C++20

The Complete Guide

Nicolai M. Josuttis

C++20 - The Complete Guide

作者: Nicolai M. Josuttis

译者：陈晓伟

版本：2022-10-30

目录

前言

C++20 是现代 C++ 的下一个版本，在最新版本的 GCC、Clang 和 Visual C++ 中已经 (部分地) 有所支持。了解 C++20 和了解 C++11 一样重要，C++20 包含了大量新特性和库，将再次改变 C++ 编程的方式。这既适用于应用程序开发者，也适用于基础库的开发者。

实践方式

本书从两个方面进行了实践:

- 我正在写一本深入实践的书，涵盖了不同开发者和 C++ 开发工作组，以及介绍了复杂的新特性，但我也会一些问题。
- 我自己在 Leanpub 上出版这本书，并按需印刷。这本书是逐步完成的，当有了重要的改进，就会出版新版本。

好的方面是:

- 读者们可以从资深应用开发者处了解语言特性的观点，资深的开发者能够感受到特性可能带来的挑战，并提出相关的问题，以便解释特性的动机、设计，以及可能产生的后果。
- 我也仍在学习和编写 C++20 的代码，读者们也可以从我的实践经验中获益。
- 本书的当前版本和目前的读者，都可以从早期版本反馈的问题中受益。

所以各位读者也是实践的一部分。所以请提供有关缺陷、错误、未解释清楚的功能或差距的反馈，以便大家都能从这些反馈中受益。

致谢

若没有他人的帮助和支持，这本书不可能完成。

首先，感谢 C++ 社区，让这本书成书成为可能。C++20 中所有特性的设计、反馈，以及好奇心都是是一门成功语言发展的基础。特别是对我所有的问题进行解释，以及给出相应的反馈。

我要特别感谢审阅过本书草稿或相应幻灯片的诸位，你们提供了宝贵的反馈和建议。这些评论大大提高了本书的质量，再次证明了好东西是许多人合作的结果。目前为止 (这个名单还在不断增长)，非常感谢 Carlos Buchart, Javier Estrada, Howard Hinnant, Yung-Hsiang Huang, Daniel Krügler, Dietmar Kühl, Jens Maurer, Paul Ranson, Thomas Symalla, Steve Vinoski, Ville Voutilainen. Andreas Weis, Hui Xie, Leor Zolman 和 Victor Zverovich。

另外，还要感谢 C++ 标准委员会的每一位成员。除了添加新语言和库功能的工作外，还花了很多很多时间与我解释和讨论他们的工作，为他们的耐心和热情点赞。在此特别感谢 Howard Hinnant, Tom Honermann, Tomasz Kaminski, Peter Sommerlad, Tim Song, Barry Revzin, Ville Voutilainen 和 Jonathan Wakely。

特别感谢 LaTeX 社区提供了一个很棒的文本系统，感谢 Frank Mittelbach 解决了关于的 LaTeX 的问题 (我太菜了)。

最后，非常感谢校对员 Tracey Duffy 的工作，将我的“德式英语”转换成地道的英语。

关于本书

本书介绍了 C++20 的所有新语言和库特性，通过示例和相关的背景信息可让读者对每个新特性的提出动机进行了解。

本书的重点在于新特性在实践中的应用，并演示了这些特性如何影响日常编程，以及如何在项目中从中受益。这既适用于应用程序开发者，也适用于提供通用框架和基础库开发者。

预备知识

读者们应该已经熟悉 C++，熟悉类和引用的概念，并且应该能够使用 C++ 标准库的组件 (如 IOStreams 和容器) 编写 C++ 代码。还应该熟悉现代 C++ 的基本特性，例如：auto 或基于范围的 for 循环，以及 C++11、C++14 和 C++17 引入的其他常用特性。

但不需要成为专家，我的目标是使普通 C++ 程序员可以理解这些内容，他们不一定知道所有的细节或所有的最新特性。本书只论基本特性，并根据需要了解更微妙的问题和情况。

这确保了专家级和中级开发者也适合阅读本书。

本书结构

本书涵盖了 C++20 引入的所有变化。既有语言和库的特性，也有影响日常应用程序编程的特性和 (基础) 库的复杂实现的特性，更常规的例子会首先给出。

章节分为不同的组，但别过分解读这里的分组，除了首先引入可能会被后续引入的功能所使用之外。原则上，可以按任意顺序阅读这些章节。若将不同章节的特征组合在一起，就会进行交叉引用。

如何阅读

别看这本书有些厚，与 C++ 一样，当深入研究细节时，事情会变得复杂。对于新特性的基本理解，介绍动机和示例通常就足够了。

根据我的经验，学习新东西的最好方法是看例子，大家会在书中找到很多例子。有些只是说明抽象概念的几行代码，而另一些则是提供材料具体应用的完整程序。后一种示例由描述包含程序代码的文件的 C++ 注释引入。读者们可以在这本书的网站http://www.cppstd20.com上找到这些文件。

执行方式

请注意以下关于我在本书中编写代码和注释的提示

初始化

我通常使用现代形式的初始化 (C++11 中作为统一初始化引入)，通常情况下使用花括号或 =:

```cpp
int i = 42;
std::string s{"hello"};
```

大括号初始化有以下优点:

- 可以与基本类型、类类型、聚合、枚举类型和 auto 一起使用
- 可用于初始化具有多个值的容器
- 可以检测窄化错误 (例如，用浮点值初始化 int)。
- 不可与函数声明或函数调用混淆

若大括号为空，则调用 (子) 对象的默认构造函数，并保证基本数据类型用 0/false/nullptr 初始化。

术语“错误”

我经常谈论编程错误，若没有特殊提示，则使用术语“错误”或注释，例如

```cpp
... // ERROR
```

表示编译时错误。相应的代码不应该编译 (使用符合标准的编译器)。

若使用术语“运行时错误”，则程序可能会编译，但行为不正确或导致未定义的行为 (因此，可能会或可能不会执行预期的操作)。

代码简化

我会用例子来解释所有的特性，但为了专注于教授的关键方面，可能会跳过代码的其他细节。

- 大多数情况下，我使用省略号 (“…”) 来表示缺少额外的代码。注意，在这里没有使用代码字体。若看到带有代码字体的省略号，则代码必须将这三个点作为语言特性 (例如“typename …”)。
- 头文件中，我通常跳过预处理器保护。所有的头文件都应该像下面这样写:

```cpp
#ifndef MYFILE_HPP
#define MYFILE_HPP
...
#endif // MYFILE_HPP
```

因此，在项目中使用这些头文件时，请补全代码。

C++ 标准

C++ 会根据不同的 C++ 标准定义的不同版本。

最初的 C++ 标准发布于 1998 年，随后在 2003 年通过技术勘误表进行了修订，该勘误表对原始标准进行了更正和澄清，“旧 C++ 标准”称为 C++98 或 C++03。

“现代 C++”是从 C++11 开始，并扩展到 C++14 和 C++17。国际 C++ 标准委员会现在的目标是每三年发布一个新标准，这使得进行大量添加的时间更少，但可以更快地为更广泛的编程社区带来改进。所以，开发更大的特性需要时间，并且可能涉及多个标准。

C++20 现在是下一个“现代的 C++”的开始，一些编程方式会有所改变。而编译器需要一些时间来提供最新的语言特性的支持。写这本书的时候，C++20 已经有部分得到了主流编译器的支持，但编译器在支持新的不同语言特性方面差异很大。有些人会编译本书中的大部分甚至全部代码，而另一些人可能只能够处理一个子集。我希望这个问题很快就会得到解决，因为各地的开发者都会要求供应商提供对相应标准的支持。

示例代码和附加信息

可以访问所有的示例程序，并从它的网站上找到更多关于这本书的信息:

http://www.cppstd20.com

反馈

欢迎诸位的建设性意见。我非常努力地编辑这本书，希望对你来说这是一本优秀的书。但有时，我不得不停止编写、审查和调整以“发布新版本”。因此可能会出现错误、不一致、可以改进的演示文稿或完全缺失的情况。您的反馈给了我一个机会来解决这些问题，通过本书的网站通知所有读者关于内容的修改，并改进后续的版本。

联系我的最好方式是发电子邮件，可以在这本书的网站上找到电子邮件地址:

http://www.cppstd20.com

若使用电子书，可能想要确保你有这本书的最新版本 (记住是增量编写和出版的)。在提交报告之前，还应该在书的网站上查看目前已知的勘误表。提供反馈时，请始终参考此版本的发布日期。目前的出版日期是 2022-10-30(也可以在封面后面的第二页看到日期)。

多谢诸位！

第 1 章 比较操作符 <=>

C++20 简化了用户定义类型的比较定义，并引入了更好的处理方法，引入了新的操作符 <=>(也称为太空船操作符)。

本章会介绍 C++20 如何使用这些新特性定义和处理比较操作 (符)。

1.1. 提出比较操作符 <=> 的动机

首先了解一下 C++20 处理比较的新方式和新操作符 <=> 提出的动机。

1.1.1 C++20 前如何定义比较运算符

C++20 之前，必须为一个类型定义六个操作符，以提供对象所有比较的支持。

例如，若要比较 Value 类型的对象 (具有整型 ID)，则须实现以下操作:

```
class Value {
private:
  long id;
  ...
public:
  ...
  // equality operators:
  bool operator== (const Value& rhs) const {
    return id == rhs.id; // basic check for equality
  }
  bool operator!= (const Value& rhs) const {
    return !(*this == rhs); // derived check
  }
  // relational operators:
  bool operator< (const Value& rhs) const {
    return id < rhs.id; // basic check for ordering
  }
  bool operator<= (const Value& rhs) const {
    return !(rhs < *this); // derived check
  }
  bool operator> (const Value& rhs) const {
    return rhs < *this; // derived check
  }
  bool operator>= (const Value& rhs) const {
    return !(*this < rhs); // derived check
  }
};
```

当使用另一个 Value(作为参数 rhs 传递) 调用 Value(为其定义操作符的对象) 的六个比较操作符中的一个。例如:

```
Value v1, v2;
... ;
if (v1 <= v2) { // calls v1.operator<=(v2)
  ...
}
```

操作符也可以间接调用 (例如，使用 sort()):

```
std::vector<Value> coll;
... ;
std::sort(coll.begin(), coll.end()); // uses operator < to sort
```

C++20 开始，可以使用 range:

```
std::ranges::sort(coll); // uses operator < to sort
```

问题是，尽管大多数操作符都是根据其他操作符定义的 (都基于 operator == 或 operator <)，但这些定义很繁琐，而且会增加很多阅读上的混乱。

此外，对于实现良好的类型，可能需要更多的声明:

- 若操作符不能抛出，就用 noexcept 声明
- 若操作符可以在编译时使用，则使用 constexpr 声明
- 若构造函数不是显式的，则将操作符声明为“隐藏友元”(在类结构中与友元一起声明，以便两个操作数都成为参数，并支持隐式类型转换)
- 声明带有 [[nodiscard]] 的操作符，以便在返回值未使用时强制发出警告

例如:

```
// lang/valueold.hpp

class Value {
private:
  long id;
  ...
public:
  constexpr Value(long i) noexcept // supports implicit type conversion
  : id{i} {
  }
  ...
  // equality operators:
  [[nodiscard]] friend constexpr
  bool operator== (const Value& lhs, const Value& rhs) noexcept {
    return lhs.id == rhs.id; // basic check for equality
  }
  [[nodiscard]] friend constexpr
  bool operator!= (const Value& lhs, const Value& rhs) noexcept {
```

```
    return !(lhs == rhs); // derived check for inequality
  }
  // relational operators:
  [[nodiscard]] friend constexpr
  bool operator< (const Value& lhs, const Value& rhs) noexcept {
    return lhs.id < rhs.id; // basic check for ordering
  }
  [[nodiscard]] friend constexpr
  bool operator<= (const Value& lhs, const Value& rhs) noexcept {
    return !(rhs < lhs); // derived check
  }
  [[nodiscard]] friend constexpr
  bool operator> (const Value& lhs, const Value& rhs) noexcept {
    return rhs < lhs; // derived check
  }
  [[nodiscard]] friend constexpr
  bool operator>= (const Value& lhs, const Value& rhs) noexcept {
    return !(lhs < rhs); // derived check
  }
};
```

1.1.2 C++20 后如何定义比较运算符

C++20 后，关于比较操作符的定义发生了一些变化。

== 与!= 操作符

为了检查是否相等，现在定义 == 操作符就够了。

当编译器找不到表达式的匹配声明 a!=b 时，编译器会重写表达式并查找!(a==b)。若这不起作用，编译器也会尝试改变操作数的顺序，所以也会尝试!(b==a):

```
a != b // tries: a!=b, !(a==b), and !(b==a)
```

因此，对于 TypeA 的 a 和 TypeB 的 b，编译器将能够识别并编译

```
a != b
```

若需要的话，可以这样做

- 一个独立函数 operator!=(TypeA, TypeB)
- 一个独立函数 operator==(TypeA, TypeB)
- 一个独立函数 operator==(TypeB, TypeA)
- 一个成员函数 TypeA::operator!=(TypeB)
- 一个成员函数 TypeA::operator==(TypeB)
- 一个成员函数 TypeB::operator==(TypeA)

直接调用已定义的 operator!= 是首选 (但类型的顺序必须合适)，更改操作数的顺序为最低的优先级。同时拥有独立函数和成员函数会出现歧义错误。

因此，

```
bool operator==(const TypeA&, const TypeB&);
```

或

```
class TypeA {
public:
  ...
  bool operator==(const TypeB&) const;
};
```

编译器可以进行编译:

```
MyType a;
MyType b;
...
a == b; // OK: fits perfectly
b == a; // OK, rewritten as: a == b
a != b; // OK, rewritten as: !(a == b)
b != a; // OK, rewritten as: !(a == b)
```

当重写将操作数转换为已定义成员函数的参数时，也可以对第一个操作数进行隐式类型转换。

请参阅示例 sentinel1.cpp，了解如何在调用具有不同顺序的操作数的!= 时仅定义成员操作符 ==。

<=> 操作符

对于所有的关系操作符，没有等价的规则说定义小于操作符就足够了。但现在，只需要定义新的操作符 <=> 即可。

事实上，以下内容足以让开发者使用所有可能的比较操作符:

```
// lang/value20.hpp

#include <compare>
class Value {
private:
  long id;
  ...
public:
  constexpr Value(long i) noexcept
  : id{i} {
  }
  ...
  // enable use of all equality and relational operators:
  auto operator<=> (const Value& rhs) const = default;
};
```

通常，== 可以通过定义 == 和!= 操作符来处理对象的相等性，而 <=> 操作符通过定义关系操作符来处理对象的顺序。若通过 =default 声明操作符 <=>，则可以使用了一个特殊的规则，即默认成员操作符 <=>:

```
class Value {
  ...
  auto operator<=> (const Value& rhs) const = default;
};
```

生成对应的成员 == 操作符，从而得到:

```
class Value {
  ...
  auto operator<=> (const Value& rhs) const = default;
  auto operator== (const Value& rhs) const = default; // implicitly generated
};
```

结果是两个操作符都使用了默认实现，逐个成员对象进行比较，所以类中成员的顺序很重要。因此，

```
class Value {
  ...
  auto operator<=> (const Value& rhs) const = default;
};
```

至此，我们得到了能够使用所有六个比较操作符所需的一切。

此外，即使将操作符声明为成员函数，也适用于生成的操作符:

- 若比较成员不抛出异常，则是 noexcept
- 若可在编译时比较成员，则是 constexpr
- 因为重写，还可以支持第一个操作数的隐式类型转换

通常情况下，== 和 <=> 操作符处理不同但相关的事情:

- == 操作符定义相等性，可由相等操作符 == 和!= 使用。
- <=> 操作符定义了排序，可以由关系操作符 <、<=、> 和 >= 使用。

注意，当默认或使用 <=> 操作符时，必须包含头文件 <compare>。

```
#include <compare>
```

不过，大多数标准类型 (字符串、容器、<utility>) 的头文件都会包含这个头文件。

<=> 操作符的实现

为了更好地控制生成的比较操作符，可以自己定义 == 和 <=> 操作符。例如:

```
// lang/value20def.hpp

#include <compare>
class Value {
private:
  long id;
  ...
public:
  constexpr Value(long i) noexcept
  : id{i} {
  }
  ...
  // for equality operators:
  bool operator== (const Value& rhs) const {
    return id == rhs.id; // defines equality (== and !=)
  }
  // for relational operators:
  auto operator<=> (const Value& rhs) const {
    return id <=> rhs.id; // defines ordering (<, <=, >, and >=)
  }
};
```

可以指定哪个成员以哪个顺序重要或实现特殊行为。

这些基本操作符的工作方式是，若表达式使用其中一个比较操作符，并且没有找到匹配的直接定义，则重写表达式，以便使用这些操作符。

与重写相等操作符调用相对应，重写也可能改变关系操作数的顺序，从而可能对第一个操作数启用隐式类型转换。例如：

```
x <= y
```

没有找到 <= 操作符的匹配定义，可以重写为

```
(x <=> y) <= 0
```

甚至

```
0 <= (y <=> x)
```

通过重写，<=> 操作符执行一个三向比较，生成一个可以与 0 比较的值:

- 若 x<=>y 的值等于 0，则 x 和 y 等于或相等。
- 若 x<=>y 小于 0，则 x 小于 y.
- 若 x<=>y 大于 0，则 x 大于 y。

但请注意，<=> 操作符的返回类型不是整数值。返回类型是表示比较类别的类型，可以是强排序、弱排序或偏排序，但这些类型支持与 0 进行比较。

1.2. 定义和使用

下面几节解释自 C++20 后比较操作符的详细情况。

1.2.1 使用 <=> 操作符

<=> 操作符是一个新的二元操作符。所有基本数据类型都有定义，对于其中定义了关系运算符的数据类型，也可以由用户自行重载定义。

<=> 操作符优先于所有其他比较操作符，所以需要在输出语句中使用括号，但不需要将其结果与其他值进行比较:

```
std::cout << (0 < x <=> y) << '\n'; // calls 0 < (x <=> y)
```

请注意，必须包含一个特定的头文件来处理 <=> 操作符的结果:

```
#include <compare>
```

这适用于声明 (默认情况下)、实现或使用。例如:

```
#include <compare> // for calling <=>

auto x = 3 <=> 4; // does not compile without header <compare>
```

大多数标准类型 (字符串、容器、<utility>) 的头文件都包含这个头文件，但要对不需要此头的值或类型使用操作符，必须包含 <compare>。

注意，<=> 操作符用于实现类型。在 <=> 操作符的实现之外，开发者不应该直接调用 <=>。虽然这样做没问题，但永远不要用 a<=>b < 0 来代替 a<b。

1.2.2 比较类别类型

<=> 操作符不返回布尔值，但其作用类似于三向比较，产生负值表示信号较少，正值表示信号较大，0 表示信号相等或等效。这种行为类似于 C 函数 strcmp() 的返回值，也有一个重要的区别: 返回值不是整数值，C++ 标准库提供了三种可能的返回类型，其反映了相应的比较类别。

比较类别

当比较两个值并按顺序排列时，有可能发生不同的“类别”行为:

- 对于**强排序** (也称为全排序)，给定类型的值都小于或等于或大于该类型的其他值 (包括其本身)。
  这一类的典型例子是整数值或常见的字符串类型，字符串 s1 小于等于或大于字符串 s2。
  若此类别的一个值既不小于也不大于另一个值，则两个值相等。若有多个对象，可以按升序或降序进行排序 (相等值的顺序任意)。

- 对于**弱排序**，给定类型的任何值都小于、等于或大于该类型的任何其他值 (包括其本身)，但相等的值不一定是相等的 (具有相同的值)。

  此类别的典型示例是用于不区分大小写的字符串的类型，字符串”hello” 小于”hello1” 且大于”hell”。尽管，”hello” 与”HELLO” 这两个字符串不相等但等价。

  若这个类别的值既不小于也不大于另一个值，那么这两个值至少是相等的 (甚至可能是相等的)。若有多个对象，则可以按升序或降序进行排序 (相等值的顺序可以任意)。
- 对于**偏排序**，给定类型的任何值都可以小于、等于或大于该类型的任何其他值 (包括其本身)，可能根本不能指定两个值之间的特定顺序。

  这种类型的典型示例是浮点类型，因为可能需要处理特殊值 NaN(“非数字”)，任何值与 NaN 的比较结果都为 false。因此，比较可能导致两个值是无序的，并且比较操作符可能返回四个值中的一个。

  若有多个对象，可能无法按升序或降序对其进行排序 (除非确保不存在无法排序的值)。

使用标准库比较类别

- std::strong_ordering 的值:
  - std::strong_ordering::less
  - std::strong_ordering::equal (也可是 std::strong_ordering::equivalent)
  - std::strong_ordering::greater
- std::weak_ordering 的值:
  - std::weak_ordering::less
  - std::weak_ordering::equivalent
  - std::weak_ordering::greater
- std::partial_ordering 的值:
  - std::partial_ordering::less
  - std::partial_ordering::equivalent
  - std::partial_ordering::greater
  - std::partial_ordering::unordered

注意，所有类型都有 less、greater 和 equivalent 值。但 strong_ordering 也有 equal，这与这里的 equal 相同，而 partial_ordering 的值为 unordered，既不表示小于，等于和大于。

较强比较类型具有向较弱比较类型的隐式类型转换，可以将 strong_ordering 值用作 weak_ordering 值，或 partial_ordering 值 (相等则为 equivalent)。

1.2.3 使用 <=> 操作符比较类别

<=> 操作符应该返回比较类别类型的值，表示比较结果，以及该结果是否能够创建强/全、弱或偏排序的信息。

例如，为 MyType 类型定义的 <=> 如下所示:

```
std::strong_ordering operator<=> (MyType x, MyOtherType y)
{
  if (xIsEqualToY) return std::strong_ordering::equal;
  if (xIsLessThanY) return std::strong_ordering::less;
  return std::strong_ordering::greater;
}
```

或者，作为一个更具体的例子，为 MyType 类型定义 <=> 操作符:

```
class MyType {
  ...
  std::strong_ordering operator<=> (const MyType& rhs) const {
    return value == rhs.value ? std::strong_ordering::equal :
          value < rhs.value ? std::strong_ordering::less :
                      std::strong_ordering::greater;
  }
};
```

通过将操作符映射到底层类型的结果，通常更容易定义操作符。所以，上面的成员 <=> 操作符最好只输出其成员值的值和类别:

```
class MyType {
  ...
  auto operator<=> (const MyType& rhs) const {
    return value <=> rhs.value;
  }
};
```

这不仅返回正确的值，还确保根据成员值的类型，返回值具有正确的比较类别类型。

1.2.4 直接调用 <=> 操作符

也可以直接使用已定义的 <=> 操作符:

```
MyType x, y;
...
x <=> y // yields a value of the resulting comparison category type
```

如前所述，实现 <=> 操作符时，应该只直接调用 <=> 操作符，了解返回的比较类别会非常有意义。

<=> 操作符是为定义了关系操作符的所有基本类型预定义的。例如:

```
int x = 17, y = 42;
x <=> y // yields std::strong_ordering::less
x <=> 17.0 // yields std::partial_ordering::equivalent
&x <=> &x // yields std::strong_ordering::equal
&x <=> nullptr // ERROR: relational comparison with nullptr not supported
```

此外，所有提供关系操作符的 C++ 标准库类型现在也提供了 <=> 操作符。例如:

```
std::string{"hi"} <=> "hi" // yields std::strong_ordering::equal;
std::pair{42, 0.0} <=> std::pair{42, 7.7} // yields std::partial_ordering::less
```

对于自己的类型，只需将 <=> 操作符定义为成员或独立函数即可。

因为返回类型取决于比较类别，所以可以根据特定的返回值进行检查:

```
if (x <=> y == std::partial_ordering::equivalent) // always OK
```

由于隐式类型转换为较弱排序类型，若 <=> 操作符产生 strong_ordering 或 weak_ordering 类型的值，这也是可以编译的。

反过来不行。若比较产生 weak_ordering 或 partial_ordering 类型的值，则不能将其与 strong_ordering 类型的值进行比较。

```
if (x <=> y == std::strong_ordering::equal) // might not compile
```

然而，与 0 的比较总是可能的，而且通常更容易理解:

```
if (x <=> y == 0) // always OK
```

此外，由于关系型操作符调用的新重写，<=> 操作符可以间接调用:

```
if (!(x < y || y < x)) // might call operator<=> to check for equality
```

或:

```
if (x <= y && y <= x) // might call operator<=> to check for equality
```

注意!= 操作符，永远不会重写为调用 <=> 操作符的方式，但可能调用由于默认操作符 <=> 成员隐式生成的操作符 ==。

1.2.5 处理多种排序条件

要基于多个属性计算运算符 <=> 的结果，通常可以实现一连串的子比较，直到结果不相等或到达要最终属性的比较:

```
class Person {
  ...
  auto operator<=> (const Person& rhs) const {
    auto cmp1 = lastname <=> rhs.lastname; // primary member for ordering
    if (cmp1 != 0) return cmp1; // return result if not equal
    auto cmp2 = firstname <=> rhs.firstname; // secondary member for ordering
    if (cmp2 != 0) return cmp2; // return result if not equal
    return value <=> rhs.value; // final member for ordering
  }
};
```

但若属性具有不同的比较类别，则返回类型不会编译。例如，若成员名是 string 类型，而成员值是 double 类型，则返回类型会冲突:

```
class Person {
  std::string name;
  double value;
  ...
  auto operator<=> (const Person& rhs) const { // ERROR: different return types deduced
    auto cmp1 = name <=> rhs.name;
    if (cmp1 != 0) return cmp1; // return strong_ordering for std::string
    return value <=> rhs.value; // return partial_ordering for double
  }
};
```

可以使用到最弱比较类型的转换。若已知最弱的比较类型，可以声明为返回类型:

```
class Person {
  std::string name;
  double value;
  ...
  std::partial_ordering operator<=> (const Person& rhs) const { // OK
    auto cmp1 = name <=> rhs.name;
    if (cmp1 != 0) return cmp1; // strong_ordering converted to return type
    return value <=> rhs.value; // partial_ordering used as the return type
  }
};
```

若不知道比较类型 (例如，类型是模板形参)，可以使用新的类型特征 std::common_comparison_category<> 来计算最强的比较类型:

```
class Person {
  std::string name;
  double value;
  ...
  auto operator<=> (const Person& rhs) const // OK
  -> std::common_comparison_category_t<decltype(name <=> rhs.name),
                       decltype(value <=> rhs.value)> {
    auto cmp1 = name <=> rhs.name;
    if (cmp1 != 0) return cmp1; // used as or converted to common comparison type
    return value <=> rhs.value; // used as or converted to common comparison type
  }
};
```

通过使用尾部返回类型语法 (auto 在前面，返回类型在-> 后面)，可以使用参数来计算比较类型。这种情况下，可以只使用 name，而非 rhs.name，这种方法通常是有效的 (例如，也适用于独立函数)。

若希望提供比内部使用的类别更强的类别，则必须将内部比较的所有可能值映射到返回类型的值。若不能映射某些值，这可能包括一些错误处理。例如:

```
class Person {
  std::string name;
  double value;
  ...
  std::strong_ordering operator<=> (const Person& rhs) const {
    auto cmp1 = name <=> rhs.name;
    if (cmp1 != 0) return cmp1; // return strong_ordering for std::string
    auto cmp2 = value <=> rhs.value; // might be partial_ordering for double
    // map partial_ordering to strong_ordering:
    assert(cmp2 != std::partial_ordering::unordered); // RUNTIME ERROR if unordered
    return cmp2 == 0 ? std::strong_ordering::equal
                     : cmp2 > 0 ? std::strong_ordering::greater
                                : std::strong_ordering::less;
  }
};
```

C++ 标准库为此提供了一些辅助函数对象。例如，映射浮点值，可以使用 std::strong_order() 来比较两个值:

```
class Person {
  std::string name;
  double value;
  ...
  std::strong_ordering operator<=> (const Person& rhs) const {
    auto cmp1 = name <=> rhs.name;
    if (cmp1 != 0) return cmp1; // return strong_ordering for std::string
    // map floating-point comparison result to strong ordering:
    return std::strong_order(value, rhs.value);
  }
};
```

如若可能，std::strong_order() 会根据传入的参数产生一个 std::strong_ordering 值，如下所示:

- 对传递的类型使用 strong_order(val1, val2)(若已定义)
- 否则，若传递的值是浮点类型，则使用 ISO/IEC/IEEE 60559 中指定的 totalOrder() 的值 (例如，-0 小于 +0，-NaN 小于任何非 NaN 值和 +NaN)
- 若是为传递的类型定义的，可使用新的函数对象 std::compare_three_way{}(val1, val2)，

这为浮点类型提供强排序的最简单方法，即使在运行时，若一个或两个操作数可能具有值 NaN，这种方法也可以工作。

std::compare_three_way 是用于调用 <=> 操作符的新函数对象类型，就像 std::less 是用于小于操作符的函数对象类型一样。

对于其他具有较弱排序和 == 和 < 操作符定义的类型，可以使用函数对象 std::compare_strong_order_fallback():

```
class Person {
  std::string name;
```

```
  SomeType value;
  ...
  std::strong_ordering operator<=> (const Person& rhs) const {
    auto cmp1 = name <=> rhs.name;
    if (cmp1 != 0) return cmp1; // return strong_ordering for std::string
    // map weak/partial comparison result to strong ordering:
    return std::compare_strong_order_fallback(value, rhs.value);
  }
};
```

用于映射比较类别类型的表，函数对象列出了用于映射比较类型的所有的辅助函数。

要为泛型类型定义 <=> 操作符，还应该考虑使用函数对象 std::compare_three_way 或算法 std::lexicographical_compare_three_way()。

1.3. 定义 <=> 和 == 操作符

自定义数据类型都可以定义 <=> 和 == 操作符:

- 要么作为带一个参数的成员函数
- 或者一个独立的函数，有两个参数

std:: 中的函数对象	**效果**
strong_order()	映射到强序值也适用于浮点值
weak_order()	映射到弱序值，也适用于浮点值
partial_order()	映射到偏序值
compare_strong_order_fallback()	映射到强序值，即使只定义了 == 和 < 操作符
compare_weak_order_fallback()	映射到弱序值，即使只定义了 == 和 < 操作符
compare_partial_order_fallback()	映射到偏序值，即使只定义了 == 和 < 操作符

表 1.1 用于映射比较类别类型的函数对象

1.3.1 默认操作符 == 和 <=>

在类或数据结构中 (作为成员或友元函数)，可以使用 =default 将所有比较操作符声明为默认值，但这通常只对 == 和 <=> 操作符有意义。成员函数必须接受第二个形参作为 const 左值引用 (const &)，友元函数也可以按值接受这两个形参。

默认操作符需要成员和可能的基类的支持:

- 默认操作符 == 要求在成员和基类中支持 ==。
- 默认操作符 <=> 要求在成员和基类中支持 == 和已实现的小于操作符或默认操作符 <=>(详细信息请参见下文)。

对于生成的默认操作符，有两点要提一下:

- 若比较成员保证不抛出异常，则操作符为 noexcept 型。

- 若可以在编译时比较成员，则操作符为 constexpr 型。

对于空类，默认操作符比较所有对象为相等:==、<= 和 >= 生成 true，!=、< 和 > 生成 false，而 <=> 生成 std::strong_ordering::equal。

1.3.2 默认 <=> 操作符实现默认 == 操作符

当操作符 <=> 成员定义为默认值时，若没有提供默认操作符 ==，则根据定义也定义相应的操作符 == 成员，可采用相应的方式 (可见性、虚拟、属性、需求等)。例如:

```
template<typename T>
class Type {
  ...
  public:
    [[nodiscard]] virtual std::strong_ordering
      operator<=>(const Type&) const requires(!std::same_as<T,bool>) = default;
};
```

等价于:

```
template<typename T>
class Type {
  ...
  public:
  [[nodiscard]] virtual std::strong_ordering
    operator<=> (const Type&) const requires(!std::same_as<T,bool>) = default;

  [[nodiscard]] virtual bool
    operator== (const Type&) const requires(!std::same_as<T,bool>) = default;
};
```

例如，下面的代码足以支持 Coord 类型对象的所有六种比较操作符:

lang/coord.hpp

```
#include <compare>
struct Coord {
  double x{};
  double y{};
  double z{};
  auto operator<=>(const Coord&) const = default;
};
```

成员函数必须是 const，形参必须声明为 const 左值引用 (const &)。

可以这样使用数据结构:

lang/coord.cpp

```cpp
#include "coord.hpp"
#include <iostream>
#include <algorithm>
int main()
{
  std::vector<Coord> coll{ {0, 5, 5}, {5, 0, 0}, {3, 5, 5},
                {3, 0, 0}, {3, 5, 7} };

  std::sort(coll.begin(), coll.end());
  for (const auto& elem : coll) {
    std::cout << elem.x << '/' << elem.y << '/' << elem.z << '\n';
  }
}
```

该程序应有如下的输出:

```
0/5/5
3/0/0
3/5/5
3/5/7
5/0/0
```

1.3.3 默认操作符 <=> 的实现

若 <=> 操作符是默认的，并且有成员或基类，并且调用关系操作符之一，则会发生以下情况:

- 若为成员或基类定义了操作符 <=>，则调用该操作符。
- 否则，使用 == 和 < 操作符来决定 (从成员或基类的角度)
  - 对象相等/等价 (运算符 == 产生 true)
  - 对象有“大”有“小”
  - 对象无序 (仅在检查偏排序时)

这种情况下，这些操作符的默认 <=> 操作符的返回类型不能为 auto。

例如，以下声明:

```cpp
struct B {
  bool operator==(const B&) const;
  bool operator<(const B&) const;
};

struct D : public B {
  std::strong_ordering operator<=> (const D&) const = default;
};
```

然后:

```
D d1, d2;
d1 > d2; // calls B::operator== and possibly B::operator<
```

若操作符 == 的结果为 true，则知道 > 的结果为 false，仅此而已。否则，使用操作符 < 来判断表达式是 true 还是 false。

```
struct D : public B {
  std::partial_ordering operator<=> (const D&) const = default;
};
```

编译器甚至可能两次调用小于操作符来确定是否有序。

```
struct B {
  bool operator==(const B&) const;
  bool operator<(const B&) const;
};

struct D : public B {
  auto operator<=> (const D&) const = default;
};
```

编译器不会编译带有关系操作符的调用，无法确定基类具有哪个排序类别在这种情况下，基类中也需要 <=> 操作符。

然而，检查相等性是有效的，在 D 中 == 操作符声明为等价于以下内容的代码:

```
struct D : public B {
  auto operator<=> (const D&) const = default;
  bool operator== (const D&) const = default;
};
```

当有以下行为:

```
D d1, d2;
d1 > d2; // ERROR: cannot deduce comparison category of operator<=>
d1 != d2; // OK (note: only tries operator<=> and B::operator== of a base class)
```

相等性检查总是只使用基类的操作符 ==(但可以根据默认操作符 <=> 生成)，可忽略基类中的 < 或!= 运算符。

同样地，若 D 中有 B 类型的成员。

1.4. 重载解析与重写表达式

最后，介绍一下如何使用重写支持的比较操作符，对表达式进行求值。

相等操作符

编译

```
x != y
```

编译器现在可能会尝试以下的操作:

```
x.operator!=(y) // calling member operator!= for x
operator!=(x, y) // calling a free-standing operator!= for x and y

!x.operator==(y) // calling member operator== for x
!operator==(x, y) // calling a free-standing operator== for x and y

!x.operator==(y) // calling member operator== generated by operator<=> for x

!y.operator==(x) // calling member operator== generated by operator<=> for y
```

最后一种形式试图支持第一个操作数的隐式类型转换，这要求操作数是一个参数。
通常，编译器会尝试调用:

- 操作符!=: 操作符!=(x, y) 或成员操作符!=:x.operator!=(y)
  同时定义两个操作符!= 会导致歧义错误。
- 操作符 ==:!operator==(x, y) 或成员 operator==: !x.operator==(y)
  注意，成员 == 操作符可以使用默认操作符 <=> 成员生成。
  同样，同时定义两个操作符 == 会导致歧义错误，这也适用于成员函数。
  == 操作符是由于默认操作符 <=> 生成。

当需要对第一个操作数 v 进行隐式类型转换时，编译器还会尝试对操作数重新排序:

```
42 != y // 42 implicitly converts to the type of y
```

编译器会按顺序调用:

- 独立或成员操作符!=
- 独立的或成员操作符 ==(注意，成员操作符 == 可以使用默认的 <=> 成员生成)[最初的 C++20 标准在http://wg21.link/p2468r2中进行了修复。]

重写表达式永远不会调用成员操作符!=。

关系操作符

对于关系操作符，有类似的行为，只是重写的语句返回到新的操作符 <=> 并将结果与 0 进行比较。操作符的行为类似于一个三方比较函数，表示 less 时返回负值，表示 equal 时返回 0，表示 greater 时返回正值 (返回值不是数值，只是一个支持相应比较的值)。

编译

```
x <= y
```

编译器现在会尝试以下的操作:

```
x.operator<=(y) // calling member operator<= for x
operator<=(x, y) // calling a free-standing operator<= for x and y

x.operator<=>(y) <= 0 // calling member operator<=> for x
operator<=>(x, y) <= 0 // calling a free-standing operator<=> for x and y

0 <= y.operator<=>(x) // calling member operator<=> for y
```

同样，最后一种形式试图支持第一个操作数的隐式类型转换，所以相应的值必须成为第一个操作数的参数。

1.5. 泛型代码中使用 <=>

泛型代码中，使用 <=> 会有一些挑战。可能存在提供操作符 <=> 的类型，也可能存在提供偏或全基本比较操作符的类型。

1.5.1 compare_three_way

std::compare_three_way 是用于调用 <=> 的函数对象类型，就像 std::less 是用于调用小于操作符的函数对象类型一样。

可以这样使用:

- 比较泛型类型的值
- 必须指定函数对象的类型时作为默认类型

例如：

```
template<typename T>
struct Value {
  T val{};
  ...
  auto operator<=> (const Value& v) const noexcept(noexcept(val<=>val)) {
    return std::compare_three_way{}(val<=>v.val);
  }
};
```

使用 std::compare_three_way(与 std::less 一样) 的好处是，定义了原始指针的总序 (操作符 <=> 或 < 不是这种情况)，所以当使用是原始指针类型的泛型类型时可以使用。

允许开发者前向声明 operator<=>()，C++20 还引入了类型特性 std::compare_three_way_result 和别名模板 std::compare_three_way_result_t:

```
template<typename T>
struct Value {
  T val{};
  ...
  std::compare_three_way_result_t<T,T>
    operator<=> (const Value& v) const noexcept(noexcept(val<=>val));
};
```

1.5.2 算法 lexicographical_compare_three_way()

为了能够比较两个范围并产生匹配比较类别的值，C++20 还引入了 lexicographical_compare_three_way() 算法。该算法对于为集合成员实现 <=> 操作符特别有帮助。

例如:

lib/lexicothreeway.cpp

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

std::ostream& operator<< (std::ostream& strm, std::strong_ordering val)
{
  if (val < 0) return strm << "less";
  if (val > 0) return strm << "greater";
  return strm << "equal";
}

int main()
{
  std::vector v1{0, 8, 15, 47, 11};
  std::vector v2{0, 15, 8};

  auto r1 = std::lexicographical_compare(v1.begin(), v1.end(),
                       v2.begin(), v2.end());

  auto r2 = std::lexicographical_compare_three_way(v1.begin(), v1.end(),
                           v2.begin(), v2.end());

  std::cout << "r1: " << r1 << '\n';
  std::cout << "r2: " << r2 << '\n';
}
```

该程序应有以下输出:

```
r1: 1
r2: less
```

注意，lexicographical_compare_three_way() 还没有支持范围。既不能将范围作为单个参数传递，也不能传递投射形参:

```
auto r3 = std::ranges::lexicographical_compare(v1, v2); // OK
auto r4 = std::ranges::lexicographical_compare_three_way(v1, v2); // ERROR
```

1.6. 比较运算符的兼容性

引入了新的比较规则之后，事实证明 C++20 引入了一些问题，当从旧 C++ 版本进行切换时，这些问题就会暴露。

1.6.1 委托独立比较操作符

下面的例子演示了最典型问题:

lang/spacecompat.cpp

```
#include <iostream>

class MyType {
private:
  int value;
public:
  MyType(int i) // implicit constructor from int:
    : value{i} {
  }
  bool operator==(const MyType& rhs) const {
    return value == rhs.value;
  }
};

bool operator==(int i, const MyType& t) {
  return t == i; // OK with C++17
}
int main()
{
  MyType x = 42;
  if (0 == x) {
    std::cout << "'0 == MyType{42}' works\n";
  }
}
```

有一个简单的类，存储整型值，并有一个隐式构造函数来初始化对象 (隐式构造函数是支持使用 = 初始化的必要条件):

```
class MyType {
  public:
```

```
  MyType(int i); // implicit constructor from int
  ...
};

MyType x = 42; // OK
```

类还声明了一个成员函数来比较对象:

```
class MyType {
  ...
  bool operator==(const MyType& rhs) const;
  ...
};
```

但 C++20 之前，这只允许对第二个操作数进行隐式类型转换，所以类或其他代码引入了一个全局操作符来交换参数的顺序:

```
bool operator==(int i, const MyType& t) {
  return t == i; // OK until C++17
}
```

对于类来说，最好将 operator==() 定义为“隐藏的友元”(在类结构中使用友元定义它，以便两个操作符都成为参数，可以直接访问成员，并且只有在至少有一个参数类型适合时才执行隐式类型转换)，但上面的代码在 C++20 之前具有相同的效果。

不幸的是，这段代码在 C++20 中不再可用，[感谢 Peter Dimov 和 Barry Revzin 指出了这个问题，并在http://stackoverflow.com/questions/65648897上进行了讨论] 其导致了无尽的递归。在全局函数内部，因为表达式 t == i，编译器会试图将调用重写为 i == t，所以也可以调用全局 operator==() 本身:

```
bool operator==(int i, const MyType& t) {
  return t == i; // finds operator==(i,t) in addition to t.operator(MyType{i})
}
```

因为不需要隐式类型转换，所以重写的语句是更好的匹配项。我们目前还没有解决方案来支持这里的向后兼容性，但编译器已经开始对这样的代码发出警告了。

若的代码只使用 C++20，则可以简单地删除独立函数。否则，就只有两个选择:

- 使用显式转换:

```
bool operator==(int i, const MyType& t) {
  return t == MyType{i}; // OK until C++17 and with C++20
}
```

- 新特性可用时，使用特性测试宏来禁用代码:

1.6.2 protected 成员的继承

对于具有默认比较操作符的派生类，若基类将操作符作为 protected 成员，则可能会出现问题。

注意，默认的比较操作符需要在基类中支持比较，以下操作不起作用:

```
struct Base {
};

struct Child : Base {
  int i;
  bool operator==(const Child& other) const = default;
};

Child c1, c2;
...
c1 == c2; // ERROR
```

还必须在基类中提供默认操作符 ==，但若不希望基类为 public 提供 == 操作符，那么可在基类中添加 protected 的默认操作符 ==:

```
struct Base {
  protected:
  bool operator==(const Base& other) const = default;
};
struct
Child : Base {
  int i;
  bool operator==(const Child& other) const = default;
};
```

这时，派生类的默认比较不起作用。其会由当前实现默认操作符的指定行为的编译器拒绝，这里的默认操作符过于严格。这个问题有望很快得到解决 (参见http://wg21.link/cwg2568)。

作为一种解决方法，开发者必须自己实现派生操作符。

1.7. 附注

Oleg Smolsky 对隐式定义比较运算符的要求描述在http://wg21.link/n3950中，Bjarne Stroustrup 随后在http://wg21.link/n4175上再次提出了这个问题。然而，Jens Maurer 在http://wg21.link/p0221r2中提出终版提议最终没采纳，因为它是一个选择退出特性 (因此，现有类型将自动具有比较操作符，除非声明不需要比较操作符)。然后考虑了其他各种建议，Herb Sutter 在http://wg21.link/p0515r0上汇集了这些建议。

最终接受的方案是由 Herb Sutter、Jens Maurer 和 Walter E. Brown 在http://wg21.link/p0515r3和http://wg21.link/p0768r1中提出。并且，Barry Revzin 在http://wg21.link/p1185r2, http://wg21.link/p1186r3, http://wg21.link/p1614r2和http://wg21.link/p1630r1中进行了大量的修改，从而才采纳。

第 2 章 函数参数的占位符类型

泛型编程是 C++ 的关键特性，C++20 还为泛型编程提供了几个新特性，其中有一个扩展在本书中或多或少会用到的: 现在可以使用 auto 和其他占位符类型来声明普通函数的参数。

后面的章节将介绍更多的通用扩展:

- 非类型模板参数的扩展
- Lambda 模板

2.1. auto 为普通函数参数

C++14 起，Lambda 可以使用 auto 等占位符来声明/定义其参数:

```
auto printColl = [] (const auto& coll) { // generic lambda
  for (const auto& elem : coll) {
    std::cout << elem << '\n';
  }
}
```

只要支持 Lambda 内部的操作，占位符允许传递任何类型的参数:

```
std::vector coll{1, 2, 4, 5};
...
printColl(coll); // compiles the lambda for vector<int>

printColl(std::string{"hello"}); // compiles the lambda for std::string
```

C++20 起，可以对所有函数 (包括成员函数和操作符) 使用 auto 等占位符:

```
void printColl(const auto& coll) // generic function
{
  for (const auto& elem : coll) {
    std::cout << elem << '\n';
  }
}
```

声明只是声明模板的快捷方式:

```
template<typename T>
void printColl(const T& coll) // equivalent generic function
{
  for (const auto& elem : coll) {
    std::cout << elem << '\n';
  }
}
```

使用 auto 替代模板参数 T，所以该特性也称为“函数模板化”的语法。

因为带有 auto 的函数是函数模板，所以适用于函数模板的所有规则。从而不能在一个翻译单元 (CPP 文件) 中实现带有自动参数的函数，并在另一个翻译单元中调用。对于具有 auto 参数的函数，实现存在于头文件，以便可以在多个 CPP 文件中使用 (否则，必须在一个翻译单元中显式地实例化函数)。另一方面，因为函数模板总是内联，所以不需要声明为内联。

另外，可以显式地指定模板参数:

```
void print(auto val)
{
  std::cout << val << '\n';
}

print(64); // val has type int
print<char>(64); // val has type char
```

2.1.1 auto 用于成员函数的参数

可以使用这个特性来定义成员函数:

```
class MyType {
  ...
  void assign(const auto& newVal);
};
```

该声明等价于 (不同之处在于没有定义类型 T):

```
class MyType {
  ...
  template<typename T>
  void assign(const T& newVal);
};
```

注意，模板不能在函数内部声明。对于使用 auto 形参的成员函数，不能再在函数内部局部定义类或数据结构:

```
void foo()
{
  struct Data {
    void mem(auto); // ERROR can't declare templates inside functions
  };
}
```

有关带有 auto 的成员 == 操作符的示例，请参阅 sentinel1.cpp。

2.2. 使用 auto 作为参数

参数使用 auto 有什么好处呢？

2.2.1 进行延迟类型检查

使用 auto 参数，实现具有循环依赖关系的代码会容易得多。

考虑两个使用其他类的对象的类。要使用另一个类的对象，需要定义它们的类型，前置声明是不够的 (除非只声明引用或指针):

```
class C2; // forward declaration

class C1 {
  public:
  void foo(const C2& c2) const { // OK
    c2.print(); // ERROR: C2 is incomplete type
  }

  void print() const;
};

class C2 {
  public:
  void foo(const C1& c1) const {
    c1.print(); // OK
  }
  void print() const;
};
```

虽然可以在类定义中实现 C2::foo()，但不能实现 C1::foo()，因为要检查 C2.print() 的调用是否有效，编译器需要知道类 C2 的定义。

必须在声明两个类结构之后实现 C2::foo():

```
class C2;

class C1 {
public:
  void foo(const C2& c2) const; // forward declaration
  void print() const;
};

class C2 {
public:
  void foo(const C1& c1) const {
    c1.print(); // OK
  }
  void print() const;
```

```
};

inline void C1::foo(const C2& c2) const { // implementation (inline if in header)
  c2.print(); // OK
}
```

泛型函数在调用时检查泛型形参的成员，所以可以使用 auto 实现:

```
class C1 {
  public:
  void foo(const auto& c2) const {
    c2.print(); // OK
  }

  void print() const;
};

class C2 {
  public:
  void foo(const auto& c1) const {
    c1.print(); // OK
  }
  void print() const;
};
```

将 C1::foo() 声明为成员函数模板时也会有同样的效果，但使用 auto 会更容易一些。

注意，auto 允许调用者传递任何类型的参数，只要该类型提供成员函数 print()。若不想这样，可以使用标准概念 std::same_as 来约束这个成员函数只对 C2 类型的形参使用:

```
#include <concepts>

class C2;

class C1 {
  public:
  void foo(const std::same_as<C2> auto& c2) const {
    c2.print(); // OK
  }
  void print() const;
};
...
```

对于概念来说，不完整的类型可以工作得很好。

2.2.2 auto 函数与 Lambda 表达式

带有 auto 参数的函数不同于 Lambda 表达式。例如，不能在不指定泛型参数的情况下，传递带有 auto 作为参数的函数:

```
bool lessByNameFunc(const auto& c1, const auto& c2) { // sorting criterion
  return c1.getName() < c2.getName(); // - compare by name
}
...
std::sort(persons.begin(), persons.end(),
      lessByNameFunc); // ERROR: can't deduce type of parameters in sorting criterion
```

记住，lessByName() 的声明相当于:

```
template<typename T1, typename T2>
bool lessByNameFunc(const T1& c1, const T2& c2) { // sorting criterion
  return c1.getName() < c2.getName(); // - compare by name
}
```

因为没有直接调用函数模板，编译器不能推断模板参数来编译调用。所以在传递函数模板作为形参时，必须显式指定模板的形参:

```
std::sort(persons.begin(), persons.end(),
      lessByName<Customer, Customer>); // OK
```

使用 Lambda 表达式，可以按原样传递表达式:

```
auto lessByNameLambda = [] (const auto& c1, const auto& c2) { // sorting criterion
              return c1.getName() < c2.getName(); // compare by name
            };
...
std::sort(persons.begin(), persons.end(),
      lessByNameLambda); // OK
```

Lambda 是一个没有泛型类型的对象，所以只有将对象作为函数使用才可通用。

另外，显式指定 (缩写) 函数模板参数更容易:

- 只需在函数名后传递指定的类型:

```
void printFunc(const auto& arg) {
  ...
}

printFunc<std::string>("hello"); // call function template compiled for
↪  std::string
```

- 对于泛型 Lambda 表达式，必须完成以下操作:

```
auto printFunc = [] (const auto& arg) {
  ...
};

printFunc.operator()<std::string>("hello"); // call lambda compiled for
↪  std::string
```

对于泛型 Lambda 表达式，运算符 operator() 为模板。所以必须将请求的类型作为参数传递给 operator()，以显式指定模板形参。

2.3. auto 作为参数的详情

再来详细了解一下简化函数模板。

2.3.1 auto 参数的限制

使用 auto 声明函数形参遵循与声明 Lambda 表达式形参相同的规则:

- 对于用 auto 声明的每个形参，函数都有隐式模板形参。
- 参数可以是参数包:

```
void foo(auto... args);
```

这相当于以下内容 (不引入类型):

```
template<typename... Types>
void foo(Types... args);
```

- 不允许使用 decltype(auto)。

简化的函数模板可以用 (部分地) 显式指定的模板参数调用，模板参数的顺序与调用参数相同。例如:

```
void foo(auto x, auto y)
{
  ...
}

foo("hello", 42); // x has type const char*, y has type int
foo<std::string>("hello", 42); // x has type std::string, y has type int
foo<std::string, long>("hello", 42); // x has type std::string, y has type long
```

2.3.2 模板参数和 auto 参数的组合

简化的函数模板仍然可以显式指定模板参数，为占位符类型生成的模板参数可添加到指定参数之后:

```
template<typename T>
void foo(auto x, T y, auto z)
{
  ...
}

foo("hello", 42, '?'); // x has type const char*, T and y are int, z is char
foo<long>("hello", 42, '?'); // x has type const char*, T and y are long, z is char
```

因此，以下声明是等价的 (除了没有为使用 auto 的类型命名):

```
template<typename T>
void foo(auto x, T y, auto z);

template<typename T, typename T2, typename T3>
void foo(T2 x, T y, T3 z);
```

稍后会介绍，通过使用概念作为类型约束，可以约束占位符参数和模板参数，可再使用模板参数来进行这样的限定。

例如，下面的声明确保第二个形参 y 是整型，而第三个形参 z 的类型可以转换为 y 的类型:

```
template<std::integral T>
void foo(auto x, T y, std::convertible_to<T> auto z)
{
  ...
}

foo(64, 65, 'c'); // OK, x is int, T and y are int, z is char
foo(64, 65, "c"); // ERROR: "c" cannot be converted to type int (type of 65)
foo<long,char>(64, 65, 'c'); // NOTE: x is char, T and y are long, z is char
```

注意，最后一条语句以错误的顺序指定了参数类型。

模板参数的顺序不像预期的那样的话，可能会导致错误:

lang/tmplauto.cpp

```
#include <vector>
#include <ranges>

void addValInto(const auto& val, auto& coll)
{
  coll.insert(val);
}

template<typename Coll> // Note: different order of template parameters
requires std::ranges::random_access_range<Coll>
void addValInto(const auto& val, Coll& coll)
{
  coll.push_back(val);
}

int main()
{
  std::vector<int> coll;
  addValInto(42, coll); // ERROR: ambiguous
}
```

addValInto() 的第二次声明中只对第一个参数使用 auto，所以模板参数的顺序不同。C++20 接受http://wg21.link/p2113r0，因为重载解析不会优先于第二个声明，而非第一个声明，所以会得到一个歧义错误。[并不是所有的编译器都能正确处理这个问题。]

因此，在混合模板参数和自动参数时要谨慎。理想情况下，与声明保持一致。

2.4. 附注

对于普通函数的参数 auto 首先是由 Ville Voutilainen, Thomas Köppe, Andrew Sutton, Herb Sutter, Gabriel Dos Reis, Bjarne Stroustrup, Jason Merrill, Hubert Tong, Eric Niebler, Casey Carter, Tom Honermann 和 Erich Keane 在http://wg21.link/p1141r0中提出，并选择使用类型约束。最终接受的提案是由 Ville Voutilainen、Thomas Köppe、Andrew Sutton、Herb Sutter、Gabriel Dos Reis、Bjarne Stroustrup、Jason Merrill、Hubert Tong、Eric Niebler、Casey Carter、Tom Honermann、Erich Keane、Walter E. Brown、Michael Spertus 和 Richard Smith 在http://wg21.link/p1141r2上制定。

第 3 章 概念、需求和约束

本章介绍了 C++ 新特性的概念，包括关键字的概念和要求，该特性用于根据需求约束代码的可用性。

概念是 C++ 的一个里程碑，因为概念为编写泛型代码时，需要提供了一种语言特性: 指定需求。虽然有一些变通方法，但现在有了一种简单易读的方法来指定泛型代码的需求，在需求不满足时可得到更好的判断，在泛型代码不起作用时禁用 (即使可编译)，并可在不同类型的泛型代码之间进行切换。

3.1. 提出概念和需求的动机

下面的函数模板，返回两个值中的最大值:

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

此函数模板可以为具有相同类型的两个实参调用，前提是对形参执行的操作 (使用小于操作符进行比较，并进行复制) 有效。

当传递的是两个指针时，将比较其地址，而不是所引用的值。

3.1.1 逐步完善模板

使用 requires 子句

为了解决这个问题，可以为模板配置一个约束，这样在传递原始指针时就不可用了:

```
template<typename T>
requires (!std::is_pointer_v<T>)
T maxValue(T a, T b)
{
  return b < a ? a : b;
}
```

约束是在 requires 子句中表述的，该子句是通过关键字 requires 引入的 (还有其他方式可用来表述约束)。

为了指定模板不能用于原始指针的约束，可以使用标准类型特征 std::is_pointer_v<>(生成标准类型特征 std::is_pointer<> 的值成员)[类型特征是在 C++11 中作为标准类型函数引入的，C++17 中引入了以_v 后缀的使用方式]。有了这个约束，就不能再为原始指针使用函数模板了:

```
int x = 42;
int y = 77;
```

```
std::cout << maxValue(x, y) << '\n'; // OK: maximum value of ints
std::cout << maxValue(&x, &y) << '\n'; // ERROR: constraint not met
```

该要求是编译时检查，对编译后代码的性能没有影响。所以模板不能用于原始指针，当传递原始指针时，编译器的行为就好像模板不存在一样。

定义和使用概念

可能要不止一次地需要指针约束，所以可以为约束引入一个概念:

```
template<typename T>
concept IsPointer = std::is_pointer_v<T>;
```

概念是一个模板，应用于传递的模板参数的一个或多个需求引入名称，以便可以将这些需求用作约束。在等号之后 (这里不能使用大括号)，必须将需求指定为在编译时求值的布尔表达式，则要求用于特化 IsPointer<> 的模板实参必须是裸指针。

可以使用这个概念来约束 maxValue() 模板:

```
template<typename T>
requires (!IsPointer<T>)
T maxValue(T a, T b)
{
  return b < a ? a : b;
}
```

重载概念

通过使用约束和概念，甚至可以重载 maxValue() 模板，为指针和其他类型分别提供实现:

```
template<typename T>
requires (!IsPointer<T>)
T maxValue(T a, T b) // maxValue() for non-pointers
{
  return b < a ? a : b; // compare values
}

template<typename T>
requires IsPointer<T>
auto maxValue(T a, T b) // maxValue() for pointers
{
  return maxValue(*a, *b); // compare values the pointers point to
}
```

注意，仅用一个概念 (或多个概念与 && 组合) 约束模板的 requires 子句不再需要括号，否定的概念总是需要括号。

现在有了两个同名的函数模板，但每种类型只能使用其中一个:

```
int x = 42;
int y = 77;
std::cout << maxValue(x, y) << '\n'; // calls maxValue() for non-pointers
std::cout << maxValue(&x, &y) << '\n'; // calls maxValue() for pointers
```

指针的实现将返回值的计算委托给指针所引用的对象，所以第二次调用同时使用了 maxValue() 模板。将指针传递给 int 时，实例化 T 为 int* 的指针模板，并将 T 为 int 的非指针基本模板 maxValue()。

现在可以进行递归了，可以请求指向 int 类型指针的指针的最大值:

```
int* xp = &x;
int* yp = &y;
std::cout << maxValue(&xp, &yp) << '\n'; // calls maxValue() for int**
```

使用概念解析的重载

重载解析认为有约束的模板比没有约束的模板更特化，所以只约束对指针实现就够了:

```
template<typename T>
T maxValue(T a, T b) // maxValue() for a value of type T
{
  return b < a ? a : b; // compare values
}

template<typename T>
requires IsPointer<T>
auto maxValue(T a, T b) // maxValue() for pointers (higher priority)
{
  return maxValue(*a, *b); // compare values the pointers point to
}
```

重载使用引用和非引用就可能出现歧义，所以要谨慎。

使用概念，甚至可以使用某些约束，但这需要使用包含其他概念的概念。

类型约束

若约束是应用于参数的单个概念，则有几种方法可以简化约束的说明。可以在声明模板形参时，直接将其指定为类型约束:

```
template<IsPointer T> // only for pointers
auto maxValue(T a, T b)
{
  return maxValue(*a, *b); // compare values the pointers point to
}
```

使用 auto 声明参数时，可以将该概念用作类型约束:

```
auto maxValue(IsPointer auto a, IsPointer auto b)
{
  return maxValue(*a, *b); // compare values the pointers point to
}
```

这也适用于通过引用传递的参数:

```
auto maxValue3(const IsPointer auto& a, const IsPointer auto& b)
{
  return maxValue(*a, *b); // compare values the pointers point to
}
```

通过直接约束这两个形参，改变了模板的规范: 不再要求 a 和 b 必须具有相同的类型，只要求两者都是类指针对象就好。

当使用模板语法时，等效代码如下所示:

```
template<IsPointer T1, IsPointer T2> // only for pointers
auto maxValue(T1 a, T2 b)
{
  return maxValue(*a, *b); // compare values the pointers point to
}
```

还应该允许比较值的基本函数模板使用不同的类型。一种方法是指定两个模板参数:

```
template<typename T1, typename T2>
auto maxValue(T1 a, T2 b) // maxValue() for values
{
  return b < a ? a : b; // compare values
}
```

另一种选择是使用 auto 参数:

```
auto maxValue(auto a, auto b) // maxValue() for values
{
  return b < a ? a : b; // compare values
}
```

现在可以传递一个指向整型类型的指针和一个指向双精度类型的指针。

尾接 requires 子句

maxValue() 的指针版本:

```
auto maxValue(IsPointer auto a, IsPointer auto b)
{
  return maxValue(*a, *b); // compare values the pointers point to
}
```

还有一个不明显的隐含要求: 解引用之后，值必须可比较。

编译器在 (递归地) 实例化 maxValue() 模板时检测到该需求。错误消息有一个问题，因为错误发生较晚，并且在指针的 maxValue() 声明中该要求是不可见的。

为了让指针版本在声明中直接要求指针所指向的值必须具有可比性，可以在函数中添加另一个约束:

```
auto maxValue(IsPointer auto a, IsPointer auto b)
requires IsComparableWith<decltype(*a), decltype(*b)>
{
  return maxValue(*a, *b);
}
```

使用后面的 requires 子句，可以在参数列表之后指定。好处是可以使用一个参数的名称，甚至可以组合多个参数名称来制定约束。

标准概念

前面的例子中，没有定义 IsComparableWith 这个概念。可以使用 require 表达式 (稍后会介绍)，也可以使用 C++ 标准库的概念:

```
auto maxValue(IsPointer auto a, IsPointer auto b)
requires std::totally_ordered_with<decltype(*a), decltype(*b)>
{
  return maxValue(*a, *b);
}
```

概念 std::totally_ordered_with 接受两个模板形参，用于检查传递的类型的值是否支持可比较操作符 ==、!=、<、<=、> 和 >=。

标准库为常见约束提供了许多标准概念，在命名空间 std 中提供 (有时使用子命名空间)。

还可以使用概念 std::three_way_comparable_with，这要求支持操作符 <=>(为概念提供名称)。要检查是否支持对相同类型的两个对象进行比较，可以使用 std::totally_ordered 这个概念。

requires 表达式

maxValue() 模板不适用于非原始指针的类指针类型，例如：智能指针。若代码也要为这些类型编译，最好将指针定义为可以调用解引用操作符的对象。

C++20 中，就很容易指定:

```
template<typename T>
concept IsPointer = requires(T p) { *p; }; // expression *p has to be well-formed
```

这个概念没有对原始指针使用类型特性，而是提出了一个简单的要求: 表达式 *p 必须对类型 T 的对象 p 有效。

再使用 requires 关键字来引入一个 requires 表达式，其可以定义类型和参数的一个或多个需求。通过声明类型为 T 的形参 p，就可以指定必须支持这种对象的哪些操作。

还可以要求多个操作、类型成员以及表达式产生约束类型。例如:

```
template<typename T>
concept IsPointer = requires(T p) {
  *p; // operator * has to be valid
  p == nullptr; // can compare with nullptr
  {p < p} -> std::same_as<bool>; // operator < yields bool
};
```

这里指定了三个要求，都适用于定义这个概念的类型，为 T 的参数 p:

- 该类型必须支持解引用操作符。
- 该类型必须支持小于操作符，该操作符必须产生 bool 类型。
- 该类型的对象必须与 nullptr 比较。

不需要两个 T 类型的形参来检查是否可以调用小于操作符，运行时值无关紧要，但对于如何指定表达式产生的结果有一些限制 (例如，不能只指定 bool 而，不指定 std::same_as<>)。

这里不要求 p 是一个等于 nullptr 的指针，只要求可以将 p 与 nullptr 进行比较。这就排除了迭代器，其不能与 nullptr 进行比较 (除非碰巧实现为原始指针，例如 std::array<> 类型通常就是这种情况)。

这是一个编译时约束，对生成的代码没有影响，只决定代码编译哪种类型，所以将形参 p 声明为值还是引用就无关紧要了。

也可以直接在 requires 子句中使用 requires 表达式作为临时约束 (这看起来有点滑稽，但当理解了 requires 子句和 requires 表达式之间的区别，并且两者都需要关键字 requires，就有意义了):

```
template<typename T>
requires requires(T p) { *p; } // constrain template with ad-hoc requirement
auto maxValue(T a, T b)
{
  return maxValue(*a, *b);
}
```

使用概念的完整例子

已经介绍了所有必要的内容，再来看一个完整的示例程序，用于计算普通值和指针类对象的最大值:

lang/maxvalue.cpp

```cpp
#include <iostream>
// concept for pointer-like objects:
template<typename T>
concept IsPointer = requires(T p) {
  *p; // operator * has to be valid
  p == nullptr; // can compare with nullptr
  {p < p} -> std::convertible_to<bool>; // < yields bool
};

// maxValue() for plain values:
auto maxValue(auto a, auto b)
{
  return b < a ? a : b;
}

// maxValue() for pointers:
auto maxValue(IsPointer auto a, IsPointer auto b)
requires std::totally_ordered_with<decltype(*a), decltype(*b)>
{
  return maxValue(*a, *b); // return maximum value of where the pointers refer to
}
int main()
{
  int x = 42;
  int y = 77;
  std::cout << maxValue(x, y) << '\n'; // maximum value of ints
  std::cout << maxValue(&x, &y) << '\n'; // maximum value of where the pointers point
  ↪ to

  int* xp = &x;
  int* yp = &y;
  std::cout << maxValue(&xp, &yp) << '\n'; // maximum value of pointer to pointer

  double d = 49.9;
  std::cout << maxValue(xp, &d) << '\n'; // maximum value of int and double pointer
}
```

不能使用 maxValue() 来检查两个迭代器值的最大值:

```cpp
std::vector coll{0, 8, 15, 11, 47};
auto pos = std::find(coll.begin(), coll.end(), 11); // find specific value
if (pos != coll.end()) {
  // maximum of first and found value:
  auto val = maxValue(coll.begin(), pos); // ERROR
}
```

原因是要求形参与 nullptr 可比较，而迭代器不需要支持 nullptr。这是否是一个设计问题？所以对于概念的定义非常重要。

3.2. 使用约束和概念

可以使用 requirea 子句或概念来约束几乎所有形式的泛型代码:

- 函数模板:

```
template<typename T>
requires ...
void print(const T&) {
  ...
}
```

- 类模板：

```
template<typename T>
requires ...
class MyType {
  ...
}
```

- 别名模板
- 变量模板
- (甚至可以约束) 成员函数

对于这些模板，可以约束类型和值参数。

但这里不能约束概念:

```
template<std::ranges::sized_range T> // ERROR
concept IsIntegralValType = std::integral<std::ranges::range_value_t<T>>;
```

必须这样指定:

```
template<typename T>
concept IsIntegralValType = std::ranges::sized_range<T> &&
              std::integral<std::ranges::range_value_t<T>>;
```

3.2.1 约束别名模板

下面是一个约束别名模板的例子 (使用声明的泛型):

```
template<std::ranges::range T>
using ValueType = std::ranges::range_value_t<T>;
```

该声明等价于:

```
template<typename T>
requires std::ranges::range<T>
using ValueType = std::ranges::range_value_t<T>;
```

类型 ValueType<> 现在只可对范围类型进行定义:

```
ValueType<int> vt1; // ERROR
ValueType<std::vector<int>> vt2; // int
ValueType<std::list<double>> vt3; // double
```

3.2.2 约束变量模板

下面是一个约束变量模板的例子:

```
template<std::ranges::range T>
constexpr bool IsIntegralValType = std::integral<std::ranges::range_value_t<T>>;
```

这相当于:

```
template<typename T>
requires std::ranges::range<T>
constexpr bool IsIntegralValType = std::integral<std::ranges::range_value_t<T>>;
```

类型 IsIntegralValType<> 现在只可在范围中定义:

```
bool b1 = IsIntegralValType<int>; // ERROR
bool b2 = IsIntegralValType<std::vector<int>>; // true
bool b3 = IsIntegralValType<std::list<double>>; // false
```

3.2.3 约束成员函数

requires 子句也可以是成员函数声明的一部分，开发者就可以根据需求和概念指定不同的 API。

lang/valorcoll.hpp

```
#include <iostream>
#include <ranges>

template<typename T>
class ValOrColl {
  T value;
  public:
  ValOrColl(const T& val)
  : value{val} {
  }
  ValOrColl(T&& val)
  : value{std::move(val)} {
  }

  void print() const {
    std::cout << value << '\n';
```

```
  }

  void print() const requires std::ranges::range<T> {
    for (const auto& elem : value) {
      std::cout << elem << ' ';
    }
    std::cout << '\n';
  }
};
```

定义了一个类 ValOrColl，可以保存一个值或一个值的集合，作为 T 类型的值。提供了两个 print() 成员函数，类使用标准概念 std::ranges::range 来决定使用哪一个:

- 若类型 T 是一个集合，则满足约束，所以两个 print() 成员函数都可用。但重载解析首选第二个 print()，因为该成员函数有约束，将遍历集合的元素。
- 若类型 T 不是集合，则只有第一个 print() 可用，因此可以使用。

例如，可以这样使用这个类:

lang/valorcoll.cpp

```
#include "valorcoll.hpp"
#include <vector>

int main()
{
  ValOrColl o1 = 42;
  o1.print();
  ValOrColl o2 = std::vector{1, 2, 3, 4};
  o2.print();
}
```

该程序应有以下输出:

```
42
1 2 3 4
```

这种方式只能约束模板，不能使用 requires 来约束普通函数:

```
void foo() requires std::numeric_limits<char>::is_signed // ERROR
{
  ...
}
```

C++ 标准库中约束成员函数的一个例子是，const 视图的 begin() 的条件可用性。

3.2.4 约束非类型模板参数

可以约束的不仅仅是类型，还可以约束作为模板参数的值 (非类型模板参数 (NTTP))。例如:

```
template<int Val>
concept LessThan10 = Val < 10;
```

或者更通用的情况:

```
template<auto Val>
concept LessThan10 = Val < 10;
```

可以这样使用:

```
template<typename T, int Size>
requires LessThan10<Size>
class MyType {
  ...
};
```

稍后将讨论更多的示例。

3.3. 概念和约束在实践中的应用

使用需求作为约束可能有有很多原因:

- 约束帮助我们理解模板上的限制，并在需求破坏时获得更容易理解的错误消息。
- 约束可以用于在代码没有意义的情况下禁用泛型代码:
  - 对于某些类型，泛型代码可能可以编译，但不能做正确的事情。
  - 可能必须修复重载解析，若有多个有效选项，重载解析将决定使用哪个操作。
- 约束可用于重载或特化泛型代码，以便针对不同的类型编译不同的代码。

通过逐步开发另一个示例来进一步了解这些原因，还可以引入一些关于约束、需求和概念的细节信息。

3.3.1 理解代码和错误消息

假设要编写将对象的值插入到集合中的泛型代码，可以将其实现为泛型代码。当明确了传递的对象类型，就会对其进行编译:

```
template<typename Coll, typename T>
void add(Coll& coll, const T& val)
{
  coll.push_back(val);
}
```

这段代码并不总是可以编译。对于传递的参数的类型有一个隐含的要求: 对于类型为 Coll 的容器，必须支持对类型为 T 的值的 push_back()。

也可以认为这是多个基本需求的组合:

- 类型 Coll 必须支持 push_back()。
- 必须进行从类型 T 到 Coll 元素类型的转换。
- 若传递的实参具有元素类型 Coll，则该类型必须支持复制 (使用传递值初始化)。

若违反这些要求中的一个，代码将无法编译。例如:

```
std::vector<int> vec;
add(vec, 42); // OK
add(vec, "hello"); // ERROR: no conversion from string literal to int

std::set<int> coll;
add(coll, 42); // ERROR: no push_back() supported by std::set<>

std::vector<std::atomic<int>> aiVec;
std::atomic<int> ai{42};
add(aiVec, ai); // ERROR: cannot copy/move atomics
```

当编译失败时，错误消息会非常清楚，例如：模板的参数类型没有找到成员 push_back() 时:

```
prog.cpp: In instantiation of ’ void add(Coll&, const T&)
             [with Coll = std::__debug::set<int>; T = int]’ :
prog.cpp:17:18:     required from here
prog.cpp:11:8: error: ’ class std::set<int>’  has no member named ’ push_back’
  11 | coll.push_back(val);
     | ~~~~~^~~~~~~~~
```

然而，一般的错误消息也很难阅读和理解。例如，当编译器必须处理不支持复制的需求时，问题就会在 std::vector<> 实现的深处出现。会看到得到 40 到 90 行错误信息，从而必须仔细寻找违反的需求:

```
...
prog.cpp:11:17: required from ’ void add(Coll&, const T&)
                    [with Coll = std::vector<std::atomic<int> >; T =
                    ↪  std::atomic<int>]’
prog.cpp:25:18:     required from here
.../include/bits/stl_construct.h:96:17:
    error: use of deleted function
’ std::atomic<int>::atomic(const std::atomic<int>&)’
    96 | -> decltype(::new((void*)0) _Tp(std::declval<_Args>()...))
       |                 ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
...
```

读者们可能认为可以通过定义和使用一个检查，确定是否可以执行 push_back() 调用的概念来改善这种情况:

```cpp
template<typename Coll, typename T>
concept SupportsPushBack = requires(Coll c, T v) {
  c.push_back(v);
};

template<typename Coll, typename T>
requires SupportsPushBack<Coll, T>
void add(Coll& coll, const T& val)
{
  coll.push_back(val);
}
```

没有找到 push_back() 的错误信息现在可能如下所示:

```
prog.cpp:27:4: error: no matching function for call to ’ add(std::set<int>&, int)’
    27 | add(coll, 42);
       | ~~~^~~~~~~~~~
prog.cpp:14:6: note: candidate: ’ template<class Coll, class T> requires ...’
     14 | void add(Coll& coll, const T& val)
        |      ^~~
prog.cpp:14:6: note: template argument deduction/substitution failed:
prog.cpp:14:6: note: constraints not satisfied
prog.cpp: In substitution of ’ template<class Coll, class T> requires ...
           [with Coll = std::set<int>; T = int]’ :
prog.cpp:27:4: required from here
prog.cpp:8:9: required for the satisfaction of ’ SupportsPushBack<Coll, T>’
                  [with Coll = std::set<int, std::less<int>, std::allocator<int> >; T
                  ↪  = int]
prog.cpp:8:28: in requirements with ’ Coll c’ , ’ T v’
                      [with T = int; Coll = std::set<int, std::less<int>,
                      ↪  std::allocator<int> >]
prog.cpp:9:16: note: the required expression ’ c.push_back(v)’ is invalid
      9 | c.push_back(v);
        | ~~~~~~~~~~~^~~
```

在传递原子类型时，仍然会在 std::vector<> 的代码深处检测可复制性检查 (这一次是在检查概念时，而不是在编译代码时)。

当指定使用 push_back() 的基本约束作为要求时，情况会有所改善:

```cpp
template<typename Coll, typename T>
requires std::convertible_to<T, typename Coll::value_type>
void add(Coll& coll, const T& val)
{
  coll.push_back(val);
}
```

使用标准概念 std::convertible_to 来要求，(隐式或显式) 将传递的实参 T 的类型转换为集合的元素类型。

若违背需求，就会得到一条错误消息，其中包含违反的概念和位置。例如 [这只是一种可能的消息例子，不一定与特定编译器的情况相匹配]:

```
...
prog.cpp:11:17: In substitution of ’ template<class Coll, class T>
                 requires convertible_to<T, typename Coll::value_type>
                 void add(Coll&, const T&)
                [with Coll = std::vector<std::atomic<int> >; T = std::atomic<int>]’ :
prog.cpp:25:18: required from here
.../include/concepts:72:13: required for the satisfaction of
                 ’ convertible_to<T, typename Coll::value_type>
                  [with T = std::atomic<int>;
                      Coll = std::vector<std::atomic<int>,
                                         std::allocator<std::atomic<int> > >]’
.../include/concepts:72:30: note: the expression ’ is_convertible_v<_From, _To>
                      [with _From = std::atomic<int>; _To = std::atomic<int>]’
                      evaluated to ’ false’
    72 | concept convertible_to = is_convertible_v<_From, _To>
       |                                  ^~~~~~~~~~~~~~~~~~~~~~~~~~~
...
```

参见 rangessort.cpp 中的另一个检查参数约束的算法示例。

3.3.2 使用概念禁用泛型代码

假设为上面介绍的 add() 函数模板提供一个特殊的实现。在处理浮点值时，应该发生一些不同的事。

一种简单的方法可能是为 double 重载函数模板:

```
template<typename Coll, typename T>
void add(Coll& coll, const T& val) // for generic value types
{
  coll.push_back(val);
}

template<typename Coll>
void add(Coll& coll, double val) // for floating-point value types
{
  ... // special code for floating-point values
  coll.push_back(val);
}
```

当传递一个 double 类型的参数作为第二个参数时，调用第二个函数; 否则，使用泛型参数:

```
std::vector<int> iVec;
add(iVec, 42); // OK: calls add() for T being int
```

```
3
4  std::vector<double> dVec;
5  add(dVec, 0.7); // OK: calls 2nd add() for double
```

当传递 double 类型时，两个函数重载匹配。首选第二个重载，因为其与第二个参数完美匹配。

若传递一个浮点数，可以有以下效果:

```
1  float f = 0.7;
2  add(dVec, f); // OOPS: calls 1st add() for T being float
```

原因在于重载解析的一些微妙细节，这两个函数都可以调用。重载解析有一些通用规则，例如:

- 没有类型转换的调用优先于具有类型转换的调用。
- 调用普通函数优于调用函数模板。

重载解析必须在调用类型转换和调用函数模板之间做出决定。按照规则，优先选择带有模板参数的版本。

修复重载解析

错误的重载解析的修复非常简单。不需要用特定类型声明第二个形参，只需要求要插入的值为浮点类型即可，可以使用新的标准概念 std::floating_point 来约束浮点值的函数模板:

```
1  template<typename Coll, typename T>
2  requires std::floating_point<T>
3  void add(Coll& coll, const T& val)
4  {
5    ... // special code for floating-point values
6    coll.push_back(val);
7  }
```

因为使用的概念只适用于单个模板形参，所以可以使用速记表示法:

```
1  template<typename Coll, std::floating_point T>
2  void add(Coll& coll, const T& val)
3  {
4    ... // special code for floating-point values
5    coll.push_back(val);
6  }
```

或者，使用 auto 参数:

```
1  void add(auto& coll, const std::floating_point auto& val)
2  {
3    ... // special code for floating-point values
4    coll.push_back(val);
5  }
```

对于 add()，现在有两个可以调用的函数模板: 一个不带特定需求，另一个带特定需求:

```
uirement:
template<typename Coll, typename T>
void add(Coll& coll, const T& val) // for generic value types
{
  coll.push_back(val);
}

template<typename Coll, std::floating_point T>
void add(Coll& coll, const T& val) // for floating-point value types
{
  ... // special code for floating-point values
  coll.push_back(val);
}
```

这就足够了，因为重载解析也更喜欢有约束的重载或特化，而不是那些约束较少或没有约束的重载或特化:

```
std::vector<int> iVec;
add(iVec, 42); // OK: calls add() for generic value types

std::vector<double> dVec;
add(dVec, 0.7); // OK: calls add() for floating-point types
```

非必须，则相同

若两个重载或特化都有约束，重载解析可以决定哪一个更好，这一点很重要。为了支持这一点，签名不应有太大的差异。

若签名差异太大，则可能不适用更受约束的重载。例如，声明浮点值的重载以按值接受实参，则传递浮点值会产生二义性:

```
template<typename Coll, typename T>
void add(Coll& coll, const T& val) // note: pass by const reference
{
  coll.push_back(val);
}

template<typename Coll, std::floating_point T>
void add(Coll& coll, T val) // note: pass by value
{
  ... // special code for floating-point values
  coll.push_back(val);
}

std::vector<double> dVec;
add(dVec, 0.7); // ERROR: both templates match and no preference
```

后一项声明不再是前一项声明的特例，只有两个不同的函数模板都可以使用。

若确实希望使用不同的签名，则必须约束第一个函数模板不能用于浮点值。

窄化限制

这个例子中还有一个有趣的问题: 两个函数模板都允许我们传递一个 double 类型的值，将其添加到 int 类型的集合中:

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

add(iVec, 1.9); // OOPS: add 1
```

原因是有从 double 到 int 的隐式类型转换 (由于与编程语言 C 兼容)，这种可能丢失部分值的隐式转换称为数据窄化。所以上面的代码在插入之前编译，会将值 1.9 转换为 1。

若不希望数据窄化，有多个选项。一种选择是通过要求传递的值类型与集合的元素类型匹配，完全禁用类型转换:

```
requires std::same_as<typename Coll::value_type, T>
```

但这也会禁用有用和安全的类型转换。

出于这个原因，最好定义一个概念来确定一个类型是否可以在不缩小的情况下转换为另一个类型，这在一个简短的需求中是可能的 [感谢 Giuseppe D 'Angelo 和 Zhihao Yuan 在http://wg21.link/p0870中介绍检查窄化转换的技巧]:

```
template<typename From, typename To>
concept ConvertsWithoutNarrowing =
  std::convertible_to<From, To> &&
  requires (From&& x) {
    { std::type_identity_t<To[]>{std::forward<From>(x)} }
    -> std::same_as<To[1]>;
  };
```

可以使用这个概念来制定相应的约束:

```
template<typename Coll, typename T>
requires ConvertsWithoutNarrowing<T, typename Coll::value_type>
void add(Coll& coll, const T& val)
{
  ...
}
```

包含约束

定义上面的窄化转换的概念可能就足够了，而不需要 std::convertible_to 概念，剩下的部分会进行隐式检查:

```
template<typename From, typename To>
concept ConvertsWithoutNarrowing = requires (From&& x) {
  { std::type_identity_t<To[]>{std::forward<From>(x)} } -> std::same_as<To[1]>;
};
```

若 ConvertsWithoutNarrowing 概念也检查 std::convertible_to 概念，编译器可以检测到 ConvertsWithoutNarrowing 比 std::convertible_to 更受约束。术语是 ConvertsWithoutNarrowing 包含 std::convertible_to。

这允许开发者做以下事情:

```
emplate<typename F, typename T>
requires std::convertible_to<F, T>
void foo(F, T)
{
  std::cout << "may be narrowing\n";
}

template<typename F, typename T>
requires ConvertsWithoutNarrowing<F, T>
void foo(F, T)
{
  std::cout << "without narrowing\n";
}
```

若没有指定 ConvertsWithoutNarrowing 包含 std::convertible_to，编译器在调用带有两个参数的 foo() 时将引发歧义错误，这两个参数相互转换而不进行窄化。

以同样的方式，概念可以包含其他概念，多以更特化地用于重载解析。事实上，C++ 标准概念构建了一个相当复杂的包含图。

稍后我们将讨论包含的细节。

3.3.3 使用需求调用不同的函数

最后，应该让 add() 函数模板更灵活:

- 支持只提供 insert() 而不是 push_back() 来插入新元素的集合。
- 支持传递一个集合 (容器或范围) 来插入多个值。

这些是不同的函数，应该有不同的名字，通常使用不同的名字会更好。C++ 标准库是一个很好的例子，说明如果协调不同的 API。例如，可以使用相同的泛型代码遍历所有容器，尽管在内部，容器使用非常不同的方式转到下一个元素并访问其值。

使用概念调用不同的函数

刚刚介绍了概念，“显而易见”的方法可能是引入一个概念来找出是否支持某个函数调用:

```
template<typename Coll, typename T>
concept SupportsPushBack = requires(Coll c, T v) {
  c.push_back(v);
};
```

也可以定义一个只需要集合作为模板形参的概念:

```
template<typename Coll>
concept SupportsPushBack = requires(Coll coll, Coll::value_type val) {
  coll.push_back(val);
};
```

不必在这里使用 typename 来使用 Coll::value_type。从 C++20 开始，当上下文明确限定成员必须是类型时，不再需要 typename。

还有其他方式来声明这个概念:

- 可以使用 std::declval<>() 来获取元素类型值:

```
template<typename Coll>
concept SupportsPushBack = requires(Coll coll) {
  coll.push_back(std::declval<typename Coll::value_type&>());
};
```

可以看到概念和需求的定义并没有创建代码,是一个未求值的上下文,可以使用 std::declval<>() 来表示“假设有一个这种类型的对象”，无论将 coll 声明为值，还是非 const 引用都无关紧要。这里的 & 很重要。若没有 &，只需要使用移动语义插入一个右值 (比如一个临时对象)。对于 &，我们创建了一个左值，因此需要 push_back() 进行复制。

- 可以使用 std::ranges::range_value_t 来代替 value_type 的成员:

```
template<typename Coll>
concept SupportsPushBack = requires(Coll coll) {
  coll.push_back(std::declval<std::ranges::range_value_t<Coll>>());
};
```

当需要集合的元素类型时，使用 std::ranges::range_value_t<> 使代码更具泛型 (例如，也适用于原始数组)。因为这里需要成员 push_back()，所以也需要成员的 value_type。

使用一个参数的 SupportPushBack 概念，这里可以提供两个实现:

```
template<typename Coll, typename T>
requires SupportsPushBack<Coll>
void add(Coll& coll, const T& val)
{
  coll.push_back(val);
}

template<typename Coll, typename T>
```

```
void add(Coll& coll, const T& val)
{
  coll.insert(val);
}
```

这里不需要命名需求 SupportsInsert，因为带有附加需求的 add() 更特殊，所以重载解析更偏向它。但只有少数容器支持只使用一个参数调用 insert()，为了避免其他重载和 add() 调用的问题，最好在这里也有一个约束。

这里将需求定义为一个概念，甚至可以将它用作模板形参的类型约束:

```
template<SupportsPushBack Coll, typename T>
void add(Coll& coll, const T& val)
{
  coll.push_back(val);
}
```

作为一个概念，也可以将其用作 auto 作为参数类型的类型约束:

```
void add(SupportsPushBack auto& coll, const auto& val)
{
  coll.push_back(val);
}

template<typename Coll, typename T>
void add(auto& coll, const auto& val)
{
  coll.insert(val);
}
```

if constexpr 的概念

也可以在 if constexpr 条件中直接使用 SupportsPushBack 这个概念:

```
if constexpr (SupportsPushBack<decltype(coll)>) {
  coll.push_back(val);
}
else {
  coll.insert(val);
}
```

组合 requires 和 if constexpr

甚至可以跳过引入的概念，直接将 requires 表达式作为条件传递给编译时 if:

```
if constexpr (requires { coll.push_back(val); }) {
  coll.push_back(val);
}
else {
  coll.insert(val);
}
```

这是在泛型代码中切换两个不同函数调用的好方法。当引入一个不值得的概念时，建议这样做。

概念与变量模板

为什么使用概念比使用 bool 类型的变量模板更好 (就像类型特性一样)，比如:

```
template<typename T>
constexpr bool SupportsPushBack = requires(T coll) {
  coll.push_back(std::declval<typename T::value_type>());
};
```

概念有以下好处:

- 可包含。
- 可以直接用作模板参数或 auto 前面的类型约束。
- 若使用特殊需求，可以与编译时一起使用。

若不需要这些，选择概念定义还是 bool 类型的变量模板的问题就变得有趣了，稍后将详细讨论这个问题。

插入单个和多个值

为了提供处理作为一个集合传递的多个值的重载，可以简单地添加约束。标准概念 std::ranges::input_range 可用于此:

```
template<SupportsPushBack Coll, std::ranges::input_range T>
void add(Coll& coll, const T& val)
{
  coll.insert(coll.end(), val.begin(), val.end());
}

template<typename Coll, std::ranges::input_range T>
void add(Coll& coll, const T& val)
{
  coll.insert(val.begin(), val.end());
}
```

同样，只要重载将此作为附加约束，这些函数将是首选。

概念 std::ranges::input_range 是一个用于处理范围的概念，范围是可以使用 begin() 和 end() 迭代的集合。但范围不需要将 begin() 和 end() 作为成员函数，所以处理范围的代码应该使用范围库提供的 std::ranges::begin() 和 std::ranges::end():

```
template<SupportsPushBack Coll, std::ranges::input_range T>
void add(Coll& coll, const T& val)
{
  coll.insert(coll.end(), std::ranges::begin(val), std::ranges::end(val));
}
template<typename Coll, std::ranges::input_range T>
void add(Coll& coll, const T& val)
{
  coll.insert(std::ranges::begin(val), std::ranges::end(val));
}
```

这些辅助程序是函数对象，因此使用它们可以避免 ADL 问题。

处理多重约束

通过汇集所有有用的概念和需求，可以将它们放在不同位置的函数中。

```
template<SupportsPushBack Coll, std::ranges::input_range T>
requires ConvertsWithoutNarrowing<std::ranges::range_value_t<T>,
typename Coll::value_type>
void add(Coll& coll, const T& val)
{
  coll.insert(coll.end(),
        std::ranges::begin(val), std::ranges::end(val));
}
```

要禁用窄化转换，可以使用 std::ranges::range_value_t 将范围的元素类型传递给 ConvertsWithoutNarrowing。std::ranges::range_value_t 是另一个范围工具，用于在迭代范围时获取元素的类型。

也可以在 requires 从句中将它们组合在一起:

```
template<typename Coll, typename T>
requires SupportsPushBack<Coll> &&
      std::ranges::input_range<T> &&
      ConvertsWithoutNarrowing<std::ranges::range_value_t<T>,
              typename Coll::value_type>
void add(Coll& coll, const T& val)
{
  coll.insert(coll.end(),
        std::ranges::begin(val), std::ranges::end(val));
}
```

声明函数模板的两种方式等价。

3.3.4 完整的例子

前面的小节提供了很大的灵活性，现在把所有的选项放在一起，就有一个完整的例子:

lang/add.cpp

```
#include <iostream>
#include <vector>
#include <set>
#include <ranges>
#include <atomic>

// concept for container with push_back():
template<typename Coll>
concept SupportsPushBack = requires(Coll coll, Coll::value_type val) {
  coll.push_back(val);
};

// concept to disable narrowing conversions:
template<typename From, typename To>
concept ConvertsWithoutNarrowing =
  std::convertible_to<From, To> &&
  requires (From&& x) {
    { std::type_identity_t<To[]>{std::forward<From>(x)} }
    -> std::same_as<To[1]>;
  };


// add() for single value:
template<typename Coll, typename T>
requires ConvertsWithoutNarrowing<T, typename Coll::value_type>
void add(Coll& coll, const T& val)
{
  if constexpr (SupportsPushBack<Coll>) {
    coll.push_back(val);
  }
  else {
    coll.insert(val);
  }
}
// add() for multiple values:
template<typename Coll, std::ranges::input_range T>
requires ConvertsWithoutNarrowing<std::ranges::range_value_t<T>,
                  typename Coll::value_type>
void add(Coll& coll, const T& val)
{
  if constexpr (SupportsPushBack<Coll>) {
    coll.insert(coll.end(),
          std::ranges::begin(val), std::ranges::end(val));
  }
```

```
  else {
    coll.insert(std::ranges::begin(val), std::ranges::end(val));
  }
}

int main()
{
  std::vector<int> iVec;
  add(iVec, 42); // OK: calls push_back() for T being int

  std::set<int> iSet;
  add(iSet, 42); // OK: calls insert() for T being int

  short s = 42;
  add(iVec, s); // OK: calls push_back() for T being short

  long long ll = 42;
  // add(iVec, ll); // ERROR: narrowing
  // add(iVec, 7.7); // ERROR: narrowing

  std::vector<double> dVec;
  add(dVec, 0.7); // OK: calls push_back() for floating-point types
  add(dVec, 0.7f); // OK: calls push_back() for floating-point types
  // add(dVec, 7); // ERROR: narrowing

  // insert collections:
  add(iVec, iSet); // OK: insert set elements into a vector
  add(iSet, iVec); // OK: insert vector elements into a set

  // can even insert raw array:
  int vals[] = {0, 8, 18};
  add(iVec, vals); // OK
  // add(dVec, vals); // ERROR: narrowing
}
```

如您所见，我决定使用概念 SupportsPushBack 作为各种函数模板中的一个模板参数，并使用 if constexpr。

3.3.5 以前的解决方法

C++20 前，约束模板已经成为可能。这样做的方法通常不容易使用，并且可能存在明显的缺点。

SFINAE

C++20 前禁用模板可用性的主要方法是 SFINAE。术语“SFINAE”(发音类似于 sfee-nay) 代表“替换失败不是错误”，若泛型代码的声明格式不佳，就忽略它，而非引发编译时错误。

例如，要在 push_back() 和 insert() 之间切换，可以在 C++20 之前的代码中声明如下的函数模板:

```cpp
template<typename Coll, typename T>
auto add(Coll& coll, const T& val) -> decltype(coll.push_back(val))
{
  return coll.push_back(val);
}

template<typename Coll, typename T>
auto add(Coll& coll, const T& val) -> decltype(coll.insert(val))
{
  return coll.insert(val);
}
```

因为把这里的 push_back() 作为第一个模板声明的一部分，所以若不支持 push_back()，则忽略此模板。对应的声明在 insert() 的第二个模板中是必需的 (重载解析不会忽略返回类型，若两个函数模板都可以使用，则会报错)。

这是放置需求的一种隐蔽的方式，开发者可能很容易忽略。

std::enable_if<>

对于更复杂的禁用泛型代码的情况，从 C++11 开始，C++ 标准库提供了 std::enable_if<>。

这是一个接受布尔条件的类型特性，若为 false 则产生无效代码。通过在声明中的某个地方使用 std::enable_if<>，这也可以是“SFINAE out”的泛型代码。

例如，可以使用 std::enable_if<> 类型特性，排除调用 add() 函数模板的类型:

```cpp
// disable the template for floating-point values:
template<typename Coll, typename T,
typename = std::enable_if_t<!std::is_floating_point_v<T>>>
void add(Coll& coll, const T& val)
{
  coll.push_back(val);
}
```

诀窍是插入一个额外的模板参数，以便能够使用特性 std::enable_if<>。若特性没有禁用模板，将产生 void(可以指定它产生另一种类型，作为可选的第二个模板参数)。

这样的代码也很难编写和阅读，并且有一些缺点。要用不同的约束提供 add() 的另一个重载，还需要另一个模板参数。否则，将对同一个函数进行两次不同的重载，因为对于编译器 std::enable_if<> 不会更改签名。此外，对于每个调用，只有一个不同的重载是可用的。

概念提供了一种更具可读性的表述约束的方式。包括只满足一个约束，具有不同约束的模板的重载就不会违反一个定义规则，只要一个约束包含另一个约束，甚至可以满足多个约束。

3.4. 语义约束

概念可同时检查语法和语义约束:

- **语法约束**在编译时，可以检查是否满足某些功能需求 (“是否支持特定的操作?”或“特定操作是否产生特定类型?”)。
- **语义约束**满足了某些只能在运行时检查的需求 (“操作是否具有相同的效果?”或“对特定值执行相同的操作是否总是产生相同的结果?”)。

有时，概念允许开发者通过接口来指定是否满足语义约束，从而将语义约束转换为语法约束。

3.4.1 语义约束的例子

来看一些语义约束的例子。

std::ranges::sized_range

语义约束的第一个例子是概念 std::ranges::sized_range，其保证可以在常量时间内计算一个范围内的元素数量 (通过成员函数 size() 或计算开始和结束之间的差值)。

若范围类型提供 size()(作为成员函数或独立函数)，则默认情况下满足此概念。要从这个概念中退出 (例如，迭代所有元素以产生结果)，可以将 std::disable_size_range<Rg> 设置为 true:

```
class MyCont {
  ...
  std::size_t size() const; // assume this is expensive, so that this is not a sized
  ↪ range
};
// opt out from concept std::ranges::sized_range:
constexpr bool std::ranges::disable_sized_range<MyCont> = true;
```

std::ranges::range 与 std::ranges::view

语义约束的一个示例是概念 std::ranges::view。除了一些语法约束外，还保证移动构造函数/赋值、复制构造函数/赋值 (如可用) 和析构函数具有恒定的复杂性 (所花费的时间不取决于元素的数量)。

实现者可以通过公开地从 std::ranges::view_base 或 std::ranges::view_interface<> 派生，或者通过将模板特化 std::ranges::enable_view<Rg> 设置为 true 来提供相应的保证。

std::invocable 和 std::regular_invocable

语义约束的简单示例是 std::invocable 和 std::regular_invocable 概念之间的区别，后者保证不修改传递的操作和传递的参数的状态。

但不能用编译器检查这两个概念之间的区别，所以 std::regular_invocable 这个概念记录了指定 API 的意图。简单起见，这里只使用了 std::invocable。

std::weakly_incrementable 和 std::incrementable

incrementable 和 weak_incrementable 这两个概念除了在语法上有一定的区别外，在语义上也有区别:

- incrementable 要求相同值的每次递增都产生相同的结果。
- weakly_incrementable 只要求类型支持自增操作符，增加相同的值，可能会产生不同的结果。

因此:

- 当满足 incrementable 时，可以从一个起始值在一个范围内迭代多次。
- 当仅满足 weakly_incrementable 条件时，只能在一个范围内迭代一次。具有相同起始值的第二次迭代可能产生不同的结果。

这种差异对迭代器很重要: 输入流迭代器 (从流中读取值的迭代器) 只能迭代一次，因为下一次迭代产生不同的值，所以输入流迭代器满足 weakly_incrementable 概念，但不满足可递增概念，但这些概念不能用于检查这种差异:

```
std::weakly_incrementable<std::istream_iterator<int>> // yields true
std::incrementable<std::istream_iterator<int>> // OOPS: also yields true
```

原因是这种差异是一种语义约束，不能在编译时检查，所以这些概念可以用来记录约束条件:

```
template<std::weakly_incrementable T>
void algo1(T beg, T end); // single-pass algorithm

template<std::incrementable T>
void algo2(T beg, T end); // multi-pass algorithm
```

这里对算法使用了不同的名称。由于无法检查约束的语义差异，因此开发者可以决定不传递输入流迭代器:

```
algo1(std::istream_iterator<int>{std::cin}, // OK
    std::istream_iterator<int>{});

algo2(std::istream_iterator<int>{std::cin}, // OOPS: violates constraint
    std::istream_iterator<int>{});
```

然而，不能根据这个差异来区分两种实现:

```
template<std::weakly_incrementable T>
void algo(T beg, T end); // single-pass implementation

template<std::incrementable T>
void algo(T beg, T end); // multi-pass implementation
```

若在这里传递一个输入流迭代器，编译器将不正确地使用多通道实现:

```
algo(std::istream_iterator<int>{std::cin}, // OOPS: calls the wrong overload
   std::istream_iterator<int>{});
```

这里有一个解决方案，因为对于这种语义差异，C++98 已经引入了迭代器特性，迭代器概念使用了这些特性。若使用这些概念 (或相应的范围概念)，就没有问题了:

```
template<std::input_iterator T>
void algo(T beg, T end); // single-pass implementation

template<std::forward_iterator T>
void algo(T beg, T end); // multi-pass implementation

algo(std::istream_iterator<int>{std::cin}, // OK: calls the right overload
  std::istream_iterator<int>{});
```

推荐使用更具体的迭代器和范围概念，其也支持新的和修改过的迭代器类别。

3.5. 概念设计指南

现在，看看如何使用概念的指导方针。请注意，我们仍在学习如何使用概念。此外，随着时间的推移，对概念的改进支持可能会改变一些指导方针。

3.5.1 概念应该分组

为类型的每个属性或功能引入一个概念肯定是粒度过细了，所以有太多编译器必须处理的概念，并且都将其指定为约束。

因此，应该提供常见和典型的概念，以区分不同类别的需求或类型，但也有一些极端情况。

C++ 标准库提供了一个很好的设计示例，可以遵循这种方法。提供的大多数概念都用于将类型分类，如范围、迭代器、函数等为一个整体，但为了支持包容并确保概念的一致性，提供了几个基本概念 (例如 std::movable)。

结果是一个相当复杂的包含图。描述 C++ 标准概念的那一章，将对概念进行分组。

3.5.2 谨慎定义概念

概念包含，一个概念可以是另一个概念的子集，因此在重载解析中更受约束的概念是首选。

然而，需求和约束可以用不同的方式定义。对于编译器来说，找出一组需求是否是另一组需求的子集可能并不容易。

例如，若两个模板参数的概念是可交换的 (因此两个参数的顺序不重要)，则需要仔细设计概念。有关详细信息和示例，请参阅如何定义概念 std::same_as 的讨论。

3.5.3 概念与类型特征和布尔表达式

概念不仅仅是在编译时计算布尔值结果的表达式。与类型特征和其他编译时表达式相比，读者们应该更倾向于使用它们。

不过，概念有几个好处:

- 互包含。
- 可以直接用作模板参数或 auto 前面的类型约束。
- 可以与前面介绍的编译时 if(if constexpr) 一起使用。

包含概念

概念的主要好处是互包含，而类型特征不可互包含。

考虑下面的例子，用定义为类型特性的两个需求重载函数 foo():

```
template<typename T, typename U>
requires std::is_same_v<T, U> // using traits
void foo(T, U)
{
  std::cout << "foo() for parameters of same type" << '\n';
}

template<typename T, typename U>
requires std::is_same_v<T, U> && std::is_integral_v<T>
void foo(T, U)
{
  std::cout << "foo() for integral parameters of same type" << '\n';
}

foo(1, 2); // ERROR: ambiguity: both requirements are true
```

问题是，若两个需求都求值为 true，则两个重载都适合，并且没有规则表明其中一个优先于另一个，所以编译器将停止编译，并出现歧义错误。

若使用相应的概念，编译器会发现第二个需求是一种特化，若两个需求都满足，则更倾向于使用它:

```
template<typename T, typename U>
requires std::same_as<T, U> // using concepts
void foo(T, U)
{
  std::cout << "foo() for parameters of same type" << '\n';
}

template<typename T, typename U>
requires std::same_as<T, U> && std::integral<T>
void foo(T, U)
{
  std::cout << "foo() for integral parameters of same type" << '\n';
}

foo(1, 2); // OK: second foo() preferred
```

使用 if constexpr 概念

C++17 引入了编译时 if，允许根据特定的编译时条件切换代码。

例如 (如前所述):

```
template<typename Coll, typename T>
void add(Coll& coll, const T& val) // for floating-point value types
{
  if constexpr(std::is_floating_point_v<T>) {
    ... // special code for floating-point values
  }
  coll.push_back(val);
}
```

当泛型代码必须为不同类型的参数提供不同的实现，但签名相同时，使用这种方法比提供重载或特化的模板更具可读性。

不能使用 if constexpr 来提供不同的 API，以允许其他人稍后添加其他重载或特化，或者完全禁用此模板，但可以根据需求约束成员函数来启用或禁用 API 的某些部分。

3.6. 附注

自 C++98 以来，C++ 语言设计者一直在探索如何用概念约束模板的参数。C++ 编程语言中引入概念有多种方法 (例如，参见 Bjarne Stroustrup 的http://wg21.link/n1510)，但 C++ 标准委员会还未能在 C++20 之前就合适的机制达成一致。

对于 C++11 的工作草案，采用了一个非常丰富的概念方法，后来因为太复杂而放弃。之后，基于http://wg21.link/n3351, Andrew Sutton, Bjarne Stroustrup 和 Gabriel Dos Reis 在http://wg21.link/n3580上提出了一种名为 Concepts Lite 的新方法，从http://wg21.link/n4549重启了概念的技术规范。

随着时间的推移，特别是根据实现 ranges 库的经验，进行了各种改进。

http://wg21.link/p0724r0建议将概念 TS 应用于 C++20 的工作草案，最终接受的方案是由 Andrew Sutton 在http://wg21.link/p0734r0中提出。

随后提出了各种修复和改进方案并接受，最明显的变化是将http://wg21.link/p1754r1中提出的标准概念的名称改为“标准大小写”(只有小写字母和下划线)。

第 4 章 详细介绍概念、需求和约束

本章讨论了概念、需求和约束的一些细节。

此外，下一章会列出并讨论 C++20 标准库提供的所有标准概念。

4.1. 约束

要指定对泛型参数的需求，需要约束，这些约束在编译时用于决定是否实例化和编译模板。

可以约束函数模板、类模板、变量模板和别名模板。

普通的约束通常使用 requires 子句来指定。例如:

```
template<typename T>
void foo(const T& arg)
requires MyConcept<T>
...
```

在模板参数或 auto 前面，也可以直接使用概念作为类型约束:

```
template<MyConcept T>
void foo(const T& arg)
...
```

或者:

```
void foo(const MyConcept auto& arg)
...
```

4.2. 需求项

requires 子句使用关键字 requires 和编译时布尔表达式来限制模板的可用性。布尔表达式可以是:

- 编译时的布尔表达式
- 概念
- requires 表达式

可以使用布尔表达式的地方都可以使用有约束 (特别是以 if constexpr 作为条件的)。

4.2.1 requires 子句中使用 && 和 ||

要在 requires 子句中组合多个约束，可以使用运算符 &&。例如:

```
template<typename T>
requires (sizeof(T) > 4) // ad-hoc Boolean expression
      && requires { typename T::value_type; } // requires expression
```

```
      && std::input_iterator<T> // concept
void foo(T x) {
  ...
}
```

约束顺序无所谓。

还可以使用运算符 || 表示“可选”约束。例如:

```
template<typename T>
requires std::integral<T> || std::floating_point<T>
T power(T b, T p);
```

很少需要指定可选约束，也不应该随意指定，在 requires 子句中过度使用运算符 || 可能会增加编译资源的负担 (使编译明显变慢)。

单个约束还可以涉及多个模板参数，约束可以在多个类型 (或值) 之间施加影响。例如:

```
template<typename T, typename U>
requires std::convertible_to<T, U>
auto f(T x, U y) {
  ...
}
```

操作符 && 和 || 是唯一可以用来组合多个约束而不必使用括号的操作符。对于其他所有内容，使用括号 (正式地将一个特殊的布尔表达式传递给 requires 子句)。

4.3. 特别的布尔表达式

为模板制定约束的基本方法是使用 requires 子句:requires 后跟一个布尔表达式。requires 之后，约束可以使用编译时布尔表达式，而不仅是概念或 requires 表达式。这些表达可以使用:

- 类型谓词，如类型特征
- 编译时变量 (用 constexpr 或 constinit 定义)
- 编译时函数 (用 constexpr 或 consteval 定义)

来看一些使用特殊布尔表达式来限制模板可用性的例子:

- 当 int 和 long 的大小不同时:

```
template<typename T>
requires (sizeof(int) != sizeof(long))
...
```

- 仅当 sizeof(T) 不是太大时:

```
template<typename T>
requires (sizeof(T) <= 64)
...
```

- 仅当非类型模板参数 Sz 大于零时:

```
template<typename T, std::size_t Sz>
requires (Sz > 0)
...
```

- 仅对裸指针和 nullptr 时:

```
template<typename T>
requires (std::is_pointer_v<T> || std::same_as<T, std::nullptr_t>)
...
```

std::same_as 是一个新的标准概念，也可以使用标准类型特性 std::is_same_v<>:

```
template<typename T>
requires (std::is_pointer_v<T> || std::is_same_v<T, std::nullptr_t>)
...
```

- 当参数不能用作字符串时:

```
template<typename T>
requires (!std::convertible_to<T, std::string>)
...
```

std::convertible_to 是一个新的标准概念，也可以使用标准类型特性 std::is_convertible_v<>:

```
template<typename T>
requires (!std::is_convertible_v<T, std::string>)
...
```

- 仅当参数是指向整型值的指针 (或类指针对象) 时可用:

```
template<typename T>
requires std::integral<std::remove_reference_t<decltype(*std::declval<T>())>>
...
```

解引用操作符通常产生一个非整型的引用，所以:

  - 假设有一个类型为 T 的对象:std::declare<T>()
  - 为对象调用解引用操作符:*
  - 获取其类型:decltype()
  - 删除引用:std::remove _reference_v<>
  - 检查是否为整型:std::integral<>

约束也可以通过 std::optional<int> 进行。

std::integral 是一个新的标准概念，也可以使用标准类型特征:std::is_integral_v<>.

