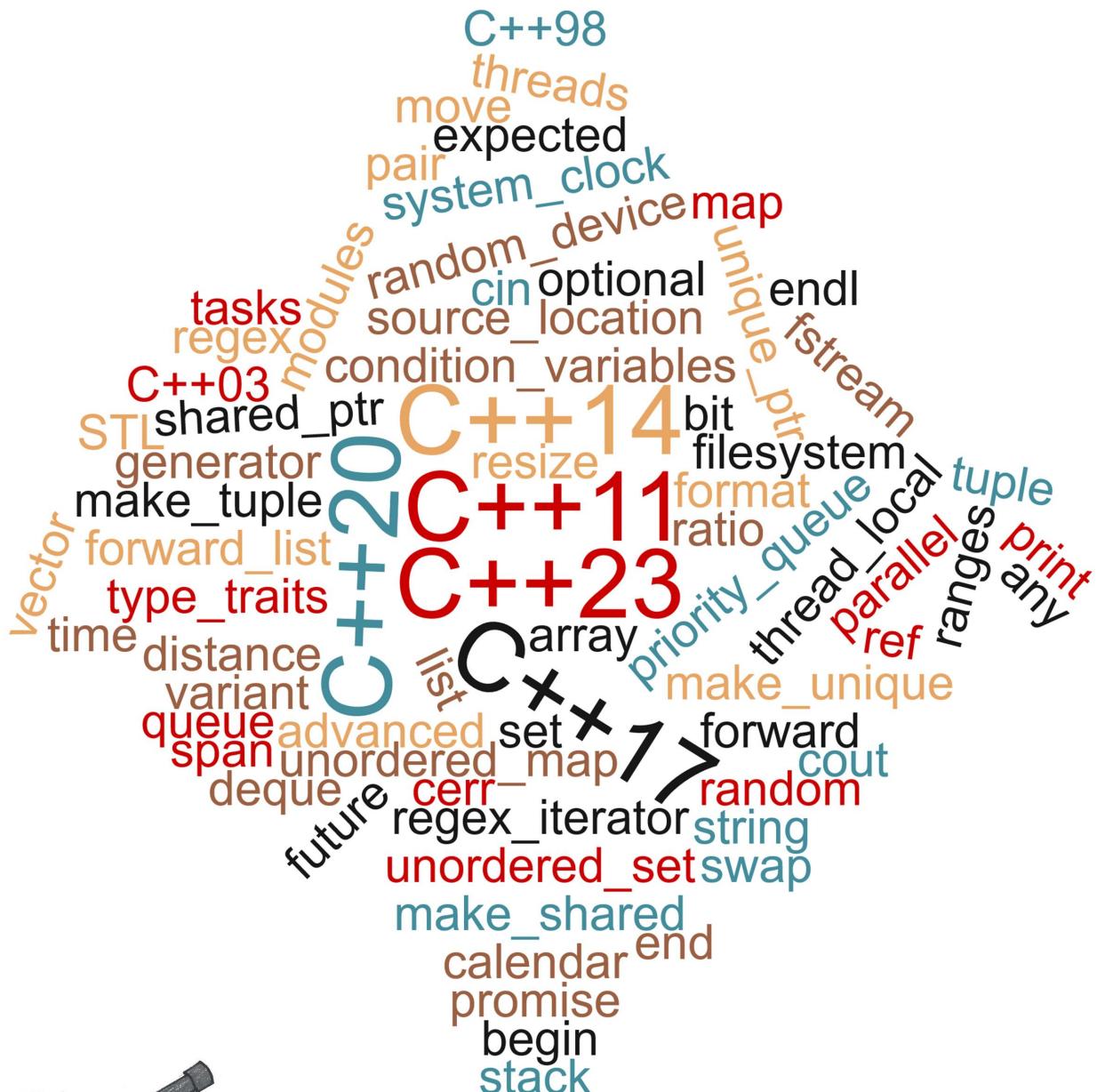


The C++ Standard Library

Fourth Edition includes C++23



Rainer Grimm

ModernesCpp.com

The C++ Standard Library

What every professional C++ programmer should know
about the C++ standard library.

作者: Rainer Grimm

译者: 陈晓伟

版本: [2023-03-11](#)

目录

本书目标	7
致谢	7
更多信息	7
Cippi	7
关于作者	8
第 1 章 标准库	9
1.1. 历史背景	9
1.2. 概述	9
1.3. 如何使用	13
第 2 章 实用工具	17
2.1. 有用的功能	17
2.2. 函数适配器	22
2.3. 数值对	24
2.4. 元组	24
2.5. 引用包装器	26
2.6. 智能指针	28
2.7. 类型特性	35
2.8. 时间库	42
2.9. std::any, std::optional 和 std::variant	49
2.10. std::expected	54
第 3 章 所有容器的接口	57
3.1. 创建和删除	57
3.2. 大小	59
3.3. 访问方式	59
3.4. 赋值和交换	61
3.5. 比较	61
3.6. 擦除	62
第 4 章 顺序容器	64
4.1. Array	65
4.2. Vector	66
4.3. Deque	68

4.4. List	69
4.5. 前向 List	70
第 5 章 关联容器	73
5.1. 概述	73
5.2. 有序关联容器	76
5.3. 无序关联容器	79
第 6 章 容器适配器	84
6.1. 线性容器	84
6.2. 关联性容器	87
第 7 章 视图	89
7.1. 连续访问	89
7.2. 多维访问	92
第 8 章 迭代器	97
8.1. 类型	97
8.2. 创建迭代器	98
8.3. 实用的功能	99
8.4. 适配器	101
第 9 章 可调用单元	104
9.1. 函数	104
9.2. 函数对象	105
9.3. Lambda 函数	106
第 10 章 算法	107
10.1. 约定	108
10.2. 迭代器是粘合剂	109
10.3. 顺序, 并行或并行执行与先向量化	109
10.4. for_each	112
10.5. 不可修改算法	113
10.6. 可修改算法	120
10.7. 分区	131
10.8. 排序	133
10.9. 二分查找	135
10.10. 合并操作	136
10.11. 堆	139
10.12. 最小和最大	141
10.13. 排列	142
10.14. 数值	143
10.15. 单元化内存	149
第 11 章 范围库	151
11.1. 范围	151

11.2. 视图	153
11.3. 范围适配器	153
11.4. 直接用于容器	156
11.5. 函数复合	157
11.6. 惰性计算	158
11.7. std 的算法与 std::ranges 的算法	160
第 12 章 数值	161
12.1. 随机数	161
12.2. 从 C 继承的数值函数	163
12.3. 数学常数	165
第 13 章 字符串	167
13.1. 创建和删除	168
13.2. C++ 和 C 字符串之间的转换	169
13.3. 大小与容量	170
13.4. 比较	171
13.5. 连接字符串	171
13.6. 访问元素	172
13.7. 输入与输出	173
13.8. 搜索	174
13.9. 检查是否有子字符串	175
13.10. 修改操作	176
13.11. 数值转换	178
第 14 章 字符串视图	181
14.1. 创建和初始化	182
14.2. 无修改的操作	182
14.3. 可修改的操作	182
第 15 章 正则表达式	185
15.1. 字符类型	186
15.2. 正则表达式对象	186
15.3. 搜索结果 <code>match_results</code>	187
15.4. 匹配	190
15.5. 搜索	191
15.6. 替换	192
15.7. 格式化	193
15.8. 反复搜索	194
第 16 章 输入和输出流	196
16.1. 层次结构	196
16.2. 输入输出函数	197
16.3. 流	203

16.4. 自定义数据类型	209
第 17 章 格式库	211
17.1. 格式化功能	211
17.2. 语法	212
17.3. 格式规范	213
17.4. 自定义格式化器	215
第 18 章 文件系统	217
18.1. 类	218
18.2. 非成员函数	220
18.3. 文件类型	223
第 19 章 多线程	226
19.1. 内存模型	226
19.2. 原子数据类型	226
19.3. 线程	232
19.4. 停止令牌	237
19.5. 共享变量	239
19.6. 线程本地数据	248
19.7. 条件变量	249
19.8. 信号量	251
19.9. 协调类型	253
19.10. 任务	256
第 20 章 协程	262
20.1. 可等待	263
20.2. 无限数据流—— <code>co_yield</code>	263

本书目标

本书是对当前 C++23 标准 ISO/IEC 14882:2023 标准库的快速参考。C++20 标准有 2100 页之多，相比之下，C++23 和 C++17 既是不大不小的 C++ 标准更新，而 C++14 是对 C++11 的补充。

2011 年发布的 C++11 有 1300 多页。这距离第一个（也是唯一一个 C++ 标准）C++98 已经过去了 13 年。当然，还有 2003 年发布的 C++03，但 C++03 更多的是作为 bug 修复的 C++ 版本。

这个快速参考的目的是提供 C++ 标准库的简要参考。本书假定读者熟悉 C++。如对 C++ 很陌生，应该从关于 C++ 核心的教科书开始学习。当掌握了一本关于核心语言的书，就可以通过阅读这本书迈出下一步。为了更便于理解，我在本书中提供了许多简短的代码段，将理论和实践进行串联。

致谢

首先，我要感谢奥莱利学院的讲师 Alexandra Follenius，她的德文书 [C++ Standardbibliothek](#) 是我《C++ Standardbibliothek》的始祖，Karsten Ahnert、Guntram Berti、Dmitry Ganyushin、Sven Johannsen、Torsten Robitzki、Bart Vandewoestyne 和 Felix Winter 都是非常有价值的校对。非常感谢他们所有人。

我开始在英文博客上请求把这本书翻译成英文 [www.ModernesCpp.com](#)。我收到的反馈比预期的要高得多。特别感谢你们所有人，也包括第一任校对——我的儿子 Marius。

以下是按字母顺序排列的名字：Mahesh Attarde, Rick Audet, Pete Barrow, Michael BenDavid, Dave Burns, Alvaro Fernandez, Juliette Grimm, George Haake, Clare Macrae, Arne Mertz, Ian Reeve, Jason Turner, Bart Vandewoestyne, Ivan Vergiliev 和 Andrzej Warzynski。

更多信息

这本书的思想相对容易解释：“关于 C++ 标准库，每个专业的 C++ 开发者都应该知道什么。”因此，我留下了许多没有回答的问题，并且在每个新主题的开头提供了详细信息的链接。

优秀的在线资源 [www.cppreference.com](#)。

Cippi

来介绍一下 Cippi。Cippi 将在这本书中陪伴你阅读，希望你能喜欢她。



我是 Cippi，我好奇、聪明，并且“女人味”十足!!

关于作者

自 1999 年以来，我一直担任软件架构师、团队领导和讲师。2002 年，我创建了公司实习生进修会议。自 2002 年以来，我一直在教授培训课程。我的第一个教程是关于专有管理软件的，但不久之后我开始教授 Python 和 C++。我喜欢在业余时间写一些关于 C++、Python 和 Haskell 的文章，也喜欢在会议上发言。我每周都会在我的英文博客[Modernes Cpp](#)和由 Heise Developer 托管的[German blog](#)上发表文章。

自 2016 年以来，我一直是一名独立讲师，讲授有关现代 C++ 和 Python 的研讨会。我已经用各种语言出版了几本关于现代 C++ 的书，特别是关于并发性的书。由于我的职业，我一直在寻找教授现代 C++ 的最佳方法。



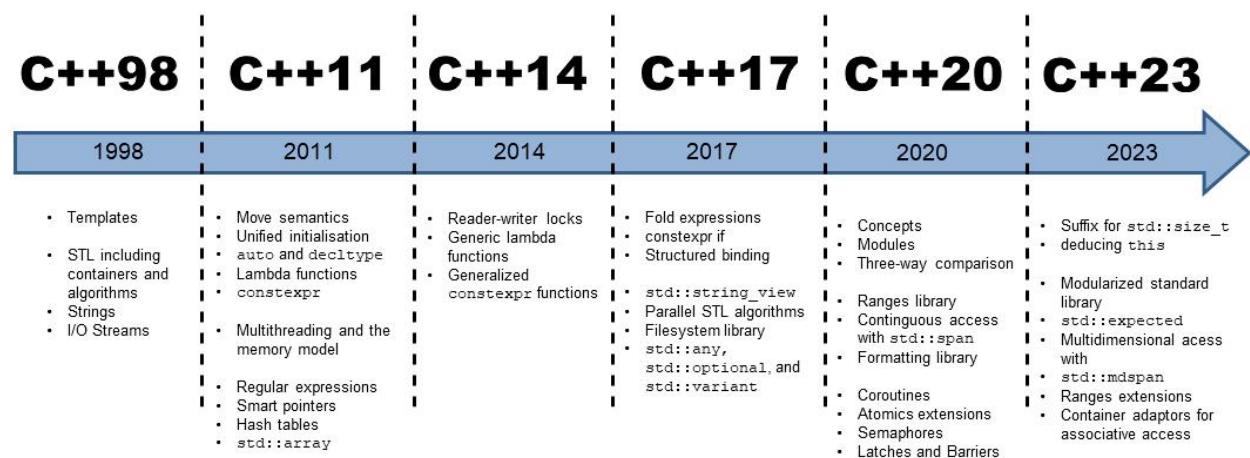
Rainer Grimm

第 1 章 标准库

C++ 标准库由许多组件组成。本章有两个目的：快速概述相应功能，以及如何使用它们。

1.1. 历史背景

C++ 和标准库有着悠久的历史，其始于 20 世纪 80 年代，结束于 2023 年。了解软件开发的人都知道我们的领域发展得有多快，所以 40 年是很长的一段时间。C++ 的第一个组件，如 I/O 流，是不同于现代标准模板库 (STL) 的思维方式设计的。C++ 最初是一种面向对象的语言，将泛型编程与 STL 结合在一起，现在采用了许多函数式编程的思想。可以在 C++ 标准库中观察到，过去 40 年的这种演变，这也是软件问题解决方式的演变。



C++ 的时间线

1998 年的第一个 C++98 标准库有三个组件。这些是前面提到的 I/O 流，主要用于文件处理、字符串库和标准模板库。

标准模板库促进了算法在容器上的透明应用。2005 年技术报告 1 (TR1) 延续了这一历史。C++ 库 ISO/IEC TR 19768 的扩展并不是官方标准，但几乎所有的组件都成为 C++11 的一部分。例如，基于 boost 库 (<http://www.boost.org/>) 的正则表达式、智能指针、哈希表、随机数和时间库。

除了 TR1 的标准化之外，C++11 还增加了一个新组件：多线程库。

C++14 只是对 C++11 标准的一个小更新，C++14 只对现有的智能指针、元组、类型特征和多线程库进行了一些改进。

C++17 包含了文件系统的库和两个新的数据类型 std::any 和 std::optional。

C++20 有四个突出的特性：概念、范围、协程和模块。除了这四大特性之外，C++20 中还有更多的特性：三方比较操作符、格式化库以及与并发相关的数据类型——信号量、锁存器和栅栏。

C++23 改进了 C++20 的四大功能：扩展范围功能、协程生成器 std::generator 和模块化的 C++ 标准库。

1.2. 概述

由于 C++ 有许多库，为每个用例很难找到最合适的库。

实用工具

实用工具是具有一般性的库，因此可以在许多上下文中应用。

其包括计算最小值或最大值、两个值的中点或交换或移动值的函数。由于保留了整数的比较，所以不会进行整型的类型提升。

其他实用工具有 `std::function`、`std::bind` 或 `std::bind_front`。使用 `std::bind` 或 `std::bind_front`，可以轻松地从现有函数创建新函数。要将它们绑定到一个变量并在以后调用，可以使用 `std::function`。

使用 `std::pair` 和泛化 `std::tuple`，可以创建任意长度的异构对和元组。

引用包装器 `std::ref` 和 `std:: cref` 非常方便。可以使用它们为变量创建引用包装器，而 `std:: cref` 是常量。

实用工具最实用是智能指针，其允许显式的自动内存管理。可以使用 `std::unique_ptr` 对显式所有权的概念进行建模，并使用 `std::shared_ptr` 对共享所有权进行建模。`shared_ptr` 使用引用计数来管理其资源。第三个是 `std::weak_ptr`，这有助于打破 `std::shared_ptr` 之间的循环依赖关系。不过，循环引用是引用计数的经典问题。

类型特征库可以在编译时检查、比较和操作类型信息。

时间库是对 C++ 新多线程功能的重要补充，可以方便地进行性能测量，并包括对日历和时区的支持。

有了 `std::any`、`std::optional` 和 `std::variant`，C++17 就有了三种特殊的数据类型。

标准模板库



STL 的三个组成部分

标准模板库 (STL) 由三个组件组成。它们是容器，在容器上运行的算法，以及连接两者的迭代器。

容器对其元素只有最小的要求，这种泛型编程能够组合算法和容器。

C++ 标准库有丰富的容器集合，有顺序容器和关联容器。关联容器可以分为有序关联容器和无序关联容器。

每个序列容器都有惟一的域。尽管如此，95% 的用例中，`std::vector` 是正确的选择。`std::vector` 可以动态调整其大小，自动管理其内存，提供出色的性能。相反，`std::array` 是唯一不能在运行时调整其大小的序列容器，其针对最小的内存和性能开销进行了优化。

虽然 `std::vector` 常将新元素放在末尾，但使用 `std::deque` 是会将元素放在开头。`std::list` 是双链表，`std::forward_list` 是单链表，并针对这两种容器中位置操作进行了优化。

关联容器是键值对的容器，通过各自的键提供它们的值。关联容器的典型用例是电话簿，其中使用键姓来检索值电话号码。C++ 有八种不同的关联容器：其一是带有有序键的关联容器：`std::set`、`std::map`、`std::multiset` 和 `std::multimap`。另外，还有无序关联容器：`std::unordered_set`、`std::unordered_map`、`std::unordered_multiset` 和 `std::unordered_multimap`。

首先是序关联容器，`std::set` 与 `std::map` 的区别在于前者没有关联值。`std::map` 和 `std::multimap` 的区别在于后者可以有多个相同的键。这些命名约定也适用于无序关联容器，其与有序容器有很多共同之处，区别在于性能。有序关联容器的访问时间为对数，而无序关联容器的访问时间为常数。因此，无序关联容器的访问时间与它们的大小无关，`std::unordered_map` 与 `std::vector` 适用相同的规则。在 95% 的用例中，若不需要排序键，`std::unordered_map` 应该是首选。

容器适配器为序列容器提供了一个简化的接口。C++ 有 `std::stack`、`std::queue` 和 `std::priority_queue`。

支持 C-array、`std::array`、`std::vector` 或 `std::string` 视图。`std::span` 是一个连续元素序列的视图，并且视图不是所有者。

迭代器是容器和算法之间的粘合剂。容器创建它们，作为通用指针，可以使用它们向前和向后迭代或跳转到容器中的相应位置。迭代器类型取决于容器，若使用迭代器适配器，就可以直接访问流。

STL 为提供了 100 多个算法。指定执行策略，可以顺序、并行或并行和向量化地运行大多数算法。算法对元素或一系列元素进行操作。两个迭代器定义一个范围。第一个定义开始迭代器，第二个称为结束迭代器，定义范围的结束，并且结束迭代器指向超出范围末端的一个元素。

该算法具有广泛的应用前景。可以查找元素，进行计数，查找范围，并进行比较或转换。有一些算法可以从容器中生成、替换或删除元素。也可以对容器进行排序、排列或分区，或者确定容器的最小或最大元素。许多算法可以通过函数、函数对象或 Lambda 函数等可调用对象进一步定制。可调用对象为元素的搜索或转换提供了特殊的标准，极大地提高了算法的能力。

范围库提供惰性算法，可以直接在容器上工作，并且很容易组合，并用函数思想扩展了 C++。此外，大多数经典 STL 算法都具有范围，支持投影并提供安全保证。

数值方面

C++ 中有两个用于数字的库：随机数库和数学函数，这是 C++ 从 C 继承来的。

随机数库由两部分组成。有随机数生成器，还有随机数的分布。随机数生成器生成一个介于最小值和最大值之间的数字流，随机数分布将其映射到具体分布上。

因为 C 语言，C++ 有很多数学标准函数。例如，有对数函数、指数函数和三角函数。

C++ 也支持基本和高级数学常数，如 e、π 或 φ。

文本处理

通过字符串和正则表达式，C++ 有两个强大的库来处理文本。

`std::string` 拥有丰富的成员函数集合来分析和修改其文本，它与字符的 `std::vector` 有很多共同之

处，所以可以将 STL 算法应用于 `std::string`。`std::string` 是 C 字符串的继承者，但使用起来更容易、更安全。C++ 字符串可以很好的管理其内存。

与 `std::string` 相比，`std::string_view` 的复制成本很低。`std::string_view` 是对 `std::string` 的非所属引用。

正则表达式是一种描述文本模式的语言。使用正则表达式确定文本模式在文本中是出现一次还是多次，但这还不是全部。正则表达式支持用文本替换匹配的模式。

输入输出

I/O 流库是一个从 C++ 开始就存在的库，可与外部进行通信。

提取操作符 (`>>`) 允许它从输入流中读取格式化或未格式化的数据，插入操作符 (`<<`) 允许将数据写入输出流，可以使用操纵符进行格式化。

流类具有复杂的类层次结构。两个流类很重要：首先，字符串流允许与字符串和流交互。其次，文件流能够轻松地读取和写入文件。流的状态保存在可以读取和操作的标志中。

通过重载输入操作符和输出操作符，自定义类可以像基本数据类型一样与外部交互。

格式化库为 `printf` 家族提供了一种安全且可扩展的替代方案，并扩展了 I/O 流库。

与 I/O 流库相比，文件系统库在 C++17 中添加到正式标准中。该库基于三个概念：文件、文件名和路径。文件可以是目录、硬链接、符号链接或普通文件，路径可以是绝对的也可以是相对的。

文件系统库支持一个强大的接口来读取和操作文件系统。

多线程

C++ 在 2011 年发布的 C++ 标准中加入了一个多线程库。这个库具有基本的构建块，如原子变量、线程、锁和条件变量，是未来 C++ 标准构建更高抽象的基础。C++11 已经在提供比前面提到的基本构建块更高的抽象了。

C++11 首次提供了内存模型和原子变量，这两个组件都是多线程编程中定义良好行为的基础。

C++ 中的新线程将立即开始工作，其可以在前台或后台运行，并通过复制或引用获取数据。有了停止令牌，就可以中断改进后的线程 `std::jthread`。

线程之间共享变量的访问必须协调，可以通过互斥锁或锁的不同方式来完成。通常，保护数据的初始化就足够了，因为它在其生命周期内是不可变的。

将变量声明为线程局部可确保线程获得其副本，因此不会产生冲突。

条件变量是实现发送方-接收方工作流的经典解决方案。关键思想是发送方在完成其工作时通知接收方，以便接收方可以开始工作。

信号量是一种同步机制，用于控制对共享资源的并发访问。信号量有一个大于零的计数器，获取信号量会减少计数器，释放信号量会增加计数器。只有当计数器大于零时，线程才能获取资源。

与信号量类似，`std::latch` 和 `std::barrier` 是允许某些线程阻塞直到计数器变为零的协调类型。与 `std::barrier` 相反，可以为新的迭代重用 `std::latch`，并为这个新迭代调整它的计数器。

任务与线程有很多共同之处。但当开发者显式地创建线程时，任务将由 C++ 运行时隐式地创建。任务就像数据通道。数据可以是一个值、一个异常，或者只是一个通知。`promise` 将数据放入数据通道，`future` 会带回相应的值。

协程是可以在保持状态的情况下暂停和恢复执行的函数。协程是编写[事件驱动型应用](#)的常用方法。事件驱动型应用程序可以是模拟、游戏、服务器、用户界面，甚至是算法。协程可启用[协调式多任务](#)，其处理的关键是每个任务都需要尽可能多的时间。

1.3. 如何使用

要使用库，必须有三个步骤。首先，必须使用 `#include` 语句包含相应头文件，以便编译器知道库的名称。或者，可以使用 `import std;` C++23 中可导入标准库。由于 C++ 标准库名称位于命名空间 `std` 中，因此可以在第二步中使用限定的名称，或者必须在全局命名空间中导入。第三步也是最后一步是指定链接器获取可执行文件所需的库。第三步通常是可选的。下面几行解释了这三个步骤。

包含头文件

预处理器在 `#include` 语句之后包含该文件。通常，这是一个头文件。头文件放在尖括号中：

```
1 #include <iostream>
2 #include <vector>
```

指定所有必要的头文件

编译器可以自由地向头文件中添加额外的头文件，代码可能拥有所有必需的头文件，尽管没有指定它们。不建议依赖此功能。所有需要的头文件都应该明确指定。否则，编译器升级或代码移植可能会引发编译错误。

导入标准库

C++23 中，可以使用 `import` 语句导入整个标准库：

```
1 import std;
```

`import std;` 语句从 C++ 头文件和 C 包装头文件（如 `std::printf` 来自于 `<cstdio>`）中导入命名空间 `std` 中的所有内容。此外，还从 `<new>` 头文件中导入 `::operator new` 或 `::operator delete`。

若还想从 C 包装器头文件 `<stdio.h>` 中导入全局命名空间对应项，例如 `::printf`，请使用 `import std.compat`。

比起包含头文件，更推荐导入模块的方式

模块与头文件相比有许多优点。这些优点不仅适用于模块化的标准库，也适用于用户定义的模块。

模块只导入一次，而且这个过程实际上无开销。导入模块的顺序没有区别，并且模块的重复名称是不可能的。模块能够表达代码的逻辑结构，可以显式指定应该导出或不导出的名称。此外，可以将几个模块捆绑成一个更广泛的模块，并将它们作为一个逻辑包提供给用户。

使用头文件和模块

在以下代码片段中，我使用了两种语法形式：头文件和模块。实际操作时应该只使用其中一种。

使用命名空间

若使用限定名，则应精确地按照定义使用它们。对于每个命名空间，必须使用范围解析操作符`::`。C++ 标准库的更多库使用嵌套命名空间。

```
1 #include <iostream> // either
2 #include <chrono> // either
3 ...
4 #import std; // or
5 ...
6 std::cout << "Hello world:" << '\n';
7 auto timeNow= std::chrono::system_clock::now();
```

使用非限定名称

C++ 中，可以通过 `using` 声明和 `using` 指令来使用名称。

使用声明

`using` 声明将一个名称添加到应用 `using` 声明的可见性范围内：

```
1 #include <iostream> // either
2 #include <chrono> // either
3 ...
4 #import std; // or
5 ...
6 using std::cout;
7 using std::endl;
8 using std::chrono::system_clock;
9 ...
10 cout << "Hello world:" << endl; // unqualified name
11 auto timeNow= now(); // unqualified name
```

`using` 声明有以下效果:

- 若在相同可见性作用域中声明了相同的名称，则会出现查找不到明确和编译器错误。
- 若在周围的可见性作用域中声明了相同的名称，则将被 `using` 声明隐藏。

使用指令

`using` 指令允许它不加限制地使用所有命名空间名称。

```
1 #include <iostream> // either
2 #include <chrono> // either
3 ...
4 #import std; // or
5 ...
6 using namespace std;
7 ...
8 cout << "Hello world:" << endl; // unqualified name
9 auto timeNow= chrono::system_clock::now(); // partially qualified name
```

`using` 指令不给当前可见性作用域添加任何名称，只能使名称可访问。所以:

- 若在相同可见性作用域中声明了相同的名称，则会出现查找不到明确和编译器错误。
- 本地名称空间中的名称隐藏了在周围名称空间中声明的名称。
- 若在不同的命名空间中可以看到相同的名称，或者命名空间中的名称隐藏了全局作用域中的名称，则会发生歧义查找，从而导致编译器错误。

源文件中小心使用 `using` 指令

在源文件中使用 `using` 指令应该非常小心，因为通过使用命名空间 `std` 的指令，所有来自 `std` 的名字都是可见的。这包括在本地或周围名称空间中意外隐藏名称的名称。

不要在头文件中使用 `using` 指令。若包含了一个带有 `using namespace std` 指令的头文件，那么 `std` 中的所有名字都是可见的。

命名空间别名

命名空间别名定义命名空间的同义词。对于较长的命名空间名称或嵌套的命名空间，使用别名通常很方便

```
1 #include <chrono> // either
2 ...
3 #import std; // or
4 ...
5 namespace sysClock= std::chrono::system_clock;
```

```
6 auto nowFirst= sysClock::now();  
7 auto nowSecond= std::chrono::system_clock::now();
```

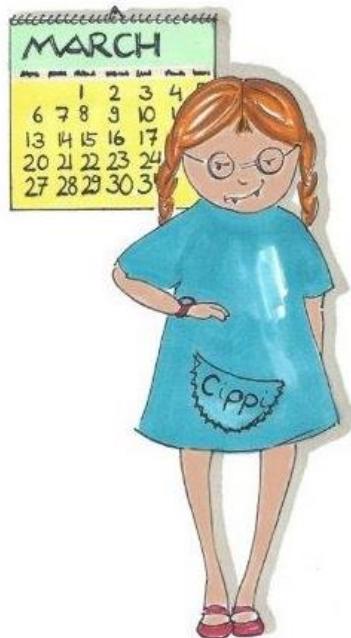
由于命名空间别名的存在，可以使用该别名来处理现在限定的函数。命名空间别名不能隐藏名称。

构建可执行文件

很少需要针对库显式链接，并且是平台相关的。例如，对于当前的 g++ 或 clang++ 编译器，应该链接到 pthread 库以获得多线程功能。

```
g++ -std=c++14 thread.cpp -o thread -pthread
```

第 2 章 实用工具



Cippi 在研究日历

实用工具是可以在不同上下文中使用的工具，其不绑定到特定的域。这句话适用于本章的所有函数和库。这里介绍了可以应用于任意值的函数，或者可以用来创建新函数并将它们绑定到变量的函数。可以将任意类型的任何值存储在对和元组中，或者构建任何值的引用。智能指针是在 C++ 中实现自动内存管理的工具。要获取类型信息，请使用类型特征库。

2.1. 有用的功能

可以将 `min`、`max` 和 `minmax` 函数的许多变体应用于值和初始化列表。这些函数需要头文件 `<algorithm>`。相反，`std::move`、`std::forward`、`std::to_underlying` 和 `std::swap` 函数在头文件 `<utility>` 中定义。

`std::min`, `std::max` 和 `std::minmax`

头文件 `<algorithm>` 中定义的函数 `std::min`、`std::max` 和 `std::minmax` 作用于值和初始化器列表，并返回所请求的值作为结果。在 `std::minmax` 的情况下，返回一个 `std::pair`。这对的第一个元素是最小值；第二个是值的最大值。默认情况下使用 `less` 操作符 (`<`)，但可以应用比较操作符。这个函数需要两个参数并返回一个布尔值，返回 `true` 或 `false` 的函数称为谓词。

`std::min`, `std::max` 和 `std::minmax`

```
1 // minMax.cpp
2 ...
3 #include <algorithm>
4 ...
```

```

5  using std::cout;
6  ...
7  cout << std::min(2011, 2014); // 2011
8  cout << std::min({3, 1, 2011, 2014, -5}); // -5
9  cout << std::min(-10, -5, [](int a, int b)
10     { return std::abs(a) < std::abs(b); });
11
12 auto pairInt= std::minmax(2011, 2014);
13 auto pairSeq= std::minmax({3, 1, 2011, 2014, -5});
14 auto pairAbs= std::minmax({3, 1, 2011, 2014, -5}, [](int a, int b)
15     { return std::abs(a) < std::abs(b); });
16
17 cout << pairInt.first << "," << pairInt.second; // 2011,2014
18 cout << pairSeq.first << "," << pairSeq.second; // -5,2014
19 cout << pairAbs.first << "," << pairAbs.second; // 1,2014

```

该表提供了 `std::min`、`std::max` 和 `std::minmax` 函数的概述

`std::min`、`std::max` 和 `std::minmax` 的不同

函数	描述
<code>min(a, b)</code>	返回 <code>a</code> 和 <code>b</code> 中较小的值
<code>min(a, b, comp)</code>	根据谓词 <code>comp</code> 返回 <code>a</code> 和 <code>b</code> 的较小值。
<code>min(initialiser list)</code>	返回初始化列表的最小值。
<code>min(initialiser list, comp)</code>	根据谓词 <code>comp</code> 返回初始化项列表的最小值。
<code>max(a, b)</code>	返回 <code>a</code> 和 <code>b</code> 中较大的值。
<code>max(a, b, comp)</code>	根据谓词 <code>comp</code> 返回 <code>a</code> 和 <code>b</code> 中较大的值。
<code>max(initialiser list)</code>	返回初始化项列表的最大值。
<code>max(initialiser list, comp)</code>	根据谓词 <code>comp</code> 返回初始化项列表的最大值。
<code>minmax(a, b)</code>	返回 <code>a</code> 和 <code>b</code> 的较大和较小的值。
<code>minmax(a, b, comp)</code>	根据谓词 <code>comp</code> 返回 <code>a</code> 和 <code>b</code> 的较大和较小的值。
<code>minmax(initialiser list)</code>	返回初始化项列表的最小值和最大值。
<code>minmax(initialiser list, comp)</code>	根据谓词 <code>comp</code> 返回初始化项列表的最小值和最大值。

`std::midpoint` 和 `std::lerp`

函数 `std::midpoint(a, b)` 计算 `a` 和 `b` 之间的中点。`a` 和 `b` 可以是整数、浮点数或指针。若 `a` 和 `b` 是指针，必须指向同一个数组对象。函数 `std::midpoint` 要求包含头文件 `<numeric>`。

函数 `std::lerp(a, b, t)` 计算两个数字的线性插值，需要头文件 `<cmath>`。返回值是 $a + t(b - a)$ 。

`std::midpoint` 和 `std::lerp`

```

1 // midpointLerp.cpp
2
3 #include <cmath>
4 #include <numeric>
5
6 ...
7
8 std::cout << "std::midpoint(10, 20): " << std::midpoint(10, 20) << '\n';
9
10 for (auto v: {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}) {
11     std::cout << "std::lerp(10, 20, " << v << "): " << std::lerp(10, 20, v) << '\n';
12 }
```

```

C:\x64 Native Tools Command Prompt for VS 2019
C:\Users\rainer>midpointLerp.exe

std::midpoint(10, 20): 15

std::lerp(10, 20, 0): 10
std::lerp(10, 20, 0.1): 11
std::lerp(10, 20, 0.2): 12
std::lerp(10, 20, 0.3): 13
std::lerp(10, 20, 0.4): 14
std::lerp(10, 20, 0.5): 15
std::lerp(10, 20, 0.6): 16
std::lerp(10, 20, 0.7): 17
std::lerp(10, 20, 0.8): 18
std::lerp(10, 20, 0.9): 19
std::lerp(10, 20, 1): 20

C:\Users\rainer>
```

std::cmp_equal, std::cmp_not_equal, std::cmp_less, std::cmp_greater, std::cmp_less_equal 和 std::cmp_greater_equal

头文件 `<utility>` 中定义的函数 `std::cmp_equal`、`std::cmp_not_equal`、`std::cmp_less`、`std::cmp_greater`、`std::cmp_less_equal` 和 `std::cmp_greater_equal` 提供了整数的安全比较。安全比较对负符号整数的比较小于对无符号整数的比较，并且对有符号整数或无符号整数以外的值的比较，会产生编译时错误。

使用内置整数进行整数转换

下面的代码片段举例说明了有符号/无符号比较的问题。

```

1 -1 < 0u; // true
2 std::cmp_greater(-1, 0u); // false
```

作为有符号整数的-1会提升为无符号类型，会导致令人意外的结果。

std::move

头文件 `<utility>` 中定义了函数 `std::move`，授权编译器移动资源。移动语义中，源对象的值移动到新对象中。之后，源处于定义良好但不确定的状态。通常，这是源的默认状态。通过使用 `std::move`，编译器将源参数转换为右值引用:`static_cast<std::remove_reference<decltype(arg)>::type&&>(arg)`。若编译器不能应用移动语义，则会退回到复制语义：

```
1 #include <utility>
2 ...
3 std::vector<int> myBigVec(10000000, 2011);
4 std::vector<int> myVec;
5
6 myVec = myBigVec; // copy semantics
7 myVec = std::move(myBigVec); // move semantics
```

移动比复制廉价

移动语义有两个优点。首先，使用便宜的移动而不是昂贵的复制通常是一个好主意，不需要额外的内存分配和释放。其次，有些对象是不能复制的。例如，线程或锁。

std::forward

头文件 `<utility>` 中定义的函数 `std::forward` 能够编写函数模板，这些函数模板可以相同地转发它们的参数。`std::forward` 的典型用例是工厂函数或构造函数。工厂函数创建一个对象，因此应该不加修改地传递其参数。构造函数通常使用实参来初始化具有相同实参的基类，所以 `std::forward` 对于泛型库作者来说是一个完美的工具：

完美转发

```
1 // forward.cpp
2 ...
3 #include <utility>
4 ...
5 using std::initializer_list;
6
7 struct MyData{
8     MyData(int, double, char) {};
9 };
10
11 template <typename T, typename... Args>
12 T createT(Args&&... args){
13     return T(std::forward<Args>(args)...);
```

```

14 }
15 ...
16 ...
17
18 int a= createT<int>();
19 int b= createT<int>(1);
20
21 std::string s= createT<std::string>("Only for testing.");
22 MyData myData2= createT<MyData>(1, 3.19, 'a');
23
24 typedef std::vector<int> IntVec;
25 IntVec intVec= createT<IntVec>(initialiser_list<int>({1, 2, 3}));

```

函数模板 `createT` 必须以通用引用: `args&&...args` 的形式接受参数。通用引用或转发引用是类型演绎上下文中的右值引用。

std::forward 支持与可变模板结合使用的完全泛型函数

若将 `std::forward` 与可变模板一起使用，则可以定义完全泛型的函数模板。函数模板可以接受任意数量的参数，并且不加修改地对其进行转发。

`std::to_underlying`

C++23 中的 `std::to_underlying` 函数将枚举 `enum` 转换为其底层类型 `Enum`。这个函数是表达式 `static_cast<std::underlying_type<Enum>::type>(Enum)` 的方便函数，使用类型特征函数 `std::underlying_type`。

`std::swap`

通过 `<utility>` 中定义的函数 `std::swap`，可以轻松地交换两个对象。C++ 标准库中的泛型实现会在内部使用 `std::move` 函数。

`std::swap` 中使用移动语义

```

1 // swap.cpp
2 ...
3 #include <utility>
4 ...
5 template <typename T>
6 inline void swap(T& a, T& b) noexcept {
7     T tmp(std::move(a));
8     a = std::move(b);
9     b = std::move(tmp);
10 }

```

2.2. 函数适配器

`std::bind`、`std::bind_front`、`std::bind_back` 和 `std::functions` 这三个函数可以组合得非常好。`std::bind`、`std::bind_front` 或 `std::bind_back` 能够动态地创建新的函数对象，而 `std::function` 接受这些临时函数对象并将它们绑定到一个变量。

`std::bind`, `std::bind_front`, `std::bind_back` 和 `std::function` 常常是多余的

`std::bind` 和 `std::function`, 是 TR1, `std::bind_front` 和 `std::bind_back` 的一部分，在 C++ 中大多是不必要的。首先，可以使用 Lambda 来代替 `std::bind`、`std::bind_front`、`std::bind_back`; 其次，可以经常使用 `auto` 来代替 `std::function` 进行自动类型推断。

`std::bind`

函数 `std::bind` 比 `std::bind_front` 或 `std::bind_back` 更强大，因为 `std::bind` 允许将参数绑定到任意位置。`std::bind`、`std::bind_front` 和 `std::bind_back` 能够动态地创建新的函数对象，而 `std::function` 接受这些临时函数对象并将它们绑定到一个变量。这些函数是函数式编程的强大工具，并且需要头文件 `<functional>`。

创建和绑定函数对象

```
1 // bindAndFunction.cpp
2 ...
3 #include <functional>
4 ...
5 // for placeholder _1 and _2
6 using namespace std::placeholders;
7
8 using std::bind;
9 using std::bind_front;
10 using std::bind_back;
11 using std::function
12 ...
13 double divMe(double a, double b){ return a/b; }
14
15 function<double(double, double)> myDiv1 = bind(divMe, _1, _2); // 200
16 function<double(double)> myDiv2 = bind(divMe, 2000, _1); // 200
17 function<double(double)> myDiv3 = bind_front(divMe, 2000); // 200
18 function<double(double)> myDiv4 = bind_back(divMe, 10); // 200
```

`std::bind`可以用多种方式创建函数对象:

- 将参数绑定到任意位置，
- 改变参数的顺序，

- 为参数引入占位符，
- 对函数部分求值，
- 调用新创建的函数对象，在 STL 算法中使用它们，或者将其存储在 std::function 中。

std::bind_front (C++20)

`std::bind_front` 从可调用对象创建可调用包装器。调用 `std::bind_front(func, arg…)` 将所有参数 `arg` 绑定到 `func` 的前端，并返回一个可调用的包装器。

std::bind_back (C++23)

`std::bind_back` 从可调用对象创建可调用包装器。调用 `std::bind_back(func, arg…)` 将所有参数 `arg` 绑定到 `func` 的后面，并返回一个可调用的包装器。

std::function

`std::function` 可以在变量中存储可调用对象，是一个多态函数包装器。可调用对象可以是 Lambda 函数、函数对象或函数。若必须显式指定可调用对象的类型，则 `std::function` 是必需的，不能用 `auto` 取代。

使用 `std::function` 的调度表

```

1 // dispatchTable.cpp
2 ...
3 #include <functional>
4 ...
5 using std::make_pair;
6 using std::map;
7
8 map<const char, std::function<double(double, double)>> tab;
9 tab.insert(make_pair('+', [](double a, double b){ return a + b; }));
10 tab.insert(make_pair('-', [](double a, double b){ return a - b; }));
11 tab.insert(make_pair('*', [](double a, double b){ return a * b; }));
12 tab.insert(make_pair('/', [](double a, double b){ return a / b; }));
13
14 std::cout << tab['+'](3.5, 4.5); // 8
15 std::cout << tab['-'](3.5, 4.5); // -1
16 std::cout << tab['*'](3.5, 4.5); // 15.75
17 std::cout << tab['/'](3.5, 4.5); // 0.777778

```

函数的类型形参定义了 `std::function` 接受的可调用对象的类型。

返回类型和参数类型

函数类型	返回类型	参数类型
double(double, double)	double	double
int()	int	
double(int, double)	double	int, double
void()		

2.3. 数值对

可以使用 `std::pair` 构建任意类型的数值对。类模板 `std::pair` 需要头文件 `<utility>`。`std::pair` 有一个默认的复制和移动构造函数，其对象可以交换：`std::swap(pair1, pair2)`。

C++ 库中经常使用 `std::pair`。例如，函数 `std::minmax` 以对的形式返回结果，关联容器 `std::map`、`std::unordered_map`、`std::multimap` 和 `std::unordered_multimap` 以对的方式管理它们的键/值关联性。

要获取 `pair p` 中的元素，可以直接访问，也可以通过索引访问。使用 `p.first` 或 `std::get<0>(p)`，将获得第一个元素；使用 `p.second` 或 `std::get<1>(p)`，将获得第二个元素。

数值对支持比较操作符 `==`、`!=`、`<`、`>`、`<=` 和 `>=`。当比较两个数值对时，成员 `pair1.first` 和 `pair2.first` 先比较，然后是 `pair1.second` 和 `pair2.second` 再比较。同样的策略也适用于其他比较操作符。

std::make_pair

C++ 有一个辅助函数 `std::make_pair`，可以在不指定类型的情况下生成数值对。`std::make_pair` 会自动推导出参数的类型。

辅助函数 `std::make_pair`

```

1 // pair.cpp
2 ...
3 #include <utility>
4 ...
5 using namespace std;
6 ...
7 pair<const char*, double> charDoub("str", 3.14);
8 pair<const char*, double> charDoub2= make_pair("str", 3.14);
9 auto charDoub3= make_pair("str", 3.14);
10
11 cout << charDoub.first << ", " << charDoub.second; // str, 3.14
12 charDoub.first="Str";
13 get<1>(charDoub)= 4.14;
14 cout << charDoub.first << ", " << charDoub.second; // Str, 4.14

```

2.4. 元组

使用 `std::tuple` 可以创建任意长度和类型的元组。类模板需要头文件 `<tuple>`。`std::tuple` 是 `std::pair` 的泛化形式。包含两个元素和对的元组可以相互转换。与 `std::pair` 一样，元组也有一个默认构造函

数、一个复制构造函数和一个移动构造函数。

可以使用 std::swap 函数交换元组。std::get 可以访问元组的第 i 个元素:std::get<i-1>(t)。通过 std::get<type>(t)，可以直接引用类型 type 的元素。

元组支持比较操作符 ==、!=、<、>、<= 和 >=。若比较两个元组，则按字典顺序比较元组的元素。比较从索引 0 开始。

std::make_tuple

辅助函数 std::make_tuple 提供了一种创建元组的方便方法。不必指定类型，编译器会自动推导出来。

辅助函数 std::make_tuple

```
1 // tuple.cpp
2 ...
3 #include <tuple>
4 ...
5 using std::get;
6
7 std::tuple<std::string, int, float> tup1("first", 3, 4.17f);
8 auto tup2 = std::make_tuple("second", 4, 1.1);
9
10 std::cout << get<0>(tup1) << ", " << get<1>(tup1) << ", "
11     << get<2>(tup1) << '\n'; // first, 3, 4.17
12 std::cout << get<0>(tup2) << ", " << get<1>(tup2) << ", "
13     << get<2>(tup2) << '\n'; // second, 4, 1.1
14 std::cout << (tup1 < tup2) << '\n'; // true
15
16 get<0>(tup2) = "Second";
17 std::cout << get<0>(tup2) << ", " << get<1>(tup2) << ", "
18     << get<2>(tup2) << '\n'; // Second, 4, 1.1
19 std::cout << (tup1 < tup2) << '\n'; // false
20
21 auto pair = std::make_pair(1, true);
22 std::tuple<int, bool> tup = pair;
```

std::tie 和 std::ignore

std::tie 允许创建引用变量的元组。可以使用 std::ignore 显式忽略元组元素。

辅助函数 std::tie 和 std::ignore

```
1 // tupleTie.cpp
2 ...
3 #include <tuple>
```

```

4   ...
5   using namespace std;
6
7   int first= 1;
8   int second= 2;
9   int third= 3;
10
11  int fourth= 4;
12  cout << first << " " << second << " "
13    << third << " " << fourth << endl; // 1 2 3 4
14
15  auto tup= tie(first, second, third, fourth) // bind the tuple
16    = std::make_tuple(101, 102, 103, 104); // create the tuple
17    // and assign it
18
19  cout << get<0>(tup) << " " << get<1>(tup) << " " << get<2>(tup)
20    << " " << get<3>(tup) << endl; // 101 102 103 104
21  cout << first << " " << second << " " << third << " "
22    << fourth << endl; // 101 102 103 104
23
24  first= 201;
25  get<1>(tup)= 202;
26  cout << get<0>(tup) << " " << get<1>(tup) << " " << get<2>(tup)
27    << " " << get<3>(tup) << endl; // 201 202 103 104
28  cout << first << " " << second << " " << third << " "
29    << fourth << endl; // 201 202 103 104
30
31  int a, b;
32  tie(std::ignore, a, std::ignore, b)= tup;
33  cout << a << " " << b << endl; // 202 104

```

2.5. 引用包装器

引用包装器是类型为引用的复制构造函数和复制赋值包装器，其在头文件`<functional>`中定义，其行为类似于引用，但可以复制的对象。与引用相反，`std::reference_wrapper`对象支持两个额外的用例：

- 可以在标准模板库的容器中使用它们。`std::vector<std::reference_wrapper<int>> myIntRefVector`
- 可以复制具有`std::reference_wrapper`对象的类的实例，但这不适用于引用。

`get`成员函数允许访问引用`:myInt.get()`，可以使用引用包装器来封装和调用可调用对象。

引用包装器

```

1 // referenceWrapperCallable.cpp
2 ...
3 #include <functional>
4 ...

```

```

5  void foo() {
6      std::cout << "Invoked" << '\n';
7  }
8
9  typedef void callableUnit();
10 std::reference_wrapper<callableUnit> refWrap(foo);
11
12 refWrap(); // Invoked

```

std::ref 和 std:: cref

使用辅助函数**std::ref**和**std:: cref**，可以轻松地为变量创建引用包装器。**std::ref**将创建一个非常量引用包装器，**std:: cref**将创建一个常量引用包装器。

辅助函数 *std::ref* 和 *std:: cref*

```

1 // referenceWrapperRefCref.cpp
2 ...
3 #include <functional>
4 ...
5 void invokeMe(const std::string& s) {
6     std::cout << s << ": const " << '\n';
7 }
8
9 template <typename T>
10 void doubleMe(T t) {
11     t *= 2;
12 }
13
14 std::string s("string");
15
16 invokeMe(std::cref(s)); // string
17
18 int i = 1;
19 std::cout << i << '\n'; // 1
20
21 doubleMe(i);
22 std::cout << i << '\n'; // 1S
23
24 doubleMe(std::ref(i));
25 std::cout << i << '\n'; // 2

```

可以使用**std::string**的常量引用和**std:: cref(s)**封装的非常量**std::string**来调用**invokeMe**函数。当在辅助函数**std::ref**中包装变量*i*时，将使用引用调用函数模板**doubleMe**，变量*i*加倍。

2.6. 智能指针

智能指针对于 C++ 来说必不可少，其能在 C++ 中实现显式的内存管理。除了已弃用的 std::auto_ptr 之外，C++ 还提供了三个智能指针，在头文件 <memory> 中定义。

首先，std::unique_ptr 对独占所有权的概念进行了建模。其次，std::shared_ptr 对共享所有权的概念进行了建模。std::weak_ptr 不是智能指针，因为它有一个有限的接口，其作用是打破 std::shared_ptr 的循环引用，模拟了临时所有权的概念。

智能指针根据 RAII 习惯用法管理资源，若智能指针超出作用域，资源将自动释放。

资源获取即初始化

资源获取即初始化 (Resource Acquisition Is Initialization，简称 RAII) 是 C++ 中的一种流行技术，其中资源获取和释放与对象的生命周期绑定在一起。对于智能指针，内存在构造函数中分配，在析构函数中释放。在 C++ 中，当对象超出作用域时自动调用析构函数。

智能指针概述

名称	标准	描述
std::auto_ptr (已废弃)	C++98	独占资源。移动资源，然后复制。
std::unique_ptr	C++11	独占资源，不能复制。
std::shared_ptr	C++11	具有共享变量的引用计数器，自动管理引用计数器。若引用计数器为 0，则删除资源。
std::weak_ptr	C++11	帮助打破 std::shared_ptr 的循环引用。不修改引用计数器。

std::unique_ptr

std::unique_ptr 负责管理其资源。若超出作用域，会自动释放资源。若没有使用复制语义，可以在标准模板库的容器和算法中使用 std::unique_ptr。若不使用特殊的删除器，std::unique_ptr 会与原始指针一样廉价和快速。

不要使用 std::auto_ptr

经典的 C++03 有一个智能指针 std::auto_ptr，负责资源的生命周期。但是 std::auto_ptr 有一个概念问题。若隐式或显式复制 std::auto_ptr，则可能会移动资源。没有复制语义，使用隐式移动语义，经常会有未定义行为。因此 std::auto_ptr 在 C++11 中已弃用，应该使用 std::unique_ptr。既不能隐式地也不能显式地复制 std::unique_ptr，只能移动它：

```

1 #include <memory>
2 ...
3 std::auto_ptr<int> ap1(new int(2011));
4 std::auto_ptr<int> ap2 = ap1; // OK
5
6 std::unique_ptr<int> up1(new int(2011));
7 std::unique_ptr<int> up2 = up1; // ERROR
8 std::unique_ptr<int> up3 = std::move(up1); // OK

```

这些都是 `std::unique_ptr` 的成员函数。

`std::unique_ptr` 的成员函数

名称	描述
<code>get</code>	返回指向资源的指针。
<code>get_deleter</code>	返回 <code>delete</code> 函数。
<code>release</code>	返回指向资源的指针并释放它。
<code>reset</code>	重置资源。
<code>swap</code>	交换资源。

下面的代码段中，可以看到这些成员函数的具体使用情况：

The `std::unique_ptr`

```

1 // uniquePtr.cpp
2 ...
3 #include <utility>
4 ...
5 using namespace std;
6
7 struct MyInt{
8     MyInt(int i):i_(i){}
9     ~MyInt(){
10         cout << "Good bye from " << i_ << endl;
11     }
12     int i_;
13 };
14
15 unique_ptr<MyInt> uniquePrt1{new MyInt(1998)};
16 cout << uniquePrt1.get() << endl; // 0x15b5010
17
18 unique_ptr<MyInt> uniquePrt2{move(uniquePrt1)};
19 cout << uniquePrt1.get() << endl; // 0
20 cout << uniquePrt2.get() << endl; // 0x15b5010
21 {

```

```

22     unique_ptr<MyInt> localPtr{new MyInt(2003)};
23 } // Good bye from 2003
24 uniquePtr2.reset(new MyInt(2011)); // Good bye from 1998
25 MyInt* myInt= uniquePtr2.release();
26 delete myInt; // Good by from 2011
27
28 unique_ptr<MyInt> uniquePtr3{new MyInt(2017)};
29 unique_ptr<MyInt> uniquePtr4{new MyInt(2022)};
30 cout << uniquePtr3.get() << endl; // 0x15b5030
31 cout << uniquePtr4.get() << endl; // 0x15b5010
32
33 swap(uniquePtr3, uniquePtr4);
34 cout << uniquePtr3.get() << endl; // 0x15b5010
35 cout << uniquePtr4.get() << endl; // 0x15b5030

```

std::unique_ptr 对数组有一个特化:

std::unique_ptr array

```

1 // uniquePtrArray.cpp
2 ...
3 #include <memory>
4 ...
5 using namespace std;
6
7 class MyStruct{
8     public:
9         MyStruct():val(count){
10             cout << (void*)this << " Hello: " << val << endl;
11             MyStruct::count++;
12         }
13         MyStruct(){
14             cout << (void*)this << " Good Bye: " << val << endl;
15             MyStruct::count--;
16         }
17     private:
18         int val;
19         static int count;
20     };
21
22 int MyStruct::count= 0;
23 ...
24 {
25     // generates a myUniqueArray with three `MyStructs`
26     unique_ptr<MyStruct[]> myUniqueArray{new MyStruct[3]};
27 }
28 // 0x1200018 Hello: 0
29 // 0x120001c Hello: 1
30 // 0x1200020 Hello: 2

```

```
31 // 0x1200020 GoodBye: 2  
32 // 0x120001c GoodBye: 1  
33 // 0x1200018 GoodBye: 0
```

特殊的删除器

`std::unique_ptr` 可以用特殊的删除器:`std::unique_ptr<int, MyIntDeleter>` `up(new int(2011), MyIntDeleter())`。默认情况下，`std::unique_ptr` 使用相应资源的删除器。

`std::make_unique`

辅助函数[std::make_unique](#)与 C++11 标准中被遗忘的 `std::make_shared` 不同，`std::make_unique` 在 C++14 标准中添加。`std::make_unique` 能够在一个步骤中创建 `std::unique_ptr`: `std::unique_ptr<int> up = std::make_unique<int>(2014)`。

`std::shared_ptr`

`std::shared_ptr` 共享资源的所有权，其有两个句柄：一个用于资源，一个用于引用计数器。通过复制 `std::shared_ptr`，引用计数增加 1。若 `std::shared_ptr` 超出作用域，则减少 1。若引用计数器的值变为 0，并且不再有 `std::shared_ptr` 引用该资源，则 C++ 运行时自动释放该资源。资源的释放恰好发生在最后一个 `std::shared_ptr` 超出作用域的时候。C++ 运行时保证引用计数器的调用是一个原子操作。由于这种管理开销，`std::shared_ptr` 比原始指针或 `std::unique_ptr` 需要更多的时间和内存。

下表是 `std::shared_ptr` 的成员函数。

`std::shared_ptr` 的成员函数

名称	描述
<code>get</code>	返回指向资源的指针。
<code>get_deleter</code>	返回 <code>delete</code> 函数。
<code>reset</code>	重置资源
<code>swap</code>	交换资源
<code>unique</code>	检查 <code>std::shared_ptr</code> 是否是资源的独占所有者
<code>use_count</code>	返回引用计数器的值。

`std::make_shared`

辅助函数[std::make_shared](#)创建资源，并在 `std::shared_ptr` 中返回它。应该选择 `std::make_shared`，而非直接创建 `std::shared_ptr`，因为 `std::make_shared` 更快。

下面的代码示例显示了 `std::shared_ptr` 的典型用例。

`std::shared_ptr`

```
1 // sharedPtr.cpp
2 ...
3 #include <memory>
4 ...
5 class MyInt{
6 public:
7     MyInt(int v):val(v){
8         std::cout << "Hello: " << val << '\n';
9     } ~
10    MyInt() {
11        std::cout << "Good Bye: " << val << '\n';
12    }
13 private:
14     int val;
15 };
16
17 auto sharPtr= std::make_shared<MyInt>(1998); // Hello: 1998
18 std::cout << sharPtr.use_count() << '\n'; // 1
19
20 {
21     std::shared_ptr<MyInt> locSharPtr(sharPtr);
22     std::cout << locSharPtr.use_count() << '\n'; // 2
23 }
24 std::cout << sharPtr.use_count() << '\n'; // 1
25
26 std::shared_ptr<MyInt> globSharPtr= sharPtr;
27 std::cout << sharPtr.use_count() << '\n'; // 2
28
29 globSharPtr.reset();
30 std::cout << sharPtr.use_count() << '\n'; // 1
31 sharPtr= std::shared_ptr<MyInt>(new MyInt(2011)); // Hello:2011
32                                     // Good Bye: 1998
33 ...
34 // Good Bye: 2011
```

