

Jacek Galowicz

• 矩形截图(R)

C++17 STL Cookbook

Discover the latest enhancements to functional
programming and lambda expressions



Packt

Table of Contents

Introduction	1.1
前言	1.2
关于本书	1.3
各章梗概	1.4
第1章 C++17的新特性	1.5
使用结构化绑定来解包绑定的返回值	1.5.1
将变量作用域限制在if和switch区域内	1.5.2
新的括号初始化规则	1.5.3
构造函数自动推导模板的类型	1.5.4
使用constexpr-if简化编译	1.5.5
只有头文件的库中启用内联变量	1.5.6
使用折叠表达式实现辅助函数	1.5.7
第2章 STL容器	1.6
擦除/移除std::vector元素	1.6.1
以O(1)的时间复杂度删除未排序std::vector中的元素	1.6.2
快速或安全的访问std::vector实例的方法	1.6.3
保持对std::vector实例的排序	1.6.4
向std::map实例中高效并有条件的插入元素	1.6.5
了解std::map::insert新的插入提示语义	1.6.6
高效的修改std::map元素的键值	1.6.7
std::unordered_map中使用自定义类型	1.6.8
过滤用户的重复输入，并以字母序将重复信息打印出——std::set	1.6.9
实现简单的逆波兰表示法计算器——std::stack	1.6.10
实现词频计数器——std::map	1.6.11
实现写作风格助手用来查找文本中很长的句子——std::multimap	1.6.12
实现个人待办事项列表——std::priority_queue	1.6.13
第3章 迭代器	1.7
建立可迭代区域	1.7.1
让自己的迭代器与STL的迭代器兼容	1.7.2
使用迭代适配器填充通用数据结构	1.7.3
使用迭代器实现算法	1.7.4
使用反向迭代适配器进行迭代	1.7.5
使用哨兵终止迭代	1.7.6
使用检查过的迭代器自动化检查迭代器代码	1.7.7
构建zip迭代适配器	1.7.8

第4章 Lambda表达式	1.8
使用Lambda表达式定义函数	1.8.1
使用Lambda为std::function添加多态性	1.8.2
并置函数	1.8.3
通过逻辑连接创建复杂谓词	1.8.4
使用同一输入调用多个函数	1.8.5
使用std::accumulate和Lambda函数实现transform_if	1.8.6
编译时生成笛卡尔乘积	1.8.7
第5章 STL基础算法	1.9
容器间相互复制元素	1.9.1
容器元素排序	1.9.2
从容器中删除指定元素	1.9.3
改变容器内容	1.9.4
在有序和无序的vector中查找元素	1.9.5
将vector中的值控制在特定数值范围内——std::clamp	1.9.6
在字符串中定位模式并选择最佳实现——std::search	1.9.7
对大vector进行采样	1.9.8
生成输入序列的序列	1.9.9
实现字典合并工具	1.9.10
第6章 STL算法的高级使用方式	1.10
使用STL算法实现单词查找树类	1.10.1
使用树实现搜索输入建议生成器	1.10.2
使用STL数值算法实现傅里叶变换	1.10.3
计算两个vector的误差和	1.10.4
使用ASCII字符曼德尔布罗特集合	1.10.5
实现分割算法	1.10.6
将标准算法进行组合	1.10.7
删除词组间连续的空格	1.10.8
压缩和解压缩字符串	1.10.9
第7章 字符串, 流和正则表达	1.11
创建、连接和转换字符串	1.11.1
消除字符串开始和结束处的空格	1.11.2
无需构造获取std::string	1.11.3
从用户的输入读取数值	1.11.4
计算文件中的单词数量	1.11.5
格式化输出	1.11.6
使用输入文件初始化复杂对象	1.11.7
迭代器填充容器——std::istream	1.11.8

迭代器进行打印—— <code>std::ostream</code>	1.11.9
使用特定代码段将输出重定向到文件	1.11.10
通过集成 <code>std::char_traits</code> 创建自定义字符串类	1.11.11
使用正则表达式库标记输入	1.11.12
简单打印不同格式的数字	1.11.13
从 <code>std::iostream</code> 错误中获取可读异常	1.11.14
第8章 工具类	1.12
转换不同的时间单位—— <code>std::ratio</code>	1.12.1
转换绝对时间和相对时间—— <code>std::chrono</code>	1.12.2
安全的标识失败—— <code>std::optional</code>	1.12.3
对元组使用函数	1.12.4
使用元组快速构成数据结构	1.12.5
将 <code>void*</code> 替换为更为安全的 <code>std::any</code>	1.12.6
存储不同的类型—— <code>std::variant</code>	1.12.7
自动化管理资源—— <code>std::unique_ptr</code>	1.12.8
处理共享堆内存—— <code>std::shared_ptr</code>	1.12.9
对共享对象使用弱指针	1.12.10
使用智能指针简化处理遗留API	1.12.11
共享同一对象的不同成员	1.12.12
选择合适的引擎生成随机数	1.12.13
让STL以指定分布方式产生随机数	1.12.14
第9章 并行和并发	1.13
标准算法的自动并行	1.13.1
让程序在特定时间休眠	1.13.2
启动和停止线程	1.13.3
打造异常安全的共享锁—— <code>std::unique_lock</code> 和 <code>std::shared_lock</code>	1.13.4
避免死锁—— <code>std::scoped_lock</code>	1.13.5
同步并行中使用 <code>std::cout</code>	1.13.6
进行延迟初始化—— <code>std::call_once</code>	1.13.7
将执行的程序推到后台—— <code>std::async</code>	1.13.8
实现生产者/消费者模型—— <code>std::condition_variable</code>	1.13.9
实现多生产者/多消费者模型—— <code>std::condition_variable</code>	1.13.10
并行ASCII曼德尔布罗特渲染器—— <code>std::async</code>	1.13.11
实现一个小型自动化并行库—— <code>std::future</code>	1.13.12
第10章 文件系统	1.14
实现标准化路径	1.14.1
使用相对路径获取规范的文件路径	1.14.2
列出目录下的所有文件	1.14.3

实现一个类似grep的文本搜索工具	1.14.4
实现一个自动文件重命名器	1.14.5
实现一个磁盘使用统计器	1.14.6
计算文件类型的统计信息	1.14.7
实现一个工具：通过符号链接减少重复文件，从而控制文件夹大小	1.14.8

C++17 STL Cook book

函数式编程和Lambda表达式的最新功能

- 作者: Jacek Galowicz
- 译者: 陈晓伟 基于提交 52399cfc

本书主旨

- 了解C++最新的特性，使用标准库(STL)编写更优秀的代码，使用最新特性和STL节省开发上的时间开销。
- 了解STL特性所适用的范围和能力，并用其特性解决实际问题。
- 简洁优雅地使用STL实现算法。

本书概述

作为对《C++17 STL Cook book》的中文翻译。

C++因其快捷、高效和灵活的特点，帮助人们解决了很多问题，在很多领域种都有所使用。其将要到来的新版本，将会改变人们的编程习惯。如果想要掌握更加高明的编程方式，或是让代码更轻松地移植，就必须熟练掌握C++17 STL。本书将会通过实际例子帮助你了解C++17 STL，并掌握C++17 STL的使用方法。

本书将帮助你了解新版本的语言机制和标准库特性，并且告诉你他们如何工作。与众不同的是，我们会采用针对问题的特定解决方案，来帮助你克服使用方面的障碍。我们使用STL来解决实际问题，这样你就能了解到STL的核心，比如容器、算法、工具类、Lambda表达式，迭代器等等。这些实际问题的解决在展示如何更好编程的同时，帮助我们更多的了解STL。

看完本书后，你将了解到C++17最新的功能，并优雅地使用STL，且高效的解决难题。

将会学到

- 了解新语言的核心特性，以及这些特性所解决的问题。
- 通过实现迭代器来了解特性的需求，以及其内部工作流程。
- 探索算法、函数编程风格和Lambda表达式。
- 使用STL中提供的丰富、可移植、快速、久经考验、精心设计的算法。
- 使用STL中的字符串代替C风格的字符串。
- 了解支持并发和同步的标准类，以及如何使用
- 使用C++17 STL中的文件系统库

作者简介

Jacek Galowicz 在德国亚琛工业大学(Rheinisch-Westfälische Technische Hochschule Aachen University)获得电气工程/计算机工程硕士学位。在校期间，他特别喜欢以学生助教的身份参加教学和研究，并且在多项科技刊物发表文章。毕业后，他选择做一名自由执业者，并涉及很多领域，比如使用C和C++编写内核驱动、3维图像编程、数据库、网络通讯和物理模拟。近几年，他在Intel和FireEye平台上为Intel x86虚拟化编写性能和安全敏感的微内核操作系统，目前常驻于不伦瑞克(德国中北部城市，属下萨克森州)。他对使用最新的C++实现低层软件有着强烈的热情，并且努力地将高性能与优雅地编码风格相结合。近年来学习纯粹的函数式编程和Haskell的经历，让他更有动力(在元编程的帮助下)实现泛型编码。

作者鸣谢

感谢支持我翻译的各位同学们！

写书的同时也在创办一家公司，给我带来很多乐趣的同时，也是次很有趣的人生体验。这些有趣的经历来源于我身边的每一个人，感谢我可爱的女友给予我的耐心和支持，以及我公司的合伙人，当然还有支持我的所有朋友。这里要特别感谢Arne Mertz为我提供的宝贵的建议，当然还有Torsten Robitzki和来自于Oliver Bruns社区C++用户组的Hannover，感谢他们对本书的反馈。

代码评审

Arne Mertz是一个具有10多年C++经验的专家。他在汉堡大学攻读物理专业，而后转行成为一名软件开发攻城狮。其主要使用C++完成金融企业的应用程序。Arne就任于德国Zuhlike Engineering公司，并且它的博客也非常出名——[Simplify C++!](#)。对于C++，其主张在使用清爽，并具有良好可维护性的代码风格。

本书相关

- **github** 翻译地址：<https://github.com/xiaoweiChen/CPP-17-STL-cookbook>
- 本书源码：<https://github.com/PacktPublishing/Cpp17-STL-Cookbook>

前言

《C++17 STL Cookbook》将结合C++代码实例和标准库(STL)，教会你如何充分使用C++17。这里要说明的是，本书会尽可能的去使用STL，从而教会大家使用C++17。

C++是一门伟大且具有力量的语言。它使用简单的高级接口，将隐藏复杂的解决方案隐藏于背后，不过这样就意味着需要编写高效和低开销的底层代码实现。国际标准化组织(ISO)C++标准委员会致力于改进C++标准。C++11标准为C++带来了大量不错的特性，C++14和C++17也为C++加入了些新的特性。

目前为止，C++作为一门编程语言提供了语言相应的语言特性个标准库工具，用于处理复杂的标准数据结构和算法，包括：智能指针、Lambda表达式、常量表达式、便捷式可控线程的并发编程、正则表达式、随机数发生器、异常、可变参数模板(C++的部分模板类是图灵完备的!)、自定义文字、便捷式文件系统遍历等等。这些功能使它成为一种通用的语言，并在软件行业的所有领域，用于实现高质量和高性能的软件。

不过，很多编程者只将C++当做一门编程语言学习，而不太重视标准库(STL)的使用。不使用C++所带的标准库，将会让C++看起来就像是具有class的C语言，21世纪的现代化程序不应该写成这样。并且，这样的使用令人沮丧，就像是卸掉了它的一条手臂一样。

Bjarne Stroustrup(C++之父)在他的《C++程序设计语言》(C++11版本)中写到

请牢记，标准库和语言功能都是为了支撑以软件质量为目标的编程技术。他们应被组合起来发挥作用——如同建房子的砖块——而非个别地采用相对孤立地去解决某个特定问题。

这段话能很明确的概括我写这本书的目的。本书的所有例子都与实际息息相关，处理这些问题时，只依赖与STL，不依赖其他的库。少了其他库的依赖，就能很容易的将程序运行起来，不必去为开发环境所困扰。我希望你们受这些例子的启发，找到使用标准库的灵感，用伟大的编程语言作为解决更高级问题的基石。

关于本书

本书中所有的例子都很简单，都可以很容易编译和运行，不过读者们还是需要注意一下自己所选择的操作系统和编译器。下面就让我们来看一下在编译和运行本书例程时，所要注意的一些内容。

编译和运行例程

本书的所有例子都在Linux和Mac OS进行开发和验证，我们使用GNU的C++编译器 **g++**，和LLVM的C++编译器 **clang++**。

shell环境下可以使用如下的命令使用g++编译例程：

```
$ g++ -std c++1z -o recipe_app recipe_code.cpp
```

要使用clang++的话，命令行类似：

```
$ clang++ -std C++ 1z -o recipe_app recipe_code.cpp
```

上面两个例子都假设我们的C++例程写在 `recipe_code.cpp` 文件中。完成编译后，生成可执行二进制文件 `recipe_app`，然后使用如下命令执行它：

```
$ . ./recipe_app
```

书中很多例子，都是通过标准输入读取整个文件的内容。遇到这样的例子时，我们使用标准UNIX管道和 `cat` 命令直接将文件内容传输给我们的应用，命令如下所示：

```
$ cat file.txt | ./recipe_app
```

上面的方法适用于Linux和Mac OS系统。在微软Windows Shell中，需要使用如下的命令：

```
> recipe_app.exe < file.txt
```

如果你不想在Shell命令行中运行，你可以在Microsoft Visual Studio IDE中运行，不过需要你修改一下配置，"Configuration properties > Debugging"，并且添加 "`<file.txt`"，使用Visual Studio加载应用就能直接运行程序了。(Visual Studio IDE的话选定对应的解决方案，右键后选择“属性”，在“调试”页面输入相应的命令行参数)

前期准备

如果最近你阅读了本书中C++17的新特性，并使用前卫的编译器编译了这些代码，你可能会在编译阶段遇到一些问题。因为你使用到的一些C++17 STL新特性可能还没有在编译器中进行实现。

运行本书代码时，需要给 `<execution_policy>` 和 `<filesystem>` 头文件添加前缀 `experimental/`。其会将你将是用到的一些STL算法、数值等等包含入你的代码中，不过这也取决于编译器标准库的更新程度和稳定性。

这同样使用于命名空间的新特性。标准库中，实验部分的实现并不在 `std` 命名空间中，而是在 `std::experimental` 中。

适读群体

如果你没有编写过C++程序的经验，那么请将本书放回书架。如果你只想学习有关语言基础的知识，那么本书不是你理想的选择。当你了解完语言基础后，本书会对你的语言技巧进行升级。

除此之外，如果你符合如下的描述的话，可以继续阅读本书：

- 已经了解过C++的基础，不过现在你不知道下一步自己该怎么走，这是因为你与资深C++达人还有很大的差距。
- C++基础十分牢靠，但是你对STL知之甚少。
- 对C++的某个老版本比较了解，比如C++98、C++11或C++14。

以上这些描述，都是基于你使用C++的频度而论。本书储备了很多优秀的STL新特性，等待你去发现。

章节设计

本书中你会发现几个经常出现的标题：

(译者：这些副标题只在本节翻译，正文中使用英文原文作为副标题)

- Getting ready
- How to do it
- How it works
- There's more
- See also

下面简单介绍一些这几个副标题所涵盖的内容：

准备开始 **Getting ready**

本节会说明我们的期望，以及如何在初期对环境或软件进行配置。

如何完成 **How to do it...**

本节包含实现所需的步骤。

如何工作 **How it works...**

本节会对前一节所发生的事情，进行详细解释。

信息补充 **There's more...**

本节包含了一些式例相关的补充信息，以便读者对式例有更深入的了解。

更多信息 See also

为式例提供一些帮助链接，有助于了解C++的更多知识。

文本样式

~~本书中，使用不同的文本样式区分不同种类的信息。下面的一些例子会解释这些风格的含义。~~

~~文本的代码，数据库表名，文件夹名，文件名，文件的扩展名，路径名，虚拟的URL，用户输入和推特引用，会展示成这种样式。“下一步需要修改build.properties文件。”~~

代码块为这种样式：

```
my_wrapper<T1, T2, T3> make_wrapper (T1 t_1, T2 t2, T3  
t3)  
{  
    return t_1, t2, t3;  
}
```

~~新术语和关键字使用粗体。你在屏幕上看到的单词，例如菜单或对话框，会是这种样式：“完成后，点击执行。”~~

~~警告或重要说明会显示在一个方框中。~~

~~提示和技巧会用斜体样式~~

读者反馈

我们欢迎读者的反馈。这样我们就知道这本书哪里好，哪里不好。读者的反馈对于我们来说十分重要，并且能帮助我们确定读者关注的重点，从而让读者在阅读本书时的收获最大化。一般的反馈可以通过发送电子邮件到 feedback@packtpub.com，并在主题中提到这本书的名字即可。如果您是某个方便的专家，并且对写作或写书感兴趣的话，可以了解一下我们的作者指南www.packtpub.com/authors。

客户支持

现在您已经是本书的主人，我们会为您购买本书的行为，提供相应的支持服务。

源码下载

可使用您在 <http://www.packtpub.com> 的账号下载本书式例代码。如果您在别处购买了本书，可以通过访问 <http://wmv.packtpub.com/support>，客服会将注册文件直接发送给您。

您可以按照以下步骤下载代码：

- 网页端使用您的电子邮件地址和密码进行登录或注册。将鼠标悬停在“SUPPORT”标签上。点击“Downloads & Errata”。搜索框内键入本书的名字。选择你所查找的书籍，并下载其代码包。只需您在购买本书的下拉菜单中点击“Code Download”即可。
- 压缩包下载完毕后，请确认您所使用的解压缩软件的版本和所解压的文件夹地址。Windows：WinRAR / 7-Zip；Mac：Zipeg / iZip / UnRarX；Linux：7-Zip / PeaZip。

本书代码github的托管地址为 <https://github.com/PacktPublishing/Cpp17-STL-Cookbook>

其他书籍的代码包和视频目录在 <https://github.com/PacktPublishing/> 下都能看到。

快去看一下吧！

勘误列表

尽管我们很认真的保证本书内容的正确性，但难免还是会出现错误。如果您在我们的书或代码中发现了疑似错误的地方，请反馈给我们，我们将感激不尽。如果这真是个错误，我们将在后续的版本中修复这个问题，以免误导更多的读者。如果您发现了任何错误，请访问 <http://mwv.packtpub.com/submit-errata> 选择本书，点击勘误提交的链接，然后详述你发现的问题。当您的勘误得到了验证，您的勘误将会记录在我们的勘误列表上。

想要了解之前的勘误列表，可以在 <https://mwv.packtpub.com/books/content/support> 上面输入书籍的名字查找对应的勘误列表。想要看到的内容将会出现在勘误栏下。

盗版必究

互联网上存在着盗版问题。Packt非常重视我们的版权和许可证。如果您在网上发现我们的作品的非法副本，请提供地址或网站名称，以便我们进行维权。

请通过 copyright@packtpub.com 联系我们，麻烦在邮件内附上与涉嫌盗版的相关资料。感谢您帮助我们保护相关作品的只是产权。

问题解答

如果您对本书有任何的问题，您可以通过向 questions@packtpub.com 发送邮件告诉我们，我们会尽可能的解答您所提出的问题。

各章梗概

第1章，C++17新特性。介绍那些对C++语言来说很重大的改变，以便后续的章节中将精力集中在STL上。

第2章，STL容器。STL容器在C++17标准中进行了升级，让我们见识一下STL容器的数据类型是多么的丰富。粗略的了解一下容器后，再仔细了解其添加的内容。

第3章，迭代器。迭代器是STL中很重要的概念，其将STL算法和容器数据类型二者紧密联系在一起。我们将用实际例子来了解如何使用迭代器，从而更好的了解迭代器的概念。

第4章，Lambda表达式。这是一种很有意思的编程模式，其为纯函数式编程的方式。C++11标准引入Lambda表达式，C++14和C++17标准为其添加了一些新特性。

第5章，STL基础算法。介绍了STL的标准算法的特点，简单易用、高效、鲁棒性好和高度通用。我们将学习如何使用它们，这样就可以集中精力在解决问题上，而不是浪费时间去重新发明轮子。

第6章，STL算法的高级使用方式。演示如何通过使用STL基本算法，以更简洁的方式编写更复杂的算法，而无需重复代码。本章中，充分利用STL解决更复杂问题的同时，将学习如何结合现有的算法，来创建真正符合需求的新算法。

第7章，字符串，流和正则表达。对STL中关于字符串、通用I/O流和正则表达式的类型进行详细概述。

第8章，工具类。了解STL如何生成随机数、测量时间、管理动态内存、优雅地提示错误等等。我们会来了解一下这些极为有用、高可移植性的工具类，并且会介绍C++17带来的全新STL工具。

第9章，并行和并发。多处理器领域编写代码时，并行和并发就变得很重要。

C++11标准首先引入并行和并发的概念，随后C++17进行加强，这对于我们编写并发程序来说有很大的帮助。

第10章，文件系统。虽然之前的STL提供对单个文件读取和操作，但这还无法达到用户的需求。C++17添加了很多新的操作(独立于操作系统库)用于处理文件系统路径，以及对目录进行遍历。

第1章 C++17的新特性

C++11, C++14和C++17标准为C++添加了许多新特性。当前的C++已经和10年前的C++完全不同了。C++标准并不是用来规范语言的，其实为了让相应编译器理解相应的语义，也是为了更好的理解C++标准模板库(STL)。

这本书中的例子展示了如何充分的利用STL。不过，作为本书的第1章，我们还是需要了解一下那些比较重要的新语言特性。掌握了这些新的语言特性，有助于你编写可读性高、可维护性强和表达性清晰的代码。

我们将了解到如何单独访问组对、元组和结构化绑定的数据结构的成员，以及如何使用新的 `if` 和 `switch` 限制变量的作用范围。新的括号初始化语法于C++11的语法有歧义，虽然看上去是相同的，不过这个已经被新括号初始化规则所修复。模板类实例的类型现在可以从构造函数的参数中自动推断出来，如果对一个模板类进行不同类型的特化，将会产生完全不同的代码，不过现在用 `constexpr-if` 就能很容易的表示。大多数情况下，使用折叠表达式处理模板函数的可变参数包，会变得更加容易。最后，在只有头文件的库中使用声明内联变量，来定义全局静态对象会变得更加舒服，这之前的標準中只能在函数中进行。

库的实现者可能比实现应用程序的开发者对本章的示例更感兴趣。虽然我们有足够的理由去了解这些特性，但为了理解本书的其余部分，无需立即理解本章的所有示例。

使用结构化绑定来解包绑定的返回值

C++17配备了一种新的特性——结构化绑定，其可以结合语法糖来自动推到类型，并可以从组对、元组和结构体中提取单独的变量。其他编程语言中，这种特性也被成为解包。

How to do it...

使用结构化绑定是为了能够更加简单的，为绑定了多个变量的结构体进行赋值。我们先来看下在C++17标准之前是如何完成这个功能的。然后，我们将会看到一些使用C++17实现该功能的例子：

- 访问 `std::pair` 中的一个元素：假设我们有一个数学函数 `divide_remainder`，需要输入一个除数和一个被除数作为参数，返回得到的分数的整数部分和余数。可以使用一个 `std::pair` 来绑定这两个值：

```
std::pair<int, int> divide_remainder(int dividend, int divisor);
```

考虑使用如下的方式访问组对中的单个值：

```
const auto result (divide_remainder(16, 3));
std::cout << "16 / 3 is "
    << result.first << " with a remainder of "
    << result.second << '\n';
```

与上面的代码段不同，我们现在可以将相应的值赋予对应的变量，这样写出来的代码可读性更高：

```
auto [fraction, remainder] = divide_remainder(16, 3);
std::cout << "16 / 3 is "
    << fraction << " with a remainder of "
    << remainder << '\n';
```

- 也能对 `std::tuple` 进行结构化绑定：让我们使用下面的实例函数，获取股票的在线信息：

```
std::tuple<std::string,
std::chrono::system_clock::time_point, unsigned>
stock_info(const std::string &name);
```

我们可以使用如下的方式获取这个例子的各个变量的值：

```
const auto [name, valid_time, price] =
stock_info("INTC");
```

- 结构化绑定也能用在自定义结构体上。假设有这么一个结构体：

```
struct employee{
    unsigned id;
    std::string name;
    std::string role;
    unsigned salary;
};
```

现在我们来看下如何使用结构化绑定访问每一个成员。我们假设有一组 `employee` 结构体的实例，存在于 `vector` 中，下面使用循环将其内容进行打印：

```
int main() {
    std::vector<employee> employees{
        /* Initialized from somewhere */
    };

    for (const auto &[id, name, role, salary] :
employees) {
        std::cout << "Name: " << name
            << "Role: " << role
            << "Salary: " << salary << '\n';
    }
}
```

How it works...

结构化绑定以以下方式进行应用：

```
auto [var1, var2, ...] = <pair, tuple, struct, or array expression>;
```

- `var1, var2, ...` 表示一个变量列表，其变量数量必须匹配表达式所对应的结构。
- `<pair, tuple, struct, or array expression>` 必须是下面的其中一种：
 - 一个 `std::pair` 实例。
 - 一个 `std::tuple` 实例。
 - 一个结构体实例。其所有成员都必须是非静态成员，每个成员以基础类定义。结构体中的第一个声明成员赋予第一个变量的值，第二个声明的编程赋予第二个变量的值，依次类推。
 - 固定长度的数组。
- `auto` 部分，也就是 `var` 的类型，可以是 `auto`, `const auto`, `const auto&` 和 `auto&&`。

Note:

不仅为了性能，还必须确保在适当的时刻使用引用，尽量减少不必要的副本。

如果中括号中变量不够，那么编译器将会报错：

```
std::tuple<int, float, long> tup(1, 2.0, 3);
auto [a, b] = tup; // Does not work
```

这个例子中想要将三个成员值，只赋予两个变量。编译器会立即发现这个错误，并且提示我们：

```
error: type 'std::tuple<int, float, long>' decomposes
into 3 elements, but only 2 names were provided
auto [a, b] = tup;
```

There's more...

STL中的基础数据结构都能通过结构化绑定直接进行访问，而无需修改任何东西。考虑下面这个例子，循环中打印 `std::map` 中的元素：

```
std::map<std::string, size_t> animal_population {
    {"humans", 7000000000},
    {"chickens", 17863376000},
    {"camels", 24246291},
    {"sheep", 1086881528},
    /* ... */
};

for (const auto &[species, count] : animal_population) {
    std::cout << "There are " << count << " " << species
        << " on this planet.\n";
}
```

从 `std::map` 容器中获取元素的方式比较特殊，我们会在每次迭代时获得一个 `std::pair<const key_type, value_type>` 实例。另外每个实例都需要进行结构化绑定（`key_type` 绑定到 `species` 字符串上，`value_type` 为一个 `size_t` 格式的统计数字），从而达到访问每一个成员的目的。

在C++17之前，使用 `std::tie` 可达到类似的效果：

```
int remainder;
std::tie(std::ignore, remainder) = divide_remainder(16,
    5);
std::cout << "16 % 5 is " << remainder << '\n';
```

这个例子展示了如何将结果组对解压到两个变量中。`std::tie` 的能力远没有结构化绑定强，因为在进行赋值的时候，所有变量需要提前定义。另外，本例也展示了一种在 `std::tie` 中有，而结构化绑定没有的功能：可以使用 `std::ignore` 的值，作为虚拟变量。分数部分将会赋予到这个虚拟变量中，因为这里我们不需要用到分数值，所以使用虚拟变量忽略分数值。

Note:

使用结构化绑定时，就不能再使用`std::tie`创建虚拟变量了，所以我们不得不绑定所有值到命名过的变量上。对部分成员进行绑定的做法是高效的，因为编译器可以很容易的对未绑定的变量进行优化。

回到之前的例子，`divide_remainder` 函数也可以通过使用传入输出参数的方式进行实现：

```
bool divide_remainder(int dividend, int divisor, int
&fraction, int &remainder);
```

调用该函数的方式如下所示：

```
int fraction, remainder;
const bool success {divide_remainder(16, 3, fraction,
remainder)};
if (success) {
    std::cout << "16 / 3 is " << fraction << " with a
remainder of "
    << remainder << '\n';
}
```

很多人都很喜欢使用特别复杂的结构，比如组对、元组和结构体，他们认为这样避免了中间拷贝过程，所以代码会更快。对于现代编译器来说，这种想法不再是正确的了，这里编译器并没有刻意避免拷贝过程，而是优化了这个过程。(其实拷贝过程还是存在的)。

Note:

与C的语法特征不同，将复杂结构体作为返回值传回会耗费大量的时间，因为对象需要在返回函数中进行初始化，之后将这个对象拷贝到相应容器中返回给调用端。现代编译器支持[返回值优化\(RVO, return value optimization\)](#)技术，这项技术可以省略中间副本的拷贝。

将变量作用域限制在**if**和**switch**区域内

将变量的生命周期尽可能的限制在指定区域内，是一种非常好的代码风格。有时我们需要在满足某些条件时获得某个值，然后对这个值进行操作。

为了让这个过程更简单，C++17中为**if**和**switch**配备了初始化区域。

How to do it...

这个案例中，我们使用初始化语句，来了解下其使用方式：

- **if**：假设我们要在一个字母表中查找一个字母，我们 `std::map` 的成员 `find` 完成这个操作：

```
if (auto itr (character_map.find(c)); itr !=  
character_map.end()) {  
    // *itr is valid. Do something with it.  
} else {  
    // itr is the end-iterator. Don't dereference.  
}  
// itr is not available here at all
```

- **switch**：这个例子看起来像是从玩家输入的字母决定某个游戏中的行为。通过使用 `switch` 查找字母相对应的操作：

```
switch (char c (getchar()); c) {  
    case 'a': move_left(); break;  
    case 's': move_back(); break;  
    case 'w': move_fwd(); break;  
    case 'd': move_right(); break;  
    case 'q': quit_game(); break;  
    case '0'...'9': select_tool('0' - c); break;  
    default:  
        std::cout << "invalid input: " << c << '\n';  
}
```

How it works...

带有初始化的 `if` 和 `switch` 相当于语法糖一样。

```
// if: before C++17
{
    auto var(init_value);
    if (condition){
        // branch A. var is accessible
    } else {
        // branch B. var is accessible
    }
    // var is still accessible
}
```

```
// if: since C++17
if (auto var (init_value); condition){
    // branch A. var is accessible
} else {
    // branch B. var is accessible
}
// var is not accessible any longer
```

```
// switch: before C++17
{
    auto var (init_value);
    switch (var) {
        case 1: ...
        case 2: ...
        ...
    }
    // var is still accessible
}
```

```
// switch: since C++17
switch(auto var (init_value); var){
    case 1: ...
    case 2: ...
    ...
}
// var is not accessible any longer
```

这些有用的特性保证了代码的简洁性。C++17之前只能使用外部括号将代码包围，就像上面的例子中展示的那样。减短变量的生命周期，能帮助我们保持代码的整洁性，并且更加易于重构。

There's more...

另一个有趣的例子是临界区限定变量生命周期。

先来看个栗子：

```
if (std::lock_guard<std::mutex> lg {my_mutex};  
    some_condition) {  
    // Do something  
}
```

首先，创建一个 `std::lock_guard`。这个类接收一个互斥量和作为其构造函数的参数。这个类在其构造函数中对互斥量上锁，之后当代码运行完这段区域后，其会在析构函数中对互斥量进行解锁。这种方式避免了忘记解锁互斥量而导致的错误。

C++17之前，为了确定解锁的范围，需要一对额外的括号对。

另一个例子中对弱指针进行区域限制：

```
if (auto shared_pointer (weak_pointer.lock());  
    shared_pointer != nullptr) {  
    // Yes, the shared object does still exist  
} else {  
    // shared_pointer var is accessible, but a null pointer  
}  
// shared_pointer is not accessible any longer
```

这个例子中有一个临时的 `shared_pointer` 变量，虽然 `if` 条件块或外部括号会让其保持一个无用的状态，但是这个变量确实会“泄漏”到当前范围内。

当要使用传统API的输出参数时，`if` 初始化段就很有用：

```
if (DWORD exit_code; GetExitCodeProcess(process_handle,  
    &exit_code)) {  
    std::cout << "Exit code of process was: " << exit_code  
    << '\n';  
}  
// No useless exit_code variable outside the if-  
conditional
```

`GetExitCodeProcess` 函数是Windows操作系统的内核API函数。其通过返回码来判断给定的进程是否合法的处理完成。当离开条件域，变量就没用了，也就可以销毁这个变量了。

具有初始化段的 `if` 代码块在很多情况下都特别有用，尤其是在使用传统API的输出参数进行初始化时。

Note:

使用带有初始化段的 `if` 和 `switch` 能保证代码的紧凑性。这使您的代码紧凑，更易于阅读，在重构过程中，会更容易改动。

新的括号初始化规则

C++11引入了新的括号初始化语法 {}。其不仅允许集合式初始化，而且还是对常构造函数的调用。遗憾的是，当与 auto 类型变量结合时，这种方式就很容易出现错误。C++17将会增强这一系列初始化规则。本节中，我们将了解到如何使用 C++17 语法规正确的初始化变量。

How to do it...

一步初始化所有变量。使用初始化语法时，注意两种不同的情况：

- 不使用auto声明的括号初始化：

```
// Three identical ways to initialize an int:  
int x1 = 1;  
int x2{1};  
int x3(1);  
  
std::vector<int> v1{1, 2, 3}; // Vector with three ints  
std::vector<int> v2 = {1, 2, 3}; // same here  
std::vector<int> v3(10, 20); // Vector with 10 ints, each  
have value 20
```

- 使用auto声明的括号初始化：

```
auto v {1}; // v is int  
auto w {1, 2}; // error: only single elements in direct  
// auto initialization allowed! (this is  
new)  
auto x = {1}; // x is std::initializer_list<int>  
auto y = {1, 2}; // y is std::initializer_list<int>  
auto z = {1, 2, 3.0}; // error: Cannot deduce element  
type
```

How it works...

无 auto 类型声明时，{} 的操作没什么可大惊小怪的。当在初始化 STL 容器时，例如 std::vector，std::list 等等，括号初始化就会去匹配 std::initializer_list (初始化列表) 的构造函数，从而初始化容器。其构造函数会使用一种“贪婪”的方式，这种方式就意味着不可能匹配非聚合构造函数(与接受初始化列表的构造函数相比，非聚合构造函数是常用构造函数)。

std::vector 就提供了一个特定的非聚合构造函数，其会使用任意个相同的数值填充 vector 容器：std::vector<int> v(N, value)。当写成 std::vector<int> v{N, value} 时，就选择使用 initializer_list 的构造函数进行初始化，其会将 vector 初

初始化成只有N和value两个元素的变量。这个“陷阱”大家应该都知道。

{ } 与 () 调用构造函数初始化的方式，不同点在于 { } 没有类型的隐式转换，比如 `int x(1.2);` 和 `int x = 1.2;` 通过静默的对浮点值进行向下取整，然后将其转换为整型，从而将x的值初始化为1。相反的，`int x{1.2};` 将会遇到编译错误，初始化列表中的初始值，需要与变量声明的类型完全匹配。

Note:

哪种方式是最好的初始化方式，目前业界是有争议的。括号初始化的粉丝们提出，使用括号的方式非常直观，直接可以调用构造函数对变量进行初始化，并且代码行不会做多余的事情。另外，使用{}括号将会是匹配构造函数的唯一选择，这是因为使用()进行初始化时，会尝试匹配最符合条件的构造函数，并且还会对初始值进行类型转换，然后进行匹配(这就会有处理构造函数二义性的麻烦)。

C++17添加的条件也适用于auto(推断类型)——C++11引入，用于正确的推导匹配变量的类型。`auto x{123};` 中 `std::initializer_list<int>` 中只有一个元素，这并不是我们想要的结果。C++17将会生成一个对应的整型值。

经验法则：

- `auto var_name {one_element};` 将会推导出var_name的类型——与one_element一样。
- `auto var_name {element1, element2, ...};` 是非法的，并且无法通过编译。
- `auto var_name = {element1, element2, ...};` 将会使用 `std::initializer_list<T>` 进行初始化，列表中elementN变量的类型均为T。

C++17加强了初始化列表的鲁棒性。

Note:

使用C++11/C++14模式的编译器解决这个问题时，有些编译器会将 `auto x{123};` 的类型推导成整型，而另外一些则会推导成 `std::initializer_list<int>`。所以，这里需要特别注意，编写这样的代码，可能会导致有关可移植性的问题！

构造函数自动推导模板的类型

C++中很多类都需要指定类型，其实这个类型可以从用户所调用的构造函数中推导出来。不过，在C++17之前，这是一个未标准化的特性。C++17能让编译器自动的从所调用的构造函数，推导出模板类型。

How to do it...

使用最简单的方法创建 `std::pair` 和 `std::tuple` 实例。其可以实现一步创建。

```
std::pair my_pair (123, "abc"); // std::pair<int, const
                                char*>
std::tuple my_tuple (123, 12.3, "abc"); // std::tuple<int, double, const char*>
```

How it works...

让我们定义一个类，了解自动化的对模板类型进行推断的价值。

```
template <typename T1, typename T2, typename T3>
class my_wrapper {
    T1 t1;
    T2 t2;
    T3 t3;
public:
    explicit my_wrapper(T1 t1_, T2 t2_, T3 t3_)
        : t1{t1_}, t2{t2_}, t3{t3_}
    {}
    /* ... */
};
```

好！我们定义了一个模板类。C++17之前，我们为了创建该类的实例：

```
my_wrapper<int, double, const char *> wrapper {123, 1.23,
                                              "abc"};
```

我们省略模板特化的部分：

```
my_wrapper wrapper {123, 1.23, "abc"};
```

C++17之前，我们可能会通过以下的方式实现一个工厂函数：

```
my_wrapper<T1, T2, T3> make_wrapper(T1 t1, T2 t2, T3 t3)
{
    return {t1, t2, t3};
}
```

使用工厂函数：

```
auto wrapper (make_wrapper(123, 1.23, "abc"));
```

Note:

STL中有很多工厂函数，比如 `std::make_shared`、`std::make_unique`、`std::make_tuple` 等等。C++17中，这些工厂函数就过时了。当然，考虑到兼容性，这些工厂函数在之后还会保留。

There's more...

我们已经了解过隐式模板类型推导。但一些例子中，不能依赖类型推导。如下面的例子：

```
// example class
template <typename T>
struct sum{
    T value;

    template <typename ... Ts>
    sum(Ts&& ... values) : value{ (values + ...) } {}
};
```

结构体中，`sum` 能接受任意数量的参数，并使用折叠表达式将它们添加到一起(本章稍后的一节中，我们将讨论折叠表达式，以便了解折叠表达式的更多细节)。加法操作后得到的结果保存在 `value` 变量中。现在的问题是，`T` 的类型是什么？如果我们不显式的进行指定，那就需要通过传递给构造函数的变量类型进行推导。当我们提供了多个字符串实例，其类型为 `std::string`。当我们提供多个整型时，其类型就为 `int`。当我们提供多个整型、浮点和双浮点时，编译器会确定哪种类型适合所有的值，而不丢失信息。为了实现以上的推导，我们提供了指导性显式推导：

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

指导性推导会告诉编译器使用 `std::common_type_t` 的特性，其能找到适合所有值的共同类型。来看下如何使用：

```
sum s {1u, 2.0, 3, 4.0f};  
sum string_sum {std::string{"abc"}, "def"};  
std::cout << s.value << '\n'  
      << string_sum.value << '\n';
```

第1行中，我们创建了一个 `sum` 对象，构造函数的参数类型为 `unsigned`，`double`，`int` 和 `float`。`std::common_type_t` 将返回 `double` 作为共同类型，所以我们获得的是一个 `sun<double>` 实例。第2行中，我们创建了一个 `std::string` 实例和一个C风格的字符串。在我们的指导下，编译器推导出这个实例的类型为 `sum<std::string>`。

当我们运行这段代码时，屏幕上会打印出10和abcdef。其中10为数值 `sum` 的值，`abcdef`为字符串 `sum` 的值。

使用constexpr-if简化编译

模板化编程中，通常要以不同的方式做某些事情，比如特化模板类型。C++17带了 `constexpr-if` 表达式，可以在很多情况下简化代码。

How to do it...

本节中，我们会实现一个很小的辅助模板类。它能处理不同模板类型的特化，因为它可以在完全不同的代码中，选取相应的片段，依据这些片段的类型对模板进行特化：

- 完成代码中的通用部分。在我们的例子中，它是一个简单的类，它的成员函数 `add`，支持对 `u` 类型值与 `t` 类型值的加法：

```
template <typename T>
class addable
{
    T val;
public:
    addable(T v) : val(v) {}
    template <typename U>
    T add(U x) const {
        return val + x;
    }
};
```

- 假设类型 `t` 是 `std::vector<something>`，而类型 `u` 是 `int`。这里就有问题了，为整个 `vector` 添加整数是为了什么呢？其应该是对 `vector` 中的每个元素加上一个整型数。实现这个功能就需要在循环中进行：

```
template <typename U>
T add(U x)
{
    auto copy (val); // Get a copy of the vector member
    for (auto &n : copy) {
        n += x;
    }
    return copy;
}
```

- 下一步也是最后一步，将两种方式结合在一起。如果 `t` 类型是一个 `vector`，其每个元素都是 `u` 类型，择进行循环。如果不是，则进行普通的加法：

```

template <typename U>
T add(U x) const{
    if constexpr(std::is_same<T,
std::vector<U>>::value) {
        auto copy(val);
        for (auto &n : copy) {
            n += x;
        }
        return copy;
    } else {
        return val + x;
    }
}

```

4. 现在就可以使用这个类了。让我们来看下其对不同类型处理的是多么完美，下面的例子中有 `int` , `float` , `std::vector<int>` 和 `std::vector<string>` :

```

addable<int> {1}.add(2); // is 3
addable<float> {1.f}.add(2); // is 3.0
addable<std::string> {"aa"}.add("bb"); // is "aabb"

std::vector<int> v{1, 2, 3};
addable<std::vector<int>> {v}.add(10); // is
std::vector<int> {11, 12, 13}

std::vector<std::string> sv{"a", "b", "c"};
addable<std::vector<std::string>>
{sv}.add(std::string("z")); // is {"az", "bz", "cz"}

```

How it works...

新特性 `constexpr-if` 的工作机制与传统的 `if-else` 类似。不同点就在于前者在编译时进行判断，后者在运行时进行判断。所以，使用 `constexpr-if` 的代码在编译完成后，程序的这一部分其实就不会有分支存在。有种方式类似于 `constexpr-if`，那就是 `#if-#else` 的预编译方式进行宏替换，不过这种方式在代码的构成方面不是那么优雅。组成 `constexpr-if` 的所有分支结构都是优雅地，没有使用分支在语义上不要求合法。

为了区分是向 `vector` 的每个元素加上`x`，还是普通加法，我们使用 `std::is_same` 来进行判断。表达式 `std::is_same<A, B>::value` 会返回一个布尔值，当A和B为同样类型时，返回`true`，反之返回`false`。我们的例子中就写为 `std::is_same<T, std::vector<U>>::value()` (`is_same_v = is_same<T, U>::value;`)，当返回为`true`时，且用户指定的T为 `std::vector<X>`，之后试图调用`add`，其参数类型 `U = X`。

当然，在一个 `constexpr-if-else` 代码块中，可以有多个条件(注意：`a`和`b`也可以依赖于模板参数，并不需要其为编译时常量)：

```

if constexpr(a) {
    // do something
} else if constexpr(b) {
    // do something else
} else {
    // do something completely different
}

```

C++17中，很多元编程的情况更容易表达和阅读。

There's more...

这里对比一下C++17之前的实现和添加 `constexpr-if` 后的实现，从而体现出这个特性的加入会给C++带来多大的提升：

```

template <typename T>
class addable{
    T val;
public:
    addable(T v):val{v}{}

    template <typename U>
    std::enable_if_t<!std::is_same<T,
    std::vector<U>>::value, T>
    add(U x) const {
        return val + x;
    }

    template <typename U>
    std::enable_if_t<!std::is_same<T,
    std::vector<U>>::value, std::vector<U>>
    add (U x) const{
        auto copy(val);
        for (auto &n: copy){
            n += x;
        }
        return copy;
    }
};

```

在没有了 `constexpr-if` 的帮助下，这个类看起特别复杂，不像我们所期望的那样。怎么使用这个类呢？

简单来看，这里重载实现了两个完全不同的 `add` 函数。其返回值的类型声明，让这两个函数看起里很复杂；这里有一个简化的技巧——表达式，例如 `std::enable_if_t<condition, type>`，如果条件为真，那么就为 `type` 类型，反

之 `std::enable_if_t` 表达式不会做任何事。这通常被认为是一个错误，不过我们能解释为什么什么都没做。

对于第二个 `add` 函数，相同的判断条件，但是为反向。这样，在两个实现不能同时为真。

当编译器看到具有相同名称的不同模板函数并不得不选择其中一个时，一个重要的原则就起作用了：替换失败不是错误(**SFINAE**, **Substitution Failure is not An Error**)。这个例子中，就意味着如果函数的返回值来源一个错误的模板表示，无法推断得出，这时编译器不会将这种情况视为错误(和 `std::enable_if` 中的条件为`false`时的状态一样)。这样编译器就会去找函数的另外的实现。

很麻烦是吧，C++17中实现起来就变得简单多了。

只有头文件的库中启用内联变量

这种库在声明函数时，始终是内联的，C++17中允许声明内联变量。C++17之前只能使用其他变通的方法实现内联变量，新标准的支持让实现只有头文件的库更加的容易。

How it's done...

本节中，我们创建一个类，可以作为典型头文件库的成员。其目的就是给定一个静态成员，然后使用 `inline` 关键字对其进行修饰，使得其实例在全局范围内都能访问到，在C++17之前这样做是不可能的。

1. `process_monitor` 类必须包含一个静态成员，并且能全局访问。当该单元被重复包含时，会产生符号重定义的问题。

```
// foo_lib.hpp
class process_monitor {
public:
    static const std::string standard_string{
        "some static globally available string"};
};

process_monitor global_process_monitor;
```

2. 多个 `.cpp` 文件中包含这个头文件时，链接阶段会出错。为了修复这个问题，添加了 `inline` 关键字：

```
// foo_lib.hpp
class process_monitor {
public:
    static const inline std::string standard_string{
        "some static globally available string"};
};

inline process_monitor global_process_monitor;
```

瞧，就是这样！

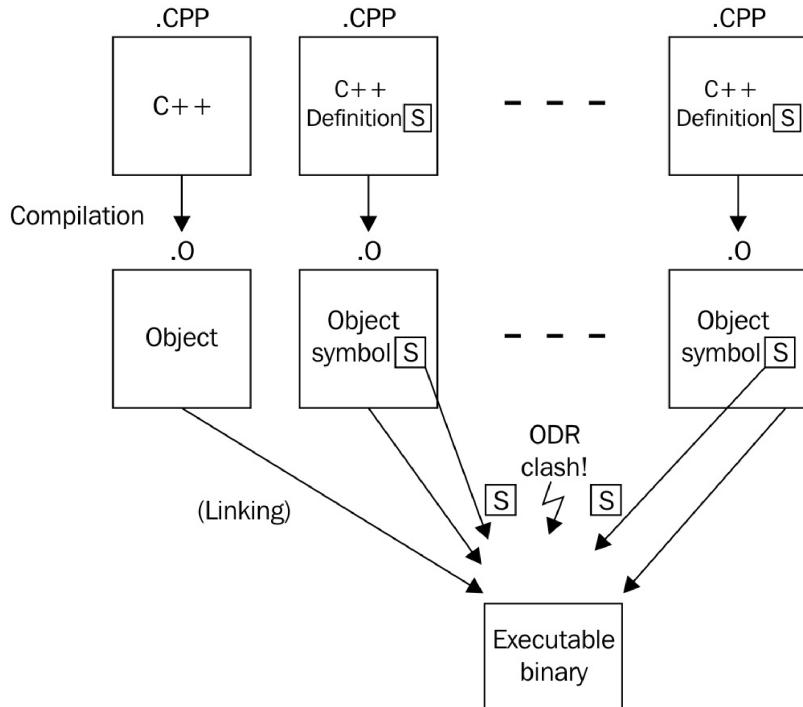
How it works...

C++程序通常都有多个C++源文件组成(其以 `.cpp` 或 `.cc` 结尾)。这些文件会单独编译成模块/二进制文件(通常以 `.o` 结尾)。链接所有模块/二进制文件形成一个单独的可执行文件，或是动态库/静态库则是编译的最后一步。

当链接器发现一个特定的符号，被定义了多次时就会报错。举个栗子，现在我们有一个函数声明 `int foo();`，当我们在两个模块中定义了同一个函数，那么哪一个才是正确的呢？链接器自己不能做主。这样没错，但是这也可能不是开发者想看到

的。

为了能提供全局可以使用的方法，通常会在头文件中定义函数，这可以让C++的所有模块都调用头文件中函数的实现(C++中，头文件中实现的函数，编译器会隐式的使用`inline`来进行修饰，从而避免符号重复定义的问题)。这样就可以将函数的定义单独的放入模块中。之后，就可以安全的将这些模块文件链接在一起了。这种方式也被称为[定义与单一定义规则\(ODR, One Definition Rule\)](#)。看了下图或许能更好的理解这个规则：



如果这是唯一的方法，就不需要只有头文件的库了。只有头文件的库非常方便，因为只需要使用`#include`语句将对应的头文件包含入C++源文件/头文件中后，就可以使用这个库了。当提供普通库时，开发者需要编写相应的编译脚本，以便连接器将库模块链接在一起，形成对应的可执行文件。这种方式对于很小的库来说是不必要的。

对于这样例子，`inline`关键字就能解决不同的模块中使用同一符号采用不同实现的方式。当连接器找到多个具有相同签名的符号时，这些函数定义使用`inline`进行声明，连接器就会选择首先找到的那个实现，然后认为其他符号使用的是相同的定义。所有使用`inline`定义的符号都是完全相同的，对于开发者来说这应该是常识。

我们的例子中，连接器将会在每个模块中找到`process_monitor::standard_string`符号，因为这些模块包含了`foo_lib.hpp`。如果没有了`inline`关键字，连接器将不知道选择哪个实现，所以其会将编译过程中断并报错。同样的原理也适用于`global_process_monitor`符号。

使用`inline`声明所有符号之后，连接器只会接受其找到的第一个符号，而将后续该符号的不同实现丢弃。

C++17之前，解决的方法是通过额外的C++模块文件提供相应的符号，这将迫使我们的库用户强制在链接阶段包含该文件。

传统的 `inline` 关键字还有另外一种功能。其会告诉编译器，可以通过实现直接放在调用它的地方来消除函数调用的过程。这样的话，代码中的函数调用会减少，这样我们会认为程序会运行的更快。如果函数非常短，那么生成的程序段也会很短(假设函数调用也需要若干个指令，保护现场等操作，其耗时会高于实际工作的代码)。当内联函数非常长，那么二进制文件的大小就会变得很大，有时并无法让代码运行的更快。因此，编译器会将 `inline` 关键字作为一个提示，可能会对内联函数消除函数调用。当然，编译器也会将一些函数进行内联，尽管开发者没有使用 `inline` 进行提示。

There's more...

C++17之前的解决方法就是将对应函数声明为静态函数，这个函数会返回某个静态对象的引用：

```
class foo{
public:
    static std::string& standard_string() {
        static std::string s{"some standard string"};
        return s;
    }
};
```

通过这种方式，将头文件包含在多个模块中是完全合法的，但仍然可以访问相同的实例。不过，对象并没有在程序开始时立即构造，而是在第一次调用这个获取函数时才进行构造。对于一些特定的情况来说，这也个问题。假设我们想要在程序开始时就构造静态和全局函数，从而完成一些比较重要的事情(就和我们的例程一样)，不过当程序运行后，在调用时去构造这些对象，就会带来比较大的性能开销。

另一个解决方法是将非模板类看做一个模板类，因此非模板类也适用于这项规则。

不过，以上的两种策略在C++17中不太适用了，C++17已经使用新的 `inline` 完美解决。

使用折叠表达式实现辅助函数

自C++11起，加入了变长模板参数包，能让函数结构任意数量的参数。有时，这些参数都组合成一个表达式，从中得出函数结果。C++17中使用折叠表达式，可以让这项任务变得更加简单。

How to do it...

首先，实现一个函数，用于将所有参数进行累加：

1. 声明该函数：

```
template <typename ... Ts>
auto sum(Ts ... ts);
```

2. 那么现在我们拥有一个参数包 `ts`，并且函数必须将参数包展开，然后使用表达式进行求和。如果我们对这些参数进行某个操作(比如：加法)，那么为了将这个操作应用于该参数包，就需要使用括号将表达式包围：

```
template<typename ... Ts>
auto sum(Ts ... ts) {
    return (ts + ...);
}
```

3. 现在我们可以调用这个函数：

```
int the_sum {sum(1, 2, 3, 4, 5)}; // value: 15
```

4. 这个操作不仅对 `int` 类型起作用，我们能对任何支持加号的类型使用这个函数，比如 `std::string`：

```
std::string a{"Hello "};
std::string b{"World"};

std::cout << sum(a, b) << '\n'; // output: Hello
World
```

How it works...

这里只是简单的对参数集进行简单的递归，然后应用二元操作符 `+` 将每个参数加在一起。这称为折叠操作。C++17中添加了折叠表达式，其能用更少的代码量，达到相同的结果。

其中有种称为一元折叠的表达式。C++17中的折叠参数包支持如下二元操作

符: + - * / % ^ & | = < > << >> += -= *= /= %= ^= &= |= <=> == != <=> && || , .* ->* 。

这样的话，在我们的例子中表达式 `(ts+...)` 和 `(...+ts)` 等价。不过，对于某些其他的例子，这就所有不同了——当 ... 在操作符右侧时，称为有“右折叠”；当 ... 在操作符左侧时，称为“左折叠”。

我们 `sum` 例子中，一元左折叠的扩展表达式为 `1+(2+(3+(4+5)))`，一元右折叠的扩展表达式为 `((1+2)+3)+4+5`。根据操作符的使用，我们就能看出差别。当用来进行整数相加，那么就没有区别。

There's more...

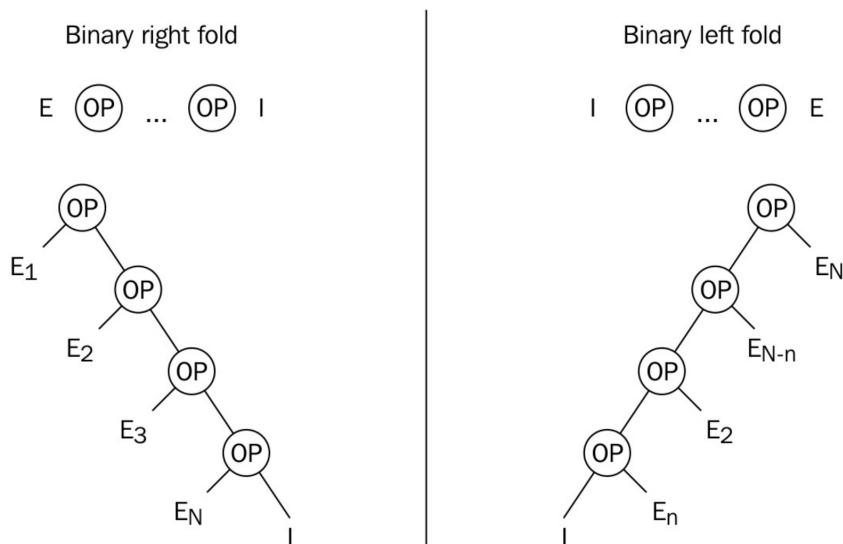
如果在调用 `sum` 函数的时候没有传入参数，那么可变参数包中就没有可以被折叠的参数。对于大多数操作来说，这将导致错误(对于一些例子来说，可能会是另外一种情况，我们后面就能看到)。这时我们就需要决定，这时一个错误，还是返回一个特定的值。如果是特定值，显而易见应该是0。

如何返回一个特定值：

```
template <typename ... Ts>
auto sume(Ts ... ts) {
    return (ts + ... + 0);
}
```

`sum()` 会返回0，`sum(1, 2, 3)` 返回 `(1+(2+(3+0)))`。这样具有初始值的折叠表达式称为二元折叠。

当我们写成 `(ts + ... + 0)` 或 `(0 + ... + ts)` 时，不同的写法就会让二元折叠表达式处于不同的位置(二元右折叠或二元左折叠)。下图可能更有助于理解左右二元折叠：



为了应对无参数传入的情况，我们使用二元折叠表达式，这里标识元素这个概念很重要——本例中，将0加到其他数字上不会有任何改变，那么0就一个标识元素。因为有这个属性，对于加减操作来说，可以将0添加入任何一个折叠表达式，当参数包中没有任何参数时，我们将返回0。从数学的角度来看，这没问题。但从工程的角度，我们需要根据我们需求，定义什么是正确的。

同样的原理也适用于乘法。这里，标识元素为1：

```
template <typename ... Ts>
auto product(Ts ... ts) {
    return (ts * ... * 1);
}
```

`product(2, 3)` 的结果是6，`product()` 的结果是1。

逻辑操作符 `and(&&)` 和 `or(||)` 具有内置的标识元素。`&&` 操作符为`true`，`||` 操作符为`false`。

对于逗号表达式来说，其标识元素为 `void()`。

为了更好的理解这特性，让我们可以使用这个特性来实现的辅助函数。

匹配范围内的单个元素

如何告诉函数在一定范围内，我们提供的可变参数至少包含一个值：

```
template <typename R, typename ... Ts>
auto matches(const R& range, Ts ... ts)
{
    return (std::count(std::begin(range),
        std::end(range), ts) + ...);
}
```

辅助函数中使用STL中的 `std::count` 函数。这个函数需要三个参数：前两个参数定义了迭代器所要遍历的范围，第三个参数则用于与范围内的元素进行比较。`std::count` 函数会返回范围内与第三个参数相同元素的个数。

在我们的折叠表达式中，我们也会将开始和结束迭代器作为确定范围的参数传入 `std::count` 函数。不过，对于第三个参数，我们将会每次从参数包中放入一个不同参数。最后，函数会将结果相加返回给调用者。

可以这样使用：

```
std::vector<int> v{1, 2, 3, 4, 5};

matches(v, 2, 5); // return 2
matches(v, 100, 200); // return 0
matches("abcdefg", 'x', 'y', 'z'); // return 0
matches("abcdefg", 'a', 'b', 'f'); // return 3
```

如我们所见，`matches` 辅助函数十分灵活——可以直接传入 `vector` 或 `string` 直接调用。其对于初始化列表也同样适用，也适用于 `std::list`，`std::array`，`std::set` 等 STL 容器的实例。

检查集合中的多个插入操作是否成功

我们完成了一个辅助函数，用于将任意数量参数插入 `std::set` 实例中，并且返回是否所有插入操作都成功完成：

```
template <typename T, typename ... Ts>
bool insert_all(T &set, Ts ... ts)
{
    return (set.insert(ts).second && ...);
}
```

那么这个函数如何工作呢？`std::set` 的 `insert` 成员函数声明如下：

```
std::pair<iterator, bool> insert(const value_type&
value);
```

手册上所述，当我们使用 `insert` 函数插入一个元素时，该函数会使用一个包含一个迭代器和一个布尔值的组对作为返回值。当该操作成功，那么迭代器指向的就是新元素在 `set` 实例中的位置。否则，迭代器指向某个已经存在的元素，这个元素与插入项有冲突。

我们的辅助函数在完成插入后，会访问 `.second` 区域，这里的布尔值反映了插入操作成功与否。如果所有插入操作都为 `true`，那么都是成功的。折叠标识使用逻辑操作符 `&&` 链接所有插入结果的状态，并且返回计算之后的结果。

可以这样使用它：

```
std::set<int> my_set{1, 2, 3};

insert_all(my_set, 4, 5, 6); // Returns true
insert_all(my_set, 7, 8, 2); // Returns false, because
the 2 collides
```

需要注意的是，当在插入3个元素时，第2个元素没有插入成功，那么 `&&` 会根据短路特性，终止插入剩余元素：

```
std::set<int> my_set{1, 2, 3};

insert_all(my_set, 4, 2, 5); // Returns flase
// set contains {1, 2, 3, 4} now, without the 5!
```

检查所有参数是否在范围内

当要检查多个变量是否在某个范围内时，可以多次使用查找单个变量是否在某个范围的方式。这里我们可以使用折叠表达式进行表示：

```

template <typename T, typename ... Ts>
bool within(T min, T max, Ts ...ts)
{
    return ((min <= ts && ts <= max) && ...);
}

```

表达式 `(min <= ts && ts <= max)` 将会告诉调用者参数包中的每一个元素是否在这个范围内。我们使用 `&&` 操作符对每次的结果进行处理，从而返回最终的结果。

如何使用这个辅助函数：

```

within(10, 20, 1, 15, 30); // --> false
within(10, 20, 11, 12, 13); // --> true
within(5.0, 5.5, 5.1, 5.2, 5.3) // --> true

```

这个函数也是很灵活的，其只需要传入的参数类型可以进行比较，且支持 `<=` 操作符即可。并且该规则对于 `std::string` 都是适用的：

```

std::string aaa {"aaa"};
std::string bcd {"bcd"};
std::string def {"def"};
std::string zzz {"zzz"};

within(aaa, zzz, bcd, def); // --> true
within(aaa, def, bcd, zzz); // --> false

```

将多个元素推入vector中

可以编写一个辅助函数，不会减少任何结果，又能同时处理同一类的多个操作。比如向 `std::vector` 传入元素：

```

template <typename T, typename ... Ts>
void insert_all(std::vector<T> &vec, Ts ... ts) {
    (vec.push_back(ts), ...);
}

int main() {
    std::vector<int> v{1, 2, 3};
    insert_all(v, 4, 5, 6);
}

```

需要注意的是，使用了逗号操作符将参数包展开，然后推入`vector`中。该函数也不惧空参数包，因为逗号表达式具有隐式标识元素，`void()` 可以翻译为什么都没做。

第2章 STL容器

C++标准库中有大量的标准容器。容器通常包含一组数据或对象的集合。容器的厉害之处在于几乎可以和任何类型的对象一起使用，所以我们只需要为程序选择合适的容器即可。STL带给我们栈、自动增长的vector、map等等。这样我们就可以集中精力于我们的应用，而不用重复制作轮子。了解所有容器，对于C++开发者来说至关重要。

STL容器的分类如下，会在各节中进行详细描述：

- 连续存储
- 列表存储
- 搜索树
- 哈希表
- 容器适配器

连续存储

想要存储一组对象最简单的方式，就是将其一个接一个的存在一块比较大的内存当中。内存可以使用随机访问的方式进行，其时间复杂度为O(1)。

最简单的方式就是使用 `std::array`，其就是对C风格数组的一种包装。不过，`std::array` 要比C风格数组要先进的多，因为其没有运行时开销，而且进行元素添加时，也会十分舒适和安全。还有一点和C风格数组一样，一旦创建，其长度就是固定的，创建过后无法改变长度。

`std::vector` 和 `std::array` 很类似，不过 `std::vector` 的长度可变。其会使用堆上的内存来存储对象。当新元素添加到 `vector` 中后，当前长度超过了原始的长度，那么 `std::vector` 会自动新分配一段更大的内存，用来放置包括新插入元素的所有元素，并且释放之前所占用的内存。此外，当新元素需要插入到两个旧元素之间时，`std::vector` 会移动当前已有的元素。当要删除 `vector` 中间的一个已存在元素，那么 `vector` 类会自动地移动其他对象，将删除后的缝隙填补起来。

如果有大量元素在 `std::vector` 的头部或尾部进行插入或删除，那么为了填补空隙和移动已有元素，将会耗费很多时间。如遇到这样的情况，建议你考虑使用 `std::deque`。对象集合会存储在多段固定长度的连续内存中，这些内存段是相互独立的。这就使得双向队列变得很简单，并且增长也很容易，因为不同的内存段相对独立，只需要将新分配的内存段加入就可以了，无需对其他已存在的内存段进行移动。减少的场景也是一样的。

列表存储

`std::list` 是一个典型的双向链表。如果是单向列表，那就需要进行遍历，所以 `std::forward_list` 的优势在维护的复杂性上，因为其指针方向只有一个方向。列表遍历的时间复杂度是线性的O(n)。其在特定位置上插入和删除元素的时间复杂度为O(1)。

搜索树

当对象集具有可进行排序的自然属性时，可以使用小于的关系将这些元素进行排序，我们就可以使用搜索树来保存这个排序关系。从名字就可以看出，搜索树可以帮助我们很容易的通过一个关键字查找到对应元素，其搜索的时间复杂度为 $O(\log(n))$ 。

STL提供了不同种类的树，`std::set` 是其中最简单的一种，保存元素不重复，存储的元素是可排序的(用一种树的结构)。

`std::map` 使用的是另一种方式，将存储的数据使用组对进行存储。一个组对有一个 **key** 值和一个对应值构成。搜索树会对 **key** 值部分进行排序，使组对能作为 `std::map` 的一种关联容器。`std::map` 的 **key** 值和 `std::set` 的值一样，在整个树中只能存在一个。

`std::multiset` 和 `std::multimap` 是被特化的，**key** 对象可以是重复的。

哈希表

讨论关联容器时，搜索树并不是唯一的方式。使用哈希表查找元素的时间复杂度只有 $O(1)$ ，不过这就会忽略其自然序，所以不能简单的使用排序的方式进行遍历。哈希表大小可由用户控制，并且可以单独选择哈希函数，这是一项很重要的特性，因为其性能与空间复杂度依赖于此。

`std::unordered_set` 和 `std::unordered_map` 具有很多接口与 `std::set` 和 `std::map` 一样，它们之间几乎可以相互替换。

搜索树的实现中，容器都具有多个变种：

`std::unordered_multiset` 和 `std::unordered_multimap`，这两种方法都取消了对象/键的唯一性，因此我们可以用相同的键存储多个元素。

容器适配器

数组、列表、树和哈希表并不是存储和访问数据的唯一方式，这里还有栈、队列等其他的方式也可以存储和访问数据。类似的，越复杂的结构可以使用越原始的方式实现，并且 STL 使用以下形式的容器适配器进行操作：`std::stack`、`std::queue` 和 `std::priority_queue`。

最牛X的是当我们需要这样的数据结构时，我们可以选择一种适配器。然后，当我们觉得到它们性能较差时，就可以改变一个模板参数，以便让适配器使用不同的容器实现。实践中，这也就意味着我们可以将 `std::stack` 实例中的元素类型从 `std::vector` 切换成 `std::deque`。

擦除/移除std::vector元素

由于 `std::vector` 能自动增长，并且使用方式简单，很受C++开发新手的喜爱。可以通过查阅手册，来了解这个容器该如何使用，比如删除元素。这样使用STL容器，只是了解容器的皮毛，容器应该帮助我们写出更简洁、维护性好和更快的代码。

本节的全部内容都是在一个 `vector` 实例中删除元素。当 `vector` 中部的一个元素消失了，那么位于消失元素右边的所有元素都要往左移(这种操作的时间复杂度为 $O(n)$)。新手们会用循环来做这件事，因为循环的确好用。不过，循环会降低代码的优化空间。最后，比起STL的方法，循环是既不快，也不美，

How to do it...

首先，我们使用整数来填充一个 `std::vector` 实例，之后剔除一些特定元素。我们演示的从 `vector` 实例中删除元素正确的方法。

1. 包含文件是首要任务。

```
#include <iostream>
#include <vector>
#include <algorithm>
```

2. 声明我们所要使用的命名空间。

```
using namespace std;
```

3. 现在我们来创建一个 `vector` 实例，并用整数填满它。

```
int main() {
    vector<int> v{1, 2, 3, 2, 5, 2, 6, 2, 4, 8};
```

4. 然后移除一些元素。需要我们移除哪些呢？2出现的太多了，就选择2吧。让我们移除它们吧。

```
const auto new_end(remove(begin(v), end(v), 2));
```

5. 已经完成了两步中的一步。`vector` 在删除这些元素之后，长度并没有发生变化。那么下一步就让这个 `vector` 变得短一些。

```
v.erase(new_end, end(v));
```

6. 我们在这里暂停一下，输出一下当前 `vector` 实例中所包含的元素。

```
for(auto i : v) {
    cout << i << ", ";
}
cout << '\n';
```

7. 现在，让我们来移除一组指定的数据。为了完成这项工作，我们先定义了一个谓词函数，其可接受一个数作为参数，当这个数是奇数时，返回true。

```
const auto odd([](int i){return i % 2 != 0;});
```

8. 这里我们使用remove_if函数，使用上面定义的谓词函数，来删除特定的元素。这里我们将上面删除元素的步骤合二为一。

```
v.erase(remove_if(begin(v), end(v), odd),
end(v));
```

9. 所有的奇数都被删除，不过vector实例的容量依旧是10。最后一步中，我们将其容量修改为正确的大小。需要注意的是，这个操作会让vector重新分配一段内存，以匹配相应元素长度，vector中已存的元素会移动到新的内存块中。

```
v.shrink_to_fit();
```

10. 打印一下现在vector实例中的元素。

```
for (auto i : v) {
    cout << i << ", ";
}
cout << '\n';
```

11. 编译完成后，运行程序，就可以了看到两次删除元素后vector实例中存在的元素。

```
$ ./main
1, 3, 5, 6, 4, 8,
6, 4, 8,
```

How it works...

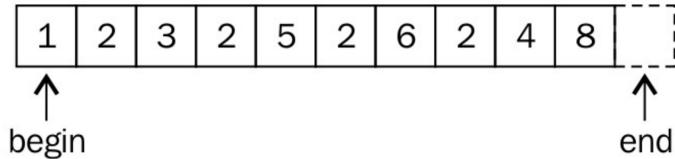
我们可以清楚的看到，要从一个vector实例中移除一个元素，首先要进行删除，然后进行擦除，这样才算真正的移除。这会让人感到困惑，那就让我们近距离观察一下这些步骤是如何工作的。

从vector中移除2的代码如下所示：

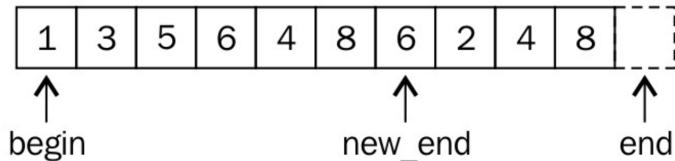
```
const auto new_end = remove(begin(v), end(v), 2));
v.erase(new_end, end(v));
```

`std::begin` 和 `std::end` 函数都以一个 `vector` 实例作为参数，并且返回其迭代器，迭代器分别指向第一个元素和最后一个元素，就如下图所示。

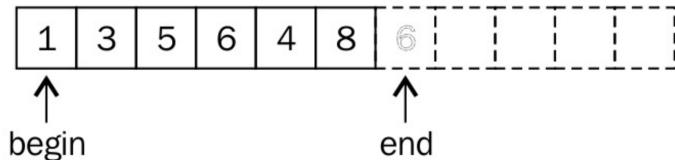
1.) initial state



2.) after `new_end = std::remove (begin, end, 2);`



3.) after `vector.erase (new_end, end);`



`std::remove` 在删除2的时候，会先将非2元素进行移动，然后修改`end`迭代器的指向。该算法将严格保留所有非2个值的顺序。

在2步中，2的值仍然存在，并且 `vector` 应该变短。并且4和8在现有的 `vector` 中重复了。这是怎么回事？

让我们再来看一下所有的元素，目前 `vector` 的范围并不是原来那样了，其是从 `begin` 迭代器，到 `new_end` 迭代器。`new_end` 之后的值其实就不属于 `vector` 实例了。我们会注意到，在这个范围内的数值，就是我们想要的正确结果，也就是所有的2都被移除了。

最后，也就是为什么要调用 `erase` 函数：我们需要告诉 `vector` 实例，`new_end` 到 `end` 之间的元素我们不需要了。我们仅需要保留 `begin` 到 `new_end` 间的元素就好了。`erase` 函数会将 `end` 指向 `new_end`。这里需要注意的是 `std::remove` 会直接返回 `new_end` 迭代器，所以我们可以直接使用它。

Note:

`vector` 在这里不仅仅移动了内部指针。如果 `vector` 中元素比较复杂，那么在移除的时候，会使用其析构函数来销毁相应的对象。

最后，这个向量就如步骤3所示：的确变短了。那些旧的元素已经不在 `vector` 的访问范围内了，不过其仍存储在内存中。

为了不让 `vector` 浪费太多的内存，我们在最后调用了 `shrink_to_fit`。该函数会为元素分配足够的空间，将剩余的元素移到该空间内，并且删除之前那个比较大的内存空间。

在上面的第8步中，我们定义了一个谓词函数，并在 `std::remove_if` 中使用了它。因为不论删除函数返回怎么样的迭代器，在对 `vector` 实例使用擦除函数都是安全的。如果 `vector` 中全是偶数，那么 `std::remove_if` 不会做任何事情，并且返回 `end` 迭代器。之后的调用就为 `v.erase(end, end);`，同样没有做任何事情。

There's more...

`std::remove` 函数对其他容器同样有效。当使用 `std::array` 时，其不支持 `erase` 操作，因为其内存空间固定，无法进行自动化处理。因为 `std::remove` 只是将要删除的元素移动到容器末尾，而不是将其真正删除，所以这个函数也可以用于不支持空间大小变化的数据类型。当然也有其他类似的方法，例如字符串中，可以用哨兵值 `\0` 来覆盖原始的 `end` 迭代所指向的值。

以 $O(1)$ 的时间复杂度删除未排序 `std::vector`中的元素

因为其他元素要填补删除元素所留下来的空隙，从而需要进行移动，所以从 `std::vector` 中删除元素的时间复杂度为 $O(n)$ 。

移动其他元素也与此类似，当很多很大或很复杂的元素需要移动，那么就会花费很长的时间。当无法保证顺序时，我们需要对其进行优化，这就是本节的内容。

How to do it...

我们继续使用一些数字来填充 `std::vector` 实例，并且实现一个快速删除函数，以 $O(1)$ 的时间复杂度删除`vector`中的元素。

1. 首先，包含必要的头文件：

```
#include <iostream>
#include <vector>
#include <algorithm>
```

2. 定义主函数，并定义一个 `vector` 实例：

```
int main() {
    std::vector<int> v{123, 456, 789, 100, 200};
```

3. 下一步就要删除索引为2的元素(789)。我们所要用的来删除元素的函数在后面进行实现，我们先假设已经实现好了。执行完成后，来看下 `vector` 中的内容。

```
quick_remove_at(v, 2);
for (int i : v) {
    std::cout << i << ", ";
}
std::cout << '\n';
```

4. 现在，我们将删除另外一个元素。我们想删除123，但是要假装不知道其索引。因此，我们要使用 `std::find` 函数在`vector`的合法范围内查找这个值，并返回其位置信息。得到索引信息后，我们就可以用 `quick_remove_at` 将对应元素删除了，这里所使用到的是一个重载版本，能接受迭代器作为输入参数。

```

    quick_remove_at(v, std::find(std::begin(v),
std::end(v), 123));
    for (int i : v) {
        std::cout << i << ", ";
    }
    std::cout << '\n';
}

```

5. 我们实现了两种 `quick_remove_at` 函数。具体实现代码中，需要注意与 `main` 函数的前后关系。两个函数都能接收一个 `vector` 实例的引用，所以这里允许用户使用各种类型的变量作为元素。对于我们来说，其类型就是 `T`。第一个 `quick_remove_at` 函数用来接收索引值，是一个具体的数，所以其接口如下所示：

```

template <typename T>
void quick_remove_at(std::vector<T> &v, std::size_t
idx)
{

```

6. 现在来展示一下本节的重点——如何在不移动其他元素的情况下，快速删除某个元素？首先，将 `vector` 中最后一个元素进行重写。第二，删除 `vector` 中最后一个元素。就这两步。我们的代码会对输入进行检查。如果输入的索引值超出了范围，函数不会做任何事情。另外，该函数会在传入空 `vector` 的时候崩溃。

```

    if (idx < v.size()) {
        v[idx] = std::move(v.back());
        v.pop_back();
    }
}

```

7. 另一个 `quick_remove_at` 实现也很类似。用 `std::vector<T>` 的迭代器替换了具体的索引数值。因为泛型容器已经定义了这样的类型，所以获取它的类型并不复杂。

```

template <typename T>
void quick_remove_at(std::vector<T> &v,
                     typename std::vector<T>::iterator
it)
{

```

8. 现在我们来访问这些迭代器所指向的值。和另一个函数一样，我们会将最后一个元素进行重写。因为这次处理的是迭代器，所以我们要对迭代器指向的位置进行检查。如果其指向了一个错误的位置，我们就会阻止其解引用。

```

    if (it != std::end(v)) {

```

9. 在该代码块中，我们会做和之前一样的事情——我们要覆盖最后一个位置上的值——然后将最后一个元素从 `vector` 中剪掉。

```
*it = std::move(v.back());  
v.pop_back();  
}  
}
```

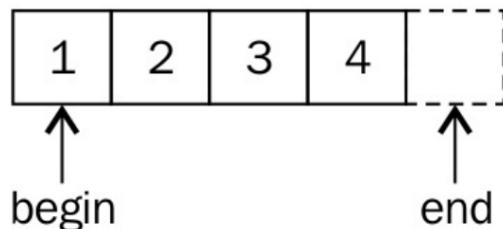
10. 这就完事了。让我们来编译程序，并运行：

```
$ ./main  
123, 456, 200, 100,  
100, 456, 200,
```

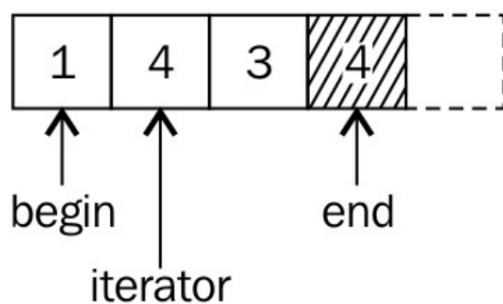
How it works...

`quick_remove_at` 函数移除元素非常快，而且不需要动其他元素。这个函数使用了更加具有创造性的做法：这是一种与实际元素交换的方式，然后将最后一个元素从 `vector` 中删除。虽然，最后一个元素与选中的元素没有实际的关联，但是它在这个特别的位置上，而且删除最后一个元素的成本最低！`vector` 的长度在删除完成后，也就减少1，这就是这个函数所要做的。并且无需移动任何元素。看一下下面的图，可能有助于你理解这个函数的原理。

1.) initial state



2.) after quick_remove_at (vec, iterator)



完成这两步的代码如下：

```
v.at(idx) = std::move(v.back());
v.pop_back();
```

迭代器版本实现几乎一模一样：

```
*it = std::move(v.back());
v.pop_back();
```

逻辑上，我们将选定元素与最后一个元素进行交换。不过，在代码中元素并没有进行交换，代码直接使用最后一个值覆盖了选定元素的值。为什么要这样？当我们交换元素时，就需要将选定的元素存储在一个临时变量中，并在最后将这个临时变量中的值放在 `vector` 的最后。这个临时变量是多余的，而且要删除的值对于我们来说是没有意义的，所以这里选择了直接覆盖的方式，更加高效的实现了删除。

好了，交换是无意义的，覆盖是一种更好的方式。让我们来看下这个，当我们要获取 `vector` 最后元素的迭代器时，只需要简单的执行 `*it = v.back();` 就行了，对吧？完全正确，不过试想我们存储了一些非常长的字符串在 `vector` 中，或存储了另一个 `vector` 或 `map` ——这种情况下，简单的赋值将对这些值进行拷贝，那么就会带来非常大的开销。这里使用 `std::move` 可将这部分开销优化掉：比如字符串，指向堆内存上存储的一个大字符串。我们无需拷贝它。只需要移动这个字符串即可，就是将目标指针指向这块地址即可。移动源保持不变，不过出于无用的状态，这样做可以类似的让目标指针指向源指针所在的位置，然后将原始位置的元素删除，这样做即完成了元素移动，又免去了移动消耗。

快速或安全的访问**std::vector**实例的方法

`std::vector` 可能是STL容器中适用范围最广的，因为其存储数据的方式和数组一样，并且还有相对完善的配套设施。不过，非法访问一个 `vector` 实例还是十分危险的。如果一个 `vector` 实例具有100个元素，那当我们想要访问索引为123的元素时，程序就会崩溃掉。如果不崩溃，那么你就麻烦了，未定义的行为会导致一系列奇奇怪怪的错误，查都不好查。经验丰富的开发者会在访问前，对索引进行检查。这样的检查其实比较多余，因为很多人不知道 `std::vector` 有内置的检查机制。

How to do it...

本节我们将使用两种不同的方式访问一个 `std::vector` 实例，并且利用其特性编写更加安全的代码。

1. 先包含相应的头文件，并且用1000个123填满一个`vector`实例：

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    const size_t container_size{1000};
    vector<int> v(container_size, 123);
```

2. 我们通过 `[]` 操作符访问范围之外的元素：

```
cout << "Out of range element value: "
     << v[container_size + 10] << '\n';
```

3. 之后我们使用 `at` 函数访问范围之外的元素：

```
cout << "Out of range element value: "
     << v.at(container_size + 10) << '\n';
}
```

4. 让我们运行程序，看下会发生什么。下面的错误信息是由GCC给出。其他编译器也会通过不同方式给出类似的错误提示。第一种方式得到的结果比较奇怪。超出范围的访问方式并没有让程序崩溃，但是访问到了与123相差很大的数字。第二种方式中，我们看不到打印出来的结果，因为在打印之前程序已经崩溃了。当越界访问发生的时候，我们可以通过异常的方式更早的得知！

```
Out of range element value: -726629391
terminate called after throwing an instance of
'std::out_of_range'
what(): array::at: __n (which is 1010) >= _Nm (which
is 1000)
Aborted (core dumped)
```

How it works...

`std::vector` 提供了 `[]` 操作符和 `at` 函数，它们的作用几乎是一样的。`at` 函数会检查给定的索引值是否越界，如果越界则返回一个异常。这对于很多情景都十分适用，不过因为检查越界要花费一些时间，所以 `at` 函数会让程序慢一些。

当需要非常快的索引成员时，并能保证索引不越界，我们会使用 `[]` 快速访问 `vector` 实例。很多情况下，`at` 函数在牺牲一点性能的基础上，有助于发现程序内在的bug。

Note:

默认使用 `at` 函数是一个好习惯。当代码的性能很差，但没有bug存在时，可以使用性能更高的操作符来替代 `at` 函数。

There's more...

当然，我们需要处理越界访问，避免整个程序崩溃。为了对越界访问进行处理，我们可以使用截获异常的方式。可以用 `try` 代码块将调用 `at` 函数的部分包围，并且定义错误处理的 `catch` 代码段。

```
try {
    std::cout << "Out of range element value: "
    << v.at(container_size + 10) << '\n';
} catch (const std::out_of_range &e) {
    std::cout << "Ooops, out of range access detected: "
    << e.what() << '\n';
}
```

Note:

顺带一提，`std::array` 也提供了 `at` 函数。

保持对**std::vector**实例的排序

`array` 和 `vector` 不会对他们所承载的对象进行排序。有时我们去需要排序，但这不代表着我们总是要去切换数据结构，需要排序能够自动完成。在我们的例子有如有一个 `std::vector` 实例，将添加元素后的实例依旧保持排序，会是一项十分有用的功能。

How to do it...

本节中我们使用随机单词对 `std::vector` 进行填充，然后对它进行排序。并在插入更多的单词的同时，保证 `vector` 实例中单词的整体排序。

1. 先包含必要的头文件。

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <iterator>
#include <cassert>
```

2. 声明所要使用的命名空间。

```
using namespace std;
```

3. 完成主函数，使用一些随机单词填充 `vector` 实例。

```
int main()
{
    vector<string> v {"some", "random", "words",
                      "without", "order", "aaa",
                      "yyy"};
```

4. 对`vector`实例进行排序。我们使用一些断言语句和STL中自带的 `is_sorted` 函数对是否排序进行检查。

```
assert(false == is_sorted(begin(v), end(v)));
sort(begin(v), end(v));
assert(true == is_sorted(begin(v), end(v)));
```

5. 这里我们使用 `insert_sorted` 函数添加随机单词到已排序的 `vector` 中，这个函数我们会在后面实现。这些新插入的单词应该在正确的位置上，并且 `vector` 实例需要保持已排序的状态。

```
insert_sorted(v, "foobar");
insert_sorted(v, "zzz");
```

6. 现在，我们来实现 `insert_sorted` 函数。

```
void insert_sorted(vector<string> &v, const string
&word)
{
    const auto insert_pos = lower_bound(begin(v),
end(v), word);
    v.insert(insert_pos, word);
}
```

7. 回到主函数中，我们将 `vector` 实例中的元素进行打印。

```
for (const auto &w : v) {
    cout << w << " ";
}
cout << '\n';
}
```

8. 编译并运行后，我们得到如下已排序的输出。

```
aaa foobar order random some without words yyy zzz
```

How it works...

程序整个过程都是围绕 `insert_sorted` 展开，这也就是本节所要说明的：对于任意的新字符串，通过计算其所在位置，然后进行插入，从而保证 `vector` 整体的排序性。不过，这里我们假设的情况是，在插入之前，`vector` 已经排序。否则，这种方法无法工作。

这里我们使用STL中的 `lower_bound` 对新单词进行定位，其可接收三个参数。头两个参数是容器开始和结尾的迭代器。这确定了我们单词 `vector` 的范围。第三个参数是一个单词，也就是要被插入的那个。函数将会找到大于或等于第三个参数的首个位置，然后返回指向这个位置的迭代器。

获取了正确的位置，那就使用 `vector` 的成员函数 `insert` 将对应的单词插入到正确的位置上。

There's more...

`insert_sorted` 函数很通用。如果需要其适应不同类型的参数，这样改函数就能处理其他容器所承载的类型，甚至是容器的类似，比如 `std::set`、`std::deque`、`std::list` 等等。(这里需要注意的是成员函

数 `lower_bound` 与 `std::lower_bound` 等价，不过成员函数的方式会更加高效，因为其只用于对应的数据集合）

```
template <typename C, typename T>
void insert_sorted(C &v, const T &item)
{
    const auto insert_pos = lower_bound(begin(v), end(v),
item));
    v.insert(insert_pos, item);
}
```

当我们要将 `std::vector` 类型转换为其他类型时，需要注意的是并不是所有容器都支持 `std::sort`。该函数所对应的算法需要容器为可随机访问容器，例如 `std::list` 就无法进行排序。

向**std::map**实例中高效并有条件的插入元素

我们需要用键值对填充一个 `map` 实例时，会碰到两种不同的情况：

1. 键不存在。创建一个全新的键值对。
2. 键已存在。修改键所对应的值。

我通常会使用 `insert` 或 `emplace` 函数对 `map` 插入新元素，如果插入不成功，那么就是第二种情况，就需要去修改现有的元素。`insert` 和 `emplace` 都会创建一个新元素尝试插入到 `map` 实例中，不过在第二种情况下，这个新生成的元素会被扔掉。两种情况下，我们都会多余调用一次构造函数。

C++17 中，添加了 `try_emplace` 函数，其只有在满足条件的情况下，才能插入新元素。让我们实现一个程序，建立一张表，列出各国亿万富翁的数量。我们例子中不会使用很大开销进行元素创建，不过我们的例子来源于生活，其能让你明白如何使用 `try_emplace`。

How to do it...