- 当非类型模板参数 Min 和 Max 的最大公约数 (GCD) 大于 1 时:

```
template<typename T>
constexpr bool gcd(T a, T b); // greatest common divisor (forward declaration)

template<typename T, int Min, int Max>
```

```
requires (gcd(Min, Max) > 1) // available if there is a GCD greater than 1
...
```

- 暂时禁用模板:

```
template<typename T>
requires false // disable the template
...
```

在一些特殊情况下，需要在整个表达式或部分表达式周围加上括号。若只使用标识符，可以将:: 和 <⋯> 与 && 和 || 组合使用。例如:

```
requires std::convertible_to<T, int> // no parentheses needed here
    &&
    (!std::convertible_to<int, T>) // ! forces the need for parentheses
```

4.4. 需求表达式

需求表达式 (不同于 requires 子句) 提供了一种简单而灵活的语法，用于在一个或多个模板参数上指定多个需求:

- 必需的类型定义
- 表达式必须有效
- 对表达式产生类型的要求

表达式以 requires 开头，后跟一个可选的参数列表，然后是一个需求块 (都以分号结束)。例如:

```
template<typename Coll>
... requires {
    typename Coll::value_type::first_type; // elements/values have first_type
    typename Coll::value_type::second_type; // elements/values have second_type
  }
```

可选参数列表允许引入一组“虚拟变量”，可用于在表达式的主体中表达需求:

```
template<typename T>
... requires(T x, T y) {
    x + y; // supports +
    x - y; // supports -
  }
```

这些形参永远不会由实参取代，所以通过值或引用声明都可以。

参数还允许引入子类型 (参数):

```
template<typename Coll>
... requires(Coll::value_type v) {
    std::cout << v; // supports output operator
  }
```

要求检查 Coll::value_type 是否有效，以及该类型的对象是否支持输出操作符。

此参数列表中的类型成员不必用 typename 限定。

当使用此方法检查 Coll::value_type 是否有效时，在需求块的主体中不需要任何内容，但不能为空，所以可以简单地使用 true:

```
template<typename Coll>
... requires(Coll::value_type v) {
    true; // dummy requirement because the block cannot be empty
  }
```

4.4.1 简单的需求

简单的需求就是必须格式良好的表达式，必须进行编译。调用不会执行，从而操作产生的结果并不重要。

```
template<typename T1, typename T2>
... requires(T1 val, T2 p) {
    *p; // operator* has to be supported for T2
    p[0]; // operator[] has to be supported for int as index
    p->value(); // calling a member function value() without arguments has to be
↪   possible
    *p > val; // support the comparison of the result of operator* with T1
    p == nullptr; // support the comparison of a T2 with a nullptr
  }
```

最后一个调用不要求 p 是 nullptr(要做到这一点，必须检查 T2 是否是类型 std::nullptr_t)。相反，可以要求将 T2 类型的对象与 nullptr 类型的对象进行比较。

通常使用运算符 || 没什么意义。一个简单的需求，例如

```
*p > val || p == nullptr;
```

不要求左子表达式或右子表达式可能成立，其规定了可以用运算符 || 组合两个子表达式结果的要求。

要满足这两个子表达式中的一个，必须使用:

```
template<typename T1, typename T2>
... requires(T1 val, T2 p) {
    *p > val; // support the comparison of the result of operator* with T1
  }
  || requires(T2 p) { // OR
    p == nullptr; // support the comparison of a T2 with nullptr
  }
```

注意，这个概念并不要求 T 是整型:

```
template<typename T>
... requires {
    std::integral<T>; // OOPS: does not require T to be integral
    ...
  };
```

概念只要求表达式 std::integral<T> 有效，这是所有类型的情况。T 是整型的要求必须这样表述:

```
template<typename T>
... std::integral<T> && // OK, does require T to be integral
  requires {
    ...
  };
```

或者:

```
template<typename T>
... requires {
    requires std::integral<T>; // OK, does require T to be integral
    ...
  };
```

4.4.2 类型的需求

类型需求是在使用类型名称时必须格式良好的表达式，所以名称必须定义为有效类型。

```
template<typename T1, typename T2>
... requires {
  typename T1::value_type; // type member value_type required for T1
  typename std::ranges::iterator_t<T1>; // iterator type required for T1
  typename std::common_type_t<T1, T2>; // T1 and T2 have to have a common type
}
```

对于所有类型需求，若类型存在但为空，则满足需求。

只能检查给定类型的名称 (类名、枚举类型的名称，来自 typedef 或 using)，不能使用类型检查其他类型声明:

```
template<typename T>
... requires {
  typename int; // ERROR: invalid type requirement
  typename T&; // ERROR: invalid type requirement
}
```

测试后者的方法是声明相应的形参:

```cpp
template<typename T>
... requires(T&) {
  true; // some dummy requirement
};
```

同样，需求检查使用传递的类型来定义另一个类型是否有效:

```cpp
template<std::integral T>
class MyType1 {
  ...
};

template<typename T>
requires requires {
  typename MyType1<T>; // instantiation of MyType1 for T would be valid
}
void mytype1(T) {
  ...
}

mytype1(42); // OK
mytype1(7.7); // ERROR
```

因此，以下需求不检查类型 T 是否存在标准哈希函数:

```cpp
template<typename T>
concept StdHash = requires {
  typename std::hash<T>; // does not check whether std::hash<> is defined for T
};
```

需要标准哈希函数的方法是尝试创建或使用:

```cpp
template<typename T>
concept StdHash = requires {
  std::hash<T>{}; // OK, checks whether we can create a standard hash function for T
};
```

注意，简单的需求只检查需求是否有效，而不检查需求是否满足。出于这个原因:

- 使用总是产生值的类型函数没有意义:

```cpp
template<typename T>
... requires {
  std::is_const_v<T>; // not useful: always valid (doesn't matter what it
↪  yields)
}
```

要检查 const 性，需要使用:

```
template<typename T>
... std::is_const_v<T> // ensure that T is const
```

requires 表达式中，可以使用嵌套的需求 (见下文)。

- 使用产生类型的类型函数没有意义:

```
template<typename T>
... requires {
  typename std::remove_const_t<T>; // not useful: always valid (yields a type)
}
```

需求只检查类型表达式是否产生类型，这总是正确的。

使用可能具有未定义行为的类型函数也没有意义。类型特性 std::make_unsigned<> 要求传递的参数是整型，而非 bool 型。若传递的类型不是整型，则有未定义行为，所以不应该使用 std::make_unsigned<> 作为需求，而不限制调用其类型:

```
template<typename T>
... requires {
  std::make_unsigned<T>::type; // not useful as type requirement (valid or undefined
  ↪ behavior)
}
```

这种情况下，需求只能满足或导致未定义行为 (这可能需求仍然满足)。相反，应该约束可以使用嵌套需求的类型 T:

```
template<typename T>
... requires {
  requires (std::integral<T> && !std::same_as<T, bool>);
  std::make_unsigned<T>::type; // OK
}
```

4.4.3 复合需求

复合需求允许将简单需求和类型需求结合起来，可以指定一个表达式 (大括号内)，然后添加以下一个或两个:

- noexcept 要求表达式保证不抛出异常
- -> type-constraint 将概念应用于表达式的求值

下面是一些例子:

```
template<typename T>
... requires(T x) {
  { &x } -> std::input_or_output_iterator;
  { x == x };
  { x == x } -> std::convertible_to<bool>;
```

```
  { x == x }noexcept;
  { x == x }noexcept -> std::convertible_to<bool>;
}
```

-> 之后的类型约束将结果类型作为其第一个模板参数:

- 第一个需求中，需求在对类型为 T 的对象使用运算符 & 时满足 std::input_or_output_iterator 的概念 (std::input_or_output_iterator<decltype(&x)> 产生 true)。
  也可以这样指定:

```
{ &x } -> std::is_pointer_v<>;
```

- 最后一个需求中，需求可以将两个 T 类型对象的 == 操作符的结果用作 bool(当将两个 T 类型对象和 bool 类型对象的 == 操作符的结果作为参数传递时，满足 std::convertible_to 的概念)。

需求表达式还可以表达对关联类型的需求:

```
template<typename T>
... requires(T coll) {
  { *coll.begin() } -> std::convertible_to<T::value_type>;
}
```

但不能使用嵌套类型指定类型需求，例如：不能使用它们来要求使用解引用操作符的返回值产生整数值。其中的问题是，返回值是一个引用，必须先解引用:

```
std::integral<std::remove_reference_t<T>>
```

而且不能在 requires 表达式的结果中，使用带有类型特性的嵌套表达式:

```
template<typename T>
concept Check = requires(T p) {
  { *p } -> std::integral<std::remove_reference_t<>>; // ERROR
  { *p } -> std::integral<std::remove_reference_t>; // ERROR
};
```

要么先定义相应的概念:

```
template<typename T>
concept UnrefIntegral = std::integral<std::remove_reference_t<T>>;

template<typename T>
concept Check = requires(T p) {
  { *p } -> UnrefIntegral; // OK
};
```

要么使用嵌套需求。

4.4.4 嵌套需求

嵌套需求可用于在 requires 表达式中指定附加约束。以 requires 开头，后跟一个编译时布尔表达式，该表达式本身也可能是或使用 requires 表达式。嵌套需求的好处是，可以确保编译时表达式 (使用所需表达式的参数或子表达式) 产生特定的结果，而不是仅仅确保表达式有效。

例如，考虑一个概念，必须确保对于给定类型，解引用和 [](下标) 操作符产生相同的类型。通过使用嵌套需求，可以这样指定:

```
template<typename T>
concept DerefAndIndexMatch = requires (T p) {
                requires std::same_as<decltype(*p),
                    decltype(p[0])>;
              };
```

好的方面是，对于“假设有一个 T 类型的对象”，这里有一个简单的语法，不必在这里使用 requires 表达式，但代码必须使用 std::declval<>():

```
template<typename T>
concept DerefAndIndexMatch = std::same_as<decltype(*std::declval<T>()),
       decltype(std::declval<T>()[0])>;
```

另一个例子，可以使用嵌套需求来解决刚才，在表达式上指定复杂类型需求的问题:

```
template<typename T>
concept Check = requires(T p) {
  requires std::integral<std::remove_cvref_t<decltype(*p)>>;
};
```

请注意 requires 表达式中的区别:

```
template<typename T>
... requires {
  !std::is_const_v<T>; // OOPS: checks whether we can call is_const_v<>
  requires !std::is_const_v<T>; // OK: checks whether T is not const
}
```

这里，使用的类型特性是 is_const_v<> 带/不带 requires，而只有第二个需求有用:

- 第一个表达式只要求检查 const 性并对结果取反有效，这个需求总是可满足 (即使 T 是 const int)。因为做这个检查总是有效的，所以这个需求毫无价值。
- 第二个需求表达式必须满足。若 T 是 int 则满足要求，但若 T 是 const int 则不满足。

4.5. 概念详解

通过定义概念，可以为一个或多个约束引入名称。

模板 (函数、类、变量和别名模板) 可以使用概念来约束其能力 (通过 requires 子句或作为模板参数的直接类型约束)，但概念也是布尔编译时表达式 (类型谓词)，可以在需要检查类型的地方使用 (if constexpr 条件中)。

4.5.1 定义概念

概念定义如下:

```
template< ... >
concept name = ... ;
```

等号是必需的 (不能在没有定义的情况下声明一个概念，也不能在这里使用大括号)。等号之后，可以指定可转换为 true 或 false 的编译时表达式。

概念很像 bool 类型的 constexpr 变量模板，但没有显式指定类型:

```
template<typename T>
concept MyConcept = ... ;

std::is_same<MyConcept< ... >, bool> // yields true
```

无论在编译时还是在运行时，都可以在需要布尔表达式值的地方使用一个概念。但不接受地址，因为其后面没有对象 (地址是一个纯右值)。

模板参数可能没有约束 (不能使用概念来定义概念)。

不能在函数中定义概念 (所有模板都是如此)。

4.5.2 概念的特殊能力

概念具有特殊的能力。

例如，考虑以下概念:

```
template<typename T>
concept IsOrHasThisOrThat = ... ;
```

与布尔变量模板的定义 (定义类型特征的常用方式) 相比:

```
template<typename T>
inline constexpr bool IsOrHasThisOrThat = ... ;
```

有以下不同:

- 概念并不表示代码，没有类型、存储、生命周期或与对象相关的其他属性。
  通过在编译时为特定模板参数实例化它们，实例化只会变为 true 或 false，所以可以在使用 true 或 false 的地方使用，可以得到这些字面量的所有属性。
- 概念不必声明为内联的，其隐式内联。
- 概念可以用作类型约束:

```
template<IsOrHasThisOrThat T>
...
```

变量模板不能这样使用。

- 概念是给约束命名的唯一方法，所以需要其来决定一个约束是否是另一个约束的特殊情况。
- 包含的概念。为了让编译器决定一个约束是否决定另一个约束 (因此是特殊的)，必须将约束公式化为概念。

4.5.3 非类型模板参数的概念

概念也可以应用于非类型模板参数 (NTTP)。例如:

```
template<auto Val>
concept LessThan10 = Val < 10;

template<int Val>
requires LessThan10<Val>
class MyType {
  ...
};
```

作为一个可用的例子，可以使用一个概念将非类型模板形参的值约束为 2 的幂:

lang/conceptnttp.cpp

```
#include <bit>

template<auto Val>
concept PowerOf2 = std::has_single_bit(static_cast<unsigned>(Val));

template<typename T, auto Val>
requires PowerOf2<Val>
class Memory {
  ...
};

int main()
{
  Memory<int, 8> m1; // OK
  Memory<int, 9> m2; // ERROR
  Memory<int, 32> m3; // OK
  Memory<int, true> m4; // OK
  ...
}
```

概念 PowerOf2 接受一个值而不是类型作为模板参数 (这里使用 auto，不需要指定类型):

```
template<auto Val>
concept PowerOf2 = std::has_single_bit(static_cast<unsigned>(Val));
```

当新的标准函数 std::has_single_bit() 对传递的值产生 true(只设置一个位意味着值是 2 的幂) 时，这个概念就满足了，std::has_single_bit() 要求我们有一个无符号整型值。通过强制转换为 unsigned，开发者可以传递有符号整型值，并拒绝不能转换为无符号整型值的类型。

接下来，使用这个概念要求 Memory 类只接受 2 的幂次大小和类型:

```
template<typename T, auto Val>
requires PowerOf2<Val>
class Memory {
  ...
};
```

注意，不能这样写:

```
template<typename T, PowerOf2 auto Val>
class Memory {
  ...
};
```

这就对 Val 类型提出了要求，但概念 PowerOf2 不约束其类型，并限制了值。

4.6. 使用概念作为类型约束

如前所述，概念可以用作类型约束。可以在不同的地方使用类型约束:

- 模板类型参数的声明中
- 用 auto 声明的调用参数的声明中
- 作为复合需求中的一个需求

例如:

```
template<std::integral T> // type constraint for a template parameter
class MyClass {
  ...
};

auto myFunc(const std::integral auto& val) { // type constraint for an auto parameter
  ...
};

template<typename T>
concept MyConcept = requires(T x) {
    { x + x } -> std::integral; // type constraint for return type
  };
```

使用一元约束，对表达式返回的单个参数或类型调用一元约束。

具有多参数的类型约束

也可以使用带有多个参数的约束，将参数类型或返回值用作第一个参数:

```cpp
template<std::convertible_to<int> T> // conversion to int required
class MyClass {
  ...
};

auto myFunc(const std::convertible_to<int> auto& val) { // conversion to int required
  ...
};

template<typename T>
concept MyConcept = requires(T x) {
    { x + x } -> std::convertible_to<int>; // conversion to int required
  };
```

另一个经常使用的例子是约束可调用对象 (函数、函数对象、Lambda) 的类型，可以使用 std::invocable 或 std::regular_invocable 概念传递一定数量的特定类型的参数，例如：要求传递一个接受 int 和 std::string 的操作，则必须声明:

```cpp
template<std::invocable<int, std::string> Callable>
void call(Callable op);
```

或:

```cpp
void call(std::invocable<int, std::string> auto op);
```

std::invocable 和 std::regular_invocable 的区别在于后者保证不修改传递的操作和参数。这是一种语义上的差异，只有助于记录意图，所以只使用 std::invocable。

类型约束和 auto

类型约束可以在所有可以使用 auto 的地方使用，该特性的主要应用是对用 auto 声明的函数参数使用类型约束。例如:

```cpp
void foo(const std::integral auto& val)
{
  ...
}
```

也可以对 auto 使用类型约束:

- 约束声明:

```cpp
std::floating_point auto val1 = f(); // valid if f() yields floating-point value

for (const std::integral auto& elem : coll) { // valid if elements are integral
↪  values
```

```
  ...
}
```

- 约束返回类型:

```
std::copyable auto foo(auto) { // valid if foo() returns copyable value
  ...
}
```

- 约束非类型模板参数:

```
template<typename T, std::integral auto Max>
class SizedColl {
  ...
};
```

也适用于接受多个参数的概念:

```
template<typename T, std::convertible_to<T> auto DefaultValue>
class MyType {
  ...
};
```

另一个例子，请参阅对 Lambda 作为非类型模板参数的支持。

4.7. 用概念包含约束

两个概念可以有一个包含关系，可以指定一个概念，使其限制一个或多个其他概念。这样做的好处是，当两个约束都得到满足时，重载解析更倾向于使用约束较多的泛型代码，而不是使用约束较少的泛型代码。

例如，假设引入以下两个概念:

```
template<typename T>
concept GeoObject = requires(T obj) {
  { obj.width() } -> std::integral;
  { obj.height() } -> std::integral;
  obj.draw();
};

template<typename T>
concept ColoredGeoObject =
  GeoObject<T> && // subsumes concept GeoObject
  requires(T obj) { // additional constraints
    obj.setColor(Color{});
    { obj.getColor() } -> std::convertible_to<Color>;
  };
```

因为它显式地规定了类型 T 也必须满足概念 GeoObject 的约束，所以概念 ColoredGeoObject 显式地包含了概念 GeoObject。

当为两个概念重载模板并且两个概念都满足时，不会得到歧义错误，重载解析更倾向于包含其他概念的概念:

```
template<GeoObject T>
void process(T) // called for objects that do not provide setColor() and getColor()
{
  ...
}

template<ColoredGeoObject T>
void process(T) // called for objects that provide setColor() and getColor()
{
  ...
}
```

约束包含仅在使用概念时起作用。当一个概念/约束比另一个更特殊时，不存在自动包含。

约束和概念不会仅仅基于需求进行包含:

```
// declared in a header file:
template<typename T>
concept GeoObject = requires(T obj) {
            obj.draw();
          };

// declared in another header file:
template<typename T>
concept Cowboy = requires(T obj) {
          obj.draw();
          obj = obj;
        };
```

假设为 GeoObject 和 Cowboy 重载函数模板:

```
template<GeoObject T>
void print(T) {
  ...
}

template<Cowboy T>
void print(T) {
  ...
}
```

对于 Circle 或 Rectangle(都有 draw() 成员函数)，我们不希望这样，因为 Cowboy 的概念更特殊，对 print() 的调用更倾向于对 Cowboy 的 print() 调用，所以希望看到在本例中有两个可能的 print() 函数发生冲突。

检查假设的工作只针对概念进行评估。若没有使用概念，则具有不同约束的重载不明确:

```
template<typename T>
requires std::is_convertible_v<T, int>
void print(T) {
  ...
}

template<typename T>
requires (std::is_convertible_v<T, int> && sizeof(int) >= 4)
void print(T) {
  ...
}

print(42); // ERROR: ambiguous (if both constraints are true)
```

当使用概念时，下面的代码可以工作:

```
template<typename T>
requires std::convertible_to<T, int>
void print(T) {
  ...
}

template<typename T>
requires (std::convertible_to<T, int> && sizeof(int) >= 4)
void print(T) {
  ...
}

print(42); // OK
```

产生这种行为的一个原因是，处理概念之间的依赖关系需要更多编译的时间。

C++ 标准库提供的概念经过精心设计，以便在有意义时包含其他概念。事实上，标准概念构建了一个相当复杂的包含图。例如:

- std::random_access_range 包含了 std::bidirectional_range，两者都包含了 std::forward_range，三者都包含了 std::input_range，所以都包含了 std::range。

  但是，std::sized_range 只包含 std::range，而不包含其他的概念。
- std::regular 包含了 std::semiregular，而两者都包含了 std::copyable 和 std::default_initializable(其中包含了其他几个概念，如 std::movable、std::copy_constructible 和 std::destructible)。
- std::sortable 包含 std::permutable，两者都包含 std::indirectly_swappable，这两个参数都是相同的类型。

4.7.1 间接包含

约束甚至可以间接包含，所以重载解析仍然可以选择一个重载或特化，而非另一个 (即使其约束不是根据彼此定义的)。

例如，假设已经定义了以下两个概念:

```
template<typename T>
concept RgSwap = std::ranges::input_range<T> && std::swappable<T>;

template<typename T>
concept ContCopy = std::ranges::contiguous_range<T> && std::copyable<T>;
```

当为这两个概念重载两个函数，并传递一个同时适合这两个概念的对象时，这并不具有二义性:

```
template<RgSwap T>
void foo1(T) {
  std::cout << "foo1(RgSwap)\n";
}

template<ContCopy T>
void foo1(T) {
  std::cout << "foo1(ContCopy)\n";
}

foo1(std::vector<int>{}); // OK: both fit, ContCopy is more constrained
```

原因是 ContCopy 包含 RgSwap:

- contiguous_range 概念是根据 input_range 概念定义。
  (包含 random_access_range，包含 bidirectional_range，包含 forward_range，包含 input_range。)
- copyable 概念是根据可交换的概念来定义。
  (包含 movable，也就包含着可互换。)

使用以下声明，当两个概念都适合时，就会产生歧义:

```
template<typename T>
concept RgSwap = std::ranges::sized_range<T> && std::swappable<T>;

template<typename T>
concept ContCopy = std::ranges::contiguous_range<T> && std::copyable<T>;
```

原因是 contiguous_range 和 sized_range 这两个概念都不包含对方。

此外，对于下列声明，没有一个概念包含另一个概念:

```
template<typename T>
concept RgCopy = std::ranges::input_range<T> && std::copyable<T>;

template<typename T>
concept ContMove = std::ranges::contiguous_range<T> && std::movable<T>;
```

一方面，ContMove 更受约束，因为 contiguous_range 包含 input_range。另一方面，因为 copyable 包含着 movable，所以 RgCopy 更受约束。

为了避免混淆，不要对相互包含的概念做太多的假设。若有疑问，请指定所需的所有具体概念。

4.7.2 定义交换概念

要正确地实现假设，必须非常谨慎。一个例子是 std::same_as 概念的实现，检查两个模板形参是否具有相同的类型。

为了理解为什么定义概念并不简单，假设定义了概念 SameAs:

```
template<typename T, typename U>
concept SameAs = std::is_same_v<T, U>; // define concept SameAs
```

这个定义对于这样的情况足够了:

```
template<typename T, typename U>
concept SameAs = std::is_same_v<T, U>; // define concept SameAs

template<typename T, typename U>
requires SameAs<T, U> // use concept SameAs
void foo(T, U)
{
  std::cout << "foo() for parameters of same type" << '\n';
}

template<typename T, typename U>
requires SameAs<T, U> && std::integral<T> // use concept SameAs again
void foo(T, U)
{
  std::cout << "foo() for integral parameters of same type" << '\n';
}

foo(1, 2); // OK: second foo() preferred
```

这里，foo() 的第二个定义包含了第一个定义，当传递了两个相同整型的参数，并且两个 foo() 的约束都满足时，会调用第二个 foo()。

但当在第二个 foo() 的约束中调用 SameAs<> 时，若稍微修改参数会发生什么:

```
template<typename T, typename U>
requires SameAs<T, U> // use concept SameAs
void foo(T, U)
{
  std::cout << "foo() for parameters of same type" << '\n';
}

template<typename T, typename U>
```

```
requires SameAs<U, T> && std::integral<T> // use concept SameAs with other order
void foo(T, U)
{
  std::cout << "foo() for integral parameters of same type" << '\n';
}

foo(1, 2); // ERROR: ambiguity: both constraints are satisfied without one subsuming
↪  the other
```

问题是编译器无法检测到 SameAs<> 是可交换的。对于编译器来说，模板参数的顺序很重要，所以第一个需求不一定是第二个需求的子集。

为了解决这个问题，必须以一种不影响参数顺序的方式来设计 SameAs 概念，这里需要一个辅助概念:

```
template<typename T, typename U>
concept SameAsHelper = std::is_same_v<T, U>;

template<typename T, typename U>
concept SameAs = SameAsHelper<T, U> && SameAsHelper<U, T>; // make commutative
```

对于 IsSame<>，参数的顺序不再重要:

```
template<typename T, typename U>
requires SameAs<T, U> // use SameAs
void foo(T, U)
{
  std::cout << "foo() for parameters of same type" << '\n';
}

template<typename T, typename U>
requires SameAs<U, T> && std::integral<T> // use SameAs with other order
void foo(T, U)
{
  std::cout << "foo() for integral parameters of same type" << '\n';
}

foo(1, 2); // OK: second foo() preferred
```

编译器可以发现第一个构建块 SameAs<U,T> 是 SameAs 定义的子概念的一部分，因此其他构建块 SameAs<T,U> 和 std::integral<T> 是扩展，所以第二个 foo() 更优先使用。

C++20 标准库提供的概念 (请参见 std::same_as) 中就包含了类似这样的细节，所以应该在适合需求时使用，而不是定义自己的概念。

```
template<typename T, typename U>
requires std::same_as<T, U> // standard same_as<> is commutative
void foo(T, U)
{
```

```
  std::cout << "foo() for parameters of same type" << '\n';
}

template<typename T, typename U>
requires std::same_as<U, T> && std::integral<T> // so different order does not matter
void foo(T, U)
{
  std::cout << "foo() for integral parameters of same type" << '\n';
}

foo(1, 2); // OK: second foo() preferred
```

对于自定义的概念，请考虑尽可能更多的使用方法 (和滥用)，越多越好。就像任何好软件一样，概念也需要好的设计和测试用例。

第 5 章 详细介绍标准概念

本章详细描述了 C++20 标准库的所有概念。

5.1. 所有标准概念的概述

“类型和对象基本概念”表列出了类型和对象的基本概念。

“范围、迭代器和算法概念”表列出了范围、视图、迭代器和算法的概念。

“辅助概念”表列出的概念主要用作其他概念的构建块，通常不会让应用程序开发者直接使用。

5.1.1 头文件和命名空间

标准概念在不同的头文件中定义:

- 头文件 <concepts> 中定义了许多基本概念，其中包括 <ranges> 和 <iterator>。
- 迭代器的概念在头文件 <iterator> 中定义。
- 范围的概念在头文件 <ranges> 中定义。
- <compare> 中定义了 three_way_comparable 的概念 (几乎包含在所有其他头文件中)。
- uniform_random_bit_generator 在 <random> 中定义。

几乎所有的概念都定义在命名空间 std 中，唯一的例外是 range 概念，它定义在命名空间 std::ranges 中。

概念	约束
integral	整型
signed_integral	有符号整型
unsigned_integral	无符号整型
floating_point	浮点类型
movable	支持移动初始化/赋值和交换
copyable	支持移动和复制初始化/赋值和交换
semiregular	支持默认初始化、复制、移动和交换
regular	支持默认初始化、复制、移动、交换和相等比较
same_as	相同类型
convertible_to	类型可转换为另一类型
derived_from	从另一类型派生的类型
constructible_from	可从其他类型构造的类型
assigneable_from	可从另一类型赋值的类型
swappable_with	类型可与其他类型交换
common_with	两种类型有一个共同的类型
common_reference_with	两个类型有一个共同的引用类型

equality_comparable	类型支持相等性检查
equality_comparable_with	可以检查两种类型是否相等
totally_ordered	类型支持严格的弱排序
totally_ordered_with	是否可以检查两种类型的严格弱排序
three_way_comparable	可以应用所有比较运算符 (包括运算符 <=>)
three_way_comparable_with	可以使用所有比较操作符 (包括 <=>)
invocable	类型是指定参数的可调用对象
regular_invocable	类型是指定参数的可调用对象 (无修改)
predicate	类型是一个谓词 (返回布尔值的可调用对象)
relation	可调用类型定义了两个类型之间的关系
equivalence_relation	可调用类型定义了两个类型之间的相等关系
strict_weak_order	可调用类型定义了两个类型之间的排序关系
uniform_random_bit_generator	可调用类型可以用作随机数生成器

表 5.1 类型和对象的基本概念

概念	约束
default_initializable	类型可默认初始化
move_constructible	类型支持移动初始化
copy_constructible	类型支持复制初始化
destructible	类型是可销毁
swappable	类型可交换
weakly_incrementables	类型支持自增操作符
incrementable	类型支持保持相等的递增运算符

表 5.2 辅助概念

概念	约束
range	类型是一个范围
output_range	类型是要写入的范围
input_range	类型是要读取的范围
forward_range	类型是一个可多次读取的范围
bidirectional_range	类型是向前和向后读取的范围
random_access_range	类型是一个支持在元素上跳跃的范围
contiguous_range	类型是在连续内存中包含元素的范围
sized_range	类型是一个大小支持范围
common_range	类型是一系列迭代器和相同类型的哨兵
borrowed_range	类型是左值或借用的范围
view	类型是视图
vieable_range	类型或可以转换为视图
indirectly_writable	类型可用于写入它所引用的位置
indirectly_readable	类型可用于从其所引用的位置进行读取
indirectly_movable	类型是指可移动的对象
indirectly_movable_storable	类型是指支持临时的可移动对象
indirectly_copyable	类型指的是可复制对象
indirectly_copyable_storable	类型是指支持临时对象的可复制对象
indirectly_swappable	类型指的是可交换的对象
indirectly_comparable	类型指的是可比较的对象
input_output_iterator	类型是一个迭代器
output_iterator	类型是一个输出迭代器
input_iterator	类型 (至少) 是输入迭代器
forward_iterator	类型 (至少) 是前向迭代器
bidirectional_iterator	类型 (至少) 是一个双向迭代器
random_access_iterator	类型 (至少) 是一个随机访问迭代器
contiguous_interator	类型是指向连续内存中元素的迭代器
sentinel_for	类型可以用作迭代器类型的标记
sized_sentinel_for	类型可以用作迭代器类型的哨点，距离计算的成本较低
permutable	类型 (至少) 是一个前向迭代器，可以对元素重新排序
mergeable	可以使用两种类型将排序后的元素合并为第三种类型
sortable	类型是可排序的 (根据比较和投影)
indirectly_unary_invocable	操作可以用迭代器的值类型调用
indirectly_regular_unary_invocable	无状态操作可以用迭代器的值类型调用
indirect_unary_predicate	一元谓词可以用迭代器的值类型调用
indirect_binary_predicate	二元谓词可以用两个迭代器的值类型调用
indirect_equivalence_relation	谓词可用于检查传递的迭代器的两个值是否相等
indirect_strict_weak_order	谓词可用于对传递的迭代器的两个值排序。

表 5.3 范围、迭代器和算法的概念

5.1.2 标准概念的包含

C++ 标准库提供的概念经过精心设计，以便包含其他概念，其建立了一个相当复杂的包含图。图 5.1 展示了其复杂程度。

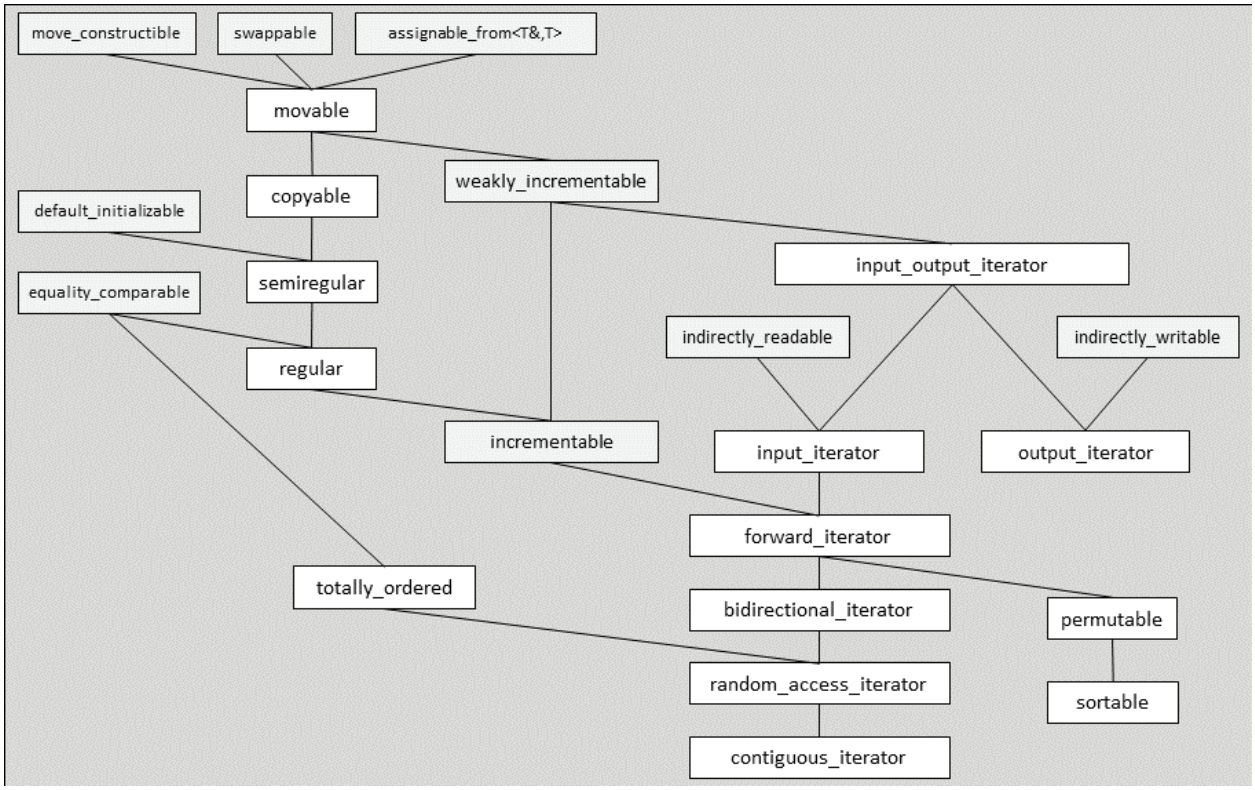

图 5.1 C++ 标准概念的包含图 (摘录)

出于这个原因，概念的描述列出了包含哪些其他关键概念。

5.2. 语言相关的概念

本节列出一般适用于对象和类型的概念。

5.2.1 算法概念

std::integral<T>

- 保证类型 T 是整型 (包括 bool 和所有字符类型)。
- 要求:
  - 类型特性 std::is_integral_v<T> 为 true

std::signed_integral<T>

- 保证类型 T 是有符号整型 (包括有符号字符类型，可能是 char)。
- 要求:
  - 需要满足 std::integral<T>
  - 类型特性 std::is_signed_v<T> 为 true

std::unsigned_integral<T>

- 保证类型 T 是无符号整型 (包括 bool 和无符号字符类型，可能是 char)。
- 要求:
  - 需要满足 std::integral<T>
  - 类型特性 std::is_signed_v<T> 为 true

std::floating_point<T>

- 保证类型 T 是浮点类型 (float、double 或 long double)。
- 引入这个概念是为了能够定义数学常数。
- 要求:
  - 类型特性 std::is_floating_point_v<T> 为 true

5.2.2 对象概念

对于对象 (既不是引用，也不是函数，也不是 void 的类型)，有一个所需基本操作的层次结构。

std::movable<T>

- 保证类型 T 是可移动和可交换的。可以移动构造、移动赋值，以及与该类型的另一个对象交换。
- 要求:
  - 需要满足 std::move_constructible<T>
  - 需要满足 std::assignable_from<T&, T>
  - 需要满足 std::swappable<T>
  - T 既不是引用，也不是函数，也不是 void

std::copyable<T>

- 保证类型 T 是可复制的 (可移动和可交换)。

- 要求:
  - 需要满足 std::movable<T>
  - 需要满足 std::copy_constructible<T>
  - std::assignable_from 用于任意 T，T&，const T，以及 const T& 转换为 T&
  - 需要满足 std::swappable<T>
  - T 既不是引用，也不是函数，也不是 void

std::semiregular<T>

- 保证类型 T 是半常规的类型 (可以默认初始化、复制、移动和交换)。
- 要求:
  - 需要满足 std::copyable<T>
  - 需要满足 std::default_initializable<T>
  - 需要满足 std::movable<T>
  - 需要满足 std::copy_constructible<T>
  - std::assignable_from 用于任意 T，T&，const T，以及 const T& 转换为 T&
  - 需要满足 std::swappable<T>
  - T 既不是引用，也不是函数，也不是 void

std::regular<T>

- 保证类型 T 是常规的 (可以默认初始化、复制、移动、交换和检查相等性)。
- 要求:
  - 需要满足 std::semiregular<T>
  - 需要满足 std::equality_comparable<T>
  - 需要满足 std::copyable<T>
  - 需要满足 std::default_initializable<T>
  - 需要满足 std::movable<T>
  - 需要满足 std::copy_constructible<T>
  - std::assignable_from 用于任意 T，T&，const T，以及 const T& 转换为 T&
  - 需要满足 std::swappable<T>
  - T 既不是引用，也不是函数，也不是 void

5.2.3 概念间的关系类型

std::same_as<T1, T2>

- 保证 T1 和 T2 类型相同。

- 概念两次调用 std::is_same_v 类型特性，以确保与参数的顺序无关。
- 要求:
  - 类型特性 std::is_same_v<T1, T2> 为 true
  - 与 T1 和 T2 的顺序无关

std::convertible_to<From, To>

- 保证 From 类型的对象隐式和显式可转换为 To 类型的对象。
- 要求:
  - 类型特性 std::is_convertible_v<From, To> 为 true
  - 支持从 From 到 To 的 static_cast
  - 可以将 From 类型的对象作为 To 返回

std::derived_from<D, B>

- 保证 D 类型是从 B 类型公开派生的 (或者 D 和 B 是相同的)，D 类型的指针都可以转换为 B 类型的指针。换句话说: 保证了 D 引用/指针可以用作 B 引用/指针。
- 要求:
  - 类型特性 std::is_base_of_v<B, D> 为 true
  - 对于类型 D 到 B 的 const 指针，类型特性 std::is_convertible_v 为 true

std::constructible_from<T, Args...>

- 保证可以用 Args…类型的参数初始化一个 T 类型的对象。
- 要求:
  - 需要满足 std::destructible<T>
  - 类型特性 std::is_constructible_v<T, Args...> 为 true

std::assignable_from<To, From>

- 保证可以将 From 类型的值移动或复制赋值给 To 类型的值。
  赋值还必须产生原始 To 对象。
- 要求:
  - To 必须是一个左值引用
  - std::common_reference_with<To, From> 满足类型的 const 左值引用
  - 必须支持赋值运算符，并产生与 To 相同的类型

std::swappable_with<T1, T2>

- 保证类型 T1 和 T2 的值可以交换。
- 要求:
  - 需要满足 std::common_reference_with<T1, T2>
  - 任意两个 T1 和 T2 类型的对象，都可以通过使用 std::ranges::swap() 交换值。

std::common_with<T1, T2>

- 保证类型 T1 和 T2 共享一个可以显式转换为的公共类型。
- 要求:
  - 类型特性 std::common_type_t<T1, T2> 会产生一个类型
  - 其通用类型支持 static_cast
  - 这两种类型的引用共享一个 common_reference 类型
  - 与 T1 和 T2 的顺序无关

std::common_reference_with<T1, T2>

- 保证类型 T1 和 T2 共享一个 common_reference 类型，可以显式和隐式地转换为该类型。
- 要求:
  - 类型特性 std::common_reference_t<T1, T2> 会产生一个类型
  - 这两种类型可通过 std::convertible_to 转换为公共引用类型
  - 与 T1 和 T2 的顺序无关

5.2.4 比较概念

std::equality_comparable<T>

- 保证类型 T 的对象可使用 == 和!= 操作符比较，与顺序无关。
- T 的 == 操作符应该是对称的和可传递的:
  - 当 t2==t1 时，t1==t2 为真
  - 若 t1==t2 和 t2==t3 为真，则 t1==t3 为真

  但这是在编译时无法检查的语义约束。
- 要求:
  - == 和!= 操作符都受支持，并产生可转换为 bool 的值

std::equality_comparable_with<T1, T2>

- 保证类型 T1 和 T2 的对象可以使用 == 和!= 操作符进行比较。

- 要求:
  - 对于涉及 T1 和/或 T2 对象的所有比较，都支持 == 和!=，并产生可转换为 bool 的相同类型的值

std::totally_ordered<T>

- 保证类型 T 的对象可与 ==、!=、<、<=、> 和 >= 操作符进行比较，以便两个值总是相互等于、小于或大于对方。
- 这个概念并没有声称类型 T 具有所有值的总顺序，即使浮点值没有顺序，但表达式 std::totally_ordered<double> 的结果仍然为 true:

```
std::totally_ordered<double> // true
std::totally_ordered<std::pair<double, double>> // true
std::totally_ordered<std::complex<int>> // false
```

  提供这个概念是为了检查能够对元素进行排序的正式需求，因使用了 std::ranges::less，这是排序算法的默认排序条件。若类型没有所有值的总顺序，排序算法就不会编译失败，支持所有六个基本比较运算符就足够了。要排序的值应该是全序或弱序，但这是在编译时无法检查的语义约束。
- 要求:
  - 需要满足 std::equality_comparable<T>
  - 与 ==、!=、<、<=、> 和 >= 操作符的所有比较都会产生可转换为 bool 的值

std::totally_ordered_with<T>

- 保证类型 T1 和 T2 的对象与 ==、!=、<、<=、> 和 >= 操作符具有可比性，以便两个值总是相互等于、小于或大于对方。
- 要求:
  - 对于涉及 T1 和/或 T2 对象的所有比较，都支持 ==，!=，<，<=，> 和 >= 操作符，并产生可转换为 bool 的相同类型的值

std::three_way_comparable<T>, std::three_way_comparable<T, Cat>

- 保证类型 T 的对象与 ==、!=、<、<=、>、>= 和 <=> 操作符可比较 (并且至少具有比较类别类型 Cat)。若没有传递 Cat，则需要 std::partial_ordering。
- 这个概念在头文件 <compare> 中定义。
- 注意，这个概念并不包含 std::equality_comparable，因为后者要求 == 操作符只对两个相等的对象产生真值。对于弱序或偏序，情况可能不是这样。
- 注意，这个概念并不暗示和包含 std::totally_ordered，因为后者要求比较类别是 std::strong_ordering 或 std::weak_ordering。

- 要求:
  - 与 ==、!=、<、<=、> 和 >= 操作符的所有比较都会产生可转换为 bool 的值
  - 与 <=> 操作符的任何比较都会产生一个比较类别 (至少是 Cat)。

std::three_way_comparable_with<T1, T2>, std::three_way_comparable_with<T1, T2, Cat>

- 保证任意两个 T1 和 T2 类型的对象都可以与 ==、!=、<、<=、>、>= 和 <=> 操作符进行比较 (并且至少具有比较类别类型 Cat)。若没有传递 Cat，则需要 std::partial_ordering。
- 这个概念在头文件 <compare> 中定义。
- 注意，这个概念并不包含 std::equality_comparable_with，因为后者要求 == 操作符只对两个相等的对象产生真值。对于弱序或偏序，情况可能并不是这样。
- 注意，这个概念并不暗示和包含 std::totally_ordered_with，因为后者要求比较类别是 std::strong_ordering 或 std::weak_ordering。
- 要求:
  - std::three_way_comparable 满足 T1 和 T2(以及 Cat) 的值和公共引用
  - 与 ==、!=、<、<=、> 和 >= 操作符的所有比较都会产生可转换为 bool 的值
  - 与 <=> 操作符的任何比较都会产生一个比较类别 (至少是 Cat)。
  - 与 T1 和 T2 的顺序无关

5.3. 迭代器和范围的概念

本节列出迭代器和范围的所有基本概念，在算法和类似函数中都很有用。

注意，范围的概念是在命名空间 std::ranges 中提供的 (不是在 std 中)，并在头文件 <ranges> 中声明。

迭代器的概念在头文件 <iterator> 中声明。

5.3.1 范围和视图的概念

定义了几个概念来约束范围，其与迭代器的概念相对应，并处理了自 C++20 以来新的迭代器类别。

std::ranges::range<Rg>

- 保证 Rg 是一个有效的范围。
- 这个概念在头文件 <compare> 中定义。
- Rg 类型的对象支持通过使用 std::ranges::begin() 和 std::ranges::end() 对元素进行迭代。
  若范围是一个数组，或者提供 begin() 和 end() 成员，或与独立的 begin() 和 end() 函数一起使用，就会出现这种情况。
- 此外，对于 std::ranges::begin() 和 std::ranges::end()，适用以下约束:
  - 必须在 (平均的) 常数时间内运行。

  - 不修改范围。
  - begin() 在多次调用时产生相同的位置 (除非该范围至少不提供前向迭代器)。

  以良好的性能迭代所有元素 (甚至多次迭代，除非有纯输入迭代器)。
- 要求:
  - 对于 Rg 类型的对象 rg，支持 std::ranges::begin(rg) 和 std::ranges::end(rg)

std::ranges::output_range<Rg, T>

- 保证 Rg 可提供接受 T 类型值的输出迭代器 (可以用来编写的迭代器) 的范围。
- 要求:
  - 需要满足 std::range<Rg>
  - 迭代器类型满足 std::output_iterator 和 T

std::ranges::input_range<Rg>

- 保证 Rg 是一个输入迭代器 (可用于读取的迭代器) 的范围。
- 要求:
  - 需要满足 std::range<Rg>
  - 迭代器类型满足 std::input_iterator

std::ranges::forward_range<Rg>

- 保证 Rg 是一个前向迭代器 (可用于读写和多次迭代的迭代器) 的范围。
- 注意，迭代器的 iterator_category 成员可能不匹配。对于产生左值的迭代器，是 std::input_iterator_tag(若可用)。
- 要求:
  - 需要满足 std::input_range<Rg>
  - 迭代器类型满足 std::forward_iterator

std::ranges::bidirectional_range<Rg>

- 保证 Rg 是一个双向迭代器的范围 (可以用于读写和向后迭代的迭代器)。
- 注意，迭代器的 iterator_category 成员可能不匹配。对于产生左值的迭代器，是 std::input_iterator_tag(若可用)。
- 要求:
  - 需要满足 std::forward_range<Rg>
  - 迭代器类型满足 std::bidirectional_iterator

std::ranges::random_access_range<Rg>

- 保证 Rg 是一个提供随机访问迭代器 (可用于读写、前后跳转和计算距离的迭代器) 的范围。
- 注意，迭代器的 iterator_category 成员可能不匹配。对于产生左值的迭代器，是 std::input_iterator_tag(若可用)。
- 要求:
  - 需要满足 std::bidirectional_range<Rg>
  - 迭代器类型满足 std::random_access_iterator

std::ranges::contiguous_range<Rg>

- 保证 Rg 是一个范围，该范围提供随机访问迭代器，并附加约束元素存储在连续内存中。
- 注意，迭代器的 iterator_category 成员不匹配。若定义了类别成员，则只有 std::random_access_iterator_tag，若值是右值，甚至只有 std::input_iterator_tag。
- 要求:
  - 需要满足 std::random_access_range<Rg>
  - 迭代器类型满足 std::contiguous_iterator
  - std::ranges::data() 将生成指向第一个元素的裸指针

std::ranges::sized_range<Rg>

- 保证 Rg 是可以在常量时间内计算出元素数量的范围 (通过调用成员 size() 或通过计算开始和结束之间的差值)。
- 若满足这个概念，std::ranges::size() 对于 Rg 类型的对象是定义良好的。
- 注意，这个概念的性能方面是一个语义约束，不能在编译时检查。为了表明一个类型即使提供了 size() 也不满足这个概念，可以定义 std::disable_size_range<Rg> 产生 true[disable_size_range 的真正含义是“忽略 size_range 的 size()”。]。
- 要求:
  - 需要满足 std::range<Rg>
  - 支持 std::ranges::size()

std::ranges::common_range<Rg>

- 保证 Rg 是一个范围，其中开始迭代器和哨兵 (结束迭代器) 具有相同的类型。
- 保证是由:
  - 所有标准容器 (vector, list 等)
  - empty_view
  - single_view

  - common_view

  保证不是由:

  - 查看视图
  - 删除 const 视图
  - 没有最终值或最终值为不同类型的 iota 视图

  对于其他视图，其取决于底层范围的类型。
- 要求:
  - 需要满足 std::range<Rg>
  - std::ranges::iterator_t<Rg> 和 std::ranges::sentinel_t<Rg> 具有相同类型

std::ranges::borrowed_range<Rg>

- 保证在当前上下文中传递的范围 Rg，产生即使该范围不再存在也可以使用的迭代器。若传递左值或传递的范围始终是租借范围，则满足该概念。
- 若满足这个概念，则范围的迭代器与范围的生命周期无关。所以当创建迭代器的范围被销毁时，迭代器不能挂起。但若范围的迭代器指向基础范围，并且基础范围不再存在，则其仍然可以悬挂。
- 形式上，若 Rg 是左值 (有名称的对象)，或者变量模板 std::ranges::enable_borrow_range<Rg> 为真，则满足该概念，以下视图都是如此:subrange、ref_view、string_view、span、iota_view 和 empty_view。
- 要求:
  - 需要满足 std::range<Rg>
  - Rg 是左值或 enable_borrow_range<Rg> 产生 true

std::ranges::view<Rg>

- 保证 Rg 是一个视图 (复制、移动、赋值和销毁成本很低的范围)。
- 视图有以下要求:[最初，C++20 要求视图也应该是默认可构造的，并且销毁应该是低成本的，但http://wg21.link/p2325r3和http://wg21.link/p2415r2删除了这些要求。]
  - 必须是一个范围 (支持对元素的迭代)。
  - 必须可移动。
  - 移动构造函数和复制构造函数 (若可用) 必须具有恒定的复杂性。
  - 移动赋值运算符和复制赋值运算符 (若可用的话)，必须是低成本的 (保持不变的复杂度，或者不比销毁加创建的复杂度更低)。

  除最后一个要求外，所有要求都通过相应的概念进行检查。最后一个要求是语义约束，必须由类型的实现者通过从 std::ranges::view_interface 公开派生或特化 std::ranges::enable_view<Rg> 生成 true 来保证。
- 要求:
  - 需要满足 std::range<Rg>

  - 需要满足 std::movable<Rg>
  - 变量模板 std::ranges::enable_view<Rg> 为 true

std::ranges::viewable_range<Rg>

- 保证 Rg 是一个可以使用 std::views::all() 适配器安全地转换为视图的范围。
- 若 Rg 已经是一个视图或一个范围的左值，或一个可移动的右值，或一个范围但不是初始化列表，则满足这个概念。[对于仅移动视图类型的左值，即使满足 viewable_range，也存在 all() 格式错误的问题。]
- 要求:
  - 需要满足 std::range<Rg>

5.3.2 类指针对象的概念

本节列出了对象的所有标准概念，可以使用解引用操作符来处理它们所指向的值。这通常适用于裸指针、智能指针和迭代器，所以这些概念通常用作处理迭代器和算法的概念的基本约束。

注意，这些概念不需要支持操作符->。

这些概念在头文件 <iterator> 中声明。

基本概念的间接支持

std::indirectly_writable<P, Val>

- 保证 P 是一个类似指针的对象，支持操作符 * 来赋值 Val。
- 满足非 const 裸指针、智能指针和迭代器的要求，提供 Val 可以赋值给 P 所指的位置。

std::indirectly_readable<P>

- 保证 P 是一个类似指针的对象，支持解引用操作符的读访问方式。
- 满足裸指针，智能指针和迭代器。
- 要求:
  - 结果值对于 const 和非 const 对象具有相同的引用类型 (排除了 std::optional<>)，这确保了 P 的常量性不会传播到所指向的位置 (当操作符返回对成员的引用时通常不会出现这种情况)。
  - std::iter_value_t<P> 必须有效，该类型不必支持操作符->。

间接可读对象的概念

对于间接可读的类指针概念，可以检查其他的约束:

std::indirectly_movable<InP,OutP>

- 保证 InP 的值可以直接赋值给 OutP。
- 有了这个概念，以下代码则有效:

```
void foo(InP inPos, OutP, outPos) {
  *outPos = std::move(*inPos);
}
```

- 要求:
  - 需要满足 std::indirectly_readable<InP>
  - std::indirectly_writable 满足从 InP 值到 OutP 的右值引用

std::indirectly_movable_storable<InP,OutP>

- 保证 InP 的值可以间接赋值给 OutP，即使在使用 OutP 指向的类型的 (临时) 对象时也是如此。
- 有了这个概念，以下代码则有效:

```
void foo(InP inPos, OutP, outPos) {
  OutP::value_type tmp = std::move(*inPos);
  *outPos = std::move(tmp);
}
```

- 要求:
  - 需要满足 std::indirectly_movable<InP, OutP>
  - std::indirectly_writable 满足 OutP 所引用对象的 InP 值
  - std::movable 满足 InP 所引用的值
  - InP 引用的右值是可复制/移动的、可构造的和可赋值的

std::indirectly_copyable<InP, OutP>

- 保证 InP 的值可以直接赋值给 OutP。
- 有了这个概念，以下代码则有效:

```
void foo(InP inPos, OutP outPos) {
  *outPos = *inPos;
}
```

- 要求:
  - 需要满足 std::indirectly_readable<InP>
  - std::indirectly_writable 满足于对 InP 值的引用 (OutP 值) 可赋值

std::indirectly_copyable_storable<InP, OutP>

- 保证 InP 的值可以间接赋值给 OutP，即使在使用 OutP 指向类型的 (临时) 对象时也是如此。
- 有了这个概念，以下代码则有效:

```
void foo(InP inPos, OutP outPos) {
  OutP::value_type tmp = *inPos;
  *outPos = tmp;
}
```

- 要求:
  - 需要满足 std::indirectly_copyable<InP, OutP>
  - std::indirectly_writable 满足 InP 值对 OutP 所引用对象的 const 左值和右值引用
  - std::copyable 满足 InP 引用的值
  - InP 引用的值是可复制/移动的、可构造的和可赋值的

std::indirectly_swappable<P>, std::indirectly_swappable<P1, P2>

- 保证 P 或 P1 和 P2 的值可以交换 (使用 std::ranges::iter_swap())。
- 要求:
  - 满足 std::indirect_readable<P1>(和 std::indirect_readable<P2>)
  - 任意两个类型为 P1 和 P2 的对象，支持 std::ranges::iter_swap()

std::indirectly_comparable<P1, P2, Comp>, std::indirectly_comparable<P1, P2, Comp, Proj1>, std::indirectly_comparable<P1, P2, Comp, Proj1, Proj2>

- 保证可以将元素 (可选地用 Proj1 和 Proj2 转换) 与 P1 和 P2 所引用的位置进行比较。
- 要求:
  - Comp 满足 std::indirect_binary_predicate
  - std::projected<P1, Proj1> 和 std::projected<P2, Proj2> 满足 std::identity 为默认投影

5.3.3 迭代器的概念

本节列出了需要不同类型迭代器的概念。其处理的是自 C++20 起的新迭代器类别，与范围的概念相对应。

这些概念在头文件 <iterator> 中提供。

std::input_output_iterator<Pos>

- 保证 Pos 支持所有迭代器的基本接口: 自加和解引用操作符，其中解引用操作符必须引用一个值。
  这个概念并不要求迭代器可复制 (所以，这低于算法使用的迭代器的基本要求)。
- 要求:
  - 满足 std::weakly_incrementable<pos>
  - 解引用操作符生成了一个引用

std::output_iterator<Pos, T>

- 保证 Pos 是一个输出迭代器 (可以为元素赋值的迭代器)，可以为其赋类型为 typeT 的值。
- Pos 类型的迭代器可用于赋值 T 类型的值 val:

```
*i++ = val;
```

- 这些迭代器只适用于单向迭代。
- 要求:
  - 满足 std::input_or_output_iterator<Pos>
  - 满足 std::indirectly_writable<Pos, I>

std::input_iterator<Pos>

- 保证 Pos 是一个输入迭代器 (可以从元素中读取值的迭代器)。
- 若 std::forward_iterator 也不满足，这些迭代器只适用于单向迭代。
- 要求:
  - 满足 std::input_or_output_iterator<Pos>
  - 满足 std::indirectly_readable<Pos>
  - Pos 有一个派生自 std::input_iterator_tag 的迭代器类别

std::forward_iterator<Pos>

- 确保 Pos 是一个前向迭代器 (一个读取迭代器，可以对元素进行多次前向迭代)。
- 注意，迭代器的 iterator_category 成员可能不匹配。对于产生左值的迭代器，是 std::input_iterator_tag(若可用)。
- 要求:
  - 满足 std::input_iterator<Pos>
  - 满足 std::incrementable<Pos>
  - Pos 有一个派生自 std::forward_iterator_tag 的迭代器类别

std::bidirectional_iterator<Pos>

- 确保 Pos 是一个双向迭代器 (一个读取迭代器，可以使用它在元素上向前或向后迭代多次)。
- 注意，迭代器的 iterator_category 成员可能不匹配。对于产生左值的迭代器，是 std::input_iterator_tag(若可用)。
- 要求:
  - 满足 std::forward_iterator<Pos>
  - 支持使用操作符向后迭代--
  - Pos 有一个派生自 std::bidirectional_iterator_tag 的迭代器类别

std::random_access_iterator<Pos>

- 确保 Pos 是一个随机访问迭代器 (可以在元素上来回跳转的读取迭代器)。
- 注意，迭代器的 iterator_category 成员可能不匹配。对于产生左值的迭代器，是 std::input_iterator_tag(若可用)。
- 要求:
  - 满足 std::bidirectional_iterator<Pos>
  - 满足 std::totally_ordered<Pos>
  - 满足 std::sized_sentinel_for<Pos, Pos>
  - 支持 +, +=, -, -=, [] 操作符
  - Pos 有一个派生自 std::random_access_iterator_tag 的迭代器类别

std::contiguous_iterator<Pos>

- 保证 Pos 是迭代器，迭代连续内存中的元素。
- 注意，迭代器的 iterator_category 成员不匹配。若定义了类别成员，则只有 std::random_access_iterator_tag，若值是右值，甚至只有 std::input_iterator_tag。
- 要求:
  - 满足 std::random_access_iterator<Pos>
  - Pos 有一个派生自 std::consecuous_iterator_tag 的迭代器类别
  - 元素的 to_address() 可指向该元素的裸指针

std::sentinel_for<S, Pos>

- 保证 S 可以用作 Pos 的哨兵 (可能是不同类型的结束迭代器)。
- 要求:
  - 满足 std::semiregular<S>
  - 满足 std::input_or_output_iterator<Pos>
  - 可以使用 == 和!= 操作符比较 Pos 和 S

std::sized_sentinel_for<S, Pos>

- 保证 S 可以用作 Pos 的哨兵 (可能是不同类型的结束迭代器)，并且可以在常数时间内计算它们之间的距离。
- 要表示不能在常数时间内计算距离 (尽管可以计算距离)，可以定义 std::disable_size_sentinel_for<Rg> 的结果为 true。
- 要求:
  - 满足 std::sentinel_for<S, Pos>
  - 对 Pos 和 S 调用操作符减号，将产生迭代器不同类型的值
  - std::disable_size_sentinel_for<S, Pos> 未定义为返回 true

5.3.4 算法的迭代器概念

std::permutable<Pos>

- 保证您可以使用向前自加操作符迭代，并通过移动和交换元素来重新排序
- 要求:
  - 满足 std::forward_iterator<Pos>
  - 满足 std::indirectly_movable_storable<Pos>
  - 满足 std::indirectly_swappable<Pos>

std::mergeable<Pos1, Pos2, ToPos>,
std::mergeable<Pos1, Pos2, ToPos, Comp>,
std::mergeable<Pos1, Pos2, ToPos, Comp, Proj1>,
std::mergeable<Pos1, Pos2, ToPos, Comp, Proj1, Proj2>,

- 保证可以通过将两个排序序列的元素复制到 ToPos 所引用的序列中，从而将其合并到 Pos1 和 Pos2 所引用的位置。顺序由运算符 < 或 Comp 定义 (应用于可选地用投影 Proj1 和 Proj2 转换的值)。
- 要求:
  - Pos1 和 Pos2 满足 std::input_iterator
  - 满足 std::weakly_incrementable<ToPos>
  - Pos1 和 Pos2 满足 std::indirect_copyable<PosN, ToPos>
  - Comp 满足 std::indirect_strict_weak_order，以及满足 std::projected<Pos1, Proj1> 和 std::projected<Pos2, Proj2>(以 < 作为默认比较，std::identity 作为默认投影):
    * Pos1 和 Pos2 满足 std::indirectly_readable
    * 满足 std::copy_constructible<Comp>
    * Comp& 和 (投影的) 值/引用类型满足 std::strict_weak_order<Comp>

std::sortable<Pos> std::sortable<Pos, Comp> std::sortable<Pos, Comp, Proj>

- 保证可以使用操作符 < 或 Comp 对迭代器 Pos 引用的元素进行排序 (在可选地对值应用投影 Proj 之后)。
- 要求:
  - 满足 std::permutable<Pos>
  - std::indirect_strict_weak_order 的 Comp 和 (投影) 值满足 (以 < 作为默认比较，std::identity 作为默认投影):
    - 满足 std::indirectly_readable<Pos>
    - 满足 std::copy_constructible<Comp>
    - Comp& 和 (投影的) 值/引用类型满足 std::strict_weak_order<Comp>

5.4. 可调用对象的概念

本节列出了可调用对象的所有概念

- 函数
- 函数对象
- Lambda 表达式
- 指向成员的指针 (带对象类型作为附加参数)

5.4.1 可调用对象的基本概念

std::invocable<Op, ArgTypes...>

- 保证可以为 ArgTypes.... 类型的参数调用 Op
- Op 可以是函数、函数对象、Lambda 或指向成员的指针。
- 要证明操作和传递的参数都没有修改，可以使用 std::regular_invocable 代替，这两个概念之间只有语义上的差异，不能在编译时进行检查，所以概念上的差异只说明了意图。
- 例如:

```
struct S {
  int member;
  int mfunc(int);
  void operator()(int i) const;
};

void testCallable()
{
  std::invocable<decltype(testCallable)> // satisfied
  std::invocable<decltype([](int){}), char> // satisfied (char converts to int)
```

```
  std::invocable<decltype(&S::mfunc), S, int> // satisfied (member-pointer, object, args)
  std::invocable<decltype(&S::member), S>; // satisfied (member-pointer and object)
  std::invocable<S, int>; // satisfied due to operator()
}
```

作为类型约束，只需要指定参数类型:

```
void callWithIntAndString(std::invocable<int, std::string> auto op);
```

完整示例，请参见使用 Lambda 作为非类型模板参数。

- 要求:
  - std::invoke(op, args) 对 Op 类型和 ArgTypes... 类型的 args 有效

std::regular_invocable<Op, ArgTypes...>

- 保证可以为 ArgTypes…类型的参数调用 Op。并且调用既不改变所传递操作的状态，也不改变所传递参数的状态。
- Op 可以是函数、函数对象、Lambda 或指向成员的指针。
- 注意，std::invocable 与 std::invocable 概念的区别纯粹是语义上的，不能在编译时检查，所以概念差异只是证明了意图。
- 要求:
  - std::invoke(op, args) 对 Op 类型和 ArgTypes... 类型的 args 有效

std::predicate<Op, ArgTypes...>

- 保证可调用的 (函数，函数对象，Lambda)Op 是一个谓词，可以为 ArgTypes... 类型的参数所调用。
- 调用既不改变所传递操作的状态，也不改变所传递参数的状态。
- 要求:
  - 满足 std::regular_invocable<Op>
  - 所有使用 ArgTypes... 的 Op 调用，都会生成一个可当作布尔值的值

std::relation<Pred, T1, T2>

- 保证任意两个类型 T1 和 T2 的对象具有二元关系，可以作为参数传递给二元谓词 Pred。
- 调用既不改变所传递操作的状态，也不改变所传递参数的状态。
- 注意，std::equivalence_relation 和 std::strict_weak_order 的区别纯粹是语义上的，不能在编译时检查，所以概念差异只能说明意图。
- 要求:

  - 对于 pred 以及两个对象类型 T1 和 T2 的任意组合，都满足 std::predicate

std::equivalence_relation<Pred, T1, T2

- 保证任意两个类型 T1 和 T2 的对象与 Pred 相比具有等价关系。
  可以作为参数传递给二进制谓词 Pred，并且这种关系是自反的、对称的和可传递的。
- 调用既不改变所传递操作的状态，也不改变所传递参数的状态。
- 注意，std::relation 和 std::strict_weak_order 的区别纯粹是语义上的，不能在编译时检查，所以概念差异只能说明意图。
- 要求:
  - 对于 pred 以及两个对象类型 T1 和 T2 的任意组合，都满足 std::predicate

std::strict_weak_order<Pred, T1, T2>

- 与 Pred 相比，任意两个类型 T1 和 T2 的对象具有严格的弱序关系。
  可以作为参数传递给二元谓词 Pred，并且这种关系是非自反的和传递的。
- 调用既不改变所传递操作的状态，也不改变所传递参数的状态。
- 注意，std::relation 和 std::equivalence_relation 的区别纯粹是语义上的，不能在编译时检查，所以概念差异只能说明意图。
- 要求:
  - 对于 pred 以及两个对象类型 T1 和 T2 的任意组合，都满足 std::predicate

std::uniform_random_bit_generator<Op>

- 保证 Op 可以用作返回 (理想情况下) 相等概率的无符号整数值的随机数生成器。
- 这个概念在头文件 <random> 中定义。
- 要求:
  - 满足 std::invocable<Op&>
  - 结果满足 std::unsigned_integral
  - 支持表达式 Op::min() 和 Op::max()，并产生与生成器调用相同的类型
  - Op::min() 小于 Op::max()

5.4.2 迭代器使用可调用对象的概念

std::indirectly_unary_invocable<Op, Pos>

- 保证可以用 Pos 所引用的值调用 Op。

- 请注意，与 indirectly_regular_unary_invocable 概念的区别纯粹是语义上的，不能在编译时检查，所以概念差异只能说明意图。
- 要求:
  - 满足 std::indirectly_readable<Pos>
  - 满足 std::copy_constructible<Op>
  - 满足 Op& 和 Pos 的值和 (常用) 引用类型的 std::invocable
  - 同时使用值和引用调用 Op 的结果，具有共同的引用类型

std::indirectly_regular_unary_invocable<Op, Pos>

- 保证可以使用 Pos 所引用的值调用 Op，并且调用不会改变 Op 的状态。
- 请注意，与 std::indirect_unary_invocable 概念的区别是纯语义的，不能在编译时检查，所以概念差异只能说明意图。
- 要求:
  - 满足 std::indirectly_readable<Pos>
  - 满足 std::copy_constructible<Op>
  - Op& 和 Pos 的值，以及 (常见) 引用类型都满足 std::regular $_{i}nvocation$
- 同时使用值和引用调用 Op 的结果是，具有共同的引用类型

std::indirect_unary_predicate<Pred, Pos>

- 保证可以用 Pos 引用的值调用一元谓词 Pred。
- 要求:
  - 满足 std::indirectly_readable<Pos>
  - 满足 std::copy_constructible<Pred>
  - Pred& 和 Pos 值，以及 (常见) 引用类型都满足 std::predicate
  - 所有这些对 Pred 的调用，都会产生一个可以当作布尔值的值

std::indirect_binary_predicate<Pred, Pos1, Pos2>

- 确保可以用 Pos1 和 Pos2 引用的值调用二元谓词 Pred。
- 要求:
  - Pos1 和 Pos2 满足了 std::indirect_readable
  - 满足 std::copy_constructible<Pred>
  - 若 Pred& 是 Pos1 的值或 (通用) 引用类型，以及 Pos2 的值或 (通用) 引用类型，则满足 std::predicate
  - 所有这些对 Pred 的调用都会产生一个可作为布尔值的值

std::indirect_equivalence_relation<Pred, Pos1>,
std::indirect_equivalence_relation<Pred, Pos1, Pos2>

- 确保可以调用二元谓词 Pred 来检查 Pos1 和 Pos1/Pos2 的两个值是否相等。
- 注意，与 std::indirectly_strict_weak_order 概念的区别纯粹是语义上的，不能在编译时检查，所以概念差异只能说明意图。
- 要求:
  - Pos1 和 Pos2 满足了 std::indirect_readable
  - 满足 std::copy_constructible<Pred>
  - 若 Pred& 是 Pos1 的值或 (通用) 引用类型，以及 Pos2 的值或 (通用) 引用类型，则满足 std::predicate
  - 所有这些对 Pred 的调用都会产生一个可以作为布尔值的值

std::indirect_strict_weak_order<Pred, Pos1>, std::indirect_strict_weak_order<Pred, Pos1, Pos2>

- 保证可以调用二元谓词 Pred 来检查 Pos1 和 Pos1/Pos2 的两个值是否具有严格的弱序。
- 注意，与 std::indirect_equivalence_relation 概念的区别纯粹是语义上的，不能在编译时检查，所以概念差异只能说明意图。
- 要求:
  - Pos1 和 Pos2 满足了 std::indirect_readable
  - 满足 std::copy_constructible<Pred>
  - 若 Pred& 是 Pos1 的值或 (通用) 引用类型，以及 Pos2 的值或 (通用) 引用类型，则满足 std::predicate
  - 所有这些对 Pred 的调用都会产生一个可以作为布尔值的值

5.5. 辅助概念

介绍一些标准化的概念，这些概念主要用于实现其他概念。通常，不应该在应用程序代码中使用。

5.5.1 特定类型属性的概念

std::default_initializable<T>

- 保证类型 T 支持默认构造 (没有初始值的声明/构造)。
- 要求:
  - 满足 std::constructible_from<T>
  - 满足 std::destructible<T>
  - T{}; 是可用的

  - T x; 是可用的

std::move_constructible<T>

- 保证类型 T 的对象可以用其类型的右值初始化。
- 以下操作是有效的 (尽管它可以复制而不是移动):

```
T t2{std::move(t1)} // for any t1 of type T
```

  之后，t2 的值应该是 t1，这是一个在编译时无法检查的语义约束。
- 要求:
  - 满足 std::constructible_from<T, T>
  - 满足 std::convertible<T, T>
  - 满足 std::destructible<T>

std::copy_constructible<T>

- 保证类型 T 的对象可以用其类型的左值初始化。
- 以下操作是有效的:

```
T t2{t1} // for any t1 of type T
```

  之后，t2 的值应该是 t1，这是一个在编译时无法检查的语义约束。
- 要求:
  - 满足 std::move_constructible<T>
  - std::constructible_from 和 std::convertible_to 适用于任何 T、T&、const T 和 const T& 转换成的 T
  - 满足 std::destructible<T>

std::destructible<T>

- 保证类型 T 的对象可析构而不抛出异常。
  注意，即使是实现的析构函数也会自动保证不抛出，除非显式地用 noexcept(false) 标记。
- 要求:
  - 类型特性 std::is_nothrow_destructible_v<T> 为 true

std::swappable<T>

- 保证可以交换两个 T 类型对象的值。
- 要求:
  - std::ranges::swap() 可以为两个类型为 T 的对象调用

5.5.2 可增量类型的概念

std::weakly_incrementable<T>

- 保证类型 T 支持自增操作符。
- 注意，这个概念并不要求相同值的两次增量产生相同的结果，所以这只是单向迭代的要求。
- 与 std::incrementable 相比，若满足以下条件，这个概念也能得到满足:
  - 该类型不是默认可构造的、不可复制的或不可相等比较的
  - 后置自增运算符返回 void(或任何其他类型)。
  - 相同值的两次增量会产生不同的结果
- 注意，与概念 std::incrementable 的区别纯粹是语义上的区别，所以对于那些增量产生不同结果的类型，可能在技术上仍然满足增量概念。要针对这种语义差异实现不同的行为，应该使用迭代器概念。
- 要求:
  - 满足 std::default_initializable<T>
  - 满足 std::movable<T>
  - std::iter_difference_t<T> 是一个有效的有符号整型

std::incrementable<T>

- 保证类型 T 是一个可递增的类型，这样您就可以对相同的值序列进行多次迭代。
- 与 std::weak_incrementable 不同，这个概念要求:
  - 相同值的两次增量产生相同的结果 (就像前向迭代器的情况一样)
  - 类型 T 是默认可构造的、可复制的和可比较的
  - 后置自增操作符返回迭代器的副本 (返回类型为 T)。
- 注意，与概念 std::weak _incrementable 的区别纯粹是语义上的区别，所以于增量产生不同结果的类型，在技术上可能仍然满足概念 incrementable。要针对这种语义差异实现不同的行为，应该使用迭代器概念。
- 要求:
  - 满足 std::weakly_incrementable<T>
  - 满足 std::regular<T>，使得该类型是默认可构造的、可复制的和可比较的

第 6 章 范围和视图

自第一个 C++ 标准以来，处理容器和其他序列元素的方法一直是使用迭代器，来确定第一个元素的位置 (开始) 和最后一个元素后面的位置 (结束)。对范围进行操作的算法通常使用两个参数来处理容器的所有元素，容器提供了 begin() 和 end() 等函数来提供这些参数。

C++20 提供了一种处理范围的新方法，支持将范围和子范围定义和使用为单个对象，例如：将其作为一个整体作为单个参数传递，而不是处理两个迭代器。

这种改变听起来很简单，会产生很多后果。对于调用者和实现者来说，处理算法的方式都发生了巨大的变化，所以 C ++20 提供了一些处理范围的新特性和工具:

- 将范围作为单个参数的新重载或标准算法的变体
- 处理范围对象的几个工具:
  - 用于创建范围对象的辅助函数
  - 处理范围对象的辅助函数
  - 用于处理范围对象的辅助类型
  - 范围的概念
- 轻量级范围，称为视图，用于引用具有可选值转换的范围 (子集)
- 管道是一种灵活的组合范围和视图处理的方式

本章介绍了范围和视图的基本方面和特点。

6.1. 使用范围和视图

让我们通过几个使用范围和视图的例子来了解其作用，并在不深入细节的情况下进行讨论。

6.1.1 将容器作为范围传递给算法

自 C++98 年发布第一个 C++ 标准以来，在处理元素集合时，可以迭代半开放范围。通过传递范围的 begin() 和 end()(通常来自容器的 begin() 和 end() 成员函数)，可以指定必须处理哪些元素:

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

std::vector<int> coll{25, 42, 2, 0, 122, 5, 7};

std::sort(coll.begin(), coll.end()); // sort all elements of the collection
```

C++20 引入了范围的概念，表示一系列值的单个对象。任何容器都可以这样使用范围。

因此，可以将容器作为一个整体传递给算法:

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

std::vector<int> coll{25, 42, 2, 0, 122, 5, 7};
```

```
5
6  std::ranges::sort(coll); // sort all elements of the collection
```

这里，将 vector coll 传递给范围算法 sort()，以便对 vector 中的所有元素进行排序。

自 C++20 以来，大多数标准算法都支持将范围作为一个参数传递，但目前还不支持并行算法和数值算法。除了在整个范围内只接受一个参数外，新算法可能有一些细微的区别:

- 使用迭代器和范围的概念来确保传递有效的迭代器和范围。
- 可能有不同的返回类型。
- 可能返回租借迭代器，因为传递了一个临时范围 (右值)，从而没有有效的迭代器。

详细信息请参见 C++20 中的算法概述。

范围的命名空间

range 库的新算法，如 sort() 在命名空间 std::ranges 中提供。一般来说，C++ 标准库提供了处理特殊命名空间中的范围的所有功能:

- 大多数在命名空间 std::ranges 中提供。
- 其中一些在 std::views 中提供，这是 std::ranges::views 的别名。

新的命名空间是必要的，因为 ranges 库引入了几个使用相同符号名称的新 API，但 C++20 不应该破坏现有代码，所以命名空间可以避免歧义和其他与现有 API 的冲突。

看到哪些属于 ranges 库及其命名空间 std::ranges，哪些属于 std，有时会感到惊讶。粗略地说，std::ranges 用于作为一个整体处理范围的工具。

例如:

- 有些概念属于 std，有些属于 std::range。
  例如，对于迭代器，我们有 std::forward_iterator 概念，相应的范围概念是 std::ranges::forward_range。
- 有些类型函数属于 std，有些属于 std::range。
  例如，有类型特性 std::iter_value_t，对应的范围类型特性是 std::ranges::range_value_t。
- 有些符号甚至在 std 和 std::ranges 中都提供了。
  例如，独立函数 std::begin() 和 std::ranges::begin()。

若两者都可以使用，最好使用 std::ranges 命名空间中的符号和工具，原因是 ranges 库的新工具可以修复旧工具所具有的缺陷，例如：最好在命名空间 std::ranges 中使用 begin() 或 cbegin()。

为命名空间 std::ranges 引入快捷方式很常见，比如 rg 或 rng。所以，上面的代码也可以如下所示:

```
1  #include <vector>
2  #include <algorithm>
3  namespace rg = std::ranges; // define shortcut for std::ranges
4
5  std::vector<int> coll{25, 42, 2, 0, 122, 5, 7};
```

```
6
7  rg::sort(coll); // sort all elements of the collection
```

不要使用 using namespace 指令来避免限定范围符号。否则，很容易得到具有编译时冲突或使用错误查找的代码，可能会发生严重的错误。

范围库的头文件

范围库的许多新特性都在新的头文件 <ranges> 中提供，但其中一些是在现有的头文件中提供的。范围算法就是一个例子，在 <algorithm> 中声明。

因此，要使用为范围提供的算法作为单个参数，只需要包含 <algorithm>。

但对于范围库提供的一些附加特性，需要头文件 <ranges>。所以，在使用 std::ranges 或 std::views 命名空间中的内容时，应该始终包含 <ranges>:

```
1  #include <vector>
2  #include <algorithm>
3  #include <ranges> // for ranges utilities (so far not necessary yet)
4
5  std::vector<int> coll{25, 42, 2, 0, 122, 5, 7};
6
7  std::ranges::sort(coll); // sort all elements of the collection
```

6.1.2 范围的约束和工具

范围的新标准算法将范围参数声明为模板参数 (没有可用于它们的通用类型)，为了在处理这些范围参数时指定和验证必要的要求，C++20 引入了几个范围概念。此外，工具可用于使用这些概念。

例如，考虑用于范围的 sort() 算法。原则上，其定义如下 (缺少一些细节，将在后面补充):

```
1  template<std::ranges::random_access_range R,
2      typename Comp = std::ranges::less>
3  requires std::sortable<std::ranges::iterator_t<R>, Comp>
4  ... sort(R&& r, Comp comp = {});
```

这个声明已经有了范围的多个新特性:

- 两个标准概念规定了通过范围 R 的要求:
  - 概念 std::ranges::random_access_range 要求 R 是提供随机访问迭代器的范围 (可以使用迭代器在元素之间来回跳转，并计算其距离)。这个概念包括 (包含) 范围:std::range 的基本概念，要求对于传递的参数，可以从 begin() 到 end() 遍历元素 (至少使用 std::ranges::begin() 和 std::ranges::end()，所以数组也满足这个概念)。
  - sortable 概念要求范围 R 中的元素可以用排序条件 Comp 进行排序。
- 新的类型工具 std::ranges::iterator_t 用于将迭代器类型传递给 std::sortable。

- 比较谓词 std::ranges::less 用作默认的比较条件，其定义了排序算法使用 < 操作符对元素进行排序。std::ranges::less 是一种概念约束的 std::less，确保支持所有比较操作符 (==、!=、<、<=、> 和 >=)，并确保值具有总排序。

所以，可以通过随机访问迭代器和可排序元素传递任何范围。例如:

ranges/rangessort.cpp

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

void print(const auto& coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector<std::string> coll{"Rio", "Tokyo", "New York", "Berlin"};

  std::ranges::sort(coll); // sort elements
  std::ranges::sort(coll[0]); // sort character in first element
  print(coll);

  int arr[] = {42, 0, 8, 15, 7};
  std::ranges::sort(arr); // sort values in array
  print(arr);
}
```

该程序有以下输出:

```
Beilnr New York Rio Tokyo
0 7 8 15 42
```

若传递没有随机访问迭代器的容器/范围，会得到一个错误消息：不满足概念 std::ranges::random_access_range:

```
std::list<std::string> coll2{"New York", "Rio", "Tokyo"};
std::ranges::sort(coll2); // ERROR: concept random_access_range not satisfied
```

若传递的容器/范围不能与小于操作符进行比较，则不满足 std::sortable:

```
std::vector<std::complex<double>> coll3;
std::ranges::sort(coll3); // ERROR: concept sortable not satisfied
```

范围的概念

“基本范围概念”表列出了定义范围需求的基本概念。

概念 (std::ranges 中的)	要求
range	可以从头到尾迭代吗
output_range	写入值的元素的范围
input_range	读取元素值的范围
forward_range	范围可多次迭代这些元素
bidirectional_range	可以向前和向后遍历元素
random_access_range	随机访问范围 (在元素之间来回跳转)
contiguous_range	连续内存中包含所有元素的范围
sized_range	具有恒定时间 size() 的范围

表 6.1 基本范围概念

注意事项如下:

- std::ranges::range 是所有其他范围概念的基本概念 (所有其他概念都包含这个概念)。
- output_range，input_range, forward_range, bidirectional_range, random_access_range 和 contiguous_range 的层次结构映射到相应的迭代器类别，并构建相应的包容层次结构。
- std::ranges::consecuous_range 是一个新的范围/迭代器类别，其保证元素存储在连续内存中，通过使用指针迭代元素。
- std::ranges::sized_range 独立于其他约束，只是它是一个范围。

注意，迭代器和相应的范围类别在 C++20 中略有变化，C++ 标准现在支持两个版本的迭代器类别:C++20 之前的版本和 C++20 之后的版本，它们可能不一样。

“其他范围概念”表列出了稍后将介绍的用于特殊情况的其他一些范围概念。

概念 (std::range 中的)	要求
view	复制或移动和分配成本较低的范围
viewable_range	可以转换为视图的范围 (使用 std::ranges::all())
borrowed_range	使用与范围的生命周期无关的迭代器进行范围操作
common_range	开始和结束 (sentine) 的范围具有相同的类型

表 6.2 其他范围概念

有关详细信息，请参见范围概念的讨论。

6.1.3 视图

为了处理范围，C++20 还引入了视图。视图是轻量级的范围，创建和复制/移动成本都很低。

- 参考范围和子范围
- 自有临时范围
- 过滤元素
- 生成转换后的元素值
- 生成一系列的值

视图通常用于在特定的基础上，处理基础范围的元素子集和/或经过一些可选转换后的值。例如，可以使用一个视图来迭代一个范围的前五个元素，如下所示:

```
for (const auto& elem : std::views::take(coll, 5)) {
  ...
}
```

std::views::take() 是一个范围适配器，用于创建对传递的范围 coll 进行操作的视图，take() 创建了传递范围 (若有) 的前 n 个元素的视图，则可将一个视图传递给 coll，该视图以其第六个元素结束 (若元素较少，则以最后一个元素结束)。

```
std::views::take(coll, 5)
```

C++ 标准库提供了几个范围适配器和工厂，用于在命名空间 std::views 中创建视图。创建的视图提供了通常的范围 API，可以使用 begin()、end() 和自加操作符来迭代元素，并且可以使用解引用操作符来处理值。

范围和视图的管道

调用作为单个参数传递的范围适配器有另一种语法:

```
for (const auto& elem : coll | std::views::take(5)) {
  ...
}
```

这两种形式等价，但管道语法使得在范围上创建视图序列更加方便，稍后将详细讨论这一点。这样，简单的视图就可以用作更复杂的元素集合处理的构建块。

假设想使用以下三个视图:

```
// view with elements of coll that are multiples of 3:
std::views::filter(coll, [] (auto elem) {
  return elem % 3 == 0;
})

// view with squared elements of coll:
std::views::transform(coll, [] (auto elem) {
  return elem * elem;
})

```

```
// view with first three elements of coll:
std::views::take(coll, 3)
```

因为视图是一个范围，可以使用视图作为另一个视图的参数:

```
// view with first three squared values of the elements in coll that are multiples of
↪  3:
auto v = std::views::take(
  std::views::transform(
    std::views::filter(coll,
      [] (auto elem) { return elem % 3 == 0; }),
    [] (auto elem) { return elem * elem; }),
  3);
```

这种嵌套很难阅读和维护，可以从另一种管道语法中让视图对范围进行操作。通过使用操作符|，可以创建视图的管道:

```
// view with first three squared values of the elements in coll that are multiples of
↪  3:
auto v = coll
  | std::views::filter([] (auto elem) { return elem % 3 == 0; })
  | std::views::transform([] (auto elem) { return elem * elem; })
  | std::views::take(3);
```

范围和视图的管道易于定义和理解。

对于集合，例如

```
std::vector coll{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
```

输出将是:

```
9 36 81
```

下面是另一个完整的程序，演示了视图与管道语法的组合:

ranges/viewspipe.cpp

```
#include <iostream>
#include <vector>
#include <map>
#include <ranges>
int main()
{
  namespace vws = std::views;
  // map of composers (mapping their name to their year of birth):
  std::map<std::string, int> composers{
    {"Bach", 1685},
    {"Mozart", 1756},
```

```
    {"Beethoven", 1770},
    {"Tchaikovsky", 1840},
    {"Chopin", 1810},
    {"Vivaldi ", 1678},
  };
  // iterate over the names of the first three composers born since 1700:
  for (const auto& elem : composers
    | vws::filter([](const auto& y) { // since 1700
      return y.second >= 1700;
    })
    | vws::take(3) // first three
    | vws::keys // keys/names only
  ) {
    std::cout << "- " << elem << '\n';
  }
}
```

例子中，对作 composers 的映射应用了两个视图，其中的元素有他们的名字和出生年份 (注意，之类引入了 vws 作为 std::views 的快捷方式):

```
std::map<std::string, int> composers{ ... };
...
composers
  | vws::filter( ... )
  | vws::take(3)
  | vws::keys
```

组合管道直接传递给基于范围的 for 循环，并产生以下输出 (map 中的元素是根据键/名称排序的):

```
- Beethoven
- Chopin
- Mozart
```

生成视图

视图本身也可以生成一系列值。例如，通过使用 iota 视图，可以遍历从 1 到 10 的所有值:

```
for (int val : std::views::iota(1, 11)) { // iterate from 1 to 10
  ...
}
```

视图类型和生命周期

可以创建视图，并在使用之前给定一个名字:

```cpp
auto v1 = std::views::take(coll, 5);
auto v2 = coll | std::views::take(5);
...
for (int val : v1) {
  ...
}
std::ranges::sort(v2);
```

大多数关于左值 (有名称的范围) 的视图都有引用语义，所以必须确保视图的引用范围和迭代器存在并且有效。

指定确切的类型可能很棘手，可以使用 auto 来声明视图，例如：对于具有 std::vector<int> 类型的 coll, v1 和 v2 都是以下类型:

```cpp
std::ranges::take_view<std::ranges::ref_view<std::vector<int>>>
```

在内部，适配器和构造函数可以创建嵌套视图类型，例如 std::ranges::take_view 或 std::ranges::iota_view，引用 std::ranges::ref_view，后者用于引用传入的外部容器的元素。

也可以直接声明和初始化真实的视图，但通常应该使用提供的适配器和工厂来创建和初始化视图。使用适配器和工厂通常更好，因为更容易使用、更智能，并且可以提供优化。例如，若传递的范围已经是 std::string_view，take() 可能只产生一个 std::string_view。

现在，可能想知道所有这些视图类型是否会在代码大小和运行时间上产生巨大的开销。注意，视图类型只使用小的或不重要的内联函数，所以至少在通常情况下，优化编译器可以避免显著的开销。

写入视图

左值视图通常具有引用语义，所以视图既可以用于读取，也可以用于写入。

例如，只能对 coll 的前五个元素进行排序:

```cpp
std::ranges::sort(std::views::take(coll, 5)); // sort the first five elements of coll
```

或:

```cpp
std::ranges::sort(coll | std::views::take(5)); // sort the first five elements of
↪  coll
```

通常:

- 若引用范围的元素修改，则视图的元素也会修改。
- 若视图的元素修改，则引用范围的元素也会修改。

延迟计算

除了具有引用语义之外，视图还使用延迟求值。所以当视图的迭代器调用 begin()、自加操作符或请求元素的值时，视图会对下一个元素进行处理:

```
auto v = coll | std::views::take(5); // neither goes to the first element nor to its value
...
auto pos = v.begin(); // goes to the first element
...
std::cout << *pos; // goes to its value
...
++pos; // goes to the next element
...
std::cout << *pos; // goes to its value
```

缓存

此外，一些视图使用缓存。若使用 begin() 访问视图的第一个元素需要一些计算 (因为跳过了前导元素)，视图可能会缓存 begin() 的返回值，以便下次调用 begin() 时，不必再次计算第一个元素的位置。

然而，这种优化有一些后果:

- 当视图为 const 时，可能无法遍历视图。
- 即使没有修改任何东西，并发迭代也会导致未定义行为。
- 早期的读访问可能会使视图元素的后期迭代失效或更改。

我们将在后面详细讨论所有这些。当修改发生时，标准视图是有害的:

- 在范围中插入或删除元素，可能会对视图的功能产生重大影响。经过这样的修改后，视图的行为可能会有所不同，甚至不再有效。

因此，强烈建议在需要视图之前使用它们——创建视图并使用。若在初始化视图和使用视图之间发生修改，则必须谨慎。

6.1.4 哨兵

为了处理范围，必须引入一个新术语——“哨兵”，表示范围的结束。

在编程中，哨兵是一个特殊的值，标志着结束或终止。典型的例子有:

- 空结束符’\0’ 作为字符序列的结尾 (例如，用于字符串字面值)
- nullptr 标记链表的结束
- -1 表示非负整数列表的结束

在范围库中，哨兵定义范围的结束。在 STL 的传统方法中，哨兵将是 end 迭代器，通常与迭代集合的迭代器具有相同的类型。但在 C++20 范围中，不再需要这种类型。

end 迭代器必须与定义范围的开始迭代器和用于遍历元素的迭代器具有相同的类型，这一要求导致了一些缺陷。创建一个 end 迭代器可能很昂贵，甚至不可能:

- 若要使用 C 字符串或字符串字面值作为范围，首先必须通过遍历字符来计算 end 迭代器，直到找到'\0'，所以在使用字符串作为范围之前，必须已经完成了第一次迭代，处理所有的元素需要第二次迭代。
- 通常，若用某个值定义范围的结束，则适用此原则。若需要一个 end 迭代器来处理范围，首先必须遍历整个范围以找到其 end。
- 有时，不可能迭代两次 (一次查找末尾，一次处理范围的元素)。这适用于使用纯输入迭代器的范围，例如：使用从其读取的输入流作为范围。为了计算输入的末尾 (可能是 EOF)，必须读取输入。再次读取输入是不可能的，或者会产生不同的值。

end 迭代器作为哨兵的泛化解决了这个难题。C++20 范围支持不同类型的哨兵 (end 迭代器)，可以表示“直到'\0'”，“直到 EOF”，或直到任何值，甚至可以发出“没有终点”的信号来定义无限的范围，或说“嘿，迭代器，自己检查是否有终点。”

注意，C++20 前，也可以有这些类型的哨兵，但必须与迭代器具有相同的类型。输入流迭代器就是一个例子:std::istream_iterator<> 类型的默认构造迭代器可用来创建一个流结束迭代器，这样就可以用算法处理流中的输入，直到文件结束或出现错误:

```
// print all int values read from standard input:
std::for_each(std::istream_iterator<int>{std::cin}, // read ints from cin
    std::istream_iterator<int>{}, // end is an end-of-file iterator
    [] (int val) {
      std::cout << val << '\n';
    });
```

通过放宽哨兵 (end 迭代器) 现在必须与迭代迭代器具有相同类型的要求，有几个好处:

- 可以在开始处理之前跳过寻找结尾的需要，同时做这两件事: 处理值；并在迭代时找到结束值。
- 对于 end 迭代器，可以使用禁用导致未定义行为的操作的类型 (例如调用解引用操作符，因为末尾没有值)。当尝试解引用 end 迭代器时，可以使用该特性在编译时发出错误信号。
- 定义 end 迭代器变得更加容易。

来看一个简单的例子，使用不同类型的哨兵来迭代迭代器类型不同的“范围”:

ranges/sentinel1.cpp

```
#include <iostream>
#include <compare>
#include <algorithm> // for for_each()

struct NullTerm {
  bool operator== (auto pos) const {
    return *pos == '\0'; // end is where iterator points to ' \0'
  }
};

int main()
{
  const char* rawString = "hello world";

```

```
  // iterate over the range of the begin of rawString and its end:
  for (auto pos = rawString; pos != NullTerm{}; ++pos) {
    std::cout << ' ' << *pos;
  }
  std::cout << '\n';

  // call range algorithm with iterator and sentinel:
  std::ranges::for_each(rawString, // begin of range
        NullTerm{}, // end is null terminator
        [] (char c) {
          std::cout << ' ' << c;
        });
  std::cout << '\n';
}
```

该程序有以下输出:

```
h e l l o   w o r l d
h e l l o   w o r l d
```

首先定义一个 end 迭代器，将 end 定义为具有等于'\0' 的值:

```
struct NullTerm {
  bool operator== (auto pos) const {
    return *pos == '\0'; // end is where iterator points to ' \0'
  }
};
```

注意，在这里结合了 C++20 的其他一些新特性:

- 定义成员函数 == 操作符时，使用 auto 作为形参类型，就使成员函数具有泛型，这样 == 操作符就可以与任意类型的对象 pos 进行比较 (前提是 pos 引用的值与'\0' 的比较是有效的)。
- 只将 == 操作符定义为泛型成员函数，尽管算法通常使用!= 操作符来比较迭代器和哨兵，C++20 现在可以用任意顺序的操作数将!= 映射到 == 操作符。

直接使用哨兵

然后，使用一个基本循环遍历字符串 rawString 的“范围”字符:

```
for (auto pos = rawString; pos != NullTerm{}; ++pos) {
  std::cout << ' ' << *pos;
}
```

将 pos 初始化为一个迭代器，该迭代器遍历字符并使用 *pos 输出其值。当其与 NullTerm{} 的比较产生 pos 的值不等于'\0' 时，循环运行。因此，NullTerm{} 就起到了哨兵的作用。其类型与 pos 不同，但支持与 pos 进行比较，从而检查 pos 所引用的当前值。

这里可以看到哨兵是 end 迭代器的泛化。可能与遍历元素的迭代器类型不同，但支持与迭代器进行比较，以确定是否位于范围的末端。

将哨兵传递给算法

C++20 为不再要求开始迭代器和哨兵 (结束迭代器) 具有相同类型的算法提供了重载，但这些重载会在命名空间 std::ranges 中提供:

```
std::ranges::for_each(rawString, // begin of range
            NullTerm{}, // end is null terminator
            ... );
```

命名空间 std 中的算法仍然要求开始和结束迭代器具有相同的类型，不能这样使用:

```
std::for_each(rawString, NullTerm{}, // ERROR: begin and end have different types
      ... );
```

若有两个不同类型的迭代器，则类型 std::common_iterator 为传统算法提供了一种协调方法，因为数字算法、并行算法和容器仍然要求开始和结束迭代器具有相同的类型。

6.1.5 用哨兵计数和定义范围

范围可以不仅仅是容器或一对迭代器，也可以通过以下方式定义:

- 同一类型的开始和结束迭代器
- 开始迭代器和哨兵 (可能是不同类型的结束标记)
- 开始迭代器和 count
- 数组

范围库支持所有这些范围的定义和使用。

首先，算法以这样一种方式实现: 范围可以是数组。例如:

```
int rawArray[] = {8, 6, 42, 1, 77};
...
std::ranges::sort(rawArray); // sort elements in the raw array
```

此外，还有几个工具用于定义由迭代器和哨兵或计数定义的范围，这些将在以下小节中介绍。

子范围

为了定义迭代器和哨兵的范围，范围库提供了 std::ranges::subrange<> 类型。

来看一个使用子范围的简单例子:

ranges/sentinel2.cpp

```cpp
#include <iostream>
#include <compare>
#include <algorithm> // for for_each()

struct NullTerm {
  bool operator== (auto pos) const {
    return *pos == '\0'; // end is where iterator points to ' \0'
  }
};

int main()
{
  const char* rawString = "hello world";

  // define a range of a raw string and a null terminator:
  std::ranges::subrange rawStringRange{rawString, NullTerm{}};

  // use the range in an algorithm:
  std::ranges::for_each(rawStringRange,
  [] (char c) {
    std::cout << ' ' << c;
  });
  std::cout << '\n';

  // range-based for loop also supports iterator/sentinel:
  for (char c : rawStringRange) {
    std::cout << ' ' << c;
  }

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

作为一个哨兵的例子，定义了类型 NullTerm 作为哨兵的类型，用于检查字符串的空结束符是否作为范围的结束。

通过使用 std::ranges::subrange，代码定义了一个范围对象，表示字符串的开始，哨兵作为字符串的结束:

```cpp
std::ranges::subrange rawStringRange{rawString, NullTerm{}};
```

子范围是泛型类型，可将迭代器和哨兵定义的范围转换为表示该范围的单个对象。实际上，范围甚至是一个视图，在内部只存储迭代器和哨兵。所以子范围具有引用语义，并且复制成本很低。

作为子范围，可以将范围传递给将范围作为单个参数的新算法:

```cpp
std::ranges::for_each(rawStringRange, ... ); // OK
```

子范围并不总是通用范围，对它们调用 begin() 和 end() 可能会产生不同的类型，子范围只产生传递定义范围的内容。

即使子范围不是公共范围，也可以将其传递给基于范围的 for 循环。基于范围的 for 循环接受开始迭代器和哨兵 (结束迭代器) 类型不同的范围 (该特性已经在 C++17 中引入，但是对于范围和视图):

```
for (char c : std::ranges::subrange{rawString, NullTerm{}}) {
  std::cout << ' ' << c;
}
```

通过定义一个类模板，可以在其中指定范围结束的值，从而使这种方法更加通用:

ranges/sentinel3.cpp

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

template<auto End>
struct EndValue {
  bool operator== (auto pos) const {
    return *pos == End; // end is where iterator points to End
  }
};

int main()
{
  std::vector coll = {42, 8, 0, 15, 7, -1};

  // define a range referring to coll with the value 7 as end:
  std::ranges::subrange range{coll.begin(), EndValue<7>{}};

  // sort the elements of this range:
  std::ranges::sort(range);

  // print the elements of the range:
  std::ranges::for_each(range,
            [] (auto val) {
              std::cout << ' ' << val;
            });
  std::cout << '\n';

  // print all elements of coll up to -1:
  std::ranges::for_each(coll.begin(), EndValue<-1>{},
              [] (auto val) {
                std::cout << ' ' << val;
              });
  std::cout << '\n';
}
```

将 EndValue<> 定义为结束迭代器，检查作为模板形参传递的结束值。EndValue<7>{} 创建一个结束迭代器，其中 7 结束范围，EndValue<-1>{} 创建一个结束迭代器，其中-1 结束范围。

程序输出如下:

```
0 8 15 42
0 8 15 42 7
```

可以定义受支持的非类型模板参数类型的值。

作为哨兵的另一个例子，请查看 std::unreachable_sentinel。这是 C++20 定义的一个值，用来表示无限范围的“结束”，可以优化代码，使其永远不会与末尾进行比较 (因为若总是产生 false，则比较是无用的)。

有关子范围的更多方面，请参阅子范围的详细信息。

范围的开始和计数

范围库提供了多种处理定义为起始和计数的范围的方法。

使用 begin 迭代器和计数创建范围的最方便方法，是使用范围适配器 std::views::counted()。创建了一个指向 begin 迭代器/指针的前 n 个元素的低成本视图。

例如:

```
std::vector<int> coll{1, 2, 3, 4, 5, 6, 7, 8, 9};

auto pos5 = std::ranges::find(coll, 5);
if (std::ranges::distance(pos5, coll.end()) >= 3) {
  for (int val : std::views::counted(pos5, 3)) {
    std::cout << val << ' ';
  }
}
```

这里，std::views::counted(pos5, 3) 创建了一个视图，该视图表示从 pos5 所引用的元素开始的三个元素。注意，counted() 不会检查是否存在元素 (传递的计数过高会导致未定义行为)，开发者要确保代码的有效性。因此，需要使用 std::ranges::distance()，检查是否有足够的元素 (注意，若集合没有随机访问迭代器，这种检查的开销可能会很大)。

若知道有一个值为 5 的元素后面至少有两个元素，也可以这样:

```
// if we know there is a 5 and at least two elements behind:
for (int val : std::views::counted(std::ranges::find(coll, 5), 3)) {
  std::cout << val << ' ';
}
```

计数可能为 0，则该范围为空。

注意，只有当你确实有一个迭代器和一个计数时，才可以使用 counted()。若已经有了一个范围，并且只想处理前 n 个元素，请使用 std::views::take()。

详细信息请参见对 std::views::counted() 的描述。

6.1.6 投影

sort() 和许多其他用于范围的算法一样，通常有一个额外的可选模板参数，一个投影:

```
template<std::ranges::random_access_range R,
      typename Comp = std::ranges::less,
      typename Proj = std::identity>
requires std::sortable<std::ranges::iterator_t<R>, Comp, Proj>
... sort(R&& r, Comp comp = {}, Proj proj = {});
```

可选的附加参数，允许在算法进一步处理之前为每个元素指定一个转换 (投影)。

例如，sort() 允许指定要排序的元素的投影，而不是结果值的比较方式:

```
std::ranges::sort(coll,
        std::ranges::less{}, // still compare with <
        [] (auto val) { // but use the absolute value
          return std::abs(val);
        });
```

这可能比以下代码更具可读性或更容易编程:

```
std::ranges::sort(coll,
        [] (auto val1, auto val2) {
          return std::abs(val1) < std::abs(val2);
        });
```

有关完整示例，请参阅 ranges/rangesproject.cpp。

默认的投影是 std::identity()，只产生传递给它的参数，因此根本不执行投影/转换。(std::identity() 在 <function> 中定义为一个新的函数对象)。

用户定义的投影只需要接受一个参数并为转换后的参数返回一个值。

元素必须可排序的要求考虑了投影:

```
requires std::sortable<std::ranges::iterator_t<R>, Comp, Proj>
```

6.1.7 实现范围代码的工具

为了方便针对所有不同类型的范围进行编程，范围库提供了以下工具:

- 泛型函数，生成迭代器或范围的大小
- 类型函数，生成迭代器的类型或元素的类型

假设想要实现一个在一个范围内产生最大值的算法:

ranges/maxvalue1.hpp

```
#include <ranges>

```

```
template<std::ranges::input_range Range>
std::ranges::range_value_t<Range> maxValue(const Range& rg)
{
  if (std::ranges::empty(rg)) {
    return std::ranges::range_value_t<Range>{};
  }
  auto pos = std::ranges::begin(rg);
  auto max = *pos;
  while (++pos != std::ranges::end(rg)) {
    if (*pos > max) {
      max = *pos;
    }
  }
  return max;
}
```

这里，使用几个标准实用程序来处理范围 rg:

- 概念 std::ranges::input_range 要求传递的参数是一个可以读取的范围
- 类型函数 std::ranges::range_value_t，产生范围内相应类型的元素
- 辅助函数 std::ranges::empty() 判断产生范围是否为空
- 辅助函数 std::ranges::begin() 生成指向第一个元素的迭代器 (若有的话)
- 辅助函数 std::ranges::end() 产生范围的哨兵 (结束迭代器)

在这些类型工具的帮助下，因为也为它们定义了工具，所以该算法甚至适用于所有范围 (包括数组)。

例如，std::ranges::empty() 尝试调用成员函数 empty()、成员函数 size()、独立函数 size()，或者检查开始迭代器和哨兵 (结束迭代器) 是否相等。稍后将详细记录范围的工具函数。

注意，泛型 maxValue() 函数应该将传递的范围 rg 声明为通用引用 (也称为转发引用)，当它们是 const 时，无法迭代一些轻量级范围 (视图):

```
template<std::ranges::input_range Range>
std::ranges::range_value_t<Range> maxValue(Range&& rg)
```

稍后将详细讨论。

6.1.8 范围的局限性和缺点

C++20 中，范围也有一些主要的限制和缺点，应该在一般介绍中提到:

- 目前还没有对数值算法的范围支持。要将一个范围传递给数值算法，必须显式地传递 begin() 和 end():

```
std::ranges::accumulate(cont, 0L); // ERROR: not provided
std::accumulate(cont.begin(), cont.end(), 0L); // OK
```

- 目前还不支持并行算法的范围。

```
std::ranges::sort(std::execution::par, cont); // ERROR: not provided
std::sort(std::execution::par, cont.begin(), cont.end()); // OK
```

通过向视图传递 begin() 和 end() 来使用现有的并行算法时，应该谨慎。对于某些视图，并发迭代会导致未定义行为，只有在将视图声明为 const 之后才这样做。

- 一些由开始迭代器和结束迭代器定义的范围的传统 api 要求迭代器具有相同的类型 (例如，这适用于容器和命名空间 std 中的算法)。可能需要与 std::views::common() 或 std::common_iterator 协调其类型。
- 对于某些视图，当视图为 const 时，不能遍历元素，所以泛型代码可能必须使用通用/转发引用。
- cbegin() 和 cend() 函数的设计目的是确保不会 (意外地) 修改所遍历的元素，但对于引用非 const 对象的视图来说无效。
- 当视图引用容器时，可能会破坏 const 的传播。
- 范围会导致严重的命名空间混乱。例如，下面的 std::find() 声明，所有标准名称都是完全限定的:

```
template<std::ranges::input_range Rg,
         typename T,
         typename Proj = std::identity>
requires std::indirect_binary_predicate<std::ranges::equal_to,
                  std::projected<std::ranges::iterator_t<Rg>, Proj>,
                  const T*>
constexpr std::ranges::borrowed_iterator_t<Rg>
  find(Rg&& rg, const T& value, Proj proj = {});
```

要知道在哪里应该使用哪个命名空间确实不容易。

此外，在命名空间 std 和命名空间 std::ranges 中，有些符号的行为略有不同。上面的声明中，equal_to 就是这样一个例子。也可以使用 std::equal_to，但 std::ranges 中的工具提供了更好的支持，并且对于极端情况的处理会更健壮。

6.2. 租借迭代器和范围

当将范围作为单个参数传递给算法时，会遇到生命周期的问题。本节描述范围库如何处理此问题。

6.2.1 租借迭代器

许多算法将迭代器返回所操作的范围，将范围作为单个参数传递时，可能会遇到一个新问题，当范围需要两个参数 (开始迭代器和结束迭代器) 时，是不可能的: 若传递一个临时范围 (函数返回的范围) 并返回迭代器，当范围销毁时，返回的迭代器可能会在语句结束时失效。使用返回的迭代器 (或其副本) 会导致未定义行为。

例如，考虑将一个临时范围传递给 find() 算法，该算法在一个范围中搜索一个值:

```
std::vector<int> getData(); // forward declaration

auto pos = find(getData(), 42); // returns iterator to temporary vector
// temporary vector returned by getData() is destroyed here
std::cout << *pos; // OOPS: using a dangling iterator
```

getData() 的返回值在使用它的语句结束时销毁，所以 pos 指的是一个不再存在的集合元素，使用 pos 会导致未定义行为 (则会得到一个段错误，这样就可以看到存在问题了)。

为了解决这个问题，范围标准库引入了租借迭代器的概念。租借迭代器确保其生命周期不依赖于可能已销毁的临时对象。若存在，使用它会导致编译时错误。租借迭代器会发出信号，表明是否可以安全地超过所传递的范围，若该范围不是临时的，或者迭代器的状态不依赖于所传递范围的状态，就会出现这种情况。若有一个租借迭代器来引用一个范围，则这个迭代器可以安全使用，即使在范围销毁时也不会悬空。[因此，范围标准库的草案版本中，这样的迭代器被称为“安全迭代器”。]

使用类型 std::ranges::borrowed_iterator_t<>，算法可以将返回的迭代器声明为租借的，所以该算法会返回一个可以在语句后安全使用的迭代器。若为悬空，则使用一个特殊的返回值来发出信号，并将可能的运行时错误转换为编译时错误。

例如，单个范围的 std::ranges::find() 声明如下:

```
template<std::ranges::input_range Rg,
      typename T,
      typename Proj = identity>
...
constexpr std::ranges::borrowed_iterator_t<Rg>
  find(Rg&& r, const T& value, Proj proj = {});
```

通过将返回类型指定为 Rg 的 std::ranges::borrow_iterator_t<>，该标准启用了编译时检查: 若传递给算法的范围 R 是临时对象 (右值)，则返回类型变为悬空迭代器。这时，返回值是 std::ranges::dangling 类型的对象，对此类对象的使用 (复制和赋值除外) 都会导致编译时错误。

下面的代码会出现编译时错误:

```
std::vector<int> getData(); // forward declaration

auto pos = std::ranges::find(getData(), 42); // returns iterator to temporary vector
// temporary vector returned by getData() was destroyed
std::cout << *pos; // compile-time ERROR
```

为了能够为临时对象使用 find()，必须将其作为左值传递，所以其必须有一个名字，所以算法确保调用集合后仍然存在，还可以检查是否找到了一个值 (这通常是合适的)。

为返回集合指定名称的最佳方法是将其绑定到引用，集合就永远不会复制。请注意，根据规则，临时对象的引用总是会延长其生命周期:

```
std::vector<int> getData(); // forward declaration

reference data = getData(); // give return value a name to use it as an lvalue
// lifetime of returned temporary vector ends now with destruction of data
...
```

可以在这里使用两种引用:

- 可以声明一个 const 左值引用:

```
std::vector<int> getData(); // forward declaration

const auto& data = getData(); // give return value a name to use it as an lvalue
auto pos = std::ranges::find(data, 42); // yields no dangling iterator
if (pos != data.end()) {
  std::cout << *pos; // OK
}
```

这个引用使返回值为 const，这可能不是期望的 (注意，当某些视图是 const 时，不能进行迭代; 但由于视图的引用语义，在返回时必须非常谨慎)。

- 更泛型的代码中，应该使用通用引用 (也称为转发引用) 或 decltype(auto)，以便保持返回值的“非”const:

```
... getData(); // forward declaration

auto&& data = getData(); // give return value a name to use it as an lvalue
auto pos = std::ranges::find(data, 42); // yields no dangling iterator
if (pos != data.end()) {
  std::cout << *pos; // OK
}
```

这个特性的副作用是，即使结果代码有效，也不能将临时对象传递给算法:

```
process(std::ranges::find(getData(), 42)); // compile-time ERROR
```

尽管迭代器在函数调用期间是有效的 (临时 vector 将在调用后销毁)，find() 返回一个 std::ranges::dangling 对象。

同样，处理此问题的最佳方法是声明 getData() 返回值的引用:

- 使用 const 左值引用:

```
const auto& data = getData(); // give return value a name to use it as an lvalue
process(std::ranges::find(data, 42)); // passes a valid iterator to process()
```

- 使用通用/转发引用:

```
auto&& data = getData(); // give return value a name to use it as an lvalue
process(std::ranges::find(data, 42)); // passes a valid iterator to process()
```

通常情况下，需要一个返回值的名称来检查返回值是否指向一个元素，而不是一个范围的 end():

```
auto&& data = getData(); // give return value a name to use it as an lvalue
auto pos = std::ranges::find(data, 42); // yields a valid iterator
if (pos != data.end()) { // OK
  std::cout << *pos; // OK
}
```

6.2.2 租借范围

范围类型可以声明其是租借范围。当范围本身不再存在时，迭代器仍可以使用。

C++20 为此提供了 std::ranges::borrowed_range 的概念。若范围类型的迭代器从不依赖于其范围的生存期，或者传递的范围对象为左值，则满足此概念，该概念检查该范围创建的迭代器在该范围不存在后是否可以使用。

迭代器迭代存储在其中的值，所有引用作为右值 (临时范围对象) 传递的范围的标准容器和视图都不是租借范围，所以有两种方法让视图成为租借范围:

- 迭代器存储所有用于本地迭代的信息。例如:
  - std::ranges::iota_view，生成一个递增的值序列。迭代器在本地存储当前值，并且不引用任何其他对象。
  - std::ranges::empty_view，任何迭代器总是在末尾，因此其根本不能迭代元素值。
- 迭代器直接引用底层范围，而不使用调用 begin() 和 end() 的视图。例如:
  - std::ranges::subrange
  - std::ranges::ref_view
  - std::span
  - std::string_view

注意，当租借迭代器引用基础范围 (上面的后一类)，并且基础范围不再存在时，仍然可以悬空。

因此，可以在编译时捕获一些 (但不是所有) 可能的运行时错误，可以用各种方法来演示如何在不同范围内查找值为 8 的元素 (是的，通常应该检查是否返回了 end 迭代器):

- 所有左值 (有名字的对象) 都是租借的范围，只要迭代器存在于范围的同一作用域或子作用域中，返回的迭代器就不能悬空。

```
std::vector coll{0, 8, 15};

auto pos0 = std::ranges::find(coll, 8); // borrowed range
std::cout << *pos0; // OK (undefined behavior if no 8)

auto pos1 = std::ranges::find(std::vector{8}, 8); // yields dangling
std::cout << *pos1; // compile-time ERROR
```

- 对于临时视图，视情况而定。例如:

```
auto pos2 = std::ranges::find(std::views::single(8), 8); // yields dangling
std::cout << *pos2; // compile-time ERROR

```

```
auto pos3 = std::ranges::find(std::views::iota(8), 8); // borrowed range
std::cout << *pos3; // OK (undefined behavior if no 8 found)

auto pos4 = std::ranges::find(std::views::empty<int>, 8); // borrowed range
std::cout << *pos4; // undefined behavior as no 8 found
```

例如，单视图迭代器引用视图中元素的值; 因此，单视图不是租借范围。

另一方面，iota 视图迭代器保存它们所引用的元素的副本，所以 iota 视图会声明为租借范围。

- 对于引用另一个范围 (作为一个整体或它的子序列) 的视图，情况更加复杂，尝试就会发现类似的问题。例如，适配器 std::views::take() 也检查右值:

```
auto pos5 = std::ranges::find(std::views::take(std::vector{0, 8, 15}, 2), 8);
// compile-time ERROR
```

这里，使用 take() 会出现一个编译时错误。

然而，使用 count()，其只接受一个迭代器，开发者有责任确保迭代器的有效性:

```
auto pos6 = std::ranges::find(std::views::counted(std::vector{0, 8, 15}.begin(),
              2), 8);
std::cout << *pos6; // runtime ERROR even if 8 found
```

这里用 count() 创建的视图，根据定义是租借范围，因为其将内部引用传递的迭代器。换句话说: 计数视图的迭代器不需要它所属的视图。然而，迭代器仍然可以引用一个不再存在的范围 (因为视图引用了一个不再存在的对象)。示例的最后一行用 pos6 演示了这种情况，即使 find() 正在查找的值可以在临时范围内找到，仍然会出现未定义行为。

若实现了一个容器或视图，可以通过特化变量模板 std::ranges::enable_borrowing_range<> 来表示租借范围。

6.3. 使用视图

如前所述，视图是轻量级范围，可以用作构建块来处理其他范围和视图的所有或部分元素的 (修改过的) 值。

C++20 提供了几个标准视图，可以用来将一个范围转换为一个视图，或者将一个视图转换为一个范围/视图，其中元素可以以各种方式修改:

- 过滤元素
- 生成转换后的元素值
- 修改迭代元素的顺序
- 拆分或合并范围

此外，还有一些视图产生的值。

对于每一种视图类型，都有一个相应的定制点对象 (通常是函数对象)，允许开发者调用函数来创建视图。若函数从传递的范围中创建视图，则这样的函数对象称为“范围适配器”。若函数在不传递现有范围的情况下创建视图，则称为“范围工厂”。大多数情况下，这些函数对象具有不带_view

后缀的视图名，但一些更通用的函数对象可能会根据传递的参数创建不同的视图。这些函数对象都定义在特殊的命名空间 std::views 中，这命名空间 std::ranges::views 的别名。

“源视图”表列出了 C++20 中从外部资源创建视图或生成值的标准视图，这些视图可以作为视图管道中的起始构建块。还可以查看哪些范围适配器或工厂可以创建它们 (若有的话)，更推荐使用，而非原始视图类型。

若没有另行指定，适配器和工厂在命名空间 std:: 视图中可用，视图类型在名称空间 std::ranges 中可用。std::string_view 已经在 C++17 中引入，所有其他视图都是在 C++20 中引入的，通常以_view 结尾。唯一不以_view 结尾的视图类型是 std::subange 和 std::span。

“适配视图”表列出了 C++20 中处理范围和其他视图的范围适配器和标准视图，可以作为视图管道中的任何地方的构建块，包括在视图的开头。同样，推荐使用。

所有视图都提供了具有恒定复杂性的移动 (和可选的复制) 操作 (这些操作所需的时间与元素的数量无关)[原始 C++20 标准还要求视图具有具有恒定复杂性的默认构造函数和析构函数，但这些要求后来因http://wg21.link/P2325R3和http://wg21.link/p2415r2删除了]。概念 std::ranges::view 检查相应的需求。

范围工厂/适配器 all()、counts() 和 common() 将在一个特殊的章节中描述，所有视图类型，以及其他适配器和工厂的详细信息将在视图类型细节一章中描述。

适配器/工厂	类型	效果
all(rg)	种类: rg 的类型 ref_view owning_view	生成的范围 rg 是视图 - 若已经是视图，则返回 rg - 若 rg 是左值，则返回 ref_view - 若 rg 是右值，则返回一个 owning_view
counted(beg, sz)	种类: std::span 子范围	从 begin 迭代器和计数产生一个视图 - 若 rg 是连续且常规的，则生成一个 span - 否则产生子范围 (若有效)
iota(val)	iota_view	生成一个无限视图，其中包含以 val 开头的值的递增序列
iota(val, endVal)	iota_view	生成一个视图，其值序列从 val 递增到 (但不包括)endVal
single(val)	signle_view	生成一个只有 val 元素的视图
empty<T> -	empty_view basic_istream_view	生成类型为 T 的元素的空视图 生成读取 T 类型的 T 个元素的视图
istream<T>(s) - - - -	istream_view wistream_view std::basic_string_view std::span subrange	生成从字符流 s 读取 Ts 的视图 生成一个从 wchar_t 流 s 中读取 Ts 的视图 生成字符数组的只读视图 生成连续内存中元素的视图 生成 begin 迭代器和哨兵的视图

表 6.3 源视图

适配器	类型	效果
take(num)	变化	第一个 (最多)num 个元素
take_while(pred)	take_while_view	匹配谓词的所有首元素
drop(num)	变化	除了第一个 num 元素之外的所有元素
drop_while(pred)	drop_while_view	除首元素外的所有匹配谓词的元素
filter(pred)	filter_view	匹配谓词的所有元素
transform(func)	transform_view	转换后的值的所有元素
elements<idx>	elements_view	所有元素的 idxth 成员/属性
keys	elements_view	所有元素的第一个成员
values	elements_view	所有元素的第二个成员
reverse	变化	所有元素按倒序排列
join	join_view	多个范围中的所有元素
split(sep)	split_view	一个范围的所有元素拆分为多个范围
lazy_split(sep)	lazy_split_view	输入或常量范围的所有元素拆分为多个范围
common	变化	迭代器和哨兵中所有类型相同的元素

表 6.4 适配器视图

6.3.1 视图的范围性

容器和字符串不是视图，因为不够轻量级: 没有提供低成本的复制构造函数，所以必须复制元素。

然而，可以很容易地使用容器作为视图:

- 通过将容器传递给范围适配器 std::views::all()，可以显式地将容器转换为视图。
- 通过将 begin 迭代器和 end(哨兵) 或大小传递给 std::ranges::subrange 或 std::views::counts()，可以显式地将容器的元素转换为视图。
- 可以通过将容器传递给其中一个自适应视图来隐式地将其转换为视图，这些视图通常通过将容器隐式地转换为视图来获取容器。

通常，应该使用最后一种选项，不过有更多方法可以用来实现此功能。例如，有以下选项来传递范围 coll 给获取视图:

- 可以将范围作为参数传递给视图的构造函数:

```
std::ranges::take_view first4{coll, 4};
```

- 可以把这个范围作为参数传递给相应的适配器:

```
auto first4 = std::views::take(coll, 4);
```

- 可以通过管道将范围传入相应的适配器:

```
auto first4 = coll | std::views::take(4);
```

视图 first4 只迭代 coll 的前 4 个元素 (若元素不够，迭代次数会更少)，但这里发生了什么取决于 coll 是什么类型:

- 若 coll 已经是一个视图，take() 会按原样获取视图。
- 若 coll 是一个容器，take() 使用一个到容器的视图，该视图是用适配器 std::views::all() 自动创建的。适配器会产生一个 ref_view，若容器通过名称传递 (作为左值)，将引用容器的所有元素。
- 若传递了一个右值 (临时范围，比如函数返回的容器或带有 std::move() 标记的容器)，则该范围将移动到 owning_view 中，然后 owning_view 直接保存传递类型的范围，其中包含所有移动的元素。[C++20 发布http://wg21.link/p2415r2后增加了对视图临时对象 (右值) 的支持。]

例如:

```
std::vector<std::string> coll{"just", "some", "strings", "to", "deal", "with"};

auto v1 = std::views::take(coll, 4); // iterates over a ref_view to coll

auto v2 = std::views::take(std::move(coll), 4); // iterates over an owning_view
// to a local vector<string>

auto v3 = std::views::take(v1, 2); // iterates over v1
```

std::views::take() 会创建一个新的获取视图，最终迭代在 coll 中初始化的值，但结果类型和确切的行为有如下不同:

- v1 是 take_view<ref_view<vector<string>>>。
  将容器 coll 作为左值 (命名对象) 传递，所以获取视图通过一个 ref 视图迭代到容器。
- v2 是 take_view<owning_view<vector<string>>>.
  将 coll 作为右值 (临时对象或用 std::move() 标记的对象) 传递，所以获取视图遍历拥有视图，该视图拥有用传递的集合初始化的字符串 vector。
- v3 是 take_view<take_view<ref_view<vector<string>>>>.
  因为传递了视图 v1, 获取视图遍历这个视图。最终遍历了 coll(其中的元素已经由第二条语句移除了，所以第二条语句之后不需要再遍历了)。

在内部，初始化使用可推导指南和类型工具 std::views::all_t<>，后面将详细解释。

注意，这种行为允许基于范围的 for 循环迭代临时范围:

```
for (const auto& elem : getColl() | std::views::take(5)) {
  std::cout << "- " << elem << '\n';
}

for (const auto& elem : getColl() | std::views::take(5) | std::views::drop(2)) {
  std::cout << "- " << elem << '\n';
}
```

通常，使用对临时对象的引用作为基于范围的 for 循环迭代的集合是一个致命的运行时错误 (这是 C++ 标准委员会多年来一直不愿意修复的一个错误，参见http://wg21.link/p2012)。由于

传递临时范围对象 (rvalue) 将范围移动到 owning_view 中，因此视图不会引用外部容器，从而不会出现运行时错误。

6.3.2 惰性计算

理解什么时候处理视图是很重要的。视图在定义后不会开始处理，其会按需运行:

- 若需要视图的下一个元素，则通过执行必要的迭代来计算其是哪一个。
- 若需要一个视图元素的值，测通过执行定义的转换来计算其值。

看下下面的代码:

ranges/filttrans.cpp

```
#include <iostream>
#include <vector>
#include <ranges>
namespace vws = std::views;

int main()
{
  std::vector<int> coll{ 8, 15, 7, 0, 9 };

  // define a view:
  auto vColl = coll
         | vws::filter([] (int i) {
           std::cout << " filter " << i << '\n';
           return i % 3 == 0;
         })
         | vws::transform([] (int i) {
           std::cout << " trans " << i << '\n';
           return -i;
         });

  // and use it:
  std::cout << "*** coll | filter | transform:\n";
  for (int val : vColl) {
    std::cout << "val: " << val << "\n\n";
  }
}
```

定义了一个视图 vColl，只过滤和转换范围 coll 的元素:

- 使用 std::views::filter()，只处理那些是 3 的倍数的元素。
- 使用 std::views::transform()，对每个值求反。

该程序有以下输出:

```
*** coll | filter | transform:
filter 8
```

```
filter 15
trans 15
val: -15

filter 7
filter 0
trans 0

val: 0
filter 9
trans 9
val: -9
```

首先，定义视图 vColl 时或之后，不会调用 filter() 和 transform()。当使用视图时，处理就开始了(这里: 迭代 vColl)。视图使用延迟求值，只是对处理的描述，当需要下一个元素或值时才进行处理。

假设对 vColl 进行更多的手动迭代，调用 begin() 和 ++ 来获取下一个值，使用解引用操作符来获取其值:

```
std::cout << "pos = vColl.begin():\n";
auto pos = vColl.begin();
std::cout << "*pos:\n";
auto val = *pos;
std::cout << "val: " << val << "\n\n";

std::cout << "++pos:\n";
++pos;
std::cout << "*pos:\n";
val = *pos;
std::cout << "val: " << val << "\n\n";
```

对于这段代码，可以得到以下输出:

```
pos = vColl.begin():
filter 8
filter 15
*pos:
trans 15
val: -15

++pos:
filter 7
filter 0
*pos:
trans 0
val: 0
```

一一步来看看会发生什么:

- 当调用 begin() 时，会发生以下情况:
  - 获取 vColl 的第一个元素的请求会传递给转换视图，转换视图将其传递给过滤器视图，过滤器视图将其传递给 coll，后者生成迭代器的第一个元素 8。
  - 过滤器视图查看第一个元素的值并拒绝，并通过调用 ++ 向 coll 请求下一个元素。过滤器获取第二个元素的位置 15，并将其位置传递给变换视图。
  - 因此，pos 初始化为指向第二个元素的位置的迭代器。
- 当使用 *pos 时，会发生以下情况:
  - 因为需要这个值，所以现在为当前元素调用转换视图，并生成其负值。
  - 用当前元素的负值初始化 val。
- 当使用 ++pos 时，同样的情况会再次发生:
  - 获取下一个元素的请求传递给过滤器，过滤器将请求传递给 coll，直到找到合适的元素 (或者到达 coll 的末尾)。
  - 因此，pos 获取第四个元素的位置。
- 通过再次调用 *pos，执行转换并为循环生成下一个值。

这个迭代会一直持续下去，直到范围或其中一个视图表示已经到达终点。

这种 pull 模型有一个很大的好处: 不处理不需要的元素。例如，使用视图来查找第一个结果值为 0:

```
1 std::ranges::find(vColl, 0);
```

那么输出将只有:

```
filter 8
filter 15
trans 15
filter 7
filter 0
trans 0
```

pull 模型的另一个好处是视图的序列或管道甚至可以在无限范围内操作。不会在不知道使用了多少的情况下计算无限个值，从而根据视图的用户请求计算尽可能多的值。

6.3.3 缓存视图

假设想要多次迭代一个视图。若一次又一次地计算第一个有效元素，似乎是对性能的浪费，而跳过的元素视图会在调用 begin() 后进行缓存。

修改上面的程序，使其在视图 vColl 的元素上迭代两次:

ranges/filttrans2.cpp

```
1 #include <iostream>
2 #include <vector>
3 #include <ranges>
4 namespace vws = std::views;
```

```cpp

int main()
{
  std::vector<int> coll{ 8, 15, 7, 0, 9 };
  // define a view:
  auto vColl = coll
  | vws::filter([] (int i) {
    std::cout << " filter " << i << '\n';
    return i % 3 == 0;
  })
  | vws::transform([] (int i) {
    std::cout << " trans " << i << '\n';
    return -i;
  });

  // and use it:
  std::cout << "*** coll | filter | transform:\n";
  for (int val : vColl) {
    ...
  }
  std::cout << "--------------------\n";

  // and use it again:
  std::cout << "*** coll | filter | transform:\n";
  for (int val : vColl) {
    std::cout << "val: " << val << "\n\n";
  }
}
```

输出如下所示：

```
*** coll | filter | transform:
filter 8
filter 15
trans 15
filter 7
filter 0
trans 0
filter 9
trans 9
--------------------
*** coll | filter | transform:
trans 15
val: -15

filter 7
filter 0
trans 0
val: 0
```

```
filter 9
trans 9
val: -9
```

第二次使用 vcol .begin() 时，不再尝试查找第一个元素，因为与过滤器元素的第一次迭代一起进行了缓存。

注意，begin() 的缓存有好有坏，可能还有意想不到的后果，最好初始化一次缓存视图并使用两次:

```cpp
// better:
auto v1 = coll | std::views::drop(5);
check(v1);
process(v1);
```

然后初始化并使用两次:

```cpp
// worse:
check(coll | std::views::drop(5));
process(coll | std::views::drop(5));
```

此外，修改范围的前置元素 (改变它们的值或插入/删除元素) 可能会使视图失效，当且仅当修改之前已经调用了 begin()。

所以:

- 若在修改之前不调用 begin()，这个视图通常是有效的，以后使用时也能正常工作:

```cpp
std::list coll{1, 2, 3, 4, 5};
auto v = coll | std::views::drop(2);
coll.push_front(0); // coll is now: 0 1 2 3 4 5
print(v); // initializes begin() with 2 and prints: 2 3 4 5
```

- 但若在修改之前调用 begin()(例如，打印元素)，就很容易得到错误的元素。例如:

```cpp
std::list coll{1, 2, 3, 4, 5};
auto v = coll | std::views::drop(2);
print(v); // init begin() with 3
coll.push_front(0); // coll is now: 0 1 2 3 4 5
print(v); // begin() is still 3, so prints: 3 4 5
```

这里，begin() 缓存为一个迭代器，若向范围中添加或删除新元素，视图就不再对基础范围中的所有元素进行操作。

也可能得到无效的值。例如:

```cpp
std::vector vec{1, 2, 3, 4};

auto biggerThan2 = [](auto v){ return v > 2; };
auto vVec = vec | std::views::filter(biggerThan2);
```

```
5
6  print(vVec); // OK: 3 4
7
8  ++vec[1];
9  vec[2] = 0; // vec becomes 1 3 0 4
10
11 print(vVec); // OOPS: 0 4;
```

注意，这是一个迭代，即使只是读，也可以算作写访问，若视图的引用范围在此期间修改，则迭代视图的元素可能会使以后的使用无效。

其效果取决于何时，以及如何进行缓存。请参阅关于缓存视图特定部分的注释:

- 过滤视图，将 begin() 缓存为迭代器或偏移量
- 丢弃视图，将 begin() 缓存为迭代器或偏移量
- 丢弃段视图，将 begin() 缓存为迭代器或偏移量
- 反向 (Reverse) 视图，将 begin() 缓存为迭代器或偏移量

这里，看看 C++ 对性能的关心:

- 若根本不遍历视图的元素，那么在初始化时进行缓存将会带来不必要的性能开销。
- 若在视图的元素上迭代一秒或更多次，完全不缓存将会带来不必要的性能成本 (某些情况下，在丢弃段视图上应用反向视图，甚至可能具有二次复杂度)。

由于缓存，使用非临时视图可能会产生非常令人惊讶的结果。所以，在修改视图使用的范围时必须小心。

另一个结果是，缓存可能要求视图在遍历其元素时不能是 const。其后果更为严重，稍后再讨论。

6.3.4 过滤器的性能问题

Pull 模式也有缺点。为了演示这一点，改变上面涉及的两个视图的顺序，以便先调用 transform()，然后调用 filter():

ranges/transfilt.cpp

```
1  #include <iostream>
2  #include <vector>
3  #include <ranges>
4  namespace vws = std::views;
5
6  int main()
7  {
8    std::vector<int> coll{ 8, 15, 7, 0, 9 };
9
10   // define a view:
11   auto vColl = coll
12   | vws::transform([] (int i) {
13     std::cout << " trans: " << i << '\n';
```

```
    return -i;
  })
  | vws::filter([] (int i) {
    std::cout << " filt: " << i << '\n';
    return i % 3 == 0;
  });

  // and use it:
  std::cout << "*** coll | transform | filter:\n";
  for (int val : vColl) {
  std::cout << "val: " << val << "\n\n";
  }
}
```

程序的输出如下所示:

```
*** coll | transform | filter:
trans: 8
filt: -8
trans: 15
filt: -15
trans: 15
val: -15

trans: 7
filt: -7
trans: 0
filt: 0
trans: 0
val: 0

trans: 9
filt: -9
trans: 9
val: -9
```

还调用了 transform 视图:

- 现在对每个元素使用 transform()。
- 对于通过过滤器的元素，并执行两次转换。

为每个元素调用转换是必要的，因为过滤器视图在之后进行，现在需要查看转换后的值。这时，取反操作不影响过滤器，所以把它放在前面会更好。

然而，为什么要对通过过滤器的元素调用两次变换呢? 原因在于使用 Pull 模型的管道的性质，以及使用迭代器的事实。

- 首先，过滤器需要转换后的值来进行检查，必须在过滤器的结果之前执行转换。

- 记住，范围和视图在范围的元素上分两步迭代: 首先计算位置/迭代器 (使用 begin() 和 ++)，然后作为单独的步骤，使用解引用操作符来获取值。所以对于每个过滤器来说，前面所有的变换都必须执行一次，才能检查元素的值。但若设置为 true，过滤器只返回元素的位置，而不返回值。所以当使用过滤器的用户需要某个值时，就必须再次执行转换操作。

实际上，每个 filter 都会增加通过过滤器的元素的所有前置变换的调用，并在之后使用这些值。给定下列转换管道 t1, t2, t3 和过滤器 f1, f2:

```
t1 | t2 | f1 | t3 | f2
```

有以下行为:

- 对于 f1 为 false 的元素:

```
t1 t2 f1
```

- 对于 f1 为真而 f2 为假的元素:

```
t1 t2 f1 t1 t2 t3 f2
```

- 对于 f1 和 f2 为真的元素:

```
t1 t2 f1 t1 t2 t3 f2 t1 t2 t3
```

查看 ranges/viewscals.cpp 获得一个完整的示例。

若担心管道的性能，可以考虑以下几点: 应该在使用过滤器之前避免耗时的转换。所有视图提供的功能通常都是以这样一种方式进行优化的，即只保留转换和过滤器中的表达式。在一些简单的情况下，过滤器检查 3 的倍数并且转换为负值，结果的行为差异实际上就像调用一样

```
if (-x % 3 == 0) return -x; // first transformation then filter
```

而非

```
if (x % 3 == 0) return -x; // first filter then transformation
```

有关过滤器的更多细节，请参阅关于过滤器视图一节。

6.4. 销毁或修改范围的视图

视图通常具有引用语义，通常指的是存在于自身之外的范围，所以必须谨慎，只有当底层范围存在并且存储在视图或其迭代器中的对它们的引用有效时，才能使用视图。

6.4.1 视图及其范围之间的生命周期依赖关系

所有对作为左值传递的范围 (作为第一个构造函数参数或使用管道) 进行操作的视图，都在内部存储对传递范围的引用。

使用视图时，底层范围必须存在。这样的代码会导致未定义行为:

```
auto getValues()
{
  std::vector coll{1, 2, 3, 4, 5};
  ...
  return coll | std::views::drop(2); // ERROR: return reference to local range
}
```

这里返回的是按值放视图。在内部，其引用 coll，在 getValues() 结束时销毁。

这段代码与返回指向局部对象的引用或指针一样糟糕，可能会正常工作或导致致命的运行时错误。不幸的是，编译器 (目前) 不会对此发出警告。

在右值范围对象上使用视图很好，可以将一个视图返回给一个临时范围对象:

```
auto getValues()
{
  ...
  return std::vector{1, 2, 3, 4, 5} | std::views::drop(2); // OK
}
```

或可以用 std::move() 标记底层范围:

```
auto getValues()
{
  std::vector coll{1, 2, 3, 4, 5};
  ...
  return std::move(coll) | std::views::drop(2); // OK
}
```

6.4.2 具有写访问权限的视图

视图不应该修改传递的参数或为其调用非 const 操作，视图及其副本对于相同的输入具有相同的行为。

对于使用辅助函数检查或转换值的视图，这些辅助函数永远不应该修改元素。理想情况下，应该按值或按 const 引用取值。若修改作为非 const 引用传递的实参，就会有未定义行为:

```
coll | std::views::transform([] (auto& val) { // better declare val as const&
  ++val; // ERROR: undefined behavior
})

coll | std::views::drop([] (auto& val) { // better declare val as const&
  return ++val > 0; // ERROR: undefined behavior
})
```

注意，编译器不能检查辅助函数或谓词是否修改了传递的值。视图要求传递的函数或谓词是 std::regular_invocable 的 (这是 std::predicate 的隐式要求)。但不修改值是语义约束，在编译时不能总是检查这一点，所以要由开发者来做。

但是，支持使用视图来限制要修改的元素子集。例如:

```
// assign 0 to all but the first five elements of coll:
for (auto& elem : coll | vws::drop(5)) {
  elem = 0;
}
```

6.4.3 更改范围的视图

缓存 begin() 的视图若没有特别使用，并且底层范围发生了变化，可能会遇到严重的问题 (请注意，我写的是在范围和调用 begin() 仍然有效时发生的问题)。

考虑下面的程序:

ranges/viewslazy.cpp

```
#include <iostream>
#include <vector>
#include <list>
#include <ranges>

void print(auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << ' ' << elem;
  }
  std::cout << '\n';
}

int main()
{
  std::vector vec{1, 2, 3, 4, 5};
  std::list lst{1, 2, 3, 4, 5};

  auto over2 = [](auto v) { return v > 2; };
  auto over2vec = vec | std::views::filter(over2);
  auto over2lst = lst | std::views::filter(over2);

  std::cout << "containers and elements over 2:\n";
  print(vec); // OK: 1 2 3 4 5
  print(lst); // OK: 1 2 3 4 5
  print(over2vec); // OK: 3 4 5
  print(over2lst); // OK: 3 4 5

  // modify underlying ranges:
  vec.insert(vec.begin(), {9, 0, -1});
  lst.insert(lst.begin(), {9, 0, -1});

  std::cout << "containers and elements over 2:\n";
  print(vec); // vec now: 9 0 -1 1 2 3 4 5
```

```
  print(lst); // lst now: 9 0 -1 1 2 3 4 5
  print(over2vec); // OOPS: -1 3 4 5
  print(over2lst); // OOPS: 3 4 5

  // copying might eliminate caching:
  auto over2vec2 = over2vec;
  auto over2lst2 = over2lst;
  std::cout << "elements over 2 after copying the view:\n";
  print(over2vec2); // OOPS: -1 3 4 5
  print(over2lst2); // OK: 9 3 4 5
  }
```

输出如下:

```
containers and elements over 2:
1 2 3 4 5
1 2 3 4 5
3 4 5
3 4 5
containers and elements over 2:
9 0 -1 1 2 3 4 5
9 0 -1 1 2 3 4 5
-1 3 4 5
3 4 5
elements over 2 after copying the view:
-1 3 4 5
9 3 4 5
```

问题在于过滤器视图的缓存。第一次迭代中，两个视图都缓存视图的开头。注意，这里不同的方式:

- 对于像 vector 这样的随机访问范围，视图将偏移量缓存到第一个元素，当再次使用视图时，使 begin() 失效的重新分配就不会导致未定义行为。
- 对于其他范围 (如列表)，视图实际上会缓存调用 begin() 的结果。若缓存的元素被删除，则缓存的 begin() 不再有效，这可能是一个严重的问题。但若前面插入新的元素，也会造成混乱。

缓存的总体效果是，对底层范围的进一步修改可能会以各种方式使视图无效:

- 缓存的 begin() 可能不再有效。
- 缓存的偏移量可能在基础范围的末尾之后。
- 符合谓词的新元素可能不会在以后的迭代中使用。
- 不适合的新元素可能会在后面的迭代中使用。

6.4.4 复制视图可能会改变行为

最后，前面的例子演示了复制视图有时可能会使缓存无效:

- 缓存了 begin() 的视图有如下输出:

```
-1 3 4 5
3 4 5
```

- 这些视图的副本有不同的输出:

```
-1 3 4 5
9 3 4 5
```

复制后，视图到 vector 的缓存偏移量仍然使用，则删除视图到列表 (begin()) 的缓存 begin()。

缓存有一个额外的后果，即视图的副本可能与其源的状态不同。由于这个原因，在复制视图之前应该三思而行 (尽管一个设计目标是通过值来降低复制的成本)。

若这种行为有一个后果，就是: 特别使用视图 (在视图定义之后立即使用)。

这有点可悲，因为视图的惰性求值。原则上，会允许一些难以置信的用例，这在实践中无法实现，因为代码的行为很难预测，若视图不是特别使用，底层范围可能会改变。

过滤器视图的写访问

当使用过滤器视图时，写访问有一些重要的附加限制:

- 首先，由于过滤视图缓存，修改后的元素可能会通过过滤器，即使不应该通过。
- 此外，必须确保修改后的值仍然满足传递给过滤器的谓词。否则，将得到未定义行为 (尽管有时会产生正确的结果)。有关详细信息和示例，请参见过滤视图的描述。

6.5. 视图和常量

在使用视图 (以及一般的范围库) 时，有一些关于常量性的事情令人惊讶，甚至是不正常的。

- 对于某些视图，当视图为 const 时，不可能遍历元素。
- 视图消除了对元素常量性的传播。
- 像 cbegin() 和 cend() 这样的函数的目标是确保元素在迭代时为 const，要么没有提供，要么已破坏。

这些问题或多或少都有充分的理由，其中一些与视图的性质和所做的一些基本设计决策有关。我认为这是一个严重的设计错误，但 C++ 标准委员会中的其他人有不同的观点。

目前正在进行修复，至少修复了 cbegin() 和 cend() 的连续性，但 C++ 标准委员会决定不在 C++20 中进行应用。这些将在 C++23 中实现 (并改变行为)，详见http://wg21.link/p2278r4。

6.5.1 视图及其范围之间的生命周期依赖关系

实现一个可以遍历每种类型的容器和视图的元素，并具有良好性能的泛型函数是非常复杂的。像下面这样声明一个打印所有元素的函数并不总有效:

```
1 template<typename T>
2 void print(const T& coll); // OOPS: might not work for some views
```

考虑下面的具体例子:

ranges/printconst.cpp

```cpp
#include <iostream>
#include <vector>
#include <list>
#include <ranges>

void print(const auto& rg)
{
  for (const auto& elem : rg) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
  std::list lst{1, 2, 3, 4, 5, 6, 7, 8, 9};

  print(vec | std::views::take(3)); // OK
  print(vec | std::views::drop(3)); // OK

  print(lst | std::views::take(3)); // OK
  print(lst | std::views::drop(3)); // ERROR
  for (const auto& elem : lst | std::views::drop(3)) { // OK
    std::cout << elem << ' ';
  }
  std::cout << '\n';

  auto isEven = [] (const auto& val) {
    return val % 2 == 0;
  };
  print(vec | std::views::filter(isEven)); // ERROR
}
```

声明一个非常简单的泛型函数 print()，输出作为 const 引用传递的范围的所有元素:

```cpp
void print(const auto& rg)
{
  ...
}
```

然后，为几个视图调用 print()，并遇到了令人惊讶的行为:

- 将 vector 传递给 take 视图和 drop 视图的效果很好:

```cpp
print(vec | std::views::take(3)); // OK
print(vec | std::views::drop(3)); // OK
```

```
3
4  print(lst | std::views::take(3)); // OK
```

- 将 drop 视图传递给 list 无法编译:

```
1  print(vec | std::views::take(3)); // OK
2  print(vec | std::views::drop(3)); // OK
3
4  print(lst | std::views::take(3)); // OK
5  print(lst | std::views::drop(3)); // ERROR
```

- 然而，直接迭代 drop 视图却运行良好:

```
1  for (const auto& elem : lst | std::views::drop(3)) { // OK
2    std::cout << elem << ' ';
3  }
```

所以，vector 不能很好地工作。例如，若传递一个过滤器视图们会得到所有范围的错误:

```
1  print(vec | std::views::filter(isEven)); // ERROR
```

const& 并不适用于所有视图

这种非常奇怪和意外的行为的原因是，当视图为 const 时，视图并不总是支持遍历元素。这是这样一个事实的结果: 迭代这些视图的元素有时需要能够修改视图的状态 (例如，由于缓存)。对于某些视图 (如过滤器视图) 永远不会工作，对于某些视图 (例如 drop 视图)，只是有时会起作用。

若用 const 声明了以下标准视图中的元素，则不能迭代:

- 永远不能迭代的 const 视图:
  - 过滤视图
  - 丢弃段视图
  - 拆分视图
  - IStream 视图
- 只能偶尔迭代的 const 视图:
  - 丢弃视图，若引用的范围没有随机访问或没有 size()
  - 反向视图，若引用的范围对于开始迭代器和前哨 (结束迭代器) 具有不同的类型
  - 连接视图，若引用的范围生成值而不是引用，则使用该视图
  - 若引用的范围本身不是 const 可迭代的，则引用其他范围的所有视图

对于这些视图，begin() 和 end() 作为 const 成员函数只能有条件地提供，或者根本不提供。对于丢弃视图，只有当传递的范围满足随机访问范围和大小范围的要求时，才会提供 const 对象的 begin():[可能会将模板参数 V 限制为视图这一事实感到惊讶，但它可能是一个从传递的容器中隐式创建的视图引用，而不是视图]。

```
1  namespace std::ranges {
2    template<view V>
```

```
  class drop_view : public view_interface<drop_view<V>> {
    public:
    ...
    constexpr auto begin() const requires random_access_range<const V>
    && sized_range<const V>;
    ...
  };
}
```

这里，使用新特性来约束成员函数。

若将形参声明为 const 引用，则无法提供可以处理所有范围和视图元素的泛型函数。

```
void print(const auto& coll); // not callable for all views

template<typename T>
void foo(const T& coll); // not callable for all views
```

是否对范围参数的类型有约束并不重要:

```
void print(const std::ranges::input_range auto& coll); // not callable for all views

template<std::ranges::random_access_range T>
void foo(const T& coll); // not callable for all views
```

非 const&& 对所有视图都有效

为了在泛型代码中支持这些视图，应该将范围参数声明为通用引用 (也称为转发引用)。这些引用可以引用所有表达式，同时保留引用对象不是 const 的事实 [C++ 标准中，通用引用称为转发引用，但在 std::forward<>() 调用之前并不真正的转发，并且完美转发并不总是使用这些引用的原因 (如本例所示)]。例如:

```
void print(std::ranges::input_range auto&& coll); // can in principle pass all views

template<std::ranges::random_access_range T>
void foo(T&& coll);
```

因此，下面的程序可以正常工作:

ranges/printranges.cpp

```
#include <iostream>
#include <vector>
#include <list>
#include <ranges>

void print(auto&& rg)
```

```
{
  for (const auto& elem : rg) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
  std::list lst{1, 2, 3, 4, 5, 6, 7, 8, 9};

  print(vec | std::views::take(3)); // OK
  print(vec | std::views::drop(3)); // OK

  print(lst | std::views::take(3)); // OK
  print(lst | std::views::drop(3)); // OK
  for (const auto& elem : lst | std::views::drop(3)) { // OK
    std::cout << elem << ' ';
  }
  std::cout << '\n';

  auto isEven = [] (const auto& val) {
    return val % 2 == 0;
  };
  print(vec | std::views::filter(isEven)); // OK
}
```

出于同样的原因，前面介绍的通用 maxValue() 函数并不总是有效:

```
template<std::ranges::input_range Range>
std::ranges::range_value_t<Range> maxValue(const Range& rg)
{
  ... // ERROR when iterating over the elements of a filter view
}

// max value of a filtered range:
auto odd = [] (auto val) {
           return val % 2 != 0;
         };
std::cout << maxValue(arr | std::views::filter(odd)) << '\n'; // ERROR
```

如前所述，最好实现泛型 maxValue() 函数如下:

ranges/maxvalue2.hpp

```
#include <ranges>

template<std::ranges::input_range Range>
```

```
std::ranges::range_value_t<Range> maxValue(Range&& rg)
{
  if (std::ranges::empty(rg)) {
    return std::ranges::range_value_t<Range>{};
  }
  auto pos = std::ranges::begin(rg);
  auto max = *pos;
  while (++pos != std::ranges::end(rg)) {
    if (*pos > max) {
      max = *pos;
    }
  }
  return max;
}
```

这里，将参数 rg 声明为通用/转发引用:

```
template<std::ranges::input_range Range>
std::ranges::range_value_t<Range> maxValue(Range&& rg)
```

这样，就可以确保传递的视图不会变成 const，所以现在也可以传递一个过滤视图:

ranges/maxvalue2.cpp

```
#include "maxvalue2.hpp"
#include <iostream>
#include <algorithm>

int main()
{
  int arr[] = {0, 8, 15, 42, 7};
  // max value of a filtered range:
  auto odd = [] (auto val) { // predicate for odd values
        return val % 2 != 0;
      };
  std::cout << maxValue(arr | std::views::filter(odd)) << '\n'; // OK
}
```

程序的输出是:

```
15
```

现在想知道如何声明一个泛型函数，可以调用所有 const 和非 const 范围和视图，并保证元素不被修改? 好吧，再一次，这里有一个重要的教训:

C++20 起，不再有一种方法可以声明一个泛型函数来接受所有标准集合类型 (const 和非 const 容器和视图)，并保证不修改元素。

所能做的就是获取范围/视图，并确保不在函数体中修改元素，但这样做可能会很复杂:

- 将视图声明为 const 并不一定使元素成为 const。
- 视图没有 const_iterator 成员。
- 视图目前还不提供 cbegin() 和 cend() 成员。
- std::ranges::cbegin() 和 Std::cbegin() 对于视图是无效的。
- 将视图元素声明为 const 可能没有效果。

对并发迭代使用 const&

关于使用通用/转发引用的建议，有一个重要的限制: 当同时遍历视图时不应该使用。

考虑下面的例子:

```
std::list<int> lst{1, 2, 3, 4, 5, 6, 7, 8};

auto v = lst | std::views::drop(2);

// while another thread prints the elements of the view:
std::jthread printThread{[&] {
    for (const auto& elem : v) {
      std::cout << elem << '\n';
    }
}};

// this thread computes the sum of the element values:
auto sum = std::accumulate(v.begin(), v.end(), // fatal runtime ERROR
0L);
```

使用 std::jthread，启动了一个线程，该线程遍历视图 v 的元素并进行输出。同时，遍历 v 来计算元素值的和。对于标准容器，只进行读操作的并行迭代是安全的 (对于容器来说，调用 begin() 可以保证作为读访问计数)，但此保证不适用于标准视图。因为可能会并发地为调用视图 v 的 begin()，所以这段代码会导致未定义行为 (可能的数据竞争)。

执行只读并发迭代的函数应该使用 const 视图，将可能的运行时错误转换为编译时错误:

```
const auto v = lst | std::views::drop(2);

std::jthread printThread{[&] {
    for (const auto& elem : v) { // compile-time ERROR
      std::cout << elem << '\n';
    }
}};

auto sum = std::accumulate(v.begin(), v.end(), // compile-time ERROR
0L);
```

视图的重载

读者们可能会声称可以简单地重载容器和视图的泛型函数。对于视图，可能只需要添加一个重载来约束视图的函数，并按值接受参数就足够了:

```
void print(const auto& rg); // for containers
void print(std::ranges::view auto rg) // for views
```

但当一次按值传递，一次按引用传递时，重载解析规则可能会变得棘手，可能会导致歧义:

```
std::vector vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
print(vec | std::views::take(3)); // ERROR: ambiguous
```

对于包含用于一般情况的概念的视图，使用概念也没有帮助:

```
void print(const std::ranges::range auto& rg); // for containers
void print(std::ranges::view auto rg) // for views

std::vector vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
print(vec | std::views::take(3)); // ERROR: ambiguous
```

相反，必须声明 const& 重载不适用于视图:

```
template<std::ranges::input_range T> // for containers
requires (!std::ranges::view<T>) // and not for views
void print(const T& rg);

void print(std::ranges::view auto rg) // for views

std::vector vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
print(vec | std::views::take(3)); // OK
```

复制视图可能会创建一个与源视图具有不同状态和行为的视图，所以按值传递所有视图都会有问题。

6.5.2 具有写访问权限的视图

容器具有很深的稳定性，具有值语义并拥有自己的元素，所以可将任何常量性传播给其元素。当容器为 const 类型时，其元素也是 const 类型，所以以下代码无法编译:

```
template<typename T>
void modifyConstRange(const T& range)
{
  range.front() += 1; // modify an element of a const range
}

std::array<int, 10> coll{}; // array of 10 ints with value 0
...
modifyConstRange(coll); // compile-time ERROR
```

这个编译时错误很有帮助，因为有助于检测在运行时可能破坏的代码。例如，若不小心使用赋值操作符而不是比较操作符，代码将无法编译:

```
template<typename T>
void modifyConstRange(const T& range)
{
  if (range[0] = 0) { // OOPS: should be ==
    ...
  }
}

std::array<int, 10> coll{}; // array of 10 ints with value 0
...
modifyConstRange(coll); // compile-time ERROR (thank goodness)
```

知道元素不能修改也有助于优化 (避免备份和检查更改)，或者确保在使用多个线程时，元素访问不会因数据竞争而导致未定义行为。

有了视图来说，情况就复杂多了。作为左值传递的范围的视图具有引用语义，引用存储在其他地方的元素。按照设计，这些视图不会将常量性传播到其元素上。

```
std::array<int, 10> coll{}; // array of 10 ints with value 0
...
std::ranges::take_view v{coll, 5}; // view to first five elements of coll
modifyConstRange(v); // OK, modifies the first element of coll
```

这适用于几乎所有关于左值的视图，不管是如何创建的:

```
modifyConstRange(coll | std::views::drop(5)); // OK, elements of coll are not const
modifyConstRange(coll | std::views::take(5)); // OK, elements of coll are not const
```

因此，可以使用范围适配器 std::views::all() 将容器传递给接受范围作为 const 引用的泛型函数:

```
modifyConstRange(coll); // compile-time ERROR: elements are const
modifyConstRange(std::views::all(coll)); // OK, elements of coll are not const
```

对右值的视图 (如果它们是有效的) 通常仍然传播常量性:

```
readConstRange(getColl() | std::views::drop(5)); // compile-time ERROR (good)
readConstRange(getColl() | std::views::take(5)); // compile-time ERROR (good)
```

另外，不能使用视图修改 const 容器的元素:

```
const std::array<int, 10> coll{}; // array of 10 ints with value 0
...
auto v = std::views::take(coll, 5); // view to first five elements of coll
modifyConstRange(v); // compile-time ERROR (good)
```

因此，通过在视图引用容器之前将容器设置为 const，可以确保容器的元素不会接受该视图的函数修改:

```
std::array<int, 10> coll{}; // array of 10 ints with value 0
...
std::ranges::take_view v{std::as_const(coll), 5}; // view to five elems of const coll
modifyConstRange(v); // compile-time ERROR
modifyConstRange(std::as_const(coll) | std::views::take(5)); // compile-time ERROR
```

函数 std::as_const() 从 C++17 开始在头文件 <utility> 中提供，但在创建视图后调用 std::as_const() 没有效果 (除了少数视图，不再能够遍历元素)。

C++23 将引入一个带有范围适配器 std::views::as_const() 的辅助视图 std::ranges::as_const_view，就可以简单地将视图的元素设置为 const，如下所示 (参见http://wg21.link/p2278r4):

```
std::views::as_const(v); // make elements of view v const
```

不能忘记这里的命名空间视图:

```
std::as_const(v); // OOPS: no effect on views
```

命名空间 std 和命名空间 std::ranges 中的函数 as_const() 都使某些东西成为 const，但前者使对象成为 const，而后者使元素成为 const[C++ 标准委员会的成员有时决定支持看起来相同的特性，即使细节不同，也可能成为麻烦的来源。] 读者们可能想知道，为什么使用引用语义的视图没有被设计成传播常量的方式。一个论点是在内部使用指针，因此应该表现得像指针。

然而，奇怪的是，右值的视图不是这样工作的。另一种说法是，const 几乎没什么作用，其可以很容易地通过将视图复制到非 const 视图来删除:

```
void foo(const auto& rg) // if rg would propagate constness for views on lvalues
{
  auto rg2 = rg; // rg2 would no longer propagate constness
  ...
}
```

初始化的行为类似于 const_cast<>。但若这是容器的泛型代码，开发者已经习惯不复制传递的集合，这样做的代价很高。因此，现在有了另一个不复制传递的范围的好理由，并将范围 (容器和视图) 声明为 const 的效果将更加一致。然而，现在设计决策已经完成，作为开发者必须处理。

6.5.3 更改范围的视图

正如所了解的那样，对作为视图的范围使用 const 是一个问题，原因有两个:

- const 可以禁用元素的迭代
- const 可能不会传播到视图的元素

显而易见的问题是如何确保代码在使用视图时不能修改范围的元素。

使元素为 const

确保视图中的元素不能修改的唯一方法是，在元素访问时强制使用 const:

- 基于范围的 for 循环中使用 const:

```
for (const auto& elem : rg) {
  ...
}
```

- 或通过传递转换为 const 的元素:

```
for (auto pos = rg.begin(); pos != rg.end(); ++pos) {
  elemfunc(std::as_const(*pos));
}
```

然而，这里有一个坏消息: 视图库的设计者计划禁用某些视图声明元素 const 的效果，比如即将到来的 zip 视图:

```
for (const auto& elem : myZipView) {
  elem.member = value; // WTF: C++23 plans to let this compile
}
```

也许还能阻止这一切，但使用 C++ 标准视图时，将集合或其元素设置为 const 可能不起作用。

const_iterator 和 cbegin() 未提供或无效

不幸的是，通常将容器的所有元素都设置为 const 的方法不适用于视图:

- 通常，视图不提供 const_iterator 来执行以下操作:

```
for (decltype(rg)::const_iterator pos = rg.begin(); // not for views
pos != range.end();
++pos) {
  elemfunc(*pos);
}
```

因此，也不能简单地将整个范围转换为:

```
std::ranges::subrange<decltype(rg)::const_iterator> crg{rg}; // not for views
```

- 通常，视图没有成员函数 cbegin() 和 cend() 来执行以下操作:

```
callTraditionalAlgo(rg.cbegin(), rg.cend()); // not for views
```

有的读者可能会建议使用独立的辅助函数 std::cbegin() 和 std::cend()，从 C++11 开始就提供了。引入它们是为了确保元素在迭代时是 const 的，因为 std::cbegin() 和 std::cend() 对于视图来说是无效的，所以里的情况会更糟。其规范没有考虑到具有浅常量的类型 (在内部，调用 const 成员函数

begin()，并且不向值类型添加 constness)。对于不传播常量的视图，这些函数被破坏了 [这时应该修复 std::cbegin() 和 std::cend()，但目前为止，C++ 标准委员会已经拒绝了所有修正它们的提案。]

```
for (auto pos = std::cbegin(range); pos != std::cend(range); ++pos) {
  elemfunc(*pos); // does not provide constness for values of views
}
```

还要注意，在使用这些辅助函数时，ADL 存在一些问题。C++20 在命名空间 std::ranges 中引入了 range 库的相应帮助函数，但 C++20 中的视图中破坏了 std::ranges::cbegin() 和 std::ranges::cend()(这将在 C++23 中修复):

```
for (auto pos = std::ranges::cbegin(rg); pos != std::ranges::cend(rg); ++pos) {
  elemfunc(*pos); // OOPS: Does not provide constness for the values in C++20
}
```

因此，C++20 中，当输入使视图的所有元素都为 const 时，还会遇到另一个严重的问题:

在泛型代码中使用 cbegin()，cend()，cdata() 时要谨慎，这些函数在某些视图中不可用或已破坏。

不管这是我们有这个问题，还是 C++ 标准委员会的，及其范围库的子小组不愿意在 C++20 中修复这个问题。可以在 std::ranges::view_interface<> 中提供 const_iterator 和 cbegin() 成员，其作为所有视图的基类。有趣的是，到目前为止只有视图提供了 const_iterator 支持:std::string_view。具有讽刺意味的是，这或多或少是唯一不需要的视图，因为在字符串视图中，字符总是 const。

然而，还是有一些希望的:http://wg21.link/p2278提供了一种方法来修复 C++23 中这个损坏的常量 (而 std::cbegin() 和 std::cend() 不适用) 使用这种方法，在泛型函数内部，可以将元素设为 const，如下所示:

```
void print(auto&& rgPassed)
{
  auto rg = std::views::as_const(rgPassed); // ensure all elements are const
  ... // use rg instead of rgPassed now
}
```

另一种方法，可以执行以下操作:

```
void print(R&& rg)
{
  if constexpr (std::ranges::const_range<R>) {
    // the passed range and its elements are constant
    ...
  }
  else {
    // call this function again after making view and elements const:
    print(std::views::as_const(std::forward<R>(rg)));
  }
}
```

不幸的是，C++23 仍然没有提供一种通用的方法来声明容器和视图的引用形参，以保证内部的元素是 const。不能声明在其签名中保证不修改元素的函数 print()。

6.6. 视图分解容器的习惯用法

正如本章所述，使用容器时可以依赖的一些习惯用法会被视图打破了。这里有一个总结，当在容器和视图中使用泛型代码时，应该考虑到:

- 当标准视图为 const 时，可能无法遍历其元素。
  因此，所有类型范围 (容器和视图) 的泛型代码必须将参数声明为通用/转发引用。
  但当有并发迭代时，不要使用通用/转发引用，则 const 很友好。
- 左值范围的标准视图不传播常量。
  将这样的视图声明为 const 并没有将元素声明为 const。
- 标准视图上的并发迭代可能导致数据争用 (由于未定义行为导致的运行时错误)，即使它们只读取。
- 读取迭代可能会影响以后的功能行为，甚至使以后的迭代无效。
  临时使用标准视图。
- 复制视图可能会创建一个与源视图具有不同状态和行为的视图。
  避免复制标准视图。
- cbegin() 和 cend() 不能使元素为 const。
  C++23 将通过提供 cbegin() 和 cend() 成员函数，以及修复 std::ranges::cbegin() 和 std::ranges::cend() 来部分修复这个问题。不幸的是，仍然会破坏 std::cbegin() 和 std::cend()。
- 类型 const_iterator 通常不可用。
- 对于 C++23，可能会有以下破坏: 对于某些标准视图，将元素声明为 const 可能没有影响，可能能够修改视图的 const 元素的成员。

所以标准视图并不总是一个限制或处理范围元素的纯子集，其可能提供在使用整个范围时不允许的选项和操作。

因此，要特别小心地使用视图，不要考虑任何临时的常量。最好避免使用标准视图，而使用具有更安全设计的视图。

6.7. 附注

第一个 C++ 标准采用 Standard Template Llibrary(该标准引入了一对迭代器作为容器/集合的抽象，以便在算法中处理) 之后，一直有关于如何处理单个范围对象，而不是传递 begin 迭代器和 end 迭代器的讨论。Boost 的 Range 库和 Adobe 源库 (ASL) 是最早提出具体库的两种方法。

2005 年，Thorsten Ottosen 在http://wg21.link/n1871上提出了将“范围”作为 C++ 标准的第一个建议。Eric Niebler 花了数年时间 (得到了许多人的支持) 推动这一进程，并于 2014 年由 Eric Niebler、Sean Parent 和 Andrew Sutton 在http://wg21.link/n4128上提出了另一项提案 (该文档包含了许多关键设计决策的基本原理)。因此，2015 年 10 月，范围技术规范以http://wg21.link/n4560起点。

范围库最终通过，并将范围的 TS 合并到由 Eric Niebler、Casey Carter 和 Christopher Di Bella 在http://wg21.link/p0896r4中提出的 C++ 标准中。

采用之后，针对 C++20 的一些建议、论文，甚至是缺陷改变了重要的方面，特别是视图方面。例如，由 Barry Revzin, Tim Song 和 Nicolai Josuttis 提出的http://wg21.link/p2210r2(修复分割视图)，http://wg21.link/p2325r3(修复视图的定义)，http://wg21.link/p2415r2(引入右值范围的拥有视图)，和http://wg21.link/p2432r1(修复 istream 视图)。

此外，视图的几个 const 问题中的一些可能会在 C++23 中通过http://wg21.link/p2278r4修复。希望厂商能在 C++23 之前提供这个修复。否则，C++20 的代码可能与 C++23 不兼容。

第 7 章 范围和视图的工具

上一章介绍了范围和视图之后，本章首先概述了 C++20 中引入的用于处理范围和视图的重要细节和工具 (函数，类型和对象)。本章特别包括:

- 用于创建视图的最重要的通用工具
- 一种对迭代器进行分类的新方法及结果
- 新迭代器和哨兵类型
- 用于处理范围和迭代器的新函数和类型函数

最后，本章还列出了 C++20 现在提供的所有算法，详细说明了这些算法是否可以作为一个整体用于范围，以及适用于它们的几个关键方面。

请注意，本书中还有其他地方详细描述了范围的特征:

- 关于概念的总论章介绍了所有的范围概念。
- 下一章将介绍标准视图类型的详情。

7.1. 范围作为视图的实用工具

正如视图及其使用方式所介绍的那样，C++ 提供了几个范围适配器和范围工厂，以便可以简单地创建具有最佳性能的视图。

其中几个适配器适用于特定的视图类型。然而，根据传递范围的特征，其中一些可能会创建不同的东西。若适配器已经具有结果的特征，可能只生成可传递的范围。

有一些关键的范围适配器和工厂，可以很容易地创建视图，或将范围转换为具有特定特征 (独立于内容) 的视图:

- std::views::all() 是将传递的范围转换为视图的主要范围适配器。
- std::views::counted() 可将传入的 begin 和 count/size 转换为视图的主要范围工厂。
- std::views::common() 是一个范围适配器，将 (开始) 迭代器和哨兵 (结束迭代器) 具有不同类型的范围转换为具有统一的开始和结束类型的视图。

本节将介绍这些适配器。

7.1.1 std::views::all()

范围适配器 std::views::all() 是将任何还不是视图的范围转换为视图的适配器，可由一个低成本的句柄来处理范围中的元素。

all(rg) 的生成以下结果 [C++20 最初声明 all() 可以产生子范围，但不能产生所属视图]。但在 C++20 发布后，这个问题已经修复了 (参见http://wg21.link/p2415):

- 若 rg 已经是视图，则为 rg 的副本
- 否则，rg 的 std::ranges::ref_view 是一个左值 (有名称的范围对象)
- 否则，若 rg 的 std::ranges::owned_view 是右值 (未命名的临时范围对象或用 std::move() 标记的范围对象)

例如:

```
std::vector<int> getColl(); // function returning a tmp. container
std::vector coll{1, 2, 3}; // a container
std::ranges::iota_view aView{1}; // a view

auto v1 = std::views::all(aView); // decltype(coll)
auto v2 = std::views::all(coll); // ref_view<decltype(coll)>
auto v3 = std::views::all(std::views::all(coll)); // ref_view<decltype(coll)>
auto v4 = std::views::all(getColl()); // owning_view<decltype(coll)>
auto v5 = std::views::all(std::move(coll)); // owning_view<decltype(coll)>
```

all() 适配器通常用于将范围作为轻量级对象传递。将范围转换成视图有两个原因:

- 性能:
  当复制范围时 (形参按值接受实参)，使用视图要廉价得多。这样做的原因是移动或 (若支持的话) 复制一个视图保证是廉价的，所以将范围作为视图传递会使调用变得低成本。
  例如:

```
void foo(std::ranges::input_range auto coll) // NOTE: takes range by value
{
  for (const auto& elem : coll) {
    ...
  }
}

std::vector<std::string> coll{ ... };

foo(coll); // OOPS: copies coll
foo(std::views::all(coll)); // OK: passes coll by reference
```

  这里使用 all() 有点像使用引用包装器 (用 std::ref() 或 std::cref() 创建)。
  但 all() 的好处是传递的内容，仍然支持通常的范围接口。
  将容器传递给协程 (协程通常必须按值接受参数) 可能是另一种应用。
- 视图的支持性要求:
  使用 all() 的另一个原因是为了满足需要视图的约束，这个要求的一个原因可能是为了确保传递参数的代价不高 (这就带来了上面提到的好处)。
  例如:

```
void foo(std::ranges::view auto coll) // NOTE: takes view by value
{
  for (const auto& elem : coll) {
    ...
  }
}

std::vector<std::string> coll{ ... };

```

```
foo(coll); // ERROR
foo(std::views::all(coll)); // OK (passes coll by reference)
```

使用 all() 可能隐式地发生。

使用 all() 进行隐式转换的示例是使用容器初始化视图类型。

std::views::all_t<> 类型

C++20 还定义了类型 std::views::all_t<> 作为 all() 产生的类型，其遵循完全转发规则，若依值类型和右值引用类型都可以用来指定右值的类型:

```
std::vector<int> v{0, 8, 15, 47, 11, -1, 13};
...
std::views::all_t<decltype(v)> a1{v}; // ERROR
std::views::all_t<decltype(v)&> a2{v}; // ref_view<vector<int>>
std::views::all_t<decltype(v)&&> a3{v}; // ERROR
std::views::all_t<decltype(v)> a4{std::move(v)}; // owning_view<vector<int>>
std::views::all_t<decltype(v)&> a5{std::move(v)}; // ERROR
std::views::all_t<decltype(v)&&> a6{std::move(v)}; // owning_view<vector<int>>
```

接受范围的视图通常使用类型 std::views::all_t<> 来确保传递的范围确实是一个视图。因此，若传递的范围还不是视图，则会隐式创建视图。例如:

```
std::views::take(coll, 3)
```

具有相同的效果:

```
std::ranges::take_view{std::ranges::ref_view{coll}, 3};
```

其工作方式如下 (以视图为例):

- 视图类型要求传递视图:

```
namespace std::ranges {
  template<view V>
  class take_view : public view_interface<take_view<V>> {
    public:
    constexpr take_view(V base, range_difference_t<V> count);
    ...
  };
}
```

- 推导指南要求视图的元素类型为 std::views::all_t:

```
namespace std::ranges {
  // deduction guide to force the conversion to a view:
  template<range R>
  take_view(R&&, range_difference_t<R>) -> take_view<views::all_t<R>>;
}
```

- 当传递容器时，视图的范围类型必须具有 std::views::all_t<> 类型，所以容器会隐式地转换为 ref_view(若是左值) 或 owned _view(若是右值)。其效果就好像下面的语句:

```
std::ranges::ref_view rv{coll}; // convert to a ref_view<>
std::ranges::take_view tv(rv, 3); // and use view and count to initialize the
↪  take_view<>
```

概念 viewable_range 可用于检查类型是否可用于 all_t<>(因此对应的对象可以传递给 all())[对于仅移动视图类型的左值，尽管 viewable_range 是满足的，但 all() 仍存在格式错误的问题]:

```
std::ranges::viewable_range<std::vector<int>> // true
std::ranges::viewable_range<std::vector<int>&> // true
std::ranges::viewable_range<std::vector<int>&&> // true
std::ranges::viewable_range<std::ranges::iota_view<int>> // true
std::ranges::viewable_range<std::queue<int>> // false
```

7.1.2 std::views::counted()

范围工厂 std::views::counted() 提供了从 begin 迭代器和计数创建视图的最灵活的方式。

```
std::views::counted(beg, sz)
```

创建一个以 beg 开头的范围的前 sz 个元素的视图。

使用视图时，由开发者来确保 begin 和 count 是有效的。否则，程序具的行为未定义。

计数可能为 0，所以该范围为空。

计数存储在视图中，因此是稳定的。即使在视图引用的范围中插入了新元素，计数也不会改变。例如:

```
std::list lst{1, 2, 3, 4, 5, 6, 7, 8};

auto c = std::views::counted(lst.begin(), 5);
print(c); // 1 2 3 4 5

lst.insert(++lst.begin(), 0); // insert new second element in lst
print(c); // 1 0 2 3 4
```

若想查看一个范围的前 num 个元素，获取视图提供了一种更方便、更安全的方式来获取:

```
std::list lst{1, 2, 3, 4, 5, 6, 7, 8};
auto v1 = std::views::take(lst, 3); // view to first three elements (if they exist)
```

视图检查是否有足够的元素，若没有，则产生更少的元素。

通过使用丢弃视图，甚至可以跳过特定数量的前置元素:

```
std::list lst{1, 2, 3, 4, 5, 6, 7, 8};
auto v2 = std::views::drop(lst, 2) | std::views::take(3); // 3rd to 5th element (if
↪  exist)
```

counted() 的类型

counted(beg, sz) 根据所调用的范围特征，会产生不同类型的结果:

- 若传递的开始迭代器是一个 contiguous_iterator(指存储在连续内存中的元素)，则会产生一个 std::span。这适用于 std::vector<> 或 std::array<> 的原始指针、原始数组和迭代器。
- 否则，若传递的 begin 迭代器是一个 random_access_iterator(支持在元素之间来回跳转)，就会产生 std::ranges::subrange。这适用于 std::deque<> 的迭代器。
- 否则，将产生 std::ranges::subrange，以 std::counts_iterator 作为开始，并以 std::default_sentinel_t 类型的伪哨兵作为结束，所以子范围中的迭代器在迭代时计数。这适用于列表、关联容器和无序容器 (哈希表) 的迭代器。

例如:

```
std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
auto vec5 = std::ranges::find(vec, 5);
auto v1 = std::views::counted(vec5, 3); // view to element 5 and two more elems in
↪  vec
  // v1 is std::span<int>
std::deque<int> deq{1, 2, 3, 4, 5, 6, 7, 8, 9};
auto deq5 = std::ranges::find(deq, 5);
auto v2 = std::views::counted(deq5, 3); // view to element 5 and two more elems in
↪  deq
  // v2 is std::ranges::subrange<std::deque<int>::iterator>
std::list<int> lst{1, 2, 3, 4, 5, 6, 7, 8, 9};
auto lst5 = std::ranges::find(lst, 5);
auto v3 = std::views::counted(lst5, 3); // view to element 5 and two more elems in
↪  lst
  // v3 is std::ranges::subrange<std::counted_iterator<std::list<int>::iterator>,
  // std::default_sentinel_t>
```

注意，这段代码有风险，因为若集合中没有 5，后面有两个元素，会在创建时出现未定义行为。因此，若不确定情况是否如此，则应该检查此选项。

7.1.3 std::views::common()

范围适配器 std::views::common() 为传递的范围生成一个具有统一类型的视图，用于开始迭代器和哨兵 (结束迭代器)。其行为类似于范围适配器 std::views::all()，若其迭代器具有不同的类型，则会从传递的参数创建 std::ranges::common_view。

假设想调用传统算法 algo()，该算法要求开始迭代器和结束迭代器的类型相同。然后，可以为不同类型的迭代器进行调用，如下所示:

```
template<typename BegT, typename EndT>
void callAlgo(BegT beg, EndT end)
{
```

```
  auto v = std::views::common(std::ranges::subrange(beg,end));
  algo(v.begin(), v.end()); // assume algo() requires iterators with the same type
}
```

common(rg) 会生成的类型为

- 若 rg 已经是一个具有相同开始和结束迭代器类型的视图，则为 rg 的副本
- 否则，若 rg 是一个具有相同开始迭代器和结束迭代器类型的范围对象，则调用 rg 的 std::ranges::ref_view
- 否则，rg 为 std::ranges::common_view 类型

例如:

```
std::list<int> lst;
std::ranges::iota_view iv{1, 10};
...
auto v1 = std::views::common(lst); // std::ranges::ref_view<decltype(lst)>
auto v2 = std::views::common(iv); // decltype(iv)
auto v3 = std::views::common(std::views::all(vec));
      // std::ranges::ref_view<decltype(lst)>

std::list<int> lst {1, 2, 3, 4, 5, 6, 7, 8, 9};
auto vt = std::views::take(lst, 5); // begin() and end() have different types
auto v4 = std::views::common(vt); // std::ranges::common_view<decltype(vt)>
```

注意，std::ranges::common_view 的构造函数和辅助类型 std::common:iterator 都要求传递的迭代器具有不同的类型，所以若不知道迭代器类型是否不同，则应该使用此适配器。

7.2. 新迭代器类别

迭代器有不同的功能，这些功能很重要，因为有些算法需要特殊的迭代器。例如，排序算法需要能够执行随机访问的迭代器，否则性能就会很差。由于这个原因，迭代器有不同的类别，C++20 中扩展了一个新的类别: 连续迭代器。这些类别的功能列在“迭代器类别”表中。

迭代器类别	功能	供应者
Output	向前写入	osteram 迭代器, 插入器
Input	向前读一次	istream 迭代器
Forward	向前读取	std::forward_list<>, 无序容器
Bidirectional	向前和向后读取	list<>, set<>, multset<>, map<>, multimap<>
Random-access	随机读取	deque<>
Contiguous	读取存储在连续内存中的元素	array<>, vector<>, string, C 风格数组

表 7.1 迭代器类别

注意，在 C++20 中，类别的细节发生了变化。最重要的变化是:

- 新的迭代器类别连续。对于这个类别，定义了个新的迭代器标记类型:std::contiguous_iterator_tag。
- 产生临时对象 (右值) 的迭代器可以拥有比输入迭代器更强的类别，生成引用的要求不再适用于前向迭代器。
- 输入迭代器不再保证是可复制的，应该只可移动。
- 输入迭代器的后置自增操作符不再需要产生任何结果。对于输入迭代器 pos，不应再使用 pos++，可使用 ++pos 代替。

这些更改向后不兼容:

- 对于 std::vector 或数组等容器的迭代器，不再检查迭代器是否为随机访问迭代器，开发者必须检查迭代器是否支持随机访问。
- 前向迭代器、双向迭代器或随机访问迭代器的值是引用的假设不再适用。

出于这个原因，C++20 引入了一个新的可选迭代器属性 iterator_concept，并定义可以设置该属性来表示一个不同于传统类别的 C++20 类别。例如:

```
std::vector vec{1, 2, 3, 4};
auto pos1 = vec.begin();
decltype(pos1)::iterator_category // type std::random_access_iterator_tag
decltype(pos1)::iterator_concept // type std::contiguous_iterator_tag

auto v = std::views::iota(1);
auto pos2 = v.begin();
decltype(pos2)::iterator_category // type std::input_iterator_tag
decltype(pos2)::iterator_concept // type std::random_access_iterator_tag
```

注意，std::iterator_traits 没有提供成员 iterator_concept，对于迭代器，成员 iterator_category 可能并不总有定义。

迭代器概念和范围概念考虑了新的 C++20 类别。对于必须处理迭代器类别的代码，C++20 起会进行如下处理:

- 使用迭代器概念和范围概念来检查类别，而不是 std::iterator_traits<I>::iterator_category。
- 若实现了自己的迭代器类型，可参考http://wg21.link/p2259中的提供的 iterator_category 和/或 iterator_concept。

有效的 C++20 输入迭代器可能根本不是 C++17 迭代器 (例如，不提供复制)。对于这些迭代器，传统的迭代器特性不起作用。由于这个原因，从 C++20 开始:

- 使用 std::iter_value_t
  代替 iterator_traits<>::value_type.
- 使用 std::iter_reference_t
  代替 iterator_traits<>::reference.
- 使用 std::iter_difference_t
  代替 iterator_traits<>::difference_type.

7.3. 新增迭代器和哨兵类型

为了 (更好地) 支持范围和视图，C++20 引入了几个新的迭代器和哨兵类型:

- std::counted_iterator 用于迭代器，该迭代器本身有一个计数来指定范围的结束
- std::common_iterator 用于公共迭代器类型，可用于不同类型的两个迭代器
- std::default_sentinel_t 用于结束迭代器，强制迭代器检查其结束
- std::unreachable_sentinel_t 表示永远无法到达的 end 迭代器，表示无限范围
- std::move_sentinel 用于将副本映射到 move 的 end 迭代器

7.3.1 std::counted_iterator

计数迭代器是一种迭代器，其有一个计数器，表示要迭代的最大元素数。
有两种方法可以使用这样的迭代器:

- 遍历并检查还剩下多少个元素:

```
for (std::counted_iterator pos{coll.begin(), 5}; pos.count() > 0; ++pos) {
  std::cout << *pos << '\n';
}
std::cout << '\n';
```

- 迭代并与默认的哨兵进行比较:

```
for (std::counted_iterator pos{coll.begin(), 5};
pos != std::default_sentinel; ++pos) {
  std::cout << *pos << '\n';
}
```

当视图适配器 std::ranges::counted() 为非随机访问迭代器生成子范围时，就会使用此功能。

“类 std::counted_iterator<> 的操作”表列出了计数迭代器的 API。

操作	效果
countedItor pos{}	创建一个不引用任何元素的计数迭代器 (count 为 0)
countedItor pos{pos2, num}	创建以 pos2 开头的 num 个元素的计数迭代器
countedItor pos{pos2}	创建一个有计数迭代器，作为有计数迭代器 pos2 的 (类型转换) 副本
pos.count()	返回剩余元素的数量 (0 表示已到结束)
pos.base()	生成底层迭代器的 (副本)
...	底层迭代器类型的所有标准迭代器操作
pos == std::default_sentinel	返回迭代器是否在末尾
pos != std::default_sentinel	返回迭代器是否在末尾

表 7.2 类 std::counted_iterator<> 的操作

开发者则负责确保:

- 初始计数不高于初始传递的迭代器计数

- 计数迭代器的增量不会超过 count 次
- 计数迭代器不访问第 n 个元素以外的元素 (通常，最后一个元素后面的位置是有效的)

7.3.2 std::common_iterator

类型 std::common_iterator<> 用于协调两个迭代器的类型，其包装两个迭代器，以便从外部看，两个迭代器具有相同的类型。在内部，存储两种类型之一的值 (迭代器通常使用 std::variant<>)。

接受开始迭代器和结束迭代器的传统算法要求这些迭代器具有相同的类型，若有不同类型的迭代器，可以使用这个类型函数来调用这些算法:

```
algo(beg, end); // if this is an error due to different types

algo(std::common_iterator<decltype(beg), decltype(end)>{beg}, // OK
std::common_iterator<decltype(beg), decltype(end)>{end});
```

若传递给 common_iterator<> 的类型相同，则会导致编译时错误，所以在泛型代码中:

```
template<typename BegT, typename EndT>
void callAlgo(BegT beg, EndT end)
{
  if constexpr(std::same_as<BegT, EndT>) {
    algo(beg, end);
  }
  else {
    algo(std::common_iterator<decltype(beg), decltype(end)>{beg},
    std::common_iterator<decltype(beg), decltype(end)>{end});
  }
}
```

要达到同样的效果，更方便的方法是使用 common() 范围适配器:

```
template<typename BegT, typename EndT>
void callAlgo(BegT beg, EndT end)
{
  auto v = std::views::common(std::ranges::subrange(beg,end));
  algo(v.begin(), v.end());
}
```

7.3.3 std::default_sentinel

默认哨兵是一个不提供任何操作的迭代器。C++20 提供了一个相应的对象 std::default_sentinel，其类型为 std::default_sentinel_t，其在 <iterator> 中提供。该类型没有成员:

```
namespace std {
  class default_sentinel_t {
```

```
  };
  inline constexpr default_sentinel_t default_sentinel{};
}
```

提供类型和值作为伪哨兵 (结束迭代器)，迭代器无需查看结束迭代器就知道其结束点。对于这些迭代器，定义了与 std::default_sentinel 的比较，但从未使用默认的哨兵。相反，迭代器在内部检查它是否在结束处 (或距离结束处有多远)。

例如，计数迭代器为自己定义了一个 == 操作符，并带有一个默认的哨兵来检查其是否位于末尾:

```
namespace std {
  template<std::input_or_output_iterator I>
  class counted_iterator {
    ...
    friend constexpr bool operator==(const counted_iterator& p,
    std::default_sentinel_t) {
      ... // returns whether p is at the end
    }
  };
}
```

代码可以这样写:

```
// iterate over the first five elements:
for (std::counted_iterator pos{coll.begin(), 5};
pos != std::default_sentinel;
++pos) {
  std::cout << *pos << '\n';
}
```

该标准用默认的哨兵定义了以下操作:

- std::counted_iterator:
  - 使用操作符 == 和!= 进行比较
  - 用运算符-计算距离
- std::views::counted() 可以创建一个计数迭代器的子范围和一个默认的哨兵
- std::istream_iterator:
  - 默认哨兵可以用作初始值，其效果与默认构造函数相同
  - 使用 == 和!= 操作符进行比较
- std::istreambuf_iterator:
  - 默认哨兵可以用作初始值，其效果与默认构造函数相同
  - 使用 == 和!= 操作符进行比较
- std::ranges::basic_istream_view<>:
  - Istream 视图生成 std::default_sentinel 作为 end()

  - Istream 视图迭代器可以使用 == 和!= 操作符进行比较
- std::ranges::take_view<>:
  - 获取视图可能产生 std::default_sentinel 作为 end()
- std::ranges::split_view<>:
  - 拆分视图可能产生 std::default_sentinel 作为 end()
  - 拆分视图迭代器可以使用 == 和!= 操作符进行比较

7.3.4 std::unreachable_sentinel

std::unreachable_sentinel_t 类型的值 std::unreachable_sentinel 是在 C++20 中引入的，用于指定不可达的哨兵 (范围的结束迭代器)，其实际上是在说: “不要和我比较。” 此类型和值可用于指定无限范围。

使用时，它可以优化生成的代码，编译器可以检测到它与另一个迭代器进行比较时永远不会返回 true，所以可能会跳过对末尾的检查。

例如，若知道 42 存在于一个集合中，就可以这样进行搜索:

```
auto pos42 = std::ranges::find(coll.begin(), std::unreachable_sentinel,
          42);
```

该算法将同时比较 42 和 coll.end()(或作为结束/前哨传递的内容)。因为使用了 unreachable_sentinel(与迭代器的任何比较总是产生 false)，编译器可以优化代码，只与 42 进行比较，所以开发者必须确保 42 是存在的。

iota 视图提供了一个对无限范围使用 unreachable_sentinel 的示例。

7.3.5 std::move_sentinel

C++11 起，C++ 标准库提供了一个迭代器类型 std::move_iterator，可用于将迭代器的行为从复制映射到移动值，C++20 引入了相应的哨兵类型。

此类型只能用于使用操作符 == 和!= 比较移动哨兵和移动迭代器，并计算移动迭代器和移动哨兵之间的差异 (若支持)。

若有一个形成有效范围的迭代器和一个哨兵 (满足 std::sentinel_for 概念)，可以将它们转换为一个移动迭代器和一个移动哨兵，以获得仍然满足 sentinel_for 的有效范围。

可以像下面这样移动哨兵:

```
std::list<std::string> coll{"tic", "tac", "toe"};
std::vector<std::string> v1;
std::vector<std::string> v2;

// copy strings into v1:
for (auto pos{coll.begin()}; pos != coll.end(); ++pos) {
  v1.push_back(*pos);
}

```

```
// move strings into v2:
for (std::move_iterator pos{coll.begin()};
pos != std::move_sentinel(coll.end()); ++pos) {
  v2.push_back(*pos);
}
```

这样做的效果是迭代器和哨兵具有不同的类型，所以不能直接初始化 vector，因为要求 begin 和 end 类型相同，所以还必须为 end 使用移动迭代器:

```
std::vector<std::string> v3{std::move_iterator{coll.begin()},
               std::move_sentinel{coll.end()}}; // ERROR
std::vector<std::string> v4{std::move_iterator{coll.begin()},
               std::move_iterator{coll.end()}}; // OK
```

7.4. 处理范围的新函数

范围库在命名空间 std::ranges 和 std 中提供了几个通用的辅助函数。

其中一些在 C++20 之前就已经存在了，在命名空间 std 中具有相同的名称或略有不同的名称 (并且仍然提供向后兼容性)。但范围工具通常为特定的功能提供更好的支持，可能会修复旧版本的缺陷，或者使用一些概念来限制旧版本的使用。

这些“函数”需要是函数对象或函子，甚至是定制点对象 (CPO 是要求是半常规的，并保证所有实例相等的函数对象)。

它们不支持依赖参数的查找 (ADL)，所以只调用函数对象不正确。但因为它们像函数一样使用，所以我仍然推荐于这样做，其余的就是实现细节了。

出于这个原因，应该选择 std::ranges 中的工具，而不是 std 中的工具。

7.4.1 处理范围 (和数组) 元素的函数

“处理范围元素的泛型函数”表列出了处理范围及其元素的新的独立函数，它们也适用于原始数组。

C++20 之前，几乎所有相应的工具函数都可以直接在命名空间 std 中使用。唯一的例外是:

- ssize()，C++20 中引入了 std::ssize()
- cdata()，不存在于命名空间 std 中

读者们可能想知道为什么命名空间 std::ranges 中会有新函数，而不是修复命名空间 std 中的现有函数。好吧，问题是若修复现有的函数，可能会破坏现有的代码，而向后兼容是 C++ 的一个重要目标。

下一个问题是什么时候使用哪个函数。这里的指导原则非常简单: 优先使用命名空间 std::ranges 中的函数/工具，而不是命名空间 std 中的函数/工具。

原因不仅在于命名空间 std::ranges 中的函数/工具使用了概念，这有助于在编译时发现问题和 bug; 更推荐命名空间 std::ranges 中的新函数的另一个原因是，命名空间 std::ranges 中的函数有时存在缺陷，std::ranges 中的新实现实则修复了这些缺陷:

- 一个问题是参数相关查找 (ADL)。
- 另一个问题是 const 的正确性。

函数	意义
std::ranges::empty(rg)	生成的范围是否为空
std::ranges::size(rg)	生成范围的大小
std::ranges::ssize(rg)	将范围的大小作为 signed 类型的值
std::ranges::begin(rg)	生成指向该范围的第一个元素的迭代器
std::ranges::end(rg)	生成范围哨兵 (一个迭代器)
std::ranges::cbegin(rg)	生成指向范围第一个元素的常量迭代器
std::ranges::cend(rg)	生成范围的一个常量哨兵 (一个到末尾的常量迭代器)
std::ranges::rbegin(rg)	生成范围内第一个元素的反向迭代器
std::ranges::rend(rg)	生成范围的反向前哨 (一个迭代器到末尾)
std::ranges::crbegin(rg)	生成指向范围第一个元素的反向常量迭代器
std::ranges::crend(rg)	生成范围的反向常量哨兵 (一个到末尾的常量迭代器)
std::ranges::data(rg)	生成该范围的原始数据
std::ranges::cdata(rg)	生成具有 const 元素的范围的原始数据

表 7.3 用于处理范围元素的泛型函数

std::ranges::begin() 如何解决 ADL 问题

来看看为什么在处理泛型代码中的范围时，使用 std::ranges::begin() 比使用 std::begin() 更好。问题是，若不巧妙地使用 std::begin()，依赖于参数的查找并不总是有效。

假设要编写泛型代码，调用为范围对象 obj 定义的 begin() 函数。std::begin() 的问题如下:

- 要支持具有 begin() 成员函数的容器等范围类型，可以始终调用成员函数 begin():

```
obj.begin(); // only works if a member function begin() is provided
```

但对于只有独立 begin() 的类型 (例如原始数组)，这不起作用。

- 但若使用独立的 begin()，则有一个问题:
  - 用于原始数组的标准独立 std::begin() 需要使用 std:: 进行限定。
  - 其他非标准范围不能处理完整的限定条件 std::。例如:

```
class MyColl {
  ...
};
... begin(MyColl); // declare free-standing begin() for MyColl

MyColl obj;
std::begin(obj); // std::begin() does not find ::begin(MyType)
```

- 必要的解决方法是在 begin() 前添加一个额外的 using 声明，并且不限定调用本身:

```
using std::begin;
begin(obj); // OK, works in all these cases
```

std::ranges::begin() 则没有这个问题:

```
std::ranges::begin(obj); // OK, works in all these cases
```

诀窍在于 begin 不是一个使用依赖参数查找的函数，而是一个实现所有可能查找的函数对象。参见 lib/begin.cpp 获得完整的示例。

这适用于 std::ranges 中定义的所有工具，所以使用 std::ranges 工具的代码通常支持更多类型和更复杂的用例。

7.4.2 用于处理迭代器的函数

“处理迭代器的泛型函数”表列出了所有用于移动迭代器、向前或向后查找以及计算迭代器之间距离的泛型函数。注意，下一小节描述了交换和移动迭代器引用的元素的函数。

函数	意义
std::ranges::distance(from, to)	生成 from 和 to 之间的距离 (元素数量)
std::ranges::distance(rg)	生成 rg 中元素的数量 (对于没有 size() 的范围也是如此)
std::ranges::next(pos)	生成 pos 后面的下一个元素的位置
std::ranges::next(pos, n)	生成 pos 后第 n 个元素的位置
std::ranges::next(pos, to)	生成 pos 后的位置
std::ranges::next(pos, n, maxpos)	返回第 n 个元素在 pos 之后但不在 maxpos 之后的位置
std::ranges::prev(pos)	生成元素在 pos 之前的位置
std::ranges::prev(pos, n)	生成第 n 个元素在 pos 之前的位置
std::ranges::prev(pos, n, minpos)	返回第 n 个元素在 pos 之前的位置，但不在 minpos 之前
std::ranges::advance(pos, n)	向前/向后推进 n 个元素
std::ranges::advance(pos, to)	将 pos 向前推进到
std::ranges::advance(pos, n, maxpos)	将 pos 向前/向后移动 n 个元素，但不超过 maxpos

表 7.4 处理迭代器的泛型函数

同样，比起命名空间 std 中对应的传统工具，更推荐喜欢这些工具。例如，与 std::ranges::next() 相比，旧的工具 std::next() 要求为传递的迭代器 It 提供 std::iterator_traits<It>::difference_type。但某些视图类型的内部迭代器不支持迭代器特性，因此若使用 std::next()，代码可能无法编译。

7.4.3 用于交换和移动元素/值的函数

范围库还提供了交换和移动值的函数。这些函数列在表中，用于交换和移动元素/值的通用函数，其不能只用于范围。

函数	意义
std::ranges::swap(val1, val2)	交换值 val1 和 val2(使用移动语义)
std::ranges::iter_swap(pos1, pos2)	交换迭代器 pos1 和 pos2 所引用的值 (使用移动语义)
std::ranges::iter_move(pos)	产生迭代器 pos 在移动时所引用的值

表 7.5 用于交换和移动元素/值的通用函数

因为标准概念 std::swappable 和 std::swappable_with 使用了它，所以函数 std::ranges::swap() 在 <concepts> 中定义 (std::swap() 在 <utility> 中定义)。函数 std::ranges::iter_swap() 和 std::ranges::iter_move() 在 <iterator> 中定义。

std::ranges::swap() 修复了在泛型代码中，std::swap() 可能找不到为某些类型提供的最佳交换函数的问题 (类似于 begin() 和 cbegin() 所存在的问题)。

但由于存在泛型回退，所以此修复不会修复无法编译的代码，而可能只会有更好的性能。

考虑下面的例子:

lib/swap.cpp