本例中，可调用对象是一个函数对象，可以很容易地计算创建了多少个类的实例。结果在静态变量 `count` 中。

`std::shared_ptr from this`

必须从 `std::enable_shared_from_this` 派生类 `public`，可以使用类`std::enable_shared_from_this`对象进行创建，该类返回自身的 `std::shared_ptr`。类支持成员函数 `shared_from_this` 返回 `std::shared_ptr` 到 `this`:

std::shared_ptr from this

```
1 // enableShared.cpp
2 ...
3 #include <memory>
4 ...
5 class ShareMe: public std::enable_shared_from_this<ShareMe>{
6     std::shared_ptr<ShareMe> getShared(){
7         return shared_from_this();
8     }
9 };
10
11 std::shared_ptr<ShareMe> shareMe(new ShareMe);
12 std::shared_ptr<ShareMe> shareMe1= shareMe->getShared();
13
14 std::cout << (void*)shareMe.get() << '\n'; // 0x152d010
15 std::cout << (void*)shareMe1.get() << '\n'; // 0x152d010
16 std::cout << shareMe.use_count() << '\n'; // 2
```

从代码示例中可以看到，`get` 成员函数引用同一个对象。

std::weak_ptr

`std::weak_ptr` 不是智能指针。`std::weak_ptr` 不支持对资源的透明访问，它只是从 `std::shared_ptr` 中借用资源。`std::weak_ptr` 并不改变引用计数器：

std::weak_ptr

```
1 // weakPtr.cpp
2 ...
3 #include <memory>
4 ...
5 auto sharedPtr= std::make_shared<int>(2011);
6 std::weak_ptr<int> weakPtr(sharedPtr);
7
8 std::cout << weakPtr.use_count() << '\n'; // 1
9 std::cout << sharedPtr.use_count() << '\n'; // 1
10
11 std::cout << weakPtr.expired() << '\n'; // false
12 if( std::shared_ptr<int> sharedPtr1= weakPtr.lock() ) {
13     std::cout << *sharedPtr << '\n'; // 2011
14 }
15 else{
16     std::cout << "Don't get it!" << '\n';
17 }
18
19 weakPtr.reset();
```

```

21 if( std::shared_ptr<int> sharedPtr1= weakPtr.lock() ) {
22     std::cout << *sharedPtr << '\n';
23 }
24 else{
25     std::cout << "Don't get it!" << '\n'; // Don't get it!
26 }

```

该表概述了 std::weak_ptr 的成员函数。

std::weak_ptr 的成员函数。

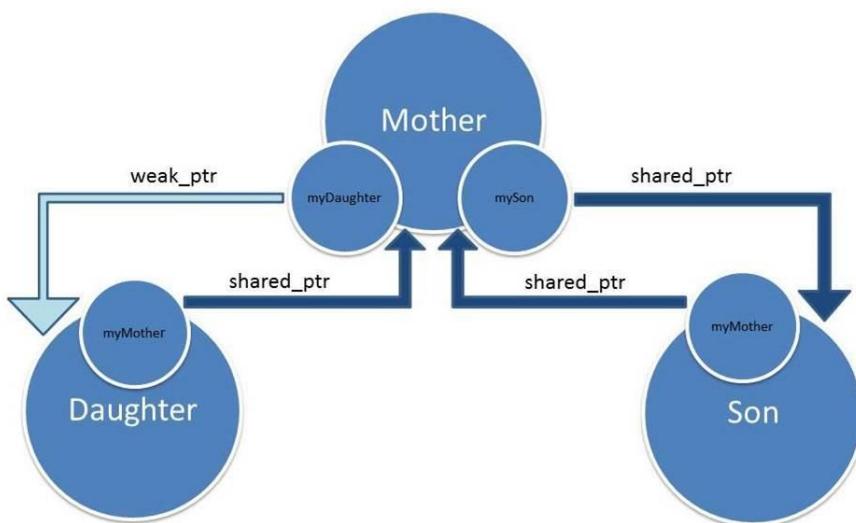
名称	描述
expired	检查资源是否删除。
lock	以现有的资源创建一个 std::shared_ptr
reset	重置资源
swap	交换资源
use_count	返回引用计数器的值。

std::weak_ptr 存在有一个原因，打破了 std::shared_ptr 的循环引用。

循环引用

若 std::shared_ptr 相互引用，则得到它们的循环引用，资源计数器永远不会变为 0，资源也不会自动释放。若在循环中嵌入一个 std::weak_ptr，就可以打破这个循环，因为 std::weak_ptr 不修改引用计数器。

代码示例的结果是，daughter 自动释放，但 son 和 mother 都没有释放。mother 通过 std::shared_ptr 指代 son，通过 std::weak_ptr 指代 daughter。图中看到代码的结构会帮助读者对此关系进行理解。



循环引用

循环引用

```
1 // cyclicReference.cpp
2 ...
3 #include <memory>
4 ...
5 using namespace std;
6
7 struct Son, Daughter;
8
9 struct Mother{
10     ~Mother() {cout << "Mother gone" << endl;}
11     void setSon(const shared_ptr<Son> s) {mySon= s;}
12     void setDaughter(const shared_ptr<Daughter> d) {myDaughter= d;}
13     shared_ptr<const Son> mySon;
14     weak_ptr<const Daughter> myDaughter;
15 };
16
17 struct Son{
18     Son(shared_ptr<Mother> m):myMother(m){}
19     ~Son() {cout << "Son gone" << endl;}
20     shared_ptr<const Mother> myMother;
21 };
22
23 struct Daughter{
24     Daughter(shared_ptr<Mother> m):myMother(m){}
25     ~Daughter() {cout << "Daughter gone" << endl;}
26     shared_ptr<const Mother> myMother;
27 };
28
29 {
30     shared_ptr<Mother> mother= shared_ptr<Mother>(new Mother);
31     shared_ptr<Son> son= shared_ptr<Son>(new Son(mother));
32     shared_ptr<Daughter> daugh= shared_ptr<Daughter>(new Daughter(mother));
33     mother->setSon(son);
34     mother->setDaughter(daugh);
35 }
36 // Daughter gone
```

2.7. 类型特性

类型特征库能够在编译时检查、比较和修改类型，所以在运行时上没有开销。使用类型特征库有两个原因：优化和正确性。优化是因为类型特征库的自省功能，可使自动选择更快的代码成为可能。正确性，可以指定在编译时检查代码的要求。

类型特征库和 static_assert 是一对强大的组合

一方面，类型特征库的函数在编译时提供类型信息；另一方面， static_assert 函数在编译时检查给定的信息。这些检查在运行时的开销。

```
1 #include <type_traits>
2 ...
3 template <typename T> T fac(T a) {
4     static_assert(std::is_integral<T>::value, "T not integral");
5     ...
6 }
7 fac(10);
8 fac(10.1); // with T= double; T not integral
```

GCC 编译器退出函数调用 fac(10.1)。编译时的消息是， T 是 double 类型，因此不是整型。

检查类型信息

使用类型特征库，可以检查基本类型和复合类型类别。属性值可提供结果。

主要类型类别

有 14 种不同的类型类别，是完全且不重叠的，每个类型只是一个类型类别的成员。若为类型检查类型类别，则请求独立于 const 或 volatile 限定符。

```
1 template <class T> struct is_void;
2 template <class T> struct is_integral;
3 template <class T> struct is_floating_point;
4 template <class T> struct is_array;
5 template <class T> struct is_pointer;
6 template <class T> struct is_null_pointer;
7 template <class T> struct is_member_object_pointer;
8 template <class T> struct is_member_function_pointer;
9 template <class T> struct is_enum;
10 template <class T> struct is_union;
11 template <class T> struct is_class;
12 template <class T> struct is_function;
13 template <class T> struct is_lvalue_reference;
14 template <class T> struct is_rvalue_reference;
```

下面的代码示例显示了所有主要类型类别。

所有主要类型类别

```
1 // typeCategories.cpp
2 ...
```

```

3 #include <type_traits>
4 using std::cout;
5
6 cout << std::is_void<void>::value; // true
7 cout << std::is_integral<short>::value; // true
8 cout << std::is_floating_point<double>::value; // true
9 cout << std::is_array<int [] >::value; // true
10 cout << std::is_pointer<int*>::value; // true
11 cout << std::is_reference<int&>::value; // true
12
13 struct A{
14     int a;
15     int f(int){ return 2011; }
16 };
17 cout << std::is_member_object_pointer<int A::*>::value; // true
18 cout << std::is_member_function_pointer<int (A::*)(int)>::value; // true
19
20 enum E{
21     e= 1,
22 };
23 cout << std::is_enum<E>::value; // true
24
25 union U{
26     int u;
27 };
28
29 cout << std::is_union<U>::value; // true
30
31 cout << std::is_class<std::string>::value; // true
32 cout << std::is_function<int * (double)>::value; // true
33 cout << std::is_lvalue_reference<int&>::value; // true
34 cout << std::is_rvalue_reference<int&&>::value; // true

```

复合类型类别

在 14 个主要类型类别的基础上，有 7 个复合类型类别。

复合类型类别

复合类型类别	主要类型类别
is_arithmetic	is_floating_point 或 is_integral
is_fundamental	is_arithmetic 或 is_void
is_object	is_arithmetic 或 is_enum 或 is_pointer 或 is_member_pointer
is_reference	is_lvalue_reference 或 is_rvalue_reference

is_compound	is_fundamental 的补充
is_member_pointer	is_member_object_pointer 或 is_member_function_pointer
is_scalar	is_arithmetic 或 is_enum 或 is_pointer 或 is_member_pointer 或 is_null_pointer

类型属性

除了主要和复合类型类别之外，还有许多类型属性。

```

1 template <class T> struct is_const;
2 template <class T> struct is_volatile;
3 template <class T> struct is_trivial;
4 template <class T> struct is_trivially_copyable;
5 template <class T> struct is_standard_layout;
6 template <class T> struct has_unique_object_represenation;
7 template <class T> struct is_empty;
8 template <class T> struct is_polymorphic;
9 template <class T> struct is_abstract;
10 template <class T> struct is_final;
11 template <class T> struct is_aggregate;
12 template <class T> struct is_implicit_lifetime;
13
14 template <class T> struct is_signed;
15 template <class T> struct is_unsigned;
16
17 template <class T> struct is_bounded_array;
18 template <class T> struct is_unbounded_array;
19 template <class T> struct is_scoped_enum;
20
21 template <class T, class... Args> struct is_constructible;
22 template <class T> struct is_default_constructible;
23 template <class T> struct is_copy_constructible;
24 template <class T> struct is_move_constructible;
25
26 template <class T, class U> struct is_assignable;
27 template <class T> struct is_copy_assignable;
28 template <class T> struct is_move_assignable;
29 template <class T> struct is_destructible;
30 template <class T, class... Args> struct is_trivially_constructible;
31 template <class T> struct is_trivially_default_constructible;
32 template <class T> struct is_trivially_copy_constructible;
33 template <class T> struct is_trivially_move_constructible;
34 template <class T, class U> struct is_triviallyAssignable;
35 template <class T> struct is_trivially_copyAssignable;
36 template <class T> struct is_trivially_moveAssignable;
37
38 template <class T> struct is_trivially_destructible;
```

```

39
40 template <class T, class... Args> struct is_nothrow_constructible;
41 template <class T> struct is_nothrow_default_constructible;
42 template <class T> struct is_nothrow_copy_constructible;
43 template <class T> struct is_nothrow_move_constructible;
44
45 template <class T, class U> struct is_nothrowAssignable;
46
47 template <class T> struct is_nothrow_copyAssignable;
48 template <class T> struct is_nothrow_moveAssignable;
49
50 template <class T> struct is_nothrow_destructible;
51 template <class T> struct has_virtual_destructor;
52
53 template <class T> struct is_swappableWith;
54 template <class T> struct is_swappable;
55 template <class T> struct is_nothrow_swappableWith;
56 template <class T> struct is_nothrow_swappable;

```

类型关系

该库支持各种类型关系

类型关系

函数	描述
template <class Base, class Derived> struct is_base_of	检查 Derived 是否从 Base 派生。
template <class From, class To> struct is_convertible struct is_nothrow_convertible	检查 From 是否可以转换为 To。
template <class T, class U> struct is_same	检查类型 T 和 U 是否相同
template <class T, class U> struct is_layout_compatible	检查类型 T 和 U 是否为布局兼容。
template <class Base, class Derived> struct is_pointer_interconvertible_base_of	检查一个类型是否是另一个类型的指针可互换的基型。
template <class Fn, class ...ArgTypes> struct is_invocable struct is_invocable_r struct is_nothrow_invocable struct is_nothrow_invocable_r	检查是否可以使用给定类型调用类型。

类型修改

类型特征库允许在编译期间更改类型，可以修改类型的常量：

类型修改

```
1 // typeTraitsModifications.cpp
2 ...
3 #include <type_traits>
4 ...
5 using namespace std;
6
7 cout << is_const<int>::value; // false
8 cout << is_const<const int>::value; // true
9 cout << is_const<add_const<int>::type>::value; // true
10
11 typedef add_const<int>::type myConstInt;
12 cout << is_const<myConstInt>::value; // true
13
14 typedef const int myConstInt2;
15 cout << is_same<myConstInt, myConstInt2>::value; // true
16
17 cout << is_same<int, remove_const<add_const<int>::type>::type>::value; // true
18 cout << is_same<const int, add_const<add_const<int>::type>::type>::value; // true
```

函数 `std::add_const` 会将 `const` 添加到类型中，而 `std::remove_const` 用来删除 `const`。

类型特征库中还有更多可用的函数，可以修改类型的 `const-volatile` 属性。

```
1 template <class T> struct remove_const;
2 template <class T> struct remove_volatile;
3 template <class T> struct remove_cv;
4
5 template <class T> struct add_const;
6 template <class T> struct add_volatile;
7 template <class T> struct add_cv;
```

可以在编译时更改符号，

```
1 template <class T> struct make_signed;
2 template <class T> struct make_unsigned;
```

或类型的引用或指针属性。

```
1 template <class T> struct remove_reference;
2 template <class T> struct remove_cvref;
3 template <class T> struct add_lvalue_reference;
4 template <class T> struct add_rvalue_reference;
5
6 template <class T> struct remove_pointer;
7 template <class T> struct add_pointer;
```

以下函数对于编写泛型库非常重要。

```
1 template <class B> struct enable_if;
2 template <class B, class T, class F> struct conditional;
3 template <class... T> common_type;
4 template <class... T> common_reference;
5 template <class... T> basic_common_reference;
6 template <class... T> void_t;
7 template <class... T> type_identity;
```

可以使用 std::enable_if 有条件地隐藏函数重载或模板特化，以避免重载解析。条件 std::conditional 在编译时提供了三元操作符，而 std::common_type 提供了所有类型中的公共类型。std::common_type、std::common_reference、std::basic_common_reference、std::void_t、std::type_identity 为可变参数模板，所以类型参数的数量可以任意。

操作特征

函数 std::connection、std::disjunction 和 std::negation 支持类型特征函数的逻辑组合，其是可变参数模板。

C++ 有::type 和::value 的简写

若想从 int 类型得到 const int 类型，必须要求类型为 std::add_const<int>::type。对于 C++14 标准，可使用 std::add_const_t<int>，而不是冗长的形式 std::add_const<int>::type。此规则适用于所有类型特征函数。

因此，C++17 中，可以使用简写 std::is_integral_v<T> 来表示谓词 std::is_integral<T>::value。

成员关系

函数 std::is_pointer_interconvertible_with_class 检查一个类型的对象是否与该类型的指定子对象进行了指针互转换，函数 std::is_corresponding_member 检查两个指定成员在两个指定类型的公共初学者序列中是否相互对应。

const 求值

`std::is_constant_evaluated` 可以检测函数调用是否在编译时发生。

检测在编译时是否发生了函数调用

```
1 // constantEvaluated.cpp
2 #include <type_traits>
3 ...
4
5 constexpr double power(double b, int x) {
6     if (std::is_constant_evaluated() && !(b == 0.0 && x < 0)) {
7         if (x == 0)
8             return 1.0;
9         double r = 1.0, p = x > 0 ? b : 1.0 / b;
10        auto u = unsigned(x > 0 ? x : -x);
11        while (u != 0) {
12            if (u & 1) r *= p;
13            u /= 2;
14            p *= p;
15        }
16        return r;
17    }
18    else {
19        return std::pow(b, double(x));
20    }
21 }
22
23 constexpr double kilo1 = power(10.0, 3); // execution at compile time
24
25 int n = 3;
26 double kilo2 = power(10.0, n); // execution at runtime
27 std::cout << "kilo2: " << kilo2 << '\n';
```

2.8. 时间库

时间库主要由时间点、时间段和时钟三个部分组成。该库还提供了时间功能、日历支持、时区支持，以及输入和输出支持。

时间点

时间点由起始点、所谓的历元和附加的时间段来定义。

时间段

时间段是两个时间点之间的差值，时钟周期数定义了这个值。

时钟

时钟由起始点(历元)和刻度组成，这样就可以计算出当前的时间点。

日期

从午夜开始的时间分为小时: 分钟: 秒。

日历

表示各种日历日期，如年、月、工作日或一周的第 n 天。

时区

表示特定于某个地理区域的时间。

时间库是多线程的关键组件

时间库是 C++ 新多线程功能的关键组件，可以将当前线程通过 std::this_thread::sleep_for(std::chrono::milliseconds(15)) 设置为 15 毫秒的睡眠状态，或者尝试获取 2 分钟的锁:lock。try_lock_until(now + std::chrono::minutes(2))。

时间点

时间段是由一段时间组成，定义为某个时间单位的时钟周期数。时间点由时钟和持续时间组成。这个持续时间可以是正的，也可以是负的。

```
1 template <class Clock, class Duration= typename Clock::duration> class time_point;
```

时钟 std::chrono::steady_clock、std::chrono::high_resolution_clock 和 std::chrono::system 没有定义历元。但是在主流平台上，std::chrono::system 的 epoch 通常定义为 1.1.1970。可以计算自 1970 年 1 月 1 日以来的时间，其分辨率为纳秒、秒和分钟。

自纪元以来的时间

```
1 // epoch.cpp
2 ...
3 #include <chrono>
4 ...
5 auto timeNow= std::chrono::system_clock::now();
6 auto duration= timeNow.time_since_epoch();
7 std::cout << duration.count() << "ns" // 1413019260846652ns
8
9 typedef std::chrono::duration<double> MySecondTick;
10 MySecondTick mySecond(duration);
11 std::cout << mySecond.count() << "s"; // 1413019260.846652s
12
13 const int minute= 60;
14 typedef std::chrono::duration<double, <minute>> MyMinuteTick;
15 MyMinuteTick myMinute(duration);
16 std::cout << myMinute.count() << "m"; // 23550324.920572m
```

由于函数 std::chrono::clock_cast，可以在不同的时钟之间转换时间点。

使用时间库进行简单的性能测试

性能测试

```
1 // performanceMeasurement.cpp
2 ...
3 #include <chrono>
4 ...
5 std::vector<int> myBigVec(10000000, 2011);
6 std::vector<int> myEmptyVec1;
7
8 auto begin= std::chrono::high_resolution_clock::now();
9 myEmptyVec1 = myBigVec;
10 auto end= std::chrono::high_resolution_clock::now() - begin;
11
12 auto timeInSeconds = std::chrono::duration<double>(end).count();
13 std::cout << timeInSeconds << '\n'; // 0.0150688800
```

时间段

时间段是两个时间点之间的差值，时间段以时钟周期数来衡量。

```
1 template <class Rep, class Period = ratio<1>> class duration;
```

若 Rep 是浮点数，则时间段支持小数。最重要的时间段在 chrono 库中定义：

```
1 typedef duration<signed int, nano> nanoseconds;
2 typedef duration<signed int, micro> microseconds;
3 typedef duration<signed int, milli> milliseconds;
4 typedef duration<signed int> seconds;
5 typedef duration<signed int, ratio< 60>> minutes;
6 typedef duration<signed int, ratio<3600>> hours;
```

时间段可以有多长？C++ 标准保证预定义的时间长度可以存储 +/-292 年。可以很容易地定义时间长度，就像一个德国学校的小时：`typedef std::chrono::duration<double, std::ratio<2700>> MyLessonTick`。以自然数表示的时间持续时间必须显式地转换为以浮点数表示的时间持续时间，该值将截断：

时间段

```
1 // duration.cpp
2 ...
```

```
3 #include <chrono>
4 #include <ratio>
5
6 using std::chrono;
7
8 typedef duration<long long, std::ratio<1>> MySecondTick;
9 MySecondTick aSecond(1);
10
11 milliseconds milli(aSecond);
12 std::cout << milli.count() << " milli"; // 1000 milli
13
14 seconds seconds(aSecond);
15 std::cout << seconds.count() << " sec"; // 1 sec
16
17 minutes minutes(duration_cast<minutes>(aSecond));
18 std::cout << minutes.count() << " min"; // 0 min
19
20 typedef duration<double, std::ratio<2700>> MyLessonTick;
21 MyLessonTick myLesson(aSecond);
22 std::cout << myLesson.count() << " less"; // 0.00037037 less
```

std::ratio

`std::ratio` 支持在编译时使用有理数进行算术运算。有理数有两个模板参数：分子和分母。C++11 预先定义了许多有理数。

C++14 为最常用的时间段提供了内置字面符。

用于时间持续时间的内置字面符

类型	后缀	举例
std::chrono::hours	h	5h
std::chrono::minutes	min	5min
std::chrono::seconds	s	5s
std::chrono::milliseconds	ms	5ms
std::chrono::microseconds	us	5us
std::chrono::nanoseconds	ns	5ns

时钟

时钟由时间起点和刻度组成，可以通过成员函数获得当前时间。

std::chrono::system_clock

系统时间，可与外部时钟同步。

std::chrono::steady_clock

时钟，无法调整。

std::chrono::high_resolution_clock

最精确的系统时间。

std::chrono::system_clock 通常指 1.1.1970，不能将 std::steady_clock 向前或向后调整为与其他两个时钟相反。成员函数 to_time_t 和 from_time_t 可用于在 std::chrono::system_clock 和 std::time_t 对象之间进行转换。

日期

std::chrono::time_of_day 将从午夜开始的持续时间分成小时: 分钟: 秒。函数 std::chrono::is_am 和 std::chrono::is_pm 检查时间是在正午之前 (子午线之前) 还是之后 (子午线之后)。

std::chrono::time_of_day 对象 tOfDay 支持各种成员函数。

std::chrono::time_of_day 的成员函数

成员函数	描述
tOfDay.hours()	返回自午夜开始的小时数。
tOfDay.minutes()	返回自午夜开始的分钟数。
tOfDay.seconds()	返回午夜之后的第二个分量。
tOfDay.subseconds()	返回从午夜开始的秒数 (小数)。
tOfDay.to_duration()	返回从午夜开始的时间。

<code>std::chrono::make12(hr)</code>	返回相当于 24 小时 (12 小时) 格式时间的 12 小时 (24 小时)。
<code>std::chrono::is_am(hr)</code> <code>std::chrono::is_pm(hr)</code>	检测 24 小时格式时间是上午还是下午。

日历

日历表示各种日期，例如年、月、工作日或一周的第 n 天。

当前时间

```

1 // currentTime.cpp
2 ...
3 #include <chrono>
4 using std::chrono;
5 ...
6 auto now = system_clock::now();
7 std::cout << "The current time is " << now << " UTC\n";
8
9 auto currentYear = year_month_day(floor<days>(now)).year();
10 std::cout << "The current year is " << currentYear << '\n';
11
12 auto h = floor<hours>(now) - sys_days(January/1/currentYear);
13 std::cout << "It has been " << h << " since New Years!\n";
14
15 std::cout << '\n';
16
17 auto birthOfChrist = year_month_weekday(sys_days(January/01/0000));
18 std::cout << "Weekday: " << birthOfChrist.weekday() << '\n';

```

程序的输出显示了与当前时间相关的信息。

```

The current time is 2020-07-18 20:39:12.356023527 UTC
The current year is 2020
It has been 4796h since New Years!

Weekday: Sat

```

下表给出了日历类型的概述。

各种日历类型

类型	描述
last_spec	表示一个月内的最后一天或最后一个工作日。
day	表示一个月中的一天。
month	表示一年中的一个月。
year	表示公历中的一年。
weekday	表示公历中一周中的一天。
weekday_indexed	表示一个月的第 n 个工作日
weekday_last	表示一个月的最后一个工作日
month_day	表示特定月份的特定日期。
month_day_last	表示特定月份的最后一天。
month_weekday	表示指定月份的第 n 个工作日。
month_weekday_last	表示特定月份的最后一个工作日。
year_month	表示特定年份的特定月份。
year_month_day	表示特定的年、月和日。
year_month_day_last	表示特定年份和月份的最后一天
year_month_weekday	表示特定的年、月和工作日
year_month_weekday_last	表示特定年份和月份的最后一个工作日。

时区

时区表示特定于一个地理区域的时间，下面的代码段显示了不同时区的本地时间。

显示不同时区的本地时间

```

1 // timezone.cpp
2 ...
3 #include <chrono>
4 using std::chrono;
5 ...
6 auto time = floor<milliseconds>(system_clock::now());
7 auto localTime = zoned_time<milliseconds>(current_zone(), time);
8 auto berlinTime = zoned_time<milliseconds>("Europe/Berlin", time);
9 auto newYorkTime = std::chrono::zoned_time<milliseconds>("America/New_York", time);
10 auto tokyoTime = zoned_time<milliseconds>("Asia/Tokyo", time);
11
12 std::cout << time << '\n';
13 std::cout << localTime << '\n';
14 std::cout << berlinTime << '\n';
15 std::cout << newYorkTime << '\n';
16 std::cout << tokyoTime << '\n';

```

时区功能支持访问[IANA 时区数据库](#)，支持不同时区的操作，并提供有关闰秒的信息。

下表给出了时区功能的概述。欲了解更多详细信息，请参阅cpprefere.com。

时区信息

类型	描述
tzdb	描述 IANA 时区数据库。
locate_zone	根据 time_zone 的名称定位时间。
current_zone	返回当前 time_zone。
time_zone	表示一个时区。
sys_info	返回特定时间点的时区信息。
local_info	表示从本地时间到 UNIX 时间的信息。
zoned_time	表示时区和时间点。
leap_second	包含有关插入闰秒的信息。

时间的 I/O

函数 std::chrono::parse 解析流中的 chrono 对象。

解析时间点和时区

```
1 std::istringstream inputStream("1999-10-31 01:30:00 -08:00 US/Pacific");
2
3 std::chrono::local_seconds timePoint;
4 std::string timeZone;
5 inputStream >> std::chrono::parse("%F %T %Ez %Z", timePoint, timeZone);
```

解析功能提供了各种格式说明符来处理一天中的时间和日历日期，如年、月、周和日。cppreference.com为格式说明符提供详细信息。

2.9. std::any, std::optional 和 std::variant

新的 C++17 数据类型 std::any、std::optional 和 std::variant 都基于[Boost 库](#)。

std::any

[std::any](#)为任何可复制构造类型的单个值的类型安全容器，该类型需要头文件 `<any>`。有几种方法可以创建 std::any 容器，可以使用各种构造函数或工厂函数 `std::make_any`。`any.emplace` 允许将一个值构造为任何值，`any.reset` 能够销毁所包含的对象。若想知道容器 std::any 是否有值，请使用成员函数 `any.has_value`，也可以通过 `any.type` 获得容器对象的类型 id。因为泛型函数 `std::any_cast`，可以访问所包含的对象。若指定了错误的类型，就会得到 `std::bad_any_cast` 异常。下面的代码片段显示了 std::any 的基本用法。

std::any

```
1 // any.cpp
2 ...
3 #include <any>
4
5 struct MyClass{};
6
7 ...
8
9 std::vector<std::any> anyVec{true, 2017, std::string("test"), 3.14, MyClass()};
10 std::cout << std::any_cast<bool>(anyVec[0]); // true
11 int myInt= std::any_cast<int>(anyVec[1]);
12 std::cout << myInt << '\n'; // 2017
13
14 std::cout << anyVec[0].type().name(); // b
15 std::cout << anyVec[1].type().name(); // i
```

这里定义了一个 `std::vector<std::any>`。要获得其元素，必须使用 `std::any_cast`。若使用了错误的类型，就会得到 `std::bad_any_cast` 异常。

typeid 的字符串表示形式

类型标识符的字符串表示形式是实现定义的。若 `anyVec[1]` 是 `int` 类型，则表达式 `anyVec[1].type().name()` 将使用[GCC C++ 编译器](#)返回 `i`，使用[Microsoft Visual C++ 编译器](#)返回 `int`。

`std::any` 可以拥有任意类型的对象；`std::optional` 可以有值，也可以没有值。

std::optional

`std::optional` 对于可能有结果的数据库查询等计算来说非常合适，这个类型需要头文件 `<optional>`。

不要定义“无结果”

C++17 之前，通常使用惟一值（如空指针、空字符串或惟一整数）来表示结果的缺失。对于类型系统，必须使用常规值（例如空字符串）来定义不规则值。这些唯一的值或没有结果很容易出错，因为必须用类型系统来检查返回值。

各种构造函数和方便的函数 `std::make_optional` 允许定义一个 `std::optional` 对象 `opt`，带或不带值。`opt.emplace` 将在原地构造包含的值，而 `opt.reset` 将销毁容器值。可以显式地询问 `std::optional` 容器是否有值，或者可以在逻辑表达式中检查它。`opt.value` 返回值，而 `opt.value_or` 返回值或默认值。若 `opt` 没有包含值，`opt.value` 将抛出 `std::bad_optional_access` 异常。

下面是一个使用 std::optional 的简短示例。

std::optional

```
1 // optional.cpp
2 ...
3 #include <optional>
4
5 std::optional<int> getFirst(const std::vector<int>& vec) {
6     if (!vec.empty()) return std::optional<int>(vec[0]);
7     else return std::optional<int>();
8 }
9
10 ...
11
12 std::vector<int> myVec{1, 2, 3};
13 std::vector<int> myEmptyVec;
14
15 auto myInt= getFirst(myVec);
16
17 if (myInt) {
18     std::cout << *myInt << '\n'; // 1
19     std::cout << myInt.value() << '\n'; // 1
20     std::cout << myInt.value_or(2017) << '\n'; // 1
21 }
22
23 auto myEmptyInt= getFirst(myEmptyVec);
24
25 if (!myEmptyInt) {
26     std::cout << myEmptyInt.value_or(2017) << '\n'; // 2017
27 }
```

函数 getFirst 中使用 std::optional。若第一个元素存在，getFirst 返回该元素。若没有，将得到一个 std::optional<int> 对象。主函数有两个 vector。两者都调用 getFirst 并返回一个 std::optional 对象。在 myInt 的情况下，对象有一个值;myEmptyInt 的情况下，该对象没有值。程序显示 myInt 和 myEmptyInt 的值。myInt.value_or(2017) 返回值，但 myEmptyInt.value_or(2017) 会返回默认值。

C++23 中，std::optional 扩展为一元操作 opt.and_then、opt.transform 和 opt.or_else。and_then 返回给定函数调用的结果 (若存在)，或者返回一个空的 std::optional。transform 返回一个 std::optional，其中包含转换后的值，或者一个空 std::optional。此外，opt.or_else 若包含值或给定函数的结果，则返回 std::optional。

这些一元操作支持 std::optional 上的复合操作：

std::optional 上的一元操作

```
1 // optionalMonadic.cpp
2
3 #include <iostream>
```

```

4 #include <optional>
5 #include <vector>
6 #include <string>
7
8 std::optional<int> getInt(std::string arg) {
9     try {
10         return {std::stoi(arg)};
11     }
12     catch (...) {
13         return { };
14     }
15 }
16
17
18 int main() {
19
20     std::vector<std::optional<std::string>> strings = {"66", "foo", "-5"};
21
22     for (auto s: strings) {
23         auto res = s.and_then(getInt)
24             .transform( [](int n) { return n + 100; })
25             .transform( [](int n) { return std::to_string(n); })
26             .or_else([] { return std::optional<std::string>("Error"); });
27         std::cout << *res << ' '; // 166 Error 95
28     }
29 }
30 }
```

基于范围的 for 循环(第 22 行)遍历 std::vector<std::optional<std::string>>。getInt 函数将每个元素转换为整数(第 23 行),为其添加 100(第 24 行),将其转换回字符串(第 25 行),最后显示字符串(第 27 行)。若初始转换为 int 失败,则返回字符串 Error(第 26 行)并显示。

std::variant

std::variant是类型安全的联合,使用该类型需要头文件 `<variant>`。`std::variant` 的实例具有来自其类型之一的值,类型不能是引用、数组或 void。`std::variant` 可以有多个类型,默认初始化的 `std::variant` 使用其第一个类型初始化,其第一个类型必须有一个默认构造函数。通过使用 `var.index`,可以获得 `std::variant` `var` 持有的备选项的从零开始的索引。若该变量持有值,则 `var.valueless_by_exception` 返回 false,可以使用 `var.emplace` 在原地创建一个新值。一些全局函数用于访问 `std::variant`。函数模板 `var.holds_alternative` 允许检查 `std::variant` 是否包含指定的替代,可以将 `std::get` 与索引和类型作为参数一起使用。通过使用索引,将获得该值。若使用类型调用 `std::get`,则只有在该值唯一的情况下才会获得该值。若使用无效索引或非唯一类型,则会得到 `std::bad_variant_access` 异常。`std::get` 最终返回一个异常,而 `std::get_if` 在发生错误时,会返回一个空指针。

下面的代码片段展示了 `std::variant` 的用法。

`std::variant`

```
1 // variant.cpp
2 ...
3 #include <variant>
4
5 ...
6
7 std::variant<int, float> v, w;
8 v = 12; // v contains int
9 int i = std::get<int>(v);
10 w = std::get<int>(v);
11 w = std::get<0>(v); // same effect as the previous line
12 w = v; // same effect as the previous line
13
14 // std::get<double>(v); // error: no double in [int, float]
15 // std::get<3>(v); // error: valid index values are 0 and 1
16
17 try{
18     std::get<float>(w); // w contains int, not float: will throw
19 }
20 catch (std::bad_variant_access&){}
21
22 std::variant<std::string> v2("abc"); // converting constructor must be unambiguous
23 v2 = "def"; // converting assignment must be unambiguous
```

v 和 w 是两个 `std::variant` 对象，两者都可以有 int 和 float 值，默认值为 0。v 为 12 时，下面的调用 `std::get<int>(V)` 返回值。接下来的三行显示了将变量 v 赋值给 w 的三种可能性，但必须记住一些规则。可以通过类型 `std::get<double>(v)` 或 `index: std::get<3>(v)` 来查询变量值。类型必须唯一且索引有效，变量 w 保存一个 int 值。若为浮点类型，就会得到 `std::bad_variant_access` 异常。若构造函数调用或赋值调用是明确的，则可以进行转换。这使得从 C-string 构造 `std::variant<std::string>`，或为该变体赋值一个新的 C-string 成为可能。

`std::variant` 有一个有趣的非成员函数 `std::visit`，允许在变量列表上执行可调用对象。可调用对象是自定义的可调用对象。通常，这可以是一个函数、函数对象或 Lambda 表达式。为简单起见，在本例中使用 Lambda 函数。

`std::visit`

```
1 // visit.cpp
2 ...
3 #include <variant>
4
5 ...
6
7 std::vector<std::variant<char, long, float, int, double, long long>>
8     vecVariant = {5, '2', 5.4, 10011, 20111, 3.5f, 2017};
```

```

10 for (auto& v: vecVariant){
11     std::visit([](auto&& arg){std::cout << arg << " ";}, v);
12     // 5 2 5.4 100 2011 3.5 2017
13 }
14
15 // display each type
16 for (auto& v: vecVariant){
17     std::visit([](auto&& arg){std::cout << typeid(arg).name() << " ";}, v);
18     // int char double __int64 long float int
19 }
20
21 // get the sum
22 std::common_type<char, long, float, int, double, long long>::type res{};
23
24 std::cout << typeid(res).name() << '\n'; // double
25
26 for (auto& v: vecVariant){
27     std::visit([&res](auto&& arg){res+= arg;}, v);
28 }
29 std::cout << "res: " << res << '\n'; // 4191.9
30
31 // double each value
32 for (auto& v: vecVariant){
33     std::visit([&res](auto&& arg){arg *= 2;}, v);
34     std::visit([](auto&& arg){std::cout << arg << " ";}, v);
35     // 10 d 10.8 200 4022 7 4034
36 }

```

本例中的每个 variant 实例都可以保存 char、long、float、int、double 或 long long 类型。第一个访问器 `[](auto&& arg){std::cout << arg << " ";}` 将输出各种变量。第二个访问器 `std::cout << typeid(arg).name() << " ";` 将显示其类型。

现在我想对这些变量的元素乘 2，在编译时需要正确的结果类型。来自类型特征库的 `std::common_type` 将其提供。`common_type` 给出了所有 `char`、`long`、`float`、`int`、`double` 和 `long long` 类型都可以隐式转换的类型。`res{}` 中的最后一个 {} 导致其初始化为 0.0，`Res` 是 `double` 类型。访问者 `[&res](auto&& arg){arg *= 2;}` 计算总和，下面一行显示相应结果。

2.10. `std::expected`

`std::expected<T, E>` 提供了一种存储两个值中的任意一个的方法。`std::expected` 的实例总是保存一个值：要么是类型 T 的期望值，要么是类型 E 的非期望值，使用该类型需要头文件 `<expected>`。`std::expected` 可以实现返回值或返回错误的函数，存储值直接在预期对象占用的存储空间中分配，从而不进行动态内存分配。

`std::expected` 有一个类似的接口，比如 `std::optional`。与 `std::optional` 相反，`std::expected` 会返回错误消息。

各种构造函数允许定义具有期望值的预期对象 `exp`。`exp.emplace` 将就地构造包含的值，

可以显式地询问 `std::expected` 容器是否有值，或者以在逻辑表达式中进行检查。`exp.value` 返回预期值，`exp.value_or` 返回预期值或默认值。若 `exp` 有一个意外值，调用 `exp.value` 将抛出 `std::bad_expected_access` 异常。

`std::unexpected` 表示存储在 `std::expected` 中的非预期值。

`std::expected`

```
1 // expected.cpp
2
3 #include <iostream>
4 #include <expected>
5 #include <vector>
6 #include <string>
7
8 std::expected<int, std::string> getInt(std::string arg) {
9     try {
10         return std::stoi(arg);
11     }
12     catch (...) {
13         return std::unexpected{std::string(arg + ": Error")};
14     }
15 }
16
17 int main() {
18
19     std::vector<std::string> strings = {"66", "foo", "-5"};
20
21     for (auto s: strings) {
22         auto res = getInt(s);
23         if (res) {
24             std::cout << res.value() << ' '; // 66 -5
25         }
26         else {
27             std::cout << res.error() << ' '; // foo: Error
28         }
29     }
30
31     std::cout << '\n';
32
33     for (auto s: strings) {
34         auto res = getInt(s);
35         std::cout << res.value_or(2023) << ' '; // 66 2023 -5
36     }
37
38 }
```

`getInt` 函数将每个字符串转换为整数，并返回 `std::expected<int, std::string>`。Int 表示预期值，`std::string` 表示意外值。两个基于范围的 `for` 循环（第 22 行和第 34 行）遍历 `std::vector<std::string>`。第

一个基于范围的 for 循环(第 22 行)中，显示预期值(第 25 行)或意外值(第 28 行)。第二个基于范围的 for 循环(第 34 行)中，显示预期值或默认值 2023(第 36 行)。

`std::expected` 支持单进操作以方便函数组合:`exp.and_then`、`exp.transform`、`exp.or_else` 和 `exp.transform_error`。`exp.and_then` 返回给定函数调用的结果(若存在)，或者返回一个空 `std::expected`。`exp.transform` 返回一个 `std::expected`，其中包含转换后的值，或者一个空 `std::expected`。若 `exp.or_else` 包含值或给定函数的结果，则返回 `std::expected`。

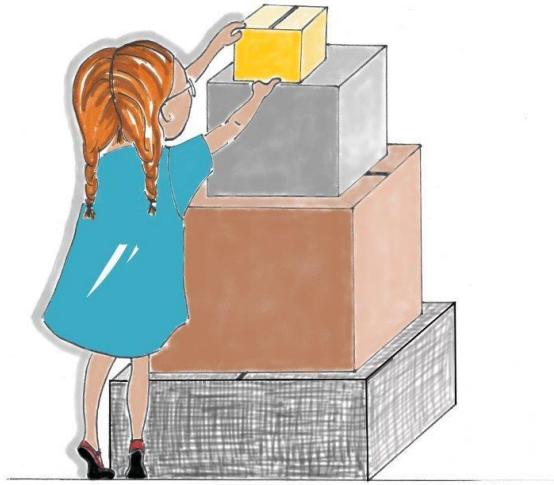
下面的程序基于前面的程序 `optionalMonadic.cpp`，`std::optional` 类型可用 `std::expected` 替换。

对 `std::expected` 的一元操作

```
1 // expectedMonadic.cpp
2 ...
3 #include <expected>
4
5 std::expected<int, std::string> getInt(std::string arg) {
6     try {
7         return std::stoi(arg);
8     }
9     catch (...) {
10         return std::unexpected{std::string(arg + ": Error")};
11     }
12 }
13
14 std::vector<std::string> strings = {"66", "foo", "-5"};
15
16 for (auto s: strings) {
17     auto res = getInt(s)
18         .transform([](int n) { return n + 100; })
19         .transform([](int n) { return std::to_string(n); });
20     std::cout << *res << ' ';// 166 foo: Error 95
21 }
```

基于范围的 for 循环(第 23 行)遍历 `std::vector<std::string>`。`getInt` 函数将每个字符串转换为整数(第 24 行)，为其添加 100(第 25 行)，将其转换回字符串(第 26 行)，最后显示字符串(第 27 行)。若初始转换为 `int` 失败，则返回 `string arg + ": Error"`(第 14 行)并显示。

第3章 所有容器的接口



Cippi 正在打包

标准模板库的序列容器和关联容器有很多共同之处。例如，创建或删除容器、确定其大小、访问其元素、赋值或交换等操作都与容器的元素类型无关。每个容器至少有一个类型参数和一个此类型的分配器。分配器大部分时间在后台工作。`vector` 就是一个例子，`std::vector<int>` 会调用 `std::vector<int, std::allocator<int>>`。由于使用了 `std::allocator`，可以动态调整除 `std::array` 之外的所有容器的大小。他们有更多的共同点，可以使用迭代器访问容器的元素。

尽管高度相似，但容器在细节上有所不同。顺序容器和关联容器章节提供了详细信息。

对于序列容器 `std::array`、`std::vector`、`std::deque`、`std::list` 和 `std::forward_list`，C++ 在每个领域都有相应的专业工具。

关联容器可以分为有序容器和无序容器。

3.1. 创建和删除

每个容器都会提供不同的构造函数。要删除容器 `cont` 中所有的元素，可以使用 `cont.clear()`。创建容器、删除容器、添加或删除元素都一样，每次容器都会自行负责内存的管理。

下表显示了容器的构造函数和析构函数。下面的表中，使用 `std::vector` 作为代表。

创建和删除容器

类型	示例
默认构造	<code>std::vector<int> vec1</code>
范围构造	<code>std::vector<int> vec2(vec1.begin(), vec1.end())</code>
复制构造	<code>std::vector<int> vec3(vec2)</code>
复制构造	<code>std::vector<int> vec3=vec2</code>

移动构造	std::vector<int>vec4(std::move(vec3))
移动构造	std::vector<int>vec4=std::move(vec3)
序列 (初始化器列表)	std::vector<int>vec5{1,2,3,4,5}
序列 (初始化器列表)	std::vector<int>vec5={1,2,3,4,5}
析构	vec5.~vector()
删除所有元素	vec5.clear()

必须在编译时指定 std::array 的大小，并使用[聚合初始化](#)进行初始化，std::array 没有用于删除其元素的成员函数。

下面的例子中，我对不同的容器使用了不同的构造函数。

不同的构造函数

```

1 // containerConstructor.cpp
2 ...
3 #include <map>
4 #include <unordered_map>
5 #include <vector>
6 ...
7 using namespace std;
8
9 vector<int> vec= {1, 2, 3, 4, 5, 6, 7, 8, 9};
10 map<string, int> m= {"bart", 12345}, {"jenne", 34929}, {"huber", 840284} ;
11 unordered_map<string, int> um{m.begin(), m.end()};
12
13 for (auto v: vec) cout << v << " "; // 1 2 3 4 5 6 7 8 9
14 for (auto p: m) cout << p.first << "," << p.second << " ";
15 // bart,12345 huber,840284 jenne,34929
16
17 for (auto p: um) cout << p.first << "," << p.second << " ";
18 // bart,12345 jenne,34929 huber,840284
19
20 vector<int> vec2= vec;
21 cout << vec.size() << endl; // 9
22 cout << vec2.size() << endl; // 9
23
24 vector<int> vec3= move(vec);
25 cout << vec.size() << endl; // 0
26 cout << vec3.size() << endl; // 9
27
28 vec3.clear();
29 cout << vec3.size() << endl; // 0

```

3.2. 大小

对于容器 `cont`, 可以使用 `cont.empty()` 检查容器是否为空。`cont.size()` 返回当前元素数, 而 `cont.max_size()` 返回 `cont` 可以拥有的最大元素数。元素的最大数量由实现定义。

容器的大小

```
1 // containerSize.cpp
2 ...
3 #include <map>
4 #include <set>
5 #include <vector>
6 ...
7 using namespace std;
8
9 vector<int> intVec{1, 2, 3, 4, 5, 6, 7, 8, 9};
10 map<string, int> str2Int = {{"bart", 12345},
11   {"jenne", 34929}, {"huber", 840284}};
12
13 set<double> douSet{3.14, 2.5};
14
15 cout << intVec.empty() << endl; // false
16 cout << str2Int.empty() << endl; // false
17 cout << douSet.empty() << endl; // false
18
19 cout << intVec.size() << endl; // 9
20 cout << str2Int.size() << endl; // 3
21 cout << douSet.size() << endl; // 2
22
23 cout << intVec.max_size() << endl; // 4611686018427387903
24 cout << str2Int.max_size() << endl; // 384307168202282325
25 cout << douSet.max_size() << endl; // 461168601842738790
```

用 `cont.empty()` 代替 `cont.size()`

对于容器 `cont`, 使用成员函数 `cont.empty()` 而非 `(cont.size() == 0)` 来确定容器是否为空。首先, `cont.empty()` 通常比 `(const.size() == 0)` 更快; 第二, `std::forward_list` 没有成员函数 `size()`。

3.3. 访问方式

迭代器允许访问容器的元素。若使用 `begin` 和 `end` 迭代器, 则有一个可以进一步处理的范围。对于容器 `cont`, 使用 `cont.begin()` 获得开始迭代器, 使用 `cont.end()` 获得结束迭代器, 其定义了半开范围。因为开始迭代器属于范围, 所以是半开的, 而结束迭代器指向范围之外的位置。迭代器对 `cont.begin()` 和 `cont.end()` 使其能够修改容器的元素。

创建和删除容器

迭代器	描述
cont.begin() and cont.end()	向前迭代的一对迭代器。
cont.cbegin() and cont.cend()	用于向前 const 迭代的一对迭代器。
cont.rbegin() and cont.rend()	一对向后迭代的迭代器。
cont.crbegin() and cont.crend()	一对迭代器，用于向后 const 迭代。

现在可以修改容器了。

访问容器的元素

```
1 // containerAccess.cpp
2 ...
3 #include <vector>
4 ...
5 struct MyInt{
6     MyInt(int i): myInt(i){};
7     int myInt;
8 };
9
10 std::vector<MyInt> myIntVec;
11 myIntVec.push_back(MyInt(5));
12 myIntVec.emplace_back(1);
13 std::cout << myIntVec.size() << '\n'; // 2
14
15 std::vector<int> intVec;
16 intVec.assign({1, 2, 3});
17 for (auto v: intVec) std::cout << v << " "; // 1 2 3
18
19 intVec.insert(intVec.begin(), 0);
20 for (auto v: intVec) std::cout << v << " "; // 0 1 2 3
21
22 intVec.insert(intVec.begin()+4, 4);
23 for (auto v: intVec) std::cout << v << " "; // 0 1 2 3 4
24
25 intVec.insert(intVec.end(), {5, 6, 7, 8, 9, 10, 11});
26
27 for (auto v: intVec) std::cout << v << " "; // 0 1 2 3 4 5 6 7 8 9 10 11
28
29 for (auto revIt= intVec.rbegin(); revIt != intVec.rend(); ++revIt)
30     std::cout << *revIt << " "; // 11 10 9 8 7 6 5 4 3 2 1 0
31
32 intVec.pop_back();
33 for (auto v: intVec ) std::cout << v << " "; // 0 1 2 3 4 5 6 7 8 9 10
```

3.4. 赋值和交换

可以将新元素分配给现有容器或交换两个容器。将容器 cont2 赋值给容器 cont1，存在复制赋值 cont = cont2 和移动赋值 cont = std::move(cont2)，一种特殊形式的赋值是带有初始化列表的赋值:cont=1,2,3,4,5，这在 std::array 中是不可能的。函数 swap 有两种形式，可以将其作为成员函数 cont.swap(cont2) 或函数模板 std::swap(cont, cont2)。

赋值和交换

```
1 // containerAssignmentAndSwap.cpp
2 ...
3 #include <set>
4 ...
5 std::set<int> set1{0, 1, 2, 3, 4, 5};
6 std::set<int> set2{6, 7, 8, 9};
7
8 for (auto s: set1) std::cout << s << " "; // 0 1 2 3 4 5
9 for (auto s: set2) std::cout << s << " "; // 6 7 8 9
10
11 set1= set2;
12 for (auto s: set1) std::cout << s << " "; // 6 7 8 9
13 for (auto s: set2) std::cout << s << " "; // 6 7 8 9
14
15 set1= std::move(set2);
16 for (auto s: set1) std::cout << s << " "; // 6 7 8 9
17 for (auto s: set2) std::cout << s << " "; //
18
19 set2= {60, 70, 80, 90};
20 for (auto s: set1) std::cout << s << " "; // 6 7 8 9
21 for (auto s: set2) std::cout << s << " "; // 60 70 80 90
22
23 std::swap(set1, set2);
24 for (auto s: set1) std::cout << s << " "; // 60 70 80 90
25 for (auto s: set2) std::cout << s << " "; // 6 7 8 9
```

3.5. 比较

容器支持比较操作符 ==、!=、<、>、<=、>=。这两个容器的比较适用于容器的元素。若比较关联容器，则比较它们的键。无序关联容器只支持比较操作符 == 和 !=。

容器的比较

```
1 // containerComparison.cpp
2 ...
3 #include <array>
4 #include <set>
5 #include <unordered_map>
```

```

6 #include <vector>
7 ...
8 using namespace std;
9
10 vector<int> vec1{1, 2, 3, 4};
11 vector<int> vec2{1, 2, 3, 4};
12 cout << (vec1 == vec2) << endl; // true
13
14 array<int, 4> arr1{1, 2, 3, 4};
15 array<int, 4> arr2{1, 2, 3, 4};
16 cout << (arr1 == arr2) << endl; // true
17
18 set<int> set1{1, 2, 3, 4};
19 set<int> set2{4, 3, 2, 1};
20 cout << (set1 == set2) << endl; // true
21
22 set<int> set3{1, 2, 3, 4, 5};
23 cout << (set1 < set3) << endl; // true
24
25 set<int> set4{1, 2, 3, -3};
26 cout << (set1 > set4) << endl; // true
27
28 unordered_map<int, string> uSet1{{1, "one"}, {2, "two"}};
29 unordered_map<int, string> uSet2{{1, "one"}, {2, "Two"}};
30 cout << (uSet1 == uSet2) << endl; // false

```

3.6. 擦除

独立函数 std::erase(cont, val) 和 std::erase_if(cont, pred) 擦除与 val 比较相等，或满足谓词 pred 的所有容器内容元素。两个函数都返回已删除元素的数目。

一致性容器的擦除

```

1 // erase.cpp
2 ...
3
4 template <typename Cont>
5 void eraseVal(Cont& cont, int val) {
6     std::erase(cont, val);
7 }
8
9 template <typename Cont, typename Pred>
10 void erasePredicate(Cont& cont, Pred pred) {
11     std::erase_if(cont, pred);
12 }
13
14 template <typename Cont>
15 void printContainer(Cont& cont) {

```

```

16     for (auto c: cont) std::cout << c << " ";
17     std::cout << '\n';
18 }
19
20 template <typename Cont>
21 void doAll(Cont& cont) {
22     printContainer(cont);
23     eraseVal(cont, 5);
24     printContainer(cont);
25     erasePredicate(cont, [](auto i) { return i >= 3; } );
26     printContainer(cont);
27 }
28
29 ...
30
31 std::string str{"A sentence with e."};
32 std::cout << "str: " << str << '\n';
33 std::erase(str, 'e');
34 std::cout << "str: " << str << '\n';
35
36 std::cout << "\nstd::vector " << '\n';
37 std::vector vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
38 doAll(vec);
39
40 std::cout << "\nstd::deque " << '\n';
41 std::deque deq{1, 2, 3, 4, 5, 6, 7, 8, 9};
42 doAll(deq);
43
44 std::cout << "\nstd::list" << '\n';
45 std::list lst{1, 2, 3, 4, 5, 6, 7, 8, 9};
46 doAll(lst);

```

erase 和 erase_if 可以应用于所有 STL 容器和 std::string。

```

str: A sentence with e.
str: A sntnc with .

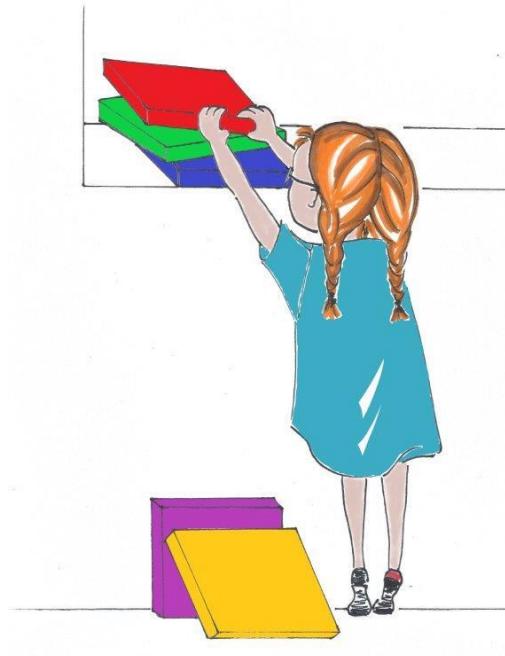
std::vector
1 2 3 4 5 6 7 8 9
1 2 3 4 6 7 8 9
1 2

std::deque
1 2 3 4 5 6 7 8 9
1 2 3 4 6 7 8 9
1 2

std::list
1 2 3 4 5 6 7 8 9
1 2 3 4 6 7 8 9
1 2

```

第4章 顺序容器



Cippi 建造堆栈

顺序容器有很多共同之处，但每个容器都有其特定的域。深入讨论之前，先了解一下 std 命名空间的五个序列容器。

顺序容器

标准	array	vector	deque	list	forward_list
大小	静态	静态	动态	动态	动态
实现	静态数组	动态数组	数组序列	双链表	单链表
访问	随机	随机	随机	前向/后向	前向
插入和删除		末尾: O(1)	开始和末尾: O(1)	开始和末尾: O(1) 任意位置: O(1)	开始位置: O(1) 任意位置: O(1)
内存预留		是	否	否	否
内存释放		shrink_to_fit	shrink_to_fit	总是	总是
优势	无内存分配; 最低内存要求	95% 的方案适用	开头和结尾 插入和删除	任意位置 插入和删除	快速插入和删除; 最低内存要求
缺点	无动态内存分配	任意位置插入 和删除: O(n)	任意位置插入 和删除: O(n)	无随机访问	无随机访问

对于这个表，我还想补充几句。

$O(i)$ 表示操作的复杂度 (运行时)。 $O(1)$ 表示容器上操作的运行时间恒定，并且与容器的大小无关。相反， $O(n)$ 表示运行时线性依赖于容器元素的数量。对于 `std::vector` 意味着什么？元素的访问时间与 `std::vector` 的大小无关，但是插入或删除的耗时和位置相关。

`std::vector` 对元素的随机访问与 `std::deque` 对元素的随机访问具有相同的复杂度 $O(1)$ ，但并不意味着这两种操作都一样快。

双链表 (`std::list`) 或单链表 (`std::forward_list`) 的插入或删除操作，只有在迭代器指向正确的元素时才能保证复杂度为 $O(1)$ 。

`std::string` 就像是 `std::vector<char>`

当然，`std::string` 不是标准模板库的容器。从行为的角度来看，类似于序列容器，尤其是 `std::vector<char>`，所以我会把 `std::string` 视为 `std::vector<char>`。

4.1. Array

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

`std::array` 将 C 数组的内存和运行时特性与 `std::vector` 接口结合起来。

`std::array` 是固定长度的均匀容器，需要头文件 `<array>`，`std::array` 的大小固定。

要初始化 `std::array`，必须遵循一些特殊规则。

`std::array<int, 10> arr`

这 10 个元素没有初始化。

`std::array<int, 10> arr`

这 10 个元素默认初始化。

`std::array<int, 10> arr{1, 2, 3, 4, 5}`

其余元素默认初始化。

`std::array` 支持三种类型的索引访问。

```
1 arr[n];
2 arr.at(n);
3 std::get<n>(arr);
```

最常用的带尖括号的第一类格式不检查 `arr` 的边界，这与 `arr.at(n)` 相反。最终会得到一个 `std::range_error` 异常。最后一个类型显示了 `std::array` 与 `std::tuple` 的关系，它们都是固定长度的容器。
这里有一些关于 `std::array` 的算术运算。