本节中，我们将实现一个应用，其能创建一张百万富翁的列表。这个列表中按国家区分，里面记录了各国富人的数量。

1. 包含头文件和声明命名空间。

```
#include <iostream>
#include <functional>
#include <list>
#include <map>

using namespace std;
```

2. 定义一个结构器，代表对应的富翁。

```
struct billionaire {
    string name;
    double dollars;
    string country;
};
```

3. 主函数中，我们定义了一个百万富翁的列表。世界上有很多百万富翁，所以我们创建一个有限列表来存储这些富翁的信息。这个列表是已排序的。2017年福布斯富豪名单，世界百万富翁排行榜可以在 <https://www.forbes.com/billionaires/list> 查到。

```

int main()
{
    list<billionaire> billionaires {
        {"Bill Gates", 86.0, "USA"},
        {"Warren Buffet", 75.6, "USA"},
        {"Jeff Bezos", 72.8, "USA"},
        {"Amancio Ortega", 71.3, "Spain"},
        {"Mark Zuckerberg", 56.0, "USA"},
        {"Carlos Slim", 54.5, "Mexico"},
        // ...
        {"Bernard Arnault", 41.5, "France"},
        // ...
        {"Liliane Bettencourt", 39.5, "France"},
        // ...
        {"Wang Jianlin", 31.3, "China"},
        {"Li Ka-shing", 31.2, "Hong Kong"}
        // ...
    };
}

```

4. 现在让我们定义一个表。这个表由表示国家名的字符串和一个组对构成。组对中会具有上面列表的一个(**const**)副本。这也就是每个国家最富有的人。组对中另一个变量是一个计数器，其会统计某国的富豪人数。

```
map<string, pair<const billionaire, size_t>> m;
```

5. 现在，让我们将列表中的数据尝试插入到组对中。每个组对中都包含了对应国家的百万富翁，并将计数器的值置成1。

```

for (const auto &b : billionaires) {
    auto [iterator, success] =
    m.try_emplace(b.country, b, 1);
}

```

6. 如果这一步成功，那就不用再做其他事了。我们使用**b**和1创建的组对已经插入到表中。如果因为键已存在而插入失败，那么组对就不会构建。当我们百万富翁结构体非常大时，我们需要将运行时拷贝的时间节省下来。不过，在不成功的情况下，我们还是要对计数器进行增加1的操作。

```

if (!success) {
    iterator->second.second += 1;
}

```

7. 现在，我们来打印一下每个国家百万富翁的数量，以及各个国家中最富有的人。

```

        for (const auto & [key, value] : m) {
            const auto &[b, count] = value;
            cout << b.country << " : " << count
                << " billionaires. Richest is "
                << b.name << " with " << b.dollars
                << " B$\n";
        }
    }
}

```

8. 编译并运行程序，就会得到下面的输出(这里的输出是不完整的，因为列表比较长)。

```

$ ./efficient_insert_or_modify
China : 1 billionaires. Richest is Wang Jianlin with
31.3 B$
France : 2 billionaires. Richest is Bernard Arnault
with 41.5 B$
Hong Kong : 1 billionaires. Richest is Li Ka-shing
with 31.2 B$
Mexico : 1 billionaires. Richest is Carlos Slim with
54.5 B$
Spain : 1 billionaires. Richest is Amancio Ortega
with 71.3 B$
USA : 4 billionaires. Richest is Bill Gates with 86
B$

```

How it works...

本节围绕着 `std::map` 中的 `try_emplace` 函数展开，这个函数是C++17添加的。下面是其函数声明之一：

```

std::pair<iterator, bool> try_emplace(const key_type& k,
Args&... args);

```

其函数第一个参数 `k` 是插入的键，`args` 表示这个键对应的值。如果我们成功的插入了元素，那么函数就会返回一个迭代器，其指向新节点在表中的位置，组对中布尔变量的值被置为`true`。当插入不成功，组对中的布尔变量值会置为`false`，并且迭代器指向与新元素冲突的位置。

这个特性在我们的例子中非常有用——可以完美处理第一次访问到，和之后访问到的情况。

Note:

`std::map` 中 `insert` 和 `emplace` 方法完全相同。`try_emplace` 与它们不同的地方在于，在遇到已经存在的键时，不会去构造组对。当相应用对象的类型需要很大开销进行构造时，这对于程序性能是帮助的。

There's more...

如果我们将表的类型从 `std::map` 换成 `std::unordered_map`，程序照样能工作。这样的话，当不同类型的表具有较好的性能特性时，我们就可以快速的进行切换。例子中，唯一可观察到的区别是，亿万富翁表不再按字母顺序打印，因为哈希表和搜索树不同，其不会对对象进行排序。

了解`std::map::insert`新的插入提示语义

`std::map` 中查找元素的时间复杂度为 $O(\log(n))$ ，与插入元素的时间复杂相同，因为要在对应位置上插入元素，那么就先要找到这个位置。通常，插入 M 个新元素的时间复杂度为 $O(M \log(n))$ 。

为了让插入更加高效，`std::map` 插入函数接受一个迭代器参数 `hint`。自 C++11 起，该参数为指向将插入新元素到其前的位置的迭代器。如果这个迭代器给定正确，那么插入的时间复杂度就为 $O(1)$ 。

How to do it...

本节会是用传入迭代器的方式向 `std::map` 实例中插入多个新元素，从而减少耗时：

1. 包含必要的头文件。

```
#include <iostream>
#include <map>
#include <string>
```

2. 创建一个 `map` 实例，并填充一些内容。

```
int main()
{
    std::map<std::string, size_t> m { {"b", 1}, {"c", 2}, {"d", 3} };
```

3. 我们将插入多个元素，对于每次插入，我们都会传入一个 `hint` 迭代器。第一次插入我们不指定其开始位置，只将插入位置指向 `map` 的 `end` 迭代器之前。

```
auto insert_it = std::end(m);
```

4. 我们将以字母表的反序进行元素的插入，然后使用 `hint` 迭代器，然后使用 `insert` 函数的返回值重新初始化迭代器的值。下一个元素将在 `hint` 迭代器前插入。

```
for (const auto &s : {"z", "y", "x", "w"}) {
    insert_it = m.insert(insert_it, {s, 1});
}
```

5. 为了展示在什么情况下 `insert` 函数不工作，我们将要插入最左侧位置的元素插入到最右侧。

```
m.insert(std::end(m), {"a", 1});
```

6. 最后我们打印当前的 `map`。

```
for (const auto & [key, value] : m) {
    std::cout << "\\" << key << ":" " << value
    << ", ";
}
std::cout << '\n';
}
```

7. 编译运行程序，错误的插入并没有对结果又什么影响，`map` 实例中对象的顺序仍然是对的。

```
"a": 1, "b": 1, "c": 2, "d": 3, "w": 1, "x": 1, "y":
1, "z": 1,
```

How it works...

本例与常用的方式不同，多了一个迭代器。并且我们提到了这个迭代器的正确与否。

正确的迭代器将会指向一个已存在的元素，其值要比要插入元素的键大，所以新元素会插在这个迭代器之前。如果用户提供的迭代器位置有误，那么插入函数会退化成未优化的版本，其时间复杂度恢复 $O(\log(n))$ 。

对于第一次插入，我们选择了 `map` 实例的 `end` 迭代器，因为没有其他更好的选择。在插入“z”之后，函数会返回相应的迭代器，这样我们就知道了要插入“y”的位置。“x”也同理，后面的元素依次类推。

Note:

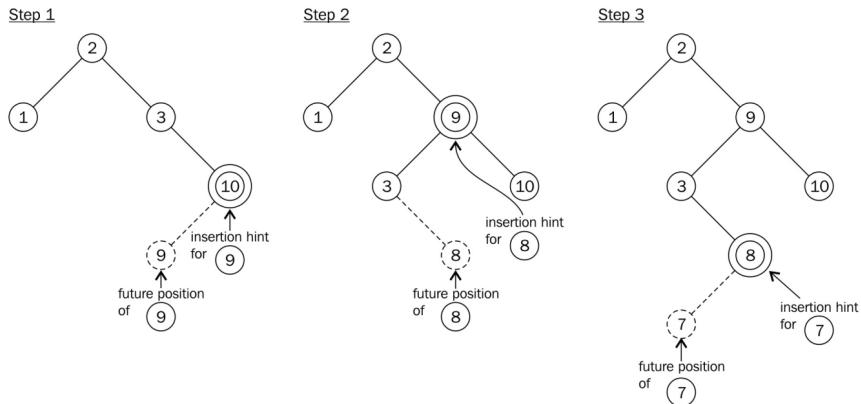
在C++11之前，`hint`迭代器只是建议作为搜索开始位置的迭代器。

There's more...

其中，比较有趣的事情是，在给定错误的迭代器，`map` 实例依旧能保持其排序。那么他是如何工作的呢？还有插入的时间复杂度为 $O(1)$ 意味着什么？

`std::map`通常使用二叉搜索树实现。当在搜索树中插入一个新键时，该键要和其他键进行比较，从末端到顶端。如果键小于或大于其他节点的键，搜索树的左侧或右侧分支则会成为更深的节点。不过，搜索算法会阻止节点达到当前搜索树的底端。否则会打破搜索树的平衡，所以为了保证正确性，需要使用一个平衡算法用来管理节点。

当我们把元素插入到树中时，这些键值就会成为邻居(就如整数1和2互邻一样)。如果有 `hint` 传入，那么很容易检查键是否正确。如果这种情况出现，则可以省去搜索的时间。而后，平衡算法会可能还要运行。虽然优化并不是总能成功，不过平均下来，性能上还是会有提升。可以使用多次插入的方式，来统计运行的耗时，这被称之为**摊销复杂度**。



如果插入的 `hint` 是错的，那么插入函数会放弃使用 `hint`，转而使用搜索算法进行查找。虽然程序不会出什么问题，但这样做会让程序变慢。

高效的修改**std::map**元素的键值

在 `std::map` 数据结构中，键-值通常都对应存在，而且键通常是唯一并排序过的，而且键值一旦设定那么就不允许用户再进行修改。为了阻止用户修改键，键的类型声明使用了 `const`。

这种限制是非常明智的，其可以保证用户很难在使用 `std::map` 的时候出错。不过，如果我们真的需要修改 `map` 的键值该怎么办呢？

C++17之前，因为对应的键已经存在，我们不得不将整个键-值对从树中移除，然后再插入。这种方法的确定很明显，其需要分配出一些不必要的内存，感觉上也会对性能有一定的影响。

从C++17起，我们无需重新分配内存，就可以删除和重新插入map键值对。下面的内容中将会展示如何操作。

How to do it...

我们使用 `std::map` 类型一个实现应用，用于确定车手在虚拟比赛中的排位。当车手在比赛中完成超越，那么我们将使用C++17的新方法改变其键值。

1. 包含必要的头文件和声明使用的命名空间。

```
#include <iostream>
#include <map>

using namespace std;
```

2. 我们会在修改map的时候打印之前和之后结果，所以这里先实现了一个辅助函数。

```
template <typename M>
void print(const M &m)
{
    cout << "Race placement:\n";
    for (const auto &[placement, driver] : m) {
        cout << placement << ":" << driver <<
    '\n';
    }
}
```

3. 主函数中，我们实例化并初始化一个 `map`，其键为整型，表示是当前的排位；值为字符型，表示驾驶员的姓名。我们在这里先打印一下这个 `map`，因为我们在下一步对其进行修改。

```
int main()
{
    map<int, string> race_placement {
        {1, "Mario"}, {2, "Luigi"}, {3, "Bowser"},
        {4, "Peach"}, {5, "Yoshi"}, {6, "Koopa"},
        {7, "Toad"}, {8, "Donkey Kong Jr."}
    };
    print(race_placement);
}
```

4. 让我来看下排位赛的某一圈的情况，**Bowser**因为赛车事故排在最后，**Donkey Kong Jr.**从最后一名超到第三位。例子中首先要从 `map` 中提取节点，因为这是唯一能修改键值的方法。`extract` 函数是C++17新加的特性。其可以从 `map` 中删除元素，并没有内存重分配的副作用。看下这里是怎么用的吧。

```
{
    auto a(race_placement.extract(3));
    auto b(race_placement.extract(8));
```

5. 现在我们要交换**Bowser**和**Donkey Kong Jr.**的键。键通常都是无法修改的，不过我们可以通过 `extract` 方法来修改元素的键。

```
swap(a.key(), b.key());
```

6. `std::map` 的 `insert` 函数在C++17中有一个新的重载版本，其接受已经提取出来的节点，就是为了在插入他们时，不会分配不必要的内存。

```

    race_placement.insert(move(a));
    race_placement.insert(move(b));
}
```

7. 最后，我们打印一下目前的排位。

```

    print(race_placement);
}
```

8. 编译并运行可以得到如下输出。我们可以看到初始的排位和最后的排位。

```
$ ./mapnode_key_modification
Race placement:
1: Mario
2: Luigi
3: Bowser
4: Peach
5: Yoshi
6: Koopa
7: Toad
8: Donkey Kong Jr.
Race placement:
1: Mario
2: Luigi
3: Donkey Kong Jr.
4: Peach
5: Yoshi
6: Koopa
7: Toad
8: Bowser
```

How it works...

在C++17中，`std::map` 有一个新成员函数 `extract`。其有两种形式：

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

在例子中，我们使用了第二个，能接受一个键值，然后找到这个键值，并提取对应的 `map` 节点。第一个函数接受一个迭代器，提取的速度会更快，应为给定了迭代器就不需要在查找。

当使用第二种方式去提取一个不存在的节点时，会返回一个空 `node_type` 实例。`empty()` 成员函数会返回一个布尔值，用来表明 `node_type` 实例是否为空。以任何方式访问一个空的实例都会产生未定义行为。

提取节点之后，我们要使用 `key()` 函数获取要修改的键，这个函数会返回一个非常量的键。

需要注意的是，要将节点重新插会到 `map` 时，我们需要在 `insert` 中移动他们。因为 `extract` 可避免不必要的拷贝和内存分配。还有一点就是，移动一个 `node_type` 时，其不会让容器的任何值发生移动。

There's more...

使用 `extract` 方法提取的 `map` 节点实际上非常通用。我们可以从一个 `map` 实例中提取出来节点，然后插入到另一个 `map` 中，甚至可以插入到 `multimap` 实例中。这种方式在 `unordered_map` 和 `unordered_multimap` 实例中也适用。同样

在 `set/multiset` 和 `unordered_set/unordered_multiset` 也适用。

为了在不同 `map` 或 `set` 结构中移动元素，键、值和分配器的类型都必须相同。需要注意的是，不能将 `map` 中的节点移动到 `unordered_map` 中，或是将 `set` 中的元素移动到 `unordered_set` 中。

std::unordered_map中使用自定义类型

当我们使用 `std::unordered_map` 代替 `std::map` 时，对于键的选择要从另一个角度出发。`std::map` 要求键的类型可以排序。因此，元素可以进行排序。不过，当我们使用数学中的向量作为键呢？这样一来就没有判断哪个向量大于另一个向量，比如向量(0, 1)和(1, 0)无法相比较，因为它们指向的方向不同。在 `std::unordered_map` 中这都不是问题，因为不需要对键的哈希值进行排序。对于我们来说只要为类型实现一个哈希函数和等同`==`操作符的实现，等同操作符的是实现是为了判断两个对象是否完全相同。本节中，我们就来实验一下这个例子。

How to do it...

本节中，我们要定义一个简单的 `coord` 数据结构，其没有默认哈希函数，所以我们必须要自行定义一个。然后我们会使用 `coord` 对象来对应一些值。

1. 包含使用 `std::unordered_map` 所必须的头文件

```
#include <iostream>
#include <unordered_map>
```

2. 自定义数据结构，这是一个简单的数据结构，还不具备对应的哈希函数：

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

3. 实现哈希函数是为了能让类型作为键存在，这里先实现比较操作函数：

```
bool operator==(const coord &l, const coord &r)
{
    return l.x == r.x && l.y == r.y;
}
```

4. 为了使用STL哈希的能力，我们打开了std命名空间，并且创建了一个特化的 `std::hash` 模板。其使用 `using` 将特化类型进行别名。

```
namespace std
{
template <>
struct hash<coord>
{
    using argument_type = coord;
    using result_type = size_t;
```

5. 下面要重载该类型的括号表达式。我们只是为 `coord` 结构体添加数字，这是一个不太理想的哈希方式，不过这里只是展示如何去实现这个函数。一个好的散列函数会尽可能的将值均匀的分布在整個取值范围内，以减少哈希碰撞。

```
    result_type operator() (const argument_type &c)
const
{
    return static_cast<result_type>(c.x)
        + static_cast<result_type>(c.y);
}
};
```

6. 我们现在可以创建一个新的 `std::unordered_map` 实例，其能结构 `coord` 结构体作为键，并且对应任意值。例子中对 `std::unordered_map` 使用自定义的类型来说，已经很不错了。让我们基于哈希进行实例化，并填充自定义类型的 `map` 表，并打印这个 `map` 表：

```
int main()
{
    std::unordered_map<coord, int> m {
        { {0, 0}, 1 },
        { {0, 1}, 2 },
        { {2, 1}, 3 }
    };
    for (const auto & [key, value] : m) {
        std::cout << "(" << key.x << ", " << key.y
            << "): " << value << "\n";
    }
    std::cout << '\n';
}
```

7. 编译运行这个例子，就能看到如下的打印信息：

```
$ ./custom_type_unordered_map
{(2, 1): 3} {(0, 1): 2} {(0, 0): 1}
```

How it works...

通常实例化一个基于哈希的 `map` 表(比如: `std::unordered_map`)时，我们会这样写：

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

编译器为我们创建特化的 `std::unordered_map` 时，这句话背后隐藏了大量的操作。所以，让我们来看一下其完整的模板类型声明：

```
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;
```

这里第一个和第二个模板类型，我么填写的是 `coord` 和 `int`。另外的三个模板类型是选填的，其会使用已有的标准模板类。这里前两个参数需要我们给定对应的类型。

对于这个例子，`class Hash` 模板参数是最有趣的一个：当我们不显式定义任何东西时，其就指向 `std::hash<key_type>`。STL已经具有 `std::hash` 的多种特化类型，比如 `std::hash<std::string>`、`std::hash<int>`、`std::hash<unique_ptr>` 等等。这些类型中可以选择最优的一种类型类解决对应的问题。

不过，STL不知道如何计算我们自定义类型 `coord` 的哈希值。所以我们要使用我们定义的类型对哈希模板进行特化。编译器会从 `std::hash` 特化列表中，找到我们所实现的类型，也就是将自定义类型作为键的类型。

如果新特化一个 `std::hash<coord>` 类型，并且将其命名成 `my_hash_type`，我们可以使用下面的语句来实例化这个类型：

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

这样命名就很直观，可读性好，而且编译器也能从哈希实现列表中找到与之对应的正确的类型。

过滤用户的重复输入，并以字母序将重 复信息打印出——`std::set`

`std::set` 是一个奇怪的容器：工作原理和 `std::map` 很像，不过 `std::set` 将键作为值，没有键值对。所以没办法与其他类型的数据进行映射。表面上看，`std::set` 因为没有太多的例子，导致很多开发者几乎不知道有这样的容器。想要使用类似 `std::set` 的功能时，只有自己去实现一遍。

本节展示如何使用 `std::set` 收集很多不同的元素，过滤这些元素，最后只输出一个元素。

How to do it...

从标准输入流中读取单词，所有不重复的单词将放到一个 `std::set` 实例中。之后，枚举出所有输入流中不重复的单词。

1. 包含必要的头文件。

```
#include <iostream>
#include <set>
#include <string>
#include <iterator>
```

2. 为了分析我们的输入，会使用到 `std` 命名空间：

```
using namespace std;
```

3. 现在来实现主函数，先来实例化一个 `std::set`。

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

4. 下一件事情就是获取用户的输入。我们只从标准输入中读取，这样我们就要用到 `istream_iterator`。

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

5. 这样就得到了一对 `begin` 和 `end` 迭代器，可以用来表示用户的输入，我们可以使用 `std::inserter` 来填满 `set` 实例。

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

6. 这样就完成了填充。为了看到从标准输入获得的不重复单词，我们可以打印当前的 `set` 实例。

```
for (const auto word : s) {
    cout << word << ", ";
}
cout << '\n';
}
```

7. 最后，让我们编译并运行这个程序。从之前的输入中，重复的单词都会去除，获得不重复的单词，然后以字母序排序输出。

```
$ echo "a a a b c foo bar foobar foo bar bar" |
./program
a, b, bar, c, foo, foobar,
```

How it works...

程序中有两个有趣的部分。第一个是使用了 `std::istream_iterator` 来访问用户输入，另一个是将 `std::set` 实例使用 `std:: inserter` 用包装后，在使用 `std::copy` 填充。这看起来像是变魔术一样，只需要一行代码，我们就能完成使用输入流填充实例，去除重复的单词和以字母序进行排序。

`std::istream_iterator`

这个例子的有趣之处在于一次性可以处理流中大量相同类型的数据：我们对整个输入进行逐字的分析，并以 `std::string` 实例的方式插入 `set`。

`std::istream_iterator` 只传入了一个模板参数。也就我们输入数据的类型。我们选择 `std::string` 是因为我们假设是文本输入，不过这里也可以是 `float` 型的输入。基本上任何类型都可以使用 `cin >> var;` 完成。构造函数接受一个 `istream` 实例。标准输入使用全局输入流 `std::cin` 表示，例子中其为 `istream` 的参数。

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

输入流迭代器 `it` 就已经实例化完毕了，其可以做两件事情：当对其解引用(`*it`)时，会得到当前输入的符号。我们通过输入迭代器构造 `std::string` 实例，每个字符串容器中都包含一个单词；当进行自加 `++it` 时，其会跳转到下一个单词，然后再解引用访问下一个单词。

不过，每次自加后的解引用时都须谨慎。当标准输入为空，迭代器就不能再解引用。另外，我们需要终止使用解引用获取单词的循环。终止的条件就是通过和 `end` 迭代器进行比较，知道何时迭代器无法解引用。如果 `it==end` 成立，那么说明输入流已经读取完毕。

我们在创建 `it` 的同时，也创建了一个 `std::istream_iterator` 的 `end` 迭代器。其目的是于 `it` 进行比较，在每次迭代中作为中止条件。

当 `std::cin` 结束时，`it` 迭代器将会与 `end` 进行比较，并返回 `true`。

std::inserter

调用 `std::copy` 时，我们使用 `it` 和 `end` 作为输入迭代器。第三个参数必须是一个输出迭代器。因此，不能使用 `s.begin()` 或 `s.end()`。一个空的 `set` 中，这二者是一致的，所以不能对 `set` 的迭代器进行解引用(无论是读取或赋值)。

这就使 `std::inserter` 有了用武之地。其为一个函数，返回一个 `std::insert_iterator`，返回值的行为类似一个迭代器，不过会完成普通迭代器无法完成的事。当对其使用加法时，其不会做任何事。当我们对其解引用，并赋值给它时，它会连接相关容器，并且将赋值作为一个新元素插入容器中。

当通过 `std::inserter` 实例化 `std::insert_iterator` 时，我们需要两个参数：

```
auto insert_it = inserter(s, s.end());
```

其中 `s` 是我们的 `set`，`s.end()` 是指向新元素插入点的迭代器。对于一个空 `set` 来说，从哪里开始和从哪里结束一样重要。当使用其他数据结构时，比如 `vector` 和 `list`，第二个参数对于定义插入新项的位置来说至关重要。

将二者结合

最后，所有的工作都在 `std::copy` 的调用中完成：

```
copy(input_iterator_begin, input_iterator_end,
      insert_iterator);
```

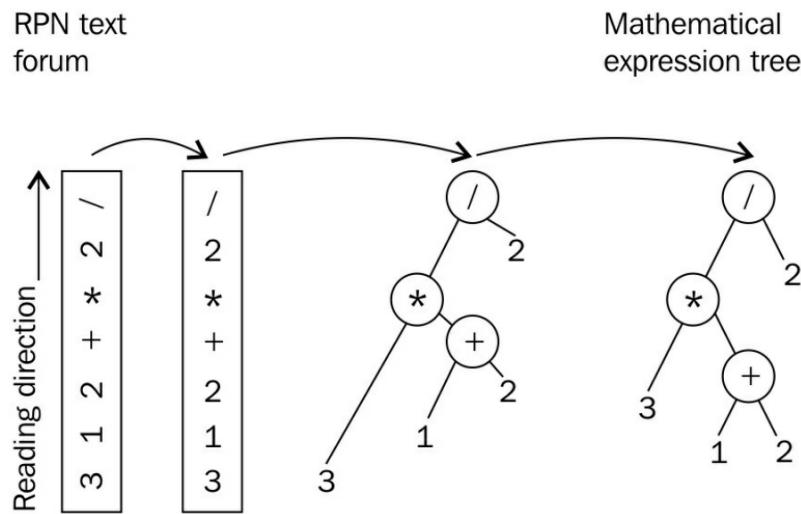
这个调用从 `std::cin` 中获取输入迭代器，并将其推入 `std::set` 中。之后，其会让迭代器自增，并且确定输入迭代器是否达到末尾。如果不是，那么可以继续从标准输入中获取单词。

重复的单词会自动去除。当 `set` 已经拥有了一个单词，再重复将这个单词添加入 `set` 时，不会产生任何效果。与 `std::multiset` 的表现不同，`std::multiset` 会接受重复项。

实现简单的逆波兰表示法计算器—— `std::stack`

`std::stack` 是一个适配类，其能让用户使用自己定义的类型作为栈中的元素。本节中，我们会使用 `std::stack` 构造一个逆波兰(RPN, reverse polish notation)计算器，为了展示如何使用 `std::stack`。

RPN是一种记号法，可以用一种非常简单的解析方式来表达数学表达式。在RPN中，`1+2`解析为`1 2 +`。操作数优先，然后是操作符。另一个例子：`(1+2)*3`表示为`1 2 + 3 *`。这两个例子已经展示了RPN可以很容易的进行解析，并且不需要小括号来定义子表达式。



How to do it...

本节中，我们将从标准输入中读取一个RPN表达式，然后根据表达式解析出正确的计算顺序，并得到结果。最后，我们将输出得到的结果。

1. 包含必要的头文件。

```
#include <iostream>
#include <stack>
#include <iterator>
#include <map>
#include <sstream>
#include <cassert>
#include <vector>
#include <stdexcept>
#include <cmath>
```

2. 声明所使用的命名空间。

```
using namespace std;
```

3. 然后，就来实现我们的RPN解析器。其能接受一对迭代器，两个迭代器分别指定了数学表达式的开始和结尾。

```
template <typename IT>
double evaluate_rpn(IT it, IT end)
{
```

4. 在遍历输入时，需要记住所经过的所有操作数，直到我们看到一个操作符为止。这也就是使用栈的原因。所有数字将会被解析出来，然后以双精度浮点类型进行保存，所以保存到栈中的数据类型为 `double`。

```
stack<double> val_stack;
```

5. 为了能更方便的访问栈中的元素，我们实现了一个辅助函数。其会修改栈中内容，弹出最顶端的元素，并返回这个元素。

```
auto pop_stack ([&] () {
    auto r (val_stack.top());
    val_stack.pop();
    return r;
});
```

6. 另一项准备工作，就是定义所支持的数学操作符。我们使用 `map` 保存相关数学操作符的作用。每个操作符的实现我们使用Lambda函数实现。

```
map<string, double (*)(double, double)> ops {
    {"+", [] (double a, double b) { return a + b; },
     {"-", [] (double a, double b) { return a - b; },
      {"*", [] (double a, double b) { return a * b; },
       {"/", [] (double a, double b) { return a / b; },
        {"^", [] (double a, double b) { return pow(a, b); } },
        {"%", [] (double a, double b) { return fmod(a, b); } },
      };
    };
};
```

7. 现在就可以对输入进行遍历了。假设我们的输入是字符串，我们使用全新的 `std::stringstream` 获取每个单词，这样就可以将操作数解析为数字了。

```
for ( ; it != end; ++it) {
    stringstream ss {*it};
```

8. 我们获得的每个操作数，都要转换成 `double` 类型。如果当前解析的字符是操作数，那么我们将转换类型后，推入栈中。

```
if (double val; ss >> val) {
    val_stack.push(val);
}
```

9. 如果不是操作数，那么就必定为一个操作符。我们支持的操作符都是二元的，所以当遇到操作符时，我们需要从栈中弹出两个操作数。

```
else {
    const auto r {pop_stack()};
    const auto l {pop_stack()};
```

10. 现在我们可以从解引用迭代器 `it` 获取操作数。通过查询 `ops_map` 表，我们可以获得参与 `Lambda` 计算的 `l` 和 `r` 值。

```
try {
    const auto & op (ops.at(*it));
    const double result {op(l, r)};
    val_stack.push(result);
}
```

11. 我们使用 `try` 代码块将计算代码包围，因为我们的计算可能会出错。在调用 `map` 的成员函数 `at` 时，可能会抛出一个 `out_of_range` 异常，由于用户具体会输入什么样的表达式，并不是我们能控制的。所以，我们将会重新抛出一个不同的异常，我们称之为 `invalid_argument` 异常，并且携带着程序未知的操作符。

```
catch (const out_of_range &) {
    throw invalid_argument(*it);
}
```

12. 这就是遍历循环的全部，我们会将栈中的操作数用完，然后得到对应的结果，并将结果保存在栈顶。所以我们要返回栈顶的元素。(我们对栈的大小进行断言，如果大小不是1，那么就有缺失的操作符)

```
}
}
return val_stack.top();
}
```

13. 现在我们可以使用这个RPN解析器了。为了使用这个解析器，我们需要将标准输入包装成一个 `std::istream_iterator` 迭代器对，并且传入RPN解析器函数。最后，我们将输出结果：

```
int main()
{
    try {
        cout << evaluate_rpn(istream_iterator<string>
{cin}, {});
        << '\n';
    }
}
```

14. 这里我们再次使用了 `try` 代码块，因为用户输入的表达式可能会存在错误，所以当解析器抛出异常时，需要在这里获取。我们需要获取对应的异常，并且打印出一条错误信息：

```
    catch (const invalid_argument &e) {
        cout << "Invalid operator: " << e.what() <<
        '\n';
    }
}
```

15. 完成编译步骤后，我们就可以使用这个解析器了。输入 `3 1 2 + * 2 /`，其为 $(3*(1+2))/2$ 数学表达式的RPN表达式，然后我们获得相应的结果：

```
$ echo "3 1 2 + * 2 /" | ./rpn_calculator
4.5
```

How it works...

整个例子通过解析我们的输入，持续向栈中压入操作数的方式完成相应的数学计算。本例中，我们会从栈中弹出最后两个操作数，然后使用操作符对这两个操作数进行计算，然后将其结果保存在栈中。为了理解本节中的所有代码，最重要的就是要理解，我们如何区分了输入中的操作数和操作符，如何管理我们的栈，以及如何选择正确的计算操作符。

栈管理

我们使用 `std::stack` 中的成员函数 `push` 将元素推入栈中：

```
val_stack.push(val);
```

出站元素的获取看起来有些复杂，因为我们使用了一个Lambda表达式完成这项操作，其能够引用 `val_stack` 对象。这里我们为代码添加了一些注释，可能会更好理解一些：

```
auto pop_stack ([&]() {
    auto r (val_stack.top()); // 获取栈顶元素副本
    val_stack.pop(); // 从栈中移除顶部元素
    return r; // 返回顶部元素副本
});
```

这个**Lambda**表达式能够一键式获取栈顶元素，并且能删除顶部元素。

在 `std::stack` 的设计当中，无法使用一步完成这些操作。不过，定义一个**Lambda**函数也是十分快捷和简介，所以我们可以使用这种方式获取值：

```
double top_value {pop_stack();}
```

从输入中区别操作数和操作符

主循环中执行 `evaluate_rpn` 时，我们会根据迭代器遍历标准输入，然后判断字符是一个操作数，还是一个操作符。如果字符可以被解析成 `double` 变量，那这就是一个数，也就是操作数。我们需要考虑有些比较难以解析的数值(比如，`+1`和`-1`)，这种数值可能会被解析成操作符(尤其是`+1`这种)。

用于区分操作数和操作符的代码如下所示：

```
stringstream ss {*it};
if (double val; ss >> val) {
    // It's a number!
} else {
    // It's something else than a number - an operation!
}
```

如果字符是一个数字，流操作符 `>>` 会告诉我们。首先，我们将字符串包装成一个 `std::stringstream`。然后使用 `stringstream` 对象的能力，将流中 `std::string` 类型解析并转换成一个 `double` 变量。解析失败时也能知道是什么，因为只解析器需要解析数字出来；否则，需要解析的就不是一个数字。

选择和应用正确的数学操作符

判断完当前用户的输入是否为一个数后，我们先假设输入了一个操作符，比如 `+` 或 `*`。然后，查询 `map` 表 `ops`，找到对应的操作，并返回相应的函数，其函数可以接受两个操作数，然后返回对应操作后的结果。

`map` 表本身的类型看起来会相对复杂：

```
map<string, double (*) (double, double)> ops { ... };
```

其将 `string` 映射到 `double (*) (double, double)`。后者是什么意思呢？这个类型是一个函数指针的声明，说明这个函数接受两个 `double` 类型的变量作为输入，并且返回值也是 `double` 类型。可以将 `(*)` 部分理解成函数的名字，例如 `double sum(double, double)`，这样就好理解多了吧。这里的重点在于我们的**Lambda**函数 `[](double, double) {return /* some double */ }`，其可转换为实际匹配指针声明的函数。这里 **Lambda**不获取任何东西，所以可以转化为函数指针。

这样，我们就可以方便的在 `map` 表中查询操作符是否支持：

```
const auto & op (ops.at(*it));
const double result {op(l, r)};
```

`map` 会为我们隐式的做另一件事：当我们执行 `ops.at("foo")` 时，如果 "foo" 是一个合法键(实际中我们不会用这个名字存任何操作)，那么在这个例子中，`map` 表将会抛出一个异常，例子中可以捕获这个异常。当我们捕获这个异常时，我们会重新抛出一个不同的异常，为了描述我们遇到了什么样的错误。相较于 `out of range`，用户也能更好的了解 `invalid argument` 异常的含义，因此我们在使用的时候，程序的 `map` 表到底支持哪些操作，我们是不知道的。

There's more...

`evaluate_rpn` 函数可以传入迭代器，感觉这样传递的方式要比传入标准输入更容易理解。这让程序更容易测试，或适应来自于用户的不同类型的输入。

使用字符串流或字符串数组的迭代器作为输入，例如下面的代码，`evaluate_rpn` 不用做任何修改：

```
int main()
{
    stringstream s {"3 2 1 + * 2 /"};
    cout << evaluate_rpn(istream_iterator<string>{s}, {})
    << '\n';
    vector<string> v {"3", "2", "1", "+", "*", "2", "/"};
    cout << evaluate_rpn(begin(v), end(v)) << '\n';
}
```

Note:

在有意义的地方使用迭代器，会使得代码可重复利用度高，模块化好。

实现词频计数器——`std::map`

`std::map` 在收集和统计数据方面非常有用，通过建立键值关系，将可修改的对象映射到对应键上，可以很容易的实现一个词频计数器。

How to do it...

本节中，我们将从标准输入中获取用户的输入，或是从记录一部小说的文本文件。我们会去标记输入单词，并统计一共有多少个单词。

1. 包含必要的头文件。

```
#include <iostream>
#include <map>
#include <vector>
#include <algorithm>
#include <iomanip>
```

2. 声明所使用的命名空间。

```
using namespace std;
```

3. 我们将使用一个辅助函数，对输入中的符号进行处理。

```
string filter_punctuation(const string &s)
{
    const char *forbidden ".,:; ";
    const auto idx_start
(s.find_first_not_of(forbidden));
    const auto idx_end
(s.find_last_not_of(forbidden));
    return s.substr(idx_start, idx_end - idx_start +
1);
}
```

4. 现在，我们来实现真正要工作的部分。使用 `map` 表对输入的每个单词进行统计。另外，使用一个变量来保存目前为止看到的最长单词的长度。程序的最后，我们将打印这个 `map` 表。

```
int main()
{
    map<string, size_t> words;
    int max_word_len {0};
```

5. 将标准输入导入 `std::string` 变量中，标准输入由空格隔开。通过如下方法获取输入单词。

```
string s;
while (cin >> s) {
```

6. 我们获得的单词可能包含标点符号，因为这些符号可能紧跟在单词后面。使用辅助函数将标点符号去除。

```
auto filtered (filter_punctuation(s));
```

7. 如果当前处理的单词是目前处理最长的单词，我们会更新 `max_word_len` 变量。

```
max_word_len = max<int>(max_word_len,
filtered.length());
```

8. 然后，我们将增加该词在 `words map` 中的频率。如果是首次处理该单词，那么将会隐式创建一个键值对，然后插入 `map`，之后再进行自加操作。

```
++words[filtered];
}
```

9. 当循环结束时，`words map` 会保存所有输入单词的频率。`map` 中单词作为键，并且键以字母序排列。我们想要以频率多少进行排序，词频最高的排第一位。为了达到这样的效果，首先实现一个 `vector`，将所有键值对放入这个 `vector` 中。

```
vector<pair<string, size_t>> word_counts;
word_counts.reserve(words.size());
move(begin(words), end(words),
back_inserter(word_counts));
```

10. 然后，`vector` 中将将具有 `words map` 中的所有元素。然后，我们来进行排序，把词频最高的单词排在最开始，最低的放在最后。

```
sort(begin(word_counts), end(word_counts),
[](const auto &a, const auto &b) {
    return a.second > b.second;
});
```

11. 现在所有元素如我们想要的顺序排列，之后将这些数据打印在用户的终端上。使用 `std::setw` 流控制器，可以格式化输出相应的内容。

```

        cout << "# " << setw(max_word_len) << "<WORD>" <<
" #<COUNT>\n";
    for (const auto & [word, count] : word_counts) {
        cout << setw(max_word_len + 2) << word << "
#"
        << count << '\n';
    }
}

```

12. 编译后运行，我们就会得到一个词频表：

```

$ cat lorem_ipsum.txt | ./word_frequency_counter
# <WORD> #<COUNT>
et #574
dolor #302
sed #273
diam #273
sit #259
ipsum #259
...

```

How it works...

本节中，我们使用 `std::map` 实例进行单词统计，然后将 `map` 中的所有元素放入 `vector` 中，然后进行排序，再打印输出。为什么要这么做？

先看一个例子。当我们要从 `a a b c b b b d c c` 字符串中统计词频时，我们的 `map` 内容如下：

```

a -> 2
b -> 4
c -> 3
d -> 1

```

不过，这是未排序的，这不是我们想要给用户展示的排序。我们的程序要首先输出 `b` 的频率，因为 `b` 的频率最高。然后是 `c, a, d`。不幸的是，我们无法要求 `map` 使用键所对应的值进行排序。

这就需要 `vector` 帮忙了，将 `map` 中的键值对放入 `vector` 中。这个方法明确的将这些元素从 `map` 中删除了。

```

vector<pair<string, size_t>> word_counts;

```

然后，我们使用 `std::move` 函数将词-频对应关系填充整个 `vector`。这样的好处是让单词不会重复，不过这样会将元素从 `map` 中完全删除。使用 `move` 方法，减少了很多不必要的拷贝。

```
move(begin(words), end(words),
back_inserter(word_counts));
```

Note

一些STL的实现使用短字符优化——当所要处理的字符串过长，这种方法将无需再在堆上分配内存，并且可以将字符串直接进行存储。在这个例子中，移动虽然不是最快的方式，但也不会慢多少。

接下来比较有趣的就是排序操作，其使用了一个**Lambda**表达式作为自定义比较谓词：

```
sort(begin(word_counts), end(word_counts),
[] (const auto &a, const auto &b) { return a.second >
b.second; });
```

排序算法将会成对的处理元素，比较两个元素。通过提供的**Lambda**函数，`sort`方法将不会再使用默认比较谓词，其会将 `a.second` 和 `b.second` 进行比较。这里的键值对中，第二个值为词频数，所以可以使用 `.second` 得到对应词频数。通过这种方式，将移动所有高频率的词到 `vector` 的开始，并且将低频率词放在末尾。

实现写作风格助手用来查找文本中很长的句子——`std::multimap`

当有超级多的元素需要排序时，某些键值描述可能会出现多次，那么使用 `std::multimap` 完成这项工作无疑是个不错的选择。

先找个应用场景：当使用德文写作时，使用很长的句子是没有问题的。不过，使用英文时，就不行了。我们将实现一个辅助工具来帮助德国作家们分析他们的英文作品，着重于所有句子的长度。为了帮助这些作家改善其写作的文本风格，工具会按句子的长度对每个句子进行分组。这样作家们就能挑出比较长的句子，然后截断这些句子。

How to do it...

本节中，我们将从标准输入中获取用户输入，用户会输入所有的句子，而非单词。然后，我们将这些句子和其长度收集在 `std::multimap` 中。之后，我们将对所有句子的长度进行排序，打印给用户看。

1. 包含必要的头文件。`std::multimap` 和 `std::map` 在同一个头文件中声明。

```
#include <iostream>
#include <iterator>
#include <map>
#include <algorithm>
```

2. 声明所使用的命名空间。

```
using namespace std;
```

3. 我们使用句号将输入字符串分成若干个句子，句子中的每个单词以空格隔开。

句子中的一些对于句子长度无意义的符号，也会计算到长度中，所以，这里要使用辅助函数将这些符号过滤掉。

```
string filter_ws(const string &s)
{
    const char *ws {" \r\n\t"};
    const auto a (s.find_first_not_of(ws));
    const auto b (s.find_last_not_of(ws));
    if (a == string::npos) {
        return {};
    }
    return s.substr(a, b);
}
```

4. 计算句子长度函数需要接收一个包含相应内容的字符串，并且返回一个 `std::multimap` 实例，其映射了排序后的句子长度和相应的句子。

```
multimap<size_t, string> get_sentence_stats(const  
string &content)  
{
```

5. 这里声明一个 `multimap` 结构，以及一些迭代器。在计算长度的循环中，我们需要 `end` 迭代器。然后，我们使用两个迭代器指向文本的开始和结尾。所有句子都在这个文本当中。

```
multimap<size_t, string> ret;  
const auto end_it (end(content));  
auto it1 (begin(content));  
auto it2 (find(it1, end_it, '.'));
```

6. `it2` 总是指向句号，而 `it1` 指向句子的开头。只要 `it1` 没有到达文本的末尾就好。第二个条件就是要检查 `it2` 是否指向字符。如果不满足这些条件，那么就意味着这两个迭代器中没有任何字符了：

```
while (it1 != end_it && distance(it1, it2) > 0) {
```

7. 我们使用两个迭代器间的字符创建一个字符串，并且过滤字符串中所有的空格，只是为了计算句子纯单词的长度。

```
string s {filter_ws({it1, it2});}
```

8. 当句子中不包含任何字符，或只有空格时，我们就不统计这句。另外，我们要计算有多少单词在句子中。这很简单，每个单词间都有空格隔开，单词的数量很容易计算。然后，我们就将句子和其长度保存在 `multimap` 中。

```
if (s.length() > 0) {  
    const auto words (count(begin(s), end(s),  
    ' ') + 1);  
    ret.emplace(make_pair(words, move(s)));  
}
```

9. 对于下一次循环迭代，我们将会让 `it1` 指向 `it2` 的后一个字符。然后将 `it2` 指向下一个句号。

```
it1 = next(it2, 1);  
it2 = find(it1, end_it, '.');
```

10. 循环结束后，`multimap` 包含所有句子以及他们的长度，这里我们将其返回。

```
    return ret;
}
```

11. 现在，我们来写主函数。首先，我们让 `std::cin` 不要跳过空格，因为我们需要句子中有空格。为了读取整个文件，我们使用 `std::cin` 包装的输入流迭代器初始化一个 `std::string` 实例。

```
int main()
{
    cin.unsetf(ios::skipws);
    string content {istream_iterator<char>{cin}, {}};
```

12. 只需要打印 `multimap` 的内容，在循环中调用 `get_sentence_stats`，然后打印 `multimap` 中的内容。

```
for (const auto & [word_count, sentence]
      : get_sentence_stats(content)) {
    cout << word_count << " words: " << sentence
    << ".\n";
}
```

13. 编译完成后，我们可以使用一个文本文件做例子。由于长句子的输出量很长，所以先把最短的句子打印出来，最后打印最长的句子。这样，我们就能首先看到最长的句子。

```
$ cat lorem_ipsum.txt | ./sentence_length
...
10 words: Nam quam nunc, blandit vel, luctus
pulvinar,
hendrerit id, lorem.
10 words: Sed consequat, leo eget bibendum sodales,
augue velit cursus nunc..
12 words: Cum sociis natoque penatibus et magnis dis
parturient montes, nascetur ridiculus mus.
17 words: Maecenas tempus, tellus eget condimentum
rhoncus,
sem quam semper libero, sit amet adipiscing sem neque
sed ipsum.
```

How it works...

整个例子中，我们将一个很长的字符串，分割成多个短句，从而评估每个句子的长度，并且在 `multimap` 中进行排序。因为 `std::multimap` 很容易使用，所以变成较为复杂的部分就在于循环，也就是使用迭代器获取每句话的内容。

```

const auto end_it (end(content));

// (1) Beginning of string
auto it1 (begin(content));

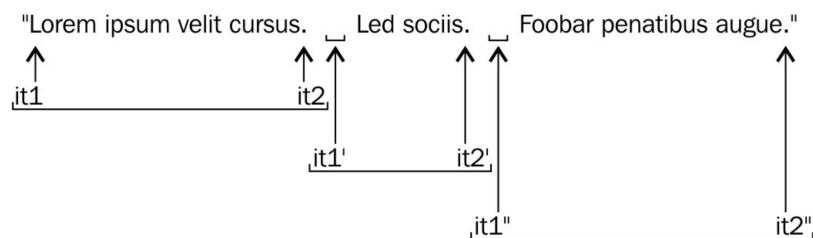
// (1) First '.' dot
auto it2 (find(it1, end_it, '.'));
while (it1 != end_it && std::distance(it1, it2) > 0) {
    string sentence {it1, it2};
    // Do something with the sentence string...

    // One character past current '.' dot
    it1 = std::next(it2, 1);

    // Next dot, or end of string
    it2 = find(it1, end_it, '.');
}

```

将代码和下面的图结合起来可能会更好理解，这里使用具有三句话的字符串来举例。



① "Lorem ipsum velit cursus."

3 spaces → 4 words

② "Led sociis."

1 space → 2 words

③ "Foobar penatibus augue."

2 spaces → 3 words

`it1` 和 `it2` 总是随着字符串向前移动。通过指向句子的开头和结尾的方式，确定一个句子中的内容。`std::find` 算法会帮助我们寻找下一个句号的位置。

`std::find` 的描述：

从当前位置开始，返回首先找到的目标字符迭代器。如果没有找到，返回结束迭代器。

这样我们就获取了一个句子，然后通过构造对应字符串的方式，将句子的长度计算出来，并将长度和原始句子一起插入 `multimap` 中。我们使用句子的长度作为元素的键，原句作为值存储在 `multimap` 中。通常一个文本中，长度相同的句子有很多。这样使用 `std::map` 就会比较麻烦。不过 `std::multimap` 就没有重复键值的问题。这些键值也是排序好的，从而能得到用户们想要的输出。

There's more...

将整个文件读入一个大字符串中后，遍历字符串时需要为每个句子创建副本。这是没有必要的，这里可以使用 `std::string_view` 来完成这项工作，该类型我们将会放在后面来介绍。

另一种从两个句号中获取句子的方法就是使用 `std::regex_iterator` (正则表达式)，我们将会在后面的章节中进行介绍。

实现个人待办事项列表—— `std::priority_queue`

`std::priority_queue` 是另一种适配容器(类似于 `std::stack`)。其实为另一种数据结构的包装器(默认的数据结构为 `std::vector`)，并且提供类似队列的接口。同样也遵循队列的特性，先进先出。这与我们之前使用的 `std::stack` 完全不同。

这里仅仅是对 `std::queue` 的行为进行描述，本节将展示 `std::priority_queue` 是如何工作的。这个适配器比较特殊，其不仅有FIFO的特性，还混合着优先级。这就意味着，FIFO的原则会在某些条件下被打破，根据优先级的顺序形成子FIFO队列。

How to do it...

本节中，我们将创建一个待办事项的结构。为了程序的简明性就不从用户输入解析输入了。这次专注于 `std::priority_queue` 的使用。所以我们使用一些待办事项和优先级填充一个优先级序列，然后以FIFO的顺序读出这些元素(这些元素是通过优先级进行过分组)。

1. 包含必要的头文件。`std::priority_queue` 在 `<queue>` 中声明。

```
#include <iostream>
#include <queue>
#include <tuple>
#include <string>
```

2. 我们怎么将待办事项存在优先级队列中呢？我们不能添加项目时，附加优先级。优先级队列将使用自然序对待队列中的所有元素。现在我们实现一个自定义的结构体 `struct todo_item`，并赋予其优先级系数，和一个字符串描述待办事件，并且为了让该结构体具有可排序性，这里会实现比较操作符 `<`。另外，我们将会使用 `std::pair`，其能帮助我们聚合两个类型为一个类型，并且能完成自动比较。

```
int main()
{
    using item_type = std::pair<int, std::string>;
```

3. 那么现在我们有了一个新类型 `item_type`，其由一个优先级数字和一个描述字符串构成。所以，我们可以使用这种类型实例化一个优先级队列。

```
std::priority_queue<item_type> q;
```

4. 我们现在来填充优先级队列。其目的就是为了提供一个非结构化列表，之后优先级队列将告诉我们以何种顺序做什么事。比如，你有漫画要看的同时，也有作业需要去做，那么你必须先去写作业。不过，`std::priority_queue` 没有构造

函数，其支持初始化列表，通过列表我们能够填充优先级队列(使用 `vector` 或 `list` 都可以对优先级队列进行初始化)。所以我们这里定义了一个列表，用于下一步的初始化。

```
std::initializer_list<item_type> il {
    {1, "dishes"},
    {0, "watch tv"},
    {2, "do homework"},
    {0, "read comics"},
};
```

5. 现在我们可以很方便的遍历列表中的所有元素，然后通过 `push` 成员函数将元素插入优先级列表中。

```
for (const auto &p : il) {
    q.push(p);
}
```

6. 这样所有的元素就都隐式的进行了排序，并且我们可以浏览列表中优先级最高的事件。

```
while(!q.empty()) {
    std::cout << q.top().first << ":" <<
    q.top().second << '\n';
    q.pop();
}
std::cout << '\n';
}
```

7. 编译运行程序。结果如我们所料，作业是最优先的，看电视和看漫画排在最后。

```
$ ./main
2: do homework
1: dishes
0: watch tv
0: read comics
```

How it works...

`std::priority_queue` 使用起来很简单。我们只是用了其三个成员函数。

1. `q.push(item)` 将元素推入队列中。
2. `q.top()` 返回队首元素的引用。
3. `q.pop()` 移除队首元素。

不过，如何做到排序的呢？我们将优先级数字和描述字符串放入一个 `std::pair` 中，然后就自然得到排序后的结果。这里有一个 `std::pair<int, std::string>` 的实例 `p`，我们可通过 `p.first` 访问优先级整型数，使用 `p.second` 访问字符串。我们在循环中就是这样打印所有待办事件的。

如何让优先级队列意识到 `{2, "do homework"}` 要比 `{0, "watch tv"}` 重要呢？

比较操作符 `<` 在这里处理了不同的元素。我们假设现在有 `left < right`，两个变量的类型都是 `pair`。

1. `left.first != right.first`，然后返回 `left.first < right.first`。
2. `left.first == right.first`，然后返回 `left.second < right.second`。

以这种方式就能满足我们的要求。最重要的就是 `pair` 中第一个成员，然后是第二个成员。否则，`std::priority_queue` 将会字母序将元素进行排序，而非使用数字优先级的顺序(这样的话，看电视将会成为首先要做的事情，而完成作业则是最后一件事。对于懒人来说，无疑是个完美的顺序)。

第3章 迭代器

迭代器是C++中非常重要的概念。STL旨在打造一组灵活和通用的工具集，迭代器是工具集中重要的一环。不过，有时候迭代器使用起来比较繁琐，所以很多编程人员还是喜欢用C的指针来完成相应的功能。一半的编程人员基本上会放弃使用STL中的迭代器。本章介绍了迭代器，并展示如何让它们很快的工作起来。快速地介绍是不能完全覆盖迭代器强大的功能，但是这种小例子能让你增加对迭代器的好感度。

大多数容器类(除了类似C风格的数组)，可包含一系列的数据项。许多日常任务会处理超大的数据量，这里先不关心如何获得这些数据。不过，如果我们考虑数组和链表，并且想要计算这两种结构所有项的和，那么将如下使用两种不同的算法：

- 通过查询数组的大小，来进行加和计算：

```
int sum {0};  
for (size_t i {0}; i < array_size; ++i) { sum +=  
array[i]; }
```

- 使用迭代器进行循环，直到数组的末尾：

```
int sum {0};  
while (list_node != nullptr) {  
    sum += list_node->value; list_node = list_node-  
>next;  
}
```

两种方法都能计算出所有项的加和，不过我们键入的代码，有多少用在实际加和任务中了呢？如果说要使用其他结构体来存储这些数据，例如 `std::map`，难道我们还要在重新实现一个函数？使用迭代器是最佳的选择。

使用迭代器的代码才更加的通用：

```
int sum {0};  
for (int i : array_or_vector_or_map_or_list) { sum += i;  
}
```

这段代码很简洁，只是使用C++11添加的for循环范围特性就完成了整体的叠加。其就像是个语法糖，将其扩展后类似如下代码：

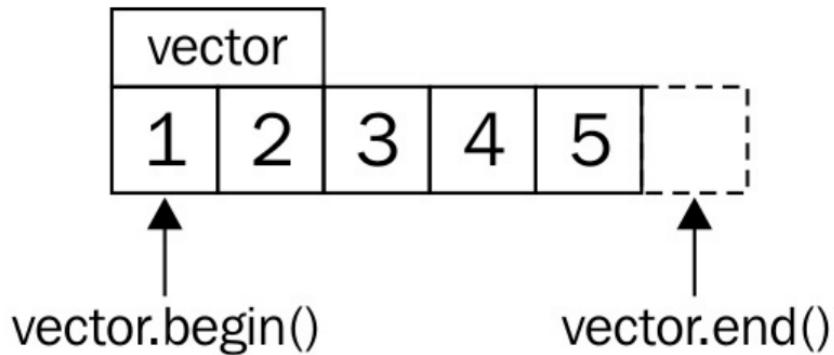
```

{
    auto && __range = array_or_vector_or_map_or_list ;
    auto __begin = std::begin(__range);
    auto __end = std::end(__range);
    for ( ; __begin != __end; ++__begin) {
        int i = *__begin;
        sum += i;
    }
}

```

这段代码对于使用迭代器的老手来说并没有什么，不过对于刚接触迭代器的新手来说就像是在变魔术。

假设我们的 `vector` 内容如下所示：



`std::begin(vector)` 和 `vector.begin()` 等价，并且返回 `vector` 中指向第一个元素的迭代器(指向1)。 `std::end(vector)` 与 `vector.end()` 等价，并返回指向 `vector` 末尾元素的迭代器(指向5的后方)。

每一次迭代，循环都会检查开始迭代器是否与末尾迭代器不同。如果是，那么可以对开始迭代器进行解引用，并获取其指向的值。然后，推动迭代器指向下一个元素，再与末尾迭代器进行比较，以此类推。这也能提升代码的可读性，这样的迭代器就类似于C风格的指针。实际上，C风格的指针也是一种迭代器。

迭代器的类型

C++中很多迭代器类型，都有各自的局限性。不用去死记这些限制，只要记住一种类型的能力是从更强大的类型继承过来的即可。当知道算法是使用何种迭代器实现时，编译器就可以以更好的方式优化这个算法。所以，开发者只要表达清楚自己想要实现的算法，那么编译器将选择优化后的实现来完成对应的任务。

让我们来看下这些迭代器吧(从左往右)：

Iterator category					Multi pass support	Defined operations
				Input Iterator	multiple passes <u>not supported</u>	*it (read-access) + +it or it + +
				Forward Iterator	multiple passes supported	+ +it or it + +
			Bidirectional Iterator			--it or it --
			Random Access Iterator			it + =n or it - =n
	Contiguous Iterator					contiguous storage (like an array)

输入迭代器

只能用来读取指向的值。当该迭代器自加时，之前指向的值就不可访问。也就是说，不能使用这个迭代器在一个范围内遍历多次。`std::istream_iterator` 就是这样的迭代器。

前向迭代器

类似于输入迭代器，不过其可以在指示范围内迭代多次。`std::forward_list` 就是这样的迭代器。就像一个单向链表一样，只能向前遍历，不能向后遍历，但可以反复迭代。

双向迭代器

从名字就能看出来，这个迭代器可以自增，也可以自减，迭代器可以向前或向后迭代。`std::list`，`std::set` 和 `std::map` 都支持双向迭代器。

随机访问迭代器

与其他迭代器不同，随机访问迭代器一次可以跳转到任何容器中的元素上，而非之前的迭代器，一次只能移动一格。`std::vector` 和 `std::deque` 的迭代器就是这种类型。

连续迭代器

这种迭代器具有前述几种迭代器的所有特性，不过需要容器内容在内存上是连续的，类似一个数组或 `std::vector`。

输出迭代器

该迭代器与其他迭代器不同。因为这是一个单纯用于写出的迭代器，其只能增加，并且将对应内容写入文件当中。如果要读取这个迭代中的数据，那么读取到的值就是未定义的。

可变迭代器

如果一个迭代器既有输出迭代器的特性，又有其他迭代器的特性，那么这个迭代器就是可变迭代器。该迭代器可读可写。如果我们从一个非常量容器的实例中获取一个迭代器，那么这个迭代器通常都是可变迭代器。

建立可迭代区域

我们已经认识了STL中提供的各种迭代器。我们只需实现一个迭代器，支持前缀加法`++`，解引用`*`和比较操作`==`，这样我们就能使用C++11基于范围的`for`循环对该迭代器进行遍历。

为了更好的了解迭代器，本节中将展示如何实现一个迭代器。迭代该迭代器时，只输出一组数字。实现的迭代器并不支持任何容器，以及类似的结构。这些数字是在迭代过程中临时生成的。

How to do it...

本节中，我们将实现一个迭代器类，并且对该迭代器进行迭代：

1. 包含必要的头文件。

```
#include <iostream>
```

2. 迭代器结构命名为`num_iterator`：

```
class num_iterator {
```

3. 其数据类型只能是整型，仅用是用来计数的，构造函数会初始化它们。显式声明构造函数是一个好习惯，这就能避免隐式类型转换。需要注意的是，我们会使用`position`值来初始化`i`。这就让`num_iterator`可以进行默认构造。虽然我们的整个例子中都没有使用默认构造函数，但默认构造函数对于STL的算法却是很重要的。

```
    int i;
public:
    explicit num_iterator(int position = 0) :
        i{position} {}
```

4. 当对迭代器解引用时`*it`，将得到一个整数：

```
int operator*() const { return i; }
```

5. 前缀加法操作`++it`：

```
num_iterator& operator++() {
    ++i;
    return *this;
}
```

6. `for`循环中需要迭代器之间进行比较。如果不相等，则继续迭代：

```
    bool operator!=(const num_iterator &other) const
{
    return i != other.i;
}
```

7. 迭代器类就实现完成了。我们仍需要一个中间对象对应于 `for (int i : intermediate(a, b)) {...}` 写法，其会从头到尾的遍历，其为一种从a到b遍历的预编程。我们称其为 `num_range`：

```
class num_range {
```

8. 其包含两个整数成员，一个表示从开始，另一个表示结束。如果我们要从0到9遍历，那么a为0，b为10(`[0, 10]`)：

```
    int a;
    int b;
public:
    num_range(int from, int to)
        : a{from}, b{to}
    {}
```

9. 该类也只有两个成员函数需要实现：`begin` 和 `end` 函数。两个函数都返回指向对应数字的指针：一个指向开始，一个指向末尾。

```
    num_iterator begin() const { return
        num_iterator{a}; }
    num_iterator end() const { return
        num_iterator{b}; }
};
```

10. 所有类都已经完成，让我们来使用一下。让我们在主函数中写一个例子，遍历100到109间的数字，并打印这些数值：

```
int main()
{
    for (int i : num_range{100, 110}) {
        std::cout << i << ", ";
    }
    std::cout << '\n';
}
```

11. 编译运行后，得到如下输出：

```
100, 101, 102, 103, 104, 105, 106, 107, 108, 109,
```

How it works...

考虑一下如下的代码段：

```
for (auto x : range) { code_block; }
```

这段代码将被编译器翻译为类似如下的代码：

```
{
    auto __begin = std::begin(range);
    auto __end = std::end(range);
    for ( ; __begin != __end; ++__begin) {
        auto x = *__begin;
        code_block
    }
}
```

这样看起来就直观许多，也能清楚的了解我们的迭代器需要实现如下操作：

- operator!=
- operator++
- operator*

也需要 `begin` 和 `end` 方法返回相应的迭代器，用来确定开始和结束的范围。

Note:

本书中，我们使用 `std::begin(x)` 替代 `x.begin()`。如果有 `begin` 成员函数，那么 `std::begin(x)` 会自动调用 `x.begin()`。当 `x` 是一个数组，没有 `begin()` 方法时，`std::begin(x)` 会找到其他方式来处理。同样的方式也适用于 `std::end(x)`。当用户自定义的类型不提供 `begin/end` 成员函数时，`std::begin/std::end` 就无法工作了。

本例中的迭代器是一个前向迭代器。再来看一下使用 `num_range` 的循环，从另一个角度看是非常的简单。

Note:

回头看下构造出迭代器的方法在 `range` 类中为 `const`。这里不需要关注编译器是否会因为修饰符 `const` 而报错，因为迭代 `const` 的对象是很常见的事。

让自己的迭代器与**STL**的迭代器兼容

上一节中，我们实现了自己的迭代器，不过为了融合**STL**提供的迭代器的优点，我们需要提供一些迭代器接口。后面我们会来学习如果实现这些接口，不过将我们自定义的迭代器与**STL**的标准迭代器放在一起时，有时会发现有编译不通过的问题。这是为什么呢？

STL算法尝试寻找更多有关于我们所使用迭代器的信息。不同迭代器的能力是不同的，不大可能用同样的算法实现不同的迭代器。例如，我们只是简单的从一个 `std::vector` 将其中的数字拷贝到另一个时，我们的实现中可以直接调用 `memcpy` 快速实现这个功能。如果容器是 `std::list` 的话，`memcpy` 的方式就不好用了，只能一个个的单独拷贝。实现者将大量的自动优化思想注入**STL**算法实现当中。为了能更好的使用，我们也会为我们的迭代器装备这些思想。

How to do it...

本节中，我们将实现一个简单的计数迭代器(与**STL**算法一起使用)，一开始这个实现是无法编译通过的。我们需要做一些兼容性操作，使得程序通过编译。

1. 包含必要的头文件。

```
#include <iostream>
#include <algorithm>
```

2. 实现一个计数迭代器，作为基础版本。当我们使用其进行遍历时，我们只需要增加计数器即可。`num_range` 用来处理 `begin` 和 `end` 迭代器。

```

class num_iterator
{
    int i;
public:
    explicit num_iterator(int position = 0) :
        i{position} {}
    int operator*() const { return i; }
    num_iterator& operator++() {
        ++i;
        return *this;
    }
    bool operator!=(const num_iterator &other) const
    {
        return i != other.i;
    }
    bool operator==(const num_iterator &other) const
    {
        return !(*this != other);
    }
};

class num_range {
    int a;
    int b;
public:
    num_range(int from, int to)
        : a{from}, b{to}
    {}
    num_iterator begin() const { return
        num_iterator{a}; }
    num_iterator end() const { return
        num_iterator{b}; }
};

```

3. 声明所使用的命名空间。

```
using namespace std;
```

4. 现在让我们来遍历100到109间的数字。这里需要注意的是，110这里是开区间，所以值无法遍历到110。

```

int main()
{
    num_range r {100, 110};

```

5. 现在，我们使用一个STL算法 `std::minmax_element`。这个算法会返回一个 `std::pair`，其具有两个迭代器：一个指向最小值的迭代器和一个指向最大值的迭代器。在这个范围中100和109即为这两个迭代器所指向的位置。

```
auto min_max(minmax_element(r.begin(), r.end()));
cout << *min_max.first << " - " <<
*min_max.second << '\n';
}
```

6. 我们在编译的时候遇倒如下的错误信息。这个错误与 `std::iterator_traits` 有关。这个错误可能在使用其他编译器时，错误信息的格式不同，或者就没有错误。这个错误在clang 5.0.0 (trunk 299766)版本出现。

```
error: no type named 'value_type' in 'std::__1::iterator_traits<num_iterator>'  
      __less<typename iterator_traits<_ForwardIterator>::value_type>());  
  
main.cpp:56:24:     in instantiation of function template specialization 'std::__1::minmax_element<num_iterator>' requested here  
        auto min_max (std::minmax_element(std::begin(r), std::end(r)));  
                           ^  
1 error generated.
```

7. 为了修正这个错误，我们需要激活迭代器的迭代功能。之后定义一个 `num_iterator` 结构体，我们会对 `std::iterator_traits` 进行特化。这个特化就是告诉STL我们的 `num_iterator` 是一种前向迭代器，并且指向的对象是 `int` 类型的值。

```
namespace std {  
template <>  
struct iterator_traits<num_iterator> {  
    using iterator_category =  
    std::forward_iterator_tag;  
    using value_type = int;  
};  
}
```

8. 让我们再对程序进行编译，之前的错误应该不存在了。输出了范围内的最大值和最小值：

```
100 - 109
```

How it works...

一些STL算法需要知道其处理容器的迭代器类型，有些还需要知道迭代器所指向的类型。这就是要有不同实现的原因。

不过，所有STL算法将会通过 `std::iterator_traits<my_iterator>` 访问对应类型的迭代器(这里假设迭代器类型为`my_iterator`)。这个特性类需要包含五种不同类型的成员定义：

- `difference_type`: `it1 - it2` 结果的类型
- `value_type`: 迭代器解引用的数据的类型(这里需要注意`void`类型)

- `pointer`: 指向元素指针的类型
- `reference`: 引用元素的类型
- `iterator_category`: 迭代器属于哪种类型

`pointer`、`reference`和`difference_type`并没有在`num_iterator`中定义，因为其实际的内存值不重复(我们只是返回int值，不想数组一样是连续的)。因此`num_iterator`并不需要定义这些类型，因为算法是依赖于解引用后指定内存上的值。如果我们的迭代器定义了这些类型，就可能会出现问题。

There's more...

C++17标准之前，C++都鼓励自定义迭代器继承于`std::iterator<...>`，这样所有主流的类型都会自动定义。C++17中这条建议仍然能工作，但是不再推荐从`std::iterator<...>`继承了。

使用迭代适配器填充通用数据结构

大多数情况下，我们想要用数据来填充任何容器，不过数据源和容器却没有通用的接口。这种情况下，我们就需要人工的去编写算法，将相应的数据推入容器中。不过，这会分散我们解决问题的注意力。

不同数据结构间的数据传递现在可以只通过一行代码就完成，这要感谢STL中的迭代适配器。本节会展示如何使用迭代适配器。

How to do it...

本节中，我们使用一些迭代器包装器，展示如何使用包装器，并了解其如何在编程任务中给予我们帮助。

1. 包含必要的头文件。

```
#include <iostream>
#include <string>
#include <iterator>
#include <sstream>
#include <deque>
```

2. 声明使用的命名空间。

```
using namespace std;
```

3. 开始使用 `std::istream_iterator`。这里我们特化为 `int` 类型。这样，迭代器就能将标准输入解析成整数。例如，当我们遍历这个迭代器，其就和 `std::vector<int>` 一样了。`end` 迭代器的类型没有变化，但不需要构造参数：

```
int main()
{
    istream_iterator<int> it_cin {cin};
    istream_iterator<int> end_cin;
```

4. 接下来，我们实例化一个 `std::deque<int>`，并且将标准输入中的所有数字拷贝到队列中。队列本身不是一个迭代器，所以我们使用 `std::back_inserter` 辅助函数将队列包装入 `std::back_insert_iterator` 中。这样指定的迭代器就能执行 `v.pack_back(item)`，将标准输入中的每个元素放入容器中。这样就能让队列自动增长。

```
deque<int> v;
copy(it_cin, end_cin, back_inserter(v));
```

5. 接下来，我们使用 `std::istringstream` 将元素拷贝到队列中部。先使用字符串，来定义一个字符流的实例：

```
istringstream sstr {"123 456 789"};
```

6. 我们需要选择列表的插入点。这个点必须在中间，我们使用队列的起始指针，然后使用 `std::next` 函数将其指向中间位置。函数第二个参数的意思就是让指针前进多少，这里选择 `v.size() / 2` 步，也就是队列的正中间位置(这里我们将 `v.size()` 强转为 `int` 类型，因为 `std::next` 第二个参数类型为 `difference_type`，是和第一个迭代器参数间的距离。因此，该类型是个有符号类型。根据编译选项，如果我们不进行显式强制转化，编译器可能会报出警告)。

```
auto deque_middle (next(begin(v),
    static_cast<int>(v.size()) / 2));
```

7. 现在，我们可以从输入流中一步步的拷贝整数到队列当中。另外，流的 `end` 包装迭代器为空的 `std::istream_iterator<int>`。这个队列已经被包装到一个插入包装器中，也就是成为 `std::insert_iterator` 的一个实例，其指向队列中间位置的迭代器，我们用 `deque_middle` 表示：

```
copy(istream_iterator<int>(sstr), {}, inserter(v,
    deque_middle));
```

8. 现在，让我们使用 `std::front_insert_iterator` 插入一些元素到队列中部：

```
initializer_list<int> il2 {-1, -2, -3};
copy(begin(il2), end(il2), front_inserter(v));
```

9. 最后一步将队列中的全部内容打印出来。`std::ostream_iterator` 作为输出迭代器，在我们的例子中其就是从 `std::cout` 拷贝打印出的信息，并将各个元素使用逗号隔开：

```
copy(begin(v), end(v), ostream_iterator<int>
{cout, ", "});
cout << '\n';
}
```

10. 编译并运行，即有如下的输出。你能找到那些数字是由哪行的代码插入的吗？

```
$ echo "1 2 3 4 5" | ./main
-3, -2, -1, 1, 2, 123, 456, 789, 3, 4, 5,
```

How it works...

本节我们使用了很多不同类型的迭代适配器。他们有一点是共同的，会将一个对象包装成迭代器。

std::back_insert_iterator

`back_insert_iterator` 可以包装 `std::vector`、`std::deque`、`std::list` 等容器。其会调用容器的 `push_back` 方法在容器最后插入相应的元素。如果容器实例不够长，那么容器的容量会自动增长。

std::front_insert_iterator

`front_insert_iterator` 和 `back_insert_iterator` 一样，不过 `front_insert_iterator` 调用的是容器的 `push_front` 函数，也就是在所有元素前插入元素。这里需要注意的是，当对类似于 `std::vector` 的容器进行插入时，其已经存在的所有元素都要后移，从而空出位置来放插入元素，这会对性能造成一定程度的影响。

std::insert_iterator

这个适配器与其他插入适配器类似，不过能在容器的中间位置插入新元素。使用 `std::inserter` 包装辅助函数需要两个参数。第一个参数是容器的实例，第二个参数是迭代器指向的位置，就是新元素插入的位置。

std::istream_iterator

`istream_iterator` 是另一种十分方便的适配器。其能对任何 `std::istream` 使用(文件流或标准输入流)，并且可以根据实例的具体特化类型，对流进行分析。本节中，我们使用了 `std::istream_iterator<int>(std::cin)`，其会将整数从标准输入中拉出来。

通常，对于流来说，其长度我们是不知道的。这就存在一个问题，也就是 `end` 迭代器指向的位置在哪里？对于流迭代器来说，它就知道相应的 `end` 迭代器的位置。这样就使得迭代器的比较更加高效，不需要通过遍历来完成。这样就是为什么 `end` 流迭代器不需要传入任何参数的原因。

std::ostream_iterator

`ostream_iterator` 和 `istream_iterator` 类似，不过是来进行输出的流迭代器。与 `istream_iterator` 不同在于，构造时需要传入两个参数，且第二个参数必须要是一个字符串，这个字符串将会在各个元素之后，推入输出流中。这样我们就能很容易的在元素中间插入逗号或者换行的符号，以便用户进行观察。

使用迭代器实现算法

迭代器通常根据指向位置的移动，来遍历容器中的元素，但不需要迭代对应的数据类型。迭代器也会被用来实现算法，其可以通过 `++it` 指向下一个元素，并且通过 `*it` 解引用得到对应的值。

本节中，我们将用迭代器来实现斐波那契函数。斐波那契函数会有类似如下的迭代： $F(n) = F(n - 1) + F(n - 2)$ 。数列的初始值 $F(0) = 0$ 和 $F(1) = 1$ 。这样下列序列就可以进行计算：

- $F(0) = 0$
- $F(1) = 1$
- $F(2) = F(1) + F(0) = 1$
- $F(3) = F(2) + F(1) = 2$
- $F(4) = F(3) + F(2) = 3$
- $F(5) = F(4) + F(3) = 5$
- $F(6) = F(5) + F(4) = 8$
- ...

我们要实现一个函数，可以输出斐波那契第n个数的值。通常我们都会使用函数迭代，或者是循环来实现这个函数。这样的话，我们只能一个个的将相应的值算出来，然后才能计算出下一个值。这里我们有两个选择——递归调用斐波那契函数计算整个数列，这样很浪费计算时间，或者将最后两个斐波那契数作为临时变量，并用它们来计算下一个数。第二种方法我们需要重新实现斐波那契算法循环。这样我们就可以将斐波那契数列计算的代码和我们实际的代码放在一起：

```
size_ta{0};  
size_tb{1};  
for(size_ti{0}; i < N; ++i) {  
    const size_t old_b{b};  
    b += a;  
    a = old_b;  
    // do something with b, which is the current  
    fibonacci number  
}
```

使用迭代器实现斐波那契数列是一件很有意思的事情。如何将循环中的迭代，使用迭代器的前向自加操作来代替呢？其实很简单，让我们来看一下。

How to do it...

本节中，我们主要关注如何用一个迭代器实现生成斐波那契数列。

1. 为了打印斐波那契数列在终端，我们需要包含标准输入输出流头文件。

```
#include <iostream>
```

2. 我们调用斐波那契迭代器——`fibit`。其会指向一个值 `i`，其保存的值为斐波那契数列对应的位置，`a` 和 `b` 保存斐波那契数列中最后两个值。实例化迭代器时，需要将斐波那契迭代器初始化为 $F(0)$ 的值：

```
class fibit
{
    size_t i {0};
    size_t a {0};
    size_t b {1};
```

3. 下一步，定义标准构造函数和另一个构造函数用来初始化迭代器。

```
public:
    fibit() = default;
    explicit fibit(size_t i_)
        : i{i_}
    {}
```

4. 当我们对迭代器解引用时，迭代器将返回对应位置的数值。

```
size_t operator*() const { return b; }
```

5. 当移动迭代器 `++` 时，其会移动到下一个斐波那契数上。这里的实现与基于循环的实现几乎是一样的。

```
fibit& operator++() {
    const size_t old_b {b};
    b += a;
    a = old_b;
    ++i;
    return *this;
}
```

6. 当使用循环时，增加后的迭代器将会和 `end` 迭代器进行比较，所以这里需要为迭代器实现不等于 `!=` 操作符。我们只比较当且迭代器所对应的步数，这比循环1000000次再结束迭代器简单许多，这样我们就不需要计算太多的斐波那契数：

```
bool operator!=(const fibit &o) const { return i
!= o.i; }
```

7. 为了能让斐波那契迭代器适应 `for` 循环的范围写法，我们需要实现一个范围类。我们称这个类为 `fib_range`，其构造函数只需要一个参数，这个参数能够告诉我们我们想要遍历的范围：

```

class fib_range
{
    size_t end_n;
public:
    fib_range(size_t end_n_)
        : end_n{end_n_}
    {}

```

8. `begin` 和 `end` 函数将会返回对应位置上的迭代器，也就是 $F(0)$ 和 $F(end_n)$ 对应的迭代器。

```

fibit begin() const { return fibit{}; }
fibit end() const { return fibit{end_n}; }
};

```

9. 好了，其他与迭代器相关的代码我们就不管了。因为我们辅助类就能很好的帮助我们将这些细节的东西隐藏掉！让我们打印10个斐波那契数字：

```

int main()
{
    for (size_t i : fib_range(10)) {
        std::cout << i << ", ";
    }
    std::cout << '\n';
}

```

10. 编译运行后，我们会在终端上看到如下的打印：

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55,
```

There's more...

为了兼容STL中的迭代器，这里实现的迭代器必须支持 `std::iterator_traits` 类。想要知道怎么做，要参考一下3.2节(让自己的迭代器与STL的迭代器兼容)，其对如何兼容进行了明确地说明。

Note:

试着从迭代器的角度思考，这样的代码在很多情况下就显得十分优雅。不用担心性能，编译器会根据模板对迭代器相关的代码进行优化。

为了保证例子的简洁性，我们并没有对其做任何事情，不过要是作为斐波那契迭代器的发布库的话，其可用性还是比较差的——`fibit` 传入一个参数的构造函数，可以直接使用 `end` 迭代器替换，因为 `fibit` 并没有包含任何一个合法的斐波那契值，这里的库并不强制使用这种方式。

还有些方面需要进行修复：

- 将 `fibit(size_t i_)` 声明为私有构造函数，并在 `fibit` 类中将 `fib_range` 类声明为一个友元类。这样用户就只能使用正确的方式进行迭代了。
- 可以使用迭代器哨兵，避免用户引用 `end` 迭代器。可以参考一下3.6节(使用哨兵终止迭代)中内容，以获得更多信息。

使用反向迭代适配器进行迭代

有时我们需要反向迭代一个范围内的内容。基于范围的for循环中，STL迭代通常都使用前向累加的方式进行迭代，那么当需要反向时，就需要对其进行递减。当然，这里可以将迭代器进行包装，将调用累加操作改为递减的操作。听起来要写好多冗余的代码，来对反向迭代进行支持。

STL中提供了反向迭代适配器，其能帮助我们对迭代器进行包装。

How to do it...

本节中，我们将用另一种方式使用反向迭代器，只为了展示如何使用它们：

1. 包含必要的头文件：

```
#include <iostream>
#include <list>
#include <iterator>
```

2. 声明所使用的命名空间：

```
using namespace std;
```

3. 为了有东西可以迭代，我们实例化一个整数列表：

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

4. 现在，让我们来反向打印这些数字。为了完成反向打印，我们调用 `std::list` 的成员函数 `rbegin` 和 `rend` 获得反向迭代器，并且将数字推入输出流 `ostream_iterator` 适配器中：

```
copy(l.rbegin(), l.rend(), ostream_iterator<int>
{cout, ", "});
cout << '\n';
```

5. 如果容器不提供 `rbegin` 和 `rend` 函数的话，就需要使用双向迭代器来帮忙了，这里可以使用工厂函数 `std::make_reverse_iterator` 创建双向迭代器。其能接受普通迭代器，然后将其转换为反向迭代器：

```

        copy(make_reverse_iterator(end(l)),
              make_reverse_iterator(begin(l)),
              ostream_iterator<int>{cout, ", "});
    cout << '\n';
}

```

6. 编译并运行该程序，就能得到如下的输出：

```

5, 4, 3, 2, 1,
5, 4, 3, 2, 1,

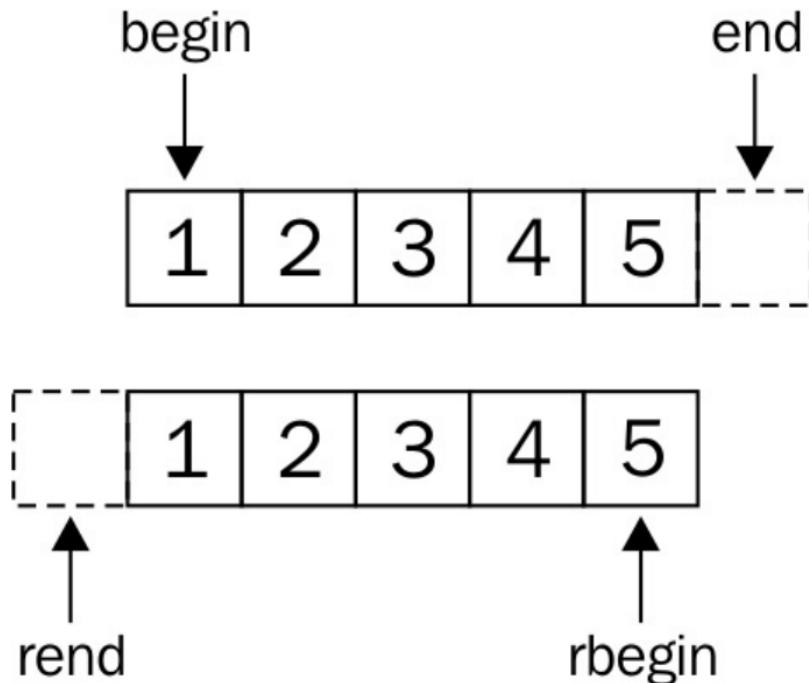
```

How it works...

为了将一个普通迭代器转换为一个反向迭代器，容器至少要支持双向迭代。这就需要双向类别或更高级的迭代器才能满足条件。

反向迭代器是普通迭代器的一种，并且接口和普通迭代器都一样，不过其累加操作会被当做递减操作来进行。

下面就来聊一下 `begin` 和 `end` 迭代器的位置。先来看一下图，迭代器区域里面是一串标准的数字序列。



如果序列是从1到5，`begin` 迭代器将指向元素1所在的位置，并且 `end` 迭代器将指向元素5后面的位置。当定义了反向迭代器，`rbegin` 迭代器就指向了元素5，并且 `rend` 迭代器指向元素1之前的位置。可以将书反过来，可以发现这两个中方式是镜像的。

当我们想让我们自定义的容器类支持反向迭代，我们不用将所有细节一一实现；我们只需使用 `std::make_reverse_iterator` 工厂函数，将普通的迭代器包装成反向迭代器即可，背后的操作STL会帮我们完成。

使用哨兵终止迭代

对于STL算法和基于范围的for循环来说，都会假设迭代的位置是提前知道的。在有些情况下，并不是这样，我们在迭代器到达末尾之前，我们是很难确定结束的位置在哪里。

这里使用C风格的字符串来举例，我们在编译时无法知道字符串的长度，只能在运行时使用某种方法进行判断。字符串遍历的代码如下所示：

```
for (const char *c_pointer = some_c_string; *c_pointer != '\0'; ++c_pointer) {
    const char c = *c_pointer;
    // do something with c
}
```

对于基于范围的for循环来说，我们可以将这段字符串包装进一个 `std::string` 实例中，`std::string` 提供 `begin()` 和 `end()` 函数：

```
for (char c : std::string(some_c_string)) { /* do
something with c */ }
```

不过，`std::string` 在构造的时候也需要对整个字符串进行遍历。`C++17`中加入了 `std::string_view`，但在构造的时候也会对字符串进行一次遍历。对于比较短的字符串来说这是没有必要的，不过对于其他类型来说就很有必要。`std::istream_iterator` 可以用来从 `std::cin` 捕获输入，当用户持续输入的时候，其 `end` 迭代器并不能指向输入字符串真实的末尾。

`C++17`添加了一项新的特性，其不需要 `begin` 迭代器和 `end` 迭代器是同一类型的迭代器。本节我们看看，这种小修改的大用途。

How to do it...

本节，我们将在范围类中构造一个迭代器，其就不需要知道字符串的长度，也就不用提前找到字符串结束的位置。

1. 包含必要的头文件。

```
#include <iostream>
```

2. 迭代器哨兵是本节的核心内容。奇怪的是，它的定义完全是空的。

```
class cstring_iterator_sentinel {};
```

3. 我们先来实现迭代器。其包含一个字符串指针，指针指向的容器就是我们要迭代的：

```
class cstring_iterator {
    const char *s {nullptr};
```

4. 构造函数只是初始化内部字符串指针，对应的字符串是外部输入。显式声明构造函数是为了避免字符串隐式转换为字符串迭代器：

```
public:
    explicit cstring_iterator(const char *str)
        : s{str}
    {}
```

5. 当对迭代器进行解引用，其就会返回对应位置上的字符：

```
char operator*() const { return *s; }
```

6. 累加迭代器只增加迭代器指向字符串的位置：

```
cstring_iterator& operator++() {
    ++s;
    return *this;
}
```

7. 这一步是最有趣的。我们为了比较，实现了 != 操作符。不过，这次我们不去实现迭代器的比较操作，这次迭代器要和哨兵进行比较。当我们比较两个迭代器时，在当他们指向的位置相同时，我们可以认为对应范围已经完成遍历。通过和空哨兵对象比较，当迭代器指向的字符为 \0 字符时，我们可以认为到达了字符串的末尾。

```
bool operator!=(const cstring_iterator sentinel)
const {
    return s != nullptr && *s != '\0';
}
```

8. 为了使用基于范围的 for 循环，我们需要一个范围类，用来指定 begin 和 end 迭代器：

```
class cstring_range {
    const char *s {nullptr};
```

9. 实例化时用户只需要提供需要迭代的字符串：

```
public:  
    cstring_range(const char *str)  
        : s{str}  
    {}
```

10. `begin()` 函数将返回一个 `cstring_iterator` 迭代器，其指向了字符串的起始位置。`end()` 函数会返回一个哨兵类型。需要注意的是，如果不使用哨兵类型，这里将返回一个迭代器，这个迭代器要指向字符串的末尾，但是我们无法预知字符串的末尾在哪里。

```
cstring_iterator begin() const {  
    return cstring_iterator{s};  
}  
cstring_iterator_sentinel end() const {  
    return {};  
}  
};
```

11. 类型定义完，我们就来使用它们。例子中字符串是用户输入，我们无法预知其长度。为了让使用者给我们一些输入，我们的例子会判断是否有输入参数。

```
int main(int argc, char *argv[])  
{  
    if (argc < 2) {  
        std::cout << "Please provide one  
parameter.\n";  
        return 1;  
    }
```

12. 当程序运行起来时，我们就知道 `argv[1]` 中包含的是使用者的字符串。

```
for (char c : cstring_range(argv[1])) {  
    std::cout << c;  
}  
std::cout << '\n';  
}
```

13. 编译运行程序，就能得到如下的输出：

```
$ ./main "abcdef"  
abcdef
```

循环会将所有的字符打印出来。这是一个很小的例子，只是为了展示如何使用哨兵确定迭代的范围。当在无法获得 `end` 迭代器的位置时，这是一种很有用的方法。当能够获得 `end` 迭代器时，就不需要使用哨兵了。

使用检查过的迭代器自动化检查迭代器代码

迭代器很有用，能提供一般化的接口供用户使用。不过，迭代器经常被当做指针误用。当指针指向一个非法的内存位置时，不能进行解引用。这对迭代器也适用，不过有大量的条件来界定迭代器指向的位置是否合法。这些可以通过看一下[STL文档](#)就能了解到，但是还会写出很容易出现bug的代码。

最好的情况是，这些问题没有在客户的机器上出现，而是开发者测试这些程序时就能暴露出来。不过，通常即使是解引用了悬垂指针和错误的迭代器，代码也不会报错。这种情况是最糟的，因为这种未定义行为的代码，没法确定会发生什么。

幸运的是，有工具可以帮助我们。[GNU STL](#)有调试模式可选，[GNU C++](#)编译器和[LLVM clang C++](#)编译器都提供这样的库，其会为我们生成具有调试信息的二进制程序，可以让错误更容易暴露出来。这种库非常容易使用，并且特别有用，我们将在本节展示。[Microsoft Visual C++](#)标准库还提供了更多的检查项。

How to do it...

本节我们将使用迭代器故意访问一个非法位置：

1. 包含头文件。

```
#include <iostream>
#include <vector>
```

2. 首先实例化一个整型类 `vector`，并且让指针指向值1。我们使用 `shrink_to_fit()` 将 `vector` 的容积设置为3，多分配的内存是不必要的，小一点的存储空间会让迭代速度更快：

```
int main()
{
    std::vector<int> v {1, 2, 3};
    v.shrink_to_fit();
    const auto it (std::begin(v));
```

3. 然后解引用迭代器，打印相应的内容：

```
std::cout << *it << '\n';
```

4. 接下来，让我们向 `vector` 中增加一个新数。这样 `vector` 的长度就不够再放下另外一个数，这里 `vector` 会自动增加其长度。通过分配一个新的更大的内存块来实现长度的增加，会将所有现存的项移到新的块，然后删除旧的内存块。

```
v.push_back(123);
```

5. 现在，让我们再次通过迭代器从1开始打印 `vector`。这就坏了。为什么呢？因为在 `vector` 自增的过程中，会分配新的内存，删除旧的内存，但是迭代器却不知道这个改变。这就意味着，迭代器将会指向旧地址，并且我们不知道这样做会怎样。

```
    std::cout << *it << '\n'; // bad bad bad!
}
```

6. 编译变这个程序并运行，我们不会看到任何错误，不过迭代器解引用所打印出来的数字看上去像是随机数。看上去没有问题，反而最有问题。如果不指出来，可能没人会发现问题。

```
$ g++ -std=c++17 main.cpp -o main
$ ./main
1
0
```

7. 这时调试工具就派上了用场。**GUN STL**支持一种预处理宏 `_GLIBCXX_DEBUG`，其会激活**STL**中对健壮性检查的代码。这会让程序变慢，不过更容易找到**Bug**。我们可以通过 `-D_GLIBCXX_DEBUG` 编译选项来启用这些代码，或者在代码的最开始加上这个宏。如你所见，其输出相关的错误信息，并关闭了应用的进程。

Microsoft Visual C++ 编译器可以通过 `/D_ITERATOR_DEBUG_LEVEL=1` 启用检查。

```
$ g++ -std=c++17 main.cpp -D_GLIBCXX_DEBUG -o main
$ ./main
1
/opt/gcc_latest/include/c++/7.0.0/debug/safe_iterator.h:270:
Error: attempt to dereference a singular iterator.