```cpp
#include <iostream>
#include <utility> // for std::swap()
#include <concepts> // for std::ranges::swap()

struct Foo {
  Foo() = default;
  Foo(const Foo&) {
    std::cout << " COPY constructor\n";
  }
  Foo& operator=(const Foo&) {
    std::cout << " COPY assignment\n";
    return *this;
  }
  void swap(Foo&) {
    std::cout << " efficient swap()\n"; // swaps pointers, no data
  }
};

void swap(Foo& a, Foo& b) {
  a.swap(b);
}

int main()
{
  Foo a, b;
  std::cout << "--- std::swap()\n";
  std::swap(a, b); // generic swap called

  std::cout << "--- swap() after using std::swap\n";
  using std::swap;
  swap(a, b); // efficient swap called
```

```
32
33    std::cout << "--- std::ranges::swap()\n";
34    std::ranges::swap(a, b); // efficient swap called
35  }
```

该程序有以下输出:

```
--- std::swap()
COPY constructor
COPY assignment
COPY assignment
--- swap() after using std::swap
efficient swap()
--- std::ranges::swap()
efficient swap()
```

注意，当 swap 函数在命名空间 std 中定义为未在 std 中定义的类型时 (这在形式上是不允许的)，使用 std::ranges::swap() 会适得其反:

```
1  class Foo {
2    ...
3  };
4
5  namespace std {
6    void swap(Foo& a, Foo& b) { // not found by std::ranges::swap()
7      ...
8    }
9  }
```

应该更改此代码。最好的选择是使用“隐藏友元”:

```
1  class Foo {
2    ...
3    friend void swap(Foo& a, Foo& b) { // found by std::ranges::swap()
4      ...
5    }
6  };
```

函数 std::ranges::iter_move() 和 std::ranges::iter_swap() 与解引用迭代器和调用 std::move() 或 swap() 相比有一个好处: 它们与代理迭代器一起工作得很好，所以要在泛型代码中支持代理迭代器:

```
1  auto val = std::ranges::iter_move(it);
2  std::ranges::iter_swap(it1, it2);
```

而不是:

```
auto val = std::move(*it);
using std::swap;
swap(*it1, *it2);
```

7.4.4 用于值比较的函数

“用于比较标准的通用函数”表列出了现在可以用作比较标准的所有范围工具，其在<functional>中定义。

函数	意义
std::ranges::equal_to(val1, val2)	生成的 val1 是否等于 val2
std::ranges::not_equal_to(val1, val2)	生成的 val1 是否不等于 val2
std::ranges::less(val1, val2)	生成的 val1 是否小于 val2
std::ranges::greater(val1, val2)	生成的 val1 是否大于 val2
std::ranges::less_equal(val1, val2)	生成的 val1 是否小于或等于 val2
std::ranges::greater_equal(val1, val2)	生成的 val1 是否大于或等于 val2

表 7.6 用于比较标准的通用函数

检查相等性时，可以使用概念 std::equality_comparable_with。

在检查顺序时，使用 std::totally_ordered_with 概念。注意，这并不要求必须支持 <=> 操作符，并且该类型不必支持全序。若可以用 <、>、<= 和 >= 比较这些值就足够了，但这是一个在编译时无法检查的语义约束。

注意，C++20 还引入了函数对象类型 std::compare_three_way，可以用它来使用新的操作符 <=>。

7.5. 用于处理范围的新类型函数/工具

本节列出了 C++20 中 ranges 库提供的所有新的辅助类型函数和工具。

至于泛型帮助函数，C++20 之前就已经存在了一些这样的工具，其在命名空间 std 中具有相同的名称或略有不同的名称 (并且仍然提供向后兼容性)，但范围工具通常为指定的功能提供更好的支持。他们可能会修复旧版本的缺陷，或者使用一些概念来限制旧版本的使用。

7.5.1 范围的泛型类型

“使用范围时产生所涉及类型的泛型函数”表列出了范围的所有泛型类型定义，其定义为别名模板。

函数类型	意义
std::ranges::iterator_t<Rg>	在 Rg 上迭代的迭代器类型 (begin() 的结果)
std::ranges::sentinel_t<Rg>	Rg 的 end 迭代器类型 (end() 的结果)
std::ranges::range_value_t<Rg>	范围内元素的类型
std::ranges::range_reference_t<Rg>	元素类型引用的类型
std::ranges::range_difference_t<Rg>	两个迭代器差的类型
std::ranges::range_size_t<Rg>	size() 函数返回的类型
std::ranges::range_rvalue_reference_t<Rg>	元素类型的右值引用的类型
std::ranges::borrowed_iterator_t<Rg>	租借范围为 std::ranges::iterator_t<Rg>，否则为 std::ranges::dangling
std::ranges::borrowed_subrange_t<Rg>	租借范围的迭代器类型的子范围类型，否则为 std::ranges::dangling

表 7.7 使用范围时产生所涉及类型的泛型函数

正如在范围介绍中提到的，这些类型函数的主要优点是它们一般适用于所有类型的范围和视图，这甚至适用于原始数组。

注意，std::ranges::range_value_t<Rg> 只是以下操作的快捷方式:

```
1 std::iter_value_t<std::ranges::iterator_t<Rg>>
```

应该优先于 Rg::value_type。

7.5.2 迭代器的泛型类型

范围库还为迭代器引入了新的类型特征，这些特征不是在命名空间 std:: 范围中定义的，而是在命名空间 std 中定义的。

“泛型函数在使用迭代器时产生的类型”表列出了迭代器的所有泛型类型特征，其定义为别名模板。

函数类型	意义
std::iter_value_t<It>	迭代器引用的值的类型
std::iter_reference_t<It>	值类型引用的类型
std::iter_rvalue_reference_t<It>	值类型的右值引用的类型
std::iter_common_reference_t<It>	引用类型的公共类型和对值类型的引用
std::iter_difference<It>	两个迭代器差的类型

表 7.8 泛型函数在使用迭代器时产生的类型

由于对新的迭代器类别有更好的支持，更推荐这些工具，而非传统的迭代器特征 (std::iterator_traits<>)。

这些函数将在关于迭代器特性的一节中详细描述。

7.5.3 新函数类型

“新函数类型”表列出了可作为辅助函数使用的新类型，在 <functional> 中定义。

函数类型	意义
std::identity	返回自身的函数对象
std::compare_three_way	调用 <=> 操作符的函数对象

表 7.9 新函数类型

若算法支持传递投影，函数对象 std::identity 通常用于在可以传递投影的地方不传递投影:

```
auto pos = std::ranges::find(coll, 25, // find value 25
              [](auto x) {return x*x;}); // for squared elements
```

使用 std::identity 允许程序员传递“无投影”:

```
auto pos = std::ranges::find(coll, 25, // find value 25
              std::identity{}); // for elements as they are
```

该函数对象用作默认模板形参，用于声明可以跳过投影的算法:

```
template<std::ranges::input_range R,
    typename T,
    typename Proj = std::identity>
constexpr std::ranges::borrowed_iterator_t<R>
find(R&& r, const T& value, Proj proj = {});
```

函数对象类型 std::compare_three_way 是一个新的函数对象类型，用于指定调用/使用新的 <=> 操作符 (就像 std::less 或 std::ranges::less 表示调用操作符 <)，这在 <=> 操作符的章节中进行了详述。

7.5.4 处理迭代器的其他新类型

“迭代器的其他新类型”表列出了用于处理迭代器的所有新类型，在 <iterator> 中定义。

函数类型	意义
std::incrementable_traits<It>	产生两个迭代器的 difference_type 的辅助类型
std::projected<It1, It2>	为投影制定约束的类型

表 7.10 迭代器的其他新类型

还要注意新的迭代器和哨兵类型。

7.6. 范围算法

由于支持将范围作为一个参数传递，并且能够处理不同类型的哨兵 (end 迭代器)，C++20 中现在提供了调用算法的新方法。

但其有一些限制，并不是所有的算法都支持传递范围。

7.6.1 范围算法的优点和限制

对于某些算法，还没有允许开发者将范围作为单个参数传递的版本:

- 范围算法不支持并行执行 (C++17 引入)，没有 API 将范围参数作为单个对象和执行策略。
- 目前还没有针对单对象范围的数值算法。要调用像 std::accumulate() 这样的算法，仍然必须传递范围的开始和结束。C++23 可能会提供数值范围算法。

若算法支持范围，则使用概念在编译时查找可能的错误:

- 迭代器和范围的概念确保传递有效的迭代器和范围。
- 可调用对象的概念确保传递有效的辅助函数。

返回类型可能会有所不同，因为:

- 支持并可能返回不同类型的迭代器，为这些迭代器定义了特殊的返回类型。这些类型列在“范围算法的新返回类型”表中。
- 可能返回租借迭代器，表明该迭代器不是有效的迭代器，因为传递了一个临时范围 (右值)。

类型	意义	成员
std::ranges::in_in_result	对于两个输入范围的位置	in1, in2
std::ranges::in_out_result	对于一个输入范围的位置和一个输出范围的位置	in, out
std::ranges::in_in_out_result	用于两个输入范围的位置和一个输出范围的位置	in1, in2, out
std::ranges::in_out_out_result	对于输入范围的一个位置和两个输出范围的位置	in, out1, out2
std::ranges::in_fun_result	对于一个输入范围和一个函数的位置	in, out
std::ranges::min_max_result	对于一个最大和一个最小的位置/值	min, max
std::ranges::in_found_result	对于输入范围和布尔值的一个位置	in, found

表 7.11 范围算法的新返回类型

下面的程序演示了如何使用这些返回类型:

ranges/results.cpp

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

void print(std::string_view msg, auto beg, auto end)
{
  std::cout << msg;
```

```
  for(auto pos = beg; pos != end; ++pos) {
    std::cout << ' ' << *pos;
  }
  std::cout << '\n';
}

int main()
{
  std::vector inColl{1, 2, 3, 4, 5, 6, 7};
  std::vector outColl{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  auto result = std::ranges::transform(inColl, outColl.begin(),
  [] (auto val) {
    return val*val;
  });

  print("processed in:", inColl.begin(), result.in);
  print("rest of in: ", result.in, inColl.end());
  print("written out: ", outColl.begin(), result.out);
  print("rest of out: ", result.out, outColl.end());
}
```

该程序有以下输出:

```
processed in: 1 2 3 4 5 6 7
rest of in:
written out: 1 4 9 16 25 36 49
rest of out: 8 9 10
```

7.6.2 算法概述

各种算法的情况变得越来越复杂。其中有些可以并行使用，其中一些是范围库支持的。本节概述了哪些算法以哪种形式可用 (不详细介绍每个算法的功能)。

后三栏含义如下所示:

- Ranges 表示该算法范围库是否支持
- _result 指示是否使用以及使用哪些_result 类型 (例如，in_out 表示 in_out_result)
- Borrowed 指示算法是否返回租借迭代器

“不可修改的算法”表列出了 C++20 中可用的不可修改标准算法。

名称	启用标准	并行性	Ranges	_result	Borrowed
for_each()	C++98	yes	yes	in_fun	yes
for_each_n()	C++17	yes	yes	in_fun	
count()	C++98	yes	yes		
count_if()	C++98	yes	yes		

min_element()	C++98	yes	yes	yes	
max_element()	C++98	yes	yes	yes	
minmax_element()	C++11	yes	yes	min_max	yes
min()	C++20	no	yes(only)		
max()	C++20	no	yes(only)		
minmax()	C++20	no	yes(only)	min_max	
find()	C++98	yes	yes		yes
find_if()	C++98	yes	yes		yes
find_if_not()	C++11	yes	yes		yes
search()	C++98	yes	yes		yes
search_n()	C++98	yes	yes		yes
find_end()	C++98	yes	yes		yes
find_first_of()	C++98	yes	yes		yes
adjacent_find()	C++98	yes	yes		yes
equal()	C++98	yes	yes		yes
is_permutation()	C++11	no	yes		
mismatch()	C++98	yes	yes	in_in	yes
lexicographical_compare()	C++98	yes	yes		
lexicographical_compare_three_way()	C++20	no	no		
is_sorted()	C++11	yes	yes		
is_sorted_until()	C++11	yes	yes		yes
is_partitioned()	C++11	yes	yes		
partition_point()	C++11	no	yes		
is_heap()	C++11	yes	yes		
is_heap_until()	C++11	yes	yes		yes
all_of()	C++11	yes	yes		
any_of()	C++11	yes	yes		
none_of()	C++11	yes	yes		

表 7.12 不可修改的算法

std::ranges::min()、std::ranges::max() 和 std::ranges::minmax() 算法在 std 中只有两个值或一个 std::initializer_list<> 的对应项。

“可修改算法”表列出了 C++20 中可用的修改标准算法。

名称	启用标准	并行性	**Ranges**	**_result**	**Borrowed**
for_each()	C++98	yes	yes	in_fun	yes
for_each_n()	C++17	yes	yes	in_fun	
copy()	C++98	yes	yes	in_out	yes

copy_if()	C++11	yes	yes	in_out	
copy_n()	C++11	yes	yes	in_out	yes
copy_backward()	C++11	no	yes	in_out	yes
move()	C++11	yes	yes	in_out	yes
move_backward()	C++11	no	yes	in_out	yes
transform() for one range	C++98	yes	yes	in_out	yes
transform() for two ranges	C++98	yes	yes	in_in_out	yes
merge()	C++98	yes	yes		
swap_ranges()	C++98	yes	yes	in_in	yes
fill()	C++98	yes	yes		yes
fill_n()	C++98	yes	yes		
generate()	C++98	yes	yes		yes
generate_n()	C++98	yes	yes		
iota()	C++11	no	no		
replace()	C++98	yes	yes		yes
replace_if()	C++98	yes	yes		yes
replcae_copy()	C++98	yes	yes	in_out	yes
replace_copy_if()	C++98	yes	yes	in_out	yes

表 7.13 可修改算法

“可移除算法”表列出了 C++20 中可用的“移除”标准算法。注意，算法从来没有真正地删除元素。当它们将整个范围作为单个参数时，这仍然适用。改变顺序，使未删除的元素打包到前面，并返回至新的前端。

名称	启用标准	并行性	Ranges	_result	Borrowed
remove()	C++98	yes	yes		yes
remove_if()	C++98	yes	yes		yes
remove_copy()	C++98	yes	yes	in_out	yes
remove_copy_if()	C++98	yes	yes	in_out	yes
unique()	C++98	yes	yes		yes
unique_copy()	C++98	yes	yes	in_out	yes

表 7.14 可移除算法

“可变异算法”表列出了 C++20 中可用的变异标准算法。

名称	启用标准	并行性	Ranges	_result	Borrowed
reverse()	C++98	yes	yes		yes
reverse_copy()	C++98	yes	yes	in_out	yes

rotate()	C++98	yes	yes		yes
rotate_copy()	C++98	yes	yes	in_out	yes
shift_left()	C++20	yes	no		
shift_right()	C++20	yes	no		
sample()	C++17	no	yes		
next_permutation()	C++98	no	yes	in_found	yes
prev_permutation()	C++98	no	yes	in_found	yes
shuffle()	C++11	no	yes		yes
random_shuffle()	C++98	no	no		
partition()	C++98	yes	yes		
stable_partition()	C++98	yes	yes		
partition_copy()	C++11	yes	yes		

表 7.15 可变异算法

“排序算法”表列出了 C++20 中可用的排序标准算法。

名称	启用标准	并行性	Ranges	_result	Borrowed
sort()	C++98	yes	yes		yes
stable_sort()	C++98	yes	yes		yes
partial_sort()	C++98	yes	yes		yes
partial_sort_copy()	C++98	yes	yes	in_out	yes
nth_element()	C++98	yes	yes		yes
partition()	C++98	yes	yes		yes
stable_partition()	C++98	yes	yes		yes
partition_copy()	C++11	yes	yes	in_out_out	yes
make_heap()	C++98	no	yes		yes
push_heap()	C++98	no	yes		yes
pop_heap()	C++98	no	yes		yes
sort_heap()	C++98	no	yes		yes

表 7.16 排序算法

“可排序范围的算法”表中列出了 C++20 中用于排序范围的可用标准算法。

名称	启用标准	并行性	Ranges	_result	Borrowed
binary_search()	C++98	no	yes		
includes()	C++98	yes	yes		
lower_bound()	C++98	no	yes		yes
upper_bound()	C++98	no	yes		yes

equal_range()	C++98	no	yes		yes
merge()	C++98	yes	yes	in_in_out	yes
inplace_merge()	C++98	yes	yes		yes
set_union()	C++98	yes	yes	in_in_out	yes
set_intersection()	C++98	yes	yes	in_in_out	yes
set_difference()	C++98	yes	yes	in_out	yes
set_symmetric_difference()	C++98	yes	yes	in_in_out	yes
partition_point()	C++11	no	yes		yes

表 7.17 可排序范围的算法

“数学算法”表列出了 C++20 中可用的数学标准算法。

名称	启用标准	并行性	Ranges	_result	Borrowed
accumulate()	C++98	no	no		
reduce()	C++17	yes	no		
transform_reduce()	C++17	yes	no		
inner_product()	C++98	no	no		
adjacent_difference()	C++98	no	no		
partial_sum()	C++98	no	no		
inclusive_scan()	C++17	yes	no		
exclusive_scan()	C++17	yes	no		
transform_inclusive_scan()	C++17	yes	no		
transform_exclusive_scan()	C++17	yes	no		
iota	C++11	no	no		

表 7.18 数学算法

第 8 章 视图类型的详情

本章会讨论 C++20 标准库引入的所有视图类型的细节。

首先，概述了所有视图类型，讨论了视图的通用部分，例如：C++ 标准库提供的通用基类。

之后，将根据视图的一般特征在多个章节中讨论 C++20 的各种视图类型: 视图是视图管道的初始构建块 (使用现有值或自己生成值)，过滤和转换视图，改变视图，最后是多范围视图。视图类型的每个描述都以最重要特征的概述开始。

请注意，关于 std::span 的下一章将讨论 span 视图的其他细节 (历史上，span 不是范围库的一部分，但其也是视图)。

8.1. 所有视图的概览

C++20 提供了大量不同的视图。其中一些封装了现有源的元素，其中一些本身生成值，还有许多对元素或值进行操作 (过滤或转换)。

本章简要概述了 C++20 中所有可用的视图类型，对于几乎所有这些类型，都有辅助的范围适配器/工厂，允许开发者通过调用函数或管道来创建视图。应该使用这些适配器和工厂，而不是直接初始化视图，因为适配器和工厂执行额外的优化 (例如在不同的视图中选择最佳的)，可以在更多的情况下工作，对需求进行双重检查，并且更容易使用。

8.1.1 概述包装和生成视图

“包装和生成视图”表列出了只能作为管道源元素的标准视图。

这些观点可能是

- 包装视图，对外部资源的元素序列进行操作 (例如：从输入流读取的容器或元素)
- 工厂视图，自己生成元素

类型	适配器/工厂	效果
std::ranges::red_view	all(rg)	引用一个范围
std::ranges::owning_view	all(rg)	包含范围的视图
std::ranges::subrange	counted(beg, sz)	开始迭代器和哨兵
std::span	counted(beg, sz)	指向连续内存和大小的开始迭代器
std::ranges::iota_view	iota(val) iota(val, endVal)	递增值的生成器
std::ranges::single_view	single(val)	视图只拥有一个值
std::ranges::empty_view	empty<T>	没有元素的视图
std::ranges::basic_istream_view std::ranges::istream_view<> std::ranges::wistream_view<>	istream<T>(strm) -	从流中读取元素 从 char 流 从 wchar_t 流

std::basic_string_view	-	字符的只读视图
std::string_view	-	转换成 char 序列
std::u8string_view	-	转换为 char8_t 序列
std::u16string_view	-	转换为 char16_t 序列
std::u32string_view	-	转换为 char32_t 序列
std::wstring_view	-	转换为 wchar_t 序列

表 8.1 包装和生成视图

该表列出了视图的类型和可用于创建视图的范围适配器/工厂的名称 (若有的话)。

对于字符串视图，C++17 中作为 std::basic_string_view<> 类型引入，C++ 提供了别名类型 std::string_view, std::u8string_view, std::u16string_view, std::u32string_view 和 std::wstring_view。

对于 istream 视图，其类型 std::basic_istream_view<>，C++ 提供了别名 std::istream_view 和 std::wistream_view。

注意，视图类型使用不同的命名空间:

- span 和 string_view 在命名空间 std 中。
- 所有范围适配器和工厂都在命名空间 std::views 中提供 (std::ranges::views 的别名)。
- 所有其他视图类型都在命名空间 std::ranges 中提供。

一般来说，最好使用范围适配器和工厂。应该倾向于使用适配器 std::views::all()，而不是直接使用类型 std::ranges::ref_view<> 和 std::ranges::owning_view<>。然而，在某些情况下，没有提供适配器/工厂，必须直接使用视图类型。最重要的例子是当必须从一对迭代器创建视图时，必须直接初始化 std::ranges::subrange。

8.1.2 自适应视图概述

“适应视图”表列出了以某种方式适应给定范围的元素的标准视图 (过滤掉元素、修改元素的值、改变元素的顺序，或者组合或创建子范围)，尤其可以在视图的管道中使用。

类型	适配器	效果
std::ranges::ref_view	all(rg)	引用一个范围
std::ranges::owning_view	all(rg)	包含范围的视图
std::ranges::take_view	take(num)	第一个 (最多)num 个元素
std::ranges::take_while_view	take_while(pred)	匹配谓词的所有前置元素
std::ranges::drop_view	drop(num)	除第一个 num 元素外的所有元素
std::ranges::drop_while_view	drop_while(pred)	除了与谓词匹配的前置元素之外的所有元素
std::ranges::filter_view	filter(pred)	匹配谓词的所有元素
std::ranges::transform_view	transform(func)	所有元素转换后的值
std::ranges::elements_view	elements<idx>	所有元素的 idxth 的成员/属性
std::ranges::keys_view	keys	所有元素的第一个成员

std::ranges::values_view	values	所有元素的第二个成员
std::ranges::reverse_view	reverse	所有元素按倒序排列
std::ranges::split_view	split(sep)	范围中的所有元素分割为子范围
std::ranges::lazy_split_view	lazy_split(sep)	输入范围或常量范围的所有元素拆分为子范围
std::ranges::join_view	join	多个范围的范围中的所有元素
std::ranges::common_view	common()	迭代器和哨兵中所有类型相同的元素

表 8.2 适应视图

同样，推荐使用范围适配器，而不是直接使用视图类型。应该倾向于使用适配器 std::views::take()，而不是类型 std::ranges::take_view<>，当可以简单地跳转到底层范围的第 n 个元素时，适配器可能根本不会创建获取视图。作为另一个例子，应该总是更喜欢使用适配器 std::views::common()，而不是类型 std::ranges::common_view<>，因为只有适配器允许传递已经通用的范围 (只是接受它们的原样)，用 common_view 包装公共范围会导致编译时错误。

8.2. 视图的基类和命名空间

视图有一个公共类型，提供了视图的大多数成员函数，这将在本节中讨论。

本节还讨论为什么范围适配器和工厂使用特殊的命名空间。

8.2.1 视图的基类

所有标准视图都派生自类 std::ranges::view_interface<viewType>。已发布的 C++20 标准中，view_interface<> 也派生自一个空基类 std::ranges::view_base，但这个问题通过http://wg21.link/lwg3549解决了。]

类模板 std::ranges::view_interface<> 基于派生视图类型的 begin() 和 end() 的定义引入了几个基本成员函数，这些函数必须作为模板参数传递给这个基类。std::ranges::view_interface<> 的操作表列出了类为视图提供的 API。

操作	效果	是否提供
r.empty()	返回 r 是否为空 (begin() == end())	前向迭代器
if (r)	若 r 不为空，则为 True	前向迭代器
r.size()	产生元素的数目	是否可以计算和 end 之间的差值
r.front()	生成的第一个元素	前向迭代器
r.back()	生成的最后一个元素	双向迭代器和 end() 产生与 begin() 相同的类型
r[idx]	生成的第 n 个元素	随机访问迭代器
r.data()	生成指向元素内存的原始指针	元素位于连续内存中

表 8.3 std::ranges::view_interface<> 的操作

类 view_interface<> 还为派生自它的每个类型初始化了 std::ranges::enable_view<>，这些类型满足 std::ranges::view 概念。

当自定义视图类型时，应该从 view_interface<> 派生，并将自己的类型作为参数传递。例如:

```
template<typename T>
class MyView : public std::ranges::view_interface<MyView<T>> {
  public:
  ... begin() ... ;
  ... end() ... ;
  ...
};
```

基于 begin() 和 end() 的返回类型，类型自动提供 std::ranges::view_interface<> 的操作表中列出的成员函数，若可用性的先决条件合适。这些成员函数的 const 版本要求，视图类型的 const 版本是一个有效范围。

C++23 将添加成员 cbegin() 和 cend()，将其映射到 std::ranges::cbegin() 和 std::ranges::cend()(将与http://wg21.link/p2278r4一起添加)。

8.2.2 为什么范围适配器/工厂有自己的命名空间

范围适配器和工厂有自己的命名空间 std::ranges::views，为其定义了命名空间别名 std::views:

```
namespace std {
  namespace views = ranges::views;
}
```

这样，就可以为可能在其他命名空间 (甚至在另一个标准命名空间) 中使用的视图使用名称。

可以 (也必须) 在使用视图时进行限定:

```
std::ranges::views::reverse // full qualification
std::views::reverse // shortcut
```

通常，不加限定地使用视图是不可能的。ADL 不会启动，因为在命名空间 std::views 中没有定义范围 (容器或视图)。

```
std::vector<int> v;
...
take(v, 3) | drop(2); // ERROR: can't find views (may find different symbol)
```

即使视图也会产生不属于视图命名空间的对象 (std::ranges 中)，使用视图时仍然需要对视图进行限定:

```
std::vector<int> values{ 0, 1, 2, 3, 4 };
auto v1 = std::views::all(values);
auto v2 = take(v1, 3); // ERROR
auto v3 = std::views::take(v1, 3); // OK
```

有一件重要的事情需要注意: 永远不要使用 using 声明跳过范围适配器的限定:

```
using namespace std::views; // do not do this
```

以作 composers 为例，若只是用定义的局部值对象过滤掉值，就会遇到麻烦:

```
std::vector<int> values;
...
std::map<std::string, int> composers{ ... };

using namespace std::views; // do not do this

for (const auto& elem : composers | values) { // OOPS: finds wrong values
  ...
}
```

本例中，将使用本地 vector 值，而非视图。若很幸运，会得到了一个编译时错误。若不够幸运，非限定视图会找到不同的符号，这可能会导致遇到，甚至出现覆盖其他对象内存的未定义行为。

8.3. 外部元素的源视图

本节讨论 C++20 中创建引用现有外部值的视图的所有特性 (通常作为单个范围参数传递，作为开始迭代器和哨兵，或作为开始迭代器和计数)。

8.3.1 子范围

类型:	std::ranges::subrange<>
内容:	从传递开始到结束的所有元素
工厂:	std::views::counted() std::views::reverse() 在逆子范围上
元素类型:	传递迭代器中值的类型
要求:	至少是输入迭代器
类别:	和传入的一样
是否为有限范围:	若传递的是普通随机访问迭代器或是长度提示
是否为常规范围:	若在通用迭代器上
是否为租借范围:	总是
缓存:	无
常量可迭代:	若传递的迭代器可复制
传播常量性:	从不

类模板 std::ranges::subrange<> 定义了一个范围元素的视图，通常以开始迭代器和哨兵 (结束迭代器) 对的形式传递，也可以传递单个范围对象，并间接传递开始迭代器和计数。在内部，视图本身通过存储开始 (迭代器) 和结束 (哨兵) 来展示元素。

子范围视图的主要用例是将一对开始迭代器和哨兵 (结束迭代器) 转换为一个对象。例如:

```
std::vector<int> coll{0, 8, 15, 47, 11, -1, 13};

std::ranges::subrange s1{std::ranges::find(coll, 15),
            std::ranges::find(coll, -1)};
print(coll); // 15 47 11
```

可以用结束值的特殊哨兵来初始化子范围:

```
std::ranges::subrange s2{coll.begin() + 1, EndValue<-1>{}};
print(s2); // 8 15 47 11
```

这两种方法在将一对迭代器转换为范围/视图时都特别有用，这样范围适配器就可以处理元素:

```
void foo(auto beg, auto end)
{
  // init view for all but the first five elements (if there are any):
  auto v = std::ranges::subrange{beg, end} | std::views::drop(5);
  ...
}
```

下面是演示子范围用法的完整示例:

ranges/subrange.cpp

```
#include <iostream>
#include <string>
#include <unordered_map>
#include <ranges>

void printPairs(auto&& rg)
{
  for (const auto& [key, val] : rg) {
    std::cout << key << ':' << val << ' ';
  }
  std::cout << '\n';
}

int main()
{
  // English/German dictionary:
  std::unordered_multimap<std::string,std::string> dict = {
    {"strange", "fremd"},
    {"smart", "klug"},
    {"car", "Auto"},
    {"smart","raffiniert"},
    {"trait", "Merkmal"},
    {"smart", "elegant"},
```

```
  };

  // get begin and end of all elements with translation of "smart":
  auto [beg, end] = dict.equal_range("smart");

  // create subrange view to print all translations:
  printPairs(std::ranges::subrange(beg, end));
}
```

该程序有以下输出:

```
smart:klug smart:elegant smart:raffiniert
```

equal_range() 返回了键为”smart” 的 dict 中所有元素的开头和结尾，这里使用一个子范围将这两个迭代器转换为一个视图 (这里是字符串对元素的视图):

```
std::ranges::subrange(beg, end)
```

可以将视图传递给 printPairs()，遍历元素并输出。

子范围可以改变它的大小，以便在 begin 和 end 都有效的情况下在 begin 和 end 之间插入或删除元素:

```
std::list coll{1, 2, 3, 4, 5, 6, 7, 8};

auto v1 = std::ranges::subrange(coll.begin(), coll.end());
print(v1); // 1 2 3 4 5 6 7 8

coll.insert(++coll.begin(), 0);
coll.push_back(9);
print(v2); // 1 0 2 3 4 5 6 7 8 9
```

子范围的工厂

没有范围工厂用于从 begin(迭代器) 和 end(哨兵) 初始化子范围，但有一个范围工厂可以从 begin 和 count 创建子范围:[C++20 最初声明 all() 也可以产生子范围。当后来引入了所属视图时，但这个选项删除了 (参见http://wg21.link/p2415)]

```
std::views::counted(beg, sz)
```

std::views::counted() 创建一个包含非连续范围的前 sz 个元素的子范围，从迭代器 beg 引用的元素开始 (对于连续范围，counts() 创建一个 span 视图)。

当 std::views::counted() 创建子范围时，子范围以 std::counted_iterator 作为开始，以 std::default_sentinel_t 类型的伪哨兵作为结束:

```
std::views::counted(rg.begin(), 5);
```

相当于:

```
std::ranges::subrange{std::counted_iterator{rg.begin(), 5},
  std::default_sentinel};
```

这样做的效果是，即使插入或删除元素，该子范围内的计数仍然保持稳定:

```
std::list coll{1, 2, 3, 4, 5, 6, 7, 8};

auto v2 = std::views::counted(coll.begin(), coll.size());
print(v2); // 1 2 3 4 5 6 7 8

coll.insert(++coll.begin(), 0);
coll.push_back(9);
print(v2); // 1 0 2 3 4 5 6 7
```

有关详细信息，请参阅 std::views::counted() 和 std::counted_iterator 类型的描述。

下面是一个完整的示例，演示了 counted() 如何使用子范围:

ranges/subrangecounted.cpp

```
#include <iostream>
#include <string>
#include <unordered_map>
#include <ranges>

void printPairs(auto&& rg)
{
  for (const auto& [key, val] : rg) {
    std::cout << key << ':' << val << ' ';
  }
  std::cout << '\n';
}

int main()
{
  // English/German dictionary:
  std::unordered_multimap<std::string,std::string> dict = {
    {"strange", "fremd"},
    {"smart", "klug"},
    {"car", "Auto"},
    {"smart","raffiniert"},
    {"trait", "Merkmal"},
    {"smart", "elegant"},
  };

```

```
  printPairs(dict);
  printPairs(std::views::counted(dict.begin(), 3));
}
```

该程序有以下输出:

```
trait:Merkmal car:Auto smart:klug smart:elegant smart:raffiniert strange:fremd
trait:Merkmal car:Auto smart:klug
```

极少数情况下，std::views::reverse() 也可能产生子范围。当反转一个反转的子范围时，就会发生这种情况，将会得到原始的子范围。

子范围的特殊特征

子范围可以是也可以不是一个长度范围。实现的方式是，子范围有第三个模板形参，枚举类型 std::ranges::subrange_kind，其值可以是 std::ranges::unsized 或 std::ranges::sized。该模板形参的值可以指定，也可以从为迭代器和哨兵类型调用的 std::sized_sentinel_for 概念派生。

会有以下后果:

- 若传递相同类型的连续迭代器或随机访问迭代器，则子范围的大小为:

```
std::vector vec{1, 2, 3, 4, 5}; // has random access
...
std::ranges::subrange sv{vec.begin()+1, vec.end()-1}; // sized
std::cout << std::ranges::sized_range<decltype(sv)>; // true
```

- 若传递的迭代器不支持随机访问或不常见，则子范围不确定大小:

```
std::list lst{1, 2, 3, 4, 5}; // no random access
...
std::ranges::subrange sl{++lst.begin(), --lst.end()}; // unsized
std::cout << std::ranges::sized_range<decltype(sl)>; // false
```

- 后一种情况下，可以传递一个长度来使子范围变成一个长度范围:

```
std::list lst{1, 2, 3, 4, 5}; // no random access
...
std::ranges::subrange sl2{++lst.begin(), --lst.end(), lst.size()-2}; // sized
std::cout << std::ranges::sized_range<decltype(sl2)>; // true
```

若用错误的大小初始化视图，或者在开始和结束之间插入或删除元素之后使用视图，则为未定义行为。

子范围的接口

“类 std::ranges::subrange<> 的操作”表列出了子范围的 API。

只有当迭代器是默认可初始化时，才提供默认构造函数。

使用 szHint 参数的构造函数，允许开发者将未设置长度的范围转换为设置长度的子范围，如上所示。

子范围的迭代器指向底层 (或临时创建的) 范围，所以子范围是租借范围，但当底层范围不再存在时，迭代器仍然悬空。

类似元组的子范围接口

std::ranges::subange 也有一个类似于元组的接口来支持结构化绑定 (C++17 中引入)，所以可以很容易地初始化一个开始迭代器和一个哨兵 (结束迭代器)，方法如下:

```
1 auto [beg, end] = std::ranges::subrange{coll};
```

为此，提供类 std::ranges::subange;

- std::tuple_size<> 的特化
- std::tuple_element<> 的特化
- 获取索引 0 和 1 的 get<>() 函数

操作	效果
subrange r{}	创建一个空子范围
subrange r{rg}	使用范围 rg 的元素创建子范围
subrange r{rg, szHint}	使用范围 rg 的元素创建子范围，指定该范围有 szHint 个元素
subrange r{beg, end}	使用 range [beg, end) 的元素创建子范围
subrange r{beg, end, szHint}	使用 range [beg, end) 的元素创建子范围，指定该范围有 szHint 个元素
r.begin()	生成 begin 迭代器
r.end()	生成哨兵 (end 迭代器)
r.empty()	生成 r 是否为空
if (r)	若 r 不为空，则为 True
r.size()	生成元素的数量 (若有长度则可用)
r.front()	生成第一个元素 (若转发可用)
r.back()	生成最后一个元素 (若为双向且通用则可用)
r[idx]	生成第 n 个元素 (如果随机访问可用)
r.data()	生成一个指向元素内存的原始指针 (若元素在连续内存中可用)。
r.next()	生成从第二个元素开始的子范围
r.next(n)	生成从第 n 个元素开始的子范围
r.prev()	生成从第一个元素之前的元素开始的子范围
r.prev(n)	生成一个子范围，从第 n 个元素开始，在第一个元素之前
r.advance(dist)	修改 r，使其稍后开始 num 元素 (以负 num 开始较早)
auto [beg, end] = r	用 begin 和 end/哨兵 (r) 初始化 beg 和 end

表 8.4 类 std::ranges::subrange<> 的操作

使用子范围协调原始数组类型

子范围的类型仅取决于迭代器的类型 (以及是否提供了 size())，这可以用来协调原始数组的类型。

考虑下面的例子:

```
int a1[5] = { ... };
int a2[10] = { ... };

std::same_as<decltype(a1), decltype(a2)> // false

std::same_as<decltype(std::ranges::subrange{a1}),
       decltype(std::ranges::subrange{a2})> // true
```

这只适用于原始数组，对于 std::array<> 和其他容器，迭代器 (可能) 具有不同的类型，所以协调的类型与子范围不可移植。

8.3.2 参考视图

类型:	std::ranges::ref_view<>
内容:	范围的所有元素
工厂:	std::views::all() and all other adaptors on lvalues
元素类型:	与传入的范围类型相同
要求:	至少为输入范围
类别:	与传入相同
是否是长度范围:	若是长度范围
是否是通用范围:	若是常规范围
是否是租借范围:	总是
缓存:	无
常量可迭代:	若在可迭代范围内
传播常量性:	从不

类模板 std::ranges::ref_view<> 定义了一个简单引用范围的视图，通过值传递视图的效果就像通过引用传递范围一样。

其效果类似于 std::reference_wrapper<> 类型 (C++11 引入)，对使用 std::ref() 和 std::cref() 创建的第一类对象进行引用，但这个包装器的好处仍然可以直接用作范围。

ref_view 的用例是将容器转换为复制成本较低的轻量级对象。例如:

```
void foo(std::ranges::input_range auto coll) // NOTE: takes range by value
{
  for (const auto& elem : coll) {
```

```
    ...
  }
}

std::vector<std::string> coll{ ... };

foo(coll); // copies coll
foo(std::ranges::ref_view{coll}); // pass coll by reference
```

将容器传递给协程 (通常必须按值接受参数) 可能是这种技术的一种应用。

请注意，只能创建一个左值 (一个有名字的范围) 的参考视图:

```
std::vector coll{0, 8, 15};
...
std::ranges::ref_view v1{coll}; // OK, refers to coll
std::ranges::ref_view v2{std::move(coll)}; // ERROR
std::ranges::ref_view v3{std::vector{0, 8, 15}}; // ERROR
```

对于右值，必须使用归属视图。

下面是一个使用参考视图的完整示例:

ranges/refview.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <ranges>

void printByVal(std::ranges::input_range auto coll) // NOTE: takes range by value
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector<std::string> coll{"love", "of", "my", "life"};

  printByVal(coll); // copies coll
  printByVal(std::ranges::ref_view{coll}); // pass coll by reference
  printByVal(std::views::all(coll)); // ditto (using a range adaptor)
}
```

参考视图的范围适配器

参考视图也可以 (通常应该) 用范围工厂创建:

```
std::views::all(rg)
```

std::views::all() 创建一个 ref_view 到传入的范围 rg，前提是 rg 是左值，而非视图 (若 rg 已经是视图，则返回 rg；若传递右值，则返回归属视图)。

此外，若传递的左值还不是视图，几乎所有其他范围适配器也会创建一个参考视图。

上面的示例程序使用这种间接的方式，通过最后一次调用 printByVal() 创建了一个参考视图，其传递一个 std::ranges::ref_view<std::vector<std::string>>>:

```
std::vector<std::string> coll{"love", "of", "my", "life"};
...
printByVal(std::views::all(coll));
```

请注意，所有其他接受范围的视图都间接地为传递的范围调用 all()(通过使用 std::views::all_t<>)。出于这个原因，若传递一个不是视图的左值给其中一个视图，一个参考视图总是自动创建的:

```
std::views::take(coll, 3)
```

基本上与调用以下代码的效果相同:

```
std::ranges::take_view{std::ranges::ref_view{coll}, 3};
```

但使用适配器时，可能会有一些优化。

详细信息请参见 std::views::all() 的描述。

参考视图的特点

参考视图存储对基础范围的引用，所以参考视图是一个租借范围。只要引用的视图有效，就可以使用它。例如，若底层视图是一个 vector，重新分配并不会使该视图失效。但当底层范围不再存在时，迭代器仍然可以悬空。

参考视图的接口

“类 std::ranges::ref_view<> 的操作”表列出了 ref_view 的 API。

操作	效果
ref_view r{rg}	创建一个引用范围 rg 的 ref_view
r.begin()	生成 begin 迭代器
r.end()	生成哨兵 (end 迭代器)
r.empty()	生成 r 是否为空 (若范围支持则可用)
if(r)	若 r 不为空则为 true(若定义了 empty() 则可用)

r.size()	生成元素的数量 (若引用了一个长度范围，则可用)
r.front()	生成第一个元素 (若转发可用)
r.back()	生成最后一个元素 (若双向且通用则可用)
r[idx]	生成第 n 个元素 (若随机访问可用)
r.data()	生成一个指向元素内存的原始指针 (若元素在连续内存中可用)
r.base()	生成对 r 所引用的范围的引用

表 8.5 类 std::ranges::ref_view<> 的操作表

注意，没有为这个视图提供默认构造函数。

8.3.3 归属视图

类型:	std::ranges::owning_view<>
内容:	移动范围内的所有元素
适配器:	std::views::all() 和所有其他对右值的适配器
元素类型:	与传递的范围类型相同
要求:	至是少输入范围
类别:	与传入的类别相同
是否是长度范围:	若为长度范围
是否是通用范围:	若为常规范围
是否是租借范围:	若为租借范围
缓存:	无
常量可迭代:	若在可迭代范围内
传播常量性:	总是

类模板 std::ranges::owning_view<> 定义了一个获取另一个范围元素所有权的视图。[最初，C++20 没有归属视图，该视图是在后来的修复中引入的 (参见http://wg21.link/p2415)。]

这是 (到目前为止) 唯一可能拥有多个元素的视图，但构造成本仍然很低，因为初始范围必须是右值 (临时对象或用 std::move() 标记的对象)，构造函数将范围移动到视图的内部成员。

例如:

```
std::vector vec{0, 8, 15};
std::ranges::owning_view v0{vec}; // ERROR
std::ranges::owning_view v1{std::move(vec)}; // OK
print(v1); // 0 8 15
print(vec); // unspecified value (was moved away)

std::array<std::string, 3> arr{"tic", "tac", "toe"};
std::ranges::owning_view v2{arr}; // ERROR
std::ranges::owning_view v2{std::move(arr)}; // OK
```

```
print(v2); // "tic" "tac" "toe"
print(arr); // "" "" ""
```

这是 C++20 中唯一一个完全不支持复制的标准视图，只能移动。例如:

```
std::vector coll{0, 8, 15};
std::ranges::owning_view v0{std::move(coll)};

auto v3 = v0; // ERROR
auto v4 = std::move(v0); // OK (range in v0 moved to v4)
```

owning_view 的主要用例是从一个范围创建视图，而不再依赖于范围的生命周期。例如:

```
void foo(std::ranges::view auto coll) // NOTE: takes range by value
{
  for (const auto& elem : coll) {
    ...
  }
}

std::vector<std::string> coll{ ... };

foo(coll); // ERROR: no view
foo(std::move(coll)); // ERROR: no view
foo(std::ranges::owning_view{coll}); // ERROR: must pass rvalue
foo(std::ranges::owning_view{std::move(coll)}); // OK: move coll as view
```

将范围转换为归属视图，通常是通过将临时容器传递给范围适配器来隐式地完成的 (见下文)。

归属视图的范围适配器

归属视图也可以 (通常应该) 用范围工厂创建:

```
std::views::all(rg)
```

std::views::all() 为传入的范围 rg 创建一个 owning_view，前提是 rg 是一个右值而不是一个视图 (若已经是一个视图则返回 rg，否则传递左值则返回 ref 视图)。

此外，若传递的右值还不是视图，几乎所有其他范围适配器也会创建一个归属视图。

因此，上面的例子可以像下面这样对 foo() 的调用:

```
foo(std::views::all(std::move(coll))); // move coll as view
```

要创建一个归属视图，适配器 all() 需要一个右值 (对于一个左值，all() 创建一个参考视图)，所以必须传递一个临时范围或一个用 std::move() 标记的命名范围 (若传递左值，all() 将产生一个参考视图)。

下面是一个完整的例子:

ranges/owningview.cpp

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <ranges>

auto getColl()
{
  return std::vector<std::string>{"you", "don't", "fool", "me"};
}

int main()
{
  std::ranges::owning_view v1 = getColl(); // view owning a vector of strings
  auto v2 = std::views::all(getColl()); // ditto
  static_assert(std::same_as<decltype(v1), decltype(v2)>);

  // iterate over drop view of owning view of vector<string>:
  for (const auto& elem : getColl() | std::views::drop(1)) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

这里，v1 和 v2 都有类型 std::ranges::owning_view<std::vector<std::string>>。然而，创建拥有视图的常用方法是在程序末尾基于范围的 for 循环中进行演示: 将一个临时容器传递给范围适配器时，会间接地创建了一个归属视图。

表达式

```cpp
getColl() | std::views::drop(1)
```

创建了

```cpp
std::ranges::drop_view<std::ranges::owning_view<std::vector<std::string>>>
```

这是通过使用 std::views::all_t<> 对传递的临时范围间接调用 all() 实现的。

详细信息请参见 std::views::all() 的描述。

归属视图的特殊特性

归属视图将传递的范围移动到自身，若拥有的范围是租借范围，则拥有视图是租借范围。

归属视图的接口

“类 std::ranges::owning_view<> 的操作”表列出了 owning_view 的 API。

只有当迭代器是默认可初始化时，才提供默认构造函数。

操作	效果
owning_view r{}	创建一个空的 owning_view
owning_view r{rg}	创建一个拥有 rg 元素的 owning_view
r.begin()	生成 begin 迭代器
r.end()	生成哨兵 (end 迭代器)
r.empty()	生成 r 是否为空 (若范围支持则可用)
if(r)	若 r 不为空则为 True(若定义了 empty() 则可用)
r.size()	生成元素的数量 (若引用了一个长度范围，则可用)
r.front()	生成第一个元素 (若转发可用)
r.back()	生成最后一个元素 (若是双向且通用的，则可用)
r[idx]	生成第 n 个元素 (若随机访问可用)
r.data()	生成一个指向元素内存的原始指针 (若元素在连续内存中可用)。
r.base()	生成对生成区域的引用

表 8.6 类 std::ranges::owning_view<> 的操作

8.3.4 通用视图

类型:	std::ranges::common_view<>
内容:	具有统一迭代器类型的范围的所有元素
适配器:	std::views::common()
元素类型:	与传入的 range 类型相同
要求:	非通用至少前向范围 (适配器接受通用范围)
类别:	通常是向前的 (若在一个连续的范围内连续)
是否是长度范围:	若为长度范围
是否是通用范围:	总是
是否是租借范围:	若为租借范围
缓存:	无
常量可迭代:	若在可迭代范围内
传播常量性:	只有在右值范围内

类模板 std::ranges::common_view<> 是一个视图，其协调范围的开始迭代器和结束迭代器类型，以便能够将其传递给需要相同迭代器类型的代码 (例如容器的构造函数或传统算法)。

注意，视图的构造函数要求传递的范围不是通用范围 (迭代器有不同的类型)。例如:

```
1 std::list<int> lst{1, 2, 3, 4, 5, 6, 7, 8, 9};
2
3 auto v1 = std::ranges::common_view{lst}; // ERROR: is already common
4
```

```
auto take5 = std::ranges::take_view{lst, 5}; // yields no common view
auto v2 = std::ranges::common_view{take5}; // OK
```

相应的范围适配器还可以处理已经很常见的范围，最好使用范围适配器 std::views::common() 来创建通用视图。

通用视图的范围适配器

通用视图也可以 (通常应该) 使用范围适配器创建:

```
std::views::common(rg)
```

std::views::common() 创建一个将 rg 引用为通用范围的视图。若 rg 已经是公共的，返回 std::views::all(rg)。

因此，可以使上面的示例始终按照以下方式编译:

```
std::list<int> lst{1, 2, 3, 4, 5, 6, 7, 8, 9};

auto v1 = std::views::common(lst); // OK

auto take5 = std::ranges::take_view{lst, 5}; // yields no common view
auto v2 = std::views::common(take5); // v2 is common view
std::vector<int> coll{v2.begin(), v2.end()}; // OK
```

通过使用适配器，从已经很常见的范围中创建通用视图并不是编译时错误。这是选择适配器，而非直接初始化视图的关键原因。

下面是使用通用视图的完整示例程序:

ranges/commonview.cpp

```
#include <iostream>
#include <string>
#include <list>
#include <vector>
#include <ranges>

void print(std::ranges::input_range auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::list<std::string> coll{"You're", "my", "best", "friend"};
```

```
18
19    auto tv = coll | std::views::take(3);
20    static_assert(!std::ranges::common_range<decltype(tv)>);
21    // could not initialize a container by passing begin and end of the view
22
23    std::ranges::common_view vCommon1{tv};
24    static_assert(std::ranges::common_range<decltype(vCommon1)>);
25
26    auto tvCommon = coll | std::views::take(3) | std::views::common;
27    static_assert(std::ranges::common_range<decltype(tvCommon)>);
28
29    std::vector<std::string> coll2{tvCommon.begin(), tvCommon.end()}; // OK
30    print(coll);
31    print(coll2);
32  }
```

这里，首先创建一个视图，若不是一个随机访问范围。这是不常见的，所以不能用它来初始化容器的元素:

```
1  std::vector<std::string> coll2{tv.begin(), tv.end()}; // ERROR: begin/end types
   ↪  differ
```

使用通用视图，初始化工作得很好。若不知道要协调的范围是否已经是公共的，则使用 std::views::common()。

通用视图的主要用例是将 begin 和 end 迭代器，具有不同类型的范围或视图传递给需要 begin 和 end 具有相同类型的泛型代码。一个典型的例子是将，范围的开始和结束传递给容器构造函数或传统算法。例如:

```
1  std::list<int> lst {1, 2, 3, 4, 5, 6, 7, 8, 9};
2
3  auto v1 = std::views::take(lst, 5); // Note: types of begin() and end() differ
4  std::vector<int> coll{v1.begin(), v1.end()}; // ERROR: containers require the same
   ↪  type
5
6  auto v2 = std::views::common(std::views::take(lst, 5)); // same type now
7  std::vector<int> coll{v2.begin(), v2.end()}; // OK
```

通用视图的接口

“类 std::ranges::common_view<> 的操作”表列出了通用视图的 API。

操作	效果
common_view r{}	创建引用默认构造范围的 common_view
common_view r{rg}	创建引用范围 rg 的 common_view

r.begin()	生成 begin 迭代器
r.end()	生成哨兵 (end 迭代器)
r.empty()	生成 r 是否为空 (若范围支持则可用)
if (r)	若 r 不为空则为 true(若定义了 empty() 则可用)
r.size()	生成元素的数量 (若引用了一个长度范围，则可用)
r.front()	生成第一个元素 (若转发可用)
r.back()	生成最后一个元素 (若双向且通用的，则可用)
r[idx]	生成第 n 个元素 (如果随机访问可用)
r.data()	产生一个指向元素内存的原始指针 (若元素在连续内存中可用)。
r.base()	产生对 r 所引用范围的引用

表 8.7 类 std::ranges::common_view<> 的操作表

只有当迭代器是默认可初始化时，才提供默认构造函数。在内部，common_view 使用 std::common_iterator 类型的迭代器。

8.4. 生成视图

本节讨论 C++20 中用于创建生成值的视图的所有特性 (不引用视图外部的元素或值)。

8.4.1 iota 视图

类型:	std::ranges::iota_view<>
内容:	值递增序列的生成器
工厂:	std::views::iota()
元素类型:	值
类别:	随机访问的输入 (取决于开始值的类型)
是否是长度范围:	若用结束值初始化
是否是通用范围:	若是有限的，end 和值的类型相同
是否是租借范围:	总是
缓存:	无
常量可迭代:	总是
传播常量性:	从不 (但元素不是左值)

类模板 std::ranges::iota_view<> 是一个生成一系列值的视图。这些值可以是整型，例如

- 1, 2, 3 ...
- ‘a’，‘b’，‘c’...

也可以使用自加操作符生成指针或迭代器序列。
这个序列可能有限，也可能无限。

iota 视图的主要用例是提供在一系列值上迭代的视图。例如:

```
std::ranges::iota_view v1{1, 100}; // view with values: 1, 2, ... 99
for (auto val : v1) {
  std::cout << val << '\n'; // print these values
}
```

iota 视图的范围工厂

iota 视图也可以 (通常应该) 使用范围工厂创建:

```
std::views::iota(val)
std::views::iota(val, endVal)
```

例如:

```
for (auto val : std::views::iota(1, 100)) { // iterate over values 1, 2, ... 99
  std::cout << val << '\n'; // print these values
}
```

下面是使用 iota 视图的完整示例程序:

ranges/iotaview.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <ranges>

void print(std::ranges::input_range auto&& coll)
{
  int num = 0;
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
    if (++num > 30) { // print only up to 30 values:
      std::cout << "...";
      break;
    }
  }
  std::cout << '\n';
}

int main()
{
  std::ranges::iota_view<int> iv0; // 0 1 2 3 ...
  print(iv0);

  std::ranges::iota_view iv1{-2}; // -2 -1 0 1 ...
```

```
  print(iv1);

  std::ranges::iota_view iv2{10, 20}; // 10 11 12 ... 19
  print(iv2);

  auto iv3 = std::views::iota(1); // -2 -1 0 1 ...
  print(iv3);

  auto iv4 = std::views::iota('a', 'z'+1); // a b c ... z
  print(iv4);

  std::vector coll{0, 8, 15, 47, 11};
  for (auto p : std::views::iota(coll.begin(), coll.end())) { // sequence of
↪  iterators
    std::cout << *p << ' '; // 0 8 15 47 11
  }
  std::cout << '\n';
}
```

该程序有以下输出:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
↪  ...
-2 -1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
↪  ...
10 11 12 13 14 15 16 17 18 19
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
↪  ...
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 8 15 47 11
```

请注意，从'a' 到'z' 的输出取决于平台的字符集。

iota 视图的特殊特征

迭代器持有当前值，所以 iota 视图是租借范围 (其迭代器的生命周期不依赖于视图)。若 iota 视图有限，并且结束值与开始值具有相同的类型，则该视图是一个通用范围。

若 iota 视图无限大，或者结束的类型不同于开始的类型，则不是通用范围，可能必须使用一个通用视图来协调开始迭代器和哨兵 (结束迭代器) 的类型。

iota 视图的接口

“类 std::ranges::iota_view<> 操作”表列出了 iota 视图的 API。

操作	效果
ioat_view<Type>v	创建从默认值 Type 开始的无限序列
iota_view v{begVal}	创建一个以 begVal 开头的无限序列
iota_view v{begVal, endVal}	创建一个从 begVal 开始直到 endVal 之前的值的序列
iota_view v{beg, end}	启用子视图的辅助构造函数
v.begin()	生成 begin 迭代器 (指向起始值)
v.end()	生成哨兵 (end 迭代器)，若无限制，则为 std::unreachable_sentinel
v.empty()	表示 v 是否为空
if (v)	若 v 不为空，则为 true
v.size()	生成元素的数量 (若有限且可计算，则可用)
v.front()	生成第一个元素
v.back()	生成最后一个元素 (若有限且通用，则可用)
v[idx]	生成的第 n 个元素

表 8.8 类 std::ranges::iota_view<> 操作

声明 iota 视图时:

```
std::ranges::iota_view v1{1, 100}; // range with values: 1, 2, ... 99
```

v1 的类型由 std::ranges::iota_view<int, int> 推导而出，会创建提供基本 API 的对象，并使用 begin() 和 end() 得到这些值的迭代器。迭代器在内部存储当前值，并在迭代器加 1 时递增当前值:

```
std::ranges::iota_view v1{1, 100}; // range with values: 1, 2, ... 99
auto pos = v1.begin(); // initialize iterator with value 1
std::cout << *pos; // print current value (here, value 1)
++pos; // increment to next value (i.e., increment value to 2)
std::cout << *pos; // print current value (here, value 2)
```

因此，值的类型必须支持自增操作符，所以 bool 或 std::string 这样的类型就不可能了。

注意，此迭代器的类型取决于 iota_view 的实现，所以在使用时必须使用 auto，或用 std::ranges::iterator_t<> 也可以声明 pos:

```
std::ranges::iterator_t<decltype(v1)> pos = v1.begin();
```

至于迭代器的范围，值的范围是半开范围，不包括结束值。要包含 end 值，需要对 end 加 1:

```
std::ranges::iota_view letters{'a', 'z'+1}; // values: 'a', 'b', ... 'z'
for (auto c : letters) {
  std::cout << c << ' '; // print these values/letters
}
```

这个循环的输出取决于字符集，只有当小写字母具有连续的值 (如 ASCII 或 ISO-Latin-1 或 UTF-8 的情况) 时，视图才只迭代小写字符。通过使用 char8_t，可以确保这是可移植的 UTF-8 字符:

```
std::ranges::iota_view letters{u8'a', u8'z'+1}; // UTF-8 values from a to z
```

若没有传递 end 值，视图将是无限的，并生成一个无限长的值序列:

```
std::ranges::iota_view v2{10L}; // unlimited range with values: 10L, 11L, 12L, ...
std::ranges::iota_view<int> v3; // unlimited range with values: 0, 1, 2, ...
```

v3 的类型是 std::ranges::iota_view<int, std::unreachable_sentinel_t>，所以 end() 会生成 std::unreachable_sentinel。

无限视图是无尽的。当迭代器表示最大值并迭代到下一个值时，调用自增操作符，这在形式上是未定义行为 (实践中，执行通常会溢出，以便下一个值是值类型的最小值)。若视图有一个永远不匹配的结束值，其行为是一样的。

iota 视图对指针和迭代器进行迭代

可以用迭代器或指针初始化 iota 视图，可使用第一个值作为 begin，最后一个值作为 end，但是元素是迭代器/指针，而非值。

可以使用它来处理指向范围中所有元素的迭代器。

```
std::list<int> coll{2, 4, 6, 8, 10, 12, 14};
...
// pass iterator to each element to foo():
for (const auto& itorElem : std::views::iota(coll.begin(), coll.end())) {
  std::cout << *itorElem << '\n';
}
```

也可以使用这个特性初始化带有迭代器的容器，该迭代器指向集合 coll 的所有元素:

```
std::ranges::iota_view itors{coll.begin(), coll.end()};
std::vector<std::ranges::iterator_t<decltype(coll)>> refColl{itors.begin(),
  itors.end()};
```

注意，若跳过 refColl 元素类型的说明，则必须使用括号。否则，可以用两个迭代器元素初始化 vector:

```
std::vector refColl(itors.begin(), itors.end());
```

提供这个构造函数的主要原因是，为了支持创建 iota 视图子视图的泛型代码，比如丢弃视图:

```
// generic function to drop the first element from a view:
auto dropFirst = [] (auto v) {
             return decltype(v){++v.begin(), v.end()};
           };
std::ranges::iota_view v1{1, 9}; // iota view with elems from 1 to 8
auto v2 = dropFirst(v1); // iota view with elems from 2 to 8
```

成员函数 size() 和 [] 操作符的可用性取决于传递的范围内对操作符的支持情况。

8.4.2 单视图

类型:	std::ranges::single_view<>
内容:	只有一个元素的范围生成器
工厂:	std::views::single()
元素类型:	引用
类别:	连续的
是否是长度范围:	长度总为 1
是否是通用范围:	总是
是否是租借范围:	从不
缓存:	无
常量可迭代:	总是
传播常量性:	总是

类模板 std::ranges::single_view<> 是拥有一个元素的视图。除非值类型是 const，否则可以对值进行。

总体效果是，单视图的行为大致类似于一个元素的低成本集合，不分配堆上的内存。

单视图的主要用例是调用泛型代码来操作只有一个元素/值的视图:

```
std::ranges::single_view<int> v1{42}; // single view
for (auto val : v1) {
  ... // called once
}
```

这可以用于测试用例，或者用于代码必须在特定上下文中提供视图的情况，该视图必须只有一个元素。

单视图的范围工厂

单视图也可以 (通常应该) 通过范围工厂创建:

```
std::views::single(val)
```

例如：

```
for (auto val : std::views::single(42)) { // iterate over the single int value 42
  ... // called once
}
```

下面是使用单视图的完整示例程序:

ranges/singleview.cpp

```cpp
#include <iostream>
#include <string>
#include <ranges>

void print(std::ranges::input_range auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::ranges::single_view<double> sv0; // single view double 0.0
  std::ranges::single_view sv1{42}; // single_view<int> with int 42

  auto sv2 = std::views::single('x'); // single_view<char>
  auto sv3 = std::views::single("ok"); // single_view<const char*>
  std::ranges::single_view<std::string> sv4{"ok"}; // single view with string "ok"

  print(sv0);
  print(sv1);
  print(sv2);
  print(sv3);
  print(sv4);
  print(std::ranges::single_view{42});
}
```

该程序有以下输出:

```
0
42
x
ok
ok
42
```

单视图的特殊特性

单视图为以下概念建模:

- std::ranges::contiguous_range
- std::ranges::sized_range

视图始终是通用范围，而不是租借范围 (其迭代器的生命周期不依赖于视图)。

单视图的接口

“类 std::ranges::single_view<> 的操作”表列出了单视图的 API。

操作	效果
single_view<Type>v	创建具有 type 类型的一个值初始化元素的视图
single_view v{val}	创建具有其类型的一个值为 val 的元素的视图
single_view<Type>v{std::in_place, arg1, arg2, ...}	创建一个视图，其中一个元素初始化为 arg1, arg2，…
v.begin()	生成指向元素的原始指针
v.end()	生成指向元素后面位置的原始指针
v.empty()	总为 false
if(v)	总为 true
v.size()	生成 1
v.front()	生成对当前值的引用
v.back()	生成对当前值的引用
v[idx]	生成 idx 0 的当前值
r.data()	产生指向元素的原始指针

表 8.9 类 std::ranges::single_view<> 的操作

若调用构造函数时没有传递初始值，则单视图中的元素将初始化。所以要么使用其 Type 的默认构造函数，要么用 0、false 或 nullptr 初始化值。

要用需要多个初始化器的对象初始化单视图时，有两个选择:

- 传递初始化的对象:

```
std::ranges::single_view sv1{std::complex{1,1}};
auto sv2 = std::views::single(std::complex{1,1});
```

- 在参数 std::in_place 后传递初始值作为参数:

```
std::ranges::single_view<std::complex<int>> sv4{std::in_place, 1, 1};
```

注意，对于非 const 单视图，可以修改“element”的值:

```
std::ranges::single_view v2{42};
std::cout << v2.front() << '\n'; // prints 42
++v2.front(); // OK, modifies value of the view
std::cout << v2.front() << '\n'; // prints 43
```

可以通过将元素类型声明为 const 来防止这种情况:

```
std::ranges::single_view<const int> v3{42};
++v3.front(); // ERROR
```

接受值的构造函数复制或移动该值到视图中，所以视图不能引用初始值 (将元素类型声明为引用不会编译)。

因为 begin() 和 end() 会生成相同的值，所以视图总是一个通用范围。因为迭代器引用存储在视图中的值，以便能对其进行修改，所以单视图不是一个租借范围 (其迭代器的生命周期确实取决于视图)。

8.4.3 空视图

类型:	std::ranges::empty_view<>
内容:	无元素范围的生成器
工厂:	std::views::empty<>
元素类型:	引用
类别:	连续的
是否是长度范围:	总是 0
是否是通用范围:	总是
是否是租借范围:	总是
缓存:	无
常量可迭代:	总是
传播常量性:	从不

类模板 std::ranges::empty_view<> 是一个没有元素的视图，但必须指定元素类型。

空视图的主要用例是调用具有低成本视图的泛型代码，该视图没有元素，并且类型系统知道它没有元素:

```
std::ranges::empty_view<int> v1; // empty view
for (auto val : v1) {
  ... // never called
}
```

这可以用于测试用例，或者用于代码必须提供在特定上下文中，其中视图不能有元素。

空视图的范围工厂

空视图也可以 (通常应该) 用范围工厂创建，其为一个变量模板，所以用类型的模板参数声明它，但没有调用参数:

```
std::views::empty<type>
```

例如:

```
for (auto val : std::views::empty<int>) { // iterate over no int values
  ... // never called
}
```

下面是一个使用空视图的完整示例程序:

ranges/emptyview.cpp

```
#include <iostream>
#include <string>
#include <ranges>

void print(std::ranges::input_range auto&& coll)
{
  if (coll.empty()) {
    std::cout << "<empty>\n";
  }
  else {
    for (const auto& elem : coll) {
      std::cout << elem << ' ';
    }
    std::cout << '\n';
  }
}

int main()
{
  std::ranges::empty_view<double> ev0; // empty view of double
  auto ev1 = std::views::empty<int>; // empty view of int

  print(ev0);
  print(ev1);
}
```

该程序有以下输出:

```
<empty>
<empty>
```

空视图的特殊特性

空视图的行为大致类似于空 vector，还模拟了以下概念:

- std::ranges::contiguous_range(暗示 std::ranges::random_access_range)
- std::ranges::sized_range

begin() 和 end() 总是产生 nullptr，所以该范围既是通用范围，又是租借范围 (其迭代器的生命周期不依赖于视图)。

空视图的接口

“类 std::ranges::empty_view<> 的操作”表列出了空视图的 API。

读者们可能想知道为什么空视图提供 front()、back() 和 [] 操作符，其行为总是未定义的，调用它们总会触发致命的运行时错误。然而，泛型代码在调用 front()、back() 或 [] 操作符之前总是必须检查 (或知道) 是否存在元素，所以泛型代码永远不会调用这些成员函数。即使是空视图，这样的代码也可以编译。

操作	效果
empty_view<Type>v	创建不包含 Type 类型元素的视图
v.begin()	生成一个原始指针，指向用 nullptr 初始化的元素类型
v.end()	生成一个原始指针，指向用 nullptr 初始化的元素类型
v.empty()	生成 true
if (v)	总是 false
v.size()	生成 0
v.front()	总是未定义的行为 (致命的运行时错误)
v.back()	总是未定义的行为 (致命的运行时错误)
v[idx]	总是未定义的行为 (致命的运行时错误)
r.data	生成一个原始指针，指向用 nullptr 初始化的元素类型

表 8.10 类 std::ranges::empty_view<> 的操作

例如，这样传递一个空视图:

```
void foo(std::ranges::random_access_range auto&& rg)
{
  std::cout << "sortFirstLast(): \n";
  std::ranges::sort(rg);
  if (!std::ranges::empty(rg)) {
    std::cout << " first: " << rg.front() << '\n';
    std::cout << " last: " << rg.back() << '\n';
  }
}

foo(std::ranges::empty_view<int>{}); // OK
```

8.4.4 istream 视图

类型:	std::ranges::basic_istream_view<> std::ranges::istream_view<> std::ranges::wistream_view<>
内容:	从流中读取元素的范围的生成器
工厂:	std::views::istream<>()
元素类型:	引用

类别:	输入
是否是长度范围:	从不
是否是通用范围:	从不
是否是租借范围:	从不
缓存:	无
常量可迭代:	从不
传播常量性:	–

类模板 std::ranges::basic_istream_view<> 是一个从输入流 (如标准输入、文件或字符串流) 中读取元素的视图。

与通常的流类型一样，该类型是字符类型的泛型，并提供 char 和 wchar_t 的特化:

- 类模板 std::ranges::istream_view<> 是一个视图，使用 char 类型的字符从输入流中读取元素。
- 类模板 std::ranges::wistream_view<> 是一个视图，使用 wchar_t 类型的字符从输入流中读取元素。

例如:

```
std::istringstream myStrm{"0 1 2 3 4"};

for (const auto& elem : std::ranges::istream_view<int>{myStrm}) {
  std::cout << elem << '\n';
}
```

istream 视图的范围工厂

Istream 视图也可以 (通常应该) 使用范围工厂创建，工厂使用传入范围的字符类型将其参数传递给 std::ranges::basic_istream_view 构造函数:

```
std::views::istream<Type>(rg)
```

例如:

```
std::istringstream myStrm{"0 1 2 3 4"};

for (const auto& elem : std::views::istream<int>(myStrm)) {
  std::cout << elem << '\n';
}
```

或:

```
std::wistringstream mywstream{L"1.1 2.2 3.3 4.4"};
auto vw = std::views::istream<double>(mywstream);
```

vw 的初始化相当于:

```
auto vw2 = std::ranges::basic_istream_view<double, wchar_t>{myStrm};
auto vw3 = std::ranges::wistream_view<double>{myStrm};
```

下面是使用 istream 视图的完整示例程序:

ranges/istreamview.cpp

```
#include <iostream>
#include <string>
#include <sstream>
#include <ranges>

void print(std::ranges::input_range auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << " ";
  }
  std::cout << '\n';
}

int main()
{
  std::string s{"2 4 6 8 Motorway 1977 by Tom Robinson"};

  std::istringstream mystream1{s};
  std::ranges::istream_view<std::string> vs{mystream1};
  print(vs);

  std::istringstream mystream2{s};
  auto vi = std::views::istream<int>(mystream2);
  print(vi);
}
```

该程序有以下输出:

```
2 4 6 8 Motorway 1977 by Tom Robinson
2 4 6 8
```

用字符串 s 初始化的字符串流上迭代两次:

- istream 视图 vs 遍历从输入字符串流 mystream1 读取的所有字符串，输出 s 的所有子字符串。
- istream 视图 vi 遍历从输入字符串流 mystream2 中读取的所有整型数，读数结束时，到达 Motorway。

istream 视图和 const

不能迭代 const istream 视图:

```
void printElems(const auto& coll) {
  for (const auto elem& e : coll) {
    std::cout << elem << '\n';
  }
}

std::istringstream myStrm{"0 1 2 3 4"};

printElems(std::views::istream<int>(myStrm)); // ERROR
```

问题是 begin() 仅为非 const istream 视图提供，因为值分两步处理 (begin()/++ 和解引用操作符)，同时值存储在视图中。

下面的例子演示了修改视图的原因，通过使用底层的方法来迭代视图的“元素”，从 istream 视图中读取字符串:

```
std::istringstream myStrm{"stream with very-very-very-long words"};

auto v = std::views::istream<std::string>(myStrm);

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

代码有以下输出:

```
stream
with
very-very-very-long
words
```

当迭代器 pos 遍历视图 va 时，使用 begin() 和 ++ 从输入流 myStrm 中读取字符串。这些字符串必须存储在内部，以便可以通过解引用操作符使用。若将字符串存储在迭代器中，迭代器可能必须为该字符串保存内存，复制迭代器必须复制该内存，复制迭代器的代价不应该太高，所以视图将字符串存储在视图中，其效果可在迭代时修改视图。

因此，必须在泛型代码中使用通用/转发引用来支持此视图:

```
void printElems(auto&& coll) {
  ...
}
```

istream 视图的接口

“类 std::ranges::basic_istream_view<> 的操作”表列出了 istream 视图的 API。

操作	效果
istream_view<Type>v{strm}	Creates an istream view of type Type reading from strm
v.begin()	读取第一个值并生成 begin 迭代器
v.end()	产生 std::default_sentinel 作为结束迭代器

表 8.11 类 std::ranges::basic_istream_view<> 的操作

所能做的就是使用迭代器逐值读取，直到到达流的末尾。没有提供其他成员函数 (如 empty() 或 front())。

C++20 标准中 istream 视图的原始规范与其他视图有一些不一致，这些不一致后来修复了 (参见http://wg21.link/p2432)。通过这个修复，有了当前的行为:

- 可以完全指定所有模板参数:

```
std::ranges::basic_istream_view<int, char, std::char_traits<char>> v1{myStrm};
```

- 最后一个模板参数有一个默认值，所以可以使用:

```
std::ranges::basic_istream_view<int, char> v2{myStrm}; // OK
```

- 但不能跳过太多参数:

```
std::ranges::basic_istream_view<int> v3{myStrm}; // ERROR
```

- istream 视图遵循通常的约定，即为没有 basic_前缀的 char 和 wchar_t 流有特殊类型:

```
std::ranges::istream_view<int> v4{myStrm}; // OK for char streams

std::wistringstream mywstream{L"0 1 2 3 4"};
std::ranges::wistream_view<int> v5{mywstream}; // OK for wchar_t streams
```

- 并提供了相应的范围工厂:

```
auto v6 = std::views::istream<int>(myStrm); // OK
```

8.4.5 字符视图

类型:	std::basic_string_view<> std::string_view std::u8string_view std::u16string_view std::u32string_view std::wstring_view
内容:	字符序列中的所有字符
工厂:	–
元素类型:	const 引用

类别:	连续
是否是长度范围:	总是
是否是通用范围:	总是
是否是租借范围:	总是
缓存:	无
常量可迭代:	总是
传播常量性:	元素总是 const

类模板 std::basic_string_view<>，及其特化 std::string_view、std::u16string_view、std::u32string_view 和 std::wstring_view 是 C++17 标准库中仅有的视图类型，C++20 为 UTF-8 字符增加了特化:std::u8strng_view。

字符串视图不遵循视图具有的几个约定:

- 视图定义在命名空间 std 中 (而不是 std::ranges)。
- 视图有自己的头文件 <string_view>。
- 视图没有范围适配器/工厂来创建视图对象。
- 视图只提供对元素/字符的读访问。
- 视图提供了 cbegin() 和 cend() 成员函数。
- 视图不支持 bool 转换。

例如:

```
for (char c : std::string_view{"hello"}) {
  std::cout << c << ' ';
}
std::cout << '\n';
```

这个循环有以下输出:

```
h e l l o
```

字符视图的特殊特性

迭代器不引用这个视图。相反，其引用底层字符序列，字符串视图是一个租借范围，但当底层字符序列不再存在时，迭代器仍可以悬空。

字符串视图特定于视图的接口

“类 std::basic_string_view<> 视图的操作”表列出了字符串视图 API 中与视图相关的部分。

操作	效果
string_view sv	创建一个没有元素的字符串视图
string_view sv{s}	使用 s 创建一个字符串视图
v.begin()	生成一个原始指针，指向用 nullptr 初始化的元素类型
v.end()	生成一个原始指针，指向用 nullptr 初始化的元素类型
sv.empty()	生成的 sv 是否为空
sv.size()	生成字符的数量
sv.front()	生成的第一个字符
sv.back()	生成的最后一个字符
sv[idx]	生成的第 n 个字符
sv.data()	产生指向字符或 nullptr 的原始指针
...	为只读字符串提供的其他几个操作

表 8.12 类 std::basic_string_view<> 视图的操作

除了通常的视图接口之外，该类型还提供了所有字符串只读操作的完整 API。

要了解更多细节，请查看我的另一本书《C++17-完整指南》(参见http://www.cppstd17.com)。

8.4.6 span

类型:	std::span<>
内容:	连续存储器中一个范围的所有元素
工厂:	std::views::counted()
元素类型:	引用
要求:	连续的范围
类别:	连续
是否是长度范围:	Always
是否是通用范围:	总是
是否是租借范围:	总是
缓存:	无
常量可迭代:	总是
传播常量性:	从不

类模板 std::span<> 是存储在连续内存中的元素序列的视图，其在单独的章节中有详细的描述。这里，仅将 span 的属性作为视图呈现。

span 的主要优点是，支持在引用范围的中间或末尾引用 n 个元素的子范围。例如:

```
1 std::vector<std::string> vec{"New York", "Tokyo", "Rio", "Berlin", "Sydney"};
2
```

```
// sort the three elements in the middle:
std::ranges::sort(std::span{vec}.subspan(1, 3));

// print last three elements:
print(std::span{vec}.last(3));
```

这个例子将在后面关于 span 的章节中讨论。

span 不遵循视图通常具有的几个约定:

- 视图定义在命名空间 std 中，而不是 std::ranges 中。
- 视图有自己的头文件 。
- 视图不支持 bool 转换。

span 的范围工厂

没有范围工厂用于从 begin(迭代器) 和 end(哨兵) 初始化 span，但有范围工厂可以使用 begin 和 count 创建一个 span:

```
std::views::counted(beg, sz)
```

std::views::counted() 创建一个动态大小的 span，从迭代器 beg 开始，用连续范围的前 sz 个元素初始化 (对于非连续范围，counted() 创建子范围)。

例如:

```
std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9};

auto v = std::views::counted(vec.begin()+1, 3); // span with 2nd to 4th elem of vec
```

当 std::views::counted() 创建一个 span 时，span 具有动态范围 (其大小可以变化)。

span 的特点

迭代器不指向 span。相反，其引用底层元素，所以 span 是一个租借范围。但当底层字符序列不再存在时，迭代器仍然可以悬空。

span 的特定视图接口

“类 std::span<> 的视图操作”表列出了 span 的 API 中与视图相关的部分。

操作	效果
span sp	创建一个没有元素的 span
span sp{s}	为 s 创建一个 span 视图
sp.begin()	生成 begin 迭代器

sp.end()	生成哨兵 (end 迭代器)
sp.empty()	生成的 sv 是否为空
sp.size()	生成字符的数量
sp.front()	生成的第一个字符
sp.back()	生成的最后一个字符
sp[idx]	生成的第 n 个字符
sp.data()	生成指向字符或 nullptr 的原始指针
...	请参阅关于 span 操作的部分

表 8.13 类 std::span<> 的视图操作

有关更多详细信息，请参阅有关 span 操作的部分。

8.5. 过滤视图

本节讨论过滤给定范围或视图元素的所有视图。

8.5.1 获取视图

类型:	std::ranges::take_view<>
内容:	范围的第一个 (最多)num 个元素
适配器:	std::views::take()
元素类型:	与传入的类型相同 range
要求:	至少为输入范围
类别:	和传入一样
是否是长度范围:	若为长度范围
是否是通用范围:	若为长度随机访问范围
是否是租借范围:	若是租借视图或左值非视图
缓存:	无
常量可迭代:	若在可迭代范围内
传播常量性:	只有在右值范围内

类模板 std::ranges::take_view<> 定义了一个引用，传入范围的前 num 个元素的视图。若传递的范围没有足够的元素，则视图引用所有元素。

例如:

```
std::vector<int> rg{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
...
for (const auto& elem : std::ranges::take_view{rg, 5}) {
  std::cout << elem << ' ';
}
```

循环会输出：

```
1 2 3 4 5
```

获取视图的范围适配器

获取视图也可以 (通常应该) 使用范围适配器创建:

```
std::views::take(rg, n)
```

例如:

```
for (const auto& elem : std::views::take(rg, 5)) {
  std::cout << elem << ' ';
}
```

或:

```
for (const auto& elem : rg | std::views::take(5)) {
  std::cout << elem << ' ';
}
```

适配器可能并不总是产生 take_view:

- 若传递一个 empty_view，则只返回该视图。
- 若传递了一定大小的随机访问范围，可以初始化相同类型的范围，其中 begin() + num，则返回这样的范围 (这适用于子范围，iota 视图，字符串视图和 span)。

例如:

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

// using constructors:
std::ranges::take_view tv1{vec, 5}; // take view of ref view of vector
std::ranges::take_view tv2{std::vector<int>{}, 5}; // take view of owing view of
↪  vector
std::ranges::take_view tv3{std::views::iota(1,9), 5}; // take view of iota view

// using adaptors:
auto tv4 = std::views::take(vec, 5); // take view of ref view of vector
auto tv5 = std::views::take(std::vector<int>{}, 5); // take view of owning view of
↪  vector
auto tv6 = std::views::take(std::views::iota(1,9), 5); // pure iota view
```

获取视图在内部存储传递的范围 (可选择以与 all() 相同的方式转换为视图)，所以其只有在传递的范围有效的情况下才有效 (除非传递了右值，则在内部使用了归属视图)。

迭代器是传递的范围的迭代器，若是一个有长度的随机访问范围，或者是一个对它进行计数的迭代器和一个默认的哨兵，所以只有在传递了一定长度的随机访问范围时，该范围才是通用的。

下面是使用获取视图的完整示例:

ranges/takeview.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <ranges>

void print(std::ranges::input_range auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector coll{1, 2, 3, 4, 1, 2, 3, 4, 1};

  print(coll); // 1 2 3 4 1 2 3 4 1
  print(std::ranges::take_view{coll, 5}); // 1 2 3 4 1
  print(coll | std::views::take(5)); // 1 2 3 4 1
}
```

该程序有以下输出:

```
1 2 3 4 1 2 3 4 1
1 2 3 4 1
1 2 3 4 1
```

获取视图的特点

只有当底层范围是长度范围和通用范围时，获取视图才通用 (迭代器和哨兵具有相同的类型)。为了协调类型，可能必须使用通用视图。

获取视图的接口

“类 std::ranges::take_view<> 的操作”表列出了获取视图的 API。

操作	效果
take_view r{}	创建一个引用默认构造范围的 take_view

take_view r{rg, num}	创建一个引用范围 rg 前 num 个元素的 take_view
r.begin()	生成 begin 迭代器
r.end()	生成哨兵 (end 迭代器)
r.empty()	生成的 r 是否为空 (若范围支持则可用)
if (r)	若 r 不为空则为 true(若定义了 empty() 则可用)
r.size()	生成元素的数量 (若引用了一个长度范围，则可用)
r.front()	生成的第一个元素 (若转发可用)
r.back()	生成的最后一个元素 (若双向且通用，则可用)
r[idx]	生成的第 n 个元素 (若随机访问可用)
r.data()	生成一个指向元素内存的原始指针 (若元素在连续内存中可用)
r.base()	生成对 r 所引用或拥有的范围的引用

表 8.14 类 std::ranges::take_view<> 的操

8.5.2 获取段视图

类型:	std::ranges::take_while_view<>
内容:	范围中与谓词匹配的所有前导元素
适配器:	std::views::take_while()
元素类型:	与传递的范围类型相同
要求:	至少为输入范围
类别:	和传递的一样
是否是长度范围:	从不
是否是通用范围:	从不
是否是租借范围:	从不
缓存:	无
常量可迭代:	若在常量可迭代的范围和谓词适用于常量的值
传播常量性:	只有在右值范围内

类模板 std::ranges::take_while_view<> 定义了一个视图，该视图引用传递的范围中匹配某个谓词的所有前置元素:

```
std::vector<int> rg{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
...
for (const auto& elem : std::ranges::take_while_view{rg, [](auto x) {
                    return x % 3 != 0;
                }}) {
  std::cout << elem << ' ';
}
```

循环的输出为:

```
1 2
```

获取段视图的范围适配器

获取段视图也可以 (通常应该) 使用范围适配器创建，适配器只需将其参数传递给 std::ranges::take_while_view 构造函数:

```
1 std::views::take_while(rg, pred)
```

例如:

```
1 for (const auto& elem : std::views::take_while(rg, [](auto x) {
2             return x % 3 != 0;
3           })) {
4   std::cout << elem << ' ';
5 }
```

或:

```
1 for (const auto& elem : rg | std::views::take_while([](auto x) {
2             return x % 3 != 0;
3           })) {
4   std::cout << elem << ' ';
5 }
```

传递的谓词必须是满足 std::predicate 概念的可调用对象。这暗示了 std::regular_invocable 的概念，所以谓词永远不应该修改底层范围的传递值，但不修改值是语义约束，所以在编译时不能总是检查这一点，所以至少应该将谓词声明为按值或按 const 引用接受实参。

获取段视图在内部存储传递的范围 (可选择以与 all() 相同的方式转换为视图)，所以只有在传递的范围有效的情况下才有效 (除非传递了右值，则在内部使用了归属视图)。

迭代器只是传递的范围和一个特殊的内部哨兵类型的迭代器。

下面是使用获取段视图的完整示例:

ranges/takewhileview.cpp

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 #include <ranges>
5
6 void print(std::ranges::input_range auto&& coll)
7 {
8   for (const auto& elem : coll) {
9     std::cout << elem << ' ';
```

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

int main()
{
  std::vector coll{1, 2, 3, 4, 1, 2, 3, 4, 1};
  print(coll); // 1 2 3 4 1 2 3 4 1
  auto less4 = [] (auto v) { return v < 4; };
  print(std::ranges::take_while_view{coll, less4}); // 1 2 3
  print(coll | std::views::take_while(less4)); // 1 2 3
}
```

该程序有以下输出:

```
1 2 3 4 1 2 3 4 1
1 2 3
1 2 3
```

获取段视图的接口

“类 std::ranges::take_while_view<> 的操作”表列出了获取段视图的 API。

操作	效果
take_while_view r{}	创建一个引用默认构造范围的 take_while_view
take_while_view r{rg. pred}	创建一个 take_while_view，引用 pred 为真的 rg 范围的前导元素
r.begin()	生成 begin 迭代器
r.end()	生成哨兵 (end 迭代器)
r.empty()	生成的 r 是否为空 (若范围支持，则可用)
if (r)	若 r 不为空，则为 true(若定义了 empty()，则可用)
r.size()	生成元素的数量 (若引用了一个长度范围，则可用)
r.front()	生成第一个元素 (若转发可用)
r[idx]	生成第 n 个元素 (若为随机访问，则可用)
r.data()	生成指向元素内存的原始指针 (若元素在连续的内存中，则可用)
r.base()	生成对 r 所引用或拥有的范围的引用
r.pred()	生成对谓词的引用

表 8.15 类 std::ranges::take_while_view<> 的操作

8.5.3 丢弃视图

类型:	std::ranges::drop_view<>
内容:	除第 num 个元素外的所有元素
适配器:	std::views::drop()
元素类型:	与传递的范围类型相同
要求:	至少为输入范围
类别:	和传递的一样
是否是长度范围:	若在长度范围内
是否是通用范围:	若在通用范围内
是否是租借范围:	若为租借视图或左值非视图
缓存:	若没有随机访问范围或没有长度范围，则缓存 begin()
常量可迭代:	若是一个可常量迭代范围，该范围提供随机访问并确定长度
传播常量性:	只有在右值范围内

类模板 std::ranges::drop_view<> 定义了一个视图，该视图引用传递的范围中除第 num 个元素外的所有元素，产生与获取视图相反的元素。

例如:

```
std::vector<int> rg{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
...
for (const auto& elem : std::ranges::drop_view{rg, 5}) {
  std::cout << elem << ' ';
}
```

循环输出:

```
6 7 8 9 10 11 12 13
```

丢弃视图的范围适配器

丢弃视图也可以 (通常应该) 使用范围适配器创建:

```
std::views::drop(rg, n)
```

例如:

```
for (const auto& elem : std::views::drop(rg, 5)) {
  std::cout << elem << ' ';
}
```

或:

```cpp
for (const auto& elem : rg | std::views::drop(5)) {
  std::cout << elem << ' ';
}
```

注意，适配器可能并不总是产生 drop_view:

- 若传递了一个空视图，则直接返回空视图。
- 若传递了一定长度的随机访问范围，可以初始化相同类型的范围，其中 begin() + num，则返回这样的范围 (这适用于子范围，iota 视图，字符串视图和 span)。

例如:

```cpp
std::vector<int> vec;

// using constructors:
std::ranges::drop_view dv1{vec, 5}; // drop view of ref view of vector
std::ranges::drop_view dv2{std::vector<int>{}, 5}; // drop view of owing view of
↪  vector
std::ranges::drop_view dv3{std::views::iota(1,10), 5}; // drop view of iota view

// using adaptors:
auto dv4 = std::views::drop(vec, 5); // drop view of ref view of vector
auto dv5 = std::views::drop(std::vector<int>{}, 5); // drop view of owing view of
↪  vector
auto dv6 = std::views::drop(std::views::iota(1,10), 5); // pure iota view
```

丢弃视图在内部存储传递的范围 (可选择以与 all() 相同的方式转换为视图)，所以只有在传递的范围有效的情况下才有效 (除非传递了右值，则在内部使用了归属视图)。

begin 迭代器在第一次调用 begin() 时初始化并缓存。在没有随机访问的范围上，这需要线性时间，所以重用丢弃视图比重新创建要好。

下面是一个使用丢弃视图的完整示例:

ranges/dropview.cpp

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <ranges>

void print(std::ranges::input_range auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
```

```
{
  std::vector coll{1, 2, 3, 4, 1, 2, 3, 4, 1};

  print(coll); // 1 2 3 4 1 2 3 4 1
  print(std::ranges::drop_view{coll, 5}); // 2 3 4 1
  print(coll | std::views::drop(5)); // 2 3 4 1
}
```

该程序有以下输出:

```
1 2 3 4 1 2 3 4 1
2 3 4 1
2 3 4 1
```

丢弃视图和缓存

为了获得更好的性能 (常数复杂度)，丢弃视图会在视图中缓存 begin() 的结果 (除非范围只是一个输入范围)，所以对丢弃视图元素的第一次迭代要比以后的迭代更慢。

出于这个原因，最好初始化一次丢弃视图，并使用它两次:

```
// better:
auto v1 = coll | std::views::drop(5);
check(v1);
process(v1);
```

而非初始化并使用两次:

```
// worse:
check(coll | std::views::drop(5));
process(coll | std::views::drop(5));
```

注意，对于具有随机访问的范围 (例如，数组，vector 和队列)，缓存的开始偏移量与视图一起复制。否则，不会复制缓存的起始位置。

当过滤器视图引用的范围被修改时，此缓存可能会产生功能性后果。

例如:

ranges/dropcache.cpp

```
#include <iostream>
#include <vector>
#include <list>
#include <ranges>

void print(auto&& coll)
{
  for (const auto& elem : coll) {
```

```
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector vec{1, 2, 3, 4};
  std::list lst{1, 2, 3, 4};

  auto vVec = vec | std::views::drop(2);
  auto vLst = lst | std::views::drop(2);

  // insert a new element at the front (=> 0 1 2 3 4)
  vec.insert(vec.begin(), 0);
  lst.insert(lst.begin(), 0);

  print(vVec); // OK: 2 3 4
  print(vLst); // OK: 2 3 4

  // insert more elements at the front (=> 98 99 0 -1 0 1 2 3 4)
  vec.insert(vec.begin(), {98, 99, 0, -1});
  lst.insert(lst.begin(), {98, 99, 0, -1});

  print(vVec); // OK: 0 -1 0 1 2 3 4
  print(vLst); // OOPS: 2 3 4

  // creating a copy heals:
  auto vVec2 = vVec;
  auto vLst2 = vLst;

  print(vVec2); // OK: 0 -1 0 1 2 3 4
  print(vLst2); // OK: 0 -1 0 1 2 3 4
}
```

从形式上讲，将无效视图复制到 vector 会创建未定义的行为，因为 C++ 标准没有指定如何进行缓存。由于 vector 的重新分配会使所有迭代器失效，因此缓存的迭代器将失效。对于随机访问范围，视图通常缓存偏移量，而不是迭代器。所以视图仍然有效，因为它仍然包含不包含前两个元素的范围。

根据经验，**不要使用在底层范围被修改后调用 begin() 的丢弃视图。**

丢弃视图和 const

注意，不能总是迭代 const 丢弃视图。实际上，所引用的范围必须是一个可随机访问范围和一定长度的范围。

例如:

```
void printElems(const auto& coll) {
  for (const auto elem& e : coll) {
    std::cout << elem << '\n';
  }
}

std::vector vec{1, 2, 3, 4, 5};
std::list lst{1, 2, 3, 4, 5};

printElems(vec | std::views::drop(3)); // OK
printElems(lst | std::views::drop(3)); // ERROR
```

要在泛型代码中支持这个视图，必须使用通用/转发引用:

```
void printElems(auto&& coll) {
  ...
}

std::list lst{1, 2, 3, 4, 5};

printElems(lst | std::views::drop(3)); // OK
```

丢弃视图的接口

“类 std::ranges::drop_view<> 的操作”表列出了丢弃视图的 API。

操作	效果
drop_view r{}	创建一个 drop_view，引用一个默认构造的范围
drop_view r{rg, num}	创建一个 drop_view，引用 rg 范围内除前 num 个元素外的所有元素
r.begin()	生成 begin 迭代器
r.end()	生成哨兵 (end 迭代器)
r.empty()	生成的 r 是否为空 (若范围支持，则可用)
if (r)	若 r 不为空则为 true(若定义了 empty()，则可用)
r.size()	生成元素的数量 (若引用了一个大小长度范围，则可用)
r.front()	生成的第一个元素 (若转发可用)
r.back()	生成的最后一个元素 (若双向且通用，则可用)
r[idx]	生成的第 n 个元素 (若随机访问，则可用)
r.data()	生成一个指向元素内存的原始指针 (若元素在连续内存中，则可用)。
r.base()	生成对 r 所引用或拥有的范围的引用

表 8.16 类 std::ranges::drop_view<> 的操作

8.5.4 丢弃段视图

类型:	std::ranges::drop_while_view<>
内容:	范围中除前导元素外与谓词匹配的所有元素
适配器:	std::views::drop_while()
元素类型:	与传递的范围类型相同
要求:	至少为输入范围
类别:	和传递的一样
是否是长度范围:	若传递了普通的随机访问范围
是否是通用范围:	若在常规范围内
是否是租借范围:	若是租借视图或左值非视图
缓存:	总是缓存 begin()
常量可迭代:	Never
传播常量性:	–(若为 const，不能调用 begin())

类模板 std::ranges::drop_while_view<> 定义了一个视图，该视图跳过与某个谓词匹配的传递范围的所有前导元素，生成了与获取段视图相反的元素:

```
std::vector<int> rg{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
...
for (const auto& elem : std::ranges::drop_while_view{rg, [](auto x) {
            return x % 3 != 0;
        }}) {
  std::cout << elem << ' ';
}
```

循环输出为:

```
3 4 5 6 7 8 9 10 11 12 13
```

丢弃段视图的范围适配器

丢弃段视图也可以 (通常应该) 使用范围适配器创建，适配器只需将其参数传递给 std::ranges::drop_while_view 构造函数:

```
std::views::drop_while(rg, pred)
```

例如:

```
for (const auto& elem : std::views::drop_while(rg, [](auto x) {
            return x % 3 != 0;
          })) {
  std::cout << elem << ' ';
}
```

或:

```
for (const auto& elem : rg | std::views::drop_while([](auto x) {
            return x % 3 != 0;
          })) {
  std::cout << elem << ' ';
}
```

传递的谓词必须是满足 std::predicate 概念的可调用对象。这暗示了 std::regular_invocable 的概念，谓词永远不应该修改底层范围的传递值，但不修改值是语义约束，所以在编译时不能总是检查这一点，所以至少应该将谓词声明为按值或按 const 引用接受实参。

丢弃段视图在内部存储传递的范围 (可选择以与 all() 相同的方式转换为视图)，所以只有在传递的范围有效的情况下才有效 (除非传递了右值，则在内部使用了归属视图)。

begin 迭代器在第一次调用 begin() 时初始化并缓存，这总是需要线性时间，所以重用丢弃段视图比重新创建要好。

下面是一个使用丢弃段视图的完整示例:

ranges/dropwhileview.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <ranges>

void print(std::ranges::input_range auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector coll{1, 2, 3, 4, 1, 2, 3, 4, 1};

  print(coll); // 1 2 3 4 1 2 3 4 1
  auto less4 = [] (auto v) { return v < 4; };
  print(std::ranges::drop_while_view{coll, less4}); // 4 1 2 3 4 1
  print(coll | std::views::drop_while(less4)); // 4 1 2 3 4 1
}
```

该程序有以下输出:

```
1 2 3 4 1 2 3 4 1
4 1 2 3 4 1
4 1 2 3 4 1
```

丢弃段视图和缓存

为了获得更好的性能 (常数复杂度)，丢弃段视图会在视图中缓存 begin() 的结果 (除非该范围只是一个输入范围)。所以对丢弃段视图元素的第一次迭代，比以后的迭代代价更慢。

出于这个原因，最好初始化一个过滤器视图，然后使用它两次:

```
// better:
auto v1 = coll | std::views::drop_while(myPredicate);
check(v1);
process(v1);
```

而非初始化并使用它两次:

```
// worse:
check(coll | std::views::drop_while(myPredicate));
process(coll | std::views::drop_while(myPredicate));
```

当修改过滤视图引用的范围时，缓存可能会产生功能性后果。例如:

ranges/dropwhilecache.cpp

```
#include <iostream>
#include <vector>
#include <list>
#include <ranges>

void print(auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector vec{1, 2, 3, 4};
  std::list lst{1, 2, 3, 4};

  auto lessThan2 = [](auto v){
    return v < 2;
  };

  auto vVec = vec | std::views::drop_while(lessThan2);
  auto vLst = lst | std::views::drop_while(lessThan2);

  // insert a new element at the front (=> 0 1 2 3 4)
  vec.insert(vec.begin(), 0);
```

```
  lst.insert(lst.begin(), 0);

  print(vVec); // OK: 2 3 4
  print(vLst); // OK: 2 3 4

  // insert more elements at the front (=> 0 98 99 -1 0 1 2 3 4)
  vec.insert(vec.begin(), {0, 98, 99, -1});
  lst.insert(lst.begin(), {0, 98, 99, -1});

  print(vVec); // OOPS: 99 -1 0 1 2 3 4
  print(vLst); // OOPS: 2 3 4

  // creating a copy heals (except with random access):
  auto vVec2 = vVec;
  auto vLst2 = vLst;

  print(vVec2); // OOPS: 99 -1 0 1 2 3 4
  print(vLst2); // OK: 98 99 -1 0 1 2 3 4
}
```

从形式上讲，将无效视图复制到 vector 会创建未定义的行为，因为 C++ 标准没有指定如何进行缓存。由于 vector 的重新分配会使所有迭代器失效，因此缓存的迭代器将失效。对于随机访问范围，视图通常缓存偏移量，而不是迭代器。因为它仍然包含一个有效范围，所以视图仍然有效，尽管开始不再适合谓词。

根据经验，**不要使用在底层范围被修改后调用 begin() 的丢弃段视图**。

丢弃段视图和 const

不能迭代 const 丢弃段视图。

例如:

```
void printElems(const auto& coll) {
  for (const auto elem& e : coll) {
    std::cout << elem << '\n';
  }
}

std::vector vec{1, 2, 3, 4, 5};

printElems(vec | std::views::drop_while( ... )); // ERROR
```

问题在于，begin() 仅为非 const 丢弃段视图提供，因为缓存迭代器会修改视图。

要在泛型代码中支持这个视图，必须使用通用/转发引用:

```
void printElems(auto&& coll) {
  ...
```

```
}

std::list lst{1, 2, 3, 4, 5};

printElems(vec | std::views::drop_while( ... )); // OK
```

丢弃段视图的接口

“类 std::ranges::drop_while_view<> 的操作”表列出了丢弃段视图的 API。

操作	效果
drop_while_view r{}	创建一个 drop_while_view，引用一个默认构造的范围
drop_while_view r{rg, pred}	创建一个 drop_while_view，引用 rg 的所有元素，除了 pred 为真的前导元素
r.begin()	生成 begin 迭代器
r.end()	生成哨兵 (end 迭代器)
r.empty()	生成的 r 是否为空 (若范围支持，则可用)
if (r)	若 r 不为空，则为 true(若定义了 empty()，则可用)
r.size()	生成元素的数量 (若引用了一个长度范围，则可用)
r.front()	生成的第一个元素 (若转发可用)
r.back()	生成的最后一个元素 (若果双向且通用，则可用)
r[idx]	生成的第 n 个元素 (若随机访问可用)
r.data()	生成生一个指向元素内存的原始指针 (若元素在连续内存中可用)。
r.base()	生成对 r 所引用或归属范围的引用
r.pred()	生成对谓词的引用

表 8.17 类 std::ranges::drop_while_view<> 的操作

8.5.5 过滤视图

类型:	std::ranges::filter_view<>
内容:	范围中与谓词匹配的所有元素
适配器:	std::views::filter()
元素类型:	与传递的范围类型相同
要求:	至少为输入范围
类别:	和传入的一样，但至多是双向的
是否是长度范围:	从不
是否是通用范围:	若在常规范围内
是否是租借范围:	从不
缓存:	总是缓存 begin()

常量可迭代:	从不
传播常量性:	– (若是 const，则不能使用 begin())

类模板 std::ranges::filter_view<> 定义了一个视图，该视图只迭代底层范围中与某个谓词匹配的元素。也就是说，过滤掉所有与谓词不匹配的元素。例如:

```
std::vector<int> rg{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
...
for (const auto& elem : std::ranges::filter_view{rg, [](auto x) {
               return x % 3 != 0;
           }}) {
  std::cout << elem << ' ';
}
```

循环输出:

```
1 2 4 5 7 8 10 11 13
```

过滤视图的范围适配器

过滤视图也可以 (通常应该) 使用范围适配器创建，适配器只需将其参数传递给 std::ranges::filter_view 构造函数:

```
std::views::filter(rg, pred)
```

例如:

```
for (const auto& elem : std::views::filter(rg, [](auto x) {
             return x % 3 != 0;
           })) {
  std::cout << elem << ' ';
}
```

或:

```
for (const auto& elem : rg | std::views::filter([](auto x) {
             return x % 3 != 0;
           })) {
  std::cout << elem << ' ';
}
```

传递的谓词必须是满足 std::predicate 概念的可调用对象。这表示了 std::regular_invocable 的概念，这意味着谓词永远不应该修改底层范围的传递值。但不修改值是语义约束，在编译时不能总是检查这一点，所以至少应该将谓词声明为按值或按 const 引用接受实参。