`std::array`

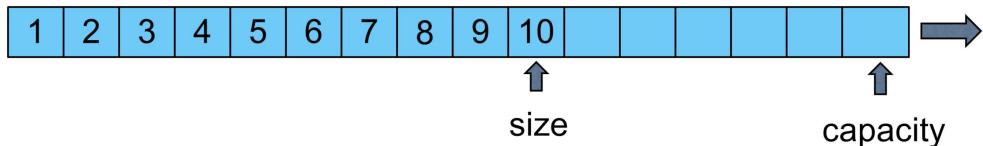
```
1 // array.cpp
2 ...
```

```

3 #include <array>
4 ...
5 std::array<int, 10> arr{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
6 for (auto a: arr) std::cout << a << " " ; // 1 2 3 4 5 6 7 8 9 10
7
8 double sum= std::accumulate(arr.begin(), arr.end(), 0);
9 std::cout << sum << '\n'; // 55
10
11 double mean= sum / arr.size();
12 std::cout << mean << '\n'; // 5.5
13 std::cout << (arr[0] == std::get<0>(arr)); // true

```

4.2. Vector



`std::vector` 是同构容器，其长度在运行时自动调整，`std::vector` 需要头文件 `<vector>`。`vector` 在内存中连续存储元素，因此 `std::vector` 支持指针算术。

```

1 for (int i= 0; i < vec.size(); ++i){
2     std::cout << vec[i] == *(vec + i) << '\n'; // true
3 }

```

创建 `std::vector` 时，要区分圆括号和花括号

构造 `std::vector` 时，必须注意一些特殊规则。下面的例子中，使用圆括号的构造函数创建了一个包含 10 个元素的 `std::vector`，使用大括号的构造函数创建了一个仅包含 10 的 `std::vector`。

```

1 std::vector<int> vec(10);
2 std::vector<int> vec{10};

```

同样的规则也适用于 `std::vector<int>(10,2011)` 或 `std::vector<int>{10,2011}`。第一种情况下，会得到一个 `std::vector`，其中十个元素初始化为 2011。在第二种情况下，会得到一个包含元素 10 和 2011 的 `std::vector`。因为花括号代表一个初始化列表，所以会使用列表构造函数。

大小与容量

`std::vector` 的元素数量通常小于已为其预留空间的元素数量，`std::vector` 的大小可以增加，而不需要分配新内存。

有一些操作可以智能地使用内存。

std::vector 的内存管理

成员函数	描述
vec.size()	vec 的元素个数。
vec.capacity()	vec 无需重新分配即可拥有的元素数。
vec.resize(n)	vec 会将增加到 n 个元素。
vec.reserve(n)	至少保留 n 个元素内存。
vec.shrink_to_fit()	将 vec 的容量减小到其大小。

vec.shrink_to_fit() 没有固定行为，所以运行时可以忽略它。但在主流的平台上，可观察到期望的行为。

我们来使用一下这些成员函数。

std::vector

```
1 // vector.cpp
2 ...
3 #include <vector>
4 ...
5 std::vector<int> intVec1(5, 2011);
6 intVec1.reserve(10);
7 std::cout << intVec1.size() << '\n'; // 5
8 std::cout << intVec1.capacity() << '\n'; // 10
9
10 intVec1.shrink_to_fit();
11 std::cout << intVec1.capacity() << '\n'; // 5
12
13 std::vector<int> intVec2(10);
14 std::cout << intVec2.size() << '\n'; // 10
15
16 std::vector<int> intVec3{10};
17 std::cout << intVec3.size() << '\n'; // 1
18
19 std::vector<int> intVec4{5, 2011};
20 std::cout << intVec4.size() << '\n'; // 2
```

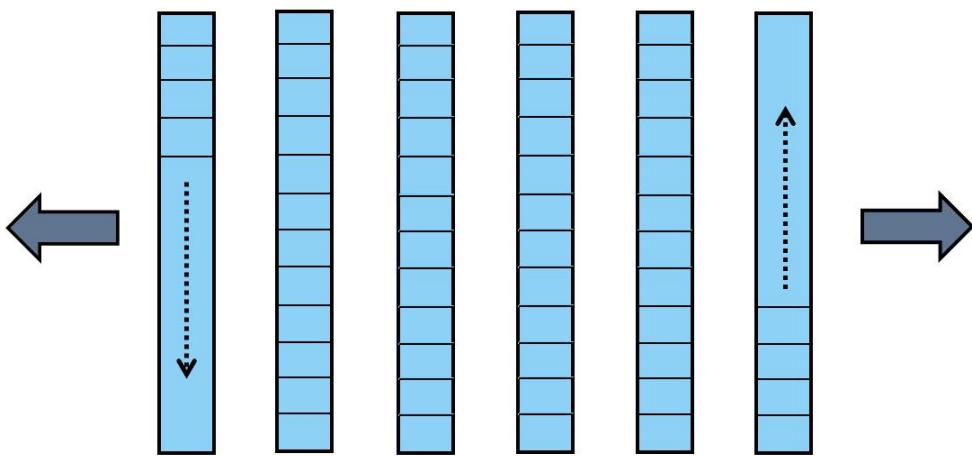
std::vector vec 有几个成员函数来访问它的元素。使用 vec.front(), 可获得第一个元素；使用 vec.back(), 可获得 vec 的最后一个元素。要读取或写入 vec 的 (n+1) 个元素，可以使用索引操作符 vec[n] 或成员函数 vec.at(n)。第二个检查 vec 的边界，因此最终得到一个 std::out_of_range 异常。

除了索引操作符，std::vector 还提供了额外的成员函数来赋值、插入、创建或删除元素。

修改 std::vector 的元素

成员函数	描述
vec.assign(...)	赋值一个或多个元素、一个范围或初始化列表。
vec.clear()	移除 vec 中所有的元素。
vec.emplace(pos, args...)	使用 vec 中的参数在 pos 之前创建一个新元素，并返回该元素的新位置。
vec.emplace_back(args...)	在 vec 中创建一个带有参数的新元素 args...
vec.erase(...)	移除一个元素或一个范围，并返回下一个位置。
vec.insert(pos, ...)	插入一个或多个元素、一个范围或一个初始化列表，并返回元素的新位置。
vec.pop_back()	删除最后一个元素。
vec.push_back(elem)	在 vec 的末尾添加 elem 的副本。

4.3. Deque



`std::deque`通常由固定大小的数组序列组成，与 `std::vector` 非常相似，需要头文件 `<deque>`。
`std::deque` 有三个额外的成员函数，`deq.push_front(elem)`, `deq.pop_front()` 和 `deq.emplace_front(args...)` 在开头添加或删除元素。

`std::deque`

```

1 // deque.cpp
2 ...
3 #include <deque>
4 ...
5 struct MyInt{
6     MyInt(int i): myInt(i){};
7     int myInt;
8 };
9
10 std::deque<MyInt> myIntDeq;
11
12 myIntDeq.push_back(MyInt(5));
13 myIntDeq.emplace_back(1);

```

```

14 std::cout << myIntDeq.size() << '\n'; // 2
15
16 std::deque<MyInt> intDeq;
17 intDeq.assign({1, 2, 3});
18 for (auto v: intDeq) std::cout << v << " "; // 1 2 3
19
20 intDeq.insert(intDeq.begin(), 0);
21 for (auto v: intDeq) std::cout << v << " "; // 0 1 2 3
22
23 intDeq.insert(intDeq.begin()+4, 4);
24 for (auto v: intDeq) std::cout << v << " "; // 0 1 2 3 4
25
26 intDeq.insert(intDeq.end(), {5, 6, 7, 8, 9, 10, 11});
27 for (auto v: intDeq) std::cout << v << " "; // 0 1 2 3 4 5 6 7 8 9 10 11
28
29 for (auto revIt= intDeq.rbegin(); revIt != intDeq.rend(); ++revIt)
30     std::cout << *revIt << " "; // 11 10 9 8 7 6 5 4 3 2 1 0
31
32 intDeq.pop_back();
33 for (auto v: intDeq) std::cout << v << " "; // 0 1 2 3 4 5 6 7 8 9 10
34
35 intDeq.push_front(-1);
36 for (auto v: intDeq) std::cout << v << " "; // -1 0 1 2 3 4 5 6 7 8 9 10

```

4.4. List



`std::list` 是一个双链表。`std::list` 需要包含头文件 `<list>`。

尽管 `std::list` 与 `std::vector` 或 `std::deque` 有类似的接口，但 `std::list` 由于它的结构，所以两者有很大的不同。

`std::list` 有以下几个特点：

- 不支持随机访问。
- 访问元素的速度很慢，最坏的情况下，必须遍历整个列表。
- 若迭代器指向正确的位置，添加或删除元素很快。
- 若添加或删除元素，迭代器持续有效。

由于其独特的结构，`std::list` 有一些特殊的成员函数。

std::list 的特殊成员函数

成员函数	描述
<code>lis.merge(c)</code>	将有序列表 <code>c</code> 合并到有序列表 <code>list</code> 中， <code>list</code> 保持有序。

lis.merge(c, op)	将有序列表 c 合并到有序列表 list 中，使 list 保持有序。其使用 op 作为排序标准。
lis.remove(val)	从列表中删除值为 val 的所有元素。
lis.remove_if(pre)	从列表中删除所有满足谓词 pre 的元素。
lis.splice(pos, ...)	拆分 pos 之前 list 中的元素。元素可以是单个元素、范围或列表。
lis.unique()	移除相同值的相邻元素。
lis.unique(pre)	移除满足谓词 pre 的相邻的元素。

下面的代码段中展示如何使用其成员函数。

std::list

```

1 // list.cpp
2 ...
3 #include <list>
4 ...
5 std::list<int> list1{15, 2, 18, 19, 4, 15, 1, 3, 18, 5,
6             4, 7, 17, 9, 16, 8, 6, 6, 17, 1, 2};
7
8 list1.sort();
9 for (auto l: list1) std::cout << l << " ";
10 // 1 1 2 2 3 4 4 5 6 6 7 8 9 15 15 16 17 17 18 18 19
11
12 list1.unique();
13 for (auto l: list1) std::cout << l << " ";
14 // 1 2 3 4 5 6 7 8 9 15 16 17 18 19
15
16 std::list<int> list2{10, 11, 12, 13, 14};
17 list1.splice(std::find(list1.begin(), list1.end(), 15), list2);
18 for (auto l: list1) std::cout << l << " ";
19 // 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

```

4.5. 前向 List



*std::forward_list*是一个单链表，需要头文件 `<forward_list>`。*std::forward_list*没有太多的接口，并针对最小的内存需求进行了优化。

*std::forward_list*与*std::list*有很多共同之处：

- 不支持随机访问。
- 访问任意元素的速度很慢，最坏的情况下，必须向前遍历整个列表。
- 若迭代器指向正确的位置，添加或删除元素很快。
- 若添加或删除元素，迭代器持续有效。

- 操作总是引用 std::forward_list 的开头或当前元素之后的位置。

可以向前迭代 std::forward_list 的特性具有重要的影响，迭代器不能自减，因此不支持在迭代器上进行自减自加之类的操作。出于同样的原因，std::forward_list 没有向后迭代器，并且是唯一不知道其大小的序列容器。

std::forward_list 有一个异常域

std::forward_list 是单链表的替代。若元素的插入、提取或移动只影响相邻的元素，则会针对最小的内存管理和性能进行优化。这是典型的排序算法。

std::forward_list 的特殊成员函数

成员函数	描述
forw.before_begin()	返回第一个元素之前的迭代器。
forw.emplace_after(pos, args...)	在 pos 之后创建一个带有参数 args... 的元素
forw.emplace_front(args...)	在 forw 开头创建一个参数为 args... 的元素
forw.erase_after(pos, ...)	从 forw 中移除元素 pos 或从 pos 开始的元素范围。
forw.insert_after(pos, ...)	在两个新元素之后插入。这些元素可以是单个元素、范围或初始化列表。
forw.merge(c)	将列表 c 合并到排序的列表 forw 中。
forw.merge(c, op)	将 forw 有序列表 c 合并到前向排序列表 forw 中，使用 op 进行排序。
forw.splice_after(pos, ...)	forw 拆分 pos 之前的元素。元素可以是单个元素、范围或列表。
forw.unique()	移除具有相同值的相邻元素。
forw.unique(pre)	移除满足谓词 pre 的相邻元素。

来使用一下 std::forward_list 的成员函数。

std::forward_list

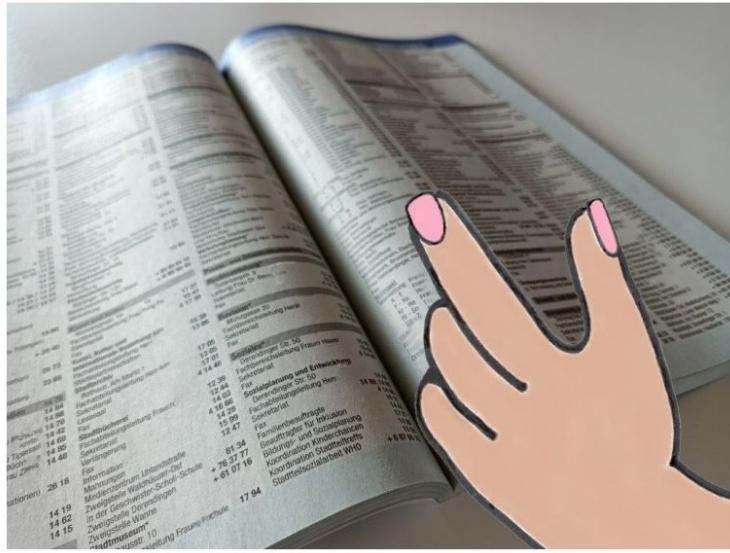
```

1 // forwardList.cpp
2 ...
3 #include<forward_list>
4 ...
5 using std::cout;
6
7 std::forward_list<int> forw;
8 std::cout << forw.empty() << '\n'; // true
9
10 forw.push_front(7);
11 forw.push_front(6);
12 forw.push_front(5);
13 forw.push_front(4);
14 forw.push_front(3);

```

```
15  forw.push_front(2);
16  forw.push_front(1);
17  for (auto i: forw) cout << i << " "; // 1 2 3 4 5 6 7
18
19  forw.erase_after(forw.before_begin());
20  cout << forw.front(); // 2
21
22  std::forward_list<int> forw2;
23  forw2.insert_after(forw2.before_begin(), 1);
24  forw2.insert_after(++forw2.before_begin(), 2);
25  forw2.insert_after(++(++forw2.before_begin()), 3);
26  forw2.push_front(1000);
27  for (auto i= forw2.cbegin(); i != forw2.cend(); ++i) cout << *i << " ";
28  // 1000 1 2 3
29
30  auto IteratorTo5= std::find(forw.begin(), forw.end(), 5);
31  forw.splice_after(IteratorTo5, std::move(forw));
32  for (auto i= forw.cbegin(); i != forw.cend(); ++i) cout << *i << " ";
33  // 2 3 4 5 1000 1 2 3 6 7
34
35  forw.sort();
36  for (auto i= forw.cbegin(); i != forw.cend(); ++i) cout << *i << " ";
37  // 1 2 2 3 3 4 5 6 7 1000
38
39  forw.reverse();
40  for (auto i= forw.cbegin(); i != forw.cend(); ++i) cout << *i << " ";
41  // 1000 7 6 5 4 3 3 2 2 1
42
43  forw.unique();
44  for (auto i= forw.cbegin(); i != forw.cend(); ++i) cout << *i << " ";
45  // 1000 7 6 5 4 3 2 1
```

第 5 章 关联容器



Cippi 在搜索电话簿

C++ 有八种不同的[关联容器](#)。四个具有排序键的关联容器:`std::set`、`std::map`、`std::multiset`和`std::multimap`。另外四个是具有未排序键的关联容器:`std::unordered_set`、`std::unordered_map`、`std::unordered_multiset`和`std::unordered_multimap`。关联容器是特殊的容器，容器支持的接口在前文中已经进行过描述。

5.1. 概述

八个有序和无序容器都有一个共同点，其将键与值关联起来，可以使用键来获取值。要对关联容器进行分类，必须了解三个问题：

- 键是否已排序？
- 键是否有关联的值？
- 一个键可以出现多次吗？

下表 $2^3 = 8$ 行给出了这三个问题的答案。我回答表格中的第四个问题。最好的情况下，键的访问时间有多快？

关联容器的特征

关联容器	排序	关联值	相同的键	访问时间
<code>std::set</code>	yes	no	no	对数
<code>std::unordered_set</code>	no	no	no	常数
<code>std::map</code>	yes	yes	no	对数
<code>std::unordered_map</code>	no	yes	no	常数
<code>std::multiset</code>	yes	no	yes	对数

std::unordered_multiset	no	no	yes	常数
std::multimap	yes	yes	yes	对数
std::unordered_multimap	no	yes	yes	常数

自 C++98 起，就有了顺序关联容器，C++11 还增加了无序关联容器。这两个类都有非常相似的接口。这就是为什么下面的代码示例对于 std::map 和 std::unordered_map 是相同的。更准确地说，std::unordered_map 的接口是 std::map 接口的超集。其余三个无序关联容器也是如此，将代码从有序容器移植到无序容器很容易。

可以使用初始化列表初始化容器，并使用索引操作符添加新元素。要访问键/值对 p 的第一个元素，首先使用 p，对于第二个元素，使用 p.second。p.first 是键，p.second 是对子的关联值。

std::map 与 std::unordered_map

```

1 // orderedUnorderedComparison.cpp
2 ...
3 #include <map>
4 #include <unordered_map>
5
6 // std::map
7
8 std::map<std::string, int> m {{"Dijkstra", 1972}, {"Scott", 1976}};
9 m["Ritchie"] = 1983;
10 std::cout << m["Ritchie"]; // 1983
11 for (auto p : m) std::cout << "{" << p.first << "," << p.second << "}";
12 // {Dijkstra,1972},{Ritchie,1983},{Scott,1976}
13 m.erase("Scott");
14 for (auto p : m) std::cout << "{" << p.first << "," << p.second << "}";
15 // {Dijkstra,1972},{Ritchie,1983}
16 m.clear();
17 std::cout << m.size() << '\n'; // 0
18
19 // std::unordered_map
20 std::unordered_map<std::string, int> um {{"Dijkstra", 1972}, {"Scott", 1976}};
21 um["Ritchie"] = 1983;
22 std::cout << um["Ritchie"]; // 1983
23 for (auto p : um) std::cout << "{" << p.first << "," << p.second << "}";
24 // {Ritchie,1983},{Scott,1976},{Dijkstra,1972}
25 um.erase("Scott");
26 for (auto p : um) std::cout << "{" << p.first << "," << p.second << "}";
27 // {Ritchie,1983},{Dijkstra,1972}
28
29 um.clear();
30 std::cout << um.size() << '\n'; // 0

```

这两个程序执行之间有一个微妙的区别：std::map 的键有序，而 std::unordered_map 的键无序。问题是：为什么在 C++ 中有这样类似的容器？我已经在表格中指出，对于 C++ 来说，答案是：性能。对

无序关联容器的键的访问时间是常量，因此与容器的大小无关。若容器足够大，性能差异则会越显著。

contains

成员函数 associativeContainer.contains(ele) 检查 associativeContainer 是否有元素 ele。

检查关联容器是否有元素

```
1 // containsElement.cpp
2 ...
3 template <typename AssozCont>
4 bool containsElement5(const AssozCont& assozCont) {
5     return assozCont.contains(5);
6 }
7
8 std::set<int> mySet{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
9 std::cout << "containsElement5(mySet): "
10      << containsElement5(mySet); // true
11
12 std::unordered_set<int> myUnordSet{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
13 std::cout << "containsElement5(myUnordSet): "
14      << containsElement5(myUnordSet); // true
15
16 std::map<int, std::string> myMap{ {1, "red"}, {2, "blue"}, {3, "green"} };
17 std::cout << "containsElement5(myMap): "
18      << containsElement5(myMap); // false
19
20 std::unordered_map<int, std::string> myUnordMap{ {1, "red"}, {2, "blue"}, 
21                                         {3, "green"} };
22 std::cout << "containsElement5(myUnordMap): "
23      << containsElement5(myUnordMap); // false
```

插入和删除

关联容器中元素的插入 (插入和放置) 和删除 (擦除) 操作类似于 std::vector 的规则。对于一个键只能有一次的关联容器，若键已经在容器中，则插入失败。此外，有序关联容器支持一个特殊的函数 ordAssCont.erase(key)，会删除所有具有键的数值对，并返回其数量。

插入和删除

```
1 // associativeContainerModify.cpp
2 ...
3 #include <set>
4 ...
5 std::multiset<int> mySet{3, 1, 5, 3, 4, 5, 1, 4, 4, 3, 2, 2, 7, 6, 4, 3, 6};
```

```

6
7 for (auto s: mySet) std::cout << s << " ";
8 // 1 1 2 2 3 3 3 3 4 4 4 4 5 5 6 6 7
9
10 mySet.insert(8);
11 std::array<int, 5> myArr{10, 11, 12, 13, 14};
12 mySet.insert(myArr.begin(), myArr.begin() + 3);
13 mySet.insert({22, 21, 20});
14 for (auto s: mySet) std::cout << s << " ";
15 // 1 1 2 2 3 3 3 3 4 4 4 4 5 5 6 6 7 10 11 12 20 21 22
16
17 std::cout << mySet.erase(4); // 4
18 mySet.erase(mySet.lower_bound(5), mySet.upper_bound(15));
19 for (auto s: mySet) std::cout << s << " ";
20 // 1 1 2 2 3 3 3 3 20 21 22

```

5.2. 有序关联容器

概述

有序关联容器 `std::map` 和 `std::multimap` 将键与一个值关联起来，两者都在头文件 `<map>` 中定义。`std::set` 和 `std::multiset` 需要头文件 `<set>`。

四个有序容器都由它们的类型、分配器和比较函数参数化。根据类型的不同，容器具有分配器和比较函数的默认值。`std::map` 和 `std::set` 的声明很好地说明了这一点。

```

1 template < class key, class val, class Comp= less<key>,
2         class Alloc= allocator<pair<const key, val> >
3 class map;
4
5 template < class T, class Comp = less<T>,
6           class Alloc = allocator<T> >
7 class set;

```

两个关联容器的声明表明 `std::map` 有一个关联值。键和值用于默认的分配器`:allocator<pair<const key, val>>`，可以从分配器派生出更多东西。`std::map` 具有 `std::pair<const key, val>` 类型的键值对。关联值 `val` 对于排序条件不重要`:less<key>`，观察结果对于 `std::multimap` 和 `std::multiset` 也成立。

键和值

对于有序关联容器的键和值有特殊的规则。

键必须是

- 可排序 (默认 `<`)，
- 可复制和可移动。

值必须是

- 默认可构造,
- 可复制和可移动。

键关联值构建一个键值对 p, 首先与成员键 p.first, 值 p.second。

```
1 #include <map>
2 ...
3 std::multimap<char, int> multiMap= {{'a', 10}, {'a', 20}, {'b', 30}};
4 for (auto p: multiMap) std::cout << "{" << p.first << ", " << p.second << "}" " ;
5 // {a,10} {a,20} {b,30}
```

比较标准

顺序关联容器的默认比较条件是 std::less。若要使用用户定义类型作为键，则必须重载操作符<。为数据类型重载操作符<就足够了，因为C++运行时在关系符(!)的帮助下进行比较。(!(elem1<elem2 || elem2<elem1)), 两个元素相等。

可以将排序条件指定为模板参数。这个排序标准必须实现严格的弱序。

严格弱序

若满足下列条件，则给出集合 S 上排序准则的严格弱序。

- 若 s 来自 S，则 ‘s < s’ 不成立。
- S 中的所有 s1 和 s2 必须成立：若 s1 < s2，则 s2 < s1 不成立。
- 对于所有 s1、s2 和 s3，其中 s1 < s2 和 s2 < s3 必须满足 s1 < s3。
- 对于所有 s1、s2 和 s3，其中 s1 与 s2 不可比较，s2 与 s3 不可比较，必须成立 s1 与 s3 不可比较。

与严格弱序的定义相反，对 std::map 使用带有严格弱序的比较条件要简单得多。

```
1 #include <map>
2 ...
3 std::map<int, std::string, std::greater<int>> int2Str{
4     {5, "five"}, {1, "one"}, {4, "four"}, {3, "three"}, 
5     {2, "two"}, {7, "seven"}, {6, "six"} };
6 for (auto p: int2Str) std::cout << "{" << p.first << ", " << p.second << "}" " ;
7 // {7,seven} {6,six} {5,five} {4,four} {3,three} {2,two} {1,one}
```

特定的搜索函数

有序关联容器针对搜索进行了优化，所以会提供了特定的搜索函数。

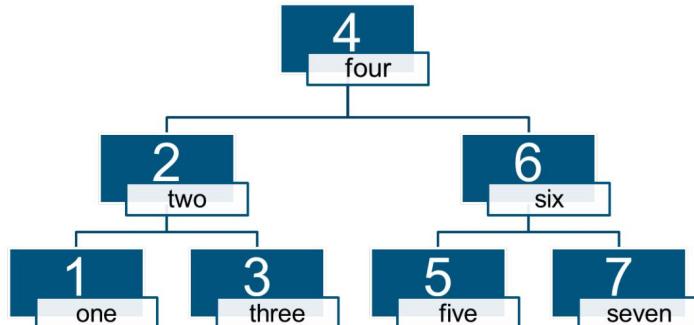
有序关联容器的特殊搜索函数

搜索函数	描述
ordAssCont.count(key)	返回带有键的值的个数。
ordAssCont.find(key)	返回 ordAssCont 中 key 的迭代器。若在 ordAssCont 中没有键，返回 ordAssCont.end()。
ordAssCont.lower_bound(key)	返回 ordAssCont 第一个插入键迭代器的位置。
ordAssCont.upper_bound(key)	返回 ordAssCont 最后插入键迭代器的位置。
ordAssCont.equal_range(key)	返回 std::pair 中的范围 ordAssCont.lower_bound(key) 和 ordAssCont.upper_bound(key)。

关联容器中进行搜索

```
1 // associativeContainerSearch.cpp
2 ...
3 #include <set>
4 ...
5 std::multiset<int> mySet{3, 1, 5, 3, 4, 5, 1, 4, 4, 3, 2, 2, 7, 6, 4, 3, 6};
6
7 for (auto s: mySet) std::cout << s << " ";
8 // 1 1 2 2 3 3 3 4 4 4 4 5 5 6 6 7
9
10 mySet.erase(mySet.lower_bound(4), mySet.upper_bound(4));
11 for (auto s: mySet) std::cout << s << " ";
12 // 1 1 2 2 3 3 3 5 5 6 6 7
13 std::cout << mySet.count(3) << '\n'; // 4
14 std::cout << *mySet.find(3) << '\n'; // 3
15 std::cout << *mySet.lower_bound(3) << '\n'; // 3
16 std::cout << *mySet.upper_bound(3) << '\n'; // 5
17 auto pair= mySet.equal_range(3);
18 std::cout << "(" << *pair.first << ", " << *pair.second << ")"; // (3,5)
```

std::map



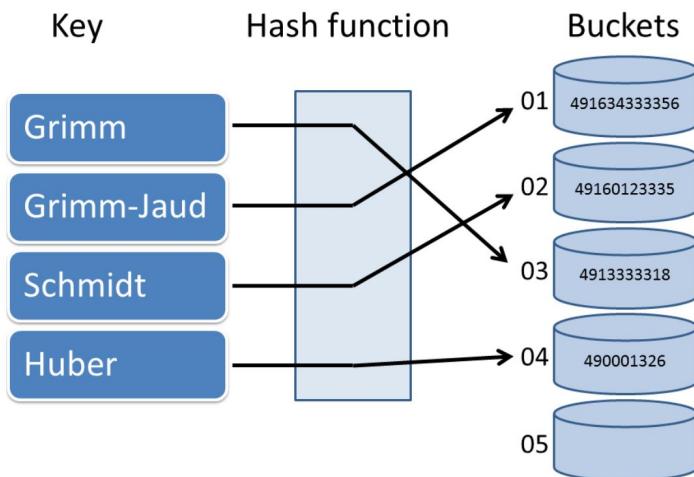
`std::map`是最常用的关联容器，它结合了通常足够的性能和便捷的接口，可以通过索引操作符访问它的元素。若键不存在，`std::map` 创建一个键/值对。对于该值，使用默认构造函数。

将 `std::map` 看作为 `std::vector` 的泛化

通常，`std::map` 称为关联数组，因为 `std::map` 像序列容器一样支持索引操作符。区别在于，其索引不像 `std::vector` 那样局限于数字。`std::map` 的索引几乎可以是任意类型。同样的观察结果也适用于 `std::unordered_map`。

除了索引操作符之外，`std::map` 还支持 `at` 成员函数。检查通过 `at` 的访问。若请求键在 `std::map` 中不存在，则抛出 `std::out_of_range` 异常。

5.3. 无序关联容器



概述

新的 C++11 标准中，有四个无序关联容器:`std::unordered_map`、`std::unordered_multimap`、`std::unordered_set` 和 `std::unordered_multiset`。它们与有序关联容器有很多共同之处。不同之处在于，无序的具有更丰富的接口，并且键不排序。

这里展示了 `std::unordered_map` 的声明。

```
1 template< class key, class val, class Hash= std::hash<key>,
2         class KeyEqual= std::equal_to<key>,
3         class Alloc= std::allocator<std::pair<const key, val>>>
4 class unordered_map;
```

和 `std::map` 一样，`std::unordered_map` 有一个分配器，但是 `std::unordered_map` 不需要比较函数。相反，`std::unordered_map` 需要两个额外的函数：一个用于确定其键的哈希值:`std::hash<key>`；第

二个用于比较键是否相等:`std::equal_to<key>`。由于有三个默认模板参数，只需要提供键类型和值给`std::unordered_map`: `std::unordered_map<char,int> unordMap`。

键和值

无序关联容器的键和值有特殊的规则。

键必须是

- 相等可比，
- 可作为哈希值，
- 可复制或可移动。

值必须是

- 可默认构造，
- 可复制或可移动。

性能

性能，这是很重要的原因——为什么无序关联容器在 C++ 中被忽略了这么久。在下面的例子中，从 1000 万个大的 `std::map` 和 `std::unordered_map` 中读取 100 万个随机创建的值。令人印象深刻的结果是，无序关联容器的线性访问时间比有序关联容器的访问时间快 20 倍，这就是这些运算的常数和对数复杂度 $O(\log n)$ 之间的区别。

比较性能

```
1 // associativeContainerPerformance.cpp
2 ...
3 #include <map>
4 #include <unordered_map>
5 ...
6 using std::chrono::duration;
7 static const long long mapSize= 10000000;
8 static const long long accSize= 1000000;
9 ...
10 // read 1 million arbitrary values from a std::map
11 // with 10 million values from randValues
12 auto start= std::chrono::system_clock::now();
13 for (long long i= 0; i < accSize; ++i){myMap[randValues[i]];}
14 duration<double> dur= std::chrono::system_clock::now() - start;
15 std::cout << dur.count() << " sec"; // 9.18997 sec
16 // read 1 million arbitrary values from a std::unordered_map
17 // with 10 million values
18 auto start2= std::chrono::system_clock::now();
19 for (long long i= 0; i < accSize; ++i){ myUnorderedMap[randValues[i]];}
20 duration<double> dur2= std::chrono::system_clock::now() - start2;
```

```
21 std::cout << dur2.count() << " sec"; // 0.411334 sec
```

哈希函数

无序关联容器访问时间恒定的原因是哈希函数。哈希函数将键映射到它的值，即所谓的哈希值。若哈希函数产生尽可能少的冲突，并将键均匀地分配到桶上，那就是好的。因为哈希函数的执行花费的时间是恒定的，所以在基本情况下，元素的访问性能也恒定。

哈希函数的特点

- 已经为布尔型、自然数和浮点数等内置类型定义，
- 可用于 std::string 和 std::wstring，
- 为 C 字符串 const char 生成指针地址的哈希值，
- 可以为用户定义的数据类型定义。

对于用作无序关联容器的键的用户定义类型，必须牢记两个要求。需要一个哈希函数，并且必须相等。

自定义哈希函数

```
1 // unorderedMapHash.cpp
2 ...
3 #include <unordered_map>
4 ...
5 struct MyInt{
6     MyInt(int v) :val(v) {}
7     bool operator==(const MyInt& other) const {
8         return val == other.val;
9     }
10    int val;
11 };
12
13 struct MyHash{
14     std::size_t operator()(MyInt m) const {
15         std::hash<int> hashVal;
16         return hashVal(m.val);
17     }
18 };
19
20 std::ostream& operator<< (std::ostream& st, const MyInt& myIn){
21     st << myIn.val ;
22     return st;
23 }
24
25 #typedef std::unordered_map<MyInt, int, MyHash> MyIntMap;
26 MyIntMap myMap{{MyInt(-2), -2}, {MyInt(-1), -1}, {MyInt(0), 0}, {MyInt(1), 1}};
27
```

```
28 for(auto m : myMap) std::cout << "(" << m.first << "," << m.second << ")" " ;
29 // {MyInt(1),1} {MyInt(0),0} {MyInt(-1),-1} {MyInt(-2),-2}
30
31 std::cout << myMap[MyInt(-2)] << '\n'; // -2
```

更多细节

无序关联容器将其索引存储在桶中。索引到哪个桶是由哈希函数决定的，哈希函数将键映射到索引。若不同的键映射到相同的索引，则称为冲突。哈希函数试图避免这种情况的出现。

索引通常以链表的形式存储在桶中。对桶的访问是恒定的；桶中的访问是线性的。桶的数量称为容量。每个桶的平均元素数称为负载因子。通常，若负载因子超过 1，C++ 运行时将生成新的桶。这个过程被称为“重哈希”，也可以显式触发：

哈希函数的更多细节

```
1 // hashInfo.cpp
2 ...
3 #include <unordered_set>
4 ...
5 using namespace std;
6
7 void getInfo(const unordered_set<int>& hash) {
8     cout << "hash.bucket_count(): " << hash.bucket_count();
9     cout << "hash.load_factor(): " << hash.load_factor();
10 }
11
12 unordered_set<int> hash;
13 cout << hash.max_load_factor() << endl; // 1
14
15 getInfo(hash);
16 // hash.bucket_count(): 1
17 // hash.load_factor(): 0
18
19 hash.insert(500);
20 cout << hash.bucket(500) << endl; // 5
21
22 // add 100 arbitrary values
23 fillHash(hash, 100);
24
25 getInfo(hash);
26 // hash.bucket_count(): 109
27 // hash.load_factor(): 0.88908
28
29 hash.rehash(500);
30
31 getInfo(hash);
```

```
32 // hash.bucket_count(): 541  
33 // hash.load_factor(): 0.17298  
34 cout << hash.bucket(500); // 500
```

通过成员函数 `max_load_factor`, 可以读取和设置负载因子, 所以可以控制碰撞和重哈希的概率。我在上面的简短例子中强调了一点。键 500 首先在第 5 个桶中, 但在重哈希之后在第 500 个桶中。

第 6 章 容器适配器



Cippi 把形状各异的玩具放进盒子里

6.1. 线性容器

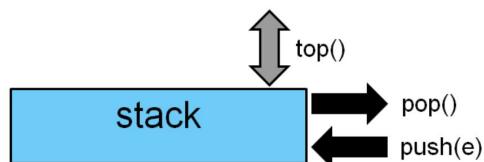
C++ 将 `std::stack`、`std::queue` 和 `std::priority_queue` 作为三个特殊的序列容器。大多数人都知道这些经典的数据结构。

容器的适配器

- 支持现有序列容器的简化接口；
- 不能与标准模板库的算法一起使用，
- 由数据类型和容器参数化的类模板 (`std::vector`、`std::list` 和 `std::deque`)；
- 默认使用 `std::deque` 作为内部序列容器：

```
1 template <typename T, typename Container= deque<T>>
2 class stack;
```

栈



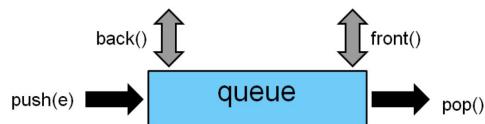
`std::stack` 遵循后进先出原则 (后进先出)。栈 `sta` 需要头文件 `<stack>`，它有三个特殊的成员函数。

使用 `sta.push(e)`，可以在堆栈的顶部插入一个新元素 `e`，使用 `sta.pop()` 可从顶部删除它，并使用 `sta.top()` 可引用它。堆栈支持比较操作符，并且知道期大小。栈操作具有恒定的复杂性。

std::stack

```
1 // stack.cpp
2 ...
3 #include <stack>
4 ...
5 std::stack<int> myStack;
6
7 std::cout << myStack.empty() << '\n'; // true
8 std::cout << myStack.size() << '\n'; // 0
9
10 myStack.push(1);
11 myStack.push(2);
12 myStack.push(3);
13 std::cout << myStack.top() << '\n'; // 3
14
15 while (!myStack.empty()){
16     std::cout << myStack.top() << " ";
17     myStack.pop();
18 } // 3 2 1
19
20 std::cout << myStack.empty() << '\n'; // true
21 std::cout << myStack.size() << '\n'; // 0
```

队列



std::queue 遵循 FIFO 原则 (先进先出)。队列 que 需要头文件 <queue>, 它有四个特殊的成员函数。

使用 queue.push(e), 可以在队列的末尾插入元素 e, 并使用 queue.pop() 从队列中删除第一个元素。queue.back() 允许引用 queue 的最后一个组件 queue.front() 到 queue 中的第一个元素。std::queue 与 std::stack 具有相似的特征, 可以比较 std::queue 实例并获得它们的大小。队列操作也具有恒定的复杂性。

std::queue

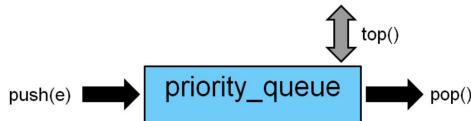
```
1 // queue.cpp
2 ...
3 #include <queue>
4 ...
5 std::queue<int> myQueue;
```

```

6
7 std::cout << myQueue.empty() << '\n'; // true
8 std::cout << myQueue.size() << '\n'; // 0
9
10 myQueue.push(1);
11 myQueue.push(2);
12 myQueue.push(3);
13 std::cout << myQueue.back() << '\n'; // 3
14 std::cout << myQueue.front() << '\n'; // 1
15
16 while (!myQueue.empty()){
17     std::cout << myQueue.back() << " ";
18     std::cout << myQueue.front() << " : ";
19     myQueue.pop();
20 } // 3 1 : 3 2 : 3 3
21
22 std::cout << myQueue.empty() << '\n'; // true
23 std::cout << myQueue.size() << '\n'; // 0

```

优先级队列



`std::priority_queue` 是一个简化的 `std::queue`, 需要头文件 `<queue>`。

与 `std::queue` 的不同之处在于, 它们的最大元素总是位于优先级队列的顶部。`std::priority_queue` `pri` 默认使用比较运算符是 `std::less`。与 `std::queue` 类似, `pri.push(e)` 在优先级队列中插入一个新元素 `e`。`pri.pop()` 删除 `pri` 的第一个元素, 但其复杂度为对数。使用 `pri.top()`, 可以引用优先级队列中的第一个元素, 它是最大的一个。`priority_queue` 知道其大小, 但不支持对其实例使用比较运算符。

`std::priority_queue`

```

1 // priorityQueue.cpp
2 ...
3 #include <queue>
4 ...
5 std::priority_queue<int> myPriorityQueue;
6
7 std::cout << myPriorityQueue.empty() << '\n'; // true
8 std::cout << myPriorityQueue.size() << '\n'; // 0
9
10 myPriorityQueue.push(3);
11 myPriorityQueue.push(1);
12 myPriorityQueue.push(2);

```

```

13 std::cout << myPriorityQueue.top() << '\n'; // 3
14
15 while (!myPriorityQueue.empty()){
16     std::cout << myPriorityQueue.top() << " ";
17     myPriorityQueue.pop();
18 } // 3 2 1
19
20 std::cout << myPriorityQueue.empty() << '\n'; // true
21 std::cout << myPriorityQueue.size() << '\n'; // 0
22
23 std::priority_queue<std::string, std::vector<std::string>,
24                     std::greater<std::string>> myPriorityQueue2;
25
26 myPriorityQueue2.push("Only");
27 myPriorityQueue2.push("for");
28 myPriorityQueue2.push("testing");
29 myPriorityQueue2.push("purpose");
30 myPriorityQueue2.push(".");
31
32 while (!myPriorityQueue2.empty()){
33     std::cout << myPriorityQueue2.top() << " ";
34     myPriorityQueue2.pop();
35 } // . Only for purpose testing

```

6.2. 关联性容器

C++23 中的四个关联容器 `std::flat_map`、`std::flat_multimap`、`std::flat_set` 和 `std::set_multiset` 是对有序关联容器 `std::map`、`std::multimap`、`std::set` 和 `std::multiset` 的临时替换。更准确地说，`std::flat_map` 是 `std::map` 的临时替代，`std::flat_multimap` 是 `std::multimap` 的临时替代，依此类推。

平序关联容器的键和值需要单独的序列容器，该序列容器必须支持随机访问迭代器。默认情况下，会使用 `std::vector`，但对于 `std::array` 或 `std::deque` 也是有效的。

下面的代码段显示了 `std::flat_map` 和 `std::flat_set` 的声明。

```

1 template<class Key, class T,
2         class Compare = less<Key>,
3         class KeyContainer = vector<Key>, class MappedContainer = vector<T>>
4 class flat_map;
5
6 template<class Key,
7         class Compare = less<Key>,
8         class KeyContainer = vector<Key>>
9 class flat_set;

```

平序关联容器提供了与有序关联容器不同的时间和空间复杂性。与非平序关联容器相比，平序关联容器需要更少的内存，并且读取速度更快。下面的比较更详细地介绍了平序和非平序关联容器。

平序关联容器与非平序关联容器的比较

平序提供了更好的读取性能，例如遍历容器，并且需要更少的内存。它们还需要元素必须是可复制的或可移动的。并且平序支持随机访问迭代器。

若插入或删除元素，非平序可以提高写入性能。此外，非平序保证迭代器在插入或删除元素后仍然有效。并且非平序支持双向迭代器。

std::sorted_unique

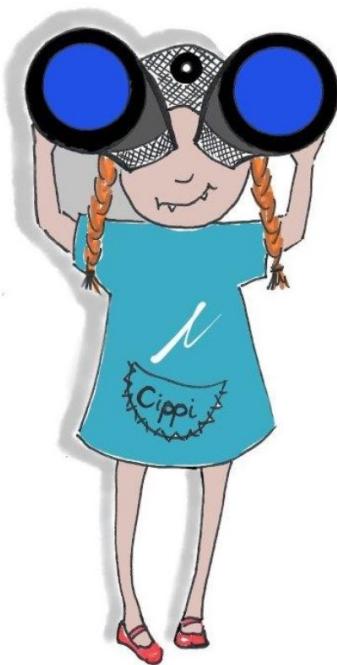
可以在构造函数调用或成员函数 insert 中，使用常量 std::sorted_unique 来指定元素已经排序。这提高了创建平序关联容器或插入元素的性能。

下面的代码片段从一个排序的初始化列表 {1,2,3,4,5} 创建一个 std::flat_map。

```
1 std::flat_map myFlatMap = { std::sorted_unique, {1, 2, 3, 4, 5}, {10, 11, 1, 5, -4}
↪   };
```

对非排序元素使用常量 std::sorted_unique 会产生未定义行为。

第 7 章 视图



Cippi 正在观察

7.1. 连续访问

`std::span` 表示引用连续对象序列的对象，`std::span`(有时也称为视图)从来不是所有者。这个连续的对象序列可以是普通的 C 数组、带大小的指针、`std::array`、`std::vector` 或 `std::string`，还可以访问其子序列。

`std::span` 不会衰变

当调用以 C 数组为参数的函数时，会发生衰变。该函数通过指向数组第一个元素的指针获取数组。C 数组到指针的转换很容易出错，因为 C 数组的所有长度信息都会丢失。

相反，`std::span` 知道其长度。

```
1 // copySpanArray.cpp
2 ...
3 #include <span>
4
5 template <typename T>
6 void copy_n(const T* p, T* q, int n) {}
7
8 template <typename T>
9 void copy(std::span<const T> src, std::span<T> des) {}
10
11 int arr1[] = {1, 2, 3};
```

```

12 int arr2[] = {3, 4, 5};
13
14 copy_n(arr1, arr2, 3); // (1)
15 copy<int>(arr1, arr2); // (2)

```

与 C 数组 (1) 相比，通过 std::span(2) 接受 C 数组的函数不需要显式的长度参数。

std::span 可以具有静态区段或动态区段，带有动态 extent 的 std::span 的实现包含指向其第一个元素和长度的指针。默认情况下，std::span 有一个动态范围：

std::span 的定义

```

1 template <typename T, std::size_t Extent = std::dynamic_extent>
2 class span;

```

下表给出了 std::span 的成员函数。

std::span sp 的成员函数

函数	描述
sp.front()	访问第一个元素。
sp.back()	访问最后一个元素。
sp[i]	访问第 i 个元素。
sp.data()	返回一个指向序列开头的指针。
sp.size_bytes()	以字节为单位返回序列的大小。
sp.empty()	如果序列为空则返回 true。
sp.first<count>() sp.first(count)	返回由序列的第一个 count 元素组成的子 span。
sp.last<count>() sp.last(count)	返回由序列的最后 count 个元素组成的子 span。
sp.subspan<first, count>() sp.subspan(first, count)	返回由 count 元素组成的子 span。

std::span 接受不同的参数

```

1 // printSpan.cpp
2 ...
3 #include <span>
4
5 void printMe(std::span<int> container) {
6     std::cout << "container.size(): " << container.size() << '\n';
7     for(auto e : container) std::cout << e << ' ';

```

```

8     std::cout << "\n\n";
9 }
10 std::cout << '\n';
11
12 int arr[] {1, 2, 3, 4};
13 printMe(arr); // (1)
14
15 std::vector vec{1, 2, 3, 4, 5};
16 printMe(vec); // (2)
17
18 std::array arr2{1, 2, 3, 4, 5, 6};
19 printMe(arr2); // (3)

```

std::span 可以用 C-array、std::vector(1) 或 std::array(2) 初始化。

```

C:\Users\seminar>printSpan.exe

container.size(): 4
1 2 3 4

container.size(): 5
1 2 3 4 5

container.size(): 6
1 2 3 4 5 6

```

自动减少 std::span 的大小

std::string_view 优于 std::span

std::string 是一个连续的字符序列，可以用 std::string 初始化 std::span。不过，应该选择 std::string_view，而不是 std::span，std::string_view 表示字符序列的视图，而不是 std::span 那样的对象序列。std::string_view 的接口类似于字符串，但是 std::span 的接口是通用的。

可以修改整个 span 或仅修改子 span。当修改一个 span 时，同时也修改了其引用的对象。

修改 std::span 引用的对象

```

1 // spanTransform.cpp
2 ...

```

```

3 #include <span>
4
5 std::vector vec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
6 printMe(vec);
7
8 std::span span1(vec); // (1)
9 std::span span2{span1.subspan(1, span1.size() - 2)}; // (2)
10 std::transform(span2.begin(), span2.end(), // (3)
11                 span2.begin(),
12                 [] (int i){ return i * i; });
13
14 printMe(vec);
15 printMe(span1);

```

我在前面的示例 quadSpan.cpp 中定义了 printMe 函数。span1 引用 std::vector vec(1)。相反，span2 只引用底层 vec 元素，不包括第一个和最后一个元素 (2)，将每个元素映射到它的平方只指向这些元素 (3)。

```

C:\Users\seminar>spanTransform.exe

container.size(): 10
1 2 3 4 5 6 7 8 9 10

container.size(): 10
1 4 9 16 25 36 49 64 81 10

container.size(): 10
1 4 9 16 25 36 49 64 81 10

C:\Users\seminar>

```

修改 std::span 引用的对象

7.2. 多维访问

std::mdspan 是一个连续对象序列的非拥有多维视图。通常，这种多维视图称为多维数组。对象的连续序列可以是 C 数组、带大小的指针、std::array、std::vector 或 std::string。

维度的数量和每个维度的大小决定了多维数组的形状。维度的数量称为秩，每个维度的大小扩展。std::mdspan 的大小，是所有不为 0 的维度的乘积，所以可以使用多维索引操作符 [] 访问 std::mdspan 的元素。

std::mdspan 的每个维度可以有一个静态区段或一个动态区段。静态范围意味着其长度在编译时指定；动态区段意味着其长度是在运行时指定的。

std::mdspan 的定义

```

1 template<
2   class T,
3   class Extents,
4   class LayoutPolicy = std::layout_right,
5   class AccessorPolicy = std::default_accessor<T>
6 > class mdspan;

```

- T: 连续的对象序列
- Extents: 指定维度的数量作为它们的大小; 每个维度可以有一个静态范围或一个动态范围
- LayoutPolicy: 指定访问底层内存的布局策略
- AccessorPolicy: 指定如何引用基础元素

由于 C++17 中的[类模板参数推演](#), [class template argument deduction \(CTAG\)](#), 编译器通常可以自动推断模板参数。

两个二维数组

```

1 // mdspan.cpp
2
3 #include <mdspan>
4 #include <iostream>
5 #include <vector>
6
7 int main() {
8
9     std::vector myVec{1, 2, 3, 4, 5, 6, 7, 8};
10
11    std::mdspan m{myVec.data(), 2, 4};
12    std::cout << "m.rank(): " << m.rank() << '\n';
13
14    for (std::size_t i = 0; i < m.extent(0); ++i) {
15        for (std::size_t j = 0; j < m.extent(1); ++j) {
16            std::cout << m[i, j] << ' ';
17        }
18        std::cout << '\n';
19    }
20
21    std::cout << '\n';
22
23    std::mdspan m2{myVec.data(), 4, 2};
24    std::cout << "m2.rank(): " << m2.rank() << '\n';
25    for (std::size_t i = 0; i < m2.extent(0); ++i) {
26        for (std::size_t j = 0; j < m2.extent(1); ++j) {
27            std::cout << m2[i, j] << ' ';
28        }
29        std::cout << '\n';
30    }

```

```
31  
32 }
```

这个例子中，我应用了三次类模板参数推导。第 9 行将其用于 std::vector，第 11 行和第 23 行将其用于 std::mdspan。第一个二维数组 m 的形状为 (2,4)，第二个二维数组 m2 的形状为 (4,2)。第 12 行和第 24 行显示了两个 std::mdspan 的秩。由于每个维度的范围 (第 14 行和第 15 行) 以及第 16 行中的索引操作符，可以直接遍历多维数组。

```
m.rank(): 2  
1 2 3 4  
5 6 7 8  
  
m2.rank(): 2  
1 2  
3 4  
5 6  
7 8
```

两个二维数组

若多维数组应该具有静态范围，则必须指定模板参数。

显式指定 std::mdspan 的模板参数

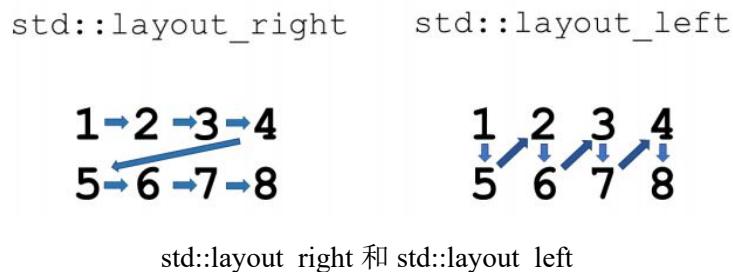
```
// staticDynamicExtent.cpp  
  
#include <mdspan>  
...  
  
std::mdspan<int, std::extents<std::size_t, 2, 4>> m{myVec.data()}; // (1)  
std::cout << "m.rank(): " << m.rank() << '\n';  
  
for (std::size_t i = 0; i < m.extent(0); ++i) {  
    for (std::size_t j = 0; j < m.extent(1); ++j) {  
        std::cout << m[i, j] << ' ';  
    }  
    std::cout << '\n';  
}  
  
std::mdspan<int, std::extents<std::size_t, std::dynamic_extent, std::dynamic_extent>>  
    m2{myVec.data(), 4, 2}; // (2)  
std::cout << "m2.rank(): " << m2.rank() << '\n';  
  
for (std::size_t i = 0; i < m2.extent(0); ++i) {  
    for (std::size_t j = 0; j < m2.extent(1); ++j) {
```

```

22     std::cout << m2[i, j] << ' ';
23 }
24 std::cout << '\n';
25 }
```

`staticDynamicExtent.cpp` 基于前面的程序 `mdspan.cpp`, 并产生相同的输出。不同之处在于, `std::mdspan m(1)` 具有静态区段。为了完整, `std::mdspan m2(2)` 具有动态范围, `m` 的形状是用模板参数指定的, 而 `m2` 的形状是用函数参数指定的。

`std::mdspan` 允许指定访问底层内存的布局策略。默认情况下, 使用 `std::layout_right`(C, C++ 或 Python 风格), 也可以指定 `std::layout_left` (Fortran 或 MATLAB 风格)。下面的图举例说明了访问 `std::mdspan` 的元素的顺序。



使用布局策略 `std::layout_right` 和 `std::layout_left` 遍历两个 `std::mdspan`, 展示了其差异。

使用 `std::mdspan`: `std::layout_right` 和 `std::layout_left`

```

1 // mdspanLayout.cpp
2 ...
3 #include <mdspan>
4
5 std::vector myVec{1, 2, 3, 4, 5, 6, 7, 8};
6
7 std::mdspan<int, // (1)
8     std::extents<std::size_t, std::dynamic_extent, std::dynamic_extent>,
9     std::layout_right> m2{myVec.data(), 4, 2};
10
11 std::cout << "m.rank(): " << m.rank() << '\n';
12
13 for (std::size_t i = 0; i < m.extent(0); ++i) {
14     for (std::size_t j = 0; j < m.extent(1); ++j) {
15         std::cout << m[i, j] << ' ';
16     }
17     std::cout << '\n';
18 }
19
20 std::cout << '\n';
21
22 std::mdspan<int,
23     std::extents<std::size_t, std::dynamic_extent, std::dynamic_extent>,
```

```

24     std::layout_left> m2{myVec.data(), 4, 2}; // (2)
25     std::cout << "m2.rank(): " << m2.rank() << '\n';
26
27     for (std::size_t i = 0; i < m2.extent(0); ++i) {
28         for (std::size_t j = 0; j < m2.extent(1); ++j) {
29             std::cout << m2[i, j] << ' ';
30         }
31         std::cout << '\n';
32     }

```

std::mdspan m 使用 std::layout_right(1)，另一个 std::mdspan std::layout_left(1)。通过类模板参数推导，std::mdspan(1) 的构造函数调用不需要显式的模板参数，相当于以下表达式:std::mdspan m2{myVec.data(), 4, 2}。

程序的输出，展示了两种不同的布局策略。

m.rank(): 2

1 2
3 4
5 6
7 8

m2.rank(): 2

1 5
2 6
3 7
4 8

std::mdspan 和 std::layout_left

下表给出了 std::mdspan 接口的概述。

std::mdspan md 的成员函数

函数	描述
md[ind]	访问第 ind 个元素。
md.size	返回多维数组的大小。
md.rank	返回多维数组的维度。
md.extents(i)	返回第 i 维的大小。
md.data_handle	返回一个指向连续元素序列的指针。

第 8 章 迭代器



Cippi 正在大迈步

一方面，[迭代器](#)是表示容器中位置的指针的泛化。另一方面，它们在容器中提供了强大的迭代和随机访问功能。

迭代器是泛型容器和标准模板库的泛型算法之间的粘合剂。

迭代器支持以下操作：

- *: 返回当前位置的元素
- ==, !=: 比较两个位置
- =: 给迭代器赋值

基于范围的 for 循环会隐式地使用迭代器。

因为不检查迭代器，所以其有和指针一样的问题。

```
1 std::vector<int> vec{1, 23, 3, 3, 3, 4, 5};  
2 std::deque<int> deque;  
3  
4 // Start iterator bigger than end iterator  
5 std::copy(vec.begin() + 2, vec.begin(), deque.begin());  
6  
7 // Target container too small  
8 std::copy(vec.begin(), vec.end(), deque.end());
```

8.1. 类型

根据功能可以对迭代器进行分类。迭代器的类别取决于所使用的容器类型。C++ 有前向、双向、随机访问和连续迭代器。使用前向迭代器，可以使用双向迭代器在两个方向上向前迭代容器。随机

访问迭代器允许它直接访问任意元素。特别是，随机迭代器支持迭代器算术和排序比较(例如: $<$)。连续迭代器是一种随机访问迭代器，并要求容器的元素连续地存储在内存中。

下表是容器及其迭代器类别的表示。双向迭代器包括正向迭代器的功能。随机访问迭代器包括前向和双向迭代器的功能。It 和 It2 是迭代器，n 是自然数。

容器迭代器的类别

迭代器类别	属性	容器
前向迭代器	++It , $\text{It}++$, $*\text{It}$ $\text{It} == \text{It2}$, $\text{It} != \text{It2}$	<code>std::unordered_set</code> <code>std::unordered_map</code> <code>std::unordered_multiset</code> <code>std::unordered_multimap</code> <code>std::forward_list</code>
双向迭代器	$-\text{It}$, $\text{It}-$	<code>std::set</code> <code>std::map</code> <code>std::multiset</code> <code>std::multimap</code> <code>std::list</code>
随机访问迭代器	$\text{It}[i]$ $\text{It} += n$, $\text{It} -= n$ $\text{It} + n$, $\text{It} - n$ $n + \text{It}$ $\text{It} - \text{It2}$	<code>std::deque</code>
连续迭代器	$\text{It} < \text{It2}$, $\text{It} \leq \text{It2}$, $\text{It} > \text{It2}$ $\text{It} \geq \text{It2}$	<code>std::array</code> <code>std::vector</code> <code>std::string</code>

输入迭代器和输出迭代器是特殊的前向迭代器：只能对所指向的元素读写一次。

8.2. 创建迭代器

每个容器根据请求生成合适的迭代器。例如，`std::unordered_map` 会生成常量和非常量前向迭代器。

```
1 std::unordered_map<std::string, int>::iterator unMapIt= unordMap.begin();  
2 std::unordered_map<std::string, int>::iterator unMapIt= unordMap.end();  
3  
4 std::unordered_map<std::string, int>::const_iterator unMapIt= unordMap.cbegin();  
5 std::unordered_map<std::string, int>::const_iterator unMapIt= unordMap.cend();
```

