_begin_{};
it_t itb_begin_{};
it_t ita_end_{};
it_t itb_end_{};
```

it_t 类型定义为目标迭代器类的类型:

```
using val_t = typename T::value_type;
using ret_t = std::pair<val_t, val_t>;
using it_t = typename T::iterator;
```

其他别名类型是 val_t，表示目标值的类型，ret_t 表示返回值对，整个类中使用这些类型定义是为了方便使用。

begin() 和 end() 函数使用只初始化 ita_和 itb_值的私有构造函数:

```
zip_iterator begin() const
  { return zip_iterator(ita_begin_, itb_begin_); }
zip_iterator end() const
  { return zip_iterator(ita_end_, itb_end_); }
```

私有构造函数:

```
// private constructor for begin() and end() objects
zip_iterator(it_t ita, it_t itb) : ita_{ita}, itb_{itb} {}
```

这是一个使用 it_t 迭代器作为参数的构造函数，只初始化 ita_和 itb_，以便在比较操作符重载中使用。

类的其余部分就像普通的迭代器一样，但操作的是目标类的迭代器:

```
zip_iterator& operator++() {
  ++ita_;
  ++itb_;
  return *this;
}
```

```
bool operator==(const zip_iterator& o) const {
  return ita_ == o.ita_ || itb_ == o.itb_;
}
bool operator!=(const zip_iterator& o) const {
  return !operator==(o);
}
```

解引用操作符返回一个 std::pair 对象 (ret_t 是 std::pair<val_t, val_t> 的别名)。这是从迭代器接口中检索出的值。

```
ret_t operator*() const {
  return { *ita_, *itb_ };
}
```

There's more…

zip_iterator 适配器可以用来将对象压缩到 map 中:

```
map<string, string> name_map{};

for(auto [a, b] : zip_iterator(vec_a, vec_b)) {
  name_map.try_emplace(a, b);
}

cout << "name_map: ";
for(auto [a, b] : name_map) {
  cout << format("[{}, {}] ", a, b);
}
cout << '\n';
```

若将这段代码添加到 main()，则会得到这样的输出:

```
name_map: [Bob, Dylan] [John, Williams] [Joni, Mitchell]
```

4.10. 创建随机访问迭代器

本节中是一个全功能连续/随机访问迭代器的例子，这是容器最完整的迭代器类型。随机访问迭代器包括所有其他类型的容器迭代器的所有特性，以及它的随机访问功能。

虽然在本章中包含一个完整的迭代器很重要，但这个示例的代码超过 700 行，比本书中的其他示例要大一些。我将在这里介绍代码的基本组件。请在此处查看完整的源代码https://github.com/PacktPublishing/CPP-20-STL-Cookbook/blob/main/chap04/container-iterator.cpp。

How to do it…

迭代器需要一个容器。我们将使用一个简单的数组，并将其称为 Container。迭代器类嵌套在 Container 类中。

所有这些设计会与 STL 容器的接口一致。

- 容器定义为模板类，private 部分只有两个元素:

```
template<typename T>
class Container {
  std::unique_ptr<T[]> c_{};
  size_t n_elements_{};
```

我们为数据使用 unique_pointer，让智能指针管理内存。这减少了对 Container() 析构函数的需求。n_elements_变量保留了容器的大小。

- public 部分中，有构造函数:

```
Container(initializer_list<T> l) : n_elements_{l.size()}
{
  c_ = std::make_unique<T[]>(n_elements_);
  size_t index{0};
  for(T e : l) {
    c_[index++] = e;
  }
}
```

第一个构造函数使用 initializer_list 为容器传递元素。使用 make_unique 来分配空间，并用基于范围的 for 循环填充容器。

- 还有一个构造函数，分配空间但不填充元素:

```
Container(size_t sz) : n_elements_{sz} {
  c_ = std::make_unique<T[]>(n_elements_);
}
```

make_unique() 函数的作用是为 element 构造空对象。

- 函数的作用是: 返回元素的个数:

```
size_t size() const {
  return n_elements_;
}
```

- operator[]() 函数，返回一个索引元素:

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

- at() 函数，对索引元素进行边界检查:

```
T& at(const size_t index) const {
  if(index > n_elements_ - 1) {
    throw std::out_of_range(
```

```
      "Container::at(): index out of range"
    );
  }
  return c_[index];
}
```

这与 STL 的用法一致，at() 函数是首选。

- begin() 和 end() 函数使用容器数据的地址调用迭代器构造函数。

```
iterator begin() const { return iterator(c_.get()); }
iterator end() const {
  return iterator(c_.get() + n_elements_);
}
```

unique_ptr::get() 函数从智能指针返回地址。

- 迭代器类作为公共成员嵌套在 Container 类中。

```
class iterator {
  T* ptr_;
```

迭代器类有一个私有成员，在 Container 类的 begin() 和 end() 方法中初始化的指针。

- 迭代器构造函数接受一个指向容器数据的指针。

```
iterator(T* ptr = nullptr) : ptr_{ptr} {}
```

因为标准需要一个默认构造函数，所以这里有一个默认值。

运算符重载

此迭代器为以下操作符提供操作符重载: 前缀 ++，后缀 ++，前缀–，后缀–，[]，默认比较 <=>(C++20)，==，*，->，+，非成员 +，数字-，对象-，+= 和-=。我们将在这里介绍几个值得注意的重载，具体请参阅这些重载的源代码。

- C++20 默认的比较运算符 <=> 提供了全套比较运算符的功能，除了 == 运算符:

```
const auto operator<=>(const iterator& o) const {
  return ptr_ <=> o.ptr_;
}
```

这是 C++20 的特性，所以需要支持 C++20 的编译器和库。

- 有两个 + 操作符重载，支持 it + n 和 n + it 操作。

```
iterator operator+(const size_t n) const {
  return iterator(ptr_ + n);
}
// non-member operator (n + it)
friend const iterator operator+(
    const size_t n, const iterator& o) {
  return iterator(o.ptr_ + n);
}
```

友元声明是一种特殊情况。在模板类成员函数中使用时，相当于一个非成员函数。这允许在类上下文中定义非成员函数。

- 运算符也有两个重载，需要同时支持数值操作数和迭代器操作数。

```
const iterator operator-(const size_t n) {
  return iterator(ptr_ - n);
}
const size_t operator-(const iterator& o) {
  return ptr_ - o.ptr_;
}
```

这允许 it - n 和 it - it 运算。不需要非成员函数，因为 (n -) 并不是有效的操作。

验证

C++20 规范 §23.3.4.13 要求一个有效的随机访问迭代器有一组特定的操作和结果，我在源代码中写了一个 unit_tests() 函数来验证这些需求。

main() 函数创建一个 Container 对象并执行一些简单的验证函数。

- 首先，创建一个 Container<string> 对象 x，其中包含 10 个值。

```
Container<string> x{"one", "two", "three", "four",
  "five",
    "six", "seven", "eight", "nine", "ten" };
cout << format("Container x size: {}\n", x.size());
```

输出给出了元素的数量:

```
Container x size: 10
```

- 用一个基于范围的 for 循环来显示容器的元素:

```
puts("Container x:");
for(auto e : x) {
  cout << format("{} ", e);
}
cout << '\n';
```

输出为:

```
Container x:
one two three four five six seven eight nine ten
```

- 接下来，测试几个直接访问的方法:

```
puts("direct access elements:");
cout << format("element at(5): {}\n", x.at(5));
cout << format("element [5]: {}\n", x[5]);
cout << format("element begin + 5: {}\n",
  *(x.begin() + 5));
cout << format("element 5 + begin: {}\n",
  *(5 + x.begin()));
```

```
cout << format("element begin += 5: {}\n",
  *(x.begin() += 5));
```

输出为:

```
direct access elements:
element at(5): six
element [5]: six
element begin + 5: six
element 5 + begin: six
element begin += 5: six
```

- 用 ranges::views 管道和 views::reverse 来测试容器:

```
puts("views pipe reverse:");
auto result = x | views::reverse;
for(auto v : result) cout << format("{} ", v);
cout << '\n';
```

输出为:

```
views pipe reverse:
ten nine eight seven six five four three two one
```

- 最后，创建一个包含 10 个未初始化元素的 Container 对象 y:

```
Container<string> y(x.size());
cout << format("Container y size: {}\n", y.size());
for(auto e : y) {
  cout << format("[{}] ", e);
}
cout << '\n';
```

输出为:

```
Container y size: 10
[] [] [] [] [] [] [] [] [] []
```

How it works…

尽管有很多代码，但这个迭代器并不比一个较小的迭代器复杂。大多数代码都在操作符重载中，每个重载通常只有一两行代码。

容器本身由智能指针管理。这是因为它是一个“平面”数组，所以不需要展开或压缩。

当然，STL 提供了一个 std::array 类，以及其他更复杂的数据结构。不过，揭开一个迭代器类的神秘面纱，还是挺有趣的。

第 5 章 Lambda 表达式

C++11 标准引入了 lambda 表达式 (有时称为 lambda 函数，或简称 lambda)。该特性允许在表达式的上下文中使用匿名函数。lambda 可用于函数调用、容器、变量和其他表达式上下文。

先简单了解一下 lambda 表达式。

5.1. Lambda

lambda 本质上是一个匿名函数作为字面表达式:

```
auto la = []{ return "Hello\n"; };
```

变量 la 现在可以像函数一样使用:

```
cout << la();
```

可以传递给另一个函数:

```
f(la);
```

也可以传递给另一个 lambda:

```
const auto la = []{ return "Hello\n"; };
const auto lb = [](auto a){ return a(); };
cout << lb(la);
```

输出为

```
Hello
```

或者它可以匿名传递 (作为一个文字):

```
const auto lb = [](auto a){ return a(); };
cout << lb([]{ return "Hello\n"; });
```

闭包

闭包这个术语通常应用于匿名函数。严格地说，闭包是一个允许在自己的词法范围之外使用符号的函数。

可能已经注意到 lambda 定义中的方括号:

```
auto la = []{ return "Hello\n"; };
```

方括号用于指定捕获列表，捕获是可从 lambda 体范围内访问的外部变量。若试图使用外部变量而没有将其列为捕获，会得到一个编译错误:

```
const char * greeting{ "Hello\n" };
const auto la = []{ return greeting; };
cout << la();
```

当尝试用 GCC 编译这个时，会得到了以下的错误:

```
In lambda function:
error: 'greeting' is not captured
```

因为 lambda 的主体有自己的词法作用域，而 greeting 变量在该作用域之外。

可以在捕获中指定 greeting 变量，这允许变量进入 lambda 的作用域:

```
const char * greeting{ "Hello\n" };
const auto la = [greeting]{ return greeting; };
cout << la();
```

现在，程序就能按预期编译和运行:

```
$ ./working
Hello
```

这种捕获自身作用域之外变量的能力使 lambda 成为闭包。人们用不同的方式使用这个词，只要能相互理解就好。尽管如此，了解这个术语的含义还是很有必要的。

lambda 表达式允许我们编写良好、干净的泛型代码，了可以使用函数式编程模式，其中可以使用 lambda 作为算法，甚至作为其他 lambda 函数的参数。

本章中，我们将在以下主题中介绍 lambda 在 STL 中的使用:

- 用于作用域可重用代码
- 算法库中作为谓词
- 与 std::function 一起作为多态包装器
- 用递归连接 lambda
- 将谓词与逻辑连接词结合起来
- 用相同的输入调用多个 lambda
- 对跳转表使用映射 lambda

5.2. 相关准备

本章所有的例子和源代码都可以在本书的 GitHub 库中找到，网址是https://github.com/PacktPublishing/CPP-20-STL-Cookbook/tree/main/chap05。

5.3. 用于作用域可重用代码

Lambda 表达式可以定义和存储供后续使用，也可以作为参数传递，存储在数据结构中，并在不同的上下文中使用不同的参数调用。它们和函数一样灵活，并且具有数据的移动性。

How to do it…

从一个简单的程序开始，可用其来测试 lambda 表达式的各种设置:

- 首先定义一个 main() 函数，并使用它来测试 lambda:

```
int main() {
  ... // code goes here
}
```

- 在 main() 函数内部，将声明两个 lambda。lambda 基本定义需要一对方括号和花括号:

```
auto one = [](){ return "one"; };
auto two = []{ return "two"; }
```

注意，第一个示例 one 在方括号后包含圆括号，而第二个示例 two 则没有。空参数括号通常包括在内，但并不总是必需的。返回类型由编译器推断。

- 可以用 cout 或 format 调用这些函数，或在接受 C-string 的上下文中使用:

```
cout << one() << '\n';
cout << format("{}\n", two());
```

- 通常，编译器可以通过自动类型推断确定返回类型。否则，可以使用-> 操作符指定返回类型:

```
auto one = [] () -> const char * { return "one"; };
auto two = [] () -> auto { return "two"; };
```

lambda 使用尾部返回类型语法，由-> 操作符和类型规范组成。若没有指定返回类型，则认为它是 auto。若使用尾部返回类型，则需要参数括号。

- 来定义一个 lambda 来输出其他 lambda 的值:

```
auto p = [](auto v) { cout << v() << '\n'; };
```

p() 需要一个 lambda(或函数) 作为参数 v，并在函数体中调用它。

auto 类型参数使此 lambda 成为缩写模板。C++20 前，这是创建 lambda 模板的唯一方法。从 C++20 开始，可以在捕获括号之后指定模板参数 (不带 template 关键字)。这与模板参数相同:

```
auto p = []<template T>(T v) { cout << v() << '\n'; };
```

缩写的 auto 版本更简单，也更常见。

- 现在可以在函数调用中传递一个匿名 lambda:

```
p([]{ return "lambda call lambda"; });
```

输出结果为:

```
lambda call lambda
```

- 若需要将参数传递给匿名的 lambda，可以将其放在 lambda 表达式后的括号中:

```
<< [](auto l, auto r){ return l + r; }(47, 73)
  << '\n';
```

函数参数 47 和 73 传递给函数体后括号中的匿名 lambda。

- 可以通过将变量包含在方括号中来访问 lambda 的外部作用域的变量:

```
int num{1};
p([num]{ return num; });
```

- 或者通过引用来捕获它们:

```
int num{0};
auto inc = [&num]{ num++; };
for (size_t i{0}; i < 5; ++i) {
  inc();
}
cout << num << '\n';
```

输出如下所示:

```
5
```

可以修改捕获的变量。

- 也可以定义一个本地捕获变量来维护其状态:

```
auto counter = [n = 0]() mutable { return ++n; };
for (size_t i{0}; i < 5; ++i) {
  cout << format("{}, ", counter());
}
cout << '\n';
```

输出为:

```
1, 2, 3, 4, 5,
```

可变说明符允许 lambda 修改它的捕获，lambda 默认限定为 const。

与尾部返回类型一样，任何说明符都需要参数圆括号。

- lambda 支持两种默认捕获类型:

```
int a = 47;
int b = 73;
auto l1 = []{ return a + b; };
```

若试图编译这段代码，会得到一个错误，其中包括:

```
note: the lambda has no capture-default
```

一种默认捕获类型用等号表示:

```
auto l1 = [=]{ return a + b; };
```

这将捕获 lambda 作用域中的所有符号，等号通过复制执行捕获。将捕获对象的副本，就像使用赋值操作符复制对象一样。

另一个默认捕获使用 & 号进行引用捕获:

```
auto l1 = [&]{ return a + b; };
```

这是通过引用进行捕获方式。

默认捕获只在引用时使用，所以并不像看起来那样混乱。我建议在可能的情况下使用显式捕获，这样可读性更高。

How it works…

lambda 表达式的语法如下:

```
// Syntax of the lambda expression
[capture-list] (parameters)
  mutable    (optional)
  constexpr (optional)
  exception attr (optional)
  -> return type (optional)
{ body }
```

lambda 表达式唯一需要的部分是捕获列表和函数体，函数体可以为空:

```
[]{}
```

这是最简单的 lambda 表达式，什么也没捕捉到，什么也没做。

来了解一下表达式的每一部分。

捕获列表

捕获列表指定我们捕获什么 (如果有的话)。不能省略，但可以是空的。可以使用 [=] 通过复制来捕获所有变量，或者使用 [&] 通过引用来捕获 lambda 范围内的所有变量。

可以通过在括号中列出单个变量来捕获:

```
[a, b]{ return a + b; }
```

指定的捕获要复制的默认值，可以通过引用操作符来获取:

```
[&a, &b]{ return a + b; }
```

当通过引用捕获时，可以修改引用的变量。

> **Note**
>
> 在类内使用 lambda 时，不能直接捕获对象成员，可以捕获 this 或 *this 来解除对类成员的引用。

参数

与函数一样，形参可在括号中指定:

```
[](int a, int b){ return a + b };
```

若没有参数、说明符或尾部返回类型，圆括号是可选的。说明符或尾部返回类型使括号成为必需项:

```
[]() -> int { return 47 + 73 };
```

可变修饰符 (可选)

lambda 表达式默认为 const 限定的，除非指定了可变修饰符。这允许在 const 上下文中使用，但不能修改复制捕获的变量。例如:

```
[a]{ return ++a; };
```

这将导致编译失败，并出现如下错误消息:

```
In lambda function:
error: increment of read-only variable 'a'
```

使用 mutable 修饰符，lambda 不再是 const 限定的，捕获的变量可能会更改:

```
[a]() mutable { return ++a; };
```

constexpr 说明符 (可选)

可以使用 constexpr 显式指定您希望将 lambda 视为常量表达式，其可以在编译时计算。弱 lambda 满足要求，即使没有说明符，也可以将其视为 constexpr。

异常属性 (可选)

可以使用 noexcept 说明符表明 lambda 表达式不抛出任何异常。

尾部的返回类型 (可选)

默认情况下，lambda 返回类型是从 return 语句中推导出来的，就像是一个 wuto 返回类型一样。可以选择使用-> 操作符指定一个尾部返回类型:

```
[](int a, int b) -> long { return a + b; };
```

若使用可选说明符或尾部？返回类型，则必须使用参数括号。

> **Note**
>
> 包括 GCC 在内的一些编译器允许省略空参数括号，即使有说明符或尾随返回类型。这是不正确的！根据规范，当包含参数、说明符和尾部返回类型都是 lambda 声明器的一部分时，都需要括号。这在 C++ 的未来版本中可能会改变。

5.4. 算法库中作为谓词

算法库中的某些函数需要使用谓词函数。谓词是测试条件并返回布尔 true/false 响应的函数 (或函子或 lambda)。

How to do it…

对于这个示例，我们将使用不同类型的谓词来测试 count_if() 算法:

- 首先，创建一个用作谓词的函数。谓词接受一定数量的参数并返回 bool 类型。count_if() 的谓词有一个参数:

```
bool is_div4(int i) {
  return i % 4 == 0;
}
```

该谓词检查 int 值是否能被 4 整除。

- 在 main() 中，定义一个 int 值的 vector，并使用它来测试 count_if() 的谓词:

```
int main() {
  const vector<int> v{ 1, 7, 4, 9, 4, 8, 12, 10, 20 };
  int count = count_if(v.begin(), v.end(), is_div4);
  cout << format("numbers divisible by 4: {}\n",
    count);
}
```

输出如下:

```
numbers divisible by 4: 5
```

(5 个可整除数分别是:4,4,8,12 和 20。)

count_if() 算法使用谓词来确定计算序列中的哪些元素，对每个元素作为参数调用谓词，并且仅在谓词返回 true 时计数元素。

本例中，我们使用函数作为谓词。

- 也可以将函子用作谓词:

```
struct is_div4 {
  bool operator()(int i) {
    return i % 4 == 0;
  }
};
```

这里唯一的变化是需要使用类的实例作为谓词:

```
int count = count_if(v.begin(), v.end(), is_div4());
```

函子的优点是可以携带上下文并访问类和实例变量。在 C++11 引入 lambda 表达式之前，这是使用谓词的常用方法。

- 使用 lambda 表达式，其具有两个优点: 函数的简单性和函子的强大功能。可以使用 lambda 作为变量:

```
auto is_div4 = [](int i){ return i % 4 == 0; };
int count = count_if(v.begin(), v.end(), is_div4);
```

或者可以使用匿名 lambda:

```
int count = count_if(v.begin(), v.end(),
  [](int i){ return i % 4 == 0; });
```

- 可以通过在函数中包装 lambda 来利用 lambda 捕获，并使用该函数上下文生成具有不同参数的相同 lambda:

```
auto is_div_by(int divisor) {
  return [divisor](int i){ return i % divisor == 0; };
}
```

该函数从捕获上下文返回一个带除数的谓词 lambda。

然后，count_if() 可以使用这个谓词

```
for( int i : { 3, 4, 5 } ) {
  auto pred = is_div_by(i);
  int count = count_if(v.begin(), v.end(), pred);
  cout << format("numbers divisible by {}: {}\n", i,
    count);
}
```

每次调用 is_div_by() 都会返回一个与 i 除数不同的谓词。

现在可得到这样的输出:

```
numbers divisible by 3: 2
numbers divisible by 4: 5
numbers divisible by 5: 2
```

How it works…

函数指针的类型表示，为调用函数操作符的指针:

```
void (*)()
```

可以声明一个函数指针，并用函数名对其进行初始化:

```
void (*fp)() = func;
```

声明后，函数指针可以解引用，并像使用函数本身一样使用:

```
func(); // do the func thing
```

lambda 表达式具有与函数指针相同的类型:

```
void (*fp)() = []{ cout << "foo\n"; };
```

无论在哪里使用具有特定签名的函数指针，都可以使用具有相同签名的 lambda。这使得函数指针、函子和 lambda 的工作方式一致:

```
bool (*fp)(int) = is_div4;
bool (*fp)(int) = [](int i){ return i % 4 == 0; };
```

由于这种等价性，像 count_if() 这样的算法接受函数、函子或 lambda(在这些函数中期望具有特定函数签名的谓词)。

这适用于所有使用谓词的算法。

5.5. 与 std::function 一起作为多态包装器

类模板 std::function 是函数的精简多态包装器，可以存储、复制和调用函数、lambda 表达式或其他函数对象，在想要存储对函数或 lambda 的引用的地方很有用。使用 std::function 允许在同一个容器中存储具有不同签名的函数和 lambda，并且其可以维护 lambda 捕获的上下文。

How to do it…

这个示例使用 std::function 类来存储 vector 中 lambda 的不同特化:

- 这个示例包含在 main() 函数中，首先声明三个不同类型的容器:

```
int main() {
  deque<int> d;
  list<int> l;
  vector<int> v;
```

这些容器是，deque, list 和 vector，将由模板 lambda 引用。

- 我们声明一个名为 print_c 的简单 lambda 函数对容器的值进行打印:

```
auto print_c = [](auto& c) {
  for(auto i : c) cout << format("{} ", i);
  cout << '\n';
};
```

- 声明一个返回匿名 lambda 的 lambda:

```
auto push_c = [](auto& container) {
  return [&container](auto value) {
    container.push_back(value);
  };
};
```

push_c 接受对容器的引用，该容器由匿名 lambda 捕获。匿名 lambda 调用捕获容器上的 push_back() 成员。push_c 的返回值是匿名 lambda。

- 现在声明一个 std::function 元素的 vector，并用 push_c() 的三个实例对其进行填充:

```
const vector<std::function<void(int)>>
  consumers { push_c(d), push_c(l), push_c(v) };
```

初始化器列表中的每个元素都是对 push_c 的函数调用。push_c 返回匿名 lambda 的实例，该实例通过函数包装器存储在 vector 中。push_c 是用 d、l 和 v 这三个容器调用的，容器用匿名 lambda 作为捕获进行传递。

- 现在循环遍历 consumers vector，并调用每个 lambda 元素 10 次，每个容器中用 0-9 填充三个容器:

```
for(auto &consume : consumers) {
  for (int i{0}; i < 10; ++i) {
    consume(i);
  }
}
```

- 现在三个容器，deque, list 和 vector，已经用整数填充。现在将它们打印出来:

```
print_c(d);
print_c(l);
print_c(v);
```

输出应是:

```
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
```

How it works…

Lambdas 经常间接使用，这就是一个很好的例子。例如，push_c 返回一个匿名 lambda:

```
auto push_c = [](auto& container) {
  return [&container](auto value) {
    container.push_back(value);
  };
};
```

这个匿名表达式存储在 vector 中:

```
const vector<std::function<void(int)>>
  consumers { push_c(d), push_c(l), push_c(v) };
```

这是 consumers 容器的定义，由三个元素初始化，其中每个元素都通过调用 push_c 进行初始化，该调用返回一个匿名 lambda。存储在 vector 中的是匿名表达式，而不是 push_c。

vector 定义使用 std::function 类作为元素的类型。函数构造函数可以接受任意可调用对象，并将其引用存储为函数目标:

```
template< class F >
function( F&& f );
```

当函数使用函数操作符时，函数对象可以使用参数调用目标函数:

```
for(auto &c : consumers) {
  for (int i{0}; i < 10; ++i) {
    c(i);
  }
}
```

这将调用存储在 consumers 容器中的每个匿名表达式 10 次，从而填充 d、l 和 v 容器。

There's more…

function 类的性质使它在很多方面都很有用，可以将其视为一个多态函数容器。可以存储为一个独立的函数:

```
void hello() {
  cout << "hello\n";
}
int main() {
  function<void(void)> h = hello;
  h();
}
```

可以存储一个成员函数，使用 std::bind 来绑定函数形参:

```
struct hello {
  void greeting() const { cout << "Hello Bob\n"; }
};
int main() {
  hello bob{};
  const function<void(void)> h =
    std::bind(&hello::greeting, &bob);
  h();
}
```

或者可以存储可执行对象:

```
struct hello {
  void operator()() const { cout << "Hello Bob\n"; }
};
int main() {
  const function<void(void)> h = hello();
  h();
}
```

输出如下所示:

```
Hello Bob
```

5.6. 用递归连接 lambda

可以使用一个简单的递归函数来级联 lambda，这样一个的输出就是下一个的输入。这创建了一种在另一个函数上构建一个函数的简单方法。

How to do it…

这是一个简单的示例，使用递归函数来完成大部分工作:

- 首先，定义连接函数 concat():

```
template <typename T, typename ...Ts>
auto concat(T t, Ts ...ts) {
  if constexpr (sizeof...(ts) > 0) {
    return [&](auto ...parameters) {
      return t(concat(ts...)(parameters...));
    };
  } else {
    return t;
  }
}
```

该函数返回一个匿名 lambda，该 lambda 再次调用该函数，直到耗尽参数包。

- main() 函数中，我们创建了一对 lambda，并使用它们调用 concat() 函数:

```
int main() {
  auto twice = [](auto i) { return i * 2; };
  auto thrice = [](auto i) { return i * 3; };
  auto combined = concat(thrice, twice,
    std::plus<int>{});
  std::cout << format("{}\n", combined(2, 3));
}
```

concat() 函数调用时带有三个参数: 两个 lambdas 和 std::plus() 函数。

随着递归的展开，函数从右到左进行，从 plus() 开始。plus() 函数接受两个参数并返回和。plus() 的返回值传递给 twice()，返回值传递给 threice()。

然后，使用 format() 将结果打印到控制台:

```
30
```

How it works…

concat() 函数很简单，但由于间接递归和返回 lambda，可能会令人困惑:

```
template <typename T, typename ...Ts>
auto concat(T t, Ts ...ts) {
  if constexpr (sizeof...(ts) > 0) {
    return [&](auto ...parameters) {
      return t(concat(ts...)(parameters...));
    };
  } else {
    return t;
  }
}
```

concat() 函数使用参数包调用。对于省略号，sizeof…运算符返回参数包中的元素数。这用于测试递归的结束。

concat() 函数返回一个 lambda，递归调用 concat() 函数。因为 concat() 的第一个参数不是参数包的一部分，所以每次递归调用都会剥离包的第一个元素。

外层 return 语句返回 lambda，内部的返回值来自于 lambda。lambda 调用传递给 concat() 的函数，并返回其值。

读者们可以把这个例子仔细拆解，细细研究。这项技术还是很有价值的。

5.7. 将谓词与逻辑连接词结合起来

此示例将 lambda 包装在函数中，以创建用于算法谓词的自定义连接。

How to do it…

copy_if() 算法需要带有一个参数的谓词。在这个示例中，我们将从其他三个 lambda 创建一个谓词 lambda:

- 首先，编写 combine() 函数。这个函数返回一个用于 copy_if() 算法的 lambda:

```
template <typename F, typename A, typename B>
auto combine(F binary_func, A a, B b) {
  return [=](auto param) {
    return binary_func(a(param), b(param));
  };
}
```

combine() 函数接受三个函数参数——一个二元连接符和两个谓词——并返回一个 lambda，该 lambda 调用带有两个谓词的连接符。

- main() 函数中，创建了用于 combine() 的 lambda:

```
int main() {
  auto begins_with = [](const string &s){
    return s.find("a") == 0;
  };
  auto ends_with = [](const string &s){
    return s.rfind("b") == s.length() - 1;
  };
  auto bool_and = [](const auto& l, const auto& r){
    return l && r;
  };
```

begin_with 和 ends_with 是简单的过滤器谓词，用于分别查找以'a' 开头和以'b' 结尾的字符串。bool_and 可对两个参数进行与操作。

- 现在，可以使用 combine() 调用 copy_if 算法:

```
std::copy_if(istream_iterator<string>{cin}, {},
      ostream_iterator<string>{cout, " "},
      combine(bool_and, begins_with,
        ends_with));
cout << '\n';
```

combine() 函数返回一个 lambda，该 lambda 将两个谓词与连接词组合在一起。

输出如下所示:

```
$ echo aabb bbaa foo bar abazb | ./conjunction
aabb abazb
```

How it works…

std::copy_if() 算法需要一个带有一个形参的谓词函数，但连接需要两个形参，每个形参都需要一个参数。我们用一个函数来解决这个问题，该函数返回一个专门用于此上下文的 lambda:

```
template <typename F, typename A, typename B>
auto combine(F binary_func, A a, B b) {
  return [=](auto param) {
    return binary_func(a(param), b(param));
  };
}
```

combine() 函数从三个形参创建一个 lambda，每个形参都是一个函数。返回的 lambda 接受谓词函数所需的一个参数。现在可以用 combine() 函数调用 copy_if():

```
std::copy_if(istream_iterator<string>{cin}, {},
      ostream_iterator<string>{cout, " "},
      combine(bool_and, begins_with, ends_with));
```

这将组合 lambda 传递给算法，以便其在该上下文中进行操作。

5.8. 用相同的输入调用多个 lambda

通过将 lambda 包装在函数中，可以轻松地创建具有不同捕获值的 lambda 的多个实例。可以使用相同的输入调用 lambda 的不同版本。

How to do it…

这是一个用不同类型的大括号包装值的简单例子:

- 首先创建包装器函数 braces():

```
auto braces (const char a, const char b) {
  return [a, b](const char v) {
    cout << format("{}{}{} ", a, v, b);
  };
}
```

braces() 函数包装了一个 lambda，该 lambda 返回一个三值字符串，其中第一个和最后一个值是作为捕获传递给 lambda 的字符，中间的值作为参数传递。

- 在 main() 函数中，使用 braces() 创建了四个 lambda，使用了四组不同的括号:

```
auto a = braces('(', ')');
auto b = braces('[', ']');
auto c = braces('{', '}');
auto d = braces('|', '|');
```

- 现在，可以在简单的 for() 循环中调用 lambda:

```
for( int i : { 1, 2, 3, 4, 5 } ) {
  for( auto x : { a, b, c, d } ) x(i);
  cout << '\n';
}
```

这是两个嵌套的 for() 循环。外部循环只是从 1 数到 5，将一个整数传递给内部循环。内部循环调用带大括号的 lambda 函数。

这两个循环都使用初始化器列表作为基于范围的 for() 循环中的容器。这是一种方便的方法，可以循环使用一小组值。

- 程序的输出如下所示:

```
(1) [1] {1} |1|
(2) [2] {2} |2|
(3) [3] {3} |3|
(4) [4] {4} |4|
(5) [5] {5} |5|
```

输出会显示了每个整数的括号组合。

How it works…

这是一个如何为 lambda 使用包装器的简单示例。braces() 函数使用传递给它的括号构造了一个 lambda:

```
auto braces (const char a, const char b) {
  return [a, b](const auto v) {
    cout << format("{}{}{} ", a, v, b);
  };
}
```

通过将括号 () 函数参数传递给 lambda，可以返回具有该上下文的 lambda。所以，main 函数中的每个赋值操作都带有这些参数:

```
auto a = braces('(', ')');
auto b = braces('[', ']');
auto c = braces('{', '}');
auto d = braces('|', '|');
```

当使用数字调用这些 lambda 时，将返回一个字符串，对应的大括号中会包含该数字。

5.9. 对跳转表使用映射 lambda

当希望从用户或其他输入中选择操作时，跳转表是一种有用的模式，跳转表通常在 if/else 或 switch 结构中实现。在这个示例中，我们将只使用 STL map 和匿名 lambda 构建一个简洁的跳转表。

How to do it…

从 map 和 lambda 构建简单的跳转表很容易。map 提供了简单的索引导航，lambda 可以存储为有效负载。

- 首先，创建一个简单的 prompt() 函数来从控制台获取输入:

```
const char prompt(const char * p) {
  std::string r;
  cout << format("{} > ", p);
  std::getline(cin, r, '\n');

  if(r.size() < 1) return '\0';
  if(r.size() > 1) {
    cout << "Response too long\n";
    return '\0';
  }
  return toupper(r[0]);
}
```

C-string 参数用作提示符，std::getline() 被调用来获取用户的输入。响应存储在 r 中，检查长度。若长度为一个字符，则将其转换为大写并返回。

- 在 main() 函数中，声明并初始化了 lambda 的 map:

```
using jumpfunc = void(*)();
map<const char, jumpfunc> jumpmap {
  { 'A', []{ cout << "func A\n"; } },
  { 'B', []{ cout << "func B\n"; } },
  { 'C', []{ cout << "func C\n"; } },
  { 'D', []{ cout << "func D\n"; } },
  { 'X', []{ cout << "Bye!\n"; } }
};
```

map 容器装载了用于跳转表的匿名 lambda。这些 lambda 可以很容易地调用其他函数或执行简单的任务。

使用别名是为了方便，为 lambda 有效负载使用函数指针类型 void(*)()。若需要更大的灵活性，或者发现它更具可读性，这里可以使用 std::function()。其开销很小:

```
using jumpfunc = std::function<void()>;
```

- 现在可以提示用户输入，并从 map 中选择一个动作:

```
char select{};
while(select != 'X') {
  if((select = prompt("select A/B/C/D/X"))) {
    auto it = jumpmap.find(select);
```

```
    if(it != jumpmap.end()) it->second();
    else cout << "Invalid response\n";
  }
}
```

这就是基于 map 的跳转表，循环直到选择“X”退出。我们使用提示字符串调用 prompt()，在 map 对象上调用 find()，然后使用 it->second() 它调用表达式。

How it works…

map 容器是一个很好的跳转表。简洁明了，易于操作:

```
using jumpfunc = void(*)();
map<const char, jumpfunc> jumpmap {
  { 'A', []{ cout << "func A\n"; } },
  { 'B', []{ cout << "func B\n"; } },
  { 'C', []{ cout << "func C\n"; } },
  { 'D', []{ cout << "func D\n"; } },
  { 'X', []{ cout << "Bye!\n"; } }
};
```

匿名 lambda 作为有效负载存储在映射容器中，键是动作菜单中的字符响应。

可以测试一个键的有效性，并在一个动作中选择一个 lambda 表达式:

```
auto it = jumpmap.find(select);
if(it != jumpmap.end()) it->second();
else cout << "Invalid response\n";
```

这是一个简单而优雅的解决方案，否则只能使用尴尬的分支代码了。

第 6 章 STL 算法

STL 的强大之处在于容器接口的标准化。若容器具有特定的功能，那么该功能的接口很可能是跨容器类型标准化的。这种标准化使得容器可以直接支持某些算法。

例如，若想对 int 类型的 vector 中所有元素求和，可以使用循环:

```
vector<int> x { 1, 2, 3, 4, 5 };
long sum{};
for( int i : x ) sum += i; // sum is 15
```

或者一个算法:

```
vector<int> x { 1, 2, 3, 4, 5 };
auto sum = accumulate(x.begin(), x.end(), 0); // sum is 15
```

同样的语法也适用于其他容器:

```
deque<int> x { 1, 2, 3, 4, 5 };
auto sum = accumulate(x.begin(), x.end(), 0); // sum is 15
```

算法版本不一定更短，但更易于阅读和维护，而且算法通常比等效循环更有效。

C++20 起，范围库提供了一组操作范围和视图的替代算法。本书将在适当的地方演示这些替代方案，有关范围和视图的更多信息，请参阅本书第 1 章中的示例。

大多数算法都在算法头中。一些数值算法，特别是 accumulate()，在 <numeric> 头文件中，而一些与内存相关的算法在 <memory> 头文件中。

我们将使用以下示例中介绍 STL 算法:

- 基于迭代器的复制
- 将容器元素连接到一个字符串中
- std::sort——排序容器元素
- std::transform——修改容器内容
- 查找特定项
- 将容器的元素值限制在 std::clamp 的范围内
- std::sample——采集样本数据集
- 生成有序数据序列
- 合并已排序容器

6.1. 相关准备

本章所有的例子和源代码都可以在本书的 GitHub 库中找到，网址是https://github.com/PacktPublishing/CPP-20-STL-Cookbook/tree/main/chap06。

6.2. 基于迭代器的复制

复制算法通常用于从容器复制到容器，若与迭代器一起工作，则要灵活得多。

How to do it…

这个示例中，将使用 std::copy 和 std::copy_n 进行实验，以便更好地理解其工作原理:

- 从一个打印容器的函数开始:

```
void printc(auto& c, string_view s = "") {
  if(s.size()) cout << format("{}: ", s);
  for(auto e : c) cout << format("[{}] ", e);
  cout << '\n';
}
```

- 在 main() 中，定义了一个 vector，并使用 printc() 将其打印出来:

```
int main() {
  vector<string> v1
  { "alpha", "beta", "gamma", "delta",
    "epsilon" };
  printc(v1);
}
```

会得到这样的输出:

```
v1: [alpha] [beta] [gamma] [delta] [epsilon]
```

- 现在，创建第二个 vector，使其有足够空间复制第一个 vector:

```
vector<string> v2(v1.size());
```

- 使用 std::copy() 算法将 v1 复制到 v2:

```
std::copy(v1.begin(), v1.end(), v2.begin());
printc(v2);
```

std::copy() 算法接受两个用于复制源范围的迭代器和一个用于复制目标范围的迭代器，给定 v1 的 begin() 和 end() 迭代器来复制整个 vector。v2 的 begin() 迭代器作为复制的目标。现在的输出是:

```
v1: [alpha] [beta] [gamma] [delta] [epsilon]
v2: [alpha] [beta] [gamma] [delta] [epsilon]
```

- copy() 算法不为目标分配空间。因此，v2 必须已经有用于复制的空间。或者，使用 back_inserter() 迭代器适配器将元素插入到 vector 的后面:

```
vector<string> v2{};
std::copy(v1.begin(), v1.end(), back_inserter(v2))
```

- 也可以使用 ranges::copy() 算法来复制整个范围。容器对象作为一个范围，因此可以使用 v1 作为源。这里，仍然使用目标容器的迭代器:

```
vector<string> v2(v1.size());
ranges::copy(v1, v2.begin());
```

这也适用于 back_inserter():

```
vector<string> v2{};
ranges::copy(v1, back_inserter(v2));
```

输出为:

```
v2: [alpha] [beta] [gamma] [delta] [epsilon]
```

- 可以使用 copy_n() 复制一定数量的元素:

```
vector<string> v3{};
std::copy_n(v1.begin(), 3, back_inserter(v3));
printc(v3, "v3");
```

第二个参数中，copy_n() 算法是要复制的元素数量的计数。输出结果为:

```
v3: [alpha] [beta] [gamma]
```

- 还有一个 copy_if() 算法，其使用布尔谓词函数来确定要复制哪些元素:

```
vector<string> v4{};
std::copy_if(v1.begin(), v1.end(), back_inserter(v4),
  [](string& s){ return s.size() > 4; });
printc(v4, "v4");
```

还有一个 copy_if() 的范围版本:

```
vector<string> v4{};
ranges::copy_if(v1, back_inserter(v4),
  [](string& s){ return s.size() > 4; });
printc(v4, "v4");
```

输出只包含长度超过 4 个字符的字符串:

```
v4: [alpha] [gamma] [delta] [epsilon]
```

注意，beta 值被排除在外。

- 可以使用这些算法中的任何一个来复制到或从任何序列，包括流迭代器:

```
ostream_iterator<string> out_it(cout, " ");
ranges::copy(v1, out_it)
cout << '\n';
```

输出为:

```
alpha beta gamma delta epsilon
```

How it works…

std::copy() 算法非常简单，等价的函数为:

```
template<typename Input_it, typename Output_it>
Output_it bw_copy(Input_it begin_it, Input_it end_it,
          Output_it dest_it) {
  while (begin_it != end_it) {
    *dest_it++ = *begin_it++;
  }
  return dest_it;
}
```

copy() 函数使用目标迭代器的赋值操作符，从输入迭代器复制到输出迭代器，直到到达输入范围的末尾。

这个算法还有一个版本叫做 std::move()，其移动元素而不是复制:

```
std::move(v1.begin(), v1.end(), v2.begin());
printc(v1, "after move: v1");
printc(v2, "after move: v2");
```

这将执行移动而不是复制赋值。移动操作之后，v1 中的元素将为空，v1 中的元素现在在 v2 中。输出如下所示:

```
after move1: v1: [] [] [] [] []
after move1: v2: [alpha] [beta] [gamma] [delta] [epsilon]
```

还有一个 move() 算法的范围版本，执行相同的操作:

```
ranges::move(v1, v2.begin());
```

这些算法的强大之处在于其简单。通过让迭代器管理数据，这些简单、优雅的函数允许在支持所需迭代器的任何 STL 容器之间无缝复制或移动。

6.3. 将容器元素连接到一个字符串中

有时，库中没有现成的算法来完成任务。可以使用迭代器，使用与算法库相同的方式，编写一个迭代器。

例如，需要使用分隔符将容器中的元素连接到一个字符串中。常见的解决方案是使用简单的 for() 循环:

```
for(auto v : c) cout << v << ', ';
```

这个方案的问题是在尾部留下了一个分隔符:

```
vector<string> greek{ "alpha", "beta", "gamma",
  "delta", "epsilon" };
for(auto v : greek) cout << v << ", ";
cout << '\n';
```

输出为:

```
alpha, beta, gamma, delta, epsilon,
```

这在测试环境中可能没问题，但在生产系统中，后面的逗号是无法接受的。

ranges::views 库有一个 join() 函数，但不提供分隔符:

```
auto greek_view = views::join(greek);
```

views::join() 函数的作用是: 返回一个 ranges::view 对象。这需要一个单独的步骤来显示或转换为字符串。我们可以使用 for() 循环遍历视图:

```
for(const char c : greek_view) cout << c;
cout << '\n';
```

输出如下所示:

```
alphabetagammadeltaepsilon
```

我们需要在元素之间设置一个适当的分隔符。

由于算法库没有满足需求的函数，就需要自己编写一个。

How to do it…

这个示例中，将一个容器的元素用分隔符连接成一个字符串:

- 在 main() 函数中，声明了一个字符串 vector:

```
int main() {
  vector<string> greek{ "alpha", "beta", "gamma",
    "delta", "epsilon" };
  ...
}
```

- 现在，来写一个简单的 join() 函数，使用 ostream 对象用分隔符连接元素:

```
namespace bw {
  template<typename I>
  ostream& join(I it, I end_it, ostream& o,
        string_view sep = "") {
    if(it != end_it) o << *it++;
    while(it != end_it) o << sep << *it++;
```

```
    return o;
  }
}
```

我把它放在 bw 命名空间中，以避免名称冲突。

可以这样与 cout 一起使用:

```
bw::join(greek.begin(), greek.end(), cout, ", ") << '\n';
```

因为其返回 ostream 对象，所以可以在它后面加上 << 来向流中添加换行符。

输出:

```
alpha, beta, gamma, delta, epsilon
```

- 这里通常需要一个字符串，而不是直接写入到 cout。对于返回 string 的版本，可以重载此函数:

```
template<typename I>
string join(I it, I end_it, string_view sep = "") {
  ostringstream ostr;
  join(it, end_it, ostr, sep);
  return ostr.str();
}
```

这也在 bw 名称空间中。这个函数创建一个 ostringstream 对象，传递给 bw::join() 的 ostream 版本，从 ostringstream 对象的 str() 方法返回一个字符串对象。

可以这样使用:

```
string s = bw::join(greek.begin(), greek.end(), ", ");
cout << s << '\n';
```

输出为:

```
alpha, beta, gamma, delta, epsilon
```

- 来添加最后一个重载，让它更容易使用:

```
string join(const auto& c, string_view sep = "") {
  return join(begin(c), end(c), sep);
}
```

这个版本只需要一个容器和一个分隔符，就可以满足大多数用例:

```
string s = bw::join(greek, ", ");
cout << s << '\n';
```

输出为:

```
alpha, beta, gamma, delta, epsilon
```

How it works…

这个示例中的大部分工作由迭代器和 ostream 对象完成:

```
namespace bw {
  template<typename I>
  ostream& join(I it, I end_it, ostream& o,
        string_view sep = "") {
    if(it != end_it) o << *it++;
    while(it != end_it) o << sep << *it++;
    return o;
  }
}
```

分隔符在第一个元素之后，在每个连续元素之间，并在最后一个元素之前停止。所以可以在每个元素之前添加分隔符，跳过第一个，也可以在每个元素之后添加分隔符，跳过最后一个。若测试并跳过第一个元素，逻辑会更简单。我们在 while() 循环之前的代码可以这样做:

```
if(it != end_it) o << *it++;
```

完成了第一个元素，就可以简单地在每个剩余元素之前添加分隔符:

```
while(it != end_it) o << sep << *it++;
```

方便起见，返回 ostream 对象。这允许用户可以向流中添加换行符或其他对象:

```
bw::join(greek.begin(), greek.end(), cout, ", ") << '\n';
```

输出为:

```
alpha, beta, gamma, delta, epsilon
```

There's more…

与标准库算法一样，join() 函数将适用于支持前向迭代器的容器。例如，下面是数字库中的 double 常量列表:

```
namespace num = std::numbers;
list<double> constants { num::pi, num::e, num::sqrt2 };
cout << bw::join(constants, ", ") << '\n';
```

输出为:

```
3.14159, 2.71828, 1.41421
```

其可以与 ranges::view 对象一起工作，就像前面在这个示例中定义的 greek_view 一样:

```
cout << bw::join(greek_view, ":") << '\n';
```

输出为:

```
a:l:p:h:a:b:e:t:a:g:a:m:m:a:d:e:l:t:a:e:p:s:i:l:o:n
```

6.4. std::sort——排序容器元素

我们早已经解决了“如何高效地对可比元素进行排序“的问题，对于大多数应用程序，没有理由重新发明这个轮子。STL 通过 std::sort() 算法提供了一个优秀的排序解决方案。虽然标准没有指定排序算法，但当应用于具有 n 个元素的范围时，最差情况的时间复杂度为 O(nlog(n))。

几十年前，快速排序算法被认为是大多数用途的一个很好的折衷，通常比其他可比算法更快。现在，我们有混合算法，可以根据情况在不同的方法之间进行选择，经常在运行中切换算法。目前大多数 C++ 库都使用一种混合方法，将插入排序和插入排序相结合。通常，std::sort() 也具有出色的性能。

How to do it…

这个示例中，我们将检查 std::sort() 算法。sort() 算法适用于具有随机访问迭代器的容器。这里，我们将使用 int 类型的 vector:

- 我们会从一个函数开始测试容器是否已排序:

```
void check_sorted(auto &c) {
  if(!is_sorted(c.begin(), c.end())) cout << "un";
  cout << "sorted: ";
}
```

使用 std::is_sorted() 算法，并根据结果打印“sorted:”或“unsorted:”。

- 这里需要一个函数来打印 vector:

```
void printc(const auto &c) {
  check_sorted(c);
  for(auto& e : c) cout << e << ' ';
  cout << '\n';
}
```

这个函数使用 check_sorted()，来查看容器排序的状态。

- 现在，可以在 main() 函数中定义并打印一个 int 类型的 vector:

```
int main() {
  vector<int> v{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
  printc(v);
  …
}
```

输出如下所示:

```
sorted: 1 2 3 4 5 6 7 8 9 10
```

- 为了测试 std::sort() 算法，需要一个未排序的 vector。下面是一个简单的随机容器的函数:

```
void randomize(auto& c) {
  static std::random_device rd;
  static std::default_random_engine rng(rd());
  std::shuffle(c.begin(), c.end(), rng);
}
```

std::random_device 类使用系统的硬件熵源。大多数现代系统都有一个，否则库将对其进行模拟。std::default_random_engine() 函数从熵源生成随机数，std::shuffle() 用其来随机化容器。现在，可以对容器调用 randomize() 并打印结果:

```
randomize(v);
printc(v);
```

输出为:

```
unsorted: 6 3 4 8 10 1 2 5 9 7
```

因为它是随机的，每个人的输出可能都会不同。事实上，我每次运行它，也会得到不同的结果:

```
for(int i{3}; i; --i) {
  randomize(v);
  printc(v);
}
```

输出为:

```
unsorted: 3 1 8 5 10 2 7 9 6 4
unsorted: 7 6 5 1 3 9 10 2 4 8
unsorted: 4 2 3 10 1 9 5 6 8 7
```

- 只需使用 std::sort() 就能对 vector 进行排序:

```
std::sort(v.begin(), v.end());
printc(v);
```

输出为:

```
sorted: 1 2 3 4 5 6 7 8 9 10
```

默认情况下，sort() 算法使用小于操作符对提供的迭代器指定范围内的元素进行排序。

- partial_sort() 算法将对容器的一部分进行排序:

```
cout << "partial_sort:\n";
randomize(v);
auto middle{ v.begin() + (v.size() / 2) };
std::partial_sort(v.begin(), middle, v.end());
printc(v);
```

partial_sort() 接受三个迭代器: 起始、中间和末尾。它对容器进行排序，使中间之前的元素排序。中间之后的元素不保证是原来的顺序。输出如下:

```
unsorted: 1 2 3 4 5 10 7 6 8 9
```

注意，前五个元素是有序的。

- partition() 算法不进行任何排序，其会重新排列容器，使某些元素出现在容器的前面:

```
coutrandomize(v);
printc(v);
partition(v.begin(), v.end(), [](int i)
  { return i > 5; });
printc(v);
```

第三个参数是谓词 lambda，将决定哪些元素会移到前面。

输出为:

```
unsorted: 4 6 8 1 9 5 2 7 3 10
unsorted: 10 6 8 7 9 5 2 1 3 4
```

注意，小于 5 的值会移动到容器的前面。

- sort() 算法支持可选的比较函数，该函数可用于非标准比较。例如，给定一个名为 things 的类:

```
struct things {
  string s_;
  int i_;
  string str() const {
    return format("({}, {})", s_, i_);
  }
};
```

可以创建一个 vector:

```
vector<things> vthings{ {"button", 40},
  {"hamburger", 20}, {"blog", 1000},
  {"page", 100}, {"science", 60} };
```

这里需要一个函数对其进行打印:

```
void print_things(const auto& c) {
  for (auto& v : c) cout << v.str() << ' ';
```

```
  cout << '\n';
}
```

- 现在可以对 vector 进行排序和打印:

```
std::sort(vthings.begin(), vthings.end(),
    [](const things &lhs, const things &rhs) {
  return lhs.i_ < rhs.i_;
});
print_things(vthings);
```

输出为:

```
(hamburger, 20) (button, 40) (science, 60) (page, 100)
(blog, 1000)
```

注意比较函数在 i_成员上排序，所以结果是按 i_排序的，可以对 s_成员进行排序:

```
std::sort(vthings.begin(), vthings.end(),
    [](const things &lhs, const things &rhs) {
  return lhs.s_ < rhs.s_;
});
print_things(vthings);
```

现在，得到了这样的输出:

```
(blog, 1000) (button, 40) (hamburger, 20) (page, 100)
(science, 60)
```

How it works…

sort() 函数的工作原理是对由两个迭代器指示的元素范围应用排序算法，用于该范围的开始和结束。

通常，这些算法使用小于操作符来比较元素。当然，也可以使用比较函数，通常以 lambda 形式提供:

```
std::sort(vthings.begin(), vthings.end(),
[](const things& lhs, const things& rhs) {
  return lhs.i_ < rhs.i_;
});
```

比较函数接受两个参数，并返回 bool 类型，其签名相当于:

```
bool cmp(const Type1& a, const Type2& b);
```

sort() 函数使用 std::swap() 来移动元素。这在计算周期和内存使用方面都很高效，这减轻了为读写排序对象分配空间的需要。这也是为什么 partial_sort() 和 partition() 函数，不能保证未排序元素顺序的原因。

6.5. std::transform——修改容器内容

std::transform() 函数非常强大和灵活，是库中最常用的算法之一。它将函数或 lambda 应用于容器中的每个元素，将结果存储在另一个容器中，同时保留原始的元素。

How to do it…

这个示例中，将探索 std::transform() 函数的某些使用方式:

- 从一个打印容器内容的函数开始:

```
void printc(auto& c, string_view s = "") {
  if(s.size()) cout << format("{}: ", s);
  for(auto e : c) cout << format("{} ", e);
  cout << '\n';
}
```

我们将使用它来查看转换的结果。

- 在 main() 函数中，声明两个 vector:

```
int main() {
  vector<int> v1{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
  vector<int> v2;
  printc(v1, "v1");
  ...
}
```

输出 v1 的内容:

```
v1: 1 2 3 4 5 6 7 8 9 10
```

- 现在可以使用 transform() 函数将每个值的平方插入到 v2 中:

```
cout << "squares:\n";
transform(v1.begin(), v1.end(), back_inserter(v2),
  [](int x){ return x * x; });
printc(v2, "v2");
```

transform() 函数有四个参数。前两个是源范围的 begin() 和 end() 迭代器，第三个参数是目标范围的 begin() 迭代器。本例中，使用 back_inserter() 算法将结果插入到 v2 中。第四个参数是变换函数，我们使用 lambda 来对值进行平方。

输出为:

```
squares:
v2: 1 4 9 16 25 36 49 64 81 100
```

- 当然，可以对任何类型使用 transform()，下面是一个将字符串对象的向量转换为小写的例子。首先，需要一个函数来返回字符串的小写值:

```
string str_lower(const string& s) {
  string outstr{};
  for(const char& c : s) {
    outstr += tolower(c);
  }
  return outstr;
}
```

现在可以在 transform 中使用 str_lower() 函数了:

```
vector<string> vstr1{ "Mercury", "Venus", "Earth",
  "Mars", "Jupiter", "Saturn", "Uranus", "Neptune",
  "Pluto" };
vector<string> vstr2;
printc(vstr1, "vstr1");
cout << "str_lower:\n";
transform(vstr1.begin(), vstr1.end(),
  back_inserter(vstr2),
  [](string& x){ return str_lower(x); });
printc(vstr2, "vstr2");
```

这将为 vstr1 中的每个元素调用 str_lower()，并将结果插入 vstr2。结果为:

```
vstr: Mercury Venus Earth Mars Jupiter Saturn Uranus
Neptune Pluto
str_lower:
vstr: mercury venus earth mars jupiter saturn uranus
neptune pluto
```

(是的，冥王星对我来说永远都是行星。)

- transform 还有一个范围版本:

```
cout << "ranges squares:\n";
auto view1 = views::transform(v1, [](int x){
  return x * x; });
printc(view1, "view1");
```

范围版本具有更简洁的语法，并返回一个视图对象，而不是填充另一个容器。

How it works…

std::transform() 函数的工作原理与 std::copy() 相似，只不过增加了用户提供的函数。输入范围内的每个元素都传递给函数，函数的返回值会复制赋值给目标迭代器。

值得注意的是，transform() 并不保证元素将按顺序处理。若需要确保转换的顺序，需要使用 for 循环:

```
v2.clear(); // reset vector v2 to empty state
```

```
for(auto e : v1) v2.push_back(e * e);
printc(v2, "v2");
```

输出为:

```
v2: 1 4 9 16 25 36 49 64 81 100
```

6.6. 查找特定项

算法库包含一组用于在容器中查找元素的函数。std::find() 函数及其派生函数在容器中依次搜索并返回指向第一个匹配元素的迭代器，若没有匹配则返回 end()。

How to do it…

find() 算法适用于满足前向或输入迭代器条件的容器。对于这个示例，我们将使用 vector 容器。find() 算法按顺序搜索容器中第一个匹配的元素。先来看几个例子:

- 首先，在 main() 函数中声明一个 int 类型的 vector:

```
int main() {
  const vector<int> v{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
  ...
}
```

- 现在，搜索值为 7 的元素:

```
auto it1 = find(v.begin(), v.end(), 7);
if(it1 != v.end()) cout << format("found: {}\n", *it1);
else cout << "not found\n";
```

find() 算法接受三个参数:begin() 和 end() 迭代器，以及要搜索的值。算法会返回其找到的第一个元素的迭代器，若搜索未能找到匹配，则返回 end() 迭代器。

输出为:

```
found: 7
```

- 也可以寻找比标量更复杂的对象，相应的对象需要支持相等比较运算符。下面是一个简单的结构体，带有 operator==() 的重载:

```
struct City {
  string name{};
  unsigned pop{};
  bool operator==(const City& o) const {
    return name == o.name;
  }
  string str() const {
```

```
    return format("[{}, {}]", name, pop);
  }
};
```

注意，operator=() 重载只比较 name 成员。

这还包含了一个 str() 函数，其返回 City 元素的字符串形式。

- 现在可以声明一个 City 元素的 vector:

```
const vector<City> c{
  { "London", 9425622 },
  { "Berlin", 3566791 },
  { "Tokyo", 37435191 },
  { "Cairo", 20485965 }
};
```

- 可以像搜索 int vector 一样搜索 City vector:

```
auto it2 = find(c.begin(), c.end(), City{"Berlin"});
if(it2 != c.end()) cout << format("found: {}\n",
  it2->str());
else cout << "not found\n";
```

输出为:

```
found: [Berlin, 3566791]
```

- 若想搜索 pop 成员而不是 name，可以使用 find_if() 函数和一个谓词:

```
auto it3 = find_if(begin(c), end(c),
  [](const City& item)
    { return item.pop > 20000000; });
if(it3 != c.end()) cout << format("found: {}\n",
  it3->str());
else cout << "not found\n";
```

谓词会测试 pop 成员，可以得到如下输出:

```
found: [Tokyo, 37435191]
```

- 注意，find_if() 的结果只返回满足谓词的第一个元素，尽管 vector 中有两个元素的 pop 值大于 20,000,000。

  find() 和 find_if() 函数只返回一个迭代器。范围库提供了 ranges::views::filter()，这是一个视图适配器，可为我们提供所有匹配的迭代器位置，而不会干扰到 vector:

```
auto vw1 = ranges::views::filter(c,
  [](const City& c){ return c.pop > 20000000; });
for(const City& e : vw1) cout << format("{}\n", e.str());
```

输出两个匹配的元素:

```
[Tokyo, 37435191]
[Cairo, 20485965]
```

How it works…

find() 和 find_if() 函数依次在容器中搜索，检查每个元素，直到找到为止。若找到匹配项，则返回指向该匹配项的迭代器。若到达 end() 迭代器而没有找到匹配项，则返回 end() 迭代器表明没有找到匹配项。

find() 函数接受三个参数，begin() 和 end() 迭代器，以及一个搜索值。其签名是这样的:

```
template<class InputIt, class T>
constexpr InputIt find(InputIt, InputIt, const T&)
```

find_if() 函数使用谓词，而不是值:

```
template<class InputIt, class UnaryPredicate>
constexpr InputIt find_if(InputIt, InputIt, UnaryPredicate)
```

There's more…

find() 函数都按顺序搜索并在找到第一个匹配时返回。若想找到更多匹配的元素，可以使用范围库中的 filter() 函数:

```
template<ranges::viewable_range R, class Pred>
constexpr ranges::view auto ranges::views::filter(R&&, Pred&&);
```

filter() 函数的作用是返回一个视图，其指向容器，其中只包含筛选过的元素。然后，可以像使用其他容器一样使用视图:

```
auto vw1 = std::ranges::views::filter(c,
  [](const City& c){ return c.pop > 20000000; });
for(const City& e : vw1) cout << format("{}\n", e.str());
```

输出为:

```
[Tokyo, 37435191]
[Cairo, 20485965]
```

6.7. 将容器的元素值限制在 std::clamp 的范围内

C++17 中添加了 std::clamp() 函数，可将标量数值的范围限制在最小值和最大值之间。该函数经过优化，可以使用移动语义，以获得最大的速度和效率。

How to do it…

可以使用 clamp() 在循环中使用容器，或者使用 transform() 算法来限制容器的值。让我们来看一些例子。

- 从一个简单的输出容器值的函数开始:

```
void printc(auto& c, string_view s = "") {
  if(s.size()) cout << format("{}: ", s);
  for(auto e : c) cout << format("{:>5} ", e);
  cout << '\n';
}
```

注意格式字符串“{:>5}”。对于表格视图，这将每个值右对齐到 5 个空格。

- 在 main() 函数中，定义一个用于容器的初始化式列表。这允许我们多次使用相同的值:

```
int main() {
  auto il = { 0, -12, 2001, 4, 5, -14, 100, 200,
    30000 };
  ...
}
```

对于使用 clamp() 来说，这是一个值的范围。

- 先来定义一些常数作为极值:

```
constexpr int ilow{0};
constexpr int ihigh{500};
```

在调用 clamp() 时使用这些值。

- 现在可以在 main() 函数中定义一个容器，这里将使用 int 类型的 vector:

```
vector<int> voi{ il };
cout << "vector voi before:\n";
printc(voi);
```

使用初始化器列表中的值，输出如下:

```
vector voi before:
0 -12 2001 4 5 -14 100 200 30000
```

- 现在可以使用一个带有 clamp() 的 for 循环来限制值在 0 到 500 之间:

```
cout << "vector voi after:\n";
for(auto& e : voi) e = clamp(e, ilow, ihigh);
printc(voi);
```

这将 clamp() 函数应用于容器中的每个值，分别使用 0 和 500 作为下限和上限，输出是:

```
vector voi before:
0 -12 2001 4 5 -14 100 200 30000
vector voi after:
0 0 500 4 5 0 100 200 500
```

clamp() 操作后，负值为 0，大于 500 的值为 500。

- 可以用 transform() 算法做同样的事情，在 lambda 中使用 clamp()。这次使用一个 list 容器:

```
cout << "list loi before:\n";
list<int> loi{ il };
printc(loi);
transform(loi.begin(), loi.end(), loi.begin(),
  [=](auto e){ return clamp(e, ilow, ihigh); });
cout << "list loi after:\n";
printc(loi);
```

输出结果与使用 for 循环的版本相同:

```
list loi before:
0 -12 2001 4 5 -14 100 200 30000
list loi after:
0 0 500 4 5 0 100 200 500
```

How it works…

clamp() 算法是一个简单的函数:

```
template<class T>
constexpr const T& clamp( const T& v, const T& lo,
    const T& hi ) {
  return less(v, lo) ? lo : less(hi, v) ? hi : v;
}
```

若 v 的值小于 lo，则返回 lo。若 hi 小于 v，则返回 hi。

我们的例子中，可以使用 for 循环对容器应用 clamp():

```
for(auto& v : voi) v = clamp(v, ilow, ihigh);
```

这里还在 lambda 中使用了 transform() 算法和 clamp():

```
transform(loi.begin(), loi.end(), loi.begin(),
  [=](auto v){ return clamp(v, ilow, ihigh); });
```

示例中，两个版本给出了相同的结果，并且都从 GCC 编译器生成了类似的代码。编译的大小略有不同 (使用 for 循环的版本更小，正如预期的那样)，性能上的差异可以忽略不计。

我更喜欢 for 循环，但 transform() 版本在其他应用中可能更灵活。

6.8. std::sample——采集样本数据集

std::sample() 算法获取值序列的随机样本，并用该样本填充目标容器。其对于分析更大的数据集很有用，其中随机样本可用来代表整个数据。

样本集允许近似一个大数据集的特征，而不需要分析整个数据集。这提供了效率与准确性的交换，在许多情况下是一种公平的交换。

How to do it…

这个示例中，我们将使用一个具有标准正态分布的 200,000 个随机整数数组。我们将对几百个值进行采样，以创建每个值频率的直方图。

- 我们将从一个简单的函数开始，从一个 double 型返回一个四舍五入的 int 型。标准库缺少这样一个函数，我们稍后会创建它:

```
int iround(const double& d) {
  return static_cast<int>(std::round(d));
}
```

  标准库提供了几个版本的 std::round()，包括一个返回长 int 的版本。但我们需要一个 int，这是一个简单的解决方案，可以避免编译器关于缩小转换的警告，同时隐藏难看的 static_cast。

- main() 函数中，可从一些有用的常量开始:

```
int main() {
  constexpr size_t n_data{ 200000 };
  constexpr size_t n_samples{ 500 };
  constexpr int mean{ 0 };
  constexpr size_t dev{ 3 };
  ...
}
```

  我们有 n_data 和 n_samples 的值，分别用于数据和样本容器的大小。也有均值和 dev 的值，随机值正态分布的均值和标准差参数。

- 现在，设置随机数生成器和分布对象，用于初始化源数据集:

```
std::random_device rd;
std::mt19937 rng(rd());
std::normal_distribution<> dist{ mean, dev };
```

  random_device 对象提供对硬件随机数生成器的访问。mt19937 类是 Mersenne Twister 随机数算法的实现，这是一种高质量的算法，在我们所使用的这种大小的数据集的大多数系统上都能很好地执行。正态分布类提供了均值附近的随机数分布，并提供了标准差。

- 现在可以用 n_data 个数的随机 int 值填充一个数组:

```
array<int, n_data> v{};
for(auto& e : v) e = iround(dist(rng));
```

  数组容器的大小固定，因此模板参数包含一个 size_t 值，表示要分配的元素数量。这里使用 for() 循环填充数组。

rng 对象是硬件随机数生成器，其可传递给 normal_distribution 对象的 dist()，然后传递给 round()，我们的整数舍入函数。

- 此时，就有一个包含 200,000 个数据点的数组。这有很多要分析的，我们将使用 sample() 算法取 500 个值的样本:

```
array<int, n_samples> samples{};
sample(data.begin(), data.end(), samples.begin(),
  n_samples, rng);
```

我们定义另一个数组对象来保存样本，这个的大小是 n_samples。然后，使用 sample() 算法用 n_samples 随机数据点填充数组。

- 我们创建一个直方图来分析样本。map 结构是完美的，可以很容易地映射每个值的频率:

```
std::map<int, size_t> hist{};
for (const int i : samples) ++hist[i];
```

for() 循环从样本容器中获取每个值，并将其用作 map 中的键。增量表达式 ++hist[i] 计算样本集中每个值出现的次数。

- 我们使用 C++20 的 format() 函数打印出直方图:

```
constexpr size_t scale{ 3 };
cout << format("{:>3} {:>5} {:<}/{}\n",
  "n", "count", "graph", scale);
for (const auto& [value, count] : hist) {
  cout << format("{:>3} ({:>3}) {}\n",
    value, count, string(count / scale, '*'));
}
```

format() 说明符看起来像 {:>3} 为一定数量的字符腾出空间。尖括号指定左对齐或右对齐。

string(count, char) 构造函数创建一个字符串，其中一个字符重复指定的次数，在本例中是 n 个星号字符 *，其中 n 是 count/scale，即直方图中某个值的频率，除以缩放常数。

输出如下所示:

```
$ ./sample
n count graph/3
-9 ( 2)
-7 ( 5) *
-6 ( 9) ***
-5 ( 22) *******
-4 ( 24) ********
-3 ( 46) ***************
-2 ( 54) ******************
-1 ( 59) *******************
 0 ( 73) ************************
 1 ( 66) **********************
 2 ( 44) **************
 3 ( 34) ***********
 4 ( 26) ********
 5 ( 18) ******
 6 ( 9) ***
 7 ( 5) *
 8 ( 3) *
 9 ( 1)
```

这是直方图的一个很好的图形表示。第一个数字是值，第二个数字是值的频率，星号是频率的直观表示，其中每个星号代表样本集中出现的规模 (3) 次。

每次运行代码时输出都会有所不同。

How it works…

std::sample() 函数的作用是: 从源容器中的随机位置选择特定数量的元素，并将其复制到目标容器中。

sample() 的签名是这样的:

```
OutIter sample(SourceIter, SourceIter, OutIter,
  SampleSize, RandNumGen&&);
```

前两个参数是包含完整数据集的容器上的 begin() 和 end() 迭代器。第三个参数是用于示例目标的迭代器。第四个参数是样本大小，最后一个参数是随机数生成器函数。

sample() 算法采用均匀分布，因此每个数据点的采样概率相同。

6.9. 生成有序数据序列

排列有许多用例，包括测试、统计、研究等。next_permutation() 算法通过将容器重新排序到下一个字典排列来生成排列。

How to do it…

对于这个示例，我们将打印出一组三个字符串的排列:

- 首先创建一个简短的函数来打印容器的内容:

```
void printc(const auto& c, string_view s = "") {
  if(s.size()) cout << format("{}: ", s);
  for(auto e : c) cout << format("{} ", e);
  cout << '\n';
}
```

使用简单的函数来打印数据集和排列。

- main() 函数中，声明了一个字符串对象的 vector，并使用 sort() 算法对其进行排序。

```
int main() {
  vector<string> vs{ "dog", "cat", "velociraptor" };
  sort(vs.begin(), vs.end());
  ...
}
```

next_permutation() 函数需要一个已排序的容器。

- 现在可以在 do 循环中使用 next_permutation() 列出排布情况:

```
do {
  printc(vs);
} while (next_permutation(vs.begin(), vs.end()));
```

next_permutation() 函数修改容器，若有另一个排列则返回 true 若没有则返回 false。
输出列出了三种宠物的六种排列:

```
cat dog velociraptor
cat velociraptor dog
dog cat velociraptor
dog velociraptor cat
velociraptor cat dog
velociraptor dog cat
```

How it works…

std::next_permutation() 算法生成一组值的字典排列，即基于字典排序的排列。必须对输入进行排序，因为算法按字典顺序逐级进行排列。所以，若从 3,2,1 这样的集合开始，其会立即终止，因为这三个元素的最后一个字符是字典顺序的最后一个。

例如:

```
vector<string> vs{ "velociraptor", "dog", "cat" };
do {
  printc(vs);
} while (next_permutation(vs.begin(), vs.end()));
```

输出为:

```
velociraptor dog cat
```

虽然术语 lexicographically 意味着字母顺序，但实现使用标准比较操作符，因此其适用于任何可排序的值。

同样，若集合中的值重复，则仅根据字典序对它们进行计数。这里有一个 int 类型的 vector，其有两个五个值的重复序列:

```
vector<int> vi{ 1, 2, 3, 4, 5, 1, 2, 3, 4, 5 };
sort(vi.begin(), vi.end());
printc(vi, "vi sorted");
long count{};
do {
  ++count;
} while (next_permutation(vi.begin(), vi.end()));
cout << format("number of permutations: {}\n", count);
```

输出为:

```
Vi sorted: 1 1 2 2 3 3 4 4 5 5
number of permutations: 113400
```

这些值有 113,400 种排列。注意，不是 10!(3,628,800)，因为有些值重复。因为 3,3 和 3,3 排序相同，所以其是字典序。

换句话说，若列出这个短集合的排列:

```
vector<int> vi2{ 1, 3, 1 };
sort(vi2.begin(), vi2.end());
do {
  printc(vi2);
} while (next_permutation(vi2.begin(), vi2.end()));
```

由于有重复，所以只有三种排列，不是 3 的阶乘 (9):

```
1 1 3
1 3 1
3 1 1
```

6.10. 合并已排序容器

merge() 算法接受两个已排序的序列，并创建第三个已合并并排序的序列。这种技术通常用作归并排序的一部分，允许将非常大量的数据分解成块，分别排序，并合并到一个排序的目标中。

How to do it…

对于这个示例，我们将使用 std::merge() 将两个已排序的 vector 容器，合并至第三个 vector。

- 从打印容器内容函数开始:

```
void printc(const auto& c, string_view s = "") {
  if(s.size()) cout << format("{}: ", s);
  for(auto e : c) cout << format("{} ", e);
  cout << '\n';
}
```

我们将使用它来打印源序列和目标序列。

- main() 函数中，将声明源 vector 和目标 vector，并将它们打印出来:

```
int main() {
  vector<string> vs1{ "dog", "cat",
    "velociraptor" };
  vector<string> vs2{ "kirk", "sulu", "spock" };
  vector<string> dest{};
  printc(vs1, "vs1");
  printc(vs2, "vs2");
  ...
}
```

输出为:

```
vs1: dog cat velociraptor
vs2: kirk sulu spock
```

- 现在可以对 vector 进行排序并再次打印:

```
sort(vs1.begin(), vs1.end());
sort(vs2.begin(), vs2.end());
printc(vs1, "vs1 sorted");
printc(vs2, "vs2 sorted");
```

输出为:

```
vs1 sorted: cat dog velociraptor
vs2 sorted: kirk spock sulu
```

- 现在源容器已经排序，可以将其合并:

```
merge(vs1.begin(), vs1.end(), vs2.begin(), vs2.end(),
  back_inserter(dest));
printc(dest, "dest");
```

输出为:

```
dest: cat dog kirk spock sulu velociraptor
```

该输出表示将两个源合并为一个排序的 vector。

How it works…

merge() 算法使用 begin() 和 end() 迭代器，分别来自源迭代器和目标迭代器的输出迭代器:

```
OutputIt merge(InputIt1, InputIt1, InputIt2, InputIt2, OutputIt)
```

其接受两个输入范围，执行归并/排序操作，并将结果序列发送给输出迭代器。

第 7 章 字符串、流和格式化

STL 字符串类是一个功能强大的全功能工具，用于存储、操作和显示基于字符的数据。在高级脚本语言中，可以找到的许多字符串相关的便利、快速和敏捷的功能。

string 类基于 basic_string，这是一个连续的容器类，可以用字符类型实例化。其类签名是这样的:

```
template<
  typename CharT,
  typename Traits = std::char_traits<CharT>,
  typename Allocator = std::allocator<CharT>
> class basic_string;
```

Trait 和 Allocator 模板参数通常保留默认值。

basic_string 的底层存储是一个连续的 CharT 序列，可以通过 data() 成员函数访问:

```
const std::basic_string<char> s{"hello"};
const char * sdata = s.data();
for(size_t i{0}; i < s.size(); ++i) {
  cout << sdata[i] << ' ';
}
cout << '\n';
```

输出为:

```
h e l l o
```

data() 成员函数返回一个指向底层字符数组的 CharT*。从 C++11 起，data() 返回的数组以空结束，使得 data() 等价于 c_str()。

basic_string 类包含许多在其他连续存储类中可以找到的方法，包括 insert()、erase()、push_back()、pop_back() 等，这些方法可以操作底层的 CharT 数组。

std::string 是 std::basic_string<char> 类型的别名:

```
using std::string = std::basic_string<char>;
```

对于大多数情况，都会使用 std::string。

7.1. String 格式化

字符串格式化一直是 STL 的弱点。直到最近，还只能在笨重的 STL iostream 和过时的遗留 printf() 之间做选择。C++20 起加入了格式库，STL 字符串格式终于发展起来了。新的格式库紧密地基于 Python 的 str.format() 方法，快速灵活，结合了 iostreams 和 printf() 的许多优点，以及良好的内存管理和类型安全。

有关格式库的更多信息，请参阅第 1 章中使用新格式库的示例。

虽然不再需要使用 iostream 进行字符串格式化，但其对于其他目的仍然有用，包括文件和流 I/O，以及一些类型转换。

- 轻量级字符串对象——string_view
- 连接字符串
- 转换字符串
- 使用格式库格式化文本
- 删除字符串中的空白
- 从用户输入中读取字符串
- 统计文件中的单词数
- 使用文件输入初始化复杂结构体
- 使用 char_traits 定义一个字符串类
- 用正则表达式解析字符串

7.2. 相关准备

本章所有的例子和源代码都可以在本书的 GitHub 库中找到，网址是https://github.com/PacktPublishing/CPP-20-STL-Cookbook/tree/main/chap07。

7.3. 轻量级字符串对象——string_view

string_view 类提供了 string 类的轻量级替代方案。string_view 不维护自己的数据存储，而是对 C-string 的视图进行操作。这使得 string_view 比 std::string 更小、更高效。在需要字符串对象，但不需要更多内存，对于计算密集型的 std::string 很有用。

How to do it…

string_view 类看起来与 STL string 类相似，但工作方式略有不同。来看一些例子:

- 这是一个从 C-string (char 数组) 初始化的 STL 字符串:

```
char text[]{ "hello" };
string greeting{ text };
text[0] = 'J';
cout << text << ' ' << greeting << '\n';
```

输出为:

```
Jello hello
```

当修改数组时，字符串不会改变，因为 string 构造函数创建了底层数据副本。

- 当对 string_view 执行同样的操作时，会得到不同的结果:

```
char text[]{ "hello" };
string_view greeting{ text };
text[0] = 'J';
cout << text << ' ' << greeting << '\n';
```

输出为:

```
Jello Jello
```

string_view 构造函数创建底层数据的视图，但不保留自己的副本。这会更高效，但也会有副作用。

- 因为 string_view 不复制底层数据，源数据必须在 string_view 对象存在期间保持在作用域内。所以，这是不行的:

```
string_view sv() {
  const char text[]{ "hello" }; // temporary storage
  string_view greeting{ text };
  return greeting;
}
int main() {
  string_view greeting = sv(); // data out of scope
  cout << greeting << '\n'; // output undefined
}
```

因为在 sv() 函数返回后底层数据超出了作用域，所以 main() 中 greeting 对象使用它时不再有效。

- string_view 类具有对底层数据有意义的构造函数, 包括字符数组 (const char*)、连续范围 (包括 std::string) 和其他 string_view 对象。下面的例子使用了范围构造函数:

```
string str{ "hello" };
string_view greeting{ str };
cout << greeting << '\n';
```

输出为:

```
hello
```

- 还有一个 stringview 文字操作符 sv，定义在 std::literals 命名空间中:

```
using namespace std::literals;
cout << "hello"sv.substr(1, 4) << '\n';
```

这构造了一个 constexpr 的 string_view 对象，并调用 substr() 来获得从索引 1 开始的 4 个值。
输出为:

```
ello
```

How it works…

string_view 类实际上是一个连续字符序列上的迭代器适配器。实现通常有两个成员: 一个 const CharT * 和一个 size_t。工作原理是在源数据周围包装一个 contiguous_iterator。

可以像 std::string 那样使用，但有一些区别:

- 复制构造函数不复制数据，当复制 string_view 时，复制操作都会对相同的底层数据进行:

```
char text[]{ "hello" };
string_view sv1{ text };
string_view sv2{ sv1 };
string_view sv3{ sv2 };
string_view sv4{ sv3 };
cout << format("{} {} {} {}\n", sv1, sv2, sv3, sv4);
text[0] = 'J';
cout << format("{} {} {} {}\n", sv1, sv2, sv3, sv4);
```

输出为:

```
hello hello hello hello
Jello Jello Jello Jello
```

- 将 string_view 传递给函数时，会使用复制构造函数:

```
void f(string_view sv) {
  if(sv.size()) {
    char* x = (char*)sv.data(); // dangerous
    x[0] = 'J'; // modifies the source
  }
  cout << format("f(sv): {} {}\n", (void*)sv.data(),
    sv);
}
int main() {
  char text[]{ "hello" };
  string_view sv1{ text };
  cout << format("sv1: {} {}\n", (void*)sv1.data(),
    sv1);
  f(sv1);
  cout << format("sv1: {} {}\n", (void*)sv1.data(),
    sv1);
}
```

输出为:

```
sv1: 0x7ffd80fa7b2a hello
f(sv): 0x7ffd80fa7b2a Jello
sv1: 0x7ffd80fa7b2a Jello
```

因为复制构造函数不会对底层数据进行复制，底层数据的地址 (由 data() 成员函数返回) 对于 string_view 的所有实例都相同。尽管 string_view 成员指针是 const 限定的，但可以取消 const 限定符。但不建议这样做，因为可能会导致意想不到的副作用。但需要注意的是，数据永远不会进行复制。

- string_view 类缺少直接操作底层字符串的方法。在 string 中支持的 append()、operator+()、push_back()、pop_back()、replace() 和 resize() 等方法在 string_view 中不支持。
  若需要用加法操作符连接字符串，需要使用 std::string。例如，这对 string_view 不起作用:

```
sv1 = sv2 + sv3 + sv4; // does not work
```

  这时需要使用 string 代替:

```
string str1{ text };
string str2{ str1 };
string str3{ str2 };
string str4{ str3 };

str1 = str2 + str3 + str4; // works
cout << str1 << '\n';
```

  输出为:

```
JelloJelloJello
```

7.4. 连接字符串

C++ 中，有几种方法可以连接字符串。本节中，我们将研究三个最常见的: 字符串类 operator+()、字符串类 append() 函数和 ostringstream 类 operator<<()。C++20 中新增了 format() 函数。

How to do it…

本示例中，将研究连接字符串的方法。然后，考虑不同的用例，执行一些基准测试。

- 从两个 std::string 对象开始:

```
string a{ "a" };
string b{ "b" };
```

  string 对象是由字面 C-string 构造的。
  C-string 构造函数生成字面值字符串的副本，并使用本地副本作为字符串对象的底层数据。
- 现在，构造一个新的空字符串对象，并用分隔符和换行符连接 a 和 b:

```
string x{};
x += a + ", " + b + "\n";
cout << x;
```

  这里，使用字符串对象的 += 和 + 操作符来连接 a 和 b 字符串，以及字面字符串 “,” 和 “\n”。
  结果字符串包含连接在一起的元素:

```
a, b
```

- 可以使用 string 对象的 append() 成员函数:

```
string x{};
x.append(a);
x.append(", ");
x.append(b);
x.append("\n");
cout << x;
```

结果相同:

```
a, b
```

- 或者，可以构造一个 ostringstream 对象，使用流接口:

```
ostringstream x{};
x << a << ", " << b << "\n";
cout << x.str();
```

得到了相同的结果:

```
a, b
```

- 也可以使用 C++20 的 format() 函数:

```
string x{};
x = format("{}, {}\n", a, b);
cout << x;
```

还是相同的结果:

```
a, b
```

How it works…

string 对象有两种不同的方法用于连接字符串: 加法操作符和 append() 成员函数。

append() 成员函数的作用是: 将数据添加到字符串对象的数据末尾，必须分配和管理内存来完成这个任务。

加法操作符使用 operator+() 重载构造一个包含新旧数据的新字符串对象，并返回新对象。

ostringstream 对象的工作方式类似于 ostream，但将其输出存储为字符串使用。

C++20 的 format() 函数使用带有可变参数的格式字符串，并返回一个新构造的字符串对象。

There's more…

如何决定哪种连接策略适合自己的代码? 可以从一些基准测试开始。

基准测试

我在 Debian Linux 上使用 GCC 11 执行了这些测试:

- 首先，使用 <chrono> 库创建一个计时器函数:

```
using std::chrono::high_resolution_clock;
using std::chrono::duration;

void timer(string(*f)()) {
  auto t1 = high_resolution_clock::now();
  string s{ f() };
  auto t2 = high_resolution_clock::now();
  duration<double, std::milli> ms = t2 - t1;
  cout << s;
  cout << format("duration: {} ms\n", ms.count());
}
```

timer 函数调用传递给它的函数，标记函数调用前后的时间，并使用 cout 显示耗时。

- 现在，使用 append() 成员函数创建了一个连接字符串的函数:

```
string append_string() {
  cout << "append_string\n";
  string a{ "a" };
  string b{ "b" };
  long n{0};
  while(++n) {
    string x{};
    x.append(a);
    x.append(", ");
    x.append(b);
    x.append("\n");
    if(n >= 10000000) return x;
  }
  return "error\n";
}
```

为了进行基准测试，该函数将重复连接 1000 万次。使用 timer() 从 main() 调用这个函数:

```
int main() {
  timer(append_string);
}
```

得到这样的输出:

```
append_string
a, b
duration: 425.361643 ms
```

所以，在这个系统上，连接字符串在大约 425 毫秒内运行了 1000 万次。

- 现在，用加法操作符重载创建相同的函数:

```
string concat_string() {
  cout << "concat_string\n";
  string a{ "a" };
  string b{ "b" };
  long n{0};
  while(++n) {
    string x{};
    x += a + ", " + b + "\n";
    if(n >= 10000000) return x;
  }
  return "error\n";
}
```

基准测试输出:

```
concat_string
a, b
duration: 659.957702 ms
```

这个版本在大约 660 毫秒内执行了 1000 万次。

- 现在，让用 ostringstream 试试:

```
string concat_ostringstream() {
  cout << "ostringstream\n";
  string a { "a" };
  string b { "b" };
  long n{0};
  while(++n) {
    ostringstream x{};
    x << a << ", " << b << "\n";
    if(n >= 10000000) return x.str();
  }
  return "error\n";
}
```

基准测试输出:

```
ostringstream
a, b
duration: 3462.020587 ms
```

这个版本在 3.5 秒内运行了 1000 万次迭代。

- 下面是 format() 版本:

```
string concat_format() {
```

```
  cout << "append_format\n";
  string a{ "a" };
  string b{ "b" };
  long n{0};
  while(++n) {
    string x{};
    x = format("{}, {}\n", a, b);
    if(n >= 10000000) return x;
  }
  return "error\n";
}
```

基准测试输出:

```
append_format
a, b
duration: 782.800547 ms
```

format() 版本在大约 783 毫秒内运行了 1000 万次迭代。

- 结果总结:

连接方式	基准 (毫秒)
append()	425 ms
operator+()	660 ms
format()	783 ms
ostringstream	3,462 ms

连接字符串的性能比较

为什么会出现性能差异?

可以从这些基准测试中看到，ostringstream 版本比基于字符串的版本花费的时间长很多倍。

append() 方法比加法运算符略快，需要分配内存，但不构造新对象。由于重复进行，所以内部可能有一些优化。

加法操作符重载可能调用 append() 方法，所以会比 append() 方法还要慢。

format() 版本创建了一个新的字符串对象，但没有 iostream 的开销。

ostringstream 操作符 << 重载为每个操作创建一个新的 ostream 对象。考虑到流对象的复杂性，以及对流状态的管理，这使得它比基于字符串其他版本都要慢得多。

如何选择?

其中会涉及到一些个人偏好的因素。操作符重载 (+ 或 <<) 可以很方便，若不考虑性能的话。

与 string 方法相比，ostringstream 类有一个明显的优势: 针对每种不同的类型，有相应的 << 操作符，因此能够在不同类型调用相同代码的情况下进行操作。

format() 函数提供了相同的类型安全和自定义选项，并且比 ostringstream 类快得多。

string 对象的加法操作符重载快速、易于使用、易于读取，但比 append() 慢了一点点。

append() 版本是最快的，但需要为每个项使用单独的函数。

通常，我更喜欢 format() 函数或字符串对象的加法运算符。若性能很重要，我会使用 append()。我将在需要 ostringstream 独特特性，且性能不是问题的地方使用它。

7.5. 转换字符串

string 类是一个连续容器，很像 vector 或数组，并且支持 contiguous_iterator 和所有算法。

string 类是具有 char 类型的 basic_string 的特化，所以容器的元素是 char 类型的。其他特化也可用，但字符串是最常见的。

其本质上是一个连续的 char 元素容器，所以 string 可以与 transform() 算法或其他使用 contiguous_iterator 的技术一起使用。

How to do it…

根据应用程序的不同，有几种方法可以进行转换。本示例将探索其中的一些。

- 我们将从几个谓词函数开始，谓词函数接受一个转换元素并返回一个相关元素。例如，这是一个返回大写字符的谓词:

```
char char_upper(const char& c) {
  return static_cast<char>(std::toupper(c));
}
```

这个函数是对 std::toupper() 的包装。由于 toupper() 函数返回 int 类型，而字符串元素是 char 类型，因此不能在转换中直接使用 toupper() 函数。

下面是一个相应的 char_lower() 函数:

```
char char_lower(const char& c) {
  return static_cast<char>(std::tolower(c));
}
```

- 用于演示目的，rot13() 函数是一个简单的转换谓词。这是一个简单的替换密码，不适合真正的加密，可以用于做模糊处理:

```
char rot13(const char& x) {
  auto rot13a = [](char x, char a)->char {
    return a + (x - a + 13) % 26;
  };
  if (x >= 'A' && x <= 'Z') return rot13a(x, 'A');
  if (x >= 'a' && x <= 'z') return rot13a(x, 'a');
  return x;
}
```

- 可以在 transform() 算法中使用这些谓词:

```
main() {
  string s{ "hello jimi\n" };
  cout << s;
  std::transform(s.begin(), s.end(), s.begin(),
    char_upper);
  cout << s;
  ...
```

transform() 函数对 s 中的每个元素调用 char_upper()，将结果放回 s，并将所有字符转换为大写:

```
hello jimi
HELLO JIMI
```

- 除了 transform()，还可以使用一个简单的带有谓词函数的 for 循环:

```
for(auto& c : s) c = rot13(c);
cout << s;
```

结果是:

```
URYYB WVZV
```

- rot13 密码的有趣之处在于可以快速破解。因为 ASCII 字母表中有 26 个字母，旋转 13 然后再旋转 13 会得到原始字符串。让我们再次转换为小写字母和 rot13 来恢复我们的字符串:

```
for(auto& c : s) c = rot13(char_lower(c));
cout << s;
```

结果是:

```
hello jimi
```

由于其统一接口，谓词函数可以作为彼此的参数链接起来。也可以使用 char_lower(rot13(c)) 得到同样的结果。

- 若需求过于复杂，无法进行简单的逐字符转换，则可以像处理任何连续容器一样使用字符串迭代器。下面是一个简单的函数，通过大写第一个字符和空格后面的每个字符将小写字符串转换为 Title Case:

```
string& title_case(string& s) {
  auto begin = s.begin();
  auto end = s.end();
  *begin++ = char_upper(*begin); // first element
  bool space_flag{ false };
  for(auto it{ begin }; it != end; ++it) {
    if(*it == ' ') {
```

```
      space_flag = true;
    } else {
      if(space_flag) *it = char_upper(*it);
      space_flag = false;
    }
  }
  return s;
}
```

因为它返回一个对转换后的字符串的引用，可以用 cout 调用，像这样:

```
cout << title_case(s);
```

输出为:

```
Hello Jimi
```

How it works…

std::basic_string 类及其特化 (包括 string) 由完全兼容 contiguous_iterator 的迭代器支持，所以适用于任何连续容器的技术，也适用于字符串。

> **Note**
>
> 这些转换将不适用于 string_view 对象，因为底层数据是 const 限定的。

7.6. 使用格式库格式化文本

C++20 引入了新的 format() 函数，该函数以字符串形式返回参数的格式化表示。format() 使用 python 风格的格式化字符串，具有简洁的语法、类型安全，以及出色的性能。

format() 函数接受一个格式字符串和一个模板形参包作为参数:

```
template< class... Args >
string format(const string_view fmt, Args&&... args );
```

格式化字符串使用大括号 {} 作为格式化参数的占位符:

```
const int a{47};
format("a is {}\n", a);
```

输出为:

```
a is 47
```

还使用大括号作为格式说明符，例如:

```
  format("Hex: {:x} Octal: {:o} Decimal {:d} \n", a, a, a);
```

输出为:

```
Hex: 2f Octal: 57 Decimal 47
```

本示例展示了如何将 format() 函数，用于一些常见的字符串格式化解决方案。

> **Note**
>
> 本章是使用 Windows 10 上的 Microsoft Visual C++ 编译器预览版开发的。撰写本文时，这是唯一完全支持 C++20 <format> 库的编译器。最终的实现可能在某些细节上有所不同。

How to do it…

使用 format() 函数来考虑一些常见的格式化解决方案:

- 先从一些变量开始格式化:

```
const int inta{ 47 };
const char * human{ "earthlings" };
const string_view alien{ "vulcans" };
const double df_pi{ pi };
```

pi 常数在 <numbers> 头文件和 std::numbers 命名空间中。

- 可以使用 cout 显示变量:

```
cout << "inta is " << inta << '\n'
<< "hello, " << human << '\n'
<< "All " << alien << " are welcome here\n"
<< "π is " << df_pi << '\n';
```

得到这样的输出:

```
a is 47
hello, earthlings
All vulcans are welcome here
π is 3.14159
```

- 现在，来看看 format() 如何对它们进行处理:

```
cout << format("Hello {}\n", human);
```

这是 format() 函数的最简单形式，格式字符串有一个占位符 {} 和一个对应的变量 human。输出结果为:

```
Hello earthlings
```

- format() 函数返回一个字符串，我们使用 cout<< 来显示该字符串。
  format() 库的建议包括一个 print() 函数，使用与 format() 相同的参数，这就可以打印格式化的字符串:

```
print("Hello {}\n", cstr);
```

  但 print() 没有进入 C++20，但有望加入 C++23。
  我们可以用一个简单的函数，使用 vformat() 提供相同的功能:

```
template<typename... Args>
constexpr void print(const string_view str_fmt,
Args&&... args) {
  fputs(std::vformat(str_fmt,
  std::make_format_args(args...)).c_str(),
  stdout);
}
```

  这个简单的单行函数提供了一个有用的 print() 函数，可以用它来代替 cout << format() 组合:

```
print("Hello {}\n", human);
```

  输出为:

```
Hello earthlings
```

  该函数的更完整版本可以在示例文件的 include 目录中找到。
- 格式字符串还提供了位置选项:

```
print("Hello {} we are {}\n", human, alien);
```

  输出为:

```
Hello earthlings we are vulcans
```

  可以在格式字符串中使用位置选项来改变参数的顺序:

```
print("Hello {1} we are {0}\n", human, alien);
```

  现在，可以得到这样的输出:

```
Hello vulcans we are earthlings
```

  注意，参数保持不变。只有大括号中的位置值发生了变化。位置索引是从零开始的，就像 [] 操作符一样。
  这个特性对于国际化 (或本地化) 非常有用，因为不同的语言在句子中，可以使用不同的顺序。
- 数字有很多格式化选项:

```
print("π is {}\n", df_pi);
```

输出为:

```
π is 3.141592653589793
```

可以指定精度的位数:

```
print("π is {:.5}\n", df_pi);
```

输出为:

```
π is 3.1416
```

冒号字符“:”用于分隔位置索引和格式化参数:

```
print("inta is {1:}, π is {0:.5}\n", df_pi, inta);
```

输出为:

```
inta is 47, π is 3.1416
```

- 若想要一个值占用一定的空间，可以这样指定字符的数量:

```
print("inta is [{:10}]\n", inta);
```

输出为:

```
inta is [ 47]
```

可以向左或向右对齐:

```
print("inta is [{:<10}]\n", inta);
print("inta is [{:>10}]\n", inta);
```

输出为:

```
inta is [47 ]
inta is [  47]
```

默认情况下，用空格字符填充，但可以进行修改:

```
print("inta is [{:*<10}]\n", inta);
print("inta is [{:0>10}]\n", inta);
```

输出为:

```
inta is [47********]
inta is [0000000047]
```

还可以将值居中:

```
print("inta is [{:^10}]\n", inta);
print("inta is [{:_^10}]\n", inta);
```

输出为:

```
inta is [ 47   ]
inta is [____47____]
```

- 可以将整数格式化为十六进制、八进制或默认的十进制表示形式:

```
print("{:>8}: [{:04x}]\n", "Hex", inta);
print("{:>8}: [{:4o}]\n", "Octal", inta);
print("{:>8}: [{:4d}]\n", "Decimal", inta);
```

输出为:

```
    Hex: [002f]
  Octal: [ 57]
Decimal: [ 47]
```

注意，这里使用右对齐来排列标签。

大写十六进制使用大写 X:

```
print("{:>8}: [{:04X}]\n", "Hex", inta);
```

输出为:

```
    Hex: [002f]
```

> **Tip**
>
> 默认情况下，Windows 使用不常见的字符编码。最新版本可能默认为 UTF-16 或 UTF-8 BOM。旧版本可能默认为“代码页”1252，这是 ISO 8859-1 ASCII 标准的超集。Windows 系统默认为更常见的 UTF-8 (No BOM)。
>
> 默认情况下，Windows 不会显示标准 UTF-8 π 字符。要使 Windows 兼容 UTF-8 编码 (以及其他编码)，请在测试时使用编译器开关/utf-8 并在命令行上发出命令 chcp 65001。现在，你也可以得到 π 并使用它。

How it works…

<format> 库使用模板形参包将参数传递给格式化器，需要单独检查参数的类和类型。标准库函数 make_format_args() 接受一个形参包并返回一个 format_args 对象，该对象需要提供格式化参数的类型擦除列表。

可以在 print() 函数中看到这些:

```
template<typename... Args>
constexpr void print(const string_view str_fmt, Args&&... args)
{
  fputs(vformat(str_fmt,
  make_format_args(args...)).c_str(),
  stdout);
}
```

make_format_args() 函数的作用是: 接受一个参数包并返回 format_args 对象。vformat() 函数的作用是: 接受格式字符串和 format_args 对象，并返回一个 std::string。然后，使用 c_str() 方法来获取用于 fputs() 的 C 字符串。

There's more…

对于自定义类重载 ostream << 操作符是常见的做法。例如，给定一个保存分数值的类 Frac:

```
template<typename T>
struct Frac {
  T n;
  T d;
};
...
Frac<long> n{ 3, 5 };
cout << "Frac: " << n << '\n';
```

我们想把对象打印成 3/5 这样的分数。因此，需要编写一个简单的操作符 << 特化，就像这样:

```
template <typename T>
std::ostream& operator<<(std::ostream& os, const Frac<T>& f) {
  os << f.n << '/' << f.d;
  return os;
}
```

现在输出是:

```
Frac: 3/5
```

为了为我们的自定义类提供 format() 支持，需要创建一个特化的格式化器对象:

```
template <typename T>
struct std::formatter<Frac<T>> : std::formatter<unsigned> {
  template <typename Context>
  auto format(const Frac<T>& f, Context& ctx) const {
```

```
    return format_to(ctx.out(), "{}/{}", f.n, f.d);
  }
};
```

formatter 类的特化重载了它的 format() 方法。简单起见，我们继承了 formatter<unsigned> 特化。format() 方法使用 Context 对象调用，该对象为格式化的字符串提供输出上下文。再将 ctx.out 传入 format_to() 函数，将返回一个正常格式字符串。

现在，可以在 Frac 类中使用 print() 函数:

```
print("Frac: {}\n", n);
```

格式化器现在可以识别我们的类，并提供我们想要的输出:

```
Frac: 3/5
```

7.7. 删除字符串中的空白

用户的输入通常在字符串的一端或两端包含无关的空格。这可能会有问题，所以经常需要移除它。在这个示例中，我们将使用字符串类方法 find_first_not_of() 和 find_last_not_of() 来去除字符串末尾的空白。

How to do it…

string 类包含用于查找字符列表中包含或不包含的元素的方法，我们将使用这些方法来修剪字符串:

- 从定义字符串开始，输入假设来自一个的“十指”用户:

```
int main() {
  string s{" \t ten-thumbed input \t \n \t "};
  cout << format("[{}]\n", s);
  ...
```

输入在内容前后有一些额外的制表符\t 和换行符\n 个字符。我们用括号打印它，以显示空格:

```
[           ten-thumbed input
       ]
```

- 下面是一个 trimstr() 函数，用于删除字符串两端的所有空白字符:

```
string trimstr(const string& s) {
  constexpr const char * whitespace{ " \t\r\n\v\f" };
  if(s.empty()) return s;
  const auto first{ s.find_first_not_of(whitespace) };
  if(first == string::npos) return {};
  const auto last{ s.find_last_not_of(whitespace) };
  return s.substr(first, (last - first + 1));
}
```

将空白字符集定义为空格、制表符、返回符、换行符、垂直制表符和换行符。其中一些比另一些更常见，但这是标准集。

这个函数使用 string 类的 find_first_not_of() 和 find_last_not_of() 方法来查找不是集合成员的第一个/最后一个元素。

- 现在，可以使用该函数来消除所有未经请求的空白:

```
cout << format("[{}]\n", trimstr(s));
```

输出为:

```
[ten-thumbed input]
```

How it works…

string 类的各种 find…() 成员函数返回一个位置作为 size_t 值:

```
size_t find_first_not_of( const CharT* s, size_type pos = 0 );
size_t find_last_not_of( const CharT* s, size_type pos = 0 );
```

返回值是第一个匹配字符的从零开始的位置 (不在 s 字符列表中)，若没有找到，则返回特殊值 string::npos。npos 是一个表示无效位置的静态成员常量。

我们测试 (first == string::npos)，如果不匹配则返回空字符串 {}。否则，使用 s.substr() 方法，通过确认第一个和最后一个字符的位置，来返回不带空格的字符串。

7.8. 从用户输入中读取字符串

STL 使用 std::cin 对象从标准输入流提供基于字符的输入。cin 对象是一个全局单例对象，可从控制台读取输入作为 istream 输入流。

默认情况下，cin 每次读取一个单词，直到流的末尾:

```
string word{};
cout << "Enter words: ";
while(cin >> word) {
  cout << format("[{}] ", word);
}
cout << '\n';
```

输出为:

```
$ ./working
Enter words: big light in sky
[big] [light] [in] [sky]
```

这可能会让一些人认为 cin 功能很弱。

虽然 cin 确实有它的怪癖，但其更容易用来提供面向行的输入。

How to do it…

要从 cin 获得基本的面向行的功能，需要理解两个重要的行为。一种是一次获得一行字的能力，而不是一次获得一个字。另一个是在出现错误条件后重置流的能力。让我们来了解一下细节:

- 首先，需要提示用户进行输入。下面是一个简单的提示函数:

```
bool prompt(const string_view s, const string_view s2 = "") {
  if(s2.size()) cout << format("{} ({}): ", s, s2);
  else cout << format("{}: ", s);
  cout.flush();
  return true;
}
```

调用 count.flush() 确保立即显示输出。有时，输出不包括换行符时，输出流可能不会自动刷新。

- cin 类有一个 getline() 方法，从输入流中获取一行文本，并将其放入 C-string 数组中:

```
constexpr size_t MAXLINE{1024 * 10};
char s[MAXLINE]{};
const char * p1{ "Words here" };
prompt(p1);
cin.getline(s, MAXLINE, '\n');
cout << s << '\n';
```

输出为:

```
Words here: big light in sky
big light in sky
```

cin.getline() 方法有三个参数:

```
getline(char* s, size_t count, char delim );
```

第一个参数是目标的 C-string 数组。第二个是数组的大小。第三个是行尾的分隔符。

该函数在数组中放置的字符不超过 count-1，为空结束符留出空间。

分隔符默认为换行符'\n' 字符。

- STL 还提供了一个独立的 getline() 函数，用于 STL 字符串对象:

```
string line{};
const char * p1a{ "More words here" };
prompt(p1a, "p1a");
getline(cin, line, '\n');
cout << line << '\n';
```

输出为:

```
$ ./working
More words here (p1a): slated to appear in east
slated to appear in east
```

std::getline() 函数有三个参数:

```
getline(basic_istream&& in, string& str, char delim );
```

第一个参数是输出流，第二个参数是对字符串对象的引用，第三个参数是行结束分隔符。若未指定，分隔符默认为换行符'\n' 字符。

我感觉 getline() 比 cin.getline() 方法更方便。

- 可以使用 cin 从输入流中获取特定的类型。要做到这一点，必须能够处理错误条件。当 cin 遇到错误时，其将流设置为错误条件并停止接受输入。要在错误后重试输入，必须重置流的状态。下面是一个在错误后重置输入流的函数:

```
void clearistream() {
  string s{};
  cin.clear();
  getline(cin, s);
}
```

clear() 函数的作用是: 重置输入流中的错误标志，但将文本保留在缓冲区中。然后，通过读取一行并丢弃，来清除缓冲区。

- 可以通过对数值类型变量使用 cin 来接受数值输入:

```
double a{};
double b{};
const char * p2{ "Please enter two numbers" };
for(prompt(p2); !(cin >> a >> b); prompt(p2)) {
  cout << "not numeric\n";
  clearistream();
}
cout << format("You entered {} and {}\n", a, b);
```

输出为:

```
$ ./working
Please enter two numbers: a b
not numeric
Please enter two numbers: 47 73
You entered 47 and 73
```

cin >> a >> b 表达式接受来自控制台的输入，并尝试将前两个单词转换为与 a 和 b 兼容的类型 (double)。若失败了，则再次使用 clearistream()。

- 可以使用 getline() 分隔符参数来获取逗号分隔的输入:

```
line.clear();
prompt(p3);
while(line.empty()) getline(cin, line);
stringstream ss(line);
while(getline(ss, word, ',')) {
  if(word.empty()) continue;
  cout << format("word: [{}]\n", trimstr(word));
}
```

输出为:

```
$ ./working
Comma-separated words: this, that, other
word: [this]
word: [that]
word: [other]
```

因为这段代码是在数字代码之后运行的，并且因为 cin 是混乱的，所以缓冲区中可能仍然有结束的行。while(line.empty()) 循环将有选择地获取空行。

使用一个 stringstream 对象来处理单词，所以不必使用 cin。可以使用 getline() 来获取一行，而无需等待文件结束状态。

然后，在 stringstream 对象上调用 getline() 来解析用逗号分隔的单词。这给了我们带有前导空格的单词。我们使用本章中的 trimstr() 函数来清理空白字符。

How it works…

std::cin 对象比它看起来更有用，但使用它可能是一个挑战。其倾向于在流上留下行结束符，并且在出现错误的情况下，可能会忽略输入。

解决方案是使用 getline()，并在必要时将行放入 stringstream 中，以便于解析。

7.9. 统计文件中的单词数

默认情况下，basic_istream 类每次读取一个单词。可以利用这个属性使用 istream_iterator 来计算单词数。

How to do it…

这是一个使用 istream_iterator 来计数单词的简单方法:

- 使用 istream_iterator 对象来计数单词:

```
size_t wordcount(auto& is) {
  using it_t = istream_iterator<string>;
  return distance(it_t{is}, it_t{});
```

```
}
```

distance() 函数接受两个迭代器，并返回它们之间的距离。using 语句为 istream_iterator 类创建了一个带有字符串特化的别名 it_t。然后，用一个迭代器调用 distance()，迭代器用输入流 it_t{is} 初始化，另一个用默认构造函数调用 distance()，作为流的结束哨点。

- 在 main() 中使用 wordcount():

```
int main() {
  const char * fn{ "the-raven.txt" };
  std::ifstream infile{fn, std::ios_base::in};
  size_t wc{ wordcount(infile) };
  cout << format("There are {} words in the
    file.\n", wc);
}
```

这将使用 fstream 对象调用 wordcount() 并打印文件中的字数。当用其处理埃德加·爱伦·坡的《乌鸦》时，可得到这样的输出:

```
There are 1068 words in the file.
```

How it works…

因为 basic_istream 默认为逐字输入，文件中的步数将是字数。distance() 函数将测量两个迭代器之间的距离，因此使用兼容对象的起始和末尾迭代器来计算文件中的字数。

7.10. 使用文件输入初始化复杂结构体

输入流的优点是能够解析文本文件中不同类型的数据，并将它们转换为相应的基本类型。下面是一个使用输入流将数据导入结构容器的简单技术。

How to do it…

这个示例中，我们将获取一个数据文件，并将其不同的字段导入到 struct 对象的 vector 中。数据文件表示城市，及其人口和地图坐标:

- 这是 cities.txt，是要读取的数据文件:

```
Las Vegas
661903 36.1699 -115.1398
New York City
8850000 40.7128 -74.0060
Berlin
3571000 52.5200 13.4050
Mexico City
21900000 19.4326 -99.1332
Sydney
5312000 -33.8688 151.2093
```

城市名称单独在一行上。第二行是人口，然后是经度和纬度。这一模式在五个城市中都重复出现。

- 我们将在一个常量中定义文件名，以便稍后可以打开它:

```
constexpr const char * fn{ "cities.txt" };
```

- 下面是一个保存数据的 City 结构体:

```
struct City {
  string name;
  unsigned long population;
  double latitude;
  double longitude;
};
```

- 我们希望读取文件并填充 City 对象的 vector:

```
vector<City> cities;
```

- 这里是输入流使这变得容易的地方。可以简单地为 City 类特化操作符 >>，如下所示:

```
std::istream& operator>>(std::istream& in, City& c) {
  in >> std::ws;
  std::getline(in, c.name);
  in >> c.population >> c.latitude >> c.longitude;
  return in;
}
```

std::ws 输入操纵符将丢弃输入流中前面的空格。

我们使用 getline() 来读取城市名称，因为可能由多个单词组成。

这利用填充 (unsigned long) 的 >> 操作符，以及纬度和经度 (都是 double) 元素来填充正确的类型。

- 现在，可以打开文件并使用 >> 操作符将文件直接读入 City 对象的 vector:

```
ifstream infile(fn, std::ios_base::in);
if(!infile.is_open()) {
```

```
    cout << format("failed to open file {}\n", fn);
    return 1;
}
for(City c{}; infile >> c;) cities.emplace_back(c);
```

- 可以使用 format() 来显示 vector:

```
for (const auto& [name, pop, lat, lon] : cities) {
    cout << format("{:.<15} pop {:<10} coords {}, {}\n",
    name, make_commas(pop), lat, lon);
}
```

输出为:

```
$ ./initialize_container < cities.txt
Las Vegas..... pop 661,903 coords 36.1699, -115.1398
New York City.. pop 8,850,000 coords 40.7128, -74.006
Berlin........ pop 3,571,000 coords 52.52, 13.405
Mexico City.... pop 21,900,000 coords 19.4326, -99.1332
Sydney........ pop 5,312,000 coords -33.8688, 151.2093
```

- make_comma() 函数也在第 2 章中使用，接受一个数值并返回一个字符串对象。为了可读性，添加了逗号:

```
string make_commas(const unsigned long num) {
    string s{ std::to_string(num) };
    for(int l = s.length() - 3; l > 0; l -= 3) {
        s.insert(l, ",");
    }
    return s;
}
```

How it works…

这个示例的核心是 istream 类操作符 >> 重载:

```
std::istream& operator>>(std::istream& in, City& c) {
    in >> std::ws;
    std::getline(in, c.name);
    in >> c.population >> c.latitude >> c.longitude;
    return in;
}
```

通过在函数头中指定我们的 City 类，每当一个 City 对象出现在输入流 >> 操作符的右侧时，将使用这个函数:

```
City c{};
infile >> c;
```

这允许我们精确地指定输入流如何将数据读入 City 对象。

There's more…

在 Windows 系统上运行这段代码时，会注意到第一行的第一个单词损坏了。这是因为 Windows 总是在任何 UTF-8 文件的开头包含一个字节顺序标记 (BOM)。因此，当在 Windows 上读取文件时，BOM 将包含在读取的第一个对象中。BOM 是不合时宜的，但在撰写本文时，没有办法阻止 Windows 使用它。

解决方案是调用一个函数来检查文件的前三个字节的 BOM。UTF-8 的 BOM 是 EF BB BF。下面是一个搜索并跳过 UTF-8 BOM 的函数:

```
// skip BOM for UTF-8 on Windows
void skip_bom(auto& fs) {
  const unsigned char boms[]{ 0xef, 0xbb, 0xbf };
  bool have_bom{ true };
  for(const auto& c : boms) {
    if((unsigned char)fs.get() != c) have_bom = false;
  }
  if(!have_bom) fs.seekg(0);
  return;
}
```

这将读取文件的前三个字节，并检查是否为 UTF-8 BOM 签名。若这三个字节中的一个不匹配，会将输入流重置到文件的开头。若文件没有 BOM，则不有任何问题。

只需在开始读取文件之前调用这个函数:

```
int main() {
  ...
  ifstream infile(fn, std::ios_base::in);
  if(!infile.is_open()) {
    cout << format("failed to open file {}\n", fn);
    return 1;
  }
  skip_bom(infile);
  for(City c{}; infile >> c;) cities.emplace_back(c);
  ...
}
```

这将确保 BOM 不包含在文件的第一个字符串中。

> **Note**
>
> 因为 cin 输入流不可定位，所以 skip_bom() 函数将不能在 cin 流上工作，只适用于可搜索的文本文件。

7.11. 使用 char_traits 定义一个字符串类

string 类是 basic_string 类的别名，签名为:

```
class basic_string<char, std::char_traits<char>>;
```

第一个模板参数提供字符类型。第二个模板形参提供一个字符特征类，为指定的字符类型提供基本的字符和字符串操作。我们通常使用的是 char_traits<char> 类。

我们可以通过提供自己的自定义字符特征类，来修改字符串的行为。

How to do it…

在这个示例中，将创建一个字符特征类，用于 basic_string，其可以进行忽略大小的比较:

- 首先，需要一个函数将字符转换为通用大小写。这里使用小写字母，大写的也可以，这是一个随意的选择:

```
constexpr char char_lower(const char& c) {
  if(c >= 'A' && c <= 'Z') return c + ('a' - 'A');
  else return c;
}
```

这个函数必须是 constexpr(对于 C++20 及更高版本)，所以现有的 std::tolower() 函数在这里不能工作。

- 我们的特征类称为 ci_traits(ci 代表大小写无关)，继承自 std::char_traits<char>:

```
class ci_traits : public std::char_traits<char> {
  public:
  ...
};
```

继承允许重载需要的函数。

- 比较函数在小于时称为 lt()，在等于时称为 eq():

```
static constexpr bool lt(char_type a, char_type b)
noexcept {
  return char_lower(a) < char_lower(b);
}
static constexpr bool eq(char_type a, char_type b)
noexcept {
  return char_lower(a) == char_lower(b);
}
```

注意，这里比较了字符的小写版本。

- 还有一个 compare() 函数，用于比较两个 c 字符串。大于则返回 +1，小于则返回-1，等于则返回 0。还可以使用三向比较操作符 <=>:

```
static constexpr int compare(const char_type* s1,
const char_type* s2, size_t count) {
  for(size_t i{0}; i < count; ++i) {
    auto diff{ char_lower(s1[i]) <=>
      char_lower(s2[i]) };
    if(diff > 0) return 1;
    if(diff < 0) return -1;
```

```
  }
  return 0;
}
```

- 最后，需要实现一个 find() 函数。这将返回一个指向已找到字符的第一个实例的指针，若未找到则返回 nullptr:

```
static constexpr const char_type* find(const char_type*
p, size_t count, const char_type& ch) {
  const char_type find_c{ char_lower(ch) };
  for(size_t i{0}; i < count; ++i) {
    if(find_c == char_lower(p[i])) return p + i;
  }
  return nullptr;
}
```

- 现在有了一个 ci_traits 类，可以为我们的字符串类定义一个别名:

```
using ci_string = std::basic_string<char, ci_traits>;
```

- main() 函数中，定义了一个字符串和一个 ci_string:

```
int main() {
  string s{"Foo Bar Baz"};
  ci_string ci_s{"Foo Bar Baz"};
  ...
```

- 想用 cout 打印它们，但现在还不行:

```
cout << "string: " << s << '\n';
cout << "ci_string: " << ci_s << '\n';
```

首先，需要重载操作符 <<:

```
std::ostream& operator<<(std::ostream& os,
const ci_string& str) {
  return os << str.c_str();
}
```

现在，会得到这样的输出:

```
string: Foo Bar Baz
ci_string: Foo Bar Baz
```

- 比较两个具有不同情况的 ci_string 对象:

```
ci_string compare1{"CoMpArE StRiNg"};
ci_string compare2{"compare string"};
if (compare1 == compare2) {
  cout << format("Match! {} == {}\n", compare1,
  compare2);
```

```
} else {
  cout << format("no match {} != {}\n", compare1,
  compare2);
}
```

输出为:

```
Match! CoMpArE StRiNg == compare string
```

比较和预期的一样。

- 在 ci_s 对象上使用 find() 函数，搜索小写的 b 并找到大写的 B:

```
size_t found = ci_s.find('b');
cout << format("found: pos {} char {}\n", found,
ci_s[found]);
```

输出为:

```
found: pos 4 char B
```

Note

注意 format() 函数不需要特化，这是用 fmt.dev 参考实现测试的，不能与 MSVC 的 format() 预览版一起工作，即使是特化也不行。希望在未来的版本中可以修复这个问题。

How it works…

这个配方的工作方式是将字符串类的模板特化中的 std::char_traits 类替换为我们的 ci_traits 类。basic_string 类使用特征类，来实现其基本的特定于字符的函数，比如：比较和搜索。当用我们自己的类来代替它时，可以改变这些基本行为。

There's more…

也可以重写 assign() 和 copy() 成员函数来创建一个存储小写字符的类:

```
class lc_traits : public std::char_traits<char> {
  public:
  static constexpr void assign( char_type& r, const
  char_type& a )
  noexcept {
    r = char_lower(a);
  }
  static constexpr char_type* assign( char_type* p,
  std::size_t count, char_type a ) {
    for(size_t i{}; i < count; ++i) p[i] =
    char_lower(a);
```

```
    return p;
  }
  static constexpr char_type* copy(char_type* dest,
    const char_type* src, size_t count) {
    for(size_t i{0}; i < count; ++i) {
      dest[i] = char_lower(src[i]);
    }
    return dest;
  }
};
```

现在，可以创建一个 lc_string 别名，对象存储小写字符:

```
using lc_string = std::basic_string<char, lc_traits>;
...
lc_string lc_s{"Foo Bar Baz"};
cout << "lc_string: " << lc_s << '\n';
```

输出为:

```
lc_string: foo bar baz
```

> **Note**
>
> 这些方法在 GCC 和 Clang 上可以正常工作，但在 MSVC 预览版上则不行。我希望在未来的版本中可以修复这个问题。

7.12. 用正则表达式解析字符串

正则表达式 (通常缩写为 regex) 通常用于文本流的词法分析和模式匹配。它们在 Unix 文本处理实用程序 (如 grep、awk 和 sed) 中很常见，并且是 Perl 语言不可分割的一部分。POSIX 标准于 1992 年获得批准，而其他常见的变体包括 Perl 和 ECMAScript (JavaScript) 方言。C++ regex 库默认使用 ECMAScript 方言。

regex 库是在 C++11 中首次引入 STL 的，其对于在文本文件中查找模式非常有用。

要了解更多关于正则表达式的语法和用法，我推荐 Jeffrey Friedl 的《Mastering Regular Expressions》这本书。

How to do it…

对于这个示例，我们将从 HTML 文件中提取超链接。超链接是这样用 HTML 编码的:

```
<a href="http://example.com/file.html">Text goes here</a>
```

我们将使用一个 regex 对象来提取链接和文本，作为两个单独的字符串。

- 我们的示例文件名为 the-end.html。它来自我的网站 (https://bw.org/end/)，并包含在 GitHub 存储库中:

```
const char *fn{ "the-end.html" };
```

- 现在，用正则表达式字符串定义 regex 对象:

```
const std::regex
  link_re{ "<a href=\"([^\"]*)\"[^<]*>([^<]*)</a>" };
```

正则表达式一开始看起来很吓人，但实际上它们很简单。

解析如下:

I. 匹配整个字符串。
II. 查找子字符串 <a href="
III. 存储到下一个双引号间的所有内容，作为子匹配 1。
IV. 跳过 > 字符。
V. 将字符串 之前的所有内容存储为子匹配 2。

- 现在，将文件放入一个字符串:

```
string in{};
std::ifstream infile(fn, std::ios_base::in);
for(string line{}; getline(infile, line);) in += line;
```

这将打开 HTML 文件，逐行读取它，并将每行追加到字符串对象中。

- 为了提取链接字符串，设置了 sregex_token_iterator 对象，来逐级遍历文件并提取每个匹配的元素:

```
std::sregex_token_iterator it{ in.begin(), in.end(),
  link_re, {1, 2} };
```

1 和 2 对应正则表达式中的子匹配项。

- 我们有一个对应的函数，用迭代器逐级遍历结果:

```
template<typename It>
void get_links(It it) {
  for(It end_it{}; it != end_it; ) {
    const string link{ *it++ };
    if(it == end_it) break;
    const string desc{ *it++ };
    cout << format("{:.<24} {}\n", desc, link);
  }
}
```

使用 regex 迭代器调用函数:

```
get_links(it);
```

用描述和链接得到了这个结果:

```
Bill Weinman.............. https://bw.org/
courses................... https://bw.org/courses/
music..................... https://bw.org/music/
books..................... https://packt.com/
back to the internet.... https://duckduckgo.com/
```

How it works…

STL 正则表达式引擎作为生成器运行，每次计算并产生一个结果。我们使用 sregex_iterator 或 sregex_token_iterator 来设置迭代器，sregex_token_iterator 支持子匹配，sregex_iterator 不支持子匹配。

正则表达式中的括号作为子匹配项，分别编号为 1 和 2:

```
const regex link_re{ "<a href=\"([^\"]*)\"[^<]*>([^<]*)</a>" };
```

正则表达式匹配的每个部分如下图所示:

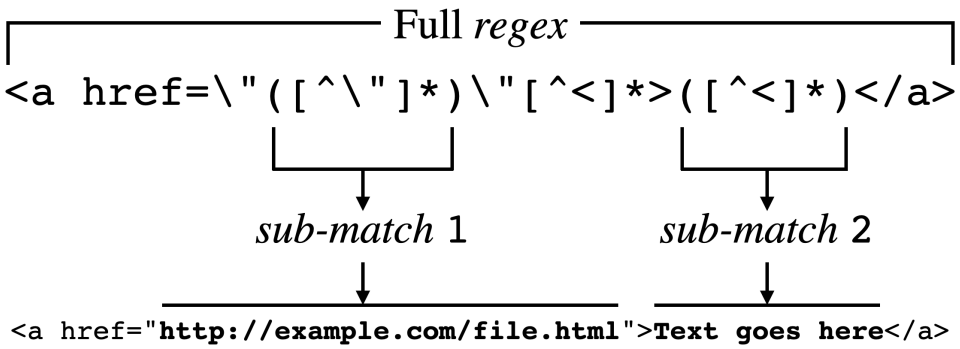


图 7.1 带有子匹配项的正则表达式

这允许我们匹配一个字符串，并使用该字符串的部分作为的结果:

```
sregex_token_iterator it{ in.begin(), in.end(), link_re, {1, 2}
};
```

子匹配项从 1 开始编号。子匹配 0 是一个特殊值，表示整个匹配。

要支持迭代器，就可以像这样使用:

```
for(It end_it{}; it != end_it; ) {
  const string link{ *it++ };
  if(it == end_it) break;
  const string desc{ *it++ };
  cout << format("{:.<24} {}\n", desc, link);
}
```

这只是通过 regex 迭代器逐步遍历结果，格式化的输出如下所示:

```
Bill Weinman..... https://bw.org/
courses.......... https://bw.org/courses/
music............ https://bw.org/music/
books............ https://packt.com/
back to the internet.... https://duckduckgo.com/
```

第 8 章　实用工具类

C++ 标准库包括为特定任务设计的各种工具类。有些是常见的，读者们可能在这本书的其他示例中见过很多这样的类。

本章在以下主题中介绍了一些通用的工具，包括时间测量、泛型类型、智能指针等:

- std::optional 管理可选值
- std::any 保证类型安全
- std::variant 存储不同的类型
- std::chrono 的时间事件
- 对可变元组使用折叠表达式
- std::unique_ptr 管理已分配的内存
- std::shared_ptr 的共享对象
- 对共享对象使用弱指针
- 共享管理对象的成员
- 比较随机数引擎
- 比较随机数分布发生器

8.1. 相关准备

本章所有的例子和源代码都可以在本书的 GitHub 库中找到，网址是https://github.com/PacktPublishing/CPP-20-STL-Cookbook/tree/main/chap08。

8.2. std::optional 管理可选值

C++17 引入的 std::optional 类保存了一个可选值。

假设有一个函数，它可能返回值，也可能不返回值——例如，一个函数检查一个数字是否是质数，但若存在第一个因数，则返回第一个因数。此函数应返回值或 bool 状态。可以创建一个同时包含值和状态的结构体:

```
struct factor_t {
  bool is_prime;
  long factor;
};
factor_t factor(long n) {
  factor_t r{};
  for(long i = 2; i <= n / 2; ++i) {
    if (n % i == 0) {
      r.is_prime = false;
      r.factor = i;
      return r;
    }
  }
  r.is_prime = true;
```

```
  return r;
}
```

这是一个笨拙的解决方案，但它是有效的，而且很常见。

optional 类可以让这个任务变得简单:

```
optional<long> factor(long n) {
  for (long i = 2; i <= n / 2; ++i) {
    if (n % i == 0) return {i};
  }
  return {};
}
```

使用 optional，可以返回值或非值。

我们可以这样使用:

```
long a{ 42 };
long b{ 73 };
auto x = factor(a);
auto y = factor(b);
if(x) cout << format("lowest factor of {} is {}\n", a, *x);
else cout << format("{} is prime\n", a);
if(y) cout << format("lowest factor of {} is {}\n", b, *y);
else cout << format("{} is prime\n", b);
```

输出为:

```
lowest factor of 42 is 2
73 is prime
```

How to do it…

这个示例中，来看看如何使用 optional 类的例子:

- optional 类非常简单。使用标准模板表示法构造一个 optional:

```
optional<int> a{ 42 };
cout << *a << '\n';
```

使用指针的解引用操作符访问可选对象的值。

输出为:

```
42
```

- 使用 bool 操作符来测试 optional 是否有值:

```
if(a) cout << *a << '\n';
else cout << "no value\n";
```

若构造 a 时不带值:

```
optional<int> a{};
```

输出将输出 else 的内容:

```
no value
```

- 我们可以通过声明类型别名来进一步简化:

```
using oint = std::optional<int>;
oint a{ 42 };
oint b{ 73 };
```

- 若想对 oint 对象进行操作，可以对操作符进行重载:

```
oint operator+(const oint& a, const oint& b) {
  if(a && b) return *a + *b;
  else return {};
}
oint operator+(const oint& a, const int b) {
  if(a) return *a + b;
  else return {};
}
```

现在，可以直接操作 oint 对象:

```
auto sum{ a + b };
if(sum) {
  cout << format("{} + {} = {}\n", *a, *b, *sum);
} else {
  cout << "NAN\n";
}
```

输出为:

```
42 + 73 = 115
```

- 假设用默认构造函数声明 b:

```
oint b{};
```

现在，可得到 else 分支的输出:

```
NAN
```

How it works…

std::optional 类是为了简单而创建的，其为许多常用函数提供操作符重载。还包括成员函数，可提供更进一步的灵活性。

optional 类提供了一个操作符 bool 重载，用于确定对象是否有值:

```
optional<int> n{ 42 };
if(n) ... // has a value
```

或者，可以使用 has_value() 成员函数:

```
if(n.has_value()) ... // has a value
```

要访问该值，可以使用解引用操作符的重载:

```
x = *n; // * retruns the value
```

或者，可以使用 value() 成员函数:

```
x = n.value(); // * retruns the value
```

reset() 成员函数可以销毁该值，并重置可选对象的状态:

```
n.reset(); // no longer has a value
```

There's more…

optional 类通过 value() 方法提供异常支持:

```
b.reset();
try {
  cout << b.value() << '\n';
} catch(const std::bad_optional_access& e) {
  cout << format("b.value(): {}\n", e.what());
}
```

输出为:

```
b.value(): bad optional access
```

Note

只有 value() 方法会抛出异常。对于无效值，解引用操作符的行为未定义。

8.3. std::any 保证类型安全

C++17 中引入了 std::any 类，可为任何类型的单个对象提供了类型安全的容器。

例如，这是一个默认构造的 any 对象:

```
any x{};
```

该对象无值，可以使用 has_value() 方法进行测试:

```
if(x.has_value()) cout << "have value\n";
else cout << "no value\n";
```

输出为:

```
no value
```

我们使用赋值操作符为 any 对象赋值:

```
x = 42;
```

现在，any 对象有一个值和一个类型:

```
if(x.has_value()) {
  cout << format("x has type: {}\n", x.type().name());
  cout << format("x has value: {}\n", any_cast<int>(x));
} else {
  cout << "no value\n";
}
```

输出为:

```
x has type: i
x has value: 42
```

type() 方法返回一个 type_info 对象，type_info::name() 方法返回 C-string 类型的实现定义的名称。在本例中，GCC 中 i 表示 int。

使用 any_cast<type>() 非成员函数强制转换值进行使用。

可以用不同类型的不同值重新赋值:

```
x = "abc"s;
cout << format("x is type {} with value {}\n",
  x.type().name(), any_cast<string>(x))
```

输出为:

```
x is type NSt7__cxx1112basic_string... with value abc
```

用 GCC 缩写了这个长类型名，相同的对象，曾经持有 int，现在包含一个 STL 字符串对象。

any 类的主要是创建一个多态函数，来看看这个示例。

How to do it…

这个示例中，将使用 any 类构建一个多态函数，可以在其参数中接受不同类型对象的函数:

- 多态函数可以接受任意对象，并打印其类型和值:

```
void p_any(const any& a) {
  if (!a.has_value()) {
    cout << "None.\n";
```

```
  } else if (a.type() == typeid(int)) {
    cout << format("int: {}\n", any_cast<int>(a));
  } else if (a.type() == typeid(string)) {
    cout << format("string: \"{}\"\n",
    any_cast<const string&>(a));
  } else if (a.type() == typeid(list<int>)) {
    cout << "list<int>: ";
    for(auto& i : any_cast<const list<int>&>(a))
      cout << format("{} ", i);
    cout << '\n';
  } else {
    cout << format("something else: {}\n",
    a.type().name());
  }
}
```

p_any() 函数首先测试对象是否有值，针对各种类型测试 type() 方法，并对每种类型采取适当的操作。

在 any 类之前，必须为这个函数编写四种不同的特化，并且不能简单地处理默认的情况。

- 在 main() 使用这个函数:

```
p_any({});
p_any(47);
p_any("abc"s);
p_any(any(list{ 1, 2, 3 }));
p_any(any(vector{ 1, 2, 3 }));
```

输出为:

```
one.
int: 47
string: "abc"
list<int>: 1 2 3
something else: St6vectorIiSaIiEE
```

我们的多态函数用最少的代码处理了各种类型的变量。

How it works…

any 复制构造函数和赋值操作符，可以直接初始化来生成目标对象的非 const 副本作为包含对象，包含的对象类型可以作为 typeid 单独存储。

初始化后，any 对象有以下方法:

- emplace() 替换包含的对象，在适当的位置构造新对象。
- reset() 替换包含的对象，在适当的位置构造新对象。
- has_value() 若有包含对象，则返回 true。

- type() 返回 typeid 对象，表示所包含对象的类型。
- operator=() 用复制或移动操作替换包含的对象。

any 类还支持以下非成员函数:

- any_cast<T>()，模板函数，提供对包含对象的类型安全访问。
  any_cast<T>() 函数返回包含对象的副本，可以使用 any_cast<T&>() 来返回引用。
- std::swap() 专门处理 std::swap 算法。

若用错误的类型强制转换 any 对象，则会抛出一个 bad_any_cast 异常:

```
try {
  cout << any_cast<int>(x) << '\n';
} catch(std::bad_any_cast& e) {
  cout << format("any: {}\n", e.what());
}
```

输出为:

```
any: bad any_cast
```

8.4. std::variant 存储不同的类型

C++17 中引入的 std::variant 类可以保存不同的值，每次一个，其中每个值必须适合相同的分配内存空间。对于保存可供在单个上下文中使用的替代类型非常有用。

与 union 的区别

variant 类是一个带标记的联合，其与原始 union 结构的不同之处在于，一次只能有一种类型起作用。

原始 union 类型继承自 C，是一种结构，在这种结构中，相同的数据可以作为不同的类型访问。例如:

```
union ipv4 {
  struct {
    uint8_t a; uint8_t b; uint8_t c; uint8_t d;
  }quad;
  uint32_t int32;
} addr;
addr.int32 = 0x2A05A8C0;
cout << format("ip addr dotted quad: {}.{}.{}.{}\n",
  addr.quad.a, addr.quad.b, addr.quad.c, addr.quad.d);
cout << format("ip addr int32 (LE): {:08X}\n", addr.int32);
```

输出为:

```
ip addr dotted quad: 192.168.5.42
ip addr int32 (LE): 2A05A8C0
```

这个例子中，union 有两个成员，类型是 struct 和 uint32_t，其中 struct 有四个 uint8_t 成员。这为我们提供了相同 32 位内存空间的两种不同视角，可以将相同的 ipv4 地址视为一个 32 位无符号整数 (Little Endian 或 LE) 或四个 8 位无符号整数。这提供了在系统级别的位多态。

variant 不是这样的。variant 类是一个带标记的 union，其中每个数据都带有其类型的标记。若将一个值存储为 uint32_t，只能以 uint32_t 的形式访问。这使得 variant 类型安全，但并不是 union 的替代品。

How to do it…

示例中，将展示 std::variant 的使用方式。

- 我们将从一个简单的类开始，先来保存一个 Animal:

```
class Animal {
  string_view _name{};
  string_view _sound{};
  Animal();
public:
  Animal(string_view n, string_view s)
  : _name{ n }, _sound{ s } {}
  void speak() const {
    cout << format("{} says {}\n", _name, _sound);
  }
  void sound(string_view s) {
    _sound = s;
  }
}
```

动物的名字和动物发出的声音被传递到构造函数中。

- 不同的物种类继承自 Animal:

```
class Cat : public Animal {
  public:
  Cat(string_view n) : Animal(n, "meow") {}
};
class Dog : public Animal {
  public:
  Dog(string_view n) : Animal(n, "arf!") {}
};
class Wookie : public Animal {
  public:
  Wookie(string_view n) : Animal(n, "grrraarrgghh!") {}
};
```

这些类中的每一个都通过调用父构造函数，来为其物种类设置声音。

- 现在，可以在别名中定义变量类型:

```
using v_animal = std::variant<Cat, Dog, Wookie>;
```

这个变体可以容纳任何类型，猫，狗或伍基。

- 在 main() 中，使用 v_animal 别名作为类型创建了一个列表:

```
int main() {
  list<v_animal> pets{
    Cat{"Hobbes"}, Dog{"Fido"}, Cat{"Max"},
    Wookie{"Chewie"}
  };
  ...
```

列表中的每个元素都是变量定义中包含的类型。

- variant 类提供了几种访问元素的不同方法，先来看看 visit()。

visit() 调用带有当前包含在变体中的对象的函子。首先，定义一个接受宠物的函子:

```
struct animal_speaks {
  void operator()(const Dog& d) const { d.speak(); }
  void operator()(const Cat& c) const { c.speak(); }
  void operator()(const Wookie& w) const {
    w.speak(); }
};
```

这是一个简单的函子类，每个 Animal 子类都有重载。使用 visit() 调用它，并使用列表的每个元素:

```
for (const v_animal& a : pets) {
  visit(animal_speaks{}, a);
}
```

会得到这样的输出:

```
Hobbes says meow
Fido says arf!
Max says meow
Chewie says grrraarrgghh!
```

- variant 类还提供了一个 index() 方法:

```
for(const v_animal &a : pets) {
  auto idx{ a.index() };
  if(idx == 0) get<Cat>(a).speak();
  if(idx == 1) get<Dog>(a).speak();
  if(idx == 2) get<Wookie>(a).speak();
}
```

可以得到这样的输出:

```
Hobbes says meow
Fido says arf!
Max says meow
Chewie says grrraarrgghh!
```

每个变量对象都根据模板参数中声明类型的顺序建立索引。av_animal 类型定义为 std::variant<Cat, Dog, Wookie>，这些类型的索引顺序为 0 - 2。

- get_if<T>() 函数的作用是: 根据类型测试给定元素:

```
for (const v_animal& a : pets) {
  if(const auto c{ get_if<Cat>(&a) }; c) {
    c->speak();
  } else if(const auto d{ get_if<Dog>(&a) }; d) {
    d->speak();
  } else if(const auto w{ get_if<Wookie>(&a) }; w) {
    w->speak();
  }
}
```

输出为:

```
Hobbes says meow
Fido says arf!
Max says meow
Chewie says grrraarrgghh!
```

get_if<T>() 函数的作用是: 若元素类型与 T 匹配，则返回一个指针; 否则，返回 nullptr。

- 最后，hold_alternative <T>() 函数返回 true 或 false。可以使用 this 来测试一个类型是否对应一个元素，而不返回该元素:

```
size_t n_cats{}, n_dogs{}, n_wookies{};
for(const v_animal& a : pets) {
  if(holds_alternative<Cat>(a)) ++n_cats;
  if(holds_alternative<Dog>(a)) ++n_dogs;
  if(holds_alternative<Wookie>(a)) ++n_wookies;
}
cout << format("there are {} cat(s), "
       "{} dog(s), "
       "and {} wookie(s)\n",
       n_cats, n_dogs, n_wookies);
```

输出为:

```
there are 2 cat(s), 1 dog(s), and 1 wookie(s)
```

How it works…

std::variant 类是一个单对象容器。variant<X, Y, Z> 实例的类型，必须是 X, Y 或 Z 对象类型的其中之一，并可以同时保存当前对象的值和类型。

index() 方法告诉我们当前对象的类型:

```
if(v.index() == 0) // if variant is type X
```

holds_alternative<T>() 非成员函数如果 T 是当前对象的类型，则返回 true:

```
if(holds_alternative<X>(v)) // if current variant obj is type X
```

可以使用 get() 非成员函数检索当前对象:

```
auto o{ get<X>(v) }; // current variant obj must be type X
```

可以将类型和检索测试与 get_if() 非成员函数结合起来:

```
auto* p{ get_if<X>(v) }; // nullptr if current obj not type X
```

visit() 非成员函数使用一个可调用对象，将当前变量对象作为其唯一形参:

```
visit(f, v); // calls f(v) with current variant obj
```

visit() 函数是在不测试对象类型的情况下，检索对象的唯一方法。结合一个可以处理每种类型的函子，就非常灵活了:

```
struct animal_speaks {
  void operator()(const Dog& d) const { d.speak(); }
  void operator()(const Cat& c) const { c.speak(); }
  void operator()(const Wookie& v) const { v.speak(); }
};
main() {
  for (const v_animal& a : pets) {
    visit(animal_speaks{}, a);
  }
}
```

输出为:

```
Hobbes says meow
Fido says arf!
Max says meow
Chewie says grrraarrgghh!
```

8.5. std::chrono 的时间事件

std::chrono 库提供了用于测量、报告时间和间隔的工具。

这些类和函数中的许多都是在 C++11 中引入的，但在 C++20 中已经有了重大的变化和更新。在撰写本文时，许多更新还没有在我测试的系统上实现。

本示例使用 chrono 库，为事件进行计时。

How to do it…

system_clock 类用于报告当前日期和时间。steady_clock 和 high_resolution_clock 类用于计时事件。让我们来看看这些时钟间的区别:

- 因为这些名字可能又长又不方便，在整个示例中使用一些类型别名:

```
using std::chrono::system_clock;
using std::chrono::steady_clock;
using std::chrono::high_resolution_clock;
using std::chrono::duration;
using seconds = duration<double>;
using milliseconds = duration<double, std::milli>;
using microseconds = duration<double, std::micro>;
using fps24 = duration<unsigned long, std::ratio<1, 24>>;
```

duration 类表示两个时间点之间的间隔。这些别名便于使用不同的间隔。

- 可以使用 system_clock 类来获取当前时间和日期:

```
auto t = system_clock::now();
cout << format("system_clock::now is {:%F %T}\n", t);
```

system_clock::now() 函数的作用是: 返回一个 time_point 对象。<chrono> 库包含了 time_point 的 format() 特化，该特化使用 strftime() 格式说明符。

输出为:

```
system_clock::now is 2022-02-05 13:52:15
```

<iomanip> 头文件包括 put_time()，其工作方式类似于 ostream 的 strftime():

```
std::time_t now_t = system_clock::to_time_t(t);
cout << "system_clock::now is "
  << std::put_time(std::localtime(&now_t), "%F %T")
  << '\n';
```

put_time() 接受一个指向 C 风格 time_t* 值的指针。system_clock::to_time_t 转换 time_point 对象为 time_t。

这与 format() 示例的输出相同:

```
system_clock::now is 2022-02-05 13:52:15
```

- 也可以使用 system_clock 为事件计时。首先，需要进行计时。下面是一个计算质数的函数:

```
constexpr uint64_t MAX_PRIME{ 0x1FFFF }
uint64_t count_primes() {
  constexpr auto is_prime = [](const uint64_t n) {
    for(uint64_t i{ 2 }; i < n / 2; ++i) {
      if(n % i == 0) return false;
    }
    return true;
  };
  uint64_t count{ 0 };
  uint64_t start{ 2 };
  uint64_t end{ MAX_PRIME };
  for(uint64_t i{ start }; i <= end ; ++i) {
    if(is_prime(i)) ++count;
  }
  return count;
}
```

这个函数对 2 到 0x1FFFF(131,071) 之间的质数进行计数，在大多数现代系统上，这需要几秒钟的时间。

- 现在，编写一个计时器函数来计时 count_primes():

```
seconds timer(uint64_t(*f)()) {
  auto t1{ system_clock::now() };
  uint64_t count{ f() };
  auto t2{ system_clock::now() };
  seconds secs{ t2 - t1 };
  cout << format("there are {} primes in range\n",
    count);
  return secs;
}
```

这个函数接受一个函数 f，并返回 duration<double>。这里使用 system_clock::now() 来标记调用 f() 前后的时间。取两个时间之间的差值，并在 duration 对象中返回。

- 可以在 main() 中使用 timer():

```
int main() {
  auto secs{ timer(count_primes) };
  cout << format("time elapsed: {:.3f} seconds\n",
  secs.count());
  ...
```

这将把 count_primes() 函数传递给 timer()，并以秒为单位存储持续时间对象。

输出为:

```
there are 12252 primes in range
time elapsed: 3.573 seconds
```

duration 对象上的 count() 方法返回指定单位的持续时间——本例中为 double，表示持续时间的秒数。

这是在一个运行 Debian 和 GCC 的 VM 上运行的，具体的时间在不同的系统上会有所不同。

- system_clock 类设计用来提供当前挂钟时间。虽然其频率可能支持计时目的，但不保证单调。换句话说，它可能不能提供一致的计时间隔。

chrono 库在 steady_clock 中提供了一个更合适的时钟。它具有与 system_clock 相同的接口，但为计时目的提供了更可靠的计时间隔:

```
seconds timer(uint64_t(*f)()) {
  auto t1{ steady_clock::now() };
  uint64_t count{ f() };
  auto t2{ steady_clock::now() };
  seconds secs{ t2 - t1 };
  cout << format("there are {} primes in range\n",
    count);
  return secs;
}
```

steady_clock 设计用于提供可靠一致的单调，适用于计时事件。它使用一个相对的时间参考，所以不用挂钟时间。system_clock 从固定的时间点 (1970 年 1 月 1 日 00:00 UTC) 开始测量，而 steady_clock 使用相对时间。

另一个选项是 high_resolution_clock，在给定系统上提供最短的计时间隔期，但在不同的实现中实现不一致。其可能是 system_clock 或 steady_clock 的别名，可能是单调的，也可能不是。high_resolution_clock 不建议通用。

- timer() 函数返回 seconds，是 duration<double> 的别名:

```
using seconds = duration<double>;
```

duration 类接受一个可选的第二个模板形参，一个 std::ratio 类:

```
template<class Rep, class Period = std::ratio<1>>
class duration;
```

<chrono> 头文件为许多十进制比率提供了方便的类型，包括 milli 和 micro:

```
using milliseconds = duration<double, std::milli>;
using microseconds = duration<double, std::micro>;
```

若需要其他工具，可以自己准备:

```
using fps24 = duration<unsigned long, std::ratio<1, 24>>;
```

fps24 表示以标准的每秒 24 帧拍摄的电影帧数。比率是 1/24 秒。

这让我们可以轻松地在不同的持续时间范围之间进行转换:

```
cout << format("time elapsed: {:.3f} sec\n", secs.
count());
cout << format("time elapsed: {:.3f} ms\n",
  milliseconds(secs).count());
cout << format("time elapsed: {:.3e} µs\n",
```

```
  microseconds(secs).count());
cout << format("time elapsed: {} frames at 24 fps\n",
  floor<fps24>(secs).count());
```

输出为:

```
time elapsed: 3.573 sec
time elapsed: 3573.077 ms
time elapsed: 3.573e+06 µs
time elapsed: 85 frames at 24 fps
```

因为 fps24 别名使用 unsigned long，而不是 double，所以需要进行类型转换。floor 函数通过丢弃小数部分来实现这一点。Round() 和 ceil() 在此上下文中也可用。

- 方便起见，chrono 库提供了标准持续时间比的 format() 特化:

```
cout << format("time elapsed: {:.3}\n", secs);
cout << format("time elapsed: {:.3}\n",
milliseconds(secs));
cout << format("time elapsed: {:.3}\n",
microseconds(secs));
```

输出为:

```
time elapsed: 3.573s
time elapsed: 3573.077ms
time elapsed: 3573076.564µs
```

这些结果在不同的实现上有所不同。

How it works…

chrono 库有两个主要部分，时钟类和间隔类。

时钟类

时钟类包括：

- system_clock –提供挂钟时间。
- steady_clock –为时间测量提供有保证的单调计时间隔。
- high_resolution_clock –提供最短的可用计时间隔周期。某些系统上，其可能是 system_clock 或 steady_clock 的别名。

我们使用 system_clock 显示当前时间和日期，用 steady_clock 来测量间隔。

每个时钟类都有一个 now() 方法，该方法返回 time_point，表示时钟的当前值。now() 是一个静态成员函数，所以可以不实例化对象:

```
auto t1{ steady_clock::now() };
```

std::duration 类

duration 类用于保存时间间隔——即两个 time_point 对象之间的差值。它通常由 time_point 对象的减法 (-) 操作符构造。

```
duration<double> secs{ t2 - t1 };
```

time_point 减去操作符作为 duration 的构造函数:

```
template<class C, class D1, class D2>
constexpr duration<D1,D2>
operator-( const time_point<C,D1>& pt_lhs,
  const time_point<C,D2>& pt_rhs );
```

duration 类有用于类型表示的模板参数和一个 ratio 对象:

```
template<class Rep, class Period = std::ratio<1>>
class duration;
```

周期模板参数默认为 1:1，即秒。

标准库为 10 次方提供了从 atto(1/1,000,000,000,000,000,000) 到 exa(1,000,000,000,000,000,000/1) 的比率别名 (例如 micro 和 milli)。我们可以创建标准的持续时间，就像示例中那样:

```
using milliseconds = duration<double, std::milli>;
using microseconds = duration<double, std::micro>;
```

count() 方法给出了 Rep 类型中的持续时间:

```
constexpr Rep count() const;
```

这使得我们可以方便地访问持续时间，并用于显示或其他目的:

```
cout << format("duration: {}\n", secs.count());
```

8.6. 对可变元组使用折叠表达式

tuple 类本质上是一个更复杂、不方便的结构体。tuple 的接口很麻烦，类模板参数推导和结构化绑定可以使这个过程变得简单。

对于大多数应用程序，我倾向于在使用 tuple 之前使用 struct，但有一个例外:tuple 的一个真正优势是，可以在可变的上下文中与折叠表达式一起使用。

折叠表达式

折叠表达式是 C++17 的一个新特性，设计目的是为了更容易地展开可变参数包。在折叠表达式之前，展开参数包需要一个递归函数:

```
template<typename T>
void f(T final) {
  cout << final << '\n';
}
template<typename T, typename... Args>
```

```
void f(T first, Args... args) {
  cout << first;
  f(args...);
}
int main() {
  f("hello", ' ', 47, ' ', "world");
}
```

输出为:

```
hello 47 world
```

使用折叠表达式后，这就会简单许多了:

```
template<typename... Args>
void f(Args... args) {
  (cout << ... << args);
  cout << '\n';
}
```

输出为:

```
hello 47 world
```

折叠表达式有四种类型:

- 一元右折叠: (args op ...)
- 一元左折叠: (... op args)
- 二元右折叠: (args op ... op init)
- 二元左折叠: (init op ... op args)

上面例子中的表达式是一个二进制左折叠:

```
(cout << ... << args);
```

可扩展为:

```
cout << "hello" << ' ' << 47 << ' ' << "world";
```

折叠表达式对于很多目的来说都是非常方便的，来看看如何在 tuple 中使用。

How to do it…

这个示例中，我们将创建一个模板函数，该函数对具有不同数量和类型元素的 tuple 进行操作:

- 这示例的核心是一个函数，其接受一个未知大小和类型的 tuple，并使用 format() 打印每个元素:

```cpp
template<typename... T>
constexpr void print_t(const tuple<T...>& tup) {
  auto lpt =
  [&tup] <size_t... I>
  (std::index_sequence<I...>)
  constexpr {
    (..., ( cout <<
      format((I? ", {}" : "{}"),
        get<I>(tup))
    ));
    cout << '\n';
  };
  lpt(std::make_index_sequence<sizeof...(T)>());
}
```

这个函数的核心在 lambda 表达式中，使用 index_sequence 对象生成索引值的参数包。然后，使用折叠表达式对每个下标值调用 get<I>。模板化的 lambda 需要编译器支持 C++20。

可以使用一个单独的函数来代替 lambda，但我更喜欢将它保持在一个作用域中。

- 现在可以在 main() 中使用各种 tuple:

```cpp
int main() {
  tuple lables{ "ID", "Name", "Scale" };
  tuple employee{ 123456, "John Doe", 3.7 };
  tuple nums{ 1, 7, "forty-two", 47, 73L, -111.11 };

  print_t(lables);
  print_t(employee);
  print_t(nums);
}
```

输出为:

```
ID, Name, Scale
123456, John Doe, 3.7
1, 7, forty-two, 47, 73, -111.11
```

How it works…

使用 tuple 的挑战在于它的限制性接口。可以使用 std::tie()、结构化绑定或 std::get<> 函数检索元素。若不知道 tuple 中元素的数量和类型，这些技术都没啥用。

可以使用 index_sequence 类来解决这个限制。index_sequence 是一个 integer_sequence 的特化，它提供了一个 size_t 的参数包，可以用它来索引 tuple。我们使用 make_index_sequence 调用 lambda 函数在 lambda 中设置一个参数包:

```cpp
lpt(std::make_index_sequence<sizeof...(T)>());
```

模板化的 lambda 是用 get() 函数 size_t 索引的参数包构造的:

```
[&tup] <size_t... I> (std::index_sequence<I...>) constexpr {
  ...
};
```

get() 函数将索引值作为模板形参，这里可以使用一元左折叠表达式调用 get<I>():

```
(..., ( cout << format("{} ", std::get<I>(tup))));
```

折叠表达式接受函数参数包中的每个元素，并应用逗号操作符。逗号的右边有一个 format() 函数，输出元组的每个元素。

这使得推断 tuple 中元素的数量成为可能，这使得它可以在可变的上下文中使用。记住，与一般的模板函数一样，编译器将为每个 tuple 形参组合生成该函数的特化版本。

There's more…

也可以将这种技术用于其他任务。例如，这里有一个函数，它返回大小未知的元组中所有 int 值的和:

```
template<typename... T>
constexpr int sum_t(const tuple<T...>& tup) {
  int accum{};
  auto lpt =
  [&tup, &accum] <size_t... I>
    (std::index_sequence<I...>)
  constexpr {
    (..., (
      accum += get<I>(tup)
    ));
  };
  lpt(std::make_index_sequence<sizeof...(T)>());
  return accum;
}
```

可以用几个不同数量 int 值的元组对象调用这个函数:

```
tuple ti1{ 1, 2, 3, 4, 5 };
tuple ti2{ 9, 10, 11, 12, 13, 14, 15 };
tuple ti3{ 47, 73, 42 };
auto sum1{ sum_t(ti1) };
auto sum2{ sum_t(ti2) };
auto sum3{ sum_t(ti3) };
cout << format("sum of ti1: {}\n", sum1);
cout << format("sum of ti2: {}\n", sum2);
cout << format("sum of ti3: {}\n", sum3);
```

输出为:

```
sum of ti1: 15
sum of ti2: 84
sum of ti3: 162
```

8.7. std::unique_ptr 管理已分配的内存

智能指针是管理已分配堆内存的优秀工具。

堆内存由 C 函数 malloc() 和 free() 在最低层进行管理。malloc() 从堆中分配一块内存，free() 将内存返还给堆。这些函数不执行初始化，也不调用构造函数或析构函数。若未能通过调用 free() 将已分配的内存返回给堆，则该行为是未定义的，通常会导致内存泄漏和安全漏洞。

C++ 提供了 new 和 delete 操作符来分配和释放堆内存，用以取代 malloc() 和 free()。new 和 delete 操作符调用对象构造函数和析构函数，但仍然不管理内存。若使用 new 分配内存，而未能使用 delete 释放它，还是会泄漏内存。

C++14 中引入的智能指针遵循资源获取即初始化 (Resource Acquisition Is Initialization, RAII) 习惯用法，当为一个对象分配内存时，将调用该对象的构造函数。当调用对象的析构函数时，内存会自动返回到堆中。

例如，当使用 make_unique() 创建一个新的智能指针时:

```
{ // beginning of scope
  auto p = make_unique<Thing>(); // memory alloc'd,
  // ctor called
  process_thing(p); // p is unique_ptr<Thing>
} // end of scope, dtor called, memory freed
```

make_unique() 为 Thing 对象分配内存，调用 Thing 默认构造函数，构造一个 unique_ptr<Thing> 对象，并返回 unique_ptr。当 p 超出作用域时，调用 Thing 析构函数，内存自动返回到堆中。

除了内存管理之外，智能指针的工作原理非常类似于基本指针:

```
auto x = *p; // *p derefs the pointer, returns Thing object
auto y = p->thname; // p-> derefs the pointer, returns member
```

unique_ptr 是一个智能指针，只允许一个指针实例。它可以移动，但不能复制。来仔细了解一下，如何使用 unique_ptr。

How to do it…

在这个示例中，用一个演示类检查 std::unique_ptr，当使用构造函数和析构函数时输出:

- 首先，创建一个简单的演示类:

```
struct Thing {
  string_view thname{ "unk" };
  Thing() {
    cout << format("default ctor: {}\n", thname);
```

```
  }
  Thing(const string_view& n) : thname(n) {
    cout << format("param ctor: {}\n", thname);
  }
  ~Thing() {
    cout << format("dtor: {}\n", thname);
  }
};
```

该类有一个默认构造函数、一个参数化构造函数和一个析构函数。每一个都有打印语句，告诉我们调用了什么。

- 当只构造一个 unique_ptr 时，不会分配内存或构造一个托管对象:

```
int main() {
  unique_ptr<Thing> p1;
  cout << "end of main()\n";
}
```

输出为:

```
end of main()
```

- 当使用 new 操作符时，会分配内存并构造一个 Thing 对象:

```
int main() {
  unique_ptr<Thing> p1{ new Thing };
  cout << "end of main()\n";
}
```

输出为:

```
default ctor: unk
end of main()
dtor: unk
```

new 操作符通过调用默认构造函数构造 Thing 对象。unique_ptr<Thing> 析构函数在智能指针到达其作用域的末尾时，调用 Thing 析构函数。

Thing 默认构造函数不初始化 thname 字符串，会保留其默认值“unk”。

- 也可以使用 make_unique():

```
int main() {
  auto p1 = make_unique<Thing>();
  cout << "end of main()\n";
}
```

输出为:

```
default ctor: unk
end of main()
dtor: unk
```

make_unique() 工厂函数负责内存分配并返回 unique_ptr 对象，这是构造 unique_ptr 的推荐方法。

- 传递给 make_unique() 的参数都将用于构造目标对象:

```
int main() {
  auto p1 = make_unique<Thing>("Thing 1") };
  cout << "end of main()\n";
}
```

输出为:

```
param ctor: Thing 1
end of main()
dtor: Thing 1
```

参数化构造函数将值赋给 thname，因此 Thing 对象现在是“Thing 1”。

- 创建一个函数，其有一个 unique_ptr<Thing> 参数:

```
void process_thing(unique_ptr<Thing> p) {
  if(p) cout << format("processing: {}\n",
    p->thname);
  else cout << "invalid pointer\n";
}
```

若尝试传递一个 unique_ptr 给这个函数，会得到一个编译器错误:

```
process_thing(p1);
```

编译器错误为：

```
error: use of deleted function...
```

这是因为函数调用试图复制 unique_ptr 对象，但删除了 unique_ptr 复制构造函数以防止复制。解决方案是让函数接受一个 const& 引用参数:

```
void process_thing(const unique_ptr<Thing>& p) {
  if(p) cout << format("processing: {}\n",
    p->thname);
  else cout << "invalid pointer\n";
}
```

输出为:

```
param ctor: Thing 1
processing: Thing 1
end of main()
dtor: Thing 1
```

- 可以用一个临时对象调用 process_thing()，其会在函数作用域的末尾立即销毁:

```
int main() {
  auto p1{ make_unique<Thing>("Thing 1") };
  process_thing(p1);
  process_thing(make_unique<Thing>("Thing 2"));
  cout << "end of main()\n";
}
```

输出为:

```
param ctor: Thing 1
processing: Thing 1
param ctor: Thing 2
processing: Thing 2
dtor: Thing 2
end of main()
dtor: Thing 1
```

How it works…

智能指针只是一个对象，在拥有和管理另一个对象的资源时，可以提供指针接口。

unique_ptr 类由其删除的复制构造函数和复制赋值操作符来区分，这可以防止智能指针的复制。不能复制 unique_ptr:

```
auto p2 = p1;
```

编译器错误:

```
error: use of deleted function...
```

但可以移动 unique_ptr:

```
auto p2 = std::move(p1);
process_thing(p1);
process_thing(p2);
```

移动后，p1 无效，p2 为“Thing 1”。

输出为:

```
invalid pointer
processing: Thing 1
end of main()
dtor: Thing 1
```

unique_ptr 接口有一个重置指针的方法:

```
p1.reset(); // pointer is now invalid
process_thing(p1);
```

输出为:

```
dtor: Thing 1
invalid pointer
```

reset() 方法也可以将管理对象替换为另一个相同类型的对象:

```
p1.reset(new Thing("Thing 3"));
process_thing(p1);
```

输出为:

```
param ctor: Thing 3
dtor: Thing 1
processing: Thing 3
```

8.8. std::shared_ptr 的共享对象

shared_ptr 类是一个智能指针，拥有自己的托管对象，并维护一个使用计数器来跟踪副本。此示例探索了 shared_ptr 的使用方式，以在共享指针副本的同时管理内存。

> Note
>
> 有关智能指针的更多详细信息，请参阅本章前面关于使用 std::unique_ptr 管理已分配内存的介绍。

How to do it…

在这个示例中，用一个演示类检查 std::shared_ptr，当调用它的构造函数和析构函数时进行打印输出:

- 首先，创建一个简单的演示类:

```
struct Thing {
  string_view thname{ "unk" };
  Thing() {
    cout << format("default ctor: {}\n", thname);
  }
  Thing(const string_view& n) : thname(n) {
    cout << format("param ctor: {}\n", thname);
  }
  ~Thing() {
    cout << format("dtor: {}\n", thname);
  }
};
```

该类有一个默认构造函数、一个参数化构造函数和一个析构函数。每一个都有打印语句，显示调用了哪些函数。

- shared_ptr 类的工作原理与其他智能指针非常相似，也可以用 new 操作符或 make_shared() 函数来构造:

```
int main() {
  shared_ptr<Thing> p1{ new Thing("Thing 1") };
  auto p2 = make_shared<Thing>("Thing 2");
  cout << "end of main()\n";
}
```

输出为:

```
param ctor: Thing 1
param ctor: Thing 2
end of main()
dtor: Thing 2
dtor: Thing 1
```

推荐使用 make_shared() 函数，它会管理构造过程，并且不太容易出错。
与其他智能指针一样，当指针超出作用域时，托管对象将销毁，其内存将返回到堆中。

- 下面是一个检查 shared_ptr 对象的使用计数的函数:

```
void check_thing_ptr(const shared_ptr<Thing>& p) {
  if(p) cout << format("{} use count: {}\n",
  p->thname, p.use_count());
  else cout << "invalid pointer\n";
}
```

thname 是 Thing 类的成员，因此可以通过带有 p-> 成员解引用操作符的指针访问它。use_count() 函数是 shared_ptr 类的成员，因此，可以使用 p.object 成员操作符访问它。
让我们使用指针来对其进行调用:

```
check_thing_ptr(p1);
check_thing_ptr(p2);
```

输出为:

```
Thing 1 use count: 1
Thing 2 use count: 1
```

- 当复制指针时，使用计数会增加，但不会构造新的对象:

```
cout << "make 4 copies of p1:\n";
auto pa = p1;
auto pb = p1;
auto pc = p1;
auto pd = p1;
check_thing_ptr(p1);
```

输出为:

```
make 4 copies of p1:
Thing 1 use count: 5
```

- 当我们检查其他副本时，会得到相同的结果:

```
check_thing_ptr(pa);
check_thing_ptr(pb);
check_thing_ptr(pc);
check_thing_ptr(pd);
```

输出为:

```
Thing 1 use count: 5
Thing 1 use count: 5
Thing 1 use count: 5
Thing 1 use count: 5
```

每个指针都会报告相同的使用计数。

- 当副本超出作用域时，将进行销毁，并使计数递减:

```
{ // new scope
  cout << "make 4 copies of p1:\n";
  auto pa = p1;
  auto pb = p1;
  auto pc = p1;
  auto pd = p1;
```

```
  check_thing_ptr(p1);
} // end of scope
check_thing_ptr(p1);
```

输出为:

```
make 4 copies of p1:
Thing 1 use count: 5
Thing 1 use count: 1
```

- 销毁副本将减少使用计数，但不会销毁管理对象。当最终副本超出作用域且使用计数为零时，对象将销毁:

```
{
  cout << "make 4 copies of p1:\n";
  auto pa = p1;
  auto pb = p1;
  auto pc = p1;
  auto pd = p1;
  check_thing_ptr(p1);
  pb.reset();
  p1.reset();
  check_thing_ptr(pd);
} // end of scope
```

输出为:

```
make 4 copies of p1:
Thing 1 use count: 5
Thing 1 use count: 3
dtor: Thing 1
```

销毁 pb(一个副本) 和 p1(原始) 会留下三个指针副本 (pa、bc 和 pd)，因此管理对象仍然存在。其余三个指针副本将在创建它们的作用域的末尾销毁。然后销毁对象，并将其内存返回到堆中。

How it works…

shared_ptr 类的区别在于它对指向同一个托管对象的多个指针的管理。

shared_ptr 对象的复制构造函数和复制赋值操作符增加一个使用计数器。析构函数将使用计数器减为零，然后销毁托管对象，并将其内存返还给堆。

shared_ptr 类同时管理托管对象和堆分配的控制块。控制块包含使用计数器以及其他管家对象，控制块与管理对象一起在副本之间进行管理和共享。所以原始 shared_ptr 对象将控制权转让给其副本，以便其他 shared_ptr 可以管理对象及其内存。

8.9. 对共享对象使用弱指针

严格地说，std::weak_ptr 不是一个智能指针。相反，它是一个与 shared_ptr 合作运行的观察者。weak_ptr 本身不保存指针。

某些情况下，shared_ptr 对象可能会创建悬空指针或数据竞争，这可能导致内存泄漏或其他问题。解决方案是使用具有 shared_ptr 的 weak_ptr 对象。

How to do it…

这个示例中，使用 std::shared_ptr 来检查 std::weak_ptr 的使用。创建一个演示类，当调用它的构造函数和析构函数时进行打印。

- 展示将要在 shared_ptr 和 unique_ptr 中创建的类型:

```
struct Thing {
  string_view thname{ "unk" };
  Thing() {
    cout << format("default ctor: {}\n", thname);
  }
  Thing(const string_view& n) : thname(n) {
    cout << format("param ctor: {}\n", thname);
  }
  ~Thing() {
    cout << format("dtor: {}\n", thname);
  }
};
```

该类有一个默认构造函数、一个参数化构造函数和一个析构函数。每一个都有一个简单的打印语句告诉我们调用了什么。

- 还需要一个函数来检查 weak_ptr 对象:

```
void get_weak_thing(const weak_ptr<Thing>& p) {
  if(auto sp = p.lock()) cout <<
    format("{}: count {}\n", sp->thname,
      p.use_count());
  else cout << "no shared object\n";
}
```

weak_ptr 本身不作为指针操作，其需要使用 shared_ptr.lock() 函数返回一个 shared_ptr 对象，然后可以使用该对象访问托管对象。

- 因为 weak_ptr 需要一个相关的 shared_ptr，将通过创建一个 shared_ptr<Thing> 对象来启动 main()。当创建一个 weak_ptr 对象而不分配 shared_ptr 时，会对过期 (expired) 标志进行设置:

```
int main() {
  auto thing1 = make_shared<Thing>("Thing 1");
  weak_ptr<Thing> wp1;
  cout << format("expired: {}\n", wp1.expired());
  get_weak_thing(wp1);
}
```

输出为:

```
param ctor: Thing 1
expired: true
no shared object
```

make_shared() 函数分配内存并构造一个 Thing 对象。

weak_ptr<Thing> 声明构造了一个 weak_ptr 对象，但没有分配 shared_ptr。因此，当检查过期标志时，其为真，表明并没有相关的 shared_ptr。

因为没有 shared_ptr 可用，所以 get_weak_thing() 函数不能获得锁。

- 将 shared_ptr 分配给 weak_ptr 时，可以使用 weak_ptr 来访问托管对象:

```
wp1 = thing1;
get_weak_thing(wp1);
```

输出为:

```
Thing 1: count 2
```

get_weak_thing() 函数现在可以获取锁并访问托管对象。lock() 方法返回一个 shared_ptr，use_count() 反映了现在有第二个 shared_ptr 管理 Thing 对象的情况。

新 shared_ptr 在 get_weak_thing() 作用域的末尾销毁。

- weak_ptr 类有一个构造函数，其接受 shared_ptr 进行一步构造:

```
weak_ptr<Thing> wp2(thing1);
get_weak_thing(wp2);
```

输出为:

```
Thing 1: count 2
```

use_count() 再次为 2，之前的 shared_ptr 在其封闭的 get_weak_thing() 作用域结束时销毁。

- 重置 shared_ptr 时，其关联的 weak_ptr 对象将过期:

```
thing1.reset();
get_weak_thing(wp1);
get_weak_thing(wp2);
```

输出为:

```
dtor: Thing 1
no shared object
no shared object
```

reset() 后，使用计数为零，管理对象销毁，内存释放。

How it works…

weak_ptr 对象是一个观察者，其对 shared_ptr 对象属于非所有引用。weak_ptr 会观察 shared_ptr，这样就知道托管对象什么时候可用，什么时候不可用。这允许在您不需要知道托管对象是否处于活动状态的情况下，使用 shared_ptr。

weak_ptr 类有一个 use_count() 函数，返回 shared_ptr 的使用计数，若管理对象已删除，则返回 0:

```
long use_count() const noexcept;
```

weak_ptr 还有一个 expired() 函数，会报告托管对象是否已删除:

```
bool expired() const noexcept;
```

lock() 函数是访问 shared_ptr 的首选方法，其检查 expired() 以查看管理对象是否可用。若可用，将返回一个新的 shared_ptr，该 shared_ptr 与管理对象共享所有权。否则，会返回一个空的 shared_ptr。这些会作为一个原子操作完成:

```
std::shared_ptr<T> lock() const noexcept;
```

There's more…

weak_ptr 的一个重要用例是有可能循环引用 shared_ptr 对象。例如，考虑两个相互链接的类的情况 (可能在一个层次结构中):

```
struct circB;
struct circA {
  shared_ptr<circB> p;
  ~circA() { cout << "dtor A\n"; }
};
struct circB {
  shared_ptr<circA> p;
  ~circB() { cout << "dtor B\n"; }
};
```

在析构函数中有 print 语句，因此可以看到对象何时销毁。现在，可以通过 shared_ptr 创建两个相互指向的对象:

```
int main() {
  auto a{ make_shared<circA>() };
  auto b{ make_shared<circB>() };
  a->p = b;
  b->p = a;
  cout << "end of main()\n";
}
```

当运行这个函数时，会发现析构函数从未调用:

```
end of main()
```

因为对象维护着相互引用的共享指针，所以使用计数永远不会达到零，并且托管对象永远不会销毁。

可以通过更改其中一个类来使用 weak_ptr 来解决这个问题:

```
struct circB {
  weak_ptr<circA> p;
  ~circB() { cout << "dtor B\n"; }
};
```

main() 中的代码保持不变，我们会得到这样的输出:

```
end of main()
dtor A
dtor B
```

通过将一个 shared_ptr 更改为 weak_ptr，解决了循环引用，并且现在在对象作用域的末尾，可以正确地销毁对象了。

8.10. 共享管理对象的成员

std::shared_ptr 类提供了一个别名构造函数，来共享由另一个不相关指针管理的指针:

```
shared_ptr( shared_ptr<Y>&& ref, element_type* ptr ) noexcept;
```

这将返回一个别名 shared_ptr 对象，该对象使用 ref 的资源，但返回一个指向 ptr 的指针，use_count 和 deleter 与 ref 共享，但是 get() 返回 ptr。这可以在不共享整个对象的情况下共享托管对象的一个成员，并且在仍然使用该成员时不允许删除对象。

How to do it…

这个示例中，我们创建了一个托管对象并共享该对象的成员:

- 从托管对象的类开始:

```
struct animal {
  string name{};
  string sound{};
  animal(const string& n, const string& a)
  : name{n}, sound{a} {
    cout << format("ctor: {}\n", name);
  }
  ~animal() {
    cout << format("dtor: {}\n", name);
  }
```

```
};
```

该类有两个成员，animal 对象的名称和声音的字符串类型。构造函数和析构函数也有 print 语句。

- 现在，需要函数来创建 animal，但只分享名字和声音:

```
auto make_animal(const string& n, const string& s) {
  auto ap = make_shared<animal>(n, s);
  auto np = shared_ptr<string>(ap, &ap->name);
  auto sp = shared_ptr<string>(ap, &ap->sound);
  return tuple(np, sp);
}
```

这个函数创建与动物对象 shared_ptr，由名字和声音构造。然后，为名字和声音创建别名 shared_ptr 对象。当返回 name 和 sound 指针时，animal 指针将超出作用域。因为别名指针可以防止使用计数达到零，所以不会删除。

- main() 函数中，使用 make_animal() 并检查结果:

```
int main() {
  auto [name, sound] =
    make_animal("Velociraptor", "Grrrr!");
  cout << format("The {} says {}\n", *name, *sound);
  cout << format("Use count: name {}, sound {}\n",
    name.use_count(), sound.use_count());
}
```

输出为:

```
ctor: Velociraptor
The Velociraptor says Grrrr!
Use count: name 2, sound 2
dtor: Velociraptor
```

每个别名指针的 use_count 值为 2。当 make_animal() 函数创建别名指针时，其会增加 animal 指针的使用次数。当函数结束时，animal 指针超出作用域，使其使用计数为 2，这反映在别名指针中。别名指针在 main() 的末尾超出了作用域，可以销毁 animal 指针。

How it works…

别名共享指针看起来有点抽象，但它比看起来要简单。

共享指针使用一个控制块来管理资源。一个控制块与一个托管对象相关联，并在共享该对象的指针之间共享。控制块一般包含:

- 指向管理对象的指针
- 删除器
- 分配器

- 拥有托管对象的 shared_ptr 对象的数量 (使用计数)
- 引用管理对象的 weak_ptr 对象的数量

使用别名共享指针的情况下，控制块包括指向别名对象的指针，其他不变。

别名共享指针参与使用计数，就像非别名共享指针一样，防止托管对象在使用计数为零之前销毁。删除器没有改变，所以它会破坏管理对象。

重要的 Note

可以使用任何指针来构造别名共享指针。通常，指针指向别名对象中的成员。若别名指针没有引用托管对象的元素，则需要分别管理其构造和销毁。

8.11. 比较随机数引擎

随机库提供了选择的随机数生成器，每个生成器具有不同的策略和属性。本节中，我们检查一个函数，通过创建输出的直方图来比较不同的选项。

How to do it…

这个示例中，比较了 C++ 随机库提供的不同随机数生成器:

- 从一些常量开始，为随机数生成器提供统一参数:

```
constexpr size_t n_samples{ 1000 };
constexpr size_t n_partitions{ 10 };
constexpr size_t n_max{ 50 };
```

n_samples 是要检查的样本数量，n_partitions 是用来显示样本的分区数量，n_max 是直方图中条形图的最大值 (由于四舍五入的关系，这个值会有所变化)。

这些数字合理地显示了引擎之间的差异。增加样本与分区的比例往往会使曲线变得平滑，并模糊引擎之间的差异。

- 这是一个收集随机数样本并显示直方图的函数:

```
template <typename RNG>
void histogram(const string_view& rng_name) {
  auto p_ratio = (double)RNG::max() / n_partitions;
  RNG rng{}; // construct the engine object

  // collect the samples
  vector<size_t> v(n_partitions);
  for(size_t i{}; i < n_samples; ++i) {
    ++v[rng() / p_ratio];
  }

  // display the histogram
  auto max_el = std::max_element(v.begin(),
    v.end());
  auto v_ratio = *max_el / n_max;
  if(v_ratio < 1) v_ratio = 1;
```

```
  cout << format("engine: {}\n", rng_name);
  for(size_t i{}; i < n_partitions; ++i) {
    cout << format("{:02}:{:*<{}}\n",
      i + 1, ' ', v[i] / v_ratio);
  }
  cout << '\n';
}
```

简而言之，这个函数将收集的样本的直方图存储在一个 vector 中。然后，在控制台上以星号的形式显示直方图。

- 在 main() 使用 histogram()，如下所示:

```
int main() {
  histogram<std::random_device>("random_device");
  histogram<std::default_random_engine>
    ("default_random_engine");
  histogram<std::minstd_rand0>("minstd_rand0");
  histogram<std::minstd_rand>("minstd_rand");
  histogram<std::mt19937>("mt19937");
  histogram<std::mt19937_64>("mt19937_64");
  histogram<std::ranlux24_base>("ranlux24_base");
  histogram<std::ranlux48_base>("ranlux48_base");
  histogram<std::ranlux24>("ranlux24");
  histogram<std::ranlux48>("ranlux48");
  histogram<std::knuth_b>("knuth_b");
}
```

输出为:

```
engine: random_device
01: ************************************************
02: *********************************************
03: ************************************************
04: *********************************************
05: *********************************************
06: *********************************************
07: ************************************************
08: *********************************************
09: ************************************************
10: ************************************************

engine: default_random_engine
01: *********************************************
02: *********************************************
03: *********************************************
04: *********************************************
05: ************************************************************
06: ************************************************
07: ************************************************
08: *********************************************
09: *********************************************
10: *********************************************
```

图 8.1 前两个随机数引擎输出的截图

截图显示了前两个随机数引擎的直方图。

若将 n_samples 的值提高到 100,000，会发现引擎之间的差异变得更难辨别:

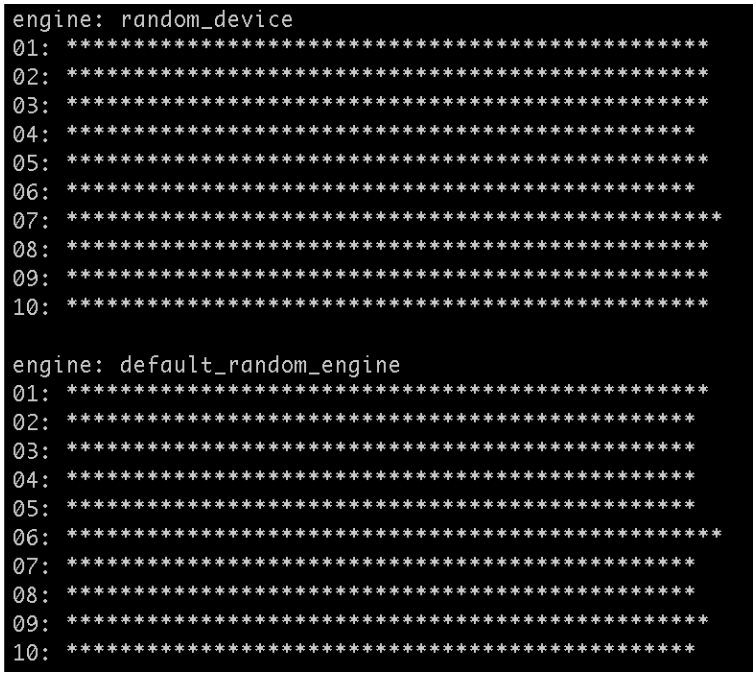


图 8.2 100,000 个样本的输出截图

How it works…

每个随机数引擎都有一个函子接口，返回序列中的下一个随机数:

```
result_type operator()();
```

函子会返回一个随机值，平均分布在 min() 和 max() 值之间。所有的随机数引擎都有这个接口。

histogram() 函数可以通过在模板中使用随机数引擎类，来利用这种一致性:

```
template <typename RNG>
```

(RNG 是随机数生成器 (Random Number Generator) 的缩写。标准库文档将这些类称为引擎，就我们的目的而言，这与 RNG 同义。)

我们用 RNG 类实例化一个对象，并在 vector 中创建一个直方图:

```
RNG rng{};
vector<size_t> v(n_partitions);
for(size_t i{}; i < n_samples; ++i) {
  ++v[rng() / p_ratio];
}
```

使用这种方式，可以很容易地比较各种随机数引擎的结果。

There's more…

库中的每个随机数引擎都有不同的方法和特征。当多次运行直方图时，会注意到大多数引擎在每次运行时都具有相同的分布，因为它们是确定的——每次生成相同的数字序列。std::random_device 在大多数系统上是不确定的。若需要更多的变化，可以使用它为其他引擎生成随机种子。使用当前日期和时间为 RNG 种子的情况也很常见。

std::default_random_engine 对于多数情况来说都是一个合适的选择。

8.12. 比较随机数分布发生器

C++ 标准库提供了一系列随机数分布生成器，每个生成器都有自己的属性。本节中，我们检查一个函数，通过创建输出的直方图来比较不同的选项。

How to do it…

与随机数引擎一样，分布生成器也有一些公共的接口元素。与随机数引擎不同，分布生成器有各种属性可以设置。可以创建一个模板函数来打印各种分布的直方图，但各种分布生成器的初始化差异很大:

- 先从一些常数开始:

```
constexpr size_t n_samples{ 10 * 1000 };
constexpr size_t n_max{ 50 };
```

n_samples 常数是为每个直方图生成的样本数量——在本例中为 10,000。
生成直方图时，n_max 常数作为除数。

- 直方图函数以分布生成器作为参数，并打印该分布算法的直方图:

```
void dist_histogram(auto distro,
    const string_view& dist_name) {
  std::default_random_engine rng{};
  map<long, size_t> m;

  // create the histogram map
  for(size_t i{}; i < n_samples; ++i)
    ++m[(long)distro(rng)];

  // print the histogram
  auto max_elm_it = max_element(m.begin(), m.end(),
    [](const auto& a, const auto& b)
    { return a.second < b.second; }
    );
  size_t max_elm = max_elm_it->second;
  size_t max_div = std::max(max_elm / n_max,
    size_t(1));
  cout << format("{}:\n", dist_name);
  for (const auto [randval, count] : m) {
    if (count < max_elm / n_max) continue;
    cout << format("{:3}:{:*<{}}\n",
      randval, ' ', count / max_div);
  }
}
```

dist_histogram() 函数的作用是: 使用 map 来存储直方图。然后，在控制台上以星号的形式显示直方图。

- 在 main() 调用 dist_histogram():

```
int main() {
```

```
  dist_histogram(std::uniform_int_distribution<int>
    {0, 9}, uniform_int_distribution");
  dist_histogram(std::normal_distribution<double>
    {0.0, 2.0}, "normal_distribution");
  ...
```

调用 dist_histogram() 函数比调用随机数生成器要复杂得多。每个随机分布类别根据其算法有不同的参数集。

要获得完整的列表，请参考 GitHub 库中的 distribution.cpp 文件。

输出为:

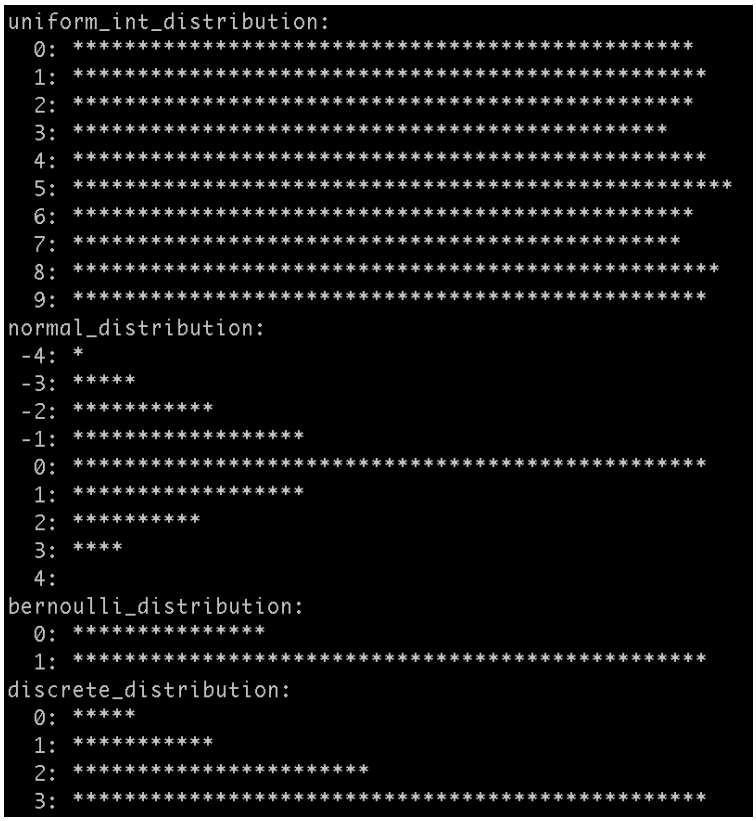


图 8.3 随机分布直方图的截图

每种分布算法产生非常不同的输出，可以试一下每个随机分布生成器的不同选项。

How it works…

每个分布生成器都有一个返回随机分布中下一个值的函子:

```
result_type operator()( Generator& g );
```

函子以随机数生成器 (RNG) 对象作为参数:

```
std::default_random_engine rng{};
map<long, size_t> m;
for (size_t i{}; i < n_samples; ++i) ++m[(long)distro(rng)];
```

现在，我们在 RNG 中使用 std::default_random_engine。

与 RNG 直方图一样，这是一个有用的工具，可以可视化随机库中可用的各种随机分布算法。可以对每种算法可用的各种参数进行试验。

第 9 章　并发和并行

并发性和并行性指的是在不同的执行线程中运行代码的能力。并发性是在后台运行线程的能力，并行性是在处理器的不同内核中同时运行线程的能力。

运行时库以及主机操作系统，将为给定硬件环境中的线程，在并发和并行执行模型之间进行选择。

在现代多任务操作系统中，main() 函数已经代表了一个执行线程。当一个新线程启动时，可由现有的线程派生。

C++ 标准库中，std::thread 类提供了线程执行的基本单元。其他类构建在线程之上，以提供锁、互斥和其他并发模式。根据系统架构的不同，执行线程可以在一个处理器上并发运行，也可以在不同的内核上并行运行。

- 休眠一定的时间
- std::thread——实现并发
- std::async——实现并发
- STL 算法与执行策略
- 互斥锁和锁——安全地共享数据
- std::atomic——共享标志和值
- std::call_once——初始化线程
- std::condition_variable——解决生产者-消费者问题
- 实现多个生产者和消费者

9.1. 相关准备

本章所有的例子和源代码都可以在本书的 GitHub 库中找到，网址是https://github.com/PacktPublishing/CPP-20-STL-Cookbook/tree/main/chap09。

9.2. 休眠一定的时间

<thread> 头文件提供了两个使线程进入休眠状态的函数，sleep_for() 和 sleep_until()。这两个函数都在 std::this_thread 命名空间中。

本节探讨了这些函数的使用，我们将在本章后面使用它们。

How to do it…

看看如何使用 sleep_for() 和 sleep_until() 函数:

- 与休眠相关的函数在 std::this_thread 命名空间中。只有几个符号，可以为 std::this_thread 和 std::chrono_literals 使用 using 指令:

```
using namespace std::this_thread;
using namespace std::chrono_literals;
```

chrono_literals 命名空间具有表示持续时间的符号，例如 1s 表示一秒，100ms 表示 100 毫秒。

- 在 main() 中，用 steady_clock::now() 标记一个时间点，这样就可以计算测试时间了:

```
int main() {
  auto t1 = steady_clock::now();
  cout << "sleep for 1.3 seconds\n";
  sleep_for(1s + 300ms);
  cout << "sleep for 2 seconds\n";
  sleep_until(steady_clock::now() + 2s);
  duration<double> dur1 = steady_clock::now() - t1;
  cout << format("total duration: {:.5}s\n",
    dur1.count());
}
```

sleep_for() 函数的作用是: 接受一个 duration 对象来指定睡眠时间。参数 (1s + 300ms) 使用 chrono_literal 操作符返回一个表示 1.3 秒的 duration 对象。

sleep_until() 函数接受一个 time_point 对象来指定从睡眠状态恢复的特定时间。使用 chrono_literal 操作符，来修改从 steady_clock::now() 返回的 time_point 对象。

输出:

```
sleep for 1.3 seconds
sleep for 2 seconds
total duration: 3.3005s
```

How it works…

sleep_for(duration) 和 sleep_until(time_point) 函数在指定的时间内暂停当前线程的执行，或者直到到达 time_point。若支持的话，sleep_for() 函数将使用 steady_clock 实现。

否则，持续时间可能会有所调整。由于调度或资源延迟，这两个功能可能会阻塞更长的时间。

There's more…

一些系统支持 POSIX 函数 sleep()，会在指定的秒数内暂停执行:

```
unsigned int sleep(unsigned int seconds);
```

sleep() 函数是 POSIX 标准的一部分，而不是 C++ 标准的一部分。

9.3. std::thread——实现并发

线程是并发的单位，main() 函数可以看作是执行的主线程。在操作系统上下文中，主线程与其他进程拥有的其他线程并发运行。

thread 类是 STL 中并发性的根本，所有其他并发特性都建立在线程类的基础上。

本节中，我们将了解 std::thread 的基础知识，以及 join() 和 detach() 如何确定其执行上下文。

How to do it…

在这个示例中，我们创建了一些 std::thread 对象，并测试了它们的执行选项。

- 从线程休眠函数开始，单位是毫秒:

```
void sleepms(const unsigned ms) {
  using std::chrono::milliseconds;
  std::this_thread::sleep_for(milliseconds(ms));
}
```

sleep_for() 函数接受一个 duration 对象，并在指定的时间内阻塞当前线程的执行。sleepms() 函数作为一个方便的包装器，其接受一个 unsigned 值，表示休眠的毫秒数。

- 现在，需要线程的一个函数。这个函数休眠的毫秒数是可变的，基于一个整数参数:

```
void fthread(const int n) {
  cout << format("This is t{}\n", n);
  for(size_t i{}; i < 5; ++i) {
    sleepms(100 * n);
    cout << format("t{}: {}\n", n, i + 1);
  }
  cout << format("Finishing t{}\n", n);
}
```

fthread() 调用 sleepms() 5 次，每次休眠 100 * n 毫秒。

- 可以在一个单独的线程中 (比如主线程) 运行 std::thread:

```
int main() {
  thread t1(fthread, 1);
  cout << "end of main()\n";
}
```

可以编译，但运行时会有错误:

```
terminate called without an active exception Aborted
```

(错误信息会有所不同。这是在 Debian 和 GCC 上的错误信息。)
问题在于，当线程对象超出作用域时，操作系统不知道该如何处理。必须指定调用者是否等待线程，或者是否分离并独立运行。

- 使用 join() 来指示调用者将等待线程完成:

```
int main() {
  thread t1(fthread, 1);
  t1.join();
  cout << "end of main()\n";
}
```

输出为:

```
This is t1
t1: 1
t1: 2
t1: 3
t1: 4
t1: 5
Finishing t1
end of main()
```

现在，main() 会等待线程完成。

- 若使用 detach()，那么 main() 不会等待，程序在线程运行之前就结束了:

```
thread t1(fthread, 1);
t1.detach();
```

输出为:

```
end of main()
```

- 当线程分离后，需要给它一些运行的时间:

```
thread t1(fthread, 1);
t1.detach();
cout << "main() sleep 2 sec\n";
sleepms(2000);
```

输出为:

```
main() sleep 2 sec
This is t1
t1: 1
t1: 2
t1: 3
t1: 4
t1: 5
Finishing t1
end of main()
```

- 分离第二个线程，看看会发生什么:

```
int main() {
  thread t1(fthread, 1);
  thread t2(fthread, 2);
```

```
  t1.detach();
  t2.detach();
  cout << "main() sleep 2 sec\n";
  sleepms(2000);
  cout << "end of main()\n";
}
```

输出为:

```
main() sleep 2 sec
This is t1
This is t2
t1: 1
t2: 1
t1: 2
t1: 3
t2: 2
t1: 4
t1: 5
Finishing t1
t2: 3
t2: 4
t2: 5
Finishing t2
end of main()
```

因为 fthread() 函数使用它的形参作为 sleepms() 的乘法器，所以第二个线程比第一个线程运行得慢一些。我们可以看到计时器在输出中交错。

- 若使用 join()，而不是 detatch() 来执行此操作，会得到类似的结果:

```
int main() {
  thread t1(fthread, 1);
  thread t2(fthread, 2);
  t1.join();
  t2.join();
  cout << "end of main()\n";
}
```

输出为:

```
This is t1
This is t2
t1: 1
t2: 1
t1: 2
t1: 3
t2: 2
t1: 4
t1: 5
Finishing t1
t2: 3
t2: 4
t2: 5
Finishing t2
end of main()
```

因为 join() 等待线程完成，所以不再需要 main() 中 2 秒的 sleepms() 来等待线程完成。

How it works…

std::thread 对象表示一个执行线程，对象和线程之间是一对一的关系。一个线程对象表示一个线程，一个线程由一个线程对象表示。线程对象不能复制或赋值，但可以移动。

其构造函数是这样:

```
explicit thread( Function&& f, Args&&… args );
```

线程是用一个函数指针和零个或多个参数构造的。函数会立即调用:

```
thread t1(fthread, 1);
```

这将创建对象 t1，并立即以字面值 1 作为参数调用函数 fthread(int)。

创建线程后，必须在线程上使用 join() 或 detach():

```
t1.join();
```

join() 方法阻塞调用线程的执行，直到 t1 线程完成:

```
t1.detach();
```

detach() 方法允许调用线程独立于 t1 线程继续运行。

There’s more…

C++20 提供了 std::jthread，其会自动在作用域的末尾汇入创建线程:

```
int main() {
  std::jthread t1(fthread, 1);
  cout "< "end of main("\n";
}
```

输出为:

```
end of main()
This is t1
t1: 1
t1: 2
t1: 3
t1: 4
t1: 5
Finishing t1
```

t1 线程可以独立执行，在其作用域的末尾会自动汇入 main() 线程。

9.4. std::async——实现并发

std::async() 异步运行一个目标函数，并返回一个携带目标函数返回值的 std::future 对象。通过这种方式，async() 的操作很像 std::thread，但允许函数有返回值。

通过几个例子来了解一下 std::async() 的使用。

How to do it…

在其最简单的形式中，std::async() 函数执行与 std::thread 相同的任务，不需要调用 join() 或 detach()，同时还允许通过 std::future 对象返回值。

这个示例中，将使用一个函数来计算一个范围内质数的数量。可以使用 chrono::steady_clock 为每个线程的进行计时。

- 先从别名开始:

```
using launch = std::launch;
using secs = std::chrono::duration<double>;
```

  std::launch 有启动策略常量，用于 async() 调用。secs 别名是一个持续时间类，用于为质数计算计时。

- 我们的目标函数计算范围内的质数。这本质上是一种通过占用一些时钟周期，来理解执行策略的方法:

```
struct prime_time {
  secs dur{};
  uint64_t count{};
};
```

```
prime_time count_primes(const uint64_t& max) {
  prime_time ret{};
  constexpr auto isprime = [](const uint64_t& n) {
    for(uint64_t i{ 2 }; i < n / 2; ++i) {
      if(n % i == 0) return false;
    }
    return true;
  };
  uint64_t start{ 2 };
  uint64_t end{ max };
  auto t1 = steady_clock::now();
  for(uint64_t i{ start }; i <= end ; ++i) {
    if(isprime(i)) ++ret.count;
  }
  ret.dur = steady_clock::now() - t1;
  return ret;
}
```

prime_time 结构用于返回值，可用于 duration 和 count，这样就可以计算循环本身的时间。若一个值是质数，isprime 返回 true。我们使用 steady_clock，来计算质数计数循环的持续时间。

- 在 main() 中，调用函数并报告其运行时间:

```
int main() {
  constexpr uint64_t MAX_PRIME{ 0x1FFFF };
  auto pt = count_primes(MAX_PRIME);
  cout << format("primes: {} {:.3}\n", pt.count,
    pt.dur);
}
```

输出为:

```
primes: 12252 1.88008s
```

- 现在，可以用 std::async() 异步运行 count_primes():

```
int main() {
  constexpr uint64_t MAX_PRIME{ 0x1FFFF };
  auto primes1 = async(count_primes, MAX_PRIME);
  auto pt = primes1.get();
  cout << format("primes: {} {:.3}\n", pt.count,
    pt.dur);
}
```

这里，使用 count_primes 函数和 MAX_PRIME 参数调用 async()，这 count_primes 函数放在后台运行。

async() 返回 std::future 对象，该对象携带异步操作的返回值。future 对象的 get() 是阻塞式的，会等到异步函数完成，获取该函数的返回对象。

这与没有 async() 时运行的时间几乎相同:

```
primes: 12252 1.97245s
```

- async() 函数可选地将执行策略标志作为其第一个参数:

```
auto primes1 = async(launch::async, count_primes, MAX_
PRIME);
```

选择是异步或延迟的，这些标志位于 std::launch 命名空间中。

async 标志启用异步操作，deferred 标志启用延迟计算。这些标志是位映设的，可以与按位或 | 操作符组合使用。

默认情况下，这两个位都要设置，就和 async|deferred 一样。

- 可以使用 async() 同时运行函数的几个实例:

```
int main() {
  constexpr uint64_t MAX_PRIME{ 0x1FFFF };
  list<std::future<prime_time>> swarm;
  cout << "start parallel primes\n";
  auto t1{ steady_clock::now() };
  for(size_t i{}; i < 15; ++i) {
    swarm.emplace_back(
    async(launch::async, count_primes,
    MAX_PRIME)
    );
  }
  for(auto& f : swarm) {
    static size_t i{};
    auto pt = f.get();
    cout << format("primes({:02}): {} {:.5}\n",
    ++i, pt.count, pt.dur);
  }
  secs dur_total{ steady_clock::now() - t1 };
  cout << format("total duration: {:.5}s\n",
    dur_total.count());
}
```

我们知道 async 返回一个 future 对象。因此，可以通过将未来对象存储在容器中来运行 15 个线程。以下是我运行在 Windows 设备 (6 核 i7) 上的输出:

```
start parallel primes
primes(01): 12252 4.1696s
primes(02): 12252 3.7754s
primes(03): 12252 3.78089s
primes(04): 12252 3.72149s
primes(05): 12252 3.72006s
primes(06): 12252 4.1306s
primes(07): 12252 4.26015s
primes(08): 12252 3.77283s
primes(09): 12252 3.77176s
primes(10): 12252 3.72038s
primes(11): 12252 3.72416s
primes(12): 12252 4.18738s
primes(13): 12252 4.07128s
primes(14): 12252 2.1967s
primes(15): 12252 2.22414s
total duration: 5.9461s
```

尽管，6 核 i7 不能在不同的核中运行所有进程，但仍然可以在 6 秒内完成 15 个实例。

看起来它在 4 秒内完成了前 13 个线程，然后再花 2 秒完成最后 2 个线程。这似乎利用了 Intel 的超线程技术，该技术允许在某些情况下在一个核心中运行 2 个线程。

当在 12 核 Xeon 上运行相同的代码时，会得到这样的结果:

```
start parallel primes
primes(01): 12252 0.96221s
primes(02): 12252 0.97346s
primes(03): 12252 0.92189s
primes(04): 12252 0.97499s
primes(05): 12252 0.98135s
primes(06): 12252 0.93426s
primes(07): 12252 0.90294s
primes(08): 12252 0.96307s
primes(09): 12252 0.95015s
primes(10): 12252 0.94255s
primes(11): 12252 0.94971s
primes(12): 12252 0.95639s
primes(13): 12252 0.95938s
primes(14): 12252 0.92115s
primes(15): 12252 0.94122s
total duration: 0.98166s
```

12 核的至强处理器可以在一秒钟内完成全部 15 个进程的任务。

How it works…

理解 std::async 的关键在于其对 std::promise 和 std::future 的使用。

promise 类允许线程存储一个对象，以后可以由 future 对象进行异步检索。

例如，有这样一个函数:

```
void f() {
  cout << "this is f()\n";
}
```

可以用 std::thread 运行:

```
int main() {
  std::thread t1(f);
  t1.join();
  cout << "end of main()\n";
}
```

这对于没有返回值的简单函数来说很好。当想从 f() 返回一个值时，可以使用 promise 和 future。

在 main() 线程中设置了 promise 和 future 对象:

```
int main() {
  std::promise<int> value_promise;
  std::future<int> value_future =
```

```
    value_promise.get_future();
  std::thread t1(f, std::move(value_promise));
  t1.detach();
  cout << format("value is {}\n", value_future.get());
  cout << "end of main()\n";
}
```

然后，将 promise 对象传递给函数:

```
void f(std::promise<int> value) {
  cout << "this is f()\n";
  value.set_value(47);
}
```

promise 对象不能复制，因此需要使用 std::move 将其传递给函数。

promise 对象充当到 future 对象的桥梁，其允许在值可用的前提下，对其进行检索。

std::async() 只是一个辅助函数，用于简化 promise 和 future 对象的创建。可以这样使用 async():

```
int f() {
  cout << "this is f()\n";
  return 47;
}

int main() {
  auto value_future = std::async(f);
  cout << format("value is {}\n", value_future.get());
  cout << "end of main()\n";
}
```

这就是 async() 的值。在很多情况下，promise 与 future 的组合使用起来更容易。

9.5. STL 算法与执行策略

从 C++17 开始，许多标准的 STL 算法可以并行执行。该特性允许算法将其工作拆分为子任务，以便在多个核上同时运行。这些算法接受一个执行策略对象，该对象指定应用于算法的并行度类型，该特性需要硬件支持。

How to do it…

执行策略在 <execution> 头文件中和 std:: Execution 命名空间中定义。本节中，我们将使用 std::transform() 算法测试可用的策略:

- 出于计时的目的，将使用带有 std::milli 比率的 duration 对象，这样就可以以毫秒为单位进行测量:

```
using dur_t = duration<double, std::milli>;
```

- 出于演示目的，将从一个具有 1000 万个随机值的 vector<unsigned> 开始:

```
int main() {
  std::vector<unsigned> v(10 * 1000 * 1000);
  std::random_device rng;
  for(auto &i : v) i = rng() % 0xFFFF;
  ...
```

- 现在，进行一个简单的变换:

```
auto mul2 = [](int n){ return n * 2; };
auto t1 = steady_clock::now();
std::transform(v.begin(), v.end(), v.begin(), mul2);
dur_t dur1 = steady_clock::now() - t1;
cout << format("no policy: {:.3}ms\n", dur1.count());
```

mul2 lambda 只是将一个值乘以 2。transform() 算法将 mul2 应用于 vector 的每个成员。
此转换不指定执行策略。
输出为:

```
no policy: 4.71ms
```

- 可以在算法的第一个参数中指定执行策略:

```
std::transform(execution::seq,
  v.begin(), v.end(), v.begin(), mul2);
```

seq 策略意味着算法不能并行化，这与没有执行策略一样。
输出为:

```
execution::seq: 4.91ms
```

注意，持续时间与没有策略时大致相同。因为它每次运行都会变化，所以永远不会是精确的。

- execution::par 策略允许算法并行化其工作负载:

```
std::transform(execution::par,
  v.begin(), v.end(), v.begin(), mul2);
```

输出为:

```
execution::par: 3.22ms
```

注意，使用并行执行策略时，算法运行得稍微快一些。

- execute::par_unseq 策略允许工作负载的非排序并行执行:

```
std::transform(execution::par_unseq,
  v.begin(), v.end(), v.begin(), mul2);
```

输出为:

```
execution::par_unseq: 2.93ms
```

这里，我们注意到该策略的另一个性能提升。

execution::par_unseq 策略对算法有更严格的要求，算法不能执行需要并发或需要顺序执行的操作。

How it works…

执行策略接口没有指定如何并行算法工作负载，其设计为在不同的负载和环境下使用不同的硬件和处理器。其可以完全在库中实现，也可以依赖于编译器或硬件支持。

并行化将在运算量超过 O(n) 的算法上表现出最大的改进。例如，sort() 显示了显著的改进。下面是一个没有并行化的 sort():

```
auto t0 = steady_clock::now();
std::sort(v.begin(), v.end());
dur_t dur0 = steady_clock::now() - t0;
cout << format("sort: {:.3}ms\n", dur0.count());
```

输出为:

```
sort: 751ms
```

使用 execution::par，就可以看到显著的性能提升:

```
std::sort(execution::par, v.begin(), v.end());
```

输出为:

```
sort: 163ms
```

execution::par_unseq 的改进会更好:

```
std::sort(execution::par_unseq, v.begin(), v.end());
```

输出为:

```
sort: 152ms
```

在使用并行算法时，做大量的测试是个好主意。若算法或谓词本身不能很好地用于并行化，那么最终可能会获得最小的性能增益，或出现意想不到的副作用。

Note

撰写本文时，GCC 对执行策略的支持很差，LLVM/Clang 也不支持。这个示例是在运行 Windows 10 和 Visual C++ 预览版的 6 核 i7 上测试的。

9.6. 互斥锁和锁——安全地共享数据

术语“互斥”指的是对共享资源的互斥访问。互斥锁通常用于避免由于多个执行线程，访问相同的数据而导致的数据损坏和竞争条件。互斥锁通常使用锁来限制线程的访问。

STL 在 <mutex> 头文件中提供了 mutex 和 lock 类。

How to do it…

这个示例中，我们将使用一个简单的 Animal 类来实验锁定和解锁互斥量:

- 首先，创建一个互斥对象:

```
std::mutex animal_mutex;
```

互斥锁是在全局作用域中声明的，因此相关对象都可以访问它。

- 我们的 Animal 类有一个名字和一个朋友列表:

```
class Animal {
  using friend_t = list<Animal>;
  string_view s_name{ "unk" };
  friend_t l_friends{};
public:
  Animal() = delete;
  Animal(const string_view n) : s_name{n} {}
  ...
}
```

添加和删除好友对互斥是一个有用的测试用例。

- 等式运算符是我们唯一需要的运算符:

```
bool operator==(const Animal& o) const {
  return s_name.data() == o.s_name.data();
}
```

s_name 成员是一个 string_view 对象，因此可以测试其数据存储地址是否相等。

- is_friend() 方法测试是否有其他动物在 l_friends 列表中:

```
bool is_friend(const Animal& o) const {
  for(const auto& a : l_friends) {
    if(a == o) return true;
  }
  return false;
}
```

- find_friend() 方法返回一个可选对象，若找到 Animal，则带有指向 Animal 的迭代器:

```
optional<friend_t::iterator>
find_friend(const Animal& o) noexcept {
  for(auto it{l_friends.begin()};
  it != l_friends.end(); ++it) {
    if(*it == o) return it;
  }
  return {};
}
```

- print() 方法输出 s_name 和 l_friends 列表中每个 Animal 对象的名称:

```
void print() const noexcept {
  auto n_animals{ l_friends.size() };
  cout << format("Animal: {}, friends: ", s_name);
  if(!n_animals) cout << "none";
  else {
    for(auto n : l_friends) {
      cout << n.s_name;
      if(--n_animals) cout << ", ";
    }
  }
  cout << '\n';
}
```

- add_friend() 方法将一个 Animal 对象添加到 l_friends 列表中:

```
bool add_friend(Animal& o) noexcept {
  cout << format("add_friend {} -> {}\n", s_name,
    o.s_name);
  if(*this == o) return false;
  std::lock_guard<std::mutex> l(animal_mutex);
  if(!is_friend(o)) l_friends.emplace_back(o);
  if(!o.is_friend(*this))
    o.l_friends.emplace_back(*this);
  return true;
}
```

- delete_friend() 方法从 l_friends 列表中删除一个 Animal 对象:

```
bool delete_friend(Animal& o) noexcept {
  cout << format("delete_friend {} -> {}\n",
    s_name, o.s_name);
  if(*this == o) return false;
  if(auto it = find_friend(o))
    l_friends.erase(it.value());
  if(auto it = o.find_friend(*this))
    o.l_friends.erase(it.value());
  return true;
}
```

- main() 函数中，我们创建了一些 Animal 对象:

```
int main() {
  auto cat1 = std::make_unique<Animal>("Felix");
  auto tiger1 = std::make_unique<Animal>("Hobbes");
  auto dog1 = std::make_unique<Animal>("Astro");
  auto rabbit1 = std::make_unique<Animal>("Bugs");
  ...
```

- 使用 async() 在对象上调用 add_friends()，在不同的线程中运行:

```
auto a1 = std::async([&]{ cat1->add_friend(*tiger1); });
auto a2 = std::async([&]{ cat1->add_friend(*rabbit1); });
auto a3 = std::async([&]{ rabbit1->add_friend(*dog1); });
auto a4 = std::async([&]{ rabbit1->add_friend(*cat1); });
a1.wait();
a2.wait();
a3.wait();
a4.wait();
```

  可以使用 wait() 对线程进行阻塞式等待。

- 可以使用 print() 来查看动物们和他们之间的关系:

```
auto p1 = std::async([&]{ cat1->print(); });
auto p2 = std::async([&]{ tiger1->print(); });
auto p3 = std::async([&]{ dog1->print(); });
auto p4 = std::async([&]{ rabbit1->print(); });
p1.wait();
p2.wait();
p3.wait();
p4.wait();
```

- 最后，使用 delete_friend() 来删除一个朋友:

```
auto a5 = std::async([&]{ cat1->delete_friend(*rabbit1);
});
a5.wait();
auto p5 = std::async([&]{ cat1->print(); });
auto p6 = std::async([&]{ rabbit1->print(); });
```

- 此时，输出是这样的:

```
add_friend Bugs -> Felix
add_friend Felix -> Hobbes
add_friend Felix -> Bugs
add_friend Bugs -> Astro
Animal: Felix, friends: Bugs, Hobbes
Animal: Hobbes, friends: Animal: Bugs, friends:
FelixAnimal: Astro, friends: Felix
, Astro
Bugs
delete_friend Felix -> Bugs
Animal: Felix, friends: Hobbes
Animal: Bugs, friends: Astro
```

这个输出有点混乱。每次运行都是不同的。有时候可能没问题，但这就可能是问题。我们需要添加一些互斥锁来控制对数据的访问。

- 使用互斥的一种方法是使用它的 lock() 和 unlock() 方法，可以将其添加到 add_friend() 函数中:

```
bool add_friend(Animal& o) noexcept {
  cout << format("add_friend {} -> {}\n", s_name, o.s_
  name);
  if(*this == o) return false;
  animal_mutex.lock();
  if(!is_friend(o)) l_friends.emplace_back(o);
  if(!o.is_friend(*this)) o.l_friends.emplace_
  back(*this);
  animal_mutex.unlock();
  return true;
}
```

lock() 方法尝试获取互斥锁。若互斥锁已经锁定，会等待 (块执行) 互斥锁打开。

- 我们还需要添加一个锁来 delete_friend():

```
bool delete_friend(Animal& o) noexcept {
  cout << format("delete_friend {} -> {}\n",
    s_name, o.s_name);
  if(*this == o) return false;
  animal_mutex.lock();
  if(auto it = find_friend(o))
    l_friends.erase(it.value());
  if(auto it = o.find_friend(*this))
    o.l_friends.erase(it.value());
  animal_mutex.unlock();
  return true;
}
```

- 现在，需要为 print() 添加一个锁，以便在打印时不更改数据:

```
void print() const noexcept {
  animal_mutex.lock();
  auto n_animals{ l_friends.size() };
  cout << format("Animal: {}, friends: ", s_name);
  if(!n_animals) cout << "none";
  else {
    for(auto n : l_friends) {
      cout << n.s_name;
      if(--n_animals) cout << ", ";
    }
  }
  cout << '\n';
  animal_mutex.unlock();
}
```

现在，输出是合理的:

```
add_friend Bugs -> Felix
add_friend Bugs -> Astro
add_friend Felix -> Hobbes
add_friend Felix -> Bugs
Animal: Felix, friends: Bugs, Hobbes
Animal: Hobbes, friends: Felix
Animal: Astro, friends: Bugs
Animal: Bugs, friends: Felix, Astro
delete_friend Felix -> Bugs
Animal: Felix, friends: Hobbes
Animal: Bugs, friends: Astro
```

由于异步操作，输出的行顺序可能不同。

- lock() 和 unlock() 方法很少直接使用。std::lock_guard 类使用适当的资源获取初始化 (RAII) 模式管理锁，该模式在销毁锁时自动释放锁。下面是带 lock_guard 的 add_friend() 方法:

```
bool add_friend(Animal& o) noexcept {
  cout << format("add_friend {} -> {}\n", s_name, o.s_
    name);
  if(*this == o) return false;
  std::lock_guard<std::mutex> l(animal_mutex);
  if(!is_friend(o)) l_friends.emplace_back(o);
  if(!o.is_friend(*this))
    o.l_friends.emplace_back(*this);
  return true;
}
```

lock_guard 对象创建并持有一个锁，直到它销毁。和 lock() 方法一样，lock_guard 也会阻塞至有锁可用为止。

- 让我们对 delete_friend() 和 print() 方法使用 lock_guard。

  delete_friend():

```
bool delete_friend(Animal& o) noexcept {
  cout << format("delete_friend {} -> {}\n",
    s_name, o.s_name);
  if(*this == o) return false;
  std::lock_guard<std::mutex> l(animal_mutex);
  if(auto it = find_friend(o))
    l_friends.erase(it.value());
  if(auto it = o.find_friend(*this))
    o.l_friends.erase(it.value());
  return true;
}
```

  print():

```
void print() const noexcept {
  std::lock_guard<std::mutex> l(animal_mutex);
  auto n_animals{ l_friends.size() };
  cout << format("Animal: {}, friends: ", s_name);
  if(!n_animals) cout << "none";
  else {
    for(auto n : l_friends) {
      cout << n.s_name;
      if(--n_animals) cout << ", ";
    }
  }
  cout << '\n';
}
```

  输出保持一致:

```
add_friend Felix -> Hobbes
add_friend Bugs -> Astro
add_friend Felix -> Bugs
add_friend Bugs -> Felix
Animal: Felix, friends: Bugs, Hobbes
Animal: Astro, friends: Bugs
Animal: Hobbes, friends: Felix
Animal: Bugs, friends: Astro, Felix
delete_friend Felix -> Bugs
Animal: Felix, friends: Hobbes
Animal: Bugs, friends: Astro
```

与前面一样，由于异步操作，输出的行顺序可能不同。

How it works…

互斥锁并不锁定数据，理解这一点很重要，它阻碍了执行。如本文所示，在对象方法中应用互斥锁时，可以使用它强制对数据进行互斥访问。

当一个线程用 lock() 或 lock_guard 锁住一个互斥量时，就说这个线程拥有这个互斥量。其他试图锁定同一个互斥锁的线程都将阻塞，直到锁的所有者将其解锁。

互斥对象在被任何线程拥有时不能被销毁。同样，当线程拥有互斥时，不能销毁。与 RAII 兼容的包装器，比如 lock_guard，将确保这种情况不会发生。

There's more…

虽然，std::mutex 提供了一个适用于许多排他性互斥锁，但 STL 确实提供了一些其他选择:

- shared_mutex 允许多个线程同时拥有一个互斥量。
- recursive_mutex 允许一个线程在一个互斥锁上叠加多个锁。
- timed_mutex 为互斥锁提供超时。shared_mutex 和 recursive_mutex 也有定时版本可用。

9.7. std::atomic——共享标志和值

std::atomic 类封装了单个对象，并保证对其的操作是原子的。

写入原子对象由内存顺序策略控制，读取可能同时发生。通常用于同步不同线程之间的访问，std::atomic 从模板类型定义原子类型。类型必须普通。

若类型占用连续的内存，没有用户定义的构造函数，也没有虚成员函数，那就是普通类型。所有的基本类型都是普通类型。

虽然可以构造普通类型，但 std::atomic 最常用于简单的基本类型，如 bool、int、long、float 和 double。

How to do it…

这个示例使用一个简单的函数，循环计数器来演示共享原子对象。我们将生成一群这样的循环，作为共享原子值的线程:

- 原子对象通常放置在全局命名空间中，可以让线程访问:

```
std::atomic<bool> ready{};
std::atomic<uint64_t> g_count{};
std::atomic_flag winner{};
```

ready 对象是一个 bool 类型，当所有线程都准备好开始计数时，该类型设置为 true。

g_count 对象是一个全局计数器，由每个线程进行递增。

winner 对象是一个特殊的 atomic_flag 类型，用于指示哪个线程先完成。

- 我们使用几个常量来控制线程数和每个线程的循环数:

```
constexpr int max_count{1000 * 1000};
constexpr int max_threads{100};
```

我将它设置为运行 100 个线程，并在每个线程中计算 1,000,000 次迭代。

- countem() 函数为每个线程生成，循环 max_count 次，并在每次循环迭代时增加 g_count。这是使用原子值的地方:

```
void countem (int id) {
  while(!ready) std::this_thread::yield();
  for(int i{}; i < max_count; ++i) ++g_count;
  if(!winner.test_and_set()) {
    std::cout << format("thread {:02} won!\n",
    id);
  }
};
```

ready 原子值用于同步线程。每个线程将调用 yield()，直到就绪值设置为 true。yield() 函数的作用是: 将执行传递给其他线程。

for 循环的每次迭代都会增加 g_count 原子值，最终值应该等于 max_count * max_threads。

循环完成后，获胜者对象的 test_and_set() 方法用于报告获胜线程。test_and_set() 是 atomic_flag 类的一个方法，设置标志并返回设置之前的 bool 值。

- 我们以前使用过 make_comma() 函数，可以显示一个带有数千个分隔符的数字:

```
string make_commas(const uint64_t& num) {
  string s{ std::to_string(num) };
  for(long l = s.length() - 3; l > 0; l -= 3) {
    s.insert(l, ",");
  }
  return s;
}
```

- main() 函数生成线程并显示结果:

```
int main() {
  vector<std::thread> swarm;
  cout << format("spawn {} threads\n", max_threads);
  for(int i{}; i < max_threads; ++i) {
    swarm.emplace_back(countem, i);
  }
  ready = true;
  for(auto& t : swarm) t.join();
  cout << format("global count: {}\n",
    make_commas(g_count));
  return 0;
}
```

这里，创建一个 vector<std::thread> 对象来保存线程。for 循环中，使用 emplace_back() 在 vector 中创建线程。线程生成后，设置 ready 标志，以便线程开始循环。

输出为:

```
spawn 100 threads
thread 67 won!
global count: 100,000,000
```

每次运行时，都会有不同的线程获胜。

How it works…

std::atomic 类封装了一个对象来同步多个线程之间的访问。

封装的对象必须是普通类型，必须使用连续的内存，没有用户定义的构造函数，也没有虚成员函数。所有的基本类型都是普通类型。

atomic 可以使用一个简单的结构体:

```
struct Trivial {
  int a;
  int b;
};
std::atomic<Trivial> triv1;
```

虽然这种用法是可能的，但并不实际。除了设置和检索复合值之外，并没有体现原子性的价值，最终需要一个互斥锁。并且，原子类最适合用于标量值。

特化

原子类的特化有以下几个不同的目的:

- 指针和智能指针:std::atomic<U*> 特化包括对原子指针算术操作的支持，包括用于加法的 fetch_add() 和用于减法的 fetch_sub()。
- 浮点类型: 当与浮点类型 float、double 和 long double 一起使用时，std::atomic 包括对原子浮点算术操作的支持，包括用于加法的 fetch_add() 和用于减法的 fetch_sub()。
- 整型类型: 当与整型类型一起使用时，std::atomic 提供了对其他原子操作的支持，包括 fetch_add()、fetch_sub()、fetch_and()、fetch_or() 和 fetch_xor()。

标准的别名

STL 为所有标准标量整型提供类型别名。所以在代码中，不需要这些声明:

```
std::atomic<bool> ready{};
std::atomic<uint64_t> g_count{};
```

我们可以用:

```
std::atomic_bool ready{};
std::atomic_uint64_t g_count{};
```

有 46 个标准的别名，每一个代表标准的整型:

atomic_bool	atomic_uint64_t
atomic_char	atomic_int_least8_t
atomic_schar	atomic_uint_least8_t
atomic_uchar	atomic_int_least16_t
atomic_short	atomic_uint_least16_t
atomic_ushort	atomic_int_least32_t
atomic_int	atomic_uint_least32_t
atomic_uint	atomic_int_least64_t
atomic_long	atomic_uint_least64_t
atomic_ulong	atomic_int_fast8_t
atomic_llong	atomic_uint_fast8_t
atomic_ullong	atomic_int_fast16_t
atomic_char8_t	atomic_uint_fast16_t
atomic_char16_t	atomic_int_fast32_t
atomic_char32_t	atomic_uint_fast32_t
atomic_wchar_t	atomic_int_fast64_t
atomic_int8_t	atomic_uint_fast64_t
atomic_uint8_t	atomic_intptr_t
atomic_int16_t	atomic_uintptr_t
atomic_uint16_t	atomic_size_t
atomic_int32_t	atomic_ptrdiff_t
atomic_uint32_t	atomic_intmax_t
atomic_int64_t	atomice_uintmax_t

无锁版本

大多数现代体系结构为执行原子操作提供了原子 CPU 指令，原子指令应该在硬件支持的情况下使用硬件支持。某些硬件可能不支持某些原子类型，std::atomic 可以使用互斥来确保那些特化的线程安全操作，导致线程在等待其他线程完成操作时阻塞。使用硬件支持的特化因为它们不需要互斥锁，所以称为无锁式方法。

is_lock_free() 方法检查特化是否无锁:

```
cout << format("is g_count lock-free? {}\n",
  g_count.is_lock_free());
```

输出为:

```
is g_count lock-free? true
```

这个结果对于大多数现代架构都可以正常处理。

有一些保证 std::atomic 的无锁版本可用。这些特化保证为每个目的使用最有效的硬件原子操作:

- std::atomic_signed_lock_free 有符号整型最有效的无锁特化的别名。
- std::atomic_unsigned_lock_free 是无符号整型最有效的无锁特化的别名。
- The std::atomic_flag 类提供了一个无锁原子布尔类型。

重要的 Note

当前 Windows 系统不支持 64 位硬件整数，即使在 64 位系统上也是如此。当在我的实验室中的一个系统上测试这段代码时，将 std::atomic<uint64_t> 替换为 std::atomic_unsigned_lock_free，性能提高了 3 倍。在 64 位 Linux 和 Mac 系统上，性能没什么变化。

There's more…

当多个线程同时读写变量时，线程可能以不同于写入变量的顺序观察变化。memory_order 指定内存访问如何围绕原子操作排序。

std::atomic 提供了访问和更改其托管值的方法。与关联操作符不同，这些访问方法为要指定的 memory_order 提供参数。例如:

```
g_count.fetch_add(1, std::memory_order_seq_cst);
```

这种情况下，memory_order_seq_cst 指定顺序一致的排序。因此，对 fetch_add() 的调用将按顺序一致地将 g_count 的值加 1。

可能的 memory_order 常量是:

- memory_order_relaxed: 这是一个松散的操作。没有同步或排序约束，只有操作的原子性得到保证。
- memory_order_consume: 这是一个消费型操作。在此加载之前，不能对依赖于该值的当前线程中的访问进行重排序。这只会影响编译器优化。
- memory_order_acquire: 这是一个获取型操作。在此加载之前，访问不能重新排序。
- memory_order_release: 这是一个存储型操作。当前线程中的访问不能在此存储之后重新排序。
- memory_order_acq_rel: 这既是获取型，也是释放型。当前线程中的访问不能在此存储之前或之后重新排序。
- memory_order_seq_cst: 这是顺序一致的排序，根据上下文获取或释放。加载执行获取，存储执行释放，读/写/修改同时执行。所有线程以相同的顺序观察所有修改。

  若没有指定 memory_order，则 memory_order_seq_cst 为默认值。

9.8. std::call_once——初始化线程

可能需要在许多线程中运行相同的代码，但只能初始化该代码一次。

一种解决方案是在运行线程之前调用初始化代码。这种方法可以工作，但有一些缺点。通过分离初始化，可以在不必要的时候调用它，也可以在必要的时候忽略它。

std::call_once 函数提供了一个更健壮的解决方案。call_once 在 <mutex> 头文件中声明。

How to do it…

本节中，我们使用 print 函数进行初始化，所以可以清楚地看到函数什么时候调用:

- 我们可以使用一个常量来表示要生成的线程数:

```
constexpr size_t max_threads{ 25 };
```

还需要一个 std::once_flag 来同步 std::call_once 函数:

```
std::once_flag init_flag;
```

- 初始化函数只是打印一个字符串，让我们知道其调用了:

```
void do_init(size_t id) {
  cout << format("do_init ({}): ", id);
}
```

- 我们的工作函数 do_print() 使用 std::call_once 来调用初始化函数，然后打印其 id:

```
void do_print(size_t id) {
  std::call_once(init_flag, do_init, id);
  cout << format("{} ", id);
}
```

- 在 main() 中，使用列表容器来管理线程对象:

```
int main() {
  list<thread> spawn;
  for (size_t id{}; id < max_threads; ++id) {
    spawn.emplace_back(do_print, id);
  }
  for (auto& t : spawn) t.join();
  cout << '\n';
}
```

输出显示初始化首先发生，并且只发生一次:

```
do_init (8): 12 0 2 1 9 6 13 10 11 5 16 3 4 17 7 15 8 14
18 19 20 21 22 23 24
```

注意，最终调用初始化函数的并不总是第一个衍生线程 (0)，但总是第一个调用。若重复运行这个命令，会看到线程 0 得到初始化，但不是每次都这样。在内核较少的系统中，初始化时更经常看到线程 0。

How it works…

std::call_once 是一个模板函数，它接受一个标志、一个可调用对象 (函数或函子) 和一个参数形参包:

```
template<class Callable, class... Args>
void call_once(once_flag& flag, Callable&& f, Args&&... args);
```

可调用的 f 只调用了一次。即使 call_once 在多个线程中并发使用，f 仍然只调用了一次。

这需要一个 std::once_flag 对象进行协调，once_flag 构造函数将其状态设置为指示可调用对象尚未调用。

当 call_once 调用可调用对象时，对同一个 once_flag 的其他调用都将阻塞，直到可调用对象返回。可调用对象返回后，可以设置 once_flag，从而使后面的 call_once 直接返回，而不在调用 f。

9.9. std::condition_variable——解决生产者-消费者问题

生产者-消费者问题的最简单版本是，一个进程生产数据，另一个进程消费数据，使用一个缓冲区或容器保存数据。这需要生产者和消费者之间的协调来管理缓冲区并防止不必要的副作用。

How to do it…

本节中，我们考虑了一个简单的解决方案，使用 std::condition_variable 来协调这个过程:

- 方便起见，我们从一些命名空间和别名声明开始说起:

```
using namespace std::chrono_literals;
namespace this_thread = std::this_thread;
using guard_t = std::lock_guard<std::mutex>;
using lock_t = std::unique_lock<std::mutex>;
```

lock_guard 和 unique_lock 的别名，可以更容易的使用这些类型而不会出错。

- 有几个常量:

```
constexpr size_t num_items{ 10 };
constexpr auto delay_time{ 200ms };
```

可以把它们放在一个地方可以更安全、更容易地试验不同的值。

- 可以使用这些全局变量来协调数据存储:

```
std::deque<size_t> q{};
std::mutex mtx{};
std::condition_variable cond{};
bool finished{};
```

可以使用 deque 将数据保存在先进先出 (FIFO) 队列中。

mutex 与 condition_variable 一起使用，以协调数据从生产者到消费者的转移。

finished 标志表示没有更多数据。

- 生产者线程将使用这个函数:

```
void producer() {
  for(size_t i{}; i < num_items; ++i) {
    this_thread::sleep_for(delay_time);
    guard_t x{ mtx };
    q.push_back(i);
    cond.notify_all();
  }
  guard_t x{ mtx };
```

```
  finished = true;
  cond.notify_all();
}
```

producer() 函数循环 num_items 迭代，并在每次循环中将一个数字压入 deque。

这里，包括使用 sleep_for() 来模拟产生每个值的延迟。

conditional_variable 需要一个互斥锁来操作，可以使用 lock_guard(通过 guard_t 别名) 来获取锁，然后将值推到 deque 上，然后在 conditional_variable 上调用 notify_all()。这将告诉使用者线程有一个可用的新值。

当循环完成时，可以设置 finished 标志，并通知消费者线程生产者已经完成。

- 消费者线程等待来自生产者的每个值，将其显示在控制台上，并等待 finished 标志:

```
void consumer() {
  while(!finished) {
    lock_t lck{ mtx };
    cond.wait(lck, [] { return !q.empty() ||
      finished; });
    while(!q.empty()) {
      cout << format("Got {} from the queue\n",
        q.front());
      q.pop_front();
    }
  }
}
```

wait() 方法等待生产者通知，使用 lambda 作为谓词继续等待，直到 deque 不为空或设置了 finished 标志。

当我们获取到一个值时显示它，然后将其从 deque 中弹出。

- 在 main() 中运行简单的线程对象:

```
int main() {
  thread t1{ producer };
  thread t2{ consumer };
  t1.join();
  t2.join();
  cout << "finished!\n";
}
```

输出为:

```
Got 0 from the queue
Got 1 from the queue
Got 2 from the queue
Got 3 from the queue
Got 4 from the queue
Got 5 from the queue
Got 6 from the queue
Got 7 from the queue
Got 8 from the queue
Got 9 from the queue
finished!
```

注意，每一行之间有 200 毫秒的延迟，所以生产者和消费者之间的协调正在按照预期的方式工作。

How it works…

生产者-消费者问题需要在写入和读取缓冲区或容器之间进行协调。在这个例子中，容器是 deque<size_t>:

```
std::deque<size_t> q{};
```

当共享变量修改时，conditional_variable 类可以阻塞一个线程或多个线程。然后，通知其他线程该值可用。

condition_variable 需要 mutex 来执行锁定操作:

```
std::lock_guard x{ mtx };
q.push_back(i);
cond.notify_all();
```

std::lock_guard 会获取锁，从而可以将一个值推入到 deque 中。

condition_variable 上的 wait() 方法用于阻塞当前线程，直到收到通知:

```
void wait( std::unique_lock<std::mutex>& lock );
void wait( std::unique_lock<std::mutex>& lock,
  Pred stop_waiting );
```

wait() 的谓词形式相当于:

```
while (!stop_waiting()) {
  wait(lock);
}
```

谓词形式用于防止在等待特定条件时出现伪唤醒。我们的例子中，将和 lambda 一起使用:

```
cond.wait(lck, []{ return !q.empty() || finished; });
```

这可以防止消费者在 deque 有数据或 finished 标志设置之前醒来。

condition_variable 类有两个通知方法:

- notify_one() 解除一个等待线程的阻塞
- notify_all() 解除所有等待线程的阻塞

示例中使用了 notify_all()。因为只有一个使用者线程，所以任何一种通知方法的工作方式都是一样的。

> **Note**
>
> 注意，unique_lock 是唯一在 condition_variable 对象上支持 wait() 的锁。

9.10. 实现多个生产者和消费者

生产者-消费者问题实际上是一组问题。若缓冲区是有界的或无界的，或者有多个生产者、多个消费者，或者两者都有，解决方案将有所不同。

来考虑一个有多个生产者、多个消费者和一个有限 (容量有限) 缓冲区的情况。

How to do it…

在这个示例中，我们将看到一个有多个生产者和消费者的情况，以及一个有界缓冲区，并使用本章中介绍的各种技术:

- 为了方便和可靠性，先从一些常量开始:

```
constexpr auto delay_time{ 50ms };
constexpr auto consumer_wait{ 100ms };
constexpr size_t queue_limit{ 5 };
constexpr size_t num_items{ 15 };
constexpr size_t num_producers{ 3 };
constexpr size_t num_consumers{ 5 };
```

  - delay_time 是一个 duration 对象，与 sleep_for() 一起使用。
  - consumer_wait 是一个持续时间对象，与消费者条件变量一起使用。
  - queue_limit 是缓冲区限制——deque 中的最大容量值。
  - num_items 是每个生产者生产的最大产品目数。
  - num_producers 是消费者的数量。

- 需要一些对象来控制这个过程:

```
deque<string> qs{};
mutex q_mutex{};
condition_variable cv_producer{};
condition_variable cv_consumer{};
bool production_complete{};
```

  - qs 是一个字符串 deque，用于保存生成的对象。

- q_mutex 可以对 deque 的访问进行控制。
- queue_limt 是缓冲区限制——deque 中项目的最大容量。
- cv_producer 是协调生产者的条件变量。
- cv_consumer 是协调使用者的条件变量。
- production_complete 当所有生产者线程完成时，将其设置为 true。

- producer() 线程运行这个函数:

```
void producer(const size_t id) {
  for(size_t i{}; i < num_items; ++i) {
    this_thread::sleep_for(delay_time * id);
    unique_lock<mutex> lock(q_mutex);
    cv_producer.wait(lock,
      [&]{ return qs.size() < queue_limit; });
    qs.push_back(format("pid {}, qs {},
      item {:02}\n", id, qs.size(), i + 1));
    cv_consumer.notify_all();
  }
}
```

传递值的 id0 用于标识生产者的连续数字。

主 for 循环重复 num_item 次。sleep_for() 函数用于模拟生成一个项所需的一些工作。

然后，从 q_mutex 中获得一个唯一的_lock，并在 cv_producer 上调用 wait()，使用 lambda 根据 queue_limit 常量检查 deque 的大小。若 deque 已经达到上限，生产者等待消费者线程来减小 deque 的体积。这表示生成器的有界缓冲区限制。

当条件满足，可以将一个产品推入 deque。元素是一个格式化的字符串，包含生产者的 id、qs 的大小和来自循环控制变量的项目编号 (i + 1)。

最后，在 cv_consumer 条件变量上使用 notify_all() 通知消费者有新数据可用。

- consumer() 线程运行这个函数:

```
void consumer(const size_t id) {
  while(!production_complete) {
    unique_lock<mutex> lock(q_mutex);
    cv_consumer.wait_for(lock, consumer_wait,
      [&]{ return !qs.empty(); });
    if(!qs.empty()){
      cout << format("cid {}: {}", id,
        qs.front());
      qs.pop_front();
    }
    cv_producer.notify_all();
  }
}
```

传递的 id 值是用于标识使用者的连续数字。

主 while() 循环继续进行，直到对 production_complete 进行设置。

我们从 q_mutex 中获得 unique_lock，并在 cv_consumer 上调用 wait_for()，使用一个超时和一个 lambda 来测试 deque 是否为空。这里需要超时，因为当一些消费者线程仍在运行时，生产者线程可能已经完成，可使 deque 为空。

当有了一个非空的 deque，就可以打印 (消费) 一个产品信息，并将其从 deque 中弹出。

- 在 main() 中，使用 async() 来生成生产者线程和消费者线程。async() 符合 RAII 模式，所以我通常更喜欢使用 async。async() 返回一个 future 对象，可以保留一个 future<void> 对象列表用于进程管理:

```
int main() {
  list<future<void>> producers;
  list<future<void>> consumers;
  for(size_t i{}; i < num_producers; ++i) {
    producers.emplace_back(async(producer, i));
  }
  for(size_t i{}; i < num_consumers; ++i) {
    consumers.emplace_back(async(consumer, i));
  }
  ...
```

使用 for 循环来创建生产者和消费者线程。

- 最后，可以使用 future 对象的列表来确定生产者和消费者线程何时完成:

```
for(auto& f : producers) f.wait();
production_complete = true;
cout << "producers done.\n";

for(auto& f : consumers) f.wait();
cout << "consumers done.\n";
```

循环生成器容器，调用 wait() 以允许生成器线程完成。然后，可以设置 production_complete 标志。同样，循环遍历消费者容器，调用 wait() 以允许消费者线程完成。并且，可以在这里执行最终的分析或完成过程。

- 输出有点长，就不全部展示了:

```
cid 0: pid 0, qs 0, item 01
cid 0: pid 0, qs 1, item 02
cid 0: pid 0, qs 2, item 03
cid 0: pid 0, qs 3, item 04
cid 0: pid 0, qs 4, item 05
...
cid 4: pid 2, qs 0, item 12
cid 4: pid 2, qs 0, item 13
cid 3: pid 2, qs 0, item 14
cid 0: pid 2, qs 0, item 15
producers done.
consumers done.
```

How it works…

这个配方的核心是使用两个 condition_variable 对象来异步控制生产者和消费者线程:

```
condition_variable cv_producer{};
condition_variable cv_consumer{};
```

在 producer() 函数中，cv_producer 对象获得 unique_lock，等待 deque 可用，并在生成产品时通知 cv_consumer 对象:

```
void producer(const size_t id) {
  for(size_t i{}; i < num_items; ++i) {
    this_thread::sleep_for(delay_time * id);
    unique_lock<mutex> lock(q_mutex);
    cv_producer.wait(lock,
      [&]{ return qs.size() < queue_limit; });
    qs.push_back(format("pid {}, qs {}, item {:02}\n",
      id, qs.size(), i + 1));
    cv_consumer.notify_all();
  }
}
```

相反，在 consumer() 函数中，cv_consumer 对象获得 unqiue_lock，等待 deque 中有产品，并在产品消费时通知 cv_producer 对象:

```
void consumer(const size_t id) {
  while(!production_complete) {
    unique_lock<mutex> lock(q_mutex);
    cv_consumer.wait_for(lock, consumer_wait,
      [&]{ return !qs.empty(); });
    if(!qs.empty()) {
      cout << format("cid {}: {}", id, qs.front());
```

```
8         qs.pop_front();
9       }
10      cv_producer.notify_all();
11    }
12  }
```

这些锁、等待和通知，构成了多个生产者和消费者之间的动态平衡。

第 10 章 文件系统

STL 文件系统库的目的是标准化跨平台的文件系统操作。文件系统库寻求规范化操作，而非 POSIX/Unix、Windows 和其他文件系统那样的不标准桥接。

文件系统库采用了相应的 Boost 库，并与 C++17 集成到 STL 中。撰写本文时，在一些系统上的实现仍然存在差距，但本章中的实力已经在 Linux、Windows 和 macOS 文件系统上进行了测试，并分别使用 GCC、MSVC 和 Clang 编译器的最新可用版本进行了编译。

标准库使用 <filesystem> 头文件，std::filesystem 命名空间通常别名为 fs:

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

fs::path 类是文件系统库的核心，在不同的环境中提供了规范化的文件名和目录路径表示。路径对象可以表示文件、目录或对象中的对象，甚至是不存在或不可能的对象。

我们将介绍使用文件系统库处理文件和目录的工具:

- 为 path 类特化 std::formatter
- 使用带有路径的操作函数
- 列出目录中的文件
- 使用 grep 实用程序搜索目录和文件
- 使用 regex 和 directory_iterator 重命名文件
- 创建磁盘使用计数器

10.1. 相关准备

本章所有的例子和源代码都可以在本书的 GitHub 库中找到，网址是https://github.com/PacktPublishing/CPP-20-STL-Cookbook/tree/main/chap10。

10.2. 为 path 类特化 std::formatter

path 类在整个文件系统库中用于表示文件或目录路径。在符合 posix 的系统上，例如 macOS 和 Linux，路径对象使用 char 类型来表示文件名。在 Windows 上，path 使用 wchar_t。在 Windows 上，cout 和 format() 不会显示 wchar_t 字符的原始字符串，所以没有简单的方法来编写使用文件系统库，并且无法直接 POSIX 和 Windows 之间编写可移植的代码。

我们需要使用预处理器指令为 Windows 编写特定版本的代码。对于某些代码库来说，这可能是一个合理的解决方案，但对于本书来说，这样写出的代码很丑陋，不能满足简单、可移植、可重用的特性。

优雅的解决方案是为 path 类使用 C++20 的 format 进行格式化特化。这就可以简单地、可移植地显示 path 对象。

How to do it…

这个示例中，我们编写了一个格式化特化，用于 fs::path 类:

- 从命名空间别名开始，所有的文件系统名称都在 std::filesystem 命名空间中:

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

- path 类的格式化特化简单而简洁:

```
template<>
struct std::formatter<fs::path>:
std::formatter<std::string> {
  template<typename FormatContext>
  auto format(const fs::path& p, FormatContext& ctx) {
    return format_to(ctx.out(), "{}", p.string());
  }
};
```

这里，专为 fs::path 类型设置了格式，使用其 string() 方法来获得可输出的字符表示。我们不能使用 c_str() 方法，因其无法在 Windows 上处理 wchar_t 字符。

本书的第 1 章中有关于格式化特化的更完整的解释。

- main() 函数中，可以使用命令行传递文件名或路径:

```
int main(const int argc, const char** argv) {
  if(argc != 2) {
    fs::path fn{ argv[0] };
    cout << format("usage: {} <path>\n",
    fn.filename());
    return 0;
  }

  fs::path dir{ argv[1] };
  if(!fs::exists(dir)) {
    cout << format("path: {} does not exist\n",
    dir);
    return 1;
  }

  cout << format("path: {}\n", dir);
  cout << format("filename: {}\n", dir.filename());
  cout << format("cannonical: {}\n",
  fs::canonical(dir));
}
```

argc 和 argv 参数是标准的命令行参数。

argv[0] 始终是可执行文件本身的完整目录路径和文件名，若没有正确数量的参数，则显示 argv[0] 的文件名部分作为使用消息的一部分。

这个例子中，使用了一些文件系统函数:

- fs::exists() 函数的作用是: 检查目录或文件是否存在。
- dir 是一个 path 对象。可以直接将 path 传递给 format()，使用特化来显示 path 的字符串表示形式。
- filename() 方法返回一个新的 path 对象，直接将其传递给 format()。

– fs::cannonical() 函数的作用是: 接受一个 path 对象，并返回一个带有规范绝对目录路径的新 path 对象。我们将这个 path 对象直接传递给 format()，然后就会从 cannical() 返回的目录路径，并进行显示。

输出为:

```
$ ./formatter ./formatter.cpp
path: ./formatter.cpp
filename: formatter.cpp
cannonical: /home/billw/working/chap10/formatter.cpp
```

How it works…

fs::path 类在整个文件系统库中用于表示目录路径和文件名。通过提供格式化特化，可以轻松地跨平台显示 path 对象。

path 类提供了一些有用的方法，可以遍历一个 path 对象:

```
fs::path p{ "~/include/bwprint.h" };
cout << format("{}\n", p);
for(auto& x : p) cout << format("[{}] ", x);
cout << '\n';
```

输出为:

```
~/include/bwprint.h
[~] [include] [bwprint.h]
```

迭代器为路径的每个元素返回一个 path 对象。

也可以得到路径的不同部分:

```
fs::path p{ "~/include/bwprint.h" };
cout << format("{}\n", p);
cout << format("{}\n", p.stem());
cout << format("{}\n", p.extension());
cout << format("{}\n", p.filename());
cout << format("{}\n", p.parent_path());
```

输出为:

```
~/include/bwprint.h
bwprint
.h
bwprint.h
~/include
```

在本章中，我们将继续为 path 类使用这个格式化特化。

10.3. 使用带有路径的操作函数

文件系统库包括用于操作 path 对象内容的函数。本节中，我们将了解其中的一些工具。

How to do it…

这个示例中，我们检查了一些操作 path 对象内容的函数:

- 从命名空间指令和格式化特化开始。本章的每个示例中都会这样做:

```
namespace fs = std::filesystem;
template<>
struct std::formatter<fs::path>:
std::formatter<std::string> {
  template<typename FormatContext>
  auto format(const fs::path& p, FormatContext& ctx) {
    return format_to(ctx.out(), "{}", p.string());
  }
};
```

- 可以使用 current_path() 函数获取当前的工作目录，该函数返回一个 path 对象:

```
cout << format("current_path: {}\n", fs::current_path());
```

输出为:

```
current_path: /home/billw/chap10
```

- absolute() 函数的作用是: 从相对路径返回绝对路径:

```
cout << format("absolute(p): {}\n", fs::absolute(p));
```

输出为:

```
absolute(p): /home/billw/chap10/testdir/foo.txt
```

absolute() 也会解除对符号链接的引用。

- /= 操作符将一个字符串追加到路径字符串的末尾，并返回一个新的路径对象:

```
cout << format("append: {}\n",
  fs::path{ "testdir" } /= "foo.txt");
```

输出为:

```
append: testdir/foo.txt
```

- canonical() 函数返回完整的规范目录路径:

```
cout << format("canonical: {}\n",
  fs::canonical(fs::path{ "." } /= "testdir"));
```

输出为:

```
canonical: /home/billw/chap
```

- equivalent() 函数测试两个相对路径，看是否解析到相同的文件系统示例:

```
cout << format("equivalent: {}\n",
  fs::equivalent("testdir/foo.txt",
    "testdir/../testdir/foo.txt"));
```

输出为:

```
equivalent: true
```

- 文件系统库包含了 filesystem_error 类用于异常处理:

```
try {
  fs::path p{ fp };
  cout << format("p: {}\n", p);
  ...
  cout << format("equivalent: {}\n",
  fs::equivalent("testdir/foo.txt",
  "testdir/../testdir/foo.txt"));
} catch (const fs::filesystem_error& e) {
  cout << format("{}\n", e.what());
  cout << format("path1: {}\n", e.path1());
  cout << format("path2: {}\n", e.path2());
}
```

filesystem_error 类包括用于显示错误消息和获取错误涉及的 path 方法。

若在 equivalent() 中引入一个错误，可以看到 fileystem_error:

```
cout << format("equivalent: {}\n",
  fs::equivalent("testdir/foo.txt/x",
    "testdir/../testdir/foo.txt/y"));
```

输出为:

```
filesystem error: cannot check file equivalence: No
such file or directory [testdir/foo.txt/x] [testdir/../
testdir/foo.txt/y]
path1: testdir/foo.txt/x
path2: testdir/../testdir/foo.txt/y
```

这是使用 GCC 在 Debian 上的输出。

filesystem_error 类通过其 path1() 和 path2() 方法提供了更多的细节，这些方法可以返回 path 对象。

- 也可以将 std::error_code 用于文件系统函数:

```
fs::path p{ fp };
std::error_code e;
cout << format("canonical: {}\n",
  fs::canonical(p /= "foo", e));
cout << format("error: {}\n", e.message());
```

输出为:

```
canonical:
error: Not a directory
```

- 即使 Windows 使用了不同的文件系统，这段代码仍会按照预期工作，使用 Windows 文件命名约定:

```
p: testdir/foo.txt
current_path: C:\Users\billw\chap10
absolute(p): C:\Users\billw\chap10\testdir\foo.txt
concatenate: testdirfoo.txt
append: testdir\foo.txt
canonical: C:\Users\billw\chap10\testdir
equivalent: true
```

How it works…

这些函数大多数接受一个 path 对象，一个可选的 std::error_code 对象，并返回一个 path 对象:

```
path absolute(const path& p);
path absolute(const path& p, std::error_code& ec);
```

equivalent() 函数接受两个 path 对象，并返回 bool 类型:

```
bool equivalent( const path& p1, const path& p2 );
bool equivalent( const path& p1, const path& p2,
  std::error_code& ec );
```

path 类具有连接和追加操作符。这两个算子都是破坏性的，会修改运算符左边的 path:

```
p1 += source; // concatenate
p1 /= source; // append
```

对于右边，这些操作符接受一个路径对象、一个字符串、一个 string_view、一个 C-string 或一对迭代器。

连接操作符将操作符右侧的字符串，添加到 p1 路径字符串的末尾。

append 操作符添加分隔符 (例如，/或\)，后面跟着从操作符右侧到路径字符串 p1 末尾的字符串。

10.4. 列出目录中的文件

文件系统库提供了一个 directory_entry 类，其中包含关于给定路径的目录相关信息，可以使用它来创建有用的目录列表。

How to do it…

在这个示例中，使用 directory_entry 类中的信息创建目录列表:

- 从显示路径对象的命名空间别名和格式化特化开始:

```
namespace fs = std::filesystem;
template<>
struct std::formatter<fs::path>:
std::formatter<std::string> {
  template<typename FormatContext>
  auto format(const fs::path& p, FormatContext& ctx) {
    return format_to(ctx.out(), "{}", p.string());
  }
};
```

- directory_iterator 类可以方便地列出目录:

```
int main() {
  constexpr const char* fn{ "." };
  const fs::path fp{fn};
  for(const auto& de : fs::directory_iterator{fp}) {
    cout << format("{} ", de.path().filename());
  }
  cout << '\n';
}
```

输出为:

```
chrono Makefile include chrono.cpp working formatter
testdir formatter.cpp working.cpp
```

- 可以添加命令行选项来实现这个功能，比如 Unix 的 ls:

```
int main(const int argc, const char** argv) {
  fs::path fp{ argc > 1 ? argv[1] : "." };
  if(!fs::exists(fp)) {
    const auto cmdname {
      fs::path{argv[0]}.filename() };
    cout << format("{}: {} does not exist\n",
      cmdname, fp);
    return 1;
  }
  if(is_directory(fp)) {
    for(const auto& de :
    fs::directory_iterator{fp}) {
      cout << format("{} ",
      de.path().filename());
    }
  } else {
    cout << format("{} ", fp.filename());
  }
  cout << '\n';
}
```

若有命令行参数，可以用它来创建一个 path 对象。否则，需要使用”.” 表示当前目录。

我们使用 if_exists() 检查路径是否存在。若不存在，则打印错误消息并退出。错误信息包括来自 argv[0] 的 cmdname。

接下来，我们检查 is_directory()。若有一个目录，可对每个条目循环使用 directory_iterator。directory_iterator 遍历 directory_entry 对象。de.path().filename() 会从每个 directory_entry 对象获取路径和文件名。

输出为:

```
$ ./working
chrono Makefile include chrono.cpp working formatter
testdir formatter.cpp working.cpp
$ ./working working.cpp
working.cpp
$ ./working foo.bar
working: foo.bar does not exist
```

- 若要对输出进行排序，可以将 directory_entry 对象存储在可排序容器中。
  文件的顶部，为 fs::directory_entry 创建一个别名:

```
using de = fs::directory_entry;
```

在 main() 的顶部，声明了一个 de 的 vector:

```
vector<de> entries{};
```

在 is_directory() 块中，我们加载 vector 对它排序，并进行显示:

```
if(is_directory(fp)) {
  for(const auto& de : fs::directory_iterator{fp}) {
    entries.emplace_back(de);
  }
  std::sort(entries.begin(), entries.end());
  for(const auto& e : entries) {
    cout << format("{} ", e.path().filename());
  }
} else { ...
```

现在输出已排序 vector:

```
Makefile chrono chrono.cpp formatter formatter.cpp
include testdir working working.cpp
```

注意，Makefile 首先排序，显然是无序的。这是因为大写字母在小写字母之前按 ASCII 顺序排序。

- 若想要一个不区分大小写的排序，需要一个忽略大小写的比较函数。首先，需要一个函数返回小写字母的字符串:

```
string strlower(string s) {
  auto char_lower = [](const char& c) -> char {
    if(c >= 'A' && c <= 'Z') return c + ('a' - 'A');
    else return c;
  };
  std::transform(s.begin(), s.end(), s.begin(),
    char_lower);
  return s;
}
```

现在，需要一个函数来比较两个 directory_entry 对象，使用 strlower():

```
bool dircmp_lc(const de& lhs, const de& rhs) {
  const auto lhstr{ lhs.path().string() };
  const auto rhstr{ rhs.path().string() };
  return strlower(lhstr) < strlower(rhstr);
}
```

现在可以在排序中使用 dircmp_lc():

```
std::sort(entries.begin(), entries.end(), dircmp_lc);
```

输出现在排序，并忽略大小写:

```
chrono chrono.cpp formatter formatter.cpp include
Makefile testdir working working.cpp
```

- 现在，有了一个简单的目录清单程序。
  从文件系统库中可以获得更多信息。来创建一个 print_dir() 函数来收集更多信息，并将其格式化为 Unix 的 ls 风格:

```
void print_dir(const de& dir) {
  using fs::perms;
  const auto fpath{ dir.path() };
  const auto fstat{ dir.symlink_status() };
  const auto fperm{ fstat.permissions() };
  const uintmax_t fsize{
    is_regular_file(fstat) ? file_size(fpath) : 0 };
  const auto fn{ fpath.filename() };

  string suffix{};
  if(is_directory(fstat)) suffix = "/";
  else if((fperm & perms::owner_exec) != perms::none) {
    suffix = "*";
  }
  cout << format("{}{}\n", fn, suffix);
}
```

print_dir() 函数接受一个 directory_entry 参数。然后，从 directory_entry 对象中检索一些有用的对象:

  - dir.path() 返回一个 path 对象
  - dir.symlink_status() 返回一个 file_status 对象，无符号链接。
  - fstat.permissions() 返回一个 perms 对象.
  - ssize 是文件的大小，fn 是文件名字符串。使用的时候，再仔细地研究吧。

Unix ls 使用文件名后面的尾随字符来表示目录或可执行文件。使用 is_directory() 测试 fstat 对象，以查看文件是否是目录，并在文件名后面添加/。同样，可以用 fperm 对象测试文件是否可执行。

在 sort() 之后的 for 循环中调用 print_dir():

```
std::sort(entries.begin(), entries.end(), dircmp_lc);
for(const auto& e : entries) {
  print_dir(e);
}
```

现在输出是这样的:

```
chrono*
chrono.cpp
formatter*
formatter.cpp
include*
Makefile
testdir/
working*
working.cpp
```

- 注意 include*，这实际上是一个符号链接。可以通过链接来正确地标记它，以获得目标路径:

```
string suffix{};
if(is_symlink(fstat)) {
  suffix = " -> ";
  suffix += fs::read_symlink(fpath).string();
}
else if(is_directory(fstat)) suffix = "/";
else if((fperm & perms::owner_exec) != perms::none)
suffix = "*";
```

read_symlink() 函数的作用是: 返回一个路径对象。我们获取返回路径对象的 string() 表示形式，并将其添加到输出的尾部:

```
chrono*
chrono.cpp
formatter*
formatter.cpp
include -> /Users/billw/include
Makefile
testdir/
working*
working.cpp
```

- Unix ls 命令还包含一串字符来表示文件的权限位。它看起来像这样:drwxr-xr-x。
  第一个字符表示文件的类型，例如:d 表示目录，l 表示符号链接，-表示普通文件。
  type_char() 函数返回相应的字符:

```
char type_char(const fs::file_status& fstat) {
     if(is_symlink(fstat)) return 'l';
  else if(is_directory(fstat)) return 'd';
```

```
  else if(is_character_file(fstat)) return 'c';
  else if(is_block_file(fstat)) return 'b';
  else if(is_fifo(fstat)) return 'p';
  else if(is_socket(fstat)) return 's';
  else if(is_other(fstat)) return 'o';
  else if(is_regular_file(fstat)) return '-';
  return '?';
}
```

剩下的字符串是三元组。每个三元组以 rwx 的形式包含读、写和执行权限位的位置。若一个位没有设置，其字符将使用-替换。三组权限分别对应三个三元组:owner、group 和 other。

```
string rwx(const fs::perms& p) {
  using fs::perms;
  auto bit2char = [&p](perms bit, char c) {
    return (p & bit) == perms::none ? '-' : c;
  };
  return { bit2char(perms::owner_read, 'r'),
    bit2char(perms::owner_write, 'w'),
    bit2char(perms::owner_exec, 'x'),
    bit2char(perms::group_read, 'r'),
    bit2char(perms::group_write, 'w'),
    bit2char(perms::group_exec, 'x'),
    bit2char(perms::others_read, 'r'),
    bit2char(perms::others_write, 'w'),
    bit2char(perms::others_exec, 'x') };
}
```

perms 对象表示 POSIX 权限位图，但它不一定以位的形式实现。每个条目必须与 perms::none 值进行比较。我们的函数满足了这个要求。

我们将这个定义添加到 print_dir() 函数的顶部:

```
const auto permstr{ type_char(fstat) + rwx(fperm) };
```

更新 format() 字符串:

```
cout << format("{} {}{}\n", permstr, fn, suffix);
```

得到这样的输出:

```
-rwxr-xr-x chrono*
-rw-r--r-- chrono.cpp
-rwxr-xr-x formatter*
-rw-r--r-- formatter.cpp
lrwxr-xr-x include -> /Users/billw/include
-rw-r--r-- Makefile
drwxr-xr-x testdir/
-rwxr-xr-x working*
-rw-r--r-- working.cpp
```

- 现在，让我们添加一定长度的字符串。fsize 值来自 file_size() 函数，该函数返回 std::uintmax_t 类型，这表示目标系统上的最大自然整数大小。uintmax_t 并不总是与 size_t 相同，并不总是可以转换。值得注意的是，uintmax_t 在 Windows 上是 32 位，而 size_t 是 64 位:

```
string size_string(const uintmax_t fsize) {
  constexpr const uintmax_t kilo{ 1024 };
  constexpr const uintmax_t mega{ kilo * kilo };
  constexpr const uintmax_t giga{ mega * kilo };
  string s;
  if(fsize >= giga ) return
    format("{}{}", (fsize + giga / 2) / giga, 'G');
  else if (fsize >= mega) return
    format("{}{}", (fsize + mega / 2) / mega, 'M');
  else if (fsize >= kilo) return
    format("{}{}", (fsize + kilo / 2) / kilo, 'K');
  else return format("{}B", fsize);
}
```

这个函数中，我选择使用 1024 作为 1K，因为这似乎是 Linux 和 BSD Unix 上的默认值。在生产环境中，这可能是一个命令行选项。

在 main() 中更新 format() 字符串:

```
cout << format("{} {:>6} {}{}\n",
  permstr, size_string(fsize), fn, suffix);
```

现在，可以得到这样的输出:

```
-rwxr-xr-x 284K chrono*
-rw-r--r-- 2K chrono.cpp
-rwxr-xr-x 178K formatter*
-rw-r--r-- 906B formatter.cpp
lrwxr-xr-x 0B include -> /Users/billw/include
-rw-r--r-- 642B Makefile
drwxr-xr-x 0B testdir/
-rwxr-xr-x 197K working*
-rw-r--r-- 5K working.cpp
```

Note

该程序是为 POSIX 系统设计的，例如 Linux 和 macOS。可以在 Windows 系统上运行，但 Windows 权限系统与 POSIX 系统不同。所以，在 Windows 上，权限位总是显示完全设置。

How it works…

文件系统库通过其 directory_entry 和相关类携带了一组丰富的信息。在这个示例中使用的主要类包括:

- path 类根据目标系统的规则表示文件系统路径。path 对象由字符串或另一个路径构造，不需要表示现有的路径，甚至不需要表示可能的路径。path 字符串可解析为多个部分，包括根名称、根目录和可选的一系列文件名和目录分隔符。
- directory_entry 类包含一个 path 对象作为成员，还可以存储其他属性，包括硬链接计数、状态、符号链接、文件大小和最后写入时间。
- file_status 类携带有关文件类型和权限的信息。perms 对象可以是 file_status 的成员，表示文件的权限结构。

有两个函数用于从 file_status 中检索 perms 对象。status() 函数和 symlink_status() 函数都返回一个 perms 对象，区别在于如何处理符号链接。status() 函数将跟随符号链接并返回目标文件中的 perms。symlink_status() 将返回符号链接本身的 perms。

There's more…

我原本打算在目录列表中包括每个文件的最后写入时间。

directory_entry 类有一个成员函数 last_write_time()，它返回一个 file_time_type 对象，表示最后一次写入文件的时间戳。

但在编写本文时，可用的实现缺乏将 file_time_type 对象转换为标准 chrono::sys_time 的可移植方法。

现在，这里有一个可以在 GCC 上工作的解决方案:

```
string time_string(const fs::directory_entry& dir) {
  using std::chrono::file_clock;
  auto file_time{ dir.last_write_time() };
  return format("{:%F %T}",
    file_clock::to_sys(dir.last_write_time()));
}
```

建议用户代码使用 std::chrono::clock_cast，而非 file::clock::to_sys 来转换时钟之间的时间点。目前的可用实现中，都没有用于此目的的 std::chrono::clock_cast 特化。

使用这个 time_string() 函数，可以将时间添加到 print_dir():

```
const string timestr{ time_string(dir) };
```

然后，可以改变 format() 字符串:

```
cout << format("{} {:>6} {} {}{}\n",
  permstr, sizestr, timestr, fn, suffix);
```

并得到这样的输出:

```
-rwxr-xr-x 248K 2022-03-09 09:39:49 chrono*
-rw-r--r-- 2K 2022-03-09 09:33:56 chrono.cpp
-rwxr-xr-x 178K 2022-03-09 09:39:49 formatter*
-rw-r--r-- 906B 2022-03-09 09:33:56 formatter.cpp
lrwxrwxrwx 0B 2022-02-04 11:39:53 include -> /home/billw/
include
-rw-r--r-- 642B 2022-03-09 14:08:37 Makefile
drwxr-xr-x 0B 2022-03-09 10:38:39 testdir/
-rwxr-xr-x 197K 2022-03-12 17:13:46 working*
-rw-r--r-- 5K 2022-03-12 17:13:40 working.cpp
```

这适用于 Debian 的 GCC-11，但不要指望它很容易的就能在其他系统上工作。

10.5. 使用 grep 实用程序搜索目录和文件

为了演示遍历和搜索目录结构，创建了一个工作方式类似 Unix grep 的简单程序。这个程序使用 recursive_directory_iterator 遍历嵌套目录，并使用正则表达式搜索文件以查找匹配。

How to do it…

这个示例中，我们编写了一个简单的 grep 程序，可以遍历目录以使用正则表达式搜索文件:

- 先从一些别名开始:

```
namespace fs = std::filesystem;
using de = fs::directory_entry;
```

```
using rdit = fs::recursive_directory_iterator;
using match_v = vector<std::pair<size_t, std::string>>;
```

match_v 是一个正则表达式匹配结果的 vector。

- 继续使用 path 对象的格式化特化:

```
template<>
struct std::formatter<fs::path>:
std::formatter<std::string> {
  template<typename FormatContext>
  auto format(const fs::path& p, FormatContext& ctx) {
    return format_to(ctx.out(), "{}", p.string());
  }
};
```

- 我们有一个简单的函数，用于从文件中获取正则表达式匹配:

```
match_v matches(const fs::path& fpath, const regex& re) {
  match_v matches{};
  std::ifstream instrm(fpath.string(),
    std::ios_base::in);
  string s;
  for(size_t lineno{1}; getline(instrm, s); ++lineno) {
    if(std::regex_search(s.begin(), s.end(), re)) {
      matches.emplace_back(lineno, move(s));
    }
  }
  return matches;
}
```

在这个函数中，使用 ifstream 打开文件，使用 getline() 从文件中读取行，并使用 regex_search() 匹配正则表达式。在 vector 中收集结果并返回。

- 现在可以在 main() 中使用这个函数:

```
int main() {
  constexpr const char * fn{ "working.cpp" };
  constexpr const char * pattern{ "path" };

  fs::path fpath{ fn };
  regex re{ pattern };
  auto regmatches{ matches(fpath, re) };
  for(const auto& [lineno, line] : regmatches) {
    cout << format("{}: {}\n", lineno, line);
  }
  cout << format("found {} matches\n", regmatches.
  size());
}
```

在本例中，使用常量作为文件名和正则表达式模式。创建 path 和 regex 对象，调用 matches() 函数，并打印结果。

输出行号和匹配行的字符串:

```
  25: struct std::formatter<fs::path>:
  std::formatter<std::string> {
  27: auto format(const fs::path& p, FormatContext& ctx) {
  32: match_v matches(const fs::path& fpath, const regex& re) {
  34: std::ifstream instrm(fpath.string(), std::ios_base::in);
  62: constexpr const char * pattern{ "path" };
  64: fs::path fpath{ fn };
  66: auto regmatches{ matches(fpath, re) };
```

- 程序需要接受 regex 模式和文件名的命令行参数，其能够遍历目录或获取文件名列表 (这可能是命令行通配符展开的结果)。这需要 main() 函数中的一些逻辑处理。
  首先，需要一个辅助函数:

```
size_t pmatches(const regex& re, const fs::path& epath,
const fs::path& search_path) {
  fs::path target{epath};
  auto regmatches{ matches(epath, re) };
  auto matchcount{ regmatches.size() };
  if(!matchcount) return 0;

  if(!(search_path == epath)) {
    target =
    epath.lexically_relative(search_path);
  }
  for (const auto& [lineno, line] : regmatches) {
    cout << format("{} {}: {}\n", target, lineno,
    line);
  }
  return regmatches.size();
}
```

  这个函数调用 matches() 函数并输出结果，其接受一个 regex 对象和两个 path 对象。epath 是目录搜索的结果，而 search_path 是搜索目录本身。我们将在 main() 中设置这些参数。
- 在 main() 中，使用 argc 和 argv 命令行参数，并声明了几个变量:

```
int main(const int argc, const char** argv) {
  const char * arg_pat{};
  regex re{};
  fs::path search_path{};
  size_t matchcount{};
  ...
```

  这里需要声明的变量是:
  - arg_pat 用于命令行中的正则表达式模式

  - re 是正则表达式对象
  - search_path 命令行搜索路径是参数
  - matchcount 是用来计数匹配的行

- 若没有参数，则打印一个简短的用法字符串:

```
if(argc < 2) {
  auto cmdname{ fs::path(argv[0]).filename() };
  cout << format("usage: {} pattern [path/file]\n",
    cmdname);
  return 1;
}
```

argv[1] 始终是来自命令行的调用命令。cmdname 使用 filename() 方法，会返回只包含调用命令路径的文件名。

- 接下来，解析正则表达式。使用 try-catch 块来捕获来自正则表达式解析器的错误:

```
arg_pat = argv[1];
try {
  re = regex(arg_pat, std::regex_constants::icase);
} catch(const std::regex_error& e) {
  cout << format("{}: {}\n", e.what(), arg_pat);
  return 1;
}
```

使用 icase 标志告诉正则表达式解析器忽略大小写。

- 若 argc == 2，只有一个参数，将其视为正则表达式模式，并且使用当前目录作为搜索路径:

```
if(argc == 2) {
  search_path = ".";
  for (const auto& entry : rdit{ search_path }) {
    const auto epath{ entry.path() };
    matchcount += pmatches(re, epath,
    search_path);
  }
}
```

rdit 是 recursive_directory_iterator 类的别名，该类从起始路径遍历目录树，为遇到的每个文件返回一个 directory_entry 对象。然后，创建一个 path 对象，并调用 pmatches() 遍历文件，从而打印所有正则表达式匹配项。

- 此时在 main() 中，argc 是 >=2。现在，处理命令行上有一个或多个文件路径的情况:

```
int count{ argc - 2 };
while(count-- > 0) {
  fs::path p{ argv[count + 2] };
  if(!exists(p)) {
    cout << format("not found: {}\n", p);
    continue;
  }
  if(is_directory(p)) {
```

```
    for (const auto& entry : rdit{ p }) {
      const auto epath{ entry.path() };
      matchcount += pmatches(re, epath, p);
    }
  } else {
    matchcount += pmatches(re, p, p);
  }
}
```

while 循环处理命令行上搜索模式以外的一个或多个参数，其检查每个文件名以确保其存在。然后，若其是一个目录，将为 recursive_directory_iterator 类使用 rdit 别名，来遍历目录并调用 pmatches() 来打印文件中的模式匹配。

若是单个文件，则可在该文件上调用 pmatches()。

- 可以用一个参数作为搜索模式运行我们的 grep:

```
$ ./bwgrep using
dir.cpp 12: using std::format;
dir.cpp 13: using std::cout;
dir.cpp 14: using std::string;
...
formatter.cpp 10: using std::cout;
formatter.cpp 11: using std::string;
formatter.cpp 13: using namespace std::filesystem;
found 33 matches
```

可以用第二个参数作为目录来运行:

```
$ ./bwgrep using ..
chap04/iterator-adapters.cpp 12: using std::format;
chap04/iterator-adapters.cpp 13: using std::cout;
chap04/iterator-adapters.cpp 14: using std::cin;
...
chap01/hello-version.cpp 24: using std::print;
chap01/chrono.cpp 8: using namespace std::chrono_
literals;
chap01/working.cpp 15: using std::cout;
chap01/working.cpp 34: using std::vector;
found 529 matches
```

注意，这里通过遍历目录树来查找子目录中的文件。

或者，可以用一个文件参数运行它:

```
$ ./bwgrep using bwgrep.cpp
bwgrep.cpp 13: using std::format;
bwgrep.cpp 14: using std::cout;
bwgrep.cpp 15: using std::string;
...
bwgrep.cpp 22: using rdit = fs::recursive_directory_
iterator;
bwgrep.cpp 23: using match_v = vector<std::pair<size_t,
std::string>>;
found 9 matches
```

How it works…

虽然，这个程序的主要任务是正则表达式匹配，但我们主要关注递归处理文件目录的技术。

recursive_directory_iterator 对象可与 directory_iterator 对象互换，不同的是 recursive_directory_iterator 对象对每个子目录的所有条目进行递归操作。

See also…

有关正则表达式的更多信息，请参见第 7 章的相关章节。

10.6. 使用 regex 和 directory_iterator 重命名文件

这是一个使用正则表达式重命名文件的简单实用程序，可使用 directory_iterator 查找目录中的文件，并使用 fs::rename() 重命名。

How to do it…

在这个示例中，我们创建了一个使用正则表达式的文件重命名实用程序:

- 先从定义一些别名开始:

```
namespace fs = std::filesystem;
using dit = fs::directory_iterator;
using pat_v = vector<std::pair<regex, string>>;
```

别名 pat_v 是一个用于正则表达式的 vector。

- 我们还继续为 path 对象使用格式化特化:

```
template<>
struct std::formatter<fs::path>:
std::formatter<std::string> {
  template<typename FormatContext>
  auto format(const fs::path& p, FormatContext& ctx) {
```

```
    return format_to(ctx.out(), "{}", p.string());
  }
};
```

- 我们有一个函数用于对文件名字符串应用正则表达式替换:

```
string replace_str(string s, const pat_v& replacements) {
  for(const auto& [pattern, repl] : replacements) {
    s = regex_replace(s, pattern, repl);
  }
  return s;
}
```

注意，我们循环遍历模式/替换对的 vector，依次应用正则表达式。这使得我们可以堆叠替换。

- main() 中，首先检查命令行参数:

```
int main(const int argc, const char** argv) {
  pat_v patterns{};
  if(argc < 3 || argc % 2 != 1) {
    fs::path cmdname{ fs::path{argv[0]}.filename() };
    cout << format(
      "usage: {} [regex replacement] ...\n",
      cmdname);
    return 1;
  }
```

命令行接受一个或多个字符串对。每对字符串都包含一个正则表达式 (正则表达式)，后面跟着一个替换。

- 现在，用 regex 和 string 对象填充 vector:

```
for(int i{ 1 }; i < argc; i += 2) {
  patterns.emplace_back(argv[i], argv[i + 1]);
}
```

pair 构造函数根据在命令行上传递的 C-string 构造适当的 regex 和 string 对象，其通过 emplace_back() 方法添加到 vector 中。

- 使用 directory_iterator 对象搜索当前目录:

```
for(const auto& entry : dit{fs::current_path()}) {
  fs::path fpath{ entry.path() };
  string rname{
    replace_str(fpath.filename().string(),
    patterns) };
  if(fpath.filename().string() != rname) {
    fs::path rpath{ fpath };
    rpath.replace_filename(rname);
    if(exists(rpath)) {
      cout << "Error: cannot rename - destination
      file exists.\n";
    } else {
      fs::rename(fpath, rpath);
```

```
      cout << format(
      "{} -> {}\n",
      fpath.filename(),
      rpath.filename());
    }
  }
}
```

这个 for 循环中，调用 replace_str() 来获取替换的文件名，然后检查新名称是不是目录中某个文件的副本。在 path 对象上使用 replace_filename() 方法创建具有新文件名的路径，并使用 fs::rename() 重命名文件。

- 为了测试这个工具，我创建了一个目录，里面有几个文件用于重命名:

```
$ ls
bwfoo.txt bwgrep.cpp chrono.cpp dir.cpp formatter.cpp
path-ops.cpp working.cpp
```

- 可以做一些简单的事情，比如把.cpp 改成.Cpp:

```
$ ../rerename .cpp .Cpp
dir.cpp -> dir.Cpp
path-ops.cpp -> path-ops.Cpp
bwgrep.cpp -> bwgrep.Cpp
working.cpp -> working.Cpp
formatter.cpp -> formatter.Cpp
```

再把它们改回来:

```
$ ../rerename .Cpp .cpp
formatter.Cpp -> formatter.cpp
bwgrep.Cpp -> bwgrep.cpp
dir.Cpp -> dir.cpp
working.Cpp -> working.cpp
path-ops.Cpp -> path-ops.cpp
```

- 使用标准的正则表达式语法，我可以在每个文件名的开头添加“bw”:

```
$ ../rerename '^' bw
bwgrep.cpp -> bwbwgrep.cpp
chrono.cpp -> bwchrono.cpp
formatter.cpp -> bwformatter.cpp
bwfoo.txt -> bwbwfoo.txt
working.cpp -> bwworking.cpp
```

注意，它重命名了开头已经有“bw”的文件。我们不想让它这样做。所以，先恢复文件名:

```
$ ../rerename '^bw' ''
bwbwgrep.cpp -> bwgrep.cpp
bwworking.cpp -> working.cpp
bwformatter.cpp -> formatter.cpp
bwchrono.cpp -> chrono.cpp
bwbwfoo.txt -> bwfoo.txt
```

现在，使用一个正则表达式来检查文件名是否已经以”bw” 开头:

```
$ ../rerename '^(?!bw)' bw
chrono.cpp -> bwchrono.cpp
formatter.cpp -> bwformatter.cpp
working.cpp -> bwworking.cpp
```

因为，使用了一个 regex/replacement 字符串的 vector，所以可以堆叠几个替换:

```
$ ../rerename foo bar '\.cpp$' '.xpp' grep grok
bwgrep.cpp -> bwgrok.xpp
bwworking.cpp -> bwworking.xpp
bwformatter.cpp -> bwformatter.xpp
bwchrono.cpp -> bwchrono.xpp
bwfoo.txt -> bwbar.txt
```

How it works…

这个示例的文件系统部分使用 directory_iterator 为当前目录中的每个文件，返回一个 directory_entry 对象:

```
for(const auto& entry : dit{fs::current_path()}) {
```

```
  fs::path fpath{ entry.path() };
  ...
}
```

然后，使用 directory_entry 对象构造一个 path 对象来处理文件。

我们在 path 对象上使用 replace_filename() 方法来创建重命名操作的目标:

```
fs::path rpath{ fpath };
rpath.replace_filename(rname);
```

这里，创建一个副本并更改它的名称:

```
fs::rename(fpath, rpath);
```

示例的正则表达式部分，regex_replace() 使用正则表达式语法在字符串中执行替换:

```
s = regex_replace(s, pattern, repl);
```

正则表达式语法非常强大，允许替换包含搜索字符串的部分:

```
$ ../rerename '(bw)(.*\.)(.*)$' '$3$2$1'
bwgrep.cpp -> cppgrep.bw
bwfoo.txt -> txtfoo.bw
```

通过在搜索模式中使用括号，可以轻松地重新排列文件名的各个部分。

10.7. 创建磁盘使用计数器

这是一个简单的程序，用于计算目录及其子目录中每个文件的大小，可以在 POSIX/Unix 和 Windows 文件系统上运行。

How to do it…

这个示例是一个程序，用于报告目录及其子目录中每个文件的大小，以及总数。我们将重用在本章其他地方使用过的一些函数:

- 先从几个别名开始说起:

```
namespace fs = std::filesystem;
using dit = fs::directory_iterator;
using de = fs::directory_entry;
```

- 还对 fs::path 对象使用格式化特化:

```
template<>
struct std::formatter<fs::path>:
std::formatter<std::string> {
  template<typename FormatContext>
  auto format(const fs::path& p, FormatContext& ctx) {
    return format_to(ctx.out(), "{}", p.string());
```

```
  }
};
```

- 为了输出目录的大小，将使用 make_comma() 函数:

```
string make_commas(const uintmax_t& num) {
  string s{ std::to_string(num) };
  for(long l = s.length() - 3; l > 0; l -= 3) {
    s.insert(l, ",");
  }
  return s;
}
```

我们以前用过这个，其从结尾开始，每三个字符前插入一个逗号。

- 要对目录进行排序，需要一个小写字符串的函数:

```
string strlower(string s) {
  auto char_lower = [](const char& c) -> char {
    if(c >= 'A' && c <= 'Z') return c + ('a' -
      'A');
    else return c;
  };
  std::transform(s.begin(), s.end(), s.begin(),
    char_lower);
  return s;
}
```

- 需要一个比较谓词，根据路径名的小写字母对 directory_entry 对象进行排序:

```
bool dircmp_lc(const de& lhs, const de& rhs) {
  const auto lhstr{ lhs.path().string() };
  const auto rhstr{ rhs.path().string() };
  return strlower(lhstr) < strlower(rhstr);
}
```

- size_string() 返回用于报告文件大小的缩写值，单位为千兆字节、兆字节、千字节或字节:

```
string size_string(const uintmax_t fsize) {
  constexpr const uintmax_t kilo{ 1024 };
  constexpr const uintmax_t mega{ kilo * kilo };
  constexpr const uintmax_t giga{ mega * kilo };

  if(fsize >= giga ) return format("{}{}",
    (fsize + giga / 2) / giga, 'G');
  else if (fsize >= mega) return format("{}{}",
    (fsize + mega / 2) / mega, 'M');
  else if (fsize >= kilo) return format("{}{}",
    (fsize + kilo / 2) / kilo, 'K');
  else return format("{}B", fsize);
}
```

- entry_size() 返回文件的大小，若是目录，则返回目录的递归大小:

```
uintmax_t entry_size(const fs::path& p) {
  if(fs::is_regular_file(p)) return
    fs::file_size(p);
  uintmax_t accum{};
  if(fs::is_directory(p) && ! fs::is_symlink(p)) {
    for(auto& e : dit{ p }) {
      accum += entry_size(e.path());
    }
  }
  return accum;
}
```

- main() 中，从声明开始，并测试是否有一个有效的目录要搜索:

```
int main(const int argc, const char** argv) {
  auto dir{ argc > 1 ?
    fs::path(argv[1]) : fs::current_path() };
  vector<de> entries{};
  uintmax_t accum{};
  if (!exists(dir)) {
    cout << format("path {} does not exist\n",
    dir);
    return 1;
  }
  if(!is_directory(dir)) {
    cout << format("{} is not a directory\n",
    dir);
    return 1;
  }
  cout << format("{}:\n", absolute(dir));
```

对于目录路径 dir，若有参数，则使用 argv[1]。否则，对当前目录使用 current_path()。然后，为用法计数器设置环境:

  - directory_entry 对象的 vector 用于排序。
  - accum 用于累积文件最终总大小的值。
  - 继续检查目录之前，需要确保 dir 存在，并且的确是一个目录。

- 接下来，一个简单的循环填充 vector。填充完成后，使用 dircmp_lc() 函数作为比较谓词对条目进行排序:

```
for (const auto& e : dit{ dir }) {
  entries.emplace_back(e.path());
}
std::sort(entries.begin(), entries.end(), dircmp_lc);
```

- 现在一切都设置好了，可以使用排序 vector 中，directory_entry 对象累积的结果了:

```
for (const auto& e : entries) {
  fs::path p{ e };
```

```
    uintmax_t esize{ entry_size(p) };
    string dir_flag{};
    accum += esize;
    if(is_directory(p) && !is_symlink(p)) dir_flag =
    " *";
    cout << format("{:>5} {}{}\n",
    size_string(esize), p.filename(), dir_flag);
}
cout << format("{:->25}\n", "");
cout << format("total bytes: {} ({})\n",
    make_commas(accum), size_string(accum));
```

对 entry_size() 的调用返回 directory_entry 对象中表示的文件或目录的大小。

若当前 entry 是一个目录 (而不是一个符号链接)，就添加一个符号来表示它是一个目录。我选择了一个倒三角，你可以选择你喜欢的符号。[译者注：由于倒三角不好显示，这里选择了使用 * 表示]

循环完成后，用逗号显示两个字节的累积大小，以及 size_string() 的缩写符号。

输出为:

```
/home/billw/working/cpp-stl-wkbk/chap10:
327K bwgrep
  3K bwgrep.cpp
199K dir
  4K dir.cpp
176K formatter
905B formatter.cpp
  0B include
  1K Makefile
181K path-ops
  1K path-ops.cpp
327K rerename
  2K rerename.cpp
 11K testdir *
 11K testdir-backup *
203K working
  3K working.cpp
-------------------------
total bytes: 1,484,398 (1M)
```

How it works…

fs::file_size() 函数的作用是: 返回一个 uintmax_t 值，该值表示文件的大小，为给定平台上最大的自然无符号整数。虽然，在大多数 64 位系统上这通常是一个 64 位整数，但 Windows 是一个例外，其使用 32 位整数。虽然，size_t 可能在某些系统上适用于这个值，但在 Windows 上无法编译，因为这里可能试图将 64 位值提升为 32 位值。

entry_size() 函数接受一个路径对象，并返回 uintmax_t 值:

```
uintmax_t entry_size(const fs::path& p) {
  if(fs::is_regular_file(p)) return fs::file_size(p);
  uintmax_t accum{};
  if(fs::is_directory(p) && !fs::is_symlink(p)) {
    for(auto& e : dit{ p }) {
      accum += entry_size(e.path());
    }
  }
  return accum;
}
```

该函数检查常规文件并返回文件的大小。否则，将检查一个目录是否也是符号链接。我们只想知道目录中文件的大小，因此不希望有符号链接。(符号链接可能会出现引用循环)

若找到一个目录，就循环遍历，为遇到的每个文件使用 entry_size()。这是一个递归循环，所以我们最终会得到整个目录的大小。

第 11 章　一些的想法

本书中，我们学习了一些有用的技术，包括 optional、容器、迭代器、算法、智能指针等等。我们已经看到了使用这些概念的例子，也有机会将它们应用到一些项目中。现在，将这些技巧应用到一些更实际的想法中。

在本章中，我们将讨论以下主题:

- 为搜索建议创建一个 trie 类
- 计算两个 vector 的误差和
- 创建自己的算法:split
- 利用现有算法:gather
- 删除连续的空格
- 数字转换为单词

11.1. 相关准备

本章所有的例子和源代码都可以在本书的 GitHub 库中找到，网址是https://github.com/PacktPublishing/CPP-20-STL-Cookbook/tree/main/chap11。

11.2. 为搜索建议创建一个 trie 类

trie，有时称为前缀树，是一种搜索树，通常用于预测文本和其他搜索应用程序。trie 是为深度优先搜索设计的递归结构，其中每个节点既是一个键，又是另一个 trie。

一个常见的用例是字符串的 trie，其中每个节点都是句子中的字符串。例如:

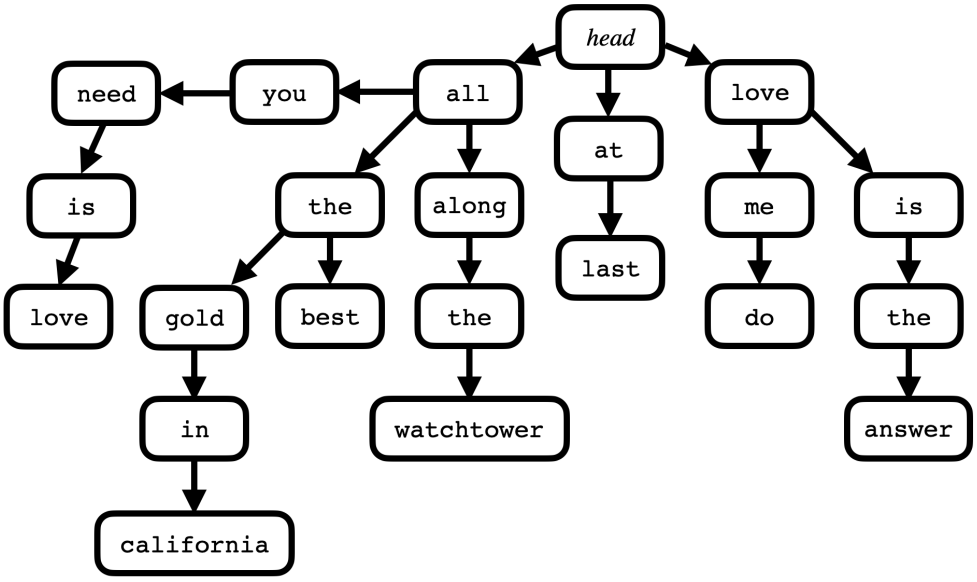


图 11.1 字符串 trie

我们经常从树的头部开始搜索，寻找以特定单词开头的句子。在这个例子中，当我搜索 all 时，我得到了三个节点:you、the 和 along。

若寻找 love，我得到的是 me 和 is。

字符串 trie 通常用于创建搜索建议。这里，我们将使用 std::map 来实现一个字符串 trie 结构。

How to do it…

在这个示例中，创建了一个递归的 trie 类，将节点存储在 std::map 容器中。对于内存中的树来说，这是一个简单的解决方案。这是一个相当大的类，所以在这里只展示重要的部分。

要获得完整的类，请参阅源代码 https://github.com/PacktPublishing/CPP-20-STL-Cookbook/blob/main/chap11/trie.cpp。

- 起一个方便的别名:

```
using ilcstr = initializer_list<const char *>;
```

使用 ilcstr 来搜索 trie。

- 把这个类放在一个私有命名空间中以避免冲突:

```
namespace bw {
  using std::map;
  using std::deque;
  using std::initializer_list;
```

方便起见，在这个命名空间中使用一些 using 语句。

- 类本身称为 trie，其有三个数据成员:

```
class trie {
  using get_t = deque<deque<string>>;
  using nodes_t = map<string, trie>;
  using result_t = std::optional<const trie*>;

  nodes_t nodes{};
  mutable get_t result_dq{};
  mutable deque<string> prefix_dq{};
```

trie 类有一些私有类型别名:

  - get_t 是字符串的 deque 的 deque，用于字符串结果。
  - nodes_t 是具有字符串键的 trie 类的 map。
  - result_t 是指向 trie 指针的可选参数，用于返回搜索结果。空 trie 也是一个有效的结果，所以可以使用一个 optional 值。

nodes 对象用于保存节点的递归 map，其中 trie 上的每个节点都是另一个 trie。

- 公共接口经常调用私有接口中的实用函数。例如，insert() 方法接受一个 initializer_list 对象，并调用私有函数_insert():

```
void insert(const ilcstr& il) {
  _insert(il.begin(), il.end());
}
```

私有函数_insert() 执行插入元素的工作:

```
template <typename It>
void _insert(It it, It end_it) {
  if(it == end_it) return;
  nodes[*it]._insert(++it, end_it);
}
```

这方便了导航 trie 所需的递归函数调用。注意，引用一个不在 map 中出现的键，会创建一个带有该键的空元素。因此，若元素不存在，在 nodes 元素上使用_insert() 的那行代码，将创建空 trie 对象。

- get() 方法返回一个 get_t 对象，我是字符串队列的队列的别名。这就可以返回多组结果:

```
get_t& get() const {
  result_dq.clear();
  deque<string> dq{};
  _get(dq, result_dq);
  return result_dq;
}
```

get() 方法调用私有的_get() 函数，该函数递归遍历 trie:

```
void _get(deque<string>& dq, get_t& r_dq) const {
  if(empty()) {
    r_dq.emplace_back(dq);
    dq.clear();
  }
  for(const auto& p : nodes) {
    dq.emplace_back(p.first);
    p.second._get(dq, r_dq);
  }
}
```

- find_prefix() 函数的作用是: 返回一个 deque 对象，其中包含与部分字符串的所有匹配项。

```
deque<string>& find_prefix(const char * s) const {
  _find_prefix(s, prefix_dq);
  return prefix_dq;
}
```

公共接口调用私有函数_find_prefix():

```
void _find_prefix(const string& s, auto& pre_dq) const {
  if(empty()) return;
  for(const auto& [k, v] : nodes) {
    if(k.starts_with(s)) {
      pre_dq.emplace_back(k);
      v._find_prefix(k, pre_dq);
    }
  }
}
```

私有_find_prefix() 函数递归遍历 trie，将前缀与每个键的开头进行比较。starts_with() 方法是 C++20 中的新方法。对于旧的 STL，可以使用 find() 方法，并检查返回值是否为 0:

```
if(k.find(s) == 0) {
  ...
```

- search() 函数返回一个可选的 <const trie*>，别名为 result_t。有两个重载:

```
result_t search(const ilcstr& il) const {
  return _search(il.begin(), il.end());
}
result_t search(const string& s) const {
  const ilcstr il{s.c_str()};
  return _search(il.begin(), il.end());
}
```

这些方法将迭代器传递给私有成员函数_search()，该函数执行搜索工作:

```
template <typename It>
result_t _search(It it, It end_it) const {
  if(it == end_it) return {this};
  auto found_it = nodes.find(*it);
  if(found_it == nodes.end()) return {};
  return found_it->second._search(++it, end_it);
}
```

_search() 函数递归搜索，直到找到匹配项，然后返回 result_t 对象中的节点。若没有找到匹配项，则返回非值 optional 值。

- 还有两个 print_trie_prefix() 函数的重载。这个函数从一个用作搜索键的前缀打印 trie 的内容。一个版本使用字符串作为前缀，另一个版本使用 C-strings 的 initializer_list:

```
void print_trie_prefix(const bw::trie& t,
    const string& prefix) {
  auto& trie_strings = t.get(\);
  cout << format("results for \"{}...\":\n", prefix);
  for(auto& dq : trie_strings) {
    cout << format("{} ", prefix);
    for(const auto& s : dq) cout << format("{} ", s);
    cout << '\n';
  }
}

void print_trie_prefix(const bw::trie& t,
    const ilcstr & prefix) {
  string sprefix{};
  for(const auto& s : prefix) sprefix +=
    format("{} ", s);
  print_trie_prefix(t, sprefix);
}
```

这些函数调用 get() 成员函数从 trie 中检索结果。

- 现在，可以在 main() 函数中测试 trie 类。首先，声明一个 trie，并插入一些句子:

```
int main() {
  bw::trie ts;
  ts.insert({ "all", "along", "the", "watchtower" });
  ts.insert({ "all", "you", "need", "is", "love" });
  ts.insert({ "all", "shook", "up" });
  ts.insert({ "all", "the", "best" });
  ts.insert({ "all", "the", "gold", "in",
    "california" });
  ts.insert({ "at", "last" });
  ts.insert({ "love", "the", "one", "you're",
    "with" });
  ts.insert({ "love", "me", "do" });
  ts.insert({ "love", "is", "the", "answer" });
  ts.insert({ "loving", "you" });
  ts.insert({ "long", "tall", "sally" });
  ...
```

insert() 调用传递一个包含句子所有字符串的 initializer_list，句子的每个字符串都会插入到树的层次结构中。

- 现在，可以搜索这个 trie。这里有一个简单的搜索单字符串“love”。

```
const auto prefix = {"love"};
if (auto st = ts.search(prefix); st.have_result) {
  print_trie_prefix(*st.t, prefix);
}
cout << '\n';
```

调用 ts.search()，initializer_list 为一个 C 字串，称为 prefix。结果连同前缀一起传递给 print_trie_prefix() 函数。

输出为：

```
results for "love...":
love is the answer
love me do
love the one you're with
```

- 下面是一个搜索双字符串前缀的例子:

```
const auto prefix = {"all", "the"};
if (auto st = ts.search(prefix); st.have_result) {
  print_trie_prefix(*st.t, prefix);
}
cout << '\n';
```

输出为:

```
results for "all the ...":
all the best
all the gold in california
```

- 下面是使用 find_prefix() 函数搜索部分前缀:

```
const char * prefix{ "lo" };
auto prefix_dq = ts.find_prefix(prefix);
for(const auto& s : prefix_dq) {
  cout << format("match: {} -> {}\n", prefix, s);
  if (auto st = ts.search(s); st.have_result) {
    print_trie_prefix(*st.t, s);
  }
}
cout << '\n';
```

输出为:

```
match: lo -> long
results for "long...":
long tall sally
match: lo -> love
results for "love...":
love is the answer
love me do
love the one you're with
match: lo -> loving
results for "loving...":
loving you
```

find_prefix() 搜索返回了几个结果，将每个结果传递给自己的搜索，每个结果产生几个结果。

How it works…

trie 类的数据存储在递归 map 容器中。map 中的每个节点都包含另一个 trie 对象，该对象又有自己的 map 节点。

```
using nodes_t = map<string, trie>
```

_insert() 函数接受 begin 和 end 迭代器，并使用它们在新节点上递归调用_insert():

```
template <typename It>
void _insert(It it, It end_it) {
  if(it == end_it) return;
```

```
  nodes[*it]._insert(++it, end_it);
}
```

同样，_search() 函数在它找到的节点上递归调用_search():

```
template <typename It>
result_t _search(It it, It end_it) const {
  if(it == end_it) return {this};
  auto found_it = nodes.find(*it);
  if(found_it == nodes.end()) return {};
  return found_it->second._search(++it, end_it);
}
```

这种使用 std::map 的递归方法可以有效地实现一个 trie 类。

11.3. 计算两个 vector 的误差和

给定两个相似的向量，仅通过量化或分辨率不同，我们可以使用 inner_product() 算法来计算误差和，定义为:

$$e = \sum_{n}^{i=1}(a_i - b_i)^2$$

图 11.2 误差和的定义

其中 e 为误差和，即两个向量中一系列点的差的平方和。

可以使用 <numeric> 中的 inner_product() 算法来计算两个向量之间的误差和。

How to do it…

在这个示例中，定义了两个向量，每个向量都有一个正弦波。一个 vector 的值是 double 类型，另一个是 int 类型。这给了我们量子化不同的向量，因为 int 类型不能表示分数值。然后我们使用 inner_product() 来计算两个向量之间的误差和:

- 在 main() 函数中，我们定义了 vector 和一个索引变量:

```
int main() {
  constexpr size_t vlen{ 100 };
  vector<double> ds(vlen);
  vector<int> is(vlen);
  size_t index{};
  ...
```

ds 是双正弦波的 vector，是 int 正弦波的 vector。每个向量有 100 个元素来保存一个正弦波，index 变量用于初始化 vector 对象。

- 用一个循环和 lambda 在 double 的 vector 中生成正弦波:

```
auto sin_gen = [&index]{
  return 5.0 * sin(index++ * 2 * pi / 100);
```

```
};
for(auto& v : ds) v = sin_gen();
```

lambda 捕获对索引变量的引用，因此可以对其进行递增。

pi 常数来自 std::numbers 库。

- 我们现在有了一个双正弦波，可以用它来推导 int 版本:

```
index = 0;
for(auto& v : is) {
  v = static_cast<int>(round(ds.at(index++)));
}
```

这将从 ds 中取每个点，四舍五入，将其转换为 int 类型，并更新它在 is 容器中的位置。

- 用一个简单的循环来显示正弦波:

```
for(const auto& v : ds) cout << format("{:-5.2f} ", v);
cout << "\n\n";
for(const auto& v : is) cout << format("{:-3d} ", v);
cout << "\n\n";
```

输出是两个容器中作为数据点的正弦波:

```
 0.00 0.31 0.63 0.94 1.24 1.55 1.84 2.13 2.41
 2.68 2.94 3.19 3.42 3.64 3.85 4.05 4.22 4.38
 4.52 4.65 4.76 4.84 4.91 4.96 4.99 5.00 4.99
 4.96 4.91 4.84 4.76 4.65 4.52 4.38 4.22 4.05
 3.85 3.64 3.42 3.19 2.94 2.68 2.41 2.13 1.84
 1.55 1.24 0.94 0.63 0.31 0.00 -0.31 -0.63 -0.94
-1.24 -1.55 -1.84 -2.13 -2.41 -2.68 -2.94 -3.19 -3.42
-3.64 -3.85 -4.05 -4.22 -4.38 -4.52 -4.65 -4.76 -4.84
-4.91 -4.96 -4.99 -5.00 -4.99 -4.96 -4.91 -4.84 -4.76
-4.65 -4.52 -4.38 -4.22 -4.05 -3.85 -3.64 -3.42 -3.19
-2.94 -2.68 -2.41 -2.13 -1.84 -1.55 -1.24 -0.94 -0.63
-0.31

 0  0  1  1  1  2  2  2  2  3  3  3  3  4  4
 4  4  4  5  5  5  5  5  5  5  5  5  5  5  5
 5  5  5  4  4  4  4  4  3  3  3  3  2  2  2
 2  1  1  1  0  0  0 -1 -1 -1 -2 -2 -2 -2 -3
-3 -3 -3 -4 -4 -4 -4 -4 -5 -5 -5 -5 -5 -5 -5
-5 -5 -5 -5 -5 -5 -5 -5 -5 -4 -4 -4 -4 -4 -4
-3 -3 -3 -3 -2 -2 -2 -2 -1 -1 -1  0
```

- 现在使用 inner_product() 计算误差和:

```
double errsum = inner_product(ds.begin(), ds.end(),
```

```
  is.begin(), 0.0, std::plus<double>(),
  [](double a, double b){ return pow(a - b, 2); });
cout << format("error sum: {:.3f}\n\n", errsum);
```

lambda 表达式返回公式的 $(a_i - b_i)^2$ 的部分。std::plus() 算法执行求和运算。

输出为:

```
error sum: 7.304
```

How it works…

inner_product() 算法计算第一个输入范围内的乘积之和，签名为:

```
T inner_product(InputIt1 first1, InputIt1 last1,
  InputIt2 first2, T init, BinaryOperator1 op1,
  BinaryOperator2 op2)
```

该函数接受两个二元算子函子 op1 和 op2。第一个 op1 用于求和，第二个 op2 用于乘积。使用 std::plus() 作为和运算符，lambda 作为积运算符。

init 参数可以用作起始值或偏置，将 0.0 传递给它。

返回值是产品的累计总和。

There's more…

可以通过在循环中放入 inner_product() 来计算累积的误差和:

```
cout << "accumulated error:\n";
for (auto it{ds.begin()}; it != ds.end(); ++it) {
  double accumsum = inner_product(ds.begin(), it,
    is.begin(), 0.0, std::plus<double>(),
    [](double a, double b){ return pow(a - b, 2); });
  cout << format("{:-5.2f} ", accumsum);
}
cout << '\n';
```

输出为:

```
accumulated error:
0.00 0.00 0.10 0.24 0.24 0.30 0.51 0.53 0.55 0.72
0.82 0.82 0.86 1.04 1.16 1.19 1.19 1.24 1.38 1.61
1.73 1.79 1.82 1.82 1.83 1.83 1.83 1.83 1.83 1.84
1.86 1.92 2.04 2.27 2.42 2.46 2.47 2.49 2.61 2.79
2.83 2.83 2.93 3.10 3.12 3.14 3.35 3.41 3.41 3.55
3.65 3.65 3.75 3.89 3.89 3.95 4.16 4.19 4.20 4.37
4.47 4.48 4.51 4.69 4.82 4.84 4.84 4.89 5.03 5.26
5.38 5.44 5.47 5.48 5.48 5.48 5.48 5.48 5.48 5.49
5.51 5.57 5.70 5.92 6.07 6.12 6.12 6.14 6.27 6.45
6.48 6.48 6.59 6.75 6.77 6.80 7.00 7.06 7.07 7.21
```

这在某些统计应用中可能有用。

11.4. 创建自己的算法:split

STL 具有丰富的算法库。然而，有时可能会发现它缺少自己需要的东西。一个常见的需求是 split 函数。

split 函数在字符分隔符上拆分字符串。例如，下面是标准 Debian 安装的 Unix /etc/passwd 文件:

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
```

每个字段用冒号:character 分隔，其中字段为:

1. 账号
2. 可加密的密码
3. 用户名
4. 组名
5. 用户名或评论
6. 主目录
7. 可选的命令解释器

这是基于 POSIX 的操作系统中的标准文件，还有其他类似的文件。大多数脚本语言都包含一个内置函数，用于在分隔符上分隔字符串。C++ 中有一些简单的方法可以做到这一点，但 std::string 只是 STL 中的另一个容器，在分隔符上分割容器的通用算法可能是有用补充。让我们来创建一个。

How to do it…

在这个示例中，我们构建了一个通用算法，它在分隔符上拆分容器，并将结果放入目标容器中。

- 我们的算法在 bw 命名空间中，以避免与 std 冲突:

```
namespace bw {
  template<typename It, typename Oc, typename V,
  typename Pred>
  It split(It it, It end_it, Oc& dest,
  const V& sep, Pred& f) {
    using SliceContainer = typename
    Oc::value_type;
    while(it != end_it) {
      SliceContainer dest_elm{};
      auto slice{ it };
      while(slice != end_it) {
        if(f(*slice, sep)) break;
        dest_elm.push_back(*slice++);
      }
      dest.push_back(dest_elm);
      if(slice == end_it) return end_it;
      it = ++slice;
    }
    return it;
  }
};
```

split() 算法在容器中搜索分隔符，并将分离的切片收集到新的输出容器中，其中每部分都是输出容器中的容器。

我希望 split() 算法尽可能通用，就像算法库中的算法一样。这样，所有的参数都是模板化的，代码可以处理各种各样的参数类型。

首先，来看看模板参数:

- It 是源容器的输入迭代器类型。
- Oc 输出容器类型。这是容器中的容器。
- V 分隔符类型。
- Pred 为谓词函数。

输出类型是容器的容器，需要容纳切片的容器。可以是 vector<string>，其中字符串值是切片，也可以是 vector<vector<int>>，其中内部的 vector<int> 包含切片，所以需要从输出容器类型派生内部容器的类型。可以通过函数体中的 using 声明来做到这一点。

```
using SliceContainer = typename Oc::value_type;
```

这也是为什么不能为输出形参使用输出迭代器的原因。根据定义，输出迭代器不能确定其内容的类型，并且将其 value_type 设置为 void。

使用 SliceContainer 定义一个临时容器，可以将其添加到输出容器中:

```
dest.push_back(dest_elm);
```

- 谓词是一个二元操作符，用于比较输入元素和分隔符。在 bw 命名空间中包含了一个默认的相等操作符:

```
constexpr auto eq = [](const auto& el, const auto& sep) {
  return el == sep;
};
```

- 还包含了 split() 的特化，默认使用 eq 操作符:

```
template<typename It, typename Oc, typename V>
It split(It it, const It end_it, Oc& dest, const V& sep)
{
  return split(it, end_it, dest, sep, eq);
}
```

- 因为分割字符串对象是这个算法的常见用例，包含了一个用于特定目的的辅助函数:

```
template<typename Cin, typename Cout, typename V>
Cout& strsplit(const Cin& str, Cout& dest, const V& sep)
{
  split(str.begin(), str.end(), dest, sep, eq);
  return dest;
}
```

- 我们测试 split 算法 main() 函数:

```
int main() {
  constexpr char strsep{ ':' };
  const string str
    { "sync:x:4:65534:sync:/bin:/bin/sync" };
  vector<string> dest_vs{};
  bw::split(str.begin(), str.end(), dest_vs, strsep,
    bw::eq);
  for(const auto& e : dest_vs) cout <<
    format("[{}] ", e);
  cout << '\n';
}
```

使用/etc/passwd 文件中的字符串来测试算法，结果如下:

```
[sync] [x] [4] [65534] [sync] [/bin] [/bin/sync]
```

- 使用 strsplit() 帮助函数会更简单:

```
vector<string> dest_vs2{};
bw::strsplit(str, dest_vs2, strsep);
for(const auto& e : dest_vs2) cout << format("[{}] ", e);
cout << '\n';
```

输出为:

```
[sync] [x] [4] [65534] [sync] [/bin] [/bin/sync]
```

这样就可以很容易地解析/etc/passwd 文件。

- 当然，可以对任何容器使用相同算法:

```
constexpr int intsep{ -1 };
vector<int> vi{ 1, 2, 3, 4, intsep, 5, 6, 7, 8, intsep,
  9, 10, 11, 12 };
vector<vector<int>> dest_vi{};
bw::split(vi.begin(), vi.end(), dest_vi, intsep);
for(const auto& v : dest_vi) {
  string s;
  for(const auto& e : v) s += format("{}", e);
  cout << format("[{}] ", s);
}
cout << '\n';
```

输出为:

```
[1234] [5678] [9101112]
```

How it works…

split 算法本身比较简单，这个示例的神奇之处在于，模板可使其通用。

using 声明中的派生类型，可以创建与输出容器一起使用的容器:

```
using SliceContainer = typename Oc::value_type;
```

这里别名一个 SliceContainer 类型，可以用它为切片创建容器:

```
SliceContainer dest_elm{};
```

这是一个临时容器，可将每个切片添加到输出容器中:

```
dest.push_back(dest_elm);
```

11.5. 利用现有算法:gather

gather 是一个利用现有算法的算法示例。

gather 算法接受一对容器迭代器，并将满足谓词的元素移动到序列中的相应 (枢轴) 位置，返回一对包含满足谓词的元素的迭代器。

例如，可以使用收集算法将所有偶数排序到 vector 的中点:

```
vector<int> vint{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
gather(vint.begin(), vint.end(), mid(vint), is_even);
for(const auto& el : vint) cout << el;
```

输出为：

```
1302468579
```

注意，偶数都在输出的中间。

在这个示例中，我们将使用标准 STL 算法实现一个 gather 算法。

How to do it…

收集算法使用 std::stable_partition() 算法，将项移动到枢轴迭代器之前，并再次将项移动到枢轴迭代器之后。

- 将算法放在 bw 命名空间中以避免冲突。

```
namespace bw {
  using std::stable_partition;
  using std::pair;
  using std::not_fn;

  template <typename It, typename Pred>
  pair<It, It> gather(It first, It last, It pivot,
  Pred pred) {
    return {stable_partition(first, pivot, not_fn(pred)),
      stable_partition(pivot, last, pred)};
  }
};
```

gather() 算法返回一对迭代器，由两次调用 stable_partition() 返回。

- 还包括了一些辅助 lambda:

```
constexpr auto midit = [](auto& v) {
  return v.begin() + (v.end() - v.begin()) / 2;
};
constexpr auto is_even = [](auto i) {
  return i % 2 == 0;
};
constexpr auto is_even_char = [](auto c) {
  if(c >= '0' && c <= '9') return (c - '0') % 2 == 0;
  else return false;
};
```

这三个 lambda 的解释如下:

  - midit 返回容器中点的迭代器，用作枢轴点。

  - is_even 如果值为偶数则返回布尔值 true，用作谓词。
  - is_even_char 如果值是'0' 和'9' 之间的字符，则返回布尔值 true，并且是偶数，用于谓词。

- 可以在 main() 函数中调用 gather()，使用 int 的 vector:

```
int main() {
  vector<int> vint{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  auto gathered_even = bw::gather(vint.begin(),
    vint.end(), bw::midit(vint), bw::is_even);
  for(const auto& el : vint) cout << el;
  cout << '\n';
}
```

输出显示偶数聚集在了中间:

```
1302468579
```

gather() 函数返回一对只包含偶数值的迭代器:

```
auto& [it1, it2] = gathered_even;
for(auto it{ it1 }; it < it2; ++it) cout << *it;
cout << '\n';
```

输出为:

```
02468
```

- 可以将枢轴点设置为 begin() 或 end() 迭代器:

```
bw::gather(vint.begin(), vint.end(), vint.begin(),
  bw::is_even);
for(const auto& el : vint) cout << el;
cout << '\n';
bw::gather(vint.begin(), vint.end(), vint.end(),
  bw::is_even);
for(const auto& el : vint) cout << el;
cout << '\n';
```

输出为:

```
0246813579
1357902468
```

- 因为 gather() 是基于迭代器的，所以可以用于任何容器。这是一串字符数字:

```
string jenny{ "867-5309" };
bw::gather(jenny.begin(), jenny.end(), jenny.end(),
  bw::is_even_char);
```

```
for(const auto& el : jenny) cout << el;
cout << '\n';
```

这将把所有偶数移动到字符串的末尾，输出为：

```
7-539860
```

How it works…

gather() 函数使用 std::stable_partition() 算法将与谓词匹配的元素移动到枢轴点。gather() 有两个对 stable_partition() 的调用，一个带有谓词，另一个使用谓词取反结果:

```
template <typename It, typename Pred>
pair<It, It> gather(It first, It last, It pivot, Pred pred) {
  return { stable_partition(first, pivot, not_fn(pred)),
    stable_partition(pivot, last, pred) };
}
```

从两个 stable_partition() 调用返回的迭代器以 pair 的形式返回。

11.6. 删除连续的空格

当从用户接收输入时，通常会在字符串中出现过多的连续空格字符。本节提供了一个删除连续空格的函数，即使其中包含制表符或其他空白字符.

How to do it…

这个函数利用 std::unique() 算法，从字符串中移除连续的空白字符。

- bw 命名空间中，先从一个检测空白的函数开始:

```
template<typename T>
bool isws(const T& c) {
  constexpr const T whitespace[]{ " \t\r\n\v\f" };
  for(const T& wsc : whitespace) {
    if(c == wsc) return true;
  }
  return false;
}
```

这个模板化的 isws() 函数应该适用于任何字符类型。

- delws() 函数使用 std::unique() 来擦除字符串中连续的空格:

```
string delws(const string& s) {
  string outstr{s};
  auto its = unique(outstr.begin(), outstr.end(),
    [](const auto &a, const auto &b) {
      return isws(a) && isws(b);
```

```
    });
  outstr.erase(its, outstr.end());
  outstr.shrink_to_fit();
  return outstr;
}
```

delws() 复制输入字符串，删除连续的空格，并返回新字符串。

- 可以使用 main() 中的字符串调用:

```
int main() {
  const string s{ "big bad \t wolf" };
  const string s2{ bw::delws(s) };
  cout << format("[{}]\n", s);
  cout << format("[{}]\n", s2);
  return 0;
}
```

输出为:

```
[big   bad             wolf]
[big bad wolf]
```

How it works…

这个函数使用 std::unique() 算法和一个比较 lambda 来查找字符串对象中的连续空格。

比较 lambda 调用 isws() 函数来确定是否找到了连续的空格:

```
auto its = unique(outstr.begin(), outstr.end(),
  [](const auto &a, const auto &b) {
    return isws(a) && isws(b);
  });
```

可以使用标准库中的 isspace() 函数，但这是一个标准 C 函数，依赖于从 int 到 char 的窄类型转换。这可能会在一些现代 C++ 编译器上触发警告，并且不能保证在没有显式强制转换的情况下可以正常工作。我们的 isws() 函数使用了模板化的类型，应该适用于任何系统，以及 std::string 的任何特化。

11.7. 数字转换为单词

在我的职业生涯中，使用过很多编程语言。学习一门新语言时，我喜欢做一个项目，让我接触到语言的细微差别。numwords 类是我为此目的最喜欢的练习之一。多年来，我用数十种语言编写过它，包括多次用 C 和 C++ 进行练习。

numwords 是一个用单词拼写数字的类，可以用于银行和会计应用。使用起来像这样:

```
int main() {
  bw::numword nw{};
```

```
  uint64_t n;
  nw = 3; bw::print("n is {}, {}\n", nw.getnum(), nw);
  nw = 47; bw::print("n is {}, {}\n", nw.getnum(), nw);
  n = 100073; bw::print("n is {}, {}\n", n,
    bw::numword{n});
  n = 1000000001; bw::print("n is {}, {}\n", n,
    bw::numword{n});
  n = 123000000000; bw::print("n is {}, {}\n", n,
    bw::numword{n});
  n = 1474142398007; bw::print("n is {}, {}\n", n,
    nw.words(n));
  n = 999999999999999999; bw::print("n is {}, {}\n", n,
    nw.words(n));
  n = 1000000000000000000; bw::print("n is {}, {}\n", n,
    nw.words(n));
}
```

输出为:

```
n is 3, three
n is 47, forty-seven
n is 100073, one hundred thousand seventy-three
n is 1000000001, one billion one
n is 123000000000, one hundred twenty-three billion
n is 1474142398007, one trillion four hundred seventy-four
billion one hundred forty-two million three hundred ninetyeight thous
n is 999999999999999999, nine hundred ninety-nine quadrillion
nine hundred ninety-nine trillion nine hundred ninety-nine
billion nine hundred ninety-nine million nine hundred ninetynine thou
n is 1000000000000000000, error
```

How to do it…

这个示例起源于创建“生产就绪”的代码练习。因此，由三个不同的文件组成:

- numword.h 是 numwords 类的头/接口文件。
- numword.cpp 是 numwords 类的实现文件。
- numword-test.cpp 是用于测试数字词类的应用程序文件。

这个类本身大约有 180 行代码。我们在这里只讨论重点代码段，读者们可以在这里找到完整的源代码，https://github.com/PacktPublishing/CPP-20STL-Cookbook/tree/main/chap11/numword。

- numword.h 文件中，将类放在 bw 命名空间中，并以一些 using 语句开始:

```
namespace bw {
```

```
  using std::string;
  using std::string_view;
  using numnum = uint64_t;
  using bufstr = std::unique_ptr<string>;
```

整个代码中使用 string 和 string_view 对象。

uint64_t 是我们的主要整数类型，因为这个类称为 numword，所以我喜欢将 numnum 作为整数类型。

_bufstr 是主要的输出缓冲区，一个包装在 unique_ptr 中的字符串，遵从 RAII 的内存管理。

- 也有一些用于不同目的的常量:

```
constexpr numnum maxnum = 999'999'999'999'999'999;
constexpr int zero_i{ 0 };
constexpr int five_i{ 5 };
constexpr numnum zero{ 0 };
constexpr numnum ten{ 10 };
constexpr numnum twenty{ 20 };
constexpr numnum hundred{ 100 };
constexpr numnum thousand{ 1000 };
```

maxnum 常数对于大多数目的应该是足够的，其余部分用于避免在代码中使用文字。

- 主要的数据结构是 string_view 对象的 constexpr 数组，表示输出中使用的单词。因为其提供了最小开销的封装，所以 string_view 类非常适合这些常量:

```
constexpr string_view errnum{ "error" };
constexpr string_view _singles[] {
  "zero", "one", "two", "three", "four", "five",
  "six", "seven", "eight", "nine"
};
constexpr string_view _teens[] {
  "ten", "eleven", "twelve", "thirteen", "fourteen",
  "fifteen", "sixteen", "seventeen", "eighteen",
  "nineteen"
};
constexpr string_view _tens[] {
  errnum, errnum, "twenty", "thirty", "forty",
  "fifty", "sixty", "seventy", "eighty", "ninety",
};
constexpr string_view _hundred_string = "hundred";
constexpr string_view _powers[] {
  errnum, "thousand", "million", "billion",
  "trillion", "quadrillion"
};
```

这些单词可以分成几个部分，在将数字转换成单词时很有用。许多语言使用类似的分解，所以这个结构应该很适用于语言翻译。

- numword 类有几个私有成员:

```
class numword {
```

```
  bufstr _buf{ std::make_unique<string>(string{}) };
  numnum _num{};
  bool _hyphen_flag{ false };
```

  - _buf 是输出字符串缓冲区，内存由 unique_ptr 管理。
  - _num 保存当前数值。
  - _hyphen_flag 在翻译过程中，用于在单词之间插入连字符，而不是空格字符。

- 这些私有方法用于操作输出缓冲区。

```
void clearbuf();
size_t bufsize();
void appendbuf(const string& s);
void appendbuf(const string_view& s);
void appendbuf(const char c);
void appendspace();
```

还有一个 pow_i() 私有方法用于计算 x^y:

```
numnum pow_i(const numnum n, const numnum p);
```

pow_i() 用于区分字输出的数字值的各个部分。

- 公共接口包括构造函数和调用 words() 方法的各种方法，该方法完成将 numnum 转换为单词字符串的工作:

```
numword(const numnum& num = 0) : _num(num) {}
numword(const numword& nw) : _num(nw.getnum()) {}
const char * version() const { return _version; }
void setnum(const numnum& num) { _num = num; }
numnum getnum() const { return _num; }
numnum operator= (const numnum& num);
const string& words();
const string& words(const numnum& num);
const string& operator() (const numnum& num) {
  return words(num); };
```

- 实现文件 numword.cpp 中，大部分工作在 words() 成员函数中处理:

```
const string& numword::words( const numnum& num ) {
  numnum n{ num };
  clearbuf();
  if(n > maxnum) {
    appendbuf(errnum);
    return *_buf;
  }
  if (n == 0) {
    appendbuf(_singles[n]);
    return *_buf;
  }
  // powers of 1000
  if (n >= thousand) {
```

```
    for(int i{ five_i }; i > zero_i; --i) {
      numnum power{ pow_i(thousand, i) };
      numnum _n{ ( n - ( n % power ) ) / power };
      if (_n) {
        int index = i;
        numword _nw{ _n };
        appendbuf(_nw.words());
        appendbuf(_powers[index]);
        n -= _n * power;
      }
    }
  }
  // hundreds
  if (n >= hundred && n < thousand) {
    numnum _n{ ( n - ( n % hundred ) ) / hundred };
    numword _nw{ _n };
    appendbuf(_nw.words());
    appendbuf(_hundred_string);
    n -= _n * hundred;
  }
  // tens
  if (n >= twenty && n < hundred) {
    numnum _n{ ( n - ( n % ten ) ) / ten };
    appendbuf(_tens[_n]);
    n -= _n * ten;
    _hyphen_flag = true;
  }
  // teens
  if (n >= ten && n < twenty) {
    appendbuf(_teens[n - ten]);
    n = zero;
  }
  // singles
  if (n > zero && n < ten) {
    appendbuf(_singles[n]);
  }
  return *_buf;
}
```

该函数的每个部分以 10 的次幂的模量剥离数字的一部分，在千万的情况下递归，并从 string_view 常量数组中追加字符串。

- appendbuf() 有三个重载。一个添加字符串:

```
void numword::appendbuf(const string& s) {
  appendspace();
  _buf->append(s);
}
```

另一个添加 string_view:

```
void numword::appendbuf(const string_view& s) {
  appendspace();
  _buf->append(s.data());
}
```

第三个添加单个字符:

```
void numword::appendbuf(const char c) {
  _buf->append(1, c);
}
```

appendspace() 方法根据上下文附加一个空格字符或连字符:

```
void numword::appendspace() {
  if(bufsize()) {
    appendbuf( _hyphen_flag ? _hyphen : _space);
    _hyphen_flag = false;
  }
}
```

- numword-test.cpp 文件是 bw::numword 的测试环境，包括格式化特化:

```
template<>
struct std::formatter<bw::numword>:
std::formatter<unsigned> {
  template<typename FormatContext>
  auto format(const bw::numword& nw,
  FormatContext& ctx) {
    bw::numword _nw{nw};
    return format_to(ctx.out(), "{}",
    nw.words());
  }
};
```

可以直接将 bw::numword 对象传递给 format()。

- 还有一个 print() 函数，绕过 cout 和 iostream，直接将格式化输出发送到 stdout:

```
namespace bw {
  template<typename... Args> constexpr void print(
  const std::string_view str_fmt, Args&&...
  args) {
    fputs(std::vformat(str_fmt,
    std::make_format_args(args...)).c_str(),
    stdout);
  }
};
```

这就可以使用 print("{}\n"，nw)，而不是通过 cout 管道。C++23 标准中会有这样一个函数。

- main() 中，声明了一个 bw::numword 对象和用于测试的 uint64_t:

```
int main() {
  bw::numword nw{};
```

```
  uint64_t n{};

  bw::print("n is {}, {}\n", nw.getnum(), nw);
  ...
```

numword 对象初始化为 0，从 print() 语句中得到如下输出:

```
n is 0, zero
```

- 我们测试了多种调用 numword 的方法:

```
nw = 3; bw::print("n is {}, {}\n", nw.getnum(), nw);
nw = 47; bw::print("n is {}, {}\n", nw.getnum(), nw);
...
n = 100073; bw::print("n is {}, {}\n", n,
bw::numword{n});
n = 1000000001; bw::print("n is {}, {}\n", n,
bw::numword{n});
...
n = 474142398123; bw::print("n is {}, {}\n", n, nw(n));
n = 1474142398007; bw::print("n is {}, {}\n", n, nw(n));
...
n = 999999999999999999; bw::print("n is {}, {}\n", n,
nw(n));
n = 1000000000000000000; bw::print("n is {}, {}\n", n,
nw(n));
```

输出为:

```
n is 3, three
n is 47, forty-seven
...
n is 100073, one hundred thousand seventy-three
n is 1000000001, one billion one
...
n is 474142398123, four hundred seventy-four billion
one hundred forty-two million three hundred ninety-eight
thousand one hundred twenty-three
n is 1474142398007, one trillion four hundred seventyfour billion
ninety-eight thousand seven
...
n is 999999999999999999, nine hundred ninety-nine
quadrillion nine hundred ninety-nine trillion nine
hundred ninety-nine billion nine hundred ninety-nine
million nine hundred ninety-nine thousand nine hundred
ninety-nine
n is 10000000000000000000, error
```

How it works…

这个类主要是由数据结构驱动的。通过将 string_view 对象组织为数组，可以轻松地将标量值转换为相应的单词:

```
appendbuf(_tens[_n]); // e.g., _tens[5] = "fifty"
```

剩下的就是数学问题了:

```
numnum power{ pow_i(thousand, i) };
numnum _n{ ( n - ( n % power ) ) / power };
if (_n) {
  int index = i;
  numword _nw{ _n };
  appendbuf(_nw.words());
  appendbuf(_powers[index]);
  n -= _n * power;
}
```

There’s more…

我还有一个程序，使用 numwords 类以文字形式展示时间。其输出如下所示:

```
$ ./saytime
three past five
```

测试模式下，输出如下所示:

```
$ ./saytime test
00:00 midnight
00:01 one past midnight
11:00 eleven o'clock
12:00 noon
13:00 one o'clock
12:29 twenty-nine past noon
12:30 half past noon
12:31 twenty-nine til one
12:15 quarter past noon
12:30 half past noon
12:45 quarter til one
11:59 one til noon
23:15 quarter past eleven
23:59 one til midnight
12:59 one til one
13:59 one til two
01:60 OOR
24:00 OOR
```

我把其的实现留给读者们，作为本书最后的家庭作业。