过滤视图是特殊的，开发者应该知道何时、何地以及如何使用它，以及使用它所产生的副作用。事实上，对管道的性能有很大的影响，并且有时会以令人惊讶的方式限制对元素的写访问。

因此，应该谨慎地使用过滤视图:

- 在管道中，应该尽可能早地获得它。
- 在使用过滤器之前，要警惕昂贵的转换。
- 当写访问破坏谓词时，不要将其用于对元素的写访问。

过滤视图在内部存储传递的范围 (可选择以与 all() 相同的方式转换为视图)，所以只有在传递的范围有效的情况下才有效 (除非传递了右值，则在内部使用了归属视图)。

begin 迭代器被初始化，通常在第一次调用 begin() 时进行缓存，这总是需要线性时间，所以重用过滤器视图比从头创建过滤器视图要好。

下面是使用过滤器视图的完整示例:

ranges/filterview.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <ranges>

void print(std::ranges::input_range auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector coll{1, 2, 3, 4, 1, 2, 3, 4, 1};

  print(coll); // 1 2 3 4 1 2 3 4 1
  auto less4 = [] (auto v) { return v < 4; };
  print(std::ranges::filter_view{coll, less4}); // 1 2 3 1 2 3 1
  print(coll | std::views::filter(less4)); // 1 2 3 1 2 3 1
}
```

该程序有以下输出:

```
1 2 3 4 1 2 3 4 1
1 2 3 1 2 3 1
1 2 3 1 2 3 1
```

过滤视图和缓存

为了获得更好的性能 (常数复杂度)，过滤器视图会在视图中缓存 begin() 的结果 (除非该范围只是一个输入范围)，所以过滤视图元素上的第一次迭代比以后的迭代更慢。

最好初始化一个过滤器视图，然后使用它两次:

```
// better:
auto v1 = coll | std::views::drop_while(myPredicate);
check(v1);
process(v1);
```

而非初始化并使用它两次:

```
// worse:
check(coll | std::views::drop_while(myPredicate));
process(coll | std::views::drop_while(myPredicate));
```

对于具有随机访问的范围 (例如，数组，vector 和队列)，缓存的开始偏移量与视图一起复制。否则，不复制缓存的开头。

当修改过滤视图引用的范围时，缓存可能会产生功能性后果。

例如:

ranges/filtercache.cpp

```
#include <iostream>
#include <vector>
#include <list>
#include <ranges>

void print(auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector vec{1, 2, 3, 4};
  std::list lst{1, 2, 3, 4};

  auto lessThan2 = [](auto v){
    return v < 2;
  };

  auto vVec = vec | std::views::filter(biggerThan2);
  auto vLst = lst | std::views::filter(biggerThan2);

  // insert a new element at the front (=> 0 1 2 3 4)
```

```
  vec.insert(vec.begin(), 0);
  lst.insert(lst.begin(), 0);

  print(vVec); // OK: 3 4
  print(vLst); // OK: 3 4

  // insert more elements at the front (=> 98 99 0 -1 0 1 2 3 4)
  vec.insert(vec.begin(), {98, 99, 0, -1});
  lst.insert(lst.begin(), {98, 99, 0, -1});

  print(vVec); // OOPS: -1 3 4
  print(vLst); // OOPS: 3 4

  // creating a copy heals (except with random access):
  auto vVec2 = vVec;
  auto vLst2 = vLst;

  print(vVec2); // OOPS: -1 3 4
  print(vLst2); // OK: 98 99 3 4
}
```

从形式上讲，将无效视图复制到 vector 会创建未定义的行为，因为 C++ 标准没有指定如何进行缓存。由于 vector 的重新分配会使所有迭代器失效，因此缓存的迭代器将失效。对于随机访问范围，视图通常缓存偏移量，而不是迭代器。因为它仍然包含一个有效的范围，所以视图仍然有效，尽管 begin 现在是第三个元素，而不管它是否适合过滤。

根据经验，**不要使用在底层范围被修改后调用 begin() 的过滤视图**。

修改元素时的过滤视图

使用过滤器视图时，对写访问有一个重要的附加限制: 必须确保修改后的值仍然满足传递给过滤器的谓词。

要理解为什么会出现这种情况，请考虑以下代码:

```
oid printElems(const auto& coll) {
  for (const auto elem& e : coll) {
    std::cout << elem << '\n';
  }
}

std::vector vec{1, 2, 3, 4, 5};

printElems(vec | std::views::drop_while( ... )); // ERROR
```

问题在于，begin() 仅为非 const 丢弃段视图提供，因为缓存迭代器会修改视图。

要在泛型代码中支持这个视图，必须使用通用/转发引用:

ranges/viewswrite.cpp

```cpp
#include <iostream>
#include <vector>
#include <ranges>
namespace vws = std::views;

void print(const auto& coll)
{
  std::cout << "coll: ";
  for (int i : coll) {
    std::cout << i << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector<int> coll{1, 4, 7, 10, 13, 16, 19, 22, 25};

  // view for all even elements of coll:
  auto isEven = [] (auto&& i) { return i % 2 == 0; };
  auto collEven = coll | vws::filter(isEven);

  print(coll);

  // modify even elements in coll:
  for (int& i : collEven) {
    std::cout << " increment " << i << '\n';
    i += 1; // ERROR: undefined behavior because filter predicate is broken
  }
  print(coll);

  // modify even elements in coll:
  for (int& i : collEven) {
    std::cout << " increment " << i << '\n';
    i += 1; // ERROR: undefined behavior because filter predicate is broken
  }
  print(coll);
}
```

对集合中的偶数元素迭代两次并自增，菜鸟开发者会假设得到以下输出:

```
coll: 1 4 7 10 13 16 19 22 25
coll: 1 5 7 11 13 17 19 23 25
coll: 1 5 7 11 13 17 19 23 25
```

但程序有以下输出:

```
coll: 1 4 7 10 13 16 19 22 25
coll: 1 5 7 11 13 17 19 23 25
coll: 1 6 7 11 13 17 19 23 25
```

第二次迭代中，再次增加前面的第一个偶数元素。为什么?

第一次使用确保只处理偶数元素的视图时，工作得很好。但视图缓存与谓词匹配的第一个元素的位置，以便 begin() 不必重新计算它。当我们访问那里的值时，过滤器不会再次应用谓词，因为它已经知道这是第一个匹配元素。当第二次迭代时，过滤器返回前一个元素。但对于所有其他元素，必须再次执行检查，因为它们现在都是奇数，所以过滤器没有找到更多的元素。

关于使用过滤器的一次写访问是否应该是格式良好的，即使它破坏了过滤器的谓词，也有一些讨论。因为修改不应违反过滤器谓词的要求，使一些非常合理的示例无效: 下面是其中之一:

```
for (auto& m : collOfMonsters | filter(isDead)) {
  m.resurrect(); // a shaman's doing, of course
}
```

这段代码通常可以编译并运行。但形式上，又有了未定义的行为，因为过滤器的谓词 (“怪物必须是死的”) 被破坏了。对 (死亡) 怪物的其他“修改”都是可能的 (例如，“烧”它们)。

要中断谓词，必须使用普通循环:

```
for (auto& m : collOfMonsters) {
  if (m.isDead()) {
    m.resurrect(); // a shaman's doing, of course
  }
}
```

过滤视图和 const

注意，不能在 const 过滤视图上迭代。

例如:

```
void printElems(const auto& coll) {
  for (const auto elem& e : coll) {
    std::cout << elem << '\n';
  }
}

std::vector vec{1, 2, 3, 4, 5};

printElems(vec | std::views::filter( ... )); // ERROR
```

问题在于，begin() 仅为非 const 过滤视图提供，因为缓存迭代器会修改视图。

要在泛型代码中支持这个视图，必须使用通用/转发引用:

```
void printElems(auto&& coll) {
  ...
}

std::list lst{1, 2, 3, 4, 5};

printElems(vec | std::views::filter( ... )); // OK
```

管道中的过滤视图

当在管道中使用过滤视图时，有几个问题需要考虑:

- 在管道中，应该尽可能早地获得它。
- 过滤器之前，要警惕昂贵的转换。
- 当在修改元素的同时使用过滤器时，确保在这些修改之后，过滤器的谓词仍然是满足的。详情见下文。

这样做的原因是，过滤器视图前面的视图和适配器可能必须多次评估元素，一次是决定它们是否通过过滤，一次是决定最终使用的值。

因此，像下面这样的管道:

```
rg | std::views::filter(pred) | std::views::transform(func)
```

要比下面的方式有更好的性能比

```
rg | std::views::transform(func) | std::views::filter(pred)
```

过滤视图的接口

“类 std::ranges::filter_view<> 的操作”表列出了过滤视图的 API。

过滤视图从不提供 size()、data() 或 [] 操作符，因其既不确定大小，也不提供随机访问。

操作	效果
filter_view r{}	创建一个引用默认构造范围的 filter_view
filter_view r{rg. pred}	创建一个 filter_view，它引用 pred 为 true 的所有元素或 rg
r.begin()	生成 begin 迭代器
r.end()	生成哨兵 (end 迭代器)
r.empty()	生成的 r 是否为空 (若范围支持，则可用)
if (r)	若 r 不为空，则为 true(若定义了 empty()，则可用)
r.front()	生成的第一个元素 (若转发可用)
r.back()	生成的最后一个元素 (若为双向且通用，则可用)

r.base()	生成指向 r 的引用范围
r.pred()	生成对谓词的引用

表 8.18 类 std::ranges::filter_view<> 的操作

8.6. 转换视图

本节讨论生成所遍历元素修改值的视图。

8.6.1 转换视图

类型:	std::ranges::transform_view<>
内容:	范围内所有元素的转换值
适配器:	std::views::transform()
元素类型:	转换的返回类型
要求:	至少为输入范围
类别:	与传递的相同，但至多是随机访问
是否是长度范围:	若在长度范围内
是否是通用范围:	若在通用范围内
是否是租借范围:	从不
缓存:	无
常量可迭代:	若有常量可迭代范围和转换作用于 const 值，则可用
传播常量性:	只有在右值范围内

类模板 std::ranges::transform_view<> 定义了一个视图，该视图在传递转换后生成底层范围内的所有元素。例如:

```
std::vector<int> rg{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
...
// print all elements squared:
for (const auto& elem : std::ranges::transform_view{rg, [](auto x) {
                return x * x;
            }}) {
  std::cout << elem << ' ';
}
```

循环输出:

```
1 4 9 16 25 36 49 64 81 100 121 144 169
```

转换视图的范围适配器

转换视图也可以 (通常应该) 使用范围适配器创建，适配器只是将其参数传递给 std::ranges::transform_view 构造函数。

```
std::views::transform(rg, func)
```

例如:

```
for (const auto& elem : std::views::transform(rg, [](auto x) {
            return x * x;
          })) {
  std::cout << elem << ' ';
}
```

或:

```
for (const auto& elem : rg | std::views::transform([](auto x) {
            x * x;
          })) {
  std::cout << elem << ' ';
}
```

下面是使用转换视图的完整示例:

ranges/transformview.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <ranges>
#include <cmath> // for std::sqrt()

void print(std::ranges::input_range auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector coll{1, 2, 3, 4, 1, 2, 3, 4, 1};

  print(coll); // 1 2 3 4 1
  auto sqrt = [] (auto v) { return std::sqrt(v); };
```

```
  print(std::ranges::transform_view{coll, sqrt}); // 1 1.41421 1.73205 2 1
  print(coll | std::views::transform(sqrt)); // 1 1.41421 1.73205 2 1
}
```

该程序有以下输出:

```
1 2 3 4 1 2 3 4 1
1 1.41421 1.73205 2 1 1.41421 1.73205 2 1
1 1.41421 1.73205 2 1 1.41421 1.73205 2 1
```

转换必须是满足 std::regular_invocable 概念的可调用对象，所以转换永远不应该修改传递的基础范围的值。但不修改值是语义约束，在编译时不能总是检查这一点，所以至少应该将可调用对象声明为通过值或 const 引用接受实参。

转换视图产生具有转换返回类型的值，所以在示例中，转换视图产生的元素类型为 double。

转换的返回类型，甚至可以是引用。例如:

ranges/transformref.cpp

```
#include <iostream>
#include <vector>
#include <utility>
#include <ranges>

void printPairs(const auto& collOfPairs)
{
  for (const auto& elem : collOfPairs) {
    std::cout << elem.first << '/' << elem.second << ' ';
  }
  std::cout << '\n';
}

int main()
{
  // initialize collection with pairs of int as elements:
  std::vector<std::pair<int,int>> coll{{1,9}, {9,1}, {2,2}, {4,1}, {2,7}};
  printPairs(coll);

  // function that yields the smaller of the two values in a pair:
  auto minMember = [] (std::pair<int,int>& elem) -> int& {
    return elem.second < elem.first ? elem.second : elem.first;
  };

  // increment the smaller of the two values in each pair element:
  for (auto&& member : coll | std::views::transform(minMember)) {
    ++member;
  }
  printPairs(coll);
}
```

该程序有以下输出:

```
1/9 9/1 2/2 4/1 2/7
2/9 9/2 3/2 4/2 3/7
```

这里传递给 transform 的 Lambda 必须返回一个引用:

```
[] (std::pair<int,int>& elem) -> int& {
  return ...
}
```

否则，Lambda 将返回每个元素的副本，以便这些副本的成员递增，而 coll 中的元素保持不变。

转换视图在内部存储传递的范围 (可选择以与 all() 相同的方式转换为视图)，所以只有在传递的范围有效的情况下才有效 (除非传递了右值，则在内部使用了归属视图)。

转换视图的特殊特性

与标准视图一样，转换视图不会将常量性委托给元素。所以即使视图是 const，传递给转换函数的元素也不是 const。

例如，使用上面的 transform.cpp 示例，可以完成以下实现:

```
std::vector<std::pair<int,int>> coll{{1,9}, {9,1}, {2,2}, {4,1}, {2,7}};

// function that yields the smaller of the two values in a pair:
auto minMember = [] (std::pair<int,int>& elem) -> int& {
  return elem.second < elem.first ? elem.second : elem.first;
};

// increment the smaller of the two values in each pair element:
const auto v = coll | std::views::transform(minMember);
for (auto&& member : v) {
  ++member;
}
```

若底层范围是长度范围和通用范围，则获取视图仅是通用的 (迭代器和哨兵具有相同的类型)。为了协调类型，可能必须使用通用视图。

开始迭代器和哨兵 (结束迭代器) 都是特殊的内部辅助类型。

转换视图的接口

“类 std::ranges::transform_view<> 的操作”表列出了转换视图的 API。

操作	效果
transform_view r{}	创建一个 transform_view，引用一个默认构造的范围

transform_view r{rg, func}	创建一个 transform_view，用函数变换 rg 范围内所有元素的值
r.begin()	生成 begin 迭代器
r.end()	生成哨兵 (end 迭代器)
r.empty()	生成的 r 是否为空 (若范围支持，则可用)
if (r)	若 r 不为空，则为 true(若定义了 empty()，则可用)
r.size()	生成元素的数量 (若引用了一个长度范围，则可用)
r.front()	生成的第一个元素 (若转发可用)
r.back()	生成的最后一个元素 (若迭代器为双向且通用，则可用)
r[idx]	生成的第 n 个元素 (若果随机访问，则可用)
r.data()	生成一个指向元素内存的原始指针 (若元素在连续内存中可用)
r.base()	产生对 r 所引用或归属范围的引用

表 8.19 类 std::ranges::transform_view<> 的操作

8.6.2 元素视图

类型:	std::ranges::elements_view<>
内容:	范围中所有类元组元素的第 n 个成员/属性
适配器:	std::views::elements<>
元素类型:	成员/属性的类型
要求:	至少为输入范围
类别:	与传递的相同，但至多是随机访问
是否是长度范围:	若在长度范围内
是否是通用范围:	若在通用范围内
是否是租借范围:	若是租借视图或左值非视图
缓存:	无
常量可迭代:	若在可迭代范围内
传播常量性:	只有在右值范围内

类模板 std::ranges::elements_view<> 定义了一个视图，该视图选择传递的范围中所有元素的第 idx 个成员/属性/元素。视图为每个元素调用 get<idx>(elem)，并且可以使用

- 获取所有 std::tuple 元素的第 idx 个成员
- 获取所有 std::variant 元素的第 idx 个成员
- 获取 std::pair 元素的第一个或第二个成员 (不过，对于关联容器和无序容器的元素，使用 keys_view 和 values_view 更方便)

对于这个视图，类模板参数推导不起作用，因为必须显式地将索引指定为参数，并且类模板不支持部分参数推导:

```
std::vector<std::tuple<std::string, std::string, int>> rg{
  {"Bach", "Johann Sebastian", 1685}, {"Mozart", "Wolfgang Amadeus", 1756},
  {"Beethoven", "Ludwig van", 1770}, {"Chopin", "Frederic", 1810},
};

for (const auto& elem
  : std::ranges::elements_view<decltype(std::views::all(rg)), 2>{rg}) {
  std::cout << elem << ' ';
}
```

循环输出:

```
1685 1756 1770 1810
```

若传递的范围不是视图，则必须使用范围适配器 all() 指定底层范围的类型:

```
std::ranges::elements_view<decltype(std::views::all(rg)), 2>{rg} // OK
```

也可以直接使用 std::views::all_t<> 指定类型:

```
std::ranges::elements_view<std::views::all_t<decltype(rg)&>, 2>{rg} // OK
std::ranges::elements_view<std::views::all_t<decltype((rg))>, 2>{rg} // OK
```

若范围还不是视图，那么 all_t<> 的参数必须是左值引用，所以需要在 rg 的类型后面加上 &，或者在 rg 周围加上双括号 (根据规则，若传递的表达式是左值，则 decltype 会产生一个左值引用)。没有 & 的单圆括号不起作用:

```
std::ranges::elements_view<std::views::all_t<decltype(rg)>, 2>{rg} // ERROR
```

声明视图更简单的方法是使用相应的范围适配器。

元素视图的范围适配器

元素视图也可以 (通常应该) 使用范围适配器创建。适配器使视图的使用更容易，不必指定底层范围的类型:

```
std::views::elements<idx>(rg)
```

例如:

```
for (const auto& elem : std::views::elements<2>(rg)) {
  std::cout << elem << ' ';
}
```

或:

```
for (const auto& elem : rg | std::views::elements<2>) {
  std::cout << elem << ' ';
}
```

使用范围适配器，elements<idx>(rg) 等价于:

```
std::ranges::elements_view<std::views::all_t<decltype(rg)&>, idx>{rg}
```

元素视图在内部存储传递的范围 (可选择以与 all() 相同的方式转换为视图)，只有在传递的范围有效的情况下才有效 (除非传递了右值，则在内部使用了归属视图)。

begin 迭代器和哨兵 (end 迭代器) 都是特殊的内部辅助类型，若传递的是通用范围，则是通用的。

下面是一个使用元素视图的完整示例:

ranges/elementsview.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <ranges>
#include <numbers> // for math constants
#include <algorithm> // for sort()

void print(std::ranges::input_range auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector<std::tuple<int, std::string, double>> coll{
    {1, "pi", std::numbers::pi},
    {2, "e", std::numbers::e},
    {3, "golden-ratio", std::numbers::egamma},
    {4, "euler-constant", std::numbers::phi},
  };

  std::ranges::sort(coll, std::less{},
            [](const auto& e) {return std::get<2>(e);});
  print(std::ranges::elements_view<decltype(std::views::all(coll)), 1>{coll});
  print(coll | std::views::elements<2>);
}
```

该程序有以下输出:

```
golden-ratio euler-constant e pi
0.577216 1.61803 2.71828 3.14159
```

不应该按以下方式对 coll 的元素进行排序:

```
std::ranges::sort(coll | std::views::elements<2>); // OOPS
```

这将只对元素的值进行排序，而不是对整个元素进行排序，并将出现以下输出:

```
pi e golden-ratio euler-constant
0.577216 1.61803 2.71828 3.14159
```

其他类元组类型的元素视图

为了能够将此视图用于用户定义的类型，需要指定一个类似元组的 API。但就目前的规范而言，这实际上并不是因为一般的类元组 api 的设计和相应概念的定义方式存在问题。严格地说，只能对 std::pair<> 和 std::tuple<> 使用 std::ranges::elements_view 类或适配器 std::views::elements<>。

但若确保在包含头文件 <ranges> 之前定义了类似元组的 API，就可以工作了。

例如:

ranges/elementsviewhack.cpp

```
#include <iostream>
#include <string>
#include <tuple>
// don't include <ranges> yet !!

struct Data {
  int id;
  std::string value;
};

std::ostream& operator<< (std::ostream& strm, const Data& d) {
  return strm << '[' << d.id << ": " << d.value << ']';
}

// tuple-like access to Data:
namespace std {
  template<>
  struct tuple_size<Data> : integral_constant<size_t, 2> {
  };

  template<>
  struct tuple_element<0, Data> {
    using type = int;
```

```
  };
  template<>
  struct tuple_element<1, Data> {
    using type = std::string;
  };

  template<size_t Idx> auto get(const Data& d) {
    if constexpr (Idx == 0) {
      return d.id;
    }
    else {
      return d.value;
    }
  }
} // namespace std
```

可以这样:

ranges/elementsviewhack.cpp

```
// don't include <ranges> before "elementsview.hpp"
#include "elementsviewhack.hpp"
#include <iostream>
#include <vector>
#include <ranges>

void print(const auto& coll)
{
  std::cout << "coll:\n";
  for (const auto& elem : coll) {
    std::cout << "- " << elem << '\n';
  }
}

int main()
{
  Data d1{42, "truth"};
  std::vector<Data> coll{d1, Data{0, "null"}, d1};
  print(coll);
  print(coll | std::views::take(2));
  print(coll | std::views::elements<1>);
}
```

输出如下:

```
coll:
- [42: truth]
- [0: null]
- [42: truth]
```

```
coll:
- [42: truth]
- [0: null]
coll:
- truth
- null
- truth
```

元素视图的接口

“类 std::ranges::elements_view<> 的操作”表列出了元素视图的 API。

操作	效果
elements_view r{}	创建一个 elements_view，引用一个默认构造的范围
elements_view r{rg}	创建一个引用范围 rg 的 elements_view
r.begin()	生成 begin 迭代器
r.end()	生成哨兵 (end 迭代器)
r.empty()	生成的 r 是否为空 (若范围支持，则可用)
if (r)	若 r 不为空，则为 true(若定义了 empty()，则可用)
r.size()	生成元素的数量 (若引用了一个长度范围，则可用)
r.front()	生成的第一个元素 (若转发可用)
r.back()	生成的最后一个元素 (若迭代器为双向且通用，则可用)
r[idx]	生成的第 n 个元素 (若随机访问可用)
r.data()	生成一个指向元素内存的原始指针 (若元素在连续内存中可用)。
r.base()	生成对 r 所引用或归属范围的引用

表 8.20 类 std::ranges::elements_view<> 的操作

8.6.3 键和值的视图

类型:	std::ranges::keys_view<>	std::ranges::values_view<>
内容:	范围中所有类元组元素的第一个/第二个成员或属性	
适配器:	std::views::keys	std::views::values
元素类型:	第一个成员的类型	第二个成员的类型
要求:	至少为输入范围	至少为输入范围
类别:	与传递的相同，但至多是随机访问	
是否是长度范围:	若在长度范围内	若在长度范围内
是否是通用范围:	若在通用范围内	若在通用范围内
是否是租借范围:	若为租借视图或左值非视图	

缓存:	无	Nothing
常量可迭代:	若在可迭代范围内	若在可迭代范围内
传播常量性:	只有在右值范围内	只有在右值范围内

类模板 std::ranges::keys_view<> 定义了一个视图，该视图从传递的范围的元素中选择第一个成员/属性/元素。只是使用索引为 0 的元素视图的快捷方式，为每个元素调用 get<0>(elem)。

类模板 std::ranges::values_view<> 定义了一个视图，该视图从传递的范围的元素中选择第二个成员/属性/元素。只是使用索引为 1 的元素视图的快捷方式，为每个元素调用 get<1>(elem)。

可以这样使用:

- 获取 std::pair 元素的第一/第二成员，这对于选择 map、unordered_map、multimap 和 unordered_multimap 的元素的键/值特别有用
- 获取 std:: 元组元素的第一/第二个成员
- 获取 std::variant 元素的第一/第二个的选项

对于后两个，直接使用索引为 0 的 elements_view 元素可能更具可读性。

类模板参数推导对这些视图还不起作用出于这个原因，必须显式地指定模板参数。例如:

```
std::map<std::string, int> rg{
  {"Bach", 1685}, {"Mozart", 1756}, {"Beethoven", 1770},
  {"Tchaikovsky", 1840}, {"Chopin", 1810}, {"Vivaldi", 1678},
};

for (const auto& e : std::ranges::keys_view<decltype(std::views::all(rg))>{rg}) {
  std::cout << e << ' ';
}
```

循环输出:

```
Bach Beethoven Chopin Mozart Tchaikovsky Vivaldi
```

键/值视图的范围适配器

键和值视图也可以 (通常应该) 使用范围适配器创建: 适配器使视图的使用更容易，不必指定底层范围的类型:

```
std::views::keys(rg)
std::views::values(rg)
```

例如:

```
for (const auto& elem : rg | std::views::keys) {
  std::cout << elem << ' ';
}
```

或:

```
for (const auto& elem : rg | std::views::values) {
  std::cout << elem << ' ';
}
```

所有其他方面都与元素视图相匹配。

下面是一个使用键和值视图的完整示例:

ranges/keysvaluesview.cpp

```
#include <iostream>
#include <string>
#include <unordered_map>
#include <ranges>
#include <numbers> // for math constants
#include <algorithm> // for sort()

void print(std::ranges::input_range auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::unordered_map<std::string, double> coll{
    {"pi", std::numbers::pi},
    {"e", std::numbers::e},
    {"golden-ratio", std::numbers::egamma},
    {"euler-constant", std::numbers::phi},
  };

  print(std::ranges::keys_view<decltype(std::views::all(coll))>{coll});
  print(std::ranges::values_view<decltype(std::views::all(coll))>{coll});

  print(coll | std::views::keys);
  print(coll | std::views::values);
}
```

该程序有以下输出:

```
euler-constant golden-ratio e pi
1.61803 0.577216 2.71828 3.14159
euler-constant golden-ratio e pi
1.61803 0.577216 2.71828 3.14159
```

8.7. 修改视图

本节讨论所有改变元素顺序的视图 (到目前为止，只有一个视图)。

8.7.1 反向视图

类型:	std::ranges::reverse_view<>
内容:	一个范围内的所有元素按倒序排列
适配器:	std::views::reverse
元素类型:	与传递的范围类型相同
要求:	至少为双向范围
类别:	与传递的相同，但至多是随机访问
是否是长度范围:	若在常长度范围内
是否是通用范围:	总是
是否是租借范围:	若是租借视图或左值非视图
缓存:	缓存 begin()，除非通用范围或随机访问和长度范围
常量可迭代:	若在可重复的通用范围内
传播常量性:	只有在右值范围内

类模板 std::ranges::reverse_view<> 定义了一个以相反顺序遍历基础范围元素的视图。

例如:

```
std::vector<int> rg{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
...
for (const auto& elem : std::ranges::reverse_view{rg}) {
  std::cout << elem << ' ';
}
```

循环输出:

```
13 12 11 10 9 8 7 6 5 4 3 2 1
```

范围适配器的反向视图

反向视图也可以 (通常应该) 使用范围适配器创建:

```
std::views::reverse(rg)
```

例如:

```
for (const auto& elem : std::views::reverse(rg)) {
  std::cout << elem << ' ';
}
```

或:

```
for (const auto& elem : rg | std::views::reverse) {
  std::cout << elem << ' ';
}
```

注意，适配器可能并不总是产生 reverse_view:

- 反向反转的范围会生成原始范围。
- 若传递反向范围，则返回原始子范围和相应的非反向迭代器。

反向视图在内部存储传递的范围 (可选择以与 all() 相同的方式转换为视图)，所以只有在传递的范围有效的情况下才有效 (除非传递了右值，则在内部使用了归属视图)。

迭代器只是传递范围的反向迭代器。

下面是一个使用反向视图的完整示例:

ranges/reverseview.cpp

```
include <iostream>
#include <string>
#include <vector>
#include <ranges>

void print(std::ranges::input_range auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector coll{1, 2, 3, 4, 1, 2, 3, 4};

  print(coll); // 1 2 3 4 1 2 3 4
  print(std::ranges::reverse_view{coll}); // 4 3 2 1 4 3 2 1
  print(coll | std::views::reverse); // 4 3 2 1 4 3 2 1
}
```

该程序有以下输出:

```
1 2 3 4 1 2 3 4
4 3 2 1 4 3 2 1
4 3 2 1 4 3 2 1
```

反向视图和缓存

为了获得更好的性能，反向视图会在视图中缓存 begin() 的结果 (除非该范围只是一个输入范围)。

注意，对于具有随机访问的范围 (例如，数组，vector 和队列)，缓存的开始偏移量与视图一起复制。否则，不复制缓存的开头。

当反向视图引用的范围被修改时，缓存可能会产生副作用。

例如:

ranges/reversecache.cpp

```cpp
#include <iostream>
#include <vector>
#include <list>
#include <ranges>

void print(auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector vec{1, 2, 3, 4};
  std::list lst{1, 2, 3, 4};

  auto vVec = vec | std::views::take(3) | std::views::reverse;
  auto vLst = lst | std::views::take(3) | std::views::reverse;
  print(vVec); // OK: 3 2 1
  print(vLst); // OK: 3 2 1

  // insert a new element at the front (=> 0 1 2 3 4)
  vec.insert(vec.begin(), 0);
  lst.insert(lst.begin(), 0);
  print(vVec); // OK: 2 1 0
  print(vLst); // OOPS: 3 2 1

  // creating a copy heals:
  auto vVec2 = vVec;
  auto vLst2 = vLst;
  print(vVec2); // OK: 2 1 0
  print(vLst2); // OK: 2 1 0
}
```

从形式上讲，将视图复制到 vector 会创建未定义的行为，因为 C++ 标准没有指定如何进行缓存。由于 vector 的重新分配会使所有迭代器失效，因此缓存的迭代器将失效。对于随机访问范围，视图通常缓存偏移量，而不是迭代器，所以对于 vector，视图仍然有效。

还需要注意的是，对整个容器的反向视图可以正常工作，因为结束迭代器是缓存的。
根据经验，**底层范围修改后不要使用一个反向视图的 begin()**。

反向视图和 const

注意，不能总是迭代 const 反向视图。实际上，引用的范围必须是通用范围。
例如:

```
void printElems(const auto& coll) {
  for (const auto elem& e : coll) {
    std::cout << elem << '\n';
  }
}

std::vector vec{1, 2, 3, 4, 5};

// leading odd elements of vec:
auto vecFirstOdd = std::views::take_while(vec, [](auto x) {
            return x % 2 != 0;
          });

printElems(vec | std::views::reverse); // OK
printElems(vecFirstOdd); // OK
printElems(vecFirstOdd | std::views::reverse); // ERROR
```

要在泛型代码中支持视图，必须使用通用/转发引用:

```
void printElems(auto&& coll) {
  ...
}

std::vector vec{1, 2, 3, 4, 5};

// leading odd elements of vec:
auto vecFirstOdd = std::views::take_while(vec, [](auto x) {
              return x % 2 != 0;
            });

printElems(vecFirstOdd | std::views::reverse); // OK
```

反向视图的接口

“类 std::ranges::reverse_view<> 的操作”表列出了反向视图的 API。

操作	效果
reverse_view r{}	创建一个指向默认构造范围的 reverse_view
reverse_view r{rg}	创建一个指向范围 rg 的 reverse_view
r.begin()	生成 begin 迭代器
r.end()	生成哨兵 (end 迭代器)
r.empty()	生成的 r 是否为空 (若范围支持，则可用)
if (r)	若 r 不为空，则为 true(若定义了 empty()，则可用)
r.size()	生成元素的数量 (若引用一个长度范围，则可用)
r.front()	生成的第一个元素 (若转发可用)
r.back()	生成的最后一个元素 (若范围为双向且通用，则可用)
r[idx]	生成的第 n 个元素 (若随机访问可用)
r.data()	生成一个指向元素内存的原始指针 (若元素在连续内存中可用)
r.base()	生成对 r 所引用或归属范围的引用

表 8.21 类 std::ranges::reverse_view<> 的操作

8.8. 多范围视图

本节讨论处理多个范围的所有视图。

8.8.1 拆分和惰性拆分视图

类型:	std::ranges::split_view<>	std::range::laze_split_view<>
内容:	一个范围的所有元素被分割成多个视图	
适配器:	std::views::split()	std::views::lazy_split()
元素类型:	集合的引用	集合的引用
要求:	至少为前向范围	至少为前向范围
类别:	总是前向	输入或前向
是否是长度范围:	从不	从不
是否是通用范围:	若在通用前向范围	若在通用前向范围
是否是租借范围:	从不	从不
缓存:	总是缓存 begin()	在输入范围内，缓存当前值
常量可迭代:	从不	若在可重复的前向范围内
传播常量性:	–	从不

std::ranges::split_view<> 和 std::ranges::lazy_split_view<> 两个类模板都定义了一个视图，该视图引用由分隔符分隔将范围拆分为多个子视图。[std::ranges::lazy_split_view<> 不是原始 C++20 标准的一部分，但后来通过http://wg21.link/p2210r2添加到 C++20 中。]

split_view<> 和 lazy_split_view<> 的区别如下:

- split_view<> 不能迭代 const 视图;lazy_split_view<> 可以 (引用的范围至少是前向范围)。
- split_view<> 只能处理至少具有前向迭代器的范围 (必须满足 forward_range 的概念)。
- split_view<> 元素只是引用范围的迭代器类型的 std::ranges::subrange(保持引用范围的类别)。lazy_split_views<> 元素是 std::ranges::lazy_split_view 类型的视图，其总是向前范围，甚至不支持 size()。
- split_view<> 具有更好的性能。

只使用输入范围或视图被用作 const 范围，应该使用 split_view<>，除非不能这样做。例如:

```
std::vector<int> rg{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
...
for (const auto& sub : std::ranges::split_view{rg, 5}) {
  for (const auto& elem : sub) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

循环输出:

```
1 2 3 4
6 7 8 9 10 11 12 13
```

只要在 rg 中找到值为 5 的元素，就结束前一个视图并开始一个新视图。

用于拆分和延迟拆分视图的范围适配器

拆分视图和延迟拆分视图也可以 (通常应该) 使用范围适配器来创建，适配器只是将其参数传递给相应的视图构造函数:

```
std::views::split(rg, sep)
std::views::lazy_split(rg, sep)
```

例如:

```
for (const auto& sub : std::views::split(rg, 5)) {
  for (const auto& elem : sub) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

或:

```
for (const auto& sub : rg | std::views::split(5)) {
  for (const auto& elem : sub) {
```

```
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

创建的视图可能是空的。对于每个前后分隔符，以及每当两个分隔符在彼此后面时，都会创建一个空视图。例如:

```
std::list<int> rg{5, 5, 1, 2, 3, 4, 5, 6, 5, 5, 4, 3, 2, 1, 5, 5};
for (const auto& sub : std::ranges::split_view{rg, 5}) {
  std::cout << "subview: ";
  for (const auto& elem : sub) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

输出如下:[最初的 C++20 标准指定最后一个分隔符可忽略，所以我们将在最后只得到一个空子视图，修复于http://wg21.link/p2210r2。]

```
subview:
subview:
subview: 1 2 3 4
subview: 6
subview:
subview: 4 3 2 1
subview:
subview:
```

除了单值之外，还可以传递一系列值作为分隔符。例如:

```
std::vector<int> rg{1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3 };
...
// split separated by a sequence of 5 and 1:
for (const auto& sub : std::views::split(rg, std::list{5, 1})) {
  for (const auto& elem : sub) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

这段代码的输出是

```
1 2 3 4
2 3 4
2 3
```

传递的元素集合必须是有效的视图和 forward_range，所以在容器中指定子序列时，必须将其转换为视图。例如:

```
// split with specified pattern of 4 and 5:
std::array pattern{4, 5};
for (const auto& sub : std::views::split(rg, std::views::all(pattern))) {
  ...
}
```

可以使用拆分视图来拆分字符串:

```
std::string str{"No problem can withstand the assault of sustained thinking"};
for (auto sub : std::views::split(str, "th"sv)) { // split by "th"
  std::cout << std::string_view{sub} << '\n';
}
```

每个子字符串 sub 的类型为 std::ranges::subrange<decltype(str.begin())>，这样的代码将不能用于延迟拆分视图。

拆分或延迟拆分视图会在内部存储传入的范围 (也可以像 all() 那样转换为视图)，只有在传递的范围有效的情况下才有效 (除非传递了右值，则在内部使用了归属视图)。

begin 迭代器和哨兵 (end 迭代器) 都是特殊的内部辅助类型，若传递的范围是通用的，则是通用的。

这里是使用拆分视图的完整示例:

ranges/splitview.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <array>
#include <ranges>

void print(auto&& obj, int level = 0)
{
  if constexpr(std::ranges::input_range<decltype(obj)>) {
    std::cout << '[';
    for (const auto& elem : obj) {
      print(elem, level+1);
    }
    std::cout << ']';
  }
  else {
    std::cout << obj << ' ';
  }
  if (level == 0) std::cout << '\n';
}

int main()
```

```
{
  std::vector coll{1, 2, 3, 4, 1, 2, 3, 4};

  print(coll); // [1 2 3 4 1 2 3 4 ]
  print(std::ranges::split_view{coll, 2}); // [[1 ][3 4 1 ][3 4 ]]
  print(coll | std::views::split(3)); // [[1 2 ][4 1 2 ][4 ]]
  print(coll | std::views::split(std::array{4, 1})); // [[1 2 3 ][2 3 4 ]]
}
```

该程序有以下输出:

```
[1 2 3 4 1 2 3 4 ]
[[1 ][3 4 1 ][3 4 ]]
[[1 2 ][4 1 2 ][4 ]]
[[1 2 3 ][2 3 4 ]]
```

拆分视图和 const

不能在 const 的拆分视图上迭代。

例如:

```
std::vector<int> coll{5, 1, 5, 1, 2, 5, 5, 1, 2, 3, 5, 5, 5};
...
const std::ranges::split_view sv{coll, 5};
for (const auto& sub : sv) { // ERROR for const split view
  std::cout << sub.size() << ' ';
}
```

特别是若将视图传递给一个将参数作为 const 引用的泛型函数:

```
void printElems(const auto& coll) {
  ...
}
printElems(std::views::split(rg, 5)); // ERROR
```

要在泛型代码中支持视图，必须使用通用/转发引用:

```
void printElems(auto&& coll) {
  ...
}
```

或者，可以使用 lazy_split_view，从而子视图元素只能用作前向范围，所以不能调用 size()、向后迭代或对元素排序:

```cpp
std::vector<int> coll{5, 1, 5, 1, 2, 5, 5, 1, 2, 3, 5, 5, 5};
...
const std::ranges::lazy_split_view sv{coll, 5};
for (const auto& sub : sv) { // OK for const lazy-split view
  std::cout << sub.size() << ' '; // ERROR
  std::sort(sub); // ERROR
  for (const auto& elem : sub) { // OK
    std::cout << elem << ' ';
  }
}
```

拆分和延迟拆分视图的接口

“类 std::ranges::split_view<> 和 std::ranges::split_view<> 的操作”表列出了拆分视图或延迟拆分视图的 API。

操作	效果
split_view r{}	创建一个 split_view，它引用一个默认构造的范围
split_view r{rg}	创建一个 split_view，引用范围 rg
r.begin()	生成 begin 迭代器
r.end()	生成哨兵 (end 迭代器)
r.empty()	生成的 r 是否为空 (若范围支持则可用)
if (r)	若 r 不为空，则为 true(若定义了 empty()，则可用)
r.size()	生成元素的数量 (若引用一个长度范围，则可用)
r.front()	生成的第一个元素 (若转发可用)
r.back()	生成的最后一个元素 (若范围为双向且通用，则可用)
r[idx]	生成的第 n 个元素 (若随机访问可用)
r.data()	生成一个指向元素内存的原始指针 (若元素在连续内存中可用)
r.base()	生成对 r 所引用或归属范围的引用

表 8.22 类 std::ranges::split_view<> 和 std::ranges::split_view<> 的操作

8.8.2 连接视图

类型:	std::ranges::join_view<>
内容:	一个范围或多个范围的所有元素作为一个视图
适配器:	std::views::join()
元素类型:	与传递的范围类型相同
要求:	至少为输入范围

类别:	双向输入
是否是长度范围:	从不
是否是通用范围:	可变
是否是租借范围:	从不
缓存:	无
常量可迭代:	若常量可迭代，范围和元素仍是 const 的引用
传播常量性:	只有在右值范围内

类模板 std::ranges::join_view<> 定义了一个遍历多个范围视图的所有元素的视图。

例如:

```
std::vector<int> rg1{1, 2, 3, 4};
std::vector<int> rg2{0, 8, 15};
std::vector<int> rg3{5, 4, 3, 2, 1, 0};
std::array coll{rg1, rg2, rg3};
...
for (const auto& elem : std::ranges::join_view{coll}) {
  std::cout << elem << ' ';
}
```

循环输出:

```
1 2 3 4 0 8 15 5 4 3 2 1 0
```

连接视图的范围适配器

连接视图也可以 (通常应该) 使用范围适配器创建，适配器只是将其参数传递给 std::ranges::join_view 构造函数:

```
std::views::join(rg)
```

例如:

```
for (const auto& elem : std::views::join(coll)) {
  std::cout << elem << ' ';
}
```

或:

```
for (const auto& elem : coll | std::views::join) {
  std::cout << elem << ' ';
}
```

下面是使用连接视图的完整示例:

ranges/joinview.cpp

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <array>
#include <ranges>
#include "printcoll.hpp"

int main()
{
  std::vector<std::string> rg1{"he", "hi", "ho"};
  std::vector<std::string> rg2{"---", "|", "---"};
  std::array coll{rg1, rg2, rg1};

  printColl(coll); // ranges of ranges of strings
  printColl(std::ranges::join_view{coll}); // range of strings
  printColl(coll | std::views::join); // range of strings
  printColl(coll | std::views::join | std::views::join); // ranges of chars
}
```

使用可以递归输出集合的辅助函数:

ranges/printcoll.cpp

```cpp
#include <iostream>
#include <ranges>

template<typename T>
void printColl(T&& obj, int level = 0)
{
  if constexpr(std::same_as<std::remove_cvref_t<T>, std::string>) {
    std::cout << "\"" << obj << "\"";
  }
  else if constexpr(std::ranges::input_range<T>) {
    std::cout << '[';
    for (auto pos = obj.begin(); pos != obj.end(); ++pos) {
      printColl(*pos, level+1);
      if (std::ranges::next(pos) != obj.end()) {
        std::cout << ' ';
      }
    }
    std::cout << ']';
  }
  else {
    std::cout << obj;
  }
  if (level == 0) std::cout << '\n';
}
```

该程序有以下输出:

```
[["he" "hi" "ho"] ["---" "|" "---"] ["he" "hi" "ho"]]
["he" "hi" "ho" "---" "|" "---" "he" "hi" "ho"]
["he" "hi" "ho" "---" "|" "---" "he" "hi" "ho"]
[h e h i h o - - - | - - - h e h i h o]
```

结合类型 std::ranges::subrange，可以使用连接视图来连接多个数组。

例如:

```
int arr1[]{1, 2, 3, 4, 5};
int arr2[] = {0, 8, 15};
int arr3[10]{1, 2, 3, 4, 5};
...
std::array<std::ranges::subrange<int*>, 3> coll{arr1, arr2, arr3};
for (const auto& elem : std::ranges::join_view{coll}) {
  std::cout << elem << ' ';
}
```

或者，可以这样声明 coll:

```
std::array coll{std::ranges::subrange{arr1},
        std::ranges::subrange{arr2},
        std::ranges::subrange{arr3}};
```

连接视图是唯一视图在 c++ 中 20 处理范围的范围，所以有外部迭代器和内部迭代器，结果视图的属性可能依赖于两者。

注意，内部迭代器不支持迭代器特性，应该使用像 std::ranges::next() 这样的工具，而非 std::next()。否则，代码可能无法编译。

连接视图和 const

不能总是迭代 const 连接视图。若范围不是 const 可迭代的，或者内部范围产生普通值而不是引用，就会发生这种情况。

对于后一种情况，请考虑以下示例:

ranges/joinconst.cpp

```
#include <vector>
#include <array>
#include <ranges>
#include "printcoll.hpp"

void printConstColl(const auto& coll)
{
  printColl(coll);
}
```

```
10
11 int main()
12 {
13   std::vector<int> rg1{1, 2, 3, 4};
14   std::vector<int> rg2{0, 8, 15};
15   std::vector<int> rg3{5, 4, 3, 2, 1, 0};
16   std::array coll{rg1, rg2, rg3};
17
18   printConstColl(coll);
19   printConstColl(coll | std::views::join);
20
21   auto collTx = [] (const auto& coll) { return coll; };
22   auto coll2values = coll | std::views::transform(collTx);
23
24   printConstColl(coll2values);
25   printConstColl(coll2values | std::views::join); // ERROR
26 }
```

当使用连接三个范围数组的元素时，可以调用 printConstColl()，将范围作为 const 引用。会得到以下输出:

```
[[1 2 3 4] [0 8 15] [5 4 3 2 1 0]]
[1 2 3 4 0 8 15 5 4 3 2 1 0]
```

然而，将整个数组传递给一个转换视图，该视图按值返回所有内部范围，调用 printConstColl() 会导致错误。

调用视图的 printColl()，其内部范围产生普通值，工作得很好。这要求 printColl() 使用 std::ranges::next()，而非 std::next()。否则，即使下面的代码也不能编译:

```
1 printColl(coll2values | std::views::join); // ERROR if std::next() used
```

连接视图的特殊特征

若外部范围和内部范围是双向的，并且内部范围是通用范围，则产生的类别是双向的。若外部范围和内部范围至少是前向范围，则生成的类别是前向的。否则，生成的类别为输入范围。

连接视图的接口

“类 std::ranges::join_view<> 的操作”表列出了连接视图的 API。

操作	效果
join_view r{}	创建一个引用默认构造范围的 join_view
join_view r{rg}	创建一个引用范围 rg 的 join_view

r.begin()	生成 begin 迭代器
r.end()	生成哨兵 (end 迭代器)
r.empty()	生成的 r 是否为空 (若该范围支持，则可用)
if (r)	若 r 不为空，则为 true(若定义了 empty()，则可用)
r.size()	生成元素的数量 (若引用了一个长度范围，则可用)
r.front()	生成的第一个元素 (若转发可用)
r.back()	生成的最后一个元素 (若范围为双向且通用，则可用)
r[idx]	生成的第 n 个元素 (若随机访问可用)
r.data()	生成一个指向元素内存的原始指针 (若元素在连续内存中可用)
r.base()	产生对 r 所引用或归属范围的引用

表 8.23 类 std::ranges::join_view<> 的操作

第 9 章 Span

为了处理范围，C++20 引入了两个视图类型，这些视图类型不会在自己的内存中存储元素，只引用存储在其他范围或视图中的元素。std::span<> 类模板就是这种视图。

历史上看，span 是 C++17 中引入的字符串视图的泛化。span 指的是任何元素类型的数组，作为原始指针和大小的组合，提供了通常的集合接口，用于读取和写入存储在连续内存中的元素。

通过要求 span 只能引用连续内存中的元素，迭代器可以是原始指针，这使得其很廉价。该集合提供随机访问 (以便跳转到范围内的任何位置)，因此可以使用此视图对元素进行排序，或者可以使用生成位于底层范围中间或末尾的 n 个元素的子序列的操作。

使用 span 既廉价又快速 (应该按值传递)，但也有潜在的危险。与原始指针一样，在使用 span 时，需要由开发者来确保引用序列的有效性。此外，span 支持写访问的情况，可能会破坏 const 的正确性 (或者不能按期望的方式工作)。

9.1. 使用 Span

来看一些使用 span 的例子。首先，必须讨论如何指定 span 中的元素数量。

9.1.1 固定和动态区段

声明 span 时，可以设置元素数量，也可以不设置，这样 span 引用的元素数量就可以改变。

具有指定固定数量元素的 span 称为具有固定区段的 span，可以通过指定元素类型和大小作为模板参数来声明，或者通过带有迭代器和大小的数组 (C 风格数组或 std::array<>) 来初始化:

```
int a5[5] = {1, 2, 3, 4, 5};
std::array arr5{1, 2, 3, 4, 5};
std::vector vec{1, 2, 3, 4, 5, 6, 7, 8};

std::span sp1 = a5; // span with fixed extent of 5 elements
std::span sp2{arr5}; // span with fixed extent of 5 elements
std::span<int, 5> sp3 = arr5; // span with fixed extent of 5 elements
std::span<int, 3> sp4{vec}; // span with fixed extent of 3 elements
std::span<int, 4> sp5{vec.data(), 4}; // span with fixed extent of 4 elements
std::span sp6 = sp1; // span with fixed extent of 5 elements
```

对于这样的 span，成员函数 size() 会生成作为类型指定部分的长度，但不能调用默认构造函数 (除非区段为 0)。

元素数量在其生命周期内不稳定的 sapn 称为具有动态区段的 span。元素的数量取决于 span 所引用的元素的顺序，并且可能由于分配了一个新的范围而改变 (没有其他方法可以改变元素的数量)。例如:

```
int a5[5] = {1, 2, 3, 4, 5};
std::array arr5{1, 2, 3, 4, 5};
std::vector vec{1, 2, 3, 4, 5, 6, 7, 8};
```

```
4
5  std::span<int> sp1; // span with dynamic extent (initially 0 elements)
6  std::span sp2{a5, 3}; // span with dynamic extent (initially 3 elements)
7  std::span<int> sp3{arr5}; // span with dynamic extent (initially 5 elements)
8  std::span sp4{vec}; // span with dynamic extent (initially 8 elements)
9  std::span sp5{arr5.data(), 3}; // span with dynamic extent (initially 3 elements)
10 std::span sp6{a5+1, 3}; // span with dynamic extent (initially 3 elements)
```

需要由开发者保证 span 指向一个具有足够元素的有效范围。

对于这两种情况，来看一下完整的例子。

9.1.2 使用带动态区段的 span

下面是第一个使用动态区段 span 的例子:

lib/spandyn.cpp

```
1  #include <iostream>
2  #include <string>
3  #include <vector>
4  #include <ranges>
5  #include <algorithm>
6  #include <span>
7
8  template<typename T, std::size_t Sz>
9  void printSpan(std::span<T, Sz> sp)
10 {
11   for (const auto& elem : sp) {
12     std::cout << '"' << elem << "\" ";
13   }
14   std::cout << '\n';
15 }
16
17 int main()
18 {
19   std::vector<std::string> vec{"New York", "Tokyo", "Rio", "Berlin", "Sydney"};
20
21   // define view to first 3 elements:
22   std::span<const std::string> sp{vec.data(), 3};
23   std::cout << "first 3: ";
24   printSpan(sp);
25
26   // sort elements in the referenced vector:
27   std::ranges::sort(vec);
28   std::cout << "first 3 after sort(): ";
29   printSpan(sp);
30
31   // insert a new element:
32   // - must reassign the internal array of the vector if it reallocated new memory
```

```
  auto oldCapa = vec.capacity();
  vec.push_back("Cairo");
  if (oldCapa != vec.capacity()) {
    sp = std::span{vec.data(), 3};
  }
  std::cout << "first 3 after push_back(): ";
  printSpan(sp);

  // let span refer to the vector as a whole:
  sp = vec;
  std::cout << "all: ";
  printSpan(sp);

  // let span refer to the last five elements:
  sp = std::span{vec.end()-5, vec.end()};
  std::cout << "last 5: ";
  printSpan(sp);

  // let span refer to the last four elements:
  sp = std::span{vec}.last(4);
  std::cout << "last 4: ";
  printSpan(sp);
}
```

程序输出如下:

```
first 3:              "New York" "Tokyo" "Rio"
first 3 after sort(): "Berlin" "New York" "Rio"
first 3 after push_back(): "Berlin" "New York" "Rio"
all:   "Berlin" "New York" "Rio" "Sydney" "Tokyo" "Cairo"
last 5: "New York" "Rio" "Sydney" "Tokyo" "Cairo"
last 4: "Rio" "Sydney" "Tokyo" "Cairo"
```

让我们一步一步地来分析这个例子。

声明 span

在 main() 中，首先用 vector 的前三个元素初始化了一个由三个常量字符串组成的 span:

```
std::vector<std::string> vec{"New York", "Rio", "Tokyo", "Berlin", "Sydney"};

std::span<const std::string> sp{vec.data(), 3};
```

初始化时，传递序列的开头和元素的数目。本例中，引用 vec 的前三个元素。
关于这个声明，有几件事需要注意:

- 由开发者来确保元素的数量与 span 的范围相匹配，并且元素是有效的。若 vector 没有足够的元素，则行为未定义:

```
std::span<const std::string> sp{vec.begin(), 10}; // undefined behavior
```

- 指定元素为 const std::string 时，就不能通过 span 进行修改。将 span 本身声明为 const，并不提供对引用元素的只读访问 (通常对于视图，const 不会传播):

```
std::span<const std::string> sp1{vec.begin(), 3}; // elements cannot be modified
const std::span<std::string> sp2{vec.begin(), 3}; // elements can be modified
```

- 为 span 使用不同于引用元素的元素类型，就像可以为转换为基础元素类型的 span 使用的类型一样。但事实并非如此，只能添加 const 这样的限定符:

```
std::vector<int> vec{ ... };
std::span<long> sp{vec.data(), 3}; // ERROR
```

传递和输出 span

接下来，输出 span，并将其传递给一个通用的 print 函数:

```
printSpan(sp);
```

print 函数可以处理 span(只要为元素定义了输出操作符):

```
template<typename T, std::size_t Sz>
void printSpan(std::span<T, Sz> sp)
{
  for (const auto& elem : sp) {
    std::cout << '"' << elem << "\" ";
  }
  std::cout << '\n';
}
```

读者们可能会感到惊讶，可以调用函数模板 printSpan<>()，即使其有一个非类型模板参数表示 span 的大小。其之所以有效，是因为 std::span<T> 是快捷方式——具有伪长度 std::dynamic_extent 的 span:

```
std::span<int> sp; // equivalent to std::span<int, std::dynamic_extent>
```

实际上，类模板 std::span<> 是这样声明的:

```
namespace std {
  template<typename ElementType, size_t Extent = dynamic_extent>
  class span {
    ...
  };
}
```

这允许开发者提供像 printSpan<>() 这样的通用代码，其既适用于固定范围的跨度，也适用于动态区段的 span。当使用具有固定区段的 span 调用 printSpan<>() 时，区段可作为模板参数传递:

```
std::span<int, 5> sp{ ... };

printSpan(sp); // calls printSpan<int, 5>(sp)
```

span 是按值传递的。这是传递 span 的推荐方法，复制它们的成本很低。在内部，span 只有一个指针和一个长度信息。

print 函数内部，使用基于范围的 for 循环遍历 span 的元素。因为 span 提供了 begin() 和 end() 的迭代器支持，所以这是可能的。

但要注意: 无论通过值传递还是通过 const 引用传递 span，只要没有将元素声明为 const，在函数内部仍可以修改元素。这就是为什么将 span 的元素声明为 const 有意义。

处理引用语义

接下来，对容器所引用的元素进行排序 (使用 std::ranges::sort()，将容器作为一个整体):

```
std::ranges::sort(vec);
```

由于 span 具有引用语义，因此这种排序也会影响 span 的元素，其结果是 span 现在引用不同的值。

若没有 const 元素的 span，也可以调用 sort() 传递 span。

引用语义在使用 span 时必须谨慎，在下个示例中就会看到相关语句。这里，将一个新元素插入 vector 中，该 vector 中保存了跨度所引用的元素。由于 span 的引用语义，若 vector 分配新的内存，会使所有迭代器和指向其元素的指针无效，所以重新分配也会使引用 vector 元素的 span 失效。span 指向了不再存在的元素。

出于这个原因，需要在插入前后都要仔细检查容量 (分配内存的最大元素数量)。若容量发生变化，则重新初始化 span，使其指向新内存中的前三个元素:

```
auto oldCapa = vec.capacity();
vec.push_back("Cairo");
if (oldCapa != vec.capacity()) {
  sp = std::span{vec.data(), 3};
}
```

只能执行这种重新初始化，因为 span 本身不为 const。

将容器分配给 span

接下来，将 vector 作为一个整体赋值给 span，并将其输出:

```
std::span<const std::string> sp{vec.begin(), 3};
...
sp = vec;
```

可以看到对具有动态区段 span 的赋值可以改变元素的数量。只要容器允许使用成员函数 data() 访问这些元素，则 span 可以使用任何类型的容器或范围来保存连续内存中的元素。

但由于模板类型推导的限制，不能将这样的容器传递给期望使用 span 的函数。必须明确指定要将 vector 转换为 span:

```
printSpan(vec); // ERROR: template type deduction doesn't work here
printSpan(std::span{vec}); // OK
```

分配不同的子序列

通常，span 的赋值操作符接受另一个元素序列。

下面的示例使用这种方式来引用 vector 中的最后三个元素:

```
std::span<const std::string> sp{vec.data(), 3};
...
// assign view to last five elements:
sp = std::span{vec.end()-5, vec.end()};
```

这里，可以使用两个迭代器指定引用的序列，将序列的开始和结束定义为半开范围 (包括开始的值，不包括结束的值)。要求满足 std::size_sentinel_for 的概念，以便构造函数可以计算差值。

下面，最后 n 个元素也可以使用 span 的成员函数来赋值:

```
std::vector<std::string> vec{"New York", "Tokyo", "Rio", "Berlin", "Sydney"};
std::span<const std::string> sp{vec.data(), 3};
...
// assign view to last four elements:
sp = std::span{vec}.last(4);
```

span 是唯一提供在范围中间或末尾生成元素序列的视图。

只要元素类型合适，就可以传递任何其他类型的元素序列:

```
std::vector<std::string> vec{"New York", "Tokyo", "Rio", "Berlin", "Sydney"};
std::span<const std::string> sp{vec.begin(), 3};
...
std::array<std::string, 3> arr{"Tick", "Trick", "Track"};
sp = arr; // OK
```

但 span 不支持元素类型的隐式类型转换 (除了添加 const)。例如，无法编译以下代码:

```
std::span<const std::string> sp{vec.begin(), 3};
...
std::array arr{"Tick", "Trick", "Track"}; // deduces std::array<const char*, 3>
sp = arr; // ERROR: different element types
```

9.1.3 使用具有非 const 元素的实例

初始化 span 时，可以使用类模板实参推导，从而推导出元素的类型 (和范围):

```
std::span sp{vec.begin(), 3}; // deduces: std::span<std::string>
```

然后，span 会声明元素的类型，使其具有基础范围的元素类型，甚至可以修改基础范围的值 (前提是基础范围没有将其元素声明为 const)。

此特性可用于允许 span 在一个语句中修改范围的元素。例如，可以对元素的一个子集进行排序:

lib/spanview.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <ranges>
#include <algorithm>
#include <span>

void print(std::ranges::input_range auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << '"' << elem << "\" ";
  }
  std::cout << '\n';
}

int main()
{
  std::vector<std::string> vec{"New York", "Tokyo", "Rio", "Berlin", "Sydney"};
  print(vec);

  // sort the three elements in the middle:
  std::ranges::sort(std::span{vec}.subspan(1, 3));
  print(vec);

  // print last three elements:
  print(std::span{vec}.last(3));
}
```

这里，创建临时 span 来对 vector vec 中的元素子集进行排序，并打印 vector 的最后三个元素。

该程序有以下输出:

```
"New York" "Tokyo" "Rio" "Berlin" "Sydney"
"New York" "Berlin" "Rio" "Tokyo" "Sydney"
"Rio" "Tokyo" "Sydney"
```

span 是视图。要处理一个范围的前 n 个元素，也可以使用范围工厂 std::views::counted()，若对一个连续内存中有元素的范围进行迭代器使用，会创建一个具有动态区段的 span:

```
std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9};

auto v = std::views::counted(vec.begin()+1, 3); // span with 2nd to 4th elem of vec
```

9.1.4 使用固定区段的 span

作为具有固定区段的 span 的第一个示例，让我们修改前面的示例，声明具有固定区段的 span:

lib/spanfix.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <ranges>
#include <algorithm>
#include <span>

template<typename T, std::size_t Sz>
void printSpan(std::span<T, Sz> sp)
{
  for (const auto& elem : sp) {
    std::cout << '"' << elem << "\" ";
  }
  std::cout << '\n';
}

int main()
{
  std::vector<std::string> vec{"New York", "Tokyo", "Rio", "Berlin", "Sydney"};

  // define view to first 3 elements:
  std::span<const std::string, 3> sp3{vec.data(), 3};
  std::cout << "first 3: ";
  printSpan(sp3);

  // sort referenced elements:
  std::ranges::sort(vec);
  std::cout << "first 3 after sort(): ";
  printSpan(sp3);

```

```
  // insert a new element:
  // - must reassign the internal array of the vector if it reallocated new memory
  auto oldCapa = vec.capacity();
  vec.push_back("Cairo");
  if (oldCapa != vec.capacity()) {
    sp3 = std::span<std::string, 3>{vec.data(), 3};
  }
  std::cout << "first 3: ";
  printSpan(sp3);

  // let span refer to the last three elements:
  sp3 = std::span<const std::string, 3>{vec.end()-3, vec.end()};
  std::cout << "last 3: ";
  printSpan(sp3);

  // let span refer to the last three elements:
  sp3 = std::span{vec}.last<3>();
  std::cout << "last 3: ";
  printSpan(sp3);
}
```

程序输出如下所示:

```
first 3:              "New York" "Tokyo" "Rio"
first 3 after sort(): "Berlin" "New York" "Rio"
first 3: "Berlin" "New York" "Rio"
last 3: "Sydney" "Tokyo" "Cairo"
last 3: "Sydney" "Tokyo" "Cairo"
```

现在，一步步地回顾程序示例中值得注意的部分。

声明 span

这一次，首先初始化一个由三个固定长度的常量字符串组成的 span:

```
std::vector<std::string> vec{"New York", "Rio", "Tokyo", "Berlin", "Sydney"};

std::span<const std::string, 3> sp3{vec.data(), 3};
```

对于固定区段，可指定元素的类型和大小，需要由开发者来保证元素的数量与 span 的区段相匹配。

若作为第二个参数传递的计数与范围不匹配，则行为未定义。

```
std::span<const std::string, 3> sp3{vec.begin(), 4}; // undefined behavior
```

分配不同的子序列

对于具有固定区段的 span，只能分配具有相同元素数量的新范围。

这一次，只赋值具有三个元素的 span:

```
std::span<const std::string, 3> sp3{vec.data(), 3};
...
sp3 = std::span<const std::string, 3>{vec.end()-3, vec.end()};
```

注意，以下代码将不可编译:

```
std::span<const std::string, 3> sp3{vec.data(), 3};
...
sp3 = std::span{vec}.last(3); // ERROR
```

这样做的原因是，赋值右侧的表达式创建了一个具有动态区段的 span。通过使用 last() 和指定元素数量作为模板参数的语法，则得到一个具有相应固定区段的 span:

```
std::span<const std::string, 3> sp3{vec.data(), 3};
...
sp3 = std::span{vec}.last<3>(); // OK
```

仍可以使用类模板实参推导来为数组元素赋值，甚至可以直接赋值:

```
std::span<const std::string, 3> sp3{vec.data(), 3};
...
std::array<std::string, 3> arr{"Tic", "Tac", "Toe"};
sp3 = std::span{arr}; // OK
sp3 = arr; // OK
```

9.1.5 固定与动态区段

固定和动态区段各有好处。

指定固定的大小使编译器能够在运行时，甚至在编译时检测长度是否违规。例如，不能将元素数目错误的 std::array<> 赋值给区段固定的 span:

```
std::vector vec{1, 2, 3};
std::array arr{1, 2, 3, 4, 5, 6};
std::span<int, 3> sp3{vec};
std::span sp{vec};
sp3 = arr; // compile-time ERROR
sp = arr; // OK
```

具有固定区段的 span 还需要更少的内存，其不需要具有实际大小的成员 (大小是类型的一部分)。

使用动态 span 提供了更大的灵活性:

```
std::span<int> sp; // OK
...
std::vector vec{1, 2, 3, 4, 5};
sp = vec; // OK (span has 5 elements)
sp = {vec.data()+1, 3}; // OK (span has 3 elements)
```

9.2. Span 的缺点

span 指的是外部值的序列，所以具有具有引用语义的类型所具有的常见问题，需要由开发者来保证 span 引用的序列的有效性。

错误出现得很快。若函数 getData() 按值返回一个整数集合 (例如，作为 vector, std::array 或 C 风格的数组)，则以下语句会出现致命的运行时错误:

```
std::span<int, 3> first3{getData()}; // ERROR: reference to temporary object

std::span sp{getData().begin(), 3}; // ERROR: reference to temporary object

sp = getData(); // ERROR: reference to temporary object
```

可能看起来很微妙，例如：使用基于范围的 for 循环:

```
// for the last 3 returned elements:
for (auto s : std::span{arrayOfConst()}.last(3)) // fatal runtime ERROR
```

这段代码会导致未定义的行为，因为基于范围的 for 循环中存在一个 bug，在对临时对象的引用上进行迭代时会使用已经销毁的值 (详细信息请参阅http://wg21.link/p2012)。

编译器可以通过对标准类型的特殊“生命周期检查”来检测这些问题，目前主要的编译器都实现了，但只能检测简单的生存期依赖关系，比如：span 和初始化的对象之间的依赖关系。

此外，必须确保所引用的元素序列保持有效。若程序的其他部分在引用序列的生命周期结束时仍在使用该 span，则可能会出现问题。

若引用对象 (比如指向一个 vector)，这个有效性问题可能在 vector 存在的时候发生:

```
std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8};

std::span sp{vec}; // view to the memory of vec
...
vec.push_back(9); // might reallocate memory
std::cout << sp[0]; // fatal runtime ERROR (referenced array no longer valid)
```

作为一种变通方法，必须用原始 vector 重新初始化 span。

一般来说，使用 span 与使用原始指针和其他视图类型一样危险，需要谨慎。

9.3. Span 的设计要点

设计一个引用元素序列的类型并不容易。需要权衡很多方面，从而做出决定:

- 性能与安全
- const 的正确性
- 可能的隐式和显式类型转换
- 支持类型的要求
- 支持的 API

先说明一下 span 不是什么:

- span 不是容器，可能具有容器的几个特点 (例如，使用 begin() 和 end() 迭代元素的能力)，但由于其引用语义已经存在几个问题:
  - 若 span 是 const，那元素也应该是 const 的吗?
  - 赋值的作用是什么: 给引用的元素分配新的序列或者赋一个新的值?
  - 应该提供 swap() 吗? 它能做什么?
- span 不是指针 (有大小)，提供解引用操作符和-> 没有意义。

std::span 类型是对元素序列的一个非常特定的引用。要正确使用这种类型，有必要具体了解一下。

Barry Revzin 有一篇非常有用的博客文章，我强烈建议大家去阅读:http://brevzin.github.io/c++/2018/12/03/span-best-span/。

C++20 还提供了其他方法来处理对 (子) 序列的引用，例如：子序列。也适用于不存储在连续内存中的序列，使用范围工厂 std::views::counted()，可以让编译器决定哪种类型最适合由开始和大小定义的范围。

9.3.1 span 生命周期的依赖性

由于引用语义，只能在存在底层值序列时下遍历 span，但迭代器并不与创建他们的 span 绑定生命周期。

span 的迭代器不引用创建它们的 span。相反，它们直接引用基础范围，所以 span 是一个租借范围。因此，即使 span 不再存在 (元素序列必须存在)，也可以使用迭代器。但当底层范围不再存在时，迭代器可悬空。

9.3.2 span 的性能

span 的设计以最佳性能为目标。在内部，仅使用指向元素序列的原始指针，但原始指针希望元素按顺序存储在一个内存块中 (否则，原始指针可以在元素来回跳转时计算元素的位置)。所以，span 要求元素存储在连续内存中。

有了这个要求，span 可以为所有类型的视图提供最佳性能。span 不需要任何内存分配，也不附带任何间接内容，使用 span 的唯一开销是构造。编译时使用概念检查引用序列是否在连续内存中

有其元素，初始化序列或赋值新序列时，迭代器必须满足 std::consecuous_iterator 概念，容器或范围必须满足 std::ranges::consecuous_range 和 std::ranges::sized_range 两个概念。

因为 span 在内部只是一个指针和一个大小，所以复制成本非常低。出于这个原因，应该更倾向于按值传递 span，而非通过 const 引用传递 (然而，对于处理容器和视图的泛型函数来说，这是一个问题)。

类型擦除

span 使用指向内存的原始指针执行元素访问，span 类型会擦除元素存储位置的信息。指向 vector 元素的 span 操作与指向数组元素的 span 操作具有相同的类型 (前提是它们具有相同的区段):

```
std::array arr{1, 2, 3, 4, 5};
std::vector vec{1, 2, 3, 4, 5};

std::span<int> vecSpanDyn{vec};
std::span<int> arrSpanDyn{arr};
std::same_as<decltype(arrSpanDyn), decltype(vecSpanDyn)> // true
```

但 span 的类模板参数推导从数组推导出固定区段，从 vector 推导出动态区段:

```
std::array arr{1, 2, 3, 4, 5};
std::vector vec{1, 2, 3, 4, 5};
std::span arrSpan{arr}; // deduces std::span<int, 5>
std::span vecSpan{vec}; // deduces std::span<int>
std::span<int, 5> vecSpan5{vec};

std::same_as<decltype(arrSpan), decltype(vecSpan)> // false
std::same_as<decltype(arrSpan), decltype(vecSpan5)> // true
```

span 与子范围

对元素连续存储的要求是与子范围的主要区别，子范围也是 C++20 引入的。在内部，子范围仍然使用迭代器，因此可以引用所有类型的容器和范围，但这可能会导致更多的开销。

此外，span 不需要对其引用的类型支持迭代器，可以传递 data() 成员函数以访问元素序列的类型。

9.3.3 span 的 const 正确性

span 是具有引用语义的视图，其行为就像指针: 若 span 是 const，并不意味着该 span 所引用的元素也是 const。

并且，可以对 const span 的元素进行写访问 (前提是这些元素不是 const):

```
std::array a1{1, 2, 3, 4, 5, 6,7 ,8, 9, 10};
std::array a2{0, 8, 15};
```

```
3
4  const std::span<int> sp1{a1}; // span/view is const
5  std::span<const int> sp2{a1}; // elements are const
6
7  sp1[0] = 42; // OK
8  sp2[0] = 42; // ERROR
9
10 sp1 = a2; // ERROR
11 sp2 = a2; // OK
```

只要 std::span<> 的元素没有声明为 const，即使对于 const span，也会有一些操作提供对这些元素的写访问 (遵循普通容器的规则):

- operator[], first(), last()
- data()
- begin(), end(), rbegin(), rend()
- std::cbegin(), std::cend(), std::crbegin(), std::crend()
- std::ranges::cbegin(), std::ranges::cend(), std::ranges::crbegin(), std::ranges::crend()

是的，所有设计用来确保元素为 const 的 c* 函数，都可使用 std::span 替代。

例如：

```
1  template<typename T>
2  void modifyElemsOfConstColl (const T& coll)
3  {
4    coll[0] = {}; // OK for spans, ERROR for regular containers
5
6    auto ptr = coll.data();
7    *ptr = {}; // OK for spans, ERROR for regular containers
8
9    for (auto pos = std::cbegin(coll); pos != std::cend(coll); ++pos) {
10     *pos = {}; // OK for spans, ERROR for regular containers
11   }
12 }
13
14 std::array arr{1, 2, 3, 4, 5, 6, 7 ,8, 9, 10};
15
16 modifyElemsOfConstColl(arr); // ERROR: elements are const
17 modifyElemsOfConstColl(std::span{arr}); // OOPS: compiles and modifies elements of a1
```

这里的问题不在于破坏了 std::span; 问题是，像 std::cbegin() 和 std::ranges::cbegin() 这样的函数破坏了具有引用语义的集合 (例如视图)。

为了确保函数只接受不能以这种方式修改元素的序列，可以要求 const 容器的 begin() 返回一个指向 const 元素的迭代器:

```
1  template<typename T>
2  void ensureReadOnlyElemAccess (const T& coll)
```

```
requires std::is_const_v<std::remove_reference_t<decltype(*coll.begin())>>
{
  ...
}
```

std::cbegin() 和 std::ranges::cbegin() 提供写访问，这是标准化 C++20 之后讨论的问题。提供 cbegin() 和 cend() 的全部意义在于确保元素在迭代时不能被修改。最初，span 确实为 const_iterator、cbegin() 和 cend() 类型提供了成员，以确保不能修改元素。但在 C++20 完成前的最后一刻，std::cbegin() 仍然迭代可变元素 (std::ranges::cbegin() 也有同样的问题)。然而，不是修复 std::cbegin() 和 std::ranges::cbegin()，而是删除了 span 中 const 迭代器的成员 (参见http://wg21.link/lwg3320)，这使得问题更加严重，因为在 C++20 中，现在没有简单的方法对 span 进行只读迭代，除非元素为 const。std::ranges::cbegin() 将在 C++23 中修复 (参见http://wg21.link/p2278)，但 std::cbegin() 仍然会遭到破坏 (叹气)。

9.3.4 泛型代码中使用 span 作为参数

如前所述，可以通过以下声明实现所有 span 的泛型函数:

```
template<typename T, std::size_t Sz>
void printSpan(std::span<T, Sz> sp);
```

这甚至适用于具有动态范围的 span，可使用特殊值 std::dynamic_extent 作为大小。

实现中，可以按以下方式处理固定与动态区段的区别:

lib/spanprint.hpp

```
#ifndef SPANPRINT_HPP
#define SPANPRINT_HPP

#include <iostream>
#include <span>

template<typename T, std::size_t Sz>
void printSpan(std::span<T, Sz> sp)
{
  std::cout << '[' << sp.size() << " elems";
  if constexpr (Sz == std::dynamic_extent) {
    std::cout << " (dynamic)";
  }
  else {
    std::cout << " (fixed)";
  }
  std::cout << ':';
  for (const auto& elem : sp) {
    std::cout << ' ' << elem;
  }
  std::cout << "]\n";
```

```
  }
#endif // SPANPRINT_HPP
```

还可以考虑将元素类型声明为 const:

```
printSpan(vec); // ERROR: template type deduction doesn't work
printSpan(std::span{vec}); // OK
printSpan<int, std::dynamic_extent>(vec); // OK (provided it is a vector of ints)
```

因此，不应将 std::span<> 用作处理存储在连续内存中元素序列泛型函数的词汇表类型。

出于性能考虑，可以这样做:

```
template<typename E>
void processSpan(std::span<typename E>) {
  ... // span specific implementation
}

template<typename T>
void print(const T& t) {
  if constexpr (std::ranges::contiguous_range<T> t) {
    processSpan<std::ranges::range_value_t<T>>(t);
  }
  else {
    ... // generic implementations for all containers/ranges
  }
}
```

使用 span 作为范围和视图

span 是视图，所以满足 std::ranges::view[最初的 C++20 标准确实要求视图必须有一个默认构造函数，而固定的 span 则不是这样，但这一要求后来删除了http://wg21.link/P2325R3。]，span 可以用于范围和视图的所有算法和函数中。在讨论所有视图细节的那一章中，列出了 span 的特定于视图的属性。

span 的一个属性是租借范围，所以迭代器的生命周期不依赖于 span，所以可以在生成迭代器的算法中使用临时 span 作为范围:

```
std::vector<int> coll{25, 42, 2, 0, 122, 5, 7};
auto pos1 = std::ranges::find(std::span{coll.data(), 3}, 42); // OK
std::cout << *pos1 << '\n';
```

但若 span 是引用的临时对象，则会出现错误。即使返回已销毁的临时对象的迭代器，以下代码也可以编译通过:

```
auto pos2 = std::ranges::find(std::span{getData().data(), 3}, 42);
std::cout << *pos2 << '\n'; // runtime ERROR
```

9.4. Span 的操作

本节详细介绍 span 的类型和操作。

9.4.1 span 的操作和成员类型概述

表“span 的操作”列出了为 span 提供的所有操作。

需要注意的是不支持的操作:

- 比较 (不包括 ==)
- swap()
- assign()
- swap()
- at()
- I/O 操作符
- cbegin(), cend(), crbegin(), crend()
- 哈希
- 用于结构化绑定的类元组 API

所以，span 既不是容器 (传统的 C++ STL 意义上)，也不是常规类型。

关于静态成员和成员类型，span 提供了容器的常用成员 (const_iterator 除外) 以及两个特殊成员:element_type 和 extent(参见表 span 的 static 和 type 成员)。

注意，std::value_type 不是指定的元素类型 (std::array 和其他几个类型的 value_type 通常是)，它是去掉了 const 和 volatile 的元素类型。

操作	效果
构造函数	创建或复制一个 span
析构函数	销毁一个 span
=	分配一个新的值序列
empty()	返回 span 是否为空
size()	返回元素的个数
size_bytes()	返回所有元素所使用的内存大小
[]	访问元素
front(),back()	访问第一个或最后一个元素
begin(), end()	提供迭代器支持 (不支持 const_iterator)
rbegin(), rend()	提供常量反向迭代器支持
first(n)	返回具有前 n 个元素动态范围的子 span
first<n>()	返回前 n 个元素的固定范围的子 span
last(n)	返回最后 n 个元素的动态范围的子 span
last<n>()	返回最后 n 个元素的固定范围的子 span
subspan(offs)	返回一个跳过第一个 off 元素的动态范围的子 span
subspan(offs, n)	在 off 元素之后返回一个动态范围为 n 的子 span

subspan<offs>()	跳过第一个 off 元素，返回具有相同范围的子 span
subspan<offs, n>()	在 off 元素之后返回一个固定范围为 n 的子 span
data()	返回指向元素的原始指针
as_bytes()	以只读 std::bytes 的 span 返回元素的内存
as_writeable_bytes()	以可写 std::bytes 的 span 返回元素的内存

表 9.1 span 的操作

成员	效果
extent	若大小不同，则表示元素数或 std::dynamic_extent
size_type	范围类型 (总是 std::size_t)
difference_type	指向元素指针的不同类型 (总是 std::difference_type)
element_type	指定的元素类型
pointer	指向元素的指针类型
const_pointer	用于对元素进行只读访问的指针类型
reference	元素引用的类型
const_reference	对元素进行只读访问的引用类型
iterator	元素迭代器的类型
reverse_iterator	元素的反向迭代器类型
value_type	不带 const 或 volatile 的元素类型

表 9.2 span 的静态和类型成员

9.4.2 构造函数

只有当它具有动态区段或范围为 0 时，才提供默认构造函数:

```
std::span<int> sp0a; // OK
std::span<int, 0> sp0b; // OK
std::span<int, 5> sp0c; // compile-time ERROR
```

若此初始化有效，则 size() 为 0，data() 为 nullptr。

原则上，可以初始化数组的 span，以哨兵 (end 迭代器) 开始的 span 和以 size 开始的 span。若 beg 指向连续内存中的元素，视图 std::views::counted(beg, sz) 会使用后者，还支持类模板参数推导。

当使用原始数组或 std::array<> 初始化 span 时，将推导出具有固定范围的 span(除非指定了元素类型):

```
int a[10] {};
std::array arr{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};

std::span sp1a{a}; // std::span<int, 10>
```

```
std::span sp1b{arr}; // std::span<int, 15>
std::span<int> sp1c{arr}; // std::span<int>
std::span sp1d{arr.begin() + 5, 5}; // std::span<int>
auto sp1e = std::views::counted(arr.data() + 5, 5); // std::span<int>
```

使用 std::vector<> 初始化 span 时，除非显式指定了 span 的大小，否则将推导出具有动态区段的 span:

```
std::vector vec{1, 2, 3, 4, 5};
std::span sp2a{vec}; // std::span<int>

std::span<int> sp2b{vec}; // std::span<int>
std::span<int, 2> sp2c{vec}; // std::span<int, 2>
std::span<int, std::dynamic_extent> sp2d{vec}; // std::span<int>
std::span<int, 2> sp2e{vec.data() + 2, 2}; // std::span<int, 2>
std::span sp2f{vec.begin() + 2, 2}; // std::span<int>
auto sp2g = std::views::counted(vec.data() + 2, 2); // std::span<int>
```

若是使用右值 (临时对象) 初始化 span，则元素必须为 const:

```
std::span sp3a{getArrayOfInt()}; // ERROR: rvalue and not const
std::span<int> sp3b{getArrayOfInt()}; // ERROR: rvalue and not const
std::span<const int> sp3c{getArrayOfInt()}; // OK
std::span sp3d{getArrayOfConstInt()}; // OK

std::span sp3e{getVectorOfInt()}; // ERROR: rvalue and not const
std::span<int> sp3f{getVectorOfInt()}; // ERROR: rvalue and not const
std::span<const int> sp3g{getVectorOfInt()}; // OK
```

使用返回的临时集合初始化 span 可能会导致致命的运行时错误。例如，永远不该使用基于范围的 for 循环来迭代 (直接初始化的)span:

```
for (auto elem : std::span{getCollOfConst()}) ... // fatal runtime error

for (auto elem : std::span{getCollOfConst()}.last(2)) ... // fatal runtime error

for (auto elem : std::span<const Type>{getColl()}) ... // fatal runtime error
```

问题在于，在迭代为临时对象返回的引用时，基于范围的 for 循环会导致未定义的行为，因为临时对象在循环开始内部迭代之前销毁。这是一个 C++ 标准委员会多年来一直不愿意修复的 bug(参见http://wg21.link/p2012)。

作为一种解决方案，可以使用初始化的新的基于范围的 for 循环:

```
for (auto&& coll = getCollOfConst(); auto elem : std::span{coll}) ... // OK
```

无论用一个迭代器和一个长度初始化 span，还是用两个定义有效范围的迭代器初始化 span，迭代器都必须指向连续内存中的元素 (满足 std::consecuous_iterator 的概念):

```
std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::span<int> sp6a{vec}; // OK, refers to all elements
std::span<int> sp6b{vec.data(), vec.size()}; // OK, refers to all elements
std::span<int> sp6c{vec.begin(), vec.end()}; // OK, refers to all elements
std::span<int> sp6d{vec.data(), 5}; // OK, refers to first 5 elements
std::span<int> sp6e{vec.begin()+2, 5}; // OK, refers to elements 3 to 7 (including)
std::list<int> lst{ ... };
std::span<int> sp6f{lst.begin(), lst.end()}; // compile-time ERROR
```

若 span 具有固定的区段，则必须匹配所传递范围中的元素数量。所以，编译器不能在编译时检查:

```
std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::span<int, 10> sp7a{vec}; // OK, refers to all elements
std::span<int, 5> sp7b{vec}; // runtime ERROR (undefined behavior)
std::span<int, 20> sp7c{vec}; // runtime ERROR (undefined behavior)
std::span<int, 5> sp7d{vec, 5}; // compile-time ERROR
std::span<int, 5> sp7e{vec.begin(), 5}; // OK, refers to first 5 elements
std::span<int, 3> sp7f{vec.begin(), 5}; // runtime ERROR (undefined behavior)
std::span<int, 8> sp7g{vec.begin(), 5}; // runtime ERROR (undefined behavior)
std::span<int, 5> sp7h{vec.begin()}; // compile-time ERROR
```

也可以使用原始数组或 std::array 直接创建和初始化 span，由于元素数量不合法而导致的一些运行时错误将成为编译时错误:

```
int raw[10];
std::array arr{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::span<int> sp8a{raw}; // OK, refers to all elements
std::span<int> sp8b{arr}; // OK, refers to all elements
std::span<int, 5> sp8c{raw}; // compile-time ERROR
std::span<int, 5> sp8d{arr}; // compile-time ERROR
std::span<int, 5> sp8e{arr.data(), 5}; // OK
```

也就是说: 要么传递一个整体上包含顺序元素的容器，要么传递两个参数来指定元素的初始范围。通常情况下，元素的数量必须匹配指定的固定范围。

带隐式转换的构造

span 必须具有它们引用的序列元素的元素类型。不支持转换 (甚至是隐式的标准转换)，但允许使用 const 等其他限定符，这也适用于复制构造函数:

```cpp
std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::span<const int> sp9a{vec}; // OK: element type with const
std::span<long> sp9b{vec}; // compile-time ERROR: invalid element type
std::span<int> sp9c{sp9a}; // compile-time ERROR: removing constness
std::span<const long> sp9d{sp9a}; // compile-time ERROR: different element type
```

为了允许容器引用用户定义容器的元素，这些容器必须发出信号，表明其迭代器要求所有元素都位于连续内存中，其必须满足 continuous_iterator 的概念。

构造函数还允许在 span 之间进行以下类型转换:

- 具有固定区段的 span 将转换为具有相同的固定区段和附加限定符的 span。
- 具有固定区段的 span 转换为具有动态区段的 span。
- 若当前区段适合，则具有动态区段的 span 将转换为具有固定区段的 span。

使用条件显式时，只有固定范围 span 的构造函数是显式的。若必须转换初始值，则无法进行复制初始化 (使用 a =):

```cpp
std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

std::span<int> spanDyn{vec.begin(), 5}; // OK
std::span<int> spanDyn2 = {vec.begin(), 5}; // OK
std::span<int, 5> spanFix{vec.begin(), 5}; // OK
std::span<int, 5> spanFix2 = {vec.begin(), 5}; // ERROR
```

因此，只有在转换为具有动态或相同固定范围的 span 时，才隐式支持转换:

```cpp
void fooDyn(std::span<int>);
void fooFix(std::span<int, 5>);

fooDyn({vec.begin(), 5}); // OK
fooDyn(spanDyn); // OK
fooDyn(spanFix); // OK
fooFix({vec.begin(), 5}); // ERROR
fooFix(spanDyn); // ERROR
fooFix(spanFix); // OK

spanDyn = spanDyn; // OK
spanDyn = spanFix; // OK
spanFix = spanFix; // OK
spanFix = spanDyn; // ERROR
```

9.4.3 子 span 的操作

创建子 span 的成员函数可以创建动态或固定区段的 span，将大小作为调用参数传递通常会产生具有动态区段的 span。将大小作为模板参数传递通常会产生具有固定范围的 span，成员函数 first()

和 last() 至少总是这样:

```
std::vector vec{1.1, 2.2, 3.3, 4.4, 5.5};
std::span spDyn{vec};

auto sp1 = spDyn.first(2); // first 2 elems with dynamic extent
auto sp2 = spDyn.last(2); // last 2 elems with dynamic extent
auto sp3 = spDyn.first<2>(); // first 2 elems with fixed extent
auto sp4 = spDyn.last<2>(); // last 2 elems with fixed extent

std::array arr{1.1, 2.2, 3.3, 4.4, 5.5};
std::span spFix{arr};

auto sp5 = spFix.first(2); // first 2 elems with dynamic extent
auto sp6 = spFix.last(2); // last 2 elems with dynamic extent
auto sp7 = spFix.first<2>(); // first 2 elems with fixed extent
auto sp8 = spFix.last<2>(); // last 2 elems with fixed extent
```

但对于 subspan()，结果有时可能令人惊讶。传递调用参数总是会产生具有动态区段的 span:

```
std::vector vec{1.1, 2.2, 3.3, 4.4, 5.5};
std::span spDyn{vec};

auto s1 = spDyn.subspan(2); // 3rd to last elem with dynamic extent
auto s2 = spDyn.subspan(2, 2); // 3rd to 4th elem with dynamic extent
auto s3 = spDyn.subspan(2, std::dynamic_extent); // 3rd to last with dynamic extent

std::array arr{1.1, 2.2, 3.3, 4.4, 5.5};
std::span spFix{arr};

auto s4 = spFix.subspan(2); // 3rd to last elem with dynamic extent
auto s5 = spFix.subspan(2, 2); // 3rd to 4th elem with dynamic extent
auto s6 = spFix.subspan(2, std::dynamic_extent); // 3rd to last with dynamic extent
```

然而，当传递模板参数时，结果可能与期望不同:

```
std::vector vec{1.1, 2.2, 3.3, 4.4, 5.5};
std::span spDyn{vec};

auto s1 = spDyn.subspan<2>(); // 3rd to last with dynamic extent
auto s2 = spDyn.subspan<2, 2>(); // 3rd to 4th with fixed extent
auto s3 = spDyn.subspan<2, std::dynamic_extent>(); // 3rd to last with dynamic extent

std::array arr{1.1, 2.2, 3.3, 4.4, 5.5};
std::span spFix{arr};

auto s4 = spFix.subspan<2>(); // 3rd to last with fixed extent
auto s5 = spFix.subspan<2, 2>(); // 3rd to 4th with fixed extent
auto s6 = spFix.subspan<2, std::dynamic_extent>(); // 3rd to last with fixed extent
```

9.5. 附注

span 是由 Lukasz Mendakiewicz 和 Herb Sutter 在http://wg21.link/n3851中以 array_views 的形式提出的，最早是由 Neil MacIntosh 在http://wg21.link/p0122r0中提出。最终接受的提案是由 Neil MacIntosh 和 Stephan T. Lavavej 在http://wg21.link/p0122r7中提出的。

后来添加了一些修改，例如删除 Tony Van Eerd 在http://wg21.link/P1085R2中提出的所有比较操作符，以及删除支持 const_iteratorhttp://wg21.link/lwg3320的决议。

C++20 发布后，视图的定义发生了变化，因此 span 现在总是视图 (参见http://wg21.link/p2325r3)。

第 10 章 格式化输出

C++ IOStream 库只提供了不方便和有限的方式来支持格式化输出 (指定字段宽度，填充字符等)，所以格式化的输出通常仍然使用 sprintf() 这样的函数。

C++20 引入了一个用于格式化输出的新库，本章将对此进行描述。其可以方便地规范格式化属性，并且具有可扩展性。

10.1. 格式化输出的例子

在了解细节之前，先看一些例子。

10.1.1 使用 std::format()

用于应用程序程序员的格式化库的基本功能是 std::format()，其允许它们根据一对大括号内指定的格式，将要用传入参数的值填充的格式化字符串组合在一起。一个简单的例子是使用每个传入参数的值:

```
#include <format>

std::string str{"hello"};
...
std::cout << std::format("String '{}' has {} chars\n", str, str.size());
```

头文件 <format> 中定义的函数 std::format() 接受一个格式字符串 (在编译时已知的字符串字面值、字符串或字符串视图)，其中 {…} 表示下一个参数的值 (这里使用其类型的默认格式)，会生成一个 std::string，并为其分配内存。

第一个示例的输出如下:

```
String ’ hello’ has 5 chars
```

右大括号后面的可选整数值指定参数的索引，以便可以以不同的顺序处理参数或多次使用:

```
std::cout << std::format("{1} is the size of string '{0}'\n", str, str.size());
```

输出如下:

```
5 is the size of string ’ hello’
```

不必显式指定参数的类型，可以很容易地在泛型代码中使用 std::format()。考虑下面的例子:

```
void print2(const auto& arg1, const auto& arg2)
{
  std::cout << std::format("args: {} and {}\n", arg1, arg2);
}
```

若按如下方式调用此函数:

```
print2(7.7, true);
print2("character:", '?');
```

输出如下:

```
args: 7.7 and true
args: character: and ?
```

若支持格式化输出，格式化甚至可以用于用户定义类型。chrono 库的格式化输出就是一个例子：

```
print2(std::chrono::system_clock::now(), std::chrono::seconds{13});
```

可能有以下输出:

```
args: 2022-06-19 08:46:45.3881410 and 13s
```

对于自定义类型，需要一个格式化器，稍后将对此进行描述。

在冒号后面格式化的占位符中，可以指定所传递参数格式化的详细信息。例如，定义一个字段宽度:

```
std::format("{:7}", 42) // yields "  42"
std::format("{:7}", 42.0) // yields "  42"
std::format("{:7}", 'x') // yields "x  "
std::format("{:7}", true) // yields "true "
```

不同的类型有不同的默认对齐方式。对于 bool 类型，输出值 false 和 true，而不是像使用 << 操作符的 iostream 输出那样输出 0 和 1。

也可以显式地指定对齐方式 (< 为左，^ 为中，> 为右) 并指定一个填充字符:

```
std::format("{:*<7}", 42) // yields "42*****"
std::format("{:*>7}", 42) // yields "*****42"
std::format("{:*^7}", 42) // yields "**42***"
```

其他的格式化规范可以强制使用特定的表示法、特定的精度 (或将字符串限制为一定的大小)、填充字符或正号:

```
std::format("{:7.2f} Euro", 42.0) // yields "  42.00 Euro"
std::format("{:7.4}", "corner") // yields "corn "
```

通过使用实参的位置，可以以多种形式输出值。例如:

```
std::cout << std::format("'{0}' has value {0:02X} {0:+4d} {0:03o}\n", '?');
std::cout << std::format("'{0}' has value {0:02X} {0:+4d} {0:03o}\n", 'y');
```

输出'?' 的十六进制、十进制和八进制值和'y' 使用实际的字符集，可能有如下输出:

```
' ?'  has value 3F +63 077
' y'  has value 79 +121 171
```

10.1.2 使用 std::format_to_n()

与其他格式化方式相比，std::format() 的实现具有相当好的性能，但必须为结果字符串分配内存。为了节省时间，可以使用 std::format_to_n()，将写入预分配的字符数组。还必须指定要写入的缓冲区及其大小:

```
char buffer[64];
...
auto ret = std::format_to_n(buffer, std::size(buffer) - 1,
              "String '{}' has {} chars\n", str, str.size());
*(ret.out) = '\0';
```

或:

```
std::array<char, 64> buffer;
...
auto ret = std::format_to_n(buffer.begin(), buffer.size() - 1,
              "String '{}' has {} chars\n", str, str.size());
*(ret.out) = '\0'; // write trailing null terminator
```

注意，std::format_to_n() 不会写入尾随的空结束符，但返回值保存了处理此问题的所有信息。其是一个 std::format_to_n_result 类型的数据结构，有两个成员:

- out 对于第一个未写字符的位置
- size 在不将其截断为传递的大小的情况下将写入的字符数。

因此，在 ret.out 指向的末尾存储一个空终止符。注意，我们只将 buffer.size()-1 传递给 std::format_to_n()，以确保为末尾的空结束符提供内存:

```
auto ret = std::format_to_n(buffer.begin(), buffer.size() - 1, ... );
*(ret.out) = '\0';
```

或者，可以用 {} 初始化缓冲区，以确保所有字符都用空终止符初始化。

若大小不适合该值，则不是错误，写入的值会删除。例如:

```
std::array<char, 5> mem{};
std::format_to_n(mem.data(), mem.size()-1, "{}", 123456.78);
std::cout << mem.data() << '\n';
```

输出如下:

```
1234
```

10.1.3 使用 std::format_to()

格式化库还提供了 std::format_to()，为格式化的输出写入无限数量的字符。在有限的内存中使用这个函数有风险，因为如果值需要太多的内存，就会有未定义行为，但通过使用输出流缓冲区迭代器，也可以安全地使用它写入流:

```
std::format_to(std::ostreambuf_iterator<char>{std::cout},
        "String '{}' has {} chars\n", str, str.size());
```

通常，std::format_to() 接受任意字符的输出迭代器。例如，也可以使用尾插入器将字符追加到字符串中:

```
std::string s;
std::format_to(std::back_inserter(s),
        "String '{}' has {} chars\n", str, str.size());
```

辅助函数 std::back_inserter() 创建一个对象，为每个字符调用 push_back()。std::format_to() 的实现可以识别传递的回插入迭代器，并一次为某些容器写入多个字符，这样仍然具有良好的性能。

10.1.4 使用 std::formatted_size()

要提前知道格式化输出将写入多少字符 (不写入任何字符)，可以使用 std::formatted_size()。例如:

```
auto sz = std::formatted_size("String '{}' has {} chars\n", str, str.size());
```

这将允许保留足够的内存，或再次检查所保留的内存是否足够。

10.2. 格式库的性能

仍然使用 sprintf() 的一个原因是，其性能明显优于使用输出字符串流或 std::to_string()。格式化库的设计就是为了更好地完成这一任务，其格式化速度至少应该和 sprintf() 一样快，甚至更快。

目前的 (草案) 实现表明，相同甚至更好的性能是可能的。粗略的测量表明

- std::format() 应该和 sprintf() 一样快，甚至更好。
- std::format_to() 和 std::format_to_n() 应该有更好的性能。

编译器通常可以在编译时检查格式字符串，这确实有很大帮助。有助于避免格式化错误，并获得更好的性能。

性能最终取决于特定平台上格式化库的实现质量。例如，在编写本书时，对于 Visual C++，/utf-8 选项显著提高了格式化性能，开发者应该自己确定性能。程序 format/formatperf10.cpp 可能会给您一些关于相关平台情况的提示。

10.2.1 使用 std::vformat() 和 vformat_to()

为了支持这一目标，C++20 标准化之后，对格式化库进行了一个重要的修复 (参见http://wg21.link/p2216r3)。通过此修复，函数 std::format()、std::format_to() 和 std::format_to_n() 要求格式字符串是编译时值，所以必须传递一个字符串字面值或 constexpr 字符串。例如:

```
const char* fmt1 = "{}\n"; // runtime format string
std::cout << std::format(fmt1, 42); // compile-time ERROR: runtime format

constexpr const char* fmt2 = "{}\n"; // compile-time format string
std::cout << std::format(fmt2, 42); // OK
```

因此，无效的格式规范成为编译时错误:

```
std::cout << std::format("{:7.2f}\n", 42); // compile-time ERROR: invalid format

constexpr const char* fmt2 = "{:7.2f}\n"; // compile-time format string
std::cout << std::format(fmt2, 42); // compile-time ERROR: invalid format
```

当然，应用有时必须在运行时计算格式细节 (例如根据传递的值计算最佳宽度)，必须使用 std::vformat() 或 std::vformat_to()，并使用 std::make_format_args() 将所有参数传递给这些函数:

```
const char* fmt3 = "{} {}\n"; // runtime format
std::cout << std::vformat(fmt3, std::make_format_args(42, 1.7)); // OK
```

若使用了运行时格式字符串，并且所传递的参数的格式无效，则调用会抛出 std::format_error 类型的运行时错误:

```
const char* fmt4 = "{:7.2f}\n";
std::cout << std::vformat(fmt4, std::make_format_args(42)); // runtime ERROR:
// throws std::format_error excep
```

10.3. 格式化输出的详情

本节来详细介绍格式化的语法。

10.3.1 格式字符串的通用格式

指定参数格式的一般方法是传递一个格式字符串，该字符串可以具有由 {…指定的替换字段。} 和字符。所有其他字符按原样打印。要打印字符 {和}，请使用 {{和}}。

例如:

```
std::format("With format {{}}: {}", 42); // yields "With format {}: 42"
```

替换字段可以有一个索引来指定参数，并在冒号后面有一个格式说明符:

- {}
  使用带有默认格式的下一个参数
- {n}
  使用默认格式的第 n 个参数 (第一个参数索引为 0)
- {:fmt}
  使用根据 fmt 格式化的下一个参数
- {n:fmt}
  使用根据 fmt 格式化的第 n 个参数

可以不使用索引指定的任何参数，也可以使用全部参数:

```
std::format("{}: {}", key, value); // OK
std::format("{1}: {0}", value, key); // OK
std::format("{}: {} or {0}", value, key); // ERROR
```

将忽略其他参数。

格式说明符的语法取决于所传递参数的类型。

- 对于算术类型、字符串和原始指针，格式化库本身已经定义了标准格式。
- C++20 还指定了时间类型 (持续时间、时间点和日历类型) 的标准格式。

10.3.2 标准格式说明符

标准格式说明符有以下格式 (每个说明符都是可选的):

```
fill align sign # 0 width .prec L type
```

- fill 是将值填充到宽度 (默认: 空格) 的字符，只能在同时指定 align 时指定。
- align 为
  - < 左对齐
  - > 右对齐
  - ^ 居中

  默认对齐方式取决于类型。
- sign 为
  - - 只有负值的负号 (默认)
  - + 正号或负号
  - space 负号或空格
- # 切换到某些符号的替代形式:
  - 为整数值的二进制、八进制和十六进制表示法写入 0b、0 和 0x 这样的前缀。
  - 强制浮点表示法总是写一个点。
- 宽度前的 0 用零填充算术值。
- width 指定最小字段宽度。
- 点后的 prec 指定精度:

  - 对于浮点类型，指定在点后面或总共打印多少位数字 (取决于表示法)。
  - 对于字符串类型，指定从字符串处理的最大字符数。
- L 激活依赖格式 (这可能会影响算术类型和 bool) 的格式
- type 指定格式化的通用表示法，将字符打印为整数值 (反之亦然) 或选择浮点值的通用表示法。

10.3.3 宽度、精度和填充字符

对于所有打印的值，冒号后面的正整数值 (不带前导点) 指定了整个值输出的最小字段宽度 (包括符号等)。它可以与对齐规范一起使用:

例如:

```
std::format("{:7}", 42);      // yields "     42"
std::format("{:7}", "hi");    // yields "hi     "
std::format("{:^7}", "hi");   // yields "  hi   "
std::format("{:>7}", "hi");   // yields "     hi"
```

还可以指定填充零和填充字符。填充 0 只能用于算术类型 (char 和 bool 除外)，若果指定了对齐方式，则忽略填充 0:

```
std::format("{:07}", 42);     // yields "0000042"
std::format("{:^07}", 42);    // yields "  42   "
std::format("{:>07}", -1);    // yields "     -1"
```

填充 0 不同于一般的填充字符，可以在冒号之后立即指定 (在对齐的前面):

```
std::format("{:^07}", 42);    // yields "  42   "
std::format("{:0^7}", 42);    // yields "0042000"
std::format("{:07}", "hi");   // invalid (padding 0 not allowed for strings)
std::format("{:0<7}", "hi"); // yields "hi00000"
```

精度用于浮点类型和字符串:

- 对于浮点类型，可以指定一个精度 (默认值为 6):

```
std::format("{}", 0.12345678);          // yields "0.12345678"
std::format("{:.5}", 0.12345678);       // yields "0.12346"
std::format("{:10.5}", 0.12345678);     // yields "   0.12346"
std::format("{:^10.5}", 0.12345678);    // yields " 0.12346  "
```

根据浮点表示法的不同，精度可能适用于整个值，也可能适用于点之后的数字。

- 对于字符串，可以用来指定最大字符数:

```
std::format("{}", "counterproductive");            // yields "counterproductive"
std::format("{:20}", "counterproductive");         // yields "counterproductive
↪   "
std::format("{:.7}", "counterproductive");         // yields "counter"
```

```
4  std::format("{:20.7}", "counterproductive");       // yields "counter
   ↪  "
5  std::format("{:^20.7}", "counterproductive");      // yields "        counter
   ↪  "
```

宽度和精度本身可以作为参数:

```
1  int width = 10;
2  int precision = 2;
3  for (double val : {1.0, 12.345678, -777.7}) {
4    std::cout << std::format("{:+{}.{}f}\n", val, width, precision);
5  }
```

输出如下:

```
    +1.00
   +12.35
  -777.70
```

在运行时指定最小字段宽度为 10，并且保留小数点后面的两个数字 (使用固定的表示法)。

10.3.4 格式/类型说明符

通过指定格式或类型说明符，可以强制为整型、浮点型和原始指针使用不同的符号。

整型的说明符

整型的表格式化选项列出了整型 (包括 bool 和 char) 可能的格式化类型选项。[根据http://wg21.link/lwg3648，对 bool 的说明符 c 的支持可能是一个错误，将被删除。]

格式	42	'@'	true	意义
none	42	@	true	默认 format
d	42	64	1	十进制 notation
b/B	101010	1000000	1	二进制 notation
#b	0b101010	0b1000000	0b1	带前缀的二进制表示法
#B	0B101010	0B1000000	0B1	带前缀的二进制表示法
o	52	100	1	八进制表示法
x	2a	40	1	十六进制表示法
X	2A	40	1	十六进制表示法
#x	0x2a	0x40	0x1	带前缀的十六进制表示法
#X	0X2A	0X40	0X1	带前缀的十六进制表示法
c	*	@	'\1'	作为值的字符
s	非法	非法	true	布尔值作为字符串

表 10.1 整型的格式化选项

例如:

```
std::cout << std::format("{:#b} {:#b} {:#b}\n", 42, '@', true);
```

输出:

```
0b101010 0b1000000 0b1
```

注意事项如下:

- 默认的符号是:
  - d (十进制) 的整数类型
  - c (作为字符) 用于字符类型
  - s (作为字符串) 用于 bool 类型
- 若在表示法之后指定 L，则使用与语言环境相关的布尔值字符序列，和与语言环境相关的千位分隔符和小数点字符来表示算术值。

浮点类型的说明符

表浮点类型的格式化选项，列出了浮点类型可能的格式化类型选项。

例如:

```
std::cout << std::format("{0} {0:#} {0:#g} {0:e}\n", -1.0);
```

输出:

```
-1 -1. -1.00000 -1.000000e+00
```

注意，传递整数-1 会导致格式错误。

格式	-1.0	0.0009765625	1785856.0	意义
none	-1	0.0009765625	1.785856e+06	默认格式
#	-1.	0.0009765628	1.785856e+06	强制十进制小数点
f/F	-1.000000	0.000977	1785856.000000	固定小数点位置 (默认点后精度:6)
g	-1	0.000976562	1.78586e+06	固定或指数表示法 (默认全精度:6)
G	-1	0.000976562	1.78586E+06	固定或指数表示法 (默认全精度:6)
#g	-1.00000	0.000976562	1.78586e+06	固定或指数符号 (强制点和零)
#G	-1.00000	0.000976562	1.78586E+06	固定或指数符号 (强制点和零)
e	-1.00000e00	9.765625e-04	1.7858560e+06	指数表示法 (默认点后精度:6)
E	-1.00000E00	9.765625E-04	1.7858560E+06	指数表示法 (默认点后精度:6)

a	-1p+0	1p-10	1.b4p+20	十六进制。浮点表示法
A	-1P+0	1P-10	1.B4P+20	十六进制。浮点表示法
#a	-1.p+0	1.p-10	1.b4p+20	十六进制。浮点表示法
#A	-1.P+0	1.P-10	1.B4P+20	十六进制。浮点表示法

表 10.2 浮点类型的格式化选项

字符串的说明符

对于字符串类型，默认的格式说明符是 s，所以不必提供此说明符。对于字符串，可以指定一定的精度，这将解释为使用的最大字符数:

```
std::format("{}", "counter"); // yields "counter"
std::format("{:s}", "counter"); // yields "counter"
std::format("{:.5}", "counter"); // yields "count"
std::format("{:.5}", "hi"); // yields "hi"
```

只支持字符类型 char 和 wchar_t 的标准字符串类型，不支持类型为 u8string 和 char8_t、u16string 和 char16_t 或 u32string 和 char32_t 的字符串和序列。实际上，C++ 标准库为以下类型提供了格式化器:

- char* 和 const char*
- const char[n] (字符串字面值)
- std::string 和 std::basic_string<char, traits, allocator>
- std::string_view 和 std::basic_string_view<char, traits>
- wchar_t* 和 const wchar_t*
- const wchar_t[n] (宽字符串字面值)
- std::wstring 和 std::basic_string<wchar_t, traits, allocator>
- std::wstring_view 和 std::basic_string_view<wchar_t, traits>

其格式字符串和参数必须具有相同的字符类型:

```
auto ws1 = std::format("{}", L"K\u00F6ln"); // compile-time ERROR
std::wstring ws2 = std::format(L"{}", L"K\u00F6ln"); // OK
```

指针的说明符

对于指针类型，默认的格式说明符是 p，通常以十六进制表示法写地址，前缀为 0x。在没有 uintptr_t 类型的平台上，格式由实现定义:

```
void* ptr = ... ;
std::format("{}", ptr) // usually yields a value such as 0x7ff688ee64
std::format("{:p}", ptr) // usually yields a value such as 0x7ff688ee64
```

只支持以下指针类型:

- void* 和 const void*
- std::nullptr_t

可以传递 nullptr 或原始指针，所以将其强制转换为类型 (const) void*:

```
int i = 42;
std::format("{}", &i) // compile-time error
std::format("{}", static_cast<void*>(&i)) // OK (e.g., 0x7ff688ee64)
std::format("{:p}", static_cast<void*>(&i)) // OK (e.g., 0x7ff688ee64)
std::format("{}", static_cast<const void*>("hi")) // OK (e.g., 0x7ff688ee64)
std::format("{}", nullptr) // OK (usually 0x0)
std::format("{:p}", nullptr) // OK (usually 0x0)
```

10.4. 全局化

若为格式指定了 L，则使用特定于语言环境的表示法:

- 对于 bool，使用 std::numpunct::truename 和 std::numpunct::falsename 上的区域设置字符串。
- 对于整数值，使用依赖于语言环境的千位分隔符。
- 对于浮点值，使用依赖于语言环境的小数点和千位分隔字符。
- 对于 chrono 库中的几种类型表示法 (持续时间、时间点等)，使用特定于语言环境的格式。

要激活特定于语言环境的表示法，还必须向 std::format() 传递一个语言环境。

例如:

```
// initialize a locale for “German in Germany”:
#ifdef _MSC_VER
std::locale locG{"deu_deu.1252"};
#else
std::locale locG{"de_DE"};
#endif

// use it for formatting:
std::format(locG, "{0} {0:L}", 1000.7) // yields 1000.7 1.000,7
```

参见 format/formatgerman.cpp 获得完整的示例。

只有在使用语言环境说明符 L 时才会使用语言环境，否则将使用默认的语言环境“C”，并使用美式格式。

或者，可以设置全局语言环境，并使用 L 说明符:

```
std::locale::global(locG); // set German locale globally
std::format("{0} {0:L}", 1000.7) // yields 1000.7 1.000,7
```

可能需要创建自己的区域设置 (通常基于带有修改 facet 的现有区域设置)。例如:

format/formatbool.cpp

```cpp
#include <iostream>
#include <locale>
#include <format>

// define facet for German bool names:
class GermanBoolNames : public std::numpunct_byname<char> {
  public:
  GermanBoolNames (const std::string& name)
  : std::numpunct_byname<char>(name) {
  }
  protected:
  virtual std::string do_truename() const {
    return "wahr";
  }
  virtual std::string do_falsename() const {
    return "falsch";
  }
};

int main()
{
  // create locale with German bool names:
  std::locale locBool{std::cin.getloc(),
             new GermanBoolNames{""}};

  // use locale to print Boolean values:
  std::cout << std::format(locBool, "{0} {0:L}\n", false); // false falsch
}
```

该程序有以下输出:

```
false falsch
```

要用宽字符串打印值 (这是 Visual C++ 的一个问题)，格式字符串和参数都必须是宽字符串。例如:

```cpp
std::wstring city = L"K\u00F6ln"; // K¨oln
auto ws1 = std::format("{}", city); // compile-time ERROR
std::wstring ws2 = std::format(L"{}", city); // OK: ws2 is std::wstring
std::wcout << ws2 << '\n'; // OK
```

目前不支持 char8_t (UTF-8 字符)、char16_t 和 char32_t 类型的字符串。

10.5. 错误处理

理想情况下，C++ 编译器应该在编译时检测 bug，而非在运行时。因为字符串字面值在编译时已知，所以当字符串字面值被用作格式字符串时，C++ 可以检查格式违规，并且在 std::format() 中

这样做 [编译时检查格式字符串的要求是通过http://wg21.link/p2216r3提出的，并在 C++20 标准化后被接受为针对 C++20 的修复]:

```
std::format("{:d}", 42) // OK
std::format("{:s}", 42) // compile-time ERROR
```

若传递的格式字符串已经初始化或计算过，格式化库会按如下方式处理格式错误:

- std::format()，std::format_to() 和 format_to_n() 只接受编译时已知的格式字符串:
  - 字符串字面值
  - constexpr 字符指针
  - 可以转换为编译时字符串视图的编译时字符串
- 要使用在运行时计算的格式字符串，请使用
  - std::vformat()
  - std::vformat_to()
- 对于 std:: 格式化_size()，只能使用编译时已知的格式字符串。

例如:

```
const char* fmt1 = "{:d}"; // runtime format string
std::format(fmt1, 42); // compile-time ERROR
std::vformat(fmt1, std::make_format_args(42)); // OK

constexpr const char* fmt2 = "{:d}"; // compile-time format string
std::format(fmt2, 42); // OK
```

使用 fmt1 不会编译，因为传递的参数不是编译时字符串，并且使用 std::format()。然而，使用 fmt1 与 std::vformat() 工作良好 (必须转换所有参数与 std::make_format_args())。将 fmt2 传递给 std::format() 时，使用 fmt2 不会编译，因为其可初始化为编译时字符串。

若想在 std::vformat() 中使用多个参数，必须一次性将其传递给 std::make_format_args():

```
const char* fmt3 = "{} {}";
std::vformat(fmt3, std::make_format_args(x, y))
```

若在运行时检测到格式化失败，则抛出类型为 std::format_error 的异常。这个新标准异常类型派生自 std::runtime_error，并提供了标准异常的常用 API，可以使用调用 what() 获得的错误消息的字符串初始化异常。

例如:

```
try {
  const char* fmt4 = "{:s}";
  std::vformat(fmt4, std::make_format_args(42)) // throws std::format_error
}
catch (const std::format_error& e) {
  std::cerr << "FORMATTING EXCEPTION: " << e.what() << std::endl;
}
```

10.6. 自定义格式化输出

格式化库可以为用户定义的类型定义格式化。需要定义一个格式化器，也很容易实现。

10.6.1 基本格式器的 API

格式器是类型的类模板 std::formatter<> 的特化。在格式化程序内部，必须定义两个成员函数:

- parse() 实现如何解析类型的格式字符串说明符
- format() 为自定义类型的对象/值执行实际格式化

来看一下第一个最小的例子 (将逐步改进它)，指定了如何格式化具有固定值的对象/值。假设类型是这样定义的 (见 format/always40.hpp):

```
class Always40 {
  public:
  int getValue() const {
    return 40;
  }
};
```

对于这种类型，可以定义第一个格式化器 (应该改进) 如下:

format/formatalways40.hpp

```
#include "always40.hpp"
#include <format>
#include <iostream>

template<>
struct std::formatter<Always40>
{
  // parse the format string for this type:
  constexpr auto parse(std::format_parse_context& ctx) {
    return ctx.begin(); // return position of } (hopefully there)
  }

  // format by always writing its value:
  auto format(const Always40& obj, std::format_context& ctx) const {
    return std::format_to(ctx.out(), "{}", obj.getValue());
  }
};
```

这已经足够好，可以进行如下工作:

```
Always40 val;
std::cout << std::format("Value: {}\n", val);
std::cout << std::format("Twice: {0} {0}\n", val);
```

输出为:

```
Value: 40
Twice: 40 40
```

将格式化器定义为类型 std::formatter<> 的特化，用于类型 Always40:

```
template<>
struct std::formatter<Always40>
{
  ...
};
```

因为只有 public 成员，所以使用 struct，而非 class。

解析格式字符串

parse() 中，实现了解析格式字符串的函数:

```
// parse the format string for this type:
constexpr auto parse(std::format_parse_context& ctx) {
  return ctx.begin(); // return position of } (hopefully there)
}
```

该函数接受一个 std::format_parse_context，提供了一个 API 来遍历传递的格式字符串的剩余字符。Ctx.begin() 指向要解析的值的格式说明符的第一个字符，若没有说明符，则指向}:

- 若格式字符串为”Value: {:7.2f}” ctx.begin() 指向: ”7.2f}”
- 若格式字符串为”Twice: {0} {0}” ctx.begin() 指向: ”} {0}” 第一次调用时
- 若格式字符串为”{}\n” ctx.begin() 指向: ”}\n”

还有一个 ctx.end()，指向整个格式字符串的末尾。开始的 {已经解析过了，所以必须解析所有的字符，直到对应的结束的}。

对于格式字符串”Val: {1:_>20}cm \n”，ctx.begin() 是_的位置，ctx.end() 是\n 之后整个格式字符串的末尾。parse() 的任务是解析传递的参数的指定格式，所以必须解析字符_>20，然后返回格式说明符末尾的位置，即字符 0 后面的末尾}。

实现中，还不支持格式说明符，所以可以简单地返回得到的第一个字符的位置，只有当下一个字符确实是} 时才会起作用 (在结束} 之前处理字符是必须改进的第一件事)。使用指定的格式字符调用 std::format() 将不起作用:

```
Always40 val;
std::format("'{:7}", val) // ERROR
```

parse() 成员函数应该是 constexpr，以支持格式字符串的编译时计算，并且代码必须接受 constexpr 函数的所有限制 (C++20 放宽了这些限制)。

但可以看到这个 API 如何允许开发者解析我们为其类型指定的格式。例如，用于支持 chrono 库的格式化输出。当然，应该遵循标准说明符的约定，以避免混淆。

执行格式化

format() 中，实现了格式化传递值的函数:

```
// format by always writing its value:
auto format(const Always40& value, std::format_context& ctx) const {
  return std::format_to(ctx.out(), "{}", value.getValue());
}
```

该函数接受两个形参:

- 作为参数传递给 std::format()(或类似函数) 的值。
- 一个 std::format_context，提供 API 来编写格式化的结果字符 (根据已解析的格式)

format 上下文最重要的函数是 out()，会产生一个对象，可以将其传递给 std::format_to() 来编写格式化的实际字符。该函数必须返回新位置以获得进一步的输出，该输出由 std::format_to() 返回。

格式化器的 format() 成员函数应该是 const。根据最初的 C++20 标准，这并不需要 (参见http://wg21.link/lwg3636了解详细信息)。

10.6.2 改进解析

改进一下前面看到的例子。首先，应该确保解析器，以更好的方式处理格式说明符:

- 应该处理到结尾} 之前的所有字符。
- 当指定非法格式化参数时，应该抛出异常。
- 应该处理有效的格式化参数 (比如：指定的字段的宽度)。

来看看对前一个格式化器的改进版本 (这次处理的类型值总是为 41):

format/formatalways41.hpp

```
#include "always41.hpp"
#include <format>
template<>
class std::formatter<Always41>
{
  int width = 0; // specified width of the field
public:
  // parse the format string for this type:
  constexpr auto parse(std::format_parse_context& ctx) {
    auto pos = ctx.begin();
    while (pos != ctx.end() && *pos != '}') {
    if (*pos < '0' || *pos > '9') {
      throw std::format_error{std::format("invalid format '{}'", *pos)};
    }
```

```
    width = width * 10 + *pos - '0'; // new digit for the width
    ++pos;
    }
    return pos; // return position of }
  }

  // format by always writing its value:
  auto format(const Always41& obj, std::format_context& ctx) const {
    return std::format_to(ctx.out(), "{:{}}", obj.getValue(), width);
  }
};
```

格式化器现在有了一个成员来存储指定字段的宽度:

```
template<>
class std::formatter<Always41>
{
  int width = 0; // specified width of the field
  ...
};
```

字段的宽度初始化为 0，但可以通过格式字符串指定。

解析器现在有一个循环，处理所有字符，直到结尾的}:

```
constexpr auto parse(std::format_parse_context& ctx) {
  auto pos = ctx.begin();
  while (pos != ctx.end() && *pos != '}') {
    ...
    ++pos;
  }
  return pos; // return position of }
}
```

循环必须检查是否还有一个字符，以及它是否是末尾的}，因为调用 std::format() 的开发者可能忘记了末尾的}。

在循环中，将当前宽度乘以数字字符的整数值:

```
width = width * 10 + *pos - '0'; // new digit for the width
```

若字符不是数字，std::format() 初始化抛出 std::format 异常:

```
if (*pos < '0' || *pos > '9') {
  throw std::format_error{std::format("invalid format '{}'", *pos)};
}
```

这里不能使用 std::isdigit()，其不是一个可以在编译时调用的函数。

可以测试这样的格式化器:format/always.cpp

该程序有以下输出:

```
41
Value: 41
Twice: 41 41
With width: ’       41’
Format Error: invalid format ’ f’
```

该值是右对齐的，这是整数值的默认对齐方式。

10.6.3 自定义格式器使用标准格式化器

仍然可以改进上面实现的格式化器:

- 可以使用对齐说明符。
- 可以支持填充字符。

幸运的是，这里不必自己实现完整的解析，而可以使用标准格式化器，以便从已支持的格式说明符中获益。

实际上，有两种方法:

- 可以将工作委托给本地标准格式化器。
- 可以从标准格式化器继承。

将格式化委托给标准格式化器

要将格式化委托给标准格式化器，必须

- 声明一个本地标准格式化器
- 将 parse() 函数将工作委托给标准格式化器
- 将 format() 函数将工作委托给标准格式化器

一般来说，应该这样:

format/formatalways42ok.hpp

```
1  #include "always42.hpp"
2  #include <format>
3
4  // *** formatter for type Always42:
5  template<>
6  struct std::formatter<Always42>
7  {
8    // use a standard int formatter that does the work:
9    std::formatter<int> f;
10
```

```
  // delegate parsing to the standard formatter:
  constexpr auto parse(std::format_parse_context& ctx) {
    return f.parse(ctx);
  }

  // delegate formatting of the value to the standard formatter:
  auto format(const Always42& obj, std::format_context& ctx) const {
    return f.format(obj.getValue(), ctx);
  }
};
```

像往常一样，为 Always42 类型声明 std::formatter<> 的特化。但这一次，使用 int 类型的本地标准格式化程序来完成这项工作，将解析和格式化都委托给它。实际上，使用 getValue() 从类型中提取值，并使用标准 int 格式化器完成其余的格式化工作。

可以用下面的程序测试格式化器:

format/always42.cpp

```
#include "always42.hpp"
#include "formatalways42.hpp"
#include <iostream>

int main()
{
  try {
    Always42 val;
    std::cout << val.getValue() << '\n';
    std::cout << std::format("Value: {}\n", val);
    std::cout << std::format("Twice: {0} {0}\n", val);
    std::cout << std::format("With width: '{:7}'\n", val);
    std::cout << std::format("With all: '{:.^7}'\n", val);
  }
  catch (std::format_error& e) {
    std::cerr << "Format Error: " << e.what() << std::endl;
  }
}
```

该程序有以下输出:

```
42
Value: 42
Twice: 42 42
With width: '      42'
With all:   '..42...'
```

默认情况下，该值仍然是右对齐的，这是 int 的默认对齐方式。

实践中，可能需要对这段代码进行一些修改，稍后将详细讨论:

- 将 format() 声明为 const 可能无法编译，除非将格式化器声明为 mutable。
- 将 parse() 声明为 constexpr 可能无法编译。

继承标准格式器

可以从标准格式化器派生，以便格式化器成员及其 parse() 函数隐式可用:

format/formatalways42inherit.hpp

```
#include "always42.hpp"
#include <format>

// *** formatter for type Always42:
// - use standard int formatter
template<>
struct std::formatter<Always42> : std::formatter<int>
{
  auto format(const Always42& obj, std::format_context& ctx) {
    // delegate formatting of the value to the standard formatter:
    return std::formatter<int>::format(obj.getValue(), ctx);
  }
};
```

实践中，可能还需要对这段代码进行一些修改:

- 将 format() 声明为 const 可能无法编译。

实践中使用标准格式化器

实践中，C++20 的标准化存在一些问题，之后必须说明:

- 最初标准化的 C++20，并不要求格式化器的 format() 成员函数必须是 const(参见http://wg21.link/lwg3636了解详细信息)。为了支持不将 format() 声明为 const 成员函数的 C++ 标准库实现，必须将其声明为非 const 函数或将本地格式化器声明为 mutable。
- 现有的实现，可能还不支持 parse() 成员函数为 constexpr 的编译时解析，因为编译时解析是在 C++20 标准化之后添加的 (参见http://wg21.link/p2216r3)。这种情况下，不能将编译时解析委托给标准格式化器。

实践中，类型 Always42 的格式化程序必须如下所示:

format/formatalways42.hpp

```
#include "always42.hpp"
#include <format>

// *** formatter for type Always42:
template<>
struct std::formatter<Always42>
{
```

```
  // use a standard int formatter that does the work:
#if __cpp_lib_format < 202106
  mutable // in case the standard formatters have a non-const format()
#endif
  std::formatter<int> f;

  // delegate parsing to the standard int formatter:
#if __cpp_lib_format >= 202106
  constexpr // in case standard formatters don't support constexpr parse() yet
#endif
  auto parse(std::format_parse_context& ctx) {
    return f.parse(ctx);
  }

  // delegate formatting of the int value to the standard int formatter:
  auto format(const Always42& obj, std::format_context& ctx) const {
    return f.format(obj.getValue(), ctx);
  }
};
```

如您所见，

- 若采用了相应的修复，则仅使用 constexpr 声明 parse()
- 可以使用 mutable 声明本地格式化器，以便 const format() 成员函数可以调用非 const 的标准 format() 函数

对于两者，实现都使用了一个特性测试宏，该宏表示支持编译时解析 (期望采用也使标准格式化器的 format() 成员函数为 const)。

10.6.4 字符串使用标准格式化器

若要格式化更复杂的类型，一种常见的方法是创建一个字符串，然后使用字符串的标准格式化程序 (若只使用字符串字面值，则使用 std::string 或 std::string_view)。

例如，可以定义枚举类型和格式化器:

format/color.hpp

```
#include <format>
#include <string>

enum class Color { red, green, blue };

// *** formatter for enum type Color:
template<>
struct std::formatter<Color> : public std::formatter<std::string>
{
  auto format(Color c, format_context& ctx) const {
    // initialize a string for the value:
    std::string value;
```

```
    switch (c) {
      using enum Color;
      case red:
        value = "red";
        break;
      case green:
        value = "green";
        break;
      case blue:
        value = "blue";
        break;
      default:
        value = std::format("Color{}", static_cast<int>(c));
        break;
    }
    // and delegate the rest of formatting to the string formatter:
    return std::formatter<std::string>::format(value, ctx);
  }
};
```

通过从字符串格式化器继承格式化器，继承了 parse() 函数，所以支持字符串具有的所有格式说明符。format() 函数中，执行到字符串的映射，然后让标准格式化器完成其余的格式化工作。

可以这样使用格式化器:

format/color.cpp

```
#include "color.hpp"
#include <iostream>
#include <string>
#include <format>

int main()
{
  for (auto val : {Color::red, Color::green, Color::blue, Color{13}}) {
    // use user-provided formatter for enum Color:
    std::cout << std::format("Color {:_>8} has value {:02}\n",
                  val, static_cast<int>(val));
  }
}
```

该程序有以下输出:

```
Color _____red has value 00
Color ___green has value 01
Color ____blue has value 02
Color _Color13 has value 13
```

若不引入自定义的格式说明符，这种方法通常可以很好地工作。若只使用字符串字面值作为可能的值，甚至可以使用 std::string_view 的格式化器。

10.7. 附注

格式化输出最初是由 Victor Zverovich 和 Lee Howes 在http://wg21.link/p0645r0中提出。最终接受的提案是由 Victor Zverovich 在http://wg21.link/p0645r10中提出。

C++20 标准化之后，接受了几个针对 C++20 的修复。最重要的是要检查格式字符串，并且必须在编译时知道，正如 Victor Zverovich 在http://wg21.link/p2216r3中阐述的那样。其他的是http://wg21.link/p2372r3(修复 chrono 格式化程序的语言环境处理) 和http://wg21.link/p2418r2(格式参数成为通用/转发引用)。

第 11 章 <chrono> 的日期和时区

C++11 引入了对持续时间和时间点提供支持的 chrono 库，可以指定和处理不同单位的持续时间和不同时钟的时间点。但目前还不支持高级持续时间和时间点，如日期 (日、月和年)、工作日，也不支持处理不同的时区。

C++20 扩展了现有的 chrono 库，支持日期、时区和其他一些特性。本章将介绍这个扩展。

11.1. 举例概述

先来看一些例子。

11.1.1 每月 5 号安排一次会议

考虑这样一个程序，希望遍历一年中的所有月份，以便在每个月的 5 号安排一次会议:

lib/chrono1.cpp

```
#include <chrono>
#include <iostream>

int main()
{
  namespace chr = std::chrono; // shortcut for std::chrono
  using namespace std::literals; // for h, min, y suffixes

  // for each 5th of all months of 2021:
  chr::year_month_day first = 2021y / 1 / 5;
  for (auto d = first; d.year() == first.year(); d += chr::months{1}) {
    // print out the date:
    std::cout << d << ":\n";

    // init and print 18:30 UTC of those days:
    auto tp{chr::sys_days{d} + 18h + 30min};
    std::cout << std::format(" We meet on {:%A %D at %R}\n", tp);
  }
}
```

程序的输出如下所示:

```
2021-01-05:
 We meet on Tuesday 01/05/21 at 18:30
2021-02-05:
 We meet on Friday 02/05/21 at 18:30
2021-03-05:
 We meet on Friday 03/05/21 at 18:30
2021-04-05:
 We meet on Monday 04/05/21 at 18:30
```

```
2021-05-05:
 We meet on Wednesday 05/05/21 at 18:30
...
2021-11-05:
 We meet on Friday 11/05/21 at 18:30
2021-12-05:
 We meet on Sunday 12/05/21 at 18:30
```

让我们一步步地完成这个程序。

声明命名空间

从命名空间开始，并使用声明使 chrono 库更方便使用:

- 首先，引入 chr:: 作为 chrono 库的标准命名空间 std::chrono 的快捷方式:

```
namespace chr = std::chrono; // shortcut for std::chrono
```

为了使示例更具可读性，只使用 chr::，而非 std::chrono::。

- 然后，需要确保可以使用字面后缀，如 s、min、h 和 y (y 是 C++20 添加的新特性，其他由 C++14 添加)。

```
using namespace std::literals; // for min, h, y suffixes
```

可以直接使用 using 声明:

```
using namespace std::chrono; // skip chrono namespace qualification
```

不过，应该尽量避免使用这种 using 声明的作用域，以避免不必要的副作用。

日历类型 year_month_day

```
chr::year_month_day first = 2021y / 1 / 5;
```

数据流首先初始化开始日期，类型为 std::chrono::year_month_day:

year_month_day 是一个日历类型，为日期的所有三个字段提供属性，以便于处理特定日期的年、月和日。

因为要想遍历每个月的所有第五天，所以用 2021 年 1 月 5 日初始化对象，使用运算符/将年与月和日的值组合在一起:

- 首先，将年份创建为 std::chrono::year 类型的对象，使用新的标准字面符 **y**:

```
2021y
```

要使这个字面符可用，必须提供以下 using 声明之一:

```
using std::literals; // enable all standard literals
using std::chrono::literals; // enable all standard chrono literals
using namespace std::chrono; // enable all standard chrono literals
using namespace std; // enable all standard literals
```

若没有这些字面符，则可使用以下等价的方式:

```
std::chrono::year{2021}
```

- 使用操作符/将 std::chrono::year 与整型值组合，创建一个 std::chrono::year_month 类型的对象。第一个操作数是年，所以很明显第二个操作数必须是月，不能在这里指定日期。
- 再次使用操作符/将 std::chrono::year_month 对象与整数值组合，以创建 std::chrono::year_month_day。

日历类型的初始化是类型安全的，只需要指定第一个操作数的类型。

操作符已经产生了正确的类型，所以可以先用 auto 声明。若没有命名空间声明，则需要这样写:

```
auto first = std::chrono::year{2021} / 1 / 5;
```

有了时间字面值，就简单了:

```
auto first = 2021y/1/5;
```

初始化日历类型的其他方法

还有其他方法可以初始化日历类型，如 year_month_day:

```
auto d1 = std::chrono::years{2021}/1/5; // January 5, 2021
auto d2 = std::chrono:month{1}/5/2021; // January 5, 2021
auto d3 = std::chrono:day{5}/1/2021; // January 5, 2021
```

操作符/的第一个参数的类型指定了如何解释其他参数。

有了时间字面符，可以简单地写为:

```
using namespace std::literals;
auto d4 = 2021y/1/5; // January 5, 2021
auto d5 = 5/1/2021; // January 5, 2021
```

没有月的标准后缀，但有预定义的标准对象:

```
auto d6 = std::chrono::January / 5 / 2021; // January 5, 2021
```

有了相应的 using 声明，可以这样写:

```
using namespace std::chrono;
auto d6 = January/5/2021; // January 5, 2021
```

所有情况下，都会初始化 std::chrono::year_month_day 类型的对象。

新持续时间类型

例子中，使用一个循环来迭代一年中的所有月份:

```
for (auto d = first; d.year() == first.year(); d += chr::months{1}) {
  ...
}
```

std::chrono::months 是一个新的持续时间类型，表示一个月。可以将它用于所有日历类型来处理日期的特定月份，像在这里添加一个月那样:

```
std::chrono::year_month_day d = ... ;
d += chr::months{1}; // add one month
```

注意，当将其用于普通持续时间和时间点时，类型 months 表示一个月的平均持续时间，即 30.436875 天。所以对于普通的时间点，应该小心使用月和年。

所有 Chrono 类型的输出操作符

循环中，输出当前日期:

```
std::cout << d << '\n';
```

C++20 起，为几乎所有可能的 chrono 类型定义了一个输出操作符。

```
std::chrono::year_month_day d = ... ;
std::cout << "d: " << d << '\n';
```

这使得输出任何计时值变得很容易，但输出并不总是适合特定的需要。对于类型 year_month_day，输出格式是我们作为开发者所期望的类型:year-month-day。例如:

```
2021-01-05
```

其他默认输出格式通常使用斜杠作为分隔符，这里有详细说明。

对于用户定义的输出，chrono 库还支持用于格式化输出的新库。稍后会使用它来输出时间点 tp:

```
std::cout << std::format("We meet on {:%D at %R}\n", tp);
```

格式化说明符，%A 表示工作日，%D 表示日期，%R 表示时间 (小时和分钟)，与 C 函数 strftime() 和 POSIX date 命令的说明符对应，因此输出可能如下所示:

```
We meet on 10/05/21 at 18:30
```

还支持特定于语言环境的输出，稍后将详细描述 chrono 类型的格式化输出。

合并日期和时间

为了初始化 tp，将循环的天数与一天中的特定时间结合起来:

```
auto tp{sys_days{d} + 18h + 30min};
```

为了组合日期和时间，必须使用时间点和持续时间。std::chrono::year_month_day 这样的日历类型不是时间点，所以首先将 year_month_day 值转换为 time_point<> 对象:

```
std::chrono::year_month_day d = ... ;
std::chrono::sys_days{d} // convert to a time_point
```

类型 std::chrono::sys_days 是以天为粒度的系统时间点的新快捷方式，相当于:std::chrono::time_point<std::chrono::system_clock, std::chrono::days>。

通过添加一些持续时间 (18 小时和 30 分钟)，可以计算一个新值，该值 (chrono 库中通常如此) 具有一个粒度足以满足计算结果的类型。将天与小时和分钟结合起来，结果类型是一个以分钟为粒度的系统时间点，但不必知道类型，只要使用 auto 就好。

为了更明确地指定 tp 的类型，还可以这样声明:

- 作为系统时间点，但不指定其粒度:

```
chr::sys_time tp{chr::sys_days{d} + 18h + 30min};
```

通过类模板参数推导，可以推导出粒度的模板参数。

- 作为具有指定粒度的系统时间点:

```
chr::sys_time<chr::minutes> tp{chr::sys_days{d} + 18h + 30min};
```

这时，初始值不能比指定类型更细粒度，否则必须使用舍入函数。

- 作为系统时间点，方便以秒为粒度的类型:

```
chr::sys_seconds tp{chr::sys_days{d} + 18h + 30min};
```

初始值不能比指定类型更细粒度，否则必须使用舍入函数。

默认输出操作符根据上述指定的格式输出时间点。例如:

```
We meet on 10/05/21 at 18:30
```

注意，处理系统时间时，默认输出是 UTC。

更细粒度的时间点，还可以使用输出精确值所需的任何参数 (例如毫秒)。

11.1.2 把会议安排在每个月的最后一天

通过迭代每个月的最后几天来修改第一个示例程序，可以这样做:

lib/chrono2.cpp

```
#include <chrono>
#include <iostream>

int main()
{
  namespace chr = std::chrono; // shortcut for std::chrono
  using namespace std::literals; // for h, min, y suffixes

  // for each last of all months of 2021:
  auto first = 2021y / 1 / chr::last;
  for (auto d = first; d.year() == first.year(); d += chr::months{1}) {
    // print out the date:
    std::cout << d << ":\n";

    // init and print 18:30 UTC of those days:
    auto tp{chr::sys_days{d} + 18h + 30min};
    std::cout << std::format(" We meet on {:%A %D at %R}\n", tp);
  }
}
```

只修改了 first 的初始化。用 auto 类型声明，用 std::chrono::last 作为 day 进行初始化:

```
auto first = 2021y / 1 / chr::last;
```

类型 std::chrono::last 不仅表示一个月的最后一天，还表示 first 具有不同的类型:std::chrono::year_month_day_last。通过增加一个月，日期的日期得到了调整。实际上，该类型总是保存最后一天。当输出日期并指定相应的输出格式时，就会发生向数字日期的转换。

结果，输出为以下内容:

```
2021/Jan/last:
 We meet on Sunday 01/31/21 at 18:30
2021/Feb/last:
 We meet on Sunday 02/28/21 at 18:30
2021/Mar/last:
 We meet on Wednesday 03/31/21 at 18:30
2021/Apr/last:
 We meet on Friday 04/30/21 at 18:30
2021/May/last:
 We meet on Monday 05/31/21 at 18:30
...
2021/Nov/last:
 We meet on Tuesday 11/30/21 at 18:30
```

```
2021/Dec/last:
 We meet on Friday 12/31/21 at 18:30
```

year_month_day_last 的默认输出格式使用 last 和斜杠，而非减号作为分隔符 (只有 year_month_day 在其默认输出格式中使用连字符)。例如:

```
2021/Jan/last
```

可以将 first 声明为 std::chrono::year_month_day:

```
// for each last days of all months of 2021:
std::chrono::year_month_day first = 2021y / 1 / chr::last;
for (auto d = first; d.year() == first.year(); d += chr::months{1}) {
  // print out the date:
  std::cout << d << ":\n";

  // init and print 18:30 UTC of those days:
  auto tp{chr::sys_days{d} + 18h + 30min};
  std::cout << std::format(" We meet on {:%D at %R}\n", tp);
}
```

会有以下输出:

```
2021-01-31:
 We meet on Sunday 01/31/21 at 18:30
2021-02-31 is not a valid date:
 We meet on Wednesday 03/03/21 at 18:30
2021-03-31:
 We meet on Wednesday 03/31/21 at 18:30
2021-04-31 is not a valid date:
 We meet on Saturday 05/01/21 at 18:30
2021-05-31:
 We meet on Monday 05/31/21 at 18:30
...
```

first 类型存储的是日期的数值，初始化为 1 月的最后一天，所以可以迭代每个月的第 31 天。若这样的日期不存在，则默认输出格式打印为无效日期，而 std::format() 甚至执行错误的计算。

处理这种情况的一种方法是检查日期是否有效，再执行要做的事情。例如:

lib/chrono3.cpp

```
#include <chrono>
#include <iostream>

int main()
{
  namespace chr = std::chrono; // shortcut for std::chrono
  using namespace std::literals; // for h, min, y suffixes
```

```
8
9     // for each last of all months of 2021:
10    chr::year_month_day first = 2021y / 1 / 31;
11    for (auto d = first; d.year() == first.year(); d += chr::months{1}) {
12      // print out the date:
13      if (d.ok()) {
14        std::cout << d << ":\n";
15      }
16      else {
17        // for months not having the 31st use the 1st of the next month:
18        auto d1 = d.year() / d.month() / 1 + chr::months{1};
19        std::cout << d << ":\n";
20      }
21
22      // init and print 18:30 UTC of those days:
23      auto tp{chr::sys_days{d} + 18h + 30min};
24      std::cout << std::format(" We meet on {:%A %D at %R}\n", tp);
25    }
26  }
```

通过使用成员函数 ok()，将无效日期调整为下个月的第一天。可以得到以下输出:

```
2021-01-31:
 We meet on Sunday 01/31/21 at 18:30
2021-02-31 is not a valid date:
 We meet on Wednesday 03/03/21 at 18:30
2021-03-31:
 We meet on Wednesday 03/31/21 at 18:30
2021-04-31 is not a valid date:
 We meet on Saturday 05/01/21 at 18:30
2021-05-31:
 We meet on Monday 05/31/21 at 18:30
...
2021-11-31 is not a valid date:
 We meet on Wednesday 12/01/21 at 18:30
2021-12-31:
 We meet on Friday 12/31/21 at 18:30
```

11.1.3 每一个第一个星期一安排会议

类似地，可以在每个月的第一个星期一安排一次会议:

lib/chrono4.cpp

```
1  #include <chrono>
2  #include <iostream>
3
4  int main()
5  {
```

```
  namespace chr = std::chrono; // shortcut for std::chrono
  using namespace std::literals; // for min, h, y suffixes

  // for each first Monday of all months of 2021:
  auto first = 2021y / 1 / chr::Monday[1];
  for (auto d = first; d.year() == first.year(); d += chr::months{1}) {
    // print out the date:
    std::cout << d << '\n';

    // init and print 18:30 UTC of those days:
    auto tp{chr::sys_days{d} + 18h + 30min};
    std::cout << std::format(" We meet on {:%A %D at %R}\n", tp);
  }
}
```

first 有一个特殊类型 std::chrono::year_month_weekday，表示一年中一个月中的工作日:
默认输出格式表示使用特定格式，但格式化的输出工作良好

```
2021/Jan/Mon[1]
 We meet on Monday 01/04/21 at 18:30
2021/Feb/Mon[1]
 We meet on Monday 02/01/21 at 18:30
2021/Mar/Mon[1]
 We meet on Monday 03/01/21 at 18:30
2021/Apr/Mon[1]
 We meet on Monday 04/05/21 at 18:30
2021/May/Mon[1]
 We meet on Monday 05/03/21 at 18:30
...
2021/Nov/Mon[1]
 We meet on Monday 11/01/21 at 18:30
2021/Dec/Mon[1]
 We meet on Monday 12/06/21 at 18:30
```

工作日的日历类型

这一次，首先初始化为一个日历类型 std::chrono::year_month_weekday 的对象，并将其初始化为 2021 年 1 月的第一个星期一:

```
auto first = 2021y / 1 / chr::Monday[1];
```

使用操作符/来组合不同的日期字段，但这一次，工作日的类型就要发挥作用了:

- 首先，调用 2021y / 1，将 std::chrono::year 与整型值结合，创建 std::chrono::year_month。
- 然后，调用 std::chrono::Monday(std::chrono::weekday 类型的标准对象) 的 [] 操作符创建一个 std::chrono::weekday_indexed 类型的对象，表示第 n 个工作日。

- 操作符/用于将 std::chrono::year_month 与 std::chrono::weekday_indexed 类型的对象组合在一起，创建一个 std::chrono::year_month_weekday 对象。

因此，完整的声明如下所示:

```
std::chrono::year_month_weekday first = 2021y / 1 / std::chrono::Monday[1];
```

year_month_weekday 的默认输出格式使用斜杠，而不是减号作为分隔符 (只有 year_month_day 在其默认输出格式中使用连字符)。例如:

```
2021/Jan/Mon[1]
```

11.1.4 不同的时区

让我们修改第一个示例程序，以使时区发挥作用。实际上，我们希望程序迭代一年中每个月的所有第一个星期一，并在不同的时区安排会议。

为此需要做以下修改:

- 迭代当前年份的所有月份
- 会议安排在当地时间 18:30
- 使用其他时区输出会议时间

可以这样编写代码:

lib/chronotz.cpp

```
#include <chrono>
#include <iostream>

int main()
{
  namespace chr = std::chrono; // shortcut for std::chrono
  using namespace std::literals; // for min, h, y suffixes

  try {
    // initialize today as current local date:
    auto localNow = chr::current_zone()->to_local(chr::system_clock::now());
    chr::year_month_day today{chr::floor<chr::days>(localNow)};
    std::cout << "today: " << today << '\n';
    // for each first Monday of all months of the current year:
    auto first = today.year() / 1 / chr::Monday[1];
    for (auto d = first; d.year() == first.year(); d += chr::months{1}) {
      // print out the date:
      std::cout << d << '\n';

      // init and print 18:30 local time of those days:
      auto tp{chr::local_days{d} + 18h + 30min}; // no timezone
      std::cout << " tp: " << tp << '\n';

```

```
      // apply this local time to the current timezone:
      chr::zoned_time timeLocal{chr::current_zone(), tp}; // local time
      std::cout << " local: " << timeLocal << '\n';

      // print out date with other timezones:
      chr::zoned_time timeUkraine{"Europe/Kiev", timeLocal}; // Ukraine time
      chr::zoned_time timeUSWest{"America/Los_Angeles", timeLocal}; // Pacific time
      std::cout << " Ukraine: " << timeUkraine << '\n';
      std::cout << " Pacific: " << timeUSWest << '\n';
    }
  }
  catch (const std::exception& e) {
    std::cerr << "EXCEPTION: " << e.what() << '\n';
  }
}
```

该计划将于 2021 年在欧洲启动，结果如下:

```
today: 2021-03-29
2021/Jan/Mon[1]
    tp:      2021-01-04 18:30:00
    local:   2021-01-04 18:30:00 CET
    Ukraine: 2021-01-04 19:30:00 EET
    Pacific: 2021-01-04 09:30:00 PST
2021/Feb/Mon[1]
    tp:      2021-02-01 18:30:00
    local:   2021-02-01 18:30:00 CET
    Ukraine: 2021-02-01 19:30:00 EET
    Pacific: 2021-02-01 09:30:00 PST
2021/Mar/Mon[1]
    tp:      2021-03-01 18:30:00
    local:   2021-03-01 18:30:00 CET
    Ukraine: 2021-03-01 19:30:00 EET
    Pacific: 2021-03-01 09:30:00 PST
2021/Apr/Mon[1]
    tp:      2021-04-05 18:30:00
    local:   2021-04-05 18:30:00 CEST
    Ukraine: 2021-04-05 19:30:00 EEST
    Pacific: 2021-04-05 09:30:00 PDT
2021/May/Mon[1]
    tp:      2021-05-03 18:30:00
    local:   2021-05-03 18:30:00 CEST
    Ukraine: 2021-05-03 19:30:00 EEST
    Pacific: 2021-05-03 09:30:00 PDT
...
2021/Oct/Mon[1]
    tp:      2021-10-04 18:30:00
    local:   2021-10-04 18:30:00 CEST
    Ukraine: 2021-10-04 19:30:00 EEST
```

```
    Pacific: 2021-10-04 09:30:00 PDT
2021/Nov/Mon[1]
    tp:      2021-11-01 18:30:00
    local:   2021-11-01 18:30:00 CET
    Ukraine: 2021-11-01 19:30:00 EET
    Pacific: 2021-11-01 10:30:00 PDT
2021/Dec/Mon[1]
    tp:      2021-12-06 18:30:00
    local:   2021-12-06 18:30:00 CET
    Ukraine: 2021-12-06 19:30:00 EET
    Pacific: 2021-12-06 09:30:00 PST
```

看看 10 月和 11 月的输出: 在洛杉矶，会议现在安排在不同的时间，尽管使用了相同的时区 PDT。这是因为会议时间的来源 (中欧) 从夏季改为冬季/标准时间。

再一次，让我们一步步地了解一下这个程序的改进。

应对今天

第一个新语句是将 today 初始化为 year_month_day 类型的对象:

```
auto localNow = chr::current_zone()->to_local(chr::system_clock::now());
chr::year_month_day today = chr::floor<chr::days>(localNow);
```

从 C++11 开始就提供了对 std::chrono::system_clock::now() 的支持，会产生一个具有系统时钟粒度的 std::chrono::time_point<>。这个系统时钟使用 UTC(从 C++20 开始，system_clock 保证使用 Unix 时间，基于 UTC)。所以首先必须将当前的 UTC 时间和日期调整为当前/本地时区的时间和日期，current_zone()->to_local() 就是这样做的。否则，本地日期可能与 UTC 日期不匹配 (因为已经过了午夜，而 UTC 还没有过，或者相反)。

直接使用 localNow 初始化 year_month_day 值不会编译，因为这会“窄化”值 (丢失值的小时、分钟、秒和次秒部分)。通过使用 floor() 这样的函数 (C++17 起可用)，可以根据所要求的粒度将值向下舍入。

若需要根据 UTC 的当前日期，以下内容就足够了:

```
chr::year_month_day today = chr::floor<chr::days>(chr::system_clock::now());
```

本地日期及时间

同样，将迭代的天数与一天中的特定时间结合起来。但这一次，首先会将每一天转换为类型 std::chrono::local_days:

```
auto tp{chr::local_days{d} + 18h + 30min};
```

std::chrono::local_days 是 time_point<local_t, days> 的快捷方式。这里使用了伪时钟 std::chrono::local_t，所以有一个本地时间点，这个时间点还没有关联时区 (甚至还没有 UTC)。

下一个语句将本地时间点与当前时区结合起来，这样就得到了一个特定于时区的时间点，类型为 std::chrono::zoned_time<>:

```
chr::zoned_time timeLocal{chr::current_zone(), tp}; // local time
```

时间点已经与系统时钟相关联，则已经将时间与 UTC 相关联。将这样的时间点与不同的时区组合将时间转换为特定的时区。

默认输出操作符演示了时间点和分区时间之间的差异:

```
auto tpLocal{chr::local_days{d} + 18h + 30min}; // local timepoint
std::cout << "timepoint: " << tpLocal << '\n';

chr::zoned_time timeLocal{chr::current_zone(), tpLocal}; // apply to local timezone
std::cout << "zonedtime: " << timeLocal << '\n';
```

这段代码可能的输出为:

```
timepoint: 2021-01-04 18:30:00
zonedtime: 2021-01-04 18:30:00 CET
```

输出的时间点没有时区，而经过时区处理的时间有时区。但两个输出输出的时间相同，因为我们将本地时间点使用的是本地时区。

事实上，时间点和分区时间的区别如下:

- 一个时间点可能与一个已定义的纪元相关联，可以定义一个独特的时间点，但它也可能与一个未定义或伪 epoch 相关联，直到将其与时区结合起来，其含义才清楚。
- 分区时间总是与时区相关联，以便 epoch 最终具有已定义的含义。epoch 总是代表一个唯一的时间点。

使用 std::chrono::sys_days，而非 std::chrono::local_days 时会发生什么:

```
auto tpSys{chr::sys_days{d} + 18h + 30min}; // system timepoint
std::cout << "timepoint: " << tpSys << '\n';

chr::zoned_time timeSys{chr::current_zone(), tpSys}; // convert to local timezone
std::cout << "zonedtime: " << timeSys << '\n';
```

这里使用一个系统时间点，有一个相关联的时区 UTC。当本地时间点应用于时区时，将系统时间点转换为时区。所以当在一个时差一小时的时区运行程序时，可得到以下输出:

```
timepoint: 2021-01-04 18:30:00
zonedtime: 2021-01-04 19:30:00 CET
```

使用其他时区

最后，使用不同的时区打印时间点，一个用于乌克兰经过基辅，另一个用于北美太平洋时区经过洛杉矶:

```
chr::zoned_time timeUkraine{"Europe/Kiev", timeLocal}; // Ukraine time
chr::zoned_time timeUSWest{"America/Los_Angeles", timeLocal}; // Pacific time
std::cout << " Ukraine: " << timeUkraine << '\n';
std::cout << " Pacific: " << timeUSWest << '\n';
```

要指定时区，必须使用 IANA 时区数据库的官方时区名称，该名称通常基于代表时区的城市。时区缩写 (如 PST) 可能会在一年中发生变化，或者适用于不同的时区。

使用这些对象的默认输出操作符添加相应的时区缩写，无论它在“冬令时”:

```
local:   2021-01-04 18:30:00 CET
Ukraine: 2021-01-04 19:30:00 EET
Pacific: 2021-01-04 09:30:00 PST
```

或者在“夏令时”:

```
local:   2021-07-05 18:30:00 CEST
Ukraine: 2021-07-05 19:30:00 EEST
Pacific: 2021-07-05 09:30:00 PDT
```

有时候，有些时区是夏令时，而有些则不是。例如，在 11 月初，美国有夏令时，但在乌克兰没有。

```
local:   2021-11-01 18:30:00 CET
Ukraine: 2021-11-01 19:30:00 EET
Pacific: 2021-11-01 10:30:00 PDT
```

不支持相应时区时

C++ 可以在小型系统上存在并发挥作用，甚至可以在烤面包机上，IANA 时区数据库的可用性将耗费太多的资源，所以不需要存在时区数据库中。

若不支持时区数据库，则所有特定于时区的调用都会抛出异常，所以在可移植的 C++ 程序中使用时区时，应该可以捕捉到异常:

```
try {
  // initialize today as current local date:
  auto localNow = chr::current_zone()->to_local(chr::system_clock::now());
  ...
}
```

```
catch (const std::exception& e) {
  std::cerr << "EXCEPTION: " << e.what() << '\n'; // IANA timezone DB missing
}
```

注意，对于 Visual C++，这也适用于较旧的 Windows 系统，所需的操作系统支持不能低于 Windows 10。此外，平台可以在不使用所有 IANA 时区数据库的情况下，支持时区 API。

例如，在德语版本的 Windows 11 安装中，我得到的输出是 GMT-8 和 GMT-7，而不是 PST 和 PDT。

11.2. 基本计时概念和术语

chrono 库的设计是为了能够处理计时器和时钟在不同系统上可能不同的事实，并随着时间的推移提高精度。为了避免每 10 年左右引入一个新的时间类型，C++11 建立的目标是通过分离时间段和时间点 (“时间点”) 来提供一个中立精度的概念。C++20 扩展了这些基本概念，支持日期和时区，以及其他的一些扩展。

因此，chrono 库的核心由以下类型或概念组成:

- 时间段定义为一个时间单位上的特定时标的数量。一个例子是时间段，如“3 分钟”(“分钟”的 3 个刻度)。其他例子是“42 毫秒”或“86400 秒”，表示 1 天的时间段。这种方法还允许指定类似于“1.5 乘以三分之一秒”的内容，其中 1.5 是时标的数量，“三分之一秒”是所用的时间单位。
- 时间点定义为时间段和时间开始的组合 (所谓的 epoch)。
  一个典型的例子是表示 2000 年 12 月 31 日午夜的系统时间点。因为系统 epoch 指定为 Unix/POSIX 的诞生时间，所以时间点定义为“自 1970 年 1 月 1 日以来的 946,684,800 秒”(或者在考虑闰秒时为 946,684,822 秒，有些时间点会考虑闰秒)。
  注意，epoch 可能是未指定的或伪 epoch。例如，本地时间点与本地时间相关联，仍然需要一个特定的 epoch 值，但代表的确切时间点不清楚，直到将这个时间点应用到特定的时区。12 月 31 日的午夜就是一个很好的例子: 庆祝活动和烟花在世界上不同的时间开始，这取决于所在的时区。
- 时钟是定义时间点的日历对象，所以不同的时钟有不同的 epoch。
  每个时间点由时钟参数化。C++11 引入了两个基本时钟 (一个用于处理系统时间的系统时钟，和一个用于测量和计时器的稳定时钟，不应受系统时钟变化的影响)。C++20 增加了新的时钟来处理 UTC 时间点 (支持闰秒)、GPS 时间点、TAI(国际原子时间) 时间点和文件系统时间点。
  为了处理本地时间点，C++20 还添加了一个伪时钟 local_t，不绑定到特定的 epoch/origin。
  处理多个时间点的操作,处理两个时间点之间的时间段/差值,通常需要使用相同的 epoch/clock，但不同时钟之间可以转换。
  时钟 (若不是本地的) 提供了一个函数 now() 来生成当前时间点。
- 日历类型 (C++20 中引入) 允许使用日、月和年等常用术语来处理日历的属性。这些类型可用于表示日期的单个属性 (日、月、年和工作日)，以及其组合 (完整日期的 year_month 或 year_month_day)。

不同的符号 (例如 Wednesday、November 和 last) 允许我们使用通用术语表示部分日期和完整日期，例如“十一月的最后一个星期三”。

可以使用系统时钟或本地时间的伪时钟，将完全指定的日历日期 (具有年、月、日或月中的特定工作日) 转换为时间点或从伪时间点转换为时间点。

- 时区 (C++20 中引入) 可以处理这样情况，当不同的时区起作用时，同时发生的事件会导致不同的时间。若在世界标准时间 18:30 在线会面，会议的当地时间在亚洲将明显晚于美国，而在美国则明显早于亚洲。

  因此，时区通过将时间点应用或转换为不同的本地时间来赋予时间点 (不同的) 含义。
- 时区时间 (C++20 引入) 是时间点和时区的组合，可用于将本地时间点应用于特定的时区 (可能是“当前”时区) 或将时间点转换为不同的时区。

  时区时间可以看作是一个日期和时间对象，是 epoch(时间点的 origin)、duration(到 origin 的距离) 和 timezone(用于调整结果时间) 的组合。

对于所有这些概念和类型，C++20 增加了对输出 (甚至格式化) 和解析的支持，就可以决定是否以如下格式输出日期/时间值:

```
Nov/24/2011
24.11.2011
2011-11-24 16:30:00 UTC
Thursday, November 11, 2011
```

11.3. 基于 C++20 的 Chrono 扩展

详细讨论如何使用 chrono 库之前，先了解一下所有的基本类型和符号。

11.3.1 时间段类型

C++20 引入了更多的时间段类型，包括天、周、月和年。表“C++20 的标准时间段类型”列出了 C++ 现在提供的所有时间段类型，以及可用于创建值的标准字面符后缀和输出值时使用的默认输出后缀。

使用 std::chrono::months 和 std::chrono::years 时要谨慎。月和年表示一个月或一年的平均时间段，这是一个小数天。平均年的时间段是通过考虑闰年来计算的:

- 每四年多一天 (366 天，而非 365 天)
- 每隔 100 年，就会多一天
- 每隔 400 年，又会多一天

因此，其值为 400 * 365 + 100−4 + 1 除以 400，平均一个月的时间是它的十二分之一。

类型	定义为	字面	输出后缀
nanoseconds	1/1,000,000,000 seconds	**ns**	**ns**
microseconds	1,000 nanoseconds	**us**	**μs or us**
milliseconds	1,000 microseconds	**ms**	**ms**

seconds	1,000 milliseconds	**s**	**s**
minutes	60 seconds	**min**	**min**
hours	60 minutes	**h**	**h**
days	24 hours	**d**	**d**
weeks	7 days		**[604800]s**
months	30.436875 days		**[2629746]s**
years	365.2425 days	**y**	**[31556952]s**

表 11.1 C++20 的标准时间段类型

11.3.2 时钟

C++20 引入了两个新的时钟 (时钟定义了时间点的 origin/epoch)。

表“C++20 以后的标准时钟类型”，描述了 C++ 标准库现在提供的所有时钟的名称和含义。

类型	含义	Epoch
system_clock	与系统时钟相关联 (C++11 起)	UTC
utc_clock	用于 UTC 时间的时钟	UTC
gps_clock	GPS 的时钟	GPS
tai_clock	用于国际原子时值的时钟	TAI
file_clock	文件系统库时间点的时钟	由实现指定
local_t	本地时间点的伪时钟	无
steady_clock	用于测量的时钟 (C++11 起)	由实现指定
high_resolution_clock	(见文本内容)	

表 11.2 C++20 后的标准时钟类型

Epoch 列指定时钟是否指定唯一的时间点，以便始终定义特定的 UTC 时间，这需要一个稳定的特定 epoch。对于 file_clock, epoch 是特定于系统的，但在程序的多次运行中是稳定的。steady_clock 的 epoch 可能会随着应用程序的每次运行而改变 (例如，当系统重新启动时)。对于伪时钟 local_t,，epoch 解释为“本地时间”，所以必须将其与时区结合起来，才能知道其代表的是哪个时间点。

C++ 标准也从 C++11 开始提供了一个高分辨率的时钟，但在将代码从一个平台移植到另一个平台时，使用它会带来一些微妙的问题。实践中，high_resolution_clock 可能是 system_clock 或 steady_clock 的别名，所以时钟有时稳定，有时不稳定，并且这个时钟可能支持或不支持转换到其他时钟或 time_t。

在任何平台上，steady_clock 要比 high_resolution_clock 好，所以应该使用 steady_clock。

稍后将讨论时钟的以下细节:

- 时钟之间的差异
- 时钟之间的转换
- 如何处理闰秒

11.3.3 时间点类型

C++20 引入了两个新的时间点类型，基于 C++11 以来的通用定义:

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

新的类型提供了一种更方便的方式来使用不同时钟的时间点。

表“C++20 后的标准时间点类型”描述了时间点类型的名称和含义。

类型	含义	定义为
local_time<Dur>	本地时间点	time_point<LocalTime, Dur>
local_seconds	本地时间点，单位为秒	time_point<LocalTime, seconds>
local_days	本地时间点，单位为天	time_point<LocalTime, days>
sys_time<Dur>	系统时间点	time_point<systime_clock, Dur>
sys_seconds	系统时间点，单位为秒	time_point<systime_clock, seconds>
sys_days	系统时间点，单位为天	time_point<systime_clodk, days>
utc_time<Dur>	UTC 时间点	time_point<utc_clock, Dur>
utc_seconds	UTC 时间点，单位为秒	time_point<utc_clock, seconds>
tai_time<Dur>	TAI 时间点	time_point<tai_clock, Dur>
tai_seconds	TAI 时间点，单位为秒	time_point<tai_clock, seconds>
gps_time<Dur>	GPS 时间点	time_point<gps_clock, Dur>
gps_seconds	GPS 时间点，单位为秒	time_point<gps_clock, seconds>
file_time<Dur>	文件系统时间点	time_point<file_clock, Dur>

表 11.3 C++20 后的标准时间点类型

对于每个标准时钟 (稳定时钟除外)，都有一个相应的_time<> 类型，允许声明表示时钟日期或时间点的对象，所以…_seconds 类型允许以秒为粒度定义相应类型的日期/时间对象。对于系统时间和当地时间，…_days 类型允许以天为粒度定义相应类型的日期/时间对象。例如:

```
std::chrono::sys_days x; // time_point<system_clock, days>
std::chrono::local_seconds y; // time_point<local_t, seconds>
std::chrono::file_time<std::chrono::seconds> z; // time_point<file_clock, seconds>
```

此类型的对象仍然表示一个时间点，尽管不幸的是该类型有一个复数名称。例如，sys_days 表示定义为系统时间点的一天 (名称来自“粒度天的系统时间点”)。

11.3.4 日历类型

作为时间点类型的扩展，C++20 在 chrono 库中引入了民用 (公历) 日历类型。

虽然时间点指定为从 epoch 开始的时间段，但日历类型具有不同的组合类型和值，适用于年、月、工作日和一个月中的天数。这两种用法都很有用:

- 时间点类型很好，只要我们只计算秒、小时和天 (比如在一年中的每一天做一些事情)。
- 日历类型对于日期计算非常有用，因为要考虑到月和年有不同的天数。此外，还可以处理诸如“本月的第三个星期一”或“本月的最后一天”之类的事情。

表“标准日历类型”列出了新的日历类型及其默认输出格式。

类型	含义	输出格式
day	日	05
month	月	Feb
year	年	1999
weekday	工作日	Mon
weekday_indexed	第 n 个工作日	Mon[2]
weekday_last	最后一个工作日	Mon[last]
month_day	一个月中的一天	Feb/05
month_day_last	一个月的最后一天	Feb/last
month_weekday	一个月的第 n 个工作日	Feb/Mon[2]
month_weekday_last	一个月最后一个工作日	Feb/Mon[last]
year_month	一年中的月份	1999/Feb
year_month_day	一个完整的日期 (一年中的一个月的一天)	1999-02-05
year_month_day_last	一年中一个月的最后一天	1999/Feb/last
year_month_weekday	一年中一个月的第 n 个工作日	1999/Feb/Mon[2]
year_month_weekday_last	一年中一个月的最后一个工作日	1999/Feb/Mon[last]

表 11.4 标准日历类型

这些类型名称并不传递日期元素进行初始化或写入格式化输出的顺序。例如，类型 std::chrono::year_month_day 可以这样使用:

```
using namespace std::chrono;

year_month_day d = January/31/2021; // January 31, 2021
std::cout << std::format("{:%D}", d); // writes 01/31/21
```

像 year_month_day 这样的日历类型可精确地计算下个月同一天的日期:

```
std::chrono::year_month_day start = ... ; // 2021/2/5
auto end = start + std::chrono::months{1}; // 2021/3/5
```

使用像 sys_days 这样的时间点，相应的代码将无法工作，因为它使用了一个月的平均时间段:

```
std::chrono::sys_days start = ... ; // 2021/2/5
auto end = start + std::chrono::months{1}; // 2021/3/7 10:29:06
```

在本例中，end 具有不同的类型，因为对于月份，小数天数也会起作用。

当添加 4 周或 28 天时，时间点类型更好，这是一个简单的算术运算，不必考虑月或年的不同长度:

```
std::chrono::sys_days start = ... ; // 2021/1/5
auto end = start + std::chrono::weeks{4}; // 2021/2/2
```

后面将讨论使用月份和年份的细节。

有特定的类型用于处理工作日以及一个月内的第 n 个和最后一个工作日，可以迭代所有第二个星期一或跳转到下个月的最后一天:

```
std::chrono::year_month_day_last start = ... ; // 2021/2/28
auto end = start + std::chrono::months{1}; // 2021/3/31
```

每种类型都有一个默认的输出格式，是一个固定的英文字符序列。对于其他格式，请使用格式化的 chrono 输出。在默认输出格式中，只有 year_month_day 使用减号作为分隔符。默认情况下，所有其他类型的都用斜杠分隔。

为了处理日历类型的值，定义了两个日历常量:

- std::chrono::last 指定一个月的最后一天/最后一个工作日，该常量的类型为 std::chrono::last_spec。
- std::chrono::Sunday，std::chrono::Monday，…，std::chrono::Saturday 指定工作日 (Sunday 的值为 0，Saturday 的值为 6)。
  这些常量的类型为 std::chrono::weekday。
- std::chrono::January、std::chrono::February、…、std::chrono::December 指定月份 (January 的值为 1，December 的值为 12)。
  这些常量的类型为 std::chrono::month。

日历类型的限制操作

日历类型的设计目的是在编译时检测操作是否有用或性能不佳，所以一些“显而易见”的操作不会按照表中列出的日历类型的标准操作进行编译:

- 日和月的计算取决于年份。不能添加天数/月份，也不能计算没有年份的所有月份类型的差值。
- 完全指定日期的日运算需要一些时间来处理不同长度的月份和闰年，可以添加/减去天数或计算这些类型之间的差异。
- chrono 不对一周的第一天做任何假设，所以无法在工作日之间获得顺序。当与工作日比较类型时，只支持 == 和!=。

类型	++/–	增/减	-(差.)	==	</<=>
day	yes	days	yes	yes	yes
month	yes	months,years	yes	yes	yes

year	yes	years	yes	yes	yes
weekday	yes	days	yes	yes	-
weekday_indexed	-	-	-	yes	-
weekday_last	-	-	-	yes	-
month_day	-	-	-	yes	yes
month_day_last	-	-	-	yes	yes
month_weekday	-	-	-	yes	-
month_weekday_last	-	-	-	yes	-
year_month	-	months,years	yes	yes	yes
year_month_day	-	months,years	-	yes	yes
year_month_day_last	-	months,years	-	yes	yes
year_month_weekday	-	months,years	-	yes	-
year_month_weekday_last	-	months,years	-	yes	-

表 11.5 日历类型的标准操作

工作日的运算是对 7 取模，所以哪一天是一周的第一天并不重要。可以计算任意两个工作日之间的差，结果总是介于 0 和 6 之间的值。将差额加到第一个工作日总是第二个工作日。例如:

```
std::cout << chr::Friday - chr::Tuesday << '\n'; // 3d (Tuesday thru Friday)
std::cout << chr::Tuesday - chr::Friday << '\n'; // 4d (Friday thru Tuesday)

auto d1 = chr::February / 25 / 2021;
auto d2 = chr::March / 3 / 2021;
std::cout << chr::sys_days{d1} - chr::sys_days{d2} << '\n'; // -6d (date diff)
std::cout << chr::weekday(d1) - chr::weekday(d2) << '\n'; // 3d (weekday diff)
```

这样，就可以很容易地计算出“下周一”与“周一减去当前工作日”的差值:

```
d1 = chr::sys_days{d1} + (chr::Monday - chr::weekday(d1)); // set d1 to next Monday
```

若 d1 是日历日期类型，首先必须将其转换为类型 std::chrono::sys_days，以便支持日期算术 (使用该类型声明 d1 可能会更好)。

同样，月的运算以 12 为模，则 12 月之后的下一个月是 1 月。若年份是类型的一部分，则相应地进行调整:

```
auto m = chr::December;
std::cout << m + chr::months{10} << '\n'; // Oct
std::cout << 2021y/m + chr::months{10} << '\n'; // 2022/Oct
```

请注意，接受整数值的时间日历类型的构造函数是显式的，所以使用整数值的显式初始化失败:

```
std::chrono::day d1{3}; // OK
std::chrono::day d2 = 3; // ERROR

d1 = 3; // ERROR
d1 = std::chrono::day{3}; // OK

passDay(3); // ERROR
passDay(std::chrono::day{3}); // OK
```

11.3.5 时间类型 hh_mm_ss

与日历类型一致，C++20 引入了一个新的时间类型 std::chrono::hh_mm_ss，将时间段转换为具有相应时间字段的数据结构。表“std::chrono::hh_mm_ss members”描述了 hh_mm_ss 成员的名称和含义。

成员函数	含义
hours()	小时数
minutes()	分钟数
seconds()	秒数
subseconds()	具有适当粒度的部分秒数
is_negative()	若值为负值，则为 true
to_duration()	转换 (返回) 到时间段
precision()	亚秒的时间段类型
operator precision()	转换为具有相应精度的值
fractional_width	亚秒精度 (静态成员)

表 11.6 std::chrono::hh_mm_ss members

这种类型对于处理时间段和时间点的不同属性非常有用，允许将时间段拆分为其属性，并作为格式化辅助。例如，可以检查特定的小时或将小时和分钟作为整数值传递给另一个函数:

```
auto dur = measure(); // process and yield some duration
std::chrono::hh_mm_ss hms{dur}; // convert to data structure for attributes
process(hms.hours(), hms.minutes()); // pass hours and minutes
```

若有一个时间点，必须首先将其转换为持续时间，通常只需计算时间点和当天午夜之间的差值 (通过将其舍入到天数粒度来计算)。例如:

```
auto tp = getStartTime(); // process and yield some timepoint
// convert time to data structure for attributes:
std::chrono::hh_mm_ss hms{tp - std::chrono::floor<std::chrono:days>(tp)};
process(hms.hours(), hms.minutes()); // pass hours and minutes
```

另一个例子，可以使用 hh_mm_ss 以不同的形式打印时间段的属性:

```
auto t0 = std::chrono::system_clock::now();
...
auto t1 = std::chrono::system_clock::now();
std::chrono::hh_mm_ss hms{t1 - t0};
std::cout << "minutes: " << hms.hours() + hms.minutes() << '\n';
std::cout << "seconds: " << hms.seconds() << '\n';
std::cout << "subsecs: " << hms.subseconds() << '\n';
```

可能会输出:

```
minutes: 63min
seconds: 19s
subsecs: 502998000ns
```

没有办法用特定的值直接初始化 hh_mm_ss 对象的不同属性。通常，应该使用 duration 类型来处理时间:

```
using namespace std::literals;
...
auto t1 = 18h + 30min; // 18 hours and 30 minutes
```

hh_mm_ss 最强大的功能是，其接受任何精度的时间段，并将其转换为通常的属性和持续时间类型小时、分钟和秒。此外，subseconds() 将以适当的持续时间类型 (如毫秒或纳秒) 生成值的其余部分。即使单位不是 10 的幂 (例如，三分之一秒)，hh_mm_ss 将其转换为最多 18 位的 10 次方次秒。若不能表示准确的值，则使用六位数的精度，可以使用标准输出操作符将结果作为一个整体输出。例如:

```
std::chrono::duration<int, std::ratio<1,3>> third{1};
auto manysecs = 10000s;
auto dblsecs = 10000.0s;

std::cout << "third: " << third << '\n';
std::cout << " " << std::chrono::hh_mm_ss{third} << '\n';
std::cout << "manysecs: " << manysecs << '\n';
std::cout << " " << std::chrono::hh_mm_ss{manysecs} << '\n';
std::cout << "dblsecs: " << dblsecs << '\n';
std::cout << " " << std::chrono::hh_mm_ss{dblsecs} << '\n';
```

这段代码有以下输出:

```
third:    1[1/3]s
          00:00:00.333333
manysecs: 10000s
          02:46:40
dblsecs:  10000.000000s
          02:46:40.000000
```

要输出特定属性，还可以使用带有特定转换说明符的格式化输出:

```
auto manysecs = 10000s;
std::cout << "manysecs: " << std::format("{:%T}", manysecs) << '\n';
```

输出为:

```
manysecs: 02:46:40
```

11.3.6 时间工具

chrono 库现在还提供了一些辅助函数来处理 12 小时和 24 小时格式。表“时间工具”列出了这些函数。

std::chrono::is_am(h)	h 是否为 0 到 11 之间的小时值
std::chrono::is_pm(h)	h 是否为 12 到 23 之间的小时值
std::chrono::make12(h)	生成小时值 h 的 12 小时等价值
std::chrono::make24(h, toPM)	生成小时值 h 的 24 小时等价值

表 11.7 时间工具

std::chrono::make12() 和 std::chrono::make24() 都要求小时值在 0 到 23 之间。std::chrono::make24() 的第二个参数指定是否将传递的小时值解释为 PM 值。若为 true，则传递的值在 0 到 11 之间，则该函数将 12 小时添加到该值中。

例如:

```
for (int hourValue : {9, 17}) {
  std::chrono::hours h{hourValue};
  if (std::chrono::is_am(h)) {
    h = std::chrono::make24(h, true); // assume a PM hour is meant
  }
  std::cout << "Tea at " << std::chrono::make12(h).count() << "pm" << '\n';
}
```

代码有以下输出:

```
Tea at 9pm
Tea at 5pm
```

11.4. I/O 与 Chrono 类型

C++20 提供了对所有计时类型的直接输出和解析的新支持。

11.4.1 默认输出格式

对于所有的 chrono 类型，标准输出操作符从 C++20 开始定义。不仅输出值，而且使用适当的格式和单位，依赖于地区的格式化也是可能的。

所有日历类型都会输出出带有日历类型输出格式的值。

所有时间段类型都以单位类型输出值，如表中时间段的输出单位所示，这适用于任何字面符操作符。

单位	输出后缀
atto	as
femto	fs
pico	ps
nano	ns
micro	μs 或 us
milli	ms
centi	cs
deci	ds
ratio<1>	s
deca	das
hecto	hs
kilo	ks
mega	Ms
giga	Gs
tera	Ts
peta	Ps
exa	Es
ratio<60>	min
ratio<3600>	h
ratio<86400>	d
ratio<num, 1>	[num]s
ratio<num, den>	[num/den]s

表 11.8 时间段的输出单位

对于所有标准时间点类型，输出操作符以以下格式输出日期和时间 (可选):

- 用于可隐式转换为天数的整数粒度单位的“年-月-日”
- 年-月-日 时: 分: 秒，表示小于等于天的整数粒度单位

若时间点值 (成员 rep) 的类型为浮点类型，则不定义输出操作符。[这个漏洞有望在 C++23 中修复。]

对于时间部分，使用类型为 hh_mm_ss 的输出运算符，这对应于格式化输出的%F %T 转换说明符。

例如:

```
auto tpSys = std::chrono::system_clock::now();
std::cout << tpSys << '\n'; // 2021-04-25 13:37:02.936314000

auto tpL = chr::zoned_time{chr::current_zone(), tpSys}.get_local_time();
std::cout << tpL; // 2021-04-25 15:37:02.936314000
std::cout << chr::floor<chr::milliseconds>(tpL); // 2021-04-25 15:37:02.936
std::cout << chr::floor<chr::seconds>(tpL); // 2021-04-25 15:37:02
std::cout << chr::floor<chr::minutes>(tpL); // 2021-04-25 15:37:00
std::cout << chr::floor<chr::days>(tpL); // 2021-04-25
std::cout << chr::floor<chr::weeks>(tpL); // 2021-04-22

auto tp3 = std::chrono::floor<chr::duration<long long, std::ratio<1, 3>>>(tpSys);
std::cout << tp3 << '\n'; // 2021-04-25 13:37:02.666666

chr::sys_time<chr::duration<double, std::milli>> tpD{tpSys};
std::cout << tpD << '\n'; // ERROR: no output operator defined

std::chrono::gps_seconds tpGPS;
std::cout << tpGPS << '\n'; // 1980-01-06 00:00:00

auto tpStd = std::chrono::steady_clock::now();
std::cout << "tpStd: " << tpStd; // ERROR: no output operator defined
```

zoned_time<> 类型输出类似于 timepoint 类型，扩展为缩写的时区名称。以下时区缩写用于标准时钟:

- UTC 有 sys_clock, utc_clock 和 file_clock
- TAI 有 tai_clock
- GPS 有 gps_clock

sys_info 和 local_info 具有未定义格式的输出操作符，只用于调试。

11.4.2 格式化输出

Chrono 支持格式化输出的新库，所以可以对 std::format() 和 std::format_to() 的参数使用日期/时间类型。

例如:

```
auto t0 = std::chrono::system_clock::now();
...
auto t1 = std::chrono::system_clock::now();

std::cout << std::format("From {} to {}\nit took {}\n", t0, t1, t1-t0);
```

使用日期/时间类型的默认输出格式。例如，可能有以下输出:

```
From 2021-04-03 15:21:33.197859000 to 2021-04-03 15:21:34.686544000
it took 1488685000ns
```

从 20121-04-03 15:21:33.197859000 到 20121-04-03 15:21:34.686544000，耗时 1488685000ns

```
std::cout << std::format("From {:%T} to {:%T} it took {:%S}s\n", t0, t1, t1-t0);
```

这可能会有以下输出:

```
From 15:21:34.686544000 to 15:21:34.686544000 it took 01.488685000s
```

chrono 类型的格式说明符是标准格式说明符语法的一部分 (每个说明符都可选):

```
fill align width .prec L spec
```

- fill, align, with 和 prec 表示与标准格式说明符相同。
- spec 指定格式化的通用符号，以% 开头。
- L 与往常一样，还为支持它的说明符打开语言环境相关的格式。

表“计时类型的转换”说明符列出了日期/时间类型格式化输出的所有转换说明符，示例基于 Sunday, June 9, 2019 17:33:16 和 850 毫秒。

若不使用特定的转换说明符，则使用默认输出操作符，该操作符标记无效日期。当使用特定的转换说明符时，情况并非如此:

```
std::chrono::year_month_day ymd{2021y/2/31}; // February 31, 2021
std::cout << std::format("{}", ymd); // 2021-02-31 is not a valid ...
std::cout << std::format("{:%F}", ymd); // 2021-02-31
std::cout << std::format("{:%Y-%m-%d}", ymd); // 2021-02-31
```

若日期/时间值类型没有为转换说明符提供必要的信息，则抛出 std::format_error 异常:

- 年份指定符用于 month_day
- 工作日指定符用于指定时间段
- 输出月份或工作日名称，并且该值无效
- 时区说明符用于本地时间点

此外，请注意以下关于转换说明符的事项:

- 负值时间段或 hh_mm_ss 值在整个值前面打印负号。例如:

```
std::cout << std::format("{:%H:%M:%S}", -10000s); // outputs: -02:46:40
```

Spec.	例子	含义
%c	Sun Jun 9 17:33:16 2019	表示标准或地区的日期和时间

日期：		
%x	06/09/19	表示标准或地区的日期
%F	2019-06-09	年-月-日，四位和两位数字
%D	06/09/19	两位数的月/日/年
%e	9	个位数时，以空格开头的日期
%d	09	两位数的日期
%b	Jun	标准或地区的缩写月名
%h	Jun	同上
%B	June	标准或地区的月份名称
%m	06	两位数的月份
%Y	2019	四位数的年份
%y	19	两位数的年份，不显示世纪
%G	2019	ISO 以周为基础的年份为四位数字 (按%V 表示周)
%g	19	iso 以周为基础的年份为两位数字 (按%V 表示周)
%C	20	两位数的世纪
工作日和周：		
%a	Sun	标准或地区的工作日名称缩写
%A	Sunday	标准或地区的工作日名称
%w	0	工作日为十进制数 (星期日为 0，星期六为 6)
%u	7	工作日为十进制数 (星期一为 1，星期天为 7)
%W	22	一年中的第几周 (00…(第 01 周从第一个星期一开始)
%U	23	一年中的第几周 (00…(第 01 周从第一个星期日开始)
%V	23	一年中的 ISO 星期 (01…53、第一周为 1 月 4 日)
时间：		
%X	17:33:16	表示标准或地区的时间
%r	05:33:16 PM	标准或当地的 12 小时时钟时间
%T	17:33:16.850	时: 分: 秒 (根据需要依赖于区域设置的亚秒)
%R	17:33	时: 分，两位数
%H	17	24 小时时钟，时间为两位数
%I	05	12 小时时钟，时间为两位数
%p	PM	按照 12 小时制是 AM 还是 PM
%M	33	分钟，两位数表示
%S	16.850	秒为十进制数 (根据需要特定于本地的亚秒)
其他：		
%Z	CEST	时区缩写 (也可以是 UTC、TAI 或 GPS)
%z	+0200	与 UTC(+02:00 时%Ez 或%Oz) 的偏移 (时和分))
%j	160	三位数年份中的第几天 (1 月 1 日是 001)
%q	ms	时间段的后缀单位

%Q	6196850	时间段的具体值
%n	\n	换行符
%t	\t	制表符
%%	%	% 字符

表 11.9 计时类型的转换说明符

- 不同的周数和年份格式可能导致不同的输出值。
  例如，2023 年 1 月 1 日，星期天，的结果如下:
  - 第 00 周，%W (第一个星期一之前的第一周)
  - 第 01 周，%U(第一个星期一的周)
  - 第 52 周，%V(ISO 周: 第 01 周周一之前的一周，即 1 月 4 日)

  ISO 周可能是上一年的最后一周，所以 ISO 年就是那一周所在的年份，可能会少一个:
  - Year 2023 ，%Y
  - Year 23 ，%y
  - Year 2022 ，%G (ISO 年的 ISO 周%V，即前一个月的最后一周)
  - Year 22 ，%g (ISO 年的 ISO 周%V，即前一个月的最后一周)
- 以下时区缩写用于标准时钟:
  - UTC 有 sys_clock, utc_clock 和 file_clock
  - TAI 有 tai_clock
  - GPS 有 gps_clock

  除了%q 和%Q 之外，的所有转换说明符也可用于格式化解析。

11.4.3 依赖于本地环境的输出

若输出流是由具有自己格式的区域设置注入的，则各种类型的默认输出操作符使用与区域设置相关的格式。例如:

```
using namespace std::literals;
auto dur = 42.2ms;

std::cout << dur << '\n'; // 42.2ms

#ifdef _MSC_VER
  std::locale locG("deu_deu.1252");
#else
  std::locale locG("de_DE");
#endif
std::cout.imbue(locG); // switch to German locale
std::cout << dur << '\n'; // 42,2ms
```

std::format() 格式化的输出像往常一样处理:[此行为指定为 C++20 的一个错误修复，使用http://wg21.link/p2372，从而 C++20 的原始措辞没有指定此行为。]

- 默认情况下，格式化输出使用独立于语言环境的“C”语言环境。
- 通过指定 L，可以切换到通过 locale 参数，或作为全局 locale 指定的依赖于 locale 的输出

则要使用依赖于语言环境的表示法，必须使用 L 说明符，并将语言环境作为第一个参数传递给 std::format()，或者在调用之前设置全局语言环境。例如:

```
using namespace std::literals;
auto dur = 42.2ms; // duration to print

#ifdef _MSC_VER
  std::locale locG("deu_deu.1252");
#else
  std::locale locG("de_DE");
#endif

std::string s1 = std::format("{:%S}", dur); // "00.042s" (not localized)
std::string s3 = std::format(locG, "{:%S}", dur); // "00.042s" (not localized)
std::string s2 = std::format(locG, "{:L%S}", dur); // "00,042s" (localized)

std::locale::global(locG); // set German locale globally
std::string s4 = std::format("{:L%S}", dur); // "00,042s" (localized)
```

某些情况下，甚至可以根据 strftime() 和 ISO 8601:2004 使用另一种语言环境的表示，可以在转换说明符前面使用前导 O 或 E 来指定:

- E 可以用作区域设置在 c，C，x，X，y，Y 和 z 前面的替代表示
- O 可以用作区域设置前面的替代数字符号 d, e, H, I, m, M, S, u, U V, w, W, y 和 z

11.4.4 格式化输入

chrono 库还支持格式化输入。这里有两个选择:

- 特定日期/时间类型提供了一个独立的函数 std::chrono::from_stream()，用于根据传入的格式字符串读取特定的值。
- 操纵符 std::chrono::parse() 允们使用 from_stream() 作为带有 >> 输入操作符解析的一部分。

使用 from_stream()

下面的代码演示了，如何使用 from_stream() 解析完整的时间点:

```
std::chrono::sys_seconds tp;
std::istringstream sstrm{"2021-2-28 17:30:00"};

std::chrono::from_stream(sstrm, "%F %T", tp);
if (sstrm) {
  std::cout << "tp: " << tp << '\n';
}
```

```
else {
  std::cerr << "reading into tp failed\n";
}
```

代码会有以下输出:

```
tp: 2021-02-28 17:30:00
```

另一个例子，可以从指定完整月份名称和年份的字符序列中解析 year_month，以及其他内容:

```
std::chrono::year_month m;
std::istringstream sstrm{"Monday, April 5, 2021"};
std::chrono::from_stream(sstrm, "%A, %B %d, %Y", m);
if (sstrm) {
  std::cout << "month: " << m << '\n'; // prints: month: 2021/Apr
}
```

格式字符串接受除%q 和%Q 以外的格式化输出的所有转换说明符，从而提高了灵活性。例如:

- %d 表示一个或两个字符来指定日期，使用%4d，可以指定最多解析四个字符。
- %n 只表示一个空白字符。
- %t 表示零或一个空白字符。
- 像空格这样的空白字符，可以表示任意数量的空白 (包括零空白)。

from_stream() 可用于下列类型:

- duration<> 类型
- sys_time<>, utc_time<>, gps_time<>, tai_time<>, local_time<> 或 file_time<> 类型
- day, month 或 year 类型
- year_month, month_day, 或 year_month_day 类型
- weekday 类型

格式必须是 const char* 类型的 C 字符串，必须匹配输入流中的字符和要解析的值。若出现以下情况，解析失败:

- 输入的字符序列与所需的格式不匹配
- 该格式没有为值提供足够的信息
- 解析的日期无效

这些情况下，设置了流的 failbit，可以通过调用 fail() 或使用流作为布尔值进行测试。

解析日期/时间的通用函数

实际上，日期和时间很少硬编码。但在测试代码时，通常需要一种简单的方法来指定日期/时间值。

下面是我用来测试本书示例的一个小辅助函数:

lib/chronoparse.hpp

```cpp
#include <chrono>
#include <string>
#include <sstream>
#include <cassert>

// parse year-month-day with optional hour:minute and optional :sec
// - returns a time_point<> of the passed clock (default: system_clock)
// in seconds
template<typename Clock = std::chrono::system_clock>
auto parseDateTime(const std::string& s)
{
  // return value:
  std::chrono::time_point<Clock, std::chrono::seconds> tp;

  // string stream to read from:
  std::istringstream sstrm{s}; // no string_view support

  auto posColon = s.find(":");
  if (posColon != std::string::npos) {
    if (posColon != s.rfind(":")) {
      // multiple colons:
      std::chrono::from_stream(sstrm, "%F %T", tp);
    }
    else {
      // one colon:
      std::chrono::from_stream(sstrm, "%F %R", tp);
    }
  }
  else {
    // no colon:
    std::chrono::from_stream(sstrm, "%F", tp);
  }

  // handle invalid formats:
  assert((!sstrm.fail()));

  return tp;
}
```

可以这样使用 parseDateTime():

lib/chronoparse.cpp

```cpp
#include "chronoparse.hpp"
#include <iostream>

int main()
{
  auto tp1 = parseDateTime("2021-1-1");
```

```
  std::cout << std::format("{:%F %T %Z}\n", tp1);

  auto tp2 = parseDateTime<std::chrono::local_t>("2021-1-1");
  std::cout << std::format("{:%F %T}\n", tp2);

  auto tp3 = parseDateTime<std::chrono::utc_clock>("2015-6-30 23:59:60");
  std::cout << std::format("{:%F %T %Z}\n", tp3);

  auto tp4 = parseDateTime<std::chrono::gps_clock>("2021-1-1 18:30");
  std::cout << std::format("{:%F %T %Z}\n", tp4);
}
```

程序有以下输出:

```
2021-01-01 00:00:00 UTC
2021-01-01 00:00:00
2015-06-30 23:59:60 UTC
2021-01-01 18:30:00 GPS
```

对于本地时间点，不能使用%Z 来输出其时区 (这样做会引发异常)。

使用 parse() 操纵符

等价 from_stream() 的方式

```
std::chrono::from_stream(sstrm, "%F %T", tp);
```

也可以这样:

```
sstrm >> std::chrono::parse("%F %T", tp);
```

最初的 C++20 标准没有直接作为字符串字面值传递格式，所以必须使用

```
sstrm >> std::chrono::parse(std::string{"%F %T"}, tp);
```

然而，这应该通过http://wg21.link/lwg3554来解决。

std::chrono::parse() 是一个 I/O 流操纵符，允许在从输入流读取的一条语句中解析多个值。使用移动语义，可以传递一个临时输入流。例如:

```
chr::sys_days tp;
chr::hours h;
chr::minutes m;
// parse date into tp, hour into h and minute into m:
std::istringstream{"12/24/21 18:00"} >> chr::parse("%D", tp)
                    >> chr::parse(" %H", h)
                    >> chr::parse(":%M", m);
std::cout << tp << " at " << h << ' ' << m << '\n';
```

这段代码输出为:

```
2021-12-24 at 18h 0min
```

可以显式地将字符串字面量“%D”、“%H”和“:%M”转换为字符串。

解析时区

解析时区有点棘手，因为时区缩写不是唯一的: 为了完成功能，from_stream() 具有以下格式:

```
istream from_stream(istream, format, value)
istream from_stream(istream, format, value, abbrevPtr)
istream from_stream(istream, format, value, abbrevPtr, offsetPtr)
```

可以选择传递 std::string 的地址，将解析后的时区缩写存储到字符串中，也可以传递 std::chrono::minutes 对象的地址以将解析后的时区偏移存储到该字符串中。在这两种情况下都可以传递 nullptr。

然而，仍然要谨慎:

- 以下代码可以工作:

```
chr::sys_seconds tp;
std::istringstream sstrm{"2021-4-13 12:00 UTC"};
chr::from_stream(sstrm, "%F %R %Z", tp);
std::cout << std::format("{:%F %R %Z}\n", tp); // 2021-04-13 12:00 UTC
```

然而，工作原因只是因为系统时间点使用的是 UTC。

- 因为忽略了时区，下面的命令不起作用:

```
chr::sys_seconds tp;
std::istringstream sstrm{"2021-4-13 12:00 MST"};
chr::from_stream(sstrm, "%F %R %Z", tp);
std::cout << std::format("{:%F %R %Z}", tp); // 2021-04-13 12:00 UTC
```

%Z 用于解析 MST，但没有参数来存储该值。

- 下面的方法似乎行得通:

```
chr::sys_seconds tp;
std::string tzAbbrev;
std::istringstream sstrm{"2021-4-13 12:00 MST"};
chr::from_stream(sstrm, "%F %R %Z", tp, &tzAbbrev);
std::cout << tp << '\n'; // 2021-04-13 12:00
std::cout << tzAbbrev << '\n'; // MST
```

但若计算时区时间，会看到正在将 UTC 时间转换为不同的时区:

```
chr::zoned_time zt{tzAbbrev, tp}; // OK: MST exists
std::cout << zt << '\n'; // 2021-04-13 05:00:00 MST
```

- 下面的方法似乎确实有效:

```
chr::local_seconds tp; // local time
std::string tzAbbrev;
std::istringstream sstrm{"2021-4-13 12:00 MST"};
chr::from_stream(sstrm, "%F %R %Z", tp, &tzAbbrev);
std::cout << tp << '\n'; // 2021-04-13 12:00
std::cout << tzAbbrev << '\n'; // MST
chr::zoned_time zt{tzAbbrev, tp}; // OK: MST exists
std::cout << zt << '\n'; // 2021-04-13 12:00:00 MST
```

  但 MST 是时区数据库中为数不多的不推荐使用的缩写之一，将此代码与 CEST 或 CST 一起使用时，会在初始化 zoned_time 时抛出异常。

- 要么只使用 tzAbbrev，而非 zoned_time 和%Z:

```
chr::local_seconds tp; // local time
std::string tzAbbrev;
std::istringstream sstrm{"2021-4-13 12:00 CST"};
chr::from_stream(sstrm, "%F %R %Z", tp, &tzAbbrev);
std::cout << std::format("{:%F %R} {}", tp, tzAbbrev); // 2021-04-13 12:00 CST
```

  或必须处理将时区缩写映射到时区的代码。

注意，%Z 不能解析伪时区 GPS 和 TAI。

11.5. 实践中使用 Chrono 扩展

已经了解了 chrono 库的新特性和新类型，本节将讨论如何在实践中使用它们。

更多的例子请参见http://github.com/HowardHinnant/date/wiki/Examples-and-Recipes。

11.5.1 无效日期

日历类型的值可能无效。这有两种情况:

- 使用无效值进行初始化。例如:

```
std::chrono::day d{0}; // invalid day
std::chrono::year_month ym{2021y/13}; // invalid year_month
std::chrono::year_month_day ymd{2021y/2/31}; // invalid year_month_day
```

- 通过导致无效日期的计算。例如:

```
auto ymd1 = std::chrono::year{2021}/1/31; // January 31, 2021
ymd1 += std::chrono::months{1}; // February 31, 2021 (invalid)

auto ymd0 = std::chrono::year{2020}/2/29; // February 29, 2020
ymd1 += std::chrono::years{1}; // February 29, 2021 (invalid)
```

表“标准日期属性的有效值”列出了日期的内部类型和不同属性的可能值。

属性	内部类型	有效值
Day	unsigned char	1 到 31
Month	unsigned char	1 到 12
Year	short	-32767 到 32767
Weekday	unsigned char	0(周日) 到 6(周六) 和 7 (还是周日)，转换为 0
Weekday index	unsigned char	1 到 5

表 11.10 标准日期属性的有效值

若单个组件有效，则所有组合类型都有效 (例如，若月份和工作日都有效，则有效的 month_weekday 有效)，需要进行以下检查:

- 必须存在类型为 year_month_day 的完整日期，这里考虑了闰年。例如:

```
2020y/2/29; // valid (there is a February 29 in 2020)
2021y/2/29; // invalid (there is no February 29 in 2021)
```

- month_day 只有当该日在当月有效时才有效。2 月 29 是有效的，30 是无效的。
  例如:

```
February/29; // valid (a February can have 29 days)
February/30; // invalid (no February can have 30 days)
```

- year_month_weekday 仅当 weekday 索引能够在一年中的指定月份中存在时才有效。
  例如:

```
2020y/1/Thursday[5]; // valid (there is a fifth Thursday in January 2020)
2020y/1/Sunday[5]; // invalid (there is no fifth Sunday in January 2020)
```

每个日历类型都提供一个成员函数 ok() 来检查值是否有效，默认输出操作符表示无效日期。处理无效日期的方式取决于编程逻辑。对于一个月中创建天数过大的典型场景，有以下选项:

- 四舍五入到每月的最后一天:

```
auto ymd = std::chrono::year{2021}/1/31;
ymd += std::chrono::months{1};
if (!ymd.ok()) {
  ymd = ymd.year()/ymd.month()/std::chrono::last; // February 28, 2021
}
```

注意，右边的表达式创建了 year_month_last，然后将其转换为 year_month_day 类型。

- 四舍五入到下个月的第一天:

```
auto ymd = std::chrono::year{2021}/1/31;
ymd += std::chrono::months{1};
if (!ymd.ok()) {
  ymd = ymd.year()/ymd.month()/1 + std::chrono::months{1}; // March 1, 2021
}
```

不要只在月份上加 1，因为对于 12 月，则创建了一个无效的月份。

- 根据所有超过的天数进行四舍五入:

```
auto ymd = std::chrono::year{2021}/1/31;
ymd += std::chrono::months{1}; // March 3, 2021
if (!ymd.ok()) {
  ymd = std::chrono::sys_days(ymd);
}
```

这使用了 year_month_day 转换的特殊功能，其中所有溢出的天数都被逻辑地添加到下一个月，但只适用于几天 (不能以这种方式添加 1000 天)。

若日期不是一个有效值，默认输出格式将用“不是一个有效类型”发出信号。例如:

```
std::chrono::day d{0}; // invalid day
std::chrono::year_month_day ymd{2021y/2/31}; // invalid year_month_day
std::cout << "day: " << d << '\n';
std::cout << "ymd: " << ymd << '\n';
```

这段代码将输出:[注意，C++20 标准中的规范在这里有点不一致，目前正在修复。例如，它声明对于无效的 year_month_days“不是一个有效日期”是输出]。

```
day: 00 is not a valid day
ymd: 2021-02-31 is not a valid year_month_day
```

当使用格式化输出的默认格式 (仅为 {}) 时，也会发生同样的情况。通过使用特定的转换说明符，可以禁用“not a valid”输出 (在银行或季度处理软件中，有时甚至使用 6 月 31 日之类的日期):

```
std::chrono::year_month_day ymd{2021y/2/31};
std::cout << ymd << '\n'; // “2021-02-31 is not a valid year_month_day”
std::cout << std::format("{:%F}\n", ymd); // “2021-02-31”
std::cout << std::format("{:%Y-%m-%d}\n", ymd); // “2021-02-31”
```

11.5.2 处理月份和年份

因为 std::chrono::months 和 std::chrono::years 不是天数的整数倍，所以在使用时必须非常小心。

- 对于仅具有年份值的标准类型 (month, year, year_month, year_month_day, year_month_weekday_last，…)，向日期添加特定的月份或年份数时可以正常工作。
- 对于整个日期只有一个值的标准时间点类型 (time_point、sys_time、sys_seconds 和 sys_days)，将相应的平均分数周期添加到日期中，这可能不会产生期望的日期。

例如，看看在 2020 年 12 月 31 日之前增加四个月或四年的不同影响:

- 当处理 year_month_day 时:

```
chr::year_month_day ymd0 = chr::year{2020}/12/31;
auto ymd1 = ymd0 + chr::months{4}; // OOPS: April 31, 2021
```

```
auto ymd2 = ymd0 + chr::years{4}; // OK: December 31, 2024

std::cout << "ymd: " << ymd0 << '\n'; // 2020-12-31
std::cout << " +4months: " << ymd1 << '\n'; // 2021-04-31 is not a valid ...
std::cout << " +4years: " << ymd2 << '\n'; // 2024-12-31
```

- 当处理 year_month_day_last 时:

```
chr::year_month_day_last yml0 = chr::year{2020}/12/chr::last;
auto yml1 = yml0 + chr::months{4}; // OK: last day of April 2021
auto yml2 = yml0 + chr::years{4}; // OK: last day of Dec. 2024

std::cout << "yml: " << yml0 << '\n'; // 2020/Dec/last
std::cout << " +4months: " << yml1 << '\n'; // 2021/Apr/last
std::cout << " as date: "
      << chr::sys_days{yml1} << '\n'; // 2021-04-30
std::cout << " +4years: " << yml2 << '\n'; // 2024/Dec/last
```

- 当处理 sys_days 时:

```
chr::sys_days day0 = chr::year{2020}/12/31;
auto day1 = day0 + chr::months{4}; // OOPS: May 1, 2021 17:56:24
auto day2 = day0 + chr::years{4}; // OOPS: Dec. 30, 2024 23:16:48

std::cout << "day: " << day0 << '\n'; // 2020-12-31
std::cout << " with time: "
      << chr::sys_seconds{day0} << '\n'; // 2020-12-31 00:00:00
std::cout << " +4months: " << day1 << '\n'; // 2021-05-01 17:56:24
std::cout << " +4years: " << day2 << '\n'; // 2024-12-30 23:16:48
```

支持此特性的唯一目的是，允许对诸如物理或生物过程之类的事物进行建模，这些过程不关心人类日历 (天气、妊娠期等) 的复杂性，并且不应将此特性用于其他目的。

注意，输出值和默认输出格式都是不同的。这是一个明显的迹象，表明使用了不同的类型:

- 当向日历类型添加月份或年份时，这些类型处理正确的逻辑日期 (同一天或下个月或年份的最后一天)。注意，这可能会导致无效日期，例如 4 月 31 日，默认输出操作符甚至在其输出中发出如下信号:[由于一些不一致，无效日期的确切格式可能与 C++20 标准不匹配，该标准在这里指定“不是有效日期”。]

```
2021-04-31 is not a valid year_month_day
```

- 当在类型为 std::chrono::sys_days 的日期上添加月或年时，结果不是类型为 sys_days。与往常一样，在 chrono 库中，结果具有能够表示结果的最佳类型:
  - 加上月份会得到一个单位为 54 秒的类型。
  - 加上年份将得到一个单位为 216 秒的类型。

  这两个单位都是秒的倍数，们可以用作 std::chrono::sys_seconds。通常，相应的输出操作符以秒为单位打印日期和时间，这不是相同的日期和后一个月或年的时间。两个时间点都不再是一个月的 31 号或最后一天，时间也不再是午夜:

```
2021-05-01 17:56:24
2024-12-30 23:16:48
```