Objects involved in the operation:
    iterator "this" @ 0x0x7ffc06323730 {
        type = __gnu_debug::__Safe_Iterator<__gnu_cxx::__normal_iterator<int*, std::__cxx1998::vector<int, std::allocator<int> >, std::__debug::vector<int, std::allocator<int> >> (mutable iterator);
        state = singular;
        references sequence with type 'std::__debug::vector<int, std::allocator<int> >' @ 0x0x7ffc06323760
    }
Aborted (core dumped)
```

8. **LLVM/clang**实现的**STL**也有调试标识，其目的是为了调试**STL**代码，而非用户的代码。对于用户的代码的调试，我们会使用不同的选项来调试。向**clang**编译器传入 `-fsanitize=address -fsanitize=undefined`，可以看看会发生什么：

```
$ clang++ -std=c++1z -fsanitize=address -fsanitize=undefined main.cpp -o main
$ ./main
=====
==20639==ERROR: AddressSanitizer: heap-use-after-free on address 0x60200000eff0 at pc 0x0000004eb519 bp 0x7fc0fe5d730 sp 0x7fc0fe5d728
READ of size 4 at 0x60200000eff0 thread T0
#0 0x4eb518 (/home/tfc/src/stl_cookbook/03/checked_iterator/main+0x4eb518)
#1 0x7f1c89f893f0 (/lib/x86_64-linux-gnu/libc.so.6+0x203f0)
#2 0x4187f9 (/home/tfc/src/stl_cookbook/03/checked_iterator/main+0x4187f9)

0x60200000eff0 is located 0 bytes inside of 12-byte region [0x60200000eff0,0x60200000effc)
freed by thread T0 here:
#0 0x4e83c0 (/home/tfc/src/stl_cookbook/03/checked_iterator/main+0x4e83c0)
#1 0x4ee64b (/home/tfc/src/stl_cookbook/03/checked_iterator/main+0x4ee64b)
#2 0x4ee619 (/home/tfc/src/stl_cookbook/03/checked_iterator/main+0x4ee619)
#3 0x4eedae (/home/tfc/src/stl_cookbook/03/checked_iterator/main+0x4eedae)
#4 0x4ef09c7 (/home/tfc/src/stl_cookbook/03/checked_iterator/main+0x4ef09c7)
#5 0x4ef2a7 (/home/tfc/src/stl_cookbook/03/checked_iterator/main+0x4ef2a7)
#6 0x4ec1af (/home/tfc/src/stl_cookbook/03/checked_iterator/main+0x4ec1af)
#7 0x4eb364 (/home/tfc/src/stl_cookbook/03/checked_iterator/main+0x4eb364)
#8 0x7f1c89f893f0 (/lib/x86_64-linux-gnu/libc.so.6+0x203f0)

previously allocated by thread T0 here:
#0 0x4e7dc0 (/home/tfc/src/stl_cookbook/03/checked_iterator/main+0x4e7dc0)
#1 0x4edfb0 (/home/tfc/src/stl_cookbook/03/checked_iterator/main+0x4edfb0)
```

WOW! **clang**编译器对于运行错误的描述非常详细。由于信息非常的多，这里只截取其中一部分。当然，这个选项并不是**clang**独有的特性，对于**GCC**同样适用。

Note:

一些运行时的问题是因为一些库的丢失，编译器不会将libasan和libubsan(AddressSanitizer内存检测工具)自动添加到程序中，需要通过包管理器或类似的工具进行安装。

How it works...

如我们之前所见，我们不需要通过修改任何代码，只需要通过为编译器添加一些编译器特性就能容易的找到代码中的Bug。

这些特性由调试器实现。一个调试器通常由一个编译器模块和一个运行时库组成。当调试器被激活时，编译器将会添加额外的信息到我们的代码中，然后形成二进制可执行文件。在运行时，调试器库由二进制文件自己去链接，例如：对应库实现会代替 `malloc` 和 `free` 函数，来分析程序到底想要多少内存。

调试器可以检测不同类型的Bug。这里只列举出一些常用的类型：

- **越界访问**: 当我们访问类似数组和 `vector` 类型的数据结构时，判别我们访问的位置是否在合法范围内。
- **释放后使用**: 当我们释放了堆上分配的指针后，再使用这个指针，则会发出这个Bug。
- **整数溢出**: 不同的机器上整数表达的范围可能是不同的，所以就会出现一些值使用整型无法进行表示。对于有符号整型，算法通常会发出一个未定义的行为。
- **指针对齐**: 一些架构中，需要指针以某种形式进行对齐，否则无法访问对应的地址。

当然，我们还能检测到更多类型的Bug。

不过，激活所有的调试器不太可行，因为这样会导致程序运行的非常缓慢。不过，在单元测试和集成测试中，激活调试器是一个很好的方式。

There's more...

对于不同类型的Bug，调试器的种类也是多种多样，并且还有很多调试器还在开发中。我们可以上网了解更多的信息，以便我们自己去调试程序。GCC和LLVM网站首页就列举了很多调试器，可以从在线文档中了解其调试能力：

- <https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html>
- <http://clang.llvm.org/docs/index.html> 可在目录中寻找调试器

使用调试器对程序进行整体测试是每个开发者都应该具有的意识。不过，在大多数公司中，开发者并没有这样的意识，即便是我们知道所有恶意软件和计算机病毒最重要的入口就是程序的Bug。

当时是一个开发新手时，看一下你的团队中是否有使用调试器的可能。如果没有，那你上班的第一天就有机会修复那些重大的Bug，并发现隐藏的Bug。

构建zip迭代适配器

不同的编程语言引领了不同的编程方式。不同语言有各自的受众群体，因为表达方式的不同，所以对于优雅地定义也不同。

纯函数式编程算是编程风格中一种比较特别的方式。其与C和C++命令方式编程的风格大相径庭。虽然风格迥异，但是纯函数式编程却能在大多数情况下产生非常优雅地代码。

这里用向量点乘为例，使用函数式方法优雅地实现这个功能。给定两个向量，然后让对应位置上的两个数字相乘，然后将所有数字加在一起。也就是 $(a, b, c) * (d, e, f)$ 的结果为 $(a * d + b * e + c * f)$ 。我们在C和C++也能完成这样的操作。代码可能类似如下的方式：

```
std::vector<double> a {1.0, 2.0, 3.0};
std::vector<double> b {4.0, 5.0, 6.0};
double sum {0};
for (size_t i {0}; i < a.size(); ++i) {
    sum += a[i] * b[i];
}
// sum = 32.0
```

如何使用其他语言让这段代码更优雅呢？

Haskell是一种纯函数式语言，其使用一行代码就能计算两个向量的点积：

```
[Prelude> a = [1.0, 2.0, 3.0]
[Prelude> b = [4.0, 5.0, 6.0]
[Prelude> sum $ zipWith (*) a b
32.0
```

Python虽然不是纯函数式编程语言，但是也会提供类似功能：

```
[>>> a = [1.0, 2.0, 3.0]
[>>> b = [4.0, 5.0, 6.0]
[>>> sum([ p[0] * p[1] for p in zip(a, b) ])
32.0
```

STL提供了相应的函数实现 `std::inner_product`，也能在一行之内完成向量点积。不过，其他语言中在没有相应的库对某种操作进行支持的情况下，也能做到在一行之内完成。

不需要对两种语言的语法进行详细了解的情况下，大家都应该能看的出，两个例子中最重要的就是zip函数。这个函数做了什么？假设我们有两个向量a和b，变换后将两个向量混合在一起。例如：`[a1, a2, a3]` 和 `[b1, b2, b3]`，使用zip函数处理的结果为 `[(a1, b1), (a2, b2), (a3, b3)]`。让我们仔细观察这个例子，就是将两个向量连接在了一起。

现在，关联的数字可以直接进行加法，然后累加在一起。在Haskell和Python的例子中我们看到，这个过程不需要任何循环或索引变量。[译者注：Python中是有循环的.....]

这里没法让C++代码如同Haskell或Python那样优雅和通用，不过本节的内容就是为了实现一个类似的迭代器——zip迭代器——然后使用这个迭代器。向量点积有特定的库支持，至于是哪些库，以及这些库如何使用，并不在本书的描述范围内。不过，本节的内容将尝试展示一种基于迭代器的方式，来帮助你使用通用的模块另外完成编程。

How to do it...

本节中，我们会实现一个类似Haskell和Python中的zip函数。为了不对迭代器的机制产生影响，`vector` 中的变量这里写死为`double`：

1. 包含头文件

```
#include <iostream>
#include <vector>
#include <numeric>
```

2. 定义`zip_iterator`类。同时也要实现一个范围类`zip_iterator`，这样我们在每次迭代时就能获得两个值。这也意味着我们同时遍历两个迭代器：

```
class zip_iterator {
```

3. zip迭代器的容器中需要保存两个迭代器：

```
using it_type = std::vector<double>::iterator;

it_type it1;
it_type it2;
```

4. 构造函数会将传入的两个容器的迭代器进行保存，以便进行迭代：

```
public:
    zip_iterator(it_type iterator1, it_type
    iterator2)
        : it1{iterator1}, it2{iterator2}
    {}
```

5. 增加zip迭代器就意味着增加两个成员迭代器：

```
zip_iterator& operator++() {
    ++it1;
    ++it2;
    return *this;
}
```

6. 如果zip中的两个迭代器来自不同的容器，那么他们一定不相等。通常，这里会用逻辑或(||)替换逻辑与(&&)，但是这里我们需要考虑两个容器长度不一样的情况。这样的话，我们需要在比较的时候同时匹配两个容器。这样，我们就能遍历完其中一个容器时，及时停下循环：

```
bool operator!=(const zip_iterator& o) const {
    return it1 != o.it1 && it2 != o.it2;
}
```

7. 逻辑等操作符可以使用逻辑不等的操作符的实现，是需要将结果取反即可：

```
bool operator==(const zip_iterator& o) const {
    return !operator!=(o);
}
```

8. 解引用操作符用来访问两个迭代器指向的值：

```
std::pair<double, double> operator*() const {
    return {*it1, *it2};
};
```

9. 迭代器算是实现了。我们需要让迭代器兼容STL算法，所以我们对标准模板进行了特化。这里讲迭代器定义为一个前向迭代器，并且解引用后返回的是一对double值。虽然，本节我们没有使用difference_type，但是对于不同编译器实现的STL可能就需要这个类型：

```
namespace std {
template <>
struct iterator_traits<zip_iterator> {
    using iterator_category =
        std::forward_iterator_tag;
    using value_type = std::pair<double, double>;
    using difference_type = long int;
};
```

10. 现在来定义范围类，其begin和end函数返回zip迭代器：

```
class zipper {
    using vec_type = std::vector<double>;
    vec_type &vec1;
    vec_type &vec2;
```

11. 这里需要从zip迭代器中解引用两个容器中的值：

```
public:  
    zipper(vec_type &va, vec_type &vb)  
        : vec1{va}, vec2{vb}  
    {}
```

12. `begin` 和 `end` 函数将返回指向两容器开始的位置和结束位置的迭代器对:

```
zip_iterator begin() const {  
    return {std::begin(vec1), std::begin(vec2)};  
}  
zip_iterator end() const {  
    return {std::end(vec1), std::end(vec2)};  
}  
};
```

13. 如Haskell和Python的例子一样，我们定义了两个 `double` 为内置类型的 `vector`。这里我们也声明了所使用的命名空间。

```
int main()  
{  
    using namespace std;  
    vector<double> a {1.0, 2.0, 3.0};  
    vector<double> b {4.0, 5.0, 6.0};
```

14. 可以直接使用两个 `vector` 对 `zipper` 类进行构造:

```
zipper zipped {a, b};
```

15. 我们将使用 `std::accumulate` 将所有值累加在一起。这里我们不能直接对 `std::pair<double, double>` 实例的结果进行累加，因为这里没有定义 `sum` 变量。因此，我们需要定义一个辅助Lambda函数来对这个组对进行操作，将两个数相乘，然后进行累加。Lambda函数指针可以作为 `std::accumulate` 的一个参数传入:

```
const auto add_product ([](double sum, const auto  
&p) {  
    return sum + p.first * p.second;  
});
```

16. 现在，让我们来调用 `std::accumulate` 将所有点积的值累加起来:

```
const auto dot_product (accumulate(  
    begin(zipped), end(zipped), 0.0,  
    add_product));
```

17. 最后，让我们来打印结果:

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

18. 编译运行后，得到正确的结果：

```
32
```

There's more...

OK，这里使用了语法糖来完成了大量的工作，不过这和Haskell的例子也相差很远，还不够优雅。我们的设计中有个很大的缺陷，那就是只能处理 `double` 类型的数据。通过模板代码和特化类，`zipper` 类会变得更通用。这样，我们就能将 `list` 和 `vector` 或 `deque` 和 `map` 这样不相关的容器合并起来。

为了让设计的类更加通用，其中设计的过程是不容忽视的。幸运的是，这样的库已经存在。Boost作为STL库的先锋，已经支持了 `zip_iterator`。这个迭代器非常简单、通用。

顺便提一下，如果你想看到了使用C++实现的更优雅的点积，并且不关心 `zip` 迭代器相关的内容，那么你可以了解一下 `std::valarray`。例子如下，自己看下：

```
#include <iostream>
#include <valarray>
int main()
{
    std::valarray<double> a {1.0, 2.0, 3.0};
    std::valarray<double> b {4.0, 5.0, 6.0};
    std::cout << (a * b).sum() << '\n';
}
```

范围库

这是C++中非常有趣的一个库，其支持 `zipper` 和所有迭代适配器、滤波器等等。其受到Boost范围库的启发，并且某段时间内里，很有可能进入C++17标准。不幸的是，我们只能在下个标准中期待这个特性的加入。这种性能可以带来更多的便利，能让我们想表达的东西通过C++快速实现，并可以通过将通用和简单的模块进行组合，来表现比较复杂的表达式。

在文档中对其描述中，有个非常简单的例子：

1. 计算从1到10数值的平方：

```
const int sum = accumulate(view::ints(1)
                           | view::transform([](int i)
                           {return i*i;})
                           | view::take(10), 0);
```

2. 从数值 `vector` 中过滤出非偶数数字，并且将剩下的数字转换成字符串：

```
std::vector<int> v {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

auto rng = v | view::remove_if([](int i){return i % 2
== 1; })
           | view::transform([](int i){return
std::to_string(i); });
// rng == {"2"s, "4"s, "6"s, "8"s, "10"s};
```

如果你等不及想要了解这些有趣的特性，可以看一下范围类的文档，<https://ericniebler.github.io/range-v3>。

第4章 Lambda表达式

Lambda表达式是C++11添加的非常重要的一个特性。C++14和C++17对Lambda进行补充，使得Lambda表达式如虎添翼。那就先了解一下，什么是Lambda表达式呢？

Lambda表达式或者Lambda函数为闭包结构。闭包是描述未命名对象的通用术语，也可以称为匿名函数。为了在C++中加入这个特性，就需要相应用对象实现()括号操作符。C++11之前想要实现类似具有Lambda的对象，代码如下所示：

```
#include <iostream>
#include <string>
int main() {
    struct name_greeter {
        std::string name;

        void operator()() {
            std::cout << "Hello, " << name << '\n';
        }
    };

    name_greeter greet_john_doe {"John Doe"};
    greet_john_doe();
}
```

构造`name_greeter`对象需要传入一个字符串。这里需要注意的是，这个结构类型，Lambda可以使用一个没有名字的实例来表示。对于闭包结构来说，我们称之为捕获一个字符串。其就像我们在构造这个例子中的实例时传入的字符串一样，不过Lambda不需要参数，就能完成打印`Hello, John Doe`。

C++11之后，使用闭包的方式来实现会更加简单：

```
#include <iostream>
int main() {
    auto greet_john_doe [] {
        std::cout << "Hello, John Doe\n";
    };
    greet_john_doe();
}
```

这样就行了！不再需要`name_greeter`结构体，直接使用Lambda表达式替代。这看起来像魔术一样，本章的第一节中会对细节进行详细的描述。

Lambda表达式对于完成通用和简介类代码是非常有帮助的。其能对通用的数据结构进行处理，这样就不惧用户指定的特殊类型。闭包结构也会被用来将运行在线程上的数据进行打包。C++11标准推出后，越来越多的库支持了Lambda表达式，因

为这对于C++来说已经是很自然的事情了。另一种使用方式是用于元编程，因为Lambda在编译时是可以进行预估的。不过，我们不会往元编程的方向去讲述，元编程的内容可能会撑爆这本书。

本章我们着重于函数式编程，对于那些对函数式编程不了解的开发者或初学者来说，这看起来非常的神奇。如果你在代码中看到**Lambda**表达式横飞，请先别沮丧。在这个函数式编程越来越流行的年代，需要拓展对于现代C++的了解。如果你看到的代码有点复杂，建议你多花点时间去分析它们。当你驯服了**Lambda**表达式，你就能驾驭它驰骋疆场，不再会为之困惑。

使用Lambda表达式定义函数

我们可以使用**Lambda**表达式来包装代码，为了在之后对其进行调用。我们可以像调用函数那样，给**Lambda**表达式传入不同的参数，从而得到不同的结果，这样我们就不需要在类中实现这个函数了。

C++11标准正式将**Lambda**语法加入C++，之后的C++14和C++17标准中对**Lambda**语法进行了升级。本节我们将看到如何使用**Lambda**表达式，以及其给我们带来的改变。

How to do it...

现在我们就来使用**Lambda**表达式完成一个程序，在实践中体验**Lambda**表达式：

1. **Lambda**表达式不需要任何库，不过我们需要将一些字符串打印在屏幕上，所以需要包含必要的头文件：

```
#include <iostream>
#include <string>
```

2. 这次我们所有内容都会在主函数中完成。我们定义了两个没有参数的函数对象，并且返回整型常量1和2。需要注意的是，返回部分在大括号对 {} 中，就像普通的函数那样，而小括号 () 表示没有参数传入，当然也可以像普通函数那样定义函数签名，对于第二个**Lambda**表达式没有添加小括号对。不过两个表达式都有中括号对 []：

```
int main()
{
    auto just_one ( [](){ return 1; } );
    auto just_two ( [ ] { return 2; } );
```

3. 那么现在我们就来调用这两个函数，就像调用普通函数那样：

```
    std::cout << just_one() << ", " << just_two() <<
    '\n';
```

4. 现在，来定义另一个函数对象，其名为**plus**，因为它要将两个参数进行加和：

```
    auto plus ( [ ](auto l, auto r) { return l + r; }
);
```

5. 这个函数对象也不难用。使用 `auto` 类型定义两个参数，只要是作为参数的实参类型支持加法操作，那么就没有任何问题：

```
    std::cout << plus(1, 2) << '\n';
    std::cout << plus(std::string{"a"}, "b") << '\n';
```

6. 当然，我们可以不使用变量的方式对Lambda表达式进行保存。我们只需要在使用到的地方对其进行定义即可：

```
std::cout
<< [](auto l, auto r){ return l + r; }(1, 2)
<< '\n';
```

7. 接下来，我们定义一个闭包，包里面装着一个计数器。当我们调用这个计数器时，其值就会自加，并且会将自加后的值返回。为了对计数变量进行初始化，我们在中括号对中对 `count` 进行了赋值。为了能让函数对获取的值进行修改，我们使用 `mutable` 关键字对函数进行修饰，否则在编译时会出问题：

```
auto counter (
    [count = 0] () mutable { return ++count; }
);
```

8. 现在让我们调用函数对象5次，并且打印其返回值，观察每次调用后计数器增加后的值：

```
for (size_t i {0}; i < 5; ++i) {
    std::cout << counter() << ", ";
}
std::cout << '\n';
```

9. 我们也可以通过捕获已经存在的变量的引用，在闭包中进行修改。这样的话，捕获到的值会自加，并且在闭包外部也能访问到这个变量。为了完成这个任务，我们在中括号对中写入 `&a`，`&` 符号就意味着捕获的是对应变量的引用，而非副本：

```
int a {0};
auto incrementer ( [&a] { ++a; } );
```

10. 如果这样能行，那我们就可以多次的调用这个函数对象，并且直接在外部对 `a` 变量的值进行观察：

```
incrementer();
incrementer();
incrementer();

std::cout
<< "Value of 'a' after 3 incrementer() calls:
"
<< a << '\n';
```

11. 最后一个例子是一个多方位展示，这个例子中一个函数对象可以接受参数，并且将其传入另一个函数对象中进行保存。在这个 `plus_ten` 函数对象中，我们会调用 `plus` 函数对象：

```
auto plus_ten ( [=] (int x) { return plus(10, x); } );
std::cout << plus_ten(5) << '\n'; }
```

12. 编译并运行代码，我们将看到如下的内容打印在屏幕上。我们也可以自己计算一下，看看打印的结果是否正确：

```
1, 2
3
ab
3
1, 2, 3, 4, 5,
Value of a after 3 incrementer() calls: 3
15
```

How it works...

上面的例子并不复杂——添加了数字，并对调用进行计数，并打印计数的结果。甚至用一个函数对象来连接字符串，并用这个函数对象对对应字符串进行计数。不过，这些实现对于对Lambda表达式不太了解的人来说，看着就很困惑了。

所以，先让我们了解一下Lambda表达式的特点：

```
[capture list] (parameters)
    mutable           (optional)
    constexpr         (optional)
    exception attr   (optional)
    -> return type   (optional)
{
    body
}
```

Lambda表达式的最短方式可以写为 `[]{}`。其没有参数，没有捕获任何东西，并且也不做实质性的执行。

那么其余的部分是什么意思呢？

捕获列表 capture list

指定我们需要捕获什么。其由很多种方式，我们展示两种比较“懒惰”的方式：

- 将Lambda表达式写成 `[=] () {...}` 时，会捕获到外部所有变量的副本。
- 将Lambda表达式写成 `[&] () {...}` 时，会捕获到外部所有变量的引用。

当然，也可以在捕获列表中单独的去写需要捕获的变量。比如 `[a, &b] () {...}`，就是捕获 `a` 的副本和 `b` 的引用，这样捕获列表就不会去捕获那些不需要捕获的变量。

本节中，我们定义了一个**Lambda**表达式：`[count=0] () {...}`，这样我们就不会捕获外部的任何变量。我们定义了一个新的 `count` 变量，其类型通过初始化的值的类型进行推断，由于初始化为0，所以其类型为 `int`。

所以，可以通过捕获列表捕获变量的副本和/或引用：

- `[a, &b] () {...}`：捕获 `a` 的副本和 `b` 的引用。
- `[&, a] () {...}`：除了捕获 `a` 为副本外，其余捕获的变量皆为引用。
- `[=, &b, i{22}, this] () {...}`：捕获 `b` 的引用，`this` 的副本，并将新变量 `i` 初始化成22，并且其余捕获的变量都为其副本。

Note:

当你需要捕获一个对象的成员变量时，不能直接去捕获成员变量。需要先去捕获对象的 `this` 指针或引用。

mutable (optional)

当函数对象需要去修改通过副本传入的变量时，表达式必须用 `mutable` 修饰。这相当于对捕获的对象使用非常量函数。

constexpr (optional)

如果我们显式的将**Lambda**表达式修饰为 `constexpr`，编译器将不会通过编译，因为其不满足 `constexpr` 函数的标准。`constexpr` 函数有很多条件，编译器会在编译时对**Lambda**表达式进行评估，看其在编译时是否为一个常量参数，这样就会让程序的二进制文件体积减少很多。

当我们不显式的将**Lambda**表达式声明为 `constexpr` 时，编译器就会自己进行判断，如果满足条件那么会将**Lambda**表达式隐式的声明为 `constexpr`。当我们需要一个**Lambda**表达式为 `constexpr` 时，我们最好显式的对**Lambda**的表达式进行声明，当编译不通过时，编译器会告诉我们哪里做错了。

exception attr (optional)

这里指定在运行错误时，是否抛出异常。

return type (optional)

当想完全控制返回类型时，我们不会让编译器来做类型推导。我们可以写成这样 `[]() -> Foo {}`，这样就告诉编译器，这个**Lambda**表达式总是返回 `Foo` 类型的结果。

使用Lambda为std::function添加多态性

我们现在想编写一些观察函数，用来观察一些变量的变化，当相应变量的数值发生改变时会进行提示，比如气压仪或是股票软件之类的东西。当有些值发生变化时，对应的观察对象就会被调用，之后以对应的方式进行反应。

为了实现这个观察器，我们存储了一些相关的函数对象在一个 `vector` 中，这些函数都接受以 `int` 变量作为参数，这个参数就是观察到的值。我们不清楚这些函数对于传入值会做什么特殊的处理，不过我们也没有必要知道。

那么 `vector` 中的函数对象类型是什么呢？`std::vector<void (*)(int)>`，只要函数声明成 `void f(int)` 就符合这个这个函数指针类型的定义。这对于Lambda表达式同样有效，不过Lambda表达式是不能捕获任何值了——`[](int x) {...}`。对于捕获列表来说，Lambda表达式确实和普通的函数指针不同，因为其就不是一个函数指针，是一个函数对象，也就是将很多数据耦合到一个函数当中！想想在C++11时代之前，C++中没有Lambda表达式，类和结构体通常会将数据和函数耦合在一起，并且当你修改一个类中的数据成员时，你得到的是一个完全不同类型的数据。

这样 `vector` 中就无法将使用同样类型名字的不同类别的对象存储在一起。不能捕获已存在的变量，这个限制对于用户来说非常的不友好，也限制了代码的使用范围。用户该如何保存不同类型的函数对象呢？对接口进行约束，采用特定的传参方式传入已经观察到的值？

本节中，我们将展示使用 `std::function` 来解决这个问题，其将扮演一个“Lambda表达式多态包装器”的角色，捕获列表是不是空的都没有关系。

How to do it...

本节我们将创建很多Lambda表达式，其捕获类型是完全不同的，但是其函数签名的类型是相同的。然后，使用 `std::function` 将这些函数对象存入一个 `vector`：

1. 包含必要的头文件：

```
#include <iostream>
#include <deque>
#include <list>
#include <vector>
#include <functional>
```

2. 我们先实现一个简单的函数，其返回值是一个Lambda表达式。其需要传入一个容器，并且返回一个函数对象，这个函数对象会以引用的方式捕获容器。且函数对象本身接受传入一个整型参数。当向函数对象传入一个整型时，表达式将会把传入的整型，添加到捕获的容器尾部：

```

template <typename C>
static auto consumer (C &container)
    return [&] (auto value) {
        container.push_back(value);
    };
}

```

3. 另一个辅助函数将会打印传入的容器中所有的内容：

```

template <typename C>
static void print (const C &c)
{
    for (auto i : c) {
        std::cout << i << ", ";
    }
    std::cout << '\n';
}

```

4. 主函数中，我们先实例化一个 `deque` 和一个 `list`，还有一个 `vector`，这些容器存放的元素都是 `int` 类型。

```

int main()
{
    std::deque<int> d;
    std::list<int> l;
    std::vector<int> v;
}

```

5. 现在使用 `consumer` 函数对象与刚刚实例化的容器进行配合：将在 `vector` 中存储生成自定义的函数对象。然后，用一个 `vector` 存放着三个函数对象。每个函数对象都会捕获对应的容器对象。这些容器对象都是不同的类型，不过都是函数对象。所以，`vector` 中的实例类型为 `std::function<void(int)>`。所有函数对象都将隐式转换成一个 `std::function` 对象，这样就可以存储在 `vector` 中了。

```

const std::vector<std::function<void(int)>>
consumers
    {consumer(d), consumer(l), consumer(v)};
}

```

6. 现在我们将10个整型值传入自定义函数对象：

```

for (size_t i {0}; i < 10; ++i) {
    for (auto &&consume : consumers) {
        consume(i);
    }
}

```

7. 三个容器都包含了同样的10个整数。让我们来打印它们：

```
    print(d);
    print(l);
    print(v);
}
```

8. 编译运行程序，就会看到如下输出，和我们的期望是一样的。

```
$ ./std_function
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...

本节中比较复杂的地方就是这一行：

```
const std::vector<std::function<void(int)>> consumers
    {consumer(d), consumer(l), consumer(v)};
```

d, l和v对象都包装进一个 `consumer(...)` 调用中。这个调用会返回多个函数对象，这样每个函数对象都能捕获这三个容器实例。虽然函数对象只能接受 `int` 型变量为参数，但是其捕获到的是完全不同的类型。这就将不同类型的A、B和C变量存入到一个 `vector` 中一样。

为了这个功能，需要找到一个共同的类型，也就是能保存不同类型的函数对象，这个类型就是 `std::function`。一个 `std::function<void(int)>` 对象可以存储我们的函数对象或传统函数，其接受只有一个整型参数和返回为空的函数类型。这里使用了多态性，为函数类型进行解耦。思考如下的写法：

```
std::function<void(int)> f (
    [&vector] (int x) { vector.push_back(x); });
```

这里有个函数对象，将Lambda表达式包装入 `std::function` 对象当中，当我们调用 `f(123)` 时，会产生一个虚函数调用，其会重定向到对象内部的实际执行函数。

当存储函数对象时，`std::function` 就显得非常智能。当我们使用Lambda表达式捕获越来越多的变量时，`std::function` 实例的体积也会越来越大。如果对象体积特别巨大，那么其将会在堆上分配出对应内存空间来存放这个函数对象。这些对于我们代码的功能性并没有什么影响，这里需要让你了解一下是因为这样的存储方式会对性能有一定的影响。

Note:

很多初学者都认为或希望 `std::function<...>` 的实际表达类型是一个Lambda表达式。不过这是错误的理解！因为有多态库的帮助，其才能将Lambda表达式进行包装，从而抹去类型的差异。

并置函数

其实很多函数没有必要完全自定义的去实现。让我们先来看一个使用Haskell实现的在文本中查找单一单词的例子。第一行定义了一个 `unique_words` 函数，在第二行中传入一个字符串：

```
|Prelude> unique_words = length . group . sort . words . (map toLower)  
|Prelude> unique_words "A B c d a b c d e"  
5
```

Wow，就是这么简单！这里不对Haskell的语法做过多的解释，让我们来看一下代码。其定义了一个 `unique_words` 的函数，该函数对其传入的参数进行了一系列的处理。首先，使用 `map toLower` 将所有字符都小写化。然后，将句子用逗号进行分割，比如 `"foo bar baz"` 就会已变成 `["foo", "bar", "baz"]`。接下来，将单词列表进行排序。这样，`["a", "b", "a"]` 就会变为 `["a", "a", "b"]`。现在，使用 `group` 函数，其会将相同的词组放到一个列表中，也就是 `["a", "a", "b"]` 成为 `[["a", "a"], ["b"]]`。现在就差不多快完事了，接下来就让我们数一下列表中一共有多少个组，这个工作由 `length` 函数完成。

多么完美的编程方式呀！我们可以从右往左看，来了解这段代码是如何工作的。这里我就不需要关心每个细节是如何进行实现(除非其性能很差，或者有Bug)。

我们不是来赞美Haskell的，而是来提升我们自己C++技能的，这样的方式在C++中同样奏效。本节的例子会展示如何使用Lambda表达式来模仿并置函数。

How to do it...

本节中定义了一些函数对象，并将它们串联起来，也就是将一个函数的输出作为另一个函数的输入，以此类推。为了很好的展示这个例子，我们编写了一些串联辅助函数：

1. 包含必要的头文件

```
#include <iostream>  
#include <functional>
```

2. 然后，我们实现一个辅助函数 `concat`，其可以去任意多的参数。这些参数都是函数，比如f, g和h。并且一个函数的结果是另一个函数的输入，可以写成 `f(g(h(...)))`：

```
template <typename T, typename ...Ts>  
auto concat(T t, Ts ...ts)  
{
```

3. 现在，代码就会变有些复杂了。当用户提供函数f, g和h时，我们现将其转换为 `f(concat(g,h))`，然后再是 `f(g(concat(h)))`，类似这样进行递归，直到得到 `f(g(h(...)))` 为止。用户提供的这些函数都可以由Lambda表达式进行捕

获，并且**Lambda**表达式将在之后获得相应的参数

，然后前向执行这些函数 `f(g(h(p)))`。这个**Lambda**表达式就是我们要返回的。`if constexpr` 结构会检查在递归步骤中，当前函数是否串联了多个函数：

```
if constexpr (sizeof... (ts) > 0) {
    return [=] (auto ...parameters) {
        return t(concat(ts...) (parameters...));
    };
}
```

4. 当我们到达递归的末尾，编译器会选择 `if constexpr` 的另一分支。这个例子中，我们只是返回函数 `t`，因为其传入的只有参数了：

```
else {
    return t;
}
```

5. 现在，让我们使用刚创建的函数连接器对函数进行串联。我们先在主函数的起始位置定义两个简单的函数对象：

```
int main()
{
    auto twice ([] (int i) { return i * 2; });
    auto thrice ([] (int i) { return i * 3; });
}
```

6. 现在，来串联他们。这里我们将两个乘法器函数和一个**STL**函数 `std::plus<int>` 放在一起，**STL**的这个函数可以接受两个参数，并返回其加和。这样我们就得到了函数 `twice(thrice(plus(a, b)))`：

```
auto combined (
    concat(twice, thrice, std::plus<int>{})
);
```

7. 我们来应用一下。`combined` 函数现在看起来和一般函数一样，并且编译器会将这些函数连接在一起，且不产生任何不必要的开销：

```
std::cout << combined(2, 3) << '\n';
}
```

8. 编译运行这个例子就会得到如下的结果，和我们的期望一致，因为 $2 * 3 * (2 + 3)$ 为30：

```
$ ./concatenation
30
```

How it works...

`concat` 函数是本节的重点。其函数体看起来非常的复杂，因为其要对另一个 Lambda 表达式传过来 `ts` 参数包进行解析，`concat` 会递归多次调用自己，每次调用参数都会减少：

```
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 [=](auto ...parameters) {
            return t(parameters...);
        };
    }
}
```

让我们写一个简单点的版本，这次串联了三个函数：

```
template <typename F, typename G, typename H>
auto concat(F f, G g, H h)
{
    return [=](auto ... params) {
        return f( g( h( params... ) ) );
    };
}
```

这个例子看起来应该很简单了吧。返回的Lambda表达式可以对f, g和h函数进行捕获。这个Lambda表达式可以接受任意多的参数传入，然后在调用f, g和h函数。我们先定义 `auto combined(concat(f, g, h))`，并在之后传入两个参数，例如 `combined(2, 3)`，这里的2和3就为 `concat` 函数的参数包。

看起来很复杂，但 `concat` 却很通用，有别于 `f(g(h(params...)))` 式的串联。我们完成的是 `f(concat(g, h))(params...)` 的串联，`f(g(concat(h)))(params...)` 为其下一次递归调用的结果，最终会的结果为 `f(g(h(params...)))`。

通过逻辑连接创建复杂谓词

当使用通用代码过滤数据时，我们通常会定义一些谓词，这些谓词就是告诉计算机，哪些数据是我们想要的，哪些数据是我们不想要的。通常谓词都是组合起来使用的。

例如，当我们在过滤字符串时，我们需要实现一个谓词，当其发现输入的字符串以 `foo` 开头就返回`true`，其他情况都返回`false`。另一个谓词，当其发现输入的字符串以“`bar`”结尾时，返回`true`，否则返回`false`。

我们也不总是自己去定义谓词，有时候可以复用已经存在的谓词，并将它们结合起来使用。比如，如果我们既想要检查输入字符串的开头是否是 `foo`，又想检查结尾是否为“`bar`”时，就可以将之前提到的两个谓词组合起来使用。本节我们使用 `Lambda` 表达式，用一种更加舒服的方式来完成这件事。

How to do it...

我们将来实现一个非常简单的字符串过滤谓词，并且将其和辅助函数结合让其变得更加通用。

1. 包含必要的头文件

```
#include <iostream>
#include <functional>
#include <string>
#include <iterator>
#include <algorithm>
```

2. 这里实现两个简单的谓词函数，后面会用到它们。第一个谓词会告诉我们字符串的首字母是否是 `a`，第二个谓词则会告诉我们字符串的结尾字母是否为 `b`：

```
static bool begins_with_a (const std::string &s)
{
    return s.find("a") == 0;
}

static bool ends_with_b (const std::string &s)
{
    return s.rfind("b") == s.length() - 1;
}
```

3. 现在，让我们来实现辅助函数，我们称其为 `combine`。其需要一个二元函数作为其第一个参数，可以是逻辑‘与’或逻辑‘或’操作。之后的两个参数为需要结合在一起的谓词函数：

```
template <typename A, typename B, typename F>
auto combine(F binary_func, A a, B b)
{
```

4. 之后，我们会返回一个**Lambda**表达式，这个表达式可以获取到两个合并后的谓词。这个表达式需要一个参数，这个参数会传入两个谓词中，然后表达式将返回这个两个谓词结合后的结果：

```
    return [=] (auto param) {
        return binary_func(a(param), b(param));
    };
}
```

5. 在实现主函数之前，先声明所使用命名空间：

```
using namespace std;
```

6. 现在，让将两个谓词函数合并在一起，形成另一个全新的谓词函数，其会告诉我们输入的字符串是否以'a'开头，并且以'b'结尾，比如"ab"或"axxxb"就会返回**true**。二元函数我们选择 `std::logical_and`。这是个模板类，需要进行实例化，所以这里我们使用大括号对创建其实例。需要注意的是，因为该类的默认类型为**void**，所以这里我们并没有提供模板参数。特化类的参数类型，都由编译器推导得到：

```
int main()
{
    auto a_xxx_b = combine(
        logical_and<>{},
        begins_with_a, ends_with_b));
}
```

7. 我们现在可以对标准输入进行遍历，然后打印出满足全新谓词的词组：

```
copy_if(istream_iterator<string>{cin}, {},
        ostream_iterator<string>{cout, ", "},
        a_xxx_b);
cout << '\n';
}
```

8. 编译边运行程序，就会得到如下输出。我们输入了四个单词，但是只有两个满足我们的谓词条件：

```
$ echo "ac cb ab axxxb" | ./combine
ab, axxxb,
```

There's more...

STL已经提供了一些非常有用的函数对象，例
如 `std::logical_and`，`std::logical_or` 等等。所以我们没有必要所有东西都自己去
实现。可以去看一下**C++**的参考手册，了解一下都有哪些函数对象已经实现：

- 英文：<http://en.cppreference.com/w/cpp/utility/functional>
- 中文：<http://zh.cppreference.com/w/cpp/utility/functional>

使用同一输入调用多个函数

当我们有很多工作要做时，可能就会导致很多代码的重复。使用**Lambda**表达式就很容易的避免重复代码，并且**Lambda**表达式将帮助你将这些重复的任务包装起来。

本节，我们将使用**Lambda**表达式接受一组参数，然后分发给相应的任务函数。这种方式并不需要添加额外的数据结构，所以编译器很容易的将这些函数打包成一个二进制文件(并且没有额外的开销)。

How to do it...

我们将要完成两个**Lambda**表达式辅助器，一个能接受一组参数，并调用多个函数对象；另一个使用一个函数调用，引发后续多个函数调用。我们的例子中，我们将使用不同的打印函数打印一些信息出来。

1. 包含打印头文件。

```
#include <iostream>
```

2. 首先，让我们实现 `multicall` 函数，这个函数是本章的重点。这个函数可以接受任意数量的参数，并且返回一个**Lambda**表达式，这个**Lambda**表达式只接受一个参数。表达式可以通过这个参数调用所有已提供的函数。这样，我们可以定义 `auto call_all (multicall(f, g, h))` 函数对象，然后调用 `call_all(123)`，从而达到同时调用 `f(123); g(123); h(123);` 的效果。这个函数看起来比较复杂，是因为我们需要一个语法技巧来展开参数包`functions`，并在 `std::initializer_list` 实例中包含一系列可调用的函数对象。

```
template <typename ... Ts>
static auto multicall (Ts ...functions)
{
    return [=] (auto x) {
        (void) std::initializer_list<int>{
            ((void) functions(x), 0) ...
        };
    };
}
```

3. 下一个辅助器能接受一个函数`f`和一个参数包 `xs`。这里要表示的就是参数包中的每个参数都会传入`f`中运行。这种方式类似于 `for_each(f, 1, 2, 3)` 调用，从而会产生一系列调用—— `f(1); f(2); f(3);`。本质上来说，这个函数使用同样的技巧来为函数展开参数包 `xs`：

```

template <typename F, typename ... Ts>
static auto for_each (F f, Ts ...xs) {
    (void) std::initializer_list<int>{
        ((void)f(xs), 0) ...
    };
}

```

4. `brace_print` 函数能接受两个字符，并返回一个新的函数对象，这个函数对象可以接受一个参数 `x`。其将会打印这个参数，当然会让之前的两个字符将这个参数包围：

```

static auto brace_print (char a, char b) {
    return [=] (auto x) {
        std::cout << a << x << b << ", ";
    };
}

```

5. 现在，我们终于可以在`main`函数中使用这些定义好的东西了。首先，我们定义函数`f`, `g`和`h`。其使用括号打印函数将其参数进行包围。`nl` 函数只打印换行符。

```

int main()
{
    auto f (brace_print('(', ')'));
    auto g (brace_print('[', ']'));
    auto h (brace_print('{', '}'));
    auto nl ([](auto) { std::cout << '\n'; });
}

```

6. 让我们将所有函数和 `multicall` 辅助器放在一起：

```

auto call_fgh (multicall(f, g, h, nl));

```

7. 这里我们提供一组数字，之后这些数字就会被相应的括号包围，然后打印出来。这样，我们现在调用一次，就等于以前调用五次主函数中定义的函数。

```

for_each(call_fgh, 1, 2, 3, 4, 5);
}

```

8. 编译运行，我们应该能得到期望的结果：

```

$ ./multicaller
(1), [1], {1},
(2), [2], {2},
(3), [3], {3},
(4), [4], {4},
(5), [5], {5},

```

How it works...

我们刚刚实现的辅助函数还是挺复杂的。我们使用了 `std::initializer_list` 来帮助我们展开参数包。为什么这里不用特殊的数据结构呢？再来看一下 `for_each` 的实现：

```
auto for_each ([]) (auto f, auto ...xs) {
    (void) std::initializer_list<int>{
        ((void)f(xs), 0) ...
    };
}
```

这段代码的核心在于 `f(xs)` 表达式。`xs` 是一个参数包，我们需要将其进行解包，才能获取出独立的参数，以便调用函数`f`。不幸的是，我们知道这里不能简单的使用 `...` 标记，写成 `f(xs)...`。

所以，我能做的只能是构造出一个 `std::initializer_list` 列表，其具有一个可变的构造函数。表达式可以直接通过 `return std::initializer_list<int>{f(xs)...};` 方式构建，不过其也有缺点。在让我们看一下 `for_each` 的实现，看起来要比之前简单许多：

```
auto for_each ([]) (auto f, auto ...xs) {
    return std::initializer_list<int>{f(xs)...};
}
```

这看起来非常简单易懂，但是我们要了解其缺点所在：

1. 其使用`f`函数的所有调用返回值，构造了一个初始化列表。但我们并不关心返回值。
2. 虽然其返回的初始化列表，但是我们想要一个“即发即弃”的函数，这些函数不用返回任何东西。
3. `f`在这里可能是一个函数，因为其不会返回任何东西，可能在编译时就会被优化掉。

要想 `for_each` 修复上面所有的问题，会让其变的更加复杂。例子中做到了一下几点：

1. 不返回初始化列表，但会将所有表达式使用 `(void)std::initializer_list<int>{...}` 转换为 `void` 类型。
2. 初始化表达式中，其将 `f(xs)...` 包装进 `(f(xs),0)...` 表达式中。这会让程序将返回值完全抛弃，不过`0`将会放置在初始化列表中。
3. `f(xs)` 在 `(f(xs),0)...` 表达式中，将会再次转换成 `void`，所以这里就和没有返回值一样。

这些不幸的事导致例程如此复杂丑陋，不过其能为所有可变的函数对象工作，并且不管这些函数对象是否返回值，或返回什么样的值。

这种技术可以很好控制函数调用的顺序，严格保证多个函数/函数对象以某种顺序进行调用。

Note:

不推荐使用C风格的类型转换，因为C++有自己的转换操作。我们可以使用`reinterpret_cast<void>(expression)`代替例程中的代码行，不过这样会降低代码的可读性，会给后面的阅读者带来一些困扰。

使用`std::accumulate`和Lambda函数实现`transform_if`

大多数用过`std::copy_if`和`std::transform`的开发者可能曾经疑惑过，为什么标准库里面没有`std::transform_if`。`std::copy_if`会将源范围内符合谓词判断的元素挑出来，不符合条件的元素忽略。而`std::transform`会无条件的将源范围内所有元素进行变换，然后放到目标范围内。这里的变换谓词是由用户提供的一个函数，这个函数不会太复杂，比如乘以多个数或将元素完全转换成另一种类型。

这两个函数很早就存在了，不过到现在还是没有`std::transform_if`函数。本节就来实现这个函数。看起来实现这个函数并不难，可以通过谓词将对应的元素选择出来，然后将这些挑选出来的元素进行变换。不过，我们会利用这个机会更加深入的了解Lambda表达式。

How to do it...

我们将来实现我们的`transform_if`函数，其将会和`std::accumulate`一起工作。

1. 包含必要的头文件。

```
#include <iostream>
#include <iterator>
#include <numeric>
```

2. 首先，我们来实现一个`map`函数。其能接受一个转换函数作为参数，然后返回一个函数对象，这个函数对象将会和`std::accumulate`一起工作。

```
template <typename T>
auto map (T fn)
{
```

3. 当传入一个递减函数时，我们会返回一个函数对象，当这个函数对象调用递减函数时，其会返回另一个函数对象，这个函数对象可以接受一个累加器和一个输入参数。递减函数会在累加器中进行调用，并且`fn`将会对输入变量进行变换。如果这里看起来比较复杂的话，我们将在后面进行详细的解析：

```
    return [=] (auto reduce_fn) {
        return [=] (auto accum, auto input) {
            return reduce_fn(accum, fn(input));
        };
    };
}
```

4. 现在，让我们来实现一个 `filter` 函数。其和 `map` 的工作原理一样，不过其不会对输入进行修改(`map` 中会对输入进行变换)。另外，我们接受一个谓词函数，并且在不接受谓词函数的情况下，跳过输入变量，而非减少输入变量：

```
template <typename T>
auto filter(T predicate)
{
```

5. 两个Lambda表达式与 `map` 函数具有相同的函数签名。其不同点在于 `input` 参数是否进行过操作。谓词函数用来区分我们是否对输入调用 `reduce_fn` 函数，或者直接调用累加器而不进行任何修改：

```
return [=] (auto reduce_fn) {
    return [=] (auto accum, auto input) {
        if (predicate(input)) {
            return reduce_fn(accum, input);
        } else {
            return accum;
        }
    };
}
```

6. 现在让我们使用这些辅助函数。我们实例化迭代器，我们会从标准输入中获取整数值：

```
int main()
{
    std::istream_iterator<int> it {std::cin};
    std::istream_iterator<int> end_it;
```

7. 然后，我们会调用谓词函数 `even`，当传入一个偶数时，这个函数会返回`true`。变换函数 `twice` 会对输入整数做乘2处理：

```
auto even ([](int i) { return i % 2 == 0; });
auto twice ([](int i) { return i * 2; });
```

8. `std::accumulate` 函数会将所对应范围内的数值进行累加。累加默认就是通过 `+` 操作符将范围内的值进行相加。我们想要提供自己的累加函数，也就是我们不想只对值进行累加。我们会将迭代器 `it` 进行解引用，获得其对应的值，之后再对其进行处理：

```
auto copy_and_advance ([](auto it, auto input) {
    *it = input;
    return ++it;
});
```

9. 我们现在将之前零零散散的实现拼组在一起。我们对标准输入进行迭代，通过输出迭代器 `ostream_iterator` 将对应的值输出在终端上。`copy_and_advance` 函数对象将会接收用户输入的整型值，之后使用输出迭代器进行输出。将值赋值给输出迭代器，将会使打印变得高效。不过，我们只会将偶数挑出来，然后对其进行乘法操作。为了达到这个目的，我们将 `copy_and_advance` 函数包装入 `even` 过滤器中，再包装入 `twice` 引射器中：

```
    std::accumulate(it, end_it,
        std::ostream_iterator<int>{std::cout, ", "},
        filter(even) (
            map(twice) (
                copy_and_advance
            )
        );
        std::cout << '\n';
}
```

10. 编译并运行程序，我们将得到如下的输出。奇数都被抛弃了，只有偶数做了乘2运算：

```
$ echo "1 2 3 4 5 6" | ./transform_if
4, 8, 12,
```

How it works...

本节看起来还是很复杂的，因为我们使用了很多嵌套Lambda表达式。为了跟清晰的了解它们是如何工作的，我们先了解一下 `std::accumulate` 的内部工作原理。下面的实现类似一个标准函数的实现：

```
template <typename T, typename F>
T accumulate(InputIterator first, InputIterator last, T
init, F f)
{
    for (; first != last; ++first) {
        init = f(init, *first);
    }
    return init;
}
```

函数参数 `f` 在这起到主要作用，所有值都会累加到用户提供的 `init` 变量上。通常情况下，迭代器范围将会传入一组数字，类似 `0, 1, 2, 3, 4`，并且 `init` 的值为0。函数 `f` 只是一个二元函数，其会计算两个数的加和。

例子中循环将会将所有值累加到 `init` 上，也就类似于 `init += (((0 + 1) + 2) + 3) + 4`。这样看起来 `std::accumulate` 就是一个通用的折叠函数。折叠范围意味着，将二值操作应用于累加器变量和迭代范围内的每一个值(累加完一个数，再累加下一个

数)。这个函数很通用，可以用它做很多事情，就比如实现 `std::transform_if` 函数！`f` 函数也会递减函数中进行调用。

`transform_if` 的一种很直接的实现，类似如下代码：

```
template <typename InputIterator, typename  
OutputIterator, typename P, typename Transform>  
OutputIterator transform_if(InputIterator first,  
InputIterator last, OutputIterator out, P predicate,  
Transform trans)  
{  
    for ( ; first != last; ++first) {  
        if (predicate(*first)) {  
            *out = trans(*first);  
            ++out;  
        }  
    }  
    return out;  
}
```

这个实现看起来和 `std::accumulate` 的实现很类似，这里的 `out` 参数可以看作 `init` 变量，并且使用函数 `f` 替换 `if`。

我们确实做到了。我们构建了 `if` 代码块，并且将二元函数对象作为一个参数提供给了 `std::accumulate`：

```
auto copy_and_advance ([](auto it, auto input) {  
    *it = input;  
    return ++it;  
});
```

`std::accumulate` 会将 `init` 值作为二元函数 `it` 的参数传入，第二个参数则是当前迭代器所指向的数据。我们提供了一个输出迭代器作为 `init` 参数。这样 `std::accumulate` 就不会做累加，而是将其迭代的内容转发到另一个范围内。这就意味着，我们只需要重新实现 `std::copy` 就可以了。

通过 `copy_and_advance` 函数对象，使用我们提供的谓词，将过滤后的结果传入另一个使用谓词的函数对象：

```

template <typename T>
auto filter(T predicate)
{
    return [=] (auto reduce_fn) {
        return [=] (auto accum, auto input) {
            if (predicate(input)) {
                return reduce_fn(accum, input);
            } else {
                return accum;
            }
        };
    };
}

```

构建过程看上去没那么简单，不过先来看一下 `if` 代码块。当 `predicate` 函数返回 `true` 时，其将返回 `reduce_fn` 函数处理后的结果，也就是 `accum` 变量。这个实现省略了使用过滤器的操作。`if` 代码块位于 **Lambda** 表达式的内部，其具有和 `copy_and_advance` 一样的函数签名，这使它成为一个合适的替代品。

现在我们就要进行过滤，但不进行变换。这个操作有 `map` 辅助函数完成：

```

template <typename T>
auto map(T fn)
{
    return [=] (auto reduce_fn) {
        return [=] (auto accum, auto input) {
            return reduce_fn(accum, fn(input));
        };
    };
}

```

这段代码看起来就简单多了。其内部有一个还有一个 **Lambda** 表达式，该表达式的函数签名与 `copy_and_advance`，所以可以替代 `copy_and_advance`。这个实现仅转发输入变量，不过会通过二元函数对 `fn` 的调用，对参数进行量化。

之后，当我们使用这些辅助函数时，我们可以写成如下的表达式：

```

filter(even) (
    map(twice) (
        copy_and_advance
    )
)

```

`filter(even)` 将会捕获 `even` 谓词，并且返回给我们一个函数，其为一个包装了另一个二元函数的二元函数，被包装的那个二元函数则是进行过滤的函数。`map(twice)` 函数做了相同的事情，`twice` 变换函数，将 `copy_and_advance` 包装入另一个二元函数中，那另一个二元函数则是对参数进行变换的函数。

虽然没有任何的优化，但我们的代码还是非常的复杂。为了让函数之间能一起工作，我们对函数进行了多层嵌套。不过，这对于编译器来说不是一件很难的事情，并且能对所有代码进行优化。程序最后的结果要比实现 `transform_if` 简单很多。这里我们没有多花一分钱，就获得了非常好的函数模组。这里我们就像堆乐高积木一样，可将 `even` 谓词和 `twice` 转换函数相结合在一起。

编译时生成笛卡尔乘积

Lambda表达式结合参数包一起使用，可以用来解决比较复杂的问题。本节中，我们将实现一个函数对象，其能接受任意多的输入参数，然后生成相应的笛卡尔乘积。

笛卡尔乘积是一个数学运算。其可以表示为 $A \times B$ ，其意思为使用集合A和集合B来结算笛卡尔乘积。结果为另一个单独的集合，其包含集合A和集合B一一对应的组对。这个运算的意义在于，将两个集合中的元素进行匹配。下图就描述了这种运算操作：

		B		
		1	2	3
A		(x, 1)	(x, 2)	(x, 3)
x	y	(y, 1)	(y, 2)	(y, 3)
z		(z, 1)	(z, 2)	(z, 3)

图中， $A = (x, y, z)$ ， $B = (1, 2, 3)$ ，所产生的笛卡尔乘积为 $(x, 1)$ ， $(x, 2)$ ， $(x, 3)$ ， $(y, 1)$ ， $(y, 2)$ 等等。如果A和B为同一个集合，比如说是 $(1, 2)$ ，那么其笛卡尔乘积为 $(1, 1)$ ， $(1, 2)$ ， $(2, 1)$ ，和 $(2, 2)$ 。有时候，这样的操作却十分冗余，比如集合 $(1, 1)$ ，或是刚才例子中的 $(1, 2)$ 和 $(2, 1)$ 。笛卡尔乘积可以通过一个简单的条件，对结果进行过滤。

How to do it...

我们实现了一个函数对形象，其能接受一个函数 f ，以及一组参数。该函数对象将会通过输出参数集合创建笛卡尔乘积，将冗余的部分进行过滤，并对每个乘积调用函数 f 。

1. 包含打印输出的头文件。

```
#include <iostream>
```

2. 然后，我们定义一个简单的辅助函数，用来对组对中的值进行打印：

```

static void print(int x, int y)
{
    std::cout << "(" << x << ", " << y << ")" \n";
}

int main()
{

```

3. 复杂的地方到了。我们先实现了一个辅助函数 `cartesian`，我们将在下一步实现这个函数。这个函数能接受一个参数 `f`，在我们使用过程中，这个 `f` 函数就是 `print` 函数。另一些参数是 `x` 和参数包 `rest`。其包含了计算笛卡尔乘积的元素。在 `f(x, rest)` 表达式中：当 `x=1` 和 `rest=2, 3, 4`，为了得到结果，我们需要调用三次: `f(1, 2); f(1, 3); f(1, 4);`。`(x < rest)` 的条件，会删除冗余的组对。我们来看下代码：

```

constexpr auto call_cart (
    [=] (auto f, auto x, auto ...rest) constexpr {
        (void) std::initializer_list<int>{
            (((x < rest)
                ? (void)f(x, rest)
                : (void)0)
            , 0) ...
        };
    });

```

4. `cartesian` 函数在本节中，算是最复杂的部分了。其能接受一个参数包 `xs`，并返回一个其捕获的函数对象。返回的函数对象能接受一个函数对象 `f`。参数包，比如 `xs = 1, 2, 3`，其内部Lambda表达式将会生成如下调用：`call_cart(f, 1, 1, 2, 3); call_cart(f, 2, 1, 2, 3); call_cart(f, 3, 1, 2, 3);`。通过对这些函数的调用，我们能得到我们想要的所有笛卡尔乘积。我们使用 `...` 对 `xs` 参数包扩展了两次，第一次看起来有些奇怪。调用 `call_cart` 时，我们第一次对 `xs` 进行了扩展。第二次扩展将会使得 `call_cart` 调用多次，并且每次的第二个参数都会不同。

```

constexpr auto cartesian ([=] (auto ...xs)
constexpr {
    return [=] (auto f) constexpr {
        (void) std::initializer_list<int>{
            ((void)call_cart(f, xs, xs...), 0)...
        };
    };
});

```

5. 那么，现在让我们使用数字集 `1, 2, 3` 来生成笛卡尔乘积，并对组对进行打印。过滤了冗余的组对，所剩的结果应该为 `(1, 2)`，`(2, 3)`，和 `(1, 3)`。我们对很多的结果进行了过滤，并且不考虑结果中组对中的数字顺序。这也就是说，我们不需要 `(1, 1)`，并且认为 `(1, 2)` 和 `(2, 1)` 为同一个组对。首先，我

们让 `cartesian` 函数产生一个函数对象，其会包含所有可能的组对，并且能够接受我们的打印函数。然后，我们将所产生的组对，使用打印函数进行打印输出。我们将 `print_cart` 变量声明为 `constexpr`，这样我们就能在编译时获得所有的乘积结果：

```
constexpr auto print_cart (cartesian(1, 2, 3));

    print_cart(print);
}
```

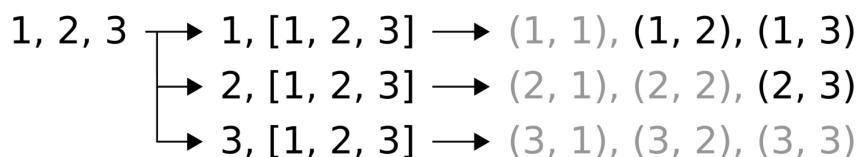
6. 编译并运行程序，我们就会得到如下的输出。通过 `call_cart` 中的 `x < rest` 判断条件，我们可以将一些冗余组对结果进行删除：

```
$ ./cartesian_product
(1, 2)
(1, 3)
(2, 3)
```

How it works...

另一个看起来比较复杂的地方就是Lambda表达式了。但当我们充分的了解后，我们就不会再对Lambda表达式有任何的困惑了！

那么，让我们来仔细的了解一下吧。我们将所发生的事情，画了一张图来说明：



这里有3步：

1. 我们将 `1, 2, 3` 作为新集合中的三个元素，其报了三个新的集合。第一个则是集合中的每一个单独向，而第二部分则是整个集合本身。
2. 我们可以将第一个元素与每一个元素相组合(包括自己)，就能得到很多组对。
3. 对于三个结果组对来说，我们只需要将其中不冗余的部分取出就好。

好了，回到我们例子：

```
constexpr auto cartesian ([=] (auto ...xs) constexpr {
    return [=] (auto f) constexpr {
        (void) std::initializer_list<int>{
            ((void) call_cart(f, xs, xs...), 0)...
        };
    };
});
```

内部表达式 `call_cart(xs, xs...)` 将会对集合 `1, 2, 3` 分别进行表示，比如：`1, [1, 2, 3]`。整个表达式 `((void)call_cart(f, xs, xs...), 0)...` 其将 ... 放在外部，其会将集合进行拆解，我们将会得到 `2, [1, 2, 3]` 和 `3, [1, 2, 3]`。

`call_cart` 完成了第2和第3步：

```
auto call_cart ([](auto f, auto x, auto ...rest)
constexpr {
    (void) std::initializer_list<int>{
        (((x < rest)
            ? (void)f(x, rest)
            : (void)0)
        , 0) ...
    };
});
```

参数 `x` 始终包含从这个集合中挑出的但选值，并且 `rest` 包含了整个集合。让我么先忽略 `x < rest` 这个条件。这里，`f(x, rest)` 表达式与 ... 参数包展开所得到的调用 `f(1, 1)`，`f(1, 2)` 等等，其就会生成将被打印的组对。这就是第2步完成的事。

第3步中，就是用 `x < rest` 条件来过滤冗余的组对了。

我们先给所有Lambda表达式和持有变量声明成 `constexpr`。通过这样做，我们可以在运行时对代码进行评估，这样编译出的二进制文件将会包含所有组对，而无需在运行时对其进行计算。需要注意的是，这里需要传入常量函数的参数为已知量，这样才能在运行时让编译器知道，并对函数进行执行。

第5章 STL基础算法

STL不仅包含数据结构，还有很多算法。数据结构可以帮助存放特定情况下需要保存的数据，而算法则会将数据结构中存储的数据进行变换。

让我们来看一个标准的例子，例如对 `vector` 实例中的数据进行累加。这个可以简单的通过循环迭代 `vector` 中的元素，将所有值累加在一个对应的值上：

```
vector<int> v {100, 400, 200 /*, ... */};

int sum {0};
for (int i : v) { sum += i; }

cout << sum << '\n';
```

不过，作为一个标准的例子，当然可以使用STL的算法来完成：

```
cout << accumulate(begin(v), end(v), 0) << '\n';
```

例子中循环变量也不是很长，不过其可读性比 `accumulate` 差很多。一个10行的循环代码看起来的确很尴尬，那么本章我们就看来了解一下标准算法(`accumulate`, `copy`, `move`, `transform` 和 `shuffle` 等等)的工作机制。

其思想就是提供丰富的算法供开发者使用，避免耗费大量的时间在重复制造轮子上面。另一方面就是，即便开发者会自己去实现相应STL中的算法，也要进行大量的测试来确保自己实现的算法是否正确，STL提供的算法都是经过了严格的测试。所以没有必要做重复的工作，这样也能节省代码审阅者的时间，否则他们还要确定算法实现中是否有Bug。

另一个重点是STL算法非常的高效。很多STL算法提供了多种特化实现，这样足以应对依赖迭代器类型的使用方式。例如，将 `vector` 中的所有元素都填充0时，就可以使用 `std::fill`。因为 `vector` 使用的是一段连续的内存，对于这种使用连续内存存放数据的结构都可以使用 `std::fill` 进行填充，这个函数类似于C中的 `memset` 函数。当开发者将容器类型从 `vector` 改为 `list`，STL算法就不能再使用 `memset` 了，并且需要逐个迭代 `list` 的元素，并将元素赋0。开发者不能为了使用 `memset` 将数据类型写死为 `vector` 或 `array`，因为实际项目中，还是有很多数据结构存储的地址并不是连续的。大多数情况下，想要自己去将代码实现的更聪明是没有太多意义的，因为STL的实现者已经考虑到了这种情况，并且STL还是免费使用的，为什么不用呢？

让我们总结一下前面提到的几点。使用STL算法的好处：

- **维护性：** 算法的名字已经说明它要做什么了。显式使用循环的方式与使用STL算法的方式没法对比。
- **正确性：** STL是由专家编写和审阅过的，并且经过了良好的测试，重新实现的复杂程度可能是你无法想象的。
- **高效性：** STL算法真的很高效，至少要比手写的循环要强许多。

很多算法都是对迭代器进行操作，第3章已经解释了迭代器的工作原理。本章专注于如何使用**STL**算法解决各种问题，了解这些**STL**应该如何使用。要展示所有**STL**算法的使用方式不是本书所要做的事情，这个事情**C++手册**已经完成了，你可以在网上进行查询，或者花钱购买电子/纸质发布版本。

作为一个**STL**“忍者”需要将**C++手册**放在手边……嗯，至少放在浏览器的书签中吧。当我们在完成一个任务的过程中，每个开发者都可以回看一下任务本身，在完成自己的任务时，确定这个**STL**算法是否适合于你的问题。

在线版本的**C++手册**: <http://cppreference.com>

其也提供离线下载功能。

Note:

在面试过程中，对于**STL**算法的熟悉程度也是判断一个开发者对**C++**的熟悉程度的标准之一。

容器间相互复制元素

大多数STL数据结构都支持迭代器。这就意味着大多数数据结构能够通过成员函数 `begin()` 和 `end()` 成员函数得到相应的迭代器，并能对数据进行迭代。迭代的过程看起来是相同的，无论是什么样的数据结构都是一样的。

我们可以对 `vector` , `list` , `deque` , `map` 等等数据结构进行迭代。我们甚至可以使用迭代器作为文件/标准输入输出的入口。此外，如之前章节介绍，我们能将迭代器接口放入算法中。这样的话，我们可以使用迭代器访问任何元素，并且可以将迭代器作为STL算法的参数传入，对特定范围内的数据进行处理。

`std::copy` 算法可以很好的展示迭代器是如何将不同的数据结构进行抽象，而后将一个容器的数据拷贝到另一个容器。类似这样的算法就与数据结构的类型完全没有关系了。为了证明这点，我们会把玩一下 `std::copy` 。

How to do it...

本节中，我们将对不同的变量使用 `std::copy` 。

- 首先，包含必要的头文件，并声明所用到的命名空间。

```
#include <iostream>
#include <vector>
#include <map>
#include <string>
#include <tuple>
#include <iterator>
#include <algorithm>

using namespace std;
```

- 我们将使用整型和字符串值进行组对。为了能很好的将其进行打印，我们将会重载 `<<` 流操作：

```
namespace std {
    ostream& operator<<(ostream &os, const pair<int,
    string> &p)
    {
        return os << "(" << p.first << ", " << p.second
        << ")";
    }
}
```

- 主函数中，我们将使用整型-字符串对填充一个 `vector`。并且我们声明一个 `map` 变量，其用来关联整型值和字符串值：

```

int main()
{
    vector<pair<int, string>> v {
        {1, "one"}, {2, "two"}, {3, "three"},
        {4, "four"}, {5, "five"}};

    map<int, string> m;
}

```

4. 现在将 `vector` 中的前几个整型字符串对使用 `std::copy_n` 拷贝到 `map` 中。因为 `vector` 和 `map` 是两种完全不同的结构体，我们需要对 `vector` 中的数据进行变换，这里就要使用到 `insert_iterator` 适配器。`std::inserter` 函数为我们提供了一个适配器。在算法中使用类似 `std::copy_n` 的算法时，需要与插入迭代器相结合，这是一种更加通用拷贝/插入元素的方式(从一种数据结构到另一种数据结构)，但这种方式不是最快的。使用指定数据结构的成员函数插入元素无疑是更加高效的方式：

```
copy_n(begin(v), 3, inserter(m, begin(m)));
```

5. 让我们打印一下 `map` 中的内容。纵观本书，我们会经常使用 `std::copy` 函数来打印容器的内容。`std::ostream_iterator` 在这里很有用，因为其可以将用户的标准输出作为另一个容器，而后将要输出的内容拷贝过去：

```

auto shell_it (ostream_iterator<pair<int,
string>>{cout,
", "});

copy(begin(m), end(m), shell_it);
cout << '\n';

```

6. 对 `map` 进行清理，然后进行下一步的实验。这次，我们会将 `vector` 的元素移动到 `map` 中，并且是所有元素：

```
m.clear();

move(begin(v), end(v), inserter(m, begin(m)));
```

7. 我们将再次打印 `map` 中的内容。此外，`std::move` 是一种改变数据源的算法，这次我们也会打印 `vector`。这样，我们就会看到算法时如何对数据源进行的移动：

```

copy(begin(m), end(m), shell_it);
cout << '\n';

copy(begin(v), end(v), shell_it);
cout << '\n';
}
```

8. 编译运行这个程序，看看会发生什么。第一二行非常简单，其反应的就是 `copy_n` 和 `move` 算法执行过后的结果。第三行比较有趣，因为移动算法将其源搬到 `map` 中，所以这时的 `vector` 是空的。在重新分配空间前，我们通常不应该访问成为移动源的项。但是为了这个实验，我们忽略它：

```
$ ./copying_items  
(1, one), (2, two), (3, three),  
(1, one), (2, two), (3, three), (4, four), (5, five),  
(1, ), (2, ), (3, ), (4, ), (5, ),
```

How it works...

`std::copy` 是STL中最简单的算法之一，其实现也非常短。我们可以看一下等价实现：

```
template <typename InputIterator, typename  
OutputIterator>  
OutputIterator copy(InputIterator it, InputIterator  
end_it,  
OutputIterator out_it)  
{  
    for (; it != end_it; ++it, ++out_it) {  
        *out_it = *it;  
    }  
    return out_it;  
}
```

这段代码很朴素，使用 `for` 循环将一个容器中的元素一个个的拷贝到另一个容器中。此时，有人就可能会发问：“使用 `for` 循环的实现非常简单，并且还不用返回值。为什么要在标准库实现这样的算法？”，这是个不错的问题。

`std::copy` 并非能让代码大幅度减少的一个实现，很多其他的算法实现其实非常复杂。这种实现其实在代码层面并不明显，但STL算法更多的在于做了很多底层优化，编译器会选择最优的方式执行算法，这些底层的东西目前还不需要去了解。

STL算法也让能避免让开发者在代码的可读性和优化性上做权衡。

Note:

如果类型只有一个或多个(使用 `class` 或 `struct` 包装)的矢量类型或是类，那么其拷贝赋值通常是轻量的，所以可以使用 `memcpy` 或 `memmove` 进行赋值操作，而不要使用自定义的赋值操作符进行操作。

这里，我们也使用了 `std::move`。其和 `std::copy` 一样优秀，不过 `std::move(*it)` 会将循环中的源迭代器，从局部值(左值)转换为引用值(右值)。这个函数就会告诉编译器，直接进行移动赋值操作来代替拷贝赋值操作。对于大多数复杂的对象，这会让程序的性能更好，但会破坏原始对象。

容器元素排序

排序是一项很常见的任务，并且可以通过各种各样的方式进行。每个计算机科学专业的学生，都学过很多排序算法(包括这些算法的性能和稳定性)。

因为这是个已解决的问题，所以开发者没必要浪费时间再次来解决排序问题，除非是出于学习的目的。

How to do it...

本节中，我们将展示如何使用 `std::sort` 和 `std::partial_sort`。

1. 首先，包含必要的头文件和声明所使用的命名空间。

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>
#include <random>

using namespace std;
```

2. 我们将打印整数在 `vector` 出现的次数，为了缩短任务代码的长度，我们在这里写一个辅助函数：

```
static void print(const vector<int> &v)
{
    copy(begin(v), end(v), ostream_iterator<int>
{cout, ", "});
    cout << '\n';
}
```

3. 我们开始实例化一个 `vector`：

```
int main()
{
    vector<int> v {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

4. 因为我们将使用不同的排序函数将 `vector` 多次打乱，所以我们需要一个随机数生成器：

```
random_device rd;
mt19937 g {rd()};
```

5. `std::is_sorted` 函数会告诉我们，容器内部的值是否已经经过排序。所以这行将打印到屏幕上：

```
cout << is_sorted(begin(v), end(v)) << '\n';
```

6. `std::shuffle` 将打乱 `vector` 中的内容，之后我们会再次对 `vector` 进行排序。前两个参数是容器的首尾迭代器，第三个参数是一个随机数生成器：

```
shuffle(begin(v), end(v), g);
```

7. 现在 `is_sorted` 函数将返回 `false`，所以 0 将打印在屏幕上，`vector` 的元素总量和具体数值都没有变，不过顺序发生了变化。我们会将函数的返回值再次打印在屏幕上：

```
cout << is_sorted(begin(v), end(v)) << '\n';
print(v);
```

8. 现在，在通过 `std::sort` 对 `vector` 进行排序。然后打印是否排序的结果：

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

cout << is_sorted(begin(v), end(v)) << '\n';
print(v);
```

9. 另一个比较有趣的函数是 `std::partition`。有时候，并不需要对列表完全进行排序，只需要比它前面的某些值小就可以。所以，让使用 `partition` 将数值小于 5 的元素排到前面，并打印它们：

```
shuffle(begin(v), end(v), g);

partition(begin(v), end(v), [] (int i) { return i
< 5; });

print(v);
```

10. 下一个与排序相关的函数是 `std::partial_sort`。我们可以使用这个函数对容器的内容进行排序，不过只是在某种程度上的排序。其会将 `vector` 中最小的 N 个数，放在容器的前半部分。其余的留在 `vector` 的后半部分，不进行排序：

```
shuffle(begin(v), end(v), g);
auto middle (next(begin(v), int(v.size()) / 2));
partial_sort(begin(v), middle, end(v));

print(v);
```

11. 当我们要对没做比较操作符的结构体进行比较，该怎么办呢？让我们来定义一个结构体，然后用这个结构体来实例化一个 `vector`：

```
struct mystuct {
    int a;
    int b;
};

vector<mystuct> mv { {5, 100}, {1, 50}, {-123,
1000},
{3, 70}, {-10, 20} };
```

12. `std::sort` 函数可以将比较函数作为第三个参数进行传入。让我们来使用它，并且传递一个比较函数。为了展示其实如何工作的，我们会对其第二个成员 `b` 进行比较。这样，我们将按 `mystuct::b` 的顺序进行排序，而非 `mystuct::a` 的顺序：

```
sort(begin(mv), end(mv),
[] (const mystuct &lhs, const mystuct &rhs) {
    return lhs.b < rhs.b;
});
```

13. 最后一步则是打印已经排序的 `vector`：

```
for (const auto &[a, b] : mv) {
    cout << "(" << a << ", " << b << ")";
}
cout << '\n';
```

14. 编译运行程序。第一个1是由 `std::is_sorted` 所返回的。之后将 `vector` 进行打乱后，`is_sorted` 就返回0。第三行是打乱后的 `vector`。下一个1是使用 `sort` 之后进行打印的。然后，`vector` 会被再次打乱，并且使用 `std::partition` 对部分元素进行排序。我们可以看到所有比5小的元素都在左边，比5大的都在右边。我们暂且将现在的顺序认为是乱序。倒数第二行展示了 `std::partial_sort` 的结果。前半部分的内容进行了严格的排序，而后半部分则没有。最后一样，我们将打印 `mystuct` 实例的结果。其结果是严格根据第二个成员变量的值进行排序的：

```
$ ./sorting_containers
1
0
7, 1, 4, 6, 8, 9, 5, 2, 3, 10,
1
1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
1, 2, 4, 3, 5, 7, 8, 10, 9, 6,
1, 2, 3, 4, 5, 9, 8, 10, 7, 6,
{-10, 20} {1, 50} {3, 70} {5, 100} {-123, 1000}
```

How it works...

这里我们使用了很多与排序算法相关的函数：

算法函数	作用
<code>std::sort</code>	接受一定范围的元素，并对元素进行排序。
<code>std::is_sorted</code>	接受一定范围的元素，并判断该范围的元素是否经过排序。
<code>std::shuffle</code>	类似于反排序函数；其接受一定范围的元素，并打乱这些元素。
<code>std::partial_sort</code>	接受一定范围的元素和另一个迭代器，前两个参数决定排序的范围，后两个参数决定不排序的范围。
<code>std::partition</code>	能够接受谓词函数。所有元素都会在谓词函数返回 <code>true</code> 时，被移动到范围的前端。剩下的将放在范围的后方。

对于没有实现比较操作符的对象来说，想要排序就需要提供一个自定义的比较函数。其签名为 `bool function_name(const T &lhs, const T &rhs)`，并且在执行过程中无副作用。

当然排序还有其他类似 `std::stable_sort` 的函数，其能保证排序后元素的原始顺序，`std::stable_partition` 也有类似的功能。

Note:

`std::sort` 对于排序有不同的实现。根据所提供的迭代器参数，其实现分为选择排序、插入排序、合并排序，对于元素数量较少的容器可以完全进行优化。在使用者的角度，我们通常都不需要了解这些。

从容器中删除指定元素

复制、转换和过滤是对一段数据常做的操作。本节，我们将重点放在过滤元素上。

将过滤出的元素从数据结构中移除，或是简单的移除其中一个，但对于不同数据结构来说，操作上就完全不一样了。在链表中(比如 `std::list`)，只要将对应节点的指针进行变动就好。不过，对于连续存储的结构体来说(比

如 `std::vector`，`std::array`，还有部分 `std::deque`)，删除相应的元素时，将会有其他元素来替代删除元素的位置。当一个元素槽空出来后，那么后面所有的元素都要进行移动，来将这个空槽填满。这个听起来都很麻烦，不过本节中我们只是想要从字符串中移除空格，这个功能没有太多的工作量。

当我们定义了一个结构体时，我们是不会考虑如何将其元素进行删除的。当需要做这件事的时候，我们才会注意到。`STL`中的 `std::remove` 和 `std::remove_if` 函数可以给我们提供帮助。

How to do it...

我们将通过不同的方式将 `vector` 中的元素进行删除：

1. 包含必要的头文件，并声明所使用的命名空间。

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

using namespace std;
```

2. 一个简单的打印辅助函数，用来打印 `vector` 中的内容：

```
void print(const vector<int> &v)
{
    copy(begin(v), end(v), ostream_iterator<int>
{cout, ", "});
    cout << '\n';
}
```

3. 我们将使用简单的整数对 `vector` 进行实例化。然后，对 `vector` 进行打印，这样就能和后面的结果进行对比：

```
int main()
{
    vector<int> v {1, 2, 3, 4, 5, 6};
    print(v);
```

4. 现在，我们移除 `vector` 中所有的2。`std::remove` 将2值移动到其他位置，这样这个值相当于消失了。因为 `vector` 长度在移除元素后变短了，`std::remove` 将会返回一个迭代器，这个迭代器指向新的末尾处。旧的 `end` 迭代器所指向的地方，实际上就没有什么意义了，所以我们可以告诉 `vector` 将这个位置进行擦除。我们使用两行代码来完成这个任务：

```
{  
    const auto new_end (remove(begin(v), end(v),  
    2));  
    v.erase(new_end, end(v));  
}  
print(v);
```

5. 现在，我们来移除奇数。为了完成移除，我们需要实现一个谓词函数，这个函数用来告诉程序哪些值是奇数，然后结合 `std::remove_if` 来使用。

```
{  
    auto odd_number ([](int i) { return i % 2 !=  
    0; });  
    const auto new_end (  
        remove_if(begin(v), end(v), odd_number));  
    v.erase(new_end, end(v));  
}  
print(v);
```

6. 下一个尝试的算法是 `std::replace`。我们使用这个函数将所有4替换成123。与 `std::replace` 函数对应，`std::replace_if` 也存在于STL中，同样可以接受谓词函数：

```
replace(begin(v), end(v), 4, 123);  
print(v);
```

7. 让我们重新初始化 `vector`，并为接下来的实验创建两个空的 `vector`：

```
v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
  
vector<int> v2;  
vector<int> v3;
```

8. 然后，我们实现两个判断奇偶数的谓词函数：

```
auto odd_number ([](int i) { return i % 2 != 0;  
});  
auto even_number ([](int i) { return i % 2 == 0;  
});
```

9. 接下来的两行做的事情完全相同。其将偶数拷贝到v2和v3中。第一行使用 `std::remove_copy_if` 函数，当相应数值不满足谓词条件时，函数会从源容器中拷贝到另一个容器中。第二行 `std::copy_if` 则是拷贝满足谓词条件的元素。

```
remove_copy_if(begin(v), end(v),
               back_inserter(v2), odd_number);
copy_if(begin(v), end(v),
        back_inserter(v3), even_number);
```

10. 然后，打印这两个 `vector`，其内容应该是完全相同的。

```
print(v2);
print(v3);
}
```

11. 编译运行程序。第一行输出的是 `vector` 初始化的值。第二行是移除2之后的内容。接下来一行是移除所有奇数后的结果。第4行是将4替换为123的结果。最后两行则是v2和v3中的内容：

```
$ ./removing_items_from_containers
1, 2, 3, 4, 5, 6,
1, 3, 4, 5, 6,
4, 6,
123, 6,
2, 4, 6, 8, 10,
2, 4, 6, 8, 10,
```

How it works...

这里我们使用了很多与排序算法相关的函数：

算法函数	作用
<code>std::remove</code>	接受一个容器范围和一个具体的值作为参数，并且移除对应的值。返回一个新的 <code>end</code> 迭代器，用于修改容器的范围。
<code>std::replace</code>	接受一个容器范围和两个值作为参数，将使用第二个数值替换所有与第一个数值相同的值。
<code>std::remove_copy</code>	接受一个容器范围，一个输出迭代器和一个值作为参数。并且将所有不满足条件的元素拷贝到输出迭代器的容器中。
<code>std::replace_copy</code>	与 <code>std::replace</code> 功能类似，但与 <code>std::remove_copy</code> 更类似些。源容器的范围并没有变化。
<code>std::copy_if</code>	与 <code>std::copy</code> 功能相同，可以多接受一个谓词函数作为是否进行拷贝的依据。

Note:

表中没有if的算法函数，都有一个*_if版本存在，其能接受谓词函数，通过谓词函数判断的结果来进行相应的操作。

改变容器内容

如果说 `std::copy` 是STL中最简单的算法，那么 `std::transform` 就是第二简单的算法。和 `copy` 类似，其可将容器某一范围的元素放置到其他容器中，在这个过程中允许进行一些变换(变换函数会对输入值进行一定操作，然后再赋给目标容器)。此外，两个具有不同元素类型的容器也可以使用这个函数。这个函数超级简单，并且非常有用，这个函数会让标准组件具有更好的可移植性。

How to do it...

本节，我们将使用 `std::transform` 在拷贝的同时，修改 `vector` 中的元素：

1. 包含必要的头文件，并且声明所使用的命名空间：

```
#include <iostream>
#include <vector>
#include <string>
#include <sstream>
#include <algorithm>
#include <iterator>

using namespace std;
```

2. `vector` 由简单的整数组成：

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

3. 为了打印元素，会将所有元拷贝到 `ostream_iterator` 适配器中。`transform` 函数可以接受一个函数对象，其能在拷贝过程中对每个元素进行操作。这个例子中，我们将计算每个元素的平方值，所以代码将打印出平方数。因为直接进行了打印，所以平方数并没有进行保存：

```
transform(begin(v), end(v),
         ostream_iterator<int>{cout, ", "},
         [] (int i) { return i * i; });
cout << '\n';
```

4. 再来做另一个变换。例如，对于数字3来说，显示成 $3^2 = 9$ 显然有更好的可读性。下面的辅助函数 `int_to_string` 函数对象就会使用 `std::stringstream` 对象进行打印操作：

```
auto int_to_string ([](int i) {
    stringstream ss;
    ss << i << "2 = " << i * i;
    return ss.str();
});
```

5. 这样就可以将整型值放入字符串中。可以说我么将这个证书映射到字符串中。

使用 `transform` 函数，使我们可以拷贝所有数值到一个字符串 `vector` 中：

```
vector<string> vs;

transform(begin(v), end(v), back_inserter(vs),
         int_to_string);
```

6. 在打印完成后，我们的例子就结束了：

```
copy(begin(vs), end(vs),
      ostream_iterator<string>(cout, "\n"));
}
```

7. 编译并运行程序：

```
$ ./transforming_items_in_containers
1, 4, 9, 16, 25,
12 = 1
22 = 4
32 = 9
42 = 16
52 = 25
```

How it works...