此外，`std::map` 还支持后向迭代器：

```
1 std::map<std::string, int>::reverse_iterator mapIt= map.rbegin();
2 std::map<std::string, int>::reverse_iterator mapIt= map.rend();
3
4 std::map<std::string, int>::const_reverse_iterator mapIt= map.crbegin();
5 std::map<std::string, int>::const_reverse_iterator mapIt= map.crend();
```

使用 auto 定义迭代器

迭代器的定义有些麻烦，使用自动类型推断可将减少所需的代码量。

```
1 std::map<std::string, int>::const_reverse_iterator
2 mapIt= map.crbegin();
3 auto mapIt2= map.crbegin();
```

创建迭代器

```
1 // iteratorCreation.cpp
2 ...
3 using namespace std;
4 ...
5 map<string, int> myMap{{"Rainer", 1966}, {"Beatrix", 1966}, {"Juliette", 1997},
6 {"Marius", 1999}};
7
8 auto endIt= myMap.end();
9 for (auto mapIt= myMap.begin(); mapIt != endIt; ++mapIt)
10 cout << "(" << mapIt->first << "," << mapIt->second << ")";
11 // {Beatrix,1966},{Juliette,1997},{Marius,1999},{Rainer,1966}
12
13 vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9};
14 vector<int>::const_iterator vecEndIt= myVec.end();
15 vector<int>::iterator vecIt;
16 for (vecIt= myVec.begin(); vecIt != vecEndIt; ++vecIt) cout << *vecIt << " ";
17 // 1 2 3 4 5 6 7 8 9
18
19 vector<int>::const_reverse_iterator vecEndRevIt= myVec.rend();
20 vector<int>::reverse_iterator revIt;
21 for (revIt= myVec.rbegin(); revIt != vecEndRevIt; ++revIt) cout << *revIt << " ";
22 // 9 8 7 6 5 4 3 2 1
```

8.3. 实用的功能

全局函数 std::begin、std::end、std::prev、std::next、std::distance 和 std::advance 使处理迭代器变得容易得多。只有 std::prev 函数需要双向迭代器。所有函数都需要头文件 <iterator>，下面表格是一

个简单的概述。

迭代器的辅助函数

辅助函数	描述
std::begin(cont)	向容器 cont 返回一个起始迭代器。
std::end(cont)	向容器 cont 返回一个结束迭代器。
std::rbegin(cont)	向容器 cont 返回一个反向起始迭代器。
std::rend(cont)	向容器 cont 返回一个反向结束迭代器。
std::cbegin(cont)	向容器 cont 返回一个常量起始迭代器。
std::cend(cont)	向容器 cont 返回一个常量结束迭代器。
std::crbegin(cont)	向容器 cont 返回一个反向常量起始迭代器。
std::crend(cont)	向容器 cont 返回一个反向常量结束迭代器。
std::prev(it)	返回一个迭代器，该迭代器指向它之前的位置。
std::next(it)	返回一个迭代器，该迭代器指向它之后的位置。
std::distance(fir, sec)	返回 fir 和 sec 之间的元素个数。
std::advance(it, n)	将迭代器前进或后退 n 个位置。

下面是辅助函数的应用。

迭代器的辅助函数

```
1 // iteratorUtilities.cpp
2 ...
3 #include <iterator>
4 ...
5 using std::cout;
6
7 std::unordered_map<std::string, int> myMap{{"Rainer", 1966}, {"Beatrix", 1966},
8 {"Juliette", 1997}, {"Marius", 1999}};
9
10 for (auto m: myMap) cout << "{" << m.first << "," << m.second << "}" " ;
11 // {Juliette,1997},{Marius,1999},{Beatrix,1966},{Rainer,1966}
12
13 auto mapItBegin= std::begin(myMap);
14 cout << mapItBegin->first << " " << mapItBegin->second; // Juliette 1997
15
16 auto mapIt= std::next(mapItBegin);
17 cout << mapIt->first << " " << mapIt->second; // Marius 1999
18 cout << std::distance(mapItBegin, mapIt); // 1
19
20 std::array<int, 10> myArr{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
21 for (auto a: myArr) std::cout << a << " "; // 0 1 2 3 4 5 6 7 8 9
22
23 auto arrItEnd= std::end(myArr);
```

```

24 auto arrIt= std::prev(arrItEnd);
25
26 cout << *arrIt << '\n'; // 9
27
28 std::advance(arrIt, -5);
29 cout << *arrIt; // 4

```

8.4. 适配器

迭代器适配器允许在插入模式或流中使用迭代器，其需要头文件 `<iterator>`。

插入迭代器

使用三个插入迭代器 `std::front_inserter`、`std::back_inserter` 和 `std::inserter`，可以分别在容器的开头、结尾或任意位置插入元素。这三个辅助函数，会将其功能映射到容器内容的底层成员函数上，元素的内存将自动提供。

下表提供了两条信息：容器的哪些成员函数是内部使用的，哪些迭代器可以使用，这取决于容器的类型。

三个插入迭代器

名称	内部使用的成员函数	容器
<code>std::front_inserter(val)</code>	<code>cont.push_front(val)</code>	<code>std::deque</code> <code>std::list</code>
<code>std::back_inserter(val)</code>	<code>cont.push_back(val)</code>	<code>std::vector</code> <code>std::deque</code> <code>std::list</code> <code>std::string</code>
<code>std::inserter(val, pos)</code>	<code>con.insert(pos, val)</code>	<code>std::vector</code> <code>std::deque</code> <code>std::list</code> <code>std::string</code> <code>std::map</code> <code>std::set</code>

可以将 STL 中的算法与三个插入迭代器结合使用。

```

1 #include <iterator>
2 ...
3 std::deque<int> deq{5, 6, 7, 10, 11, 12};
4 std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
5
6 std::copy(std::find(vec.begin(), vec.end(), 13),

```

```

7     vec.end(), std::back_inserter(deq));
8
9 for (auto d: deq) std::cout << d << " ";
10 // 5 6 7 10 11 12 13 14 15
11
12 std::copy(std::find(vec.begin(), vec.end(), 8),
13 std::find(vec.begin(), vec.end(), 10),
14 std::inserter(deq,
15 std::find(deq.begin(), deq.end(), 10)));d
16 for (auto d: deq) std::cout << d << " ";
17 // 5 6 7 8 9 10 11 12 13 14 15
18
19 std::copy(vec.rbegin() + 11, vec.rend(),
20 std::front_inserter(deq));
21 for (auto d: deq) std::cout << d << " ";
22 // 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

```

流迭代器

流迭代器适配器可以使用流作为数据源或数据接收器。C++ 提供了两个函数来创建 `istream` 迭代器和两个函数来创建 `ostream` 迭代器。创建的 `istream` 迭代器的行为类似于输入迭代器，`ostream` 迭代器的行为类似于插入迭代器。

四个流迭代器

功能	描述
<code>std::istream_iterator<T></code>	创建 <code>end-if-stream</code> 迭代器。
<code>std::istream_iterator<T>(istream)</code>	为 <code>istream</code> 创建 <code>istream</code> 迭代器。
<code>std::ostream_iterator<T>(ostream)</code>	为 <code>ostream</code> 创建 <code>ostream</code> 迭代器。
<code>std::ostream_iterator<T>(ostream, delim)</code>	用分隔符 <code>delim</code> 为 <code>ostream</code> 创建 <code>ostream</code> 迭代器。

有了流迭代器适配器，就可以直接从流中读取或写入流。

下面的交互式程序段在一个无限循环中从 `std::cin` 中读取自然数，并将其压入 `vector myIntVec`。若输入不是自然数，则会在输入流中出现错误。`myIntVec` 中的所有数字都复制到 `std::cout` 中，中间用: 分隔。

```

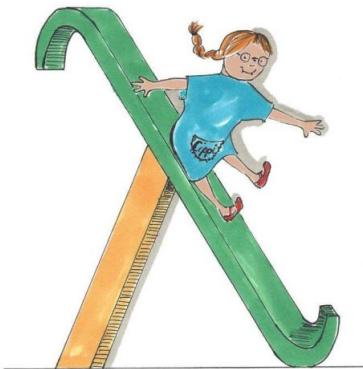
1 #include <iostream>
2 ...
3 std::vector<int> myIntVec;
4 std::istream_iterator<int> myIntStreamReader(std::cin);
5 std::istream_iterator<int> myEndIterator;
6
7 // Possible input
8 // 1

```

```
9 // 2
10 // 3
11 // 4
12 // z
13 while(myIntStreamReader != myEndIterator) {
14     myIntVec.push_back(*myIntStreamReader);
15     ++myIntStreamReader;
16 }
17
18 std::copy(myIntVec.begin(), myIntVec.end(),
19           std::ostream_iterator<int>(std::cout, ":"));
```

// 1:2:3:4:

第 9 章 可调用单元



Cippi 从滑梯上滑下来

本章有意不做详尽介绍

这本书是关于 C++ 标准库的，所以不会深入可调用单元的细节。我提供了在标准模板库算法中正确使用它们所需的尽可能多的信息。关于可调用单元的详尽讨论，应该成为关于 C++ 核心语言书籍的一部分。

许多 STL 算法和容器都可以用可调用单元（简称可调用）进行参数化。可调用对象的行为类似于函数。不仅是这些函数，还有函数对象和 Lambda 函数。谓词是返回布尔值作为结果的特殊函数。若一个谓词只有一个参数，则称为一元谓词。若一个谓词有两个参数，则称为二元谓词。函数也是如此，接受一个参数的函数是一元函数；接受两个参数的函数是二元函数。

要更改容器的元素，算法应该通过引用来获取

可调用对象可以通过值或引用从其容器接收参数。要修改容器元素，必须直接寻址，因此可调用对象必须通过引用获取。

9.1. 函数

函数是最简单的可调用对象——除了静态变量——可以没有状态。由于函数的定义通常与它的使用广泛分离，甚至在不同的翻译单元中，编译器优化代码的机会较少。

```
1 void square(int& i){ i = i*i; }
2 std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
3
4 std::for_each(myVec.begin(), myVec.end(), square);
5 for (auto v: myVec) std::cout << v << " "; // 1 4 9 16 25 36 49 64 81 100
```

9.2. 函数对象

首先，不要叫他们**函数**，这是范畴论中定义很好的术语。

函数对象是行为类似函数的对象。由于函数对象是对象，它们可以有属性，因为调用操作符实现了，所以也可以有状态。

```
1 struct Square{  
2     void operator()(int& i){i= i*i;}  
3 };  
4  
5 std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
6  
7 std::for_each(myVec.begin(), myVec.end(), Square());  
8  
9 for (auto v: myVec) std::cout << v << " "; // 1 4 9 16 25 36 49 64 81 100
```

实例化函数对象以供使用

在算法:std::for_each(myVec.begin(), myVec.end(), Square) 中只使用函数对象的名称 (Square)，而不使用函数对象的实例 (Square()) 是一个常见的错误。当然，这是一个错误。必须使用实例:std::for_each(myVec.begin(), myVec.end(), Square())

预定义函数对象

C++ 提供了一系列预定义的函数对象，需要头文件 <functional>。这些预定义的函数对象有助于更改容器的默认行为。例如，有序关联容器的键在默认情况下使用预定义的函数对象 std::less 进行排序，也可以用 std::greater 代替：

```
1 std::map<int, std::string> myDefaultMap; // std::less<int>  
2 std::map<int, std::string, std::greater<int> mySpecialMap>; // std::greater<int>
```

标准模板库中有用于算术、逻辑、位运算、求反和比较的函数对象。

预定义功能对象

函数对象	代表
否定	std::negate<T>()
算术	std::plus<T>(), std::minus<T>() std::multiplies<T>(), std::divides<T>() std::modulus<T>()

比较	std::equal_to<T>(), std::not_equal_to<T>(); std::less<T>(), std::greater<T>(); std::less_equal<T>(), std::greater_equal<T>()
逻辑	std::logical_not<T>() std::logincl_and<T>(), std::logical_or<T>()
位域	std::bit_and<T>(), std::bit_or<T>(); std::bit_xor<T>()

9.3. Lambda 函数

[Lambda 函数](#)提供就地功能。Lambda 函数可以通过值或引用来接收它们的参数。它们可以通过值、引用和 C++14 的移动来捕获环境。编译器获得了很多洞察力，并具有出色的优化潜力。

```

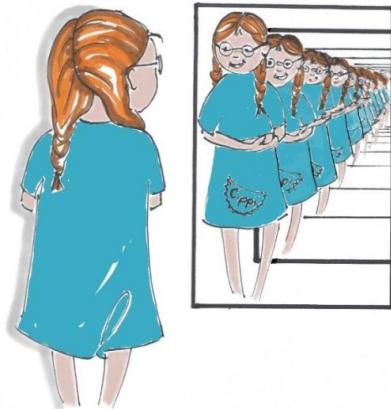
1 std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
2 std::for_each(myVec.begin(), myVec.end(), [] (int& i) { i = i*i; });
3 // 1 4 9 16 25 36 49 64 81 100

```

Lambda 函数应该是首选

若可调用对象的功能简短且不言自明，请使用 Lambda 函数。Lambda 函数通常更快、更容易理解。

第 10 章 算法



Cippi 的自我复制

标准模板库有许多[算法](#)来处理容器及其元素。由于算法是函数模板，它们与容器元素的类型无关。容器和算法之间的粘合剂是迭代器。若容器支持 STL 容器的接口，则可以将算法应用于自定义容器。

泛型编程与算法

```
1 // algorithm.cpp
2 ...
3 #include <algorithm>
4 ...
5 template <typename Cont, typename T>
6 void doTheSame(Cont cont, T t){
7     for (const auto c: cont) std::cout << c << " ";
8     std::cout << cont.size() << '\n';
9     std::reverse(cont.begin(), cont.end());
10    for (const auto c: cont) std::cout << c << " ";
11    std::reverse(cont.begin(), cont.end());
12    for (const auto c: cont) std::cout << c << " ";
13    auto It= std::find(cont.begin(), cont.end(), t);
14    std::reverse(It, cont.end());
15    for (const auto c: cont) std::cout << c << " ";
16 }
17
18 std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
19 std::deque<std::string> myDeq({"A", "B", "C", "D", "E", "F", "G", "H", "I"});
20 std::list<char> myList({'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'});
21
22 doTheSame(myVec, 5);
23 // 1 2 3 4 5 6 7 8 9 10
24 // 10
25 // 10 9 8 7 6 5 4 3 2 1
26 // 1 2 3 4 5 6 7 8 9 10
```

```

27 // 1 2 3 4 10 9 8 7 6 5
28
29 doTheSame (myDeq, "D");
// A B C D E F G H I
// 9
// I H G F E D C B A
// A B C D E F G H I
// A B C I H G F E D
30
31
32
33
34
35
36 doTheSame (myList, 'd');
// a b c d e f g h
// 8
// h g f e d c b a
// a b c d e f g h
// a b c h g f e d
37
38
39
40
41

```

10.1. 约定

使用算法时，你必须牢记一些规则。

算法在不同的头文件中定义。

<algorithm>: 包含通用算法。<numeric>: 包含数值算法。

许多算法的名称后缀都是_if 和_copy。

_if: 该算法可以通过谓词参数化。_copy: 该算法将其元素复制到另一个范围。

像 auto num= std::count(InpIt first, InpIt last, const T& val) 这样的算法返回与 val 相等的元素数量。num 类型为 iterator_traits<InpIt>::difference_type，可以保证 num 足以保存结果。由于 auto 的自动返回类型推导，编译器将提供正确的类型。

若容器使用了额外的范围，则其必须是有效的

算法 std::copy_if 使用迭代器到目标范围的开始。此目标范围必须有效。

算法的命名约定

我对参数的类型和算法的返回类型使用了一些命名约定，以便它们更容易阅读。

算法的签名

名称	描述
InIt	输入迭代器
FedIt	前向迭代器

BiIt	双向迭代器
UnFunc	一元可调用
BiFunc	二元可调用
UnPre	一元谓词
BiPre	二元谓词
Search	searcher封装了搜索算法。
ValType	从输入范围，自动推导出值类型。
Num	<code>typename std::iterator_traits<ForwardIt>::difference_type</code>
ExePol	执行策略

10.2. 迭代器是粘合剂

迭代器定义算法工作的容器的范围。其描述了一个半开放的范围。在半开范围内，开始迭代器指向开始，结束迭代器指向范围后的一个位置。

迭代器可以根据它们的功能进行分类，算法为迭代器提供条件。与 `std::rotate` 的情况类似，通常前向迭代器就足够了。但 `std::reverse` 不常用，因为 `std::reverse` 需要双向迭代器。

10.3. 顺序，并行或并行执行与先向量化

使用 C++17 中的执行策略，可以指定算法应该顺序运行、并行运行还是与向量化并行运行。

并行 STL 的可用性

2023 年，只有微软编译器本地实现了 STL 算法的并行版本。对于 GCC 或 Clang 编译器，必须安装并使用[Threading Building Blocks](#)。TBB 是 Intel 为在多核处理器上并行编程而开发的 C++ 模板库。

执行策略

策略标签指定算法应该顺序运行、并行运行还是与向量化并行运行。

- `std::execution::seq`: 串行运行算法
- `std::execution::par`: 多个线程上并行运行算法
- `std::execution::par_unseq`: 多个线程上并行运行算法，并允许各个循环交错；允许有 SIMD(单指令多数据) 扩展的向量化版本。

下面的代码段显示了所有执行策略。

执行策略

```
1 std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9};
2
3 // standard sequential sort
4 std::sort(v.begin(), v.end());
5
6 // sequential execution
7 std::sort(std::execution::seq, v.begin(), v.end());
8
9 // permitting parallel execution
10 std::sort(std::execution::par, v.begin(), v.end());
11
12 // permitting parallel and vectorized execution
13 std::sort(std::execution::par_unseq, v.begin(), v.end());
```

该示例表明，可以在没有执行策略的情况下使用 `std::sort` 的重载。在 C++17 中，可以显式指定应该使用顺序、并行还是并行和向量化版本。

并行和向量化执行

算法是否以并行和向量化的方式运行取决于许多因素，这取决于 CPU 和操作系统是否支持 SIMD 指令。此外，还取决于编译器和编译代码的优化级别。

下面的示例展示了一个用于创建新 `vector` 的简单循环。

```
1 const int SIZE= 8;
2
3 int vec[] = {1, 2, 3, 4, 5, 6, 7, 8};
4 int res[] = {0, 0, 0, 0, 0, 0, 0, 0};
5
6 int main() {
7     for (int i = 0; i < SIZE; ++i) {
8         res[i] = vec[i] + 5;
9     }
10 }
```

表达式 `res[i] = vec[i] + 5` 是这个小示例中的关键行。有了[Compiler Explorer](#)，我们可以仔细查看由 x86-64 clang 3.6 生成的汇编指令。

无优化

这是汇编说明，每个加法都是顺序完成的。

```
movslq -8(%rbp), %rax  
movl vec(,%rax,4), %ecx  
addl $5, %ecx  
movslq -8(%rbp), %rax  
movl %ecx, res(,%rax,4)
```

最大化优化

使用最高的优化级别-O3，可以使用特殊的寄存器，例如可以保存 128 位或 4 ‘int’ 的 xmm0，在四个向量元素上并行地进行加法运算。

```
movdqa .LCPI0_0(%rip), %xmm0 # xmm0 = [5,5,5,5]  
movdqa vec(%rip), %xmm1  
paddd %xmm0, %xmm1  
movdqa %xmm1, res(%rip)  
paddd vec+16(%rip), %xmm0  
movdqa %xmm0, res+16(%rip)  
xorl %eax, %eax
```

有 77 种 STL 算法可以通过执行策略进行参数化。

具有并行版本的算法

下面是 77 种并行化的算法。

77 种并行化的算法

std::adjacent_difference	std::adjacent_find	std::all_of
std::any_of	std::copy	std::copy_if
std::copy_n	std::count	std::count_if
std::equal	std::exclusive_scan	std::fill
std::fill_n	std::find	std::find_end
std::find_first_of	std::find_if	std::find_if_not
std::for_each	std::for_each_n	std::generate
std::inner_product	std::includes	std::inclusive_scan
std::is_heap_until	std::inplace_merge	std::is_heap
std::is_sorted_until	std::is_partitioned	std::is_sorted
std::merge	std::lexicographical_compare	std::max_element
std::mismatch	std::min_element	std::minmax_element
std::nth_element	std::move	std::none_of
std::partition	std::partial_sort	std::partial_sort_copy

std::remove	std::partition_copy	std::reduce
std::remove_if	std::remove_copy	std::remove_copy_if
std::replace_copy_if	std::replace	std::replace_copy
std::replace_copy	std::replace_if	std::reverse
std::reverse_copy	std::rotate	std::rotate_copy
std::search	std::search_n	std::set_difference
std::set_intersection	std::set_symmetric_difference	std::set_union
std::sort	std::stable_partition	std::stable_sort
std::swap_ranges	std::transform	std::transform_exclusive_scan
std::transform_inclusive_scan	std::transform_reduce	std::uninitialized_copy
std::uninitialized_copy_n	std::uninitialized_fill	std::uninitialized_fill_n
std::unique	std::unique_copy	

constexpr 容器和算法

C++20 支持 `constexpr` 容器 `std::vector` 和 `std::string`。`constexpr` 表示，可以在编译时应用这两个容器的成员函数。此外，标准模板库的超过 100 个算法可声明为 `constexpr`。

10.4. `for_each`

`std::for_each` 对其范围内的每个元素应用一元可调用对象，输入迭代器可提供范围。

```
1 UnFunc std::for_each(InpIt first, InpIt second, UnFunc func)
2 void std::for_each(ExePol pol, FwdIt first, FwdIt second, UnFunc func)
```

没有显式执行策略的情况下使用 `std::for_each` 是一个特殊的算法，可返回其可调用的参数。若使用函数对象调用 `std::for_each`，则可以将函数调用的结果直接存储在函数对象中。

```
1 InpIt std::for_each_n(InpIt first, Size n, UnFunc func) FwdIt std::for_each_n(ExePol
2   → pol, FwdIt first, Size n, UnFunc func)
```

`std::for_each_n` 是 C++17 的新特性，并对其范围的前 `n` 个元素应用一元可调用对象，输入迭代器和大小提供了范围。

`std::for_each`

```
1 // forEach.cpp
2 ...
3 #include <algorithm>
4 ...
```

```

5  template <typename T>
6  class ContInfo{
7  public:
8      void operator() (T t) {
9          num++;
10         sum+= t;
11     }
12     int getSum() const{ return sum; }
13     int getSize() const{ return num; }
14     double getMean() const{
15         return static_cast<double>(sum) / static_cast<double>(num);
16     }
17 private:
18     T sum{0};
19     int num{0};
20 };
21
22 std::vector<double> myVec{1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9};
23 auto vecInfo= std::for_each(myVec.begin(), myVec.end(), ContInfo<double>());
24
25 std::cout << vecInfo.getSum() << '\n'; // 49
26 std::cout << vecInfo.getSize() << '\n'; // 9
27 std::cout << vecInfo.getMean() << '\n'; // 5.5
28
29 std::array<int, 100> myArr{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
30 auto arrInfo= std::for_each(myArr.begin(), myArr.end(), ContInfo<int>());
31
32 std::cout << arrInfo.getSum() << '\n'; // 55
33 std::cout << arrInfo.getSize() << '\n'; // 100
34 std::cout << arrInfo.getMean() << '\n'; // 0.55

```

10.5. 不可修改算法

不可修改算法是用于搜索和计数元素的算法，也可以检查范围上的属性、比较范围或在范围内搜索范围。

搜索元素

可以用三种不同的方式搜索元素。

返回一个范围内的元素：

```

1 InpIt find(InpIt first, InpI last, const T& val)
2 InpIt find(ExePol pol, FwdIt first, FwdIt last, const T& val)
3
4 InpIt find_if(InpIt first, InpIt last, UnPred pred)
5 InpIt find_if(ExePol pol, FwdIt first, FwdIt last, UnPred pred)
6

```

```
7 InpIt find_if_not(InpIt first, InpIt last, UnPred pre)
8 InpIt find_if_not(ExePol pol, FwdIt first, FwdIt last, UnPred pre)
```

返回一个范围的第一个元素:

```
1 FwdIt1 find_first_of(InpIt1 first1, InpIt1 last1,
2                     FwdIt2 first2, FwdIt2 last2)
3 FwdIt1 find_first_of(ExePol pol, FwdIt1 first1, FwdIt1 last1,
4                     FwdIt2 first2, FwdIt2 last2)
5
6 FwdIt1 find_first_of(InpIt1 first1, InpIt1 last1,
7                     FwdIt2 first2, FwdIt2 last2, BiPre pre)
8 FwdIt1 find_first_of(ExePol pol, FwdIt1 first1, FwdIt1 last1,
9                     FwdIt2 first2, FwdIt2 last2, BiPre pre)
```

返回范围内相同的相邻元素:

```
1 FwdIt adjacent_find(FwdIt first, FwdIt last)
2 FwdIt adjacent_find(ExePol pol, FwdIt first, FwdIt last)
3
4 FwdIt adjacent_find(FwdIt first, FwdIt last, BiPre pre)
5 FwdIt adjacent_find(ExePol pol, FwdIt first, FwdIt last, BiPre pre)
```

这些算法需要输入或前向迭代器作为参数，并在成功找到元素时返回一个迭代器。若搜索不成功，则返回结束迭代器。

std::find, std::find_if, std::find_if_not, std::find_of 和 std::adjacent_fint

```
1 // find.cpp
2 ...
3 #include <algorithm>
4 ...
5 using namespace std;
6
7 bool isVowel(char c){
8     string myVowels{"aeiouäöü"};
9     set<char> vowels(myVowels.begin(), myVowels.end());
10    return (vowels.find(c) != vowels.end());
11 }
12
13 list<char> myCha{'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'};
14 int cha[] = {'A', 'B', 'C'};
15
16 cout << *find(myCha.begin(), myCha.end(), 'g'); // g
17 cout << *find_if(myCha.begin(), myCha.end(), isVowel); // a
18 cout << *find_if_not(myCha.begin(), myCha.end(), isVowel); // b
19
```

```

20 auto iter= find_first_of(myCha.begin(), myCha.end(), cha, cha + 3);
21 if (iter == myCha.end()) cout << "None of A, B or C." // None of A, B or C.
22 auto iter2= find_first_of(myCha.begin(), myCha.end(), cha, cha+3,
23 [](char a, char b){ return toupper(a) == toupper(b); });
24
25 if (iter2 != myCha.end()) cout << *iter2; // a
26 auto iter3= adjacent_find(myCha.begin(), myCha.end());
27 if (iter3 == myCha.end()) cout << "No same adjacent chars.";
28 // No same adjacent chars.
29
30 auto iter4= adjacent_find(myCha.begin(), myCha.end(),
31 [](char a, char b){ return isVowel(a) == isVowel(b); });
32 if (iter4 != myCha.end()) cout << *iter4; // b

```

元素计数

可以使用 STL 对元素进行计数，无论是否使用谓词。

返回元素的个数：

```

1 Num count(InpIt first, InpIt last, const T& val)
2 Num count(ExePol pol, FwdIt first, FwdIt last, const T& val)
3
4 Num count_if(InpIt first, InpIt last, UnPred pre)
5 Num count_if(ExePol pol, FwdIt first, FwdIt last, UnPred pre)

```

计数算法接受输入迭代器作为参数，并返回与 val 或谓词匹配的元素个数。

std::count 和 *std::count_if*

```

1 // count.cpp
2 ...
3 #include <algorithm>
4 ...
5 std::string str{"abcdabAAAaefaBqeaaBCQEaadsfdewAAQAaafbd"};
6 std::cout << std::count(str.begin(), str.end(), 'a'); // 9
7 std::cout << std::count_if(str.begin(), str.end(),
8     [] (char a){ return std::isupper(a); }); // 12

```

范围的检查条件

三个函数 *std::all_of*、*std::any_of* 和 *std::none_of* 可回答这个问题，若一个范围的所有、至少一个或没有元素满足条件。这些函数需要作为参数输入迭代器和一元谓词，并返回一个布尔值。

检查范围内的所有元素是否满足条件：

```
1 bool all_of(InpIt first, InpIt last, UnPre pre)
2 bool all_of(ExePol pol, FwdIt first, FwdIt last, UnPre pre)
```

检查范围内是否至少有一个元素满足条件:

```
1 bool any_of(InpIt first, InpIt last, UnPre pre)
2 bool any_of(ExePol pol, FwdIt first, FwdIt last, UnPre pre)
```

检查范围内是否没有元素满足条件:

```
1 bool none_of(InpIt first, InpIt last, UnPre pre)
2 bool none_of(ExePol pol, FwdIt first, FwdIt last, UnPre pre)
```

如前所述, 示例如下:

std::all_of, std::any_of 和 std::none_of

```
1 // allAnyNone.cpp
2 ...
3 #include <algorithm>
4 ...
5 auto even= [](int i){ return i%2; };
6 std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9};
7 std::cout << std::any_of(myVec.begin(), myVec.end(), even); // true
8 std::cout << std::all_of(myVec.begin(), myVec.end(), even); // false
9 std::cout << std::none_of(myVec.begin(), myVec.end(), even); // false
```

范围比较

使用 `std::equal`, 可以比较相等的范围。使用 `std::lexicographical_compare`、`std::lexicographical_compare_three_way` 和 `std::mismatch`, 可以用于较小的范围。

检查两个范围是否相等:

```
1 bool equal(InpIt first1, InpIt last1, InpIt first2)
2 bool equal(ExePol pol, FwdIt first1, FwdIt last1, FwdIt first2)
3
4 bool equal(InpIt first1, InpIt last1, InpIt first2, BiPre pred)
5 bool equal(ExePol pol, FwdIt first1, FwdIt last1, FwdIt first2, BiPre pred)
6
7 bool equal(InpIt first1, InpIt last1,
8           InpIt first2, InpIt last2)
9 bool equal(ExePol pol, FwdIt first1, FwdIt last1,
10          FwdIt first2, FwdIt last2)
```

```
12 bool equal(InpIt first1, InpIt last1,  
13     InpIt first2, InpIt last2, BiPre pred)  
14 bool equal(ExePol pol, FwdIt first1, FwdIt last1,  
15     FwdIt first2, FwdIt last2, BiPre pred)
```

检查第一个范围是否小于第二个范围:

```
1 bool lexicographical_compare(InpIt first1, InpIt last1,  
2     InpIt first2, InpIt last2)  
3 bool lexicographical_compare(ExePol pol, FwdIt first1, FwdIt last1,  
4     FwdIt first2, FwdIt last2)  
5  
6 bool lexicographical_compare(InpIt first1, InpIt last1,  
7     InpIt first2, InpIt last2, BiPre pred)  
8 bool lexicographical_compare(ExePol pol, FwdIt first1, FwdIt last1,  
9     FwdIt first2, FwdIt last2, BiPre pred)
```

检查第一个范围是否小于第二个范围。应用[三路比较](#)，可返回最适用比较类别的类型。

```
1 bool lexicographical_compare_three_way(InpIt first1, InpIt last1,  
2     InpIt first2, InpIt last2)  
3 bool lexicographical_compare_three_way(InpIt first1, InpIt last1,  
4     InpIt first2, InpIt last2,  
5     BiPre pre)
```

查找两个范围不相等的第一个位置:

```
1 pair<InpIt, InpIt> mismatch(InpIt first1, InpIt last1,  
2     InpIt first2)  
3 pair<InpIt, InpIt> mismatch(ExePol pol, FwdIt first1, FwdIt last1,  
4     FwdIt first2)  
5  
6 pair<InpIt, InpIt> mismatch(InpIt first1, InpIt last1,  
7     InpIt first2, BiPre pred)  
8 pair<InpIt, InpIt> mismatch(ExePol pol, FwdIt first1, FwdIt last2,  
9     FwdIt first2, BiPre pred)  
10  
11 pair<InpIt, InpIt> mismatch(InpIt first1, InpIt last1,  
12     InpIt first2, InpIt last2)  
13 pair<InpIt, InpIt> mismatch(ExePol pol, FwdIt first1, FwdIt last1,  
14     FwdIt first2, FwdIt last2)  
15  
16 pair<InpIt, InpIt> mismatch(InpIt first1, InpIt last1,  
17     InpIt first2, InpIt last2, BiPre pred)  
18 pair<InpIt, InpIt> mismatch(ExePol pol, FwdIt first1, FwdIt last1,  
19     FwdIt first2, FwdIt last2, BiPre pred)
```

这些算法接受输入迭代器和最终的二元谓词。`std::mismatch` 返回一对输入迭代器。`pa.first` 保存第一个不相等元素的输入迭代器。`pa.second` 保存第二个范围对应的输入迭代器。若两个范围相同，则得到两个 `end` 迭代器。

`std::equal`, `std::lexicographical_compare` 和 `std::mismatch`

```
1 // equalLexicographicalMismatch.cpp
2 ...
3 #include <algorithm>
4 ...
5 using namespace std;
6
7 string str1{"Only For Testing Purpose." };
8 string str2{"only for testing purpose." };
9 cout << equal(str1.begin(), str1.end(), str2.begin()); // false
10 cout << equal(str1.begin(), str1.end(), str2.begin(),
11                 [] (char c1, char c2){ return toupper(c1) == toupper(c2); } );
12                         // true
13
14 str1= {"Only for testing Purpose." };
15 str2= {"Only for testing purpose." };
16 auto pair= mismatch(str1.begin(), str1.end(), str2.begin());
17 if (pair.first != str1.end()){
18     cout << distance(str1.begin(), pair.first)
19         << "at (" << *pair.first << "," << *pair.second << ")"; // 17 at (P,p)
20 }
21
22 auto pair2= mismatch(str1.begin(), str1.end(), str2.begin(),
23                     [] (char c1, char c2){ return toupper(c1) == toupper(c2); });
24 if (pair2.first == str1.end()){
25     cout << "str1 and str2 are equal"; // str1 and str2 are equal
26 }
```

在范围内搜索范围

`std::search` 从开头搜索另一个范围，`std::find_end` 从结尾搜索 `std::search_n` 搜索范围内连续的 n 个元素。

所有算法都采用前向迭代器，可以用二进制谓词参数化，若搜索不成功，则返回第一个范围的结束迭代器。

在第一个区域中搜索第二个区域并返回位置，从开头开始：

```
1 FwdIt1 search(FwdIt1 first1, FwdIt1 last1, FwdIt2 first2, FwdIt2 last2)
2 FwdIt1 search(ExePol pol, FwdIt1 first1, FwdIt1 last1,
3                 FwdIt2 first2, FwdIt2 last2)
4
5 FwdIt1 search(FwdIt1 first1, FwdIt1 last1,
```

```

6     FwdIt2 first2, FwdIt2 last2, BiPre pre)
7 FwdIt1 search(ExePol pol, FwdIt1 first1, FwdIt1 last1,
8                 FwdIt2 first2, FwdIt2 last2, BiPre pre)
9
10    FwdIt1 search(FwdIt1 first, FwdIt last1, Search search)

```

在第一个范围中搜索第二个范围并返回位置，从结尾开始：

```

1 FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1, FwdIt2 first2 FwdIt2 last2)
2 FwdIt1 find_end(ExePol pol, FwdIt1 first1, FwdIt1 last1,
3                   FwdIt2 first2 FwdIt2 last2)
4
5 FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1, FwdIt2 first2, FwdIt2 last2,
6                   BiPre pre)
7 FwdIt1 find_end(ExePol pol, FwdIt1 first1, FwdIt1 last1,
8                   FwdIt2 first2, FwdIt2 last2, BiPre pre)

```

搜索计算第一个范围内的连续值：

```

1 FwdIt search_n(FwdIt first, FwdIt last, Size count, const T& value)
2 FwdIt search_n(ExePol pol, FwdIt first, FwdIt last, Size count, const T& value)
3
4 FwdIt search_n(FwdIt first, FwdIt last, Size count, const T& value, BiPre pre)
5 FwdIt search_n(ExePol pol, FwdIt first,
6                  FwdIt last, Size count, const T& value, BiPre pre)

```

特殊算法 search_n

算法 FwdIt search_n(FwdIt first, FwdIt last, Size count, const T& value, BiPre pre) 非常特殊。二元谓词 BiPre 将范围的值用作第一个参数，将 value 用作第二个参数。

std::find, std::find_end 和 std::search_n

```

1 // search.cpp
2 ...
3 #include <algorithm>
4 ...
5 using std::search;
6
7 std::array<int, 10> arr1{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
8 std::array<int, 5> arr2{3, 4, -5, 6, 7};
9
10 auto fwdIt= search(arr1.begin(), arr1.end(), arr2.begin(), arr2.end());
11 if (fwdIt == arr1.end()) std::cout << "arr2 not in arr1." // arr2 not in arr1.

```

```
12
13 auto fwdIt2= search(arr1.begin(), arr1.end(), arr2.begin(), arr2.end(),
14     [](int a, int b){ return std::abs(a) == std::abs(b); });
15 if (fwdIt2 != arr1.end()) std::cout << "arr2 at position "
16     << std::distance(arr1.begin(), fwdIt2) << " in arr1.";
17     // arr2 at position 3 in arr1.
```

10.6. 可修改算法

C++ 有许多修改元素和范围的算法。

复制元素和范围

可以使用 `std::copy` 向前复制范围，使用 `std::copy_backward` 向后复制范围，使用 `std::copy_if` 有条件地复制范围。若要复制 `n` 个元素，可以使用 `std::copy_n`。

复制范围：

```
1 OutIt copy(InIt first, InIt last, OutIt result)
2 FwdIt2 copy(ExePol pol, FwdIt first, FwdIt last, FwdIt2 result)
```

复制 `n` 个元素：

```
1 OutIt copy_n(InIt first, Size n, OutIt result)
2 FwdIt2 copy_n(ExePol pol, FwdIt first, Size n, FwdIt2 result)
```

复制依赖谓词 `pre` 的元素。

```
1 OutIt copy_if(InIt first, InIt last, OutIt result, UnPre pre)
2 FwdIt2 copy_if(ExePol pol, FwdIt first, FwdIt last, FwdIt2 result, UnPre pre)
```

向后复制范围：

```
1 BiIt copy_backward(BiIt first, BiIt last, BiIt result)
```

算法需要输入迭代器，并将其元素复制到 `result`，返回一个结束迭代器到目标范围内。

复制元素和范围

```
1 // copy.cpp
2 ...
3 #include <algorithm>
4 ...
5
6 std::vector<int> myVec{0, 1, 2, 3, 4, 5, 6, 7, 9};
7 std::vector<int> myVec2(10);
```

```

8
9 std::copy_if(myVec.begin(), myVec.end(), myVec2.begin() + 3,
10   [] (int a){ return a%2; });
11
12 for (auto v: myVec2) std::cout << v << " "; // 0 0 0 1 3 5 7 9 00
13
14 std::string str{"abcdefghijklmop"};
15 std::string str2{"-----"};
16
17 std::cout << str2; // -----
18 std::copy_backward(str.begin(), str.end(), str2.end());
19 std::cout << str2; // ----abcdefghijklmop
20 std::cout << str; // abcdefghijklmop
21
22 std::copy_backward(str.begin(), str.begin() + 5, str.end());
23 std::cout << str; // abcdefghijkabcde

```

替换元素和范围

可以使用 `std::replace`、`std::replace_if`、`std::replace_copy` 和 `std::replace_copy_if` 来替换范围中的元素。算法在两个方面有所不同。首先，算法需要谓词吗？其次，算法是否复制目标范围内的元素？若旧元素的值为 `old`，则用 `newValue` 替换范围内的旧元素。

```

1 void replace(FwdIt first, FwdIt last, const T& old, const T& newValue)
2 void replace(ExePol pol, FwdIt first, FwdIt last, const T& old,
3               const T& newValue)

```

若旧元素满足谓词 `pred`，则用 `newValue` 替换该范围的旧元素：

```

1 void replace_if(FwdIt first, FwdIt last, UnPred pred, const T& newValue)
2 void replace_if(ExePol pol, FwdIt first, FwdIt last, UnPred pred,
3                 const T& newValue)

```

若旧元素的值为 `old`，则用 `newValue` 替换范围内的旧元素。将结果复制到 `result`：

```

1 OutIt replace_copy(InpIt first, InpIt last, OutIt result, const T& old,
2                     const T& newValue)
3 FwdIt2 replace_copy(ExePol pol, FwdIt first, FwdIt last,
4                     FwdIt2 result, const T& old, const T& newValue)

```

若旧元素满足谓词 `pred`，则用 `newValue` 替换该范围的旧元素。

将结果复制到 `result`：

```

1 OutIt replace_copy_if(InpIt first, InpIt last, OutIt result, UnPre pred,
2                       const T& newValue)
3 FwdIt2 replace_copy_if(ExePol pol, FwdIt first, FwdIt last,
4                       FwdIt2 result, UnPre pred, const T& newValue)

```

替换元素和范围

```
1 // replace.cpp
2 ...
3 #include <algorithm>
4 ...
5
6 std::string str{"Only for testing purpose." };
7 std::replace(str.begin(), str.end(), ' ', '1');
8 std::cout << str; // Only1for1testing1purpose.
9
10 std::replace_if(str.begin(), str.end(), [] (char c){ return c == '1'; }, '2');
11 std::cout << str; // Only2for2testing2purpose.
12
13 std::string str2;
14 std::replace_copy(str.begin(), str.end(), std::back_inserter(str2), '2', '3');
15 std::cout << str2; // Only3for3testing3purpose.
16
17 std::string str3;
18 std::replace_copy_if(str2.begin(), str2.end(),
19 std::back_inserter(str3), [] (char c){ return c == '3'; }, '4');
20 std::cout << str3; // Only4for4testing4purpose.
```

删除元素和范围

`std::remove`、`std::remove_if`、`std::remove_copy` 和 `std::remove_copy_if` 支持两种操作。一方面，从范围内删除带有或不带有谓词的元素。另一方面，将修改的结果复制到新的范围。

从范围内删除值为 `val` 的元素：

```
1 FwdIt remove(FwdIt first, FwdIt last, const T& val)
2 FwdIt remove(ExePol pol, FwdIt first, FwdIt last, const T& val)
```

从范围内移除元素，满足谓词 `pred`：

```
1 FwdIt remove_if(FwdIt first, FwdIt last, UnPred pred)
2 FwdIt remove_if(ExePol pol, FwdIt first, FwdIt last, UnPred pred)
```

从范围内移除值为 `val` 的元素。将结果复制到 `result`：

```
1 OutIt remove_copy(InpIt first, InpIt last, OutIt result, const T& val)
2 FwdIt2 remove_copy(ExePol pol, FwdIt first, FwdIt last,
3 FwdIt2 result, const T& val)
```

从范围内删除满足谓词 `pred` 的元素，将结果复制到 `result`。

```
1 OutIt remove_copy_if(InpIt first, InpIt last, OutIt result, UnPre pred)
2 FwdIt2 remove_copy_if(ExePol pol, FwdIt first, FwdIt last,
3                         FwdIt2 result, UnPre pred)
```

算法需要源范围的输入迭代器和目标范围的输出迭代器。作为结果，会返回目标范围的结束迭代器。

应用擦除-删除习语

移除变量不会从范围内移除元素，值会返回范围的新逻辑结束位置，必须使用擦除-删除习惯用法来调整容器的大小。

删除元素和范围

```
1 // remove.cpp
2 ...
3 #include <algorithm>
4 ...
5
6 std::vector<int> myVec{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
7
8 auto newIt= std::remove_if(myVec.begin(), myVec.end(),
9                           [] (int a){ return a%2; });
10 for (auto v: myVec) std::cout << v << " "; // 0 2 4 6 8 5 6 7 8 9
11
12 myVec.erase(newIt, myVec.end());
13 for (auto v: myVec) std::cout << v << " "; // 0 2 4 6 8
14
15 std::string str{"Only for Testing Purpose." };
16 str.erase( std::remove_if(str.begin(), str.end(),
17                           [] (char c){ return std::isupper(c); }), str.end() );
18 std::cout << str << '\n'; // nly for esting urpose.
```

填充和创建范围

可以用 `std::fill` 和 `std::fill_n`，可以使用 `std::generate` 和 `std::generate_n` 生成新元素。
用元素填充一个范围：

```
1 void fill(FwdIt first, FwdIt last, const T& val)
2 void fill(ExePol pol, FwdIt first, FwdIt last, const T& val)
```

用 `n` 个新元素填充一个范围：

```
1 OutIt fill_n(OutIt first, Size n, const T& val)
2 FwdIt fill_n(ExePol pol, FwdIt first, Size n, const T& val)
```

使用生成器 gen 生成一个范围:

```
1 void generate(FwdIt first, FwdIt last, Generator gen)
2 void generate(ExePol pol, FwdIt first, FwdIt last, Generator gen)
```

使用生成器 gen 生成一个范围的 n 个元素:

```
1 OutIt generate_n(OutIt first, Size n, Generator gen)
2 FwdIt generate_n(ExePol pol, FwdIt first, Size n, Generator gen)
```

算法期望值 val 或 Generator gen 作为参数, gen 必须是一个不接受参数并返回新值的函数。
std::fill_n 和 std::generate_n 算法的返回值是一个输出迭代器, 指向最后创建的元素。

填充和创建范围

```
1 // fillAndCreate.cpp
2 ...
3 #include <algorithm>
4 ...
5
6 int getNext() {
7     static int next{0};
8     return ++next;
9 }
10
11 std::vector<int> vec(10);
12 std::fill(vec.begin(), vec.end(), 2011);
13 for (auto v: vec) std::cout << v << " ";
14     // 2011 2011 2011 2011 2011 2011 2011 2011 2011 2011
15
16 std::generate_n(vec.begin(), 5, getNext);
17 for (auto v: vec) std::cout << v << " ";
18     // 1 2 3 4 5 2011 2011 2011 2011 2011
```

移动范围

std::move 可向前移动范围, std::move_backward 可向后移动范围。

向前移动范围:

```
1 OutIt move(InpIt first, InpIt last, OutIt result)
2 FwdIt2 move(ExePol pol, FwdIt first, FwdIt last, Fwd2It result)
```

向后移动范围:

```
1 BiIt move_backward(BiIt first, BiIt last, BiIt result)
```

这两种算法都需要一个目标迭代器 `result`, 将范围移动到该结果中。对于 `std::move` 算法, 这是一个输出迭代器。对于 `std::move_backward` 算法, 这是一个双向迭代器。这些算法返回一个输出或一个双向迭代器, 指向目标范围内的初始位置。

源范围可以修改

`std::move` 和 `std::move_backward` 应用移动语义, 所以源范围是有效的, 但之后不一定有相同的元素。

移动范围

```
1 // move.cpp
2 ...
3 #include <algorithm>
4 ...
5
6 std::vector<int> myVec{0, 1, 2, 3, 4, 5, 6, 7, 9};
7 std::vector<int> myVec2(myVec.size());
8 std::move(myVec.begin(), myVec.end(), myVec2.begin());
9 for (auto v: myVec2) std::cout << v << " "; // 0 1 2 3 4 5 6 7 9 0
10
11 std::string str{"abcdefghijklmop"};
12 std::string str2{"-----"};
13 std::move_backward(str.begin(), str.end(), str2.end());
14 std::cout << str2; // -----abcdefghijklmop
```

交换范围

`std::swap` 和 `std::swap_ranges` 可以交换对象和范围。

互换对象:

```
1 void swap(T& a, T& b)
```

交换范围:

```
1 FwdIt swap_ranges(FwdIt1 first1, FwdIt1 last1, FwdIt first2)
2 FwdIt swap_ranges(ExePol pol, FwdIt1 first1, FwdIt1 last1, FwdIt first2)
```

返回的迭代器指向目标范围内最后交换的元素。

范围不能重叠

交换算法

```
1 // swap.cpp
2 ...
3 #include <algorithm>
4 ...
5
6 std::vector<int> myVec{0, 1, 2, 3, 4, 5, 6, 7, 9};
7 std::vector<int> myVec2(9);
8 std::swap(myVec, myVec2);
9 for (auto v: myVec) std::cout << v << " "; // 0 0 0 0 0 0 0 0 0
10 for (auto v: myVec2) std::cout << v << " "; // 0 1 2 3 4 5 6 7 9
11
12 std::string str{"abcdefghijklmnp"};
13 std::string str2{"-----"};
14 std::swap_ranges(str.begin(), str.begin() + 5, str2.begin() + 5);
15 std::cout << str << '\n'; // -----fghijklmnp
16 std::cout << str2 << '\n'; // -----abcde-----
```

变换范围

`std::transform` 算法将一元或二进制可调用对象应用于范围，并将修改后的元素复制到目标范围。
将一元可调用函数 `fun` 应用于输入范围的元素，并将结果复制到 `result`:

```
1 OutIt transform(InpIt first1, InpIt last1, OutIt result, UnFun fun)
2 FwdIt2 transform(ExePol pol, FwdIt first1, FwdIt last1, FwdIt2 result, UnFun fun)
```

将二进制可调用对象 `fun` 应用于两个输入范围，并将结果复制到 `result`:

```
1 OutIt transform(InpIt1 first1, InpIt1 last1, InpIt2 first2, OutIt result,
2                 BiFun fun)
3 FwdIt3 transform(ExePol pol, FwdIt1 first1, FwdIt1 last1,
4                 FwdIt2 first2, FwdIt3 result, BiFun fun)
```

两个版本的不同之处在于，第一个版本将可调用单元应用于范围的每个元素，第二个版本并行地将可调用对象应用于两个范围对。返回的迭代器指向最后一个转换元素之后的一个位置。

变换算法

```
1 // transform.cpp
2 ...
3 #include <algorithm>
4 ...
5
```

```

6 std::string str{"abcdefghijklmnopqrstuvwxyz"};
7 std::transform(str.begin(), str.end(), str.begin(),
8     [] (char c){ return std::toupper(c); });
9 std::cout << str; // ABCDEFGHIJKLMNOPQRSTUVWXYZ
10
11 std::vector<std::string> vecStr{"Only", "for", "testing", "purpose", ". "};
12 std::vector<std::string> vecStr2(5, "-");
13 std::vector<std::string> vecRes;
14 std::transform(vecStr.begin(), vecStr.end(),
15     vecStr2.begin(), std::back_inserter(vecRes),
16     [] (std::string a, std::string b){ return std::string(b)+a+b; });
17 for (auto str: vecRes) std::cout << str << " ";
18                                     // -Only- -for- -testing- -purpose- -.

```

反向范围

`std::reverse` 和 `std::reverse_copy` 将其范围内元素的顺序颠倒。

反转范围内元素的顺序:

```

1 void reverse(BiIt first, BiIt last)
2 void reverse(ExePol pol, BiIt first, BiIt last)

```

反转范围内元素的顺序，并将结果复制到 `result`:

```

1 OutIt reverse_copy(BiIt first, BiIt last, OutIt result)
2 FwdIt reverse_copy(ExePol pol, BiIt first, BiIt last, FwdIt result)

```

这两种算法都需要双向迭代器，返回的迭代器指向复制元素之前的输出范围结果位置。

反向范围的算法

```

1 // algorithmen.cpp
2 ...
3 #include <algorithm>
4 ...
5
6 std::string str{"123456789"};
7 std::reverse(str.begin(), str.begin() + 5);
8 std::cout << str; // 543216789

```

旋转范围

`std::rotate` 和 `std::rotate_copy` 旋转其元素。

以这样的方式旋转元素，`middle` 成为新的第一个元素:

```
1 FwdIt rotate(FwdIt first, FwdIt middle, FwdIt last)
2 FwdIt rotate(ExePol pol, FwdIt first, FwdIt middle, FwdIt last)
```

旋转元素，使 middle 成为新的第一个元素。将结果复制到 result:

```
1 OutIt rotate_copy(FwdIt first, FwdIt middle, FwdIt last, OutIt result)
2 FwdIt2 rotate_copy(ExePol pol, FwdIt first, FwdIt middle, FwdIt last,
3                     FwdIt2 result)
```

两种算法都需要前向迭代器，返回的迭代器是复制范围的 end 迭代器。

旋转算法

```
1 // rotate.cpp
2 ...
3 #include <algorithm>
4 ...
5
6 std::string str{"12345"};
7 for (auto i = 0; i < str.size(); ++i) {
8     std::string tmp{str};
9     std::rotate(tmp.begin(), tmp.begin() + i, tmp.end());
10    std::cout << tmp << " ";
11 } // 12345 23451 34512 45123 51234
```

移位范围

C++20 函数 std::shift_left 和 std::shift_right 移动范围中的元素，这两种算法都返回前向迭代器。
将范围的元素向范围的开头移动 n，返回结果范围的末尾位置。

```
1 FwdIt shift_left(FwdIt first, FwdIt last, Num n)
2 FwdIt shift_left(ExePol pol, FwdIt first, FwdIt last, Num n)
```

将范围的元素向范围的末尾移动 n，返回结果范围的起始位置。

```
1 FwdIt shift_right(FwdIt first, FwdIt last, Num n)
2 FwdIt shift_right(ExePol pol, FwdIt first, FwdIt last, Num n)
```

当 $n == 0 \text{ || } n >= \text{last} - \text{first}$ 时，对 std::shift_left 和 std::shift_right 的操作无效，区域的元素移动了。

移动范围中的元素

```
1 // shiftRange.cpp
2 ...
```

```
3 #include <algorithm>
4 ...
5
6 std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7};
7 for (auto v: myVec) std::cout << v << " "; // 1 2 3 4 5 6 7
8
9 auto newEnd = std::shift_left(myVec.begin(), myVec.end(), 2);
10 myVec.erase(newEnd, myVec.end());
11 for (auto v: myVec) std::cout << v << " "; // 3 4 5 6 7
12
13 auto newBegin = std::shift_right(myVec.begin(), myVec.end(), 2);
14 myVec.erase(myVec.begin(), newBegin);
15 for (auto v: myVec) std::cout << v << " "; // 3 4 5
```

随机混洗范围

可以使用 `std::random_shuffle` 和 `std::shuffle` 随机混洗范围。

随机混洗范围内的元素:

```
1 void random_shuffle(RanIt first, RanIt last)
```

使用随机数生成器 `gen` 随机混洗范围内的元素:

```
1 void random_shuffle(RanIt first, RanIt last, RanNumGen&& gen)
```

随机混洗一个范围内的元素，使用均匀随机数生成器 `gen`:

```
1 void shuffle(RanIt first, RanIt last, URNG&& gen)
```

这些算法需要随机访问迭代器。`RanNumGen&& gen` 必须是可调用的，接受参数并在其参数内返回值。`URNG&& gen` 必须是一个均匀随机数生成器。

首选 `std::shuffle`

使用 `std::shuffle` 代替 `std::random_shuffle`。`std::random_shuffle` 自 C++14 以来已弃用，并在 C++17 中删除，因为其在内部使用 C 函数 `rand`。

随机混洗算法

```
1 // shuffle.cpp
2 ...
3 #include <algorithm>
4 ...
```

```

5
6 using std::chrono::system_clock;
7 using std::default_random_engine;
8 std::vector<int> vec1{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
9 std::vector<int> vec2(vec1);
10
11 std::random_shuffle(vec1.begin(), vec1.end());
12 for (auto v: vec1) std::cout << v << " "; // 4 3 7 8 0 5 2 1 6 9
13
14 unsigned seed= system_clock::now().time_since_epoch().count();
15 std::shuffle(vec2.begin(), vec2.end(), default_random_engine(seed));
16 for (auto v: vec2) std::cout << v << " "; // 4 0 2 3 9 6 5 1 8 7

```

Seed 初始化随机数生成器。

删除重复

使用 std::unique 和 std::unique_copy 算法，可删除相邻的重复项。使用或不使用二元谓词都可以做到这一点。

删除相邻的重复项:

```

1 FwdIt unique(FwdIt first, FwdIt last)
2 FwdIt unique(ExePol pol, FwdIt first, FwdIt last)

```

删除相邻的重复项，满足二元谓词:

```

1 FwdIt unique(FwdIt first, FwdIt last, BiPred pre)
2 FwdIt unique(ExePol pol, FwdIt first, FwdIt last, BiPred pre)

```

删除相邻的重复项并将结果复制到 result:

```

1 OutIt unique_copy(InpIt first, InpIt last, OutIt result)
2 FwdIt2 unique_copy(ExePol pol, FwdIt first, FwdIt last, FwdIt2 result)

```

删除相邻的重复项，满足二元谓词，并将结果复制到 result:

```

1 OutIt unique_copy(InpIt first, InpIt last, OutIt result, BiPred pre)
2 FwdIt2 unique_copy(ExePol pol, FwdIt first, FwdIt last,
3                     FwdIt2 result, BiPred pre)

```

唯一算法返回新的逻辑结束迭代器

唯一算法返回范围的逻辑结束迭代器。必须使用擦除-删除习惯用法删除元素。

删除重复算法

```

1 // removeDuplicates.cpp
2 ...
3 #include <algorithm>
4 ...
5
6 std::vector<int> myVec{0, 0, 1, 1, 2, 2, 3, 4, 4, 5,
7     3, 6, 7, 8, 1, 3, 3, 8, 8, 9};
8
9 myVec.erase(std::unique(myVec.begin(), myVec.end()), myVec.end());
10 for (auto v: myVec) std::cout << v << " "; // 0 1 2 3 4 5 3 6 7 8 1 3 8 9
11
12 std::vector<int> myVec2{1, 4, 3, 3, 3, 5, 7, 9, 2, 4, 1, 6, 8, 0, 3, 5, 7, 8, 7, 3, 9, 2, 4, 2, 5, 7, 3};
13
14 std::vector<int> resVec;
15 resVec.reserve(myVec2.size());
16 std::unique_copy(myVec2.begin(), myVec2.end(), std::back_inserter(resVec),
17     [] (int a, int b){ return (a%2) == (b%2); } );
18 for(auto v: myVec2) std::cout << v << " ";
19     // 1 4 3 3 3 5 7 9 2 4 1 6 8 0 3 5 7 8 7 3 9 2 4 2 5 7 3
20 for(auto v: resVec) std::cout << v << " "; // 1 4 3 2 1 6 3 8 7 2 5

```

10.7. 分区

什么是分区?