两者都是有用的，但使用带有时间点的月份和年份通常只适用于计算许多月份和/或年份之后的大致日期。

11.5.3 解析时间点和时间段

若有一个时间点或时间段，可以访问不同的字段，如下程序所示:

lib/chronoattr.cpp

```cpp
#include <chrono>
#include <iostream>

int main()
{
  auto now = std::chrono::system_clock::now(); // type is sys_time<>
  auto today = std::chrono::floor<std::chrono::days>(now); // type is sys_days
  std::chrono::year_month_day ymd{today};
  std::chrono::hh_mm_ss hms{now - today};
  std::chrono::weekday wd{today};

  std::cout << "now: " << now << '\n';
  std::cout << "today: " << today << '\n';
  std::cout << "ymd: " << ymd << '\n';
  std::cout << "hms: " << hms << '\n';
  std::cout << "year: " << ymd.year() << '\n';
  std::cout << "month: " << ymd.month() << '\n';
  std::cout << "day: " << ymd.day() << '\n';
  std::cout << "hours: " << hms.hours() << '\n';
  std::cout << "minutes: " << hms.minutes() << '\n';
  std::cout << "seconds: " << hms.seconds() << '\n';
  std::cout << "subsecs: " << hms.subseconds() << '\n';
  std::cout << "weekday: " << wd << '\n';

  try {
    std::chrono::sys_info info{std::chrono::current_zone()->get_info(now)};
    std::cout << "timezone: " << info.abbrev << '\n';
  }
  catch (const std::exception& e) {
    std::cerr << "no timezone database: (" << e.what() << ")\n";
  }
}
```

程序的输出可能如下所示:

```
now:       2021-04-02 13:37:34.059858000
today:     2021-04-02
```

```
ymd:       2021-04-02
hms:       13:37:34.059858000
year:      2021
month:     Apr
day:       02
hours:     13h
minutes:   37min
seconds:   34s
subsecs:   59858000ns
weekday:   Fri
timezone: CEST
```

now() 产生一个与系统时钟粒度相同的时间点:

```
auto now = std::chrono::system_clock::now();
```

结果具有 std::chrono::sys_time<> 类型和一些特定于实现的解析时间段类型。

若要处理本地时间，必须实现以下几点:

```
auto tpLoc = std::chrono::zoned_time{std::chrono::current_zone(),
                    std::chrono::system_clock::now()
                  }.get_local_time();
```

为了处理时间点的日期部分，需要按天数粒度:

```
auto today = std::chrono::floor<std::chrono::days>(now);
```

现在初始化的变量的类型是 std::chrono::sys_days，可以把它赋值给 std::chrono::year_month_day 类型的对象，这样就可以用相应的成员函数访问 year、month 和 day:

```
std::chrono::year_month_day ymd{today};
std::cout << "year: " << ymd.year() << '\n';
std::cout << "month: " << ymd.month() << '\n';
std::cout << "day: " << ymd.day() << '\n';
```

为了处理时间点的时间部分，需要一个时间段，当计算原始时间点和午夜值之间的差值时，就会得到这个时间段。这里，使用 duration 来初始化 hh_mm_ss 对象:

```
std::chrono::hh_mm_ss hms{now - today};
```

可以直接得到小时、分钟、秒和亚秒:

```
std::cout << "hours: " << hms.hours() << '\n';
std::cout << "minutes: " << hms.minutes() << '\n';
std::cout << "seconds: " << hms.seconds() << '\n';
std::cout << "subsecs: " << hms.subseconds() << '\n';
```

对于亚秒，hh_mm_ss 决定必要的粒度并使用适当的单位。例子中，其输出为纳秒:

```
hms:      13:37:34.059858000
hours:    13h
minutes:  37min
seconds:  34s
subsecs:  59858000ns
```

对于工作日，只需要用粒度日类型初始化即可。这适用于 sys_days、local_days 和 year_month_day(后者可隐式转换为 std::sys_days):

```
std::chrono::weekday wd{today}; // OK (today has day granularity)
std::chrono::weekday wd{ymd}; // OK due to implicit conversion to sys_days
```

对于时区，必须将时间点 (这里是现在) 与时区 (这里是当前时区) 结合起来。生成的 std::chrono::sys_info 对象包含如下信息:

```
std::chrono::sys_info info{std::chrono::current_zone()->get_info(now)};
std::cout << "timezone: " << info.abbrev << '\n';
```

注意，并非每个 C++ 平台都支持时区数据库。某些系统上，这部分代码可能会抛出异常。

11.6. 时区

大国或国际交流中，仅仅说“中午见面吧”是不够的，因为我们有不同的时区。新 chrono 库支持这一情况，它有一个 API 来处理地球上不同的时区，包括处理标准时间 (“冬季”) 和夏令时 (“夏季”)。

11.6.1 时区的特点

处理时区有点棘手，因为这是一个非常复杂的主题。例如，必须考虑以下几点:

- 时区差异不一定是数小时，也有 30 分钟甚至 15/45 分钟的差异。例如，澳大利亚的很大一部分 (北领地和南澳大利亚与阿德莱德) 的标准时区是 UTC+9:30，尼泊尔的时区是 UTC+5:45。两个时区也可能相差 0 分钟 (例如：北美和南美的时区)。
- 时区缩写可能指不同的时区，CST 可能代表中央标准时间 (芝加哥、墨西哥城和哥斯达黎加的标准时区) 或中国标准时间 (北京和上海时区的国际名称)，或古巴标准时间 (哈瓦那的标准时区)。
- 时区变化，当国家决定改变他们的时区或夏令时开始时。这种情况就会发生，所以在处理时区时，可能不得不考虑每年的多次更新。

11.6.2 IANA 时区数据库

为了处理时区，C++ 标准库使用 IANA 时区数据库，该数据库可在http://www.iana.org/time-zones上获得。注意，时区数据库可能不是在所有平台上都可用。

时区数据库的关键项是时区名称：

- 代表时区的某一地区或国家的城市。
  例如: America/Chicago, Asia/Hong_Kong, Europe/Berlin, Pacific/Honolulu
- UTC 时间的负偏移 GMT。
  例如: Etc/GMT 或 Etc/GMT+6 表示 UTC-6, 或 Etc/GMT-8 表示 UTC+8
  UTC 时区偏移量有意地反转为 GMT，不能搜索像 UTC+6 或 UTC+5:45 这样的东西。

支持一些额外的规范和别名 (例如，UTC 或 GMT)，也可以使用一些已弃用的时区 (例如，PST8PDT、US/Hawaii、Canada/Central 或仅日本)。但是，不能搜索单个时区缩写项，如 CST 或 PST。我们将在后面讨论如何处理时区缩写。

另请参见http://en.wikipedia.org/wiki/List_of_tz_database_time_zones获取时区名称列表。

访问时区数据库

IANA 时区数据库每年都会进行多次更新，以确保在时区更改时数据库是最新的，以便程序能够做出相应的反应。

系统处理时区数据库的方式是特定于实现的，操作系统必须决定如何提供必要的数据，以及如何使其保持最新。时区数据库的更新通常作为操作系统更新的一部分进行，此类更新通常需要重新启动计算机，因此在更新期间不会运行 C++ 应用程序。

C++20 标准提供了对该数据库的低级支持，用于处理时区的高级函数:

- std::chrono::get_tzdb_list() 产生对时区数据库的引用，该数据库是 std::chrono::tzdb_list 类型的单例。
  它是一个时区数据库列表，能够并行支持多个版本的时区数据库。
- std::chrono::get_tzdb() 产生对 Std::chrono::tzdb 类型的当前时区数据库的引用。该类型有以下多个成员:
  - version, 数据库版本的字符串
  - zones, 一个类型为 std::chrono::time_zone 的时区信息向量

  处理时区的标准函数 (例如 std::chrono::current_zone() 或 std::chrono::zoned_time 类型的构造函数) 会在内部使用。
  例如，std::chrono::current_zone() 是以下方式的快捷方式:

```
std::chrono::get_tzdb().current_zone()
```

当平台不提供时区数据库时，这些函数将抛出 std::runtime_error 类型的异常。

对于支持更新 IANA 时区数据库而不需要重新启动的系统，提供了 std::chrono::reload_tzdb()。更新不会从旧数据库中删除内存，应用可能仍然有指向它的 (time_zone) 指针。相反，新数据库被自动推到时区数据库列表的前面。

可以使用 get_tzdb() 返回的 tzdb 类型的字符串成员版本，来检查时区数据库的当前版本。例如:

```
std::cout << "tzdb version: " << chr::get_tzdb().version << '\n';
```

通常，输出是年份和该年份更新的升序字母字符 (例如，“2021b”)。remote_version() 提供最新可用时区数据库的版本，使用它来决定是否调用 reload_tzdb()。

若一个长时间运行的程序正在使用 chrono 时区数据库，但从不调用 reload_tzdb()，则该程序将不会意识到数据库的更新，其将继续使用程序第一次访问数据库时存在的数据库版本。

有关为长时间运行的程序重新加载 IANA 时区数据库的详细信息和示例，请参阅http://github.com/HowardHinnant/date/wiki/Examples-and-Recipes#tzdb_manage。

11.6.3 使用时区

要处理时区，要使用两种类型:

- std::chrono::time_zone，表示特定时区的类型。
- std::chrono::zoned_time，表示与特定时区相关联的特定时间点的类型。

详细地了解一下。

time_zone 类型

所有可能的时区值都由 IANA 时区数据库预定义，所以不能仅仅通过声明 time_zone 对象来创建。这些值来自时区数据库，通常处理的是指向这些对象的指针:

- std::chrono::current_zone() 生成指向当前时区的指针。
- std::chrono::locate_zone(name) 生成一个指向时区名称的指针。
- 由 std::chrono::get_tzdb() 返回的时区数据库具有包含所有时区的非指针集合:
  - 成员区域拥有所有规范。
  - 成员链接拥有所有带有链接的别名。

  可以使用根据诸如缩写名称之类的特征来查找时区。

例如:

```
auto tzHere = std::chrono::current_zone(); // type const time_zone*
auto tzUTC = std::chrono::locate_zone("UTC"); // type const time_zone*
...
std::cout << tzHere->name() << '\n';
std::cout << tzUTC->name() << '\n';
```

输出取决于当前的时区。对我来说，德国是这样的:

```
Europe/Berlin
Etc/UTC
```

可以搜索时区数据库中的条目 (如“UTC”)，但得到的是它的规范。例如，”UTC” 只是指向”Etc/UTC” 的时区链接。若 locate_zone() 没有找到与该名称对应的信息，则抛出 std::runtime_error 异常。

time_zone 不能做很多事情，但最重要的是将它与系统时间点或本地时间点结合起来使用。

若输出一个 time_zone，会得到一些特定于实现的输出，仅用于调试:

```
std::cout << *tzHere << '\n'; // some implementation-specific debugging output
```

chrono 库还允许定义和使用自定义时区的类型。

zoned_time 类型

std::chrono::zoned_time 类型的对象将时间点应用于时区。执行此转换有两个选项:

- 将系统时间点 (属于系统时钟的时间点) 应用于时区，将事件的时间点转换为与另一个时区的本地时间同时发生。
- 将本地时间点 (属于伪时钟 local_t 的时间点) 应用于时区，将一个时间点作为本地时间应用到另一个时区。

此外，可以通过用另一个带 zoned_time 对象初始化新的 zoned_time 对象，将 zoned_time 的时间点转换为不同的时区。

例如，将每个办公室安排 18:00 的本地派对，并在 2021 年 9 月底安排一个跨越多个时区的公司派对。

```
auto day = 2021y/9/chr::Friday[chr::last]; // last Friday of month
chr::local_seconds tpOfficeParty{chr::local_days{day} - 6h}; // 18:00 the day before
chr::sys_seconds tpCompanyParty{chr::sys_days{day} + 17h}; // 17:00 that day

std::cout << "Berlin Office and Company Party:\n";
std::cout << " " << chr::zoned_time{"Europe/Berlin", tpOfficeParty} << '\n';
std::cout << " " << chr::zoned_time{"Europe/Berlin", tpCompanyParty} << '\n';

std::cout << "New York Office and Company Party:\n";
std::cout << " " << chr::zoned_time{"America/New_York", tpOfficeParty} << '\n';
std::cout << " " << chr::zoned_time{"America/New_York", tpCompanyParty} << '\n';
```

这段代码有如下输出:

```
Berlin Office and Company Party:
  2021-09-23 18:00:00 CEST
  2021-09-24 19:00:00 CEST
New York Office and Company Party:
  2021-09-23 18:00:00 EDT
  2021-09-24 13:00:00 EDT
```

组合时间点和时区时。例如，下面的代码:

```cpp
auto sysTp = chr::floor<chr::seconds>(chr::system_clock::now()); // system timepoint
auto locTime = chr::zoned_time{chr::current_zone(), sysTp}; // local time
...
std::cout << "sysTp:          " << sysTp << '\n';
std::cout << "locTime:        " << locTime << '\n';
```

首先，将 sysTp 初始化为以秒为单位的当前系统时间点，并将该时间点与当前时区结合起来。输出显示是同一时间点的系统和本地时间点:

```
sysTp:      2021-04-13 13:40:02
locTime:    2021-04-13 15:40:02 CEST
```

现在初始化一个本地时间点。一种方法是将系统时间点转换为本地时间点，现在需要一个时区。若使用当前时区，本地时间将转换为 UTC:

```cpp
auto sysTp = chr::floor<chr::seconds>(chr::system_clock::now()); // system timepoint
auto curTp = chr::current_zone()->to_local(sysTp); // local timepoint
std::cout << "sysTp:             " << sysTp << '\n';
std::cout << "locTp:             " << locTp << '\n';
```

输出结果如下:

```
sysTp:      2021-04-13 13:40:02
curTp:      2021-04-13 13:40:02
```

但若使用 UTC 作为时区，则本地时间用作本地时间，并不会关联时区:

```cpp
auto sysTp = chr::floor<chr::seconds>(chr::system_clock::now()); // system timepoint
auto locTp = std::chrono::locate_zone("UTC")->to_local(sysTp); // use local time as
↪  is
std::cout << "sysTp:               " << sysTp << '\n';
std::cout << "locTp:               " << locTp << '\n';
```

根据输出，两个时间点看起来是一样的:

```
sysTp:      2021-04-13 13:40:02
locTp:      2021-04-13 13:40:02
```

然而，它们是不一样的。sysTp 有一个相关的 UTC epoch，而 locTp 没有。若现在将本地时间点应用于时区，则不会进行转换。这里指定时区，保持时间不变:

```cpp
auto timeFromSys = chr::zoned_time{chr::current_zone(), sysTp}; // converted time
auto timeFromLoc = chr::zoned_time{chr::current_zone(), locTp}; // applied time
std::cout << "timeFromSys: " << timeFromSys << '\n';
std::cout << "timeFromLoc: " << timeFromLoc << '\n';
```

输出如下:

```
timeFromSys: 2021-04-13 15:40:02 CEST
timeFromLoc: 2021-04-13 13:40:02 CEST
```

现在将这四个对象与纽约时区结合起来:

```
std::cout << "NY sysTp: "
      << std::chrono::zoned_time{"America/New_York", sysTp} << '\n';
std::cout << "NY locTP: "
      << std::chrono::zoned_time{"America/New_York", locTp} << '\n';
std::cout << "NY timeFromSys: "
      << std::chrono::zoned_time{"America/New_York", timeFromSys} << '\n';
std::cout << "NY timeFromLoc: "
      << std::chrono::zoned_time{"America/New_York", timeFromLoc} << '\n';
```

输出结果如下:

```
Y sysTp:         2021-04-13 09:40:02 EDT
NY locTP:        2021-04-13 13:40:02 EDT
NY timeFromSys: 2021-04-13 09:40:02 EDT
NY timeFromLoc: 2021-04-13 07:40:02 EDT
```

系统时间点和由它导出的本地时间，都将现在的时间转换为纽约时区。与往常一样，将本地时间点应用于纽约，因此现在拥有与删除时区后的原始时间相同的值。timeFromLoc 是中欧的初始当地时间 13:40:02，适用于纽约时区。

11.6.4 处理时区缩写

由于时区缩写可能引用不同的时区，因此不能通过缩写定义唯一的时区。相反，必须将缩写映射到多个 IANA 时区项中的一个。

下面的程序演示了时区缩写 CST 的用法:

lib/chronocst.cpp

```
#include <iostream>
#include <chrono>
using namespace std::literals;

int main(int argc, char** argv)
{
  auto abbrev = argc > 1 ? argv[1] : "CST";

  auto day = std::chrono::sys_days{2021y/1/1};
  auto& db = std::chrono::get_tzdb();

  // print time and name of all timezones with abbrev:
```

```
  std::cout << std::chrono::zoned_time{"UTC", day}
        << " maps to these '" << abbrev << "' entries:\n";
  // iterate over all timezone entries:
  for (const auto& z : db.zones) {
    // and map to those using my passed (or default) abbreviation:
    if (z.get_info(day).abbrev == abbrev) {
      std::chrono::zoned_time zt{&z, day};
      std::cout << " " << zt << " " << z.name() << '\n';
    }
  }
}
```

不传递命令行参数或“CST”的程序可能会有如下输出

```
2021-01-01 00:00:00 UTC maps these ’ CST’ entries:
  2020-12-31 18:00:00 CST America/Bahia_Banderas
  2020-12-31 18:00:00 CST America/Belize
  2020-12-31 18:00:00 CST America/Chicago
  2020-12-31 18:00:00 CST America/Costa_Rica
  2020-12-31 18:00:00 CST America/El_Salvador
  2020-12-31 18:00:00 CST America/Guatemala
  2020-12-31 19:00:00 CST America/Havana
  2020-12-31 18:00:00 CST America/Indiana/Knox
  2020-12-31 18:00:00 CST America/Indiana/Tell_City
  2020-12-31 18:00:00 CST America/Managua
  2020-12-31 18:00:00 CST America/Matamoros
  2020-12-31 18:00:00 CST America/Menominee
  2020-12-31 18:00:00 CST America/Merida
  2020-12-31 18:00:00 CST America/Mexico_City
  ...
  2020-12-31 18:00:00 CST America/Winnipeg
  2021-01-01 08:00:00 CST Asia/Macau
  2021-01-01 08:00:00 CST Asia/Shanghai
  2021-01-01 08:00:00 CST Asia/Taipei
  2020-12-31 18:00:00 CST CST6CDT
```

因为 CST 可能代表中央标准时间、中国标准时间或古巴标准时间，所以可以看到美国和中国的大多数时区之间有 14 个小时的差异。此外，古巴的哈瓦那与这些时区有 1 或 13 个小时的时差。

注意，当在夏季的某一天搜索“CST”时，输出要小得多，因为美国条目和古巴的时区会切换到“CDT”(相应的夏令时)。然而，仍然需要一些信息，因为中国和哥斯达黎加没有夏令时。

对于 CST，可能根本找不到时区，因为时区数据库不可用，或者使用 GMT-6 之类的东西代替了 CST。

11.6.5 自定义的时区

chrono 库允许使用自定义时区。一个常见的例子是需要有一个时区，与 UTC 有一个直到运行时才知晓的偏移量。

下面是一个提供自定义时区 OffsetZone 的示例，该时区可以保存以分钟为精度的 UTC 偏移量:

lib/offsetzone.hpp

```
#include <chrono>
#include <iostream>
#include <type_traits>

class OffsetZone
{
  private:
  std::chrono::minutes offset; // UTC offset
  public:
  explicit OffsetZone(std::chrono::minutes offs)
  : offset{offs} {
  }

  template<typename Duration>
  auto to_local(std::chrono::sys_time<Duration> tp) const {
    // define helper type for local time:
    using LT
    = std::chrono::local_time<std::common_type_t<Duration,
                          std::chrono::minutes>>;
    // convert to local time:
    return LT{(tp + offset).time_since_epoch()};
  }

  template<typename Duration>
  auto to_sys(std::chrono::local_time<Duration> tp) const {
    // define helper type for system time:
    using ST
    = std::chrono::sys_time<std::common_type_t<Duration,
                          std::chrono::minutes>>;
    // convert to system time:
    return ST{(tp - offset).time_since_epoch()};
  }

  template<typename Duration>
  auto get_info(const std::chrono::sys_time<Duration>& tp) const {
    return std::chrono::sys_info{};
  }
};
```

要定义的只是本地时间和系统时间之间的转换。

可以像使用任何其他 time_zone 指针一样使用 timezone:

lib/offsetzone.cpp

```
#include "offsetzone.hpp"
#include <iostream>

```

```cpp
int main()
{
  using namespace std::literals; // for h and min suffix
  // timezone with 3:45 offset:
  OffsetZone p3_45{3h + 45min};

  // convert now to timezone with offset:
  auto now = std::chrono::system_clock::now();
  std::chrono::zoned_time<decltype(now)::duration, OffsetZone*> zt{&p3_45, now};

  std::cout << "UTC: " << zt.get_sys_time() << '\n';
  std::cout << "+3:45: " << zt.get_local_time() << '\n';
  std::cout << zt << '\n';
}
```

程序可能有以下输出:

```
UTC:    2021-05-31 13:01:19.0938339
+3:45:  2021-05-31 16:46:19.0938339
```

11.7. 时钟的详情

C++20 现在支持两种时钟，本节讨论它们之间的区别，以及如何使用特殊时钟。

11.7.1 具有指定 epoch 的时钟

C++20 现在提供了以下与 epoch 相关联的时钟 (以便定义唯一的时间点):

- 系统时钟是操作系统的时钟。从 C++20 开始，其指定为 Unix Time[有关 Unix 时间的详细信息，例如，http://en.wikipedia.org/wiki/Unix_time.]，计算从 UTC 时间 1970 年 1 月 1 日 00:00:00 开始的时间。
  闰秒的处理方式使得某些秒可能会花费更长的时间，所以永远不会有 61 秒的小时，而所有 365 天的年都有相同的 31,536,000 秒。
- UTC 时钟是代表协调世界时的时钟，通常称为 GMT(格林威治标准时间) 或祖鲁时间。本地时间与 UTC 的差异在于您所在时区的 UTC 偏移量。
  它使用与系统时钟相同的历元 (1970 年 1 月 1 日 00:00:00 UTC)。
  闰秒的处理使得某些分钟可能有 61 秒。例如，有一个时间点 1972-06-30 23:59:60，因为在 1972 年 6 月的最后一分钟增加了一个闰秒。
  因此，365 天的年份有时可能有 31,536,001，甚至 31,536,002 秒。
- GPS 时钟使用的是全球定位系统的时间，是 GPS 地面控制站和卫星上的原子钟实现的原子时标。GPS 时间从 1980 年 1 月 6 日 00:00:00 UTC 的 epoch 开始。
  每分钟有 60 秒，但 GPS 通过提前切换到下一个小时来考虑闰秒，所以 GPS 比 UTC 快越来越多的秒 (或者在 1980 年之前比 UTC 慢越来越多的秒)。例如，时间点“20121-01-01 00:00:00

UTC”表示为 GPS 时间点“20121-01-01 00:00:18 GPS”。在 2021 年，当我写这本书的时候，GPS 时间点领先了 18 秒。

所有具有 365 天的 GPS 年 (GPS 日期的午夜与一年后 GPS 日期的午夜之间的差值) 都有相同的 31,536,000 秒，但可能比“真正的”年短一到两秒。

- TAI 时钟使用国际原子时，这是基于 SI 秒连续计数的国际原子时标。TAI 时间从 UTC 时间 1958 年 1 月 1 日 00:00:00 开始。

  就像 GPS 时间一样，每分钟有 60 秒，而闰秒会通过提前切换到下一个小时来考虑，所以 TAI 比 UTC 快越来越多的秒，但与 GPS 总是有 19 秒的恒定偏移。例如，时间点“2021-01-01 00:00:00 UTC”表示为 TAI 时间点“2021-01-01 00:00:37 TAI”。在 2021 年，当我写这本书的时候，TAI 的时间点提前了 37 秒。

11.7.2 伪时钟 local_t

如前所述，有一个类型为 std::chrono::local_t 的特殊时钟。这个时钟允许指定没有时区 (甚至还没有 UTC) 的本地时间点。其 epoch 解释为“本地时间”，所以必须将它与时区结合起来才能知道代表的是哪个时间点。

local_t 是一个“伪时钟”，它不满足时钟的所有要求。事实上，它现在没有提供成员函数 now():

```
auto now1 = std::chrono::local_t::now(); // ERROR: now() not provided
```

相反，若需要一个系统时钟时间点和一个时区，可以使用时区 (如当前时区或 UTC) 将时间点转换为本地时间点:

```
auto sysNow = chr::system_clock::now(); // NOW as UTC timepoint
...
chr::local_time now2
  = chr::current_zone()->to_local(sysNow); // NOW as local timepoint
chr::local_time now3
  = chr::locate_zone("Asia/Tokyo")->to_local(sysNow); // NOW as Tokyo timepoint
```

另一种获得相同结果的方法是调用 get_local_time() 来获取分区时间 (具有关联时区的时间点):

```
chr::local_time now4 = chr::zoned_time{chr::current_zone(),
                     sysNow}.get_local_time();
```

另一种方法是将字符串解析为本地时间点:

```
chr::local_seconds tp; // time_point<local_t, seconds>
std::istringstream{"2021-1-1 18:30"} >> chr::parse(std::string{"%F %R"}, tp);
```

记住本地时间点与其他时间点之间的细微差别:

- 系统/UTC/GPS/TAI 时间点表示一个特定的时间点，将应用于时区将转换它所表示的时间值。
- 本地时间点表示本地时间。当与时区结合，全局时间点就会变得清晰。

例如:

```
auto now = chr::current_zone()->to_local(chr::system_clock::now());
std::cout << now << '\n';
std::cout << "Berlin: " << chr::zoned_time("Europe/Berlin", now) << '\n';
std::cout << "Sydney: " << chr::zoned_time("Australia/Sydney", now) << '\n';
std::cout << "Cairo: " << chr::zoned_time("Africa/Cairo", now) << '\n';
```

这里，将当前的本地时间应用于三个不同的时区:

```
2021-04-14 08:59:31.640004000
Berlin: 2021-04-14 08:59:31.640004000 CEST
Sydney: 2021-04-14 08:59:31.640004000 AEST
Cairo:  2021-04-14 08:59:31.640004000 EET
```

当使用本地时间点时，不能对时区使用转换说明符:

```
chr::local_seconds tp; // time_point<local_t, seconds>
...
std::cout << std::format("{:%F %T %Z}\n", tp); // ERROR: invalid format
std::cout << std::format("{:%F %T}\n", tp); // OK
```

11.7.3 处理闰秒

前面对具有指定 epoch 的时钟的讨论已经介绍了处理闰秒的基本方式。

为了更清楚地处理闰秒，我们使用不同的时钟迭代闰秒的时间点:

lib/chronoclocks.cpp

```
#include <iostream>
#include <chrono>

int main()
{
  using namespace std::literals;
  namespace chr = std::chrono;

  auto tpUtc = chr::clock_cast<chr::utc_clock>(chr::sys_days{2017y/1/1} - 1000ms);
  for (auto end = tpUtc + 2500ms; tpUtc <= end; tpUtc += 200ms) {
    auto tpSys = chr::clock_cast<chr::system_clock>(tpUtc);
    auto tpGps = chr::clock_cast<chr::gps_clock>(tpUtc);
    auto tpTai = chr::clock_cast<chr::tai_clock>(tpUtc);
    std::cout << std::format("{:%F %T} SYS ", tpSys);
    std::cout << std::format("{:%F %T %Z} ", tpUtc);
    std::cout << std::format("{:%F %T %Z} ", tpGps);
    std::cout << std::format("{:%F %T %Z}\n", tpTai);
  }
}
```

该程序有以下输出:

```
2016-12-31 23:59:59.000 SYS 2016-12-31 23:59:59.000 UTC 2017-01-01 00:00:16.000 GPS
↪  2017-01-01 00:00:35.000 TAI
2016-12-31 23:59:59.200 SYS 2016-12-31 23:59:59.200 UTC 2017-01-01 00:00:16.200 GPS
↪  2017-01-01 00:00:35.200 TAI
2016-12-31 23:59:59.400 SYS 2016-12-31 23:59:59.400 UTC 2017-01-01 00:00:16.400 GPS
↪  2017-01-01 00:00:35.400 TAI
2016-12-31 23:59:59.600 SYS 2016-12-31 23:59:59.600 UTC 2017-01-01 00:00:16.600 GPS
↪  2017-01-01 00:00:35.600 TAI
2016-12-31 23:59:59.800 SYS 2016-12-31 23:59:59.800 UTC 2017-01-01 00:00:16.800 GPS
↪  2017-01-01 00:00:35.800 TAI
2016-12-31 23:59:59.999 SYS 2016-12-31 23:59:60.000 UTC 2017-01-01 00:00:17.000 GPS
↪  2017-01-01 00:00:36.000 TAI
2016-12-31 23:59:59.999 SYS 2016-12-31 23:59:60.200 UTC 2017-01-01 00:00:17.200 GPS
↪  2017-01-01 00:00:36.200 TAI
2016-12-31 23:59:59.999 SYS 2016-12-31 23:59:60.400 UTC 2017-01-01 00:00:17.400 GPS
↪  2017-01-01 00:00:36.400 TAI
2016-12-31 23:59:59.999 SYS 2016-12-31 23:59:60.600 UTC 2017-01-01 00:00:17.600 GPS
↪  2017-01-01 00:00:36.600 TAI
2016-12-31 23:59:59.999 SYS 2016-12-31 23:59:60.800 UTC 2017-01-01 00:00:17.800 GPS
↪  2017-01-01 00:00:36.800 TAI
2017-01-01 00:00:00.000 SYS 2017-01-01 00:00:00.000 UTC 2017-01-01 00:00:18.000 GPS
↪  2017-01-01 00:00:37.000 TAI
2017-01-01 00:00:00.200 SYS 2017-01-01 00:00:00.200 UTC 2017-01-01 00:00:18.200 GPS
↪  2017-01-01 00:00:37.200 TAI
2017-01-01 00:00:00.400 SYS 2017-01-01 00:00:00.400 UTC 2017-01-01 00:00:18.400 GPS
↪  2017-01-01 00:00:37.400 TAI
```

这里的闰秒是写这本书时的最后一个闰秒 (事先不知道闰秒在未来什么时候会发生)，输出带有相应时区的时间点 (对于系统时间点，这里输出的是 SYS，而非其默认时区 UTC)。可以观察到以下几点:

- 闰秒时:
  - UTC 时间的秒可到 60。
  - 系统时钟使用在插入闰秒之前最后一个可表示的 sys_time 值，行为由 C++ 标准保证。
- 闰秒前:
  - GPS 时间比 UTC 时间快 17 秒。
  - TAI 时间比 UTC 时间早 36 秒 (一如既往，比 GPS 时间早 19 秒)。
- 闰秒后:
  - GPS 时间比 UTC 时间快 18 秒。
  - TAI 时间比 UTC 时间早 37 秒 (仍比 GPS 时间早 19 秒)。

11.7.4 时钟间的转换

可以在时钟间转换时间点，前提是这种转换有意义。为此，chrono 库提供了一个 clock_cast<>，只能在具有指定稳定 epoch (sys_time<>、utc_time<>、gps_time<>、tai_time<>) 的时钟时间点和文件系统时间点间进行转换。

强制转换需要目标时钟，可以选择传递不同的时间段。

下面的程序创建一个 UTC 闰秒到其他几个时钟的输出:

lib/chronoconv.cpp

```
#include <iostream>
#include <sstream>
#include <chrono>

int main()
{
  namespace chr = std::chrono;

  // initialize a utc_time<> with a leap second:
  chr::utc_time<chr::utc_clock::duration> tp;
  std::istringstream{"2015-6-30 23:59:60"}
    >> chr::parse(std::string{"%F %T"}, tp);

  // convert it to other clocks and print that out:
  auto tpUtc = chr::clock_cast<chr::utc_clock>(tp);
  std::cout << "utc_time: " << std::format("{:%F %T %Z}", tpUtc) << '\n';
  auto tpSys = chr::clock_cast<chr::system_clock>(tp);
  std::cout << "sys_time: " << std::format("{:%F %T %Z}", tpSys) << '\n';
  auto tpGps = chr::clock_cast<chr::gps_clock>(tp);
  std::cout << "gps_time: " << std::format("{:%F %T %Z}", tpGps) << '\n';
  auto tpTai = chr::clock_cast<chr::tai_clock>(tp);
  std::cout << "tai_time: " << std::format("{:%F %T %Z}", tpTai) << '\n';
  auto tpFile = chr::clock_cast<chr::file_clock>(tp);
  std::cout << "file_time: " << std::format("{:%F %T %Z}", tpFile) << '\n';
}
```

该程序有以下输出:

```
utc_time: 2015-06-30 23:59:60.0000000 UTC
sys_time: 2015-06-30 23:59:59.9999999 UTC
gps_time: 2015-07-01 00:00:16.0000000 GPS
tai_time: 2015-07-01 00:00:35.0000000 TAI
file_time: 2015-06-30 23:59:59.9999999 UTC
```

对于所有这些时钟，都为格式化的输出提供了伪时区。

转换规则如下:

- 从本地时间点进行的任何转换都只添加 epoch，时间值保持不变。

- UTC, GPS 和 TAI 时间点之间的转换增加或减少必要的偏移量。
- UTC 和系统时间之间的转换不改变时间值，但对于 UTC 闰秒的时间点，使用前面的最后一个时间点作为系统时间。

还支持从本地时间点到其他时钟的转换，大门要转换为本地时间点，必须使用 to_local()(在转换为系统时间点之后)。

不支持与 steady_clock 的时间点之间的转换。

内部时钟的转换

在内部，clock_cast<> 是一个“双轮毂辐条和车轮系统”。这两个中心是 system_clock 和 utc_clock，每个可转换时钟都必须转换到这些中心中的一个 (但不是两个)。到中心的转换是通过时钟的 to_sys() 或 to_utc() 静态成员函数完成的。对于来自中心的转换，提供了 from_sys() 或 from_utc()。clock_cast<> 将这些成员函数串在一起，用于将一种时钟转换为其他时钟的表述。

不能处理闰秒的时钟应该转换为/从 system_clock。例如，utc_clock 和 local_t 提供给 to_sys()。

可以以某种方式处理闰秒的时钟 (这并不一定说明其具有闰秒值) 应该转换为/从 utc_clock。这适用于 gps_clock 和 tai_clock，即使 gps 和 tai 没有闰秒，也有唯一到 UTC 闰秒的双向映射。

对于 file_clock，是否提供与 system_clock 或 utc_clock 之间的转换取决于具体实现。

11.7.5 处理文件时钟

时钟 std::chrono::file_clock 是文件系统库用于文件系统 (文件、目录等) 的时间点的时钟，是一种特定于实现的时钟类型，反映文件系统时间值的分辨率和范围。

例如，可以使用文件时钟来更新最后访问文件的时间:

```
// touch file with path p (update last write access to file):
std::filesystem::last_write_time(p,
                 std::chrono::file_clock::now());
```

还可以使用 C++17 中的文件系统时钟

```
std::filesystem::file_time_type:
  std::filesystem::last_write_time(p,
                   std::filesystem::file_time_type::clock::now());
```

从 C++20 开始，文件系统类型名称 file_time_type 定义如下:

```
namespace std::filesystem {
  using file_time_type = chrono::time_point<chrono::file_clock>;
}
```

C++17 中，只能使用未指定的普通时钟。

对于文件系统的时间点，现在也定义了类型 file_time:

```cpp
namespace std::chrono {
  template<typename Duration>
  using file_time = time_point<file_clock, Duration>;
}
```

没有定义像 file_seconds 这样的类型 (就像其他时钟一样)。

file_time 类型的新定义现在允许开发者可移植地将文件系统时间点转换为系统时间点。例如，可以输出最后一次访问传入文件的时间:

```cpp
void printFileAccess(const std::filesystem::path& p)
{
  std::cout << "\"" << p.string() << "\":\n";

  auto tpFile = std::filesystem::last_write_time(p);
  std::cout << std::format(" Last write access: {0:%F} {0:%X}\n", tpFile);

  auto diff = std::chrono::file_clock::now() - tpFile;
  auto diffSecs = std::chrono::round<std::chrono::seconds>(diff);
  std::cout << std::format(" It is {} old\n", diffSecs);
}
```

这段代码可能的输出:

```
"chronoclocks.cpp":
 Last write access: 2021-07-12 16:50:08
 It is 18s old
```

若要使用默认的时间点输出操作符，其会根据文件时钟的粒度输出亚秒:

```cpp
std::cout << " Last write access: " << diffSecs << '\n';
```

输出可能如下所示:

```
Last write access: 2021-07-12 16:50:08.3680536
```

要将文件访问时间作为系统时间或本地时间处理，必须使用 clock_cast<>()(内部可能会将静态 file_clock 成员函数调用 to_sys() 或 to_utc())。例如:

```cpp
auto tpFile = std::filesystem::last_write_time(p);
auto tpSys = std::chrono::file_clock::to_sys(tpFile);
auto tpSys = std::chrono::clock_cast<std::chrono::system_clock>(tpFile);
```

11.8. Chrono 的其他新功能

新的 chrono 库还增加了以下特性:

- 为了检查一个类型是否为时钟，提供了一个新的类型特征，std::chrono::is_clock<> 及其相应的变量模板 std::chrono::is_clock_v<>。例如:

```
std::chrono::is_clock_v<std::chrono::system_clock> // true
std::chrono::is_clock_v<std::chrono::local_t> // false
```

  伪时钟 local_t 在这里产生 false，不提供成员函数 now()。
- 对于时间段和时间点，定义了操作符 <=>。

11.9. 附注

chrono 库是由 Howard Hinnant 开发。当时间段和时间点的基本部分在 C++11 中标准化时，计划已经扩展到支持日期、日历和时区。

C++20 的 chrono 扩展最初是由 Howard Hinnant 在http://wg21.link/p0355r0中提出。这个扩展的最终接受的提案是由 Howard Hinnant 和 Tomasz Kaminski 在http://wg21.link/p0355r7制定。

Howard Hinnant 在http://wg21.link/p1466r3和 Tomasz Kaminski 在http://wg21.link/p1650r0添加了一些小的修复。最终接受将 chrono 库与格式化库集成的提案是由 Victor Zverovich、Daniela Engert 和 Howard Hinnant 在http://wg21.link/p1361r2中提出。

完成 C++20 之后，应用了一些修复程序来修复 C++20 标准的 chrono 扩展:

- http://wg21.Link/p2372阐明了与语言环境相关的格式化输出行为。
- http://wg21.Link/lwg3554确保可以将字符串字面量作为格式传递给 parse()。

第 12 章 std::jthread 和停止令牌

C++20 引入了一个表示线程的新类型:std::jthread, 修复了 std::thread 的一个严重的设计问题，并受益于信号取消的新特性。

本章既解释了异步场景中的取消信号特性，也介绍了新类 std::jthread。

12.1. 添加 std::jthread 的动机

C++11 引入了 std::thread 类型，其与操作系统提供的线程对应，但该类型有一个严重的设计缺陷: 不是 RAII 类型。

让我们看看为什么会有这么一个问题，以及新的线程类型如何解决这个问题。

12.1.1 std::thread 的问题

std::thread 要求在其生命周期结束时，若表示正在运行的线程，则调用 join()(等待线程结束) 或 detach()(让线程在后台运行)。若两者都没有调用，析构函数会立即导致异常的程序终止 (在某些系统上导致段错误)。由于这个原因，下面的代码会出现错误 (除非不关心程序异常的终止):

```
void foo()
{
  ...
  // start thread calling task() with name and val as arguments:
  std::thread t{task, name, val};
  ... // neither t.join() nor t.detach() called
} // std::terminate() called
```

当没有调用 join() 或 detach() 就表示正在运行的线程 t 在析构时，程序会调用 std::terminate()，后者调用 std::abort()。

即使使用 join() 来等待正在运行的线程结束，仍然会有一个严重的问题:

```
void foo()
{
  ...
  // start thread calling task() with name and val as arguments:
  std::thread t{task, name, val};
  ... // calls std::terminate() on exception
  // wait for tasks to finish:
  t.join();
  ...
}
```

这段代码还可能导致程序异常终止，线程开始和调用 join() 之间的 foo() 中发生异常时 (或者控制流从未到达 join())，没有调用 t.join()。

代码如下所示:

```
void foo()
{
  ...
  // start thread calling task() with name and val as arguments:
  std::thread t{task, name, val};
  try {
    ... // might throw an exception
  }
  catch (...) { // if we have an exception
    // clean up the thread started:
    t.join(); // - wait for thread (blocks until done)
    throw; // - and rethrow the caught exception
  }
  // wait for thread to finish:
  t.join();
  ...
}
```

这里，通过确保在离开作用域时调用 join() 来对异常作出反应，而不解决异常。不幸的是，这可能会导致阻塞 (永远)。然而，调用 detach() 也是一个问题，因为线程在程序的后台继续运行，使用 CPU 时间和资源，而这些时间和资源现在可能会销毁。

若在更复杂的上下文中使用多个线程，问题会变得更糟，并且会产生非常糟糕的代码。例如，当只启动两个线程时:

```
void foo()
{
  ...
  // start thread calling task1() with name and val as arguments:
  std::thread t1{task1, name, val};
  std::thread t2;
  try {
    // start thread calling task2() with name and val as arguments:
    t2 = std::thread{task2, name, val};
    ...
  }
  catch (...) { // if we have an exception
    // clean up the threads started:
    t1.join(); // wait for first thread to end
    if (t2.joinable()) { // if the second thread was started
      t2.join(); // - wait for second thread to end
    }
    throw; // and rethrow the caught exception
  }
  // wait for threads to finish:
  t1.join();
  t2.join();
  ...
```

```
}
```

启动第一个线程之后，启动第二个线程可能会抛出异常，因此启动第二个线程必须在 try 子句中进行。另一方面，可在同一范围内使用和 join() 两个线程。为了满足这两个需求，必须前向声明第二个线程，并在第一个线程的 try 子句中对其进行赋值。此外，对于异常，必须检查第二个线程是否启动，对没有关联线程的线程对象调用 join() 会导致异常。

另一个问题是，对两个线程调用 join() 可能会花费大量时间 (甚至永远)。注意，不能“杀死”已经启动的线程。线程不是进程，线程只能通过结束自身或结束整个程序来结束。

因此，在调用 join() 之前，应该确保等待的线程将取消其执行。不过，对于 std::thread，没有这样的机制，必须自己实现取消请求和对它的响应。

12.1.2 使用 std::jthread

std::jthread 解决了这些问题，它是 RAII 类型。若线程是可汇入的 (“j”代表“汇入”)，析构函数会自动调用 join()。这让上面的复杂代码变简单了:

```
void foo()
{
  ...
  // start thread calling task1() with name and val as arguments:
  std::jthread t1{task1, name, val};
  // start thread calling task2() with name and val as arguments:
  std::jthread t2{task2, name, val};
  ...
  // wait for threads to finish:
  t1.join();
  t2.join();
  ...
}
```

使用 std::jthread 就不再存在导致异常程序终止的危险，也不需要异常处理。为了支持尽可能容易地切换到 std::jthread 类，该类提供了与 std::thread 相同的 API，包括:

- 使用相同的头文件 <thread>
- 当调用 get_id() 时返回 std::thread::id (std::jthread::id 类型只是一个别名类型)
- 提供静态成员 hardware_concurrency()

只需用 std::jthread 替换 std::thread 并重新编译，代码就会变得更安全 (若还没有自己实现异常处理的话)。[估计大家想知道为什么不直接修复 std::thread，而是引入一个新的 std::jthread 类型。其原因很简单，即向后兼容性，可能有一些应用希望在离开正在运行的线程的作用域时终止程序。对于接下来讨论的一些新功能，我们也会打破兼容性。]

12.1.3 停止令牌和停止回调

类 std::jthread 做的更多: 提供了一种使用停止令牌发出取消信号的机制，这些令牌由 jthreads 的析构函数在调用 join() 之前使用，由线程启动的可调用对象 (函数、函数对象或 Lambda) 必须支持此请求:

- 若可调用对象只为所有传递的参数提供参数，将忽略停止请求:

```
void task (std::string s, double value)
{
  ... // join() waits until this code ends
}
```

- 为了响应停止请求，可调用对象可以添加一个新的可选第一个类型为 std::stop_token 的参数，并不时检查是否请求了停止:

```
void task (std::stop_token st,
std::string s, double value)
{
  while (!st.stop_requested()) { // stop requested (e.g., by the destructor)?
    ... // ensure we check from time to time
  }
}
```

所以，std::jthread 提供了一种协作机制来表示线程不应该再运行。它是“协作的”，因为该机制不会杀死正在运行的线程 (因为 C++ 线程根本不支持杀死线程，所以杀死线程的操作可能很容易使程序处于损坏状态)。为了响应停止请求，已启动的线程必须声明停止令牌作为附加的第一个参数，并使用它不时的检查是否应该继续运行。

也可以手动请求已经启动的 jthread 停止。例如:

```
void foo()
{
  ...
  // start thread calling task() with name and val as arguments:
  std::jthread t{task, name, val};
  ...
  if ( ... ) {
    t.request_stop(); // explicitly request task() to stop its execution
  }
  ...
  // wait for thread to finish:
  t.join();
  ...
}
```

此外，还有另一种对停止请求作出反应的方法: 可以为停止令牌注册回调，该回调将在请求停止时自动调用:

```
void task (std::stop_token st,
std::string s, double value)
{
  std::stop_callback cb{st, [] {
                  ... // called on a stop request
                }};
  ...
}
```

因此，停止执行 task() 的线程的请求 (无论是显式调用 request_stop() 还是由析构函数引起) 调用已注册为停止回调的 Lambda。请注意，回调通常由请求停止的线程调用。

停止回调函数 cb 的生命周期结束时，析构函数自动注销回调函数，以便在之后发出停止信号时不再调用该回调函数，也可以通过这种方式注册任意数量的可调用对象 (函数、函数对象或 Lambda)。

停止机制比最初看起来更灵活:

- 可以传递句柄来请求停止，并传递令牌来检查请求的停止。
- 支持条件变量，这样一个有信号的停止就可以中断一个等待。
- 也可以独立于 std::jthread 使用这种机制来请求和检查停止。

线程、停止源、停止令牌和停止回调之间也没有生命周期约束，存储停止状态的位置在堆上分配。当线程和使用此状态的最后一个停止源、停止令牌或停止回调函数销毁时，用于停止状态的内存将释放。

下面几节中，我将讨论请求停止的机制及其在线程中的应用。之后，会讨论了底层的停止令牌机制。

12.1.4 停止令牌和条件变量

当请求停止时，线程可能会因等待条件变量的通知而阻塞 (这是避免活动轮询的重要场景)。停止令牌的回调接口也支持这种情况。可以使用传递的停止令牌为条件变量调用 wait()，以便在请求停止时暂停等待。由于技术原因，必须使用 std::condition_variable_any 类型作为条件变量。

下面是一个演示在条件变量中使用停止令牌的例子:

lib/stopcv.cpp

```
include <iostream>
#include <queue>
#include <thread>
#include <stop_token>
#include <mutex>
#include <condition_variable>
using namespace std::literals; // for duration literals

int main()
{
  std::queue<std::string> messages;
  std::mutex messagesMx;
```

```
  std::condition_variable_any messagesCV;

  // start thread that prints messages that occur in the queue:
  std::jthread t1{[&] (std::stop_token st) {
      while (!st.stop_requested()) {
        std::string msg;
        {
          // wait for the next message:
          std::unique_lock lock(messagesMx);
          if (!messagesCV.wait(lock, st,
                    [&] {
                      return !messages.empty();
                    })) {
            return; // stop requested
          }
          // retrieve the next message out of the queue:
          msg = messages.front();
          messages.pop();
        }

        // print the next message:
        std::cout << "msg: " << msg << std::endl;
      }
  }};

  // store 3 messages and notify one waiting thread each time:
  for (std::string s : {"Tic", "Tac", "Toe"}) {
    std::scoped_lock lg{messagesMx};
    messages.push(s);
    messagesCV.notify_one();
  }

  // after some time
  // - store 1 message and notify all waiting threads:
  std::this_thread::sleep_for(1s);
  {
    std::scoped_lock lg{messagesMx};
    messages.push("done");
    messagesCV.notify_all();
  }

  // after some time
  // - end program (requests stop, which interrupts wait())
  std::this_thread::sleep_for(1s);
}
```

启动一个线程，循环等待消息队列不是空的，若不为空则进行输出:

```
while (!st.stop_requested()) {
```

```
  std::string msg;
  {
    // wait for the next message:
    std::unique_lock lock(messagesMx);
    if (!messagesCV.wait(lock, st,
              [&] {
                return !messages.empty();
              })) {
      return; // stop requested
    }
    // retrieve the next message out of the queue:
    msg = messages.front();
    messages.pop();
  }

  // print the next message:
  std::cout << "msg: " << msg << std::endl;
}
```

条件变量 messagesCV 的类型为 std::condition_variable_any:

```
std::condition_variable_any messagesCV;
```

用停止令牌调用 wait()，在这里传递停止令牌信号来停止线程，所以等待现在可能会结束。原因有两个:

- 有一个通知 (队列不再为空)
- 要求停止

wait() 的返回值表示是否满足条件。若返回 false，则结束 wait() 的原因是请求停止，则可以做出相应的反应 (这里，停止循环)。

表“condition_variable_any 用于停止令牌的成员函数”列出了使用锁保护 lg 的 std::condition_variable_any 类型的新成员函数。

操作	效果
cv.wait(lg, st, pred)	等待 pred 为真或 st 请求停止的通知
cv.wait_for(lg, dur, st, pred)	pred 为真或 st 请求停止时，最多等待时间为 dud
cv.wait_until(lg, tp, st, pred)	等待时间点 tp，等待 pred 为真或 st 请求停止的通知

表 12.1 condition_variable_any 成员函数用于停止令牌

目前还不支持其他阻塞函数的停止令牌。

12.2. 停止来源和停止令牌

C++20 不仅为线程提供了停止令牌。它是一种通用机制，可以异步请求停止，并使用各种方式对该请求作出响应。

其机制如下所示:

- C++20 标准库可创建一个共享的停止状态。默认情况下，停止状态不会发出信号。
- 类型为 std::stop_source 的停止源可以在其关联的共享停止状态下请求停止。
- std::stop_token 类型的停止令牌可用于，在其关联的共享停止状态下响应停止请求。可以主动轮询是否请求了停止，或者注册一个类型为 std::stop_callback 的回调，该回调将在请求停止时调用。
- 当停止请求发出，就不能撤销 (后续的停止请求无效)。
- 可以复制和移动停止源和停止令牌，允许在代码的多个位置发出停止信号或对停止做出反应。复制源代码或令牌的成本相对较低，因此按值传递可避免生命周期问题。
  然而，复制并不像传递整型值或原始指针那么简单，更像是传递一个共享指针。若经常将其传递给子函数，那么通过引用传递可能会更好。
- 该机制是线程安全的，可以在并发情况下使用。停止请求、检查请求的停止，以及对注册或取消注册的回调调用需要同步，并且当最后一个用户 (停止源、停止令牌或停止回调) 销毁时，关联的共享停止状态将自动销毁。

下面的例子展示了如何创建停止源和停止令牌:

```
#include <stop_token>
...

// create stop_source and stop_token:
std::stop_source ssrc; // creates a shared stop state
std::stop_token stok{ssrc.get_token()}; // creates a token for the stop state
```

第一步是简单地创建 stop_source 对象，该对象提供了请求停止的 API。构造函数还创建关联的共享停止状态，就可以向停止源请求 stop_token 对象，该对象提供对停止请求作出反应的 API(通过轮询或注册回调)。

然后，可以将令牌 (和/或源) 传递给位置/线程，以在可能请求停止的位置和可能对停止做出反应的位置之间建立异步通信。

没有其他方法可以创建具有关联的共享停止状态的停止令牌，停止令牌的默认构造函数没有关联的停止状态。

12.2.1 停止源和停止令牌的详情

详细的了解一下停止源、停止令牌和停止回调的 API，所有类型都在头文件 <stop_token> 中声明。

停止源的详情

“stop_source 类对象的操作”表列出了 std::stop_source 的 API。

构造函数通常在堆上为停止状态分配内存，所有停止令牌和停止回调都使用该内存，没有办法用分配器指定不同的位置。

停止源、停止令牌和停止回调之间没有生命周期约束。当使用该状态的最后一个停止源、停止令牌或停止回调销毁时，用于停止状态的内存将自动释放。

为了能够创建没有关联停止状态的停止源 (这可能很有用，因为停止状态需要资源)，可以用一个特殊的构造函数创建一个停止源，并在稍后分配一个停止源:

```
std::stop_source ssrc{std::nostopstate}; // no associated shared stop state
...
ssrc = std::stop_source{}; // assign new shared stop state
```

停止令牌的详情

“stop_token 类对象的操作”表列出了 std::stop_token 的 API。

stop_possible() 会产生是否仍然可以停止的错误信号，其会在两种情况下产生 false:

- 若没有关联的停止状态
- 若存在停止状态，但不再有停止源，并且从未请求过停止

这可以用来避免定义反应停止 (永远不会发生)。

操作	效果
stop_source s	默认构造函数; 创建具有关联停止状态的停止源
stop_source s{nostopstate}	创建没有关联停止状态的停止源
stop_source s{s2}	复制构造函数; 创建一个停止源，该停止源共享 s2 的关联停止状态
stop_source s{move(s2)}	移动构造函数; 创建一个获取 s2 的关联停止状态的停止源 (s2 不再具有关联的停止状态)
s.~stop_source()	析构函数; 若这是相关的共享停止状态的最后一个用户，则销毁
s = s2	复制赋值；复制赋值 s2 的状态，这样 s 现在也共享了 s2 的停止状态 (s 之前的停止状态都释放了)
s = move(s2)	移动赋值: 移动赋值 s2 的状态，使 s 现在共享 s2 的停止状态 (s2 不再有停止状态，释放 s 之前的停止状态)
s.get_token()	为关联的停止状态生成一个 stop_token (若没有可共享的停止状态，则返回一个没有关联的停止状态的停止令牌)
s.request_stop()	若其中任何一个尚未完成 (返回是否请求了停止)，则相关的停止状态下请求停止
s.stop_possible()	生成 s 是否具有关联的停止状态
s.stop_requested()	生成 s 是否具有请求停止的关联停止状态
s1 == s2	生成 s1 和 s2 是否共享相同的停止状态 (或者两者都不共享)
s1 != s2	生成 s2 和 s2 是否共享相同的停止状态
s1.swap(s2)	交换 s1 和 s2 的状态
swap(s1, s2)	交换 s1 和 s2 的状态

表 12.2 类 stop_source 对象的操作

12.2.2 使用停止回调

停止回调是 RAII 类型 std::stop_callback 的对象。构造函数注册一个可调用对象 (函数、函数对象或 Lambda)，以便在为指定的停止令牌请求停止时调用:

```
void task(std::stop_token st)
{
  // register temporary callback:
  std::stop_callback cb{st, []{
      std::cout << "stop requested\n";
      ...
  }};
  ...
} // unregisters callback
```

操作	效果
stop_token t	默认构造函数; 创建没有关联停止状态的停止令牌
stop_token t{t2}	复制构造函数; 创建一个停止令牌，共享 t2 的相关停止状态
stop_token t{move(t2)}	移动构造函数; 创建一个停止令牌，获取 t2 的关联停止状态 (t2 不再具有关联停止状态)
t.~stop_token()	析构函数; 若这是相关的共享停止状态的最后一个用户，则销毁
t = t2	复制赋值; 复制分配 t2 的状态，使 t 现在也共享 t2 的停止状态 (释放任何以前的 t 停止状态)
t = move(t2)	移动赋值; 移动赋值 t2 的状态，使 t 现在共享 t2 的停止状态 (t2 不再具有停止状态，并且释放 t 之前的停止状态)
t.stop_possible()	生成的 t 是否具有关联的停止状态，以及是否已经或仍可以请求停止
t.stop_requested()	生成 t 是否具有请求停止的关联停止状态
t1 == t2	输出 t1 和 t2 是否共享相同的停止状态 (或者两者都不共享)
t1 != t2	生成 t1 和 t2 是否共享相同的停止状态
t1.swap(t2)	交换 t1 和 t2 的状态
swap(t1. t2)	交换 t1 和 t2 的状态
stop_token cb{t, f}	将 cb 注册为 t 调用 f 的停止回调

表 12.3 类 stop_token 对象的操作

假设已经创建了共享停止状态，并创建了一个异步情况，其中一个线程可能请求停止，另一个线程可能运行 task()。可以用下面的代码来模拟这种情况:

```
// create stop_source with associated stop state:
std::stop_source ssrc;

// register/start task() and pass the corresponding stop token to it:
registerOrStartInBackgound(task, ssrc.get_token());
...
```

函数 registerOrStartInBackgound() 可以立即开始 task()。或启动 task() 后通过调用 std::async() 初始化一个 std::thread，称之为协程，或注册一个事件处理程序。

每当我们请求停止时:

```
ssrc.request_stop();
```

可能发生以下情况:

- 若 task() 在初始化回调时已经启动，并且仍在运行，并且还没有调用回调的析构函数，则在调用 request_stop() 的线程中立即调用已注册的可调用对象。request_stop() 阻塞，直到所有注册的可调用对象调用，且调用的顺序没有定义。
- 若 task() 在初始化回调时已经启动，并且仍在运行，并且还没有调用回调的析构函数，则在调用 request_stop() 的线程中立即调用已注册的可调用对象。请求_stop() 阻塞，直到所有注册的可调用对象被调用，且调用的顺序没有定义。
- 若 task() 已经完成 (或者至少已经调用了回调函数的析构函数)，则永远不会调用可调用对象。回调生命周期的结束表明不再需要调用可调用对象。

这些场景都会精心同步。若正在初始化一个 stop_callback，以便注册可调用对象，就会发生上述情况。若在可调用对象由于破坏了停止回调而未注册时请求停止，则同样适用。若可调用对象已经用另一个线程启动，则析构函数阻塞，直到可调用对象完成。

对于编程逻辑，则从初始化回调到其销毁结束的那一刻起，可能会调用已注册的可调用对象。直到构造函数结束，回调函数在初始化的线程中运行; 之后，在请求停止的线程中运行。请求停止的代码可能会立即调用已注册的可调用对象，也可能会稍后调用 (若回调稍后初始化)，也可能永远不会调用 (若调用回调为时已晚)。

例如，下面的代码:

lib/stop.cpp

```
#include <iostream>
#include <stop_token>
#include <future> // for std::async()
#include <thread> // for sleep_for()
#include <syncstream> // for std::osyncstream
#include <chrono>
using namespace std::literals; // for duration literals

auto syncOut(std::ostream& strm = std::cout) {
  return std::osyncstream{strm};
}

void task(std::stop_token st, int num)
{
  auto id = std::this_thread::get_id();
  syncOut() << "call task(" << num << ")\n";

  // register a first callback:
  std::stop_callback cb1{st, [num, id]{
```

```
    syncOut() << "- STOP1 requested in task(" << num
      << (id == std::this_thread::get_id() ? ")\n"
                          : ") in main thread\n");
  }};
  std::this_thread::sleep_for(9ms);

  // register a second callback:
  std::stop_callback cb2{st, [num, id]{
    syncOut() << "- STOP2 requested in task(" << num
      << (id == std::this_thread::get_id() ? ")\n"
                          : ") in main thread\n");
  }};
  std::this_thread::sleep_for(2ms);
}

int main()
{
  // create stop_source and stop_token:
  std::stop_source ssrc;
  std::stop_token stok{ssrc.get_token()};

  // register callback:
  std::stop_callback cb{stok, []{
      syncOut() << "- STOP requested in main()\n" << std::flush;
  }};

  // in the background call task() a bunch of times:
  auto fut = std::async([stok] {
              for (int num = 1; num < 10; ++num) {
                task(stok, num);
              }
            });

  // after a while, request stop:
  std::this_thread::sleep_for(120ms);
  ssrc.request_stop();
}
```

使用同步输出流来确保不同线程的 print 语句逐行同步。

例如，输出可能如下所示:

```
call task(1)
call task(2)
...
call task(7)
call task(8)
- STOP2 requested in task(8) in main thread
- STOP1 requested in task(8) in main thread
- STOP requested in main()
```

```
call task(9)
- STOP1 requested in task(9)
- STOP2 requested in task(9)
```

或者也可能是这样的:

```
call task(1)
call task(2)
call task(3)
call task(4)
- STOP2 requested in task(4) in main thread
call task(5)
- STOP requested in main()
- STOP1 requested in task(5)
- STOP2 requested in task(5)
call task(6)
- STOP1 requested in task(6)
- STOP2 requested in task(6)
call task(7)
- STOP1 requested in task(7)
- STOP2 requested in task(7)
...
```

或者像这样:

```
call task(1)
call task(2)
call task(3)
call task(4)
- STOP requested in main()
call task(5)
- STOP1 requested in task(5)
- STOP2 requested in task(5)
call task(6)
- STOP1 requested in task(6)
- STOP2 requested in task(6)
...
```

甚至可能只会这样:

```
call task(1)
call task(2)
...
call task(8)
call task(9)
- STOP requested in main()
```

若不使用 syncOut()，输出甚至可能有交错字符，因为主线程和运行 task() 的线程的输出可能完全混合在一起。

停止回调的详情

停止回调的类型，stop_callback，是一个具有非常有限 API 的类模板，只提供了一个构造函数来为停止令牌注册可调用对象，并提供了一个析构函数来取消可调用对象的注册。删除复制和移动，不提供其他成员函数。

模板形参是可调用对象的类型，通常在初始化构造函数时推导出来:

```
auto func = [] { ... };

std::stop_callback cb{myToken, func}; // deduces stop_callback<decltype(func)>
```

除了构造函数和析构函数之外，唯一的公共成员是 callback_type，存储的可调用对象的类型。

构造函数接受左值 (有名字的对象) 和右值 (临时对象或用 std::move() 标记的对象):

```
auto func = [] { ... };

std::stop_callback cb1{myToken, func}; // copies func
std::stop_callback cb2{myToken, std::move(func)}; // moves func
std::stop_callback cb3{myToken, [] { ... }}; // moves the lambda
```

12.2.3 停止令牌的限制和保证

对于可能发生在异步上下文中的几个场景，处理停止请求的特性是相当健壮的，但并不能避免所有的陷阱。

C++20 标准库保证了以下内容:

- 所有的 request_stop()、stop_requested() 和 stop_possible() 调用都会同步。
- 保证自动执行回调注册。若另一个线程并发调用 request_stop()，则当前线程将看到请求停止并立即调用当前线程上的回调，或者另一个线程将看到回调注册并在从 request_stop() 返回之前调用回调。
- stop_callback 的可调用对象保证在 stop_callback 的析构函数返回后不会调用。
- 若回调函数的析构函数刚刚在另一个线程调用，则等待可调用对象完成 (不等待其他可调用对象完成)。

然而，请注意以下限制:

- 回调不应该抛出异常，若对其可调用对象的调用通过异常退出，则调用 std::terminate()。
- 不要在回调的可调用对象中销毁回调，析构函数不会等待回调完成的。

12.3. std::jthread 的详情

“jthread 类对象的操作”表列出了 std::thread 的 API。列 Diff 记录了 Mod(修改) 了行为或与 std::thread 相比是 New 的成员函数。

操作	效果	Diff
jthread t	默认构造函数; 创建一个不可汇入的线程对象	
jthread t{f,...}	创建一个表示新线程的对象，该线程调用 f(带有附加参数) 或抛出 std::system_error	
jthread t{rv}	移动构造函数; 创建一个新的线程对象，该对象获取 rv 的状态，并使 rv 不可连接	
t.∼jthread()	析构函数; 若对象是可连接的，调用 request_stop() 和 join()	Mod
t = rv	移动赋值; 将 rv 的状态移动赋值给 t(若 t 是可连接的，则调用 request_stop() 和 join()))	Mod
t.joinable()	若有关联的线程 (可连接) 则返回 true	
t.join()	等待相关线程完成并使对象不可汇入 (若线程不可汇入则抛出 std::system_error)	
t.detach()	线程继续运行时释放 t 与其线程的关联，并使对象不可汇入 (若线程不可连接则抛出 std::system_error)	
t.request_stop()	关联的停止令牌上请求停止	New
t.get_stop_source()	从请求停止生成一个对象来	New
t.get_stop_token()	生成一个对象来检查请求的停止	New
t.get_id()	若可汇入，则返回成员类型 ID 的唯一线程 ID; 若不可汇入，则返回默认构造 ID	
t.native_handle()	返回一个平台特定的成员类型 native_handle_type，用于不可移植的线程处理	
t1.swap(t2)	交换 t1 和 t2 的状态	
swap(t1, t2)	交换 t1 和 t2 的状态	
hardware_concurrency()	带有可能的硬件线程提示的静态函数	

表 12.4 jthread 类对象的操作

std::thread 和 std::jthread 的成员类型 id 和 native_handle_type 相同，所以使用 decltype(mythread)::id 或 std::thread::id 都没有关系，直接用 std::jthread 替换现有代码中的 std::thread 就好了。

12.3.1 对 std::jthread 使用停止令牌

除了析构函数连接之外，std::jthread 的主要优点是会自动建立停止信号的机制。为此，启动线程的构造函数创建一个停止源，将其存储为线程对象的成员，并将相应的停止令牌传递给被调用的函数，以避免该函数将额外的 stop_token 作为第一个参数。

也可以通过线程的成员函数获得停止源和停止令牌:

```
std::jthread t1{[] (std::stop_token st) {
            ...
        }};
...
foo(t1.get_token()); // pass stop token to foo()
...
std::stop_source ssrc{t1.get_stop_source()};
ssrc.request_stop(); // request stop on stop token of t1
```

因为 get_token() 和 get_stop_source() 按值返回，所以停止源和停止令牌甚至可以在线程分离时，在后台运行使用。

std::jthread 集合中使用停止令牌

若启动多个 jthread，每个线程都有自己的停止令牌，这可能会导致停止所有线程的时间可能比预期的要长:

```
{
  std::vector<std::jthread> threads;
  for (int i = 0; i < numThreads; ++i) {
    pool.push_back(std::jthread{[&] (std::stop_token st) {
        while (!st.stop_requested()) {
          ...
        }
    }});
  }
  ...
} // destructor stops all threads
```

循环结束时，析构函数停止所有正在运行的线程:

```
for (auto& t : threads) {
  t.request_stop();
  t.join();
}
```

则总是等待一个线程结束，然后向下一个线程发出停止信号。

这样的代码可以为所有线程调用 join() 之前，请求所有线程停止 (通过析构函数) 进行改进。

```
{
  std::vector<std::jthread> threads;
  for (int i = 0; i < numThreads; ++i) {
    pool.push_back(std::jthread{[&] (std::stop_token st) {
        while (!st.stop_requested()) {
          ...
```

```
        }
    }});
  }
  ...
  // BETTER: request stops for all threads before we start to join them:
  for (auto& t : threads) {
    t.request_stop();
  }
} // destructor stops all threads
```

首先请求所有线程停止，线程可能在析构函数调用 join() 让线程结束之前就结束了。协程线程池的析构函数也有使用这种技术。

对多个 std::jthread 使用相同的停止令牌

可能还需要使用相同的停止令牌为多个线程请求停止，只需自己创建停止令牌，或者从已经启动的第一个线程中获取停止令牌，并将此停止令牌作为第一个参数启动 (其他) 线程。例如:

```
// initialize a common stop token for all threads:
std::stop_source allStopSource;
std::stop_token allStopToken{allStopSource.get_token()};
for (int i = 0; i < 9; ++i) {
  threads.push_back(std::jthread{[] (std::stop_token st) {
      ...
      while (!st.stop_requested()) {
        ...
      }
    },
    allStopToken // pass token to this thread
  });
}
```

可调用对象通常只接受传递的所有参数。仅当存在未传递参数的附加停止令牌参数时，才使用已启动线程的内部停止令牌。

请参阅 lib/atomicref.cpp 获得完整的示例。

12.4. 附注

作为对 std::thread 的补丁，线程应该汇入的请求最初是由 Herb Sutter 在http://wg21.link/n3630中提出。最终接受的提案是由 Nicolai Josuttis, Lewis Baker, Billy O' Neal, Herb Sutter 和 Anthony Williams 在http://wg21.link/p0660r10上制定。

第 13 章 并发特性

上一章介绍了 std::jthread 和停止令牌之后，本章将介绍 C++20 引入的所有其他并发特性:

- 锁存器和栅栏
- 计数和二值信号量
- 原子类型的各种扩展
- 同步输出流

13.1. 使用锁存器和栅栏的线程同步

两个新的类型为多线程同步异步计算/处理提供了新的机制:

- 锁存器可以进行一次性同步，线程可以等待多个任务完成。
- 栅栏对多个线程进行重复同步，当其都完成当前/下一个处理时，必须进行响应。

13.1.1 锁存器

锁存器是用于并发执行的一种新的同步机制，支持单次使用异步倒计时。从初始整数值开始，各种线程可以自动将该值计数到零。当计数器达到零时，等待此倒计时的所有线程继续运行。

考虑下面的例子:

lib/latch.cpp

```
#include <iostream>
#include <array>
#include <thread>
#include <latch>
using namespace std::literals; // for duration literals

void loopOver(char c) {
  // loop over printing the char c:
  for (int j = 0; j < c/2; ++j) {
    std::cout.put(c).flush();
    std::this_thread::sleep_for(100ms);
  }
}

int main()
{
  std::array tags{'.', '?', '8', '+', '-'}; // tags we have to perform a task for

  // initialize latch to react when all tasks are done:
  std::latch allDone{tags.size()}; // initialize countdown with number of tasks

  // start two threads dealing with every second tag:
  std::jthread t1{[tags, &allDone] {
      for (unsigned i = 0; i < tags.size(); i += 2) { // even indexes
```

```
        loopOver(tags[i]);
        // signal that the task is done:
        allDone.count_down(); // atomically decrement counter of latch
      }
      ...
  }};
  std::jthread t2{[tags, &allDone] {
      for (unsigned i = 1; i < tags.size(); i += 2) { // odd indexes
        loopOver(tags[i]);
        // signal that the task is done:
        allDone.count_down(); // atomically decrement counter of latch
      }
      ...
  }};
  ...
  // wait until all tasks are done:
  std::cout << "\nwaiting until all tasks are done\n";
  allDone.wait(); // wait until counter of latch is zero
  std::cout << "\nall tasks done\n"; // note: threads might still run
  ...
}
```

本例中，启动两个线程 (使用 std::jthread) 来执行两个任务，每个任务处理数组标记的一个字符，所以标签的大小决定了任务的数量。主线程阻塞，直到所有任务完成。:

- 用标签/任务的数量初始化锁存器:

```
std::latch allDone{tags.size()};
```

- 每个任务完成后减少计数器:

```
allDone.count_down();
```

- 主线程等待直到所有任务完成 (计数器为零):

```
allDone.wait();
```

程序的输出可能如下所示:

```
?
waiting until all tasks are done
.??..??.?..?.??.?..??.?..?.??..?.??.?.?.?.?.8??88??88??8?8?8+8+88++8+8+8+88++
8+8+8+8+88++8+8+88++8+8-+--------------------
all tasks done
```

主线程不应该假设任务完成后，所有线程都完成了。线程可能仍然会做其他事情，系统可能仍然会清理它们。要等到所有线程都完成，必须为两个线程调用 join()。

还可以使用锁存器在特定点同步多个线程，然后继续，但每个锁存器只能执行一次此操作。这样做的一个应用是确保 (尽可能地) 多个线程一起启动它们的实际工作，即使启动和初始化线程可能需要一些时间。

看下下面的例子:

lib/latch.cpp

```cpp
#include <iostream>
#include <array>
#include <vector>
#include <thread>
#include <latch>
using namespace std::literals; // for duration literals

int main()
{
  std::size_t numThreads = 10;
  // initialize latch to start the threads when all of them have been initialized:

  std::latch allReady = 10; // initialize countdown with number of threads

  // start numThreads threads:
  std::vector<std::jthread> threads;
  for (int i = 0; i < numThreads; ++i) {
    std::jthread t{[i, &allReady] {
        // initialize each thread (simulate to take some time):
        std::this_thread::sleep_for(100ms * i);
        ...
        // synchronize threads so that all start together here:
        allReady.arrive_and_wait();
        // perform whatever the thread does
        // (loop printing its index):
        for (int j = 0; j < i + 5; ++j) {
          std::cout.put(static_cast<char>('0' + i)).flush();
          std::this_thread::sleep_for(50ms);
        }
    }};
    threads.push_back(std::move(t));
  }
  ...
}
```

启动 numThreads 线程 (使用 std::jthread)，这些线程需要一些时间来初始化和启动。为了启动它们的功能，这里使用一个锁存器来进行阻塞，直到所有启动的线程都初始化并启动为止:

- 用线程数量初始化锁存器:

```cpp
std::latch allReady{numThreads};
```

- 每个线程递减计数器，直到所有线程初始化完成:

```cpp
allReady.arrive_and_wait(); // count_down() and wait()
```

程序的输出可能如下所示:

```
867534210986753420199014253768867352419078635249107683524919425386794538769458769578697867898999
```

可以看到，10 个线程 (每个线程都打印其索引) 或多或少是一起启动的。

若没有锁存器，输出可能如下所示:

```
001010210213213241324352435243654635476357468547685479685796857965879687698769879879879898999
```

这里，早启动的线程已经在运行，而之后的可能还没有启动。

锁存器的详情

类 std::latch 在头文件 <latch> 中声明，“latch 类对象的操作”表列出了 std::latch 的 API。

操作	效果
latch l{counter}	创建一个锁存器，将计数器作为倒计时的起始值
l.count_down()	自动减少计数器 (若不为 0)
l.count_down(val)	计数器按 val 自动递减
l.wait()	阻塞直到锁存器的计数器为 0
l.try_wait()	生成锁存器的计数器是否为 0
l.arrive_and_wait()	调用 count_down() 和 wait()
l.arrive_and wait(val)	调用 count_down(val) 和 wait()
max()	产生 counter 最大可能值的静态函数

表 13.1 类 latch 对象的操作

注意，不能复制或移动 (分配) 锁存器。

将容器的大小 (std::array 除外) 作为计数器的初始值是错误的。构造函数接受一个带符号的 std::ptrdiff_t，会得到以下结果:

```
1 std::latch l1{10}; // OK
2 std::latch l2{10u}; // warnings may occur
3 std::vector<int> coll{ ... };
4 ...
5 std::latch l3{coll.size()}; // ERROR
6 std::latch l4 = coll.size(); // ERROR
7 std::latch l5(coll.size()); // OK (no narrowing checked)
8 std::latch l6{int(coll.size())}; // OK
9 std::latch l7{ssize(coll)}; // OK (see std::ssize())
```

13.1.2 栅栏

栅栏是用于并发执行的新的同步机制，允许多次同步多个异步任务。设置初始计数后，多个线程可以对其进行计数，并等待计数器达到零。与锁存器相比，当达到零时，将调用一个 (可选的) 回调，计数器将重新初始化为初始计数。

当多个线程一起重复计算/执行某些事情时，栅栏是有用的。当所有线程都完成了它们的任务时，可选回调可以处理结果或新状态，再异步计算/处理可以继续进行下一轮。

例如，重复使用多个线程来计算多个值的平方根:

lib/barrier.cpp

```
#include <iostream>
#include <format>
#include <vector>
#include <thread>
#include <cmath>
#include <barrier>
int main()
{
  // initialize and print a collection of floating-point values:
  std::vector values{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0};

  // define a lambda function that prints all values
  // - NOTE: has to be noexcept to be used as barrier callback
  auto printValues = [&values] () noexcept{
            for (auto val : values) {
              std::cout << std::format(" {:<7.5}", val);
            }
            std::cout << '\n';
          };
  // print initial values:
  printValues();

  // initialize a barrier that prints the values when all threads have done their
↪  computations:
  std::barrier allDone{int(values.size()), // initial value of the counter
             printValues}; // callback to call whenever the counter is 0

  // initialize a thread for each value to compute its square root in a loop:
  std::vector<std::jthread> threads;
  for (std::size_t idx = 0; idx < values.size(); ++idx) {
    threads.push_back(std::jthread{[idx, &values, &allDone] {
        // repeatedly:
        for (int i = 0; i < 5; ++i) {
          // compute square root:
          values[idx] = std::sqrt(values[idx]);
          // and synchronize with other threads to print values:
          allDone.arrive_and_wait();
        }
    }});
  }
  ...
}
```

声明了一个浮点值数组之后，定义一个函数来输出它们 (使用 std::format() 来输出格式化的输

出):

```
// initialize and print a collection of floating-point values:
std::vector values{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0};

// define a lambda function that prints all values
// - NOTE: has to be noexcept to be used as barrier callback
auto printValues = [&values] () noexcept{
  for (auto val : values) {
    std::cout << std::format(" {:<7.5}", val);
  }
  std::cout << '\n';
};
```

回调必须用 noexcept 声明。

我们的目标是使用多个线程，以便每个线程处理一个值，所以会重复计算这些值的平方根。因此:

- 初始化栅栏 allDone，以便在所有线程完成下一个计算时打印所有值: 标签/任务的数量:

```
std::barrier allDone{int(values.size()), // initial value of the counter
           printValues}; // callback to call whenever the counter is 0
```

构造函数应该接受带符号整数值。否则，代码可能无法编译。

- 循环中，每个线程在完成计算后减少计数器，并等待计数器为零 (即所有其他线程也已经发出信号，表示已经完成):

```
..
allDone.arrive_and_wait();
```

- 当计数器达到零时，将调用回调函数来输出结果。
  回调是由最终将计数器减为零的线程调用的，所以在循环中，回调由不同的线程调用。

程序的输出可能如下所示:

1	2	3	4	5	6	7	8
1	1.4142	1.7321	2	2.2361	2.4495	2.6458	2.8284
1	1.1892	1.3161	1.4142	1.4953	1.5651	1.6266	1.6818
1	1.0905	1.1472	1.1892	1.2228	1.251	1.2754	1.2968
1	1.0443	1.0711	1.0905	1.1058	1.1185	1.1293	1.1388
1	1.0219	1.0349	1.0443	1.0516	1.0576	1.0627	1.0671

栅栏的 API 还提供了从该机制中删除线程的函数，例如：为了避免死锁，在循环运行时发出停止信号。

启动的线程的代码可以这样写:

```
// initialize a thread for each value to compute its square root in a loop:
std::vector<std::jthread> threads;
```

```
for (std::size_t idx = 0; idx < values.size(); ++idx) {
  threads.push_back(std::jthread{[idx, &values, &allDone] (std::stop_token st) {
                    // repeatedly:
                    while (!st.stop_requested()) {
                      // compute square root:
                      values[idx] = std::sqrt(values[idx]);
                      // and synchronize with other threads to print values:
                      allDone.arrive_and_wait();
                    }
                    // drop thread from barrier so that other threads not wait:
                    allDone.arrive_and_drop();
                }});
}
```

每个线程调用的 Lambda 现在接受一个停止令牌，以便在请求停止时做出反应 (显式地或通过调用线程的析构函数)。若主线程通知线程停止计算 (例如，通过调用 threads.clear())，重要的是每个线程会将自己从栅栏中删除:

```
allDone.arrive_and_drop();
```

该调用对计数器进行计数，并确保该值的下一个初始值递减。下一轮 (其他线程可能仍在运行) 中，栅栏将不再等待丢弃的线程。

参见 lib/barrierstop.cpp 获得完整的示例。

栅栏的详情

类 std::barrier 在头文件 <barrier> 中声明。“类 barrier 对象的操作”表列出了 std::barrier 的 API。

操作	效果
barrier b{num}	为 num 异步任务创建栅栏
barrier b{num, cb}	为 num 异步任务和 cb 作为回调创建栅栏
b.arrive()	将任务标记为已完成，并产生到达令牌
b.arrive(val)	将所有任务标记为已完成，并产生到达令牌
b.wait(arrivalToken)	阻塞，直到所有的任务都已经完成，并回调已调用 (若有的话)
b.arrive_and_wait()	将一个任务标记为完成并阻塞，直到所有任务都完成，并回调已调用 (若有的话)
b.arrive_and_drop()	将一个任务标记为已完成，并减少重复执行的任务数量
max()	生成 num 的最大可能值的静态函数

表 13.2 类 barrier 对象的操作

std::barrier<> 是一个类模板，其回调类型作为模板参数。通常，类型是通过类模板参数推导出来的:

```
void callback() noexcept; // forward declaration
...
std::barrier b{6, callback}; // deduces std::barrier<decltype(callback)>
```

C++ 标准要求栅栏的回调保证不会抛出异常。为了便于移植，必须用 noexcept 声明函数或 Lambda。若没有传递回调，则使用特定于实现的类型，表示没有效果的操作。

```
l.arrive_and_wait();
```

等价于

```
l.wait(l.arrive());
```

也就是说，arrive() 函数会返回一个类型为 std::barrier::arrival_token 的到达令牌，以确保 barrier 知道要等待哪个线程。否则，将无法正确处理 arrive_and_drop()。

注意，不能复制或移动 (分配) 栅栏。

将容器的大小 (std::array 除外) 作为计数器的初始值是错误的。构造函数接受一个带符号的 std::ptrdiff_t，会看到以下行为:

```
std::barrier b1{10, cb}; // OK
std::barrier b2{10u, cb}; // warnings may occur

std::vector<int> coll{ ... };
...
std::barrier b3{coll.size(), cb}; // ERROR
std::barrier b4(coll.size(), cb); // OK (no narrowing checked)
std::barrier b5{int(coll.size()), cb}; // OK
std::barrier b6{std::ssize(coll), cb}; // OK (see std::ssize())
```

函数 std::ssize() 是在 C++20 中加入的。

13.2. 信号量

C++20 引入了处理信号量的新类型。信号量是轻量级同步原语，允许同步或限制对一个或一组资源的访问。

可以像使用互斥锁一样使用，这样做的好处是，授予对资源访问权限的线程，不必是获得对资源访问权限的线程。还可以使用它们来限制资源的可用性，例如：启用和禁用线程池中线程的使用。

C++ 标准库提供了两种信号量类型:

- std::counting_semaphore<> 将多个资源的使用限制在最大值
- std::binary_semaphore<> 限制对单一资源的使用

13.2.1 使用计数信号量

下面的代码演示了信号量的常规操作方式:

lib/semaphore.cpp

```
#include <iostream>
#include <queue>
#include <chrono>
#include <thread>
#include <mutex>
#include <semaphore>
using namespace std::literals; // for duration literals

int main()
{
  std::queue<char> values; // queue of values
  std::mutex valuesMx; // mutex to protect access to the queue

  // initialize a queue with multiple sequences from 'a' to 'z':
  // - no mutex because no other thread is running yet
  for (int i = 0; i < 1000; ++i) {
    values.push(static_cast<char>('a' + (i % ('z'-'a'))));
  }

  // create a pool of numThreads threads:
  // - limit their availability with a semaphore (initially none available):
  constexpr int numThreads = 10;
  std::counting_semaphore<numThreads> enabled{0};

  // create and start all threads of the pool:
  std::vector<std::jthread> pool;
  for (int idx = 0; idx < numThreads; ++idx) {
    std::jthread t{[idx, &enabled, &values, &valuesMx] (std::stop_token st) {
        while (!st.stop_requested()) {
          // request thread to become one of the enabled threads:
          enabled.acquire();
          // get next value from the queue:
          char val;
          {
            std::lock_guard lg{valuesMx};
            val = values.front();
            values.pop();
          }
          // print the value 10 times:
          for (int i = 0; i < 10; ++i) {
            std::cout.put(val).flush();
            auto dur = 130ms * ((idx % 3) + 1);
            std::this_thread::sleep_for(dur);
          }
```

```
            // remove thread from the set of enabled threads:
            enabled.release();
            }
          }};
      pool.push_back(std::move(t));
    }

    std::cout << "== wait 2 seconds (no thread enabled)\n" << std::flush;
    std::this_thread::sleep_for(2s);

    // enable 3 concurrent threads:
    std::cout << "== enable 3 parallel threads\n" << std::flush;
    enabled.release(3);
    std::this_thread::sleep_for(2s);

    // enable 2 more concurrent threads:
    std::cout << "\n== enable 2 more parallel threads\n" << std::flush;
    enabled.release(2);
    std::this_thread::sleep_for(2s);

    // Normally we would run forever, but let's end the program here:
    std::cout << "\n== stop processing\n" << std::flush;
    for (auto& t : pool) {
    t.request_stop();
    }
}
```

代码中，启动了 10 个线程，但限制了允许其中多少线程主动运行和处理数据，因此将信号量初始化为最大数量 (10) 和初始资源数量 (0):

```
constexpr int numThreads = 10;
std::counting_semaphore<numThreads> enabled{0};
```

读者们可能想知道，为什么必须指定最大值作为模板参数。原因是，有了这个编译时值，库可以决定切换到最有效的实现 (本机支持可能只能达到某个值，或者若最大值为 1，我们可以使用简化版本)。

每个线程中，使用信号量来请求运行许可。可尝试“获取”一个可用资源来启动任务，并在任务完成后“释放”该资源以供其他使用:

```
std::jthread{[&, idx] (std::stop_token st) {
        while (!st.stop_requested()) {
          // request thread to become one of the enabled threads:
          enabled.acquire();
          ...
          // remove thread from the set of enabled threads:
          enabled.release();
        }
    }}
```

因为信号量初始化为零，所以最初的情况是阻塞，因此没有可用的资源。

可以使用信号量来允许三个线程进行并发操作:

```
// enable 3 concurrent threads:
enabled.release(3);
```

之后，允许两个线程并发运行:

```
// enable 2 more concurrent threads:
enabled.release(2);
```

若不能获得资源，就休眠或做其他事情，可以使用 try_acquire():

```
std::jthread{[&, idx] (std::stop_token st) {
        while (!st.stop_requested()) {
          // request thread to become one of the enabled threads:
          if (enabled.try_acquire()) {
            ...
            // remove thread from the set of enabled threads:
            enabled.release();
          }
          else {
            ...
          }
        }
    }}
```

还可以使用 try_acquire_for() 或 try_acquire_until() 尝试在有限的时间内获取资源:

```
std::jthread{[&, idx] (std::stop_token st) {
    while (!st.stop_requested()) {
      // request thread to become one of the enabled threads:
      if (enabled.try_acquire_for(100ms)) {
        ...
        // remove thread from the set of enabled threads:
        enabled.release();
        }
      }
    }}
```

可不时地再次检查停止令牌的状态。

线程调度不公平

注意，线程不一定是公平调度的。因此，当主线程想要结束程序时，可能需要一些时间 (甚至永远) 来调度。调用线程的析构函数之前请求所有线程停止也是一种很好的方法；否则，当前面的线程完成后，会在稍后请求停止。

线程没有公平调度的另一个后果是，不能保证等待时间最长的线程是首选。通常情况正好相反: 若线程调度器已经有一个正在运行的线程调用 release()，并且立即使用 acquire()，则调度器会保持线程运行 (“太好了，不需要上下文切换”)，所以无法保证 acquire() 会唤醒等待的多个线程中的哪一个，可能总是使用相同的线程 (伪唤醒)。

因此，在请求运行权限之前，不应该从队列中取出下一个值。

```
// BAD if order of processing matters:
{
  std::lock_guard lg{valuesMx};
  val = values.front();
  values.pop();
}
enabled.acquire();
...
```

可能是在读取其他几个值之后才处理值 val，或者甚至从未处理过。在读取下一个值之前请求许可:

```
enabled.acquire();
{
  std::lock_guard lg{valuesMx};
  val = values.front();
  values.pop();
}
...
```

出于同样的原因，不能轻易地减少已启用线程的数量。可以试着调用:

```
// reduce the number of enabled concurrent threads by one:
enabled.acquire();
```

但不知道这个报表是什么时候处理的。由于线程没有得到公平的调度，减少启用的线程数量的反应可能会花费很长时间，甚至可能会饿死 (饥饿)。

为了公平地处理队列和对资源限制的即时响应，应该使用原子的新 wait() 和通知机制。

13.2.2 使用二值信号量的例子

对于信号量，定义了一个特殊类型 std::binary_semaphore，这只是 std::counting_semaphore<1> 的快捷方式，因此它只能启用或禁用单个资源的使用。

还可以将其用作互斥锁，这样做的好处是释放资源的线程不必是以前获取资源的线程，但更常用的应用程序是一种从另一个线程发出信号/通知线程的机制。与条件变量不同，可以多次执行此操作。

考虑下面的例子:

lib/semaphorenotify.cpp

```cpp
#include <iostream>
#include <chrono>
#include <thread>
#include <semaphore>
using namespace std::literals; // for duration literals

int main()
{
  int sharedData;
  std::binary_semaphore dataReady{0}; // signal there is data to process
  std::binary_semaphore dataDone{0}; // signal processing is done

  // start threads to read and process values by value:
  std::jthread process{[&] (std::stop_token st) {
            while(!st.stop_requested()) {
              // wait until the next value is ready:
              // - timeout after 1s to check stop_token
              if (dataReady.try_acquire_for(1s)) {
                int data = sharedData;

                // process it:
                std::cout << "[process] read " << data << std::endl;
                std::this_thread::sleep_for(data * 0.5s);
                std::cout << "[process] done" << std::endl;

                // signal processing done:
                dataDone.release();
              }
              else {
                std::cout << "[process] timeout" << std::endl;
              }
              }
            }};

  // generate a couple of values:
  for (int i = 0; i < 10; ++i) {
    // store next value:
    std::cout << "[main] store " << i << std::endl;
    sharedData = i;

    // signal to start processing:
    dataReady.release();

    // wait until processing is done:
    dataDone.acquire();
    std::cout << "[main] processing done\n" << std::endl;
  }
  // end of loop signals stop
```

```
49 }
```

使用两个二值信号量让一个线程通知另一个线程:

- 使用 dataReady，主线程通知线程进程 sharedData 中有新数据需要处理。
- 使用 dataDone，处理线程通知主线程数据已处理完成。

两个信号量都初始化为零，因此在默认情况下，获取线程阻塞。发出通知的线程调用 release() 的那一刻，获取消息的线程解除阻塞，以便程序做出反应。

程序的输出如下所示:

```
[main] store 0
[process] read 0
[process]      done
[main] processing done

[main] store 1
[process] read 1
[process]      done
[main] processing done

[main] store 2
[process] read 2
[process]      done
[main] processing done

[main] store 3
[process] read 3
[process]      done
[main] processing done
...

[main] store 9
[process] read 9
[process]      done
[main] processing done

[process] timeout
```

处理线程只能使用 try_acquire_for() 获取有限的时间，返回值表示是否通知了它 (访问资源)。这允许线程不时地检查是否发出了停止信号 (就像主线程结束后那样)。

二值信号量的另一个应用是，等待在不同线程中运行的协程结束。

信号量的详情

类模板 std::counting_semaphore<> 与 std::binary_semaphore<1> 的快捷方式 std::binary_semaphore 也声明在头文件 <semaphore> 中:

```
namespace std {
  template<ptrdiff_t least_max_value = implementation-defined >
  class counting_semaphore;
  using binary_semaphore = counting_semaphore<1>;
}
```

“类 counting_semaphore<> 和 binary_semaphore 对象的操作”表列出了信号量的 API。

注意，不能复制或移动 (分配) 信号量。

将容器的大小 (std::array 除外) 作为计数器的初始值是错误的。构造函数接受一个带符号的 std::ptrdiff_t，会看到以下结果:

```
std::counting_semaphore s1{10}; // OK
std::counting_semaphore s2{10u}; // warnings may occur

std::vector<int> coll{ ... };
...
std::counting_semaphore s3{coll.size()}; // ERROR
std::counting_semaphore s4 = coll.size(); // ERROR
std::counting_semaphore s4(coll.size()); // OK (no narrowing checked)
std::counting_semaphore s6{int(coll.size())}; // OK
std::counting_semaphore s7{std::ssize(coll)}; // OK (see std::ssize())
```

操作	效果
semaphore s{num}	创建一个用 num 初始化计数器的信号量
s.acquire()	阻塞，直到可以自动减少计数器 (请求更多的资源)。
s.try_acquire()	尝试立即自动减少计数器 (请求更多资源)，若成功则返回 true
s.try_acquire_for(dur)	尝试在时间段内自动减少计数器 (请求更多资源)，若成功则返回 true
s.tre_acquire_until(tp)	尝试直到时间点 tp 自动减少计数器 (请求更多资源)，若成功则返回 true
s.release()	自动增加计数器 (多启用一个资源)
s.release(num)	自动将 num 添加到计数器 (启用更多资源)
max()	产生计数器最大可能值的静态函数

表 13.3 类 counting_semaphore<> 和 binary_semaphore 对象的操作

13.3. 原子类型的扩展

C++20 引入了两个新的原子类型 (处理引用和共享指针) 和原子类型的新特性。另外，现在有了一个类型 std::atomic<char8_t>。

13.3.1 原子引用 std::atomic_ref<>

C++11 起，标准库提供了类模板 std::atomic<> 来为普通可复制类型提供原子 API。

C++20 现在引入了类模板 std::atomic_ref<>，为普通的可复制引用类型提供原子 API，允许为通常不是原子的现有对象提供临时原子 API。一种应用程序是初始化对象而不关心并发性，然后在不同的线程中使用。

之所以给这个类型指定的名称为 atomic_ref(而不只是为引用类型提供 std::atomic<>)，是因为用户应该看到这个对象可能提供非原子访问，而且比 std::atomic<> 的保障更弱。

使用原子引用

下面的程序演示了如何使用原子引用:

lib/atomicref.cpp

```
#include <iostream>
#include <array>
#include <algorithm> // for std::fill_n()
#include <vector>
#include <format>
#include <random>
#include <thread>
#include <atomic> // for std::atomic_ref<>
using namespace std::literals; // for duration literals

int main()
{
  // create and initialize an array of integers with the value 100:
  std::array<int, 1000> values;
  std::fill_n(values.begin(), values.size(), 100);

  // initialize a common stop token for all threads:
  std::stop_source allStopSource;
  std::stop_token allStopToken{allStopSource.get_token()};

  // start multiple threads concurrently decrementing the value:
  std::vector<std::jthread> threads;
  for (int i = 0; i < 9; ++i) {
    threads.push_back(std::jthread{
      [&values] (std::stop_token st) {
        // initialize random engine to generate an index:
        std::mt19937 eng{std::random_device{}()};
        std::uniform_int_distribution distr{0, int(values.size()-1)};

        while (!st.stop_requested()) {
          // compute the next index:
          int idx = distr(eng);

          // enable atomic access to the value with the index:
          std::atomic_ref val{values[idx]};

          // and use it:
```

```
          --val;
          if (val <= 0) {
            std::cout << std::format("index {} is zero\n", idx);
          }
        }
      },
      allStopToken // pass the common stop token
    });
  }

  // after a while/event request to stop all threads:
  std::this_thread::sleep_for(0.5s);
  std::cout << "\nSTOP\n";
  allStopSource.request_stop();
  ...
}
```

首先创建并初始化包含 1000 个整数值的数组，不使用原子:

```
std::array<int, 1000> values;
std::fill_n(values.begin(), values.size(), 100);
```

稍后启动多个线程并发地递减这些值。关键是，只有在这种上下文中，才将值用作原子整数，所以需要初始化 std::atomic_ref<>:

```
std::atomic_ref val{values[idx]};
```

由于类模板实参推导，不必指定所引用对象的类型。

这个初始化的效果是，当使用原子引用时，对值的访问是原子地进行的:

- –val 以原子的方式递减
- val <= 0 以原子方式加载值，将其与 0 进行比较 (表达式在读取值后使用隐式类型转换到基础类型)

可以考虑将后者实现为 val.load() <= 0，以使用原子接口。

这个程序的不同线程不使用相同的 atomic_ref<> 对象。std::atomic_ref<> 保证通过为特定对象创建的任何 atomic_ref<> 对该对象的所有并发访问都是同步的 [如何确保这一点取决于标准库的实现]。若需要互斥锁或锁，则可以使用全局哈希表，其中为包装对象的每个地址存储一个关联的锁，这与为用户定义类型实现 std::atomic<> 的方式没有什么不同。

原子引用的特性

原子引用的头文件也是 <atomic>。对于原始指针、整型和浮点型 (C++20 起) 的 std::atomic<> 的特化也在该头文件中:

```
namespace std {
  template<typename T> struct atomic_ref; // primary template

  template<typename T> struct atomic_ref<T*>; // partial specialization for pointers

  template<> struct atomic_ref<integralType>; // full specializations for integral
↪  types

  template<> struct atomic_ref<floatType>; // full specializations for floating-point
↪  types
}
```

与 std::atomic<> 相比，原子引用有以下限制:

- 不支持 volatile
- 对象是指可能需要对齐，通常需要大于对齐的基础类型。静态成员 std::atomic_ref<type>::required_alignment 提供了这种最小对齐方式。

与 std::atomic<> 相比，原子引用有以下扩展:

- 复制构造函数，用于创建对相同基础对象的另一个引用 (但提供赋值操作符仅用于赋值基础值)。
- 常量不会传播到包装对象，所以可以给 const std::atomic_ref<> 赋一个新值:

```
MyType x, y;
const std::atomic_ref cr{x};
cr = y; // OK (would not be OK for const std::atomic<>)
```

- 线程同步支持使用 wait()、notify_one() 和 notify_all()，就像现在为所有原子类型提供的那样。

其他方面，提供了与 std::atomic<> 相同的特性:

- 该类型既提供具有内存屏障的高级 API，也提供禁用其底层 API。
- 静态成员 is_always_lock_free，非静态成员函数 is_lock_free() 表示原子支持是否无锁，这可能取决于对齐方式。

原子引用的详情

原子引用提供与相应原子类型相同的 API。对于普通的可复制类型、指针类型、整型类型或浮点类型 T，C++ 标准库提供了 std::atomic_ref<T> 与 std::atomic<T> 相同的原子 API。原子引用类型也在头文件 <atomic> 中提供。

使用静态成员 is_always_lock_free() 或非静态成员函数 is_lock_free() 时，可以检查原子类型是否在内部使用锁作为原子。若没有，则有对原子操作的本机硬件支持 (这是在信号处理程序中使用原子的先决条件)。类型的检查如下所示:

```
if constexpr(std::atomic<int>::is_always_lock_free) {
  ...
```

```
}
else {
  ... // special handling if locks are used
}
```

对特定对象的检查 (是否无锁可能取决于对齐) 如下所示:

```
std::atomic_ref val{values[idx]};
if (val.is_lock_free()) {
  ...
}
else {
  ... // special handling if locks are used
}
```

对于类型 T，尽管类型 std::atomic<T> 是无锁的，std::atomic_ref<T> 可能不是无锁的。

由原子引用引用的对象可能还必须满足特定于架构的约束，例如：对象可能需要在内存中正确对齐，或者可能不允许缓存在 GPU 寄存器内存中。对于所需的对齐，有一个静态成员:

```
namespace std {
  template<typename T> struct atomic_ref {
    static constexpr size_t required_alignment;
    ...
  };
}
```

该值至少为 alignof(T)。但该值可以是，例如 2*alignof(double)，以支持 std::complex<double> 的无锁操作。

13.3.2 原子共享指针

C++11 引入了带有可选原子接口的共享指针。使用像 atomic_load()、atomic_store() 和 atomic_exchange() 这样的函数，可以并发访问共享指针引用的值。但问题是可以将这些共享指针与非原子接口一起使用，这将破坏共享指针的所有原子使用。

C++20 现在为共享指针和弱指针提供了偏特化:

- std::atomic<std::shared_ptr<T>>
- std::atomic<std::weak_ptr<T>>

以前用于共享指针的原子 API 现在已弃用。

原子共享/弱指针不提供原子原始指针提供的额外操作，提供的 API 与 std::atomic<> 为普通可复制类型 T 提供的 API 相同。

提供了 wait()、notify()_one() 和 notify_all() 的线程同步支持，还支持具有使用内存顺序参数选项的低层原子接口。

使用原子共享指针的例子

下面的示例演示了原子共享指针的用法，原子共享指针用作共享值链表的头。

lib/atomicshared.cpp

```
#include <iostream>
#include <thread>
#include <memory> // includes <atomic> now
using namespace std::literals; // for duration literals

template<typename T>
class AtomicList {
  private:
  struct Node {
    T val;
    std::shared_ptr<Node> next;
  };
  std::atomic<std::shared_ptr<Node>> head;
public:
  AtomicList() = default;

  void insert(T v) {
    auto p = std::make_shared<Node>();
    p->val = v;
    p->next = head;
    while (!head.compare_exchange_weak(p->next, p)) { // atomic update
    }
  }

  void print() const {
    std::cout << "HEAD";
    for (auto p = head.load(); p; p = p->next) { // atomic read
      std::cout << "->" << p->val;
    }
    std::cout << std::endl;
  }
};

int main()
{
  AtomicList<std::string> alist;

  // populate list with elements from 10 threads:
  {
    std::vector<std::jthread> threads;
    for (int i = 0; i < 100; ++i) {
      threads.push_back(std::jthread{[&, i]{
          for (auto s : {"hi", "hey", "ho", "last"}) {
```

```
            alist.insert(std::to_string(i) + s);
            std::this_thread::sleep_for(5ns);
          }
      }});
    }
  } // wait for all threads to finish

  alist.print(); // print resulting list
  }
```