`std::transform` 函数工作原理和 `std::copy` 差不多，不过在拷贝的过程中会对源容器中的元素进行变换，这个变换函数由用户提供。

在有序和无序的vector中查找元素

通常，需要确定某种元素在某个容器范围内是否存在。如果存在，我们会对这个值进行修改，或者访问与其相关的值。

查找元素的目的是不同的。当想要让在一段已排序的元素中进行查找，可以使用二分查找法，这种方法要比线性查找快的多。如果没有排序，那么就只能进行线性遍历来查找对应的值。

传统的STL查找算法我们都可以使用，所以了解一下这些算法。本节将会使用两个不同的算法，线性查找算法 `std::find`，二分查找算法 `std::equal_range`。

How to do it...

本节，我们将对一个比较小的数据集进行线性和二分查找：

1. 包含必要的头文件，以及声明所使用的命名空间。

```
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
#include <string>

using namespace std;
```

2. 数据集会包含 `city` 结构体，只是存储的城市的名字和人口数量：

```
struct city {
    string name;
    unsigned population;
};
```

3. 搜索算法需要将元素与目标对象进行对比，所以我们需要重载 `city` 结构体的 `==` 操作符：

```
bool operator==(const city &a, const city &b) {
    return a.name == b.name && a.population ==
b.population;
}
```

4. 我们也需要将 `city` 实例进行打印，所以我们对其输出操作符 `<<` 也进行了重载：

```

ostream& operator<<(ostream &os, const city &city) {
    return os << "(" << city.name << ", "
        << city.population << ")";
}

```

5. 查找函数通常会返回迭代器。当函数找到相应的元素时，会返回指向其的迭代器，否则就会返回容器的 `end` 迭代器。第二种情况下，我们就不能对该迭代器进行访问。因为要打印输出结果，所以需要实现一个函数，这个函数会返回另一个函数对象，并会将数据结构的 `end` 迭代器进行包装。当要对结果进行打印时，会与容器的 `end` 迭代器相比较，如果不是 `end`，那么打印出查找到的值；如果是 `end`，则仅打印 `<end>`：

```

template <typename C>
static auto opt_print (const C &container)
{
    return [end_it = end(container)] (const auto
&item) {
        if (item != end_it) {
            cout << *item << '\n';
        } else {
            cout << "<end>\n";
        }
    };
}

```

6. 我们使用德国的一些城市对 `vector` 进行实例化：

```

int main()
{
    const vector<city> c {
        {"Aachen", 246000},
        {"Berlin", 3502000},
        {"Braunschweig", 251000},
        {"Cologne", 1060000}
    };
}

```

7. 使用这个辅助函数构造一个城市打印函数，其会获取到城市 `vector` 容器的 `end` 迭代器 `c`：

```

auto print_city (opt_print(c));

```

8. 使用 `std::find` 在 `vector` 中找到相应的元素——科隆(Cologne)。因为可以直接获得这个元素，所以这个搜索看起来毫无意义。不过，在查找之前并不知道这个元素在 `vector` 中的位置，而 `find` 函数告诉我们这个元素的具体位置。我们也可以写一个循环，仅对城市名进行比较，而无需比较人口数量。不过，这是个不是很好的设计。下一步，我们将做另外一个实验：

```

    {
        auto found_cologne (find(begin(c), end(c),
                                city{"Cologne", 1060000}));
        print_city(found_cologne);
    }

```

9. 当不需要知道对应城市的人口数量时，就不需要使用`==`操作符，只需要比较城市名称就好。`std::find_if`函数可以接受一个函数对象作为谓词函数。这样，就能只使用城市名来查找“科隆”了：

```

    {
        auto found_cologne (find_if(begin(c), end(c),
                                    [] (const auto &item) {
                                        return item.name == "Cologne";
                                    }));
        print_city(found_cologne);
    }

```

10. 为了让搜索更加优雅，可以实现谓词构建器。`population_higher_than`函数对象能接受一个人口数量，并且返回人口数量比这个数量多的城市。在这个小数据集中找一下多于2百万人口的城市。例子中，只有柏林(Berlin)符合条件：

```

    {
        auto population_more_than ([](unsigned i) {
            return [=] (const city &item) {
                return item.population > i;
            };
        });
        auto found_large (find_if(begin(c), end(c),
                                  population_more_than(2000000)));
        print_city(found_large);
    }

```

11. 使用的查找函数，线性的遍历容器，查找的时间复杂度为 $O(n)$ 。STL也有二分查找函数，其时间复杂度为 $O(\log(n))$ 。让我们生成一个新的数据集，其包含了一些整数，并构建了另一个`print`函数：

```

    const vector<int> v {1, 2, 3, 4, 5, 6, 7, 8, 9,
                          10};

    auto print_int (opt_print(v));

```

12. `std::binary_search`函数会返回一个布尔值，这个布尔值会告诉你函数是否找到了相应的元素，但是不会将指向元素的迭代器返回。二分查找需要查找的列表是已排序的，否则二分查找将出错：

```
    bool contains_7 {binary_search(begin(v), end(v),
7)};
    cout << contains_7 << '\n';
```

13. 如果需要获得查找的元素，就需要使用其他STL函数。其中之一就是 `std::equal_range`。其不会返回对应元素的迭代器给我们，不过会返回一组迭代器。第一个迭代器是指向第一个不小于给定值的元素。第二个迭代器指向第一个大于给定值的元素。我们的范围为数字1到10，那么第一个迭代器将指向7，因为其是第一个不小于7的元素。第二个迭代器指向8，因为其实第一个大于7的元素：

```
auto [lower_it, upper_it] (
    equal_range(begin(v), end(v), 7));
print_int(lower_it);
print_int(upper_it);
```

14. 当需要其中一个迭代器，可以使用 `std::lower_bound` 或 `std::upper_bound`。`lower_bound` 函数只会返回第一个迭代器，而 `upper_bound` 则会返回第二个迭代器：

```
print_int(lower_bound(begin(v), end(v), 7));
print_int(upper_bound(begin(v), end(v), 7));
}
```

15. 编译并运行这个程序，我们看到如下输出：

```
$ ./finding_items
{Cologne, 1060000}
{Cologne, 1060000}
{Berlin, 3502000}
1
7
8
7
8
```

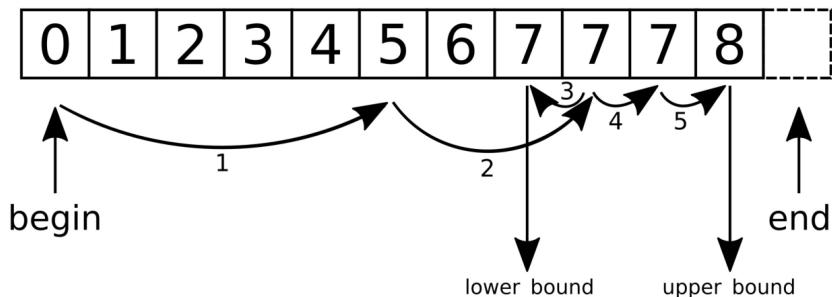
How it works...

本节使用的STL查找算法：

算法函数	作用
<code>std::find</code>	可将一个搜索范围和一个值作为参数。函数将返回找到的第一个值的迭代器。线性查找。
<code>std::find_if</code>	与 <code>std::find</code> 原理类似，不过其使用谓词函数替换比较值。
<code>std::binary_search</code>	可将一个搜索范围和一个值作为参数。执行二分查找，当找到对应元素时，返回 <code>true</code> ；否则，返回 <code>false</code> 。
<code>std::lower_bound</code>	可将一个查找返回和一个值作为参数，并且执行二分查找，返回第一个不小于给定值元素的迭代器。
<code>std::upper_bound</code>	与 <code>std::lower_bound</code> 类似，不过会返回第一个大于给定值元素的迭代器。
<code>std::equal_range</code>	可将一个搜索范围和一个值作为参数，并且返回一对迭代器。其第一个迭代器和 <code>std::lower_bound</code> 返回结果一样，第二个迭代器和 <code>std::upper_bound</code> 返回结果一样。

所有这些函数，都能接受一个自定义的比较函数作为可选参数传入。这样就可以自定义的进行查找，就如我们在本章做的那样。

来看一下 `std::equal_range` 是如何工作的。假设我们有一个 `vector`，`v = {0, 1, 2, 3, 4, 5, 6, 7, 7, 7, 8}`，并且调用 `equal_range(begin(v), end(v), 7)`，为了执行对7的二分查找。如 `equal_range` 要返回一对上下限迭代器那样，这个结果将返回一段区域 `{7, 7, 7}`，因为原始 `vector` 中有很多7，所以这个子队列中也有很多7。下图能说明其运行的原理：



首先，`equal_range` 会使用典型的二分查找，直到其找到那个不小于查找值的那个元素。而后，另一个迭代器也是用同样的方式找到。如同分开调用 `lower_bound` 和 `upper_bound` 一样。

为了获得一个二分查找函数，并返回其第一个适配条件的元素。我们可以按照如下的方式实现：

```

template <typename Iterator, typename T>
Iterator standard_binary_search(Iterator it, Iterator
end_it, T value)
{
    const auto potential_match (lower_bound(it, end_it,
value));
    if (potential_match != end_it && value ==
*potential_match) {
        return potential_match;
    }
    return end_it;
}

```

这个函数使用 `std::lower_bound`，为的就是找到第一个不大于 `value` 的元素。返回结果 `potential_match`，有三种情况：

- 没有值不小于 `value`。这样，返回值和 `end_it` (`end` 迭代器)一样。
- 遇到的第一个不小于 `value` 的元素，同时也大于 `value`。因此需要返回 `end_it`，表示没有找到相应的值。
- `potential_match` 指向的元素与 `value` 相同。这个匹配没毛病。因此就返回相应的迭代器。

当类型 `T` 没有 `==` 操作符时，需要为二分查找提供一个 `<` 操作实现。然后，可以将比较重写为 `!(value < *potential_match) && !(*potential_match < value)`。如果它们不小于，也不大于，那么必定等于。

STL 中因为缺少对多次命中的“定义”，所以并没有提供相应的函数来适配多次命中。

Note:

需要留意 `std::map` 和 `std::set` 等数据结构，它们有自己的 `find` 函数。它们携带的 `find` 函数要比通用的算法快很多，因为他们的实现与数据结构强耦合。

将vector中的值控制在特定数值范围内 ——std::clamp

很多应用中，需要获得相应的数据。在其进行绘制或进行其他处理前，会先对这些数据进行归一化，因为这些数据的差距很大。

通常可以使用 `std::transform` 通过传入一个谓词函数，对数据结构中的所有数据进行处理。不过，当不知道这些值有多大时或多小时，需要通过相应的函数找到数值的范围。

STL就包含这样的函数，比如 `std::minmax_element` 和 `std::clamp`。将这些函数与 Lambda函数相结合，可以解决一些简单的任务。

How to do it...

本节，将 `vector` 中的值使用两种不同的方式进行归一化，一种使用 `std::minmax_element`，另一种使用 `std::clamp`：

1. 包含必要的头文件，并声明所使用的命名空间。

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

using namespace std;
```

2. 将实现一个获得最大值和最小值的函数。这里最大值和最小值会更新，以便我们进行处理。函数对象会获取最大最小值，并返回另一个函数对象，这个返回的函数对象会做一些实际的转换。为了简单起见，新的最小值为0，所以旧值不需要进行偏移，并且值的归一化都是相对于0。为了有更好的可读性，这里忽略了最大值和最小值可能是一个值的可能性，不过在实际程序中需要格外注意这点，否则就会遇到除零问题：

```
static auto norm (int min, int max, int new_max)
{
    const double diff (max - min);
    return [=] (int val) {
        return int((val - min) / diff * new_max);
    };
}
```

3. 另一个函数对象构造器成为 `clampval`，其会返回一个函数对象用于捕获最小值和最大值，并调用 `std::clamp` 将值控制在一定范围内：

```
static auto clampval (int min, int max)
{
    return [=] (int val) -> int {
        return clamp(val, min, max);
    };
}
```

4. `vector` 中需要归一化的值大小不一。这些数据可能是热度数据、海拔高度或股票金额：

```
int main()
{
    vector<int> v {0, 1000, 5, 250, 300, 800, 900,
321};
```

5. 为对这些值进行归一化，我们需要找到这个 `vector` 中的最大值和最小值。`std::minmax_element` 函数将帮助我们获得这两个值。其会返回一组迭代器来代表这两个值：

```
const auto [min_it, max_it] =
minmax_element(begin(v), end(v));
```

6. 我们会将所有值从第一个 `vector` 拷贝到另一个中。让我们实例化第二个 `vector`，并且让其接收第一个 `vector` 中的值：

```
vector<int> v_norm;
v_norm.reserve(v.size());
```

7. 使用 `std::transform` 从第一个 `vector` 拷贝到第二个 `vector`。拷贝过程中，将会使用到归一化辅助函数。之前的最大值和最小值为0和1000。在归一化之后，为0和255：

```
transform(begin(v), end(v),
back_inserter(v_norm),
norm(*min_it, *max_it, 255));
```

8. 在实现另一个归一化策略之前，先将这个操作过后的结果进行打印：

```
copy(begin(v_norm), end(v_norm),
ostream_iterator<int>{cout, ", "});
cout << '\n';
```

9. 对已经归一化的 `vector` 使用 `clampval`，这时的最大值和最小值分别为255和0：

```
    transform(begin(v), end(v), begin(v_norm),
              clampval(0, 255));
```

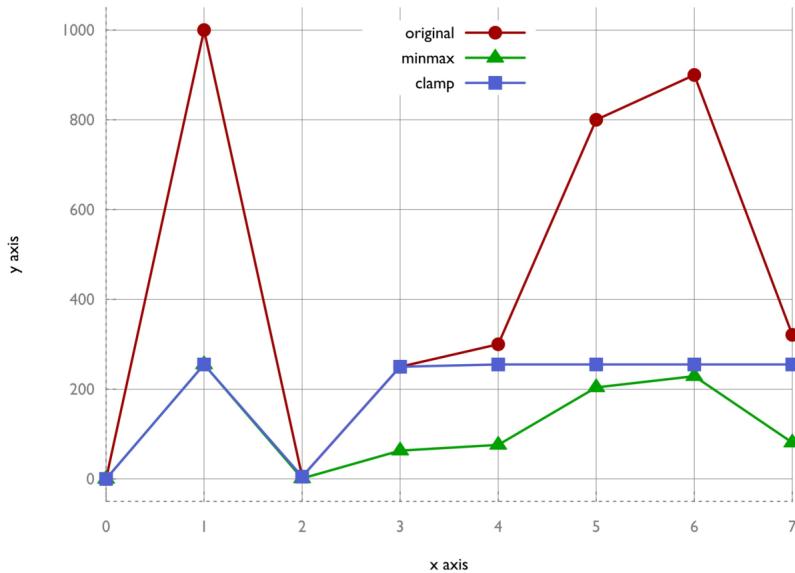
10. 完成之后，打印所有元素：

```
    copy(begin(v_norm), end(v_norm),
          ostream_iterator<int>{cout, ", "});
    cout << '\n';
}
```

11. 编译并运行程序。当前值的范围都在0到255之间，我们可以将其认为是RGB颜色的亮度值：

```
$ ./reducing_range_in_vector
0, 255, 1, 63, 76, 204, 229, 81,
0, 255, 5, 250, 255, 255, 255, 255,
```

12. 我们将对应的数据进行绘制，就得到了如下的图像。我们可以看到，使用最大最小值对原始数据进行变换，得到的数据时线性的。`clamp`曲线会损失一些信息。两种不同的结果在不同的情况下会很有用：



How it works...

除了 `std::transform`，我们使用量两个算法：

`std::minmax_element` 能接受一对 `begin` 和 `end` 迭代器作为输入。其会对这个范围进行遍历，然后找到这个范围内的最大值和最小值。其返回值是一个组对，我们会在我们的缩放函数中使用这个组对。

`std::clamp` 函数无法对一个范围进行可迭代操作。其接受三个值作为参数：一个给定值，一个最小值，一个最大值。这个函数的返回值则会将对应的值截断在最大值和最小值的范围内。我们也能使用 `max(min_val, min(max_val, x))` 来替代 `std::clamp(x, min_val, max_val)`。

在字符串中定位模式并选择最佳实现 ——`std::search`

在一个字符串中查找另一个字符串，与在一个范围内查找一个对象有些不同。首先，字符串是可迭代的对象。另一方面，从一个字符串中查询另一个字符串，就意味着就在一个范围内查询另一个范围。所以在查找过程中，有多次的比较，所以我们需要其他算法参与。

`std::string` 就包含 `find` 函数，其能实现我们想要的；不过，本节我们将使用 `std::search` 来完成这个任务。虽然，`std::search` 在字符串中会大量的用到，不过很多种容器都能使用这个算法来完成查找任务。**C++17**之后，`std::search` 添加了更多有趣的特性，并且其本身可使用简单地交换搜索算法。这些算法都优化过，并且免费提供给开发者使用。另外，我们可以实现自己的搜索算法，并且可以将我们实现的算法插入 `std::search` 中。

How to do it...

我们将对字符串使用新 `std::search` 函数，并且尝试使用其不同的查找对象进行应用：

1. 首先，包含必要的头文件，和声明所要使用的命名空间。

```
#include <iostream>
#include <string>
#include <algorithm>
#include <iterator>
#include <functional>

using namespace std;
```

2. 我们将实现一个辅助函数，用于打印查找算法所范围的位置，从而输出子字符串。

```
template <typename Itr>
static void print(Itr it, size_t chars)
{
    copy_n(it, chars, ostream_iterator<char>{cout});
    cout << '\n';
}
```

3. 我们例子输入的一个勒庞风格的字符串，其中包含我们要查找的字符串。本例中，这个需要查找的字符串为"elitr":

```

int main()
{
    const string long_string {
        "Lorem ipsum dolor sit amet, consetetur"
        " sadipscing elitr, sed diam nonumy eirmod";
    const string needle {"elitr"};

```

4. 旧 `std::search` 接口接受一组 `begin` 和 `end` 迭代器，用于确定子字符串的查找范围。这个接口会返回一个迭代器指向所查找到的子字符串。如果接口没有找到对应的字符串，其将返回该范围的 `end` 迭代器：

```

{
    auto match (search(begin(long_string),
end(long_string),
begin(needle),
end(needle)));
    print(match, 5);
}

```

5. C++17版本的 `std::search` 将会使用一组 `begin/end` 迭代器和一个所要查找的对象。`std::default_searcher` 能接受一组子字符串的 `begin` 和 `end` 迭代器，再在一个更大的字符串中，查找这个字符串：

```

{
    auto match (search(begin(long_string),
end(long_string),
default_searcher(begin(needle),
end(needle))));
    print(match, 5);
}

```

6. 这种改变就很容易切换搜索算法。`std::boyer_moore_searcher` 使用Boyer-Moore查找算法进行快速的查找：

```

{
    auto match (search(begin(long_string),
end(long_string),
boyer_moore_searcher(begin(needle),
end(needle))));
    print(match, 5);
}

```

7. C++17标准中，有三种不同的搜索器对象实现。其中还有一种是Boyer-Moore-Horspool查找算法实现：

```

    {
        auto match (search(begin(long_string),
                           end(long_string),
                           boyer_moore_horspool_searcher(begin(needle),
                                                         end(needle))));

        print(match, 5);
    }
}

```

8. 我们编译并运行这个程序。我们可以看到相同的字符串输出：

```

$ ./pattern_search_string
elitr
elitr
elitr
elitr
elitr

```

How it works...

我们在 `std::search` 中使用了4种查找方式，得到了相同的结果。这几种方式适用于哪种情况呢？

让我们假设大字符串为 `s`，要查找的部分为 `p`。然后，调用 `std::search(begin(s), end(s), begin(p), end(p))` 和 `std::search(begin(s), end(s), default_searcher(begin(p), end(p)))` 做相同的事情。

其他搜索方式将会以更复杂的方式实现：

- `std::default_searcher`：其会重定向到 `std::search` 的实现。
- `std::boyer_moore_searcher`：使用Boyer-Moore查找算法。
- `std::boyer_moore_horspool_searcher`：使用Boyer-Moore-Horspool查找算法。

为什么会有这些特殊的算法呢？Boyer-Moore算法起源于一个特殊想法——查找部分与原始字符串进行比较，其始于查找字符串的尾部，并且从右到左查找。如果查找的字符多个位置不匹配，并且对应部分没有出现，那么就需要在整个字符串进行平移，然后在进行查找。下图可能会看的更加明白一些。先来看一下第一步发生了什么：因为算法已知所要匹配字符串的长度，所以需要对相应位置上的字符进行比较，然后在平移到下一个长度点进行比较。在图中，这发生在第二步。这样Boyer-Moore算法就能避免对不必要的字符进行比较。



当然，在我们没有提供新的比较查找算法时，Boyer-Moore为默认的查找算法。其要比原先默认的算法快很多，不过其需要快速查找的数据结果进行支持，以判断搜索字符是否存在于查找块中，以及以多少为偏移进行定位。编译器将选择不同复杂度的算法实现，这取决于其所使用到的数据类型(对复杂类型的哈希映射和类型的原始查找表进行更改)。最后，默认的查找算法在查询不是很长的字符串也是非常快的。如果需要查找算法提高性能，那么Boyer-Moore将会是个不错的选择。

Boyer-Moore-Horspool为简化版的Boyer-Moore算法。其丢弃了“坏字符”规则，当对应字符串没有找到时，将会对整个查找块进行偏移。需要权衡的是，这个算法要比Boyer-Moore算法慢，但是其不需要对那么多特殊的数据结构进行操作。

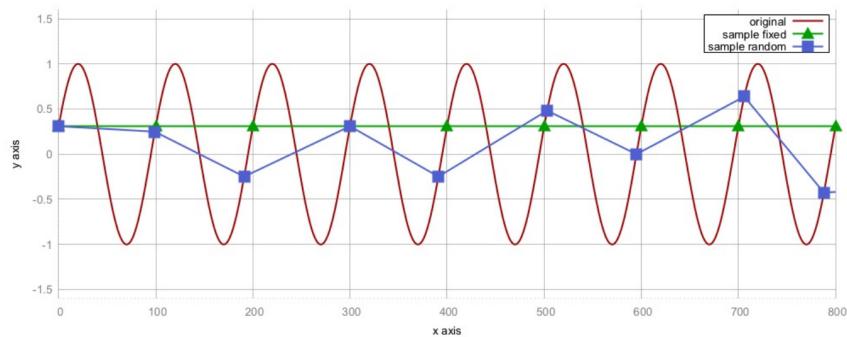
Note:

不要试图尝试比较哪种算法在那种情况下更快。你可以使用自己实际的例子进行测试，并且基于你得到的结果进行讨论。

对大**vector**进行采样

有时我们需要处理非常庞大的数据量，不可能在短时间内处理完这些数据。这样的话，数据可能就需要采样来减少要处理的数据量，从而加速整个处理过程。另一些情况下，不减少数据量也能加快程序处理的速度，不过这需要对一些数据进行存储或变换。

采样最原始的方式是每隔N个数据点，采样一次。在大多数情况下这样做没有问题，但是在信号处理中，其会引发一种称为混淆的数学情况。当减少两个随机采样点的距离时，这种现象会减弱。我们看一下下面的图，这张图就很能说明问题——当原始信号为一个sin波时，图例为三角的曲线就表示对这个曲线进行每隔100个点的取样。



不幸的是，其采样得到的值都是同一个Y值！连接起来就是与X轴平行的一条线。平方点采样，其每隔 `100+random(-15, +15)` 个值进行采样。不过，这样连接起来的曲线看起来和原始的曲线还是相差很远，所以在这个例子中就不能以固定的步长进行采样。

`std::sample` 函数不会添加随机值来改变采样的步长，而是采用完全随机的点进行采样。所以其工作方式与上图所显示的大为不同。

How to do it...

我们将对一个具有随机值的大**vector**进行采样。随机数据符合正态分布。采样结果也要符合正态分布，来让我们看下代码：

- 首先包含必要的头文件，以及声明所使用的命名空间。

```
#include <iostream>
#include <vector>
#include <random>
#include <algorithm>
#include <iterator>
#include <map>
#include <iomanip>

using namespace std;
```

2. 使用常数直接对变量进行初始化。第一个值代表了 `vector` 的的长度，第二个数代表了采样的步长：

```
int main()
{
    const size_t data_points {100000};
    const size_t sample_points {100};
```

3. 我们要使用符合正态分布的随机值生成器来将 `vector` 填满。这里先来确定正太分布的平均值和标准差：

```
const int mean {10};
const size_t dev {3};
```

4. 现在，我们来设置随机数生成器。首先，我们实例化一个随机设备，然后给定一个随机种子，对生成器进行初始化。然后，就可以得到对应分布的随机生成器：

```
random_device rd;
mt19937 gen {rd()};
normal_distribution<> d {mean, dev};
```

5. 对 `vector` 进行初始化，并用随机值将 `vector` 进行填充。这里会使用到 `std::generate_n` 算法，其会将随机值，通过 `back_inserter` 迭代器插入 `vector` 中。生成函数对象包装成了 `d(gen)` 表达式，其能生成符合分布的随机值：

```
vector<int> v;
v.reserve(data_points);

generate_n(back_inserter(v), data_points,
[&] { return d(gen); });
```

6. 我们再实例化另一个 `vector`，其来放采样过后的数值：

```
vector<int> samples;
v.reserve(sample_points);
```

7. `std::sample` 算法与 `std::copy` 的原理类似，不过其需要两个额外的参数：采样数量和随机值生成对象。前者确定输入范围，后者去确定采样点：

```
sample(begin(v), end(v), back_inserter(samples),
sample_points, mt19937{random_device{}()});
```

8. 这样就完成了采样。代码的最后展示一下我们的采样结果。输入数据符合正态分布，如果采样算法可行，那么其采样的结果也要符合正态分布。为了展示采样后的值是否符合正态分布，我们将数值的直方图进行打印：

```
map<int, size_t> hist;

for (int i : samples) { ++hist[i]; }
```

9. 最后，我们使用循环打印出直方图：

```
for (const auto &[value, count] : hist) {
    cout << setw(2) << value << " "
        << string(count, '*') << '\n';
}
```

10. 编译并运行程序，我们将看到采样后的结果，其也符合正态分布：

```
$ ./sampling_vectors
1 *
3 *
4 **
5 ****
6 *****
7 ******
8 *********
9 *********
10 *********
11 ******
12 *****
13 *****
14 ***
15 ***
16 ***
```

How it works...

`std::sample` 算法是C++17添加的。其函数签名如下：

```
template<class InIterator, class OutIterator,
         class Distance, class UniformRandomBitGenerator>
OutIterator sample(InIterator first, InIterator last,
                    SampleIterator out, Distance n,
                    UniformRandomBitGenerator&& g);
```

其输入范围有`first`和`last`迭代器确定，`out`迭代器作为采样输出。这些迭代器对于该函数来说和`std::copy`类似，元素从一个容器拷贝到另一个。`std::sample`算法只会拷贝输入中的一部分，因为采样结果只有`n`个元素。其在内部使用均匀分布，所以能以相同的概率选择输入范围中的每个数据点。

生成输入序列的序列

当测试代码需要处理参数顺序不重要的输入序列时，有必要测试它是否对所有可能的输入产生相同的输出。当你自己实现了一个排序算法时，就要写这样的测试代码来确定自己的实现是否正确。

`std::next_permutation` 在任何时候都能帮我们将序列进行打乱。我们在可修改的范围内可以调用它，其会将字典序进行置换。

How to do it...

本节，我们将从标准输入中读取多个字符串，然后使用 `std::next_permutation` 生成已排序的序列，并且打印这个序列：

1. 首先，包含必要的头文件，并声明所使用的命名空间。

```
#include <iostream>
#include <vector>
#include <string>
#include <iterator>
#include <algorithm>

using namespace std;
```

2. 使用标准数组对 `vector` 进行初始化，接下来对 `vector` 进行排序：

```
int main()
{
    vector<string> v {istream_iterator<string>{cin},
{}};
    sort(begin(v), end(v));
```

3. 现在来打印 `vector` 中的内容。随后，调用 `std::next_permutation`，其会打乱已经排序的 `vector`，再对其进行打印。直到 `next_permutation` 返回`false`时，代表 `next_permutation` 完成了其操作，循环结束：

```
do {
    copy(begin(v), end(v),
        ostream_iterator<string>{cout, ", "});
    cout << '\n';
} while (next_permutation(begin(v), end(v)));
}
```

4. 编译运行这个程序，会有如下的打印：

```
$ echo "a b c" | ./input_permutations
a, b, c,
a, c, b,
b, a, c,
b, c, a,
c, a, b,
c, b, a,
```

How it works...

`std::next_permutation` 算法使用起来有点奇怪。因为这个函数接受一组开始/结束迭代器，当其找到下一个置换时返回`true`；否则，返回`false`。不过“下一个置换”又是什么意思呢？

当 `std::next_permutation` 算法找到元素中的下一个字典序时，其会以如下方式工作：

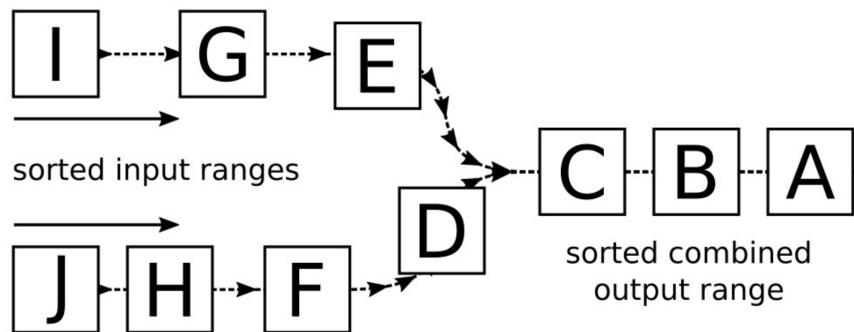
1. 通过 `v[i - 1] < v[i]` 的方式找到最大索引*i*。如果这个最大索引不存在，那么返回`false`。
2. 再找到最大所以j，这里j需要大于等于*i*，并且 `v[j] > v[i - 1]`。
3. 将位于索引位置j和*i - 1*上的值进行交换。
4. 将从*i*到范围末尾的元素进行反向。
5. 返回`true`。

每次单独的置换顺序，都会在同一个序列中呈现。为了看到所有置换的可能，我们先对数组进行了排序。如果我们输入“`c b a`”到算法中，算法会立即终止，因为每个元素都以反字典序排列。

实现字典合并工具

假设我们有一个已经排序的列表，有人有另一个已排序的列表，我们想要将这两个列表进行共享。那么最好的方式就是将这两个列表合并起来。我们需要合并后的列表也是有序的，这样我们查找元素就会十分方便。

为了将两个已排序列表中的元素进行合并，我们本能的会想需要创建一个新的列表来放置这两个列表中的元素。对于要加入的元素，我们需要将队列中的元素进行对比，然后找到最小的那个元素将其放到列表的最前面。不过，这样输出队列的顺序会被打乱。下面的图就能很好的说明这个问题：



`std::merge` 算法就可以直接来帮助我们做这个事情，这样我们就无需过多的参与。本节我们将展示如何使用这个算法。

How to do it...

我们将创建一个简单的字典，其为英语单词和德语单词一对一的翻译，之后将其存储在 `std::deque` 数据结构中。程序将标注输入中获取这个字典，并且打印合并之后的字典。

1. 包含必要的头文件，并声明所使用的命名空间。

```
#include <iostream>
#include <algorithm>
#include <iterator>
#include <deque>
#include <tuple>
#include <string>
#include <fstream>

using namespace std;
```

2. 字典是一对字符串，两两对应：

```
using dict_entry = pair<string, string>;
```

3. 我们将在屏幕上打印这个组对，并且要从用户输入中读取这个组对，所以我们必须要重载 `>>` 和 `<<` 操作符：

```
namespace std {
ostream& operator<<(ostream &os, const dict_entry p)
{
    return os << p.first << " " << p.second;
}
istream& operator>>(istream &is, dict_entry &p)
{
    return is >> p.first >> p.second;
}
```

4. 这里需要创建一个辅助函数，其能接受任何流对象作为输入，帮助我们构建字典。其会构建一个 `std::deque` 来存放一对一的字符串对，并且其会读取标准输入中的所有字符。并在返回字典前，对字典进行排序：

```
template <typename IS>
deque<dict_entry> from_instream(IS &&is)
{
    deque<dict_entry> d {istream_iterator<dict_entry>
{is}, {}};
    sort(begin(d), end(d));
    return d;
}
```

5. 这里使用不同的输入流，创建两个不同的字典。其中一个是从 `dict.txt` 文件中读取出的字符，我们先假设这个文件存在。其每一行为一个组对，另一个流就是标准输入：

```
int main()
{
    const auto dict1
(from_instream(ifstream{"dict.txt"}));
    const auto dict2 (from_instream(cin));
```

6. 作为辅助函数 `from_instream` 将返回给我们一个已经排过序的字典，这样我们就可以将两个字典直接放入 `std::merge` 算法中。其能通过给定两个的 `begin` 和 `end` 迭代器组确定输入的范围，并在最后给定输出。这里的输出将会打印在用户的屏幕上：

```
merge(begin(dict1), end(dict1),
begin(dict2), end(dict2),
ostream_iterator<dict_entry>{cout, "\n"});
}
```

7. 可以编译这个程序，不过在运行之前，我们需要创建 `dict.txt` 文件，并且写入如下内容：

```
car auto
cellphone handy
house haus
```

8. 现在我们运行程序了，输入一些英文单词，将其翻译为德文。这时的输出仍旧是一个排序后的字典，其可以将输入的所有单词进行翻译。

```
$ echo "table tisch fish fisch dog hund" |
./dictionary_merge
car auto
cellphone handy
dog hund
fish fisch
house haus
table tisch
```

How it works...

`std::meger` 算法接受两对 `begin/end` 迭代器，这两对迭代器确定了输入范围。这两对迭代器所提供的输入范围也必须是已排序的。第五个参数就是输出容器的迭代器，其接受两段范围合并的元素。

其有一个变体 `std::inplace_merge`。两个算法几乎一样，不过这个变体只需要一对迭代器，并且没有输出，和其名字一样，其会直接在输入范围上进行操作。比如对 `{A, C, B, D}` 这个序列来说，可以将第一个子序列定义为 `{A, C}`，第二个子序列定义为 `{B, D}`。使用 `std::inplace_merge` 算法将两个序列进行合并，其结果为 `{A, B, C, D}`。

第6章 **STL**算法的高级使用方式

上一章，我们了解了基础**STL**算法，并且使用简单的例子实践操作了一下这些**STL**接口：大多数**STL**算法都将一个或多个迭代器对，作为其输入或输出参数。**STL**算法也能接受谓词函数，自定义比较函数和转换函数。最后，有些接口会返回迭代器，因为其他算法还会用到这个迭代器。

STL算法旨在保持简单和通用。这样大多数代码就可以使用**STL**算法，从而让代码看起来简单明了。一个经验丰富的C++开发者对于**STL**算法非常了解，其会在代码中尽可能使用**STL**算法，这会让其他人更容易明白这段代码存在的原因，这样能帮助开发者和阅读者产生最大程度的共鸣。一个开发者的大脑会很容易的解析这些普及度很高的算法，这要比解析一段复杂的循环简单的多。虽然实现的主要方式一样，但是细节方面还有不同。

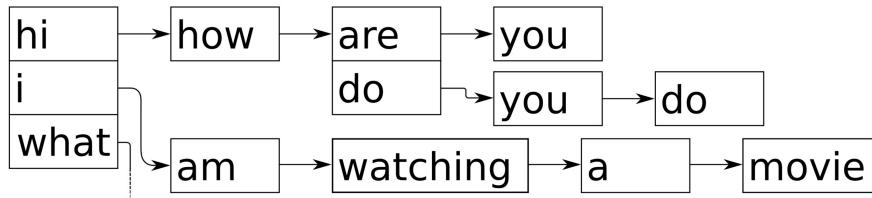
我们使用**STL**数据结构能够很好的避免指针、裸数组和粗犷的结构体。那么接下来我们将升级对**STL**算法的理解，以便使用通用的**STL**算法来替代复杂的循环控制符合代码块。因为代码变得短小，在提高可读性的同时，又增加了通用性。这就能够避免循环，仅调用 `std` 命名空间的算法即可，不过有时也会造成很糟糕的代码。我们不会去衡量代码是否糟糕，只会讨论可能性。

本章，我们将使用**STL**算法，以创造性的视野去了解现代C++能做些什么。我们将会实现属于我们自己的类**STL**算法，其能和已存在的数据结构完美结合，并且其他算法也会以同样的方式进行设计。我们也会将现有的**STL**算法与新算法相融合。这样的结合可以塑造出更加复杂的算法，不过其实现会更加短小，更具有可读性。过程中，我们可以看到**STL**是多么的简单和优雅。只有了解了所有方法，才能在使用时选择最合适的一种。

使用STL算法实现单词查找树类

所谓的trie数据类型，能够对感兴趣的数据进行存储，并且易于查找。将文本语句分割成多个单词，放置在列表中，我们能发现其开头一些单词的共性。

让我们看下下图，这里有两个句子“hi how are you”和“hi how do you do”，存储在一个类似于树的结构体中。其都以“hi how”开头，句子后面不同的部分，划分为树结构：



因为trie数据结构结合了相同的前缀，其也称为前缀树，很容易使用STL的数据结构实现。本章我们将关注如何实现我们自己的trie类。

How to do it...

本节，我们将使用STL数据结构和算法实现前缀树结构。

1. 包含必要的头文件和声明所使用的命名空间

```
#include <iostream>
#include <optional>
#include <algorithm>
#include <functional>
#include <iterator>
#include <map>
#include <vector>
#include <string>

using namespace std;
```

2. 我们首先实现一个类。我们的实现中，trie为map的递归映射。每个trie节点够包含一个map，节点的有效值`T`映射了下一个节点：

```
template <typename T>
class trie
{
    map<T, trie> tries;
```

3. 将新节点插入队列的代码很简单。使用者需要提供一个`begin/end`迭代器对，并且会通过循环进行递归。当用户输入的序列为{1, 2, 3}时，我们可以将1作为一个子trie，2为下一个子trie，以此类推。如果这些子trie在之前不存在，其将会通过`std::map`的`[]`操作符进行添加：

```

public:
    template <typename It>
    void insert(It it, It end_it) {
        if (it == end_it) { return; }
        tries[*it].insert(next(it), end_it);
    }

```

4. 我们这里也会定义一个辅助函数，用户只需要提供一个容器，之后辅助函数就会通过迭代器自动进行查询：

```

template <typename C>
void insert(const C &container) {
    insert(begin(container), end(container));
}

```

5. 调用我们的类时，可以写成这样 `my_trie.insert({"a", "b", "c"})`，必须帮助编译器正确的判断出这段代码中的所有类型，因此我们又添加了一个函数，这个函数用于重载的插入接口：

```

void insert(const initializer_list<T> &il) {
    insert(begin(il), end(il));
}

```

6. 我们也想了解，`trie`中有什么，所以我们需要一个打印函数。为了打印，我们可以对`trie`进行深度遍历。这样根节点下面的是第一个叶子节点，我们会记录我们所看到的元素的负载。当我们达到叶子节点，那么就可以进行打印了。我们会看到，当到达叶子的时候 `tries.empty()` 为`true`。递归调用`print`后，我们将再次弹出最后添加的负载元素：

```

void print(vector<T> &v) const {
    if (tries.empty()) {
        copy(begin(v), end(v),
              ostream_iterator<T>{cout, " "});
        cout << '\n';
    }
    for (const auto &p : tries) {
        v.push_back(p.first);
        p.second.print(v);
        v.pop_back();
    }
}

```

7. 打印函数需要传入一个可打印负载元素的列表，不过用户不需要传入任何参数就能调用它。这样，我们就定义了一个无参数的打印函数，其构造了辅助列表对象：

```

void print() const {
    vector<T> v;
    print(v);
}

```

8. 现在，我们就可以创建和打印trie了，我们将先搜索子trie。当trie包含的序列为 {a, b, c} 和 {a, b, d, e}，并且我们给定的序列为 {a, b}，对于查询来说，返回的子序列为包含 {c} 和 {d, e} 的部分。当我们找到子trie，将返回一个 const 的引用。在搜索中，也会出现没有要搜索序列的情况。即便如此，我们还是要返回些什么。`std::optional` 是一个非常好的帮手，因为当没有找到匹配的序列，我们可以返回一个空的 optional 对象：

```

template <typename It>
optional<reference_wrapper<const trie>>
subtrie(It it, It end_it) const {
    if (it == end_it) { return ref(*this); }
    auto found (tries.find(*it));
    if (found == end(tries)) { return {}; }

    return found->second.subtrie(next(it),
end_it);
}

```

9. 与 `insert` 方法类似，我们将提供一个只需要一个参数的 `subtrie` 方法，其能自动的从输入容器中获取迭代器：

```

template <typename C>
auto subtrie(const C &c) {
    return subtrie(begin(c), end(c));
}

```

10. 这样就实现完了。我们在主函数中使用我们trie类，使用 `std::string` 类型对类进行特化，并实例化对象：

```

int main()
{
    trie<string> t;
    t.insert({"hi", "how", "are", "you"});
    t.insert({"hi", "i", "am", "great", "thanks"});
    t.insert({"what", "are", "you", "doing"});
    t.insert({"i", "am", "watching", "a", "movie"});
}

```

11. 打印整个trie：

```
    cout << "recorded sentences:\n";
    t.print();
```

12. 而后，我们将获取输入语句的子trie，其以“hi”开头：

```
cout << "\npossible suggestions after \"hi\":\n";

if (auto st (t.subtrie(initializer_list<string>
{"hi"}));
st) {
st->get().print();
}
```

13. 编译并运行程序，其会返回两个句子的以“hi”开头的子trie：

```
$ ./trie
recorded sentences:
hi how are you
hi i am great thanks
i am watching a movie
what are you doing

possible suggestions after "hi":
how are you
i am great thanks
```

How it works...

有趣的是，单词序列的插入代码要比在子trie查找给定字母序列的代码简单许多。所以，我们首先来看一下插入代码：

```
template <typename It>
void trie::insert(It it, It end_it) {
    if (it == end_it) { return; }
    tries[*it].insert(next(it), end_it);
}
```

迭代器对 `it` 和 `end_it`，表示要插入的字符序列。`tries[*it]` 代表在子trie中要搜索的第一个字母，然后调用 `.insert(next(it), end_it);` 对更低级的子trie序列使用插入函数，使用迭代器一个词一个词的推进。`if (it == end_it) { return; }` 行会终止递归。返回语句不会做任何事情，这到有点奇怪了。所有插入操作都在 `tries[*it]` 语句上进行，`std::map` 的中括号操作将返回键所对应的值或是创建该键，相关的值（本节中映射类型是一个trie）由默认构造函数构造。这样，当我们查找不到理解的单词时，就能隐式的创建一个新的trie分支。

查找子trie看起来十分复杂，因为我们没有必要隐藏那么多的代码：

```
template <typename It>
optional<reference_wrapper<const trie>>
subtrie(It it, It end_it) const {
    if (it == end_it) { return ref(*this); }
    auto found (tries.find(*it));
    if (found == end(tries)) { return {}; }

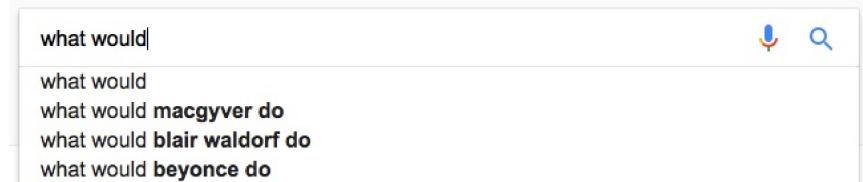
    return found->second.subtrie(next(it), end_it);
}
```

这段代码的主要部分在于 `auto found (tries.find(*it));`。我们使用 `find` 来替代中括号操作符。当我们使用中括号操作符进行查找时，`trie` 将会为我们创建丢失的元素（顺带一提，当我们尝试这样做，类的函数为 `const`，所以这样做事不可能的。这样的修饰能帮助我们减少 bug 的发生）。

另一个细节是返回值 `optional<reference_wrapper<const trie>>`。我们选择 `std::optional` 作为包装器，因为其可能没有我们所要找打 `tire`。当我们仅插入 “hello my friend”，那么就不会找到“goodbye my friend”。这样，我们仅返回 `{}` 就可以了，其代表返回一个空 `optional` 对象给调用者。不过这还是没有解释，我们为什么使用 `reference_wrapper` 代替 `optional<const trie &>`。`optional` 的实例，其为 `trie&` 类型，是不可赋值的，因此不会被编译。使用 `reference_wrapper` 实现一个引用，就是用来对对象进行赋值的。

使用树实现搜索输入建议生成器

上网时，在搜索引擎中输入要查找的东西时，对应下拉选项中会尝试猜测你想要查找什么。这种猜测是基于之前相关主题被查找的数量。有时搜索引擎十分有趣，其会显示一些奇怪的主题。



本章，我们将使用树类实现一个简单的搜索建议引擎。

How to do it...

本节，我们将实现一个终端应用，其能接受输入，并且能对所要查找的内容进行猜测，当然猜测的依据是我们用文本完成的“数据库”。

1. 包含必要的头文件和声明所使用的命名空间：

```
#include <iostream>
#include <optional>
#include <algorithm>
#include <functional>
#include <iterator>
#include <map>
#include <list>
#include <string>
#include <sstream>
#include <fstream>

using namespace std;
```

2. 我们将使用上一节实现的trie类：

```

template <typename T>
class trie
{
    map<T, trie> tries;
public:
    template <typename It>
    void insert(It it, It end_it) {
        if (it == end_it) { return; }
        tries[*it].insert(next(it), end_it);
    }

    template <typename C>
    void insert(const C &container) {
        insert(begin(container), end(container));
    }

    void insert(const initializer_list<T> &il) {
        insert(begin(il), end(il));
    }

    void print(list<T> &l) const {
        if (tries.empty()) {
            copy(begin(l), end(l),
                  ostream_iterator<T>(cout, " "));
            cout << '\n';
        }
        for (const auto &p : tries) {
            l.push_back(p.first);
            p.second.print(l);
            l.pop_back();
        }
    }

    void print() const {
        list<T> l;
        print(l);
    }

    template <typename It>
    optional<reference_wrapper<const trie>>
    subtrie(It it, It end_it) const {
        if (it == end_it) { return ref(*this); }
        auto found (tries.find(*it));
        if (found == end(tries)) { return {}; }

        return found->second.subtrie(next(it),
                                       end_it);
    }
}

```

```

template <typename C>
auto subtrie(const C &c) const {
    return subtrie(begin(c), end(c));
}
};

```

3. 实现一个简单的辅助函数，这个函数将用于提示用户输入他们想要查找的东西：

```

static void prompt()
{
    cout << "Next input please:\n > ";
}

```

4. 主函数中，我们打开一个文本文件，其作为我们的基础数据库。我们逐行读取文本文件的内容，并且将数据放入trie中解析：

```

int main()
{
    trie<string> t;
    fstream infile {"db.txt"};
    for (string line; getline(infile, line);) {
        istringstream iss {line};
        t.insert(istream_iterator<string>{iss}, {});
    }
}

```

5. 现在可以使用构建好的trie类，并且需要实现接收用户查询输入的接口。会提示用户进行输入，并且将用户的输入整行读取：

```

prompt();
for (string line; getline(cin, line);) {
    istringstream iss {line};
}

```

6. 通过文本输入，可以使用trie对其子trie进行查询。如果在数据库中已经有相应的语句，那么会对输入进行建议，否则会告诉用户没有建议给他们：

```

if (auto st (t.subtrie(istream_iterator<string>
{iss}, {})));
    st) {
    cout << "Suggestions:\n";
    st->get().print();
} else {
    cout << "No suggestions found.\n";
}

```

7. 之后，将打印一段分割符，并且再次等待用户的输入：

```
        cout << "-----\n";
        prompt();
    }
}
```

8. 运行程序之前，我们需要将db.txt文件进行设置。查找的输入可以是任何字符，并且其不确保是已经排过序的。进入trie类的所有语句：

```
do ghosts exist
do goldfish sleep
do guinea pigs bite
how wrong can you be
how could trump become president
how could this happen to me
how did bruce lee die
how did you learn c++
what would aliens look like
what would macgiver do
what would bjarne stroustrup do
...
```

9. 创建完db.txt之后，我们就可以运行程序了。其内容如下所示：

```
hi how are you
hi i am great thanks
do ghosts exist
do goldfish sleep
do guinea pigs bite
how wrong can you be
how could trump become president
how could this happen to me
how did bruce lee die
how did you learn c++
what would aliens look like
what would macgiver do
what would bjarne stroustrup do
what would chuck norris do
why do cats like boxes
why does it rain
why is the sky blue
why do cats hate water
why do cats hate dogs
why is c++ so hard
```

10. 编译并运行程序，然后进行输入查找：

```
$ ./word_suggestion
Next input please:
> what would
Suggestions:
aliens look like
bjarne stroustrup do
chuck norris do
macgiver do
-----
Next input please:
> why do
Suggestions:
cats hate dogs
cats hate water
cats like boxes
-----
Next input please:
>
```

How it works...

`trie`是如何工作的，已经在上一节中介绍过了，不过本节我们对其进行填充和查找的过程看起来有些奇怪。让我们来仔细观察一下代码片段，其使用文本数据库文件对空`trie`类进行填充：

```
fstream infile {"db.txt"};
for (string line; getline(infile, line);) {
    istringstream iss {line};
    t.insert(istream_iterator<string>{iss}, {});
}
```

这段代码会逐行的将文本文件中的内容读取出来。然后，我们将字符串拷贝到一个`istringstream`对象中。我们可以根据输入流对象，创建一个`istream_iterator`迭代器，其能帮助我们查找子`trie`。这样，我们就需要将字符串放入`vector`或`list`中了。上述代码中，有一段不必要的内存分配，可以使用移动方式，将`line`中的内容移动到`iss`中，避免不必要的内存分配。不过，`std::istringstream`没有提供构造函数，所以只能将`std::string`中的内容移动到流中。不过，这里会对输入字符串进行复制。

当在`trie`中查询用户的输入时，使用了相同的策略，但不使用输入文件流。我们使用`std::cin`作为替代，因为`trie::subtrie`对迭代器的操作，和`trie::insert`如出一辙。

There's more...

这里有必要对每个trie节点添加统计变量，这样我们就能知道各种前缀被查询的频率。因此，我们就可以将程序的建议进行排序，当前的搜索引擎就是这样做的。智能手机触摸屏文本输入的建议，也可以通过这种方式实现。

这个修改就留给读者当作业了。：）

使用STL数值算法实现傅里叶变换

信号处理领域傅里叶变换是非常重要和著名的公式。这个公式发现于200年前，其计算机用例实很多了。傅里叶变换可以用于音频/图像/视频压缩、音频滤波、医疗图像设备和用于辨识声音的手机引用。

因为其应用领域广泛，STL也试图将其用在数值计算领域。傅里叶变换只是其中一个例子，同样也是非常棘手的一个。其公式如下所示：

$$\hat{S}_k = \sum_{j=0}^{N-1} S_j \cdot e^{-i2\pi \frac{kj}{N}}$$

公式基于累加和的变换。累加中的每个元素是输入信号向量中的一个数据点和表达式 $\exp(-2 * i * \dots)$ 的乘积。这里需要一些工程数学的知识，你需要简单的了解复数的概念，如果你没有相关的知识，了解概念就可以了。仔细观察这个公式，其就是将信号中的所有数据点进行加和(信号数据的长度为N)，其循环索引值为j。其中k是另一个循环变量，因为傅里叶变换计算出的是一组值。在这组值中，每一个数据点都表示着一段重复波形的幅值和相位，这些信息不包含在原始数据中。当使用循环对其进行实现时，代码可能就会写成下面这样：

```
csignal fourier_transform(const csignal &s) {
    csignal t(s.size());
    const double pol {-2.0 * M_PI / s.size()};
    for (size_t k {0}; k < s.size(); ++k) {
        for (size_t j {0}; j < s.size(); ++j) {
            t[k] += s[j] * polar(1.0, pol * k * j);
        }
    }
    return t;
}
```

这里 `csignal` 的类型可能是 `std::vector`，其每个元素都是一个复数。对于复数而言，STL中已经有了对应的数据结构可以对其进行表示

—— `std::complex`。 `std::polar` 函数计算得是 $\exp(-2 * i * \dots)$ 部分。

这样实现看起来也挺好，不过本节中我们将使用STL工具对其进行实现。

How to do it...

本节，我们将实现傅里叶变换和逆变换，然后会对一些信号进行转换：

1. 首先，包含必要的头文件和声明所使用的命名空间：

```

#include <iostream>
#include <complex>
#include <vector>
#include <algorithm>
#include <iterator>
#include <numeric>
#include <valarray>
#include <cmath>

using namespace std;

```

2. 信号点的值一个复数，我们使用 `std::complex` 来表示，并使用 `double` 进行特化。我们可以对类型进行别名操作，使用 `cplx` 表示两个 `double` 值，这两个 `double` 值分别表示复数的实部和虚部。使用 `csignal` 来别名相应的 `vector` 对象：

```

using cplx = complex<double>;
using csignal = vector<cplx>;

```

3. 我们需要使用数值指针遍历数值序列。公式中的变量 `k` 和 `j` 就会随着序列进行累加：

```

class num_iterator {
    size_t i;
public:
    explicit num_iterator(size_t position) :
        i{position} {}

    size_t operator*() const { return i; }

    num_iterator& operator++() {
        ++i;
        return *this;
    }

    bool operator!=(const num_iterator &other) const
    {
        return i != other.i;
    }
};

```

4. 傅里叶变换需要接收一个信号，并返回一个新的信号。返回的信号表示已经经过傅里叶变换的信号。通过傅里叶逆变换，我们可以将一个经过傅里叶变换的信号，还原成原始信号，这里我们会提供一个可选的 `bool` 参数，其会决定变换的方向。`bool` 参数作为参数是一种不好习惯，特别是在一个函数的签名中出现多次。我们这有个很简洁的例子。我们做的第一件事，是使用原始信号的尺寸来分配新的信号数组：

```

    csignal fourier_transform(const csignal &s, bool back
= false)
{
    csignal t (s.size());

```

5. 公式中有两个因子，其看起来是相同的。让我们将其打包成一个变量：

```

        const double pol {2.0 * M_PI * (back ? -1.0 :
1.0)};
        const double div {back ? 1.0 : double(s.size())};

```

6. `std::accumulate` 很适合用来执行公式中的累加部分，我们将对一个范围内的数值使用 `accumulate`。对于每个值，我们将逐步的进行单个相加。`std::accumulate` 算法会调用一个二元函数。该函数的第一个参数为目前为止我们所累加的变量 `sum`，第二个参数为范围内下一个要累加的值。我们会在信号 `s` 中对当前为止的值进行查找，并且会将其和复数因子 `pol` 相乘。然后，我们返回新的 `sum`。这里的二元函数，使用Lambda表达式进行包装，因为我们在每次 `accumulate` 的调用时，`j` 变量的值是不同的。因为其是二维循环算法，所以内层Lambda做内部的循环，外层Lambda做外层的循环：

```

auto sum_up ([=, &s] (size_t j) {
    return [=, &s] (cmplx c, size_t k) {
        return c + s[k] *
            polar(1.0, pol * k * j /
double(s.size()));
    };
});

```

7. 傅里叶的内部循环，现在使用 `std::accumulate` 进行，算法中每个位置都会进行加和。我们使用Lambda表达式来实现，这样我们就能计算出傅里叶变换数组中的每个数据点的值：

```

auto to_ft ([=, &s] (size_t j) {
    return accumulate(num_iterator{0},
                     num_iterator{s.size()},
                     cmplx{},
                     sum_up(j))
    / div;
});

```

8. 目前位置，还没有执行傅里叶变换的代码。我们会准备大量的功能性代码，他们帮助我们完成很多事情。`std::transform` 的调用将会使 `j` 的值在 $[0, N)$ 间变换（这步是在外层循环完成）。变换之后的值将全部放入 `t` 中，`t` 就是我们要返回给用户的值：

```

        transform(num_iterator{0},
num_iterator{s.size()},
begin(t), to_ft);
return t;
}

```

9. 我们将会实现一些辅助函数帮助我们生成信号。首先实现的是一个余弦信号生成器，其会返回一个**Lambda**表达式，这个表达式通过传入的长度参数，产生对应长度的余弦信号数据。信号本身的长度是不固定的，但是其有固定的周期。周期为N，意味着该信号会在N步之后重复。返回的**Lambda**表达式不接受任何参数。我们可以重复的对其进行调用，并且每次调用表达式将会返回给我们下一个时间点的信号值：

```

static auto gen_cosine (size_t period_len) {
    return [period_len, n{0}] () mutable {
        return cos(double(n++) * 2.0 * M_PI /
period_len);
    };
}

```

10. 我们所要生成另一个波形是方波。该波形会在 -1 和 +1 两值间震荡，其中不会有其他的值。公式看起来有点复杂，但是其变换非常简单，也就是将值n置为 +1 或 -1， 并且其震荡周期为 `period_len`。这里要注意，我们没有使用0对n进行初始化。这样，我们的方波的其实位置就在 +1 上：

```

static auto gen_square_wave (size_t period_len)
{
    return [period_len, n{period_len*7/4}] () mutable
{
    return ((n++ * 2 / period_len) % 2) * 2 -
1.0;
}
}

```

11. 产生实际信号可以通过 `vector` 和信号生成器联合进行，使用重复调用信号生成器对 `vector` 数组进行填充。`std::generate` 就用来完成这个任务的。其接受一组 `begin/end` 迭代器组和一个生成函数。对于每个合法的迭代器，都会进行 `*it = gen()`。通过将这些代码包装成一个函数，我们可以很容易的生成一个信号数组：

```

template <typename F>
static csignal signal_from_generator(size_t len, F
gen)
{
    csignal r (len);
    generate(begin(r), end(r), gen);
    return r;
}

```

12. 最后，我们需要将信号的结果进行打印。我们可以将数组中的值拷贝到输出流迭代器中进行输出，不过我们需要先将数据进行变换，因为我们的信号数据都是复数对。这样，我们只需要在意每个点的实部就好；所以，我们可以将数组扔到 `std::transform` 中进行变换，然后将实部提取出来：

```

static void print_signal (const csignal &s)
{
    auto real_val ([](cmplx c) { return c.real(); });
    transform(begin(s), end(s),
              ostream_iterator<double>{cout, " "},
              real_val);
    cout << '\n';
}

```

13. 目前为止，傅里叶公式就已经实现了，不过现在还没有信号进行变换。这个工作我们将在主函数中完成。我们先来定义信号数据的长度：

```

int main()
{
    const size_t sig_len {100};
}

```

14. 现在来生成信号数据，转换他们，然后进行打印。首先，生成一个余弦信号和一个方波信号。这两组信号的长度和周期数相同：

```

auto cosine (signal_from_generator(sig_len,
                                    gen_cosine(sig_len / 2)));
auto square_wave (signal_from_generator(sig_len,
                                         gen_square_wave(sig_len / 2)));

```

15. 那么现在有了两个波形信号。为了生成第三个信号，我们对方波信号进行傅里叶变换，并且保存在 `trans_sqw` 数组中。方波的傅里叶变换有些特殊，我们在后面会进行介绍。索引从 10 到 `(signal_length - 10)` 都设置为 0.0。经过傅里叶变换之后，原始信号将发生很大的变化。我们将在最后看到结果：

```

    auto trans_sqw (fourier_transform(square_wave));

    fill (next(begin(trans_sqw), 10),
prev(end(trans_sqw), 10), 0);
    auto mid (fourier_transform(trans_sqw, true));

```

16. 现在，我们有三个信号：余弦、mid和方波。对于每个信号，我们将会打印其原始波形，和傅里叶变换过后的波形。输出将有六条曲线组成：

```

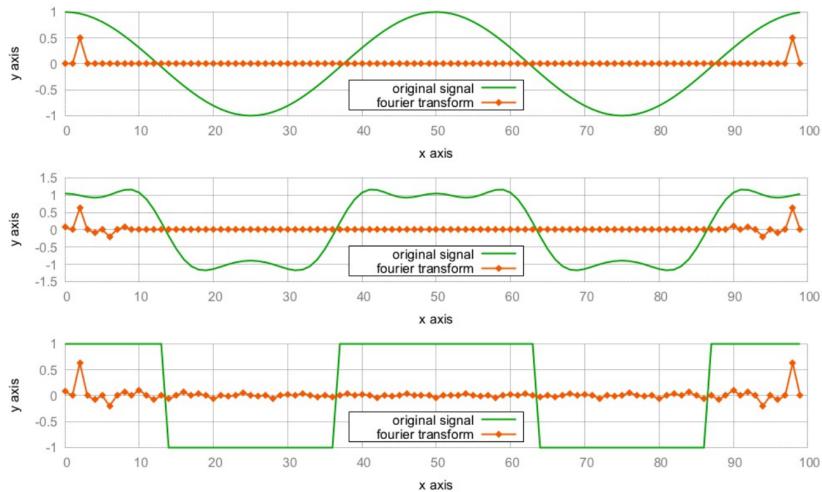
print_signal(cosine);
print_signal(fourier_transform(cosine));

print_signal(mid);
print_signal(trans_sqw);

print_signal(square_wave);
print_signal(fourier_transform(square_wave));
}

```

17. 编译并运行程序，终端上会打印出大量的数据。如果这里使用绘图输出，就可以看到如下的结果：



How it works...

这段代码又两个比较难理解的部分。第一个是傅里叶变换本身，另一个是使用可变Lambda表达式生成信号数据。

首先，我们来看一下傅里叶变换。其核心部分在循环中实现(虽然没有在我们实现中这样做，但可以结合代码看下介绍中的公式)，可能会以如下方式实现：

```

for (size_t k {0}; k < s.size(); ++k) {
    for (size_t j {0}; j < s.size(); ++j) {
        t[k] += s[j] * polar(1.0, pol * k * j /
double(s.size()));
    }
}

```

基于**STL**算法 `std::transform` 和 `std::accumulate`，我们完成了自己的例子，总结一下就类似如下的伪代码：

```

transform(num_iterator{0}, num_iterator{s.size()}, ...
accumulate((num_iterator{0}), num_iterator{s.size()},
...
c + s[k] * polar(1.0, pol * k * j /
double(s.size())));

```

和循环相比，结果完全一样。当然，使用**STL**算法也可以产生不太好的代码。不管怎么样吧，这个实现是不依赖所选用的数据结构。其对于列表也起作用(虽然这没有太大的意义)。另一个好处是，在**C++17**中**STL**很容易并行(将在本书的另一个章节进行介绍)，当需要并行的时候，我们就需要对纯循环进行重构和拆分，将其放入指定的线程中(除非使用类似**OpenMP**这样的并发库，其会自动的将循环进行重构)。

下一个难点是信号生成。让我来看一下另一个 `gen_cosine`：

```

static auto gen_cosine (size_t period_len)
{
    return [period_len, n{0}] () mutable {
        return cos(double(n++) * 2.0 * M_PI /
period_len);
    };
}

```

每一个**Lambda**表达式代表一个函数对象，其会在每次调用时改变自身的状态。其状态包含两个变量 `period_len` 和 `n`。变量 `n`会在每次调用时，进行变更。在不同的时间点上，得到的是不同的信号值，并且在时间增加时会使用 `n++` 对 `n` 的值进行更新。为了获得信号值的数组，我们创建了辅助函数 `signal_from_generator`：

```

template <typename F>
static auto signal_from_generator(size_t len, F gen)
{
    csignal r (len);
    generate(begin(r), end(r), gen);
    return r;
}

```

这个函数会通过所选长度创建一个信号 `vector`，并且调用 `std::generate` 对数据点进行填充。数组r中的每一个元素，都会调用一个 `gen` 函数。`gen` 函数是一种自修改函数对象，我们使用相同的方式创建了 `gen_cosine` 对象。

Note:

本节例子中，STL没有让代码更加的优雅。如果将范围库添加入STL(希望在C++20时加入)，那么可能就会有改观。

计算两个**vector**的误差和

对两个值进行计算的时候，计算机的计算结果与我们期望的结果有一定的差别。比如，测量由多个数据点组成的信号之间的差异，通常会涉及相应数据点的循环和减法等计算。

我们给出一个简单的计算信号**a**与信号**b**之间的误差公式：

$$e = \sum_{i=0}^{N-1} (a_i - b_i)^2$$

对于每一个 `i`，都会计算一次 `a[i] - b[i]`，对差值求平方(负值和正值就能进行比较)，最后计算平方差的和。通常我们会使用循环来做这件事，但是为了让事情更加好玩，我们决定使用**STL**算法来完成。使用**STL**的好处是，无需耦合特定的数据结果。我们的算法能够适应 `vector` 和类似链表的数据结构，不用直接进行索引。

How to do it...

本节，我们将创建两个信号，并计算这两个信号之间的误差：

- 依旧是包含必要的头文件和声明所使用的命名空间。

```
#include <iostream>
#include <cmath>
#include <algorithm>
#include <numeric>
#include <vector>
#include <iterator>

using namespace std;
```

- 我们将对两个信号的误差和进行计算。这两个信号一个是 `sine`，另一个信号也是 `sine`，不过其中之一的使用 `double` 类型进行保存，另一个使用 `int` 类型进行保存。因为 `double` 和 `int` 类型表示数值的范围有差异，就像是模拟信号 `as` 转换成数字信号 `ds`。

```
int main()
{
    const size_t sig_len {100};
    vector<double> as (sig_len); // a for analog
    vector<int> ds (sig_len); // d for digital
```

3. 为了生成一个 `sin` 波形，我们事先了一个简单的Lambda表达式，并可以传入一个可变的计数变量 `n`。我们可以经常在需要的时候调用表达式，其将返回下一个时间点的 `sine` 波形。`std::generate` 可以使用信号值来填充数组，并且使用 `std::copy` 将数组中的 `double` 类型的变量，转换成 `int` 类型变量：

```
auto sin_gen ([n{0}] () mutable {
    return 5.0 * sin(n++ * 2.0 * M_PI / 100);
});
generate(begin(as), end(as), sin_gen);
copy(begin(as), end(as), begin(ds));
```

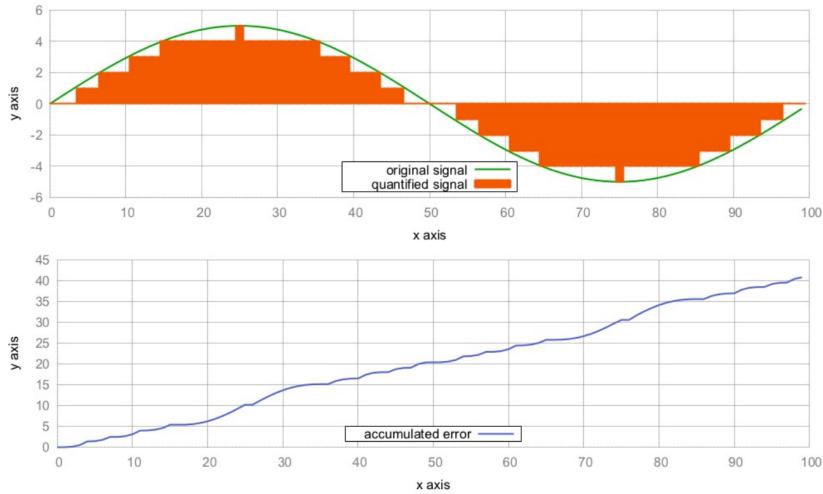
4. 我们可以对信号进行打印，也可以使用绘图进行显示：

```
copy(begin(as), end(as),
      ostream_iterator<double>{cout, " "});
cout << '\n';
copy(begin(ds), end(ds),
      ostream_iterator<double>{cout, " "});
cout << '\n';
```

5. 现在来计算误差和，我们使用 `std::inner_product`，因为这个函数能帮助我们计算两个信号矢量的差异。该函数能在指定范围内进行迭代，然后选择相应位置上进行差值计算，然后在进行平方，再进行相加：

```
cout << inner_product(begin(as), end(as),
begin(ds),
0.0, std::plus<double>{},
[] (double a, double b) {
    return pow(a - b, 2);
})
<< '\n';
}
```

6. 编译并运行程序，我们就能得到两条曲线，还有一条曲线代表的是两个信号的误差和。最终这两个信号的误差为**40.889**。当我们使用连续的方式对误差进行统计，要对值进行逐对匹配，然后得到无法曲线，其就像我们在下图中看到的一样：



How it works...

本节，我们需要将两个向量放入循环中，然后对不同位置的值计算差值，然后差值进行平方，最后使用 `std::inner_product` 将差的平方进行加和。这样，我们可以使用 Lambda 表达式来完成求差值平方的操作——`[](double a, double b){return pow(a - b), 2}`，这样就可以通过传入不同的参数来计算差值平方。

这里我们可以看下 `std::inner_product` 是如何工作的：

```
template<class Init1, class Init2, class T, class F,
class G>
T inner_product(Init1 it1, Init1 end1, Init2 it2, T val,
                 F bin_op1, G bin_op2)
{
    while(it1 != end1) {
        val = bin_op1(val, bin_op2(*it1, *it2));
        ++it1;
        ++it2;
    }
    return value;
}
```

算法会接受一对 `begin/end` 迭代器作为第一个输入范围，另一个 `begin` 迭代器代表第二个输入范围。我们的例子中，这些迭代器所指向的是 `vector`，并对这两个 `vector` 进行误差和的计算。`val` 是一个初始化值。我们这里将其设置为 `0.0`。然后，算法可以接受两个二元函数，分别为 `bin_op1` 和 `bin_op2`。

我们会发现，这个算法与 `std::accumulate` 很相似。不过 `std::accumulate` 只对一个范围进行操作。当将 `bin_op2(*it1, *it2)` 看做一个迭代器，那么我们可以简单的是用 `accumulate` 算法进行计算了。所以，我们可以将 `std::inner_product` 看成是带有打包输入范围的 `std::accumulate`。

例子中，打包函数就是 `pow(a - b, 2)`。因为我们需要将所有元素的差平方进行加和，所以我们选择 `std::plus<double>` 作为 `bin_op1`。

使用ASCII字符曼德尔布罗特集合

1975年，数学家贝诺曼德尔布罗特(Benoit Mandelbrot)创造了一个术语——分形。分形是一个数学图像或者集合，这个术语中包含了很多有趣的数学特性，不过最后看起来分形更像是艺术品。分形图像看起来是无限重复的缩小。其中最为众人所知的分形是曼德尔布罗特(Mandelbrot)集合，其集合看起来就像下图一样：



曼德尔布罗特集合可以通过迭代下面的等式得到：

$$z_0 = 0$$

$$z_{n+1} = z_n^2 + c$$

z 和 c 变量都是复数。曼德尔布罗特集合包含等式所覆盖所有让方程收敛的 c 值，也就是海报彩色的部分。有些值收敛的早，有些值收敛的晚，这里用不同的颜色对这些值进行描述，所以我们能在海报中看到各种不同的颜色。对于那些不收敛的值，我们则直接将其所在区域直接涂黑。

使用STL的 `std::complex` 类，且不使用循环来实现上面的等式。这并不是炫技，只是为了让大家更容易理解STL相关特性的使用方式。

How to do it...

本节，我们将打印类似墙上海报的图，不过是使用ASCII字符将图像打印在终端上：

1. 包含必要的头文件并声明所使用的命名空间：

```

#include <iostream>
#include <algorithm>
#include <iterator>
#include <complex>
#include <numeric>
#include <vector>

using namespace std;

```

2. 曼德尔布罗特集合和之前的等式，都是对复数进行操作。所以，我们需要使用类型别名，使用 `cmplx` 来代表 `std::complex`，并特化为 `double` 类型：

```
using cmplx = complex<double>;
```

3. 我们将使用大约20行的代码来完成一个ASCII组成的曼德尔布罗特集合图像，不过我们会将逻辑逐步实现，最后将所有结果进行组合。第一步就是实现一个函数，用于将整型坐标缩放为浮点坐标。这也就是我们如何在屏幕上特定的位置上打印相应的字符。我们想要的是曼德尔布罗特集合中复数的坐标，就需要实现一个函数，用于将对应的坐标转换成相应的几何图形。用一个Lambda表达式来构建这些变量，并将其返回。该函数能将 `int` 类型的函数转换成一个 `double` 类型的函数：

```

static auto scaler(int min_from, int max_from,
double min_to, double max_to)
{
    const int w_from {max_from - min_from};
    const double w_to {max_to - min_to};
    const int mid_from {(max_from - min_from) / 2 +
min_from};
    const double mid_to {(max_to - min_to) / 2.0 +
min_to};

    return [=] (int from) {
        return double(from - mid_from) / w_from *
w_to + mid_to;
    };
}

```

4. 现在需要在一个维度上进行坐标变换，不过曼德尔布罗特集合使用的是二维坐标系。为了能将(x, y)坐标系统转换成另一个，我们需要将 `x-scaler` 和 `y-scaler` 相结合，并且构建一个 `cmplx` 实例作为输出：

```

template <typename A, typename B>
static auto scaled_cmplx(A scaler_x, B scaler_y)
{
    return [=] (int x, int y) {
        return cmplx(scaler_x(x), scaler_y(y));
    };
}

```

5. 将坐标转换到正确的维度上后，就可以来实现曼德尔布罗特方程。现在不管怎么打印输出，一心只关注于实现方程即可。循环中，对 z 进行平方，然后加上 c ，知道 abs 的值小于2。对于一些坐标来说，其值永远不可能比2小，所以当循环次数达到 `max_iterations` 时，我们就决定放弃。最后，将会返回那些 abs 值收敛的迭代次数：

```

static auto mandelbrot_iterations(cmplx c)
{
    cmplx z {};
    size_t iterations {0};
    const size_t max_iterations {1000};
    while (abs(z) < 2 && iterations < max_iterations)
    {
        ++iterations;
        z = pow(z, 2) + c;
    }
    return iterations;
}

```

6. 那么现在我们就来实现主函数。在主函数中我们会定义缩放函数对象 `scale`，用于对坐标值进行多维变换：

```

int main()
{
    const size_t w {100};
    const size_t h {40};

    auto scale (scaled_cmplx(
        scaler(0, w, -2.0, 1.0),
        scaler(0, h, -1.0, 1.0)
    ));
}

```

7. 为了可以在一维上迭代器整个图形，需要完成另一个转换函数，用于将二维图像进行降维操作。其会根据我们所设置的字符宽度进行计算。其会将一维上的长度进行折断，然后进行多行显示，通过使用 `scale` 函数对坐标进行变换，然后返回复数坐标：

```
    auto i_to_xy ([=] (int i) { return scale(i % w, i / w); });
```

8. 我们将图像的二维坐标(int, int类型)转换为一维坐标(int类型), 再将坐标转换成曼德尔布罗特结合坐标(cmplx类型)。让我们将所有功能放入一个函数, 我们将使用一组调用链:

```
auto to_iteration_count ([=] (int i) {
    return mandelbrot_iterations(i_to_xy(i));
});
```

9. 现在我们可以来设置所有数据。假设我们的结果ASCII图像的字符宽度为 `w`, 高度为 `h`。这样就能将结果存储在一个长度为 `w * h` 数组中。我们使用 `std::iota` 将数值范围进行填充。这些数字可以用来作为转换的输入源, 我们将变换过程包装在 `to_iteration_count` 中:

```
vector<int> v (w * h);
iota(begin(v), end(v), 0);
transform(begin(v), end(v), begin(v),
to_iteration_count);
```

10. 现有一个`v`数组, 其使用一维坐标进行初始化, 不过后来会被曼德尔布罗特迭代计数所覆盖。因此, 我们就可以对图像进行打印。可以将终端窗口设置为 `w` 个字符宽度, 这样我们就不需要打印换行符。不过, 可能会有对 `std::accumulate` 有一种创造性的误用。`std::accumulate` 使用二元函数对处理范围进行缩小。我们可以对其提供一个二元函数, 其能接受一个输出迭代器(并且我们将在下一步进行终端打印), 并使用范围内的单个值进行计算。如果相应值的迭代次数大于50次时, 我们会打印 * 字符到屏幕上。否则, 会打印空字符串在屏幕上。在每行结束时(因为计数器变量`n`可被`W`均匀地分割), 我们会打印一个换行符:

```
auto binfunc ([w, n{0}] (auto output_it, int x)
mutable {
    ++output_it = (x > 50 ? '*' : ' ');
    if (++n % w == 0) { ++output_it = '\n'; }
    return output_it;
});
```

11. 通过对输入范围使用 `std::accumulate`, 我们将二元打印函数和 `ostream_iterator` 相结合, 我们需要在屏幕上刷新计算出的曼德尔布罗特集合:

```
accumulate(begin(v), end(v),
ostream_iterator<char>{cout},
binfunc);
}
```

12. 编译并运行程序，就可以看到如下的输出，其看起来和墙上的海报很像吧！

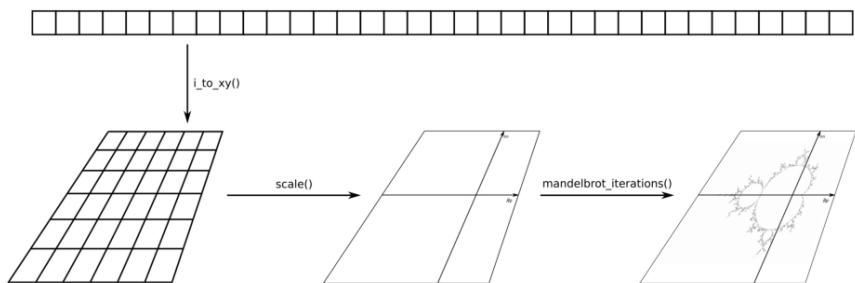


How it works...

整个计算过程都使用 `std::transform` 对一维数组进行处理：

```
vector<int> v (w * h);
iota(begin(v), end(v), 0);
transform(begin(v), end(v), begin(v),
to_iteration_count);
```

所以，会发生什么呢？我们为什么要这么做？`to_iteration_count` 函数是基于从 `i_to_xy` 开始的调用链，从 `scale` 到 `mandelbrot_iterations`。下面的图像就能展示我们的转换步骤：



这样，我们就可以使用一维数组作为输入，并且获得曼德尔布罗特方程的迭代次数（使用一维坐标表示的二维坐标上的值）。三个互不相关的转换是件好事。这样代码就可以独立的进行测试，这样就不用互相牵制了。同样，这样更容易进行正确性测试，并寻找并修复bug。

实现分割算法

很多情况下，**STL**中的算法并不够我们使用，有些算法需要我们自己去实现。解决具体问题之前，我们需要确定，这个问题是否有通解。当我们自己遇到一些问题时，我们可以实现一些辅助函数帮助我们，这些辅助函数逐渐的就可以形成库。这里关键是要明白什么样的代码是足够通用的，否则我们就需要创造一套通用语言了。

本节我们将实现一个算法，叫做分割(split)。该算法可以通过给定的值，对任何范围的元素进行分割，将分割后的结果块拷贝到输出区域中。

How to do it...

本节，将实现类似于**STL**的算法叫做分割，并且用这个算法对字符串进行分割：

1. 首先，包含必要的头文件，并声明相应的命名空间。

```
#include <iostream>
#include <string>
#include <algorithm>
#include <iterator>
#include <list>

using namespace std;
```

2. 本节的所有算法都围绕分割来进行。其接受一对 `begin/end` 迭代器和一个输出迭代器，其用法和 `std::copy` 或 `std::transform` 类似。其他参数为 `split_val` 和 `bin_func`。`split_val` 参数是要在输入范围内要查找的值，其表示要当碰到这个值时，要对范围进行分割。`bin_func` 参数是一个函数，其为分割的子序列的开始和结尾。我们可以使用 `std::find` 对输入范围进行迭代查找，这样就能直接跳转到 `split_val` 所在的位置。当将一个长字符串分割成多个单词，可以通过分割空格字符达到目的。对于每一个分割值，都会做相应的分割，并将对应的分割块拷贝到输出范围内：

```

template <typename InIt, typename OutIt, typename T,
typename F>
InIt split(InIt it, InIt end_it, OutIt out_it, T
split_val,
           F bin_func)
{
    while (it != end_it) {
        auto slice_end (find(it, end_it, split_val));
        *out_it++ = bin_func(it, slice_end);

        if (slice_end == end_it) { return end_it; }
        it = next(slice_end);
    }
    return it;
}

```

3. 现在尝试一下我们的新算法，构建一个需要进行分割的字符串。其中的字符使用 - 进行连接：

```

int main()
{
    const string s {"a-b-c-d-e-f-g"};

```

4. 创建一个 `bin_func` 对象，其能接受一组迭代器，我们需要通过该函数创建一个新的字符串：

```

auto binfunc ([](auto it_a, auto it_b) {
    return string(it_a, it_b);
});

```

5. 输出的子序列将保存在 `std::list` 中。我们现在可以调用 `split` 算法：

```

list<string> l;
split(begin(s), end(s), back_inserter(l), '-',
binfunc);

```

6. 为了看一下结果，我们将对子字符串进行打印：

```

copy(begin(l), end(l), ostream_iterator<string>
{cout, "\n"});

```

7. 编译并运行程序，就可以看到如下输出。其子序列将不会包含破折号，只有单个单词(在我们的例子中，为单个字母)：

```
$ ./split
a
b
c
d
e
f
g
```

How it works...

`split` 算法与 `std::transform` 的工作原理很类似，因为其能接受一对 `begin/end` 迭代器和一个输出迭代器。其也会将最终的算法结果拷贝到输出迭代器所在的容器。除此之外，其接受一个 `split_val` 值和一个二元函数。让我们再来看一起其整体实现：

```
template <typename InIt, typename OutIt, typename T,
          typename F>
InIt split(InIt it, InIt end_it, OutIt out_it, T
           split_val, F bin_func)
{
    while (it != end_it) {
        auto slice_end (find(it, end_it, split_val));
        *out_it++ = bin_func(it, slice_end);

        if (slice_end == end_it) { return end_it; }
        it = next(slice_end);
    }
    return it;
}
```

实现中的循环会一直进行到输入范围结束。每次迭代中都会调用 `std::find` 用来在输入范围内查找下一个与 `split_val` 匹配的元素。在我们的例子中，分割字符就是 `-`。每次的下一个减号字符的位置会存在 `slice_end`。每次循环迭代之后，`it` 迭代器将会更新到下一个分割字符所在的位置。循环起始范围将从一个减号跳到下一个减号，而非每一个独立的元素。

这一系列的操作中，迭代器 `it` 指向的是最后子字符串的起始位置，`slice_end` 指向的是子字符串的末尾位置。通过这两个迭代器，就能表示分割后的子字符串。对于字符串 `foo-bar-baz` 来说，循环中就有三个迭代器。对于用户而言，迭代器什么的并不重要，他们想要的是子字符串，所以这里就是 `bin_func` 来完成这个任务。当我们调用 `split` 时，我们可以给定其一个如下的二元函数：

```
[] (auto it_a, auto it_b) {
    return string(it_a, it_b);
}
```

`split` 函数会将迭代器传递给 `bin_func`，并通过迭代器将结果放入输出迭代器中。这样我们就能通过 `bin_func` 获得相应的单词，这里的结果是 `foo`，`bar` 和 `baz`。

There's more...

我们也可以实现相应的迭代器来完成这个算法的实现。我们现在不会去实现这样一个迭代器，但是可以简单的看一下。

迭代的每次增长，都会跳转到下一个限定符。

当对迭代器进行解引用时，其会通过迭代器指向的当前位置，创建一个字符串对象，就如同之前用到的 `bin_func` 函数那样。

迭代器类可以称为 `split_iterator`，用来替代算法 `split`，用户的代码可以写成如下的样式：

```
string s {"a-b-c-d-e-f-g"};
list<string> l;

auto binfunc ([](auto it_a, auto it_b) {
    return string(it_a, it_b);
});

copy(split_iterator{begin(s), end(s), " - ", binfunc}, {},
back_inserter(l));
```

虽然在使用中很方便，但是在实现时，迭代器的方式要比算法的形式复杂许多。并且，迭代器实现中很多边缘值会触发代码的bug，并且迭代器实现需要经过非常庞大的测试。不过，其与其他STL算法能够很好的兼容。

将标准算法进行组合

`gather` 为STL算法中最好的组合性例子。Sean Parent在任Adobe系统首席科学家时，就在向世人普及这个算法，因为其本身短小精悍。其使用方式就如同做一件艺术品一样。

`gather` 算法能操作任意的元素类型。其更改元素的顺序，通过用户的选择，其会将对应的数据放置在对应位置上。

How to do it...

本节，我们来实现 `gather` 算法，并对其进行一定的修改。最后，将展示如何进行使用：

1. 包含必要的头文件，声明所使用的命名空间。

```
#include <iostream>
#include <algorithm>
#include <string>
#include <functional>

using namespace std;
```

2. `gather` 算法是表现标准算法组合性很好的一个例子。`gather` 接受一对 `begin/end` 迭代器和另一个迭代器 `gather_pos`，其指向 `begin` 和 `end` 迭代器的中间位置。最后一个参数是一个谓词函数。使用谓词函数，算法会将满足谓词条件的所有元素放置在 `gather_pos` 迭代器附近。使用 `std::stable_partition` 来完成移动元素的任务。`gather` 算法将会返回一对迭代器。这两个迭代器由 `stable_partition` 所返回，其表示汇集范围的起始点和结束点：

```
template <typename It, typename F>
pair<It, It> gather(It first, It last, It gather_pos,
F predicate)
{
    return {stable_partition(first, gather_pos,
not_fn(predicate)),
            stable_partition(gather_pos, last,
predicate)};
```

3. 算法的另一种变体为 `gather_sort`。其工作原理与 `gather` 相同，不过其不接受一元谓词函数；其能接受一个二元比较函数。这样，其就也能将对应值汇集在 `gather_pos` 附近，并且能知道其中最大值和最小值：