一个集合的划分是一个集合在子集中的分解，使得集合的每个元素精确地在一个子集中。二元谓词定义子集，以便第一个子集的成员满足谓词。剩下的元素在第二个子集中。

C++ 提供了一些处理分区的函数，都需要一个一元谓词。`std::partition` 和 `std::stable_partition` 划分一个范围并返回分区点。使用 `std::partition_point`，可以获得分区的分区点。之后，可以使用 `std::is_partitioned` 检查分区，或者使用 `std::partition_copy` 复制分区。

检查范围是否被分区：

```

1 bool is_partitioned(InIt first, InIt last, UnPre pre)
2 bool is_partitioned(ExePol pol, FwdIt first, FwdIt last, UnPre pre)

```

范围分区：

```

1 FwdIt partition(FwdIt first, FwdIt last, UnPre pre)
2 FwdIt partition(ExePol pol, FwdIt first, FwdIt last, UnPre pre)

```

范围分区(稳定)：

```
1 BiIt stable_partition(FwdIt first, FwdIt last, UnPre pre)
2 BiIt stable_partition(ExePol pol, FwdIt first, FwdIt last, UnPre pre)
```

在两个范围内复制分区:

```
1 pair<OutIt1, OutIt2> partition_copy(InIt first, InIt last,
2           OutIt1 result_true, OutIt2 result_false, UnPre pre)
3 pair<FwdIt1, FwdIt2> partition_copy(ExePol pol, FwdIt1 first, FwdIt1 last,
4           FwdIt2 result_true, FwdIt3 result_false, UnPre pre)
```

返回分区点:

```
1 FwdIt partition_point(FwdIt first, FwdIt last, UnPre pre)
```

与 std::partition 相反, std::stable_partition 保证元素保持它们的相对顺序。返回的迭代器 FwdIt 和 BiIt 指向分区初始位置的第二个子集。算法 std::partition_copy 的 pair std::pair<OutIt, OutIt> 包含子集 result_true 和 result_false 的结束迭代器。若范围未分区, 则 std::partition_point 的行为未定义。

分区算法

```
1 // partition.cpp
2 ...
3 #include <algorithm>
4 ...
5 using namespace std;
6
7 bool isOdd(int i){ return (i%2) == 1; }
8 vector<int> vec{1, 4, 3, 4, 5, 6, 7, 3, 4, 5, 6, 0, 4,
9                 8, 4, 6, 6, 5, 8, 8, 3, 9, 3, 7, 6, 4, 8};
10 auto parPoint= partition(vec.begin(), vec.end(), isOdd);
11 for (auto v: vec) cout << v << " ";
12     // 1 7 3 3 5 9 7 3 3 5 5 0 4 8 4 6 6 6 8 8 4 6 4 4 6 4 8
13
14 for (auto v= vec.begin(); v != parPoint; ++v) cout << *v << " ";
15     // 1 7 3 3 5 9 7 3 3 5 5
16 for (auto v= parPoint; v != vec.end(); ++v) cout << *v << " ";
17     // 4 8 4 6 6 6 8 8 4 6 4 4 6 4 8
18
19 cout << is_partitioned(vec.begin(), vec.end(), isOdd); // true
20 list<int> le;
21 list<int> ri;
22 partition_copy(vec.begin(), vec.end(), back_inserter(le), back_inserter(ri),
23                 [] (int i) { return i < 5; });
24 for (auto v: le) cout << v << ""; // 1 3 3 3 3 0 4 4 4 4 4 4
25 for (auto v: ri) cout << v << ""; // 7 5 9 7 5 5 8 6 6 6 8 8 6 6 8
```

10.8. 排序

可以使用 `std::sort` 或 `std::stable_sort` 对范围进行排序，或者使用 `std::partial_sort` 对位置进行排序。此外，`std::partial_sort_copy` 复制部分排序的范围。使用 `std::nth_element`，可以为元素分配在范围中的排序位置，可以使用 `std::is_sorted` 检查是否对范围进行了排序。若想知道一个范围被排序到哪个位置，使用 `std::is_sorted_until`。

默认情况下，使用预定义的函数对象 `std::less` 作为排序标准，可以使用自己的排序标准。这必须服从严格的弱排序。

对范围内的元素进行排序：

```
1 void sort(RaIt first, RaIt last)
2 void sort(ExePol pol, RaIt first, RaIt last)
3
4 void sort(RaIt first, RaIt last, BiPre pre)
5 void sort(ExePol pol, RaIt first, RaIt last, BiPre pre)
```

对范围内的元素进行排序(稳定)：

```
1 void stable_sort(RaIt first, RaIt last)
2 void stable_sort(ExePol pol, RaIt first, RaIt last)
3
4 void stable_sort(RaIt first, RaIt last, BiPre pre)
5 void stable_sort(ExePol pol, RaIt first, RaIt last, BiPre pre)
```

对范围内的元素进行部分排序，直到中间：

```
1 void partial_sort(RaIt first, RaIt middle, RaIt last)
2 void partial_sort(ExePol pol, RaIt first, RaIt middle, RaIt last)
3
4 void partial_sort(RaIt first, RaIt middle, RaIt last, BiPre pre)
5 void partial_sort(ExePol pol, RaIt first, RaIt middle, RaIt last, BiPre pre)
```

对范围内的元素进行部分排序，并将其复制到目标范围 `result_first` 和 `result_last` 中：

```
1 RaIt partial_sort_copy(Init first, Init last,
2                     RaIt result_first, RaIt result_last)
3 RaIt partial_sort_copy(ExePol pol, FwdIt first, FwdIt last,
4                     RaIt result_first, RaIt result_last)
5
6 RaIt partial_sort_copy(Init first, Init last,
7                     RaIt result_first, RaIt result_last, BiPre pre)
8 RaIt partial_sort_copy(ExePol pol, FwdIt first, FwdIt last,
9                     RaIt result_first, RaIt result_last, BiPre pre)
```

检查一个范围是否已排序：

```
1 bool is_sorted(FwdIt first, FwdIt last)
2 bool is_sorted(ExePol pol, FwdIt first, FwdIt last)
3
4 bool is_sorted(FwdIt first, FwdIt last, BiPre pre)
5 bool is_sorted(ExePol pol, FwdIt first, FwdIt last, BiPre pre)
```

返回第一个不满足排序条件的元素的位置:

```
1 FwdIt is_sorted_until(FwdIt first, FwdIt last)
2 FwdIt is_sorted_until(ExePol pol, FwdIt first, FwdIt last)
3
4 FwdIt is_sorted_until(FwdIt first, FwdIt last, BiPre pre)
5 FwdIt is_sorted_until(ExePol pol, FwdIt first, FwdIt last, BiPre pre)
```

重新排序范围，使第 n 个元素有正确的(排序)位置:

```
1 void nth_element(RaIt first, RaIt nth, RaIt last)
2 void nth_element(ExePol pol, RaIt first, RaIt nth, RaIt last)
3
4 void nth_element(RaIt first, RaIt nth, RaIt last, BiPre pre)
5 void nth_element(ExePol pol, RaIt first, RaIt nth, RaIt last, BiPre pre)
```

下面是一个代码段。

排序算法

```
1 // sort.cpp
2 ...
3 #include <algorithm>
4 ...
5
6 std::string str{"RUDAjDkaACsdfjwldXmnEiVSEZTiepfgOIkue"};
7 std::cout << std::is_sorted(str.begin(), str.end()); // false
8
9 std::partial_sort(str.begin(), str.begin() + 30, str.end());
10 std::cout << str; // AACDEEIORSTUVXZaddddeeffgijjjkwspnmluk
11
12 auto sortUntil = std::is_sorted_until(str.begin(), str.end());
13 std::cout << *sortUntil; // s
14 for (auto charIt = str.begin(); charIt != sortUntil; ++charIt)
15     std::cout << *charIt; // AACDEEIORSTUVXZaddddeeffgijjjkw
16
17 std::vector<int> vec{1, 0, 4, 3, 5};
18 auto vecIt = vec.begin();
19
20 while(vecIt != vec.end()) {
21     std::nth_element(vec.begin(), vecIt++, vec.end());
```

```

22     std::cout << std::distance(vec.begin(), vecIt) << "-th ";
23     for (auto v: vec) std::cout << v << "/";
24 }
25 // 1-th 01435/2-th 01435/3-th 10345/4-th 30145/5-th 10345

```

10.9. 二分查找

二分搜索算法利用了范围已经排序的事实。要搜索一个元素，使用 `std::binary_search`。使用 `std::lower_bound`，将获得不小于给定值的第一个元素的迭代器。使用 `std::upper_bound`，将获得第一个元素的迭代器，该元素大于给定值。`std::equal_range` 结合了这两种算法。

若容器有 n 个元素，则平均需要进行 $\log_2(n)$ 次比较，二进制搜索要求使用与对容器排序相同的比较条件。默认的比较条件是 `std::less`，可以调整它。排序标准必须遵守严格的弱排序，若不是，则程序有未定义行为。

无序关联容器的成员函数通常更快。

在范围内搜索元素 `val`:

```

1 bool binary_search(FwdIt first, FwdIt last, const T& val)
2 bool binary_search(FwdIt first, FwdIt last, const T& val, BiPre pre)

```

返回范围内第一个元素的位置，不小于 `val`:

```

1 FwdIt lower_bound(FwdIt first, FwdIt last, const T& val)
2 FwdIt lower_bound(FwdIt first, FwdIt last, const T& val, BiPre pre)

```

返回范围内第一个元素的位置，大于 `val`:

```

1 FwdIt upper_bound(FwdIt first, FwdIt last, const T& val)
2 FwdIt upper_bound(FwdIt first, FwdIt last, const T& val, BiPre pre)

```

返回元素 `val` 的 `std::lower_bound` 和 `std::upper_bound` 对:

```

1 pair<FwdIt, FwdIt> equal_range(FwdIt first, FwdIt last, const T& val)
2 pair<FwdIt, FwdIt> equal_range(FwdIt first, FwdIt last, const T& val, BiPre pre)

```

最后，下面是代码段。

二进制搜索算法

```

1 // binarySearch.cpp
2 ...
3 #include <algorithm>
4 ...
5 using namespace std;
6
7 bool isLessAbs(int a, int b) {

```

```

8     return abs(a) < abs(b);
9 }
10 vector<int> vec{-3, 0, -3, 2, -3, 5, -3, 7, -0, 6, -3, 5,
11      -6, 8, 9, 0, 8, 7, -7, 8, 9, -6, 3, -3, 2};
12
13 sort(vec.begin(), vec.end(), isLessAbs);
14 for (auto v: vec) cout << v << " ";
15 // 0 0 0 2 2 -3 -3 -3 -3 3 -3 5 5 -6 -6 6 7 -7 7 8 8 8 9 9
16 cout << binary_search(vec.begin(), vec.end(), -5, isLessAbs); // true
17 cout << binary_search(vec.begin(), vec.end(), 5, isLessAbs); // true
18
19 auto pair= equal_range(vec.begin(), vec.end(), 3, isLessAbs);
20 cout << distance(vec.begin(), pair.first); // 5
21 cout << distance(vec.begin(), pair.second)-1; // 11
22
23 for (auto threeIt= pair.first;threeIt != pair.second; ++threeIt)
24   cout << *threeIt << " "; // -3 -3 -3 -3 -3 3 -3

```

10.10. 合并操作

合并操作能够将已排序的范围合并到新的已排序范围内，该算法要求范围和算法使用相同的排序标准。若不是，则程序具有未定义行为。默认情况下，使用预定义的排序条件 `std::less`。若使用排序标准，则必须遵守严格的弱排序。若不是，则程序未定义。

可以使用 `std::inplace_merge` 和 `std::merge` 合并两个排序的范围。可以使用 `std::includes` 检查一个排序范围是否在另一个排序范围内。可以将 `std::set_difference`, `std::set_intersection`, `std::set_symmetric_difference` 和 `std::set_union` 这两个排序范围合并为一个新的排序范围。

将两个排序好的子范围 `[first, mid]` 和 `[mid, last)` 合并:

```

1 void inplace_merge(BiIt first, BiIt mid, BiIt last)
2 void inplace_merge(ExePol pol, BiIt first, BiIt mid, BiIt last)
3
4 void inplace_merge(BiIt first, BiIt mid, BiIt last, BiPre pre)
5 void inplace_merge(ExePol pol, BiIt first, BiIt mid, BiIt last, BiPre pre)

```

合并两个排序范围并将结果复制到 `result`:

```

1 OutIt merge(InpIt first1, InpIt last1, InpIt first2, InpIt last2, OutIt result)
2 FwdIt3 merge(ExePol pol, FwdIt1 first1, FwdIt1 last1,
3               FwdIt2 first2, FwdIt2 last2, FwdIt3 result)
4
5 OutIt merge(InpIt first1, InpIt last1, InpIt first2, InpIt last2, OutIt result,
6             BiPre pre)
7 FwdIt3 merge(ExePol pol, FwdIt1 first1, FwdIt1 last1,
8               FwdIt2 first2, FwdIt2 last2, FwdIt3 result, BiPre pre)

```

检查第二个范围的所有元素是否都在第一个范围内:

```

1 bool includes(InpIt first1, InpIt last1, InpIt1 first2, InpIt1 last2)
2 bool includes(ExePol pol, FwdIt first1, FwdIt last1, FwdIt1 first2, FwdIt1 last2)
3
4 bool includes(InpIt first1, InpIt last1, InpIt first2, InpIt last2, BiPre pre)
5 bool includes(ExePol pol, FwdIt first1, FwdIt last1,
6           FwdIt1 first2, FwdIt1 last2, BiPre pre)

```

将不属于第二个范围的第一个范围的元素复制到 result:

```

1 OutIt set_difference(InpIt first1, InpIt last1, InpIt1 first2, InpIt2 last2,
2           OutIt result)
3 FwdIt2 set_difference(ExePol pol, FwdIt first1, FwdIt last1,
4           FwdIt1 first2, FwdIt1 last2, FwdIt2 result)
5
6 OutIt set_difference(InpIt first1, InpIt last1, InpIt1 first2, InpIt2 last2,
7           OutIt result, BiPre pre)
8 FwdIt2 set_difference(ExePol pol, FwdIt first1, FwdIt last1,
9           FwdIt1 first2, FwdIt1 last2, FwdIt2 result, BiPre pre)

```

确定第一个范围与第二个范围的交集，并将结果复制到 result:

```

1 OutIt set_intersection(InpIt first1, InpIt last1, InpIt1 first2, InpIt2 last2,
2           OutIt result)
3 FwdIt2 set_intersection(ExePol pol, FwdIt first1, FwdIt last1,
4           FwdIt1 first2, FwdIt1 last2, FwdIt2 result)
5
6 OutIt set_intersection(InpIt first1, InpIt last1, InpIt1 first2, InpIt2 last2,
7           OutIt result, BiPre pre)
8 FwdIt2 set_intersection(ExePol pol, FwdIt first1, FwdIt last1,
9           FwdIt1 first2, FwdIt1 last2, FwdIt2 result, BiPre pre)

```

确定第一个范围与第二个范围的对称差，并将结果复制到 result:

```

1 OutIt set_symmetric_difference(InpIt first1, InpIt last1,
2           InpIt1 first2, InpIt2 last2, OutIt result)
3 FwdIt2 set_symmetric_difference(ExePol pol, FwdIt first1, FwdIt last1,
4           FwdIt1 first2, FwdIt1 last2, FwdIt2 result)
5
6 OutIt set_symmetric_difference(InpIt first1, InpIt last1,
7           InpIt1 first2, InpIt2 last2, OutIt result,
8           BiPre pre)
9 FwdIt2 set_symmetric_difference(ExePol pol, FwdIt first1, FwdIt last1,
10          FwdIt1 first2, FwdIt1 last2, FwdIt2 result,
11          BiPre pre)

```

确定第一个范围与第二个范围的并集，并将结果复制到 result:

```

1 OutIt set_union(InpIt first1, InpIt last1,
2                 InpIt1 first2, InpIt2 last2, OutIt result)
3 FwdIt2 set_union(ExePol pol, FwdIt first1, FwdIt last1,
4                   FwdIt1 first2, FwdIt1 last2, FwdIt2 result)
5
6 OutIt set_union(InpIt first1, InpIt last1,
7                 InpIt1 first2, InpIt2 last2, OutIt result, BiPre pre)
8 FwdIt2 set_union(ExePol pol, FwdIt first1, FwdIt last1,
9                   FwdIt1 first2, FwdIt1 last2, FwdIt2 result, BiPre pre)

```

返回的迭代器是目标范围的 end 迭代器。std::set_difference 的目标范围包含第一个范围中的所有元素，但不包含第二个范围中的元素。相反，std::symmetric_difference 的目标范围只有属于一个范围的元素，而不是同时属于两个范围的元素。std::union 决定了这两个已排序范围的并集。

合并算法

```

// merge.cpp
...
#include <algorithm>
...
std::vector<int> vec1{1, 1, 4, 3, 5, 8, 6, 7, 9, 2};
std::vector<int> vec2{1, 2, 3};

std::sort(vec1.begin(), vec1.end());
std::vector<int> vec(vec1);

vec1.reserve(vec1.size() + vec2.size());
vec1.insert(vec1.end(), vec2.begin(), vec2.end());
for (auto v: vec1) std::cout << v << " "; // 1 1 2 3 4 5 6 7 8 9 1 2 3

std::inplace_merge(vec1.begin(), vec1.end()-vec2.size(), vec1.end());
for (auto v: vec1) std::cout << v << " "; // 1 1 1 2 2 3 3 4 5 6 7 8 9

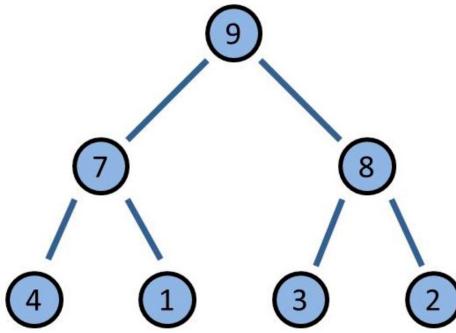
vec2.push_back(10);
for (auto v: vec) std::cout << v << " "; // 1 1 2 3 4 5 6 7 8 9
for (auto v: vec2) std::cout << v << " "; // 1 2 3 10

std::vector<int> res;
std::set_symmetric_difference(vec.begin(), vec.end(), vec2.begin(), vec2.end(),
                             std::back_inserter(res));
for (auto v : res) std::cout << v << " "; // 1 4 5 6 7 8 9 10

res= {};
std::set_union(vec.begin(), vec.end(), vec2.begin(), vec2.end(),
               std::back_inserter(res));
for (auto v : res) std::cout << v << " "; // 1 1 2 3 4 5 6 7 8 9 10

```

10.11. 堆



什么是堆?

堆是一个二叉搜索树，其中父元素总是比子元素大。堆树针对元素的高效排序进行了优化。

可以用 `std::make_heap` 创建一个堆，使用 `std::push_heap` 向堆上推送新元素，也可以使用 `std::pop_heap` 从堆中取出最大的元素。这两种操作都尊重堆特征。`std::push_heap` 移动堆上范围的最后一个元素，`std::pop_heap` 将堆中最大的元素移动到范围内的最后一个位置。可以使用 `std::is_heap` 检查一个范围是否为堆，也可以使用 `std::is_heap_until` 来确定该范围在哪个位置是堆。`std::sort_heap` 可对堆进行排序。

堆算法要求范围和算法使用相同的排序标准。若不是，则程序具有未定义行为。默认情况下，使用预定义的排序条件 `std::less`。若使用排序标准，则必须遵守严格的弱排序。若不是，则程序具有未定义行为。

使用范围创建一个堆:

```
1 void make_heap(RaiT first, RaiT last)
2 void make_heap(RaiT first, RaiT last, BiPre pre)
```

检查范围是否为堆:

```
1 bool is_heap(RaiT first, RaiT last)
2 bool is_heap(ExePol pol, RaiT first, RaiT last)
3
4 bool is_heap(RaiT first, RaiT last, BiPre pre)
5 bool is_heap(ExePol pol, RaiT first, RaiT last, BiPre pre)
```

确定在哪个位置之前范围是堆:

```
1 RaiT is_heap_until(RaiT first, RaiT last)
2 RaiT is_heap_until(ExePol pol, RaiT first, RaiT last)
3
4 RaiT is_heap_until(RaiT first, RaiT last, BiPre pre)
5 RaiT is_heap_until(ExePol pol, RaiT first, RaiT last, BiPre pre)
```

对堆进行排序:

```
1 void sort_heap(RaIt first, RaIt last)
2 void sort_heap(RaIt first, RaIt last, BiPre pre)
```

将范围的最后一个元素压入堆中, [first, last-1) 必须是堆。

```
1 void push_heap(RaIt first, RaIt last)
2 void push_heap(RaIt first, RaIt last, BiPre pre)
```

从堆中移除最大的元素, 并将其放在范围的末尾:

```
1 void pop_heap(RaIt first, RaIt last)
2 void pop_heap(RaIt first, RaIt last, BiPre pre)
```

使用 std::pop_heap, 可以从堆中删除最大的元素, 最大的元素是范围的最后一个元素。要从堆 h 中删除元素, 使用 h.pop_back。

堆算法

```
1 // heap.cpp
2 ...
3 #include <algorithm>
4 ...
5
6 std::vector<int> vec{4, 3, 2, 1, 5, 6, 7, 9, 10};
7 std::make_heap(vec.begin(), vec.end());
8 for (auto v: vec) std::cout << v << " "; // 10 9 7 4 5 6 2 3 1
9 std::cout << std::is_heap(vec.begin(), vec.end()); // true
10
11 vec.push_back(100);
12 std::cout << std::is_heap(vec.begin(), vec.end()); // false
13 std::cout << *std::is_heap_until(vec.begin(), vec.end()); // 100
14 for (auto v: vec) std::cout << v << " "; // 10 9 7 4 5 6 2 3 1 100
15
16 std::push_heap(vec.begin(), vec.end());
17 std::cout << std::is_heap(vec.begin(), vec.end()); // true
18 for (auto v: vec) std::cout << v << " "; // 100 10 7 4 9 6 2 3 1 5
19
20 std::pop_heap(vec.begin(), vec.end());
21 for (auto v: vec) std::cout << v << " "; // 10 9 7 4 5 6 2 3 1 100
22 std::cout << *std::is_heap_until(vec.begin(), vec.end()); // 100
23
24 vec.resize(vec.size()-1);
25 std::cout << std::is_heap(vec.begin(), vec.end()); // true
26 std::cout << vec.front() << '\n'; // 10
```

10.12. 最小和最大

可以使用 `std::min_element`、`std::max_element` 和 `std::minmax_element` 算法来确定范围的最小、最大、最小和最大对。每个算法都可以用二进制谓词调用。此外，C++17 允许在一对边界值之间夹紧一个值。

返回范围的最小元素：

```
1 constexpr FwdIt min_element(FwdIt first, FwdIt last)
2 FwdIt min_element(ExePol pol, FwdIt first, FwdIt last)
3
4 constexpr FwdIt min_element(FwdIt first, FwdIt last, BinPre pre)
5 FwdIt min_element(ExePol pol, FwdIt first, FwdIt last, BinPre pre)
```

返回范围的最大元素：

```
1 constexpr FwdIt max_element(FwdIt first, FwdIt last)
2 FwdIt max_element(ExePol pol, FwdIt first, FwdIt last)
3
4 constexpr FwdIt max_element(FwdIt first, FwdIt last, BinPre pre)
5 FwdIt max_element(ExePol pol, FwdIt first, FwdIt last, BinPre pre)
```

返回范围的 `std::min_element` 和 `std::max_element` 对：

```
1 constexpr pair<FwdIt, FwdIt> minmax_element(FwdIt first, FwdIt last)
2 pair<FwdIt, FwdIt> minmax_element(ExePol pol, FwdIt first, FwdIt last)
3
4 constexpr pair<FwdIt, FwdIt> minmax_element(FwdIt first, FwdIt last, BinPre pre)
5 pair<FwdIt, FwdIt> minmax_element(ExePol pol, FwdIt first, FwdIt last,
6 BinPre pre)
```

若范围有多个最小或最大元素，则返回第一个。

最小和最大算法

```
1 // minMax.cpp
2 ...
3 #include <algorithm>
4 ...
5
6 int toInt(const std::string& s) {
7     std::stringstream buff;
8     buff.str("");
9     buff << s;
10    int value;
11    buff >> value;
12    return value;
13 }
```

```

14
15 std::vector<std::string> myStrings{"94", "5", "39", "-4", "-49", "1001", "-77",
16             "23", "0", "84", "59", "96", "6", "-94"};
17 auto str= std::minmax_element(myStrings.begin(), myStrings.end());
18
19 std::cout << *str.first << ":" << *str.second; // -4:96
20
21 auto asInt= std::minmax_element(myStrings.begin(), myStrings.end(),
22     [] (std::string a, std::string b){ return toInt(a) < toInt(b); });
23 std::cout << *asInt.first << ":" << *asInt.second; // -94:1001

```

std::clamp 在一对边界值之间夹住一个值:

```

1 constexpr const T& clamp(const T& v, const T& lo, const T& hi);
2 constexpr const T& clamp(const T& v, const T& lo, const T& hi, BinPre pre);

```

若 lo 小于 v 且 v 小于 hi, 则返回对 v 的引用。否则, 返回对 lo 或 hi 的引用。默认情况下, < 用作比较标准, 也可以提供一个二进制谓词 pre。

箱位一个值

```

1 // clamp.cpp
2 ...
3 #include <algorithm>
4 ...
5
6 auto values = {1, 2, 3, 4, 5, 6, 7};
7 for (auto v: values) std::cout << v << ' '; // 1 2 3 4 5 6 7
8
9 auto lo = 3;
10 auto hi = 6;
11 for (auto v: values) std::cout << std::clamp(v, lo, hi) << ' '; // 3 3 3 4 5 6 6

```

10.13. 排列

std::prev_permutation 和 std::next_permutation 返回新排序范围的前一个较小或下一个较大的排列。若没有更小或更大的排列, 则算法返回 false。两种算法都需要双向迭代器。默认情况下, 使用预定义的排序条件 std::less。若使用排序标准, 则必须遵守严格的弱排序; 否则, 程序具有未定义行为。

将前面的排列应用于范围:

```

1 bool prev_permutation(BiIt first, BiIt last)
2 bool prev_permutation(BiIt first, BiIt last, BiPred pred)

```

将下一个排列应用于范围:

```
1 bool next_permutation(BiIt first, BiIt last)
2 bool next_permutation(BiIt first, BiIt last, BiPred pre)
```

可以用这两种算法生成范围的所有排列。

置换算法

```
1 // permutation.cpp
2 ...
3 #include <algorithm>
4 ...
5
6 std::vector<int> myInts{1, 2, 3};
7 do{
8     for (auto i: myInts) std::cout << i;
9     std::cout << " ";
10 } while(std::next_permutation(myInts.begin(), myInts.end()));
11 // 123 132 213 231 312 321
12
13 std::reverse(myInts.begin(), myInts.end());
14 do{
15     for (auto i: myInts) std::cout << i;
16     std::cout << " ";
17 } while(std::prev_permutation(myInts.begin(), myInts.end()));
18 // 321 312 231 213 132 123
```

10.14. 数值

数值算法 `std::accumulate`、`std::nearby_difference`、`std::partial_sum`、`std::inner_product` 和 `std::iota`，以及另外六个 C++17 算法 `std::exclusive_scan`、`std::inclusive_scan`、`std::transform_exclusive_scan`、`std::transform_inclusive_scan`、`std::reduce` 和 `std::transform_reduce` 是特殊的，都在头文件 `<numeric>` 中定义。其广泛适用，可用可调用对象调用。

累加范围的元素。`Init` 是起始值：

```
1 T accumulate(InpIt first, InpIt last, T init)
2 T accumulate(InpIt first, InpIt last, T init, BiFun fun)
```

计算范围内相邻元素之间的差，并将结果存储在 `result` 中：

```
1 OutIt adjacent_difference(InpIt first, InpIt last, OutIt result)
2 FwdIt2 adjacent_difference(ExePol pol, FwdIt first, FwdIt last, FwdIt2 result)
3
4 OutIt adjacent_difference(InpIt first, InpIt last, OutIt result, BiFun fun)
5 FwdIt2 adjacent_difference(ExePol pol, FwdIt first, FwdIt last,
6                             FwdIt2 result, BiFun fun)
```

计算范围的部分和:

```
1 OutIt partial_sum(InpIt first, InpIt last, OutIt result)
2 OutIt partial_sum(InpIt first, InpIt last, OutIt result, BiFun fun)
```

计算两个范围的内积(标量积)并返回结果:

```
1 T inner_product(InpIt first1, InpIt last1, OutIt first2, T init)
2 T inner_product(InpIt first1, InpIt last1, OutIt first2, T init,
3     BiFun fun1, BiFun fun2)
```

为范围中的每个元素赋一个依次递增的值。初始值为 val:

```
1 void iota(FwdIt first, FwdIt last, T val)
```

算法很难得到这样的结果。

不带可调用的 std::accumulate 可以使用以下策略:

```
1 result = init;
2 result += *(first+0);
3 result += *(first+1);
4 ...
```

不带可调用的 std::adjacent_difference 可使用以下策略:

```
1 *(result) = *first;
2 *(result+1) = *(first+1) - *(first);
3 *(result+2) = *(first+2) - *(first+1);
4 ...
```

不带可调用的 Std::partial_sum 可使用以下策略:

```
1 *(result) = *first;
2 *(result+1) = *first + *(first+1);
3 *(result+2) = *first + *(first+1) + *(first+2)
4 ...
```

具有挑战性的算法变化 inner_product(InpIt, InpIt, OutIt, T, BiFun fun1, BiFun fun2) 具有两个二进制可调用对象，使用以下策略: 第二个可调用对象 fun2 应用于每一对范围，以生成临时目标范围 tmp，第一个可调用对象应用于目标范围 tmp 的每个元素，进行累积，从而生成最终结果。

数值算法

```

1 // numeric.cpp
2 ...
3 #include <numeric>
4 ...
5
6 std::array<int, 9> arr{1, 2, 3, 4, 5, 6, 7, 8, 9};
7 std::cout << std::accumulate(arr.begin(), arr.end(), 0); // 45
8 std::cout << std::accumulate(arr.begin(), arr.end(), 1,
9     [] (int a, int b){ return a*b; } ); // 362880
10
11 std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
12 std::vector<int> myVec;
13 std::adjacent_difference(vec.begin(), vec.end(),
14     std::back_inserter(myVec), [] (int a, int b){ return a*b; });
15 for (auto v: myVec) std::cout << v << " "; // 1 2 6 12 20 30 42 56 72
16 std::cout << std::inner_product(vec.begin(), vec.end(), arr.begin(), 0); // 285
17
18 myVec= {};
19 std::partial_sum(vec.begin(), vec.end(), std::back_inserter(myVec));
20 for (auto v: myVec) std::cout << v << " "; // 1 3 6 10 15 21 28 36 45
21
22 std::vector<int> myLongVec(10);
23 std::iota(myLongVec.begin(), myLongVec.end(), 2000);
24 for (auto v: myLongVec) std::cout << v << " ";
25 // 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009

```

新并行算法与 C++17

通常用于并行执行的六种新算法称为前缀求和。若给定的二元可调用对象不可关联和交换，则算法的行为未定义。

reduce: 减少范围内的元素。init 是初始值。

- 其行为与 std::accumulate 相同，但范围可能会重新排列。

```

1 ValType reduce(InpIt first, InpIt last)
2 ValType reduce(ExePol pol, InpIt first, InpIt last)
3
4 T reduce(InpIt first, InpIt last, T init)
5 T reduce(ExePol pol, InpIt first, InpIt last, T init)
6
7 T reduce(InpIt first, InpIt last, T init, BiFun fun)
8 T reduce(ExePol pol, InpIt first, InpIt last, T init, BiFun fun)

```

transform_reduce: 转换和归约一个或两个范围的元素。Init 是起始值。

- 行为类似于 std::inner_product，但范围可能会重新排列。

- 若应用于两个范围
 - 若没有提供，则使用乘法将范围转换为一个范围，使用加法将中间范围归约为结果
 - 若有提供，`fun1` 用于变换步骤，`fun2` 用于归约步骤
- 若应用于单个范围
 - `fun2` 用于转换给定的范围

```

1 T transform_reduce(InpIt first, InpIt last, InpIt first2, T init)
2 T transform_reduce(InpIt first, InpIt last,
3                   InpIt first2, T init, BiFun fun1, BiFun fun2)
4
5 T transform_reduce(FwdIt first, FwdIt last, FwdIt first2, T init)
6 T transform_reduce(ExePol pol, FwdIt first, FwdIt last,
7                   FwdIt first2, T init, BiFun fun1, BiFun fun2)
8
9 T transform_reduce(InpIt first, InpIt last, T init, BiFun fun1, UnFun fun2)
10 T transform_reduce(ExePol pol, FwdIt first, FwdIt last,
11                   T init, BiFun fun1, UnFun fun2)
```

C++17 中的 MapReduce

[Haskell](#) 函数映射在 C++ 中称为 `std::transform`。当用 `std::transform_reduce` 名称中的 `map` 替换 `transform` 时，将获得 `std::map_reduce`。[MapReduce](#) 是一个著名的并行框架，首先将每个值映射到一个新值，然后在第二阶段将所有值归约到结果。

该算法直接适用于 C++17。在映射阶段，每个单词会映射到它的长度，然后在归约阶段，所有单词的长度会缩减为它们的和。结果是所有单词的长度之和。

```

1 std::vector<std::string> str{"Only", "for", "testing", "purpose"};
2
3 std::size_t result = std::transform_reduce(std::execution::par,
4                                           str.begin(), str.end(), 0,
5                                           [] (std::size_t a, std::size_t b){ return a + b; },
6                                           [] (std::string s){ return s.length(); });
7
8 std::cout << result << '\n'; // 21
```

`exclusive_scan`: 使用二进制操作计算互斥前缀和。

- 行为类似于 `std::reduce`，但提供了所有前缀和的范围
- 排除每次迭代中的最后一个元素

```

1 OutIt exclusive_scan(InpIt first, InpIt last, OutIt first, T init)
2 FwdIt2 exclusive_scan(ExePol pol, FwdIt first, FwdIt last,
3                       FwdIt2 first2, T init)
4
```

```
5 OutIt exclusive_scan(InpIt first, InpIt last, OutIt first, T init, BiFun fun)
6 FwdIt2 exclusive_scan(ExePol pol, FwdIt first, FwdIt last,
7     FwdIt2 first2, T init, BiFun fun)
```

inclusive_scan: 使用二元运算计算包含前缀和。

- 行为类似于 std::reduce，但提供了所有前缀和的范围
- 包括每次迭代中的最后一个元素

```
1 OutIt inclusive_scan(InpIt first, InpIt last, OutIt first2)
2 FwdIt2 inclusive_scan(ExePol pol, FwdIt first, FwdIt last, FwdIt2 first2)
3
4 OutIt inclusive_scan(InpIt first, InpIt last, OutIt first, BiFun fun)
5 FwdIt2 inclusive_scan(ExePol pol, FwdIt first, FwdIt last,
6     FwdIt2 first2, BiFun fun)
7
8 OutIt inclusive_scan(InpIt first, InpIt last, OutIt first2, BiFun fun, T init)
9 FwdIt2 inclusive_scan(ExePol pol, FwdIt first, FwdIt last,
10    FwdIt2 first2, BiFun fun, T init)
```

transform_exclusive_scan: 首先变换每个元素，然后计算互斥前缀和。

```
1 OutIt transform_exclusive_scan(InpIt first, InpIt last, OutIt first2, T init, BiFun
2     ↳ fun, UnFun fun2)
3 FwdIt2 transform_exclusive_scan(ExePol pol, FwdIt first, FwdIt last, FwdIt2 first2, T
4     ↳ init, BiFun fun, UnFun fun2)
```

transform_inclusive_scan: 首先变换输入范围中的每个元素，然后计算包含前缀和。

```
1 OutIt transform_inclusive_scan(InpIt first, InpIt last, OutIt first2,
2     BiFun fun, UnFun fun2)
3
4 FwdIt2 transform_inclusive_scan(ExePol pol, FwdIt first, FwdIt last,
5     FwdIt first2,
6     BiFun fun, UnFun fun2)
7
8 OutIt transform_inclusive_scan(InpIt first, InpIt last, OutIt first2,
9     BiFun fun, UnFun fun2,
10    T init)
11
12 FwdIt2 transform_inclusive_scan(ExePol pol, FwdIt first, FwdIt last,
13     FwdIt first2,
14     BiFun fun, UnFun fun2,
15     T init)
```

下面的示例说明了使用并行执行策略六种算法的用法。

新算法

```

1 // newAlgorithms.cpp
2 ...
3 #include <execution>
4 #include <numeric>
5 ...
6 ...
7 std::vector<int> resVec{1, 2, 3, 4, 5, 6, 7, 8};
8 std::exclusive_scan(std::execution::par,
9                     resVec.begin(), resVec.end(), resVec.begin(), 1,
10                    [](int fir, int sec){ return fir * sec; });
11
12 for (auto v: resVec) std::cout << v << " "; // 1 1 2 6 24 120 720 5040
13
14 std::vector<int> resVec2{1, 2, 3, 4, 5, 6, 7, 8};
15
16 std::inclusive_scan(std::execution::par,
17                      resVec2.begin(), resVec2.end(), resVec2.begin(),
18                      [](int fir, int sec){ return fir * sec; }, 1);
19
20 for (auto v: resVec2) std::cout << v << " "; // 1 2 6 24 120 720 5040 40320
21
22 std::vector<int> resVec3{1, 2, 3, 4, 5, 6, 7, 8};
23 std::vector<int> resVec4(resVec3.size());
24 std::transform_exclusive_scan(std::execution::par,
25                             resVec3.begin(), resVec3.end(),
26                             resVec4.begin(), 0,
27                             [](int fir, int sec){ return fir + sec; },
28                             [](int arg){ return arg *= arg; });
29
30 for (auto v: resVec4) std::cout << v << " "; // 0 1 5 14 30 55 91 140
31
32 std::vector<std::string> strVec{"Only", "for", "testing", "purpose"};
33 std::vector<int> resVec5(strVec.size());
34 std::transform_inclusive_scan(std::execution::par,
35                             strVec.begin(), strVec.end(),
36                             resVec5.begin(),
37                             [] (auto fir, auto sec){ return fir + sec; },
38                             [] (auto s){ return s.length(); }, 0);
39
40 for (auto v: resVec5) std::cout << v << " "; // 4 7 14 21
41
42 std::vector<std::string> strVec2{"Only", "for", "testing", "purpose"};
43
44 std::string res = std::reduce(std::execution::par,
45                             strVec2.begin() + 1, strVec2.end(), strVec2[0],
46                             [] (auto fir, auto sec){ return fir + ":" + sec; });
47
48

```

```

49 std::cout << res; // Only:for:testing:purpose
50
51 std::size_t res7 = std::transform_reduce(std::execution::par,
52 strVec2.begin(), strVec2.end(), 0,
53 [](std::size_t a, std::size_t b){ return a + b; },
54 [](std::string s){ return s.length(); });
55
56 std::cout << res7; // 21

```

10.15. 单元化内存

头文件 `<memory>` 中的下列函数对未初始化的内存进行操作。

将一个对象范围复制到一个统一的单元化内存中。

```

1 FwdIt uninitialized_copy(Init first, Init last, FwdIt out)
2 FwdIt uninitialized_copy(ExePol pol, FwdIt first, FwdIt last, FwdIt out)

```

将 `count` 对象复制到一个单元化内存中。

```

1 FwdIt uninitialized_copy_n(Init first, Size count, FwdIt out)
2 FwdIt uninitialized_copy_n(ExePol pol, FwdIt first, Size count, FwdIt out)

```

将对象值复制到内存的单元化内存中。

```

1 void uninitialized_fill(FwdIt first, FwdIt last, const T& value)
2 void uninitialized_fill(ExePol pol, FwdIt first, FwdIt last, const T& value)

```

从 `first` 开始，将计数对象复制到一个单元化内存中。

```

1 void uninitialized_fill_n(FwdIt first, Size count, const T& value)
2 FwdIt uninitialized_fill_n(FwdIt first, Size count, const T& value)
3 FwdIt uninitialized_fill_n(ExePol pol, FwdIt first, Size count, const T& value)

```

将一个对象范围移动到一个统一的内存区域。

```

1 FwdIt uninitialized_move(Init first, Init last, FwIt out)
2 FwdIt uninitialized_move(ExePol pol, FwdIt first, FwdIt last, FwIt out)

```

将 `count` 对象移动到一个单元化内存。

```

1 std::pair<Init, FwdIt> uninitialized_move_n(Init first, Size count, FwIt out)
2 std::pair<FwdIt, FwdIt> uninitialized_move_n(ExePol pol, FwdIt first, Size count,
3 FwIt out)

```

通过[默认初始化](#)(一个单元化内存) 构造对象。

```
1 void uninitialized_default_construct(FwdIt first, FwdIt last)
2 void uninitialized_default_construct(ExePol pol, FwdIt first, FwdIt last)
```

通过[默认初始化](#)(一个单元化内存) 构造对象。

```
1 FwdIt uninitialized_default_construct_n(FwdIt first, Size n)
2 FwdIt uninitialized_default_construct_n(ExePol pol, FwdIt first, Size n)
```

通过[值初始化](#)(一个单元化内存) 构造对象。

```
1 void uninitialized_value_construct(FwdIt first, FwdIt last)
2 void uninitialized_value_construct(ExePol pol, FwdIt first, FwdIt last)
```

通过[值初始化](#)(一个单元化内存) 构造 n 个对象。

```
1 FwdIt uninitialized_value_construct_n(FwdIt first, Size n)
2 FwdIt uninitialized_value_construct_n(ExePol pol, FwdIt first, Size n)
```

销毁对象。

```
1 constexpr void destroy(FwdIt first, FwdIt last)
2 void destroy(ExePol pol, FwdIt first, FwdIt last)
```

销毁 n 个对象。

```
1 constexpr void destroy_n(FwdIt first, Size n)
2 void destroy_n(ExePol pol, FwdIt first, Size n)
```

销毁给定地址 p 上的对象。

```
1 constexpr void destroy_at(T* p)
```

在给定地址 p 上创建对象。

```
1 constexpr T* construct_at(T* p, Args&& ... args)
```

第 11 章 范围库



Cippi 启动流水线

范围库是在 C++20 中添加的，但在 C++23 中进行扩展。范围库具有惰性算法，直接在容器上操作，并且可以组合。此外，大多数经典 STL 算法都具有范围挂件，支持投影并提供额外的安全保证。

范围的组成

```
1 // rangesFilterTransform.cpp
2 ...
3 #include <ranges>
4
5 std::vector<int> numbers = {1, 2, 3, 4, 5, 6};
6
7 auto results = numbers | std::views::filter([](int n){ return n % 2 == 0; })
8     | std::views::transform([](int n){ return n * 2; });
9
10 for (auto v: results) std::cout << v << " "; // 4 8 12
```

必须从左到右读这个表达式。管道符号 | 代表函数组合：首先，所有偶数都可以通过 (std::views::filter([](int n){return n % 2 == 0;}))。之后，每个剩余的数字映射到双倍结果 (std::views::transform([](int n){return n * 2;}))。

11.1. 范围

范围和视图为[概念](#)。C++20 支持多种范围：

std::range

范围是一组可以迭代的项，提供了开始迭代器和结束迭代器，结束迭代器是个特殊的哨兵。

std::range 还有进一步的改进：

std::range 的改进

范围 (省略命名空间 std)	描述
ranges::input_range	指定一个迭代器类型满足 input_iterator 的范围
ranges::output_range	指定一个迭代器类型满足 output_iterator 的范围
ranges::forward_range	指定一个迭代器类型满足 forward_iterator 的范围
ranges::bidirectional_range	指定一个迭代器类型满足 bidirectional_iterator 的范围
ranges::random_access_range	指定一个迭代器类型满足 random_access_iterator 的范围
ranges::contiguous_range	指定一个迭代器类型满足 contiguous_iterator 的范围

random_access_iterator 提供对其元素的随机访问，是隐式 bidirectional_iterator; bidirectional_iterator 允许在两个方向上迭代，并且是隐式 forward_iterator; 一个方向上的 forward_iterator。contiguous_iterator 的接口都相同，例如 bidirectional_iterator 接口。contiguous_iterator 保证迭代器指向连续存储的元素。

哨兵

哨兵指定一个范围的结束。对于 STL 的容器，结束迭代器是哨兵。C++20 中，哨兵的类型可以不同于起始迭代器的类型。下面的示例使用空格作为哨兵。

范围的组成

```
1 // sentinelSpace.cpp
2 ...
3 #include <algorithm>
4 ...
5 ...
6
7 struct Space {
8     bool operator==(auto pos) const {
9         return *pos == ' ';
10    }
11 ...
12 ...
13
14 const char* rainerGrimm = "Rainer Grimm";
15 std::ranges::for_each(rainerGrimm, Space{}, [] (char c) { std::cout << c; }); // ← Rainer
```

由于空格作为哨兵，最后一行只显示 Rainer。

ranges::to

std::ranges::to 是 C++23 中使用范围构造容器的一种简单方法：

```

1 std::vector<int> range(int begin, int end, int stepsize = 1) {
2     auto boundary = [end](int i){ return i < end; };
3     std::vector<int> result = std::ranges::views::iota(begin)
4         | std::views::stride(stepsize)
5         | std::views::take_while(boundary)
6         | std::ranges::to<std::vector>();
7     return result;
8 }
```

函数 `range` 创建一个 `std::vector<int>`, 由从开始到结束的所有元素组成。`begin` 必须小于 `end`。

11.2. 视图

视图是轻量级范围。视图允许访问范围、遍历范围或修改或过滤范围的元素。视图不拥有数据，其复制、移动或分配的时间复杂度恒定。

11.3. 范围适配器

范围适配器将范围转换为视图。

```

1 std::vector<int> numbers = {1, 2, 3, 4, 5, 6};
2 auto results = numbers | std::views::filter([](int n){ return n % 2 == 0; })
3     | std::views::transform([](int n){ return n * 2; });
```

代码片段中, `numbers` 是范围, `std::views::filter` 和 `std::views::transform` 是视图。

C++20 中的范围库有一组丰富的视图。

C++20 中的视图

视图	描述
<code>std::views::all_t</code> <code>std::views::all</code>	获取所有元素。
<code>std::ranges::ref_view</code>	获取另一个范围的所有元素。
<code>std::ranges::filter_view</code> <code>std::views::filter</code>	获取满足谓词的元素。
<code>std::ranges::transfrom_view</code> <code>std::views::transform</code>	转换每个元素。
<code>std::ranges::take_view</code> <code>std::views::take</code>	取另一个视图的前 <code>n</code> 个元素。
<code>std::ranges::take_while_view</code> <code>std::views::take_while</code>	只要谓词返回 <code>true</code> , 就接受另一个视图的元素。

<code>std::ranges::drop_view</code> <code>std::views::drop</code>	跳过另一个视图的前 n 个元素。
<code>std::ranges::drop_while_view</code> <code>std::views::drop_while</code>	跳过另一个视图的初始元素，直到谓词返回 false。
<code>std::ranges::join_view</code> <code>std::views::join</code>	加入范围视图。
<code>std::ranges::split_view</code> <code>std::views::split</code>	使用分隔符拆分视图。
<code>std::ranges::common_view</code> <code>std::views::common</code>	将视图转换为 <code>std::ranges::common_range</code> 。
<code>std::ranges::reverse_view</code> <code>std::views::reverse</code>	以反向顺序迭代。
<code>std::ranges::basic_istream_view</code> <code>std::views::istream_view</code>	可对相应视图使用操作符 <code>>></code> 。
<code>std::ranges::elements_view</code> <code>std::views::elements</code>	在元组的第 n 个元素上创建视图。
<code>std::ranges::keys_views</code> <code>std::views::keys</code>	在类似对值的第一个元素上创建视图。
<code>std::ranges::values_views</code> <code>std::views::values</code>	在类似对值的第二个元素上创建视图。

通常，可以使用 `std::views::transform` 这样的视图，并将其替换为 `std::ranges::transform_view`。有了范围库，算法可以直接应用于容器，可以组合，并且是惰性的。

C++23 支持更多的视图：

C++23 中的视图

视图	描述
<code>std::ranges::zip_view</code> <code>std::views::zip</code>	创建元组的视图。
<code>std::ranges::zip_transform_view</code> <code>std::views::zip_transform</code>	通过转换函数创建元组的视图。
<code>std::ranges::adjacent_view</code> <code>std::views::adjacent</code>	创建相邻元素的视图。
<code>std::ranges::adjacent_transform_view</code> <code>std::views::adjacent_transform</code>	通过转换函数创建相邻元素的视图。
<code>std::ranges::join_with_view</code> <code>std::views::join_with</code>	通过分隔符将现有范围连接到视图中。

std::ranges::slide_view std::views::slide	通过获取一个视图和一个数字 N 来创建 N 个元组。
std::ranges::chunk_view std::views::chunk	创建 N 个视图块和一个数字 N。
std::ranges::chunk_by_view std::views::chunk_by	基于谓词创建视图块。
std::ranges::as_const_view std::views::as_const	将视图转换为常量范围。
std::ranges::as_rvalue_view std::views::as_rvalue	将每个元素强制转换为右值。
std::ranges::stride_view std::views::stride	创建另一个视图第 n 个元素的视图。

下面的代码段中使用了 C++23 视图。

C++23 中的新视图

```

1 // cpp23Ranges.cpp
2 ...
3 #include <ranges>
4 ...
5
6 std::vector vec = {1, 2, 3, 4};
7
8 for (auto i : vec | std::views::adjacent<2>) {
9     std::cout << '(' << i.first << ", " << i.second << ") "; // (1, 2) (2, 3) (3, 4)
10 }
11
12 for (auto i : vec | std::views::adjacent_transform<2>(std::multiplies())) {
13     std::cout << i << ' '; // 2 6 12
14 }
15
16 std::print("{}\n", vec | std::views::chunk(2)); // [[1, 2], [3, 4],
17 std::print("{}\n", vec | std::views::slide(2)); // [[1, 2], [2, 3], [3, 4]
18 for (auto i : vec | std::views::slide(2)) {
19     std::cout << '[' << i[0] << ", " << i[1] << "] "; // [1, 2] [2, 3] [3, 4] [4, 5]
20 }
21
22 std::vector vec2 = {1, 2, 3, 0, 5, 2};
23 std::print("{}\n", vec2 | std::views::chunk_by(std::ranges::less_equal{}));
24 // [[1, 2, 3], [0, 5], [2]]
25
26 for (auto i : vec | std::views::slide(2)) {
27     std::cout << '[' << i[0] << ", " << i[1] << "] "; // [1, 2] [2, 3] [3, 4] [4, 5]
28 }
```

生成器

C++23 中的 `std::generator` 是第一个具体的协程生成器，生成器通过反复恢复暂停协程来生成元素序列。

协程生成器 `std::generator`

```
1 // generator.cpp
2 ...
3 #include <generator>
4 #include <ranges>
5 ...
6 ...
7 ...
8 std::generator<int> fib() {
9     co_yield 0; // 1
10    auto a = 0;
11    auto b = 1;
12    for(auto n : std::views::iota(0)) {
13        auto next = a + b;
14        a = b;
15        b = next;
16        co_yield next; // 2
17    }
18 }
19 ...
20 ...
21 ...
22 for (auto f : fib() | std::views::take(10)) { // 3
23     std::cout << f << " "; // 0 1 1 2 3 5 8 13 21 34
24 }
```

函数 `fib` 返回一个协程，协程创建了一个无限的斐波那契数流。数字流从 0(1) 开始，继续到后面的斐波那契数 (2)。基于范围的 for 循环显式地请求前 10 个斐波那契数 (3)。

11.4. 直接用于容器

标准模板库的算法需要一个开始迭代器和一个结束迭代器。

```
1 std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9};
2 auto res = std::accumulate(std::begin(myVec), std::end(myVec), 0);
3 std::cout << res << '\n'; // 45
```

范围库允许直接在 `std::unordered_map` 的键 (1) 或值 (3) 上创建视图。

范围直接作用于容器

```

1 // rangesEntireContainer.cpp
2 ...
3 #include <ranges>
4
5 std::unordered_map<std::string, int> freqWord{ {"witch", 25}, {"wizard", 33},
6                                         {"tale", 45}, {"dog", 4},
7                                         {"cat", 34}, {"fish", 23} };
8 std::cout << "Keys" << '\n';
9 auto names = std::views::keys(freqWord); // (1)
10 for (const auto& name : names){ std::cout << name << " "; };
11
12 for (const auto& na : std::views::keys(freqWord)){ std::cout << na << " "; }; // (2)
13
14 std::cout << "Values: " << '\n';
15 auto values = std::views::values(freqWord); // (3)
16 for (const auto& value : values){ std::cout << value << " "; };
17
18 for (const auto& value : std::views::values(freqWord)){ // (4)
19     std::cout << value << " ";
20 }

```

当然，键和值可以直接显示 ((2) 和 (4))，输出是相同的。

```

Start
Keys
fish cat tale dog wizard witch
fish cat tale dog wizard witch

Values:
23 34 45 4 33 25
23 34 45 4 33 25

0

Finish

```

11.5. 函数复合

范围库支持使用 | 符号组合函数。

范围组合

```

1 // rangesComposition.cpp
2 ...
3 #include <ranges>
4 std::map<std::string, int> freqWord{ {"witch", 25}, {"wizard", 33}, {"tale", 45},
5                                         {"dog", 4}, {"cat", 34}, {"fish", 23} };

```

```

6
7 std::cout << "All words: " // (1)
8 for (const auto& name : std::views::keys(freqWord)) { std::cout << name << " "; } \
9
10 std::cout << "All words reverse: " // (2)
11 for (const auto& name : std::views::keys(freqWord) | std::views::reverse) {
12     std::cout << name << " ";
13 }
14
15 std::cout << "The first 4 words: " // (3)
16 for (const auto& name : std::views::keys(freqWord) | std::views::take(4)) {
17     std::cout << name << " ";
18 }
19
20 std::cout << "All words starting with w: " // (4)
21 auto firstw = [] (const std::string& name){ return name[0] == 'w'; };
22 for (const auto& name : std::views::keys(freqWord) | std::views::filter(firstw)) {
23     std::cout << name << " ";
24 }

```

这种情况下，我只对键值感兴趣。显示了所有键(1)，所有键的反转(2)，前四个键(3)，以及以字母“w”开头的键(4)。

管道符号|是函数组成的语法糖。而不是 C(R)，也可以写 R | C。因此，接下来的三行是等价的。

```

1 auto rev1 = std::views::reverse(std::views::keys(freqWord));
2 auto rev2 = std::views::keys(freqWord) | std::views::reverse;
3 auto rev3 = freqWord | std::views::keys | std::views::reverse;

```

最后，程序的输出为：

```

Start
All words: cat dog fish tale witch wizard
All words reverse: wizard witch tale fish dog cat
The first 4 words: cat dog fish tale
All words starting with w: witch wizard

0
Finish

```

11.6. 惰性计算

`std::views::iota`是一个范围工厂，用于通过连续递增初始值来创建元素序列。这个序列可以是有限的，也可以是无限的。有了这个函数，就可以找到以 1000000 开头的前 20 个质数。

找出 20 个以 1000000 开头的质数

```

1 // rangesLazy.cpp
2 ...
3 #include <ranges>
4
5 bool isPrime(int i) {
6     for (int j=2; j*j <= i; ++j) {
7         if (i % j == 0) return false;
8     }
9     return true;
10 }
11
12 std::cout << "Numbers from 1000000 to 1001000 (displayed each 100th): " << '\n';
13 for (int i: std::views::iota(1000000, 1001000)) {
14     if (i % 100 == 0) std::cout << i << " ";
15 }
16
17 auto odd = [] (int i){ return i % 2 == 1; };
18 std::cout << "Odd numbers from 1000000 to 1001000 (displayed each 100th): " << '\n';
19 for (int i: std::views::iota(1000000, 1001000) | std::views::filter(odd)) {
20     if (i % 100 == 1) std::cout << i << " ";
21 }
22
23
24 std::cout << "Prime numbers from 1000000 to 1001000: " << '\n';
25 for (int i: std::views::iota(1000000, 1001000) | std::views::filter(odd)
26             | std::views::filter(isPrime)) {
27     std::cout << i << " ";
28 }
29
30 std::cout << "20 prime numbers starting with 1000000: " << '\n';
31 for (int i: std::views::iota(1000000) | std::views::filter(odd)
32             | std::views::filter(isPrime)
33             | std::views::take(20)) {
34     std::cout << i << " ";
35 }

```

以下是我的迭代策略：

1. 我不知道什么时候有 20 个大于 1000000 的质数。为了安全起见，我创建了 1000 个数字，只显示每 100 个。
2. 素数是奇数；因此，去掉了偶数。
3. 谓词 isPrime 返回一个数字是否为质数。我得到 75 个质数，但我只想要 20 个。
4. 我使用 std::iota 作为无限数字工厂，从 1000000 开始，并精确地要求 20 个质数。

```

Numbers from 1000000 to 1001000 (displayed each 100th):
1000000 1000100 1000200 1000300 1000400 1000500 1000600 1000700 1000800 1000900

Odd numbers from 1000000 to 1001000 (displayed each 100th):
1000001 1000101 1000201 1000301 1000401 1000501 1000601 1000701 1000801 1000901

Prime numbers from 1000000 to 1001000:
1000003 1000033 1000037 1000081 1000099 1000117 1000121 1000133 1000151 1000159 1000171 1000183 1000187 1000193 1000199 1000211 1000213
1000231 1000249 1000253 1000273 1000289 1000291 1000303 1000333 1000357 1000367 1000381 1000393 1000397 1000403 1000409 1000423 1000427
1000429 1000453 1000457 1000507 1000537 1000541 1000547 1000577 1000579 1000589 1000609 1000619 1000621 1000639 1000651 1000667 1000669 1000679
1000691 1000697 1000721 1000723 1000763 1000777 1000793 1000829 1000847 1000849 1000859 1000861 1000889 1000907 1000919 1000921 1000931 1000969
1000973 1000981 1000999

20 prime numbers starting with 1000000:
1000003 1000033 1000037 1000081 1000099 1000117 1000121 1000133 1000151 1000159 1000171 1000183 1000187 1000193 1000199 1000211 1000213
1000231 1000249

```

11.7. std 的算法与 std::ranges 的算法

算法库和内存库的算法有[范围挂件](#)，其以命名空间 std::ranges 开头。[数值库](#)没有范围挂件。在下面的代码中，我展示了 std::sort 算法的五个重载中的一个，以及新的 std::ranges::sort 算法的两个重载中的一个。

```

1 template< class ExecutionPolicy, class RandomIt, class Compare >
2 void sort( ExecutionPolicy&& policy,
3     RandomIt first, RandomIt last, Compare comp );
4
5 template <std::random_access_iterator I, std::sentinel_for<I> S,
6     class Comp = ranges::less, class Proj = std::identity>
7 requires std::sortable<I, Comp, Proj>
8 constexpr I sort(I first, S last, Comp comp = {}, Proj proj = {});

```

首先，std::sort 接受 begin 和 end 迭代器给出的范围，迭代器必须是随机访问迭代器，并需要一个执行策略。Compare 允许指定 std::sort 的排序策略。

此外，当研究 std::ranges::sort 的重载时，它接受一个由随机访问 begin 迭代器和哨兵定义的范围。此外，这个重载接受一个谓词 Comp 和一个投影 Proj。Comp 使用谓词表示 default less，而投影表示 std::identity。投影是一个集合到一个子集的映射：

```

1 struct PhoneBookEntry{
2     std::string name;
3     int number;
4 };
5
6 std::vector<PhoneBookEntry> phoneBook{ {"Brown", 111}, {"Smith", 444}, {"Grimm",
7     666},
8     {"Butcher", 222}, {"Taylor", 555}, {"Wilson", 333} };
9
std::ranges::sort(phoneBook, std::ranges::greater(), &PhoneBookEntry::name);

```

phoneBook 根据投影 &PhoneBookEntry::name 降序排序。

std::random_access_iterator, std::sortable 和 std::sentinel_for 都是概念。

std::ranges::sort 不支持执行策略。

第 12 章 数值



Cippi 正在学习数学

C++ 继承了 C 的数学函数，还有一个随机数库。

12.1. 随机数

随机数是许多领域所必需的，例如，测试软件，生成加密密钥，或电脑游戏。C++ 的随机数功能由两个部分组成，是随机数的生成和这些随机数的分布。这两个部分都需要头文件 `<random>`。

随机数生成器

随机数生成器在最小值和最大值之间生成随机数流，流由“所谓的”种子初始化，保证不同的随机数序列。

```
1 #include <random>
2 ...
3 std::random_device seed;
4 std::mt19937 generator(seed);
```

generator 类型的随机数生成器支持四种不同的请求：

Generator::result_type

生成随机数的数据类型。

gen()

返回一个随机数。

gen.min()

返回 gen() 可以返回的最小随机数。

gen.max()

返回 gen 可产生的最大随机数。

随机数库支持几个随机数生成器。最著名的是 Mersenne Twister, `std::default_random_engine` 和 `std::random_device`。`Std::random_device` 是唯一真正的随机数生成器，但并非所有平台都有提供。

随机数分布

随机数分布将具有随机数生成器的随机数映射到所选分布。

```
1 #include <random>
2 ...
3
4 std::random_device seed;
5 std::mt19937 gen(seed());
6 std::uniform_int_distribution<> unDis(0, 20); // distribution between 0 and 20
7 unDis(gen); // generates a random number
```

C++ 有几个离散的和连续的随机数分布。离散随机数分布生成整数，连续随机数分布生成浮点数。

```
1 class bernoulli_distribution;
2 template<class T = int> class uniform_int_distribution;
3 template<class T = int> class binomial_distribution;
4 template<class T = int> class geometric_distribution;
5 template<class T = int> class negative_binomial_distribution;
6 template<class T = int> class poisson_distribution;
7 template<class T = int> class discrete_distribution;
8 template<class T = double> class exponential_distribution;
9 template<class T = double> class gamma_distribution;
10 template<class T = double> class weibull_distribution;
11 template<class T = double> class extreme_value_distribution;
12 template<class T = double> class normal_distribution;
13 template<class T = double> class lognormal_distribution;
14 template<class T = double> class chi_squared_distribution;
15 template<class T = double> class cauchy_distribution;
16 template<class T = double> class fisher_f_distribution;
17 template<class T = double> class student_t_distribution;
18 template<class T = double> class piecewise_constant_distribution;
19 template<class T = double> class piecewise_linear_distribution;
20 template<class T = double> class uniform_real_distribution;
```

带有默认模板实参 int 的类模板是离散的。伯努利分布会产生布尔值。

下面是一个使用 Mersenne Twister std::mt19937 作为伪随机数生成器生成一百万个随机数的示例。随机数流映射为均匀和正态 (或高斯) 分布。

随机数

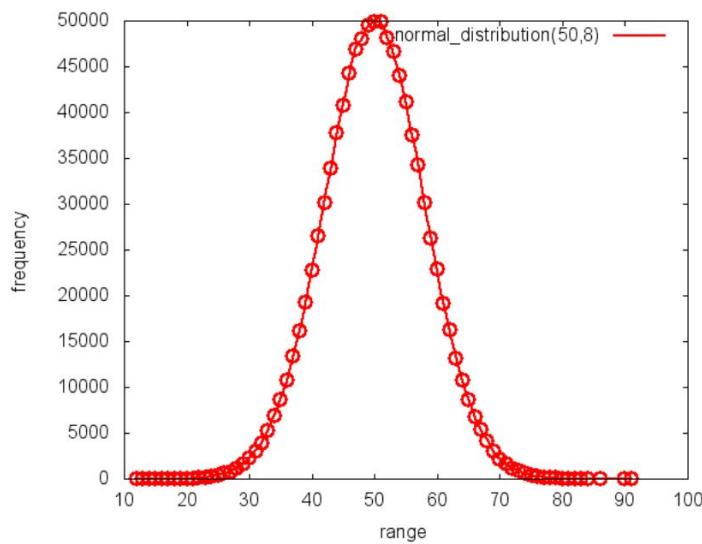
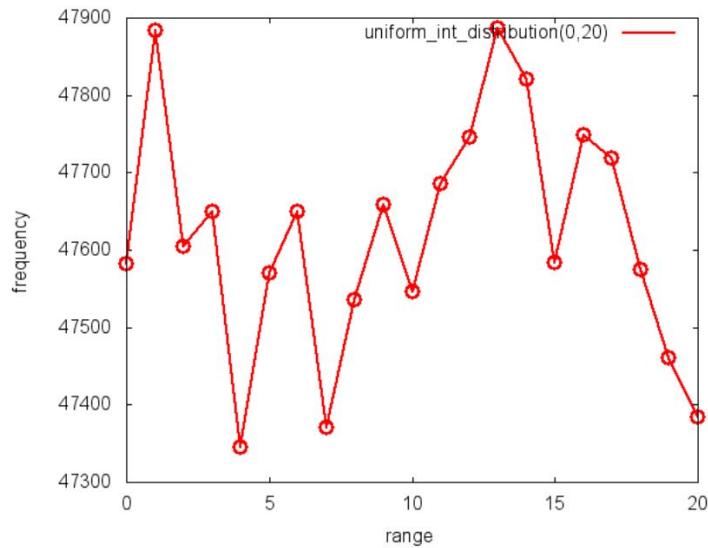
```
1 // random.cpp
2 ...
3 #include <random>
4 ...
5
6 static const int NUM= 1000000;
7 std::random_device seed;
8 std::mt19937 gen(seed());
```

```

9 std::uniform_int_distribution<> uniformDist(0, 20); // min= 0; max= 20
10 std::normal_distribution<> normDist(50, 8); // mean= 50; sigma= 8
11
12 std::map<int, int> uniformFrequency;
13 std::map<int, int> normFrequency;
14 for (int i= 1; i <= NUM; ++i){
15     ++uniformFrequency[uniformDist(gen)];
16     ++normFrequency[round(normDist(gen))];
17 }

```

下图显示了 100 万个随机数的均匀分布和正态分布。



12.2. 从 C 继承的数值函数

C++ 从 C 继承了许多数值函数，需要头文件[`<cmath>`](#)。下表显示了这些函数的名称。

[`<cmath>`](#) 中的数学函数

```
pow    sin     tanh    asinh    fabs  
exp    cos     asin    aconsh   fmod  
sqrt   tan     acos    atanh    frexp  
log    sinh    atan    ceil     ldexp  
log10  cosh    atan2   floor    modf
```

另外，C++ 从 C 中继承了数学函数。它们在头文件[`<cstdlib>`](#)中定义。

再说一遍，这些都是名字。

`<cstdlib>` 中的数学函数

```
abs    llabs   ldiv    srand  
labs    div    lldiv   rand
```

所有用于整型的函数都可用于 `int`、`long` 和 `long long`；所有用于浮点数的函数都可用于 `float`、`double` 和 `long double` 类型。

数值函数需要使用命名空间 `std` 进行限定。

数学函数

```
1 // mathFunctions.cpp  
2 ...  
3 #include <cmath>  
4 #include <cstdlib>  
5 ...  
6  
7 std::cout << std::pow(2, 10); // 1024  
8 std::cout << std::pow(2, 0.5); // 1.41421  
9 std::cout << std::exp(1); // 2.71828  
10 std::cout << std::ceil(5.5); // 6  
11 std::cout << std::floor(5.5); // 5  
12 std::cout << std::fmod(5.5, 2); // 1.5  
13  
14 double intPart;  
15 auto fracPart= std::modf(5.7, &intPart);  
16 std::cout << intPart << " + " << fracPart; // 5 + 0.7  
17 std::div_t divresult= std::div(14, 5);  
18 std::cout << divresult.quot << " " << divresult.rem; // 2 4  
19  
20 // seed  
21 std::srand(time(nullptr));  
22 for (int i= 0;i < 10; ++i) std::cout << (rand()%6 + 1) << " "  
23 // 3 6 5 3 6 5 6 3 1 5
```

12.3. 数学常数

C++ 支持基本和高级数学常数。数学常数的数据类型为 `double`, 其位于命名空间 `std::numbers` 中, 是头文件 `<numbers>` 的一部分。

数学常数

数学常数	表示
<code>std::numbers::e</code>	e
<code>std::numbers::log2e</code>	$\log_2 e$
<code>std::numbers::log10e</code>	$\log_{10} e$
<code>std::numbers::pi</code>	π
<code>std::numbers::inv_pi</code>	$\frac{1}{\pi}$
<code>std::numbers::inv_sqrtpi</code>	$\frac{1}{\sqrt{\pi}}$
<code>std::numbers::ln2</code>	$\ln 2$
<code>std::numbers::ln10</code>	$\ln 10$
<code>std::numbers::sqrt2</code>	$\sqrt{2}$
<code>std::numbers::sqrt3</code>	$\sqrt{3}$
<code>std::numbers::inv_sqrt3</code>	$\frac{1}{\sqrt{3}}$
<code>std::numbers::egamma</code>	Euler-Mascheroni constant
<code>std::numbers::phi</code>	ϕ

下面的代码段展示了所有的数学常数。

数学常数

```
1 // mathematicalConstants.cpp
2 #include <numbers>
3 ...
4
5 std::cout << std::setprecision(10);
6
7 std::cout << "std::numbers::e: " << std::numbers::e << '\n';
8 std::cout << "std::numbers::log2e: " << std::numbers::log2e << '\n';
9 std::cout << "std::numbers::log10e: " << std::numbers::log10e << '\n';
10 std::cout << "std::numbers::pi: " << std::numbers::pi << '\n';
11 std::cout << "std::numbers::inv_pi: " << std::numbers::inv_pi << '\n';
12 std::cout << "std::numbers::inv_sqrtpi: " << std::numbers::inv_sqrtpi << '\n';
13 std::cout << "std::numbers::ln2: " << std::numbers::ln2 << '\n';
14 std::cout << "std::numbers::sqrt2: " << std::numbers::sqrt2 << '\n';
15 std::cout << "std::numbers::sqrt3: " << std::numbers::sqrt3 << '\n';
16 std::cout << "std::numbers::inv_sqrt3: " << std::numbers::inv_sqrt3 << '\n';
17 std::cout << "std::numbers::egamma: " << std::numbers::egamma << '\n';
18 std::cout << "std::numbers::phi: " << std::numbers::phi << '\n';
```

```
Windows PowerShell

C:\Users\rainer>mathematicalConstants.exe

std::numbers::e: 2.718281828
std::numbers::log2e: 1.442695041
std::numbers::log10e: 0.4342944819
std::numbers::pi: 3.141592654
std::numbers::inv_pi: 0.3183098862
std::numbers::inv_sqrtpi: 0.5641895835
std::numbers::ln2: 0.6931471806
std::numbers::sqrt2: 1.414213562
std::numbers::sqrt3: 1.732050808
std::numbers::inv_sqrt3: 0.5773502692
std::numbers::egamma: 0.5772156649
std::numbers::phi: 1.618033989

C:\Users\rainer>
```

第 13 章 字符串



Cippi 正在玩蛇型玩具

string是一个字符序列。C++有许多成员函数来分析或修改字符串，C++字符串是C字符串的安全替代品:`const char*`。`std::string`需要头文件`<string>`。



字符串类似于 `std::vector`

字符串感觉像是包含字符的 `std::vector`，支持一个相似的接口，可使用标准模板库的算法来操作字符串。

下面的代码段使用 `std::string` 命名，值为 RainerGrimm。我使用 STL 算法 `std::find_if` 来获取大写字母，然后将我的名字和姓氏提取到变量 `firstName` 和 `lastName` 中。表达式 `name.begin() + 1` 表明，字符串支持随机访问迭代器：

`string` 和 `vector`

```
1 // string.cpp
2 ...
3 #include <algorithm>
4 #include <string>
5
6 std::string name{ "RainerGrimm" };
7 auto strIt= std::find_if(name.begin() + 1, name.end(),
8     [] (char c){ return std::isupper(c); });
9 if (strIt != name.end()){
10     firstName= std::string(name.begin(), strIt);
11     lastName= std::string(strIt, name.end());
12 }
```

字符串是由其字符、字符特征和分配器参数化的类模板，字符特征和分配器都有默认值。

```

1 template <typename charT,
2     typename traits= char_traits<charT>,
3     typename Allocator= allocator<charT> >
4 class basic_string;

```

C++ 有字符类型 `char`、`wchar_t`、`char16_t` 和 `char32_t` 的同义词

```

1 typedef basic_string<char> string;
2 typedef basic_string<wchar_t> wstring;
3 typedef basic_string<char16_t> u16string;
4 typedef basic_string<char32_t> u32string;

```

`std::string` 是字符串

若在 C++ 中谈论字符串，有 99% 的概率引用字符类型 `char` 的特化 `std::basic_string`，这也适用于本书。

13.1. 创建和删除

C++ 提供了许多成员函数来从 C 或 C++ 字符串中创建字符串。底层 C 字符串总是用于创建 C++ 字符串，这在 C++14 中有所改变，因为新的 C++ 标准支持 C++ 字符串字面值:`std::string str{"string"s}`。C 字符串字面值”`string literal`”加上后缀 `s` 就变成了 C++ 字符串字面值”`string literal`”`s`。

该表概述了用于创建和删除 C++ 字符串的成员函数。

用于创建和删除字符串的函数

成员函数	示例
默认构造	<code>std::string str</code>
从 C++ 字符串复制	<code>std::string str(oth)</code>
从 C++ 字符串移动	<code>std::string str(std::move(oth))</code>
使用 C++ 字符串的范围	<code>std::string(oth.begin(), oth.end())</code>
使用 C++ 字符串的子字符串	<code>std::string(oth, otherIndex)</code>
使用 C++ 字符串的子字符串	<code>std::string(oth, otherIndex, strlen)</code>
使用 C 字符串	<code>std::string str("c-string")</code>
使用 C 数组	<code>std::string str("c-array", len)</code>
使用字符	<code>std::string str(num, 'c')</code>
使用初始化列表	<code>std::string str({'a', 'b', 'c', 'd'})</code>
使用子字符串	<code>str = other.substring(3, 10)</code>

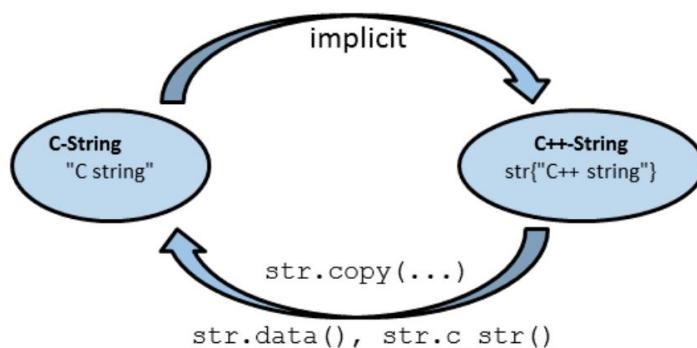
创建字符串

```

1 // stringConstructor.cpp
2 ...
3 #include <string>
4 ...
5 std::string defaultString;
6 std::string other{"123456789"};
7 std::string str1(other); // 123456789
8 std::string tmp(other); // 123456789
9 std::string str2(std::move(tmp)); // 123456789
10 std::string str3(other.begin(), other.end()); // 123456789
11 std::string str4(other, 2); // 3456789
12 std::string str5(other, 2, 5); // 34567
13 std::string str6("123456789", 5); // 12345
14 std::string str7(5, '1'); // 11111
15 std::string str8({'1', '2', '3', '4', '5'}); // 12345
16 std::cout << str6.substr(); // 12345
17 std::cout << str6.substr(1); // 2345
18 std::cout << str6.substr(1, 2); // 23

```

13.2. C++ 和 C 字符串之间的转换



虽然将 C 字符串隐式地转换为 C++ 字符串，但必须显式地请求从 C++ 字符串转换为 C 字符串。
`str.copy()` 复制 C++ 字符串的内容，但不包含终止符 `\0`。`str.data()` 和 `str.c_str()` 包含终止字符。

注意 `str.data()` 和 `str.c_str()`

若修改了 `str`，则两个成员函数 `str.data()` 和 `str.c_str()` 的返回值无效。

C 和 C++ 字符串

```

1 // stringCversusC++.cpp
2 ...
3 #include<string>
4 ...
5
6 std::string str{"C++-String"};
7 str += " C-String";
8 std::cout << str; // C++-String C-String
9 const char* cString= str.c_str();
10 char buffer[10];
11 str.copy(buffer, 10);
12 str+= "works";
13 // const char* cString2= cString; // ERROR
14 std::string str2(buffer, buffer+10);
15 std::cout<< str2; // C++-String

```

13.3. 大小与容量

字符串拥有的元素数量 (str.size()) 通常小于保留空间的元素数量: str.capacity()。若向字符串中添加元素，将不会自动分配新的内存。std::max_size() 返回一个字符串最大可以包含多少个元素。对于这三个成员函数，下列关系成立: str.size() <= str.capacity() <= str.max_size()。

下表展示了处理字符串的内存管理的成员函数。

用于创建和删除字符串的函数

成员函数	描述
str.empty()	检查 str 是否有元素。
str.size(), str.length()	字符串的元素个数。
str.capacity()	无需重新分配的 str 可以拥有的元素数。
str.max_size()	str 可以包含的最大元素数。
std.resize(n)	将 str 调整为 n 个元素。
str.resize_and_overwrite(n, op)	将 str 的大小调整为 n 个元素，并对其元素应用 op 操作。
str.reserve(n)	至少保留 n 个元素的内存。
std.shrink_to_fit()	将字符串的容量调整为其大小。

请求 str.shrink_to_fit() 与 std::vector 的情况一样，非绑定。

大小与容量

```

1 // stringSizeCapacity.cpp
2 ...
3 #include <string>

```

```

4 ...
5
6 void showStringInfo(const std::string& s) {
7     std::cout << s << ":" " ;
8     std::cout << s.size() << " " ;
9     std::cout << s.capacity() << " " ;
10    std::cout << s.max_size() << " " ;
11 }
12
13 std::string str;
14 showStringInfo(str); // "": 0 0 4611686018427387897
15
16 str += "12345";
17 showStringInfo(str); // "12345": 5 5 4611686018427387897
18
19 str.resize(30);
20 showStringInfo(str); // "12345": 30 30 4611686018427387897
21
22 str.reserve(1000);
23 showStringInfo(str); // "12345": 30 1000 4611686018427387897
24
25 str.shrink_to_fit();
26 showStringInfo(str); // "12345": 30 30 4611686018427387897

```

13.4. 比较

字符串支持众所周知的比较操作符 ==、!=、<、>、>=。两个字符串的比较发生在它们的元素上。

字符串的比较

```

1 // stringComparisonAndConcatenation.cpp
2 ...
3 #include <string>
4 ...
5
6 std::string first{"aaa"};
7 std::string second{"aaaa"};
8
9 std::cout << (first < first) << '\n'; // false
10 std::cout << (first <= first) << '\n'; // true
11 std::cout << (first < second) << '\n'; // true

```

13.5. 连接字符串

字符串重载了 + 操作符，可以使用该操作符添加字符串。

+ 操作符仅对 C++ 字符串重载

C++ 类型系统允许将 C++ 和 C 字符串连接到 C++ 字符串，但不允许将 C++ 和 C 字符串连接到 C 字符串。原因是 + 操作符仅对 C++ 字符串重载。只有第二行是有效的 C++ 代码，因为 C 字符串会隐式转换为 C++ 字符串：

连接字符串

```
1 // stringComparisonAndConcatenation.cpp
2 ...
3 #include <string>
4 ...
5 std::string wrong= "1" + "1"; // ERROR
6 std::string right= std::string("1") + "1"; // 11
```

13.6. 访问元素

访问字符串 str 的元素非常方便，字符串支持随机访问迭代器。可以使用 str.front() 访问字符串的第一个字符，并使用 str.back() 访问字符串的最后一个字符。使用 str[n] 和 str.at(n)，可以根据索引获得第 n 个元素。

下表提供了一个概述。

访问字符串的元素

成员函数	示例
str.front()	返回 str 的第一个字符。
str.back()	返回 str 的最后一个字符。
std[n]	返回 str 的第 n 个字符，不检查边界。
std.at(n)	返回 str 的第 n 个字符，检查边界。若超出边界，则抛出 std::out_of_range 异常。

访问元素

```
1 // stringAccess.cpp
2 ...
3 #include <string>
4 ...
5
6 std::string str= {"0123456789"};
7 std::cout << str.front() << '\n'; // 0
8 std::cout << str.back() << '\n'; // 9
9 for (int i= 0; i <= 3; ++i){
```

```

10     std::cout << "str[" << i << "]:" << str[i] << ";" ;
11 } // str[0]: 0; str[1]: 1; str[2]: 2; str[3]: 3;
12
13 std::cout << str[10] << '\n'; // undefined behaviour
14 try{
15     str.at(10);
16 }
17 catch (const std::out_of_range& e){
18     std::cerr << "Exception: " << e.what() << '\n';
19 } // Exception: basic_string::at
20
21 std::cout << *(&str[0]+5) << '\n'; // 5
22 std::cout << *(&str[5]) << '\n'; // 5
23 std::cout << str[5] << '\n'; // 5

```

示例中看到执行 str[10] 是令人惊讶的行为，因为对于字符串边界之外的访问是未定义行为。相反，当调用 str.at(10) 时会报错。

13.7. 输入与输出

字符串可以通过 >> 从输入流中读取，并通过 << 向输出流写入。

全局函数 getline 能够从输入流中逐行读取，直到文件结束字符。

getline 函数有四种重载。前两个参数是输入流 is 和保存行 read 的字符串 line，还可以指定一个特殊的行分隔符。

该函数通过对输入流的引用返回。

```

1 istream& getline (istream& is, string& line, char delim);
2 istream& getline (istream&& is, string& line, char delim);
3 istream& getline (istream& is, string& line);
4 istream& getline (istream&& is, string& line);

```

getline 表示获取整行，包括空格，只有行分隔符会忽略。该函数需要头文件 <string>。

使用字符串输入和输出

```

1 // stringInputOutput.cpp
2 ...
3 #include <string>
4 ...
5
6 std::vector<std::string> readFromFile(const char* fileName) {
7     std::ifstream file(fileName);
8     if (!file) {
9         std::cerr << "Could not open the file " << fileName << ".";
10        exit(EXIT_FAILURE);
11    }

```

```

12     std::vector<std::string> lines;
13     std::string line;
14     while (getline(file, line)) lines.push_back(line);
15     return lines;
16 }
17
18 std::string fileName;
19 std::cout << "Your filename: ";
20 std::cin >> fileName;
21 std::vector<std::string> lines = readFromFile(fileName.c_str());
22 int num{0};
23 for (auto line: lines) std::cout << ++num << ":" << line << '\n';

```

该程序显示任意文件的行，包括行号。表达式 `std::cin >> fileName` 读取文件名。`readFromFile` 函数使用 `getline` 读取所有文件行，并将其压入 `vector` 对象。

13.8. 搜索

C++ 提供了多种形式的字符串搜索功能，每一种都以各种重载的形式存在。

搜索功能叫”find”

奇怪的是，搜索字符串的算法以名称 `find` 开头。若搜索成功，将获得 `std::string::size_type` 类型的索引。若不是，则得到常数 `std::string::npos`。第一个字符的索引为 0。

`find` 算法支持：

- 搜索一个字符，一个 C 字符串，或者一个 C++ 字符串，
- 从 C 或 C++ 字符串中搜索一个字符，
- 前向与后向搜索，
- 从 C 或 C++ 字符串中的字符进行正(包含)或负(不包含)搜索，
- 从字符串中的任意位置开始搜索。

`find` 函数的所有六个重载的参数都遵循类似的模式，第一个参数是要搜索的文本，第二个参数保存搜索的起始位置，第三个参数保存从第二个参数开始的字符数。

以下是六种重载。

查找字符串的重载

成员函数	描述
<code>str.find(...)</code>	返回字符在 str 中为 C 或 C++ 字符串的第一个位置。
<code>str.rfind(...)</code>	返回字符在 str 中为 C 或 C++ 字符串的最后位置。
<code>str.find_first_of(...)</code>	返回 C 或 C++ 字符串中字符的第一个位置。

<code>str.find_last_of(...)</code>	返回 C 或 C++ 字符串中给定字符的最后一个位置。
<code>str.find_first_not_of(...)</code>	返回 C 或 C++ 字符串字符串中非给定字符的第一个位置。
<code>std.find_last_not_of(...)</code>	返回 C 或 C++ 字符串中非给定字符的最后一个位置。

在字符串中查找 (搜索)

```

1 // stringFind.cpp
2 ...
3 #include <string>
4 ...
5 std::string str;
6 auto idx= str.find("no");
7 if (idx == std::string::npos) std::cout << "not found"; // not found
8 str= {"dkeu84kf8k48kdj39kdj74945du942"};
9 std::string str2{"84"};
10 std::cout << str.find('8'); // 4
11 std::cout << str.rfind('8'); // 11
12 std::cout << str.find('8', 10); // 11
13 std::cout << str.find(str2); // 4
14 std::cout << str.rfind(str2); // 4
15 std::cout << str.find(str2, 10); // 18446744073709551615
16 str2="0123456789";
17 std::cout << str.find_first_of("678"); // 4
18 std::cout << str.find_last_of("678"); // 20
19 std::cout << str.find_first_of("678", 10); // 11
20 std::cout << str.find_first_of(str2); // 4
21 std::cout << str.find_last_of(str2); // 29
22 std::cout << str.find_first_of(str2, 10); // 10
23 std::cout << str.find_first_not_of("678"); // 0
24 std::cout << str.find_last_not_of("678"); // 29
25 std::cout << str.find_first_not_of("678", 10); // 10
26 std::cout << str.find_first_not_of(str2); // 0
27 std::cout << str.find_last_not_of(str2); // 26
28 std::cout << str.find_first_not_of(str2, 10); // 12

```

`std::find(str2, 10)` 返回 `std::string::npos`。若显示这个值，在我的机器上会得到 18446744073709551615。

13.9. 检查是否有子字符串

C++20 成员函数 `str.starts_with(prefix)` 和 `str.end_with(suffix)` 检查字符串 `str` 是否以前缀开始或以后缀结束。在 C++23 中，可以使用 `str.contains` 检查字符串是否有子字符串。

检查是否有前缀或后缀

成员函数 `str.starts_with(prefix)` 和 `str.end_with(suffix)` 检查给定字符串 `str` 是否以前缀开始或以后缀结束。子字符串可以是 `std::string_view`、单个字符或字符串。

检查字符串是否以前缀开始或后缀结束

```
1 // startWithEndsWith.cpp
2 ...
3 #include <string>
4 ...
5 std::string helloWorld = "hello world";
6
7 std::cout << helloWorld.starts_with("hello") << '\n'; // true
8 std::cout << helloWorld.starts_with("llo") << '\n'; // false
9
10 std::cout << helloWorld.ends_with("world") << '\n'; // true
11 std::cout << helloWorld.ends_with("wo") << '\n'; // false
```

检查是否包含子字符串

`str.contains` 检查字符串是否有子字符串。子字符串可以是 `std::string_view`、单个字符或字符串。

检查字符串是否包含子字符串

```
1 // containsString.cpp
2 ...
3 #include <string>
4 ...
5 std::string helloWorld = "hello world";
6
7 std::cout << helloWorld.contains("hello") << '\n'; // true
8 std::cout << helloWorld.contains("llo") << '\n'; // true
9 std::cout << helloWorld.contains('w') << '\n'; // true
10 std::cout << helloWorld.contains('W') << '\n'; // false
```

13.10. 修改操作

字符串有很多可以修改的操作。`str.assign` 将一个新字符串赋给字符串 `str`，使用 `str.swap` 可以交换两个字符串。要从字符串中删除一个字符，请使用 `str.pop_back` 或 `str.erase`。相反，`str.clear` 或 `str.erase` 删除整个字符串。要在字符串中添加新字符，请使用 `+=`、`std.append` 或 `str.push_back`，可以使用 `str.insert` 插入新字符或使用 `str.replace` 替换字符。

用于修改字符串的函数

成员函数	描述
str=str2	将 str2 赋值给 str。
str.assign(...)	将一个新字符串赋给 str。
str.swap(str2)	交换 str 和 str2。
str.pop_back()	从 str 中移除最后一个字符。
str.erase(...)	从 str 中删除字符。
str.clear()	清除 str 中的字符。
str.append(...)	向 str 追加字符。
str.push_back(s)	将字符 s 追加到 str。
str.insert(pos, ...)	在 str 中插入从 pos 开始的字符。
str.replace(pos, len, ...)	替换 str 中从 pos 开始的 len 字符

这些操作在许多重载版本中都是可用的，成员函数 str.assign、str.append、str.insert 和 str.replace 非常相似。这四个函数都可以用——字符串和子字符串、字符、C 字符串、C 字符串数组、范围和初始化列表。str.erase 可以擦除单个字符、范围和从给定位置开始的多个字符。

下面的代码片段显示了许多变体。简单起见，只显示字符串修改的效果：

修改字符串

```

1 // stringModification.cpp
2 ...
3 #include <string>
4 ...
5
6 std::string str{"New String"};
7 std::string str2{"Other String"};
8
9 str.assign(str2, 4, std::string::npos); // r String
10 str.assign(5, '-'); // -----
11
12 str= {"0123456789"};
13 str.erase(7, 2); // 01234569
14 str.erase(str.begin()+2, str.end()-2); // 012
15 str.erase(str.begin()+2, str.end()); // 01
16 str.pop_back(); // 0
17 str.erase(); //
18
19 str= "01234";
20 str+= "56"; // 0123456
21 str+= '7'; // 01234567
22 str+= {'8', '9'}; // 0123456789
23 str.append(str); // 01234567890123456789
24 str.append(str, 2, 4); // 012345678901234567892345
25 str.append(3, '0'); // 012345678901234567892345000
26 str.append(str, 10, 10); // 01234567890123456789234500001234567989

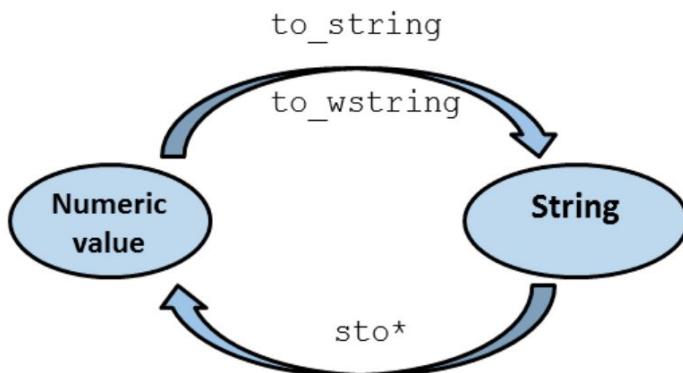
```

```

27 str.push_back('9'); // 012345678901234567892345000012345679899
28
29 str= {"345"};
30 str.insert(3, "6789"); // 3456789
31 str.insert(0, "012"); // 0123456789
32
33 str= {"only for testing purpose."};
34 str.replace(0, 0, "0"); // Only for testing purpose.
35 str.replace(0, 5, "Only", 0, 4); // Only for testing purpose.
36 str.replace(16, 8, ""); // Only for testing.
37
38 str.replace(4, 0, 5, 'y'); // Onlyyyyyy for testing.
39 str.replace(str.begin(), str.end(), "Only for testing purpose.");
40 // Only for testing purpose.
41 str.replace(str.begin()+4, str.end()-8, 10, '#');
42 // Only#####purpose.

```

13.11. 数值转换



可以使用 `std::to_string(val)` 和 `std::to_wstring(val)` 数字或浮点数转换为相应的 `std::string` 或 `std::wstring`。可使用 `sto*` 函数族，用于将数字或浮点数转换为字符串，所有函数都需要头文件 `<string>`。

`sto*` 可看作为 string to

将字符串转换为自然数或浮点数的七种方法遵循一个简单的模式。所有函数都以 `sto` 开头，并添加其他字符，表示字符串应转换为的类型。例如，`stol` 将字符串转换为长整型值，而 `stod` 将字符串转换为双精度浮点型值。

`sto` 函数都有相同的接口。下面的示例显示了 `long` 类型的示例。

```
1 std::stol(str, idx= nullptr, base= 10)
```

该函数接受一个字符串，并确定基基的长表示形式。`stol` 忽略前置空格，并可选择返回 `idx` 中第一个无效字符的索引。默认情况下，基数为 10。基数的有效值为 0 和 2，直到 36。若使用基数 0，编译器将根据字符串的格式自动确定类型。若基数大于 10，编译器将其编码为字符 a 到 z，类似于十六进制数的表示。

该表给出了所有函数的概述。

字符串的数值转换

相应函数	描述
<code>std::to_string(val)</code>	将 <code>val</code> 转换为 <code>std::string</code> 。
<code>std::to_wstring(val)</code>	将 <code>val</code> 转换为 <code>std::wstring</code> 。
<code>std::stoi(str)</code>	返回 <code>int</code> 值。
<code>std::stol(str)</code>	返回一个 <code>long</code> 型值。
<code>std::stoll(str)</code>	返回一个 <code>long long</code> 型值。
<code>std::stoul(str)</code>	返回一个 <code>unsigned long</code> 型值。
<code>std::stoull(str)</code>	返回一个 <code>unsigned long long</code> 型值。
<code>std::stof(str)</code>	返回一个 <code>float</code> 型值。
<code>std::stod(str)</code>	返回一个 <code>double</code> 型值。
<code>std::stold(str)</code>	返回一个 <code>long double</code> 型值。

有 `stou` 函数吗？

若对此很好奇，C++ 的 `sto` 函数是 C 的 `strto*` 函数的简单包装，但是 C 中没有 `strto*` 函数。因此，C++ 没有 `stou` 函数。

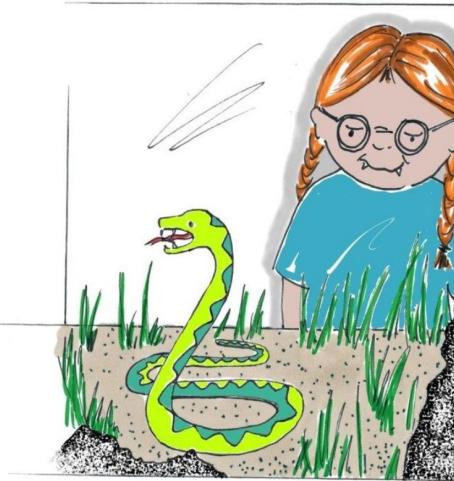
若不能转换，函数会抛出 `std::invalid_argument` 异常。若确定的值对于目标类型来说太大，则会得到 `std::out_of_range` 异常。

数值转换

```
1 // stringNumericConversion.cpp
2 ...
3 #include <string>
4 ...
5
6 std::string maxLongLongString=
7     std::to_string(std::numeric_limits<long long>::max());
8 std::wstring maxLongLongWstring=
9     std::to_wstring(std::numeric_limits<long long>::max());
10
11 std::cout << std::numeric_limits<long long>::max(); // 9223372036854775807
12 std::cout << maxLongLongString; // 9223372036854775807
```

```
13 std::wcout << maxLongLongWstring; // 9223372036854775807
14
15 std::string str("10010101");
16 std::cout << std::stoi(str); // 10010101
17 std::cout << std::stoi(str, nullptr, 16); // 268501249
18 std::cout << std::stoi(str, nullptr, 8); // 2101313
19 std::cout << std::stoi(str, nullptr, 2); // 149
20
21 std::size_t idx;
22 std::cout << std::stod(" 3.5 km", &idx); // 3.5
23 std::cout << idx; // 6
24
25 try{
26     std::cout << std::stoi(" 3.5 km") << '\n'; // 3
27     std::cout << std::stoi(" 3.5 km", nullptr, 2) << '\n';
28 }
29 catch (const std::exception& e){
30     std::cerr << e.what() << '\n';
31 } // stoi
```

第 14 章 字符串视图



Cippi 在观察蛇

`string view`是对字符串的非所属引用，表示一个字符序列的视图。这个字符序列可以是 C++ 字符串或 C 字符串，这需要头文件 `<string_view>`。

字符串视图是对复制优化字符串的优化

`std::string_view` 的目的是避免复制已经由其他人拥有的数据，并允许对类似 `std::string` 的对象进行不可变访问。字符串视图是受限制的字符串，仅支持不可变操作。此外，字符串视图还有两个额外的变异操作: `sv.remove_prefix` 和 `sv.remove_suffix`。

字符串视图是由字符和字符特征参数化的类模板，字符特性有一个默认值。与字符串相反，字符串视图是非所有者，不需要分配器。

```
1 template<
2     class CharT,
3     class Traits = std::char_traits<CharT>
4 > class basic_string_view;
```

根据字符串，字符串视图存在底层字符类型 `char`、`wchar_t`、`char16_t` 和 `char32_t` 的四个同义词。

```
1 typedef std::string_view std::basic_string_view<char>
2 typedef std::wstring_view std::basic_string_view<wchar_t>
3 typedef std::u16string_view std::basic_string_view<char16_t>
4 typedef std::u32string_view std::basic_string_view<char32_t>
```

std::string_view 是字符串视图

若谈论字符串视图，有 99% 的概率引用字符类型 char 的特化 std::basic_string_view，这也适用于本书。

14.1. 创建和初始化

可以创建一个空字符串视图。还可以从现有字符串、字符数组或字符串视图创建字符串视图。下表概述了创建字符串视图的各种方法。

用于创建和设置字符串视图的函数

功能	示例
空字符串视图	std::string_view str_view
来自 C 字符串	std::string_view str_view2("C-string")
来自字符串视图	std::string_view str_view3(str_view2)
来自 C 数组	std::string_view str_view4(arr, sizeof arr)
来自 string_view	str_view4 = str_view3.substring(2, 3)
来自字符串视	std::string_view str_view5 = str_view4

14.2. 无修改的操作

为了使本章简洁，避免重复关于字符串的详细描述，我只提到字符串视图的非修改操作。请使用字符串章节中的相关文档链接了解更多详细信息。

- 元素访问: 操作符 [], at, front, back, data
- 容量: size, length, max_size, empty
- 查找: find, rfind, find_first_of, find_last_of, find_first_not_of, find_last_not_of
- 复制: copy

14.3. 可修改的操作

stringView.swap(stringView2) 交换两个字符串视图的内容。对于字符串视图来说，成员函数 remove_prefix 和 remove_suffix 是唯一的，因为字符串不支持这两种方法。remove_prefix 向前对其起始进行收缩，remove_suffix 向后对其结尾进行收缩。

无修改操作

```
1 // string_view.cpp
2 ...
3 #include <string_view>
4
```

```

5   ...
6
7 using namespace std;
8 string str = " A lot of space";
9 string_view strView = str;
10 strView.remove_prefix(min(strView.find_first_not_of(" "), strView.size()));
11 cout << str << endl // " A lot of space
12     << strView << endl; // "A lot of space"
13
14 char arr[] = {'A', ' ', 'l', 'o', 't', ' ', 'o', 'f', ' ', '
15     's', 'p', 'a', 'c', 'e', '\0', '\0', '\0' };
16 string_view strView2(arr, sizeof arr);
17 auto trimPos = strView2.find('\0');
18 if(trimPos != strView2npos) strView2.remove_suffix(strView2.size() - trimPos);
19 cout << arr << ":" << sizeof arr << endl // A lot of space: 17
20     << strView2 << ":" << strView2.size() << endl; // A lot of space: 14

```

使用字符串视图不分配内存

若创建字符串视图或复制字符串视图，则不需要分配内存。这与字符串形成对比，创建或复制字符串需要分配内存。

分配内存

```

1 // stringView.cpp
2 ...
3 include <string_view>
4 ...
5
6 void* operator new(std::size_t count) {
7     std::cout << " " << count << " bytes" << '\n';
8     return malloc(count);
9 }
10
11 void getString(const std::string&){}
12
13 void getStringView(std::string_view){}
14
15 std::string large = "012345678901234567890"
16     "1234567890123456789"; // 41 bytes allocated
17 std::string substr = large.substr(10); // 31 bytes allocated
18
19 std::string_view largeStringView{large.c_str(), // 0 bytes allocated
20                             large.size()};
21 largeStringView.remove_prefix(10); // 0 bytes allocated
22

```

```
23 getString(large);
24 getString("012345678901234567890"
25     "1234567890123456789"); // 41 bytes allocated
26 const char message []= "0123456789012345678901234567890123456789";
27 getString(message); // 41 bytes allocated
28
29 getStringView(large); // 0 bytes allocated
30 getStringView("012345678901234567890"
31     "1234567890123456789"); // 0 bytes allocated
32 getStringView(message); // 0 bytes allocated
```

有了全局重载操作符 new，就可以观察每个内存分配操作了。

第 15 章 正则表达式



Cippi 正在分析雪地里的脚印

正则表达式是描述文本模式的语言，其需要头文件 `<regex>`。

正则表达式是执行以下任务的强大工具：

- 检查文本是否与文本模式匹配: `std::regex_match`
- 在文本中搜索文本模式: `std::regex_search`
- 将文本模式替换为文本: `std::regex_replace`
- 遍历文本中的所有文本模式: `std::regex_iterator` 和 `std::regex_token_iterator`

C++ 支持六种不同的正则表达式语法。默认情况下，使用 ECMAScript 语法。这是六种语法中最强大的语法，与 Perl 5 中使用的语法非常相似。其他五种语法是基本语法、扩展语法、awk 语法、grep 语法和 egrep 语法。

使用原始字符串

正则表达式中使用原始字符串文字。C++ 文本的正则表达式非常难懂: `C\\+\\\\+`。每个 + 号必须使用两个反斜杠。首先，+ 号是正则表达式中的唯一字符。其次，反斜杠是字符串中的一个特殊字符。一个反斜杠转义 + 号，另一个反斜杠转义反斜杠。通过使用原始字符串字面值，第二个反斜杠不再是必要的，因为反斜杠在字符串中不会转义。

```
1 #include <regex>
2 ...
3 std::string regExpr("C\\+\\\\+");
4 std::string regExprRaw(R"(C\+\+\+)");
```

处理正则表达式通常分三步完成：

I. 定义正则表达式:

```
1 std::string text="C++ or c++.";
2 std::string regExpr(R"(C\+\+\+)");
3 std::regex rex(regExpr);
```

II. 存储搜索结果:

```
1 std::smatch result;
2 std::regex_search(text, result, rgx);
```

III. 处理结果:

```
1 std::cout << result[0] << '\n';
```

15.1. 字符类型

文本类型决定正则表达式的字符类型和搜索结果的类型。

下表显示了四种不同的组合。

文本类型、正则表达式、搜索结果和操作的组合

文本类型	正则表达式类型	结果类型
const char*	std::regex	std::cmatch
std::string	std::regex	std::smatch
const wchar_t*	std::wregex	std::wcmatch
std::wstring	std::wregex	std::wsmatch

本章“搜索”一节中的程序详细展示了这四种组合。

15.2. 正则表达式对象

正则表达式类型的对象是类模板 `template <class charT, class traits= regex_traits <charT>>` 类 `basic_regex` 的实例，由字符类型和特征类参数化，特征类定义了正则语法属性的解释。C++ 中有两种同义类型：

```
1 typedef basic_regex<char> regex;
2 typedef basic_regex<wchar_t> wregex;
```

可以进一步自定义正则表达式类型的对象，可以指定所使用的语法或调整语法。如前所述，C++ 支持基本、扩展、awk、grep 和 egrep 语法。由 `std::regex_constants::icase` 标志限定的正则表达式不区分大小写。若要采用该语法，则必须显式指定语法。

指定语法

```
1 // regexGrammar.cpp
2 ...
3 #include <regex>
4 ...
5 using std::regex_constants::ECMAScript;
6 using std::regex_constants::icase;
```

```

7
8 std::string theQuestion="C++ or c++, that's the question.";
9 std::string regExprStr(R"(c\+\+\+)");
10
11 std::regex rgx(regExprStr);
12 std::smatch smatch;
13
14 if (std::regex_search(theQuestion, smatch, rgx)){
15     std::cout << "case sensitive: " << smatch[0]; // c++
16 }
17
18 std::regex rgxIn(regExprStr, ECMAScript|icase);
19 if (std::regex_search(theQuestion, smatch, rgxIn)){
20     std::cout << "case insensitive: " << smatch[0]; // C++
21 }

```

若使用区分大小写的正则表达式 rgx，则在文本 theQuestion 中搜索的结果是“c++”。若使用不区分大小写的正则表达式 rgxIn，则不是这种情况。现在，得到了匹配的字符串是“C++”。

15.3. 搜索结果 match_results

std::match_results 类型的对象是 std::regex_match 或 std::regex_search 的结果。std::match_results 是一个序列容器，至少有一个 std::sub_match 对象的捕获组，std::sub_match 对象是字符序列。

什么是捕获组？

捕获组允许它进一步分析正则表达式中的搜索结果，由一对括号 () 定义。正则表达式 ((a+)(b+)(c+)) 有四个捕获组：((a+)(b+)(c+))、(a+)、(b+) 和 (c+)，总结果是第 0 个捕获组。

C++ 有四种类型的 std::match_results 类型的同义词：

```

1 typedef match_results<const char*> cmatch;
2 typedef match_results<const wchar_t*> wcmatch;
3 typedef match_results<string::const_iterator> smatch;
4 typedef match_results<wstring::const_iterator> wsmatch;

```

搜索结果 std::smatch smatch 具有强大的接口。

std::smatch 的接口

成员函数	描述
smatch.size()	返回捕获组的数目。
smatch.empty()	若搜索结果有捕获组，则返回。

smatch[i]	返回第 i 个捕获组。
smatch.length(i)	返回第 i 个捕获组的长度。
smatch.position(i)	返回第 i 个捕获组的位置。
smatch.str(i)	以字符串形式返回第 i 个捕获组。
smatch.prefix() and smatch.suffix()	返回捕获组前后的字符串。
smatch.begin() and smatch.end()	返回捕获组的开始和结束迭代器。
smatch.format(...)	为输出格式化 std::smatch 对象。

下面的代码展示了针对不同正则表达式，前四个捕获组的输出。

捕获组

```

1 // captureGroups.cpp
2 ...
3 #include<regex>
4 ...
5 using namespace std;
6
7 void showCaptureGroups(const string& regEx, const string& text) {
8     regex rgx(regEx);
9     smatch smatch;
10    if (regex_search(text, smatch, rgx)) {
11        cout << regEx << text << smatch[0] << " " << smatch[1]
12        << " " << smatch[2] << " " << smatch[3] << endl;
13    }
14 }
15
16 showCaptureGroups("abc+", "abccccc");
17 showCaptureGroups("(a+) (b+) ", "aaabccc");
18 showCaptureGroups("((a+) (b+))", "aaabccc");
19 showCaptureGroups("(ab) (abc)+", "abababc");
20 ...

```

正则表达式	文本	smatch[0]	smatch[1]	smatch[2]	smatch[3]
abc+	abccccc	abccccc			
(a+)(b+)(c+)	aaabccc	aaabccc	aaa	b	ccc
((a+)(b+)(c+))	aaabccc	aaabccc	aaabccc	aaa	b
(ab)(abc)+	abababc	abababc	ab	abc	

std::sub_match

捕获组的类型为 std::sub_match。与 std::match_results 一样，C++ 定义了以下四个同义类型。

```

1 typedef sub_match<const char*> csub_match;
2 typedef sub_match<const wchar_t*> wsub_match;
3 typedef sub_match<string::const_iterator> ssub_match;
4 typedef sub_match<wstring::const_iterator> wssub_match;

```

可以进一步分析捕获组的内容

std::sub_match

成员函数	描述
cap.matched()	指示匹配是否成功。
cap.first() and cap.end()	返回字符序列的开始和结束迭代器。
cap.length()	返回捕获组的长度。
cap.str()	以字符串形式返回捕获组。
cap.compare(other)	将当前捕获组与其他捕获组进行比较。

下面的代码片段显示了搜索结果 std::match_results 和其捕获组 std::sub_match 之间的影响。

std::sub_match

```

1 // subMatch.cpp
2 ...
3 #include <regex>
4 ...
5 using std::cout;
6 std::string privateAddress="192.168.178.21";
7 std::string regEx(R"((\d{1,3})\.(\d{1,3})\.(\d{1,3})\.(\d{1,3}))");
8 std::regex rgx(regEx);
9 std::smatch smatch;
10 if (std::regex_match(privateAddress, smatch, rgx)) {
11     for (auto cap: smatch) {
12         cout << "capture group: " << cap << '\n';
13         if (cap.matched) {
14             std::for_each(cap.first, cap.second, [](int v) {
15                 cout << std::hex << v << " "; });
16             cout << '\n';
17         }
18     }
19 }
20 ...

```

```

capture group: 192.168.178.21
31 39 32 2e 31 36 38 2e 31 37 38 2e 32 31

capture group: 192

```

```
31 39 32

capture group: 168
31 36 38

capture group: 178
31 37 38

capture group: 21
32 31
```

正则表达式 regEx 表示 IPv4 地址， regEx 使用捕获组提取地址的组件。最后，捕获组和 ASCII 字符以十六进制值显示。

15.4. 匹配

`std::regex_match` 确定文本是否与文本模式匹配，可以进一步分析 `std::match_results` 类型的搜索结果。

下面的代码片段展示了 `std::regex_match` 的三个简单应用：一个 C 字符串、一个 C++ 字符串和一个只返回布尔值的范围。这三个类型分别可用于 `std::match_results` 对象。

`std::match`

```
1 // match.cpp
2 ...
3 #include <regex>
4 ...
5
6 std::string numberRegEx(R"([+-]?([0-9]*\.[0-9]+|[0-9]+))");
7 std::regex rgx(numberRegEx);
8 const char* numChar{"2011"};
9
10 if (std::regex_match(numChar, rgx)) {
11     std::cout << numChar << "is a number." << '\n';
12 } // 2011 is a number.
13
14 const std::string numStr{"3.14159265359"};
15 if (std::regex_match(numStr, rgx)) {
16     std::cout << numStr << " is a number." << '\n';
17 } // 3.14159265359 is a number.
18
19 const std::vector<char> numVec{{'-'}, '2', '.', '7', '1', '8', '2',
20                                '8', '1', '8', '2', '8'};
21 if (std::regex_match(numVec.begin(), numVec.end(), rgx)) {
22     for (auto c: numVec){ std::cout << c ;};
23     std::cout << "is a number." << '\n';
24 } // -2.718281828 is a number
```

15.5. 搜索

`std::regex_search` 检查文本是否包含文本模式，可以带 `std::match_results` 对象或不带 `std::match_results` 对象使用该函数，并将其应用于 C 字符串、C++ 字符串或范围。

下面的示例展示了如何对类型为 `const char*`、`std::string`、`const wchar_t*` 和 `std::wstring` 的文本使用 `std::regex_search`。

`std::search`

```
1 // search.cpp
2 ...
3 #include <regex>
4 ...
5
6 // regular expression holder for time
7 std::regex crgx("([01]?[0-9]|2[0-3]):[0-5][0-9]");
8
9 // const char*
10 std::cmatch cmatch;
11
12 const char* ctime{"Now it is 23:10." };
13 if (std::regex_search(ctime, cmatch, crgx)){
14     std::cout << ctime << '\n'; // Now it is 23:10.
15     std::cout << "Time: " << cmatch[0] << '\n'; // Time: 23:10
16 }
17
18 // std::string
19 std::smatch smatch;
20
21 std::string stime{"Now it is 23:25." };
22 if (std::regex_search(stime, smatch, crgx)){
23     std::cout << stime << '\n'; // Now it is 23:25.
24     std::cout << "Time: " << smatch[0] << '\n'; // Time: 23:25
25 }
26
27 // regular expression holder for time
28 std::wregex wrgx(L"([01]?[0-9]|2[0-3]):[0-5][0-9]");
29
30 // const wchar_t*
31 std::wcmatch wcmatch;
32
33 const wchar_t* wctime{L "Now it is 23:47." };
34 if (std::regex_search(wctime, wcmatch, wrgx)){
35     std::wcout << wctime << '\n'; // Now it is 23:47.
36     std::wcout << "Time: " << wcmatch[0] << '\n'; // Time: 23:47
37 }
38
39 // std::wstring
40 std::wsmatch wsmatch;
```

```

41
42 std::wstring wstime{L "Now it is 00:03." };
43 if (std::regex_search(wstime, wsmatch, wrgx)){
44     std::wcout << wstime << '\n'; // Now it is 00:03.
45     std::wcout << "Time: " << wsmatch[0] << '\n'; // Time: 00:03
46 }

```

15.6. 替换

`std::regex_replace` 替换与文本模式匹配的文本中的序列，以简单的形式 `std::regex_replace(text, regex, replString)` 返回其结果为字符串。函数用 `replString` 替换文本中出现的 `regex`。

std::replace

```

1 // replace.cpp
2 ...
3 #include <regex>
4 ...
5 using namespace std;
6
7 string future{"Future"};
8 string unofficialName{"The unofficial name of the new C++ standard is C++0x."};
9
10 regex rgxCpp{R"(C\+\+0x)"};
11 string newCppName{"C++11"};
12 string newName{regex_replace(unofficialName, rgxCpp, newCppName)};
13
14 regex rgxOff{"unofficial"};
15 string makeOfficial{"official"};
16 string officialName{regex_replace(newName, rgxOff, makeOfficial)};
17
18 cout << officialName << endl;
19 // The official name of the new C++ standard is C++11.