```

template <typename It, typename F>
void gather_sort(It first, It last, It gather_pos, F
comp_func)
{
    auto inv_comp_func ([&] (const auto &...ps) {
        return !comp_func(ps...);
    });

    stable_sort(first, gather_pos, inv_comp_func);
    stable_sort(gather_pos, last, comp_func);
}

```

4. 让我们来使用一下这些算法。先创建一个谓词函数，其会告诉我们当前的字母是不是 `a`，再构建一个字符串，仅包含 `a` 和 `-` 字符：

```

int main()
{
    auto is_a ([](char c) { return c == 'a'; });
    string a {"a_a_a_a_a_a_a_a_a_a_a"};

```

5. 继续构造一个迭代器，让其指向字符串的中间位置。这时可以调用 `gather` 算法，然后看看会发生什么。`a` 字符将汇集在字符串中间的某个位置附近：

```

auto middle (begin(a) + a.size() / 2);

gather(begin(a), end(a), middle, is_a);
cout << a << '\n';

```

6. 再次调用 `gather`，不过这次 `gather_pos` 的位置在字符串的起始端：

```

gather(begin(a), end(a), begin(a), is_a);
cout << a << '\n';

```

7. 再将 `gather_pos` 的位置放在末尾试试：

```

gather(begin(a), end(a), end(a), is_a);
cout << a << '\n';

```

8. 最后一次，这次将再次将迭代器指向中间位置。这次与我们的期望不相符，后面我们来看看发生了什么：

```

// This will NOT work as naively expected
gather(begin(a), end(a), middle, is_a);
cout << a << '\n';

```

9. 再构造另一个字符串，使用下划线和一些数字组成。对于这个输入队列，我们使用 `gather_sort`。`gather_pos` 迭代器指向字符串的中间，并且比较函数为 `std::less<char>`：

```
string b {"_9_2_4_7_3_8_1_6_5_0_"};
gather_sort(begin(b), end(b), begin(b) + b.size()
/ 2,
less<char>{});
cout << b << '\n';
}
```

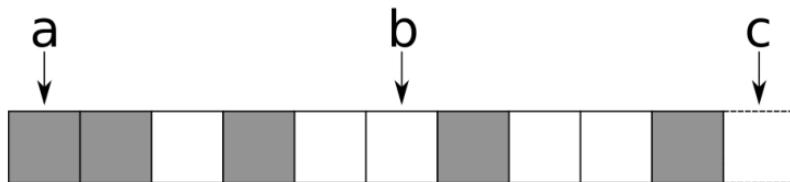
10. 编译并运行程序，就会看到如下的输出。对于前三行来说和预期一样，不过第四行貌似出现了一些问题。最后一行中，我们可以看到 `gather_short` 函数的结果。数字的顺序是排过序的(中间小，两边大)：

```
$ ./gather
_____aaaaaaaa_____
aaaaaaaaaa_____
_____aaaaaaaaaa
_____aaaaaaaaaa
_____ 9743201568 _____
```

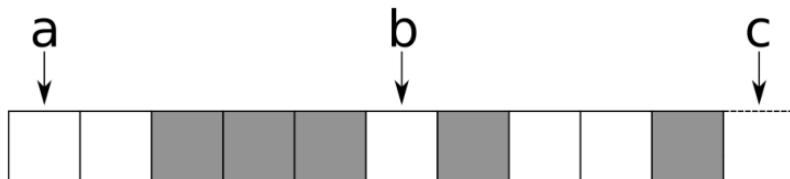
How it works...

`gather` 算法有点难以掌握，因为其非常短，但处理的是比较复杂的问题。让我们来逐步解析这个算法：

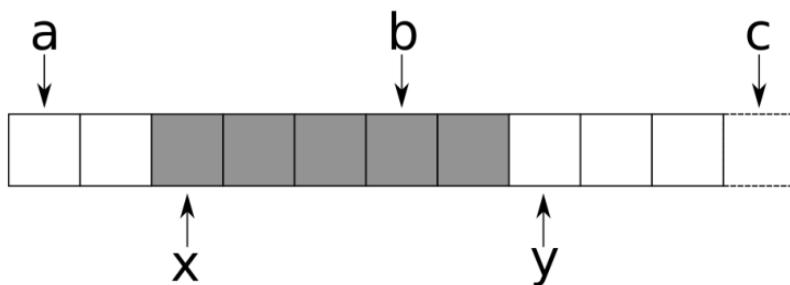
1.) initial state



2.) `stable_partition(a, b, !predicate);`



3.) `stable_partition(b, c, predicate);`



4.) `return {x, y};`

1. 初始化相应元素，并提供一个谓词函数。图中满足谓词函数条件的元素为灰色，其他的为白色。迭代器 a 和 c 表示整个范围的长度，并且迭代器 b 指向了最中间的元素。这就表示要将所有灰色的格子聚集在这个迭代器附近。
2. `gather` 算法会对范围 `(a, b]` 调用 `std::stable_partition`，并对另一边使用不满足谓词条件的结果。这样就能将所有灰色格子集中在**b**迭代器附近。
3. 另一个 `std::stable_partition` 已经完成，不过在 `[b, c)` 间我们将使用满足谓词函数的结果。这样就将灰色的格子汇集在**b**迭代器附近。
4. 所有灰色的格子都汇集在 `b` 迭代器附近，这是就可以返回起始迭代器 `x` 和末尾迭代器 `y`，用来表示所有连续灰色格子的范围。

我们对同一个范围多次调用 `gather` 算法。最初，将所有元素放在范围中间位置。然后，尝试放在开始和末尾。这种实验很有趣，因为这会让其中一个 `std::stable_partition` 无元素可以处理。

最后一次对 `gather` 进行调用时，参数为 `(begin, end, middle)`，但没有和我们预期的一样，这是为什么呢？这看起来像是一个bug，实际上不是。

试想一个字符组 `aabb`，使用谓词函数 `is_character_a`，用来判断元素是否为 `a`，当我们将第三个迭代器指向字符范围的中间时，会复现这个bug。原因：第一个 `stable_partition` 调用会对子范围 `aa` 进行操作，并且另一个 `stable_partition` 会对子范围 `bb` 上进行操作。这种串行的调用时无法得到 `baab` 的，其结果看起来和开始一样，没有任何变化。

Note:

要是想得到我们预期的序列，我们可以使用 `std::rotate(begin, begin + 1, end);`

`gather_sort` 基本上和 `gather` 差不多。签名不同的就是在谓词函数。实现的不同在于 `gather` 调用了两次 `std::stable_partition`，而 `gather_sort` 调用了两次 `std::stable_sort`。

这是由于 `not_fn` 不能作用域二元函数，所以反向比较不能由 `not_fn` 完成。

删除词组间连续的空格

我们会经常从输入中读取字符串，这些字符串会包含一些原生格式，需要进行清洗。其中一个例子就是字符串中包含了太多的空格。

本节，我们将实现一个聪明的空格滤波算法，其会删除多余的空格，会给单词间留下一个空格。我们可以将这个算法称为 `remove_multi_whitespace`，并且接口与STL很像。

How to do it...

本节，我们将实现过滤空格的算法，并了解其是如何进行工作的：

1. 包含必要的头文件和声明所使用的命名空间：

```
#include <iostream>
#include <string>
#include <algorithm>

using namespace std;
```

2. `remove_multi_whitespace` 看起来与STL的风格非常类似。这个算法会移除多余的空格，只保留一个空格。当字符串为 `a b`，算法是不会进行任何操作的；当字符串为 `a b` 时，算法会返回 `a b`。为了完成这个算法，我们使用 `std::unique` 通过对一段区域的迭代，用来查找一对连续的元素。然后，通过谓词函数进行判断，确定两个元素是否相等。如果相等，那么 `std::unique` 会将其中一个移除。这样，子范围内就不会存在相等的元素了。谓词函数会通过读取到的内容来判断二者是否相等。我们需要给 `std::unique` 怎么样一个谓词函数呢？其需要判断两个元素是否是连续的空格；如果是，就要移除一个空格。与 `std::unique` 类似，也需要传入一对 `begin/end` 迭代器，然后返回的迭代器将返回新范围的末尾迭代器：

```
template <typename It>
It remove_multi_whitespace(It it, It end_it)
{
    return unique(it, end_it, [](const auto &a, const
auto &b) {
        return isspace(a) && isspace(b);
    });
}
```

3. 万事俱备，就来进行测试，尝试使用算法将不必要的空格进行删除：

```

int main()
{
    string s {"foo bar \t baz"};
    cout << s << '\n';
}

```

4. 对字符串使用过滤算法，去掉多余的空格：

```

s.erase(remove_multispace(begin(s),
end(s)), end(s));

cout << s << '\n';
}

```

5. 编译并运行程序，就会得到如下的输出：

```

$ ./remove_consecutive_whitespace
foo bar      baz
foo bar baz

```

How it works...

整个问题的解决中，我们没有使用循环或者元素间的互相比较。我们只使用谓词函数来完成判断两个给定字符是否是空格的任务。然后，将谓词函数与 `std::unique` 相结合，所有多余的空格就都消失了。本章中有些算法可能会有些争议，不过这个算法的确算的上短小精悍的典范了。

我们如何在将算法进行组合的呢？我们来看一下 `std::unique` 可能的实现代码：

```

template<typename It, typename P>
It unique(It it, It end, P p)
{
    if (it == end) { return end; }

    It result {it};
    while (++it != end) {
        if (!p(*result, *it) && ++result != it) {
            *result = std::move(*it);
        }
    }
    return ++result;
}

```

其中循环会迭代到范围的最后，当元素满足谓词条件，就会从原始位置上移除一个元素。这个版本的 `std::unique` 不接受多余的谓词函数，来判断两个相邻的元素是否相等。这样的话，只能将重复的字符去除，比如会将 `aaaaaaaa` 变换成 `abc`。

那么，我们应该怎么做才能不去除除了空格之外的重复的元素呢？这样，谓词函数不能告诉程序“两个输入字符是相同的”，而是要说“两个输入字符都是空格”。

最后需要注意的是，无论是 `std::unique` 还是 `remove_multi whitespace` 都会从字符串中移除字母元素。根据字符串的语义来移动字符串，并表明新的结尾在哪里。新的尾部到旧的尾部的元素依旧存在，所以我们必须将它们删除：

```
s.erase(remove_multi_whitespace(begin(s), end(s)),
end(s));
```

和 `vector` 和 `list` 一样，`erase` 成员函数其会对元素进行擦除和删除。

压缩和解压缩字符串

压缩问题在编程面试中出现的相对较多。就是使用一个函数将 `aaaaabbbbbccc` 字符串转换成一个短字符串 `a5b7c3`。`a5` 表示原始字符串中有5个`a`, `b7` 表示原始字符串中有7个`b`。这就一个相对简单的压缩算法。对于普通的文本，并不需要使用这个算法，因为文本中重复的东西很少，不需要进行压缩。不过，这套算法就算没有计算机，我们也很容易的对其进行实现。如果代码一开始没有进行很好的设计，那么就很容易出现bug。虽然，处理字符串并不是一件很困难的事情，但是代码中大量使用C风格的字符串时，很有可能遇到缓冲区溢出的bug。

本节让我们使用STL来对字符压缩和解压进行实现。

How to do it...

本节，我们将对字符串实现简单的 `compress` 和 `decompress` 函数：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <string>
#include <algorithm>
#include <sstream>
#include <tuple>

using namespace std;
```

2. 对于我们的压缩算法，我们会尝试去找到文本中连续相同的字符，并且对他们进行单独的进行压缩处理。当我们拿到一个字符串，我们需要知道与第一个字符不同的字符在哪里。这里使用 `std::find` 来寻找与第一个位置上的元素不同的元素位置。先将起始位置的字符赋予 `c`。经过查找后就会返回一个迭代器，其指向第一个不同的元素。两个不同字符间的距离，会放到元组中返回：

```
template <typename It>
tuple<It, char, size_t> occurrences(It it, It end_it)
{
    if (it == end_it) { return {it, '?', 0}; }

    const char c {*it};
    const auto diff (find_if(it, end_it,
                           [c](char x) { return c != x; }));
    return {diff, c, distance(it, diff)};
}
```

3. `compress` 会连续的对 `occurrences` 函数进行调用。这样，就能从同一个字符组，跳转到另一个。`r << c << n` 行表示将字符 `c` 推入到输出流中，并且将 `occurrences` 函数的调用次数作为结果字符串的一部分。最后会返回一个字符串对象，就包含了压缩过的字符串：

```
string compress(const string &s)
{
    const auto end_it (end(s));
    stringstream r;

    for (auto it (begin(s)); it != end_it;) {
        const auto [next_diff, c, n] (occurrences(it,
end_it));
        r << c << n;
        it = next_diff;
    }

    return r.str();
}
```

4. `decompress` 的原理也不复杂，但会更简短。其会持续的从输入流中获取字符，字符串包括字符和数字。对于这两种值，函数会构造一个字符串用于解压所获取到的字符串。最后，会再次返回一个字符串。顺带一提，这里的 `decompress` 函数是不安全的。其很容易被破解。我们会在后面来看下这个问题：

```
string decompress(const string &s)
{
    stringstream ss{s};
    stringstream r;

    char c;
    size_t n;

    while (ss >> c >> n) { r << string(n, c); }
    return r.str();
}
```

5. 主函数中会构造一个简单的字符串，里面有很多重复的字符。打印压缩过后，和解压过后的字符串。最后，我们应该会得到原始的字符串：

```
int main()
{
    string s {"aaaaaaaaabbbbbbbccc";
    cout << compress(s) << '\n';
    cout << decompress(compress(s)) << '\n';
}
```

6. 编译并运行程序，我们就会得到如下的输出：

```
$ ./compress  
a9b9c11  
aaaaaaaaabbbbbbbccc
```

How it works...

这里我们使用两个函数 `compress` 和 `decompress` 来解决这个问题。

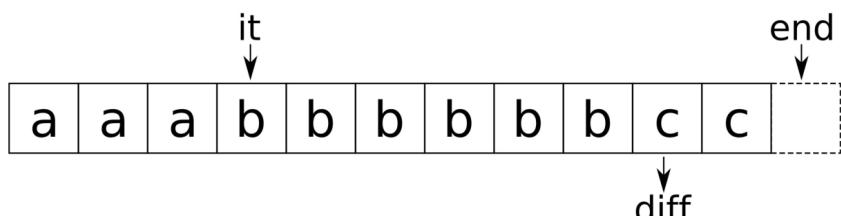
解压函数这里实现的十分简单，因为其就包含一些变量的声明，其主要工作的代码其实只有一行：

```
while (ss >> c >> n) { r << string(n, c); }
```

其能持续将字符读取到 `c` 当中，并且将数字变量读取到 `n` 中，然后输出到 `r` 中。`stringstream` 类在这里会隐藏对字符串解析的细节。当成功进行解压后，解压的字符串将输入到字符流中，这也就是 `decompress` 最后的结果。如果 `c = 'a'` 并且 `n = 5`，那么 `string(n, c)` 的字符串为 `aaaaa`。

压缩函数比较复杂，我们为其编写了一个小的辅助函数。这个辅助函数就是 `occurrences`。那么我们就先来看一下 `occurrences` 函数。下面的图展示了 `occurrences` 函数工作的方式：

1.) occurrences(it, end);



2.) return {diff, 'b', 6};

`occurrences` 函数能够接受两个参数：指向字符序列起始点的迭代器和末尾点的迭代器。使用 `find_if` 能找到第一个与起始点字符不同的字符的位置，也就是图中的 `diff` 迭代器的位置。起始位置与 `diff` 位置之间元素就与起始字符相同，图中相同的字符有6个。在我们计算出这些信息后，`diff` 迭代器就可以在下次查询时，进行重复利用。因此，我们将 `diff`、子序列范围和子序列范围的长度包装在一个元组中进行返回。

根据这些信息，我们就能在子序列之间切换，并且将结果推入到目标字符串中：

```
for (auto it (begin(s)); it != end_it;) {
    const auto [next_diff, c, n] (occurrences(it,
end_it));
    r << c << n;
    it = next_diff;
}
```

There's more...

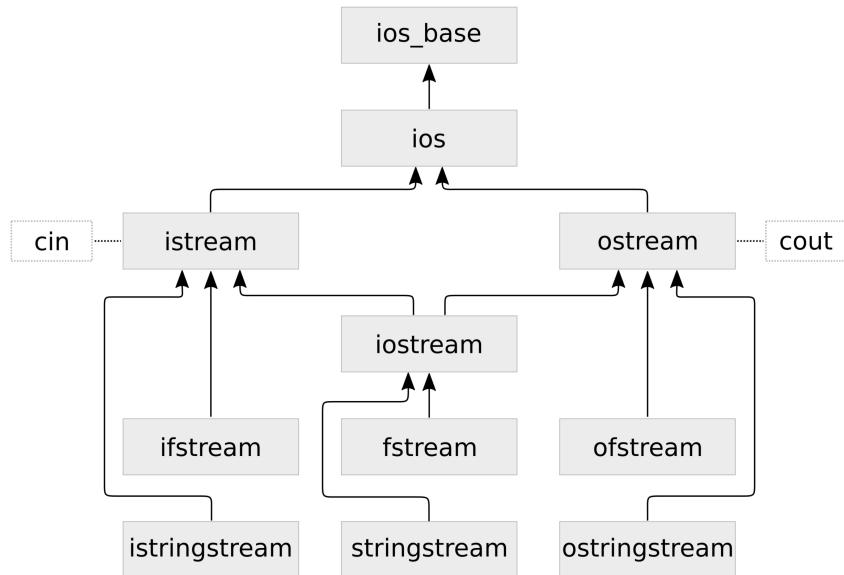
还记得在第4步的时候，我们说过 `decompress` 不安全吗？这个函数确实容易被利用。

试想我们传入一个字符串：`a00000`。压缩的第一个结果为 `a1` 因为其只包含了一个字母 `a`。然后，对后面5个0进行处理，结果为 `05`。然后将两个结果合并，那么结果就为 `a105`。不幸的是，外部对这个字符串的解读是“`a`连续出现了105次”。我们的输入字符串并没有什么错。这里最糟糕的情况就是，我们将这个字符串进行了压缩，然后我们通过输入的六个字符得到了一个长度为105的字符串。试想当用户得到了这样的结果会不会感到愤怒？因为我们的算法并没有准备好应对这样的输入。

为了避免这样的事情发生，我们只能在 `compress` 函数中禁止数字的输入，或者将数字使用其他的方式进行处理。之后，`decompress` 算法需要加入一个条件，就是需要固定输出字符串的最大长度。这个就当做作业，交由读者自行完成。

第7章 字符串, 流和正则表达

本章, 我们将对字符串的处理进行介绍, 其中包括处理、解析和打印任意的数据。对于这样的需求, STL提供了I/O流库进行支持。这些库由以下的类组成, 对应的类使用灰色框表示:



箭头的指向代表了每个类之间的继承关系。这里面类的数量还挺多, 本章中, 我们会逐个来熟悉。我们使用类型名称在STL的手册中进行查找时, 不一定能直接找到这些类。因为图中的这些名字是对于应用开发者来说的, 其中大多数类型的名称都是以 `basic_` 为前缀(例如: 我们能很容易的在STL文档中找到 `basic_istream`, 而 `istream` 却很难找到)。以 `basic_` 为前缀的I/O流类型为模板类, 可以将其特化成不同的字符类型。图中的类型都可以以 `char` 类型进行特化。我们将会在本书剩下的章节中, 使用以 `char` 为特化的版本。当我们看到类型名前面以 `w` 开头时(例如: `wistream`, `wostream`), 将使用 `wchar_t` 类型代替 `char` 类型。

图的最顶端, 能看到 `std::ios_base` 类。我们不能直接对其进行使用, 不过其他的类型都是其子类。其一种特化为 `std::ios`, 这个类型对象会包含流数据, 其能通过 `good` 成员函数对流的状态进行查询, 还能通过 `empty` 成员函数数据状态是否为(EOF)。

我们经常使用的特化类有两个: `std::istream` 和 `std::ostream`。两个类型的前缀 `i` 和 `o` 代表着输入和输出。我们在之前的代码使用使用其 `std::cout` 和 `std::cin` (还有 `std::cerr`) 对象对字符串进行过输入和输出。其都是这些类型的实例, 也是非常通用的。我们通过 `ostream` 进行数据输出, 使用 `istream` 进行数据输入。

`iostream` 类则是对 `istream` 和 `ostream` 的继承, 其将输入和输出的能力进行合并。当我们对流数据进行输入和输出的时候, 我们就有三个类可供使用。

`ifstream`, `ofstream` 和 `fstream` 继承于 `istream`, `ostream` 和 `iostream`, 不过为I/O流添加了文件的读入写出功能。

`istringstream`, `ostringstream` 和 `iostreamstream` 原理十分类似, 会将字符串读入内存中, 并在内存中对数据进行处理。

创建、连接和转换字符串

熟悉C++的“老人”们对 `std::string` 一定不会陌生。在处理C风格的字符串时，会感觉冗余和痛苦，特别是在于解析、连接和复制这些字符串的时候，而使用 `std::string` 确实一种简单安全的方法。

要特别感谢C++11添加了移动的特性，这样我们就可以对字符串的所有权进行转移。这样，很多情况下的开销就能降的很低。

`std::string` 也随着标准的更新添加了新的特性。C++17中添加了一个全新的类——`std::string_view`。本节我们将在C++17下感受一下这些新特性(将在其他节中使用 `std::string_view` 新类，来连接多个字符串)。

How to do it...

本节，将创建几个字符串和几个字符串代理，并使用它们进行对字符串的连接和转换：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <string>
#include <string_view>
#include <sstream>
#include <algorithm>

using namespace std;
using namespace std::literals;
```

2. 首先来创建字符串对象，这里 `a` 就为一个 `string` 对象。我们使用C风格的字符串对其进行构造(编译后，C风格的字符串就成为静态数组)。构造函数将对其进行拷贝，然后构成一个字符串对象。或者也可直接使用字符字面值操作符 `"s"` 来代替C风格的字符串。其也会在运行时创建一个字符串对象，这里 `b` 也是一个字符串对象，不过这里我们让程序自己去推断这个类型：

```
int main()
{
    string a { "a" };
    auto b ( "b"s );
```

3. 构造字符串对象的时候，会将相应的内容拷贝到字符串的内部内存中。为了不拷贝，可以直接对输入字符串进行引用，这里就用一下 `string_view`。这个类具有一个字面值操作，称为 `"sv"`：

```
string_view c { "c" };
auto d ( "d"sv );
```

4. OK! 现在就让我们来用一下字符串和代理字符串吧！对于这两种类型，其 `operator<<` 都是对 `std::ostream` 类型重载的类型，所以这两种类型可以直接打印：

```
cout << a << ", " << b << '\n';
cout << c << ", " << d << '\n';
```

5. 字符串类也对 `operator+` 操作进行了重载，所以可以直接将两个字符串进行连接。连接 `a` 和 `b` 只要使用 `+` 操作就能完成。对于 `a` 和 `c` 来说，就没有那么简单了，因为 `c` 不是一个字符串对象，而是一个字符串代理对象。我们必须先获取 `c` 的字符串，这步可以通过对 `c` 进行新的字符串构造得到，然后再和 `a` 进行相加。这里就有一个问题，“等下！只是为了和 `a` 相加，我们就要将 `c` 的内容拷贝到临时字符串对象中吗？没有避免使用 `c.data()` 进行内容拷贝的方法吗？”这个想法很好，但是类型本身具有缺陷——`string_view` 实例中没有终止符。这个很可能导致缓存溢出：

```
cout << a + b << '\n';
cout << a + string{c} << '\n';
```

6. 我们来创建一个新的字符串，其包含我们之前创建的所有字符串和字符串代理。使用 `std::ostringstream`，我们能将任意的变量通过流对象进行打印(类似 `std::cout`)，不过其不会显示在终端，而是打印到一个字符串缓存中。对于所有的字符串，我们是用空格对这些字符串进行分割，并使用 `operator<<` 将这些字符串打印到新的字符串对象中(使用 `o.str()`)：

```
ostringstream o;

o << a << " " << b << " " << c << " " << d;
auto concatenated = o.str();
cout << concatenated << '\n';
```

7. 这时我们可以通过相应的函数，将新字符串中所有的字符转换成大写字符。这里使用C库中的 `toupper` 函数来完成将字母转换为大写的工作，可将其与 `std::transform` 相结合。因为这里字符串的基础类型为 `char`，所以可以直接使用：

```
transform(begin(concatenated), end(concatenated),
          begin(concatenated), ::toupper);
cout << concatenated << '\n';
}
```

8. 编译并运行程序，将会得到如下输出：

```
$ ./creating_strings
a, b
c, d
ab
ac
a b c d
A B C D
```

How it works...

显然，字符串可以通过加法操作进行连接。如果要对 `string_view` 使用这个特性，我们首先需要将其转化为 `std::string`。

不过，进行字符串和字符串代理编码时要格外注意，`string_view` 的内容中没有终止符！这也就是为什么我们宁愿写成 `"abc"s + string{some_string_view}`，而不写成 `"abc"s + some_string_view.data()` 的原因。除此之外，`std::string` 也提供了 `append` 成员函数，其能对 `string_view` 实例进行处理，不过其会对字符串的内容直接进行操作。

Note:

`std::string_view` 是非常有用的，不过为了将其与字符串和字符串函数相混合。我们不能假设其具有终止符，其会在标准字符串环境中快速的跳出。幸运的是，通常一些函数的重载版本，可以对其进行正确的处理。

如果我们想要将更为复杂的字符串进行格式化连接，不需要对字符串实例进行逐个处理。`std::stringstream`，`std::ostringstream` 和 `std::istringstream` 类就适合来处理这种任务，它们能对通过对内存的管理来进行字符串的添加，并且能提供流所具有的所有通用格式化特性。这也就是本节为什么选择 `std::ostringstream` 类的原因，其可以很方便的对变量类型进行解析，然后将其放入字符串中。如果想将输入输出进行结合，那么 `std::stringstream` 则是一个不错的选择。

消除字符串开始和结束处的空格

应用从用户端获取到的输入，经常会有很多不必要的空格存在。之前的章节中，将单词间多余的空格进行移除。

现在让我们来看看，被空格包围的字符串应该怎么去移除多余的空格。`std::string` 具有很多不错的辅助函数能完成这项工作。

Note:

这节看完后，下节也别错过。将会在下节看到我们如何使用 `std::string_view` 类来避免不必要的拷贝或数据修改。

How to do it...

本节，我们将完成一个辅助函数的实现，其将判断是否有多余的空格在字符串开头和结尾，并复制返回去掉这些空格的字符串，并进行简单的测试：

1. 包含必要的头文件，并声明所使用的命名空间。

```
#include <iostream>
#include <string>
#include <algorithm>
#include <cctype>

using namespace std;
```

2. 函数将对一个常量字符串进行首尾空格的去除，并返回首尾没有空格的新字符串：

```
string trim_whitespace_surrounding(const string &s)
{
```

3. `std::string` 能够提供两个函数，这两个函数对我们很有帮助。第一个就是 `string::find_first_not_of`，其能帮助我们找到我们想要跳过的字符。本节中毫无疑问就是空格，其包括空格、制表符和换行符。函数能返回第一个非空格字符的位置。如果字符串里面只有空格，那么会返回 `string::npos`。这意味着没有找到除了空格的其他字符。如果这样，我们就会返回一个空的字符串：

```
const char whitespace[] {"\t\n"};
const size_t first
(s.find_first_not_of(whitespace));
if (string::npos == first) { return {}; }
```

4. 现在我们知道新字符串从哪里开始，但是再哪里结尾呢？因此，需要使用另一个函数 `string::find_last_not_of`，其能找到最后一个非空格字符的位置：

```
    const size_t last  
(s.find_last_not_of whitespace));
```

5. 使用 `string::substr` 就能返回子字符串，返回的字符串没有空格。这个函数需要两个参数——一个是字符串的起始位置，另一个是字符串的长度：

```
    return s.substr(first, (last - first + 1));  
}
```

6. 这就完成了。现在让我们来编写主函数，创建字符串，让字符串的前后充满空格，以便我们进行移除：

```
int main()  
{  
    string s {" \t\n string surrounded by ugly"  
              " whitespace \t\n "};
```

7. 我们将打印去除前和去除后的字符串。将字符串放入大括号中，这样就很容易辨别哪里有空格了：

```
cout << "{" << s << " }\n";  
cout << "{"  
     << trim_whitespace_surrounding(s)  
     << " }\n";  
}
```

8. 编译运行程序，就会得到如下的结果：

```
$ ./trim_whitespace  
{  
    string surrounded by ugly whitespace  
}  
{string surrounded by ugly whitespace}
```

How it works...

本节，我们使用了 `string::find_first_not_of` 和 `string::find_last_not_of` 函数。这两个函数也能接受C风格的字符串，会将其当做字符链表进行搜索。当有一个字符串 `foo bar` 时，当调用 `find_first_not_of("bfo ")` 时返回5，因为'a'字符是第一个不属于 `bfo` 的字符。参数中字符的顺序，在这里并不重要。

倒装的函数也是同样的原理，当然还有两个没有使用到的函数：`string::find_first_of` 和 `string::find_last_of`。

同样也是基于迭代器的函数，需要检查函数是否返回了合理的位置，当没有找到时，函数会返回一个特殊的位置——`string::npos`。

我们可以从辅助函数中找出字符所在的位置，并且使用 `string::substr` 来构造前后没有空格的字符串。这个函数接受一个首字符相对位置和字符串长度，然后就会构造一个子字符串进行返回。举个栗子，`string{"abcdef"}.substr(2, 2)` 将返回 `cd`。

无需构造获取`std::string`

`std::string` 类是一个十分有用的类，因为其对字符串的处理很方便。其有一个缺陷，当我们想要根据一个字符串获取其子字符串时，我们需要传入一个指针和一个长度变量，两个迭代器或一段拷贝的子字符串。我在之前的章节也这样使用过，消除字符串前后的空格的最后，使用的是拷贝的方式获得前后无空格的字符串。

当我们想要传递一个字符串或一个子字符串到一个不支持 `std::string` 的库中时，需要提供裸指针，这样的用法就回退到C的时代。与子字符串问题一样，裸指针不携带字符串长度信息。这样的话就需要将指针和字符串长度进行捆绑。

另一个十分简单的方式就是使用 `std::string_view`。这个类是C++17添加的新特性，并且能提供将字符串指针与其长度捆绑的方法，其体现了数组引用的思想。

当设计函数时，将 `std::string` 实例作为参数，但在函数中使用了额外的内存来存储这些字符，以确保原始的字符串不被修改，这时就可以使用 `std::string_view`，其可移植性很好，与STL无关。可以让其他库来提供一个 `string_view` 实现，然后将复杂的实现隐藏在背后，并且可以将其用在我们的STL代码中。这样，`string_view` 类就显得非常小，非常好用，因为其能在不同的库间都可以用。

`string_view` 另一个很酷的特性，就是可以使用非拷贝的方式引用大字符串中的子字符串。本节将使用 `string_view`，从而了解其优点和缺点。我们还会看到如何使用字符串代理来去除字符两端的空格，并不对原始字符串进行修改和拷贝。

How to do it...

本节，将使用 `string_view` 的一些特性来实现一个函数，我们将会看到有多少种类型可以输入：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <string_view>

using namespace std;
```

2. 将 `string_view` 作为函数的参数：

```
void print(string_view v)
{
```

3. 对输入字符串做其他事情之前，将移除字符开头和末尾的空格。将不会对字符串进行修改，仅适用字符串代理获取没有空格字符串。`find_first_not_of` 函数将会在字符串找到第一个非空格的字符，适用 `remove_prefix`，`string_view` 将指向第一个非空格的字符。当字符串只有空格，`find_first_not_of` 函数会返回 `npos`，其为 `size_type(-1)`。`size_type` 是一个无符号类型，其可以是一个非常大的值。所以，会在字符串代理的长度和 `words_begin` 中选择较小的那个：

```
    const auto words_begin (v.find_first_not_of("\t\n"));
    v.remove_prefix(min(words_begin, v.size()));
```

4. 我们对尾部的空格做同样的事情。`remove_suffix` 将收缩到代理的大小:

```
    const auto words_end (v.find_last_not_of("\t\n"));
    if (words_end != string_view::npos) {
        v.remove_suffix(v.size() - words_end - 1);
    }
```

5. 现在可以打印字符串代理和其长度:

```
cout << "length: " << v.length()
<< " [" << v << "] \n";
}
```

6. 主函数中, 将使用 `print` 的函数答应一系列完全不同的参数类型。首先, 会通过 `argv` 传入 `char*` 类型的变量, 运行时其会包含可执行文件的名字。然后, 传入一个 `string_view` 的实例。然后, 使用C风格的静态字符串, 并使用 "`sv` 字面字符构造的 `string_view` 类型。最后, 传入一个 `std::string`。 `print` 函数不需要对参数进行修改和拷贝。这样就没有多余的内存分配发生。对于很多大型的字符串, 这将会非常有效:

```
int main(int argc, char *argv[])
{
    print(argv[0]);
    print({});
    print("a const char * array");
    print("an std::string_view literal"sv);
    print("an std::string instance"s);
```

7. 这里还没对空格移除特性进行测试。这里也给出一个头尾都有空格的字符串:

```
print(" \t\n foobar \n \t ");
```

8. `string_view` 另一个非常酷的特性是, 其给予的字符串是不包含终止符的。当构造一个字符串, 比如"abc", 没有终止符, `print` 函数就能很安全的对其进行处理, 因为 `string_view` 携带字符串的长度信息和指向信息:

```
char cstr[] {'a', 'b', 'c'};
print(string_view(cstr, sizeof(cstr)));
}
```

9. 编译并运行程序，就会得到如下的输出，所有字符串都能被正确处理。前后有很多空格的字符串都被正确的处理，`abc` 字符串没有终止符也能被正确的打印，而没有任何内存溢出：

```
$ ./string_view
length: 17 [./string_view]
length: 0 []
length: 20 [a const char * array]
length: 27 [an std::string_view literal]
length: 23 [an std::string instance]
length: 6 [foobar]
length: 3 [abc]
```

How it works...

我们可以看到，函数可以接受传入一个 `string_view` 的参数，其看起来与字符串类型没有任何区别。我们实现的 `print`，对于传入的字符串不进行任何的拷贝。

对于 `print(argv[0])` 的调用是非常有趣的，字符串代理会自动的推断字符串的长度，因为需要将其适用于无终止符的字符串。另外，我们不能通过查找终止符的方式来确定 `string_view` 实例的长度。正因如此，当使用裸指针(`string_view::data()`)的时候就需要格外小心。通常字符串函数都会认为字符串具有终止符，这样就很难出现使用裸指针时出现内存溢出的情况。这里还是使用字符串代理的接口比较好。

除此之外，`std::string` 接口阵容已经非常豪华了。

Note:

使用 `std::string_view` 用于解析字符或获取子字符串时，能避免多余的拷贝和内存分配，并且还不失代码的舒适感。不过，对于 `std::string_view` 将终止符去掉这点，需要特别注意。

从用户的输入读取数值

本书中大多数例程的输入都是从文件或标准输入中获得。这次我们重点来了解一下读取，以及当遇到一些有问题的流时，不能直接终止程序，而是要做一些错误处理的工作。

本节只会从用户输入中读取，知道如何读取后，将了解如何从其他的流中读取数据。用户输入通常通过 `std::cin`，其为最基础的输入流对象，类似这样的类还有 `ifstream` 和 `istringstream`。

How to do it...

本节，将从用户输入中读取不同值，并且了解如何进行错误处理，并对输入中有用的部分进行较为复杂的标记。

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>

using namespace std;
```

2. 首先，提示用户输入两个数字。将这两个数字解析为 `int` 和 `double` 类型。例如，用户输入 `1 2.3`：

```
int main()
{
    cout << "Please Enter two numbers:\n> ";
    int x;
    double y;
```

3. 解析和错误检查同时在 `if` 判断分支中进行。只有两个数都被解析成有效的数字，才能对其进行打印：

```
if (cin >> x >> y) {
    cout << "You entered: " << x
        << " and " << y << '\n';
```

4. 如果因为任何原因，解析不成功，那么我们要告诉用户为什么会出错。`cin` 流对象现在处于失败的状态，并且将错误状态进行清理之前，无法为我们提供输入功能。为了能够对新的输入进行解析需要调用 `cin.clear()`，并且将之前接收到的字符丢弃。使用 `cin.ignore` 完成丢弃的任务，这里我们指定了丢弃字符的数量，直到遇到下一个换行符为止。完成这些事之后，输入有可以用了：

```

    } else {
        cout << "Oh no, that did not go well!\n";
        cin.clear();
        cin.ignore(
            std::numeric_limits<std::streamsize>::max(),
            '\n');
    }
}

```

5. 让用户输入一些其他信息。我们让用户输入名字，名字由多个字母组成，字母间使用空格隔开。因此，可以使用 `std::getline` 函数，其需要传入一个流对象和一个分隔字符。我们选逗号作为分隔字符。这里使用 `getline` 来代替 `cin >> ws` 的方式读入字符，这样我们就能丢弃在名字前的所有空格。对于每一个循环中都会打印当前的名字，如果名字为空，那么我们会将其丢弃：

```

cout << "now please enter some "
      "comma-separated names:\n> ";
for (string s; getline(cin >> ws, s, ',');) {
    if (s.empty()) { break; }
    cout << "name: \" " << s << "\"\n";
}

```

6. 编译并运行程序，就会得到如下的输出，其会让用户进行输入，然后我们输入合法的字符。数字 1 2 都能被正确的解析，并且后面输入的名字也能立即排列出来。两个逗号间没有单词的情况将会跳过：

```

$ ./strings_from_user_input
Please Enter two numbers:
> 1 2
You entered: 1 and 2
now please enter some comma-separated names:
> john doe,ellen ripley, alice,chuck norris,,
name: "john doe"
name: "ellen ripley"
name: "alice"
name: "chuck norris"

```

7. 再次运行程序，这次将在一开始就输入一些非法数字，可以看到程序就会走到不同的分支，然后丢弃相应的输入，并继续监听正确的输入。可以看到 `cin.clear()` 和 `cin.ignore(...)` 的调用如何对名字读取进行影响：

```
$ ./strings_from_user_input
Please Enter two numbers:
> a b
Oh no, that did not go well!
now please enter some comma-separated names:
> bud spencer, terence hill,
name: "bud spencer"
name: "terence hill"
```

How it works...

本节，我们对一些复杂输入进行了检索。首要注意的是，我们的检索和错误处理是同时进行。

表达式 `cin >> x` 是对 `cin` 的再次引用。因此，就可以将输入些为 `cin >> x >> y >> z >> ...`。与此同时，其也能将输入内容转换成为一个布尔值，并在 `if` 条件中使用。这个布尔值告诉我们最后一次读取是否成功，这也就是为什么我们会将代码写成 `if (cin >> x >> y) { ... }` 的原因。

当我们想要读取一个整型，但输入中包含 `foobar` 为下一个表示，那么流对象将无法对这段字符进行解析，并且这让输入流的状态变为失败。这对于解析来说是非常关键的，但对于整个程序来说就不是了。这里可以将输入流的状态进行重置，然后在进行其他的操作。在我们的例程中，我们尝试在读取两个数值失败后，读取一组姓名。例子中，我们使用 `cin.clear()` 对 `cin` 的工作状态进行了重置。不过，这样内部的光标就处于我们的现在的类型上，而非之前的数字。为了将之前输入的内容丢弃，并对姓名输入进行流水式的读取，我们使用了一个比较长的表达式，`cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');`。这里对内存的清理是十分有必要的，因为我们需要在用户输入一组姓名时，对缓存进行刷新。

下面的循环看起来也挺奇怪的：

```
for (string s; getline(cin >> ws, s, ',')); { ... }
```

`for` 循环的判断部分，使用了 `getline` 函数。`getline` 函数接受一个输入流对象，一个字符串引用作为输出，以及一个分隔符。通常，分隔字符代表新的一行。这里使用逗号作为分隔符，所以姓名输入列表为 `john, carl, frank`，这样就可以单个的进行读取。

目前为止还不错。不过，`cin >> ws` 的操作是怎么回事呢？这可以让 `cin` 对所有空格进行刷新，其会读取下一个非空格字符到下一个逗号间的字符。回看一下"john, carl, frank"例子，当我们不使用 `ws` 时，将获取到"john", " carl"和" frank"字符串。这里需要注意"carl"和"frank"开头不必要的空格，因为在 `ws` 中对输入流进行了预处理，所以能够避免开头出现空格的情况。

计算文件中的单词数量

我们在读取一个文件的时候，也想知道这个文件中包含的单词数量。我们定义的单词是位于两个空格之间的字符组合。那要如何进行统计呢？

根据对单词的定义，我们可以统计空格的数量。例如句子 `John has a funny little dog.`，这里有五个空格，所以说这句话有六个单词。

如果句子中有空格干扰怎么办，例如：`John has \t a\nfunny little dog .`。这句中有很多不必要的空格、制表符和换行符。本书的其他章节中，我们已经了解如何将多余空格从字符串中去掉。所以，可以对字符串进行预处理，将不必要的空格都去掉。这样做的确可行，不过我们有更加简单的方法。

为了寻找最优的解决方案，我们将让用户选择，是从标准输入中获取数据，还是从文本文件中获取数据。

How to do it...

本节，我们将完成一个单行统计函数，其可以对输入的数据进行计数，数据源的具体方式我们可以让用户来选择。

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
#include <iterator>

using namespace std;
```

2. `wordcount` 函数能接受一个输入流，例如 `cin`。其能创建一个 `std::input_iterator` 迭代器，其能对输出字符进行标记，然后交由 `std::distance` 进行计算。`distance` 接受两个迭代器作为参数，并确定从一个迭代器到另一个迭代器要用多少步(距离)。对于随机访问迭代器，因为有减法操作符的存在，所以实现起来非常简单。其迭代器如同指针一样，可以直接进行减法，计算出两点的距离。不过 `istream_iterator` 就不行，因为其是前向迭代器，只能向前读取，直至结束。最后所需要的步数也就是单词的数量：

```
template <typename T>
size_t wordcount(T &is)
{
    return distance(istream_iterator<string>(is),
{});
```

3. 主函数中，我们会让用户来选择输入源：

```
int main(int argc, char **argv)
{
    size_t wc;
```

4. 如果用户选择使用文件进行输入(例如: `./count_all_words some_textfile.txt`), 我们可以通过 `argv` 获取命令行中的文件名称, 并将文件打开, 读取数据, 从而对文本进行单词统计:

```
if (argc == 2) {
    ifstream ifs {argv[1]};
    wc = wordcount(ifs);
```

5. 如果用户没有传入任何参数, 就认为用户要使用标准输入流输入数据:

```
} else {
    wc = wordcount(cin);
}
```

6. 然后只需要将统计出的单词数量保存在变量 `wc` 中即可:

```
cout << "There are " << wc << " words\n";
};
```

7. 编译并运行程序。首先, 从标准输入中进行输入。我们可以这里通过 `echo` 命令将字符串, 通过管道传递给程序。当然, 我们也可以直接进行输入, 并使用 `ctrl+D` 来结束输入:

```
$ echo "foo bar baz" | ./count_all_words
There are 3 words
```

8. 这次我们使用文件作为输入源, 并对其中单词数量进行统计:

```
$ ./count_all_words count_all_words.cpp
There are 61 words
```

How it works...

本节也没有什么好多说的; 实现很短, 难度很低。需要提及的可能就是我们对 `std::cin` 和 `std::ifstream` 的实例进行了互换。`cin` 是 `std::istream` 的类型之一, 并且 `std::ifstream` 继承于 `std::istream`。可以回顾一下本章开头的类型继承表。这两种类型即使在运行时, 都能进行互换。

Note:

使用流来保持代码的模块性, 这有助于减少代码的耦合性。因为其可以匹配任意类型的流对象, 所以更容易对代码进行测试。

格式化输出

很多情况下，仅打印字符串和数字是不够的。数字通常都以十进制进行打印，有时我们需要使用十六进制或八进制进行打印。并且在打印十六进制的时候，我们希望看到以 `0x` 为前缀的十六进制的数字，但有时却不希望看到这个前缀。

当对浮点数进行打印的时候，也需要注意很多。以何种精度进行打印？要将数中的所有内容进行打印吗？或者是如何打印科学计数法样式的数？

除了数值表示方面的问题外，还需要规范我们打印的格式。有时我们要以表格的方式进行打印，以确保打印数据的可读性。

这所有的一切都与输出流有关，对输入流的解析也十分重要。本节中，我们将来感受一下格式化输出。有些显示也会比较麻烦，不过我们会对其进行解释。

How to do it...

为了让大家熟悉格式化输出，本节我们将使用各种各样的格式进行打印：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <iomanip>
#include <locale>

using namespace std;
```

2. 接下来，定义一个辅助函数，其会以不同的方式打印出一个数值。其能接受使用一种字符对宽度进行填充，其默认字符为空格：

```
void print_aligned_demo(int val,
                        size_t width,
                        char fill_char = ' ')
{
```

3. 使用 `setw`，我们可以设置打印数字的最小字符数输出个数。当我们想要将123的输出宽度设置为6时，我们会得到"abc "或" abc"。我们也可以使用 `std::left`，`std::right` 和 `std::internal` 控制从哪边进行填充。当我们以十进制的方式对数字进行输出，`internal` 看起来和 `right` 的作用一样。不过，当打印 `0x1` 时，打印宽度为6时，`internal` 会得到"0x 6"。`setfill` 控制符可以用来定义填充字符。我么可以尝试使用以下方式进行打印：

```

    cout << "=====\n";
    cout << setfill(fill_char);
    cout << left << setw(width) << val << '\n';
    cout << right << setw(width) << val << '\n';
    cout << internal << setw(width) << val << '\n';
}

```

4. 主函数中，我们使用已经实现的函数。首先，打印数字12345，其宽度为15。
我们进行两次打印，不过第二次时，将填充字符设置为'_':

```

int main()
{
    print_aligned_demo(123456, 15);
    print_aligned_demo(123456, 15, '_');
}

```

5. 随后，我们将打印 `0x123abc`，并使用同样的宽度。不过，打印之前需要使用的是 `std::hex` 和 `std::showbase` 告诉输出流对象 `cout` 输出的格式，并且添加 `0x` 前缀，看起来是一个十六进制数：

```

cout << hex << showbase;
print_aligned_demo(0x123abc, 15);

```

6. 对于八进制我们也可以做同样的事：

```

cout << oct;
print_aligned_demo(0123456, 15);

```

7. 通过 `hex` 和 `uppercase`，我们可以将 `0x` 中的x转换成大写字母。`0x123abc` 中的 `abc` 同样也转换成大写：

```

cout << "A hex number with upper case letters: "
<< hex << uppercase << 0x123abc << '\n';

```

8. 如果我们要以十进制打印100，我们需要将输出从 `hex` 切换回 `dec`：

```

cout << "A number: " << 100 << '\n';
cout << dec;

cout << "Oops. now in decimal again: " << 100 <<
'\n';

```

9. 我们可以对布尔值的输出进行配置，通常，`true`会打印出1，`false`为0。使用 `boolalpha`，我们就可以得到文本表达：

```
cout << "true/false values: "
<< true << ", " << false << '\n';
cout << boolalpha
<< "true/false values: "
<< true << ", " << false << '\n';
```

10. 现在让我们来一下浮点型变量 `float` 和 `double` 的打印。当我们有一个数`12.3`，那么打印也应该是`12.3`。当我们有一个数`12.0`，打印时会将小数点那一位进行丢弃，不过我们可以通过 `showpoint` 来控制打印的精度。使用这个控制符，就能显示被丢弃的一位小数了：

```
cout << "doubles: "
<< 12.3 << ", "
<< 12.0 << ", "
<< showpoint << 12.0 << '\n';
```

11. 可以使用科学计数法或固定浮点的方式来表示浮点数。`scientific` 会将浮点数归一化成一个十进制的小数，并且其后面的位数使用`10`的幂级数表示，其需要进行乘法后才能还原成原始的浮点数。比如，`300.0`科学计数法就表示为"`3.0E2`"，因为 $300 = 3.0 \times 10^2$ 。`fixed` 将会恢复普通小数的表达方式：

```
cout << "scientific double: " << scientific
<< 123000000000.123 << '\n';
cout << "fixed double: " << fixed
<< 123000000000.123 << '\n';
```

12. 除此之外，我们也能对打印的精度进行控制。我们先创建一个特别小的浮点数，并对其小数点后的位数进行控制：

```
cout << "Very precise double: "
<< setprecision(10) << 0.0000000001 << '\n';
cout << "Less precise double: "
<< setprecision(1) << 0.0000000001 << '\n';
}
```

13. 编译并运行程序，我们就会得到如下的输出。前四个块都是有打印辅助函数完成，其使用 `setw` 对字符串进行了不同方向的填充。此外，我们也进行了数字的进制转换、布尔数表示和浮点数表示。通过实际操作，我们会对其更加熟悉：

```
$ ./formatting
=====
123456
    123456
        123456
=====
123456 _____
      _____ 123456
          _____ 123456
=====
0x123abc
    0x123abc
0x      123abc
=====
0123456
    0123456
        0123456
A hex number with upper case letters: 0X123ABC
A number: 0X64
Ooop. now in decimal again: 100
true/false values: 1, 0
true/false values: true, false
doubles: 12.3, 12, 12.0000
scientific double: 1.230000E+12
fixed double: 1230000000000.123047
Very precise double: 0.0000000001
Less previse double: 0.0
```

How it works...

例程看起来有些长，并且 `<< foo << bar` 的方式对于初级读者来说会感觉到困惑。因此，让我们来看一下格式化修饰符的表。其都是用 `input_stream >> modifier` 或 `output_stream << modifier` 来对之后的输入输出进行影响：

符号	描述
<code>setprecision(int)</code>	打印浮点数时，决定打印小数点后的位数。
<code>showpoint / noshowpoint</code>	启用或禁用浮点数字小数点的打印，即使没有小数位。
<code>fixed /scientific / hexfloat /defaultfloat</code>	数字可以以固定格式和科学表达式的方式进行打印。 <code>fixed</code> 和 <code>scientific</code> 代表了相应的打印模式。 <code>hexfloat</code> 将会同时激活这两种模式，用十六进制浮点表示法格式化浮点数。 <code>defaultfloat</code> 则会禁用这两种模式。
<code>showpos / noshowpos</code>	启用或禁用使用'+'来标志正浮点数。
<code>setw(int n)</code>	设置打印的宽度 <code>n</code> 。在读取的时候，这种设置会截断输入。当打印位数不够时，其会使用填充字符将输出填充到 <code>n</code> 个字符。
<code>fill(char c)</code>	当我们 <code>setw</code> 时，会涉及填充字符的设置。 <code>fill</code> 可以将填充字符设置为 <code>c</code> 。其默认填充字符为空格。
<code>internal / left / right</code>	<code>left</code> 和 <code>right</code> 控制填充的方向。 <code>internal</code> 会将填充字符放置在数字和符号之间，这对于十六进制打印和一些金融数字来说，十分有用。
<code>dec / hex / oct</code>	整数打印的类型，十进制、十六进制和八进制。
<code>setbase(int n)</code>	数字类型的同义函数，当 <code>n</code> 为 <code>10/16/8</code> 时，与 <code>dec / hex / oct</code> 完全相同。当传入 <code>0</code> 时，则会恢复默认输出，也就是十进制，或者使用数字的前缀对输入进行解析。
<code>quoted(string)</code>	将带有引号的字符串的引号去掉，对其实际字符进行打印。这里 <code>string</code> 的类型可以是 <code>string</code> 类的实例，也可以是一个C风格的字符串。
<code>boolalpha / noboolalpha</code>	打印布尔变量，是打印字符形式的，还是数字形式的。
<code>showbase / noshowbase</code>	启用或禁用基于前缀的数字解析。对于 <code>hex</code> 来说就是 <code>0x</code> ，对于 <code>octal</code> 来说就是 <code>0</code> 。
<code>uppercase / nouppercase</code>	启用或禁用将浮点数中的字母或十六进制中的字符进行大写输出。

看起来很多，想要熟悉这些控制符的最好方式，还是尽可能多的使用它们。

在使用中会发现，其中有一些控制符具有粘性，另一些没有。这里的粘性是说其会持续影响接下来的所有输入或输出，直到对控制符进行重置。表格中没有粘性的为 `setw` 和 `quoted` 控制符。其只对下一次输入或输出有影响。了解这些非常重要，当我们要持续使用一个格式进行打印时，对于有粘性的控制符我们设置一次即可，其余的则需要进行设置。这些对输入解析同样适用，不过错误的设置了控制符则会得到错误的输入信息。

下面的一些控制符我们没有使用它们，因为他们对于格式化没有任何影响，但出于完整性的考量我们在这里也将这些流状态控制符列出来：

符号	描述
<code>skipws / noskipws</code>	启用或禁用输入流对空格进行略过的特性。
<code>unitbuf / nounitbuf</code>	启用或禁用在进行任何输出操作后，就立即对输出缓存进行刷新。
<code>ws</code>	从输入流舍弃前导空格。
<code>ends</code>	向流中输入一个终止符 <code>\0</code> 。
<code>flush</code>	对输出缓存区进行刷新。
<code>endl</code>	向输出流中插入 <code>\n</code> 字符，并且刷新输出缓存区。

这些控制符中，只有 `skipws / noskipws` 和 `unitbuf / nounitbuf` 是具有粘性的。

使用输入文件初始化复杂对象

将整型、浮点型和字符串分开读取不是困难，因为流操作符 `>>` 对于基础类型有重载的版本，并且输入流会将输入中的空格去除。

不过，对于更加复杂的结构体来说，我们应该如何将其从输入流中读取出来，并且当我们的字符串中需要多个单词的时候应该怎么做呢(在空格处不断开)？

对于任意类型，我们都可以对输入流 `operator>>` 操作符进行重载，接下来我们就要看下如何做这件事：

How to do it...

本节，我们将定义一个数据结构，并从标准输入中获取数据：

1. 包含必要的头文件和声明所使用的命名空间：

```
#include <iostream>
#include <iomanip>
#include <string>
#include <algorithm>
#include <iterator>
#include <vector>

using namespace std;
```

2. 创建一个复杂的对象，我们定义了一个名为 `city` 的结构体。城市需要有名字，人口数量和经纬坐标。

```
struct city {
    string name;
    size_t population;
    double latitude;
    double longitude;
};
```

3. 为了从输入流中读取一个城市的信息，这时我们就需要对 `operator>>` 进行重载。对于操作符来说，会跳过 `ws` 开头的所有空格，我们不希望空格来污染城市的名称。然后，会对一整行的文本进行读取。这样类似于从输入文件中读取一行，行中只包含城市的信息。然后，我们就可以用空格将人口，经纬度进行区分：

```

istream& operator>>(istream &is, city &c)
{
    is >> ws;
    getline(is, c.name);
    is >> c.population
        >> c.latitude
        >> c.longitude;
    return is;
}

```

4. 主函数中，我们创建一个 `vector`，其包含了若干城市元素，使用 `std::copy` 将其进行填充。我们会将输入的内容拷贝到 `istream_iterator` 中。通过给定的 `city` 结构体作为模板参数，其会使用重载过的 `operator>>` 进行数据的读取：

```

int main()
{
    vector<city> l;

    copy(istream_iterator<city>{cin}, {},
         back_inserter(l));
}

```

5. 为了了解城市信息是否被正确解析，我们会将其进行打印。使用格式化输出 `left << setw(15) <<`，城市名称左边必有很多的空格，这样我们的输出看起来就很漂亮：

```

for (const auto &[name, pop, lat, lon] : l) {
    cout << left << setw(15) << name
        << " population=" << pop
        << " lat=" << lat
        << " lon=" << lon << '\n';
}
}

```

6. 例程中所用到的文件内容如下。我们将四个城市的信息写入文件：

```

Braunschweig
250000 52.268874 10.526770
Berlin
4000000 52.520007 13.404954
New York City
8406000 40.712784 -74.005941
Mexico City
8851000 19.432608 -99.133208

```

7. 编译并运行程序，将会得到如下输入。我们在输入文件中为城市名称前添加一些不必要的空白，以查看空格是如何被过滤掉的：

```
$ cat cities.txt | ./initialize_complex_objects
Braunschweig    population = 250000 lat = 52.2689 lon
= 10.5268
Berlin          population = 4000000 lat = 52.52 lon
= 13.405
New York City   population = 8406000 lat = 40.7128
lon = -74.0059
Mexico City     population = 8851000 lat = 19.4326
lon = -99.1332
```

How it works...

本节也非常短。我们只是创建了一个新的结构体 `city`，我们对 `std::istream` 迭代器的 `operator>>` 操作符进行重载。这样也就允许我们使用 `istream_iterator<city>` 对数据进行反序列化。

关于错误检查则是一个开放性的问题。我们现在再来看下 `operator>>` 实现：

```
istream& operator>>(istream &is, city &c)
{
    is >> ws;
    getline(is, c.name);
    is >> c.population >> c.latitude >> c.longitude;
    return is;
}
```

我们读取了很多不同的东西。读取数据发生了错误，下一个应该怎么办？这是不是意味着我们有可能读取到错误的数据？不会的，这不可能发生。即便是其中一个元素没有被输入流进行解析，那么输入流对象则会置于错误的状态，并且拒绝对剩下的输入进行解析。这样就意味着，如果 `c.population` 或 `c.latitude` 没有被解析出来，那么对应的输入数据将会被丢弃，并且我们可以看到反序列化了一半的 `city` 对象。

站在调用者的角度，我们需要注意这句 `if(input_stream >> city_object)`。这也就表明流表达式将会被隐式转换成一个布尔值。当其返回`false`时，输入流对象则处于错误状态。如果出现错误，就需要采取相应的措施对流进行重置。

本节中没有使用 `if` 判断，因为我们让 `std::istream_iterator<city>` 进行反序列化。`operator++` 在迭代器的实现中，会在解析时对其状态进行检查。当遇到错误时，其将会停止之后的所有迭代。当前迭代器与 `end` 迭代器比较返回`true`时，将终止 `copy` 算法的执行。如此，我们的代码就很安全了。

迭代器填充容器——`std::istream`

上节中，我们学习了如何从输入流中向数据结构中读入数据，然后用这些数据填充列表或向量。

这次，我们将使用标准输入来填充 `std::map`。问题在于我们不能将一个结构体进行填充，然后从后面推入到线性容器中，例如 `list` 和 `vector`，因为 `map` 的负载分为键和值两部分。

完成本节后，我们会了解到如何从字符流中将复杂的数据结构进行序列化和反序列化。

How to do it...

本节，我们会定义一个新的结构体，不过这次将其放入 `map` 中，这会让问题变得复杂，因为容器中使用键值来表示所有值。

1. 包含必要的头文件，并声明所使用的命名空间。

```
#include <iostream>
#include <iomanip>
#include <map>
#include <iterator>
#include <algorithm>
#include <numeric>

using namespace std;
```

2. 我们会引用网络上的一些梗。这里的梗作为一个名词，我们记录其描述和诞生年份。我们会将这些梗放入 `std::map`，其名称为键，包含在结构体中的其他信息作为值：

```
struct meme {
    string description;
    size_t year;
};
```

3. 我们暂时先不去管键，我们先来实现结构体 `meme` 的流操作符 `operator>>`。我们假设相关梗的描述由双引号括起来，后跟对应年份。举个栗子，“`some description" 2017`”。通过使用 `is >> quoted(m.description)`，双引号会被当做限定符，直接被丢弃。这就非常的方便。然后我们继续读取年份即可：

```
istream& operator>>(istream &is, meme &m) {
    return is >> quoted(m.description) >> m.year;
}
```

4. OK, 现在将梗的名称作为键插入 `map` 中。为了实现插入 `map`，需要一个 `std::pair<key_type, value_type>` 实例。`key_type` 为 `string`，那么 `value_type` 就是 `meme` 了。名字中允许出现空格，所以可以使用 `quoted` 对名称进行包装。`p.first` 是名称，`p.second` 代表的是相关 `meme` 结构体变量。可以使用 `operator>>` 实现直接对其进行赋值：

```
istream& operator >>(istream &is,
                        pair<string, meme> &p) {
    return is >> quoted(p.first) >> p.second;
}
```

5. 现在来写主函数，创建一个 `map` 实例，然后对其进行填充。因为对流函数 `operator>>` 进行了重载，所以可以直接对 `istream_iterator` 类型直接进行处理。我们将会从标准输入中解析出更多的信息，然后使用 `inserter` 迭代器将其放入 `map` 中：

```
int main()
{
    map<string, meme> m;

    copy(istream_iterator<pair<string, meme>>{cin},
         {},
         inserter(m, end(m)));
}
```

6. 对梗进行打印前，先在 `map` 中找到名称最长的梗吧。可以对其使用 `std::accumulate`。累加的初始值为 `0u`(`u`为无符号类型)，然后逐个访问 `map` 中的元素，将其进行合并。使用 `accumulate` 合并，就意味着叠加。例子中，并不是对数值进行叠加，而是对最长字符串的长度进行累加。为了得到长度，我们为 `accumulate` 提供了一个辅助函数 `max_func`，其会将当前最大的变量与当前梗的名字长度进行比较(这里两个数值类型需要相同)，然后找出这些值中最大的那个。这样 `accumulate` 函数将会返回当前梗中，名称最长的梗：

```
auto max_func ([](size_t old_max,
                    const auto &b) {
    return max(old_max, b.first.length());
});
size_t width {accumulate(begin(m), end(m),
                        0u, max_func)};
```

7. 现在，对 `map` 进行遍历，然后打印其中每一个元素。使用 `<< left << setw(width)` 打印出漂亮的“表格”：

```

        for (const auto &[meme_name, meme_desc] : m) {
            const auto &[desc, year] = meme_desc;

            cout << left << setw(width) << meme_name
                << " : " << desc
                << ", " << year << '\n';
        }
    }
}

```

8. 现在需要一些梗的数据，我们写了一些梗在文件中：

```

"Doge" "Very Shiba Inu. so dog. much funny. wow."
2013
"Pepe" "Anthropomorphic frog" 2016
"Gabe" "Musical dog on maximum borkdrive" 2016
"Honey Badger" "Crazy nastyass honey badger" 2011
"Dramatic Chipmunk" "Chipmunk with a very dramatic
look" 2007

```

9. 编译并运行程序，将文件作为数据库进行输入：

```

$ cat memes.txt | ./filling_containers
Doge: Very Shiba Inu. so dog. much funny. wow., 2013
Dramatic Chipmunk : Chipmunk with a very dramatic
look, 2007
Gabe: Musical dog on maximum borkdrive, 2016
Honey Badger: Crazy nastyass honey badger, 2011
Pepe: Anthropomorphic frog, 2016

```

How it works...

本节有三点需要注意。第一，没有选择 `vector` 或 `list` 比较简单的结构，而是选择了 `map` 这样比较复杂的结构。第二，使用了 `quoted` 控制符对输入流进行处理。第三，使用 `accumulate` 来找到最长的键值。

我们先来看一下 `map`，结构体 `meme` 只包含一个 `description` 和 `year`。因为我们将梗的名字作为键，所以没有将其放入结构体中。可以将 `std::pair` 实例插入 `map` 中，首先实现了结构体 `meme` 的流操作符 `operator>>`，然后对 `pair<string, meme>` 做同样的事。最后，使用 `istream_iterator<pair<string, meme>>{cin}` 从标准输入中获取每个元素的值，然后使用 `inserter(m, end(m))` 将组对插入 `map` 中。

当我们使用流对 `meme` 元素进行赋值时，允许梗的名称和描述中带有空格。我们使用引号控制符，很轻易的将问题解决，得到的信息类似于这样，“`Name with spaces`”
“`Description with spaces`” 123。

当输入和输出都对带有引号的字符串进行处理时，`std::quoted` 就能帮助到我们。当有一个字符串 `s`，使用 `cout << quoted(s)` 对其进行打印，将会使其带引号。当对流中的信息进行解析时，`cin >> quoted(s)` 其就能帮助我们将引号去掉，保留引号

中的内容。

叠加操作是对 `max_func` 的调用看起来很奇怪：

```
auto max_func ([](size_t old_max, const auto &b) {
    return max(old_max, b.first.length());
});

size_t width {accumulate(begin(m), end(m), 0u,
max_func)};
```

实际上，`max_func` 能够接受一个 `size_t` 和一个 `auto` 类型的参数，这两个参数将转换成一个 `pair`，从而就能插入 `map` 中。这看起来很奇怪，因为二元函数会将两个相同类型的变量放在一起操作，例如 `std::plus`。我们会从每个组对中获取键值的长度，将当前元素的长度值与之前的最长长度相对比。

叠加调用会将 `max_func` 的返回值与 `0u` 值进行相加，然后作为左边参数的值与下一个元素进行比较。第一次左边的参数为 `0u`，所以就可以写成 `max(0u, string_length)`，这时返回的值就作为之前最大值，与下一个元素的名称长度进行比较，以此类推。

迭代器进行打印——`std::ostream`

使用输出流进行打印是一件很容易的事情，STL中的大多数基本类型都对 `operator<<` 操作符进行过重载。所以使用 `std::ostream_iterator` 类，就可以将数据类型中所具有的的元素进行打印，我们已经在之前的章节中这样做了。

本节中，我们将关注如何将自定义的类型进行打印，并且可以通过模板类进行控制。对于调用者来说，无需写太多的代码。

How to do it...

我们将对一个新的自定义的类使用 `std::ostream_iterator`，并且看起来其具有隐式转换的能力，这就能帮助我们进行打印：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <vector>
#include <iterator>
#include <unordered_map>
#include <algorithm>

using namespace std;
using namespace std::string_literals;
```

2. 让我们实现一个转换函数，其会将数字和字符串相对应。比如输入1，就会返回“one”；输入2，就会返回“two”，以此类推：

```
string word_num(int i) {
```

3. 将会对哈希表进行填充，我们后续可以对它进行访问：

```
unordered_map<int, string> m {
    {1, "one"}, {2, "two"}, {3, "three"},
    {4, "four"}, {5, "five"}, //...
};
```

4. 现在可以使用哈希表的 `find` 成员函数，通过传入相应的键值，返回对应的值。如果 `find` 函数找不到任何东西，我们就会得到一个“unknown”字符串：

```
const auto match (m.find(i));
if (match == end(m)) { return "unknown"; }
return match->second;
};
```

5. 接下来我们就要定义一个结构体 `bork`。其仅包含一个整型成员，其可以使用一个整型变量进行隐式构造。其具有 `print` 函数，其能接受一个输出流引用，通过 `borks` 结构体的整型成员变量，重复打印"bork"字符串：

```
struct bork {
    int borks;

    bork(int i) : borks(i) {}

    void print(ostream& os) const {
        fill_n(ostream_iterator<string>{os, " "},
                borks, "bork!"s);
    }
};
```

6. 为了能够更方便的对 `bork` 进行打印，对 `operator<<` 进行了重载，当通过输出流对 `bork` 进行输出时，其会自动的调用 `bork::print`：

```
ostream& operator<<(ostream &os, const bork &b) {
    b.print(os);
    return os;
}
```

7. 现在来实现主函数，先来初始化一个 `vector`：

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

8. `ostream_iterator` 需要一个模板参数，其能够表述哪种类型的变量我们能够进行打印。当使用 `ostream_iterator<T>` 时，其会使用 `ostream& operator(ostream&, const T&)` 进行打印。这也就是之前在 `bork` 类型中重载的输出流操作符。我们这次只对整型数字进行打印，所以使用 `ostream_iterator<int>`。使用 `cout` 进行打印，并可以将其作为构造参数。我们使用循环对 `vector` 进行访问，并且对每个输出迭代器 `i` 进行解引用。这也就是在STL算法中流迭代器的用法：

```
ostream_iterator<int> oit {cout};
for (int i : v) { *oit = i; }
cout << '\n';
```

9. 使用的输出迭代器还不错，不过其打印没有任何分隔符。当需要空格分隔符对所有打印的元素进行分隔时，我们可以将空格作为第二个参数传入输出流构造函数中。这样，其就能打印"1, 2, 3, 4, 5, "，而非"12345"。不过，不能在打印最后一个数字的时候将“逗号-空格”的字符串丢弃，因为迭代器并不知道哪个数字是最后一个：

```
ostream_iterator<int> oit_comma {cout, ", "};

for (int i : v) { *oit_comma = i; }
cout << '\n';
```

10. 为了将其进行打印，我们将值赋予一个输出流迭代器。这个方法可以和算法进行结合，其中最简单的方式就是 `std::copy`。我们可以通过提供 `begin` 和 `end` 迭代器来代表输入的范围，在提供输出流迭代器作为输出迭代器。其将打印 `vector` 中的所有值。这里我们会将两个输出循环进行比较：

```
copy(begin(v), end(v), oit);
cout << '\n';

copy(begin(v), end(v), oit_comma);
cout << '\n';
```

11. 还记得 `word_num` 函数吗？其会将数字和字符串进行对应。我们也可以使用进行打印。我们只需要使用一个输出流操作符，因为我们不需要对整型变量进行打印，所以这里使用的是 `string` 的特化版本。使用 `std::transform` 替代 `std::copy`，因为需要使用转换函数将输入范围内的值转换成其他值，然后拷贝到输出中：

```
transform(begin(v), end(v),
         ostream_iterator<string>{cout, " "},
         word_num);
cout << '\n';
```

12. 程序的最后一行会对 `bork` 结构体进行打印。可以直接使用，也并不需要为 `std::transform` 函数提供任何转换函数。另外，可以创建一个输出流迭代器，其会使用 `bork` 进行特化，然后再调用 `std::copy`。`bork` 实例可以通过输入范围内的整型数字进行隐式创建。然后，将会得到一些有趣的输出：

```
copy(begin(v), end(v),
      ostream_iterator<bork>{cout, "\n"});
}
```

13. 编译并运行程序，就会得到以下输出。前两行和第三四行的结果非常类似。然后，会得到数字对应的字符串，然后就会得到一堆 `bork!` 字符串。其会打印很多行，因为我们使用换行符替换了空格：

```
$ ./ostream_printing
12345
1, 2, 3, 4, 5,
12345
1, 2, 3, 4, 5,
one two three four five
bork!
bork! bork!
bork! bork! bork!
bork! bork! bork! bork!
bork! bork! bork! bork! bork!
```

How it works...

作为一个语法黑客，我们应知道 `std::ostream_iterator` 可以用来对数据进行打印，其在语法上为一个迭代器，对这个迭代器进行累加是无效的。对其进行解引用会返回一个代理对象，这些赋值操作符会将这些数字转发到输出流中。

输出流迭代器会对类型T进行特化(`ostream_iterator<T>`)，对于所有类型的 `ostream& operator<<(ostream&, const T&)` 来说，都需要对其进行实现。

`ostream_iterator` 总是会调用 `operator<<`，通过模板参数，我们已经对相应类型进行了特化。如果类型允许，这其中会发生隐式转换。当A可以隐式转换为B时，我们可以对A类型的元素进行迭代，然后将这些元素拷贝到 `output_iterator` 的实例中。我们会对 `bork` 结构体做同样的事情： `bork` 实例也可以隐式转换为一个整数，这也就是我们能够很容易的在终端输出一堆 `bork!` 的原因。

如果不能进行隐式转换，可使用 `std::transform` 和 `word_num` 函数相结合，对元素类型进行转换。

Note:

通常，对于自定义类型来说，隐式转换是一种不好的习惯，因为这是一个常见的Bug源，并且这种Bug非常难找。例子中，隐式构造函数有用程度要超过其危险程度，因为相应的类只是进行打印。

使用特定代码段将输出重定向到文件

`std::cout` 为我们提供了一种非常方便的打印方式，使用起来也十分方便，易于扩展，并可全局访问。即使我们想打印对应的信息时，比如错误信息，我们可以使用错误输出 `std::cerr` 进行输出，其和 `cout` 的用法一样，只不过一个从标准通道进行输出，另一个从错误通道进行输出。

当我们要打印比较复杂的日志信息时。比如，要将函数的输出重定向到一个文件中，或者将函数的打印输出处于静默状态，而不需要对函数进行任何修改。或许这个函数为一个库函数，我们没有办法看到其源码。可能，这个函数并没有设计为写入到文件的函数，但是我们还是想将其输出输入到文件中。

这里可以重定向输出流对象的输出。本节中，我们将看到如何使用一种简单并且优雅地方式来完成输出流的重定向。

How to do it...

我们将实现一个辅助类，其能在构造和析构阶段，帮助我们完成流的重定向，以及对流的定向进行恢复。然后，我们来看其是怎么使用的：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <fstream>

using namespace std;
```

2. 我们实现了一个类，其具有一个文件输出流对象和一个指向流内部缓冲区的指针。`cout` 作为流对象，其内部具有一个缓冲区，其可以用来进行数据交换，我们可以保存我们之前做过的事情，这样就很方便进行对后续修改的撤销。我们可以在C++手册中查询对其返回类型的解释，也可以使用 `decltype` 对 `cout.rdbuf()` 所返回的类型进行查询。这并不是一个很好的体验，在我们的例子中，其就是一个指针类型：

```
class redirect_cout_region
{
    using buftype = decltype(cout.rdbuf());

    ofstream ofs;
    buftype buf_backup;
```

3. 类的构造函数接受一个文件名字符串作为输入参数。这个字符串用来初始化文件流成员 `ofs`。对其进行初始化后，可以将其输入到 `cout` 作为一个新的流缓冲区。`rdbuf` 在接受一个新缓冲区的同时，会将旧缓冲区以指针的方式进行返回，这样当需要对缓冲区进行恢复时，就可以直接使用了：

```
public:  
    explicit  
    redirect_cout_region (const string &filename)  
    : ofs{filename}  
    , buf_backup{cout.rdbuf(ofs.rdbuf())}  
    {}
```

4. 默认构造函数和其他构造函数做的事情几乎一样。其区别在于，默认构造函数不会打开任何文件。默认构造的文件流会直接替换 `cout` 的流缓冲，这样会导致 `cout` 的一些功能失效。其会丢弃一些要打印的东西。这在某些情况下是非常有用的：

```
redirect_cout_region()  
: ofs{}  
, buf_backup{cout.rdbuf(ofs.rdbuf())}  
{}
```

5. 析构函数会对重定向进行恢复。当类在运行过程中超出了范围，可以使用原始的 `cout` 流缓冲区对其进行还原：

```
~redirect_cout_region() {  
    cout.rdbuf(buf_backup);  
}  
};
```

6. 让我们模拟一个有很多输出的函数：

```
void my_output_heavy_function()  
{  
    cout << "some output\n";  
    cout << "this function does really heavy work\n";  
    cout << "... and lots of it...\n";  
    // ...  
}
```

7. 主函数中，我们先会进行一次标准打印：

```
int main()  
{  
    cout << "Readable from normal stdout\n";
```

8. 现在进行重定向，首先使用一个文本文件名对类进行实例化。文件流会使用读取和写入模式作为默认模式，所以其会创建一个文件。所以即便是后续使用 `cout` 进行打印，其输出将会重定向到这个文件中：

```
{  
    redirect_cout_region _ {"output.txt"};  
    cout << "Only visible in output.txt\n";  
    my_output_heavy_function();  
}
```

9. 离开这段代码后，文件将会关闭，打印输出也会重归标准输出。我们再开启一个代码段，并使用默认构造函数对类进行构造。这样后续的打印信息将无法看到，都会被丢弃：

```
{  
    redirect_cout_region _;  
    cout << "This output will "  
        "completely vanish\n";  
}
```

10. 离开这段代码后，我们的标准输出将再度恢复，并且将程序的最后一行打印出来：

```
cout << "Readable from normal stdout again\n";  
}
```

11. 编译并运行这个程序，其输出和我们期望的一致。我们只看到了第一行和最后一行输出：

```
$ ./log_regions  
Readable from normal stdout  
Readable from normal stdout again
```

12. 我们可以将新文件output.txt打开，其内容如下：

```
$ cat output.txt  
Only visible in output.txt  
some output  
this function does really heavy work  
... and lots of it...
```

How it works...

每个流对象都有一个内部缓冲区，这样的缓冲区可以进行交换。当我们有一个流对象 `s` 时，我们将其缓冲区存入到变量 `a` 中，并且为流对象换上了一个新的缓冲区 `b`，这段代码就可以完成上述的过程：`a = s.rdbuf(b)`。需要恢复的时候只需要执行 `s.rdbuf(a)`。

这就如同我们在本节所做的。另一件很酷的事情是，可以将这些 `redirect_cout_region` 辅助函数放入堆栈中：

```
{  
    cout << "print to standard output\n";  
  
    redirect_cout_region la {"a.txt"};  
    cout << "print to a.txt\n";  
  
    redirect_cout_region lb {"b.txt"};  
    cout << "print to b.txt\n";  
}  
cout << "print to standard output again\n";
```

这也应该好理解，通常析构的顺序和构造的顺序是相反的。这种模式是将对象的构造和析构进行紧耦合，其也称作为资源获得即初始化(**RAII**)。

这里有一个很重要的点需要注意——`redirect_cout_region` 类中成员变量的初始化顺序：

```
class redirect_cout_region {  
    using buftype = decltype(cout.rdbuf());  
    ofstream ofs;  
    buftype buf_backup;  
public:  
    explicit  
    redirect_cout_region(const string &filename)  
    : ofs{filename},  
    buf_backup{cout.rdbuf(ofs.rdbuf())}  
    {}  
    ...
```

我们可以看到，成员 `buf_backup` 的初始化需要依赖成员 `ofs` 进行。有趣的是，这些成员初始化的顺序，不会按照初始化列表中给定元素的顺序进行初始化。这里初始化的顺序值与成员变量声明的顺序有关！

Note:

当一成员变量需要在另一个成员变量之后进行初始化，其需要在类声明的时候以相应的顺序进行声明。初始化列表中的顺序，对于构造函数来说没有任何影响。

通过集成**std::char_traits**创建自定义字符串类

我们知道 `std::string` 非常好用。不过，对于一些朋友来说他们需要对自己定义的字符串类型进行处理。

使用它们自己的字符串类型显然不是一个好主意，因为对于字符串的安全处理是很困难的。幸运的是，`std::string` 是 `std::basic_string` 类型的一个特化版本。这个类中包含了所有复杂的内存处理，不过其对字符串的拷贝和比较没有添加任何条件。所以我们可以基于 `basic_string`，将其所需要包含的自定义类作为一个模板参数传入。

本节中，我们将来看下如何传入自定义类型。然后，在不实现任何东西的情况下，如何对自定义字符串进行创建。

How to do it...

我们将实现两个自定义字符串类：`lc_string` 和 `ci_string`。第一个类将通过输入创建一个全是小写字母的字符串。另一个字符串类型不会对输入进行任何变化，不过其会对字符串进行大小写不敏感的比较：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <algorithm>
#include <string>

using namespace std;
```

2. 然后要对 `std::tolower` 函数进行实现，其已经定义在头文件 `<cctype>` 中。其函数也是现成的，不过其不是 `constexpr` 类型。`C++17` 中一些 `string` 函数可以声明成 `constexpr` 类型，但是还要使用自定义的类型。所以对于输入字符串，只将大写字母转换为小写，而其他字符则不进行修改：

```
static constexpr char tolow(char c) {
    switch (c) {
        case 'A'...'Z': return c - 'A' + 'a'; // 读者自行将
        case展开
        default:           return c;
    }
}
```

3. `std::basic_string` 类可以接受三个模板参数：字符类型、字符特化类和分配器类型。本节中我们只会修改字符特化类，因为其定义了字符串的行为。为了重新实现与普通字符串不同的部分，我们会以 `public` 方式继承标准字符特化类：

```
class lc_traits : public char_traits<char> {
public:
```

4. 我们类能接受输入字符串，并将其转化成小写字母。这里有一个函数，其是字符级别的，所以我们可以对其使用 `tolow` 函数。我们的这个函数为 `constexpr`：

```
static constexpr
void assign(char_type& r, const char_type& a) {
    r = tolow(a);
}
```

5. 另一个函数将整个字符串拷贝到我们的缓冲区内。使用 `std::transform` 将所有字符从源字符串中拷贝到内部的目标字符串中，同时将每个字符与其小写版本进行映射：

```
static char_type* copy(char_type* dest,
                      const char_type* src,
                      size_t count) {
    transform(src, src + count, dest, tolow);
    return dest;
}
};
```

6. 上面的特化类可以帮助我们创建一个字符串类，其能有效的将字符串转换成小写。接下来我们在实现一个类，其不会对原始字符串进行修改，但是其能对字符串做大小写敏感的比较。其继承于标准字符串特征类，这次将对一些函数进行重新实现：

```
class ci_traits : public char_traits<char> {
public:
```

7. `eq` 函数会告诉我们两个字符是否相等。我们也会实现一个这样的函数，但是我们只实现小写字母的版本。这样'A'与'a'就是相等的：

```
static constexpr bool eq(char_type a, char_type
b) {
    return tolow(a) == tolow(b);
}
```

8. `lt` 函数会告诉我们两个字符在字母表中的大小情况。这里使用了逻辑操作符，并继续对两个字符使用转换成小写的函数：

```

    static constexpr bool lt(char_type a, char_type
b) {
    return tolow(a) < tolow(b);
}

```

9. 最后两个函数都是字符级别的函数，接下来两个函数都为字符串级别的函数。`compare` 函数与 `strcmp` 函数差不多。当两个字符串的长度 `count` 相等，那么就返回0。如果不相等，会返回一个负数或一个正数，返回值就代表了其中哪一个在字母表中更小。并计算两个字符串中所有字符之间的距离，当然这些都是在小写情况下进行的操作。C++14后，这个函数可以声明成 `constexpr` 类型：

```

static constexpr int compare(const char_type*
s1,
                           const char_type* s2,
                           size_t count) {
    for (; count; ++s1, ++s2, --count) {
        const char_type diff (tolow(*s1) -
tolow(*s2));
        if (diff < 0) { return -1; }
        else if (diff > 0) { return +1; }
    }
    return 0;
}

```

10. 我们所需要实现的最后一个函数就是大小写不敏感的 `find` 函数。对于给定输入字符串 `p`，其长度为 `count`，我们会对某个字符 `ch` 的位置进行查找。然后，其会返回一个指向第一个匹配字符位置的指针，如果没有找到则返回 `nullptr`。这个函数比较过程中我们需要使用 `tolow` 函数将字符串转换成小写，以匹配大小写不敏感的查找。不幸的是，我们不能使用 `std::find_if` 来做这件事，因为其是非 `constexpr` 函数，所以我们需要自己去写一个循环：

```

static constexpr
const char_type* find(const char_type* p,
                      size_t count,
                      const char_type& ch) {
    const char_type find_c {tolow(ch)};
    for (; count != 0; --count, ++p) {
        if (find_c == tolow(*p)) { return p; }
    }
    return nullptr;
}

```

11. OK，所有自定义类都完成了。这里我们可以定义两种新字符串类的类型。`lc_string` 代表小写字符串，`ci_string` 代表大小写不敏感字符串。这两种类型与 `std::string` 都有所不同：

```
using lc_string = basic_string<char, lc_traits>;
using ci_string = basic_string<char, ci_traits>;
```

12. 为了能让输出流接受新类，我们需要对输出流操作符进行重载：

```
ostream& operator<<(ostream& os, const lc_string& str) {
    return os.write(str.data(), str.size());
}

ostream& operator<<(ostream& os, const ci_string& str) {
    return os.write(str.data(), str.size());
}
```

13. 现在我们来对主函数进行编写。先让我们创建一个普通字符串、小写字符串和大小写不敏感字符串的实例，然后直接将其进行打印。其在终端上看起来都很正常，不过小写字符串将所有字符转换成了小写：

```
int main()
{
    cout << " string: "
        << string{"Foo Bar Baz"} << '\n'
        << "lc_string: "
        << lc_string{"Foo Bar Baz"} << '\n'
        << "ci_string: "
        << ci_string{"Foo Bar Baz"} << '\n';
```

14. 为了测试大小写不敏感字符串，可以实例化两个字符串，这两个字符串只有在大小写方面有所不同。当我们将这两个字符串进行比较时，其应该是相等的：

```
ci_string user_input {"MaGiC PaSsWoRd!"};
ci_string password {"magic password!"};
```

15. 之后，对其进行比较，然后将匹配的结果进行打印：

```
if (user_input == password) {
    cout << "Passwords match: \""
        << "\"" == "\"" << password << "\"\n";
}
```

16. 编译并运行程序，其输出和我们期望的相符。开始的三行并未对输入进行修改，除了 `lc_string` 将所有字符转换成了小写。最后的比较，在大小写不敏感的前提下，也是相等的：

```
$ ./custom_string
string: Foo Bar Baz
lc_string: foo bar baz
ci_string: Foo Bar Baz
Passwords match: "MaGiC PaSsWoRd!" == "magic
password!"
```

How it works...

我们完成的所有子类和函数实现，在新手看来十分的不可思议。这些函数签名都来自于哪里？为什么我们为了函数签名，就要对相关功能性的函数进行重新实现呢？

首先，来看一下 `std::string` 的类声明：

```
template <
    class CharT,
    class Traits = std::char_traits<CharT>,
    class Allocator = std::allocator<CharT>
>
class basic_string;
```

可以看出 `std::string` 其实就是一个 `std::basic_string<char>` 类，并且其可以扩展为 `std::basic_string<char, std::char_traits<char>, std::allocator<char>>`。OK，这是一个非常长的类型描述，不过其意义何在呢？这就表示字符串可以不限于有符号类型 `char`，也可以是其他类型。其对于字符串类型都是有效的，这样就不限于处理 ASCII 字符集。当然，这不是我们的关注点。

`char_traits<char>` 类包含 `basic_string` 所需要的算法。`char_traits<char>` 可以进行字符串间的比较、查找和拷贝。

`allocator<char>` 类也是一个特化类，不过其运行时给字符串进行空间的分配和回收。这对于现在的我们来说并不重要，我们只使用其默认的方式就好。

当我们想要一个不同的字符串类型时，可以尝试对 `basic_string` 和 `char_traits` 类中提供的方法进行复用。我们实现了两个 `char_traits` 子类：`case_insensitive` 和 `lower_caser` 类。我们可以将这两个字符串类替换标准 `char_traits` 类型。

Note:

为了探寻 `basic_string` 适配的可能性，我们需要查询 C++ STL 文档中关于 `std::char_traits` 的章节，然后去了解还有哪些函数需要重新实现。

使用正则表达式库标记输入

当我们需要使用一些较为复杂的方式解析或转换字符串时，正则表达式是个不错的选择。因为非常好用，很多编程语言中都会内置正则表达式。

如果你还对正则表达式不太了解，可以去维基百科的相关页面进行了解。我相信其会扩展你的视野，正则表达式对于文本解析来说十分好用。正则表达式能用来检查一个电子邮件或IP地址是否合法，也能从长字符串中找到对应的子字符串等等。

本节中，我们将提取HTML文件中的链接，并且将这些链接为使用者罗列出来。因为正则表达式在C++11标准中正式加入C++ STL，所以例程很短。

How to do it...