```

除了简单的版本之外，C++ 还有一个版本的 `std::regex_replace` 可以处理范围，可将修改后的字符串直接推入另一个字符串：

```

1 typedef basic_regex<char> regex;
2 std::string str2;
3 std::regex_replace(std::back_inserter(str2),
4     text.begin(), text.end(), regex,replString);

```

`std::regex_replace` 的所有版本都有一个额外的可选参数。若将参数设置为 `std::regex_constants::format_no_copy`，将获得与正则表达式匹配的文本部分，只复制匹配的文本。若将参数设置为 `std::regex_constants::format_first_only`，则 `std::regex_replace` 将只应用一次。

15.7. 格式化

`std::regex_replace` 和 `std::match_results`, 格式与捕获组的组合能够格式化文本, 可以使用格式化字符串和占位符来插入值。

以下是两种可能性:

使用正则表达式进行格式化

```
1 // format.cpp
2 ...
3 #include <regex>
4 ...
5
6 std::string future{"Future"};
7 const std::string unofficial{"unofficial, C++0x"};
8 const std::string official{"official, C++11"};
9
10 std::regex regValues{".*, .*"};
11 std::string standardText{"The $1 name of the new C++ standard is $2."};
12 std::string textNow= std::regex_replace(unofficial, regValues, standardText);
13 std::cout << textNow << '\n';
14     // The unofficial name of the new C++ standard is C++0x.
15
16 std::smatch smatch;
17 if (std::regex_match(official, smatch, regValues)){
18     std::cout << smatch.str(); // official,C++11
19     std::string textFuture= smatch.format(standardText);
20     std::cout << textFuture << '\n';
21 } // The official name of the new C++ standard is C++11.
```

函数 `std::regex_replace(unofficial, regValues, standardText)` 中, 从字符串非官方中提取与正则表达式 `regValues` 的第一和第二个捕获组匹配的文本, 将文本 `standardText` 中的占位符 `$1` 和 `$2` 替换为提取的值。`smatch.format(standardTest)` 的策略是类似的, 但有一个区别:

搜索结果 `smatch` 的创建与格式化字符串时的使用是分开的。

除了捕获组之外, C++ 还支持其他格式转义序列。可以在格式字符串中使用:

格式化转义序列

格式化转义序列	描述
<code>\$&</code>	返回总匹配 (第 0 个捕获组)。
<code>\$.</code>	返回 <code>\$</code> 。
<code>\$‘(backward tic)</code>	返回全部匹配之前的文本。
<code>\$‘ (forward tic)</code>	返回全部匹配后的文本。
<code>'\$ i'</code>	返回第 <code>i</code> 个捕获组。

15.8. 反复搜索

使用 std::regex_iterator 和 std::regex_token_iterator 遍历匹配的文本非常方便。std::regex_iterator 支持匹配及其捕获组。std::regex_token_iterator 支持更多。可以处理每个捕获的组件，使用负索引使它能够访问匹配之间的文本。

std::regex_iterator

C++ 为 std::regex_iterator 定义了以下四个同义类型。

```
1 typedef cregex_iterator regex_iterator<const char*>
2 typedef wcregex_iterator regex_iterator<const wchar_t*>
3 typedef sregex_iterator regex_iterator<std::string::const_iterator>
4 typedef wsregex_iterator regex_iterator<std::wstring::const_iterator>
```

可以使用 std::regex_iterator 来计算文本中单词的出现次数：

std::regex_iterator

```
1 // regexIterator.cpp
2 ...
3 #include <regex>
4 ...
5
6 using std::cout;
7 std::string text{"That's a (to me) amazingly frequent question. It may be the most
8   ↪   freque\
9   ntly asked question. Surprisingly, C++11 feels like a new language: The pieces just
10   ↪   fit t\
11   ogether better than they used to, and I find a higher-level style of programming more
12   ↪   nat\
13   ural than before and as efficient as ever." };
14
15 std::regex wordReg{R"(\w+)"};
16 std::sregex_iterator wordItBegin(text.begin(), text.end(), wordReg);
17 const std::sregex_iterator wordItEnd;
18 std::unordered_map<std::string, std::size_t> allWords;
19 for (auto wordIt: allWords) cout << "(" << wordIt.first << ":" 
20           << wordIt.second << ")";
21           // (as:2) (of:1) (level:1) (find:1) (ever:1) (and:2) (natural:1) ...
```

一个单词至少包含一个字符 (\w+)，这个正则表达式用于定义开始迭代器 wordItBegin 和结束迭代器 wordItEnd。匹配项的迭代发生在 for 循环中，每个单词增加计数

器:++allWords[wordItBegin]->str()。若有单词不存在于 allWords 中，则创建一个计数器等于 1 的单词。

std::regex_token_iterator

C++ 为 std::regex_token_iterator 定义了以下四个同义类型。

```
1 typedef cregex_iterator regex_iterator<const char*>
2 typedef wcregex_iterator regex_iterator<const wchar_t*>
3 typedef sregex_iterator regex_iterator<std::string::const_iterator>
4 typedef wsregex_iterator regex_iterator<std::wstring::const_iterator>
```

std::regex_token_iterator 能够使用索引显式地指定感兴趣的捕获组。若不指定索引，将获得所有捕获组，也可以使用它们各自的索引请求特定的捕获组。-1 索引是特殊的：可以使用 -1 来定位匹配之间的文本：

std::regex_token_iterator

```
1 // tokenIterator.cpp
2 ...
3 using namespace std;
4
5 std::string text{"Pete Becker, The C++ Standard Library Extensions, 2006:
6     "Nicolai Josuttis, The C++ Standard Library, 1999:
7     "Andrei Alexandrescu, Modern C++ Design, 2001"};
8
9 regex regBook(R"((\w+)\s(\w+),([\w\s\+]*) ,(\d{4}))");
10 sregex_token_iterator bookItBegin(text.begin(), text.end(), regBook);
11
12 const sregex_token_iterator bookItEnd;
13 while (bookItBegin != bookItEnd){
14     cout << *bookItBegin++ << endl;
15 } // Pete Becker,The C++ Standard Library Extensions,2006
16 // Nicolai Josuttis,The C++ Standard Library,1999
17
18 sregex_token_iterator bookItNameIssueBegin(text.begin(), text.end(),
19                                         regBook, {{2,4}});
20 const sregex_token_iterator bookItNameIssueEnd;
```

第 16 章 输入和输出流



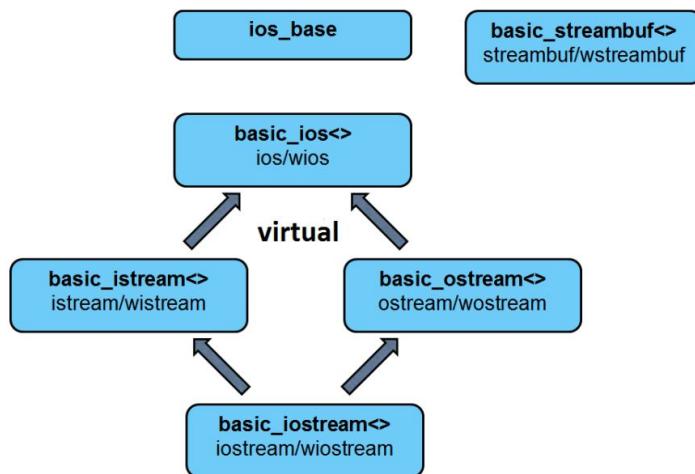
Cippi 在汹涌的河流上划船

[输入输出流](#)能够与外部进行通信。流是一个无限字符流，可以在其上推或拉数据。推叫“写入”，拉叫“读取”。

简单介绍一下输入和输出流

- 早在 1998 年第一个 C++ 标准 (C++98) 出现之前，
- 为可扩展性而设计，
- 根据面向对象和通用范式实现。

16.1. 层次结构



basic_streambuf<>

读取和写入数据。

ios_base

独立于字符类型的所有流类的属性。

basic_ios<>

依赖字符类型的所有流类的属性。

basic_istream<>

用于读取数据的流类的基类。

basic_ostream<>

用于写入数据的流类的基类。

basic_iostream<>

用于读写数据的流类的基类。

类层次结构具有字符类型 `char` 和 `wchar_t` 的同义类型。不以 `w` 开头的名字是 `char` 的同义类型；以 `w` 开头的是 `wchar_t` 的同义类型。

类 `std::basic_iostream<>` 的基类实际上派生自 `std::basic_ios<>`，因此 `std::basic_iostream<>` 只有一个 `std::basic_ios` 实例。

16.2. 输入输出函数

流类 `std::istream` 和 `std::ostream` 通常用于数据的读写。使用 `std::istream` 类需要 `<iostream>` 头文件；使用 `std::ostream` 类需要 `<ostream>` 头文件，可以同时使用 `<iostream>` 头文件。`std::istream` 分别是类 `basic_istream` 和类 `basic_ostream` 的字符类型 `char`, `std::ostream` 的类型定义：

```
1 typedef basic_istream<char> istream;
2 typedef basic_ostream<char> ostream;
```

为了方便处理键盘和显示器，C++ 有四个预定义的流对象。

四个预定义的流对象

流对象	C	设备	是否缓冲
<code>std::cin</code>	<code>stdin</code>	<code>keyboard</code>	<code>yes</code>
<code>std::cout</code>	<code>stdout</code>	<code>console</code>	<code>yes</code>
<code>std::cerr</code>	<code>stderr</code>	<code>console</code>	<code>no</code>
<code>std::clog</code>		<code>monitor</code>	<code>yes</code>

流对象也可用于 `wchar_t`

到目前为止，`wchar_t`、`std::wcin`、`std::wcout`、`std::wcerr` 和 `std::wclog` 的四个流对象并没有像它们的字符那样大量使用。这里，只是稍微提一下。

流对象足以编写从命令行读取并返回总和的程序。

流对象

```

1 // IOSTreams.cpp
2 ...
3 #include <iostream>
4
5 int main() {
6     std::cout << "Type in your numbers";
7     std::cout << "(Quit with an arbitrary character): " << std::endl;
8         // 2000 <Enter> 11 <a>
9     int sum{0};
10    int val;
11    while (std::cin >> val) sum += val;
12    std::cout << "Sum: " << sum; // Sum: 2011
13 }

```

上面的小程序使用了流操作符 `<<` 和 `>>`，以及流操纵符 `std::endl`。

插入操作符 `<<` 将字符压入输出流 `std::cout`; 提取操作符 `>>` 从输入流 `std::cin` 中提取字符。可以构建插入或提取操作符链，这两个操作符都返回对自己的引用。

`std::endl` 是一个流操纵符，因为它将一个'\n'字符放到 `std::cout` 上，并刷新输出缓冲区。

下面是最常用的流操纵符。

最常用的流操纵符

操纵符	流类型	描述
<code>std::endl</code>	output	插入新行字符并刷新流。
<code>std::flush</code>	output	刷新流。
<code>std::ws</code>	input	丢弃前置空格。

输入

C++ 中，从输入流中读取数据有两种方式：用提取器 `>>` 格式化和用显式成员函数不格式化。

格式化输入

提取操作符 `>>`

- 为所有内置类型和字符串预定义，
- 可为用户定义的数据类型进行实现，
- 可以通过格式说明符进行配置。

默认情况下，`std::cin` 会忽略前置空格

```
1 #include <iostream>
2 ...
3 int a, b;
4 std::cout << "Two natural numbers: " << '\n';
5 std::cin >> a >> b; // < 2000 11>
6 std::cout << "a: " << a << " b: " << b;
```

非格式化输入

对于来自输入流的未格式化输入，有许多成员函数。

来自输入流的未格式化输入

成员函数	描述
<code>is.get(ch)</code>	将一个字符读入 <code>ch</code> 。
<code>is.get(buf, num)</code>	在缓冲区中最多读取 <code>num</code> 个字符。
<code>is.getline(buf, num[, delim])</code>	在缓冲区中最多读取 <code>num</code> 个字符。 可选地使用行分隔符 <code>delim</code> (默认 <code>\n</code>)。
<code>is.gcount()</code>	返回通过未格式化操作从数组中提取的最后字符数。
<code>is.ignore(streamsize sz=1, int delim=end-of-file)</code>	忽略 <code>sz</code> 字符直到 <code>delim</code> 。
<code>is.peek()</code>	从 <code>is</code> 中获取一个字符，但不使用它。
<code>is.unget()</code>	将最后一个读取的字符回推到 <code>is</code> 。
<code>is.putback(ch)</code>	将字符 <code>ch</code> 引入流 <code>is</code> 。

`std::string` 有一个 `getline` 函数

`std::string` 的 `getline` 函数比 `istream` 的 `getline` 函数有一个很大的优势，字符串会自动处理它的内存，但必须为 `is.get(buf, num)` 所使用的缓冲区保留内存。

```
1 // inputUnformatted.cpp
2 ...
3 #include <iostream>
4 ...
5 std::string line;
6 std::cout << "Write a line: " << '\n';
7
8 std::getline(std::cin, line); // <Only for testing purpose.>
9 std::cout << line << '\n'; // Only for testing purpose.
```

```
10
11 std::cout << "Write numbers, separated by;" << '\n';
12 while (std::getline(std::cin, line, ';') ) {
13     std::cout << line << " ";
14 } // <2000;11;a>
15 // 2000 11
```

输出

可以使用插入操作符 `<<` 将字符推入输出流。

插入操作符 `<<`

- 为所有内置类型和字符串预定义，
- 可以为用户定义的数据类型实现，
- 可以通过格式说明符进行调整。

格式说明符

格式说明符能够显式地调整输入和输出数据。

操纵符作为格式说明符

格式说明符可用作操纵符和标志。本书中只介绍操纵符，因为它们的功能非常相似，而且操纵符使用起来更顺手。

操纵符作为格式说明符

```
// formatSpecifier.cpp
...
#include <iostream>
...
int num{2011};

std::cout.setf(std::ios::hex, std::ios::basefield);
std::cout << num << '\n'; // 7db
std::cout.setf(std::ios::dec, std::ios::basefield);
std::cout << num << '\n'; // 2011

std::cout << std::hex << num << '\n'; // 7db
std::cout << std::dec << num << '\n'; // 2011
```

下表给出了重要的格式说明符。除了字段宽度之外，格式说明符是固定的，在每个程序结束后重置。

没有实参的操作符需要头文件 <iostream>, 带参数的操作符需要头文件 <iomanip>。

布尔值的显示

操纵符	流类型	描述
std::boolalpha	输入和输出	将布尔值显示为字符。
std::noboolalpha	输入和输出	将布尔值显示为数字(默认)。

设置字段宽度和填充字符

操纵符	流类型	描述
std::setw(val)	输入和输出	将字段宽度设置为 val。
std::setfill(c)	输出流	将填充字符设置为 c 字符(默认为空格)。

文本对齐

操纵符	流类型	描述
std::left	输出	向左对齐输出。
std::right	输出	向右对齐输出。
std::internal	输出	左对齐数字符号, 右对齐数值符号。

正号和大写或小写

操纵符	流类型	描述
std::showpos	输出	显示正号。
std::noshowpos	输出	不显示正号(默认)。
std::uppercase	输出	对数值使用大写字符(默认值)。
std::lowercase	输出	对数值使用小写字符。

显示数字基数

操纵符	流类型	描述
std::oct	输出和输出	使用八进制格式的自然数。
std::dec	输出和输出	使用十进制格式的自然数(默认)。
std::hex	输出和输出	使用十六进制格式的自然数。
std::showbase	输出	显示数字基数。
std::noshowbase	输出	不显示数字基数(默认)。

对于浮点数有一些特殊的规则:

- 默认情况下，有效数字(小数点后面的数字)的个数为6。
- 若有效位数不够大，则以科学记数法显示。
- 不显示前导零和尾随零。
- 若可能，则不显示小数点。

浮点数

操纵符	流类型	描述
std::setprecision(val)	输出	将输出精度调整为val。
std::showpoint	输出	显示小数点。
std::noshowpoint	输出	不显示小数点(默认)。
std::fixed	输出	以十进制格式显示浮点数。
std::scientific	输出	以科学计数法显示浮点数。
std::hexfloat	输出	以十六进制格式显示浮点数。
std::defaultfloat	输出	以默认浮点表示法显示浮点数。

格式说明符

```

1 // formatSpecifierOutput.cpp
2 ...
3 #include <iomanip>
4 #include <iostream>
5 ...
6
7 std::cout.fill('#');
8 std::cout << -12345;
9 std::cout << std::setw(10) << -12345; // #####-12345
10 std::cout << std::setw(10) << std::left << -12345; // -12345#####
11 std::cout << std::setw(10) << std::right << -12345; // #####-12345
12 std::cout << std::setw(10) << std::internal << -12345; //---#####12345
13
14 std::cout << std::oct << 2011; // 3733
15 std::cout << std::hex << 2011; // 7db
16
17 std::cout << std::showbase;
18 std::cout << std::dec << 2011; // 2011
19 std::cout << std::oct << 2011; // 03733
20 std::cout << std::hex << 2011; // 0x7db
21
22 std::cout << 123.456789; // 123.457
23 std::cout << std::fixed;
24 std::cout << std::setprecision(3) << 123.456789; // 123.457
25 std::cout << std::setprecision(6) << 123.456789; // 123.456789
26 std::cout << std::setprecision(9) << 123.456789; // 123.456789000
27

```

```

28 std::cout << std::scientific;
29 std::cout << std::setprecision(3) << 123.456789; // 1.235e+02
30 std::cout << std::setprecision(6) << 123.456789; // 1.234568e+02
31 std::cout << std::setprecision(9) << 123.456789; // 1.234567890e+02
32
33 std::cout << std::hexfloat;
34 std::cout << std::setprecision(3) << 123.456789; // 0x1.edd3c07ee0b0bp+6
35 std::cout << std::setprecision(6) << 123.456789; // 0x1.edd3c07ee0b0bp+6
36 std::cout << std::setprecision(9) << 123.456789; // 0x1.edd3c07ee0b0bp+6
37
38 std::cout << std::defaultfloat;
39 std::cout << std::setprecision(3) << 123.456789; // 123
40 std::cout << std::setprecision(6) << 123.456789; // 123.457
41 std::cout << std::setprecision(9) << 123.456789; // 123.456789
42

```

16.3. 流

流是一个无限的数据流，可以在其上推或拉数据。字符串流和文件流允许字符串和文件直接与流交互。

字符串流

字符串流需要头文件 `<sstream>`。不连接到输入或输出流，并将其数据存储在字符串中。

无论是使用字符串流作为输入或输出，还是使用字符类型 `char` 或 `wchar_t`，都有各种字符串流类：

`std::istringstream` 和 `std::wistringstream`

用于 `char` 和 `wchar_t` 类型输入数据的字符串流。

`std::ostringstream` 和 `std::wostringstream`

用于 `char` 和 `wchar_t` 类型输出数据的字符串流。

`std::stringstream` 和 `std::wstringstream`

用于 `char` 和 `wchar_t` 类型的输入或输出数据的字符串流。

对字符串流的操作有：

- 字符串流中写入数据：

```

1 std::stringstream os;
2 os << "New String";
3 os.str("Another new String");

```

- 从字符串流中读取数据：

```

1 std::stringstream os;
2 std::string str;

```

```
3 | os >> str;
4 | str= os.str();
```

- 清除字符串流:

```
1 std::stringstream os;
2 os.str("");
```

字符串流通常用于字符串和数值之间的类型安全转换:

字符串流

```
1 // stringStreams.cpp
2 ...
3 #include <sstream>
4 ...
5
6 template <typename T>
7 T StringTo ( const std::string& source ){
8     std::istringstream iss(source);
9     T ret;
10    iss >> ret;
11    return ret;
12 }
13
14 template <typename T>
15 std::string ToString(const T& n){
16     std::ostringstream tmp ;
17     tmp << n;
18     return tmp.str();
19 }
20
21 std::cout << "5= " << StringTo<int>("5"); // 5
22 std::cout << "5 + 6= " << StringTo<int>("5") + 6; // 11
23 std::cout << ToString(StringTo<int>("5") + 6); // "11"
24 std::cout << "5e10: " << std::fixed << StringTo<double>("5e10"); // 50000000000
```

文件流

文件流能够处理文件。文件流自动管理其文件的生命周期，需要头文件 `<fstream>`。

无论使用文件流作为输入或输出，还是使用字符类型 `char` 或 `wchar_t`，都有各种文件流类：

`std::ifstream` 和 `std::wifstream`

用于 `char` 和 `wchar_t` 类型输入数据的文件流。

`std::ofstream` 和 `std::wofstream`

用于 `char` 和 `wchar_t` 类型输出数据的文件流。

std::fstream 和 std::wfstream

用于 char 和 wchar_t 类型的输入和输出数据的文件流。

std::filebuf 和 std::wfilebuf

类型为 char 和 wchar_t 的数据内存。

设置文件位置指针

用于读写的文件流必须在竞赛改变后设置文件位置指针。

标志能够设置文件流的打开模式。

打开文件流的标志

标志	描述
std::ios::in	打开文件流进行读取 (默认为 std::ifstream 和 std::wifstream)。
std::ios::out	打开文件流进行写入 (默认为 std::ofstream 和 std::wofstream)。
std::ios::app	将字符追加到文件流的末尾。
std::ios::ate	设置文件流末尾的文件位置指针的初始位置。
std::ios::trunc	删除原始文件。
std::ios::binary	禁止文件流中转义序列的解释。

将名为 in 的文件，复制到以文件缓冲区 in.rdbuf() 命名的文件中非常容易。在这个简短的示例中需要包含错误处理。

```

1 #include <fstream>
2 ...
3 std::ifstream in("inFile.txt");
4 std::ofstream out("outFile.txt");
5 out << in.rdbuf();

```

下面的表中，我比较了 C++ 和 C 打开文件的模式。

用 C++ 和 C 打开一个文件

C++ 模式	描述	C 模式
std::ios::in	读取文件	"r"
std::ios::out	写入文件	"w"
std::ios::out std::ios::app	追加到文件。	"a"
std::ios::in std::ios::out	读取并写入文件。	"r+"

std::ios::in std::ios::out std::ios::trunc	写入和读取文件。	”w+”
--	----------	------

文件必须以模式“r”和“r+”存在。相反，文件是用“a”和“w+”创建的，该文件将被“w”覆盖。

可以显式地管理文件流的生命周期。

管理文件流的生命周期

函数	描述
infile.open(name)	打开文件 name 以供读取。
infile.open(name, flags)	打开带有 flags 的文件 name 以供读取。
infile.close()	关闭文件。
infile.is_open()	检查文件是否已打开。

随机访问

随机访问可以设置文件的位置指针。

构造文件流时，文件的位置指针指向文件的开头。通过文件流文件的成员函数，可以调整文件流文件的位置。

导航文件流

成员函数	描述
file.tellg()	返回文件的读取位置。
file.tellp()	返回文件的写入位置。
file.seekg(pos)	设置文件的读取位置为 pos。
file.seekp(pos)	设置文件的写位置为 pos。
file.seekg(off, rpos)	将文件的读位置设置为相对于 rpos 的偏移量。
file.seekp(off, rpos)	将文件的写位置设置为相对于 rpos 的偏移量。

off 必须是一个数字。rpos 可以是以下三个值：

std::ios::beg

文件开头的位置。

std::ios::cur

当前位置。

std::ios::end

位于文件末尾的位置。

尊重文件边界

若随机访问一个文件，C++ 运行时不会检查文件边界。读取或写入边界之外的数据是未定义行为。

随机访问

```
1 // randomAccess.cpp
2 ...
3 #include <fstream>
4 ...
5
6 void writeFile(const std::string name){
7     std::ofstream outFile(name);
8     if (!outFile){
9         std::cerr << "Could not open file " << name << '\n';
10    exit(1);
11 }
12 for (unsigned int i= 0; i < 10 ; ++i){
13     outFile << i << " 0123456789" << '\n';
14 }
15 }
16 std::string random{"random.txt"};
17 writeFile(random);
18 std::ifstream inFile(random);
19
20 if (!inFile){
21     std::cerr << "Could not open file " << random << '\n';
22     exit(1);
23 }
24
25 std::string line;
26
27 std::cout << inFile.rdbuf();
28 // 0 0123456789
29 // 1 0123456789
30 ...
31 // 9 0123456789
32
33 std::cout << inFile.tellg() << '\n'; // 200
34
35 inFile.seekg(0); // inFile.seekg(0, std::ios::beg);
36 getline(inFile, line);
37 std::cout << line; // 0 0123456789
38
39 inFile.seekg(20, std::ios::cur);
40 getline(inFile, line);
41 std::cout << line; // 2 0123456789
```

```
42
43 inFile.seekg(-20, std::ios::end);
44 getline(inFile, line);
45 std::cout << line; // 9 0123456789
```

流的状态

标志表示流的状态，处理这些标志的成员函数需要头文件 `<iostream>`。

流的状态

下面是导致流的不同状态的条件示例：

`std::ios::eofbit`

- 读取超出最后一个有效字符的内容。

`std::ios::failbit`

- 错误的格式化读取。
- 读取超出最后一个有效字符的内容。
- 打开文件出错。

`std::ios::badbit`

- 无法调整流缓冲区的大小。
- 流缓冲区的代码转换出错。
- 部分流抛出了异常。

`stream.fail()` 返回是否设置了 `std::ios::failbit` 或 `std::ios::badbit`。

流的状态可以读取和设置。

stream.clear()

初始化标志并将流设置为 `goodbit` 状态。

stream.clear(sta)

初始化标志并将流设置为 `sta` 状态。

stream.rdstate()

返回当前状态。

stream.setstate(fla)

设置附加标志 `fla`。

流上的操作只有在流处于好位状态时才有效。若流处于 `badbit` 状态，则不能将其设置回 `goodbit` 状态。

流的状态

```

1 // streamState.cpp
2 ...
3 #include <iostream>
4 ...
5 std::cout << std::cin.fail() << '\n'; // false
6
7 int myInt;
8 while (std::cin >> myInt){ // <a>
9     std::cout << myInt << '\n'; //
10    std::cout << std::cin.fail() << '\n'; //
11 }
12
13 std::cin.clear();
14 std::cout << std::cin.fail() << '\n'; // false

```

字符 a 的输入导致流 std::cin 处于 std::ios::failbit 状态。因此不能显示 a 和 std::cin.fail(), 必须首先初始化流 std::cin。

16.4. 自定义数据类型

若重载输入和输出操作符，则数据类型的行为就像内置数据类型一样。

```

1 friend std::istream& operator>> (std::istream& in, Fraction& frac);
2 friend std::ostream& operator<< (std::ostream& out, const Fraction& frac);

```

对于重载输入和输出操作符，必须牢记一些规则：

- 为了支持输入和输出操作的链接，必须通过非常量引用获取和返回输入和输出流。
- 要访问类的私有成员，输入和输出操作符必须是数据类型的友元。
- 输入操作符 >> 将其数据类型作为非常量引用。
- 输出操作符 << 将其数据类型作为常量引用。

重载输入和输出操作符

```

1 // overloadingInOutput.cpp
2 class Fraction{
3 public:
4     Fraction(int num= 0, int denom= 0):numerator(num), denominator(denom) {}
5     friend std::istream& operator>> (std::istream& in, Fraction& frac);
6     friend std::ostream& operator<< (std::ostream& out, const Fraction& frac);
7 private:
8     int numerator;
9     int denominator;
10 };
11
12 std::istream& operator>> (std::istream& in, Fraction& frac) {

```

```
13     in >> frac.numerator;
14     in >> frac.denominator;
15     return in;
16 }
17
18 std::ostream& operator<< (std::ostream& out, const Fraction& frac) {
19     out << frac.numerator << "/" << frac.denominator;
20     return out;
21 }
22
23 Fraction frac(3, 4);
24 std::cout << frac; // 3/4
25
26 std::cout << "Enter two numbers: ";
27 Fraction fracDef;
28
29 std::cin >> fracDef; // <1 2>
30 std::cout << fracDef; // 1/2
```

第 17 章 格式库



Cippi 捏了一个杯子

C++20 中的[格式库](#)为 printf 系列提供了一种安全且可扩展的替代方法，并扩展了 Input 和 Output 流。该库需要 <format>，格式字符串语法遵循 Python 语法。

有了格式化库，就可以使用的格式字符串来指定格式规范

- 文本的填充字符和对齐方式。
- 符号，宽度和数字的精度。
- 数据类型。

17.1. 格式化功能

C++ 支持各种格式化函数。基本编译时格式化函数 std::format、std::format_to 和 std::format_to_n。运行时格式化函数 std::vformat 和 std::vformat_to 与函数 std::make_format_args 结合使用。最后是，std::print 和 std::println。

格式化库有三个基本的格式化功能。

std::format

返回格式化字符串。

std::format_to

通过输出迭代器写入格式化字符串

std::format_to_n

通过输出迭代器写入格式化字符串，但不超过 n 个字符

```
1 #include <format>
2 ...
3 std::format("{1} {0}!", "world", "Hello"); // Hello world!
4
5 std::string buffer;
6 std::format_to(std::back_inserter(buffer),
7   "Hello, C++{}!\n", "20"); // Hello, C++20!
```

std::vformat, std::vformat_to 和 std::make_format_args

三个格式化函数 `std::format`、`std::format_to` 和 `std::format_to_n` 使用格式字符串创建格式化字符串。此格式字符串必须是编译时值，所以无效的格式字符串会导致编译时错误。

对于运行时格式字符串，有两个可选函数 `std::vformat` 和 `std::vformat_to`，必须与 `std::make_format_args` 结合使用。

```
1 #include <format>
2 ...
3 std::string formatString = "{1} {0}!";
4 std::vformat(formatString, std::make_format_args("world", "Hello")); // Hello world
```

std::print 和 std::println

函数 `std::print` 和 `std::println` 写入输出控制台。`println` 在输出中添加一个换行符。此外，这两个函数都使它能够写入输出文件流并支持[Unicode](#)，必须包含头文件 `<print>`。

```
1 #include <print>
2 ...
3 std::print("{1} {0}!", "world", "Hello"); // prints "Hello world!"
4
5 std::ofstream outFile("testfile.txt");
6 std::print(outFile, "{1} {0}!", "world", "Hello"); // writes "Hello world!" into the
7   ↪ out\
File
```

本章剩下的部分使用 `std::format`

编译时格式化函数 `std::format`、`std::format_to` 和 `std::format_to_n`，运行时格式化函数 `std::vformat` 和 `std::vformat_n`，以及函数 `std::print` 和 `std::println` 对格式字符串应用相同的语法。为简单起见，本章剩下的部分将使用 `std::format`。

17.2. 语法

`std::format` 的用法: `std::format(FormatString, Args)`

格式字符串 `FormatString` 由以下几个部分组成

- 普通字符，(除了 { 和 })
- 转义序列，{{ 和 }} 取而代之的是 { 和 }
- 替换字段

替换字段具有以下格式

- 起始字符 {
 - argument-id(可选)
 - Colon : 格式规范 (可选)
- 关闭字符 }

argument-id 允许指定 Args 中参数的索引，id 从 0 开始。当不提供参数 id 时，将按照给定的方式使用参数。所有替换字段要么必须使用参数 id，要么不使用。

std::formatter 及其特化定义了参数的格式规范。

- 基本类型和字符串类型: 基于[Python 的格式规范](#)的标准格式规范。
- 时间类型: [Chrono 格式规范](#)
- 其他类型: 用户自定义格式规范

17.3. 格式规范

可以指定文本的填充字符和对齐方式、符号、宽度、数字精度和数据类型。

填充字符和文本对齐

- 填充字符: 默认空格
- 对齐方式:
 - <: 左
 - >: 右
 - ^: 居中

```
1 std::format(":7", 42);      // "        42"
2 std::format(":7", 'x');     // "x      "
3 std::format(":*<7", 'x');  // "*****x"
4 std::format(":*>7", 'x');  // "*****x"
5 std::format(":*^7", 'x');   // "***x***"
6 std::format(":7d", c);     // "      120"
7 std::format(":7", true);   // "true    "
```

数字的符号、宽度和精度

- 数字符号
 - +: 数字得到一个正符号
 - -: 数字得到一个负符号 (默认)
 - space: 正数有一个空格

```
1 double inf = std::numeric_limits<double>::infinity();
2 double nan = std::numeric_limits<double>::quiet_NaN();
```

```

3   std::format("{0:}, {0:+}, {0:-}, {0: }", 1); // "1,+1,1, 1"
4   std::format("{0:}, {0:+}, {0:-}, {0: }", -1); // "-1,-1,-1,-1"
5   std::format("{0:}, {0:+}, {0:-}, {0: }", inf); // "inf,+inf,inf, inf"
6   std::format("{0:}, {0:+}, {0:-}, {0: }", nan); // "nan,+nan,nan, nan"
7

```

- #:
 - 使用替代格式
 - 整型:
 - * 二进制数: 0b
 - * 八进制数: 0
 - * 十六进制数: 0x
- 0: 前置使用零填充

```

1 std::format("{:+06d}", 120); // "+00120"
2 std::format("{:#0f}", 120)); // "120.000000"
3 std::format("{:0>15f}", 120)); // "00000120.000000"
4 std::format("{:#06x}", 0xa); // "0x000a"

```

- Width: 指定最小宽度。
- Precision: 可以应用于浮点数和字符串
 - 浮点数: 小数点后的位数
 - 字符串: 最大字符数

```

1 double d = 20.11;
2
3 std::format("{} ", d)); // " 20.11 "
4 std::format(":10", d)); // " 20.11 "
5 std::format(":10.3", d)); // " 20.1 "
6 std::format(":.3", "Only a test."); // " Onl "

```

数据类型

若没有其他指定，则将值复制到输出中。可以显式地指定一个值的表示:

- 字符串的表示: s
- 整数的表示:
 - b, B: 二进制
 - d: 十进制
 - o: 八进制
 - x, X: 十六进制
- 字符的表示:
 - b, B, d, o, x, X: 整型
- Bool:
 - s: true 或 false

- b, B, d, o, x, X: 整型
- 浮点:
 - e, E: 科学计数法
 - f, F: 小数方式

17.4. 自定义格式化器

要格式化用户定义类型，我必须为用户定义类型专门化`std::formatter`类。所以，必须实现成员函数`parse`和`format`。

- `parse`:
 - 接受解析上下文。
 - 解析解析后的上下文
 - 返回一个迭代器到格式规范的末尾
 - 出现错误时抛出`std::format_error`
- `format`:
 - 获取应进行格式化的值`t`和格式上下文`fc`
 - 根据格式上下文进行格式化
 - 将输出写入`fc.out()`
 - 返回一个表示输出结束的迭代器

把理论付诸实践，格式化一个`std::vector`。

格式化一个`std::vector`

类`std::formatter`的特化尽可能简单，指定了应用于容器的每个元素的格式规范。

```

1 // formatVector.cpp
2
3 #include <format>
4
5 ...
6
7 template <typename T>
8 struct std::formatter<std::vector<T>> {
9
10     std::string formatString;
11
12     auto constexpr parse(format_parse_context& ctx) { // (3)
13         formatString = "{:";
14         std::string parseContext(std::begin(ctx), std::end(ctx));
15         formatString += parseContext;
16         return std::end(ctx) - 1;
17     }
18
19     template <typename FormatContext>
20     auto format(const std::vector<T>& v, FormatContext& ctx) {

```

```

20     auto out= ctx.out();
21     std::format_to(out, "[" );
22     if (v.size() > 0) std::format_to(out, formatString, v[0]);
23     for (int i= 1; i < v.size(); ++i) {
24         std::format_to(out, ", " + formatString, v[i]); // (1)
25     }
26     std::format_to(out, "]"); // (2)
27     return std::format_to(out, "\n" );
28 }
29 };
30 ...
31 ...
32 std::vector<int> myInts{1, 2, 3, 4, 5};
33 std::cout << std::format("{:}", myInts); // [1, 2, 3, 4, 5] // (4)
34 std::cout << std::format("{:+}", myInts); // [+1, +2, +3, +4, +5]
35 std::cout << std::format("{:03d}", myInts); // [001, 002, 003, 004, 005]
36 std::cout << std::format("{:b}", myInts); // [1, 10, 11, 100, 101] // (5)
37
38 std::vector<std::string> myStrings{"Only", "for", "testing"};
39 std::cout << std::format("{:}", myStrings); // [Only, for, testing]
40 std::cout << std::format("{:.3}", myStrings); // [Onl, for, tes]

```

`std::vector` 的特化具有成员函数 `parse` 和 `format`。`parse` 实际上创建了一个 `formatString`，应用于 `std::vector(1 和 2)` 的每个元素。解析上下文 `ctx(3)` 包含冒号 `(:)` 和右花括号 `(})` 之间的字符，该函数返回一个指向右花括号 `(})` 的迭代器。成员函数格式化的工作更有趣，`format` 上下文返回输出迭代器。使用了输出迭代器和 `std::format_to` 函数，所以 `std::vector` 的元素可以很好地显示出来。

`std::vector` 的元素有几种格式化方式。第一行 (4) 显示数字。下面一行在每个数字前写一个符号，将它们对齐为 3 个字符，并使用 0 作为填充字符，第 (5) 行以二进制格式显示。剩下的两行输出 `std::vector` 的每个字符串，最后一行将每个字符串截断为三个字符。

第 18 章 文件系统



Cippi 在整理绘画的东西

[filesystem](#)是基于[boost::filesystem](#)的，一些组件是可选的。所以，并非在文件系统库的每个实现上都可以使用 std::filesystem 的所有功能。例如，FAT-32 上就不支持符号链接。

该库基于三个概念：文件、文件名和路径。

- 文件是一个对象，它包含可以写入或读取的数据。文件有名称和文件类型。文件类型包括目录、硬链接、符号链接和普通文件。
 - 目录是存放其他文件的容器。当前目录由点“.”表示；两个点表示父目录“..”。
 - 硬链接将名称与现有文件关联起来。
 - 符号链接将名称与可能存在的路径关联起来。
 - 常规文件是一个目录条目，既不是目录，也不是硬链接，也不是符号链接。
- 文件名是一个表示文件的字符串，由实现定义的，允许使用哪些字符，名称可以有多长，或者名称是否区分大小写。
- 路径是标识文件位置的条目序列，有一个可选的根名称，如 Windows 上的“C:”，后面跟着一个根目录，如 Unix 上的“/”。附加部分可以是目录、硬链接、符号链接或常规文件。路径可以是绝对的、规范的或相对的。
 - 绝对路径是用来标识文件的路径。
 - 规范路径是既不包含符号链接，也不包含相对路径的路径。“.”(当前目录) 或“..”(父目录)。
 - 相对路径指定相对于文件系统中某个位置的路径。路径，如“.”(当前目录)，“..”(父目录) 或“home/rainer”是相对路径。在 Unix 上，不是从根目录“/”开始的。

下面是文件系统的一个介绍性示例。

文件系统库的概述

```
1 // filesystem.cpp
2 ...
3 #include <filesystem>
4 ...
5 ...
6 ...
```

```

7  namespace fs = std::filesystem;
8
9  std::cout << "Current path: " << fs::current_path() << '\n'; // (1)
10
11 std::string dir= "sandbox/a/b";
12 fs::create_directories(dir); // (2)
13
14 std::ofstream("sandbox/file1.txt");
15 fs::path symPath= fs::current_path() /= "sandbox"; // (3)
16 symPath /= "syma";
17 fs::create_symlink("a", "symPath"); // (4)
18
19 std::cout << "fs::is_directory(dir): " << fs::is_directory(dir) << '\n';
20 std::cout << "fs::exists(symPath): " << fs::exists(symPath) << '\n';
21 std::cout << "fs::symlink(symPath): " << fs::is_symlink(symPath) << '\n';
22
23 for(auto& p: fs::recursive_directory_iterator("sandbox")){ // (5)
24     std::cout << p << '\n';
25 }
26 fs::remove_all("sandbox");

```

`ss::current_path()`(1) 返回当前路径，可以使用 `std::filesystem::create_directories` 创建目录层次结构(2)。`/=` 运算符对于路径(3)是重载的，可以直接创建符号链接(4)或检查文件的属性。调用 `recursive_directory_iterator`(5) 允许递归遍历目录。

Output:

```

Current path: "/tmp/1469540273.75652"

fs::is_directory(dir): true
fs::exists(symPath): true
fs::symlink(symPath): true

"sandbox/syma"
"sandbox/file1.txt"
"sandbox/a"
"sandbox/a/b"
"sandbox/a/b/c"

```

18.1. 类

许多类封装了文件系统的一个特定功能。

文件系统中的各种类

类	描述
path	表示路径。
filesystem_error	定义异常对象。
directory_entry	表示目录项。
directory_iterator	定义目录迭代器。
recursive_directory_iterator	定义递归目录迭代器。
file_status	存储有关文件的信息。
space_info	表示文件系统信息。
file_type	文件类型。
perms	表示文件访问权限。
perm_options	代表权限选项功能。
copy_options	表示 copy 和 copy_file 函数的选项。
directory_options	表示函数 directory_iterator 和 recursive_directory_iterator 的选项。
file_time_type	表示文件时间。

操纵文件的权限

文件的权限由类 std::filesystem::perms 表示，是一个位掩码类型，因此可以通过位操作来操作，访问权限基于POSIX。

来自en.cppreference.com的程序展示了如何读取和操作文件的所有者、组和其他位。

文件权限

```

1 // perms.cpp
2 ...
3 #include <filesystem>
4 ...
5 namespace fs = std::filesystem;
6 void printPerms(fs::perms perm){
7     std::cout << ((perm & fs::perms::owner_read) != fs::perms::none ? "r" : "-")
8         << ((perm & fs::perms::owner_write) != fs::perms::none ? "w" : "-")
9         << ((perm & fs::perms::owner_exec) != fs::perms::none ? "x" : "-")
10        << ((perm & fs::perms::group_read) != fs::perms::none ? "r" : "-")
11        << ((perm & fs::perms::group_write) != fs::perms::none ? "w" : "-")
12        << ((perm & fs::perms::group_exec) != fs::perms::none ? "x" : "-")
13        << ((perm & fs::perms::others_read) != fs::perms::none ? "r" : "-")
14        << ((perm & fs::perms::others_write) != fs::perms::none ? "w" : "-")
15        << ((perm & fs::perms::others_exec) != fs::perms::none ? "x" : "-")
16        << '\n';
17 }
18 ...
19 ...
20 }
```

```

21 std::ofstream("rainer.txt");
22
23 std::cout << "Initial file permissions for a file: ";
24 printPerms(fs::status("rainer.txt").permissions()); // (1)
25
26 fs::permissions("rainer.txt", fs::perms::add_perms | // (2)
27     fs::perms::owner_all | fs::perms::group_all);
28 std::cout << "Adding all bits to owner and group: ";
29 printPerms(fs::status("rainer.txt").permissions());
30
31 fs::permissions("rainer.txt", fs::perms::remove_perms | // (3)
32     fs::perms::owner_write | fs::perms::group_write | fs::perms::others_write);
33 std::cout << "Removing the write bits for all: ";
34 printPerms(fs::status("rainer.txt").permissions());

```

通过 `fs::status("rainer.txt").permissions()`, 获得了文件 `rainer.txt` 的权限, 并可以在函数 `printPerms(1)` 中显示它们。通过设置类型 `std::filesystem::add_perms`, 可以向文件的所有者和组添加权限 (2)。或者, 可以设置常量 `std::filesystem::remove_perms` 来删除权限 (3)。

```

Initial file permissions for a file: rw-r--r--
Adding all bits to owner and group: rwxrwxr--
Removing the write bits for all:      r-xr-xr--

```

18.2. 非成员函数

存在许多非成员函数用于操作文件系统。

用于操作文件系统的非成员函数

非成员函数	描述
<code>absolute</code>	组成绝对路径。
<code>canonical</code> 和 <code>weakly_canonical</code>	组成规范路径。
<code>relative</code> 和 <code>proximate</code>	组成相对路径。
<code>copy</code>	复制文件或目录。
<code>copy_file</code>	复制文件内容。
<code>copy_symlink</code>	复制一个符号链接。
<code>create_directory</code> 和 <code>create_directories</code>	创建一个新目录。
<code>create_hard_link</code>	创建一个硬链接。
<code>create_symlink</code> 和 <code>create_directory_symlink</code>	创建符号链接。
<code>current_path</code>	返回当前工作目录。
<code>exists</code>	检查路径是否引用现有文件。
<code>equivalent</code>	检查两个路径是否指向同一个文件。

file_size	返回文件的大小。
hard_link_count	返回文件的硬链接数。
last_write_time	获取并设置最后一次文件修改的时间。
permissions	修改文件访问权限。
read_symlink	获取符号链接的目标。
remove	删除文件或空目录。
remove_all	递归地删除文件或目录及其所有内容。
rename	移动或重命名文件或目录。
resize_file	通过截断更改文件的大小。
space	返回文件系统上的可用空间。
status	确定文件属性。
symlink_status	确定文件属性并检查符号链接目标。
temp_directory_path	返回临时文件的目录。

读取和设置文件的最后写入时间

通过全局函数 `std::filesystem::last_write_time`, 可以读取和设置文件的最后写入时间。下面是一个基于en.cppreference.com中的 `last_write_time` 示例的示例。

文件创建的时间

```

1 // fileTime.cpp
2 ...
3 #include <filesystem>
4 ...
5 ...
6
7 namespace fs = std::filesystem;
8 using namespace std::chrono_literals;
9 ...
10 ...
11
12 fs::path path = fs::current_path() / "rainer.txt";
13 std::ofstream(path.c_str());
14 const auto fTime = fs::last_write_time(path); // (1)
15 const auto sTime = std::chrono::clock_cast<std::chrono::system_clock>(fTime); // (2)
16
17
18 std::time_t cftime = std::chrono::system_clock::to_time_t(sTime);
19 std::cout << "Write time on server " // (3)
20     << std::asctime(std::localtime(&cftime));
21 std::cout << "Write time on server " // (4)
22     << std::asctime(std::gmtime(&cftime)) << '\n';

```

```

23
24 const auto fTime2 = fTime + 2h; // (5)
25 const auto sTime2 = std::chrono::clock_cast<std::chrono::system_clock>(fTime2); // 
26 ↵ (6)
27 std::time_t cftime2 = std::chrono::system_clock::to_time_t(sTime2);
28 std::cout << "Local time on client "
     << std::asctime(std::localtime(&cftime2)) << '\n';

```

第 (1) 行给出了新创建文件的写时间，可以使用 (2) 中的 `fTime` 将其转换为一个实时挂钟 `sTime`。这个 `sTime` 在下面一行初始化 `std::chrono::system_clock`，`cftime` 的类型为 `std::filesystem::file_time_type`，在本例中是 `std::chrono::system_clock` 的别名；可以在 (3) 中初始化 `std::localtime`，并以文本形式表示日历时间。若使用 `std::gmtime(4)` 而不是 `std::localtime`，则不会有任何变化。这让我很困惑，因为协调世界时 (UTC) 与德国当地时间相差 2 小时。这是由于服务器的在线编译器在 en.cppreference.com，服务器将 UTC 时间与本地时间设置为相同时间。

下面是程序的输出。我将文件的写入时间移到未来 2 小时 (5)，并从文件系统 (6) 中读取它。这将调整时间，因此其与德国的当地时间相对应。

```

Write time on server Tue Oct 10 06:28:04 2017
Write time on server Tue Oct 10 06:28:04 2017

Local time on client Tue Oct 10 08:28:04 2017

```

文件系统空间信息

全局函数 `std::filesystem::space` 返回一个 `std::filesystem::space_info` 对象，该对象有三个成员：`capacity`、`free` 和 `available`。

- `capacity`: 文件系统的总大小
- `free`: 文件系统上的空闲空间
- `available`: 给非特权进程的自由空间 (等于或小于自由)

所有大小都以字节为单位。

下面程序的输出来自于 cppreference.com。这里尝试的所有路径都在同一个文件系统上，因此总是得到相同的答案。

空间信息

```

1 // space.cpp
2 ...
3 #include <filesystem>
4 ...
5 ...
6

```

```

7  namespace fs = std::filesystem;
8
9  ...
10
11 fs::space_info root = fs::space("/");
12 fs::space_info usr = fs::space("/usr");
13
14 std::cout << ". Capacity Free Available\n"
15     << "/" << root.capacity << " "
16     << root.free << " " << root.available << "\n"
17     << "usr " << usr.capacity << " "
18     << usr.free << " " << usr.available;

```

	Capacity	Free	Available
/	42140499968	18342744064	17054289920
usr	42140499968	18342744064	17054289920

18.3. 文件类型

通过使用以下谓词，可以轻松地查询文件类型。

文件系统的文件类型

文件类型	描述
is_block_file	检查路径是否指向一个块文件。
is_character_file	检查路径是否指向字符文件。
is_directory	检查路径是否指向一个目录。
is_empty	检查路径是否指向空文件或目录。
is_fifo	检查路径是否指向 named pipe 。
is_other	检查路径是否指向另一个文件。
is_regular_file	检查路径是否指向一个常规文件。
is_socket	检查路径是否指向 IPC 套接字。
is_symlink	检查路径是否指向符号链接。
status_known	检查文件状态是否已知。

获取文件的类型

谓词提供有关文件类型的信息，一个文件可能有多个谓词。引用普通文件的符号链接，既是普通文件，又是符号链接。

文件类型

```

1 // fileType.cpp
2 ...
3 #include <filesystem>
4 ...
5 ...
6
7 namespace fs = std::filesystem;
8
9 void printStatus(const fs::path& path_) {
10     std::cout << path_;
11     if(!fs::exists(path_)) std::cout << " does not exist";
12     else{
13         if(fs::is_block_file(path_)) std::cout << " is a block file\n";
14         if(fs::is_character_file(path_)) std::cout << " is a character device\n";
15         if(fs::is_directory(path_)) std::cout << " is a directory\n";
16         if(fs::is_fifo(path_)) std::cout << " is a named pipe\n";
17         if(fs::is_regular_file(path_)) std::cout << " is a regular file\n";
18         if(fs::is_socket(path_)) std::cout << " is a socket\n";
19         if(fs::is_symlink(path_)) std::cout << " is a symlink\n";
20     }
21 }
22 ...
23 ...
24
25 fs::create_directory("rainer");
26 printStatus("rainer");
27
28 std::ofstream("rainer/regularFile.txt");
29 printStatus("rainer/regularFile.txt");
30
31 fs::create_directory("rainer/directory");
32 printStatus("rainer/directory");
33
34 mkfifo("rainer/namedPipe", 0644);
35 printStatus("rainer/namedPipe");
36
37 struct sockaddr_un addr;
38 addr.sun_family = AF_UNIX;
39 std::strcpy(addr.sun_path, "rainer/socket");
40 int fd = socket(PF_UNIX, SOCK_STREAM, 0);
41 bind(fd, (struct sockaddr*)&addr, sizeof(addr));
42 printStatus("rainer/socket");
43
44 fs::create_symlink("rainer/regularFile.txt", "symlink");
45 printStatus("symlink");
46
47 printStatus("dummy.txt");
48

```

```
fs::remove_all("rainer");
```

```
"rainer" is a directory  
"rainer/regularFile.txt" is a regular file  
"rainer/directory" is a directory  
"rainer/namedPipe" is a named pipe  
"rainer/socket" is a socket  
"symlink" is a regular file  
      is a symlink  
"dummy.txt" does not exist
```

第 19 章 多线程



Cippi 在扎辫子

C++11 中，首次支持本机多线程。这种支持由两部分组成：定义良好的内存模型和标准化的线程接口。

19.1. 内存模型

多线程的基础是定义良好的内存模型，内存模型必须处理以下几点：

- 原子操作：可以不中断地执行的操作。
- 部分排序操作：不能重新排序的操作序列。
- 操作的可见性：保证对共享变量的操作在其他线程中可见。

C++ 内存模型与其前身 Java 内存模型有很多共同之处。

另外，C++ 允许打破顺序一致性，这是原子操作的默认行为。

顺序一致性提供了两个保证。

1. 程序的指令按源代码顺序执行。
2. 所有线程上的所有操作都遵循一个全局序。

19.2. 原子数据类型

C++ 有一组原子数据类型。首先是 `std::atomic_flag`，其次是类模板 `std::atomic`，还可以使用 `std::atomic` 定义原子数据类型。

`std::atomic_flag`

`std::atomic_flag` 是一个原子布尔值，有一个清晰和固定的状态。这里将 `clear` 状态称为 `false`，将 `set` 状态称为 `true`。`clear` 成员函数允许将其值设置为 `false`。使用 `test_and_set` 成员函数，可以将值设

置回 true，并返回前一个值。没有成员函数可以查询当前值，不过这在 C++20 中有所改变。C++20 中，`std::atomic_flag` 有一个测试成员函数，可以通过成员函数 `notify_one`、`notify_all` 和 `wait` 进行线程同步。

std::atomic_flag atomicFlag 的所有操作

成员函数	描述
<code>atomicFlag.clear()</code>	清除原子标志。
<code>atomicFlag.test_and_set()</code>	设置原子标志并返回旧值。
<code>atomicFlag.test()(C++20)</code>	返回标志的值。
<code>atomicFlag.notify_one()(C++20)</code>	通知一个等待原子标志的线程。
<code>atomicFlag.notify_all()(C++20)</code>	通知所有等待原子标志的线程。
<code>atomicFlag.wait(b)(C++20)</code>	阻塞线程，直到收到通知并且原子值发生变化。

`atomicFlag.test()` 返回 `atomicFlag` 的值而不改变它，也可以使用 `std::atomic_flag` 进行线程同步：`atomicFlag.wait()`、`atomicFlag.notify_one()` 和 `atomicFlag.notify_all()`。成员函数 `notify_one` 或 `notify_all` 通知一个或所有等待的原子标志。`atomicFlag.wait(boo)` 需要一个布尔值 `b`，`atomicFlag.wait(b)` 会阻塞直到收到通知。检查 `atomicFlag` 的值是否等于 `b`，若不等于 `b` 则解除阻塞。

必须使用 `ATOMIC_FLAG_INIT`:`std::atomic_flag` 标志 (`ATOMIC_FLAG_INIT`) 显式初始化 `std::atomic_flag`。C++20 中，默认构造的 `std::atomic_flag` 处于 `false` 状态。

`std::atomic_flag` 有一个突出的属性，它是唯一有保证的无锁的原子变量。其他功能强大的原子变量，可以通过 `std::mutex` 锁定机制来完成相同的功能。

std::atomic

类模板 `std::atomic` 有多种特化可用，需要头文件 `<atomic>`。`std::atomic<bool>` 和 `std::atomic<user-defined type>` 使用主模板。指针 `std::atomic<T*>` 提供偏特化，C++20 中智能指针 `std::atomic<smart T*>` 提供整型指针 `std::atomic<整型指针>` 的全特化，C++20 中浮点类型 `std::atomic<floating-point 针>` 的全特化，也可以使用 `std::atomic` 定义原子数据类型。

原子变量的基本接口

三个偏特化 `std::atomic<bool>`、`std::atomic<user-defined type>` 和 `std::atomic<smart T*>` 都支持基本原子接口。

成员函数	描述
<code>is_lock_free</code>	检查原子对象是否无锁。
<code>atomic_ref<T>::is_always_lock_free</code>	编译时检查原子类型是否总无锁。
<code>load</code>	原子地返回值。

operator T	原子地返回值。 相当于 atom.load()。
store	用非原子值自动替换原子值。
exchange	用新值自动替换该值，返回旧值。
compare_exchange_strong compare_exchange_weak	自动比较，并交换值。
notify_one(C++20)	通知一个原子等待操作。
notify_all(C++20)	通知所有原子等待操作。
wait(C++20)	阻塞，直到收到通知为止。 若与旧值相比不相等则返回

compare_exchange_strong 具有如下语法: `bool compare_exchange_strong(T& expected, T& desired)`。下面是对其行为的描述:

- 若 atomicValue 与期望的原子比较返回 true，则 atomicValue 在相同的原子操作中设置所需的价值。
- 若比较返回 false，则 expected 设置为 atomicValue。

自定义原子变量 `std::atomic<user-defined type>`

有了类模板 `std::atomic`，就可以定义用户定义的原子类型。

若将用户定义类型用于原子类型 `std::atomic<user-defined type>`，则对该类型有许多实质性的限制。原子类型 `std::atomic<user-defined type>` 支持与 `std::atomic<bool>` 相同的接口。

以下是用户定义类型成为原子类型的限制:

- 用户定义类型的复制赋值操作符对于其所有基类和非静态成员必须是简单数据类型。不能定义复制赋值操作符，但可以使用[default](#)从编译器请求它。
- 用户定义类型不能有虚成员函数或虚基类
- 用户定义的类型必须具有位可比性，这样才能应用 C 函数[memcpy](#)或[memcmp](#)

若用户定义类型的大小不大于 `int`，那么在主流平台上都可以对 `std::atomic<user-defined type>` 使用原子操作。

原子智能指针 `std::atomic<smart T*>` (C++20)

`std::shared_ptr` 由一个控制块和资源组成。控制块是线程安全的，但对资源的访问不是。所以修改引用计数器是一个原子操作，可以保证资源会精确地删除一次，这些都是 `std::shared_ptr` 提供的保证。使用偏特化模板 `std::atomic<std::shared_ptr<T>>` 和 `std::atomic<std::weak_ptr<T>>` 可以额外保证底层对象的访问是线程安全的。在 2022 年，原子智能指针上的所有操作都有锁。

`std::atomic<floating-point type>` (C++20)

除了基本原子接口之外，`std::atomic<floating-point type>` 还支持加法和减法。

基本原子接口的其他操作

成员函数	描述
<code>fetch_add, +=</code>	自动添加(减去)值。
<code>fetch_sub, -=</code>	返回旧值。

可以使用 `float`、`double` 和 `long double` 类型的全特化。

`std::atomic<T*>`

`std::atomic<T*>` 是类模板 `std::atomic` 的偏特化。它的行为就像一个普通的指针 `T*`。除了 `std::atomic<floating-point type>` 之外，`std::atomic<T*>` 还支持前后自增或前后自减操作。

`std::atomic<floating-point type>` 的其他操作

成员函数	描述
<code>++, -</code>	递增或递减(递增前和递增后)原子变量。

来看个简短的例子。

```
1 int intArray[5];
2 std::atomic<int*> p(intArray);
3 p++;
4 assert(p.load() == &intArray[1]);
5 p+=1;
6 assert(p.load() == &intArray[2]);
7 --p;
8 assert(p.load() == &intArray[1]);
```

`std::atomic<integral type>`

对于每个整型都有 `std::atomic` 的全特化 `std::atomic<integral type>`。`std::atomic<integer type>` 支持 `std::atomic<T*>` 或 `std::atomic<float-point type>` 支持的所有操作。此外，`std::atomic<integral type>` 支持与、或和异或的位逻辑操作符。

原子变量上的所有操作

成员函数	描述
fetch_or, = fetch_and, &=	自动对值执行位(与、或、异或)操作。 返回旧值。

复合位赋值操作和取操作之间有细微的区别。复合按位赋值操作返回新值，fetch 返回旧值。

更深入的观察可以提供更多的见解：没有原子乘法、原子除法或原子移位操作可用。这不是一个重要的限制，很少需要这些操作，并且可以很容易地实现。下面是一个原子 fetch_mult 函数的示例。

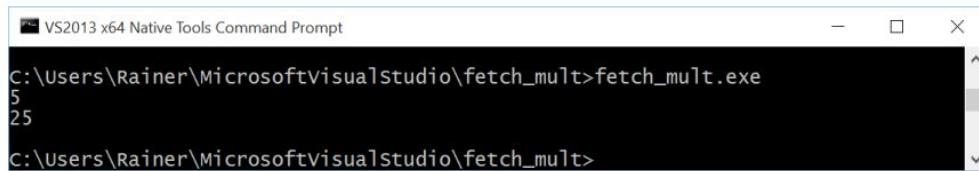
使用 *compare_exchange_strong* 的原子乘法

```

1 // fetch_mult.cpp
2
3 #include <atomic>
4 #include <iostream>
5
6 template <typename T>
7 T fetch_mult(std::atomic<T>& shared, T mult) {
8     T oldValue = shared.load();
9     while (!shared.compare_exchange_strong(oldValue, oldValue * mult));
10    return oldValue;
11 }
12
13 int main() {
14     std::atomic<int> myInt{5};
15     std::cout << myInt << '\n';
16     fetch_mult(myInt, 5);
17     std::cout << myInt << '\n';
18 }
```

第 9 行中的乘法只有在关系 `oldValue == shared` 成立时才会发生。我将乘法放入 while 循环中，以确保乘法总是发生，因为第 8 行中有两条指令用于读取 `oldValue`，第 9 行中有两条指令用于使用它。

这是原子乘法的结果。



std::atomic_ref

此外，类模板 std::atomic_ref 将原子操作应用于引用对象，原子对象的并发读写是线程安全的。引用对象的生存期必须超过 atomic_ref 的生存期。std::atomic_ref 支持相同的类型和操作，就像 std::atomic 支持其底层类型一样。

```
1 struct Counters {
2     int a;
3     int b;
4 };
5
6 Counter counter;
7 std::atomic_ref<Counters> cnt(counter);
```

所有原子操作

下表显示了所有原子操作。要获得有关这些操作的进一步信息，请参阅前面关于原子数据类型的章节。

所有原子操作取决于原子类型

成员函数	atomic_flag	atomic<bool> atomic<user> atomic<smart T*>	atomic<float>	atomic<T*>	atomic<int>
test_and_set	yes				
clear	yes				
is_lock_free		yes	yes	yes	yes
atomic<T>:: is_always_lock_free		yes	yes	yes	yes
load		yes	yes	yes	yes
operator T		yes	yes	yes	yes
store		yes	yes	yes	yes
exchange		yes	yes	yes	yes
compare_exchange_strong		yes	yes	yes	yes
compare_exchange_weak		yes	yes	yes	yes
fetch_add, +=			yes	yes	yes
fetch_sub, -=			yes	yes	yes
fetch_or, =					yes
fetch_and, &=					yes
fetch_xor, ^=					yes
++, -				yes	yes
notify_one(C++20)	yes	yes	yes	yes	yes

notify_all(C++20)	yes	yes	yes	yes	yes
wait(C++20)	yes	yes	yes	yes	yes

std::atomic<float> 表示原子浮点数类型, Std::atomic<int> 表示原子整数类型。

19.3. 线程

要使用 C++ 的多线程接口, 需要头文件 <thread>。C++11 中支持 std::thread, C++20 中支持改进的 std::jthread。

std::thread

创建

线程 std::thread 表示一个可执行单元。线程立即启动的可执行单元将其工作包作为可调用单元。可调用单元可以是函数、函数对象或 Lambda 函数:

创建线程

```

1 // threadCreate.cpp
2 ...
3 #include <thread>
4 ...
5 using namespace std;
6
7 void helloFunction() {
8     cout << "function" << endl;
9 }
10
11 class HelloFunctionObject {
12 public:
13     void operator()() const {
14         cout << "function object" << endl;
15     }
16 };
17
18 thread t1(helloFunction); // function
19
20 HelloFunctionObject helloFunctionObject;
21 thread t2(helloFunctionObject); // function object
22
23 thread t3([]{ cout << "lambda function"; }); // lambda function

```

生命周期

线程的创建者必须关心所创建线程的生命周期，创建的线程的可执行单元以可调用线程的结尾结束。要么创建者等待创建的线程 t 完成 (t.join())，要么创建者将自己从创建的线程中分离出来:t.detach()。若没有调用 t.join() 或 t.detach()，则线程 t 是可汇入的。可接合线程在其析构函数中抛出异常 std::terminate，程序终止。

线程的生命周期

```
1 // threadLifetime.cpp
2 ...
3 #include <thread>
4 ...
5
6 thread t1(helloFunction); // function
7
8 HelloFunctionObject helloFunctionObject;
9 thread t2(helloFunctionObject); // function object
10
11 thread t3([]{ cout << "lambda function"; }); // lambda function
12
13 t1.join();
14 t2.join();
15 t3.join();
```

从其创建者分离的线程通常称为守护线程，因为它在后台运行。

小心移动线程

线程可以移动，但不能复制。

```
1 #include <thread>
2 ...
3 std::thread t([]{ cout << "lambda function"; });
4 std::thread t2;
5 t2 = std::move(t);
6
7 std::thread t3([]{ cout << "lambda function"; });
8 t2 = std::move(t3); // std::terminate
```

通过执行 t2 = std::move(t)，线程 t2 获得线程 t 的可调用对象。假设线程 t2 已经有一个可调用对象并且是可连接的，C++ 运行时将调用 std::terminate。这恰好发生在 t2 = std::move(t3) 中，因为 t2 之前既没有执行 t2.join()，也没有执行 t2.detach()。

参数

`std::thread` 是可变模板，可以通过复制或引用获得任意数量的参数。可调用对象或线程都可以获得参数，线程将它们委托给可调用对象`tPerCopy2` 和 `tPerReference2`。

```
1 #include <thread>
2 ...
3
4 using namespace std;
5
6 void printStringCopy(string s){ cout << s; }
7 void printStringRef(const string& s){ cout << s; }
8
9 string s{"C++"};
10
11 thread tPerCopy([=]{ cout << s; }); // C++
12 thread tPerCopy2(printStringCopy, s); // C++
13 tPerCopy.join();
14 tPerCopy2.join();
15
16 thread tPerReference([&]{ cout << s; }); // C++
17 thread tPerReference2(printStringRef, s); // C++
18 tPerReference.join();
19 tPerReference2.join();
```

前两个线程通过复制获得参数，后两个线程通过引用获得参数。

默认情况下，线程应该通过复制获得参数

线程的参数

```
1 // threadArguments.cpp
2 ...
3 #include <thread>
4 ...
5
6 using std::this_thread::sleep_for;
7 using std::this_thread::get_id;
8
9 struct Sleeper{
10     Sleeper(int& i_):i{i_}(){}
11     void operator() (int k){
12         for (unsigned int j= 0; j <= 5; ++j){
13             sleep_for(std::chrono::milliseconds(100));
14             i += k;
15         }
16         std::cout << get_id(); // undefined behaviour
```

```

17 }
18 private:
19   int& i;
20 };
21
22 int valSleeper= 1000;
23
24 std::thread t(Sleeper(valSleeper), 5);
25 t.detach();
26
27 std::cout << valSleeper; // undefined behaviour

```

这个程序段有未定义行为。首先，`std::cout` 的生命周期绑定到主线程的生命周期，创建的线程通过引用获得它的变量 `valSleeper`。问题是创建的线程可能比它的创建者活得更长，若主线程结束，`std::cout` 和 `valSleeper` 将失去其有效性。其次，`valSleeper` 是一个共享变量，由主线程和子线程并发使用，这会是一场数据竞赛。

操作

可以在一个线程上执行许多操作。

`std::thread` 的操作

成员函数	描述
<code>t.join()</code>	直到线程 <code>t</code> 完成了可执行单元。
<code>t.detach()</code>	独立于创建者执行创建的线程。
<code>t.joinable()</code>	检查线程 <code>t</code> 是否支持 <code>join</code> 或 <code>detach</code> 调用。
<code>t.get_id()</code> and <code>std::this_thread::get_id()</code>	返回线程的标识。
<code>std::thread::hardware_concurrency()</code>	表示可以并行运行的线程数。
<code>std::this_thread::sleep_until(absTime)</code>	让线程进入睡眠状态，直到 <code>absTime</code> 。
<code>std::this_thread::sleep_for(relTime)</code>	使线程在 <code>relTime</code> 持续时间内休眠。
<code>std::this_thread::yield()</code>	让系统运行另一个线程。
<code>t.swap(t2)</code> and <code>std::swap(t1, t2)</code>	交换线程。

只能在线程 `t` 上调用一次 `t.join()` 或 `t.detach()`。若多次调用，会得到异常 `std::system_error`。
`std::thread::hardware_concurrency` 返回内核数，若运行时无法确定内核数，则返回 0。`sleep_until` 和 `sleep_for` 操作需要一个时间点或时间持续时间作为参数。

线程不能复制，但可以移动。交换操作在可能的情况下执行移动操作。

线程上的操作

```
1 // threadMember Functions.cpp
2 ...
3 #include <thread>
4 ...
5 using std::this_thread::get_id;
6
7 std::thread::hardware_concurrency() // 4
8
9 std::thread t1([]{ get_id(); }); // 139783038650112
10 std::thread t2([]{ get_id(); }); // 139783030257408
11 t1.get_id(); // 139783038650112
12 t2.get_id(); // 139783030257408
13
14 t1.swap(t2);
15
16 t1.get_id(); // 139783030257408
17 t2.get_id(); // 139783038650112
18 get_id(); // 140159896602432
```

std::jthread

std::jthread 代表可汇入线程。除了 C++11 中的 std::thread 之外，std::jthread 可以自动加入已启动的线程并发出中断信号。

自动汇入

std::thread 的非直观行为如下：当 std::thread 仍然可汇入，std::terminate 在其析构函数中调用。相反，若 std::jthread 仍然可汇入，则自动连接到其析构函数中。

终止仍然可汇入的 std::jthread

```
1 // jthreadJoinable.cpp
2 ...
3 #include <thread>
4 ...
5
6 std::jthread thr=[]{ std::cout << "std::jthread" << "\n"; }; // std::jthread
7
8 std::cout << "thr.joinable(): " << thr.joinable() << "\n"; // thr.joinable(): true
```

除了 std::thread 之外，std::jthread 也是可中断的。

19.4. 停止令牌

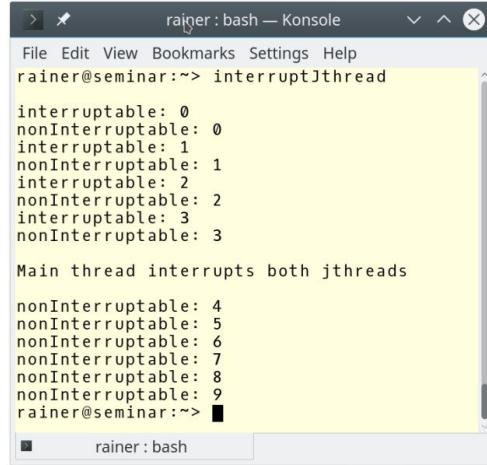
汇入线程的附加功能基于 std::stop_token、std::stop_callback 和 std::stop_source。下面的代码会给你有一个大致的了解。

中断不可中断和可中断的 *std::jthread*

```
1 // interruptJthread.cpp
2
3 ...
4
5 #include <thread>
6 #include <stop_token>
7
8 using namespace::std::literals;
9
10 ...
11
12 std::jthread nonInterruptable([]{ // (1)
13     int counter{0};
14     while (counter < 10){
15         std::this_thread::sleep_for(0.2s);
16         std::cerr << "nonInterruptable: " << counter << '\n';
17         ++counter;
18     }
19 });
20
21 std::jthread interruptable([](std::stop_token stoken){ // (2)
22     int counter{0};
23     while (counter < 10){
24         std::this_thread::sleep_for(0.2s);
25         if (stoken.stop_requested()) return; // (3)
26         std::cerr << "interruptable: " << counter << '\n';
27         ++counter;
28     }
29 });
30
31 std::this_thread::sleep_for(1s);
32
33 std::cerr << "Main thread interrupts both jthreads" << std::endl;
34 nonInterruptable.request_stop(); // (4)
35 interruptable.request_stop();
36
37 ...
```

主程序中启动了两个线程 nonInterruptable 和 interruptable((1) 和 (2))。与线程 nonInterruptable 相反，线程 interruptable 获得一个 std::stop_token，并在 (3) 中使用它来检查它是否中断:stoken_stop_requested()。中断的情况下，Lambda 函数返回，线程

结束。调用 `interruptable.request_stop()`(4) 触发线程的结束。这并不适用于之前的调用 `nonInterruptable.request_stop()`，后者没有效果。



```
File Edit View Bookmarks Settings Help
rainer@seminar:~> interruptJthread
interruptable: 0
nonInterruptable: 0
interruptable: 1
nonInterruptable: 1
interruptable: 2
nonInterruptable: 2
interruptable: 3
nonInterruptable: 3
Main thread interrupts both jthreads
nonInterruptable: 4
nonInterruptable: 5
nonInterruptable: 6
nonInterruptable: 7
nonInterruptable: 8
nonInterruptable: 9
rainer@seminar:~>
```

`std::stop_token`, `std::stop_source` 和 `std::stop_callback`

`stop_token`、`std::stop_callback` 或 `std::stop_source` 使其能够异步请求执行停止或询问执行是否获得停止信号。`std::stop_token` 可以传递给操作，然后用于主动轮询令牌以获得停止请求或通过 `std::stop_callback` 注册回调。`std::stop_source` 发送停止请求，此信号影响所有相关的 `std::stop_token`。`std::stop_source`、`std::stop_token` 和 `std::stop_callback` 这三个类共享一个相关停止状态的所有权。`request_stop()`、`stop_requested()` 和 `stop_possible()` 都是原子操作。

组件 `std::stop_source` 和 `std::stop_token` 为停止处理提供了以下属性。

`std::stop_source` 的构造函数

```
1 stop_source();
2 explicit stop_source(std::nostopstate_t) noexcept;
```

默认构造的 `std::stop_source` 获得一个停止源。接受 `std::nostopstate_t` 参数的构造函数构造了一个空的 `std::stop_source`，没有关联的停止状态。

`std::stop_source` 的成员函数

成员函数	描述
<code>src.get_token()</code>	若 <code>stop_possible()</code> ，则返回一个对应停止状态的 <code>stop_token</code> 。 否则，返回一个默认构造的(空) <code>stop_token</code> 。
<code>src.stop_possible()</code>	若 <code>src</code> 可以请求停止，则为 <code>true</code> 。
<code>src.stop_requested()</code>	若其中一个所有者调用 <code>stop_possible()</code> 和 <code>request_stop()</code> ，则为 <code>true</code> 。
<code>src.request_stop()</code>	若 <code>stop_possible()</code> 和 <code>!stop_requested()</code> ，调用一个停止请求。 否则，调用无效。

`src.stop_possible()` 表示具有关联的停止状态。`src.stop_requested()` 在具有关联的停止状态，并且在请求停止时返回 `true`。`src.request_stop()` 成功，若具有相关的停止状态并已接收到停止请求，则返回 `true`。

`src.get_token()` 返回停止令牌，可以检查是否已经发出停止请求，或者是否可以针对其关联的停止源 `src` 发出停止请求。停止令牌符号观察停止源 `src`。

std::stop_token stoken 的成员函数

成员函数	描述
<code>src.stop_possible()</code>	若 <code>stoken</code> 有关联的停止状态，则返回 <code>true</code> 。
<code>src.stop_requested()</code>	若在相应的 <code>std::stop_source src</code> 上调用了 <code>request_stop()</code> ，则返回 <code>true</code> ；否则，返回 <code>false</code> 。

默认构造的停止令牌没有关联的停止状态。若停止请求已经发出，`stoken.stop_possible()` 也返回 `true`。`stoken.stop_requested()` 在停止令牌具有关联的停止状态，并且已经接收到停止请求时返回 `true`。

若 `std::stop_token` 应该暂时禁用，可以用默认构造的 `token` 替换。默认构造的令牌没有关联的停止状态，下面的代码段展示了如何禁用和启用线程接受停止请求的能力。

暂时禁用停止令牌

```
1 std::jthread jthr([](std::stop_token stoken) {
2     ...
3     std::stop_token interruptDisabled;
4     std::swap(stoken, interruptDisabled); // (1)
5     ... // (2)
6     std::swap(stoken, interruptDisabled);
7     ...
8 }
```

`std::stop_token interruptDisabled` 没有关联的停止状态，所以线程 `jthr` 可以接受除 (1) 和 (2) 行以外的所有行中的停止请求。

19.5. 共享变量

若多个线程共享一个变量，则必须协调访问。这就是 C++ 中互斥锁的工作。

数据竞争

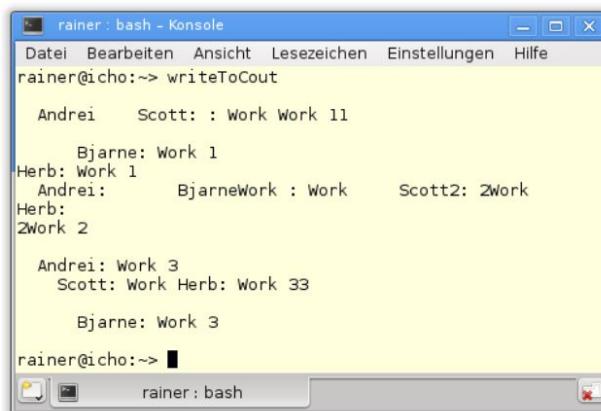
数据竞争

数据竞争是指至少有两个线程同时访问共享数据，并且至少有一个线程是写线程的状态，程序具有未定义行为。

若有几个线程写入 std::cout，可以很好地观察到线程的交错。本例中，输出流 std::cout 是共享变量。

不同步的写入 std::cout

```
1 // withoutMutex.cpp
2 ...
3 #include <thread>
4 ...
5
6 using namespace std;
7
8 struct Worker{
9     Worker(string n):name(n) {};
10    void operator() () {
11        for (int i= 1; i <= 3; ++i) {
12            this_thread::sleep_for(chrono::milliseconds(200));
13            cout << name << ":" << "Work " << i << endl;
14        }
15    }
16    private:
17    string name;
18 };
19
20 thread herb= thread(Worker("Herb"));
21 thread andrei= thread(Worker(" Andrei"));
22 thread scott= thread(Worker (" Scott"));
23 thread bjarne= thread(Worker(" Bjarne"));
```



std::cout 的输出不协调。

流是线程安全的

C++11 标准保证字符自动写入，不需要保护。只有当整个读操作不交错时，才需要保护流上线程的交错。这种保证适用于输入和输出流。

C++20 中，已经同步了像 std::osyncstream 和 std::wosyncstream 这样的输出流，保证对一个输出流的写入是同步的。输出写入内部缓冲区，并在超出作用域时刷新。同步输出流可以有一个名称，比如 synced_out，也可以没有名称。

同步输出流

```
1 {
2     std::osyncstream synced_out(std::cout);
3     synced_out << "Hello, ";
4     synced_out << "World!";
5     synced_out << '\n'; // no effect
6     synced_out << "and more!\n";
7 } // destroys the synced_output and emits the internal buffer
8
9 std::osyncstream(std::cout) << "Hello, " << "World!" << "\n";
```

本例中，std::cout 是共享变量，应该对流具有独占访问权。

互斥锁

互斥 (互斥量) 保证一次只有一个线程可以访问临界区域，需要头文件 <mutex>。互斥锁 m 通过调用 m.lock() 来锁定临界区，并通过 m.unlock() 来解锁。

使用 std::mutex 进行同步

```
1 // mutex.cpp
2 ...
3 #include <mutex>
4 #include <thread>
5 ...
6
7 using namespace std;
8
9 std::mutex mutexCout;
10
11 struct Worker{
12     Worker(string n):name(n){};
13     void operator() () {
14         for (int i= 1; i <= 3; ++i){
15             this_thread::sleep_for(chrono::milliseconds(200));
16             mutexCout.lock();
17             cout << name << ":" << "Work " << i << endl;
18             mutexCout.unlock();
```

```

19     }
20 }
21 private:
22     string name;
23 };
24
25 thread herb= thread(Worker("Herb"));
26 thread andrei= thread(Worker(" Andrei"));
27 thread scott= thread(Worker (" Scott"));
28 thread bjarne= thread(Worker(" Bjarne"));

```

因为使用相同的互斥锁 mutexCout，所以每个线程在彼此之后协调写入 std::cout。



C++ 有五种不同的互斥量。可以递归地锁定，暂时有或没有时间限制。

C++ 的互斥量

成员函数	mutex	recursive_mutex	timed_mutex	recursive_timed_mutex	shared_timed_mutex
m.lock	yes	yes	yes	yes	yes
m.unlock	yes	yes	yes	yes	yes
m.try_lock	yes	yes	yes	yes	yes
m.try_lock_for			yes	yes	yes
m.try_lock_until			yes	yes	yes

`std::shared_time_mutex` 使它能够实现读写锁。成员函数 `m.try_lock_for(relTime)` 需要一个相对时间段；成员函数 `m.try_lock_until(absTime)` 一个绝对时间点。

死锁

死锁是两个或多个线程被阻塞的状态，因为每个线程在释放其资源之前等待资源的释放。

若忘记调用 `m.unlock()`，可能很快就会出现死锁。例如，在函数 `getVar()` 中出现异常的情况下，就会发生这种情况。

```
1 m.lock();
2 sharedVar= getVar();
3 m.unlock()
```

不要在持有锁时调用未知函数

若函数 `getVar` 试图通过调用 `m.lock()` 来获取相同的锁，将会死锁。因为它不会成功，并且调用将永远阻塞。

以错误的顺序锁定两个互斥锁，是导致死锁的另一个典型原因。

死锁

```
1 // deadlock.cpp
2 ...
3 #include <mutex>
4 ...
5
6 struct CriticalData{
7     std::mutex mut;
8 };
9
10 void deadLock(CriticalData& a, CriticalData& b){
11     a.mutex.lock();
12     std::cout << "get the first mutex\n";
13     std::this_thread::sleep_for(std::chrono::milliseconds(1));
14     b.mutex.lock();
15     std::cout << "get the second mutex\n";
16     a.mutex.unlock(), b.mutex.unlock();
17 }
18
19 CriticalData c1;
20 CriticalData c2;
21
22 std::thread t1([&]{ deadLock(c1, c2); });
23 std::thread t2([&]{ deadLock(c2, c1); });
```

一毫秒的短时间窗口 (`std::this_thread::sleep_for(std::chrono::milliseconds(1))`) 足以产生高概率的死锁，因为每个线程都在等待另一个互斥锁。



将互斥锁封装在锁中

很容易忘记解锁互斥锁或以不同的顺序锁定互斥锁。为了克服互斥锁的大多数问题，可以将互斥锁封装在锁中。

锁

应该将互斥锁封装在锁中以自动释放互斥锁。锁是 RAII 习惯用法的一种实现，因为锁将互斥锁的生命周期与它的生命周期绑定在一起。C++11 分别为简单用例提供了 `std::lock_guard`，为高级用例提供了 `std::unique_lock`，都需要头文件 `<mutex>`。C++14 中，有一个 `std::shared_lock`，与互斥锁 `std::shared_time_mutex` 结合在一起使用，是实现读写锁的基础。

`std::lock_guard`

`std::lock_guard` 只支持简单的用例，所以只能在构造函数中绑定互斥锁，并在析构函数中释放互斥锁，`worker` 示例的同步可简化为构造函数的调用。

使用 `std::lock_guard` 进行同步

```
1 // lockGuard.cpp
2 ...
3 std::mutex coutMutex;
4
5 struct Worker{
6     Worker(std::string n):name(n){}
7     void operator() () {
8         for (int i= 1; i <= 3; ++i){
9             std::this_thread::sleep_for(std::chrono::milliseconds(200));
10            std::lock_guard<std::mutex> myLock(coutMutex);
11            std::cout << name << ":" << "Work " << i << '\n';
12     }
13 }
```

```
13     }
14 private:
15     std::string name;
16 }
```

std::unique_lock

std::unique_lock 的开销比 std::lock_guard 的开销大。相反，std::unique_lock 可以在有互斥锁和没有互斥锁的情况下创建，显式地锁定或释放它的互斥锁，或者延迟它的互斥锁。

下表显示了 std::unique_lock 类的成员函数。

std::unique_lock lk 的接口

成员函数	描述
lk.lock()	锁定关联的互斥锁。
std::lock(lk1, lk2, ...)	自动锁定任意数量的关联互斥锁。
lk.try_lock() and lk.try_lock_for(relTime) and lk.try_lock_until(absTime)	尝试锁定关联的互斥锁。
lk.release()	释放互斥锁。互斥锁保持锁定状态。
lk.swap(lk2) and std::swap(lk, lk2)	交换锁。
lk.mutex()	返回一个指向关联互斥对象的指针。
lk.owns_lock()	检查锁是否有互斥锁。

由于获取锁的顺序不同而导致的死锁，可以很通过 std::atomic 解决。

std::unique_lock

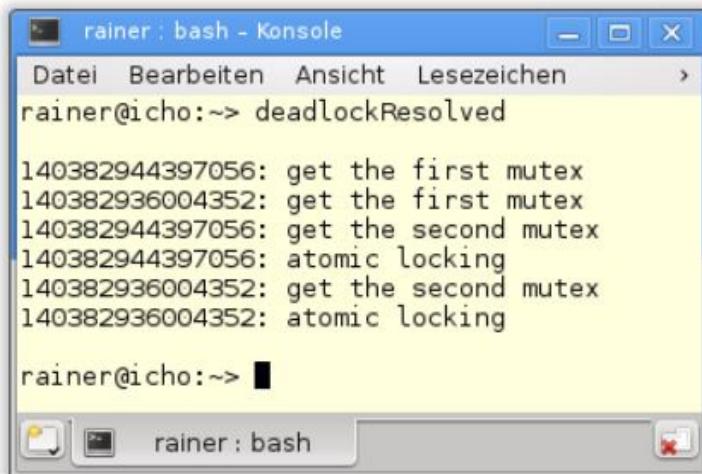
```
1 // deadLockResolved. cpp
2 ...
3 #include <mutex>
4 ...
5
6 using namespace std;
7 struct CriticalData{
8     mutex mut;
9 }
10
11 void deadLockResolved(CriticalData& a, CriticalData& b){
12     unique_lock<mutex>guard1(a.mut, defer_lock);
13     cout << this_thread::get_id() << ": get the first lock" << endl;
14     this_thread::sleep_for(chrono::milliseconds(1));
15 }
```

```

15     unique_lock<mutex>guard2(b.mut, defer_lock);
16     cout << this_thread::get_id() << ": get the second lock" << endl;
17     cout << this_thread::get_id() << ": atomic locking";
18     lock(guard1, guard2);
19 }
20
21 CriticalData c1;
22 CriticalData c2;
23
24 thread t1([&]{ deadLockResolved(c1, c2); });
25 thread t2([&]{ deadLockResolved(c2, c1); });

```

由于 std::unique_lock 的参数 std::defer_lock, a.mut 和 b.mut 则延迟锁定。锁定在调用 std::lock(guard1, guard2) 时自动发生。



std::shared_lock

std::shared_lock 与 std::unique_lock 具有相同的接口，std::shared_lock 支持多个线程共享同一个锁定互斥量的情况。对于这个特殊的用例，必须将 std::shared_lock 与 std::shared_timed_mutex 结合使用。但若多个线程在一个 std::unique_lock 中使用同一个 std::shared_time_mutex，则只有一个线程可以拥有它。

```

1 #include <mutex>
2 ...
3
4 std::shared_timed_mutex sharedMutex;
5
6 std::unique_lock<std::shared_timed_mutex> writerLock(sharedMutex);
7
8 std::shared_lock<std::shared_time_mutex> readerLock(sharedMutex);
9 std::shared_lock<std::shared_time_mutex> readerLock2(sharedMutex);

```

该示例展示了典型的读写锁场景，类型 `std::unique_lock<std::shared_timed_mutex>` 的 `writerLock` 只能独占 `sharedMutex`。`std::shared_lock<std::shared_time_mutex>` 类型的读锁 `readerLock` 和 `readerLock2` 可以共享同一个互斥锁 `sharedMutex`。

线程安全的初始化

若不修改数据，则以线程安全的方式初始化就足够了。C++ 提供了多种方法来实现这一点：使用常量表达式，使用具有块作用域的静态变量，或者使用函数 `std::call_once` 结合标志 `std::once_flag`。

常数表达式

常量表达式在编译时初始化，其本身是线程安全的。通过在变量前使用关键字 `constexpr`，变量就变成了常量表达式。用户定义类型的实例也可以是常量表达式，若将成员函数声明为常量表达式，则可以以线程安全的方式初始化。

```
1 struct MyDouble {
2     constexpr MyDouble(double v) : val(v) {}
3     constexpr double getValue() { return val; }
4     private:
5         double val
6     };
7
8 constexpr MyDouble myDouble(10.5);
9 std::cout << myDouble.getValue(); // 10.5
```

块中的静态变量

若在块中定义静态变量，C++11 运行时保证以线程安全的方式初始化它。

```
1 void blockScope() {
2     static int MySharedDataInt= 2011;
3 }
```

`std::call_once` 和 `std::once_flag`

`std::call_once` 接受两个参数：标志 `std::once_flag` 和一个可调用对象。C++ 运行时借助标志 `std::once_flag` 保证可调用对象只执行一次。

线程安全的初始化

```
1 // callOnce.cpp
2 ...
3 #include <mutex>
```

```

4   ...
5
6 using namespace std;
7
8 once_flag onceFlag;
9 void do_once() {
10    call_once(onceFlag, []{ cout << "Only once." << endl; });
11 }
12 thread t1(do_once);
13 thread t2(do_once);

```

虽然两个线程都执行了函数 `do_once`，但只有一个线程执行成功，而 Lambda 函数 `[]{cout << "only once"。<< endl;}` 只执行一次。



可以使用相同的 `std::once_flag` 来注册不同的可调用对象，并且只调用其中一个可调用对象。

19.6. 线程本地数据

使用关键字 `thread_local`，就拥有了线程本地数据，也称为线程本地存储。每个线程都有自己的数据副本，线程本地数据的行为类似于静态变量。它们是在第一次使用时创建的，其生命周期与线程的生命周期绑定在一起。

线程本地数据

```

1 // threadLocal.cpp
2 ...
3 std::mutex coutMutex;
4 thread_local std::string s("hello from ");
5
6 void addThreadLocal(std::string const& s2) {
7     s+= s2;
8     std::lock_guard<std::mutex> guard(coutMutex);
9     std::cout << s << '\n';
10    std::cout << "&s: " << &s << '\n';
11    std::cout << '\n';
12 }
13

```

```

14     std::thread t1(addThreadLocal, "t1");
15     std::thread t2(addThreadLocal, "t2");
16     std::thread t3(addThreadLocal, "t3");
17     std::thread t4(addThreadLocal, "t4");

```

每个线程都有一个 `thread_local` 字符串的副本，每个字符串都独立地修改它的字符串，并且每个字符串都有唯一地址：

The screenshot shows a terminal window titled "rainer : bash - Konsole <2>". The command "threadLocal" was run, which triggered four separate threads (t1, t2, t3, t4). Each thread prints its name followed by "hello from" and its unique memory address. The output is:

```

Datei  Bearbeiten  Ansicht  >
rainer@icho:~> threadLocal
hello from t1
&s: 0x7f64a4b256f8

hello from t4
&s: 0x7f64a33226f8

hello from t2
&s: 0x7f64a43246f8

hello from t3
&s: 0x7f64a3b236f8

rainer@icho:~> ■

```

19.7. 条件变量

条件变量允许通过消息同步线程，则需要头文件 `<condition_variable>`。一个线程充当消息的发送者，另一个线程充当消息的接收者。接收方等待发送方的通知。条件变量的典型用例是生产者-消费者工作流。

条件变量可以是消息的发送方和接收方。

条件变量 cv 的成员函数

成员函数	描述
<code>cv.notify_one()</code>	通知一个等待线程。
<code>cv.notify_all()</code>	通知所有等待的线程。
<code>cv.wait(lock, ...)</code>	在持有 <code>std::unique_lock</code> 的同时等待通知。
<code>cv.wait_for(lock, relTime, ...)</code>	在持有 <code>std::unique_lock</code> 的同时，在给定时间段内等待等待通知。
<code>cv.wait_until(lock, absTime, ...)</code>	在持有 <code>std::unique_lock</code> 的同时，在给定时间段内等待等待通知。

发送方和接收方需要一个锁。在发送方的情况下，`std::lock_guard` 就足够了，只调用一次 `lock` 和 `unlock`。对于接收方来说，`std::unique_lock` 是必要的，通常会锁定和解锁它的互斥锁几次。

条件变量

```

1 // conditionVariable.cpp
2 ...
3 #include <condition_variable>
4 ...
5
6 std::mutex mutex_;
7 std::condition_variable condVar;
8 bool dataReady= false;
9
10 void doTheWork() {
11     std::cout << "Processing shared data." << '\n';
12 }
13
14 void waitingForWork() {
15     std::cout << "Worker: Waiting for work." << '\n';
16     std::unique_lock<std::mutex> lck(mutex_);
17     condVar.wait(lck, []{ return dataReady; });
18     doTheWork();
19     std::cout << "Work done." << '\n';
20 }
21
22 void setDataReady() {
23     std::lock_guard<std::mutex> lck(mutex_);
24     dataReady=true;
25     std::cout << "Sender: Data is ready." << '\n';
26     condVar.notify_one();
27 }
28
29 std::thread t1(waitingForWork);
30 std::thread t2(setDataReady);

```



使用条件变量听起来很简单，但是有两个关键问题。

避免伪唤醒

为了保护自身免受伪唤醒，条件变量的 wait 调用应该使用相关的谓词。谓词确保通知确实来自发送方。我使用 Lambda 函数 []{return dataReady;} 作为谓词。发送方将 dataReady 设置为 true。

避免未唤醒

为了保护自己未唤醒，条件变量的 wait 调用应该使用额外的谓词。谓词确保发送方的通知不会丢失。若发送方在接收方等待之前通知接收方，则通知丢失。接收者将永远等待。接收方现在首先检查它的谓词:[]{return dataReady;}。

19.8. 信号量

信号量是一种同步机制，用于控制对共享资源的并发访问。计数信号量是计数器大于零的特殊信号量，计数器在构造函数中初始化。获取信号量会减少计数器，释放信号量会增加计数器。若一个线程试图在计数器为零时获取信号量，则该线程将阻塞，直到另一个线程通过释放信号量来增加计数器。

C++20 支持 std::binary_semaphore，它是 std::counting_semaphore<1> 的别名。在这种情况下，最小最大值为 1。

```
1 using binary_semaphore = std::counting_semaphore<1>;
```

与 std::mutex 不同，std::counting_semaphore 不绑定到线程，所以获取和释放调用可以发生在不同的线程上。下表给出了 std::counting_semaphore 的接口。

信号量 sem 的成员函数

成员函数	描述
counting_semaphore::max()	返回计数器的最大值。
sem.release(upd=1)	自动使计数器增加 upd。
sem.acquire()	执行 semt.try_acquire() 并阻塞，直到 counter 大于零。
sem.try_acquire()	自动减少计数器。
sem.try_acquire_for(relTime)	在一段时间内执行 sem.try_acquire()。
sem.try_acquire_intil(absTime)	执行 semt.try_acquire() 直到时间点。

成员函数 sem.try_lock_for(relTime) 需要一个相对时间段；成员函数 sem.try_lock_until(absTime) 需要一个绝对时间点。

信号量通常是条件变量更安全、更快的替代方案：

std::counting_semaphore

```

1 // threadSynchronisationSemaphore.cpp
2 #include <semaphore>
3
4 ...
5
6 std::counting_semaphore<1> prepareSignal(0); // (1)
7
8 void prepareWork() {
9
10    myVec.insert(myVec.end(), {0, 1, 0, 3});
11    std::cout << "Sender: Data prepared." << '\n';
12    prepareSignal.release(); // (2)
13 }
14
15 void completeWork() {
16
17    std::cout << "Waiter: Waiting for data." << '\n';
18    prepareSignal.acquire(); // (3)
19    myVec[2] = 2;
20    std::cout << "Waiter: Complete the work." << '\n';
21    for (auto i: myVec) std::cout << i << " ";
22    std::cout << '\n';
23 }
24 ...
25
26 std::thread t1(prepareWork);
27 std::thread t2(completeWork);
28
29 t1.join();
30 t2.join();

```

std::counting_semaphore prepareSignal(第一行) 可以有值 0 和 1。在示例中，初始化为 0。所以，prepareSignal.release() 将该值设置为 1(第 2 行)，并解除调用 prepareSignal.acquire()(第 3 行) 的阻塞。

```

C:\Users\seminar>threadSynchronisationSemaphore.exe
Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>threadSynchronisationSemaphore.exe
Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>threadSynchronisationSemaphore.exe
Waiter: Waiting for data.
Sender: Data prepared.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>threadSynchronisationSemaphore.exe
Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>

```

19.9. 协调类型

锁存器和栅栏是允许某些线程阻塞直到计数器变为零的协调类型。在 C++20 中，有两个版本的锁存器和栅栏:std::latch 和 std::barrier。

std::latch

现在，来仔细看看 std::latch 的接口。

std::latch lat 的成员函数

成员函数	描述
lat.count_down(upd=1)	不阻塞调用者的情况下，通过 upd 自动减少计数器。
lat.try_wait()	若 counter == 0 则返回 true
lat.wait()	若 counter == 0，立即返回。否则，阻塞直到 counter == 0。
lat.arrive_and_wait(upd=1)	相当于 count_down(upd);wait();。

upd 的默认值为 1。当 upd 大于计数器或为负值时，程序具有未定义行为。调用 lat.try_wait() 不像它的名字所示的那样，会等待。

下面的程序使用两个 std::latch 来构建一个“老板与雇员们”的工作流。我使用 synchronizedOut 函数将输出同步到 std::cout(第 13 行)。这种同步使遵循工作流变得更加容易。

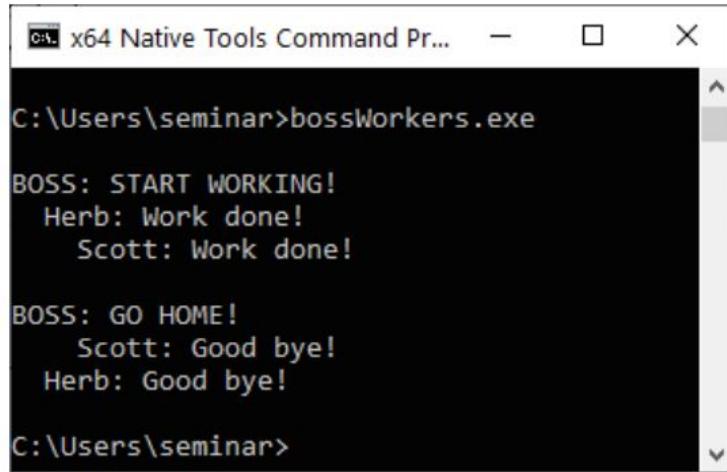
std::latch

```

1 // bossWorkers.cpp
2
3 #include <latch>
4 ...
5
6 std::latch workDone(2);
7 std::latch goHome(1); // (5)
8
9 void synchronizedOut(const std::string s) {
10    std::lock_guard<std::mutex> lo(coutMutex);
11    std::cout << s;
12 }
13
14 class Worker {
15 public:
16     Worker(std::string n): name(n) { };
17     void operator() () {
18         // notify the boss when work is done
19         synchronizedOut(name + ": " + "Work done!\n");
20         workDone.count_down(); // (3)
21         // waiting before going home
22         goHome.wait();
23         synchronizedOut(name + ": " + "Good bye!\n");
24     }
25 private:
26     std::string name;
27 };
28
29 ...
30
31 std::cout << "BOSS: START WORKING! " << '\n';
32 Worker herb("    Herb"); // (1)
33 std::thread herbWork(herb);
34
35 Worker scott("    Scott"); // (2)
36 std::thread scottWork(scott);
37
38 workDone.wait(); // (4)
39
40 std::cout << '\n';
41
42 goHome.count_down();
43
44 std::cout << "BOSS: GO HOME!" << '\n';
45
46 herbWork.join();
47 scottWork.join();

```

工作流的概念很简单。两个雇员 herb 和 scott(第 1 行和第 2 行)必须完成他们的工作。当它们完成它们的工作(第 3 行)时，其倒数 std::latch workDone。boss(主线程)在第(4)行被阻塞，直到计数器变为 0。当计数器为 0 时，老板使用第二个 std::latch goHome 来通知它的雇员回家。在这种情况下，初始计数器为 1(第 5 行)。调用 goHome.wait(0) 阻塞，直到计数器变为 0。



```
C:\Users\seminar>bossWorkers.exe

BOSS: START WORKING!
    Herb: Work done!
    Scott: Work done!

BOSS: GO HOME!
    Scott: Good bye!
    Herb: Good bye!

C:\Users\seminar>
```

std::barrier 类似于 std::latch。

std::barrier

std::latch 和 std::barrier 有两个不同之处。首先，可以多次使用 std::barrier；其次，可以为下一步(迭代)设置计数器。计数器变为零后，立即开始所谓的完成步骤。在这个完成步骤中，将调用[可调用对象](#)。栅栏在其构造函数中获得可调用对象。

完成步骤如下：

1. 所有线程都阻塞。
2. 任意线程解除阻塞，并执行可调用对象。
3. 若完成步骤，所有线程都将解除阻塞。

std::barrier bar 的成员函数

成员函数	描述
bar.arrive(upd)	计数器按 upd 自动递减。
bar.wait()	阻塞在同步点，直到完成步骤。
bar.arrive_and_wait()	相当于 wait(arrive())
bar.arrive_and_drop()	将当前和后续相位的计数器减 1。
std::barrier::max	实现支持的最大值。

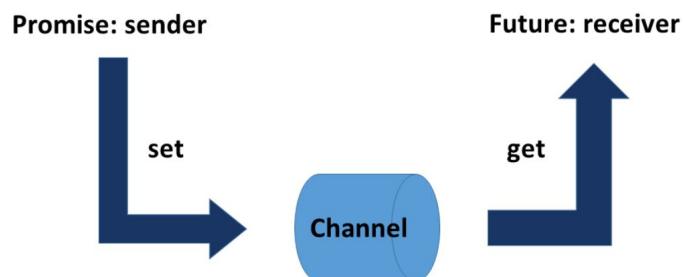
bar.arrive_and_drop() 调用本质上意味着计数器在下一阶段减 1。

19.10. 任务

除了线程之外，C++ 还有异步执行工作的任务，需要头文件 `<future>`。任务是由两个相关组件组成的工作包参数化的`:promise` 和 `future`，两者都通过数据通道连接。`promise` 执行工作包并将结果放入数据通道；相关的 `future` 获取结果。两个通信端点都可以在单独的线程中运行。特别的是，`future` 可以在以后获取结果，所以 `promise` 对结果的计算独立于相关 `future` 对结果的查询。

将任务视为数据通道

任务的行为类似于数据通道。`promise` 将其结果放在数据通道中，`future` 等着获取结果。



线程与任务

线程与任务不同。

必须使用共享变量来实现创建线程和被创建线程之间的通信。任务通过其数据通道进行通信，该数据通道受到隐式保护，任务不能使用像互斥锁这样的保护机制。

创建线程正在用 `join` 调用等待它的子线程。若没有结果则阻塞，可使用 `fut.get()` 获取结果。

若在创建的线程中发生异常，则创建的线程、创建者和整个进程终止。相反，`promise` 可以向 `future` 发送异常，`future` 必须处理该异常。

一个 `promise` 可以服务于一个或多个 `future`，可以发送值、异常或仅发送通知。

可以使用任务作为条件变量的安全替代。

```
1 #include <future>
2 #include <thread>
3 ...
4
5 int res;
6 std::thread t([&]{ res= 2000+11; });
7 t.join();
8 std::cout << res << '\n'; // 2011
9
10 auto fut= std::async([]{ return 2000+11; });
11 std::cout << fut.get() << '\n'; // 2011
```

子线程 t 和异步函数调用 std::async 计算 2000 和 11 的和，创建线程通过共享变量 res 从子线程 t 获取结果。调用 std::async 在发送方 (promise) 和接收方 (future) 之间创建数据通道。future 使用 fut.get() 向数据通道请求计算结果，fut.get() 会阻塞线程。

std::async

std::async 的行为类似于异步函数调用。这个函数调用接受一个可调用对象及其参数。std::async 是一个可变的模板，可以接受任意数量的参数。调用 std::async 返回一个 future 对象 fut，通过 fut.get() 获取结果的句柄。还可以为 std::async 指定一个启动策略，可以使用启动策略显式地确定异步操作是应该在同一个线程 (std::launch::deferred) 中执行，还是应该在另一个线程 (std::launch::async) 中执行。

调用 auto fut= std::async(std::launch::deferred, ...) 不会立即执行。调用 fut.get() 会惰性地启动 promise。

使用 std::async 实现惰性和立即求值

```
1 // asyncLazyEager.cpp
2 ...
3 #include <future>
4 ...
5 using std::chrono::duration;
6 using std::chrono::system_clock;
7 using std::launch;
8
9 auto begin= system_clock::now();
10
11 auto asyncLazy= std::async(launch::deferred, []{ return system_clock::now(); });
12 auto asyncEager= std::async(launch::async, []{ return system_clock::now(); });
13 std::this_thread::sleep_for(std::chrono::seconds(1));
14
15 auto lazyStart= asyncLazy.get() - begin;
16 auto eagerStart= asyncEager.get() - begin;
17
18 auto lazyDuration= duration<double>(lazyStart).count();
19 auto eagerDuration= duration<double>(eagerStart).count();
20
21 std::cout << lazyDuration << " sec"; // 1.00018 sec.
22 std::cout << eagerDuration << " sec"; // 0.00015489 sec.
```

程序的输出显示，与 future 的 asyncLazy 相关联的 promise 比与 future 的 asyncEager 相关联的 promise 晚一秒执行。一秒是创建者在 future asynlazy 请求其结果之前的休眠时间。

`std::async` 应该是首选

C++ 运行时确定 `std::async` 是否在单独的线程中执行。C++ 运行时的决定可能取决于内核的数量、系统的利用率或工作包的大小。

`std::packaged_task`

`std::packaged_task` 能够为可调用对象构建一个简单的包装器，稍后可以在单独的线程上执行。因此，四个步骤是必要的。

I. 打好工作包:

```
1 std::packaged_task<int(int, int)> sumTask([](int a, int b){ return a+b; });
```

II. 创建 future:

```
1 std::future<int> sumResult= sumTask.get_future();
```

III. 执行计算:

```
1 sumTask(2000, 11);
```

IV. 查询结果:

```
1 sumResult.get();
```

可以在单独的线程中移动 `std::package_task` 或 `std::future`。

`std::packaged_task`

```
1 // packaged_task.cpp
2 ...
3 #include <future>
4 ...
5
6 using namespace std;
7
8 struct SumUp{
9     int operator()(int beg, int end){
10         for (int i= beg; i < end; ++i) sum += i;
11         return sum;
12     }
13     private:
14     int beg;
15     int end;
16     int sum{0};
17 };
18
19 SumUp sumUp1, sumUp2;
20
```

```

21 packaged_task<int(int, int)> sumTask1(sumUp1);
22 packaged_task<int(int, int)> sumTask2(sumUp2);
23
24 future<int> sum1= sumTask1.get_future();
25 future<int> sum2= sumTask2.get_future();
26
27 deque< packaged_task<int(int, int)>> allTasks;
28 allTasks.push_back(move(sumTask1));
29 allTasks.push_back(move(sumTask2));
30
31 int begin{1};
32 int increment{5000};
33 int end= begin + increment;
34
35 while (not allTasks.empty()) {
36     packaged_task<int(int, int)> myTask= move(allTasks.front());
37     allTasks.pop_front();
38     thread sumThread(move(myTask), begin, end);
39     begin= end;
40     end += increment;
41     sumThread.detach();
42 }
43
44 auto sum= sum1.get() + sum2.get();
45 cout << sum; // 50005000

```

promise(std::packaged_task) 移动到 std::deque allTasks 中，程序在 while 循环中遍历所有 promise。每个 promise 在其线程中运行，并在后台执行其添加 (sumThread.detach())。结果是 1 到 100000 之间所有数字的和。

std::promise 和 std::future

std::promise 和 std::future 对提供了对任务的完全控制。

promise prom 的成员函数

成员函数	描述
prom.swap(prom2) std::swap(prom, prom2)	交换 promises。
prom.get_future()	返回 future。
prom.set_value(val)	设置值。
prom.set_exception(ex)	设置异常。
prom.set_value_at_thread_exit(val)	存储该值，并使其在 promise 退出时准备好。
prom.set_exception_at_thread_exit(ex)	存储异常，并使其在 promise 退出时准备好。

若 promise 不止一次设置值或异常，则抛出 std::future_error 异常。

future fut 的成员函数

成员函数	描述
fut.share()	返回 std::shared_future。
fut.get()	返回结果，该结果可以是一个值或异常。
fut.valid()	检查结果是否可用。调用 fut.get() 后返回 false。
fut.wait()	等待结果。
fut.wait_for(relTime)	等待结果的一段时间。
fut.wait_until(absTime)	等待结果的直到某个绝对时间点。

若 future fut 不止一次请求结果，则抛出 std::future_error 异常。future 可通过 fut.share() 创建共享 future。共享 future 与他们的 promise 相关联，并且可以独立地要求结果。共享 future 与 future 具有相同的接口。

下面是 promise 和 future 的用法。

promise 和 future

```
1 // promiseFuture.cpp
2 ...
3 #include <future>
4 ...
5
6 void product(std::promise<int>&& intPromise, int a, int b) {
7     intPromise.set_value(a*b);
8 }
9
10 int a= 20;
11 int b= 10;
12
13 std::promise<int> prodPromise;
14 std::future<int> prodResult= prodPromise.get_future();
15
16 std::thread prodThread(product, std::move(prodPromise), a, b);
17 std::cout << "20*10= " << prodResult.get(); // 20*10= 200
```

promise prodPromise 移动到一个单独的线程中，并执行它的计算。future 通过 prodResult.get() 获取结果。

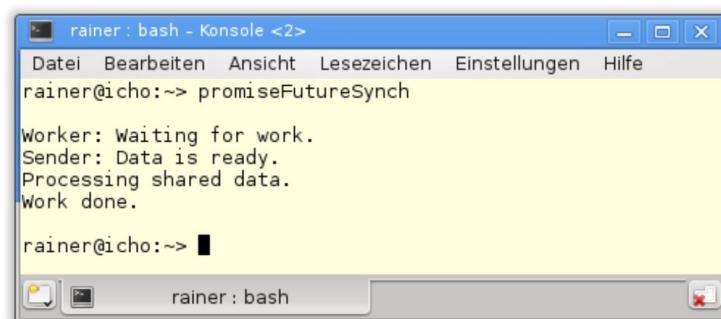
同步

future fut 可以通过调用 fut.wait() 与它关联的 promise 同步。与条件变量相反，不需要锁和互斥锁，并且不可能出现伪唤醒和未唤醒。

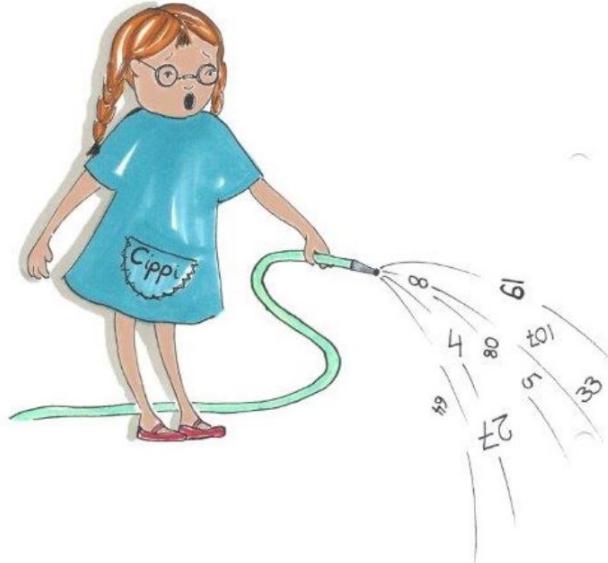
同步任务

```
1 // promiseFutureSynchronise.cpp
2 ...
3 #include <future>
4 ...
5
6 void doTheWork() {
7     std::cout << "Processing shared data." << '\n';
8 }
9
10 void waitingForWork(std::future<void>&& fut) {
11     std::cout << "Worker: Waiting for work." <<
12     '\n';
13     fut.wait();
14     doTheWork();
15     std::cout << "Work done." << '\n';
16 }
17
18 void setDataReady(std::promise<void>&& prom) {
19     std::cout << "Sender: Data is ready." <<
20     '\n';
21     prom.set_value();
22 }
23
24 std::promise<void> sendReady;
25 auto fut= sendReady.get_future();
26
27 std::thread t1(waitingForWork, std::move(fut));
28 std::thread t2(setDataReady, std::move(sendReady));
```

promise prom.set_value() 的调用唤醒 future，然后可以执行其工作。



第 20 章 协程



Cippi 在浇花

C++20 没有具体的协程，只有一个实现协同程序的框架。C++23 中，在范围库中有了第一个使用 `std::generator` 的具体协程。本章只展示一个具有挑战性的协程框架的大致概念。

协程是可以在保持状态的情况下暂停和恢复执行的函数。为了实现这一点，协程由三部分组成：`:promise` 对象、协程句柄和协程框架。

- `:promise` 对象在协程中被操作，并通过 `:promise` 对象返回结果。
- 协程句柄是一个非拥有的句柄，用于从外部恢复或销毁协程帧。
- 协程帧是一个内部的、典型的堆分配状态，由前面提到的 `:promise` 对象、协程的复制参数、挂起点的表示、声明周期在当前挂起点之前结束的局部变量，以及生命周期超过当前挂起点的局部变量组成。

优化协程的分配有两个必要条件：

使用关键字 `co_return` 而不是 `return`、`co_yield` 或 `co_await` 的函数隐式地成为协程。

新的关键字用两个新概念扩展了 C++ 函数的执行。

`co_yield` 表达式

允许编写生成器函数。生成器函数每次都返回一个新值。生成器函数是一个数据流，可以从中选择值。数据流可以无限。

生成器

```
1 Generator<int> getNext(int start = 0, int step = 1) noexcept {
2     auto value = start;
3     for (int i = 0;; ++i) {
4         co_yield value;
5         value += step;
```

```
6     }
7 }
```

co_await 表达式

挂起并恢复表达式的执行。若在函数 func 中使用 co_await 表达式，则若函数的结果不可用，则调用 auto getResult = func() 不会阻塞。而不是消耗资源的阻塞，是资源友好的等待。表达式必须是一个所谓的可等待表达式，并且必须支持以下三个函数 await_ready、await_suspend 和 await_resume。

一个协程

```
1 Acceptor acceptor{443};
2
3 while (true) {
4     Socket socket= co_await acceptor.accept();
5     auto request= co_await socket.read();
6     auto response= handleRequest(request);
7     co_await socket.write(response);
8 }
```

co_await expr 中的可等待表达式 expr 必须实现 await_ready、await_suspend 和 await_resume 函数。

20.1. 可等待

C++20 标准已经定义了两个可等待对象作为基本构建块:std::suspend_always 和 std::suspend_never。

预定义的可等待对象

```
1 struct suspend_always {
2     constexpr bool await_ready() const noexcept { return false; }
3     constexpr void await_suspend(coroutine_handle<>) const noexcept {}
4     constexpr void await_resume() const noexcept {}
5 };
6
7 struct suspend_never {
8     constexpr bool await_ready() const noexcept { return true; }
9     constexpr void await_suspend(coroutine_handle<>) const noexcept {}
10    constexpr void await_resume() const noexcept {}
11};
```

20.2. 无限数据流——co_yield

下面的程序产生一个无限数据流。协程 getNext 使用 co_yield 创建一个数据流，该数据流在开始时开始，并在请求时给出下一个值，按步递增。

无限数据流

```
1 // infiniteDataStream.cpp
2
3 ...
4
5 #include <coroutine>
6
7 template<typename T>
8 struct Generator {
9
10     struct promise_type;
11     using handle_type = std::coroutine_handle<promise_type>;
12
13     Generator(handle_type h) : coro(h) {} // (3)
14     handle_type coro;
15
16     ~Generator() {
17         if ( coro ) coro.destroy();
18     }
19     Generator(const Generator&) = delete;
20     Generator& operator = (const Generator&) = delete;
21     Generator(Generator&& oth) noexcept : coro(oth.coro) {
22         oth.coro = nullptr;
23     }
24     Generator& operator = (Generator&& oth) noexcept {
25         coro = oth.coro;
26         oth.coro = nullptr;
27         return *this;
28     }
29     T getValue() {
30         return coro.promise().current_value;
31     }
32     bool next() { // (5)
33         coro.resume();
34         return not coro.done();
35     }
36     struct promise_type {
37         promise_type() = default; // (1)
38
39         ~promise_type() = default;
40
41         auto initial_suspend() { // (4)
42             return std::suspend_always{};
43         }
44         auto final_suspend() noexcept {
45             return std::suspend_always{};
46         }
47     };
48 }
```

```

47     auto get_return_object() { // (2)
48         return Generator{handle_type::from_promise(*this)};
49     }
50
51     auto return_void() {
52         return std::suspend_never{};
53     }
54
55     auto yield_value(const T value) { // (6)
56         current_value = value;
57         return std::suspend_always{};
58     }
59
60     void unhandled_exception() {
61         std::exit(1);
62     }
63     T current_value;
64 };
65
66 Generator<int> getNext(int start = 0, int step = 1) {
67     auto value = start;
68     for (int i = 0;; ++i){
69         co_yield value;
70         value += step;
71     }
72 }
73
74 int main() {
75     std::cout << '\n';
76
77     std::cout << "getNext():";
78     auto gen = getNext();
79     for (int i = 0; i <= 10; ++i) {
80         gen.next();
81         std::cout << " " << gen.getValue(); // (7)
82     }
83
84     std::cout << '\n';
}

```

主程序创建一个协程。协同程序 gen(7) 返回从 0 到 10 的值。

getNext(): 0 1 2 3 4 5 6 7 8 9 10

无限数据流

程序 infiniteDataStream.cpp 中的数字代表工作流的第一次迭代。

1. 创建 promise

2. 调用 `promise.get_return_object()` 并将结果保存在一个局部变量中
3. 创建生成器
4. 调用 `promise.initial_suspend()`, 因为生成器是惰性的, 因此总是挂起。
5. 请求下一个值, 并在生成器使用时返回
6. 由 `co_yield` 调用触发, 下一个值随后可用。
7. 获取下一个值。

在剩余的迭代中, 只执行步骤 5 和 6。