我们将定义一个正则表达式来检测链接，并且将其作用于一个HTML文件，并将获得的链接打印出来：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <iterator>
#include <regex>
#include <algorithm>
#include <iomanip>

using namespace std;
```

2. 在后面将会生成一段可迭代器的区间，这个区间中只包含字符串。这里会以链接地址字符串和链接描述字符串配对出现。因此，我们也要写一个辅助函数来打印这些字符串：

```
template <typename InputIt>
void print(InputIt it, InputIt end_it)
{
    while (it != end_it) {
```

3. 每次循环中，我们对迭代器增加了两次，这是因为要对链接地址和链接描述进行拷贝。两个迭代器引用间，我们添加了一个if条件，为了保证程序的安全，这个条件句会检查迭代器是否过早的到达了最后：

```
    const string link {*it++};
    if (it == end_it) { break; }
    const string desc {*it++};
```

4. 现在我们就可以对链接及其描述进行打印：

```

        cout << left << setw(28) << desc
        << " : " << link << '\n';
    }
}

```

5. 主函数中，我们将从标准输入中获取所要读取的数据。这样，需要将全部标准输入通过一个输入流迭代器构造为一个字符串。为了避免符号化，为了确保我们所得到的输入与用户输入的一样，我们使用了 `noskipws` 控制符。这个控制符将会禁用空格跳过和符号化：

```

int main()
{
    cin >> noskipws;
    const std::string in {
        istream_iterator<char>{cin}, {}
    };
}

```

6. 现在我们需要定义一个正则表达式，来对HTML文件进行查找。小括号在正在表达式中代表的是组，这里我们要获取我们想要访问的链接——其为URL地址，并且还要获取其描述：

```

const regex link_re {
    "<a href=\"([^\"]*)\"[^<]*>([^\<]*)</a>";
}

```

7. `sregex_token_iterator` 类具有相同的功能，并且能对 `istream_iterator` 直接操作。我们将可迭代的输入范围和刚刚定义的正则表达式传给它。不过，这里还有第三个参数 `{1, 2}`。其表示我们想要表达式组1和组2中的结果：

```

sregex_token_iterator it {
    begin(in), end(in), link_re, {1, 2}
};

```

8. 现在我们有一个迭代器，如果找到了连接，其会返回连接地址和相应的描述。这里对第二个参数直接进行初始化，其类型与第一个参数类型相同，然后传入我们之前实现的 `print` 函数中：

```

    print(it, {});
}

```

9. 编译并运次那个程序，就会得到如下的输出。我们使用curl获取ISO C++首页的信息，其会将HTML页面直接从网上下载下来。当然，这里也能写成 `cat some_html_file.html | ./link_extraction`。正则表达式可以很方便对硬编码进行解析，通过HTML固定的格式对其进行解析。当然，你可以让其变得更加通用：

```

$ curl -s "https://isocpp.org/blog" |
./link_extraction
Sign In / Suggest an Article :
https://isocpp.org/member/login
Register : https://isocpp.org/member/register
Get Started! : https://isocpp.org/get-started
Tour : https://isocpp.org/tour
C++ Super-FAQ: https://isocpp.org/faq
Blog : https://isocpp.org/blog
Forums : https://isocpp.org/forums
Standardization: https://isocpp.org/std
About: https://isocpp.org/about
Current ISO C++ status :
https://isocpp.org/std/status
(...and many more...)

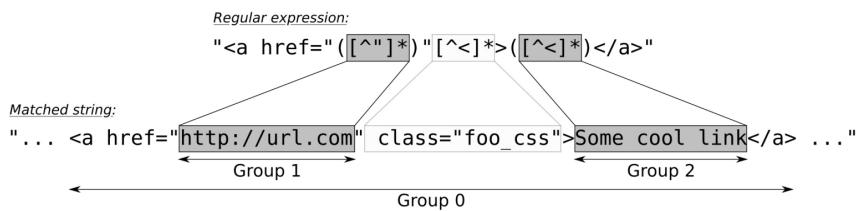
```

How it works...

正则表达式非常有用，看起来好像特别神秘，但值得学习。一个短小的表达式就能节省对我们多行的代码进行手动匹配的时间。

本节中，我们第一次实例化了一个正则类型的对象。我们使用一个用于描述的字符串对正则表达式进行构造。最简单的正则表达式是".", 其会对每个字符进行匹配，因为它是正则表达式的通配符。表达式为"a"时，其就只会对'a'字符进行匹配。表达式为"ab*"时，其表示"只有一个a，和零个或若干个b"，以此类推。正则表达式本身是一个很大的主题，维基百科和一些教学网站，还有一些学术论文中，对其都有非常详尽的描述。

让我们来看一下本节中的正则表达式，是如何对HTML连接进行匹配的。一个简单HTML连接可写为 `A great link`。我们只需要 `some_url.com/foo` 和 `A great link` 部分。所以，我们可以使用如下的正则表达式进行匹配，其会将字符串对应每个组，从而分解成多个字符串：



字符串本身为第**0**组，也就是整个字符串。引号中 `href` 的URL地址部分分在第**1**组中。正则表达式中，使用小括号来定义组，所以这个表达式中有两个组。另一个组则获取的是连接描述。

有很多**STL**函数可以接受正则对象，不过直接使用一个正则字符迭代器适配器，其是对使用 `std::regex_search` 进行自动化匹配的高阶抽象。我们可用如下的代码对其进行实例化：

```
sregex_token_iterator it {begin(in), end(in), link_re,
{1, 2}};
```

开始和结束部分表示我们的输入字符串，正则迭代器则在该字符串上进行迭代，并匹配所有链接。`link_re` 则为用于匹配连接的正则表达式，`{1, 2}` 可用来表示我们需要的部分。其第一组匹配的是我们想要的连接，进行自增后，就到了第二组，匹配的则是我们想要的连接描述，依次循环。其能非常智能的帮助我们对多行数据进行处理。

让我们来看另一个例子，以确保我们明白了上面的内容。让我们创建一个正则表达式 `"a(b*)(c*)"`。其会对具有一个'a'字符，之后有或没有'b'字符，再之后有或没有'c'字符的字符串进行匹配：

```
const string s {" abc abbccc "};
const regex re {"a(b*)(c*)"};

sregex_token_iterator it {
    begin(s), end(s), re, {1, 2}
};

print( *it ); // prints b
++it;
print( *it ); // prints c
++it;
print( *it ); // prints bb
++it;
print( *it ); // prints ccc
```

当然也可以使用 `std::regex_iterator` 类，其会将匹配的子字符串进行直接输出。

简单打印不同格式的数字

之前的章节中，我们已经了解如何打印出带格式的输出，同时也意识到了两点：

- 输入输出控制符是有粘性的，所以当我们要临时使用的时候，需要在用完之后进行还原。
- 其控制符和较少的要打印的对象相比，会显得很冗长。

这些原因导致一些开发者使用C++的时候，还是依旧使用 `printf` 进行打印输出。

本节，我们将来看一下如何不用太多代码就能进行很好的类型打印。

How to do it...

我们会先来实现一个类 `format_guard`，其会自动的将打印格式进行恢复。另外，我们添加了一个包装类型，其可以包含任意值，当对其进行打印时，其能使用相应的格式进行输出，而无需添加冗长的控制符：

1. 包含必要的头文件，并声明所使用的命名空间。

```
#include <iostream>
#include <iomanip>

using namespace std;
```

2. 辅助类在调用 `format_guard` 时，其会对输出流的格式进行清理。其构造函数保存了格式符，也就是在这里对 `std::cout` 进行设置。析构函数会将这些状态进行去除，这样就不会后续的打印有所影响：

```
class format_guard {
    decltype(cout.flags()) f {cout.flags()};
public:
    ~format_guard() { cout.flags(f); }
};
```

3. 定义另一个辅助类 `scientific_type`。因为其是一个模板类，所以其能拥有任意类型的成员变量。这个类没有其他任何作用：

```
template <typename T>
struct scientific_type {
    T value;

    explicit scientific_type(T val) : value{val} {}
};
```

4. 封装成 `scientific_type` 之后，可以对任意类型进行自定义格式设置，当对 `operator>>` 进行重载后，输出流就会在执行时，运行完全不同的代码。这样就能在使用科学计数法表示浮点数时，以大写的格式，并且其为正数时，数字前添加'+'号。我们也会在跳出函数时，使用 `format_guard` 类对打印格式进行清理：

```
template <typename T>
ostream& operator<<(ostream &os, const
scientific_type<T> &w) {
    format_guard _;
    os << scientific << uppercase << showpos;
    return os << w.value;
}
```

5. 主函数中，我们将使用到 `format_guard` 类。我们会创建一段新的代码段，首先对类进行实例化，并且对 `std::cout` 进行输出控制符的设置：

```
int main()
{
{
    format_guard _;
    cout << hex << scientific << showbase <<
uppercase;

    cout << "Numbers with special formatting:\n";
    cout << 0x123abc << '\n';
    cout << 0.123456789 << '\n';
}
```

6. 使用控制符对这些数字进行打印后，跳出这个代码段。这时 `format_guard` 的析构函数会将格式进行清理。为了对清理结果进行测试，会再次打印相同的数字。其将会输出不同的结果：

```
cout << "Same numbers, but normal formatting
again:\n";
cout << 0x123abc << '\n';
cout << 0.123456789 << '\n';
```

7. 现在使用 `scientific_type`，将三个浮点数打印在同一行。我们将第二个数包装成 `scientific_type` 类型。这样其就能按照我们指定的风格进行打印，不过在之前和之后的输出都是以默认的格式进行。与此同时，我们也避免了冗长的格式设置代码：

```
cout << "Mixed formatting: "
<< 123.0 << " "
<< scientific_type{123.0} << " "
<< 123.456 << '\n';
}
```

8. 编译并运行程序，我们就会得到如下的输出。前两行按照我们的设定进行打印。接下来的两行则是以默认的方式进行打印。这样就证明了我们的 `format_guard` 类工作的很好。最后三个数在一行上，也是和我们的期望一致。只有中间的数字是 `scientific_type` 类型的，前后两个都是默认类型：

```
$ ./pretty_print_on_the_fly
Numbers with special formatting:
0X123ABC
1.234568E-01
Same numbers, but normal formatting again:
1194684
0.123457
Mixed formatting: 123 +1.230000E+02 123.456
```

从std::iostream错误中获取可读异常

本书之前的章节中，我们还没对异常进行过捕获。不过对于流对象不会抛出异常，所以很容易使用。当我们想要解析10个数，不过解析过程在中途失败了，那么流对象将会将自身设置为失败状态，并且不会继续对数字进行解析。这样，我们就不会让程序处于危险当中。我们可以将解析过程转换为一个条件变量，比如 `if (cin >> foo >> bar >> ...)`。如果这个判断失败了，那我们将对输入进行处理。所以，这里并不会出现 `try-catch` 代码块。

实际上，之前的C++输入输出流是会抛出异常的。异常这个特性是不是一开始就有 的，所以这也可能是流对象库并不是第一个支持异常特性的原因。

为了对流使用异常，我们必须对每个流对象单独进行配置，让其在失败的时候抛出一个异常。不幸的是，我们可以对对象的异常进行捕获，但是这步并没有标准化。这就导致我们无法获得有效的错误信息，我们将在后续的实例中看到。如果我们很想对流对象使用异常，那么可以使用C库中有关文件系统错误状态，来获取更多的信息。

本节中，我们将会通过不同的方式，让程序运行失败，然后来处理这些异常，并且了解如何获取更多的有效信息。

How to do it...

我们将会让程序打开一个文件(这个过程可能会失败)，并且将会从文件中读取一个整型数字(也可能会失败)。我们可以通过激活异常的方式来发现错误，然后再来看如何对这些错误进行处理：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <fstream>
#include <system_error>
#include <cstring>

using namespace std;
```

2. 当我们要将流对象和异常一起使用时，首先需要启动异常。为了获取一个文件流对象，在指定文件并不存在时，抛出一个异常；或是在解析错误时，我们需要将对应的失败原因设置到异常掩码的对应位上。当执行失败的时候，将触发一个异常。并通过激活的 `failbit` 和 `badbit`，我们能让文件系统的错误抛出异常，并对这个错误进行解析：

```
int main()
{
    ifstream f;
    f.exceptions(f.failbit | f.badbit);
```

3. 现在可以使用 `try` 块进行对文件的访问。文件打开成功，那我们将继续读取文件中的整型数字。并且，只有在读取数字成功的情况下，我们才会对数字进行打印：

```
try {
    f.open("non_existant.txt");

    int i;
    f >> i;

    cout << "integer has value: " << i << '\n';
}
```

4. 对于可能发生的两种错误，一个 `std::ios_base::failure` 实例将会抛出。这个对象有一个 `what()` 成员函数，其会为我们解释触发了哪种异常。不幸的是，并不存在标准化的信息，所以我们不会得到太多有用的信息。不过，我们至少可以区分，触发异常的是一个文件系统问题，还是一个格式解析问题。全局变量 `errno`，其在C++诞生前就存在，其会设置为一个错误值，可供我们进行查看。`strerror` 函数会将一个错误值，翻译为我们可以读懂的字符串。当 `errno` 是0时，就代表文件系统没有任何错误：

```
catch (ios_base::failure& e) {
    cerr << "Caught error: ";
    if (errno) {
        cerr << strerror(errno) << '\n';
    } else {
        cerr << e.what() << '\n';
    }
}
```

5. 编译并运行程序，两种错误可能都会在运行时发生。当文件不存在时，我们就不可能从文件中获取数值，所以我们会得到一个 `iostream_category` 错误信息：

```
$ ./readable_error_msg
Caught error: ios_base::clear: unspecified
iostream_category
```

6. 如果文件不存在，`strerror(errno)` 将会返回不同的错误信息：

```
$ ./readable_error_msg
Caught error: No such file or directory
```

How it works...

我们可以通过 `s.exceptions(s.failbit | s.badbit)` 使能流对象 `s` 抛出异常的能力。不过，这也意味着有些情况无法使用异常，例如 `std::ifstream` 的实例需要打开一个文件进行构造，所以我们不能在之后对异常进行设置。

```
ifstream f {"non_existant.txt"};
f.exceptions(...); // too late for an exception
```

这就十分遗憾了，因为异常处理与原始C风格的方式进行对比，无需被 `if` 困扰，其每一步都是在处理异常。

当我们使用各种方法让流处于失败的状态，就会发现抛出的这些异常并没有什么区别。这样只需要了解何时捕获错误，而非捕获了什么错误(对于流是这样，而对于 **STL** 中的其他类型就不是了)，这也就是为什么我们要对 `errno` 的值进行检查的原因。这个全局变量在 C++ 和异常诞生之前，就已经存在了。

如果有任何与系统相关的函数发生了错误，其会将 `errno` 设置为除0之外的其他值(0代表没有错误)，然后调用者可以通过对 `errno` 值的查询，来了解到底出现了什么问题。这个问题我们在多线程程序会经常遇到，并且所有线程都会对一个全局变量进行修改，那么当出现错误了，是哪个线程造成的呢？幸运的是，这个设计已经在 C++11 中进行了修改，每个线程都只能看到属于自己的 `erron` 变量。

对于原始的错误处理方式，我们就不进行详细的描述了，不过其能为我们提供额外有用的信息，比如流对象触发了基于系统的异常。异常会告诉我们发生了什么，而 `erron` 则会告诉我们会发生哪种级别的错误。

第8章 工具类

本章将关注工具类，这些类能帮助我们很快地解决一些特定的任务。有些工具类我们将会在本书后续的章节中经常使用。

前两节与时间测量有关，我们将了解到如何在两种不同的时间单位间互相转换，并如何确定两个时间点。

然后，我了解一下 `optional`、`variant` 和 `any` 类型(都是在C++14和C++17中添加的新类)，在接下来的5节中，我们将介绍有关 `tuple` 的内容。

C++11之后，C++中添加了新的智能指针类型，分别为：`unique_ptr`，`shared_ptr` 和 `weak_ptr`，因为智能指针方便对内存的管理，所以给智能指针设置了5节内容。

最后，将从大体上浏览一下STL中有关于随机数生成的部分。除了学习STL中随机数引擎的特性之外，还将了解到如何在实际应用中选择合适的随机数分布。

转换不同的时间单位——`std::ratio`

C++11之后，STL具有了很多用来测量和显示时间的新类型和函数。STL这部分内容放在 `std::chrono` 命名空间中。

本节我们将关注测量时间，以及如何对两种不同的时间单位进行转换，比如：秒到毫秒和微秒的转换。STL已经提供了现成的工具，我们可以自定义时间单位，并且可以无缝的在不同的时间单位间进行转换。

How to do it...

本节，我们写一个小游戏，会让用户输入一个单词，然后记录用户打字的速度，并以不同的时间单位显示所用时间：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <chrono>
#include <ratio>
#include <cmath>
#include <iomanip>
#include <optional>

using namespace std;
```

2. `chrono::duration` 经常用来表示所用时间的长度，其为秒的倍数或小数，所有STL的程序都由整型类型进行特化。本节中，将使用 `double` 进行特化。本节之后，我们更多的会关注已经存在于STL的时间单位：

```
using seconds = chrono::duration<double>;
```

3. 1毫秒为1/1000秒，可以用这个单位来定义秒。`ratio_multiply` 模板参数可以使STL预定义的 `milli` 用来表示 `seconds::period`，其会给我们相应的小数。`ratio_multiply` 为基本时间的倍数：

```
using milliseconds = chrono::duration<
    double, ratio_multiply<seconds::period, milli>>;
```

4. 对于微秒来说也是一样的。可以使用 `micro` 表示：

```
using microseconds = chrono::duration<
    double, ratio_multiply<seconds::period, micro>>;
```

5. 现在我们实现一个函数，会用来从用户的输入中读取一个字符串，并且统计用户输入所用的时间。这个函数没有参数，在返回用户输入的同时，返回所用的时间，我们用一个组对(pair)将这两个数进行返回：

```
static pair<string, seconds> get_input()
{
    string s;
```

6. 我们需要从用户开始输入时计时，记录一个时间点的方式可以写成如下方式：

```
const auto tic (chrono::steady_clock::now());
```

7. 现在可以来获取用户的输入了。当我们没有获取成功，将会返回一个默认的元组对象。这个元组对象中的元素都是空：

```
if (!(cin >> s)) {
    return { {}, {} };
}
```

8. 成功获取输入后，我们会打上下一个时间戳。然后，返回用户的输入和输入所用的时间。注意这里获取的都是绝对的时间戳，通过计算这两个时间戳的差，我们得到了打印所用的时间：

```
const auto toc (chrono::steady_clock::now());

return {s, toc - tic};
```

9. 现在让我们来实现主函数，使用一个循环获取用户的输入，直到用户输入正确的字符串为止。在每次循环中，我们都会让用户输入"C++17"，然后调用 `get_input` 函数：

```
int main()
{
    while (true) {
        cout << "Please type the word \"C++17\" as"
             " fast as you can.\n> ";

        const auto [user_input, diff] = get_input();
```

10. 然后对输入进行检查。当输入为空，程序会终止：

```
if (user_input == "") { break; }
```

11. 当用户正确的输入"C++17"，我们将会对用户表示祝贺，然后返回其输入所用时间。 `diff.count()` 函数会以浮点数的方式返回输入所用的时间。当我们使用 STL 原始的 `seconds` 时间类型时，将会得到一个已舍入的整数，而不是一个小数。通过使用以毫秒和微秒为单位的计时，我们将获得对应单位的计数，然后通过相应的转换方式进行时间单位转换：

```
if (user_input == "C++17") {
    cout << "Bravo. You did it in:\n"
    << fixed << setprecision(2)
    << setw(12) << diff.count()
    << " seconds.\n"
    << setw(12) <<
milliseconds(diff).count()
    << " milliseconds.\n"
    << setw(12) <<
microseconds(diff).count()
    << " microseconds.\n";
break;
```

12. 如果用户输入有误时，我们会提示用户继续输入：

```
} else {
    cout << "Sorry, your input does not
match."
    " You may try again.\n";
}
}
```

13. 编译并运行程序，就会得到如下的输出。第一次输入时，会有一个错误，程序会让我们重新进行输入。在正确输入之后，我们就会得到输入所花费的时间：

```
$ ./ratio_conversion
Please type the word "C++17" as fast as you can.
> c+17
Sorry, your input does not match. You may try again.
Please type the word "C++17" as fast as you can.
> C++17
Bravo. You did it in:
2.82 seconds.
2817.95 milliseconds.
2817948.40 microseconds.
```

How it works...

本节中对不同时间单位进行转换是，我们需要先选择三个可用的时钟对象的一个。其分别为 `system_clock`，`steady_clock` 和 `high_resolution_clock`，这三个时钟对象都在 `std::chrono` 命名空间中。他们有什么区别呢？让我们来看一下：

时钟类型	特性
<code>system_clock</code>	表示系统级别的实时挂钟。想要获取本地时间的话，这是个正确的选择。
<code>steady_clock</code>	表示单调型的时间。这个时间是不可能倒退的，而时间倒退可能会在其他时钟上发生，比如：其最小精度不同，或是在冬令时和夏令时交替时。
<code>high_resolution_clock</code>	STL中可统计最细粒度时钟周期的时钟。

当我们衡量时间的“距离”，或者计算两个时间点的绝对间隔。即便时钟是112年，5小时，10分钟，1秒(或其他)之后或之前的时间，这都不影响两个时间点间的相对距离。这里我们唯一关注的就是打的两个时间点 `toc` 和 `tic`，时钟需要是微秒级别的(许多系统都使用这样的时钟)，因为不同的时钟对于我们的测量有一定的影响。对于这样的需求，`steady_clock` 无疑是最佳的选择。其能根据处理器的时间戳计数器进行实现，只要该时钟开始计数(系统开始运行)就不会停止。

OK，现在来对合适的时间对象进行选择，可以通过 `chrono::steady_clock::now()` 对时间点进行保存。`now` 函数会返回一个 `chrono::time_point<chrono::steady_clock>` 类的值。两个点之间的差就是所用时间间隔，或 `chrono::duration` 类型的时间长度。这个类型是本节的核心类型，其看起来有点复杂。让我们来看一下 `duration` 模板类的签名：

```
template<
    class Rep,
    class Period = std::ratio<1>
> class duration;
```

我们需要改变的参数类为 `Rep` 和 `Period`。`Rep` 很容易解释：其只是一个数值类型用来保存时间点的值。对于已经存在的STL时间单位，都为 `long long int` 型。本节中，我们选择了 `double`。因为我们的选择，保存的时间描述也可以转换为毫秒或微秒。当 `chrono::seconds` 类型记录的时间为1.2345秒时，其会舍入成一个整数秒数。这样，我们就能使用 `chrono::microseconds` 来保存 `tic` 和 `toc` 之间的时间，并且将其转化为粒度更加大的时间。正因为选择 `double` 作为 `Rep` 传入，可以对计时的精度在丢失较少精度的情况下，进行向上或向下的调整。

对于我们的计时单位，我们采取了 `Rep = double` 方式，所以会在 `Period` 上有不同的选择：

```
using seconds = chrono::duration<double>;
using milliseconds = chrono::duration<double,
    ratio_multiply<seconds::period, milli>>;
using microseconds = chrono::duration<double,
    ratio_multiply<seconds::period, micro>>;
```

`seconds` 是最简单的时间单位，其为 `Period = ratio<1>`，其他的时间单位就只能进行转换。1毫秒是千分之一秒，所以我们将使用 `milli` 特化的 `seconds::period` 转化为秒时，就要使用 `std::ratio<1, 1000>` 类型(`std::ratio<a, b>` 表示分数值 a/b)。`ratio_multiply` 类型是一个编译时函数，其表示对应类型的结果是多个 `ratio` 值累加。

可能这看起来非常复杂，那就让我们来看一个例子吧：`ratio_multiply<ratio<2, 3>, ratio<4, 5>>` 的结果为 `ratio<8, 15>`，因为 $(2/3) * (4/5) = 8/15$ 。

我们结果类型定义等价情况如下：

```
using seconds = chrono::duration<double, ratio<1, 1>>;
using milliseconds = chrono::duration<double, ratio<1,
1000>>;
using microseconds = chrono::duration<double, ratio<1,
1000000>>;
```

上面列出的类型，很容易的就能进行转换。当我们具有一个时间间隔 `d`，其类型为 `seconds`，我们就能将其转换成 `milliseconds`。转换只需要通过构造函数就能完成——`milliseconds(d)`。

There's more...

其他教程和书籍中，你可以会看到使用 `duration_cast` 的方式对时间进行转换。当我们具有一个时间间隔类 `chrono::milliseconds` 和要转换成的类型 `chrono::hours` 时，就需要转换为 `duration_cast<chrono::hours>(milliseconds_value)`，因为这些时间单位都是整型。从一个细粒度的时间单位，转换成一个粗粒度的时间单位，将会带来时间精度的损失，这也是为什么我们使用 `duration_cast` 的原因。基于 `double` 和 `float` 的时间间隔类型不需要进行强制转换。

转换绝对时间和相对时间—— std::chrono

C++11之前，想要获取时间并对其进行打印是有些困难的，因为C++并没有标准时间库。想要对时间进行统计就需要调用C库，并且我们要考虑这样的调用是否能很好的封装到我们的类中。

C++11之后，STL提供了 chrono 库，其让对时间的操作更加简单。

本节，我们将会使用本地时间，并对本地时间进行打印，还会给时间加上不同的偏移，这些操作很容易使用 std::chrono 完成。

How to do it...

我们将会对当前时间进行保存，并对其进行打印。另外，我们的程序还会为已经保存的时间点添加不同的偏移，并且打印偏移之后的时间：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <iomanip>
#include <chrono>

using namespace std;
```

2. 我们将打印绝对时间点。使用 chrono::time_point 模板类型来获取，因此需要对输出流操作符进行重载。对时间点的打印方式有很多，我们将会使用 %c 来表示标准时间格式。当然，可以只打印时间、日期或是我们需要的信息。调用 put_time 之前对不同类型的变量进行转换的方式看起来有些笨拙，不过这里只这么做一次：

```
ostream& operator<<(ostream &os,
    const chrono::time_point<chrono::system_clock>
    &t)
{
    const auto tt
        (chrono::system_clock::to_time_t(t));
    const auto loct (std::localtime(&tt));
    return os << put_time(loct, "%c");
}
```

3. STL已经定义了 seconds , minutes , hours 等时间类型，所以我们只需要为其添加 days 类型就好。这很简单，只需要对 chrono::duration 模板类型进行特化，将hours类型乘以24，就表示一天具有24个小时：

```
using days = chrono::duration<
    chrono::hours::rep,
    ratio_multiply<chrono::hours::period,
    ratio<24>>>;
```

4. 为了用很有优雅的方式表示很多天，我们定义属于 `days` 类型的字面值操作符。现在我们程序中写 `3_days` 就代表着3天：

```
constexpr days operator ""_days(unsigned long long d)
{
    return days{d};
}
```

5. 实际程序中，我们将会对时间点进行记录，然后就会对这个时间点进行打印。因为已经对操作符进行了重载，所以完成这样的事情就变得很简单：

```
int main()
{
    auto now (chrono::system_clock::now());

    cout << "The current date and time is " << now <<
    '\n';
```

6. 我们使用 `now` 函数来获得当前的时间点，并可以为这个时间添加一个偏移，然后再对其进行打印。为当前的时间添加12个小时，其表示的为12个小时之后的时间：

```
chrono::hours chrono_12h {12};

cout << "In 12 hours, it will be "
<< (now + chrono_12h)<< '\n';
```

7. 这里将使用 `chrono_literals` 命名空间中的函数，声明使用这个命名空间会解锁小时、秒等等时间类型的间隔字面值类型。这样我们就能很优雅的对12个小时15分之前的时间或7天之前的时间进行打印：

```
using namespace chrono_literals;

cout << "12 hours and 15 minutes ago, it was "
<< (now - 12h - 15min) << '\n'
<< "1 week ago, it was "
<< (now - 7_days) << '\n';
}
```

8. 编译并运行程序，我们将会获得如下的输出。因为使用 `%c` 格式对时间进行打印，所以得到还不错的时间输出格式。通过对不同格式的字符串进行操作，我们可以获得想要的格式。要注意的是，这里的时间格式并不是以12小时 AM/PM 方式表示，因为程序运行在欧洲操作系统上，所以使用24小时表示的方式：

```
$ ./relative_absolute_times
The current date and time is Fri May 5 13:20:38 2017
In 12 hours, it will be Sat May 6 01:20:38 2017
12 hours and 15 minutes ago, it was Fri May 5 01:05:38
2017
1 week ago, it was Fri Apr 28 13:20:38 2017
```

How it works...

我们可以通过 `std::chrono::system_clock` 来获取当前时间点。这个STL时钟类是唯一一个能将时间点的值转换成一个时间结构体的类型，其能将时间点以能够看懂的方式进行输出。

为了打印这样的时间点，我们可以对 `operator<<` 操作符进行重载：

```
ostream& operator<<(ostream &os,
                      const
 chrono::time_point<chrono::system_clock> &t)
{
    const auto tt (chrono::system_clock::to_time_t(t));
    const auto loct (std::localtime(&tt));
    return os << put_time(loct, "%c");
}
```

首先，将 `chrono::time_point<chrono::system_clock>` 转换为 `std::time_t`。然后，使用 `std::localtime` 将这个时间值进行转换，这样就能获取到一个本地时钟的相对时间值。这个函数会给我们返回一个转换后的指针(对于这个指针背后的内存不用我们多操心，因为其是一个静态对象，并不是从堆上分配的内存)，这样我们就能完成最终的打印。

`std::put_time` 函数接受一个流对象和一个时间格式字符串。`%c` 表示标准时间格式字符串，例如 `Sun Mar 12 11:33:40 2017`。我们也可以写成 `%m/%d/%y`；之后，时间就会按照这个格式进行打印，比如 `03/12/17`。时间格式符的描述很长，想要了解其具体描述的最好方式就是去查看C++参考手册。

除了打印，我们也会为我们的时问点添加偏移。这也很简单，比如：12小时15分钟就可以表示为 `12h+15min`。`chrono_literals` 命名空间为我们提供了字面类型：`hours(h), minutes(min), seconds(s), milliseconds(ms), microseconds(us), nanoseconds(ns)`。

通过对两个时间间隔的相加，我们会得到一个新的时间点。要实现这样的操作就需要对左加 `operator+` 和左减 `operator-` 操作符进行重载，这样对时间偏移的操作就会变得非常简单。

安全的标识失败——`std::optional`

当程序与外界的联系只依赖于一些变量时，那么各种失败都可能发生。

也就是，我们写了一个函数，其会返回一个值，但是当函数接口进行变更后，可能就无法获取这个返回值了。我们来看下对一个返回字符串的函数，怎样的接口会容易出现失败的情况：

- 使用引用值作为返回值： `bool get_string(string&);`
- 返回一个可以被设置为`nullptr`的指针(或智能指针)： `string* get_string();`
- 当函数出错时，直接抛出异常： `string get_string();`

以上的方式有缺点，也有优点。在C++17之后，我们会使用一种新类型来解决这个问题：`std::optional`。可选值的概念来自于纯函数式编程语言(在纯函数式语言中，这个类型为**Maybe**类型)，并且可以让代码看上去很优雅。

我们可以将 `optional` 包装到我们的类型中，其可以表示空值或错误值。本节中，我们就会来学习怎么使用这个类型。

How to do it...

本节，我们将实现一个程序用于从用户输入中读取整型数，然后将这些数字加起来。因为不确定用户会输入什么，所以我们会使用 `optional` 进行错误处理：

1. 包含必要的头文件，并声明所使用的命名空间。

```
#include <iostream>
#include <optional>

using namespace std;
```

2. 定义一个整型类型，其可能会包含一个值，使用 `std::optional` 类型来完成这件事。将目标类型包装进 `optional`，我们会给其一个附加状态，其表示当前对象中没有值：

```
using oint = optional<int>;
```

3. 使用包装后的整型类型，我们用其来表示函数返回失败的情况。当从用户输入中获取一个整数时，这个函数可能会失败，因为用户可能输入的就不是我们想要的东西，返回可选整型就能很好的解决这个问题。当成功的读取一个整数，我们会将其放入 `optional<int>` 的构造函数中。否则，我们将返回一个默认构造的 `optional`，其代表没有获取成功：

```

oint read_int()
{
    int i;
    if (cin >> i) { return {i}; }
    return {};
}

```

4. 除了获取整数，我们还能做的更多。那怎么使用两个可选整数进行相加呢？如果两个可选整数中具有相应的整数值，那么使用实际的数值直接相加。存在有空的可选变量时，我们会返回一个空的可选变量。这个函数需要简单的来解释一下：通过隐式转换，将 `optional<int>` 变量 `a` 和 `b` 转化成一个布尔表达式(写成`!a`和`!b`)，这就能让我们确定可选变量中是否有值。如果其中有值，我们将对其使用指针或是迭代器的方式，对 `a` 和 `b` 直接解引用：

```

oint operator+(oint a, oint b)
{
    if (!a || !b) { return {};}
    return {*a + *b};
}

```

5. 重载加法操作，可以直接和一个普通整数进行相加：

```

oint operator+(oint a, int b)
{
    if (!a) { return {};}

    return {*a + b};
}

```

6. 现在来完成主函数部分，我们会让用户输入两个数值：

```

int main()
{
    cout << "Please enter 2 integers.\n> ";

    auto a {read_int()};
    auto b {read_int()};
}

```

7. 然后，将获取的数值进行相加，并再与 `10` 进行相加。这里 `a` 和 `b` 为可选整型类变量，`sum` 也为可选整型类变量：

```

auto sum (a + b + 10);

```

8. 当 `a` 和/或 `b` 中不包含一个值时，`sum` 就也不包含任何值。可选整型可依然我们不必显式的对 `a` 和 `b` 进行检查。当遇到空值的时，我们定义的操作符能很完美的处理这样的情况。这样，我们只需要对结果可选整型变量进行检查即

可。如果包含一个值，那就可以安全的对这个值进行访问，并将其进行打印：

```
if (sum) {
    cout << *a << " + " << *b << " + 10 = "
    << *sum << '\n';
```

9. 当用户输入了非数字内容，我们将会输出错误信息：

```
} else {
    cout << "sorry, the input was "
        "something else than 2 numbers.\n";
}
```

10. 完成了！编译并运行程序，我们将会得到如下输出：

```
$ ./optional
Please enter 2 integers.
> 1 2
1 + 2 + 10 = 13
```

11. 当输入中包含非数字元素，我们将会得到如下输出：

```
$ ./optional
Please enter 2 integers.
> 2 z
sorry, the input was something else than 2 numbers.
```

How it works...

`optional` 非常简单易用。其可以帮助我们对错误的情况进行处理，当我们所需要的类型为T时，可以将其特化 `std::optional<T>` 版本类型进行封装。

当需要从一些地方获取一些值时，我们可以用其来检查我们是否成功的获取了对应的数值。`bool optional::has_value()` 可以帮助我们完成这件事。当其包含值时，其会返回`true`，我们就能直接对数值进行访问，对可选类型的值访问也可以通过函数 `T& optional::value()` 进行。

例子中，使用 `if (x) {...}` 和 `*x` 来替代 `if (x.has_value())` `{...}` 和 `x.value()`。`std::optional` 类型可以隐式的转换成 `bool` 类型，并且使用解引用操作符的方式和普通指针差不多。

另一个方便辅助操作符就是对 `optional` 的 `operator->` 操作符进行重载。当有一个结构体 `struct Foo { int a; string b; }` 类型，并且我们想要通过一个 `optional<Foo>` 来访问其成员变量`x`，那么就可以写成 `x->a` 或 `x->b`。当然，需要对`x`和`b`进行检查，确定其是否有值。

当可选变量中没有值时，我们还要对其进行访问，其就会抛出一个 `std::logic_error` 异常。这样，就可以对大量的可选值在不进行检查的情况下进行使用。`try-catch` 块的代码如下：

```
cout << "Please enter 3 numbers:\n";

try {
    cout << "Sum: "
    << (*read_int() + *read_int() + *read_int())
    << '\n';
} catch (const std::bad_optional_access &) {
    cout << "Unfortunately you did not enter 3
numbers\n";
}
```

`std::optional` 具有一个有趣的 `optional::value_or` 操作。当我们想要在失败的时候，可选变量包含一个默认值进行返回时，这个操作就很有用了。`x = optional_var.value_or(123)` 就能将123作为可选变量失败时的默认数值。

对元组使用函数

C++11中，STL添加了 `std::tuple`，这种类型可以用来将多个不同类型的值捆绑在一起。元组这种类型已经存在与很多编程语言中，本书的一些章节已经在使用这种类型，这种类型的用途很广泛。

不过，我们有时会将一些值捆绑在一个元组中，然后我们需要调用函数来获取其中每一个元素。对于元素的解包的代码看起来非常的冗长(并且易于出错)。其冗长的方式类似这样：`func(get<0>(tup), get<1>(tup), get<2>(tup), ...);`。

本节中，你将了解如何使用一种优雅地方式对元组进行打包和解包。调用函数时，你无需对元组特别地了解。

How to do it...

我们将实现一个程序，其能对元组值进行打包和解包。然后，我们将看到在不了解元组中元素的情况下，如何使用元组：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <iomanip>
#include <tuple>
#include <functional>
#include <string>
#include <list>

using namespace std;
```

2. 首先定义一个函数，这个函数能接受多个参数，其描述的是一个学生，并将学生相关信息进行打印。其和C风格的函数看起来差不多：

```
static void print_student(size_t id, const string
    &name, double gpa)
{
    cout << "Student " << quoted(name)
        << ", ID: " << id
        << ", GPA: " << gpa << '\n';
}
```

3. 主函数中，将对一种元组类型进行别名，然后将具体学生的信息填入到这种类型的实例中：

```
int main()
{
    using student = tuple<size_t, string, double>;
    student john {123, "John Doe"s, 3.7};
```

4. 为了打印这种类型的实例，我们将会对元组中的元素进行分解，然后调用 `print_student` 函数将这些值分别进行打印：

```
{
    const auto &[id, name, gpa] = john;
    print_student(id, name, gpa);
}
cout << "----\n";
```

5. 然后，我们来创建一个以元组为基础类型的多个学生：

```
auto arguments_for_later = {
    make_tuple(234, "John Doe"s, 3.7),
    make_tuple(345, "Billy Foo"s, 4.0),
    make_tuple(456, "Cathy Bar"s, 3.5),
};
```

6. 这里，我们依旧可以通过对元素进行分解，然后对其进行打印。当要写这样的代码时，我们需要在函数接口变化时，对代码进行重构：

```
for (const auto &[id, name, gpa] :
    arguments_for_later) {
    print_student(id, name, gpa);
}
cout << "----\n";
```

7. 当然可以做的更好，我们无需知道 `print_student` 的参数的个数，或学生元组中元素的个数，我们使用 `std::apply` 对直接将元组应用于函数。这个函数能够接受一个函数指针或一个函数对象和一个元组，然后会将元组进行解包，然后与函数参数进行对应，并传入函数：

```
apply(print_student, john);
cout << "----\n";
```

8. 循环中可以这样用：

```
for (const auto &args : arguments_for_later) {
    apply(print_student, args);
}
cout << "----\n";
```

9. 编译并运行程序，我们就能得到如下的输出：

```
$ ./apply_functions_on_tuples
Student "John Doe", ID: 123, GPA: 3.7
-----
Student "John Doe", ID: 234, GPA: 3.7
Student "Billy Foo", ID: 345, GPA: 4
Student "Cathy Bar", ID: 456, GPA: 3.5
-----
Student "John Doe", ID: 123, GPA: 3.7
-----
Student "John Doe", ID: 234, GPA: 3.7
Student "Billy Foo", ID: 345, GPA: 4
Student "Cathy Bar", ID: 456, GPA: 3.5
-----
```

How it works...

`std::apply` 是一个编译时辅助函数，可以帮助我们处理不确定的类型参数。

试想，我们有一个元组 `t`，其有元素 `(123, "abc"s, 456.0)`。那么这个元组的类型为 `tuple<int, string, double>`。另外，有一个函数 `f` 的签名为 `int f(int, string, double)`（参数类型也可以为引用）。

然后，我们就可以这样调用函数 `x = apply(f, t)`，其和 `x = f(123, "abc"s, 456.0)` 等价。`apply` 方法还是会返回 `f` 的返回值。

使用元组快速构成数据结构

我们已经在上一节中了解元组的基本使用方法。现在我们使用一个结构体，对一些变量进行捆绑：

```
struct Foo {  
    int a;  
    string b;  
    float c;  
};
```

之前的章节中，为了替代定义结构体，我们可以定义一个元组：

```
using Foo = tuple<int, string, float>;
```

我们可以根据类型列表中的索引，从而获取相应变量的具体值。如要访问元组他的第一个类型变量，可以使用 `std::get<0>(t)`，第二个类型变量为 `std::get<1>(t)`，以此类推。如果索引值过大，编译器会在编译时进行报错。

之前的章节中已经展示了C++17对元组的分解能力，允许我们使用如下的方式快速分解元素，并能对单独元素进行访问：

```
auto [a, b, c] = some_tuple
```

绑定和分解单个数据结构，只是元组能力之一。我们也可以想尽办法对元组进行连接和分割。本节中，我们将学习如何完成这样的任务。

How to do it...

本节，我们将完成对任意元组进行打印的任务。另外，我们将完成一个函数，可以对元组进行zip操作：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>  
#include <tuple>  
#include <list>  
#include <utility>  
#include <string>  
#include <iterator>  
#include <numeric>  
#include <algorithm>  
  
using namespace std;
```

2. 我们对要处理的元组中的内容非常感兴趣，所以想要对其内容进行展示。因此，将实现一个非常通用的函数，能对任意具有可打印变量的元组进行打印。这个函数能接受一个输出流引用 `os` 和一个可变的参数列表，其中具有元组中的所有成员。为了解析这些参数，我们将一个参数放在 `v` 中，其余的放在参数包 `vs...` 中：

```
template <typename T, typename ... Ts>
void print_args(ostream &os, const T &v, const Ts
&... vs)
{
    os << v;
```

3. 然后，我们就对参数包 `vs` 进行处理，其会使用逗号将 `initializer_list` 中的元素进行隔开。你可以回看一下第4章的[使用同一输入调用多个函数](#)，了解下如何使用Lambda表达式来完成这个操作：

```
(void) initializer_list<int>{((os << ", " << vs),
0) ...};
```

4. 现在就可以对任意的变量进行打印了，例如：`print_args(cout, 1, 2, "foo", 3, "bar")`。不过，依旧无法对元组进行处理。为了实现打印元组的功能，我们会对输出操作符 `<<` 进行重载，通过实现一个模板函数来匹配任意元组类型：

```
template <typename ... Ts>
ostream& operator<<(ostream &os, const tuple<Ts...>
&t)
{
```

5. 接下来会有些复杂。首先，使用Lambda表达式来接收任意多个参数。当调用 Lambda 表达式时，启用 `os` 参数就会传入 `print_args` 函数中组成新的参数列表。这也就意味着，对 `capt_tup(...some parameters...)` 的调用，会形成对 `print_args(os, ...some parameters...)` 的调用：

```
auto print_to_os ([&os] (const auto &...xs) {
    print_args(os, xs...);
});
```

6. 现在就可以来完成对元组解包的工作了。使用 `std::apply` 对元组进行解包，所有值将会解析成单独的变量，然后传入到所调用的函数中。当元组 `t` 为 `(1, 2, 3)` 时，调用 `apply(capt_tup, t)` 等价于 `capt_tup(1, 2, 3)` 的调用，随后就会调用 `print_args(os, 1, 2, 3)`。最后，我们用小括号来包围所要打印的信息：

```
os << "(";
apply(print_to_os, t);
return os << ")";
}
```

7. Okay, 现在已经完成打印元组代码的编写, 这将让后续的工作会变得更容易。不过, 需要为元组做更多的事情。例如, 编写一个可以接受迭代范围的函数。这个函数可以帮助对对应范围进行迭代, 然后返回这段范围中所有值的加和, 并且找到这个范围内的最小值和最大值, 还要能对所有值求平均。并将这四个值打包入一个元组中, 我们可以不添加任何新的结构体类型类, 来获取其中每一个成员的值:

```
template <typename T>
tuple<double, double, double, double>
sum_min_max_avg(const T &range)
{
```

8. `std::minmax_element` 会返回一对迭代器, 其分别表示输入范围内的最小值和最大值。`std::accumulate` 将会返回输入范围内所有值的加和。这样就能获得我们元组中的所有元素了!

```
    auto min_max (minmax_element(begin(range),
end(range)));
    auto sum (accumulate(begin(range), end(range),
0.0));
    return {sum, *min_max.first, *min_max.second,
            sum / range.size()};
}
```

9. 实现主函数之前, 我们将实现最后一个神奇辅助函数。为什么说这个函数神奇呢? 因为这个函数看起来非常复杂, 但当了解工作原理后, 你就能理解了, 这个函数会对两个元组进行`zip`操作。也就是说, 当传入两个元组 `(1, 2, 3)` 和 `('a', 'b', 'c')` 时, 函数将会返回一个值为 `(1, 'a', 2, 'b', 3, 'c')` 的元组:

```
template <typename T1, typename T2>
static auto zip(const T1 &a, const T2 &b)
{
```

10. 接下来, 我们将会看到本节中最为复杂的几行代码。我们会创建一个函数对象 `z`, 其能接受任意数量的参数。其会返回另一个函数对象, 返回的函数对象将获取的所有参数打包成 `xs`, 不过其也能接受任意数量的参数。其内部的函数对象可以对参数列表包 `xs` 和 `ys` 进行访问。现在就让我们看一下, 如何对这两个参数列表包进行操作。`make_tuple(xs, ys)...` 会将参数分组。当 `xs = 1, 2, 3` 并且 `ys = 'a', 'b', 'c'` 时, 我们将会返回一个新的参数包 `(1, 'a'), (2, 'b'), (3, 'c')`。三个元组中, 用逗号来对每个成员进行区分。为了获取分组后的元组, 我们使用了 `std::tuple_cat`, 其能接受任意数量的元组, 并且将其解包后放入一个元组中。这样我们就可以获得一个新元组 `(1, 'a', 2, 'b', 3, 'c')`:

```
auto z ([](auto ...xs) {
    return [xs...](auto ...ys) {
        return tuple_cat(make_tuple(xs, ys) ...);
    };
});
```

11. 最后一步就是将所有输入元组中的成员解包出来，也就是将 `a` 和 `b` 进行解包后放入 `z` 中。`apply(z, a)` 就表示将 `a` 中的所有值放入 `xs` 中，`apply(..., b)` 就表示将 `b` 中的所有值放入 `ys` 中。最后的结果元组就是 `zip` 后的一个非常大的元组，其会返回给调用者：

```
return apply(apply(z, a), b);
```

12. 我们写了非常多的辅助代码。现在，我们就来使用这些辅助函数。首先，构造出一些元组。`student` 类型包括 ID，名字，和 GPA 分数。`student_desc` 使用人类可读的格式对学生进行介绍。`std::make_tuple` 是一个非常不错的工厂函数，因为其能通过传入的参数，自适应的生成对应的元组类型：

```
int main()
{
    auto student_desc (make_tuple("ID", "Name",
"GPA"));
    auto student (make_tuple(123456, "John Doe",
3.7));
```

13. 我们对这些信息进行打印。因为已经对输出流操作符进行过重载，所以打印并不是什么难事：

```
cout << student_desc << '\n'
<< student << '\n';
```

14. 我们也可以通过 `std::tuple_cat` 将所有元组进行连接，然后进行打印：

```
cout << tuple_cat(student_desc, student) << '\n';
```

15. 我们有可以通过我们的 `zip` 函数创建新的元组：

```
auto zipped (zip(student_desc, student));
cout << zipped << '\n';
```

16. 别忘记 `sum_min_max_avg` 函数。我们将初始化列表中具有一些数字，并且会将这些数字传入这个函数中。创建了另一个同等大小的元组，其包含了一些描述字符串，这可能会让程序变得复杂一些。通过 `zip` 这些元组，并将这些元组交错的存储在了一起：

```

    auto numbers = {0.0, 1.0, 2.0, 3.0, 4.0};
    cout << zip(
        make_tuple("Sum", "Minimum", "Maximum",
        "Average"),
        sum_min_max_avg(numbers))
    << '\n';
}

```

17. 编译并运行程序，我们就会得到如下输出。前两行

是 `student` 和 `student_desc` 元组的打印结果。第3行是使用 `tuple_cat` 组合后的输出结果。第4行是将学生元组进行 `zip` 后的结果。最后一行我们将会看到对应数字列表的和值、最小值、最大值和均值。因为有 `zip` 操作，我们可以清楚地了解这些数字的意义：

```

$ ./tuple
(ID, Name, GPA)
(123456, John Doe, 3.7)
(ID, Name, GPA, 123456, John Doe, 3.7)
(ID, 123456, Name, John Doe, GPA, 3.7)
(Sum, 10, Minimum, 0, Maximum, 4, Average, 2)

```

How it works...

本节的有些代码的确比较复杂。我们对元组的 `operator<<` 操作符进行了重载实现，这样看起来比较复杂，但是这样就能对元组中的成员进行打印。然后我们实现 `sum_min_max_avg` 函数，其会返回一个元组。另外，`zip` 应该是个比较复杂的函数。

这里最简单的函数是 `sum_min_max_avg`。当我们定义一个函数 `tuple<Foo, Bar, Baz> f()` 时，我们可以将返回语句写成 `return {foo_instance, bar_instance, baz_instance};`，这样函数将会自动的构建一个元组进行返回。如果你对 `sum_min_max_avg` 中所使用的STL函数有疑问，那可以回看一下第5章，其中有一些STL的基本函数操作。

其他较为复杂的部分，就是一些辅助函数：

`operator<< for tuples`

使用 `operator<<` 对输出流进行输出时，我们实现了 `print_args` 函数。其可以接受任意个参数，不过第一个参数必须是一个 `ostream` 实例：

```

template <typename T, typename ... Ts>
void print_args(ostream &os, const T &v, const Ts &...vs)
{
    os << v;

    (void)initializer_list<int>{((os << ", " << vs),
0)...};
}

```

这个函数打印的第一个元素是 `v`，然后会将参数包 `vs` 中的元素进行打印。我们将第一个元素单独拎出来的原因是要使用逗号将所有元素进行分隔，但是我们不确定哪个参数是头或是尾(也就是要打印成“1, 2, 3”或是“1, 2, 3”)。我们在第4章了解到使用Lambda表达式对 `initializer_list` 进行扩展，也就是[使用同一输入调用多个函数](#)这一节。这个函数，就能帮我们对元组进行打印。`operator<<` 实现如下所示：

```

template <typename ... Ts>
ostream& operator<<(ostream &os, const tuple<Ts...> &t)
{
    auto capt_tup ([&os] (const auto &...xs) {
        print_args(os, xs...);
    });

    os << "(";
    apply(capt_tup, t);
    return os << ")";
}

```

首先我们定义了一个函数对象 `capt_tup`。当我们调用 `capt_tup(foo, bar, whatever)` 时，其实际调用的是 `print_args(os, foo, bar, whatever)`。这个函数只会做一件事，就是将可变列表中的参数输出到输出流对象 `os` 中。

之后，我们使用 `std::apply` 对元组 `t` 进行解包。如果这步看起来很复杂，那么可以看一下前一节，以了解 `std::apply` 的工作原理。

元素的zip函数

`zip` 函数能够接收两个元组，虽然其实现很清晰，但是看起来还是异常复杂：

```

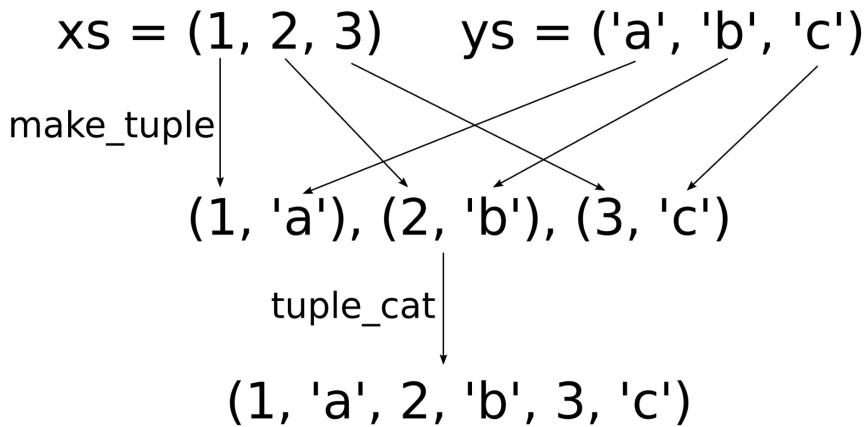
template <typename T1, typename T2>
auto zip(const T1 &a, const T2 &b)
{
    auto z ([](auto ...xs) {
        return [xs...](auto ...ys) {
            return tuple_cat(make_tuple(xs, ys) ...);
        };
    });
    return apply(apply(z, a), b);
}

```

为了能更好的了解这段代码，我们可以假设有两个元组，一个元组`a`为`(1, 2, 3)`，另一个元组`b`为`('a', 'b', 'c')`。

例程中，我们调用了 `apply(z, a)`，也就相当于调用函数 `z(1, 2, 3)`，其会构造一个哈数对象将这些参数捕获后进行返回，这样`1, 2, 3`就被放入参数包 `xs` 中了。这里会再次调用，`apply(z(1, 2, 3), b)`，会将`'a', 'b', 'c'`放入参数包 `ys` 中。

Okay，现在 `xs = (1, 2, 3)`，`ys = ('a', 'b', 'c')`，然后会发生什么呢？`tuple_cat(make_tuple(xs, ys) ...)` 就会完成下图所描述的过程：



首先，`xs` 和 `ys` 中的成员将会被 `zip` 到一起，也就是交叉配对在一起。这个交叉配对发生在 `make_tuple(xs, ys)...` 部分，这会将两个元组组成一个元组。为了获得一个大元组，我们使用了 `tuple_cat`，通过对元组的级联获取一个大的元组，其包含了所有元组中的成员，并进行了交叉配对。

将 `void*` 替换为更为安全的`std::any`

有时我们会需要将一个变量保存在一个未知类型中。对于这样的变量，我们通常会对其进行检查，以确保其是否包含一些信息，如果是包括，那我们将会去判别所包含的内容。以上的所有操作，都需要在一个类型安全的方法中进行。

以前，我们会将可变对象存与 `void*` 指针当中。`void`类型的指针无法告诉我们其所指向的对象类型，所以我们需要将其进行手动转换成我们期望的类型。这样的代码看起来很诡异，并且不安全。

C++17在STL中添加了一个新的类型——`std::any`。其设计就是用来持有任意类型的变量，并且能提供类型的安全检查和安全访问。

本节中，我们将会来感受一下这种工具类型。

How to do it...

我们将实现一个函数，这个函数能够打印所有东西。其就使用 `std::any` 作为参数：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <iomanip>
#include <list>
#include <any>
#include <iterator>

using namespace std;
```

2. 为了减少后续代码中尖括号中的类型数量，我们对 `list<int>` 进行了别名处理：

```
using int_list = list<int>;
```

3. 让我们实现一个可以打印任何东西的函数。其确定能打印任意类型，并以 `std::any` 作为其参数：

```
void print_anything(const std::any &a)
{
```

4. 首先，要做的事就是对传入的参数进行检查，确定参数中是否包含任何东西，还是只是一个空实例。如果为空，那就没有必要再进行接下来的打印了：

```
if (!a.has_value()) {
    cout << "Nothing.\n";
```

5. 当非空时，就要需要对其进行类型比较，直至匹配到对应类型。这里第一个类型为 `string`，当传入的参数是一个 `string`，我们可以使用 `std::any_cast` 将 `a` 转化成一个 `string` 类型的引用，然后对其进行打印。我们将双引号当做打印字符串的修饰：

```
    } else if (a.type() == typeid(string)) {
        cout << "It's a string: "
        << quoted(any_cast<const string&>(a)) <<
        '\n';
```

6. 当其不是 `string` 类型时，其也可能是一个 `int` 类型。当与之匹配是使用 `any_cast<int>` 将 `a` 转换成 `int` 型数值：

```
    } else if (a.type() == typeid(int)) {
        cout << "It's an integer: "
        << any_cast<int>(a) << '\n';
```

7. `std::any` 并不只对 `string` 和 `int` 有效。我们将 `map` 或 `list`，或是更加复杂的数据结构放入一个 `any` 变量中。让我们输入一个整数列表看看，按照我们的预期，函数也将会打印出相应的列表：

```
    } else if (a.type() == typeid(int_list)) {
        const auto &l (any_cast<const int_list&>(a));

        cout << "It's a list: ";
        copy(begin(l), end(l),
            ostream_iterator<int>(cout, ", "));
        cout << '\n';
```

8. 如果没有类型能与之匹配，那就不会进行猜测了。我们会放弃对类型进行匹配，然后告诉使用者，我们对输入毫无办法：

```
    } else {
        cout << "Can't handle this item.\n";
    }
}
```

9. 主函数中，我们能够对调用函数传入任何类型的值。我们可以使用大括号来构建一个空的 `any` 变量，或是直接输入字符串“abc”，或是一个整数。因为 `std::any` 可以由任何类型隐式转换而成，这里并没有语法上的开销。我们也可以直接构造一个列表，然后丢入函数中：

```
int main()
{
    print_anything({});
    print_anything("abc"s);
    print_anything(123);
    print_anything(int_list{1, 2, 3});
```

10. 当我们想要传入的参数比较大，那么拷贝到 `any` 变量中就会花费很长的时间，这是可以使用立即构造的方式。`in_place_type_t<int_list>{}` 表示一个空的对象，对于 `any` 来说其就能够知道应该如何去构建对象了。第二个参数为 `{1,2,3}` 其为一个初始化列表，其会用来初始化 `int_list` 对象，然后被转换成 `any` 变量。这样，我们就避免了不必要的拷贝和移动：

```
print_anything(any(in_place_type_t<int_list>{},
{1, 2, 3}));
```

11. 编译并运行程序，我们将得到如下的输入出：

```
$ ./any
Nothing.
It's a string: "abc"
It's an integer: 123
It's a list: 1, 2, 3,
It's a list: 1, 2, 3,
```

How it works...

`std::any` 类型与 `std::optional` 类型很类似——具有一个 `has_value()` 成员函数，能告诉我们其是否携带一个值。不过这里，我们还需要对字面的数据进行保存，所以 `any` 要比 `optional` 类型复杂的多。

访问`any`变量的内容前，我们需要知道其所承载的类型，然后将 `any` 变量转换成那种类型。

这里，使用的比较方式为 `x.type == typeid(T)`。如果比较结果匹配，那么就使用 `any_cast` 对其内容进行转换。

需要注意的是 `any_cast<T>(x)` 将会返回 `x` 中 `T` 值的副本。如果想要避免对复杂对象不必要的拷贝，那就需要使用 `any_cast<T&>(x)`。本节的代码中，我们使用引用的方式来获取 `string` 和 `list<int>` 对象的值。

Note:

如果 `any` 变量转换成为一种错误的类型，其将会抛出 `std::bad_any_cast` 异常。

存储不同的类型——`std::variant`

C++中支持使用 `struct` 和 `class` 的方式将不同类型的变量进行包装。当我们想要使用一种类型来表示多种类型时，也可以使用 `union`。不过 `union` 的问题在于我们无法知道，其是以哪种类型为基础进行的初始化。

看一下下面的代码：

```
union U {
    int a;
    char *b;
    float c;
};

void func(U u) { std::cout << u.b << '\n'; }
```

当我们调用 `func` 时，其会将已整型 `a` 为基础进行初始化的联合体 `t` 进行打印，当然也无法阻止我们对其他成员进行访问，就像使用字符串指针对成员 `b` 进行初始化了一样，这段代码会引发各种bug。当我们开始对联合体进行打包之前，有一种辅助变量能够告诉我们其对联合体进行的初始化是安全的，其就是 `std::variant`，在 C++17 中加入 STL。

`variant` 是一种新的类型，类型安全，并高效的联合体类型。其不使用堆上的内存，所以在时间和空间上都非常高效。基于联合体的解决方案，我们就不用自己再去进行实现了。其能单独存储引用、数组或 `void` 类型的成员变量。

本节中，我们将会了解一下由 `variant` 带来的好处。

How to do it...

我们实现一个程序，其中有两个类型：`cat` 和 `dog`。然后将猫狗混合的存储于一个列表中，这个列表并不具备任何运行时多态性：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <variant>
#include <list>
#include <string>
#include <algorithm>

using namespace std;
```

2. 接下来，我们将实现两个具有类似功能的类，不过两个类型之间并没有什么联系。第一个类型是 `cat`。`cat` 对象具有名字，并能喵喵叫：

```

class cat {
    string name;

public:
    cat(string n) : name{n} {}

    void meow() const {
        cout << name << " says Meow!\n";
    }
};

```

3. 另一个类是 `dog`。 `dog` 能汪汪叫：

```

class dog {
    string name;

public:
    dog(string n) : name{n} {}

    void woof() const {
        cout << name << " says Woof!\n";
    }
};

```

4. 现在我们就可以来定义一个 `animal` 类型，其为 `std::variant<dog, cat>` 的别名类型。其和以前的联合体一样，同时具有 `variant` 的特性：

```
using animal = variant<dog, cat>;
```

5. 编写主函数之前，我们再来实现两个辅助者。其中之一为动物判断谓词，通过调用 `is_type<cat>(...)` 或 `is_type<dog>(...)`，可以判断动物实例中的动物为 `cat` 或 `dog`。其实现只需要对 `holds_alternative` 进行调用即可，其为 `variant` 类型的一个通用谓词函数：

```

template <typename T>
bool is_type(const animal &a) {
    return holds_alternative<T>(a);
}

```

6. 第二个辅助者为一个结构体，其看起来像是一个函数对象。其实际是一个双重函数对象，因为其 `operator()` 实现了两次。一种实现是接受 `dog` 作为参数输入，另一个实现是接受 `cat` 类型作为参数输入。对于两种实现，其会调用 `woof` 或 `meow` 函数：

```

struct animal_voice
{
    void operator() (const dog &d) const { d.woof(); }
    void operator() (const cat &c) const { c.meow(); }
};

```

7. 现在让我们使用这些辅助者和类型。首先，定义一个 `animal` 变量的实例，然后对其进行填充：

```

int main()
{
    list<animal> l {cat{"Tuba"}, dog{"Balou"},  

    cat{"Bobby"}};

```

8. 现在，我们会将列表的内容打印三次，并且每次都使用不同的方式。第一种使用 `variant::index()`。因为 `animal` 类型是 `variant<dog, cat>` 类型的别名，其返回值的0号索引代表了一个 `dog` 的实例。1号索引则代表了 `cat` 的实例。这里的关键是变量特化的顺序。`switch-cast` 代码块中，可以通过 `get<T>` 的方式获取内部的 `cat` 或 `dog` 实例：

```

for (const animal &a : l) {
    switch (a.index()) {
        case 0:
            get<dog>(a).woof();
            break;
        case 1:
            get<cat>(a).meow();
            break;
    }
}
cout << "-----\n";

```

9. 我们也可以显示的使用类型作为其索引。`get_if<dog>` 会返回一个指向 `dog` 类型的指针。如果没有 `dog` 实例在列表中，那么指针则为 `null`。这样，我们可以尝试获取下一种不同类型的实例，直到成功为止：

```

for (const animal &a : l) {
    if (const auto d (get_if<dog>(&a)); d) {
        d->woof();
    } else if (const auto c (get_if<cat>(&a)); c)
    {
        c->meow();
    }
}
cout << "-----\n";

```

10. 使用 `variant::visit` 是一种非常优雅的方式。这个函数能够接受一个函数对象和一个 `variant` 实例。函数对象需要对 `variant` 中所有可能类型进行重载。我们在之前已经对 `operator()` 进行了重载，所以这里可以直接对其进行使用：

```
for (const animal &a : l) {
    visit(animal_voice{}, a);
}
cout << "----\n";
```

11. 最后，我们将回来数一下 `cat` 和 `dog` 在列表中的数量。`is_type<T>` 的 `cat` 和 `dog` 特化函数，将会与 `std::count_if` 结合起来使用，用来返回列表中不同实例的个数：

```
cout << "There are "
<< count_if(begin(l), end(l), is_type<cat>)
<< " cats and "
<< count_if(begin(l), end(l), is_type<dog>)
<< " dogs in the list.\n";
}
```

12. 编译并运行程序，我们就会看到打印三次的结果都是相同的。然后，可以看到 `is_type` 和 `count_if` 配合的很不错：

```
$ ./variant
Tuba says Meow!
Balou says Woof!
Bobby says Meow!
-----
Tuba says Meow!
Balou says Woof!
Bobby says Meow!
-----
Tuba says Meow!
Balou says Woof!
Bobby says Meow!
-----
There are 2 cats and 1 dogs in the list.
```

How it works...

`std::variant` 与 `std::any` 类型很相似，因为这两个类型都能持有不同类型的变量，并且我们需要在运行时对不同对象进行区分。

另外，`std::variant` 有一个模板列表，需要传入可能在列表中的类型，这点与 `std::any` 截然不同。也就是说 `std::variant<A, B, C>` 必须是 A、B 或 C 其中一种实例。当然这也意味着其就不能持有其他类型的变量，除了列表中的类型 `std::variant` 没有其他选择。

`variant<A, B, C>` 的类型定义，与以下联合体定义类似：

```
union U {
    A a;
    B b;
    C c;
};
```

当我们对 `a`, `b` 或 `c` 成员变量进行初始化时，联合体中对其进行构建机制需要我们自行区分。`std::variant` 类型就没有这个问题。

本节的代码中，我们使用了三种方式来处理 `variant` 中成员的内容。

首先，使用了 `variant` 的 `index()` 成员函数。对变量类型进行索引，`variant<A, B, C>` 中，索引值0代表A类型，1为B类型，2为C类型，以此类推来访问复杂的 `variant` 对象。

下一种就是使用 `get_if<T>` 函数进行获取。其能接受一个 `variant` 对象的地址，并且返回一个类型 `T` 的指针，指向其内容。如果 `T` 类型是错误，那么返回的指针就为 `null` 指针。其也可能对 `variant` 变量使用 `get<T>(x)` 来获取对其内容的引用，不过当这样做失败时，函数将会抛出一个异常(使用 `get`-系列函数进行转换之前，需要使用 `holds_alternative<T>(x)` 对其类型进行检查)。

最后一种方式就是使用 `std::visit` 函数来进行，其能接受一个函数对象和一个 `variant` 实例。`visit` 函数会对 `variant` 中内容的类型进行检查，然后调用对应的函数对象的重载 `operator()` 操作符。

为了这个目的，我们实现为了 `animal_voice` 类型，将 `visit` 和 `variant<dog, cat>` 类型结合在了一起：

```
struct animal_voice
{
    void operator()(const dog &d) const { d.woof(); }
    void operator()(const cat &c) const { c.meow(); }
};
```

以 `visit` 的方式对 `variant` 进行访问看起来更加的优雅一些，因为使用这种方法就不需要使用硬编码的方式对 `variant` 内容中的类型进行判别。这就让我们的代码更加容易扩展。

Note:

`variant` 类型不能为空的说法并不完全正确。将 `std::monostate` 类型添加到其类型列表中，其就能持有空值了。

自动化管理资源——`std::unique_ptr`

C++11之后，STL提供了新的智能指针，能对动态内存进行跟踪管理。C++11之前，C++中也有一个智能指针 `auto_ptr`，也能对内存进行管理，但是很容易被用错。

不过，使用C++11添加的智能指针的话，我们就很少需要使用到 `new` 和 `delete` 操作符。智能指针是自动化内存管理的一个鲜活的例子。当我们使用 `unique_ptr` 来动态分配对象，基本上不会遇到内存泄漏，因为在析构时会自动的为其所拥有内存使用 `delete` 操作。

唯一指针表达了其对对象指针的所有权，当对这段内存不再进行使用时，我们会将相关的对象所具有的内存进行释放。这个类将让我们永远远离内存泄漏(智能指针还有 `shared_ptr` 和 `weak_ptr`，不过本节中，我们只关注于 `unique_ptr`)。其不会多占用空间，并且不会影响运行时性能，这相较于原始的裸指针和手动内存管理来说十分便捷。(当我们对相应的对象进行销毁后，其内部的裸指针将会被设置为 `nullptr`)。

本节中，我们将来看一下 `unique_ptr` 如何使用。

How to do it...

我们将创建一个自定义的类型，在构造和析构函数中添加一些调试打印信息，之后展示 `unique_ptr` 如何对内存进行管理。我们将使用 `unique` 指针，并使用动态分配的方式对其进行实例化：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <memory>

using namespace std;
```

2. 我们将实现一个小类型，后面会使用 `unique_ptr` 对其实例进行管理。其构造函数和析构函数都会在终端上打印相应的信息，所以之后的自动删除中，我们会看到相应输出的打印：

```

class Foo
{
public:
    string name;

    Foo(string n)
        : name{move(n)}
    { cout << "CTOR " << name << '\n'; }

    ~Foo() { cout << "DTOR " << name << '\n'; }
};

```

3. 为了了解函数对唯一指针在作为参数传入函数的限制，我们可以实现一个这样的函数。其能处理一个 `Foo` 类型实例，并能将其名称进行打印。注意，`unique` 指针是非常智能的，其无额外开销，并且类型安全，也可以为 `null`。这就意味着我们仍然要在解引用之前，对指针进行检查：

```

void process_item(unique_ptr<Foo> p)
{
    if (!p) { return; }

    cout << "Processing " << p->name << '\n';
}

```

4. 主函数中，我们将开辟一个代码段，在堆上创建两个 `Foo` 对象，并且使用 `unique` 指针对内存进行管理。我们显式的使用 `new` 操作符创建第一个对象实例，并且将其用来创建 `unique_ptr<Foo>` 变量 `p1`。我们通过 `make_unique<Foo>` 的调用创建第二个 `unique` 指针 `p2`，我们直接传入参数对 `Foo` 实例进行构建。这种方式更加的优雅，因为我们使用 `auto` 类型对类型进行推理，并且能在第一时间对对象进行访问，并且其已经使用 `unique_ptr` 进行管理：

```

int main()
{
{
    unique_ptr<Foo> p1 {new Foo{"foo"}};
    auto p2 (make_unique<Foo>("bar"));
}

```

5. 离开这个代码段时，所创建的对象将会立即销毁，并且将内存进行释放。让我们来看一下 `process_item` 函数和如何使用 `unique_ptr`。当创建一个新的 `Foo` 实例时，其就会被 `unique_ptr` 进行管理，然后参数的生命周期就在这个函数中。当 `process_item` 返回时，这个对象就会被销毁：

```
process_item(make_unique<Foo>("foo1"));
```

6. 如将已经存在的对象传入 `process_item` 函数，就需要将指针的所有权进行转移，因为函数需要使用 `unique_ptr` 作为输入参数，这就会有一次拷贝。但是，`unique_ptr` 是无法进行拷贝的，其只能移动。现在让我们来创建两个 `Foo` 对象，并且将其中一个移动到 `process_item` 函数中。通过对输出的查阅，我们可以了解到 `foo2` 在 `process_item` 返回时会被析构，因为其所有权已经被转移。`foo3` 将会持续留存于主函数中，直到主函数返回时才进行析构：

```
auto p1 (make_unique<Foo>("foo2"));
auto p2 (make_unique<Foo>("foo3"));

process_item(move(p1));

cout << "End of main()\n";
}
```

7. 编译并运行程序。首先，我们将看到 `foo` 和 `bar` 的构造和析构的输出，离开代码段时就被销毁。我们要注意的是，销毁的顺序与创建的顺序相反。下一个构造的就是 `foo1`，其在对 `process_item` 调用时进行创建。当函数返回时，其就会被立即销毁。然后，我们会创建 `foo2` 和 `foo3`。因为之前转移了指针的所有权，`foo2` 会在 `process_item` 函数调用返回时被立即销毁。另一个元素 `foo3` 将会在主函数返回时进行销毁：

```
$ ./unique_ptr
CTOR foo
CTOR bar
DTOR bar
DTOR foo
CTOR foo1
Processing foo1
DTOR foo1
CTOR foo2
CTOR foo3
Processing foo2
DTOR foo2
End of main()
DTOR foo3
```

How it works...

使用 `std::unique_ptr` 来处理堆上分配的对象非常简单。我们初始化 `unique` 指针之后，其就会指向对应的对象，这样程序就能自动的对其进行释放操作。

当我们把 `unique` 指针赋予一些新指针时，其就会先删除原先指向的对象，然后再存储新的指针。一个 `unique` 指针变量 `x`，我们可以使用 `x.reset()` 将其目前所指向的对象进行销毁，然后在指向新的对象。另一种等价方式：`x = new_pointer` 与 `x.reset(new_pointer)` 的方式等价。

Note:

的确只有一种方式对 `unique_ptr` 所指向对象的内存进行释放，那就是使用成员函数 `release`，但这种方式并不推荐使用。

解引用之前，需要对指针进行检查，并且其能适用于裸指针相同的方式进行运算。条件语句类似于 `if (p){...}` 和 `if (p != nullptr){...}`，这与我们检查裸指针的方式相同。

解引用一个 `unique` 指针可以通过 `get()` 函数完成，其会返回一个指向对应对象的裸指针，并且可以直接进行解引用。

`unique_ptr` 有一个很重要的特性——实例无法进行拷贝，只能移动。这就是我们会将已经存在的 `unique` 指针的所有权转移到 `process_item` 参数的原因。当我们想要拷贝 `unique` 指针时，就意味着两个 `unique` 指针指向相应的对象，这与该指针的设计理念不符，所以 `unique` 指针对其指向对象的所有权必须唯一。

Note:

对于其他的数据类型，由于智能指针的存在，所以很少使用 `new` 和 `delete` 对其进行手动操作。尽可能的使用智能指针！特别是 `unique_ptr`，其在运行时无任何额外开销。

处理共享堆内存——`std::shared_ptr`

上一节中，我们了解了如何使用 `unique_ptr`。这个类型非常有用，能帮助我们管理动态分配的对象。不过，所有权只能让一个类型对象所有，不能让多个对象指向同一个动态分配的对象。

指针类型 `shared_ptr` 就是为了应对这种情况所设计的。共享指针可以随时进行拷贝，其内部有一个计数器，记录了有多少对象持有这个指针。只有当最后一个持有者被销毁时，才会对动态分配的对象进行删除。同样，其也不会让我们陷入内存泄漏的窘境，因为对象也会在使用之后进行自动删除。同时，需要确定对象没有过早的被删除，或是删除的过于频繁(每次对象的创建都要进行一次删除)。

本节中，你将了解到如何使用 `shared_ptr` 自动的对动态对象进行管理，并且能在多个所有者间共享动态对象，而后了解其与 `unique_ptr` 之间的区别：

How to do it...

我们将完成一个与 `unique_ptr` 节类似的程序，以展示 `shared_ptr` 的用法：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <memory>

using namespace std;
```

2. 然后定义一个辅助类，其能帮助我们了解类何时创建和销毁。我们将会使用 `shared_ptr` 对内存进行管理：

```
class Foo
{
public:
    string name;

    Foo(string n)
        : name{move(n)}
    { cout << "CTOR " << name << '\n'; }

    ~Foo() { cout << "DTOR " << name << '\n'; }
};
```

3. 接下来，我们将实现一个函数 `foo`，其参数的类型为共享指针。接受共享指针作为参数的方式，要比引用有意思得多，因为这样我们不会进行拷贝，但是会改变共享这指针内部的计数器：

```

void f(shared_ptr<Foo> sp)
{
    cout << "f: use counter at "
        << sp.use_count() << '\n';
}

```

4. 主函数中声明一个空的共享指针。通过默认构造方式对其进行构造，其实上是一个 `null` 指针：

```

int main()
{
    shared_ptr<Foo> fa;
}

```

5. 下一步，我们将创建一个代码段，并创建两个 `Foo` 对象。使用 `new` 操作符对第一个对象进行创建，然后使用构造函数在 `shared_ptr` 中创建这一对象。直接使用 `make_shared<Foo>` 对第二个实例进行创建，使用我们提供的参数创建一个 `Foo` 实例。这种创建的方式很优雅，使用 `auto` 进行类型推断，对象也算第一次访问。这里与 `unique_ptr` 很类似：

```

{
    cout << "Inner scope begin\n";

    shared_ptr<Foo> f1 {new Foo{"foo"}};
    auto f2 (make_shared<Foo>("bar"));
}

```

6. 当共享指针被共享时，需要记录有多少个指针共享了这个对象。这需要内部引用计数器或使用独立计数器完成，我们可以使用 `use_count` 将这个值进行输出。现在其值为1，因为其还没进行拷贝。我们可以 `f1` 拷贝到 `fa`，其计数值将会为2。

```

cout << "f1's use counter at " << f1.use_count()
<< '\n';
fa = f1;
cout << "f1's use counter at " << f1.use_count()
<< '\n';
}

```

7. 离开这个代码段时，共享指针 `f1` 和 `f2` 将会被销毁。`f1` 变量引用计数将会减少1，现在只有 `fa` 拥有这个 `Foo` 实例。当 `f2` 被回收时，其引用计数将减为0。因此，`shared_ptr` 指针将对对象进行销毁：

```

}
cout << "Back to outer scope\n";

cout << fa.use_count() << '\n';
}

```

8. 现在，让我们用两种方式调用 `f` 函数。第一种，我们使用直接拷贝 `fa` 的方式。`f` 函数将会将引用计数输出，值为2。在第二次对 `f` 的调用时，我们将指针移动到函数中。现在只有 `f` 函数对其指向的对象具有所有权：

```
cout << "first f() call\n";
f(fa);
cout << "second f() call\n";
f(move(fa));
```

9. `f` 返回之后，`foo` 实例就被立即销毁，因为没有任何指针对其具有所有权。因此，在主函数返回前，所有对象就都会被销毁：

```
cout << "end of main()\n";
}
```

10. 编译并运行程序就会得到如下输出。起初，我们可以看到 `foo` 和 `bar` 被创建。然后指针的副本 `f1` 出现(其指向 `foo` 实例)，引用计数增加到2。当离开代码段时，因为没有任何指针在对指向 `bar` 实例的共享指针具有所有权，所以其会自动进行销毁。现在 `fa` 的引用计数为1，因为现在只有 `fa` 对 `foo` 对象具有所有权。之后，我们调用了两次 `f` 函数。第一次调用，我们对 `fa` 进行了拷贝，会再次将引用计数增为2。第二次调用时，我们将 `fa` 移动到 `f` 中，其对引用计数的数值并无影响。此外，因为 `f` 函数具有了 `foo` 对象指针的所有权，所以当 `f` 函数结束时，`foo` 对象就自动销毁了。主函数打印出最后一行前，堆上分配的动态对象就会被全部销毁：

```
$ ./shared_ptr
Inner scope begin
CTOR foo
CTOR bar
f1's use counter at 1
f1's use counter at 2
DTOR bar
Back to outer scope
1
first f() call
f: use counter at 2
second f() call
f: use counter at 1
DTOR foo
end of main()
```

How it works...

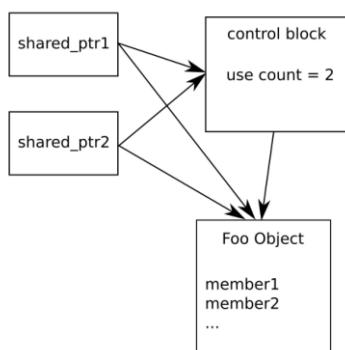
`shared_ptr` 的工作方式与 `unique_ptr` 的类似。构造共享指针和唯一指针的方法也非常类似(使用 `make_shared` 函数创建共享对象的指针，使用 `make_unique` 创建 `unique_pointer`)。

`unique_ptr` 和 `shared_ptr` 的最大区别在于可复制性上，因为共享指针内部具有一块控制区域(**control block**)，其中有用来管理对象的指针，还有一个计数器。当有N个`shared_ptr`实例指向某个对象时，其内部的计数器的值就为N。

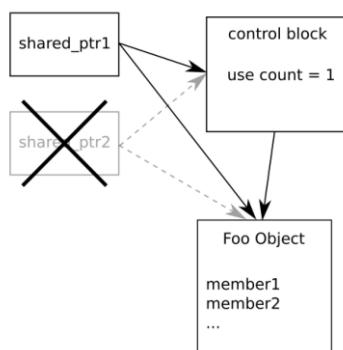
当`shared_ptr`实例销毁时，内部计数器会减1。当没有指针对对象具有所有权时，计数器的值即为0，对象就会被自动销毁。这样我们就不用担心内存泄漏了。

为了更加形象的说明，我们来看一下下面的图：

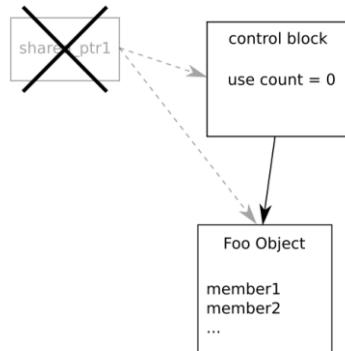
1.)



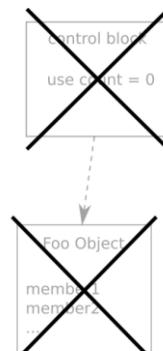
2.)



3.)



4.)



第1步中，我们具有两 `shared_ptr` 实例用于管理 `Foo` 类型的一个对象。所以其引用个数为2。然后，`shared_ptr2` 被销毁，计数就会变为1。因为还有指针指向其实例，所以 `Foo` 对象并未被销毁。第3步中，最后一个共享指针也被销毁了，这就导致引用计数为0。第4步会很快在第3步之后发生，所有控制块和 `Foo` 实例都会被销毁，并且其内存也会在堆上释放。

了解了 `shared_ptr` 和 `unique_ptr`，我们将能很容易的对动态分配的对象进行管理，并且不用担心出现内存泄漏。不过，这里有个忠告——共享指针避免在循环引用的指针间进行，这样会让计数器无法归零，导致内存泄漏。

There's more...

来看一下下面的代码。你能告诉我这段代码是否会发生内存泄漏吗？

```
void function(shared_ptr<A>, shared_ptr<B>, int);
// "function" is defined somewhere else

// ...somewhere later in the code:
function(new A{}, new B{}, other_function());
```

你可能会反问我，“怎么可能有内存泄漏呢？”，`A`和`B`在分配后就放入`shared_ptr`类型中，并且其之后会进行释放，所以不会有内存泄漏。

你说的没错，当我们确定`shared_ptr`实例获取了对应的指针时，那么我们不会遇到内存泄漏。不过这个问题又好像有点琢磨不透。

我们调用函数`f(x(), y(), z())`时，编译器需要在`f`前找到`x`，`y`，`z`函数的定义，并先运行这些函数，然后将其返回值传入`f`函数中。结合我们上面的例子来说的话，对于编译器来说执行`x`，`y`和`z`函数的顺序并没有被规定。

回看下我们给出的例子，当编译器决定首先进行`new A{}`的操作，然后进行`other_function()`，再进行`new B{}`操作时，那么能确保这些操作的结果会传到`function`中吗？当`other_function()`抛出一个异常，因为没有使用`shared_ptr`对`A`进行管理，所以会造成内存泄漏。无论我们如何捕获这个异常，对这个对象的处理机会已经不在，我们无法将其删除。

这里有两种方法可以规避这个问题：

```
// 1.)
function(make_shared<A>(), make_shared<B>(),
other_function());

// 2.)
shared_ptr<A> ap {new A{}};
shared_ptr<B> bp {new B{}};
function(ap, bp, other_function());
```

这样，对象在传入函数之前就被`shared_ptr`所保管，也就无所谓函数是否会在中途抛出异常了。

对共享对象使用弱指针

本节和 `shared_ptr` 有关，我们已经了解了如何使用共享指针。和 `unique_ptr` 一样，其提升了C++对动态分配对象的管理能力。

拷贝 `shared_ptr` 时，我们会将内部计数器加1。当持有共享指针的拷贝时，其指向的对象则不会被删除。但是，当使用某种弱指针时，其能像普通指针一样对指向对象进行操作，但依旧能让所指向对象被销毁？然而，销毁之后我们应该如何确定对象是否存在呢？

这种情况下 `weak_ptr` 就是我们最佳的选择。其相对于 `unique_ptr` 和 `shared_ptr` 来说有些复杂，但是在本节随后的内容中，我们将会对其进行使用。

How to do it...

我们将使用 `shared_ptr` 对一个实例进行管理，然后我们将 `weak_ptr` 混入其中，从而了解对 `weak_ptr` 的操作对智能指针的内存处理有何影响：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <iomanip>
#include <memory>

using namespace std;
```

2. 接下来，我们将实现一个类，将在析构函数的实现中进行打印。当类型被析构时，我们可以从打印输出进行判断：

```
struct Foo {
    int value;

    Foo(int i) : value(i) {}

    ~Foo() { cout << "DTOR Foo " << value << '\n'; }
};
```

3. 让我们来实现一个函数用于对弱指针的信息进行打印，这样我们就可以了解弱指针不同指向时的状态。`expired` 成员函数会告诉我们，弱指针指向的对象是否依旧存在，因为使用弱指针持有这个对象并无法让其生命周期延长！`use_count` 计数器告诉我们，当前 `shared_ptr` 实例中对象的引用次数：

```

void weak_ptr_info(const weak_ptr<Foo> &p)
{
    cout << "-----" << boolalpha
    << "\nexpired: " << p.expired()
    << "\nuse_count: " << p.use_count()
    << "\ncontent: ";
}

```

4. 当我们要访问一个实际对象时，需要调用 `lock` 函数，会返回一个指向对象的共享指针。当对象不存在时，返回的共享指针则是一个空指针。我们将对其进行检查，然后对其进行访问：

```

if (const auto sp (p.lock()); sp) {
    cout << sp->value << '\n';
} else {
    cout << "<null>\n";
}
}

```

5. 主函数中实例化一个空的弱指针，并且对其内容进行打印：

```

int main()
{
    weak_ptr<Foo> weak_foo;
    weak_ptr_info(weak_foo);
}

```

6. 新的代码段中，使用 `foo` 类实例化了一个共享指针，再将其拷贝给弱指针。需要注意的是，这个操作并不会对共享指针的引用计数有任何影响。其引用计数依旧为1，因为只有共享指针对其具有所有权：

```

{
    auto shared_foo (make_shared<Foo>(1337));
    weak_foo = shared_foo;
}

```

7. 离开代码段前，我们对弱指针的状态进行打印；离开时候，再打印一次。虽然弱指针依旧指向 `foo` 的对象，但是 `foo` 实例还是会在离开代码段时立即被销毁：

```

    weak_ptr_info(weak_foo);
}

weak_ptr_info(weak_foo);
}

```

8. 编译并运行程序，就会看到 `weak_ptr_info` 函数的输出。第一次，是弱指针为空的时候。第二次，是其指向我们创建的 `foo` 实例，并在弱指针锁定后对其进行解引用。第三次调用之前，我们离开了内部代码区域，会触发 `foo` 类型的析

构。之后，我们就无法通过弱指针获取已经删除的 `Foo` 对象，并且在这时弱指针也意识到，原先指向的对象已经不存在了：

```
$ ./weak_ptr
-----
expired: true
use_count: 0
content: <null>
-----
expired: false
use_count: 1
content: 1337
DTOR Foo 1337
-----
expired: true
use_count: 0
content: <null>
```

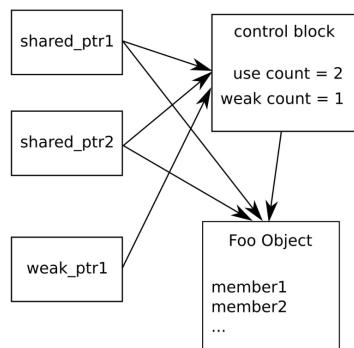
How it works...

弱指针为我们提供了一种指向共享指针对象，但不会增加其引用计数的方式。

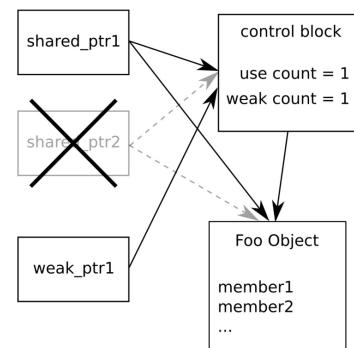
Okay，一个裸指针也可以做这样的事，不过裸指针无法告诉我们其是否处于悬垂的状态，而弱指针可以！

为了能更好的了解弱指针为共享指针添加的功能，我们画了一张图供大家参考：

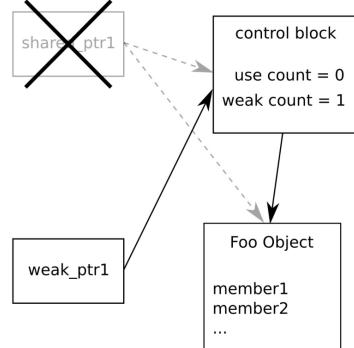
1.)



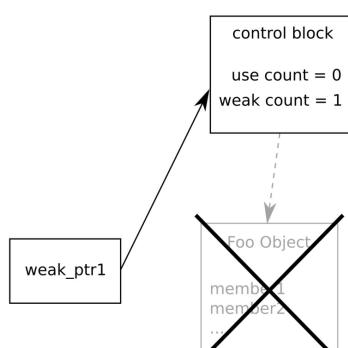
2.)



3.)



4.)



流程与共享指针的图类似。第1步中，我们有两个共享指针和一个弱指针，都指向 `Foo` 类型的实例。虽然有三个指针指向这个对象，但是其共享指针引用数依旧为 2，弱指针在控制块有属于自己的计数器。第2和3步中，共享指针的实例被销毁，这步将会让引用计数归0。第4步，`Foo` 对象也被销毁了，不过控制块依旧存在。因为弱指针依旧需要控制块来对其是否悬垂进行判断。只有当最后一个指向对象的弱指针被销毁，那么控制块才会被销毁。

也可以说处于悬垂状态的弱指针是无效的。为了对这个属性进行检查，我们可以调用 `weak_ptr` 指针的 `expired` 成员函数，其将会为我们返回一个布尔值。当其返回 `true` 时，我们就不能对这个弱指针进行解引用，因为其说明这个对象已经不存在了。

为了对弱指针解引用，我们需要调用 `lock()` 函数。这是一种安全和方便的方法，因为函数返回给我们一个共享指针。当持有这个共享指针时，我们对其进行锁定，所以这时对象的计数器无法进行变化。`lock()` 之后，对象被删除，我们将会得到一个空的共享指针。

使用智能指针简化处理遗留API

智能指针(`unique_ptr`，`shared_ptr`和`weak_ptr`)非常有用，并且对于开发者来说，可以使用其来代替手动分配和释放空间。

当有对象不能使用`new`操作进行创建，或不能使用`delete`进行释放呢？过去有很多库都有自己的分配和释放函数。这看起来好像是个问题，因为我么了解的智能指针都依赖于`new`和`delete`。那么如何在智能指针中，使用指定的工厂函数对特定类型的对象进行创建或是销毁呢？

这个问题一点都不难。本节中，我们将来了解一下如何为智能指针指定特定的分配器和销毁器。

How to do it...

本节中，我们将定义一种不能使用`new`创建的类型，并且也不能使用`delete`进行释放。对于这种限制，我们依旧选择直接使用智能指针，这里使用`unique_ptr`和`shared_ptr`实例来进行演示。

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <memory>
#include <string>

using namespace std;
```

2. 声明一个类，将其构造函数和析构函数声明为`private`。我们使用这样的方式来模拟无法直接和销毁对象实例的情况：

```
class Foo
{
    string name;

    Foo(string n)
        : name(n)
    { cout << "CTOR " << name << '\n'; }

    ~Foo() { cout << "DTOR " << name << '\n'; }
```

3. 然后，声明两个静态函数`create_foo`和`destroy_foo`，这两个函数用来对`Foo`实例进行创建和销毁，其会对裸指针进行操作。这是用来模拟使用旧C风格的API，这样我们就不能用之前的方式直接对`shared_ptr`指针进行使用：

```

public:
    static Foo* create_foo(string s) {
        return new Foo{move(s)};
    }

    static void destroy_foo(Foo *p) { delete p; }
};

```

4. 现在，我们用 `shared_ptr` 来对这样的对象进行管理。对于共享指针，我们可以
通过 `create_foo` 函数来构造相应的对象。只有销毁的方式有些问题，因
为 `shared_ptr` 默认的销毁方式会有问题。解决方法就是我们将自定义的销毁器
给予 `shared_ptr`。删除函数或删除可调用对象的函数签名需要需要
与 `destroy_foo` 函数统一。当我们的删除函数非常复杂，那我们可以使用
Lambda 表达式对其进行包装：

```

static shared_ptr<Foo> make_shared_foo(string s)
{
    return {Foo::create_foo(move(s)),
            Foo::destroy_foo};
}

```

5. 需要注意的是 `make_shared_foo` 函数，将会返回一个普通的 `shared_ptr<Foo>` 实
例，因为设置了自定义的销毁器，并不会对其类型有所影响。从编程角度上，
之前是因为 `shared_ptr` 调用了虚函数，将设置销毁器的步骤隐藏了。唯一指针
(`unique_ptr`)不会带来任何额外开销，所以这种方式不适合唯一指针。目前，
我们就需要对 `unique_ptr` 所持有的类型进行修改。我们将 `void(*)(Foo*)` 类型作
为第二个模板参数传入，其也就是 `destroy_foo` 函数的类型：

```

static unique_ptr<Foo, void (*)(Foo*)>
make_unique_foo(string s)
{
    return {Foo::create_foo(move(s)),
            Foo::destroy_foo};
}

```

6. 主函数中，我们直接使用函数对两个智能指针进行实例化。程序的输出中，我
们将看到相应的对象会被创建，然后自动销毁：

```

int main()
{
    auto ps (make_shared_foo("shared Foo instance"));
    auto pu (make_unique_foo("unique Foo instance"));
}

```

7. 编译并运行程序，我们就会得到如下输出，输出与我们的期望一致：

```
$ ./legacy_shared_ptr
CTOR shared Foo instance
CTOR unique Foo instance
DTOR unique Foo instance
DTOR shared Foo instance
```

How it works...

通常来说，当 `unique_ptr` 和 `shared_ptr` 要销毁其持有的对象时，只会对内部指针使用 `delete`。本节中，我们的类无法使用C++常用的方式进行创建和销毁。`Foo::create_foo` 函数会返回一个构造好的 `Foo` 指针，这对于智能指针来说没什么，因为指针指针也可以对裸指针进行管理。

其问题在于，当对象不能使用默认方式删除时，如何让 `unique_ptr` 和 `shared_ptr` 接触到对象的析构函数。

在这方面，两种智能指针有些不同。为了为 `unique_ptr` 设置一个自定义销毁器，我们需要对其类型进行修改。因为 `Foo` 的销毁函数为 `void Foo::destroy_foo(Foo*);`，那么 `unique_ptr` 所有是 `Foo` 的类型必须为 `unique_ptr<Foo, void(*)(Foo*)>`。现在，`unique_ptr` 也就获取了 `destroy_foo` 的指针了，在 `make_unique_foo` 函数中其作为构造的第二个模板参数传入。

`unique_ptr` 为了自定义销毁器函数，需要对持有类型进行修改，那么为什么 `shared_ptr` 就不需要呢？我们也能向 `shared_ptr` 的第二个模板参数传入对应的类型的呀。为什么 `shared_ptr` 的操作就要比 `unique_ptr` 简单呢？

这是因为 `shared_ptr` 支持可调用删除器对象，而不用影响共享指针做指向的类型，这种功能在控制块中进行。共享指针的控制块是一个对象的虚函数。这也就意味着标准共享指针的控制块，与给定了自定义的销毁器的共享指针的控制块不同！当我们想要一个唯一指针使用一个自定义销毁器时，就需要改变唯一指针所指向的类型。当我们想让共享指针使用自定义销毁器时，只需要对内部控制块的类型进行修改即可，这种修改的过程对我们是不可见的，因为其不同隐藏在虚函数的函数接口中。

当然，我们可以手动的为 `unique_ptr` 做发生在 `shared_ptr` 上的事情，不过这会增加运行时的开销。这是我们所不希望看到的，因为 `unique_ptr` 能够保证在运行时无任何额外开销。

共享同一对象的不同成员

试想我们在一个共享指针中持有一个组成非常复杂的动态分配对象，然后使用新的线程完成一些特别耗时的任务。当我们想要对共享指针所持有的对象进行释放时，线程很有可能仍旧会对这个对象进行访问。当然，我们并不想把这个非常复杂的对象交给线程，因为这样的设计有违我们的初衷。那么就意味着我们要手动对内存进行管理了么？

非也！这个问题可以由共享指针来解决，指向一个非常大的共享对象。另外，可以在初始化阶段就接管对对象的内存管理任务。

在这个例子中，我们将模拟这样一种情况(为了简单，不使用线程)，让我们来看一下 `shared_ptr` 是如何来解决这个问题的。

How to do it...

我们将定义一个结构体，其中包含了多个成员。然后，我们会使用共享指针来管理这个类型的动态分配实例。对于共享指针来说，不会直接指向这个对象的本身，而会指向其成员：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <memory>
#include <string>

using namespace std;
```

2. 定义一个类型，其中包含了不同的成员，将使用共享指针指向这些成员。为了能清晰的了解类型实例何时被创建与销毁，我们让构造和析构函数都打印一些信息：

```
struct person {
    string name;
    size_t age;

    person(string n, size_t a)
        : name{move(n)}, age{a}
    { cout << "CTOR " << name << '\n'; }

    ~person() { cout << "DTOR " << name << '\n'; }
};
```

3. 再来创建几个共享指针，用于指向 `person` 类型实例中的 `name` 和 `age` 成员变量：

```
int main()
{
    shared_ptr<string> shared_name;
    shared_ptr<size_t> shared_age;
```

4. 接下来，创建一个新代码段，并创建一个 `person` 对象，并且用共享指针对其进行管理：

```
{
    auto sperson (make_shared<person>("John Doe",
30));
```

5. 使用之前定义的两个共享指针，分别指向 `name` 和 `age` 成员。使用了 `shared_ptr` 的特定构造函数，其能接受一个共享指针和一个共享指针持有对象的成员变量。这样就能对整个对象进行管理，但不指向其本身！

```
    shared_name = shared_ptr<string>(sperson,
&sperson->name);
    shared_age = shared_ptr<size_t>(sperson,
&sperson->age);
}
```

6. 离开代码段后，我们将会打印 `person` 的 `name` 和 `age` 的值。这个操作只是用来验证，对象是否依旧存在：

```
cout << "name: " << *shared_name
<< "\nage: " << *shared_age << '\n';
}
```

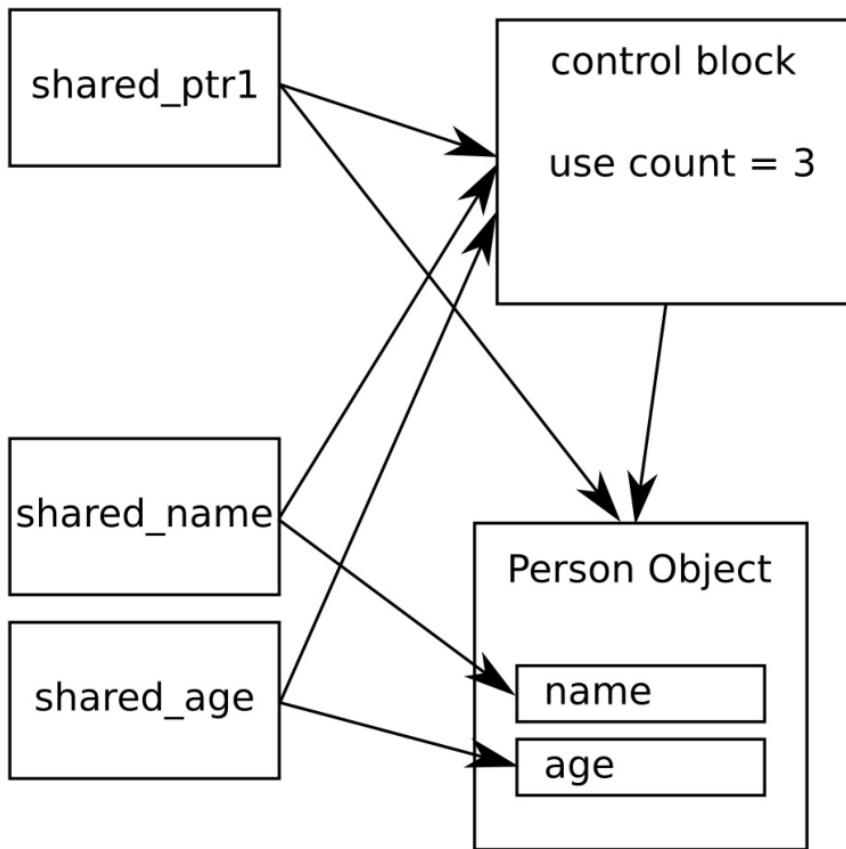
7. 编译并运行程序，我们就是会看到如下的输出。从析构函数的信息中，我们看到通过指向成员的智能指针，打印 `person` 的 `name` 和 `age` 时，对象依旧存在：

```
$ ./shared_members
CTOR John Doe
name: John Doe
age:30
DTOR John Doe
```

How it works...

本节中，我们首先动态创建了一个 `person` 对象，交给共享指针进行管理。然后，我们创建两个智能指针，分别指向 `person` 对象的两个成员变量。

为了描述我们创建了一个什么样的情景，可以看一下下面的图：



注意 `shared_ptr1` 是直接指向 `person` 对象，而 `shared_name` 和 `shared_age` 则指向的是同一个对象的 `name` 和 `age` 成员变量。显然，这些指针管理着整个对象的生命周期。可能是因为内部控制块都指为同一个控制块，这样就无所谓是否仅指向对象的子对象了。

这种情况下，控制块中的使用计数为3。`person` 对象在 `shared_ptr1` 销毁时，其对象也不会被销毁，因为还有其他指针指向它。

创建指向对象成员的指针的写法，看起来有些奇怪。为了让 `shared_ptr<string>` 指向 `person` 对象的 `name` 成员，我们的代码需要这样写：

```
auto sperson (make_shared<person>("John Doe", 30));
auto sname (shared_ptr<string>(sperson, &sperson->name));
```

为了得到指向共享对象成员的指针，我们使用成员的类型对共享指针进行特化，以便其能对成员进行访问。这也就是为什么我们在上面的代码中，创建智能指针的部分写成 `shared_ptr<string>` 的原因。构造函数中，我们提供了持有 `person` 对象的原始共享指针，第二个参数则是新共享指针所要指向对象的地址。

选择合适的引擎生成随机数

有时我们在写程序的时候需要用随机数，C++11之前开发者通常会使用C函数 `rand()` 获取随机数。在C++11之后，STL中添加了一整套随机数生成器，每一个随机数生成器都有自己的特性。

这些生成器并非都是以自解释的方式命名，所以我们要在本节对它们进行了解。本节最后，我们会了解它们有什么不同，哪种情况下应该选择哪一个。不过，这么多生成器，我们不会全部用到。

How to do it...

我们将实现一个生产者，通过随机生成器画出漂亮的直方图。然后，我们将通过这个生成器运行STL中所有的随机值生成引擎，并且对其产生的结果进行了解。这个程序有很多重复的代码，所以你可以从本书的代码库中直接对源码进行拷贝，这样要比手动输入快得多。

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <string>
#include <vector>
#include <random>
#include <iomanip>
#include <limits>
#include <cstdlib>
#include <algorithm>

using namespace std;
```

2. 然后，实现一个辅助函数，其能帮助我们将各种类型的随机数生成引擎的结果进行统计。其接受两个参数：一个 `partitions` 数和一个 `samples` 数。随机生成器的类型是通过模板参数 `RD` 定义的。这个函数中做的第一件事，就是给结果数值类型进行别名。我们同样要保证至少要将所有数字分成10份：

```
template <typename RD>
void histogram(size_t partitions, size_t samples)
{
    using rand_t = typename RD::result_type;
    partitions = max<size_t>(partitions, 10);
```

3. 接下来，我们将使用 `RD` 类型实例化一个生成器。然后，我们定义一个除数变量 `div`。所有随机数引擎所产生的随机数都在 `0` 到 `RD::max()` 之间。函数参数 `partitions`，允许我们将生成数分成几类。通过对可能的最大值进行分组，我们就能了解每一类的大小如何：

```
RD rd;
rand_t div ((double(RD::max()) + 1) /
partitions);
```

4. 接着，将使用一个 `vector` 对生成数进行计数，与我们类型的个数相同。然后，从随机引擎中获取很多随机值，其个数与 `samples` 数一致。`rd()` 表达式会从生成器中得到一个随机数，并且调整内部状态以生成下一个随机数。每个随机数都会与 `div` 进行相除，这样我们就得到了其所在类的索引号，然后将 `vector` 对应位置的计数器进行加1：

```
vector<size_t> v (partitions);
for (size_t i {0}; i < samples; ++i) {
    ++v[rd() / div];
}
```

5. 现在就有了一个粗粒度的直方图。为了将其进行打印，就要知道实际计数器的值。可以使用 `max_element` 算法提取计数器的最大值。然后，将计数器的最大值除以 100。这样就可以将所有计数器的值除以 `max_div`，得到的结果就在0到100的范围内，我们要打印多少星号。当计数器最大值小于100时，因为我们采样的数量也不是很多，所以我们使用 `max` 函数将被除数的值设置为1：

```
rand_t max_elm (*max_element(begin(v), end(v)));
rand_t max_div (max(max_elm / 100, rand_t(1)));
```

6. 将直方图打印在终端上，每个类都有自己的一行。通过对 `max_div` 的除法确定有多少 * 要进行打印，我们将会在终端上得到一段固定长度的直方图打印：

```
for (size_t i {0}; i < partitions; ++i) {
    cout << setw(2) << i << ":" "
        << string(v[i] / max_div, '*') << '\n';
}
```

7. 现在可以来完成主函数了。我们让用户来确定分成多少类，并对多少数进行采样：

```
int main(int argc, char **argv)
{
    if (argc != 3) {
        cout << "Usage: " << argv[0]
            << " <partitions> <samples>\n";
        return 1;
    }
```

8. 然后，就可以从命令行来获取这两个值。当然，从命令行获取到的是字符串，我们需要使用 `std::stoull` 将其转换成数字(`stoull`为“**string to unsigned long long**”的缩写):

```
size_t partitions {stoull(argv[1])};  
size_t samples {stoull(argv[2])};
```

9. 现在我们就可以为STL提供的每种随机数引擎，使用我们的直方图辅助函数。这里就是本节代码最长的部分。你可以选择从代码库中直接拷贝代码过来。然后对程序的输出进行观察。我们从 `random_device` 开始。这个设备试图将所有随机值均匀分配:

```
cout << "random_device" << '\n';  
histogram<random_device>(partitions, samples);
```

10. 下一个随机引擎为 `default_random_engine`，这种引擎的具体实现需要用实现来指定。其可能是后面任何一种随机引擎:

```
cout << "\ndefault_random_engine" << '\n';  
histogram<default_random_engine>(partitions,  
samples);
```

11. 然后，我们将尝试其他引擎:

```
cout << "\nminstd_rand0" << '\n';  
histogram<minstd_rand0>(partitions, samples);  
cout << "\nminstd_rand" << '\n';  
histogram<minstd_rand>(partitions, samples);  
  
cout << "\nmt19937" << '\n';  
histogram<mt19937>(partitions, samples);  
cout << "\nmt19937_64" << '\n';  
histogram<mt19937_64>(partitions, samples);  
  
cout << "\nrانlux24_base" << '\n';  
histogram<ranlux24_base>(partitions, samples);  
cout << "\nrانlux48_base" << '\n';  
histogram<ranlux48_base>(partitions, samples);  
  
cout << "\nrانlux24" << '\n';  
histogram<ranlux24>(partitions, samples);  
cout << "\nrانlux48" << '\n';  
histogram<ranlux48>(partitions, samples);  
  
cout << "\nknuth_b" << '\n';  
histogram<knuth_b>(partitions, samples);  
}
```

12. 编译并运行程序，就会得到我们想要的结果。我们将看到一段很长的打印信息，并且将看到所有引擎的不同特点。这里我们将类别分为10个，并对1000个数进行采样：

```
$ ./random_generator 10 1000
random_device
0: ****
1: ****
2: ****
3: ****
4: ****
5: ****
6: ****
7: ****
8: ****
9: ****

default_random_engine
0: ****
1: ****
2: ****
3: ****
4: ****
5: ****
6: ****
7: ****
8: ****
9: ****

minstd_rand0
0: ****
1: ****
2: ****
3: ****
4: ****
5: ****
6: ****
7: ****
8: ****
9: ****

minstd_rand
```

13. 然后我们再次执行程序。这次我们仍旧分成10类，但是对1,000,000个数进行采样。其将会生成非常直观的直方图，能更加清晰表现各种引擎的不同点。所以，对于这个程序来说，观察很重要：

```
$ ./random_generator 10 1000000
random_device
0: ****
1: ****
2: ****
3: ****
4: ****
5: ****
6: ****
7: ****
8: ****
9: ****

default_random_engine
0: ****
1: ****
2: ****
3: ****
4: ****
5: ****
6: ****
7: ****
8: ****
9: ****

minstd_rand0
0: ****
1: ****
2: ****
3: ****
4: ****
5: ****
6: ****
7: ****
8: ****
9: ****

minstd_rand
```

How it works...

通常，任何随机数生成器都需要在使用前进行实例化。生成的对象可以像函数一样调用，并无需传入参数，因为其对 `operator()` 操作符进行了重载。每一次调用都会产生一个新的随机数。其使用起来非常的简单。

本节中，我们写了一个比较复杂的程序，从而对各种随机数生成器进行了了解。可以使用我们的程序，在命令行传入不同的参数，得到如下的结论：

- 我们进行的采样次数越多，计数器分布就越均匀。
- 各个引擎中，计数器的分布有很大差异。

- 进行大量的样本采样时，每个随机数引擎所表现出的性能也是不同的。
- 用少量的采样进行多次的执行。每个分布生成的图形，每次都是一样的——因为随机引擎在每一次重复时，都会生成同样的随机数，这就意味着其生成的不是真正的随机数。这样的引擎具有某种确定性，因为其生成的随机数可以进行预测。唯一的例外就是 `std::random_device`。

如同我们所看到的，这里有一些需要进行考量的特性。对于大多数标准应用来说，`std::default_random_engine` 完全够用。对于密码学专家或是类似安全敏感的课题，都会有更加多的引擎可供选择。不过，对于一般开发者来说，这里的是否真正随机，对我们的影响并不大。

我们需要从本节总结出三个实际情况：

1. 通常，选择使用 `std::default_random_engine` 就够用了。
2. 需要生成不确定的随机数时，我们可以使用 `std::random_device`。
3. 通过 `std::random_device` (或从系统时钟获取的时间戳)对随机数引擎进行初始化，这是为了让其在每次调用时，生成不同的随机数。这种方式也叫做“设置种子”。

Note:

如果实际实现库对不确定的随机引擎不支持，那么 `std::random_device` 将退化成其他随机数引擎。

让STL以指定分布方式产生随机数

上一节中，我们了解了STL中的随机数引擎。使用引擎或其他方式生成随机数，只是完成了一半的工作。

另一个问题就是，我们需要怎么样的随机数？如何以编码的方式进行掷硬币？通常都会使用 `rand()%2` 的方式进行，因为其结果为0或1，也就分别代表着硬币的正反面。很公平，不是么；这样就不需要任何多余的库(估计只有数学专业知道，这样的方式获取不到高质量随机数)。

如果我们想要做成一个模型要怎么弄？比如：写成 `(rand() % 6) + 1`，这个就表示所要生成的结果。对于这样简单的任务，有没有库进行支持？

当我们想要表示某些东西，并明确其概率为66%呢？Okay，我们可以写一个公式出来 `bool yesno = (rand() % 100 > 66)`。（这里需要注意，是使用 `>=` 合适？还是使用 `>` 合适？）

如何设置一个不平等的模型，也就是生成数的概率完全不同呢？或是，让我们生成的随机数符合某种复杂的分布？有些问题会很快发展为一个科学问题。为了回到我们的关注点上，让我们了解一下STL所提供的具体工具。

STL有超过10种分布算法，能用来指定随机数的生成方式。本节中，我们将简单的了解一下这些分布，并且近距离了解一下其使用方法。

How to do it...

我们生成随机数，对生成数进行统计，并且将其分布部分在终端上进行打印。我们将了解到，当想要以特定的分布获取随机值时，应该使用哪种方式：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <iomanip>
#include <random>
#include <map>
#include <string>
#include <algorithm>

using namespace std;
```

2. 对于STL所提供的分布来说，我们将从打印出的直方图中看出每种分布的不同。随机采样时，可以将某种分布作为参数传入随机数生成器。然后，我们将实例化默认随机引擎和一个 `map`。这个 `map` 将获取的值与其计数器进行映射，计数器表示这个数产生的频率。我们使用一个限定函数作为随机引擎的指定分布，然后通过随机引擎生成对应分布的随机值：

```
template <typename T>
void print_distro(T distro, size_t samples)
{
    default_random_engine e;
    map<int, size_t> m;
```

3. 使用 `samples` 变量来表示我们要进行多少次采样，并且在采样过程中对 `map` 中的计数器进行操作。这样，就能获得非常漂亮的直方图。单独调用 `e()` 时，随机数引擎将生成一个随机数，`distro(e)` 会通过分布对象对随机数的生成进行限定。

```
for (size_t i {0}; i < samples; ++i) {
    m[distro(e)] += 1;
}
```

4. 为了输出到终端窗口中的数据的美观性，需要了解计数器的最大值。`max_element` 函数能帮助我们找到 `map` 中所有计数器中的最大的那一个，然后返回指向具有最大计数器那个节点的迭代器。知道了最大值，就可以让所有计数器对其进行除法，这样就能在终端窗口生成固定长度的图像了：

```
size_t max_elm (max_element(begin(m), end(m),
[] (const auto &a, const auto &b) {
    return a.second < b.second;
}) ->second);
size_t max_div (max(max_elm / 100, size_t(1)));
```

5. 现在来遍历 `map`，然后对 * 进行打印，对于每一个计数器来说都有一个固定的长度：

```
for (const auto [randval, count] : m) {
    if (count < max_elm / 200) { continue; }

    cout << setw(3) << randval << " : "
        << string(count / max_div, '*') << '\n';
}
```

6. 主函数中，会对传入的参数进行检查，我们会指定每个分布所使用的采样率。如果用户给定的参数不合适，程序将报错：

```

int main(int argc, char **argv)
{
    if (argc != 2) {
        cout << "Usage: " << argv[0]
        << " <samples>\n";
        return 1;
    }
}

```

7. `std::stoull` 会将命令行中的参数转换成数字:

```
size_t samples {stoull(argv[1])};
```

8. 首先, 来尝试 `uniform_int_distribution` 和 `normal_distribution`, 这两种都是用来生成随机数的经典分布。学过概率论的同学应该很熟悉。均匀分布能接受两个值, 确定生成随机数的上限和下限。例如, 0和9, 那么生成器将会生成 [0, 9] 之间的随机数。正态分布能接受平均值和标准差作为传入参数:

```

cout << "uniform_int_distribution\n";
print_distro(uniform_int_distribution<int>{0, 9},
samples);

cout << "normal_distribution\n";
print_distro(normal_distribution<double>{0.0,
2.0}, samples);

```

9. 另一个非常有趣的分布是 `piecewise_constant_distribution`。其能接受两个输入范围作为参数。比如定义为 0, 5, 10, 30, 那么其中的间隔就是0到4, 然后是5到9, 最后一个间隔是10到29。另一个输入范围定义了权重。比如权重 0.2, 0.3, 0.5, 那么最后生成随机数落在以上三个间隔中的概率为20%, 30%和50%。在每个间隔内, 生成数的概率都是相同的:

```

initializer_list<double> intervals {0, 5, 10,
30};
initializer_list<double> weights {0.2, 0.3, 0.5};
cout << "piecewise_constant_distribution\n";
print_distro(
    piecewise_constant_distribution<double>{
        begin(intervals), end(intervals),
        begin(weights)},
    samples);

```

10. `piecewise_linear_distribution` 的构造方式与上一个类似, 不过其权重值的特性却完全不同。对于每一个间隔的边缘值, 只有一种权重值。从一个边界转换到另一个边界中, 概率是线性的。这里我们使用同样的间隔列表, 但权重值不同的方式对分布进行实例化:

```
cout << "piecewise_linear_distribution\n";
initializer_list<double> weights2 {0, 1, 1, 0};
print_distro(
    piecewise_linear_distribution<double>{
        begin(intervals), end(intervals),
        begin(weights2)),
    samples);
```

11. 伯努利分布是另一个非常重要的分布，因为其分布只有“是/否”，“命中/未命中”或“头/尾”值，并且这些值的可能性是指定的。其输出只有0或1。另一个有趣的分布，就是 `discrete_distribution`。例子中，我们离散化了一组值 `1, 2, 4, 8`。这些值可被解释为输出为0至3的权重：

```
cout << "bernoulli_distribution\n";
print_distro(std::bernoulli_distribution{0.75},
samples);

cout << "discrete_distribution\n";
print_distro(discrete_distribution<int>{ {1, 2,
4, 8} }, samples);
```

12. 不同分布引擎之间有很大的不同。都非常特殊，也都在特定环境下非常有用。如果你没有听说过这些分布，应该对其特性不是特别的了解。不过，我们的程序中会生成非常漂亮的直方图，通过打印图，你会对这些分布有所了解：

```

    cout << "binomial_distribution\n";
    print_distro(binomial_distribution<int>{10, 0.3},
samples);
    cout << "negative_binomial_distribution\n";
    print_distro(
        negative_binomial_distribution<int>{10, 0.8},
samples);
    cout << "geometric_distribution\n";
    print_distro(geometric_distribution<int>{0.4},
samples);
    cout << "exponential_distribution\n";
    print_distro(exponential_distribution<double>
{0.4}, samples);
    cout << "gamma_distribution\n";
    print_distro(gamma_distribution<double>{1.5,
1.0}, samples);
    cout << "weibull_distribution\n";
    print_distro(weibull_distribution<double>{1.5,
1.0}, samples);
    cout << "extreme_value_distribution\n";
    print_distro(
        extreme_value_distribution<double>{0.0, 1.0},
samples);
    cout << "lognormal_distribution\n";
    print_distro(lognormal_distribution<double>{0.5,
0.5}, samples);
    cout << "chi_squared_distribution\n";
    print_distro(chi_squared_distribution<double>
{1.0}, samples);
    cout << "cauchy_distribution\n";
    print_distro(cauchy_distribution<double>{0.0,
0.1}, samples);
    cout << "fisher_f_distribution\n";
    print_distro(fisher_f_distribution<double>{1.0,
1.0}, samples);
    cout << "student_t_distribution\n";
    print_distro(student_t_distribution<double>{1.0},
samples);
}

```

13. 编译并运行程序，就可以得到如下输入。首先让我们对每个分布进行1000个的采样看看：

```
$ ./random_distro 1000
uniform_int_distribution
0 : ****
1 : ****
2 : ****
3 : ****
4 : ****
5 : ****
6 : ****
7 : ****
8 : ****
9 :
normal_distribution
-7 : 
-6 : 
-5 : ** 
-4 : *** 
-3 : **** 
-2 : ***** 
-1 : ***** 
0 : ***** 
1 : ***** 
2 : ***** 
3 : ***** 
4 : ***** 
5 : * 
piecewise_constant_distribution
0 : ****
1 : ****
2 : ****
3 : ****
4 : ****
5 : ****
6 : ****
7 : ****
```

14. 然后在用1,000,000个采样，这样获得的每个分布的直方图会更加的清楚。不过，在生成随机数的过程中，我们也会了解到那种引擎比较快，哪种比较慢：

```
$ ./random_distro 1000000
uniform_int_distribution
0 : ****
1 : ****
2 : ****
3 : ****
4 : ****
5 : ****
6 : ****
7 : ****
8 : ****
9 : ****
normal_distribution
-5 : * 
-4 : *** 
-3 : **** 
-2 : ***** 
-1 : ***** 
0 : ***** 
1 : ***** 
2 : ***** 
3 : ***** 
4 : ***** 
5 : * 
piecewise_constant_distribution
0 : ****
1 : ****
2 : ****
3 : ****
4 : ****
5 : ****
6 : ****
7 : ****
8 : ****
9 : ****
10 : ****
11 : ****
12 : ****
13 : ****
14 : ****
15 : ****
16 : ****
17 : ****
18 : ****
19 : ****
20 : ****
```

How it works...

我们通常都不会太在意随机数引擎，不过随着我们对随机数分布的要求和对生成速度的要求，我们就需要通过随机数引擎来解决这个问题。

为了使用任意的分布，首先实例化一个分布对象。会看到不同分布的构造函数所需要的构造参数并不相同。本节的描述中，只使用了一部分分布引擎，因为其中大部分的用途非常特殊，或是使用起来非常复杂。不用担心，所有分布的描述都可以在[C++ STL文档](#)中查到。

不过，当具有一个已经实例化的分布时，我们可以像函数一样对其进行调用(只需要一个随机数引擎对象作为参数)。然后，随机数生成引擎会生成一个随机数，通过特定的分布进行对随机值进行限定，然后得到了所限定的随机数。这就导致不同的直方图具有不同的分布，也就是我们程序输出的结果。

可以使用我们刚刚编写的程序，来确定不同分布的功能。另外，我们也对几个比较重要的分布进行了总结。程序中使用到分布并不都会在下表出现，如果你对某个没出现的分布感兴趣，可以查阅C++ STL文档的相关内容。

分布	描述
uniform_int_distribution	该分布接受一组上下限值作为构造函数的参数。之后得到的随机值就都是在这个范围中。我们可能得到的每个值的可能性是相同的，直方图将会是一个平坦的图像。这个分布就像是掷骰子，因为掷到骰子的每一个面的概率都是相等的。
normal_distribution	正态分布或高斯分布，自然界中几乎无处不在。其STL版本能接受一个平均值和一个标准差作为构造函数的参数，其直方图的形状像是屋顶一样。其分布于人类个体高度或动物的IQ值，或学生成绩，都符合这个分布。
bernoulli_distribution	当我们想要一个掷硬币的结果时，使用伯努利分布就非常完美。其只会产生0和1，并且其构造函数的参数是产生1的概率。
discrete_distribution	当我们有一些限制的时候，我们就可以使用离散分布，需要我们为每个间隔的概率进行定义，从而得到离散集合的值。其构造函数会接受一个权重值类别，并且通过权重值的可能性产生相应的随机数。当我们想要对血型进行随机建模时，每种血型的概率是不一样，这样这种引擎就能很完美地契合这种需求。

第9章 并行和并发

C++11之前，C++原生不支持并发和并发。但这并不意味着无法对线程进行操作，只不过需要使用系统库的API进行操作(因为线程与操作系统是不可分开的)。

随着C++11标准的完成，我们有了 `std::thread`，其能给予我们在所有操作系统上可移植的线程操作。为了同步线程，C++11也添加了互斥量，并且对一些RAII类型的锁进行了封装。另外，`std::condition_variable` 也能够灵活的在线程间，进行唤醒操作。

另一些有趣的东西就是 `std::async` 和 `std::future` ——我们可以将普通的函数封装到 `std::async` 中，可以在后台异步的运行这些函数。包装后函数的返回值则用 `std::future` 来表示，函数的结果将会在运行完成后，放入这个对象中，所以可以在函数完成前，做点别的事情。

另一个STL中值得一提提升就是执行策略，其被添加到已有的69种算法中。这样就可以对现有的STL算法不做任何修改，就能享受其并行化带来的性能提升。

本章中，我们将通过例子来了解其中最为核心的的部分。之后，我们也将了解到C++17对并行的支持。不会覆盖所有的细节，但是比较重要的部分肯定会介绍。本书会快速的帮助你了解并行编程机制，至于详细的介绍，可以在线对C++17 STL文档进行查阅。

最后，本章中最后两节值的注意。倒数第二节中，我们将并行化[第6章的ASCII曼德尔布罗特渲染器](#)，使用STL进阶用法让代码改动程度最小。最后一节中，我们将实现一个简单的库，其可以用来隐式并行复杂的任务。

标准算法的自动并行

C++17对并行化的一个重要的扩展，就是对标准函数的执行策略进行了修改。69个标准算法都能并行到不同的核上运行，甚至是向量化。

对于使用者来说，如果经常使用STL中的算法，那么就能很轻易的进行并行。可以通过基于现存的STL算法一个执行策略，然后就能享受并行带来的好处。

本节中，我们将实现一个简单的程序(通过一个不太严谨的使用场景)，其中使用了多个STL算法。使用这些算法时，我们将看到如何在C++17中，使用执行策略让这些算法并行化。本节最后一个子节中，我们会了解不同执行策略的区别。

How to do it...

本节，将使用标准算法来完成一个程序。这个程序本身就是在模拟我们实际工作中的场景。当使用这些标准算法时，我们为了得到更快的性能，将执行策略嵌入其中：

1. 包含必要的头文件，并声明所使用的命名空间。其中 `execution` 头文件是 C++17之后加入的：

```
#include <iostream>
#include <vector>
#include <random>
#include <algorithm>
#include <execution>

using namespace std;
```

2. 这里声明一个谓词函数，其用来判断给定数值的奇偶：

```
static bool odd(int n) { return n % 2; }
```

3. 主函数中先来定义一个很大的 `vector`。我们将对其进行填充，并对其中数值进行计算。这个代码的执行速度是非常非常慢的。对于不同配置的电脑来说，这个 `vector` 的尺寸可能会有变化：

```
int main()
{
    vector<int> d (50000000);
```

4. 为了向 `vector` 中塞入随机值，我们对随机数生成器进行了实例化，并选择了一种分布进行生成，并且将其打包成为一个可调用的对象。如果你对随机数生成器不太熟，那么你可以回看一下本书的第8章：

```
mt19937 gen;

uniform_int_distribution<int> dis(0, 100000);
auto rand_num [=] () mutable { return dis(gen);
});
```

5. 现在，`std::generate` 算法会用随机值将 `vector` 填满。这个算法是C++17新加入的算法，其能接受一种新的参数——执行策略。我们在这个位置上填入 `std::execution::par`，其能让代码进行自动化并行。通过这个参数的传入，可以使用多线程的方式对 `vector` 进行填充，如果我们的电脑有多核CPU，那么就可以大大节约我们的时间：

```
generate(execution::par, begin(d), end(d),
rand_num);
```

6. `std::sort` 想必大家都是非常熟悉了。C++17对其也提供了执行策略的参数：

```
sort(execution::par, begin(d), end(d));
```

7. 还有 `std::reverse`：

```
reverse(execution::par, begin(d), end(d));
```

8. 然后，我们使用 `std::count_if` 来计算 `vector` 中奇数的个数。并且也可以通过添加执行策略参数对该算法进行加速：

```
auto odds (count_if(execution::par, begin(d),
end(d), odd));
```

9. 最后，将结果进行打印：

```
cout << (100.0 * odds / d.size())
<< "% of the numbers are odd.\n";
}
```

10. 编译并运行程序，就能得到下面的输出。整个程序中我们就使用了一种执行策略，我们对不同执行策略之间的差异也是非常感兴趣。这个就留给读者当做作业。去了解一下不同的执行策略，比如 `seq`，`par` 和 `par_vec`。不过，对于不同的执行策略，我们肯定会得到不同的执行时间：

```
$ ./auto_parallel
50.4% of the numbers are odd.
```

How it works...

本节并没有设计特别复杂的使用场景，这样就能让我们集中精力与标准库函数的调用上。并行版本的算法和标准串行的算法并没有什么区别。其差别就是多了一个参数，也就是执行策略。

让我们结合以下代码，来看三个核心问题：

```
generate(execution::par, begin(d), end(d), rand_num);
sort( execution::par, begin(d), end(d));
reverse( execution::par, begin(d), end(d));

auto odds = count_if(execution::par, begin(d), end(d),
odd);
```

哪些**STL**可以使用这种方式进行并行？

69种存在的**STL**算法在C++17标准中，都可以使用这种方式进行并行，还有7种新算法也支持并行。虽然这种升级对于很多实现来说很伤，但是也只是在接口上增加了一个参数——执行策略参数。这也不是意味着我们总要提供一个执行策略参数。并且执行策略参数放在了第一个参数的位置上。

这里有69个升级了的算法。并且有7个新算法在一一开始就支持了并发：

```
adjacent difference, adjacent find.  
all_of, any_of, none_of  
copy  
count  
equal  
fill  
find  
generate  
includes  
inner product  
in place merge, merge  
is heap, is partitioned, is sorted  
lexicographical_compare  
min element, minmax element  
mismatch  
move  
n-th element  
partial sort, sort copy  
partition  
remove + variations  
replace + variations  
reverse / rotate  
search  
set difference / intersection / union /symmetric  
difference  
sort  
stable partition  
swap ranges  
transform  
unique
```

详细的内容可以查看[C++ Reference](#)。[\(参考页面\)](#)

这些算法的升级是一件令人振奋的事！如果我们之前的程序使用了很多的**STL**算法，那么就很容易的将这些算法进行并行。这里需要注意的是，这样的的改变并不意味着每个程序自动化运行N次都会很快，因为多核编程更为复杂，所要注意的事情更多。

不过，在这之前我们现在都会用 `std::thread`，`std::async` 或是第三方库进行复杂的并行算法设计，而现在我们可以以更加优雅、与操作系统不相关的方式进行算法的并行化。

执行策略是如何工作的？

执行策略会告诉我们的标准函数，以何种方式进行自动化并行。

`std::execution` 命名空间下面，有三种策略类型：

策略	含义
<code>sequenced_policy</code>	算法使用串行的方式执行，这与原始执行方式没有什么区别。全局可用的实例命名为 <code>std::execution::seq</code> 。
<code>parallel_policy</code>	算法使用多线程的方式进行执行。全局可用的实例命名为 <code>std::execution::par</code> 。
<code>parallel_unsequenced_policy</code>	算法使用多线程的方式进行执行。并允许对代码进行向量化。在这个例子中，线程间可以对内存进行交叉访问，向量化的内客可以在同一个线程中执行。全局可用的实例命名为 <code>std::execution::par_unseq</code> 。

执行策略意味着我们需要进行严格限制。严格的约定，让我们有更多并行策略可以使用：

- 并行算法对所有元素的访问，必须不能导致死锁或数据竞争。
- 向量化和并行化中，所有可访问的函数不能使用任何一种阻塞式同步。

我们需要遵守这些规则，这样才不会将错误引入到程序中。

Note:

STL的自动并行化，并总能保证有加速。因为具体的情况都不一样，所以可能在很多情况下并行化并没有加速。多核编程还是很有难度的。

向量化是什么意思？

向量化的特性需要编译器和CPU都支持，让我们先来简单的了解一下向量化是如何工作的。假设我们有一个非常大的 `vector`。简单的实现可以写成如下的方式：

```
std::vector<int> v {1, 2, 3, 4, 5, 6, 7 /*...*/};

int sum {std::accumulate(v.begin(), v.end(), 0)};
```

编译器将会生成一个对 `accumulate` 调用的循环，其可能与下面代码类似：

```
int sum {0};

for (size_t i {0}; i < v.size(); ++i) {
    sum += v[i];
}
```

从这点说起，当编译器开启向量化时，就会生成类似如下的代码。每次循环会进行4次累加，这样循环次数就要比之前减少4倍。为了简单说明问题，我们这里没有考虑不为4倍数个元素的情况：

```
int sum {0};  
for (size_t i {0}; i < v.size() / 4; i += 4) {  
    sum += v[i] + v[i+1] + v[i + 2] + v[i + 3];  
}  
// if v.size() / 4 has a remainder,  
// real code has to deal with that also.
```

为什么要这样做呢？很多CPU指令都能支持这种操作 `sum += v[i] + v[i+1] + v[i+2] + v[i+3];`，只需要一个指令就能完成。使用尽可能少的指令完成尽可能多的操作，这样就能加速程序的运行。

自动向量化非常困难，因为编译器需非常了解我们的程序，这样才能进行加速的情况下，不让程序的结果出错。目前，至少可以通过使用标准算法来帮助编译器。因为这样能让编译器更加了解哪些数据流能够并行，而不是从复杂的循环中对数据流的依赖进行分析。

让程序在特定时间休眠

C++11中对于线程的控制非常优雅和简单。在 `this_thread` 的命名空间中，包含了只能被运行线程调用的函数。其包含了两个不同的函数，让线程睡眠一段时间，这样就不需要使用任何额外的库，或是操作系统依赖的库来执行这个任务。

本节中，我们将关注于如何将线程暂停一段时间，或是让其休眠一段时间。

How to do it...

我们将完成一个短小的程序，并让主线程休眠一段时间：

1. 包含必要的头文件，并声明所使用的命名空间。`chrono_literals` 空间包含一段时间的缩略值：

```
#include <iostream>
#include <chrono>
#include <thread>

using namespace std;
using namespace chrono_literals;
```

2. 我们直接写主函数，并让主线程休眠5秒和300毫秒。感谢 `chrono_literals`，我们可以写成一种非常可读方式：

```
int main()
{
    cout << "Going to sleep for 5 seconds"
        " and 300 milli seconds.\n";

    this_thread::sleep_for(5s + 300ms);
```

3. 休眠状态是相对的。当然，我们能用绝对时间来表示。让休眠直到某个时间点才终止，这里我们在 `now` 的基础上加了3秒：

```
cout << "Going to sleep for another 3
seconds.\n";

this_thread::sleep_until(
    chrono::high_resolution_clock::now() + 3s);
```

4. 在程序退出之前，我们会打印一个表示睡眠结束：

```
cout << "That's it.\n";
}
```

5. 编译并运行程序，我们就能获得如下的输出。Linux, Mac或其他类似UNIX的操作系统会提供**time**命令，其能对一个可运行程序的耗时进行统计。使用**time**对我们的程序进行耗时统计，其告诉我们花费了8.32秒，因为我们让主线程休眠了5.3秒和3秒。最后还有一个打印，用来告诉我们主函数的休眠终止：

```
$ time ./sleep
Going to sleep for 5 seconds and 300 milli seconds.
Going to sleep for another 3 seconds.
That's it.

real 0m8.320s
user 0m0.005s
sys 0m0.003s
```

How it works...

`sleep_for` 和 `sleep_until` 函数都已经在C++11中加入，存放于 `std::this_thread` 命名空间中。其能对当前线程进行限时阻塞(并不是整个程序或整个进程)。线程被阻塞时不会消耗CPU时间，操作系统会将其标记挂起的状态，时间到了后线程会自动醒来。这种方式的好处在于，不需要知道操作系统对我们运行的程序做了什么，因为STL会将其中的细节进行包装。

`this_thread::sleep_for` 函数能够接受一个 `chrono::duration` 值。最简单的方式就是 `1s` 或 `5s+300ms`。为了使用这种非常简洁的字面时间表示方式，我们需要对命名空间进行声明 `using namespace std::chrono_literals;`。

`this_thread::sleep_until` 函数能够接受一个 `chrono::time_point` 参数。这就能够简单的指定对应的壁挂钟时间，来限定线程休眠的时间。

唤醒时间和操作系统的时间的精度一样。大多数操作系统的时间精度通常够用，但是其可能对于一些时间敏感的应用非常不利。

另一种让线程休眠一段时间的方式是使用 `this_thread::yield`。其没有参数，也就意味着这个函数不知道这个线程要休眠多长时间。所以，这个函数并不建议用来对线程进行休眠或停滞一个线程。其只会以协作的方式告诉操作系统，让操作系统对线程和进程重新进行调度。如果没有其他可以调度的线程或进程，那么这个“休眠”线程则会立即执行。正因为如此，很少用 `yield` 让线程休眠一段时间。

启动和停止线程

C++11中添加了 `std::thread` 类，并能使用简洁的方式能够对线程进行启动或停止，线程相关的东西都包含在STL中，并不需要额外的库或是操作系统的实现来对其进行支持。

本节中，我们将实现一个程序对线程进行启动和停止。如果是第一次使用线程的话，就需要了解一些细节。

How to do it...

我们将会使用多线程进行编程，并且会了解到，当程序的某些部分使用多线程时，代码会如何进行操作：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <thread>

using namespace std;
using namespace chrono_literals;
```

2. 启动一个线程时，我们需要告诉代码如何执行。所以，先来定义一个函数，这个函数会在线程中执行。这个函数可接受一个参数 `i`，可以看作为线程的ID，这样就可以了解打印输出对应的是哪个线程。另外，我们使用线程ID来控制线程休眠的时间，避免多个线程在同时执行 `cout`。如果出现了同时打印的情况，那就会影响到输出。本章的另一个章节会来详述这个问题：

```
static void thread_with_param(int i)
{
    this_thread::sleep_for(1ms * i);

    cout << "Hello from thread " << i << '\n';

    this_thread::sleep_for(1s * i);

    cout << "Bye from thread " << i << '\n';
}
```

3. 主函数中，会先了解在所使用的系统中能够同时运行多少个线程，使用 `std::thread::hardware_concurrency` 进行确定。这个数值通常依赖于机器上有多少个核，或是STL实现中支持多少个核。这也就意味着，对于不同机器，这个函数会返回不同的值：

```
int main()
{
    cout << thread::hardware_concurrency()
        << " concurrent threads are supported.\n";
```

4. 现在让我们来启动线程，每个线程的ID是不一样的，这里我们启动三个线程。我们使用实例化线程的代码行为 `thread t {f, x}`，这就等于在新线程中调用 `f(x)`。这样，在不同的线程中就可以给予 `thread_with_param` 函数不同的参数：

```
thread t1 {thread_with_param, 1};
thread t2 {thread_with_param, 2};
thread t3 {thread_with_param, 3};
```

5. 当启动线程后，我们就需要在其完成其工作后将线程进行终止，使用 `join` 函数来停止线程。调用 `join` 将会阻塞调用线程，直至对应的线程终止为止：

```
t1.join();
t2.join();
```

6. 另一种方式终止的方式是分离。如果不以 `join` 或 `detach` 的方式进行终止，那么程序只有在 `thread` 对象析构时才会终止。通过调用 `detech`，我们将告诉3号线程，即使主线程终止了，你也可以继续运行：

```
t3.detach();
```

7. 主函数结束前将打印一段信息：

```
cout << "Threads joined.\n";
}
```

8. 编译并运行程序，就会得到如下的输出。我们可以看到我们的机器上有8个CPU核。然后，我们可以看到每个线程中打印出的*hello*讯息，但是在主线程最后，我们只对两个线程使用 `join`。第3个线程等待了3秒，但是再主线程结束的时候，其只完成了2秒的等待。这样，我们就没有办法看到线程3的结束信息，因为主函数在结束之后，我们就没有任何机会将其进行杀死了：

```
$ ./threads
8 concurrent threads are supported.
Hello from thread 1
Hello from thread 2
Hello from thread 3
Bye from thread 1
Bye from thread 2
Threads joined.
```

How it works...

启动和停止线程其实没有什么困难的。多线程编程的难点在于，如何让线程在一起工作(共享资源、互相等待，等等)。

为了启动一个线程，我们首先定义一些执行函数。没有特定的规定，普通的函数就可以。我们来看一个简化的例子，启动线程并等待线程结束：

```
void f(int i) { cout << i << '\n'; }

int main()
{
    thread t {f, 123};
    t.join();
}
```

`std::thread` 的构造函数允许传入一个函数指针或一个可调用的对象，通过这个参数，我们就可以对函数进行调用。当然，我们也可以使用没有任何参数的函数。

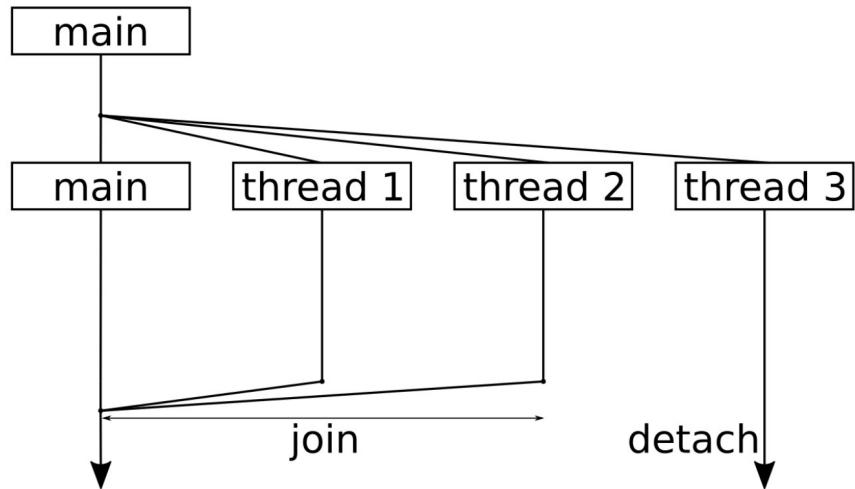
如果系统中有多个CPU核，那么线程就可以并行或并发的运行。并行与并发之间有什么区别呢？当计算机只有一个CPU核时，也可以有很多线程并行，但就不可能是并发的了，因为在单核CPU上，每个时间片上只有一个线程在执行。线程在单核上交错着运行，当一个时间片结束后，会对下一个线程进行执行(不过对于使用者来说，看起来就像是同时在运行)。如果线程间可以不去分享一个CPU和，那么这些线程就是并发运行。其实并发才是真正的同时运行。

这样，以下几点是我们绝对无法控制的：

- 共享一个CPU核时，无法控制线程交替运行的顺序。
- 线程也是有优先级的，优先级会影响线程执行的顺序。
- 实际上线程是分布在所有CPU核上的，当然操作系统也可以将线程绑定在一个核上。这也就意味着所有的线程可以运行在单核上，也可以运行在具有100个CPU核的机器上。

大多数操作系统都会提供对多线程编程提供一些可能性，不过这些特性并没包含在STL中。

在启动和停止线程的时候，告诉他们要做什么样的工作，并且什么时候线程停止工作。对于大多数应用来说就够用了。本节中，我们启动的3个线程。之后，对其中两个进行了 `join`，另一个进行 `detach`。让我们使用一个简单的图来总结一下本节的代码：



这幅图的顺序是自顶向下，你会看到我们将整个程序分成了4个线程。一开始，启动了额外3个线程来完成一些事情，之后主线程仅等待其他线程的结束。

线程结束对函数的执行后，会从函数中返回。标准库会进行相关的操作，将线程从操作系统的中删除，或用其他方式销毁，所以这里就不用操心了。

我们需要关心的就是 `join`。当对线程对象调用函数 `x.join()` 时，其会让调用线程休眠，直至 `x` 线程返回。如果线程处于一个无限循环中，就意味着程序无法终止。如果想要一个线程继续存活，并持续到自己结束的时候，那就可以调用 `x.detach()`。之后，就不会在等这个线程了。不管我们怎么做——都必须 `join` 或 `detach` 线程。如果不使用这两种方式，可以在线程对象的析构函数中调用 `std::terminate()`，这个函数会让程序“突然死亡”。

主函数返回时，整个程序也就结束了。不过，第3个线程 `t3` 还在等待，并将对应的信息打印到终端。操作系统才不在乎——会直接将我们的程序终止，并不管是否有线程还未结束。要怎么解决这个问题，就是开发者要考虑的事情了。

打造异常安全的共享锁—— `std::unique_lock`和`std::shared_lock`

由于对于线程的操作严重依赖于操作系统，所以STL提供与系统无关的接口是非常明智的，当然STL也会提供线程间的同步操作。这样就不仅是能够启动和停止线程，使用STL库也能完成线程的同步操作。

本节中，我们将了解到STL中的互斥量和RAII锁。我们使用这些工具对线程进行同步时，也会了解STL中更多同步辅助的方式。

How to do it...

我们将使用`std::shared_mutex`在独占(exclusive)和共享(shared)模式下来完成一段程序，并且也会了解到这两种方式意味着什么。另外，我们将不会对手动的对程序进行上锁和解锁的操作，这些操作都交给RAII辅助函数来完成：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <shared_mutex>
#include <thread>
#include <vector>

using namespace std;
using namespace chrono_literals;
```

2. 整个程序都会围绕共享互斥量展开，为了简单，我们定义了一个全局实例：

```
shared_mutex shared_mut;
```

3. 接下来，我们将会使用`std::shared_lock`和`std::unique_lock`这两个RAII辅助者。为了让其类型看起来没那么复杂，这里进行别名操作：

```
using shrd_lck = shared_lock<shared_mutex>;
using uniq_lck = unique_lock<shared_mutex>;
```

4. 开始写主函数之前，先使用互斥锁的独占模式来实现两个辅助函数。下面的函数中，我们将使用`unique_lock`实例来作为共享互斥锁。其构造函数的第二个参数`defer_lock`会告诉对象让锁处于解锁的状态。否则，构造函数会尝试对互斥量上锁阻塞程序，直至成功为止。然后，会对`exclusive_lock`的成员函数`try_lock`进行调用。该函数会立即返回，并且返回相应的布尔值，布尔值表示互斥量是否已经上锁，或是在其他地方已经锁住：

```

static void print_exclusive()
{
    uniq_lck l {shared_mut, defer_lock};

    if (l.try_lock()) {
        cout << "Got exclusive lock.\n";
    } else {
        cout << "Unable to lock exclusively.\n";
    }
}

```

5. 另一个函数也差不多。其会将程序阻塞，直至其获取相应的锁。然后，会使用抛出异常的方式来模拟发生错误的情况(只会返回一个整数，而非一个非常复杂的异常对象)。虽然，其会立即退出，并且在上下文中我们获取了一个锁住的互斥量，但是这个互斥量也可以被释放。这是因为 `unique_lock` 的析构函数在任何情况下都会将对应的锁进行释放：

```

static void exclusive_throw()
{
    uniq_lck l {shared_mut};
    throw 123;
}

```

6. 现在，让我们来写主函数。首先，先开一个新的代码段，并且实例化一个 `shared_lock` 实例。其构造函数将会立即对共享模式下的互斥量进行上锁。我们将会在下一步了解到这一动作的意义：

```

int main()
{
{
    shrd_lck s11 {shared_mut};

    cout << "shared lock once.\n";
}

```

7. 现在我们开启另一个代码段，并使用同一个互斥量实例化第二个 `shared_lock` 实例。现在具有两个 `shared_lock` 实例，并且都具有同一个互斥量的共享锁。实际上，可以使用同一个互斥量实例化很多的 `shared_lock` 实例。然后，调用 `print_exclusive`，其会尝试使用互斥量的独占模式对互斥量进行上锁。这样的调用当然不会成功，因为互斥量已经在共享模式下锁住了：

```

{
    shrd_lck sl2 {shared_mut};

    cout << "shared lock twice.\n";

    print_exclusive();
}

```

8. 离开这个代码段后，`shared_locks12` 的析构函数将会释放互斥量的共享锁。`print_exclusive` 函数还是失败，这是因为互斥量依旧处于共享锁模式：

```

cout << "shared lock once again.\n";

print_exclusive();
}

cout << "lock is free.\n";

```

9. 离开这个代码段时，所有 `shared_lock` 对象就都被销毁了，并且互斥量再次处于解锁状态，现在我们可以在独占模式下对互斥量进行上锁了。调用 `exclusive_throw`，然后调用 `print_exclusive`。不过因为 `unique_lock` 是一个 RAII 对象，所以是异常安全的，也就是无论 `exclusive_throw` 返回了什么，互斥量最后都会再次解锁。这样即便是互斥量处于锁定状态，`print_exclusive` 也不会被错误的状态所阻塞：

```

try {
    exclusive_throw();
} catch (int e) {
    cout << "Got exception " << e << '\n';
}

print_exclusive();
}

```

10. 编译并运行这段代码则会得到如下的输出。前两行展示的是两个共享锁实例。然后，`print_exclusive` 函数无法使用独占模式对互斥量上锁。在离开内部代码段后，第二个共享锁解锁，`print_exclusive` 函数依旧会失败。在离开这个代码段后，将会对互斥量所持有的锁进行释放，这样 `exclusive_throw` 和 `print_exclusive` 最终才能对互斥量进行上锁：

```

$ ./shared_lock
shared lock once.
shared lock twice.
Unable to lock exclusively.
shared lock once again.
Unable to lock exclusively.
lock is free.
Got exception 123
Got exclusive lock.

```

How it works...

查阅C++文档时，我们会对不同的互斥量和RAII辅助锁有些困惑。在我们回看这节的代码之前，让我们来对STL的这两个部分进行总结。

互斥量

其为**mutual exclusion**的缩写。并发时不同的线程对于相关的共享数据同时进行修改时，可能会造成结果错误，我们在这里就可以使用互斥量对象来避免这种情况的发生，STL提供了不同特性的互斥量。不过，这些互斥量的共同点就是具有`lock` 和 `unlock` 两个成员函数。

一个互斥量在解锁状态下时，当有线程对其使用`lock()` 时，这个线程就获取了互斥量，并对互斥量进行上锁。这样，但其他线程要对这互斥量进行上锁时，就会处于阻塞状态，知道第一个线程对该互斥量进行释放。`std::mutex` 就可以做到。

这里将STL一些不同的互斥量进行对比：

类型名	描述
<code>mutex</code>	具有 <code>lock</code> 和 <code>unlock</code> 成员函数的标准互斥量。并提供非阻塞函数 <code>try_lock</code> 成员函数。
<code>timed_mutex</code>	与互斥量相同，并提供 <code>try_lock_for</code> 和 <code>try_lock_until</code> 成员函数，其能在某个时间段内对程序进行阻塞。
<code>recursive_mutex</code>	与互斥量相同，不过当一个线程对实例进行上锁，其可以对同一个互斥量对象多次调用 <code>lock</code> 而不产生阻塞。持有线程可以多次调用 <code>unlock</code> ，不过需要和 <code>lock</code> 调用的次数匹配时，线程才不再拥有这个互斥量。
<code>recursive_timed_mutex</code>	提供与 <code>timed_mutex</code> 和 <code>recursive_mutex</code> 的特性。
<code>shared_mutex</code>	这个互斥量在这方面比较特殊，它可以被锁定为独占模式或共享模式。独占模式时，其与标准互斥量的行为一样。共享模式时，其他线程也可能在共享模式下对其进行上锁。其会在最后一个处于共享模式下的锁拥有者进行解锁时，整个互斥量才会解锁。其行为有些类似于 <code>shared_ptr</code> ，只不过互斥量不对内存进行管理，而是对锁的所有权进行管理。
<code>shared_timed_mutex</code>	同时具有 <code>shared_mutex</code> 和 <code>timed_mutex</code> 两种互斥量独占模式和共享模式的特性。

锁

线程对互斥量上锁之后，很多事情都变得非常简单，我们只需要上锁、访问、解锁三步就能完成我们想要做的工作。不过对于有些比较健忘的开发者来说，在上锁之后，很容易忘记对其进行解锁，或是互斥量在上锁状态下抛出一个异常，如果要对这个异常进行处理，那么代码就会变得很难看。最优的方式就是程序能够自动来处理这种事情。这种问题很类似与内存泄漏，开发者在分配内存之后，忘记使用 `delete` 操作进行内存释放。

内存管理部分，我们有 `unique_ptr`，`shared_ptr` 和 `weak_ptr`。这些辅助类可以很完美帮我们避免内存泄漏。互斥量也有类似的帮手，最简单的一个就是 `std::lock_guard`。使用方式如下：

```
void critical_function()
{
    lock_guard<mutex> l {some_mutex};

    // critical section
}
```

`lock_guard` 的构造函数能接受一个互斥量，其会立即自动调用 `lock`，构造函数会直到获取互斥锁为止。当实例进行销毁时，其会对互斥量再次进行解锁。这样互斥量就很难陷入到 `lock/unlock` 循环错误中。

C++17 STL提供了如下的RAII辅助锁。其都能接受一个模板参数，其与互斥量的类型相同(在C++17中，编译器可以自动推断出相应的类型)：

名称	描述
<code>lock_guard</code>	这个类没有什么其他的，构造函数中调用 <code>lock</code> ，析构函数中调用 <code>unlock</code> 。
<code>scoped_lock</code>	与 <code>lock_guard</code> 类似，构造函数支持多个互斥量。析构函数中会以相反的顺序进行解锁。
<code>unique_lock</code>	使用独占模式对互斥量进行上锁。构造函数也能接受一个参数用于表示超时到的时间，并不会让锁一直处于上锁的状态。其也可能不对互斥量进行上锁，或是假设互斥量已经锁定，或是尝试对互斥量进行上锁。另外，函数可以在 <code>unique_lock</code> 锁的声明周期中，对互斥量进行上锁或解锁。
<code>shared_lock</code>	与 <code>unique_lock</code> 类似，不过所有操作都是在互斥量的共享模式下进行操作。

`lock_guard` 和 `scoped_lock` 只拥有构造和析构函数，`unique_lock` 和 `shared_lock` 就比较复杂了，但也更为通用。我们将在本章的后续章节中了解到，这些类型如何用于更加复杂的情况。

现在我们来回看一下本节的代码。虽然，只在单线程的上下文中运行程序，但是我们可以了解到如何对辅助锁进行使用。`shrd_lck` 类型为 `shared_lock<shared_mutex>` 的缩写，并且其允许我们在共享模式下对一个实例多次上锁。当 `s1` 和 `s2` 存在的情况下，`print_exclusive` 无法使用独占模式对互斥量进行上锁。

现在来看看处于独占模式的上锁函数：

```

int main()
{
{
    shrd_lck sl1 {shared_mut};
{
    shrd_lck sl2 {shared_mut};
    print_exclusive();
}
print_exclusive();
}

try {
    exclusive_throw();
} catch (int e) {
    cout << "Got exception " << e << '\n';
}

print_exclusive();
}

```

`exclusive_throw` 的返回也比较重要，即便是抛出异常退出，`exclusive_throw` 函数依旧会让互斥量再度锁上。

因为 `print_exclusive` 使用了一个奇怪的构造函数，我们就再来看一下这个函数：

```

void print_exclusive()
{
    uniq_lck l {shared_mut, defer_lock};

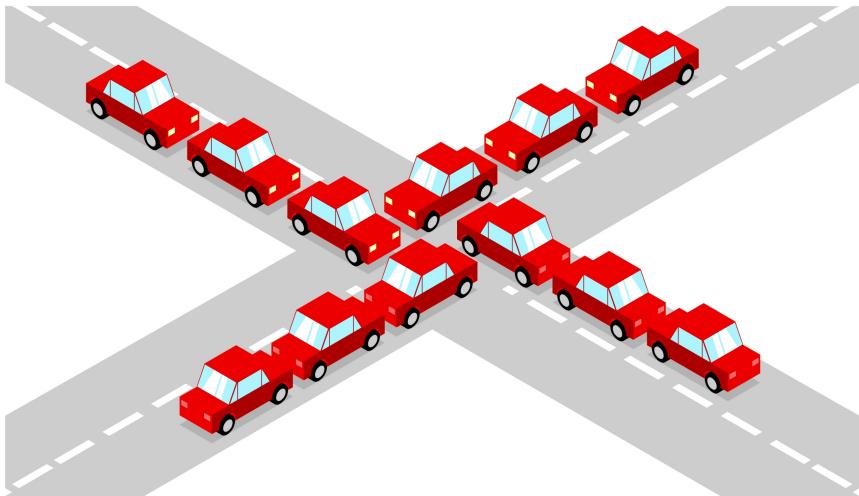
    if (l.try_lock()) {
        // ...
    }
}

```

这里我们不仅提供了 `shared_mut`，还有 `defer_lock` 作为 `unique_lock` 构造函数的参数。`defer_lock` 是一个空的全局对象，其不会对互斥量立即上锁，所以我们可以通过对这个参数对 `unique_lock` 不同的构造函数进行选择。这样做之后，我们可以调用 `l.try_lock()`，其会告诉我们有没有上锁。在互斥量上锁的情况下，就可以做些别的事情了。如果的确有机会获取锁，依旧需要析构函数对互斥量进行清理。

避免死锁——`std::scoped_lock`

如果在路上发生了死锁，就会像下图一样：



为了让交通顺畅，可能需要一个大型起重机，将路中间的一辆车挪到其他地方去。如果找不到起重机，那么我们就希望这些司机们能互相配合。当几个司机愿意将车往后退，留给空间给其他车通行，那么每辆车就不会停在原地了。

多线程编程中，开发者肯定需要避免这种情况的发生。不过，程序比较复杂的情况下，这种情况其实很容易发生。

本节中，我们将会故意的创造一个死锁的情况。然后，在相同资源的情况下，如何创造出一个死锁的情形。再使用C++17中，STL的`std::scoped_lock`如何避免死锁的发生。

How to do it...

本节中有两对函数要在并发的线程中执行，并且有两个互斥量。其中一对制造死锁，另一对解决死锁。主函数中，我们将使用这两个互斥量：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <thread>
#include <mutex>

using namespace std;
using namespace chrono_literals;
```

2. 实例化两个互斥量对象，制造死锁：

```
mutex mut_a;
mutex mut_b;
```

3. 为了使用两个互斥量制造死锁，我们需要有两个函数。其中一个函数试图对互斥量 A 进行上锁，然后对互斥量 B 进行上锁，而另一个函数则试图使用相反的方式运行。让两个函数在等待锁时进行休眠，我们确定这段代码永远处于一个死锁的状态。(这就达到了我们演示的目的。当我们重复运行程序，那么程序在没有任何休眠代码的同时，可能会有成功运行的情况。)需要注意的是，这里我们没有使用 \n 字符作为换行符，我们使用的是 endl。endl 会输出一个换行符，同时也会对 cout 的流缓冲区进行刷新，所以我们可以确保打印信息不会有延迟或同时出现：

```
static void deadlock_func_1()
{
    cout << "bad f1 acquiring mutex A..." << endl;

    lock_guard<mutex> la {mut_a};

    this_thread::sleep_for(100ms);

    cout << "bad f1 acquiring mutex B..." << endl;

    lock_guard<mutex> lb {mut_b};

    cout << "bad f1 got both mutexes." << endl;
}
```

4. deadlock_func_2 和 deadlock_func_1 看起来一样，就是 A 和 B 的顺序相反：

```
static void deadlock_func_2()
{
    cout << "bad f2 acquiring mutex B..." << endl;

    lock_guard<mutex> lb {mut_b};

    this_thread::sleep_for(100ms);

    cout << "bad f2 acquiring mutex A..." << endl;

    lock_guard<mutex> la {mut_a};

    cout << "bad f2 got both mutexes." << endl;
}
```

5. 现在我们将完成与上面函数相比，两个无死锁版本的函数。它们使用了 scoped_lock，其会作为构造函数参数的所有互斥量进行上锁。其析构函数会进行解锁操作。锁定这些互斥量时，其内部应用了避免死锁的策略。这里需要注意的是，两个函数还是对 A 和 B 互斥量进行操作，并且顺序相反：

```

static void sane_func_1()
{
    scoped_lock l {mut_a, mut_b};

    cout << "sane f1 got both mutexes." << endl;
}

static void sane_func_2()
{
    scoped_lock l {mut_b, mut_a};

    cout << "sane f2 got both mutexes." << endl;
}

```

6. 主函数中观察这两种情况。首先，我们使用不会死锁的函数：

```

int main()
{
{
    thread t1 {sane_func_1};
    thread t2 {sane_func_2};

    t1.join();
    t2.join();
}

```

7. 然后，调用制造死锁的函数：

```

{
    thread t1 {deadlock_func_1};
    thread t2 {deadlock_func_2};

    t1.join();
    t2.join();
}

```

8. 编译并运行程序，就能得到如下的输出。前两行为无死锁情况下，两个函数的打印结果。接下来的两个函数则产生死锁。因为我们能看到f1函数始终是在等待互斥量B，而f2则在等待互斥量A。两个函数都没做成功的对两个互斥量上锁。我们可以让这个程序持续运行，不管时间是多久，结果都不会变化。程序只能从外部进行杀死，这里我们使用 `ctrl + c` 的组合键，将程序终止：

```
$ ./avoid_deadlock
sane f1 got both mutexes
sane f2 got both mutexes
bad f2 acquiring mutex B...
bad f1 acquiring mutex A...
bad f1 acquiring mutex B...
bad f2 acquiring mutex A...
```

How it works...

例子中，我们故意制造了死锁，我们也了解了这样一种情况发生的有多快。在一个很大的项目中，多线程开发者在编写代码的时候，都会共享一些互斥量用于保护资源，所有开发者都需要遵循同一种加锁和解锁的顺序。这种策略或规则是很容易遵守的，不过也是很容易遗忘的。另一个问题则是**锁序倒置**。

`scoped_lock` 对于这种情况很有帮助。其实在C++17中添加，其工作原理与 `lock_guard` 和 `unique_lock` 一样：其构造函数会进行上锁操作，并且析构函数会对互斥量进行解锁操作。`scoped_lock` 特别之处是，可以指定多个互斥量。

`scoped_lock` 使用 `std::lock` 函数，其会调用一个特殊的算法对所提供的互斥量调用 `try_lock` 函数，这是为了避免死锁。因此，在加锁与解锁的顺序相同的情况下，使用 `scoped_lock` 或对同一组锁调用 `std::lock` 都是非常安全的。

同步并行中使用std::cout

多线程中的一个麻烦的地方在于，需要对并发线程所要访问的共享数据使用互斥量或其他方式进行保护，以避免让多线程修改失控。

其中 `std::cout` 打印函数通常被使用到。如果多个线程同时调用 `cout`，那么其输出将会混合在一起。为了避免输出混在一起，我们将要用我们的函数进行并发安全的打印。

我们将会了解到，如何完成对 `cout` 的包装，并使用最少量的代码进行最优的打印。

How to do it...

本节中，将实现一个并发打印安全的函数。避免将打印信息全部混在一起，我们实现了一个辅助类来帮助我们在线程间同步打印信息。

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <thread>
#include <mutex>
#include <sstream>
#include <vector>

using namespace std;
```

2. 然后实现辅助类，其名字为 `pcout`。其中字母p代表parallel，因为其会将并发的上下文进行同步。`pcout` 会 `public` 继承于 `stringstream`。这样，我们就能直接对其实例使用 `<<` 操作符了。当 `pcout` 实例销毁时，其析构函数会对一个互斥量进行加锁，然后将 `stringstream` 缓冲区中的内容进行打印。我们将在下一步了解，如何对这个类进行使用：

```
struct pcout : public stringstream {
    static inline mutex cout_mutex;

    ~pcout() {
        lock_guard<mutex> l {cout_mutex};
        cout << rdbuf();
        cout.flush();
    }
};
```

3. 现在，让我们来完成两个函数，这两个函数可运行在额外的线程上。每个线程都有一个线程ID作为参数。这两个函数的区别在于，第一个就是简单的使用 `cout` 进行打印。另一个使用 `pcout` 来进行打印。对应的实例都是一个临时变量，只存在于一行代码上。在所有 `<<` 调用执行完成后，我们想要的字符流

则就打印在屏幕上。然后，调用 `pcout` 实例的析构函数。我们可以了解到析构函数做了什么事：其对一个特定的互斥量进行上锁，所有 `pcout` 的实例都会这个互斥量进行共享：

```
static void print_cout(int id)
{
    cout << "cout hello from " << id << '\n';
}

static void print_pcout(int id)
{
    pcout{} << "pcout hello from " << id << '\n';
}
```

4. 首先，我们使用 `print_cout`，其会使用 `cout` 进行打印。我们并发的启动10个线程，使用其打印相应的字符串，并等待打印结束：

```
int main()
{
    vector<thread> v;

    for (size_t i {0}; i < 10; ++i) {
        v.emplace_back(print_cout, i);
    }

    for (auto &t : v) { t.join(); }
```

5. 然后，使用 `print_pcout` 来完成同样的事情：

```
cout << "=====\\n";

v.clear();
for (size_t i {0}; i < 10; ++i) {
    v.emplace_back(print_pcout, i);
}

for (auto &t : v) { t.join(); }
```

6. 编译并运行程序，我们就会得到如下的输出。如我们所见，前10行打印完全串行了。我们无法了解到哪条信息是由哪个线程所打印的。后10行的打印中，我们使用 `print_pcout` 进行打印，就不会造成任何串行的情况。可以清楚的看到不同线程所打印出的信息，因为每次运行的时候打印顺序都是以类似随机数的方式出现：

```

$ ./sync_cout
cout hello from cout hello from cout hello from cout hello from cout hello from
cout hello from cout hello from cout hello from cout hello from cout hello from
cout hello from 0123cout hello f
rom 45678

9

=====

pcout hello from 0
pcout hello from 2
pcout hello from 4
pcout hello from 1
pcout hello from 3
pcout hello from 5
pcout hello from 6
pcout hello from 7
pcout hello from 8
pcout hello from 9

```

How it works...

OK，我们已经构建了“cout包装器”，其可以在并发程序中串行化的对输出信息进行打印。其是如何工作的呢？

当我们一步一步的了解 pcout 的原理，就会发现其工作的原理并不神奇。首先，实现一个字符流，能接受我们输入的字符串：

```

stringstream ss;
ss << "This is some printed line " << 123 << '\n';

```

然后，其会对全局互斥量进行锁定：

```

{
    lock_guard<mutex> l {cout_mutex};
}

```

锁住的区域中，其能访问到字符流 ss，并对其进行打印。离开这个代码段时，对互斥锁进行释放。`cout.flush()` 这会告诉字符流对象立即将其内容打印到屏幕上。如果没有这一行，程序将会运行的更快，因为多次的打印可能会放在一起打印。我们的代码中，想立即看到打印信息，所以我们使用了 `flush`：

```

cout << ss.rdbuf();
cout.flush();
}

```

OK，这就很简单了吧，但每次都写这几行代码，就会让整体的代码变的很冗长。我们可以将 `stringstream` 的实例化简写为如下的方式：

```

stringstream{} << "This is some printed line " << 123 <<
'\n';

```

这个字符串流对象的实例，可以容纳我们想打印的任何字符，最后对字符串进行析构。字符串流的声明周期只在这一行内存在。之后，我们无法打印任何东西，因为我们无法对其进行访问。最后，哪段代码能访问流的内容呢？其就是 `stringstream` 的析构函数。

我们无法对 `stringstream` 实例的成员函数进行修改，但是可以对通过继承的方式包装成我们想要的类型：

```
struct pcout : public stringstream {
    ~pcout() {
        lock_guard<mutex> l {cout_mutex};
        cout << rdbuf();
        cout.flush();
    }
};
```

这个类依旧是一个字符串流，并且可以像字符串流一样对这个类型进行使用。不同的是，其会对互斥量进行上锁，并且将其内容使用 `cout` 进行输出。

我们也会将 `cout_mutex` 对象作为静态实例移入 `pcout` 结构体中，所以可以让不同的实例共享一个互斥量。

进行延迟初始化——`std::call_once`

有时我们有一些特定的代码段，可以在多个线程间并行执行，不过其中有一些功能需要在进行执行前，完成一次初始化的过程。一个很简单的方式，就是在程序进入并行前，执行已存在的准备函数。

这种方法有如下几个缺点：

- 当并行线程来自于一个库，使用者肯定会忘记调用准备函数。这样会让库函数不是那么容易的让人使用。
- 当准备函数特别复杂，并且在某些条件下我们要通过条件来判断，是否要执行这个准备函数。

本节中，我们将来了解一下 `std::call_once`，其能帮助使用简单且优雅的方式解决上面提到的问题。

How to do it...

我们将完成一个程序，我们使用多线程对同一段代码进行执行。虽然这里执行的是相同的代码，但是我们的准备函数只需要运行一次：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <thread>
#include <mutex>
#include <vector>

using namespace std;
```

2. 我们将使用 `std::call_once`。为了对其进行使用，需要对 `once_flag` 进行实例化。在对指定函数使用 `call_once` 时，需要对所有线程进行同步：

```
once_flag callflag;
```

3. 现在来定义一个只需要执行一次的函数，就让这个函数打印一个感叹号吧：

```
static void once_print()
{
    cout << '!';
}
```

4. 再来定义所有线程都会运行的函数。首先，要通过 `std::call_once` 调用 `once_print`。`call_once` 需要我们之前定义的变量 `callflag`。其会被用来对线程进行安排：

```
static void print(size_t x)
{
    std::call_once(callflag, once_print);
    cout << x;
}
```

5. OK，让我们启动10个线程，并且让他们使用 `print` 函数进行执行：

```
int main()
{
    vector<thread> v;

    for (size_t i {0}; i < 10; ++i) {
        v.emplace_back(print, i);
    }

    for (auto &t : v) { t.join(); }
    cout << '\n';
}
```

6. 编译并运行程序，我们就会得到如下的输出。首先，我们可以看到由 `once_print` 函数打印出的感叹号。然后，我么可以看到线程对应的ID号。另外，其会对所有线程进行同步，所以不会有ID在 `once_print` 函数执行前被打印：

```
$ ./call_once
!1239406758
```

How it works...

`std::call_once` 工作原理和栅栏类似。其能对一个函数(或是一个可调用的对象)进行访问。第一个线程达到 `call_once` 的线程会执行对应的函数。直到函数执行结束，其他线程才能不被 `call_once` 所阻塞。当第一个线程从准备函数中返回后，其他线程也就都结束了阻塞。

我们可以对这个过程进行安排，当有一个变量决定其他线程的运行时，线程则必须对这个变量进行等待，直到这个变量准备好了，所有变量才能运行。这个变量就是 `once_flag callflag;`。每一个 `call_once` 都需要一个 `once_flag` 实例作为参数，来表明预处理函数是否运行了一次。

另一个细节是：如果 `call_once` 执行失败了(因为准备函数抛出了异常)，那么下一个线程则会再去尝试执行(这种情况发生在下一次执行不抛出异常的时候)。

将执行的程序推到后台——`std::async`

当我们想要将一些可以执行的代码放在后台，可以用线程将这段程序运行起来。然后，我们就等待运行的结果就好：

```
std::thread t {my_function, arg1, arg2, ...};  
// do something else  
t.join(); // wait for thread to finish
```

这里 `t.join()` 并不会给我们 `my_function` 函数的返回值。为了获取返回值，需要先实现 `my_function` 函数，然后将其返回值存储到主线程能访问到的地方。如果这样的情况经常发生，我们就要重复的写很多代码。

C++11之后，`std::async` 能帮我们完成这项任务。我们将写一个简单的程序，并使用异步函数，让线程在同一时间内做很多事情。`std::async` 其实很强大，让我们先来了解其一方面。

How to do it...

我们将在一个程序中并发进行多个不同事情，不显式创建线程，这次使用 `std::async` 和 `std::future`：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>  
#include <iomanip>  
#include <map>  
#include <string>  
#include <algorithm>  
#include <iterator>  
#include <future>  
  
using namespace std;
```

2. 实现了三个函数，算是完成些很有趣的任务。第一个函数能够接收一个字符串，并且创建一个对于字符串中的字符进行统计的直方图：

```
static map<char, size_t> histogram(const string &s)  
{  
    map<char, size_t> m;  
  
    for (char c : s) { m[c] += 1; }  
  
    return m;  
}
```

3. 第二个函数也能接收一个字符串，并返回一个排序后的副本：

```
static string sorted(string s)
{
    sort(begin(s), end(s));
    return s;
}
```

4. 第三个函数会对传入的字符串中元音字母进行计数：

```
static bool is_vowel(char c)
{
    char vowels[] {"aeiou"};
    return end(vowels) !=
        find(begin(vowels), end(vowels), c);
}

static size_t vowels(const string &s)
{
    return count_if(begin(s), end(s), is_vowel);
}
```

5. 主函数中，我们从标准输入中获取字符串。为了不让输入字符串分段，我们禁用了 `ios::skipws`。这样就能得到一个很长的字符串，并且不管这个字符串中有多少个空格。我们会对结果字符串使用 `pop_back`，因为这种方式会让一个字符串中包含太多的终止符：

```
int main()
{
    cin.unsetf(ios::skipws);
    string input {istream_iterator<char>{cin}, {}};
    input.pop_back();
```

6. 为了获取函数的返回值，并加快对输入字符串的处理速度，我们使用了异步的方式。`std::async` 函数能够接收一个策略和一个函数，以及函数对应的参数。我们对于这个三个函数均使用 `launch::async` 策略。并且，三个函数的输入参数是完全相同的：

```
auto hist (async(launch::async,
                  histogram, input));
auto sorted_str (async(launch::async,
                      sorted, input));
auto vowel_count (async(launch::async,
                        vowels, input));
```

7. `async` 的调用会立即返回，因为其并没有执行我们的函数。另外，准备好同步的结构体，用来获取函数所返回的结果。目前的结果使用不同的线程并发的进行计算。此时，我们可以做其他事情，之后再来获取函数的返回值。`hist`，`sorted_str` 和 `vowel_count` 分别为函数 `histogram`，`sorted` 和 `vowels` 的返回值，不过其会通过 `std::async` 包装入 `future` 类型中。这个对象表示在未来某个时间点上，对象将会获取返回值。通过对 `future` 对象使用 `.get()`，我们将会阻塞主函数，直到相应的值返回，然后再进行打印：

```
for (const auto &[c, count] : hist.get()) {
    cout << c << ":" << count << '\n';
}

cout << "Sorted string: "
<< quoted(sorted_str.get()) << '\n'
<< "Total vowels: "
<< vowel_count.get() << '\n';
}
```

8. 编译并运行代码，就能得到如下的输出。我们使用一个简短的字符串的例子时，代码并不是真正的在并行，但这个例子中，我们能确保代码是并发的。另外，程序的结构与串行版本相比，并没有改变多少：

```
$ echo "foo bar baz foobazinga" | ./async
: 3
a: 4
b: 3
f: 2
g: 1
i: 1
n: 1
o: 4
r: 1
z: 2
Sorted string:     aaaabbbffginoooorzz"
Total vowels: 9
```

How it works...

如果你没有使用过 `std::async`，那么代码可以简单的写成串行代码：

```

auto hist (histogram(input));
auto sorted_str (sorted( input));
auto vowel_count (vowels( input));

for (const auto &[c, count] : hist) {
    cout << c << ":" << count << '\n';
}
cout << "Sorted string: " << quoted(sorted_str) << '\n';
cout << "Total vowels: " << vowel_count << '\n';

```

下面的代码，则是并行的版本。我们将三个函数使用 `async(launch::async, ...)` 进行包装。这样三个函数都不会由主函数来完成。此外，`async` 会启动新线程，并让线程并发的完成这几个函数。这样我们只需要启动一个线程的开销，就能将对应的工作放在后台进行，而后可以继续执行其他代码：

```

auto hist (async(launch::async, histogram, input));
auto sorted_str (async(launch::async, sorted, input));
auto vowel_count (async(launch::async, vowels, input));

for (const auto &[c, count] : hist.get()) {
    cout << c << ":" << count << '\n';
}

cout << "Sorted string: "
<< quoted(sorted_str.get()) << '\n'
<< "Total vowels: "
<< vowel_count.get() << '\n';

```

例如 `histogram` 函数则会返回一个 `map` 实例，`async(..., histogram, ...)` 将返回给我们的 `map` 实例包装进之前就准备好的 `future` 对象中。`future` 对象时一种空的占位符，直到线程执行完函数返回时，才有具体的值。结果 `map` 将会返回到 `future` 对象中，所以我们可以对对象进行访问。`get` 函数能让我们得到被包装起来的结果。

让我们来看一个更加简单的例子。看一下下面的代码：

```

auto x (f(1, 2, 3));
cout << x;

```

与之前的代码相比，我们也可以以下面的方式完成代码：

```

auto x (async(launch::async, f, 1, 2, 3));
cout << x.get();

```

这都是最基本的。后台执行的方式可能要比标准C++出现还要早。当然，还有一个问题要解决：`launch::async` 是什么东西？`launch::async` 是一个用来定义执行策略的标识。其有两种独立方式和一种组合方式：

策略选择	意义	
<code>launch::async</code>	运行新线程，以异步执行任务	
<code>launch::deferred</code>	在调用线程上执行任务(惰性求值)。在对 <code>future</code> 调用 <code>get</code> 和 <code>wait</code> 的时候，才进行执行。如果什么都没有发生，那么执行函数就没有运行。	
<code>launch::async \</code>	<code>launch::deferred</code>	具有两种策略共同的特性，STL的 <code>async</code> 实现可以选择策略。当没有提供策略时，这种策略就作为默认的选择。

Note:

不使用策略参数调用 `async(f, 1, 2, 3)`，我们将会选择都是用的策略。`async` 的实现可以自由的选择策略。这也就意味着，我们不能确定任务会执行在一个新的线程上，还是执行在当前线程上。

There's more...

还有件事情我们必须要知道，假设我们写了如下的代码：

```
async(launch::async, f);
async(launch::async, g);
```

这就会让 `f` 和 `g` 函数并发执行(这个例子中，我们并不关心其返回值)。运行这段代码时，代码会阻塞在这两个调用上，这并不是我们想看到的情况。

所以，为什么会阻塞呢？`async` 不是非阻塞式、异步的调用吗？没错，不过这里有特殊：当对一个 `async` 使用 `launch::async` 策略时，获取一个 `future` 对象，之后其析构函数将会以阻塞式等待方式运行。

这也就意味着，这两次调用阻塞的原因就是，`future` 生命周期只有一行的时间！我们可以以获取其返回值的方式，来避免这个问题，从而让 `future` 对象的生命周期更长。

实现生产者/消费者模型—— **std::condition_variable**

本节中，我们将使用多线程实现一个经典的生产者/消费者模型。其过程就是一个生产者线程将商品放到队列中，然后另一个消费者线程对这个商品进行消费。如果不需要生产，生产者线程休眠。如果队列中没有商品能够消费，消费者休眠。

这里两个线程都需要对同一个队列进行修改，所以我们需要一个互斥量来保护这个队列。

需要考虑的事情是：如果队列中没有商品了，那么消费者做什么呢？需要每秒对队列进行检查，直到看到新的商品吗？当然，我们可以通过生产者触发一些事件叫醒消费者，这样消费者就能在第一时间获取到新的商品。

C++11中提供了一个很不错的数据结构 `std::condition_variable`，其很适合处理这样的情况。本节中，我们实现一个简单的生产者/消费者应用，来对这个数据结构进行使用。

How to do it...

我们将实现一个单生产者/消费者程序，每个角色都有自己的线程：

1. 包含必要的头文件，并且声明所使用的命名空间：

```
#include <iostream>
#include <queue>
#include <tuple>
#include <condition_variable>
#include <thread>

using namespace std;
using namespace chrono_literals;
```

2. 队列进行实例化，并且队列 `q` 里只放简单的数字。生产者将商品放入队列中，消费者将商品从队列中取出。为了进行同步，我们需要一个互斥量。这就需要我们对 `condition_variable` 进行实例化，其变量名为 `cv`。`finished` 变量将会告诉生产者，无需在继续生产：

```
queue<size_t> q;
mutex mut;
condition_variable cv;
bool finished {false};
```

3. 首先，我们来实现一个生产者函数。其能接受一个参数 `items`，其会限制生产者生产的最大数量。一个简单的循环中，我们将会隔100毫秒生产一个商品，这个耗时就是在模拟生产的复杂性。然后，我们会对队列的互斥量进行上锁，

并同步的对队列进行访问。成功的生产后，将商品加入队列时，我们需要调用 `cv.notify_all()`，函数会叫醒所有消费线程。我们将在后面看到消费者那边是如何工作的：

```
static void producer(size_t items) {
    for (size_t i {0}; i < items; ++i) {
        this_thread::sleep_for(100ms);
        {
            lock_guard<mutex> lk {mut};
            q.push(i);
        }
        cv.notify_all();
    }
}
```

4. 生产完所有商品后，我们会将互斥量再度上锁，因为需要对 `finished` 位进行设置。然后，再次调用 `cv.notify_all()`：

```
{
    lock_guard<mutex> lk {mut};
    finished = true;
}
cv.notify_all();
}
```

5. 现在来实现消费者函数。因为只是对队列上的数值进行消费，直到消费完所有的数值，所以这个函数不需要参数。当 `finished` 未被设置时，循环会持续执行，并且会对保护队列的互斥量进行上锁，将对队列和 `finished` 标识同时进行保护。当互斥量上锁，则锁就会调用 `cv.wait`，并以Lambda表达式为参数。这个Lambda表达式其实就是一个条件谓词，如果生产者线程还在继续工作，并且还有商品在队列上，消费者线程就不能停下来：

```
static void consumer() {
    while (!finished) {
        unique_lock<mutex> l {mut};

        cv.wait(l, [] { return !q.empty() || finished;
    });
}
```

6. `cv.wait` 的调用会对锁进行解锁，并且会等到给予的条件达成时才会继续运行。然后，其会再次对互斥量上锁，并对队列上的商品进行消费，直到队列为空。如果生成者还在继续生成，那么这个循环可能会一直进行下去。否则，当 `finished` 被设置时，循环将会终止，这也表示生产者不会再进行生产：

```

        while (!q.empty()) {
            cout << "Got " << q.front()
            << " from queue.\n";
            q.pop();
        }
    }
}

```

7. 主函数中，我们让生产者生产10个商品。然后，我们就等待程序的结束：

```

int main() {
    thread t1 {producer, 10};
    thread t2 {consumer};
    t1.join();
    t2.join();
    cout << "finished!\n";
}

```

8. 编译并运行程序，我们将会得到下面的输出。当程序在运行阶段时，我们将看到每一行，差不多隔100毫秒打印出来，因为生产者需要时间进行生产：

```

$ ./producer_consumer
Got 0 from queue.
Got 1 from queue.
Got 2 from queue.
Got 3 from queue.
Got 4 from queue.
Got 5 from queue.
Got 6 from queue.
Got 7 from queue.
Got 8 from queue.
Got 9 from queue.
finished!

```

How it works...

本节中，我们只启动了两个线程。第一个线程会生产一些商品，并放到队列中。另一个则是从队列中取走商品。当其中一个线程需要对队列进行访问时，其否需要对公共互斥量 `mut` 进行上锁，这样才能对队列进行访问。这样，我们就能保证两个线程不能再同时对队列进行操作。

除了队列和互斥量，我们还声明了2个变量，其也会对生产者和消费者有所影响：

```
queue<size_t> q;
mutex mut;
condition_variable cv;
bool finished {false};
```

`finished` 变量很容易解释。当其设置为`true`时，生产者则会对固定数量的产品进行生产。当消费者看到这个值为`true`的时候，其就要将队列中的商品全部消费完。但是 `condition_variable cv` 代表了什么呢？我们在两个不同上下文中使用 `cv`。其中一个上下文则会去等待一个特定的条件，并且另一个会达成对应的条件。

消费者这边将会等待一个特殊的条件。消费者线程会在对互斥量 `mut` 使用 `unique_lock` 上锁后，进行阻塞循环。然后，会调用 `cv.wait`：

```
while (!finished) {
    unique_lock<mutex> l{mut};

    cv.wait(l, [] { return !q.empty() || finished; });

    while (!q.empty()) {
        // consume
    }
}
```

我们写了下面一段代码，这上下来两段代码看起来是等价的。我们会在后面了解到，这两段代码真正的区别到底在哪里：

```
while (!finished) {
    unique_lock<mutex> l{mut};

    while (q.empty() && !finished) {
        l.unlock();
        l.lock();
    }

    while (!q.empty()) {
        // consume
    }
}
```

这就意味着，我们先要进行上锁，然后对我们的应对方案进行检查：

1. 还有商品能够消费吗？有的话，继续持有锁，消费，释放锁，结束。
2. 如果没有商品可以消费，但是生产者依旧存在，释放互斥锁，也就是给生产者一个机会向队列中添加商品。然后，尝试再对现状进行检查，如果现状有变，则跳转到1方案中。

`cv.wait` 为什么与 `while(q.empty() && ...)` 不等价呢？因为在 `wait` 不需要再循环中持续的进行上锁和解锁的循环。如果生产者线程处于未激活状态时，这就会导致互斥量持续的被上锁和解锁，这样的操作是没有意义的，而且还会耗费掉不必要的 CPU 周期。

`cv.wait(lock, predicate)` 将会等到 `predicate()` 返回 `true` 时，结束等待。不过其不会对 `lock` 持续的进行解锁与上锁的操作。为了将使用 `wait` 阻塞的线程唤醒，我们就需要使用 `condition_variable` 对象，另一个线程会对同一个对象调用 `notify_one()` 或 `notify_all()`。等待中的线程将从休眠中醒来，并检查 `predicate()` 条件是否成立。

`wait` 还有一个很好的地方在于，如果出现了伪唤醒操作，那么线程则会再次进行休眠状态。这也就是当我们发出了过多的叫醒信号时，其不会对程序流有任何影响（但是会影响到性能）。

生产者端，在向队列输出商品后，调用 `cv.notify_all()`，并且在生产最后一个商品时，将 `finished` 设置为 `true`，这就等于引导了消费者进行消费。

实现多生产者/多消费者模型—— **std::condition_variable**

让我们再来看看一下生产者/消费者问题，这要比上一节的问题更加复杂。我们创建了多个生成者和多个消费者。并且，我们定义的队列没有限制上限。

当队列中没有商品时，消费者会处于等待状态，而当队列中没有空间可以盛放商品时，生产者也会处于等待状态。

我们将会使用多个 `std::condition_variable` 对象来解决这个问题，不过使用的方式与上节有些不同。

How to do it...

本节中，我们将实现一个类似于上节的程序，这次我们有多个生产者和消费者：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <iomanip>
#include <sstream>
#include <vector>
#include <queue>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <chrono>

using namespace std;
using namespace chrono_literals;
```

2. 接下来从本章的其他小节中拿过来一个同步打印的辅助类型，因为其能帮助我们在大量并发时进行打印：

```
struct pcout : public stringstream {
    static inline mutex cout_mutex;
    ~pcout() {
        lock_guard<mutex> l {cout_mutex};
        cout << rdbuf();
    }
};
```

3. 所有生产者都会将值写入到同一个队列中，并且所有消费者也会从这个队列中获取值。对于这个队列，我们需要使用一个互斥量对队列进行保护，并设置一个标识，其会告诉我们生产者是否已经停止生产：

```
queue<size_t> q;
mutex q_mutex;
bool production_stopped {false};
```

4. 我们将在程序中使用两个 `condition_variable` 对象。单生产者/消费者的情况下，只需要一个 `condition_variable` 告诉我们队列上面摆放了新商品。本节的例子中，我们将处理更加复杂的情况。我们需要生产者持续生产，以保证队列上一直有可消费的商品存在。如果商品囤积到一定程度，则生产者休息。`go_consume` 变量就用来提醒消费者消费的，而 `go_produce` 则是用来提醒生产者进行生产的：

```
condition_variable go_produce;
condition_variable go_consume;
```

5. 生产者函数能够接受一个生产者ID，所要生产的商品数量，以及囤积商品值的上限。然后，生产者就会进入循环生产阶段。这里，首先其会对队列的互斥量进行上锁，然后在通过调用 `go_produce.wait` 对互斥量进行解锁。队列上的商品数量未达到囤积阈值时，满足等待条件：

```
static void producer(size_t id, size_t items, size_t stock)
{
    for (size_t i = 0; i < items; ++i) {
        unique_lock<mutex> lock(q_mutex);
        go_produce.wait(lock,
                         [&] { return q.size() < stock; });
    }
}
```

6. 生产者开始生产后，就会有商品放入队列中。队列商品的表达式为 `id * 100 + i`。因为百位和线程id强相关，这样我们就能了解到哪些商品是哪些生产者生产的。我们也能将生产事件打印到终端上。格式看起来可能有些奇怪，不过其会与消费者打印输出对齐：

```
q.push(id * 100 + i);

pcout{} << " Producer " << id << " --> item "
    << setw(3) << q.back() << '\n';
```

7. 生产商品之后，我们叫醒沉睡中的消费者。每个睡眠周期为90毫秒，这用来模拟生产者生产商品的时间：

```
go_consume.notify_all();
this_thread::sleep_for(90ms);
}

pcout{} << "EXIT: Producer " << id << '\n';
}
```

8. 现在来实现消费者函数，其只接受一个消费者ID作为参数。当生产者停止生产，或是队列上没有商品，消费者就会继续等待。队列上没有商品时，生产者还在生产的话，那么可以肯定的是，队列上肯定会有新商品的产生：

```
static void consumer(size_t id)
{
    while (!production_stopped || !q.empty()) {
        unique_lock<mutex> lock(q_mutex);
```

9. 对队列的互斥量上锁之后，我们将会在等待 `go_consume` 事件变量时对互斥量进行解锁。`Lambda`表达式表明，当队列中有商品的时候我们就会对其进行获取。第二个参数 `1s` 说明，我们并不想等太久。如果等待时间超过1秒，我们将不会等待。当谓词条件达成，则 `wait_for` 函数返回；否则就会因为超时而放弃等待。如果队列中有新商品，我们将会获取这个商品，并进行相应的打印：

```
if (go_consume.wait_for(lock, 1s,
    [] { return !q.empty(); })) {
    pcout{} << " item "
        << setw(3) << q.front()
        << " --> Consumer "
        << id << '\n';
    q.pop();
```

10. 商品被消费之后，我们将会提醒生产者，并进入130毫秒的休眠状态，这个时间用来模拟消费时间：

```
    go_produce.notify_all();
    this_thread::sleep_for(130ms);
}
```

```
}
```

11. 主函数中，我们对工作线程和消费线程各自创建一个 `vector`：

```
int main()
{
    vector<thread> workers;
    vector<thread> consumers;
```

12. 然后，我们创建3个生产者和5个消费者：

```
    for (size_t i = 0; i < 3; ++i) {
        workers.emplace_back(producer, i, 15, 5);
    }

    for (size_t i = 0; i < 5; ++i) {
        consumers.emplace_back(consumer, i);
    }
```

13. 我们会先让生产者线程终止。然后返回，并对 `production_stopped` 标识进行设置，这将会让消费者线程同时停止。然后，我们要将这些线程进行回收，然后退出程序：

```
    for (auto &t : workers) { t.join(); }
    production_stopped = true;
    for (auto &t : consumers) { t.join(); }
}
```

14. 编译并运行程序，我们将获得如下的输出。输出特别长，我们进行了截断。我们能看到生产者偶尔会休息一下，并且消费者会消费掉对应的商品，直到再次生产。若是将生产者/消费者的休眠时间进行修改，则会得到完全不一样的结果：

```
$ ./multi_producer_consumer
Producer 0 --> item 0
Producer 1 --> item 100
item 0 --> Consumer 0
Producer 2 --> item 200
item 100 --> Consumer 1
item 200 --> Consumer 2
Producer 0 --> item 1
Producer 1 --> item 101
item 1 --> Consumer 0
...
Producer 0 --> item14
EXIT: Producer 0
Producer 1 --> item 114
EXIT: Producer 1
item14 --> Consumer 0
Producer 2 --> item 214
EXIT: Producer 2
item 114 --> Consumer 1
item 214 --> Consumer 2
EXIT: Consumer 2
EXIT: Consumer 3
EXIT: Consumer 4
EXIT: Consumer 0
EXIT: Consumer 1
```

How it works...

这节可以作为之前章节的扩展。与单生产者和消费者不同，我们实现了M个生产者和N个消费者之间的同步。因此，程序中不是消费者因为队列中没有商品而等待，就是因为队列中囤积了太多商品让生产者等待。

当有多个消费者等待同一个队列中出现新的商品时，程序的模式就又和单生产者/消费者工作的模式相同了。当有一个线程对保护队列的互斥量上锁时，然后对货物进行添加或减少，这样代码就是安全的。这样的话，无论有多少线程在同时等待这个锁，对于我们来说都无所谓。生产者也同理，其中最重要的就是，队列不允许两个及两个以上的线程进行访问。

比单生产者/消费者原理复杂的原因在于，当商品的数量在队列中囤积到一定程度，我们将会让生产者线程停止。为了迎合这个需求，我们使用两个不同的 `condition_variable`：

1. `go_produce` 表示队列没有被填满，并且生产者会继续生产，而后将商品放置在队列中。
2. `go_consume` 表示队列已经填满，消费者可以继续消费。

这样，生产者会将队列用货物填满，并且 `go_consume` 会用如下代码，提醒消费者线程：

```
if (go_consume.wait_for(lock, 1s, [] { return !q.empty(); })) {
    // got the event without timeout
}
```

生产者也会进行等待，直到可以再次生产：

```
go_produce.wait(lock, [&] { return q.size() < stock; });
```

还有一个细节就是我们不会让消费者线程等太久。在对 `go_consume.wait_for` 的调用中，我们添加了超时参数，并且设置为1秒。这对于消费者来说是一种退出机制：当队列为空的状态持续多于1秒，那么就可能没有生产者在工作。

这个处理起来很简单，代码会尽可能让队列中商品的数量达到阈值的上限。更复杂的程序中，当商品的数量为阈值上限的一半时，消费者线程会对生产者线程进行提醒。这样生产者就会在队列为空前继续生产。

`condition_variable` 帮助我们完美的解决了一个问题：当一个消费者触发了 `go_produce` 的提醒，那么将会有很多生产者竞争的去生产下一个商品。如果只需要生产一个商品，那么只需要一个生产者就好。当 `go_produce` 被触发时，所有生产者都争相生产这一个商品，我们将会看到的情况就是商品在队列中的数量超过了阈值的上限。

我们试想一下这种情况，我们有 `(max - 1)` 个商品在队列中，并且想在要一个商品将队列填满。不论是一个消费者线程调用了 `go_produce.notify_one()` (只叫醒一个等待线程)或 `go_produce.notify_all()` (叫醒所有等待的线程)，都需要保证只有一个生产者线程调用了 `go_produce.wait`，因为对于其他生成线程来说，一旦互斥锁解锁，那么 `q.size() < stock` (`stock` 货物阈值上限)的条件将不复存在。

并行ASCII曼德尔布罗特渲染器—— std::async

还记得第6章中的ASCII曼德尔布罗特渲染器吗？本节中，我们将使用多线程来加速其计算的过程。

原始代码中会限定每个坐标的迭代次数，坐标的迭代会让程序变得很慢，现在我们使用并行方式对其进行实现。

然后，我们对代码做少量的修改，并且将 `std::async` 和 `std::future` 加入到程序中，让程序运行的更快。想要完全理解本节，就要对原始的程序有个较为完整的了解。

How to do it...

本节中，我们将对曼德尔布罗特渲染器进行升级。首先，要提升对选定坐标迭代计算的次数。然后，通过程序并行化，来提高运行的速度：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <algorithm>
#include <iterator>
#include <complex>
#include <numeric>
#include <vector>
#include <future>

using namespace std;
```

2. `scaler` 和 `scaled_cmplx` 没有任何改动：

```

using cmplx = complex<double>

static auto scaler(int min_from, int max_from,
                  double min_to, double max_to)
{
    const int w_from {max_from - min_from};
    const double w_to {max_to - min_to};
    const int mid_from {(max_from - min_from) / 2 +
min_from};
    const double mid_to {(max_to - min_to) / 2.0 +
min_to};

    return [=] (int from) {
        return double(from - mid_from) / w_from *
w_to + mid_to;
    };
}

template <typename A, typename B>
static auto scaled_cmplx(A scaler_x, B scaler_y)
{
    return [=] (int x, int y) {
        return cmplx{scaler_x(x), scaler_y(y)};
    };
}

```

3. `mandelbrot_iterations` 函数中会增加迭代的次数，为的就是增加计算负载：

```

static auto mandelbrot_iterations(cmplx c)
{
    cmplx z {};
    size_t iterations {0};
    const size_t max_iterations {100000};
    while (abs(z) < 2 && iterations < max_iterations)
    {
        ++iterations;
        z = pow(z, 2) + c;
    }
    return iterations;
}

```

4. 主函数中的部分代码也不需要进行任何修改：

```

int main()
{
    const size_t w {100};
    const size_t h {40};

    auto scale (scaled_cmplx(
        scaler(0, w, -2.0, 1.0),
        scaler(0, h, -1.0, 1.0)
    ));

    auto i_to_xy ([=](int x) {
        return scale(x % w, x / w);
    });
}

```

5. `to_iteration_count` 函数中，不能直接调用 `mandelbrot_iterations(x_to_xy(x))`，需要使用异步函数 `std::async`：

```

auto to_iteration_count ([=](int x) {
    return async(launch::async,
        mandelbrot_iterations,
        i_to_xy(x));
});

```

6. 进行最后的修改之前，函数 `to_iteration_count` 会返回特定坐标需要迭代的次数。那么就会返回一个 `future` 变量，这个变量用于在后面获取异步结果时使用。因此，需要一个 `vector` 来盛放所有 `future` 变量，所以我们就在这里添加了一个。将输出迭代器作为第三个参数传入 `transform` 函数，并在 `vector` 变量 `r` 中放入新的输出：

```

vector<int> v (w * h);
vector<future<size_t>> r (w * h);
iota(begin(v), end(v), 0);
transform(begin(v), end(v), begin(r),
    to_iteration_count);

```

7. `accumulate` 不会在对第二个参数中 `size_t` 的值进行打印，不过这次改成了 `future<size_t>`。我们需要花点时间对这个类型进行适应(对于一些初学者来说，这里使用 `auto&` 类型的话可能会让其产生疑惑)，之后需要调用 `x.get()` 来访问 `x` 中的值，如果 `x` 中的值还没计算出来，程序将会阻塞进行等待：

```

    auto binfunc ([w, n{0}] (auto output_it,
future<size_t> &x)
    mutable {
        *++output_it = (x.get() > 50 ? '*' : ' ');
        if (++n % w == 0) { ++output_it = '\n'; }
        return output_it;
    });

    accumulate(begin(r), end(r),
        ostream_iterator<char>{cout}, binfunc);
}

```

8. 编译并运行程序，我们也能得到和之前一样的输出。唯一不同的就是执行的速度。我们增加了原始版本的迭代次数，程序应该会更慢，不过好在有并行化的帮助，我们能够计算的更快。我的机器上有4个CPU核，并且支持超线程(也就是有8个虚拟核)，我使用GCC和clang得到了不同结果。最好的加速效果有5.3倍，最差也有3.8倍。当然，这个结果和机器的很多状态有关。

How it works...

理解本节代码的关键就在于下面这句和CPU强相关的代码行：

```

transform(begin(v), end(v), begin(r),
to_iteration_count);

```

`vector v` 中包含了所有复数坐标，然后这些坐标会通过曼德尔布罗特算法进行迭代。每次的迭代结果则会保存在 `vector r` 中。

原始代码中，我们将所要绘制的分形图形保存为一维数据。代码则会对之前所有的工作结果进行打印。这也就意味着并行化是提升性能的一个关键因素。

唯一可能并行化的部分就是从 `begin(v)` 到 `end(v)` 的处理，每块都具有相同尺寸，并能够分布在所有核上。这样所有核将会对输入数据进行共享。如果使用并行版本的 `std::transform`，就需要带上一个执行策略。不幸的是，这不是问题的正确解决方式，因为每一个曼德尔布罗特集合中的点，迭代的次数是不同的。

我们的方式是使用一个 `vector` 收集将要获取每个点所要计算的数量的 `future` 变量。代码中 `vector` 能容纳 `w * h` 个元素，例子中就是 `100 * 40`，也就是说 `vector` 实例中存储了4000个 `future` 变量，这些变量都会在异步计算中得到属于自己的值。如果我们的系统有4000个CPU核，就可以启动4000个并发的对坐标进行迭代计算。一个常见的机器上并没有那么多核，CPU只能是异步的对于一个元素进行处理，处理完成后再继续下一个。

`to_iteration_count` 中调用异步版本的 `transform` 时，并不是去计算，而是对线程进行部署，然后立即获得对应的 `future` 对象。原始版本会在每个点上阻塞很久，因为迭代需要花费很长时间。

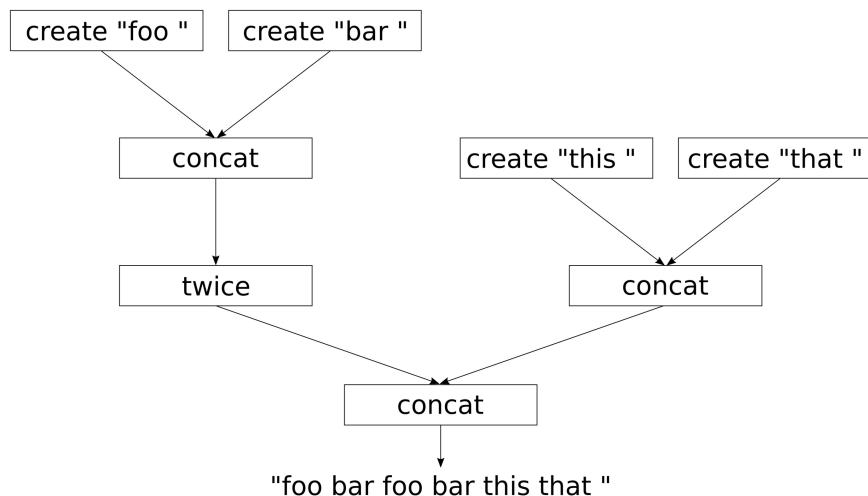
并行版本的程序，也有可能会在那里发生阻塞。打印函数所打印出的结果必须要从 `future` 对象中获取，为了完成这个目的，我们调用 `x.get()` 来获取所有结果。诀窍就在这里：等待第一个值被打印时，其他值也同时在计算。所以，当调用 `get()` 返回时，下一个 `future` 的结果也会很快地被打印出来！

当 `w * h` 是一个非常大的数时，创建 `future` 对象和同步 `future` 对象的开销将会非常可观。本节的例子中，这里的开销并不明显。我的笔记本上有一个i7 4核超线程的CPU(也就是有8个虚拟核)，并行版本与原始版本对比有3-5倍的加速，理想的并行加速应该是8倍。当然，影响机器的因素有很多，并且不同的机器也会有不同的加速比。

实现一个小型自动化并行库—— **std::future**

大多数复杂的任务都能分解为很多子任务。对于所有子任务，我们可以通过画一张有向图无环图来描述哪些子任务键是有依赖的。我们来看一个例子，假设我们想要产出一个字符串 "foo bar foo bar this that "，我们只能通过一个个字符进行产生，然后将这些词汇拼接在一起。为了完成这项工作，我们提供了三个函数 `create`，`concat` 和 `twice`。

考虑到这一点，我们可以通过绘制DAG图来看一下词组间相互的依赖关系：



实现过程中，当一个CPU核上串行的完成这些工作并没有什么问题。通过依赖关系在多个CPU核上执行任务，当所依赖的任务未完成时，只能处于等待状态。

即使使用 `std::async`，这样写出的代码也太无趣了。因为子任务间的依赖关系需要提前建模。本节中，我们将实现两个简单的辅助库，帮助我们将 `create`，`concat` 和 `twice` 函数转换成异步的。这样，我们就能找到一种更为优雅的方式，来设置依赖关系图。执行过程中，代码将会以一种智能的方式进行并行计算，并尽快将整个图完成。

How to do it...

本节中，我们将实现一些函数用来模拟计算敏感型任务，这些任务会互相依赖，我们的任务就是让这些任务尽可能的并行执行：

1. 包含必要的头文件，并声明所使用的命名空间：

```

#include <iostream>
#include <iomanip>
#include <thread>
#include <string>
#include <sstream>
#include <future>

using namespace std;
using namespace chrono_literals;

```

2. 需要对输出进行同步，所以可以使用之前章节中的同步辅助函数来帮助我们：

```

struct pcout : public stringstream {
    static inline mutex cout_mutex;

    ~pcout() {
        lock_guard<mutex> l {cout_mutex};
        cout << rdbuf();
        cout.flush();
    }
};

```

3. 现在，我们对三个字符串转换函数进行实现。第一个函数会通过一个C风格的字符串来创建一个 `std::string` 对象。我们会让这个函数休眠3秒，以模拟计算复杂度：

```

static string create(const char *s)
{
    pcout{} << "3s CREATE " << quoted(s) << '\n';
    this_thread::sleep_for(3s);
    return {s};
}

```

4. 下一个函数需要两个字符串对象作为参数，并且返回拼接后的结果。我们让其休眠5秒：

```

static string concat(const string &a, const string
&b)
{
    pcout{} << "5s CONCAT "
        << quoted(a) << " "
        << quoted(b) << '\n';
    this_thread::sleep_for(5s);
    return a + b;
}

```

5. 最后一个函数接收一个字符串作为参数，并返回自己和自己拼接后的结果。我们让其休眠3秒：

```
static string twice(const string &s)
{
    pcout{} << "3s TWICE " << quoted(s) << '\n';
    this_thread::sleep_for(3s);
    return s + s;
}
```

6. 对于串行任务来说，这就已经准备好了，但是我们想使用并行的方式来完成。所以，我们还需要实现一些辅助函数。这里需要注意了，下面三个函数看起来有些复杂。`asynchronize` 能接收一个函数 `f`，并返回一个其捕获到的可调用对象。我们可以传入任意数量的参数到这个可调用的对象中，然后其会将这些参数连同 `f` 捕获到另一个可调用对象中，并且将这个可调用对象返回给我们。最后一个可调用对象不需要任何参数。之后，其会将参数传入 `f` 中，并异步的执行函数 `f`：

```
template <typename F>
static auto synchronize(F f)
{
    return [f] (auto ... xs) {
        return [=] () {
            return async(launch::async, f, xs...);
        };
    };
}
```

7. 接下来这个函数，将会使用下一步(也就是第8步)中我们声明的函数。其能接受一个函数 `f`，并且将该函数捕获到一个可调用的对象中并返回。该对象可以被多个 `future` 对象所调用。然后，对 `future` 对象使用 `.get()`，来获取 `f` 中的结果：

```
template <typename F>
static auto fut_unwrap(F f)
{
    return [f] (auto ... xs) {
        return f(xs.get()...);
    };
}
```

8. 最后一个辅助函数能够接受一个函数 `f`。其会返回一个持有 `f` 函数的可调用对象。这个可调用对象可以传入任意个参数，并且会将函数 `f` 与这些参数让另一个可调用对象获取。最后，返回给我们的可调用对象无需任何参数。然后，就可以调用 `xs...` 包中获取到所有可调用对象。这些对象会返回很多 `future`，

这些 `future` 对象需要使用 `fut_unwrap` 进行展开。`future` 展开，并会通过 `std::async` 对实际函数 `f` 进行执行，在通过 `future` 返回函数 `f` 执行的结果：

```
template <typename F>
static auto async_adapter(F f)
{
    return [f] (auto ... xs) {
        return [=] () {
            return async(launch::async,
                         fut_unwrap(f), xs() ...);
        };
    };
}
```

9. OK，完成以上工作的感觉就是“疯狂”，这种表达式的嵌套让我想起了电影《盗梦空间》的场景(上一步的代码中，`Lambda`表达式会继续返回一个`Lambda`表达式)。这段带有魔法的代码，我们会在后面来详细的了解。现在，让我们异步的使用 `create`，`concat` 和 `twice` 函数。`async_adapter` 是一个非常简单的函数，其会等待 `future` 参数，并返回一个 `future` 的结果，其会将同步世界转换成异步世界。我们对 `concat` 和 `twice` 使用这个函数。我们必须对 `create` 使用 `asynchronize`，因为其会返回一个 `future`，不过我们会使用 `future` 对象获取到的值，而非 `future` 对象本身。任务的依赖链，需要从 `create` 开始：

```
int main()
{
    auto pcreate (asynchronize(create));
    auto pconcat (async_adapter(concat));
    auto ptwice (async_adapter(twice));
```

10. 现在我们有了可以自动并行化的函数，其与同步代码的函数名相同，不过添加了前缀 `p`。现在，让我们来设置一些比较复杂依赖关系树。首先，我们创建两个字符串 `"foo"` 和 `"bar"`，然后进行拼接，返回 `"foo bar"`。在 `twice` 中，字符串将会和自身进行拼接。然后，创建了字符串 `"this"` 和 `"that"`，拼接得到 `"this that"`。最后，我们拼接的结果为 `"foo bar foo bar this that"`，结果将会保存在变量 `callable` 中。最后，调用 `callable().get()` 进行计算，并等待返回值，然后将返回值进行打印。我们没有调用 `callable()` 时，计算不会开始，在我们对其进行调用后，就是见证奇迹的时刻：

```

auto result (
    pconcat(
        ptwice(
            pconcat(
                pcreate("foo "),
                pcreate("bar "))),
        pconcat(
            pcreate("this "),
            pcreate("that ")));
    cout << "Setup done. Nothing executed yet.\n";

    cout << result().get() << '\n';
}

```

11. 编译并运行程序，我们就会看到 `create` 每一次调用所产生的字符串，然后其他函数也开始执行。这个过程好像是通过智能调度来完成的，整个程序使用**16秒**完成。如果使用串行的方式，将会使用**30s**完成。需要注意的是，我们使用**4核**的机器来运行程序，也就是有**4次** `create` 调用在同时进行。如果机器没有太多和CPU，那么运行时间会更长：

```

$ ./chains
Setup done. Nothing executed yet.
3s CREATE "foo "
3s CREATE "bar "
3s CREATE "this "
3s CREATE "that "
5s CONCAT "this " "that "
5s CONCAT "foo " "bar "
3s TWICE"foo bar "
5s CONCAT "foo bar foo bar " "this that "
foo bar foo bar this that

```

How it works...

本节例子的串行版本，可能看起来如下：

```

int main()
{
    string result {
        concat(
            twice(
                concat(
                    create("foo "),
                    create("bar ")),
                concat(
                    create("this "),
                    create("that "))) );
    };

    cout << result << '\n';
}

```

本节中，我们完成了一些辅助函数，`async_adapter` 和 `asynchronize`，其能帮助我们对 `create`，`concat` 和 `twice` 函数进行包装。然后调用其异步版本 `pcreate`，`pconcat` 和 `ptwice`。

先不看这两个函数复杂的实现，我们先来看一下我们获得了什么。

串行版本的代码可能类似如下写法：

```

string result {concat( ... )};
cout << result << '\n';

```

并行版本的写法：

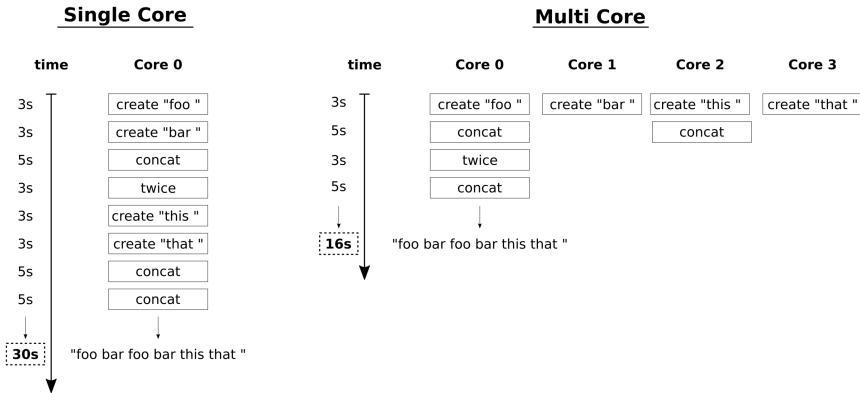
```

auto result (pconcat( ... ));
cout << result().get() << '\n';

```

好了！现在就是最复杂的环节了。并行最后的结果并不是 `string`，而是一个能够返回一个 `future<string>` 实例的可调用对象，我们可以对返回值调用 `get()` 得到函数运算后的值。这看起来可能很疯狂。

所以，我们为什么要返回 `future` 对象呢？问题在于我们的 `create`，`concat` 和 `twice` 函数运行起来都非常慢。不过，我们通过依赖关系树可以看到，数据流还是有可以独立的部分，也就是可并行的部分。让我们来看一下下面两个例子的流水：



左侧边是单核的流水。所有函数一个接一个的在CPU上进行。这样时间累加起来就是30秒。

右侧边是多核的流水。函数会通过依赖关系并行的运行。在有4个核的机器上，我们将同时创建4个子字符串，然后对其进行拼接，等等的操作。并行版本需要16秒就能完成任务。如果我们没法让函数本身变的更快，则我们无法再进行加速。4个CPU的情况下，我们能有如此的加速，其实我们可以以更好的方式进行调度。

应该怎么做？

我们通常会写成如下的模式：

```
auto a (async(launch::async, create, "foo"));
auto b (async(launch::async, create, "bar"));
auto c (async(launch::async, create, "this"));
auto d (async(launch::async, create, "that"));
auto e (async(launch::async, concat, a.get(), b.get()));
auto f (async(launch::async, concat, c.get(), d.get()));
auto g (async(launch::async, twice, e.get()));
auto h (async(launch::async, concat, g.get(), f.get()));
```

`a`, `b`, `c` 和 `d` 都可以作为一个不错的开始，因为会创建对应的子字符串，并且会在后台同时进行创建。不幸的是，这段代码将会在初始化 `e` 的时候被阻塞。为了拼接 `a` 和 `b`，我们需要调用 `get()` 函数来获取这两个值，函数会对程序进行阻塞，直到获得相应的值为止。这明显不是一个好方法，因为并行代码会在第一个 `get()` 调用时阻塞。我们需要更好的策略来解决这个问题。

OK，现在让我们来看看我们在例子中完成的比较复杂的辅助函数。第一个就是 `asynchronize`：

```

template <typename F>
static auto synchronize(F f)
{
    return [f](auto ... xs) {
        return [=] () {
            return async(launch::async, f, xs...);
        };
    };
}

```

当我们有一个函数 `int f(int, int)` 时，我们可以进行如下的操作：

```

auto f2 = synchronize(f);
auto f3 = f2(1, 2);
auto f4 = f3();
int result = f4.get();

```

`f2` 就是异步版本的 `f`。其调用方式与 `f` 完全相同。之后，其会返回可调用对象，并保存在 `f3` 中。现在 `f3` 得到了 `f` 和参数 `1` 和 `2`，不过函数还没运行，只是捕获过程。

我们调用 `f3()` 时，最后就会得到一个 `future` 实例，因为 `f3` 中的返回值是 `async(launch::async, f, 1, 2);` 的返回值。某种意义上来说 `f3` 表示为集获取函数和函数参数，与抛出 `std::async` 返回值与一身的变量。

内部Lambda表达式只通过捕获进行获取，但不接受任何输入参数。因此，可以让任务并行的方式分发，而不会遭遇任何方式的阻塞。我们对同样复杂的 `async_adapter` 函数采取同样的策略：

```

template <typename F>
static auto async_adapter(F f)
{
    return [f](auto ... xs) {
        return [=] () {
            return async(launch::async, fut_unwrap(f),
            xs(...));
        };
    };
}

```

函数能够返回一个函数 `f` 的模拟函数，因为其能接受相同的参数。然后，函数会返回一个可调用对象，并且也不接受任何参数，这里返回的可调用对象与其他辅助函数所返回的有所不同。

`async(launch::async, fut_unwrap(f), xs(...));` 是什么意思呢？其中 `xs(...)` 部分意味着，所有参数都保存在 `xs` 包中，供可调用对象使用，并且返回的可调用对象都不需要参数。那些可调用对象通过自身的方式生产 `future` 变量，通过对 `future` 变量调用 `get()` 获得实际返回值。这也就是 `fut_unwrap` 所要完成的事情：

```
template <typename F>
static auto fut_unwrap(F f)
{
    return [f](auto ... xs) {
        return f(xs.get()...);
    };
}
```

`fut_unwrap` 会将函数 `f` 转换为一个可调用对象，其能接受一组参数。函数对象执行之后可以对所有的 `future` 对象调用 `.get()`，从而获得 `f` 函数实际的执行结果。

我们花点时间来消化一下上面的内容。当主函数中调用这些函数，使用 `auto result(pconcat(...))` 的方式创建调用链，将所有子字符串最后拼接成一个长字符串。这时对 `async` 的调用还未完成。然后，当调用 `result()` 时，我们则获得 `async` 的返回值，并对其返回值调用 `.get()`，这就能保证任何线程不会发生阻塞。实际上，在 `async` 调用前，不会有 `get()` 的调用。

最后，我们可以对 `result()` 的返回值调用 `.get()`，从而获取最终的结果字符串。

第10章 文件系统

如果没有第三方库来帮助我们处理系统路径，那么对文件系统的编程就会非常冗余，因为我们需要处理很多的条件。

有些路径是绝对路径，而有些是相对路径，有时候路径还会互相包含。`.` 表示当前目录，`..` 表示上级目录。然后，不同系统用来分隔目录的斜杠也不同，Linux, MacOS和各种UNIX变体操作系统上使用的是`/`，而Windows下使用的是`\`，这样就会导致文件的不同。

因为有时程序难免要实现和文件系统相关功能，所以C++17的STL中添加了对文件系统友好的库。其好的一点就在于可移植性，所以在一个系统中写好后，就可以在不同的系统间运行。

本章中，我们会了解到 `path` 类是如何工作的，因为其是库中最重要的角色。而后，我们将会了解到，强大但简单的 `directory_iterator` 和 `recursive_directory_iterator` 类，其会对文件操作很有帮助。最后，我们将通过一些小并简单的式例工具来完成一些与文件系统相关的任务。对简单的工具有所了解后，大家就可以构建更加复杂的工具了。

实现标准化路径

本节中我们通过一个非常简单的例子来了解 `std::filesystem::path` 类，并实现一个智能标准化系统路径的辅助函数。

本节中的例子可以在任意的文件系统中使用，并且返回一种标准化格式的路径。标准化就意味着获取的是绝对路径，路径中不包括`.`和`..`。

实现函数的时候，我们将会了解，当使用文件系统库的基础部分时，需要注意哪些细节。

How to do it...

本节中，我们的程序可以从命令行参数中获得文件系统路径，并使用标准化格式进行打印：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <filesystem>

using namespace std;
using namespace filesystem;
```

2. 主函数中，会对命令行传入的参数进行检查。如果没有传入，我们将会直接返回，并在终端上打印程序具体的使用方式。当提供了一个路径，那我们将用其对 `filesystem::path` 对象进行实例化：

```
int main(int argc, char *argv[])
{
    if (argc != 2) {
        cout << "Usage: " << argv[0] << " <path>\n";
        return 1;
    }

    const path dir {argv[1]};
```

3. 实例化 `path` 对象之后，还不能确定这个路径是否真实存在于计算机的文件系统中。这里我们使用了 `filesystem::exists` 来确认路径。如果路径不存在，我们会再次返回：

```
if (!exists(dir)) {
    cout << "Path " << dir << " does not
exist.\n";
    return 1;
}
```

4. Okay, 如果完成了这个检查, 我们就能确定这是一个正确的路径, 并且将会对这个路径进行标准化, 然后将其进行打印。`filesystem::canonical` 将会为我们返回另一个 `path` 对象, 可以直接对其进行打印, 不过 `path` 的 `<<` 重载版本会将双引号进行打印。为了去掉双引号, 我们通过 `.c_str()` 或 `.string()` 方法对路径进行打印:

```
    cout << canonical(dir).c_str() << '\n';
}
```

5. 编译代码并运行。当我们在家目录下输入相对地址 “`src`” 时, 程序将会打印出其绝对路径:

```
$ ./normalizer src
/Users/tfc/src
```

6. 当我们打印一些更复杂的路径时, 比如: 给定路径中包含桌面文件夹的路径, `..`, 还会有 `Documents` 文件夹, 然后在到 `src` 文件夹。然而, 程序会打印出与上次相同的地址!

```
$ ./normalizer Desktop/../Documents/../src
/Users/tfc/src
```

How it works...

作为一个 `std::filesystem` 的新手, 看本节的代码应该也没有什么问题。通过文件系统路径字符串初始化了一个 `path` 对象。`std::filesystem::path` 类为文件系统库的核心, 因为库中大多数函数和类与之相关。

`filesystem::exists` 函数可以用来检查给定的地址是否存在。检查文件路径的原因是, `path` 对象中的地址, 不确定在文件系统中是否存在。`exists` 能够接受一个 `path` 实例, 如果地址存在, 则返回 `true`。`exists` 无论是相对地址和绝对地址都能够进行判断。

最后, 我们使用了 `filesystem::canonical` 将给定路径进行标准化。

```
path canonical(const path& p, const path& base =
current_path());
```

`canonical` 函数能接受一个 `path` 对象和一个可选的第二参数, 也就是另一个地址。如果 `p` 路径是一个相对路径, 那么 `base` 就是其基础路径。完成这些后, `canonical` 会将 `.` 和 `..` 从路径中移除。

打印时对标准化地址使用了 `.c_str()` 函数, 这样我们打印出来的地址前后就没有双引号了。

There's more...

`canonical` 在对应地址不存在时，会抛出一个 `filesystem_error` 类型的异常。为了避免函数抛出异常，我们需要使用 `exists` 函数对提供路径进行检查。这样的检查仅仅就是为了避免函数抛出异常吗？肯定不是。

`exists` 和 `canonical` 函数都能抛出 `bad_alloc` 异常。如果我们遇到了，那程序肯定会失败。更为重要的是，当我们对路径进行标准化处理时，其他人将对应的文件重命名或删除了，则会造成更严重的问题。这样的话，即便是之前进行过检查，`canonical` 还是会抛出一个 `filesystem_error` 异常。

大多数系统函数都会有一些重载，他们能够接受相同的参数，甚至是一个 `std::error_code` 引用：

```
path canonical(const path& p, const path& base =
current_path());
path canonical(const path& p, error_code& ec);
path canonical(const std::filesystem::path& p,
               const std::filesystem::path& base,
               std::error_code& ec );
```

我们可以使用 `try-catch` 将系统函数进行包围，手动的对其进行处理。需要注意的是，这里只会改变系统相关错误的动作，而无法对其他进行修改。带或不带 `ec` 参数，更加基于异常，例如当系统没有可分配内存时，还是会抛出 `bad_alloc` 异常。

使用相对路径获取规范的文件路径

上一节中，我们对路径进行了标准化输出。使用了 `filesystem::path` 类，并且了解了如何获取路径，并进行检查，以及其他一些原理性的东西。也能帮助我们将字符串组成路径，从而对路径进行再次解析。

`path` 已经将操作系统的一些细节为我们进行了封装，不过我们还是需要了解一些细节。

本节我们也将了解到，如何将绝对路径和相对路径进行合并和分解。

How to do it...

本节中，我们将围绕着相对路径和绝对路径进行，从而了解 `path` 类的有时，以及其对应的辅助函数。

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <filesystem>

using namespace std;
using namespace filesystem;
```

2. 然后，我们实例化一个 `path` 对象。不过这次，路径中的文件是否存在就没有那么重要了。这里有些函数，在文件不存在的时候会抛出异常。

```
int main()
{
    path p {"testdir/foobar.txt"};
```

3. 现在我们来了解一下不同的文件系统库函数。`current_path` 将返回我们执行程序的路径，也就是工作目录。`absolute` 能接受一个相对地址，就像我们定义的 `p` 一样。`system_complete` 在 Linux, MacOS 和类 UNIX 操作系统上与 `absolute` 的功能相同。在 Windows 下我们将获取一个带有盘符(比如 `c:`)的绝对地址。`canonical` 与 `absolute` 的功能相同，不过其删除了所有的 `.` 和 `..`。我们可以使用如下的方式使用这些函数：

```
cout << "current_path : " << current_path()
    << "\nabsolute_path : " << absolute(p)
    << "\nsystem_complete : " <<
system_complete(p)
    << "\ncanonical(p) : " << canonical(p)
    << '\n';
```

4. `path` 另一个优势在于，其对 / 操作符进行了重载。通过这种方式我们可以连接文件夹和文件。让我们组合一个，然后进行打印：

```
cout << path{"testdir"} / "foobar.txt" << '\n';
```

5. 我们将 `canonical` 与合并的路径一起使用。通过给定 `canonical` 一个相对地址，比如 "foobar.txt"，和一个合并的绝对地址 `current_path() / "testdir"`，其将会返回给我们一个绝对地址。在另一个调用中，我们给定我么的路径为 `p` (假设为 "testdir/foobar.txt")，并且通过 `current_path()` 获取当前位置的绝对路径，我们这里就使用 "testdir" 好了。其结果与 `current_path()` 相同，因为间接获得了绝对地址。在这两次调用中，`canonical` 将会返回给我们相同的绝对地址：

```
cout << "canonical testdir : "
     << canonical("foobar.txt",
                  current_path() / "testdir")
     << "\ncanonical testdir 2 : "
     << canonical(p, current_path() / "testdir/..")
     << '\n';
```

6. 我们也可以对两个非标准化的路径进行比较。`equivalence` 能接受两个路径，并在内部将两个路径进行标准化，如果这两个路径相同，就会返回 `true`，否则会返回 `false`。这个例子中，相应的路径必须存在，否则就会抛出一个异常：

```
cout << "equivalence: "
     << equivalent("testdir/foobar.txt",
                   "testdir/../testdir/foobar.txt")
     << '\n';
}
```

7. 编译并运行代码，将会得到如下的输出。`current_path()` 会返回我笔记本上的 `HOME` 目录，因为我在这个路径下执行的程序。相对路径 `p` 会通过 `absolute_path`，`system_complete` 和 `canonical` 预先进行准备。我们能看到 `absolute_path` 和 `system_complete` 的结果都一样，因为我使用的是 `Mac` 系统。在使用 `Windows` 操作系统的机器上，`system_complete` 将会前置一个 `c:`，或者是工作路径的磁盘盘符：

```
$ ./canonical_filepath
current_path: "/Users/tfc"
absolute_path : "/Users/tfc/testdir/foobar.txt"
system_complete : "/Users/tfc/testdir/foobar.txt"
canonical(p): "/Users/tfc/testdir/foobar.txt"
"testdir/foobar.txt"
canonical testdir : "/Users/tfc/testdir/foobar.txt"
canonical testdir 2 : "/Users/tfc/testdir/foobar.txt"
equivalence: 1
```

8. 这个简单的程序中，就不对异常进行处理了。当从 `testdir` 文件夹中将 `foobar.txt` 文件删除时，程序将因为抛出异常的原因而终止。`canonical` 函数需要路径真实存在。还有一个 `weakly_canonical`，但其不符合我们的要求。

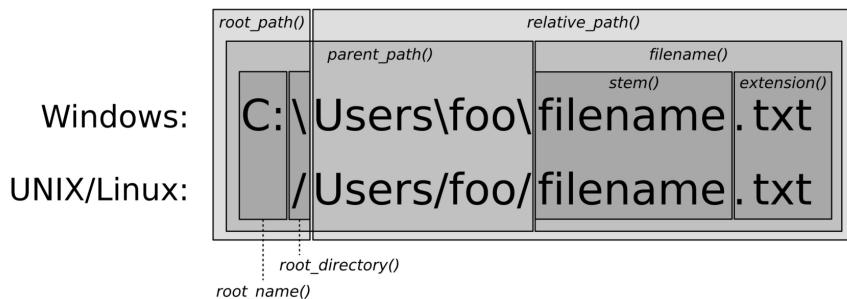
```
$ ./canonial_filepath
current_path: "/Users/tfc"
absolute_path : "/Users/tfc/testdir/foobar.txt"
system_complete : "/Users/tfc/testdir/foobar.txt"
terminate called after throwing an instance of
'std::filesystem::v1::__cxx11::filesystem_error'
what(): filesystem error: cannot canonicalize:
No such file or directory [testdir/foobar.txt]
[/Users/tfc]
```

How it works...

本节的目的就是如何快速的组成新的路径。其主要还有通过 `path` 类重载的 / 操作符来完成。另外，文件系统函数的相对路径和绝对路径是一致的，并且路径中包含 . 和 ..。

很多函数会返回一个转换或未转换的 `path` 实例。我们不会将所有函数都列在这里，如果想要了解它们，去看下C++手册是个不错的选择。

`path` 类中有很多的成员函数，很值得一看。让我们来了解一下，对于一个路径来说，成员函数返回的是哪一部分。下面的图就为我们描述了在Windows和UNIX/Linux下，对应函数所返回的路径：



这样我们就能很容易的了解到，`path` 的那个函数返回的是绝对地址。相对地址中，`root_path`，`root_name` 和 `root_directory` 部分都空的。`relative_path` 将会返回一个相对地址。

列出目录下的所有文件

每个操作系统都会提供一些工具，以列出目录下的所有文件。Linux, MacOS和类UNIX的操作系统中，`ls` 就是一个最简单的例子。Windows和Dos系统下，命令为`dir`。其会提供一些文件的补充信息，比如文件大小，访问权限等。

可以通过对文件夹的递归和文件遍历来对这样的工具进行实现。所以，让我们来试一下吧！

我们的`ls/dir`命令会将目录下的文件名，元素索引，以及一些访问权限标识，以及对应文件的文件大小，分别进行展示。

How to do it...

本节中，我们将实现一个很小的工具，为使用者列出对应文件夹下的所有文件。会将文件名，文件类型，大小和访问权限分别列出来。

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <sstream>
#include <iomanip>
#include <numeric>
#include <algorithm>
#include <vector>
#include <filesystem>

using namespace std;
using namespace filesystem;
```

2. `file_info` 是我们要实现的一个辅助函数。其能接受一个`directory_entry`对象的引用，并从这个路径中提取相应的信息，实例化`file_status`对象(使用`status`函数)，其会包含文件类型和权限信息。最后，如果是一个常规文件，则会提取其文件大小。对于文件夹或一些特殊的文件，我们将返回大小设置为0。所有的信息都将会封装到一个元组中：

```
static tuple<path, file_status, size_t>
file_info(const directory_entry &entry)
{
    const auto fs (status(entry));
    return {entry.path(),
            fs,
            is_regular_file(fs) ?
            file_size(entry.path()) : 0u};
}
```

3. 另一个辅助函数就是 `type_char`。路径不能仅表示目录和简单文本/二进制文件。操作系统提供了多种抽象类型，比如字符/块形式的硬件设备接口。**STL**库也提供了为此提供了很多为此函数。我们通过返回 '`d`' 表示文件夹，通过返回 '`f`' 表示普通文件等。

```
static char type_char(file_status fs)
{
    if (is_directory(fs)) { return 'd'; }
    else if (is_symlink(fs)) { return 'l'; }
    else if (is_character_file(fs)) { return 'c'; }
    else if (is_block_file(fs)) { return 'b'; }
    else if (is_fifo(fs)) { return 'p'; }
    else if (is_socket(fs)) { return 's'; }
    else if (is_other(fs)) { return 'o'; }
    else if (is_regular_file(fs)) { return 'f'; }

    return '?';
}
```

4. 下一个辅助函数为 `rwx`。其能接受一个 `perms` 变量(其为文件系统库的一个 `enum` 类)，并且会返回一个字符串，比如 `rwxrwxrwx`，用来表示文件的权限设置。“`rwx`”分别为 **read**, **write** 和 **execution**，分别代表了文件的权限属性。每三个字符表示一个组，也就代表对应的组或成员，能对文件进行的操作。`rwxrwxrwx` 则代表着每个人多能对这个文件进行访问和修改。`rw-r--r--` 代表着所有者可以对文件进行读取和修改，不过其他人只能对其进行读取操作。我们将这些 `读取/修改/执行` 所代表的字母进行组合，就能形成文件的访问权限列表。**Lambda**表达式可以帮助我们完成重复性的检查工作，检查 `perms` 变量 `p` 中是否包含特定的掩码位，然后返回 ‘-’ 或正确的字符。

```
static string rwx(perms p)
{
    auto check ([p] (perms bit, char c) {
        return (p & bit) == perms::none ? '-' : c;
    });

    return {check(perms::owner_read, 'r'),
            check(perms::owner_write, 'w'),
            check(perms::owner_exec, 'x'),
            check(perms::group_read, 'r'),
            check(perms::group_write, 'w'),
            check(perms::group_exec, 'x'),
            check(perms::others_read, 'r'),
            check(perms::others_write, 'w'),
            check(perms::others_exec, 'x')};
}
```

5. 最后一个辅助函数能接受一个整型的文件大小，并将其转换为跟容易读懂的模式。将其大小除以表示的对应边界，然后使用**K**, **M**或**G**来表示这个文件的大小：

```

static string size_string(size_t size)
{
    stringstream ss;
    if (size >= 10000000000) {
        ss << (size / 10000000000) << 'G';
    } else if (size >= 1000000) {
        ss << (size / 1000000) << 'M';
    } else if (size >= 1000) {
        ss << (size / 1000) << 'K';
    } else { ss << size << 'B'; }

    return ss.str();
}

```

6. 现在来实现主函数。我们会对用户在命令行输入的路径进行检查。如果没有传入，则默认为当前路径。然后，再来检查文件夹是否存在。如果不存在，就不会列出任何文件：

```

int main(int argc, char *argv[])
{
    path dir {argc > 1 ? argv[1] : "."};

    if (!exists(dir)) {
        cout << "Path " << dir << " does not
exist.\n";
        return 1;
    }
}

```

7. 现在，将使用文件信息元组来填充一个 `vector`。实例化一个 `directory_iterator`，并且将其传入 `path` 对象的构造函数中。并通过目录迭代器对文件进行迭代，我们将 `directory_entry` 对象转换成文件信息元组，然后将其插入相应的 `vector`。

```

vector<tuple<path, file_status, size_t>> items;

transform(directory_iterator{dir}, {},
         back_inserter(items), file_info);

```

8. 现在，将所有文件的信息都存在于 `vector` 之中，并且使用辅助函数将其进行打印：

```

    for (const auto &[path, status, size] : items) {
        cout << type_char(status)
            << rwx(status.permissions()) << " "
            << setw(4) << right << size_string(size)
            << " " << path.filename().c_str()
            << '\n';
    }
}

```

9. 编译并运行程序，并通过命令行传入C++文档文件所在的地址。我们能了解到对应文件夹所包含的文件，因为文件夹下只有‘d’和‘f’作为输出的表示。这些文件具有不同的权限，并且都有不同的大小。需要注意的是，这些文件的显示顺序，是按照名字在字母表中的顺序排序，不过我们不依赖这个顺序，因为C++17标准不需要字母表排序：

```

$ ./list ~/Documents/cpp_reference/en/cpp
drwxrwxr-x 0B      algorithm
frw-r--- 88K     algorithm.html
drwxrwxr-x 0B      atomic
frw-r--- 35K     atomic.html
drwxrwxr-x 0B      chrono
frw-r--- 34K     chrono.html
frw-r--- 21K     comment.html
frw-r--- 21K     comments.html
frw-r--- 220K    compiler_support.html
drwxrwxr-x 0B      concept
frw-r--- 67K     concept.html
drwxr-xr-x 0B      container
frw-r--- 285K    container.html
drwxrwxr-x 0B      error
frw-r--- 52K     error.html

```

How it works...

本节中，我们迭代了文件夹中的所有文件，并且对每个文件的状态和大小进行检查。对于每个文件的操作都非常直接和简单，我们对文件夹的遍历看起来也很魔幻。

为了对我们的文件夹进行遍历，只是对 `directory_iterator` 进行实例化，然后对该对象进行遍历。使用文件系统库来遍历一个文件夹是非常简单的。

```

for (const directory_entry &e : directory_iterator{dir})
{
    // do something
}

```

除了以下几点，`directory_iterator` 也没有什么特别的：

- 会对文件夹中的每个文件访问一次
- 文件中元素的遍历顺序未指定
- 文件节元素中 . 和 .. 都已经被过滤掉

不过，`directory_iterator` 看起来是一个迭代器，并且同时具有一个可迭代的范围。为什么需要注意这个呢？对于简单的 `for` 循环来说，其需要一个可迭代的范围。本节例程中，我们会将其当做一个迭代器使用：

```
transform(directory_iterator{dir}, {},  
        back_inserter(items), file_info);
```

实际上，就是一个迭代器类型，只不过这个类将 `std::begin` 和 `std::end` 函数进行了重载。当调用 `begin` 和 `end` 时，其会返回相应的迭代器。虽说第一眼看起来比较奇怪，但是让这个类型的确更加有用。

实现一个类似grep的文本搜索工具

大多数操作系统都会提供本地的搜索引擎。用户可以使用一些快捷键，对本地文件进行查找。

这种功能出现之前，命令行用户会通过 `grep` 或 `awk` 工具对文件进行查找。用户可以简单的输入 `grep -r foobar .`，然后工具将会基于当前目录，进行递归的查找，并显示包含有 "foobar" 名字的文件。

本节中，将实现这样一种应用。我们的 `grep` 使用命令行方式使用，并基于给定文件夹递归的对文件进行查找。然后，将找到的文件名打印出来。我们将使用线性的模式匹配方式，将匹配文件中的对应行号进行打印。

How to do it...

我们将实现小工具，用于查找与用户提供的文本段匹配的文件。这工具与UNIX中的 `grep` 工具类似，不过为了简单起见，其功能没有那么强大：

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <fstream>
#include <regex>
#include <vector>
#include <string>
#include <filesystem>

using namespace std;
using namespace filesystem;
```

2. 先来实现一个辅助函数，这个函数能接受一个文件地址和一个正则表达式对象，正则表达式对象用来描述我们要查找的文本段。然后，实例化一个 `vector`，用于保存匹配的文件行和其对应的内容。然后，实例化一个输入文件流对象，读取文件，并进行逐行的文本匹配。

```
static vector<pair<size_t, string>>
matches(const path &p, const regex &re)
{
    vector<pair<size_t, string>> d;
    ifstream is {p.c_str()};
```

3. 通过 `getline` 函数对文件进行逐行读取，当字符串中包含有我们提供文本段，则 `regex_search` 返回`true`，如果匹配会将字符串和对应的行号保存在 `vector` 中。最后，我们将返回所有匹配的结果：

```

        string s;
        for (size_t line {1}; getline(is, s); ++line) {
            if (regex_search(begin(s), end(s), re)) {
                d.emplace_back(line, move(s));
            }
        }

        return d;
    }
}

```

4. 主函数会先对用户提供的文本段进行检查，如果这个文本段不能用，则返回错误信息：

```

int main(int argc, char *argv[])
{
    if (argc != 2) {
        cout << "Usage: " << argv[0] << "
<pattern>\n";
        return 1;
    }
}

```

5. 接下来，会通过输入文本创建一个正则表达式对象。如果表达式是一个非法的正则表达式，这将会导致一个异常抛出。如果触发了异常，我们将对异常进行捕获并处理：

```

regex pattern;

try { pattern = regex{argv[1]}; }
catch (const regex_error &e) {
    cout << "Invalid regular expression
provided.\n";
    return 1;
}

```

6. 现在，可以对文件系统进行迭代，然后对我们提供的文本段进行匹配。使用 `recursive_directory_iterator` 对工作目录下的所有文件进行迭代。原理和之前章节的 `directory_iterator` 类似，不过会对子目录进行递归迭代。对于每个匹配的文件，我们都会调用辅助函数 `matches`：

```

for (const auto &entry :
    recursive_directory_iterator{current_path()}) {
    auto ms (matches(entry.path(), pattern));
}

```

7. 如果有匹配的结果，我们将会对文件地址，对应文本行数和匹配行的内容进行打印：

```
        for (const auto &[number, content] : ms) {
            cout << entry.path().c_str() << ":" << number
            << " - " << content << '\n';
        }
    }
}
```

8. 现在，准备一个文件 `foobar.txt`，其中包含一些测试行：

```
foo
bar
baz
```

9. 编译并运行程序，就会得到如下输出。我们在 `/Users/tfc/testdir` 文件夹下运行这个程序，我们先来对 `bar` 进行查找。在这个文件夹下，其会在 `foobar.txt` 的第二行和 `testdir/dir1` 文件夹下的另外一个文件 `text1.txt` 中匹配到：

```
$ ./grepper bar
/Users/tfc/testdir/dir1/text1.txt:1 - foo bar bla
blubb
/Users/tfc/testdir/foobar.txt:2 - bar
```

10. 再次运行程序，这次我们对 `baz` 进行查找，其会在第三行找到对应内容：

```
$ ./grepper baz
/Users/tfc/testdir/foobar.txt:3 - baz
```

How it works...

本节的主要任务是使用正则表达式对文件的内容进行查找。不过，让我们关注一下 `recursive_directory_iterator`，因为我们会使用这个迭代器来进行本节的子文件夹的递归迭代。

与 `directory_iterator` 和 `recursive_directory_iterator` 迭代类似，其可以用来对子文件夹进行递归，就如其名字一样。当进入文件系统中的一个文件夹时，将会产生一个 `directory_entry` 实例。当递归到子文件夹时，也会产生对应的 `directory_entry` 实例。

`recursive_directory_iterator` 具有一些有趣的成员函数：

- `depth()` 代表我们需要迭代多少层子文件夹。
- `recursion_pending()` 代表在进行当前迭代器后，是否会在进行对子文件夹进行迭代。
- `disable_recursion_pending()` 当迭代器需要再对子文件夹进行迭代时，提前调用这个函数，则会让递归停止。
- `pop()` 将会终止当前级别的迭代，并返回上一级目录。

There's more...

我们需要了解的另一个就是 `directory_options` 枚举类。`recursive_directory_iterator` 能将 `directory_options` 的实例作为其构造函数的第二个参数，通常将 `directory_options::none` 作为默认值传入。其他值为：

- `follow_directory_symlink` 能允许对符号链接的文件夹进行递归迭代。
- `skip_permission_denied` 这会告诉迭代器，是否跳过由于权限错误而无法访问的目录。

这两个选项可以通过 | 进行组合。

实现一个自动文件重命名器

本节的动机是因为我自己经常需要使用到这样的功能。我们将假日中的照片汇总在一起时，不同朋友的照片和视频都在一个文件夹中，并且每个文件的后缀看起来都不一样。一些JPEG文件有 `.jpg` 的扩展，而另一些为 `.jpeg`，还有一些则为 `.JPEG`。

一些人会让文件具有统一的扩展，其会使用一些有用的命令对于所有文件进行重命名。同时，我们会将使用下划线来替代空格。

本节中，我们将试下一个类似的工具，叫做 `renamer`。其能接受一些列输入文本段，作为其替代，类似如下的方式：

```
$ renamer jpeg jpg JPEG jpg
```

本节中，重命名器将会对当前目录进行递归，然后找到文件后缀为 `jpeg` 和 `JPEG` 的所有文件，并将这些文件的后缀统一为 `jpg`。

How to do it...

我们将实现一个工具，通过对文件夹的递归对于所有文件名匹配的文件进行重命名。所有匹配到的文件，都会使用用户提供的文本段进行替换。

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <regex>
#include <vector>
#include <filesystem>

using namespace std;
using namespace filesystem;
```

2. 我们将实现一个简单的辅助函数，其能接受一个使用字符串表示的输入文件地址和一组替换对。每一个替换对都有一个文本段和其要替换文本段。对替换范围进行循环时，我们使用了 `regex_replace` 用于对输入字符串进行匹配，然后返回转换后的字符串。之后，我们将返回结果字符串。

```

template <typename T>
static string replace(string s, const T
&replacements)
{
    for (const auto &[pattern, repl] : replacements)
    {
        s = regex_replace(s, pattern, repl);
    }

    return s;
}

```

3. 主函数中，我们首先对命令行的正确性进行检查。可以成对的接受命令行参数，因为我们想要匹配段和替换段相对应。`argc` 的第一个元素为执行文件的名字。当用户提供了成对的匹配段和替换段时，`argc` 肯定是大于3的奇数：

```

int main(int argc, char *argv[])
{
    if (argc < 3 || argc % 2 != 1) {
        cout << "Usage: " << argv[0]
             << " <pattern> <replacement> ...\\n";
        return 1;
    }
}

```

4. 我们对输入对进行检查时，会将对应的 `vector` 进行填充：

```

vector<pair<regex, string>> patterns;

for (int i {1}; i < argc; i += 2) {
    patterns.emplace_back(argv[i], argv[i + 1]);
}

```

5. 现在，可以对整个文件系统进行遍历。简单起见，将当前目录作为遍历的默认起始地址。对于每一个文件夹入口，我们将其原始路径命名为 `opath`。然后，只在没有剩余路径的情况下使用文件名，并根据之前创建的匹配列表，对对应的匹配段进行替换。我们将拷贝 `opath` 到 `rpath` 中，并且将文件名进行替换。

```

for (const auto &entry :

recursive_directory_iterator{current_path()} ) {
    path opath {entry.path()};
    string rname
{replace(opath.filename().string(),
        patterns)};

    path rpath {opath};
    rpath.replace_filename(rname);

```

6. 对于匹配的文件，我们将打印其重命名后的名字。当重命名后的文件存在，我们将不会对其进行处理。会跳过这个文件。当然，我们也可以添加一些数字或其他字符到地址中，从而解决这个问题：

```

if (opath != rpath) {
    cout << opath.c_str() << " --> "
        << rpath.filename().c_str() << '\n';
    if (exists(rpath)) {
        cout << "Error: Can't rename."
            " Destination file exists.\n";
    } else {
        rename(opath, rpath);
    }
}
}

```

7. 编译并运行程序，我们将会得到如下的输出。我的文件夹下面有一些JPEG文件，但是都是以不同的后缀名结尾，有 `jpg`，`jpeg` 和 `JPEG`。然后，执行程序将 `jpeg` 和 `JPEG` 替换成 `jpg`。这样，就可以对文件名进行统一化。

```

$ ls
birthday_party.jpeg
holiday_in_dubai.jpgholiday_in_spain.jpg
trip_to_new_york.JPG
$ ./renamer jpeg jpg JPEG jpg
/Users/tfc/pictures/birthday_party.jpeg -->
birthday_party.jpg
/Users/tfc/pictures/trip_to_new_york.JPG -->
trip_to_new_york.jpg
$ ls
birthday_party.jpg holiday_in_dubai.jpg
holiday_in_spain.jpg
trip_to_new_york.jpg

```

实现一个磁盘使用统计器

我们已经实现了一个列出文件夹下所有文件的工具，不过和系统自带的工具一样，其都不会对文件夹的大小进行打印。

为了获取文件夹的大小，我们需要将其子文件夹进行迭代，然后将其包含的所有文件的大小进行累加，才能得到该文件夹的大小。

本节中，我们将实现一个工具用来做这件事。这个工具能在任意的文件夹下运行，并且对文件夹中包含的文件总体大小进行统计。

How to do it...

本节中，我们将会实现一个程序用于迭代目录中的所有文件，并将所有文件的大小进行统计。对于统计一个文件的大小就很简单，但是要统计一个文件夹的大小，就需要将文件夹下的所有文件的大小进行相加。

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <sstream>
#include <iomanip>
#include <numeric>
#include <filesystem>

using namespace std;
using namespace filesystem;
```

2. 我们将实现一个辅助函数使用 `directory_entry` 对象作为其参数，然后返回其在文件系统中对应的大小。如果传入的不是一个文件夹地址，将通过 `file_size` 获得文件的大小。

```
static size_t entry_size(const directory_entry
&entry)
{
    if (!is_directory(entry)) { return
file_size(entry); }
```

3. 如果传入的是一个文件夹，需要对其中所有元素进行文件大小的计算。需要调用辅助函数 `entry_size` 对子文件夹进行再次递归：

```

        return accumulate(directory_iterator{entry}, {}, 
0u,
[] (size_t accum, const directory_entry &e) {
    return accum + entry_size(e);
});
}

```

4. 为了具有更好的可读性，本节使用了其他章节中的 `size_string` 函数。

```

static string size_string(size_t size)
{
    stringstream ss;
    if (size >= 1000000000) {
        ss << (size / 1000000000) << 'G';
    } else if (size >= 1000000) {
        ss << (size / 1000000) << 'M';
    } else if (size >= 1000) {
        ss << (size / 1000) << 'K';
    } else { ss << size << 'B'; }

    return ss.str();
}

```

5. 主函数中，首先就是要检查用户通过命令行提供的文件系统路径。如果没有提供，则默认为当前文件夹。处理之前，我们会检查路径是否存在。

```

int main(int argc, char *argv[])
{
    path dir {argc > 1 ? argv[1] : "."};

    if (!exists(dir)) {
        cout << "Path " << dir << " does not exist.\n";
        return 1;
    }
}

```

6. 现在，我们可以对所有的文件夹进行迭代，然后打印其名字和大小：

```

for (const auto &entry : directory_iterator{dir})
{
    cout << setw(5) << right
        << size_string(entry_size(entry))
        << " " <<
    entry.path().filename().c_str()
        << '\n';
}
}

```

7. 编译并运行程序，我们将获得如下的输出。我提供了一个C++离线手册的目录，其当然具有子目录，我们可以用我们的程序对其进行统计：

```
$ ./file_size ~/Documents/cpp_reference/en/
19M c
12K c.html
147M cpp
17K cpp.html
22K index.html
22K Main_Page.html
```

How it works...

整个程序通过 `file_size` 对普通的文件进行大小的统计。当程序遇到一个文件夹，其将会对子文件夹进行递归，然后通过 `file_size` 计算出文件夹中包含所有文件的大小。

有件事我们需要区别一下，当我们直接调用 `file_size` 时，或需要进行递归时，需要通过 `is_directory` 谓词进行判断。这对于只包含有普通文件和文件夹的文件夹是有用的。

与我们的简单程序一样，程序会在如下的情况下崩溃，因为有未处理的异常抛出：

- `file_size` 只能对普通文件和符号链接有效。否则，会抛出异常。
- `file_size` 对符号链接有效，如果链接失效，函数还是会抛出异常。

为了让本节的程序更加成熟，我们需要更多的防御性编程，避免遇到错误的文件和手动处理异常。

计算文件类型的统计信息

上一节中，我们实现了一个用于统计任意文件夹中所有文件大小的工具。

本节中，我们将递归的对文件夹中的文件名后缀进行统计。这样对每种文件类型的文件进行个数统计，并且计算每种文件类型大小的平均值。

How to do it...

本节中将实现一个简单的工具用于对给定的文件夹进行递归，同时对所有文件的数量和大小进行统计，并通过文件后缀进行分组。最后，会对文件夹中具有的文件名扩展进行打印，并打印出有多少个对应类型扩展的文件和文件的平均大小。

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <sstream>
#include <iomanip>
#include <map>
#include <filesystem>

using namespace std;
using namespace filesystem;
```

2. `size_string` 函数已经在上一节中使用过了。这里我们继续使用：

```
static string size_string(size_t size)
{
    stringstream ss;
    if (size >= 1000000000) {
        ss << (size / 1000000000) << 'G';
    } else if (size >= 1000000) {
        ss << (size / 1000000) << 'M';
    } else if (size >= 1000) {
        ss << (size / 1000) << 'K';
    } else { ss << size << 'B'; }

    return ss.str();
}
```

3. 然后，实现一个函数用于接受一个 `path` 对象，并对该路径下的所有文件进行遍历。我们使用一个 `map` 来收集所有的信息，用对应的扩展名与总体数量和所有文件的总大小进行统计：

```
static map<string, pair<size_t, size_t>>
ext_stats(const path &dir)
{
    map<string, pair<size_t, size_t>> m;

    for (const auto &entry :
        recursive_directory_iterator{dir}) {
```

4. 如果目录入口是一个目录，我们将跳过这个入口。跳过的意思就是不会对这个目录进行递归操作。`recursive_directory_iterator` 可以完成这个工作，但是不需要去查找所有文件夹中的文件。

```
    const path p {entry.path()};
    const file_status fs {status(p)};

    if (is_directory(fs)) { continue; }
```

5. 接下来，会对文件的扩展名进行提取。如果文件没有扩展名，就会对其进行忽略：

```
    const string ext {p.extension().string()};

    if (ext.length() == 0) { continue; }
```

6. 接着，计算我们查找到文件的总体大小。然后，将会对 `map` 中同一扩展的对象进行聚合。如果对应类型还不存在，创建起来也很容易。我们可以简单的对文件计数进行增加，并且对扩展总体大小进行累加：

```
    const size_t size {file_size(p)};

    auto &[size_accum, count] = m[ext];

    size_accum += size;
    count += 1;
}
```

7. 之后，我们会返回这个 `map`：

```
return m;
```

8. 主函数中，我们会从用户提供的路径中获取对应的路径，或是使用当前路径。当然，需要对地址是否存在进行检查，否则继续下去就没有任何意义：

```

int main(int argc, char *argv[])
{
    path dir {argc > 1 ? argv[1] : "."};

    if (!exists(dir)) {
        cout << "Path " << dir << " does not
exist.\n";
        return 1;
    }
}

```

9. 可以对 `ext_stats` 进行遍历。因为 `map` 中的 `accum_size` 元素包含有同类型扩展文件的总大小，然后用其除以总数量，以计算出平均值：

```

for (const auto &[ext, stats] : ext_stats(dir)) {
    const auto &[accum_size, count] = stats;

    cout << setw(15) << left << ext << ":" "
        << setw(4) << right << count
        << " items, avg size "
        << setw(4) << size_string(accum_size /
count)
        << '\n';
}
}

```

10. 编译并运行程序，我们将会得到如下的输出。我将C++离线手册的地址，作为命令行的参数：

```

$ ./file_type ~/Documents/cpp_reference/
.css :2 items, avg size 41K
.gif :7 items, avg size 902B
.html: 4355 items, avg size 38K
.js:3 items, avg size 4K
.php :1 items, avg size 739B
.png : 34 items, avg size 2K
.svg : 53 items, avg size 6K
.ttf :2 items, avg size 421K

```

实现一个工具：通过符号链接减少重复文件，从而控制文件夹大小

很多工具以不同的方式对数据进行压缩。其中最著名的文件压缩算法/格式就是**ZIP**和**RAR**。这种工具通过减少文件内部冗余，从而减少文件的大小。

将文件压缩成压缩包外，另一个非常简单的减少磁盘使用率的范式就是删除重复的文件。本节中，我们将实现一个小工具，其会对目录进行递归。递归中将对文件内容进行对比，如果找到相同的文件，我们将对其中一个进行删除。所有删除的文件则由一个符号链接替代，该链接指向目前唯一存在的文件。这种方式可以不通过压缩对空间进行节省，同时对所有的数据能够进行保存。

How to do it...

本节中，将实现一个小工具用来查找那些重复的文件。我们将会删除其中一个重复的文件，并使用符号链接的方式对其进行替换，这样就能减小文件夹的大小。

Note:

为了对系统数据进行备份，我们将使用**STL**函数对文件进行删除。一个简单的拼写错误就可能会删除很多并不想删除的文件。

1. 包含必要的头文件，并声明所使用的命名空间：

```
#include <iostream>
#include <fstream>
#include <unordered_map>
#include <filesystem>

using namespace std;
using namespace filesystem;
```

2. 为了查找重复的文件，我们将构造一个哈希表，并将文件哈希值与其第一次产生的地址相对应。最好的方式就是通过哈希算法，对文件计算出一个**MD5**或**SHA**码。为了保证例子的简洁，我们将会把文件读入一个字符串中，然后使用**hash**函数计算出对应的哈希值：

```

static size_t hash_from_path(const path &p)
{
    ifstream is {p.c_str(),
                 ios::in | ios::binary};
    if (!is) { throw errno; }

    string s;

    is.seekg(0, ios::end);
    s.reserve(is.tellg());
    is.seekg(0, ios::beg);

    s.assign(istreambuf_iterator<char>{is}, {});
}

return hash<string>{}(s);
}

```

3. 然后，我们会实现一个哈希表，并且删除重复的文件。其会对当前文件夹和其子文件夹进行遍历：

```

static size_t reduce_dupes(const path &dir)
{
    unordered_map<size_t, path> m;
    size_t count {0};

    for (const auto &entry :
        recursive_directory_iterator{dir}) {

```

4. 对于每个文件入口，我们都会进行检查，当其是文件夹时就会跳过。对于每一个文件，我们都会产生一个哈希值，并且尝试将其插入哈希表中。当哈希表已经包含有相同的哈希值，这也就意味着有文件重复了。并且插入操作会终止，`try_emplace` 所返回的第二个值就是`false`：

```

const path p {entry.path()};

if (is_directory(p)) { continue; }

const auto &[it, success] =
    m.try_emplace(hash_from_path(p), p);

```

5. `try_emplace` 的返回值将告诉我们，该键是否是第一次插入的。这样我们就能找到重复的，并告诉用户文件有重复的，并将重复的进行删除。删除之后，我们将为重复的文件创建符号链接：

```

    if (!success) {
        cout << "Removed " << p.c_str()
            << " because it is a duplicate of "
            << it->second.c_str() << '\n';

        remove(p);
        create_symlink(absolute(it->second), p);
        ++count;
    }
}

```

6. 对文件系统进行插入后，我们将会返回重复文件的数量：

```

    }

    return count;
}

```

7. 主函数中，我们会对用户在命令行中提供的目录进行检查。

```

int main(int argc, char *argv[])
{
    if (argc != 2) {
        cout << "Usage: " << argv[0] << " <path>\n";
        return 1;
    }

    path dir {argv[1]};

    if (!exists(dir)) {
        cout << "Path " << dir << " does not
exist.\n";
        return 1;
    }
}

```

8. 现在我们只需要对 `reduce_dupes` 进行调用，并打印出有多少文件被删除了：

```

const size_t dupes {reduce_dupes(dir)};

cout << "Removed " << dupes << " duplicates.\n";
}

```

9. 编译并运行程序，输出中有一些看起来比较复杂的文件。程序执行之后，我会使用 `du` 工具来检查文件夹的大小，并证明这种方法是有效的。

```

$ du -sh dupe_dir
1.1Mdupe_dir

$ ./dupe_compress dupe_dir
Removed dupe_dir/dir2/bar.jpg because it is a
duplicate of
dupe_dir/dir1/bar.jpg
Removed dupe_dir/dir2/base10.png because it is a
duplicate of
dupe_dir/dir1/base10.png
Removed dupe_dir/dir2/baz.jpeg because it is a
duplicate of
dupe_dir/dir1/baz.jpeg
Removed dupe_dir/dir2/feed_fish.jpg because it is a
duplicate of
dupe_dir/dir1/feed_fish.jpg
Removed dupe_dir/dir2/foo.jpg because it is a
duplicate of
dupe_dir/dir1/foo.jpg
Removed dupe_dir/dir2/fox.jpg because it is a
duplicate of
dupe_dir/dir1/fox.jpg
Removed 6 duplicates.

$ du -sh dupe_dir
584Kdupe_dir

```

How it works...

使用 `create_symlink` 函数在文件系统中链接一个文件，指向另一个地方。这样就能避免重复的文件出现，也可以使用 `create_hard_link` 设置一些硬链接。硬链接和软连接相比，有不同的技术含义。有些格式的文件系统可能不支持硬链接，或者是使用一定数量的硬链接，指向相同的文件。另一个问题就是，硬链接没有办法让两个文件系统进行链接。

不过，除开实现细节，使用 `create_symlink` 或 `create_hard_link` 时，会出现一个明显的错误。下面的几行代码中就有一个bug。你能很快的找到它吗？

```

path a {"some_dir/some_file.txt"};
path b {"other_dir/other_file.txt"};
remove(b);
create_symlink(a, b);

```

在程序执行的时候，不会发生任何问题，不过符号链接将失效。符号链接将错误的指向 `some_dir/some_file.txt`。正确指向的地址应该是 `/absolute/path/some_dir/some_file.txt` 或 `../some_dir/some_file.txt`。`create_symlink` 使用正确的绝对地址，可以使用如下写法：

```
create_symlink(absolute(a), b);
```

Note:

`create_symlink` 不会对链接进行检查

There's more...

可以看到，哈希函数非常简单。为了让程序没有多余的依赖，我们采用了这种方式。

我们的哈希函数有什么问题呢？有两个问题：

- 会将一个文件完全读入到字符串中。如果对于很大的文件来说，这将是一场灾难。
- C++ 中的哈希函数 `hash<string>` 可能不是这样使用的。

要寻找一个更好的哈希函数时，我们需要找一个快速、内存友好、简单的，并且保证不同的文件具有不同的哈希值，最后一个需求可能是最关键的。因为我们使用哈希值了判断两个文件是否一致，当我们认为两个文件一致时，但哈希值不一样，就能肯定有数据受到了损失。

比较好的哈希算法有MD5和SHA(有变体)。为了让我们程序使用这样的函数，可能需要使用OpenSSL中的密码学API。