该程序可能有以下输出:

```
HEAD->94last->94ho->76last->68last->57last->57ho->60last->72last-> ... ->1hey->1hi
```

和通常的原子操作一样，也可以这样写:

```
for (auto p = head.load(); p; p = p->next)
```

而非：

```
for (auto p = head.load(); p.load(); p = p->next)
```

使用原子弱指针的例子

下面的例子演示了原子弱指针的用法:

lib/atomicweak.cpp

```
#include <iostream>
#include <thread>
#include <memory> // includes <atomic> now
using namespace std::literals; // for duration literals

int main()
{
  std::atomic<std::weak_ptr<int>> pShared; // pointer to current shared value (if one
↪  exists)

  // loop to set shared value for some time:
  std::atomic<bool> done{false};
  std::jthread updates{[&] {
              for (int i = 0; i < 10; ++i) {
                {
                  auto sp = std::make_shared<int>(i);
                  pShared.store(sp); // atomic update
                  std::this_thread::sleep_for(0.1s);
                }
```

```
                std::this_thread::sleep_for(0.1s);
              }
              done.store(true);
          }};

  // loop to print shared value (if any):
  while (!done.load()) {
    if (auto sp = pShared.load().lock()) { // atomic read
      std::cout << "shared: " << *sp << '\n';
    }
    else {
      std::cout << "shared: <no data>\n";
    }
    std::this_thread::sleep_for(0.07s);
  }
}
```

这里，不必使共享指针原子化，因为它只可让一个线程使用。唯一的问题是两个线程并发地更新或使用弱指针。

该程序可能有以下输出:

```
shared: <no data>
shared: 0
shared: <no data>
shared: 1
shared: <no data>
shared: <no data>
shared: 2
shared: <no data>
shared: <no data>
shared: 3
shared: <no data>
shared: 4
shared: 4
shared: <no data>
shared: 5
shared: 5
shared: <no data>
shared: 6
shared: <no data>
shared: <no data>
shared: 7
shared: <no data>
shared: 8
shared: 8
shared: <no data>
shared: 9
shared: 9
```

```
shared: <no data>
```

和通常的原子操作一样，也可以这样写:

```
pShared = sp;
```

而非:

```
pShared.store(sp);
```

13.3.3 浮点原子类型

std::atomic<> 和 std::atomic_ref<> 现在都为 float、double 和 long double 类型提供了完整的特化。

与任意可复制类型的主模板相比，其提供了额外的原子操作来添加和减去值 [与整型的特化相反，整型的特化还提供了对自增/自减值的原子支持，并可执行按位修改]:

- fetch_add(), fetch_sub()
- operator+=, operator-=

因此，现在可以做到:

```
std::atomic<double> d{0};
...
d += 10.3; // OK since C++20
```

13.3.4 使用原子类型的线程同步

所有原子类型 (std::atomic<>、std::atomic_ref<> 和 std::atomic_flag) 现在都提供了一个简单的 API，让线程阻塞并等待其他线程对其值的更改。

对于原子值:

```
std::atomic<int> aVal{100};
```

或者原子引用:

```
int value = 100;
std::atomic_ref<int> aVal{value};
```

可以定义想要等待的值，直到引用值已经改变:

```
int lastValue = aVal.load();
aVal.wait(lastValue); // block unless/until value changed (and notified)
```

若引用对象的值与传递的参数不匹配，则立即返回。否则，会阻塞，直到为原子值或引用调用了 notify_one() 或 notify_all():

```
--aVal; // atomically modify the (referenced) value
aVal.notify_all(); // notify all threads waiting for a change
```

但对于条件变量，wait() 可能会由于伪唤醒而结束 (因此没有调用通知)，所以开发者需要在 wait() 之后再次检查该值。

等待特定原子值:

```
while ((int val = aVal.load()) != expectedVal) {
  aVal.wait(val);
  // here, aVal may or may not have changed
}
```

不能保证会将获得所有更新:

lib/atomicwait.cpp

```
#include <iostream>
#include <thread>
#include <atomic>
using namespace std::literals;
int main()
{
  std::atomic<int> aVal{0};
  // reader:
  std::jthread tRead{[&] {
            int lastX = aVal.load();
            while (lastX >= 0) {
              aVal.wait(lastX);
              std::cout << "=> x changed to " << lastX << std::endl;
              lastX = aVal.load();
            }
            std::cout << "READER DONE" << std::endl;
        }};

  // writer:
  std::jthread tWrite{[&] {
            for (int newVal : { 17, 34, 3, 42, -1}) {
              std::this_thread::sleep_for(5ns);
              aVal = newVal;
              aVal.notify_all();
            }
        }};
  ...
}
```

输出可能是:

```
=> x changed to 17
=> x changed to 34
=> x changed to 3
=> x changed to 42
=> x changed to -1
READER DONE
```

或:

```
=> x changed to 17
=> x changed to 3
=> x changed to -1
READER DONE
```

或者是:

```
READER DONE
```

通知函数是 const 成员函数。

使用原子通知的公平票务系统

使用原子 wait() 和通知的一个应用是像使用互斥锁一样使用，因为使用互斥锁的成本可能会高得多。

下面是一个例子，使用原子来实现队列中值的公平处理 (与使用信号量的不公平版本相比)。虽然可能有多个线程等待，但只有有限数量的线程可以运行。通过使用一个售票系统，确保队列中的元素按顺序处理:[这个例子的想法是基于 Bryce Adelstein Lelbach 在 CppCon 2029 上的演讲《C++20 同步库》中的一个例子 (参见http://youtu.be/Zcqwb3CWqs4?t=1810)。]

lib/atomicticket.cpp

```
#include <iostream>
#include <queue>
#include <chrono>
#include <thread>
#include <atomic>
#include <semaphore>
using namespace std::literals; // for duration literals
int main()
{
  char actChar = 'a'; // character value iterating endless from 'a' to 'z'
  std::mutex actCharMx; // mutex to access actChar
  // limit the availability of threads with a ticket system:
  std::atomic<int> maxTicket{0}; // maximum requested ticket no
  std::atomic<int> actTicket{0}; // current allowed ticket no
  // create and start a pool of numThreads threads:
```

```
constexpr int numThreads = 10;
std::vector<std::jthread> threads;
for (int idx = 0; idx < numThreads; ++idx) {
  threads.push_back(std::jthread{[&, idx] (std::stop_token st) {
                    while (!st.stop_requested()) {
                      // get next character value:
                      char val;
                      {
                        std::lock_guard lg{actCharMx};
                        val = actChar++;
                        if (actChar > 'z') actChar = 'a';
                        }

                        // request a ticket to process it and wait until enabled:
                        int myTicket{++maxTicket};
                        int act = actTicket.load();
                        while (act < myTicket) {
                        actTicket.wait(act);
                        act = actTicket.load();
                        }

                        // print the character value 10 times:
                        for (int i = 0; i < 10; ++i) {
                        std::cout.put(val).flush();
                        auto dur = 20ms * ((idx % 3) + 1);
                        std::this_thread::sleep_for(dur);
                        }

                        // done, so enable next ticket:
                        ++actTicket;
                        actTicket.notify_all();
                      }
                    }});
}

// enable and disable threads in the thread pool:
auto adjust = [&, oldNum = 0] (int newNum) mutable {
        actTicket += newNum - oldNum; // enable/disable tickets
        if (newNum > 0) actTicket.notify_all(); // wake up waiting threads
        oldNum = newNum;
        };

for (int num : {0, 3, 5, 2, 0, 1}) {
  std::cout << "\n====== enable " << num << " threads" << std::endl;
  adjust(num);
  std::this_thread::sleep_for(2s);
}

```

```
  for (auto& t : threads) { // request all threads to stop (join done when leaving
  ↪ scope)
    t.request_stop();
  }
}
```

每个线程请求下一个字符值，请求一个票据来处理这个值:

```
int myTicket{++maxTicket};
```

然后线程等待，直到票据启用:

```
int act = actTicket.load();
while (act < myTicket) {
  actTicket.wait(act);
  act = actTicket.load();
}
```

每次线程唤醒时，都会仔细检查其票据是否启用 (actTicket 至少是 myTicket)。通过增加 actTicket 的值并通知所有等待线程来启用新票据。

这发生在线程完成处理时:

```
++actTicket;
actTicket.notify_all();
```

或者当启用的票数变化时:

```
actTicket += newNum - oldNum; // enable/disable tickets
if (newNum > 0) actTicket.notify_all(); // wake up waiting threads
```

该示例可能有以下输出:

```
====== enable 0 threads

====== enable 3 threads
acbabacabacbaacbabacbacdbdbdcdbdcdecdeddcedegfgegfegegfgegfgegfegfhhfhfhhfhfhijhjjhijhji
jkjkijkjkijkkliklkilkkilmmlimlmilmmlnmlmnml
====== enable o5 threads
pmpnoqrpropnqpronpqorppnorqppornqsrosnqrsosnrqosrsntqstustvqstuvstqtvwuwtvxwtxuvwtxwtxvu
wyxwvyxuwvxwyxuvwyxzxvuyzabyuzbabyzubyabzybczabyzbcabdzbdcazdeedczadedecfafdefcedaefdedf
caefgfecghfigfichfgijhcjgijhgkijjkgihjkgjihjkgij
====== enable khg2 threads
ijkihkhkhkkllmllmlmlllmllmnnmnnmnnmnnmnmnoopoopoopoopoopqpqqpqpqpqpqqrrqrqrrsrsrrsrsrsts
tsts
====== enable 0 threads
ststtttt
====== enable 1 threads
uuuuuuuuuuvvvvvvvvvvwwwwwwwwwwxxxxxxxxxxyyyyyyyyyyzzzzzzzzzzaaaaaaaaaabbbbbbbbbbccccccccc
ccddddddddddeeeeeeeeeeffffffffffgggggggggghhhhhhhhhh
```

使用同步输出流来确保输出没有交错字符。

13.3.5 std::atomic_flag 的扩展

C++20 之前，若不设置 std::atomic_flag，就无法检查其值，因此 C++20 添加了全局函数和成员函数来检查当前值:

- atomic_flag_test(const atomic_flag*) noexcept;
- atomic_flag_test_explicit(const atomic_flag*, memory_order) noexcept;
- atomic_flag::test() const noexcept;
- atomic_flag::test(memory_order) const noexcept;

13.4. 同步输出流

C++20 提供了一种将并发输出同步到流的新机制。

13.4.1 添加同步输出流的动机

若多个线程并发地写一个流，输出通常必须同步:

- 通常，流的并发输出会导致未定义行为 (这是数据竞争，指具有未定义行为的竞争条件)。
- 支持并发输出到标准流 (如 std::cout)，但结果不是很有用，因为来自不同线程的字符可能以任意顺序混合在一起。

例如，考虑下面的程序:

lib/atomicticket.cpp

```
#include <iostream>
#include <cmath>
#include <thread>

void squareRoots(int num)
{
  for (int i = 0; i < num ; ++i) {
    std::cout << "squareroot of " << i << " is "
    << std::sqrt(i) << '\n';
  }
}

int main()
{
  std::jthread t1(squareRoots, 5);
  std::jthread t2(squareRoots, 5);
  std::jthread t3(squareRoots, 5);
}
```

三个线程并发地写 std::cout。这是有效的，因为使用了标准输出流。

然而，输出可能是这样的:

```
squareroot of squareroot of 0 is 0 is 0
0squareroot of squareroot of
01squareroot of is is 101 is

1squareroot of squareroot of
12squareroot of is is 21 is 1.41421

1.41421squareroot of squareroot of
23squareroot of is is 31.41421 is 1.73205
1.73205squareroot of squareroot of
34squareroot of is is 41.73205 is 2

2squareroot of
4 is 2
```

13.4.2 使用同步输出流

通过使用同步输出流，现在可以将多个线程的并发输出同步到同一个流，只需要使用用相应的输出流初始化的 std::osyncstream。例如:

lib/atomicticket.cpp

```
#include <iostream>
#include <cmath>
#include <thread>
#include <syncstream>

void squareRoots(int num)
{
  for (int i = 0; i < num ; ++i) {
    std::osyncstream coutSync{std::cout};
    coutSync << "squareroot of " << i << " is "
    << std::sqrt(i) << '\n';
  }
}

int main()
{
  std::jthread t1(squareRoots, 5);
  std::jthread t2(squareRoots, 5);
  std::jthread t3(squareRoots, 5);
}
```

同步输出缓冲区将输出与其他输出同步到同步输出缓冲区，以便仅在调用同步输出缓冲区的析构函数时刷新输出。

结果，输出如下所示:

```
squareroot of 0 is 0
squareroot of 0 is 0
squareroot of 1 is 1
squareroot of 0 is 0
squareroot of 1 is 1
squareroot of 2 is 1.41421
squareroot of 1 is 1
squareroot of 2 is 1.41421
squareroot of 3 is 1.73205
squareroot of 2 is 1.41421
squareroot of 3 is 1.73205
squareroot of 4 is 2
squareroot of 3 is 1.73205
squareroot of 4 is 2
squareroot of 4 is 2
```

这三个线程现在逐行写入 std::cout，但确切顺序仍然不确定。可以通过实现循环得到相同的结果:

```
for (int i = 0; i < num ; ++i) {
  std::osyncstream{std::cout} << "squareroot of " << i << " is "
                << std::sqrt(i) << '\n';
}
```

'\n'、std::endl 和 std::flush 都不会写入输出，这里需要析构函数。若在循环外创建同步输出流，则在到达析构函数时将所有线程的输出一起打印。

然而，在调用析构函数之前，有一个新的操纵符用于写入输出:std::flush_emit，所以可以创建和初始化同步输出流，并逐行输出:

```
std::osyncstream coutSync{std::cout};
for (int i = 0; i < num ; ++i) {
  coutSync << "squareroot of " << i << " is "
       << std::sqrt(i) << '\n' << std::flush_emit;
}
```

13.4.3 为文件使用同步输出流

还可以为文件使用同步输出流。考虑下面的例子:

lib/syncfilestream.cpp

```
#include <fstream>
#include <cmath>
#include <thread>
#include <syncstream>

```

```
void squareRoots(std::ostream& strm, int num)
{
  std::osyncstream syncStrm{strm};
  for (int i = 0; i < num ; ++i) {
    syncStrm << "squareroot of " << i << " is "
         << std::sqrt(i) << '\n' << std::flush_emit;
  }
}

int main()
{
  std::ofstream fs{"tmp.out"};
  std::jthread t1(squareRoots, std::ref(fs), 5);
  std::jthread t2(squareRoots, std::ref(fs), 5);
  std::jthread t3(squareRoots, std::ref(fs), 5);
}
```

该程序使用三个并发线程逐行写入同一个文件。

每个线程都使用自己的同步输出流，但都必须使用相同的文件流。因此，若每个线程都打开文件，程序将无法工作。

13.4.4 使用同步输出流作为输出流

同步输出流是一个流。类 std::osyncstream 派生自 std::ostream(准确地说: 与通常的流类一样，类 std::basic_osyncstream<> 派生自类 std::basic_ostream<>)。还可以按如下方式实现上述程序:

lib/syncfilestream2.cpp

```
#include <fstream>
#include <cmath>
#include <thread>
#include <syncstream>

void squareRoots(std::ostream& strm, int num)
{
  for (int i = 0; i < num ; ++i) {
    strm << "squareroot of " << i << " is "
       << std::sqrt(i) << '\n' << std::flush_emit;
  }
}

int main()
{
  std::ofstream fs{"tmp.out"};
  std::osyncstream syncStrm1{fs};
  std::jthread t1(squareRoots, std::ref(syncStrm1), 5);
  std::osyncstream syncStrm2{fs};
  std::jthread t2(squareRoots, std::ref(syncStrm2), 5);
```

```
  std::osyncstream syncStrm3{fs};
  std::jthread t3(squareRoots, std::ref(syncStrm3), 5);
}
```

操纵符 std::flush_emit 通常是为输出流定义的，可以在这里使用。对于未同步的输出流，没有影响。

创建一个同步输出流并将其传递给所有三个线程将无法工作，多个线程将写入一个流:

```
// undefined behavior (concurrent writes to the same stream):
std::osyncstream syncStrm{fs};
std::jthread t1(squareRoots, std::ref(syncStrm), 5);
std::jthread t2(squareRoots, std::ref(syncStrm), 5);
std::jthread t3(squareRoots, std::ref(syncStrm), 5);
```

13.4.5 实践同步输出流

C++20 起，经常使用同步输出流来“调试”带有 print 语句的多线程程序。只需要定义以下内容:

```
#include <iostream> // for std::cout
#include <syncstream> // for std::osyncstream

inline auto syncOut(std::ostream& strm = std::cout) {
  return std::osyncstream{strm};
}
```

有了这个定义，可只使用 syncOut()，而非 std::cout，来确保并发输出是逐行写入的。例如:

```
void foo(std::string name) {
  syncOut() << "calling foo(" << name
        << ") in thread " << std::this_thread::get:id() << '\n';
  ...
}
```

我们在本书中使用它来可视化并发协程输出。

为了能够关闭这样的调试输出，我有时会这样做:

```
#include <iostream>
#include <syncstream> // for std::osyncstream

constexpr bool debug = true; // switch to false to disable output

inline auto coutDebug() {
  if constexpr (debug) {
    return std::osyncstream{std::cout};
  }
```

```
  else {
    struct devnullbuf : public std::streambuf {
      int_type overflow (int_type c) { // basic output primitive
        return c; // - without any print statement
      }
    };
    static devnullbuf devnull;
    return std::ostream{&devnull};
  }
}
```

13.5. 附注

锁存器和栅栏最早是由 Alasdair Mackintosh 在http://wg21.link/n3600中提出。最终接受的提案是由 Bryce Adelstein Lelbach、Olivier Giroux、JF Bastien、Detlef Vollmann 和 David Olsen 在http://wg21.link/p1135r6上制定。

信号量最早由 Olivier Giroux 在http://wg21.link/p0514r1中提出。最终接受的提案是由 Bryce Adelstein Lelbach、Olivier Giroux、JF Bastien、Detlef Vollmann 和 David Olsen 在http://wg21.link/p1135r6上制定。

原子引用首先由 H. Carter Edwards、Hans Boehm、Olivier Giroux 和 James Reus 在http://wg21.link/p0019r0中以 std::atomic __ view<> 的形式提出。最终接受的提案是由 Daniel Sunderland、H. Carter Edwards、Hans Boehm、Olivier Giroux、Mark Hoemmen、D. Hollman、Bryce Adelstein Lelbach 和 Jens Maurer 在http://wg21.link/p0019r8上制定。wait() 和通知的 API 是由 David Olsen 在http://wg21.link/p1643r1中最终提出。

原子共享指针最早由 Herb Sutter 在http://wg21.link/n4058中以 atomic_shared_ptr<> 的形式提出，随后用于并发 TS (http://wg21.link/n4577)。最终接受的将其作为 std::atomic<> 的偏特化集成到 C++ 标准中的提案是由 Alisdair Meredith 在http://wg21.link/p0718r2中提出。

浮点类型的原子特化最早是由 H. Carter Edwards、Hans Boehm、Olivier Giroux、JF Bastien 和 James Reus 在http://wg21.link/p0020r0中提出。最终接受的提案是由 H. Carter Edwards、Hans Boehm、Olivier Giroux、JF Bastien 和 James Reus 在http://wg21.link/p0020r6上制定。

原子类型的线程同步最早是由 Olivier Giroux 在http://wg21.link/p0514r0中提出。最终接受的提案是由 Bryce Adelstein Lelbach、Olivier Giroux、JF Bastien、Detlef Vollmann 和 David Olsen 在http://wg21.link/p1135r6、http://wg21.link/p1643r1和http://wg21.link/p1644r0中提出。

同步输出流首先由 Lawrence Crowl 在http://wg21.link/n3750中提出。最终接受的提案是由 Lawrence Crowl、Peter Sommerlad、Nicolai Josuttis 和 Pablo Halpern 在http://wg21.link/p0053r7以及 Peter Sommerlad 和 Pablo Halpern 在http://wg21.link/p0753r2中共同制定。

第 14 章 协程

C++20 引入了对协程的支持。协程 (1958 年由 Mel Conway 发明) 是可以挂起的函数，本章将介绍协程的一般概念和详情。

C++20 从对协程的基本支持开始，引入了一些核心语言特性和一些处理协程的基本库特性。然而，必须编写重要的粘合代码才能处理协同程序，这使得使用它们非常灵活，但即使在简单的情况下也需要一些努力。计划是在 C++23 及以后的版本中，为 C++ 标准库中的协程的用提供更多标准类型和函数。

这一章是在 Lewis Baker, Frank Birbacher, Michael Eiler, Bjorn fahler, Dietmar Köuhl, Phil Nash 和 Charles Tolman 的大力帮助和支持下完成的。

14.1. 什么是协程?

调用普通函数 (或过程) 时，然后运行到它们的结束 (或直到到达返回语句或抛出异常)，而协程是可以分多个步骤运行的函数 (参见图 14.1)。

某些时刻，可以挂起一个协程，所以该函数暂停其计算，直到恢复。挂起可能是因为函数必须等待某些东西，有其他 (更重要的) 事情要做，或者有一个中间结果要给调用者。

因此，启动协程意味着启动另一个函数，直到它的一部分完成。调用函数和协程都在它们的两条执行路径之间来回切换。注意，这两个函数不是并行运行的，我们用控制流来打乒乓球:

- 函数可以通过开始或继续协程的语句来决定启动或恢复其当前控制流。
- 当协程运行时，协程可以决定挂起或结束其执行，启动或恢复协程的函数将继续执行其控制流。

协程的最简单形式中，主控制流和协程的控制流都在同一个线程中运行。不需要使用多线程，也不需要处理并发访问，但可以在不同的线程中运行协程。甚至可以在不同的线程上将协程恢复到先前挂起的位置。协程有一种正交特性，但其可以与多个线程一起使用。甚至可以在不同的线程上将协程恢复到先前挂起的位置。

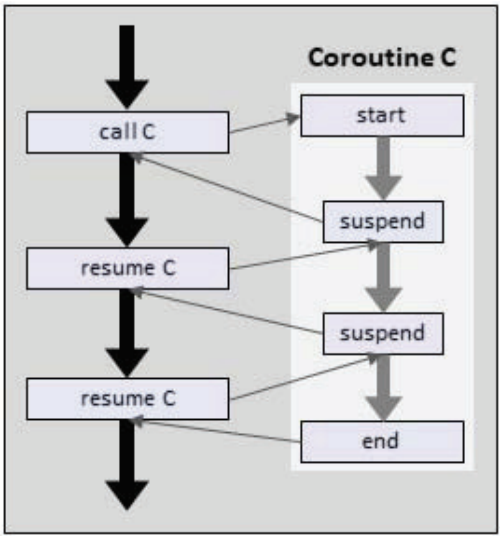

图 14.1 协程

使用协程就像在后台有一个函数，可以不时地启动和继续。然而，由于协程的生命周期超出了嵌套作用域，因此协程也是一个将其状态存储在某些内存中并提供处理状态的 API。

C++ 中有几个关于协程的基础工具:

- 只需在函数中使用以下关键字即可隐式定义协程:
  - co_await
  - co_yield
  - co_return

  若这些关键字在协程中都不没有，则必须显式地使用 co_return; 语句。
- 协程通常返回一个对象，作为调用者的协程接口。根据协程的目的和用途，该对象可以表示一个不时挂起或切换上下文的正在运行的任务，不时产生值的生成器，或者一个按需惰性地返回一个或多个值的工厂。
- 协程无堆栈。不挂起外部协程的情况下，无法挂起在外部协程中调用的内部协程，只能将外部协程作为一个整体挂起。

  当协程挂起时，协程的状态作为一个整体被存储在与堆栈分开的对象中，以便它可以在完全不同的上下文中 (在不同的调用堆栈中，在另一个线程中等) 恢复。

14.2. 第一个协程示例

这里，我们希望下面的函数是协程:

```
1  void coro(int max)
2  {
3    std::cout << "CORO " << max << " start\n";
4
5    for (int val = 1; val <= max; ++val) {
6      std::cout << "CORO " << val << '/' << max << '\n';
7    }
8
9    std::cout << "CORO " << max << " end\n";
10 }
```

该函数有一个表示最大值的形参，首先将其打印出来。然后，从 1 循环到这个最大值，并打印每个值，最后有一个 print 语句。

当用 coro(3) 调用该函数时，有以下输出:

```
CORO 3 start
CORO 1/3
CORO 2/3
CORO 3/3
CORO 3 end
```

但是，我们希望将其编程为一个协程，每次执行循环中的 print 语句时，都会挂起。因此，函数可中断，协程的用户可以通过恢复触发下一个值的输出。

14.2.1 定义协程

以下是协程定义的完整代码:

lib/atomicwait.cpp

```
#include <iostream>
#include "corotask.hpp" // for CoroTask

CoroTask coro(int max)
{
  std::cout << "              CORO " << max << " start\n";

  for (int val = 1; val <= max; ++val) {
    // print next value:
    std::cout << "           CORO " << val << '/' << max << '\n';

    co_await std::suspend_always{}; // SUSPEND
  }

  std::cout << " CORO " << max << " end\n";
}
```

仍然有一种循环遍历这些值直到参数 max 的函数，但有两点与普通函数不同:

- print 语句后的循环中，有一个 co_await 表达式，其可挂起协程并阻塞它，直到协程恢复，这称为挂起点。
  挂起调用的确切行为由紧跟在 co_await 后面的表达式定义，使开发者能够控制挂起的确切行为。
  目前，将使用 std::suspend_always 类型的默认构造对象，其接受挂起并将控制权交还给调用者。也可以通过将特殊操作数传递给 co_await，来拒绝挂起或恢复另一个协程。
- 虽然协程没有返回语句，但其有一个返回类型 CoroTask。此类型用作协程调用者的协程接口，但不能将返回类型声明为 auto。
  返回类型是必需的，因为调用者需要一个接口来处理协程 (例如恢复它)。C++20 中，协程接口类型必须由程序员 (或第三方库) 提供，稍后将看到是如何实现的。计划是，即将到来的 C++ 标准将在其库中提供一些标准的协程接口类型。

14.2.2 使用协程

可以这样使用协程:

coro/coro.cpp

```
#include <iostream>
#include "coro.hpp"

int main()
{
```

```
  // start coroutine:
  auto coroTask = coro(3); // initialize coroutine
  std::cout << "coro() started\n";

  // loop to resume the coroutine until it is done:
  while (coroTask.resume()) { // RESUME
    std::cout << "coro() suspended\n";
  }

  std::cout << "coro() done\n";
}
```

初始化协程后，产生协程接口 coroTask，启动一个循环，在协程挂起后一次再次恢复协程:

```
auto coroTask = coro(3); // initialize coroutine

while (coroTask.resume()) { // RESUME
  ...
}
```

通过调用 coro(3)，像调用函数一样调用协程。然而，与函数调用相比，不等待协程结束。相反，在协程初始化之后，调用返回协程接口来处理协程 (在开始处有一个隐式挂起点)。

在这里使用 auto 作为协程接口类型，也可以使用它的类型，这是协程的返回类型:

```
CoroTask coroTask = coro(3); // initialize coroutine
```

类 CoroTask 提供的 API 提供了一个成员函数 resume()，可用于恢复协程。每次调用 resume() 都允许协程继续运行，直到下一次挂起或直到协程结束。请注意，挂起不会在协程中留下任何作用域。在恢复时，可继续暂停状态下的协程。

其效果是，在 main() 的循环中，调用协程中的下一组语句，直到挂起点或结束:

- 首先，初始输出，val 的初始化，以及循环中的第一个输出:

```
std::cout << "                CORO " << max << " start\n";
for (int val = 1; val <= max; ... ) {
  std::cout << "                CORO " << val << '/' << max << '\n';
  ...
}
```

- 然后是两次，循环中的下一个迭代:

```
  for (...; val <= max; ++val) {
    std::cout << "                  CORO " << val << '/' << max << '\n';
    ...
  }
```

- 最后，在循环中的最后一次迭代之后，协程执行最后的 print 语句:

```
for ( ... ; val <= max; ++val) {
  ...
}

std::cout << " CORO " << max << " end\n";
```

程序输出如下所示:

```
coro() started
        CORO 3 start
        CORO 1/3
coro() suspended
        CORO 2/3
coro() suspended
        CORO 3/3
coro() suspended
        CORO 3 end
coro() done
```

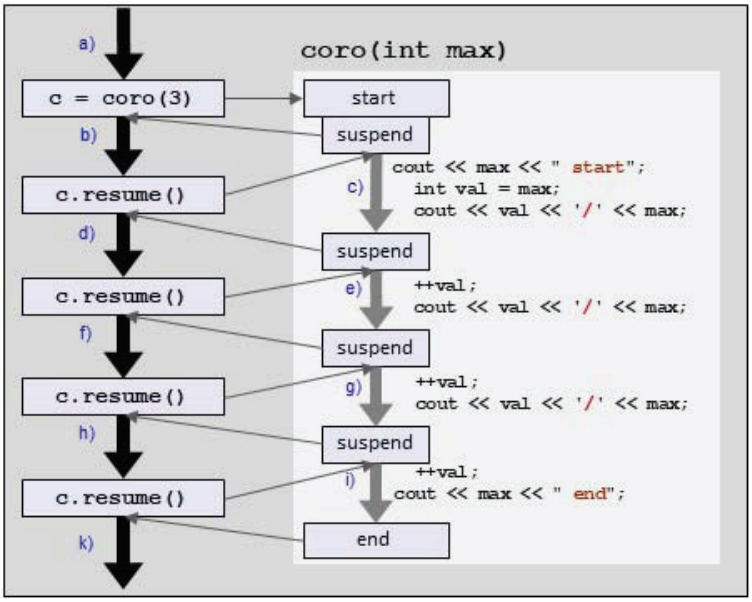

图 14.2 协同程序的例子

结合接口类型 CoroTask，我们马上就会看到，可得到以下控制流 (见图 14.2):

a) 首先，调用协程，使其启动。协程立即挂起，调用返回处理协程的接口对象。
b) 然后，可以使用接口对象来恢复协程，以便执行其语句。
c) 协程内部，处理开始语句直到第一个挂起点: 第一个 print 语句，在循环头部初始化局部计数器 val，以及 (在检查 val 小于或等于 max 之后) 循环内的 print 语句。在这部分结束时，协程暂停。
d) 挂起将控制转移回主函数，然后主函数继续执行，直到再次恢复协程。
e) 协程继续执行下一个语句，直到再次到达挂起点，增加 val 并 (在检查 val 仍然小于或等于 max 之后) 在循环中执行 print 语句。在这部分结束时，协程再次挂起。

这里继续使用之前协程挂起时 val 的值。

f) 再次将控制转移回主函数，然后主函数继续运行，直到再次恢复协程。

g) 只要增量 val 小于或等于 max，协程就继续循环。

h) 只要协程 co_await 挂起，主函数就会恢复协程。

i) 最后，协程在 val 加 1 之后离开循环，调用最后的 print 语句，值为 max。

j) 协程结束时，最后一次将控制转回主函数。然后 main 函数结束循环，并继续运行直到结束。

协程的初始化和接口的确切行为取决于接口类型 CoroTask，可能只是启动协程或提供具有一些初始化的上下文，例如打开文件或启动线程，还定义协程是立即启动还是惰性启动 (立即挂起)。在 CoroTask 的当前实现中，是惰性启动的，所以协程的初始调用还没有执行 coro()。

这里没有异步通信或控制流，coroTask.resume() 更像是 coro() 下一部分的函数调用。

当到达协程的末尾时，循环结束，resume() 的实现使得它返回协程是否 (尚未) 完成。当 resume() 返回 true 时，继续循环 (在打印暂停之后)。

多次使用协程

协程是一种准并行函数，通过来回切换控制流来顺序执行。其状态存储在由协程句柄控制的堆内存中，该句柄通常由协程接口持有。通过拥有多个协程接口对象，可以处理多个活动协程的状态，这些协程可能彼此独立地运行或挂起。

假设用不同的 max 值启动两个协程:

coro/coro2.cpp

```cpp
#include <iostream>
#include "coro.hpp"

int main()
{
  // start two coroutines:
  auto coroTask1 = coro(3); // initialize 1st coroutine
  auto coroTask2 = coro(5); // initialize 2nd coroutine
  std::cout << "coro(3) and coro(5) started\n";

  coroTask2.resume(); // RESUME 2nd coroutine once

  // loop to resume the 1st coroutine until it is done:
  while (coroTask1.resume()) { // RESUME 1st coroutine
    std::cout << "coro() suspended\n";
  }

  std::cout << "coro() done\n";

  coroTask2.resume(); // RESUME 2nd coroutine again
}
```

对于初始化的协程，可得到两个不同的接口对象，coroTask1 和 coroTask2。通过有时恢复第一个协程，有时恢复第二个协程，控制流在主函数和这两个协程之间跳转。

例子中，恢复第二个协程 (最大 5 个) 一次，然后循环恢复第一个协程，最后再次恢复第二个协程。结果，程序有以下输出:

```
coro(3) and coro(5) started
         CORO 5 start
         CORO 1/5
         CORO 3 start
         CORO 1/3
coro() suspended
         CORO 2/3
coro() suspended
         CORO 3/3
coro() suspended
         CORO 3 end
coro() done
         CORO 2/5
```

甚至可以将协程接口对象传递给不同的函数或线程那里恢复，其将永远保持目前的状态。

14.2.3 引用调用的生命周期问题

协程的生命周期通常比最初调用它的语句要长。这有一个重要的后果: 若通过引用传递临时对象，可能会遇到致命的运行时问题。

考虑以下稍微修改过的协程，现在所做的就是通过引用，取最大值:

coro/cororef.hpp

```
#include <iostream>
#include <coroutine> // for std::suspend_always{}
#include "corotask.hpp" // for CoroTask

CoroTask coro(const int& max)
{
  std::cout << "   CORO " << max << " start\n"; // OOPS: value of max still valid?

  for (int val = 1; val <= max; ++val) { // OOPS: value of max still valid?
    std::cout << " CORO " << val << '/' << max << '\n';
    co_await std::suspend_always{}; // SUSPEND
  }

  std::cout << " CORO " << max << " end\n"; // OOPS: value of max still valid?
}
```

问题是，传递一个临时对象 (甚至可能是一个文字对象) 会创建未定义行为。可能会看到它，也可能不会看到它，这取决于平台、编译器设置和其他代码。例如，考虑使用如下方式调用协程:

coro/cororef.hpp

```cpp
#include <iostream>
#include "cororef.hpp"

int main()
{
  auto coroTask = coro(3); // OOPS: creates reference to temporary/literal
  std::cout << "coro(3) started\n";
  coro(375); // another temporary coroutine
  std::cout << "coro(375) started\n";
  // loop to resume the coroutine until it is done:

  while (coroTask.resume()) { // ERROR: undefined behavior
    std::cout << "coro() suspended\n";
  }
  std::cout << "coro() done\n";
}
```

某些平台上，此代码运行良好。然而，在我测试这段代码的一个平台上，得到了以下输出:

```
coro(3) started
coro(375) started
  CORO -2147168984 start
  CORO -2147168984 end
coro() done
```

初始化协程的语句之后，max 引用所引用的传入参数 3 的位置不再可用。当第一次恢复协程时，val 的输出和初始化使用对已销毁对象的引用。

通常: **不要使用引用来声明协程参数。**

若复制参数的代价太大，可以通过使用 std::ref() 或 std::cref() 创建的引用包装器“按引用传递”。对于容器，可以使用 std::views::all()，将容器作为视图传递，因此可以使用所有标准范围函数，而无需将参数转换回来。例如:

```cpp
CoroTask printElems(auto coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << '\n';
    co_await std::suspend_always{}; // SUSPEND
  }
}
std::vector<std::string> coll;
...
// start coroutine that prints the elements:
// - use view created with std::views::all() to avoid copying the container
auto coPrintElems = printElems(std::views::all(coll));

while (coPrintElems.resume()) { // RESUME
```

```
...
}
```

14.2.4 调用协程

协程可以调用其他协程 (甚至是间接的)，调用和被调用的协程都可能有挂起点。

考虑一个具有一个挂起点的协程，以便其分两部分运行:

```
CoroTask coro()
{
  std::cout << " coro(): PART1\n";
  co_await std::suspend_always{}; // SUSPEND
  std::cout << " coro(): PART2\n";
}
```

这样使用这个协程时:

```
auto coroTask = coro(); // initialize coroutine
std::cout << "MAIN: coro() initialized\n";

while (coroTask.resume()) { // RESUME
  std::cout << "MAIN: coro() suspended\n";
}

std::cout << "MAIN: coro() done\n";
```

可得到以下输出:

```
MAIN: coro() initialized
    coro(): PART1
MAIN: coro() suspended
    coro(): PART2
MAIN: coro() done
```

现在，通过另一个协程间接调用 coro()。main() 调用 callCoro()，而非 coro():

```
auto coroTask = callCoro(); // initialize coroutine
std::cout << "MAIN: callCoro() initialized\n";

while (coroTask.resume()) { // RESUME
  std::cout << "MAIN: callCoro() suspended\n";
}

std::cout << "MAIN: callCoro() done\n";
```

有趣的部分是如何实现 callCoro()。

无内部 resume()

可以尝试通过 coro() 来实现 callCoro():

```
CoroTask callCoro()
{
  std::cout << " callCoro(): CALL coro()\n";
  coro(); // CALL sub-coroutine
  std::cout << " callCoro(): coro() done\n";
  co_await std::suspend_always{}; // SUSPEND
  std::cout << " callCoro(): END\n";
}
```

这段代码可编译。然而，并没有像预期的那样工作，正如程序的输出所示:

```
MAIN: callCoro() initialized
  callCoro(): CALL coro()
  callCoro(): coro() done
MAIN: callCoro() suspended
  callCoro(): END
MAIN: callCoro() done
```

coro() 函数体中的 print 语句永远不会调用。

原因是

```
coro();
```

只初始化协程并立即挂起它，Coro() 永远不会恢复。其无法恢复，因为返回的协程接口甚至没有使用。外部协程的 resume() 不会自动恢复任何内部协程。

为了在使用协程 (如函数) 时至少获得编译器警告，类 CoroTask 将使用 [[nodiscard]] 声明。

内部使用 resume()

必须像处理外部协程一样处理内部协程:resume() 在循环中:

```
CoroTask callCoro()
{
  std::cout << " callCoro(): CALL coro()\n";
  auto sub = coro(); // init sub-coroutine
  while (sub.resume()) { // RESUME sub-coroutine
    std::cout << " callCoro(): coro() suspended\n";
  }
  std::cout << " callCoro(): coro() done\n";
  co_await std::suspend_always{}; // SUSPEND
```

```
  std::cout << " callCoro(): END\n";
}
```

通过 callCoro() 的实现，就可以得到想要的行为和输出:

```
MAIN: callCoro() initialized
  callCoro(): CALL coro()
    coro(): PART1
  callCoro(): coro() suspended
    coro(): PART2
  callCoro(): coro() done
MAIN: callCoro() suspended
  callCoro(): END
MAIN: callCoro() done
```

请参阅 coro/corcoro.cpp 获得完整的示例。

内部使用 resume() 一次

值得注意的是，若在 callCoro() 中只恢复一次 coro()，会发生什么:

```
CoroTask callCoro()
{
  std::cout << " callCoro(): CALL coro()\n";
  auto sub = coro(); // init sub-coroutine
  sub.resume(); // RESUME sub-coroutine
  std::cout << " callCoro(): call.resume() done\n";
  co_await std::suspend_always{}; // SUSPEND
  std::cout << " callCoro(): END\n";
}
```

输出变成:

```
MAIN: callCoro() initialized
  callCoro(): CALL coro()
    coro(): PART1
  callCoro(): call.resume() done
MAIN: callCoro() suspended
  callCoro(): END
MAIN: callCoro() done
```

callCoro() 初始化 coro() 之后，coro() 只恢复一次，所以只调用它的第一部分。之后，coro() 的挂起将控制流传输回 callCoro()，然后 callCoro() 挂起自己，则控制流返回到 main()。当 main() 恢复 callCoro() 时，程序完成 callCoro()，而 Coro() 永远不会完成。

委托 resume()

可以实现 CoroTask，以便 co_await 以一种方式注册子协程，从而处理子协程中的挂起，就像处理调用协程中的挂起一样。callCoro() 看起来如下所示:

```
CoroTaskSub callCoro()
{
  std::cout << " callCoro(): CALL coro()\n";
  co_await coro(); // call sub-coroutine
  std::cout << " callCoro(): coro() done\n";
  co_await std::suspend_always{}; // SUSPEND
  std::cout << " callCoro(): END\n";
}
```

然而，resume() 随后必须将恢复请求委托给子协同程序 (若有的话)，所以 CoroTask 接口必须成为一个可等待对象。稍后，在介绍了可等待对象之后，将看到这样一个协程接口，将恢复委托给子协程。

14.2.5 实现协程接口

我已经说过几次了，在例子中，类 CoroTask 在处理协程方面起着重要的作用，是编译器和协程调用者通常处理的接口。协程接口汇集了一些要求，让编译器处理协程，并为调用者提供 API 来创建、恢复和销毁协程。

要处理 C++ 中的协程，需要做两件事:

- promise 类型
  此类型用于定义处理协同例程的某些自定义点，特定的成员函数定义了在特定情况下调用的回调函数。
- std::coroutine_handle<> 类型的内部协程句柄
  此对象在调用协程时创建 (使用上述 promise 类型的标准回调之一)，可以通过提供一个底层接口来恢复协程以及处理协程的结束，从而用于管理协程的状态。

处理协程返回类型的类型通常的目的是将这些需求结合在一起:

- 必须定义使用的 promise 类型 (通常定义为类型成员 promise_type)。
- 必须定义协程句柄存储的位置 (通常定义为数据成员)。
- 必须为调用者提供处理协程的接口 (本例中是成员函数 resume())。

协程接口 CoroTask

CoroTask 类提供 promise_type，存储协程句柄，并定义协程调用者的 API，定义如下所示:
coro/corotask.hpp

```
#include <coroutine>

// coroutine interface to deal with a simple task
// - providing resume() to resume the coroutine
```

```
class [[nodiscard]] CoroTask {
public:
  // initialize members for state and customization:
  struct promise_type; // definition later in corotaskpromise.hpp
  using CoroHdl = std::coroutine_handle<promise_type>;
  private:
  CoroHdl hdl; // native coroutine handle

public:
  // constructor and destructor:
  CoroTask(auto h)
  : hdl{h} { // store coroutine handle in interface
  }
  ~CoroTask() {
    if (hdl) {
      hdl.destroy(); // destroy coroutine handle
    }
  }
  // don't copy or move:
  CoroTask(const CoroTask&) = delete;
  CoroTask& operator=(const CoroTask&) = delete;

  // API to resume the coroutine
  // - returns whether there is still something to process
  bool resume() const {
    if (!hdl || hdl.done()) {
      return false; // nothing (more) to process
    }
    hdl.resume(); // RESUME (blocks until suspended again or the end)
    return !hdl.done();
  }
};

#include "corotaskpromise.hpp" // definition of promise_type
```

CoroTask 类中，首先定义基本类型和成员来处理协程的原生 API，编译器在协程接口类型中查找的关键成员是类型成员 promise_type。通过 promise 类型，定义了协程句柄的类型，并引入了私有成员，用于存储针对 promise 类型的协程句柄:

```
class [[nodiscard]] CoroTask {
public:
  // initialize members for state and customization:
  struct promise_type; // definition later in corotaskpromise.hpp
  using CoroHdl = std::coroutine_handle<promise_type>;
private:
  CoroHdl hdl; // native coroutine handle
  ...
};
```

引入 promise_type(每个协程类型都必须拥有)，并声明本地协程句柄 hdl，它管理协程的状态。原生协程句柄 std::coroutine_handle<> 的类型是用 promise 类型参数化的，存储在 promise 中的任何数据都是句柄的一部分，promise 中的函数可以通过句柄访问。

promise 类型必须是 public 的才能从外部可见，提供协程句柄类型的公共名称 (在本例中为 CoroHdl) 通常也很有帮助。为了简化，可以将句柄本身设为 public。这样，就可以用

```
CoroTask::CoroHdl
```

而非

```
std::coroutine_handle<CoroTask::promise_type>
```

也可以在这里直接内联地定义 promise 类型。但在本例中，将定义推迟到后来包含的 corotaskpromise.hpp。

协程接口类型的构造函数和析构函数初始化协程句柄的成员，并在协程接口销毁之前将其清除:

```
class CoroTask {
  ...
  public:
  CoroTask(auto h)
  : hdl{h} { // store coroutine handle internally
  }
  ~CoroTask() {
    if (hdl) {
      hdl.destroy(); // destroy coroutine handle (if there is one)
    }
  }
  ...
};
```

通过用 [[nodiscard]] 声明类，在创建协程但未使用时强制编译器发出警告 (当意外地将协程用作普通函数时尤其可能发生这种情况)。

简单起见，禁用了复制和移动。提供复制或移动语义是可能的，但必须小心正确地处理。

最后，为调用者定义了唯一的接口 resume():

```
class CoroTask {
  ...
  bool resume() const {
    if (!hdl || hdl.done()) {
      return false; // nothing (more) to process
    }
    hdl.resume(); // RESUME (blocks until suspended again or the end)
    return !hdl.done();
  }
};
```

关键的 API 是 resume()，在协程挂起时恢复协程，其或多或少地将恢复请求传播到原生协程句柄，其返回表示是否有必要再次恢复协程。

首先，函数检查是否有句柄，或者协程是否已经结束。

尽管在这个实现中协程接口总是有一个句柄，但这是一个必要的检查，例如，若接口支持移动语义。

只有当协程挂起且尚未结束时才允许调用 resume()，所以检查是否 done() 是必要的。调用本身恢复挂起的协程并阻塞，直到下一个挂起点或结束。

```
hdl.resume(); // RESUME (blocks until suspended again or the end)
```

也可以使用 operator():

```
hdl(); // RESUME (blocks until suspended again or the end)
```

因为 resume() 接口返回是否有必要再次恢复协程，所以返回协程是否已经结束:

```
bool resume() const {
  ...
  return !hdl.done();
}
```

成员函数 done() 由原生协程句柄提供，就是为了这个目的。

协程调用者的接口完全包装了原生协程句柄及其 API，我们决定调用者如何处理协程。可以使用不同的函数名，甚至操作符，或者分割调用以恢复和检查结束。稍后，将看到一些示例，其中迭代协程在每次挂起时交付的值，甚至可以将值发送回协程，并且可以提供一个 API 将协程置于不同的上下文中。

最后，包含了一个头文件，里面有 promise 类型的定义:

```
#include "corotaskpromise.hpp" // definition of promise_type
```

通常在接口类的声明中完成，或者至少在同一个头文件中完成。通过这种方式，可以将这个示例的细节拆分到不同的文件中。

promise_type 的实现

唯一缺少的部分是 promise 类型的定义。其目的是:

- 定义如何创建或获取协程的返回值 (通常包括创建协程句柄)
- 决定协同程序是应该在开始还是结束时挂起
- 处理协程调用者与协程之间交换的值
- 处理未处理的异常

下面是 CoroTask 的 promise 类型，及其协程句柄类型 CoroHdl 的常规基本实现:

coro/corotaskpromise.hpp

```cpp
struct CoroTask::promise_type {
  auto get_return_object() { // init and return the coroutine interface
    return CoroTask{CoroHdl::from_promise(*this)};
  }
  auto initial_suspend() { // initial suspend point
    return std::suspend_always{}; // - suspend immediately
  }
  void unhandled_exception() { // deal with exceptions
    std::terminate(); // - terminate the program
  }
  void return_void() { // deal with the end or co_return;
  }
  auto final_suspend() noexcept { // final suspend point
    return std::suspend_always{}; // - suspend immediately
  }
};
```

(必须) 定义以下成员 (使用它们的常规使用顺序):

- 调用 get_return_object() 来初始化协程接口。创建对象，该对象稍后返回给协程的调用者，其实现通常是这样的:
  - 首先，它为调用此函数的 promise 创建原生协程句柄:

    ```cpp
    coroHdl = CoroHdl::from_promise(*this)
    ```

    调用该成员函数的 promise 是在启动协程时自动创建的。

    from_promise() 是类模板 std::coroutine_handle<> 为此目的提供的静态成员函数。
  - 然后，创建协程接口对象，用刚刚创建的句柄初始化它:

    ```cpp
    coroIf = CoroTask{coroHdl}
    ```

  - 最后，返回接口对象:

    ```cpp
    return coroIf
    ```

  实现在一个语句中完成所有这些:

  ```cpp
  auto get_return_object() {
    return CoroTask{CoroHdl::from_promise(*this)};
  }
  ```

  也可以在这里返回协程句柄，而无需从中显式地创建协程接口:

  ```cpp
  auto get_return_object() {
    return CoroHdl::from_promise(*this);
  }
  ```

  在内部，返回的协程句柄然后通过使用初始化协程接口来自动转换。但不建议使用这种方法，因为若 CoroTask 的构造函数是显式的，并且在创建接口时不清楚，则这种方法不起作用。
- initial_suspend() 允许额外的初始准备，并定义协程是主动启动还是惰性启动:

  - 返回 std::suspend_never{} 表示立即启动，协程在用第一个语句初始化之后立即启动。
  - 返回 std::suspend_always{} 表示惰性启动。协程立即挂起，不执行任何语句，与恢复一起处理。

  例子中，要求立即暂停。

- return_void() 定义到达结束时的反应 (或 co_return; 声明)。若声明了这个成员函数，协程应该永远不会返回值。若协程产生或返回数据，则必须使用另一个成员函数。
- unhandled_exception() 定义了如何处理协程中未本地处理的异常。这里，指定这会导致程序的异常终止。稍后将讨论处理异常的其他方法。
- final_suspend() 定义是否应该最终挂起协程。指定要这样做，通常是正确的做法。但这个成员函数必须保证不抛出异常，应该返回 std::suspend_always{}。

这些 promise 类型成员的目的和使用将在后面详细说明。

14.2.6 引导接口、句柄和 promise

来概括一下处理协例程需要做些什么:

- 对于每个协程，都有一个 promise，在协程调用时自动创建。
- 协程状态存储在协程句柄中，其的类型是 std::coroutine_handle<PrmType>。该类型提供了恢复协程的原生 API(并检查是否处于结束状态或销毁其内存)。
- 协程接口是将所有内容组合在一起的场所。保存并管理本机协程句柄，并由协程调用返回，且提供成员函数来处理协程。

有多种方法可以声明 promise 类型和协程句柄 (以及两者的类型)。没有很好的方法可以做到这一点，因为协程句柄的类型需要 promise 类型，而 promise 类型的定义使用协程句柄。

实践中，通常可以这样做:

- 声明 promise 类型，声明协程句柄的类型，并定义 promise 类型:

```
class CoroTask {
public:
  struct promise_type; // promise type
  using CoroHdl = std::coroutine_handle<promise_type>;
private:
  CoroHdl hdl; // native coroutine handle
public:
  struct promise_type {
    auto get_return_object() {
      return CoroTask{CoroHdl::from_promise(*this)};
    }
    ...
  };
  ...
};
```

- 定义 promise 类型并声明协程句柄:

```cpp
class CoroTask {
public:
  struct promise_type { // promise type
    auto get_return_object() {
      return std::coroutine_handle<promise_type>::from_promise(*this);
    }
    ...
  };
private:
  std::coroutine_handle<promise_type> hdl; // native coroutine handle
  public:
  ...
};
```

- 外部定义 promise 类型为泛型辅助类型:

```cpp
template<typename CoroIf>
struct CoroPromise {
  auto get_return_object() {
    return std::coroutine_handle<CoroPromise<CoroIf>>::from_promise(*this);
  }
  ...
};

class CoroTask {
  public:
  using promise_type = CoroPromise<CoroTask>;
  private:
  std::coroutine_handle<promise_type> hdl; // native coroutine handle
  public:
  ...
};
```

因为 promise 类型通常是特定于接口的 (有不同或额外的成员)，通常使用以下简化的形式:

```cpp
class CoroTask {
public:
  struct promise_type;
  using CoroHdl = std::coroutine_handle<promise_type>;
private:
  CoroHdl hdl; // native coroutine handle
public:
  struct promise_type {
    auto get_return_object() { return CoroHdl::from_promise(*this); }
    auto initial_suspend() { return std::suspend_always{}; }
    void return_void() { }
    void unhandled_exception() { std::terminate(); }
    auto final_suspend() noexcept { return std::suspend_always{}; }
    ...
```

```
  };
  ...
};
```

请注意，目前为止描述的所有内容都是处理协程的常规方式。协程库设计了更多的灵活性。例如:

- 可以在容器或调度器中存储或管理协同程序接口。
- 甚至可以完全跳过协程接口，这是一个很少使用的选项。

14.2.7 内存管理

协程具有在不同上下文中使用的状态，因此协程句柄通常将协程的状态存储在堆内存中。堆内存分配可以优化，也可以进行更改。

调用 destroy()

为了使协程处理的成本更低，没有智能处理此内存的方法。协程句柄在初始化时只指向内存，直到调用 destroy()，所以当协程接口销毁时，应该显式调用 destroy():

```
class CoroTask {
  ...
  CoroHdl hdl; // native coroutine handle

  public:
  ~CoroTask() {
    if (hdl) {
      hdl.destroy(); // destroy coroutine handle (if there is one)
    }
  }
  ...
};
```

复制和移动协程

协程句柄的低成本/原生的实现也使得有必要处理复制和移动。默认情况下，复制协程接口将复制协程句柄，这将产生两个协程接口/句柄共享同一个协程的效果。当一个协程句柄将协程带入另一个句柄不知道的状态时，会带来风险。移动协程对象具有相同的效果。默认情况下，指针是通过移动复制的。

为了减少这种危险，应该小心地为协程提供复制或移动语义。最简单的方法是禁用复制和移动:

```
class CoroTask {
  ...
  // don't copy or move:
```

```
  CoroTask(const CoroTask&) = delete;
  CoroTask& operator=(const CoroTask&) = delete;
  ...
};
```

然而，不能移动协程 (例如将它们存储在容器中)，所以支持移动语义可能有意义，但应该确保已移动的对象不再引用协程，并且获得新值的协程会破坏旧值:

```
class CoroTask {
  ...
  // support move semantics:
  CoroTask(CoroTask&& c) noexcept
  : hdl{std::move(c.hdl)} {
    c.hdl = nullptr;
  }
  CoroTask& operator=(CoroTask&& c) noexcept {
    if (this != &c) { // if no self-assignment
      if (hdl) {
        hdl.destroy(); // - destroy old handle (if there is one)
      }
      hdl = std::move(c.hdl); // - move handle
      c.hdl = nullptr; // - moved-from object has no handle anymore
    }
    return *this;
  }
  ...
};
```

严格地说，这里的句柄不需要 std::move()，但它不会造成伤害，并提醒开发者将移动语义委托给成员。

14.3. 产生或返回值的协程

介绍了使用 co_await 的示例之后，还应该介绍协程的其他两个关键字:

- co_yield 允许协程在每次挂起时产生一个值。
- co_return 允许协程在其结束时返回一个值。

14.3.1 使用 co_yield

通过使用 co_yield，协程可以在挂起时产生中间结果。

一个明显的例子是“生成”值的协程。作为示例的一个变体，可以循环一些值直到最大值，并将其输出给协程的调用者，而不是仅仅输出。因此，协程看起来如下所示:

coro/coyield.hpp

```
#include <iostream>
#include "corogen.hpp" // for CoroGen
```

```
3
4   CoroGen coro(int max)
5   {
6     std::cout << "             CORO " << max << " start\n";
7
8     for (int val = 1; val <= max; ++val) {
9       // print next value:
10      std::cout << "               CORO " << val << '/' << max << '\n';
11
12      // yield next value:
13      co_yield val; // SUSPEND with value
14    }
15    std::cout << "             CORO " << max << " end\n";
16  }
```

通过使用 co_yield，协程产生中间结果。当协程到达 co_yield 时，会挂起协程，并提供 co_yield 后面的表达式的值:

```
1   co_yield val; // calls yield_value(val) on promise
```

协程帧将此映射到对 yield_value() 的调用，用于协程的承诺，可以定义如何处理此中间结果。我们的例子中，将值存储在 promise 的成员中，这使得它可以在协程接口中使用:

```
1   struct promise_type {
2     int coroValue = 0; // last value from co_yield
3
4     auto yield_value(int val) { // reaction to co_yield
5     coroValue = val; // - store value locally
6     return std::suspend_always{}; // - suspend coroutine
7     }
8     ...
9   };
```

将值存储在 promise 中之后，返回 std::suspend_always{}，这实际上是挂起协程。可以在这里编写不同的行为，以便协同程序 (有条件地) 继续。

协程可以这样使用:

coro/coyield.hpp

```
1   #include "coyield.hpp"
2   #include <iostream>
3
4   int main()
5   {
6     // start coroutine:
7     auto coroGen = coro(3); // initialize coroutine
8     std::cout << "coro() started\n";
9
```

```
  // loop to resume the coroutine until it is done:
  while (coroGen.resume()) { // RESUME
    auto val = coroGen.getValue();
    std::cout << "coro() suspended with " << val << '\n';
  }

  std::cout << "coro() done\n";
}
```

通过调用 coro(3)，初始化协程计数到 3，。每当我们为返回的协程接口调用 resume() 时，协程就会“计算”并产生下一个值。

resume() 并不返回产生的值 (仍然返回是否有必要再次恢复协程)，要访问下一个值，需要使用 getValue()，所以程序有如下输出:

```
coro() started
         CORO 3 start
         CORO 1/3
coro() suspended with 1
         CORO 2/3
coro() suspended with 2
         CORO 3/3
coro() suspended with 3
         CORO 3 end
coro() done
```

协程接口必须处理产生的值，并为调用者提供稍微不同的 API，所以使用不同的类型名称:CoroGen。这个名字表明，没有执行任务，而是有一个协程，在每次挂起时生成值。CoroGen 类型可以定义如下:

coro/corogen.hpp

```
#include <coroutine>
#include <exception> // for terminate()

class [[nodiscard]] CoroGen {
public:
  // initialize members for state and customization:
  struct promise_type;
  using CoroHdl = std::coroutine_handle<promise_type>;
private:
  CoroHdl hdl; // native coroutine handle
public:
  struct promise_type {
    int coroValue = 0; // recent value from co_yield

    auto yield_value(int val) { // reaction to co_yield
      coroValue = val; // - store value locally
      return std::suspend_always{}; // - suspend coroutine
```

```
    }

    // the usual members:
    auto get_return_object() { return CoroHdl::from_promise(*this); }
    auto initial_suspend() { return std::suspend_always{}; }
    void return_void() { }
    void unhandled_exception() { std::terminate(); }
    auto final_suspend() noexcept { return std::suspend_always{}; }
  };

  // constructors and destructor:
  CoroGen(auto h) : hdl{h} { }
  ~CoroGen() { if (hdl) hdl.destroy(); }

  // no copying or moving:
  CoroGen(const CoroGen&) = delete;
  CoroGen& operator=(const CoroGen&) = delete;

  // API:
  // - resume the coroutine:
  bool resume() const {
    if (!hdl || hdl.done()) {
      return false; // nothing (more) to process
      }
      hdl.resume(); // RESUME
      return !hdl.done();
    }

  // - yield value from co_yield:
  int getValue() const {
    return hdl.promise().coroValue;
  }
};
```

通常，这里使用的协程接口的定义遵循协程接口的一般原则。有两点不同:

- promise 提供了成员 yield_value()，在到达 co_yield 时调用。
- 对于协程接口的外部 API，CoroGen 提供了 getValue()，其返回存储在 promise 中的最后一个生成值:

```
class CoroGen {
  public:
  ...
  int getValue() const {
    return hdl.promise().coroValue;
  }
};
```

迭代协程产生的值

也可以使用协程接口，像范围一样使用 (提供 API 来迭代挂起产生的值):

coro/cororange.cpp

```
#include "cororange.hpp"
#include <iostream>
#include <vector>

int main()
{
  auto gen = coro(3); // initialize coroutine
  std::cout << "--- coro() started\n";

  // loop to resume the coroutine for the next value:
  for (const auto& val : gen) {
    std::cout << " val: " << val << '\n';
  }

  std::cout << "--- coro() done\n";
}
```

只需要协程返回一个稍微不同的生成器 (参见 coro/cororange.hpp):

```
Generator<int> coro(int max)
{
  std::cout << "CORO " << max << " start\n";
  ...
}
```

为传递的类型 (本例中为 int) 的生成器值使用了泛型协程接口，一个非常简单的实现 (它显示了这一点，但有一些缺陷) 可能如下所示:

coro/generator.hpp

```
#include <coroutine>
#include <exception> // for terminate()
#include <cassert> // for assert()

template<typename T>
class [[nodiscard]] Generator {
  public:
  // customization points:
  struct promise_type {
    T coroValue{}; // last value from co_yield

    auto yield_value(T val) { // reaction to co_yield
      coroValue = val; // - store value locally
      return std::suspend_always{}; // - suspend coroutine
```

```
15      }
16
17      auto get_return_object() {
18        return std::coroutine_handle<promise_type>::from_promise(*this);
19      }
20      auto initial_suspend() { return std::suspend_always{}; }
21      void return_void() { }
22      void unhandled_exception() { std::terminate(); }
23      auto final_suspend() noexcept { return std::suspend_always{}; }
24    };
25  private:
26    std::coroutine_handle<promise_type> hdl; // native coroutine handle
27
28  public:
29
30    // constructors and destructor:
31    Generator(auto h) : hdl{h} { }
32    ~Generator() { if (hdl) hdl.destroy(); }
33
34    // no copy or move supported:
35    Generator(const Generator&) = delete;
36    Generator& operator=(const Generator&) = delete;
37
38    // API to resume the coroutine and access its values:
39    // - iterator interface with begin() and end()
40    struct iterator {
41      std::coroutine_handle<promise_type> hdl; // nullptr on end
42      iterator(auto p) : hdl{p} {
43      }
44      void getNext() {
45        if (hdl) {
46          hdl.resume(); // RESUME
47          if (hdl.done()) {
48            hdl = nullptr;
49          }
50        }
51      }
52      int operator*() const {
53        assert(hdl != nullptr);
54        return hdl.promise().coroValue;
55      }
56      iterator operator++() {
57        getNext(); // resume for next value
58        return *this;
59      }
60      bool operator== (const iterator& i) const = default;
61    };
62
63    iterator begin() const {
```

```
    if (!hdl || hdl.done()) {
      return iterator{nullptr};
    }
    iterator itor{hdl}; // initialize iterator
    itor.getNext(); // resume for first value
    return itor;
  }

  iterator end() const {
    return iterator{nullptr};
  }
};
```

关键的方法是协程接口提供了成员 begin() 和 end()，以及迭代器来迭代值:

- begin() 在恢复第一个值后生成迭代器。迭代器在内部存储协程句柄，以便知道协程的状态。
- operator++() 迭代器会返回下一个值。
- end() 生成一个表示范围结束的状态，hdl 是 null。这也是迭代器在没有更多值时的状态。
- operator==() 默认通过比较两个迭代器的句柄来比较状态。

C++23 可能会在其标准库中提供这样一个协程接口，如 std::generator<>，其具有更加复杂和健壮的 API(参见http://wg21.link/p2502)。

14.3.2 使用 co_return

通过使用 co_return，协程可以在其结束时向调用者返回结果。

考虑下面的例子:

coro/coreturn.cpp

```
#include <iostream>
#include <vector>
#include <ranges>
#include <coroutine> // for std::suspend_always{}
#include "resulttask.hpp" // for ResultTask

ResultTask<double> average(auto coll)
{
  double sum = 0;
  for (const auto& elem : coll) {
    std::cout << " process " << elem << '\n';
    sum = sum + elem;
    co_await std::suspend_always{}; // SUSPEND
  }
  co_return sum / std::ranges::ssize(coll); // return resulting average
}

int main()
{
```

```
  std::vector values{0, 8, 15, 47, 11, 42};
  // start coroutine:
  auto task = average(std::views::all(values));

  // loop to resume the coroutine until all values have been processed:
  std::cout << "resume()\n";
  while (task.resume()) { // RESUME
    std::cout << "resume() again\n";
  }

  // print return value of coroutine:
  std::cout << "result: " << task.getResult() << '\n';
}
```

main() 启动协程 average()，遍历传递的集合中的元素，并将它们的值添加到初始和中。处理完每个元素后，协程挂起。

最后，协例程通过将总和除以元素数量返回平均值。

注意，需要 co_return 来返回协程的结果，不允许使用 return。

协程接口是在 ResultTask 类中定义的，该类是根据返回值的类型参数化的。这个接口提供 resume()，以便在协程挂起时恢复，所以还提供了 getResult() 来请求协程完成后的返回值。

协同程序接口 ResultTask<> 如下所示:

coro/resulttask.hpp

```
#include <coroutine>
#include <exception> // for terminate()
template<typename T>
class [[nodiscard]] ResultTask {
public:
  // customization points:
  struct promise_type {
    T result{}; // value from co_return

    void return_value(const auto& value) { // reaction to co_return
      result = value; // - store value locally
    }

    auto get_return_object() {
      return std::coroutine_handle<promise_type>::from_promise(*this);
    }

    auto initial_suspend() { return std::suspend_always{}; }
    void unhandled_exception() { std::terminate(); }
    auto final_suspend() noexcept { return std::suspend_always{}; }
  };

private:
  std::coroutine_handle<promise_type> hdl; // native coroutine handle
```

```
25
26  public:
27    // constructors and destructor:
28    // - no copying or moving is supported
29    ResultTask(auto h) : hdl{h} { }
30    ~ResultTask() { if (hdl) hdl.destroy(); }
31    ResultTask(const ResultTask&) = delete;
32    ResultTask& operator=(const ResultTask&) = delete;
33
34    // API:
35    // - resume() to resume the coroutine
36    bool resume() const {
37      if (!hdl || hdl.done()) {
38        return false; // nothing (more) to process
39      }
40      hdl.resume(); // RESUME
41      return !hdl.done();
42    }
43
44    // - getResult() to get the last value from co_yield
45    T getResult() const {
46      return hdl.promise().result;
47    }
48  };
```

同样，该定义遵循协程接口的一般原则。

但这次支持返回值，所以在 promise 类型中，不再提供自定义点 return_void()。相反，提供了 return_value()，当协程到达 co_return 表达式时调用:

```
1  template<typename T>
2  class ResultTask {
3    public:
4    struct promise_type {
5      T result{}; // value from co_return
6      void return_value(const auto& value) { // reaction to co_return
7        result = value; // - store value locally
8      }
9      ...
10   };
11   ...
12 };
```

当调用 getResult() 时，协程接口只返回这个值:

```
1  template<typename T>
2  class ResultTask {
3    public:
4    ...
```

```
  T getResult() const {
    return hdl.promise().result;
  }
};
```

return_void() 和 return_value()

若协程以有时可能返回值，有时可能不返回值的方式实现，则这是未定义行为。则这个协程无效:

```
ResultTask<int> coroUB( ... )
{
  if ( ... ) {
    co_return 42;
  }
}
```

14.4. 协程的 Awaitable 对象和 Awaiter

目前为止，已经看到了如何使用协程接口从外部控制协程 (包装协程句柄及其承诺)。然而，协程可以 (而且必须) 提供另一个配置点:Awaitables(实现方式是使用 Awaiter)。

相关术语如下所示:

- Awaitables 是运算符 co_await 需要作为其操作数的术语，所以 awaitables 是 co_await 可以处理的所有对象。
- Awaiter 是实现 Awaitables 的一种特定 (和典型) 方式的术语。其必须提供三个特定的成员函数来处理协程的暂停和恢复。

Awaitables 在调用 co_await(或 co_yield) 时使用，允许提供拒绝请求的代码来挂起 (暂时或在某些条件下) 或执行一些用于挂起和恢复的逻辑。

我们已经使用过两种 Awaiter 类型:std::suspend_always 和 std::suspend_never。

14.4.1 Awaiters

Awaiters 协同程序时可以调用暂停或恢复，”Awaiter 的特殊成员函数” 表列出了 awaiter 必须提供的操作。

操作	效果
await_ready()	生成是否 (当前) 禁用挂起
await_suspend(awaitHdl)	处理挂起
await_resume()	处理恢复

表 14.1 Awaiters 的特殊成员函数

关键函数是 await_suspend() 和 await_resume():

- await_ready()
  这个函数会在协程被挂起之前调用，提供 (暂时) 完全关闭挂起。若它返回 true，则协程根本不会挂起。
  这个函数通常只返回 false(“不，不要阻止/忽略任何挂起”)。为了节省挂起的成本，当挂起无意义时 (例如，若挂起依赖于可用的某些数据)，可能会有条件地产生 true。
  这个函数中，协程还没有挂起。所以，其不应该用来直接或间接地调用其所调用的协程的 resume() 或 destroy()。
- auto await_suspend(awaitHdl)
  在协程挂起后立即为协程调用此函数。参数 awaitHdl 是被挂起的协程的句柄，其的类型是 std::coroutine_handle<PromiseType>。
  这个函数中，可以指定下一步要做什么，包括立即恢复挂起的或等待的协程。不同的返回类型允许以不同的方式指定，还可以通过直接将控制转移到另一个协程来有效地跳过暂停。
  甚至可以破坏协程，但请确保不再在其他任何地方使用协程 (例如：在协程接口中调用 done() 或 destroy())。
- auto await_resume()
  当成功挂起后恢复协程时，将为协程调用此函数。
  可以返回一个值，这个值就是 co_await 表达式产生的值。

考虑下面这个简单的 awaiter 使用:

coro/awaiter.hpp

```cpp
#include <iostream>

class Awaiter {
  public:
  bool await_ready() const noexcept {
    std::cout << " await_ready()\n";
    return false; // do suspend
  }

  void await_suspend(auto hdl) const noexcept {
    std::cout << " await_suspend()\n";
  }

  void await_resume() const noexcept {
    std::cout << " await_resume()\n";
  }
};
```

这个 awaiter 中，跟踪该 awaiter 的哪个函数何时调用。因为 await_ready() 返回 false，而 await_suspend() 不返回任何值，所以 awaiter 接受挂起 (不恢复其他操作)。这是标准的 await std::suspend_always 结合一些 print 语句的行为，其他返回类型/值可以提供不同的行为。

协程可以像下面这样使用这个 awaiter(完整代码参见 coro/awaiter.cpp):

```
CoroTask coro(int max)
{
  std::cout << "   CORO start\n";
  for (int val = 1; val <= max; ++val) {
    std::cout << "   CORO " << val << '\n';
    co_await Awaiter{}; // SUSPEND with our own awaiter
  }
  std::cout << "   CORO end\n";
}
```

假设这样使用协程:

```
auto coTask = coro(2);
std::cout << "started\n";

std::cout << "loop\n";
while (coTask.resume()) { // RESUME
  std::cout << " suspended\n";
}
std::cout << "done\n";
```

可得到以下输出:

```
started
loop
  CORO start
  CORO 1
    await_ready()
    await_suspend()
  suspended
    await_resume()
  CORO 2
    await_ready()
    await_suspend()
  suspended
    await_resume()
  CORO end
done
```

稍后将讨论更多关于 awaiters 更多的细节。

14.4.2 标准 Awaiters

C++ 标准库提供了已经使用过的两个简单的 awaiter:

- std::suspend_always

- std::suspend_never

其定义非常简单:

```
namespace std {
  struct suspend_always {
    constexpr bool await_ready() const noexcept { return false; }
    constexpr void await_suspend(coroutine_handle<>) const noexcept { }
    constexpr void await_resume() const noexcept { }
  };

  struct suspend_never {
    constexpr bool await_ready() const noexcept { return true; }
    constexpr void await_suspend(coroutine_handle<>) const noexcept { }
    constexpr void await_resume() const noexcept { }
  };
}
```

两者都没有暂停或恢复，但 await_ready() 的返回值有所不同:

- 若在 await_ready() 中返回 false(而在 await_suspend() 中没有返回任何值)，则 suspend_always 接受每个挂起，并将协程返回给其调用者。
- 若在 await_ready() 中返回 true，则 suspend_never 永远不会接受任何挂起，则协程继续 (永远不会调用 await_suspend())。

awaiter 通常用作 co_await 的基本 awaiter:

```
co_await std::suspend_always{};
```

也可以在协程承诺的 initial_suspend() 和 finally_suspend() 中返回。

14.4.3 恢复子协程

通过将协程接口设置为可等待的 (提供 awaiter API)，允许它以一种方式处理子协程，使子协程的挂起点成为主协程的挂起点。

这允许开发者避免使用嵌套循环进行恢复，如示例 corocoro.cpp 所介绍的那样。

以 CoroTask 的第一个实现为例，只需要以下修改:

- 协程接口必须知道它的子协程 (若有的话):

```
class CoroTaskSub {
public:
  struct promise_type;
  using CoroHdl = std::coroutine_handle<promise_type>;
private:
  CoroHdl hdl; // native coroutine handle
public:
  struct promise_type {
```

```
    CoroHdl subHdl = nullptr; // sub-coroutine (if there is one)
    ...
  }
  ...
};
```

- 协程接口必须提供 awaiter 的 API，以便该接口可以作为 co_await 的 awaitable 对象使用:

```
class CoroTaskSub {
  public:
  ...
  bool await_ready() { return false; } // do not skip suspension
  void await_suspend(auto awaitHdl) {
    awaitHdl.promise().subHdl = hdl; // store sub-coroutine and suspend
  }
  void await_resume() { }
};
```

- 协程接口必须恢复尚未完成的最深层子协程 (若有的话):

```
class CoroTaskSub {
  public:
  ...
  bool resume() const {
    if (!hdl || hdl.done()) {
      return false; // nothing (more) to process
    }
    // find deepest sub-coroutine not done yet:
    CoroHdl innerHdl = hdl;
    while (innerHdl.promise().subHdl && !innerHdl.promise().subHdl.done()) {
      innerHdl = innerHdl.promise().subHdl;
    }
    innerHdl.resume(); // RESUME
    return !hdl.done();
  }
  ...
};
```

coro/corottasksub.hpp 中为完整的代码。

有了这个，就可以做以下事情了:

coro/corocorosub.cpp

```
#include <iostream>
#include "corotasksub.hpp" // for CoroTaskSub

CoroTaskSub coro()
{
  std::cout << " coro(): PART1\n";
  co_await std::suspend_always{}; // SUSPEND
  std::cout << " coro(): PART2\n";
```

```
}

CoroTaskSub callCoro()
{
  std::cout << " callCoro(): CALL coro()\n";
  co_await coro(); // call sub-coroutine
  std::cout << " callCoro(): coro() done\n";
  co_await std::suspend_always{}; // SUSPEND
  std::cout << " callCoro(): END\n";
}

int main()
{
  auto coroTask = callCoro(); // initialize coroutine
  std::cout << "MAIN: callCoro() initialized\n";

  while (coroTask.resume()) { // RESUME
    std::cout << "MAIN: callCoro() suspended\n";
  }

  std::cout << "MAIN: callCoro() done\n";
}
```

callCoro() 内部，可以通过将其传递给 co_await() 来调用 coro():

```
CoroTaskSub callCoro()
{
  ...
  co_await coro(); // call sub-coroutine
  ...
}
```

这里发生了如下的事情:

- coro() 的调用初始化协程，并产生 CoroTaskSub 类型的协程接口。
- 因为这个类型有一个 awaiter 接口，协程接口可以作为 co_await 的 awaitable 接口使用。然后用两个操作数调用操作符 co_await:
  - 调用它的等待协程
  - 调用的协程
- awaiter 通常的行为是:
  - await_ready() 通常询问是否拒绝等待请求。答案是“不”(错误)。
  - await_suspend() 以等待协程句柄作为参数调用。
- 通过存储子协程的句柄 (调用 await_suspend() 的对象) 作为传递的等待句柄的子协程，callCoro() 现在知道其子协程。
- 通过在 await_suspend() 中不返回任何值，挂起最终被接受，这意味着：

```
co_await coro();
```

挂起 callCoro() 并将控制流传输回 main()。

- 当 main() 然后恢复 callCoro() 时，CoroTaskSub::resume() 的实现发现 coro() 作为其最深层的子协程并恢复它。
- 每当子协程挂起时，CoroTaskSub::resume() 返回给调用者。
- 这个过程一直持续到子协同程序完成，接下来将恢复 callCoro()。

最后，程序有以下输出:

```
MAIN: callCoro() initialized
  callCoro(): CALL coro()
MAIN: callCoro() suspended
    coro(): PART1
MAIN: callCoro() suspended
    coro(): PART2
MAIN: callCoro() suspended
  callCoro(): coro() done
MAIN: callCoro() suspended
  callCoro(): END
MAIN: callCoro() done
```

直接恢复调用的子协程

注意，前面的 CoroTaskSub 实现调用

```
co_await coro(); // call sub-coroutine
```

有一个挂起点。初始化 coro()，但在启动它之前，将控制流转移回 callCoro() 的调用者。

也可以在这里直接启动 coro()，只需要对 await_suspend() 进行如下修改:

```
auto await_suspend(auto awaitHdl) {
  awaitHdl.promise().subHdl = hdl; // store sub-coroutine
  return hdl; // and resume it directly
}
```

若 await_suspend() 返回协程句柄，则立即恢复该协程。这会将程序的行为改变为以下输出:

```
MAIN: callCoro() initialized
  callCoro(): CALL coro()
    coro(): PART1
MAIN: callCoro() suspended
  callCoro(): coro() done
MAIN: callCoro() suspended
  callCoro(): END
MAIN: callCoro() done
```

将另一个协程返回到 resume 可用于恢复 co_await 上的任何其他协程，稍后将使用它进行对称传输。

如前所见，await_suspend() 的返回类型可以有所不同。

通常，若 await_suspend() 发出不应该有挂起的信号，则永远不会调用 await_resume()。

14.4.4 挂起后将值传递回协程

awaitables 和 awaiter 的另一个应用是允许在挂起后将值传递回协程。考虑以下协程:

coro/coyieldback.hpp

```
#include <iostream>
#include "corogenback.hpp" // for CoroGenBack

CoroGenBack coro(int max)
{
  std::cout << "               CORO " << max << " start\n";

  for (int val = 1; val <= max; ++val) {
    // print next value:
    std::cout << " CORO " << val << '/' << max << '\n';

    // yield next value:
    auto back = co_yield val; // SUSPEND with value with response
    std::cout << "            CORO => " << back << "\n";
  }

  std::cout << "               CORO " << max << " end\n";
}
```

同样，协程迭代到最大值，并在挂起时将当前值返回给调用者。这一次，co_yield 从调用者返回一个值给协程:

```
auto back = co_yield val; // SUSPEND with value with response
```

为了支持这一点，协程接口提供了一些修改后的常用 API:

coro/corogenback.hpp

```
#include "backawaiter.hpp"
#include <coroutine>
#include <exception> // for terminate()
#include <string>

class [[nodiscard]] CoroGenBack {
public:
  struct promise_type;
  using CoroHdl = std::coroutine_handle<promise_type>;
private:
```

```
  CoroHdl hdl; // native coroutine handle

public:
  struct promise_type {
    int coroValue = 0; // value TO caller on suspension
    std::string backValue; // value back FROM caller after suspension
    auto yield_value(int val) { // reaction to co_yield
      coroValue = val; // - store value locally
      backValue.clear(); // - reinit back value
      return BackAwaiter<CoroHdl>{}; // - use special awaiter for response
    }

    // the usual members:
    auto get_return_object() { return CoroHdl::from_promise(*this); }
    auto initial_suspend() { return std::suspend_always{}; }
    void return_void() { }
    void unhandled_exception() { std::terminate(); }
    auto final_suspend() noexcept { return std::suspend_always{}; }
  };

  // constructors and destructor:
  CoroGenBack(auto h) : hdl{h} { }
  ~CoroGenBack() { if (hdl) hdl.destroy(); }

  // no copying or moving:
  CoroGenBack(const CoroGenBack&) = delete;
  CoroGenBack& operator=(const CoroGenBack&) = delete;

  // API:
  // - resume the coroutine:
  bool resume() const {
  if (!hdl || hdl.done()) {
    return false; // nothing (more) to process
  }
  hdl.resume(); // RESUME
  return !hdl.done();
  }

  // - yield value from co_yield:
  int getValue() const {
    return hdl.promise().coroValue;
  }

  // - set value back to the coroutine after suspension:
  void setBackValue(const auto& val) {
    hdl.promise().backValue = val;
  }
};
```

修改内容如下:

- promise_type 有一个新成员 backValue。
- yield_value() 返回一个特殊的 BackAwaiter 类型。
- 协程接口有一个新成员 setBackValue()，用于将值发送回协程。

让我们详细讨论一下这些变化。

promise 的数据成员

通常，协程接口的 promise 类型是协程与调用者共享和交换数据的最佳位置:

```
class CoroGenBack {
  ...
  public:
  struct promise_type {
    int coroValue = 0; // value TO caller on suspension
    std::string backValue; // value back FROM caller after suspension
    ...
  };
  ...
};
```

现在这里有两个数据成员:

- coroValue 从协程传递给调用者的值
- backValue 从调用者传回给协程的值

co_yield 的新 awaiter

要通过 promise 将值传递给调用者，还需要实现 yield_value()。这一次，yield_value() 不会返回 std::suspend_always{}。相反，它会返回一个特殊的类型为 BackAwaiter 的 awaiter 对象，能够返回调用者返回的值:

```
class CoroGenBack {
  ...
  public:
  struct promise_type {
    int coroValue = 0; // value TO caller on suspension
    std::string backValue; // value back FROM caller after suspension

    auto yield_value(int val) { // reaction to co_yield
      coroValue = val; // - store value locally
      backValue.clear(); // - reinit back value
      return BackAwaiter<CoroHdl>{}; // - use special awaiter for response
    }
    ...
```

```
  };
  ...
};
```

awaiter 的定义如下所示:

coro/backawaiter.hpp

```
template<typename Hdl>
class BackAwaiter {
  Hdl hdl = nullptr; // coroutine handle saved from await_suspend() for
  ↪ await_resume()
  public:
  BackAwaiter() = default;

  bool await_ready() const noexcept {
    return false; // do suspend
  }

  void await_suspend(Hdl h) noexcept {
    hdl = h; // save handle to get access to its promise
  }

  auto await_resume() const noexcept {
    return hdl.promise().backValue; // return back value stored in the promise
  }
};
```

awaiter 所做的只是将传递给 await_suspend() 的协程句柄存储在本地，以便在调用 await_resume() 时拥有它。在 await_resume() 中，使用此句柄从其承诺 (由调用者使用 setBackValue() 存储) 中产生 backValue。

yield_value() 的返回值用作 co_yield 表达式的值:

```
auto back = co_yield val; // co_yield yields return value of yield_value()
```

BackAwaiter 是通用的，因为它可以用于具有任意类型成员 backValue 的所有协程句柄。

挂起后返回值的协程接口

通常，必须决定协程接口提供的 API，以让调用者返回响应。一个简单的方法是提供成员函数 setBackValue():

```
class CoroGenBack {
  ...
  public:
  struct promise_type {
    int coroValue = 0; // value TO caller on suspension
```

```
    std::string backValue; // value back FROM caller after suspension
    ...
  };
  ...
  // - set value back to the coroutine after suspension:
  void setBackValue(const auto& val) {
    hdl.promise().backValue = val;
  }
};
```

也可以提供一个接口，生成一个 backValue 的引用，以支持对它的直接赋值。

使用协程

协程也可以这样用:

coro/coyieldback.cpp

```
#include "coyieldback.hpp"
#include <iostream>
#include <vector>

int main()
{
  // start coroutine:
  auto coroGen = coro(3); // initialize coroutine
  std::cout << "**** coro() started\n";

  // loop to resume the coroutine until it is done:
  std::cout << "\n**** resume coro()\n";
  while (coroGen.resume()) { // RESUME
    // process value from co_yield:
    auto val = coroGen.getValue();
    std::cout << "**** coro() suspended with " << val << '\n';

    // set response (the value co_yield yields):
    std::string back = (val % 2 != 0 ? "OK" : "ERR");
    std::cout << "\n**** resume coro() with back value: " << back << '\n';
    coroGen.setBackValue(back); // set value back to the coroutine
  }

  std::cout << "**** coro() done\n";
}
```

该程序有以下输出:

```
**** coro() started

**** resume coro()
```

```
         CORO 3 start
         CORO 1/3
**** coro() suspended with 1

**** resume coro() with back value: OK
         CORO => OK
         CORO 2/3
**** coro() suspended with 2

**** resume coro() with back value: ERR
         CORO => ERR
         CORO 3/3
**** coro() suspended with 3

**** resume coro() with back value: OK
         CORO => OK
         CORO 3 end
**** coro() done
```

协程接口具有以不同方式处理事物的所有灵活性。例如，可以将响应作为 resume() 的参数传递，或者在一个成员中共享生成值及其响应。

14.5. 附注

支持协程的请求最初是由 Oliver Kowalke 和 Nat Goodspeed 在http://wg21.link/n3708中作为纯库扩展提出。

由于该特性的复杂性，通过http://wg21.link/n4403建立了协程 TS(实验技术规范) 来处理细节。

最终将协程合并到 C++20 标准中的提案是由 Gor Nishanov 在http://wg21.link/p0912r5中提出。

第 15 章 协程详情

我们已经在前一章学习了协程，本章将详细讨论协程。

15.1. 协程的约束

协程具有以下属性和限制:

- 协程不允许有返回语句。
- 协程不能是 constexpr 或 consteval。
- 协程不能有返回类型 auto 或其他占位符类型。
- main() 不能是协程。
- 构造函数或析构函数不可为协程。

协程可以是静态的。协程若不是构造函数或析构函数，可以是成员函数。

协程甚至可以是 Lambda，但在这种情况下，必须谨慎使用。

15.1.1 Lambda 协程

协程可以是 Lambda，也可以像这样实现第一个协程示例:

```
auto coro = [] (int max) -> CoroTask {
        std::cout << "          CORO " << max << " start\n";
        for (int val = 1; val <= max; ++val) {
          // print next value:
          std::cout << " CORO " << val << '/' << max << '\n';
          co_await std::suspend_always{}; // SUSPEND
        }
        std::cout << "         CORO " << max << " end\n";
};
```

但请注意协程 Lambda 不应该捕获任何内容。这是因为 Lambda 是定义函数对象的快捷方式，该对象在定义 Lambda 的作用域中创建。当离开该作用域时，协程 Lambda 可能仍然会恢复，这是 Lambda 对象已销毁了。

下面是一个导致致命运行时错误的代码示例:

```
auto getCoro()
{
  string str = "      CORO ";
  auto coro = [str] (int max) -> CoroTask {
        std::cout << str << max << " start\n";
        for (int val = 1; val <= max; ++val) {
          std::cout << str << val << '/' << max << '\n';
          co_await std::suspend_always{}; // SUSPEND
        }
        std::cout << str << max << " end\n";
```

```
    };
  return coro;
}

auto coroTask = getCoro()(3); // initialize coroutine
// OOPS: lambda destroyed here
coroTask.resume(); // FATAL RUNTIME ERROR
```

通常，返回一个按值捕获的 Lambda 没有生命周期问题。

可以像下面这样使用协程 Lambda:

```
auto coro = getCoro(); // initialize coroutine lambda
auto coroTask = coro(3); // initialize coroutine
coroTask.resume(); // OK
```

还可以通过引用将 str 传递给 getCoro()，并确保只要使用协程，str 就存在。

但不能将两个初始化组合到一个语句中，因为很容易出现更微妙的生命周期问题，所以强烈建议不要在协程 Lambda 中使用捕获。

15.2. 协程框架和 promise

当协程启动时，会发生三件事:

- 创建协程框架来存储协程的必要数据，这通常发生在堆上。

  但允许编译器将协程帧放在堆栈上。若协程的生命周期在调用者的生命周期内，并且编译器有足够的信息来计算帧的大小，则会发生这种情况。

- 协程的所有参数都会复制到帧中。

  注意，引用复制为引用; 这并不是说他们的值复制了。所以只要协程在运行，引用形参所引用的实参必须有效。

  建议是永远不要将协程参数声明为引用。否则，可能会发生带有未定义行为的致命运行时错误。

- promise 对象是在框架内创建的，其目的是存储协程的状态，并在协程运行时提供用于自定义的钩子。

  可以将这些对象视为“协程状态控制器”(一个控制协程行为，并可用于跟踪其状态的对象)。

图 15.1 可视化了这个初始化过程，并显示了在协程运行时使用 promise 对象的哪些自定义点。

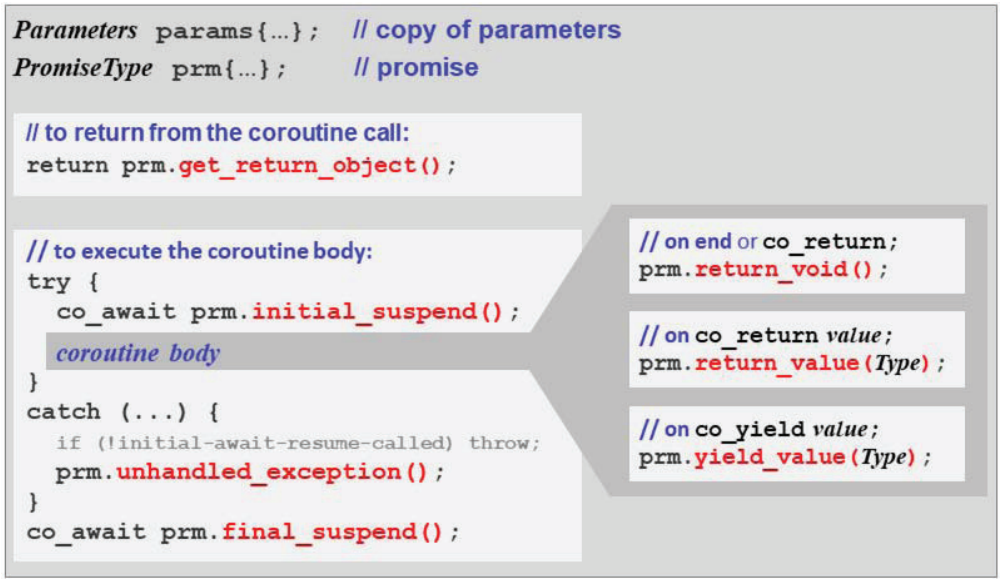

图 15.1 协程框架和 promise

15.2.1 如何与协程接口、promise 和 awaitable 对象交互

让我们再次使用 awaitable，看看如何作为一个整体进行组织的:

- 对于每一个启动的协程，编译器都会创建一个 promise。
- 该 promise 嵌入在协程句柄中，然后放置在协程接口中，协程接口通常控制句柄及其 promise 的生命周期。
- 暂停时，协程使用 awaitable 对象来控制暂停和恢复时发生的事情。

下面的程序通过使用跟踪协程接口、promise 和 awaiter 来演示精确的控制流。跟踪协程接口和 promise 的实现如下所示:

coro/tracingcoro.hpp

```
#include <iostream>
#include <coroutine>
#include <exception> // for terminate()

// coroutine interface to deal with a simple task
// - providing resume() to resume it
class [[nodiscard]] TracingCoro {
  public:
  // native coroutine handle and its promise type:
  struct promise_type;
  using CoroHdl = std::coroutine_handle<promise_type>;
  CoroHdl hdl; // coroutine handle

  // helper type for state and customization:
  struct promise_type {
    promise_type() {
      std::cout << " PROMISE: constructor\n";
    }
```

```
19      ~promise_type() {
20        std::cout << " PROMISE: destructor\n";
21      }
22      auto get_return_object() { // init and return the coroutine interface
23        std::cout << " PROMISE: get_return_object()\n";
24        return TracingCoro{CoroHdl::from_promise(*this)};
25      }
26      auto initial_suspend() { // initial suspend point
27        std::cout << " PROMISE: initial_suspend()\n";
28        return std::suspend_always{}; // - start lazily
29      }
30      void unhandled_exception() { // deal with exceptions
31        std::cout << " PROMISE: unhandled_exception()\n";
32        std::terminate(); // - terminate the program
33      }
34      void return_void() { // deal with the end or co_return;
35        std::cout << " PROMISE: return_void()\n";
36      }
37      auto final_suspend() noexcept { // final suspend point
38        std::cout << " PROMISE: final_suspend()\n";
39        return std::suspend_always{}; // - suspend immediately
40      }
41    };
42
43    // constructor and destructor:
44    TracingCoro(auto h)
45    : hdl{h} { // store coroutine handle in interface
46    std::cout << " INTERFACE: construct\n";
47    }
48    ~TracingCoro() {
49      std::cout << " INTERFACE: destruct\n";
50      if (hdl) {
51        hdl.destroy(); // destroy coroutine handle
52      }
53    }
54
55    // don't copy or move:
56    TracingCoro(const TracingCoro&) = delete;
57    TracingCoro& operator=(const TracingCoro&) = delete;
58
59    // API to resume the coroutine
60    // - returns whether there is still something to process
61    bool resume() const {
62      std::cout << " INTERFACE: resume()\n";
63      if (!hdl || hdl.done()) {
64        return false; // nothing (more) to process
65      }
66      hdl.resume(); // RESUME
67      return !hdl.done();
```

```
  }
};
```

我们跟踪:

- 当协程接口初始化和销毁时
- 当协程接口恢复协程时
- 每个 promise 操作

跟踪 awaiter 的实现如下:

coro/tracingawaiter.hpp

```cpp
#include <iostream>

class TracingAwaiter {
  inline static int maxId = 0;
  int id;
  public:
  TracingAwaiter() : id{++maxId} {
    std::cout << " AWAITER" << id << ": ==> constructor\n";
  }
  ~TracingAwaiter() {
    std::cout << " AWAITER" << id << ": <== destructor\n";
  }
  // don't copy or move:
  TracingAwaiter(const TracingAwaiter&) = delete;
  TracingAwaiter& operator=(const TracingAwaiter&) = delete;

  // constexpr
  bool await_ready() const noexcept {
    std::cout << " AWAITER" << id << ": await_ready()\n";
    return false; // true: do NOT (try to) suspend
  }

  // Return type/value means:
  // - void: do suspend
  // - bool: true: do suspend
  // - handle: resume coro of the handle
  // constexpr
  bool await_suspend(auto) const noexcept {
    std::cout << " AWAITER" << id << ": await_suspend()\n";
    return false;
  }

  // constexpr
  void await_resume() const noexcept {
    std::cout << " AWAITER" << id << ": await_resume()\n";
  }
};
```

这里，还跟踪每个操作。

成员函数不能是 constexpr，因为它们有 I/O。

协程的实现和使用如下所示:

coro/corotrace.cpp

```
#include "tracingcoro.hpp"
#include "tracingawaiter.hpp"
#include <iostream>

TracingCoro coro(int max)
{
  std::cout << " START coro(" << max << ")\n";
  for (int i = 1; i <= max; ++i) {
    std::cout << "  CORO: " << i << '/' << max << '\n';
    co_await TracingAwaiter{}; // SUSPEND
    std::cout << "   CONTINUE coro(" << max << ")\n";
  }
  std::cout << "  END coro(" << max << ")\n";
}

int main()
{
  // start coroutine:
  std::cout << "**** start coro()\n";
  auto coroTask = coro(3); // init coroutine
  std::cout << "**** coro() started\n";

  // loop to resume the coroutine until it is done:
  std::cout << "\n**** resume coro() in loop\n";
  while (coroTask.resume()) { // RESUME
    std::cout << "**** coro() suspended\n";
    ...
    std::cout << "\n**** resume coro() in loop\n";
  }

  std::cout << "\n**** coro() loop done\n";
}
```

该程序有以下输出:

```
**** start coro()
      PROMISE: constructor
      PROMISE: get_return_object()
        INTERFACE: construct
      PROMISE: initial_suspend()
**** coro() started

**** resume coro() in loop
```

```
        INTERFACE: resume()
  START coro(3)
  CORO: 1/3
          AWAITER1: ==> constructor
          AWAITER1: await_ready()
          AWAITER1: await_suspend()
**** coro() suspended

**** resume coro() in loop
        INTERFACE: resume()
          AWAITER1: await_resume()
          AWAITER1: <== destructor
  CONTINUE coro(3)
  CORO: 2/3
          AWAITER2: ==> constructor
          AWAITER2: await_ready()
          AWAITER2: await_suspend()
**** coro() suspended

**** resume coro() in loop
        INTERFACE: resume()
          AWAITER2: await_resume()
          AWAITER2: <== destructor
  CONTINUE coro(3)
  CORO: 3/3
          AWAITER3: ==> constructor
          AWAITER3: await_ready()
          AWAITER3: await_suspend()
**** coro() suspended

**** resume coro() in loop
        INTERFACE: resume()
          AWAITER3: await_resume()
          AWAITER3: <== destructor
  CONTINUE coro(3)
  END coro(3)
      PROMISE: return_void()
      PROMISE: final_suspend()

**** coro() loop done
        INTERFACE: destruct
      PROMISE: destructor
```

当调用协程时，发生的第一件事就是创建协程的 promise。

对创建的 promise 调用 get_return_object() 方法，这个函数通常初始化协程句柄并返回用它初始化的协程接口。要创建句柄，通常会调用 from_promise() 中的静态成员函数，然后传递句柄来初始化 TracingCoro 类型的协程接口，再将协程接口返回给 get_return_object() 的调用者，以便可以用作协程调用的返回值 (极少数情况下可能有其他返回类型)。

然后，调用 initial_suspend() 来查看是否应该立即挂起协程 (惰性启动)，所以控制流会返回给协程的调用者。

稍后，main() 调用 resume()，resume() 为协程句柄调用 resume()。此调用恢复协程，所以协程将处理以下语句，直到下一次挂起或结束。

对于每个暂停，co_await 都会使用一个 awaitable 对象，在本例中它是一个类型为 TracingAwaiter 的 awaiter 对象，是用默认构造函数创建的。对于 awaiter，调用成员函数 await_ready() 和 await_suspend() 来控制挂起 (甚至可能拒绝挂起)。因为 await_ready() 返回 false，而 await_suspend() 不返回任何值，所以接受挂起请求。因此，协程的恢复结束，main() 继续。下一次恢复时，调用 await_resume() 并继续协程。

当协程到达其结束或 co_return 语句时，将调用处理其结束的相应成员函数。首先，根据是否有返回值，调用 return_void() 或 return_value()。然后，调用用于最终挂起的函数 final_suspend()。注意，即使在 final_suspend() 中，协程仍在“运行”，则再次调用 resume()，或在 final_suspend() 中调用 destroy()，将导致运行时错误。

协程接口生命周期结束时，调用其析构函数，析构函数将销毁协程句柄，销毁 promise。

作为一种替代方案，假设初始化的_suspend() 返回一个 promise 类型，表示提前启动协程，而不是一开始就挂起它:

```
class [[nodiscard]] TracingCoro {
  public:
  ...
  struct promise_type {
    ...
    auto initial_suspend() { // initial suspend point
      std::cout << " PROMISE: initial_suspend()\n";
      return std::suspend_never{}; // - start eagerly
    }
    ...
  };
  ...
};
```

将得到以下输出:

```
**** start coro()
      PROMISE: constructor
      PROMISE: get_return_object()
        INTERFACE: construct
      PROMISE: initial_suspend()
  START coro(3)
  CORO: 1/3
          AWAITER1: ==> constructor
          AWAITER1: await_ready()
          AWAITER1: await_suspend()
**** coro() started
```

```
**** resume coro() in loop
        INTERFACE: resume()
          AWAITER1: await_resume()
          AWAITER1: <== destructor
  CONTINUE coro(3)
  CORO: 2/3
          AWAITER2: ==> constructor
          AWAITER2: await_ready()
          AWAITER2: await_suspend()
**** coro() suspended

**** resume coro() in loop
        INTERFACE: resume()
          AWAITER2: await_resume()
          AWAITER2: <== destructor
  CONTINUE coro(3)
  CORO: 3/3
          AWAITER3: ==> constructor
          AWAITER3: await_ready()
          AWAITER3: await_suspend()
**** coro() suspended

**** resume coro() in loop
        INTERFACE: resume()
          AWAITER3: await_resume()
          AWAITER3: <== destructor
  CONTINUE coro(3)
  END coro(3)
      PROMISE: return_void()
      PROMISE: final_suspend()

**** coro() loop done
        INTERFACE: destruct
      PROMISE: destructor
```

协程框架创建了 promise，并对其调用 get_return_object()，这初始化了协程句柄和协程接口，但 initial_suspend() 不会挂起。因此，协程开始急切地执行第一个语句，直到第一个 co_await 第一次挂起。当最初在 get_return_object() 中创建的 TracingCoro 对象返回给协程的调用者时，就触发了这个暂停。之后，和前面一样，恢复循环遍历。

15.3. 协程 promise 的详情

“协程 promise 的特殊成员函数”表列出了 promise 类型中为协程句柄提供的所有操作。

操作	效果
Constructor	初始化 promise 对象

get_return_object()	定义协程返回的对象 (通常为协程接口类型)
initial_suspend()	初始挂起点 (让协程惰性启动)
yield_value(val)	处理 co_yield 的值
unhandled_exception()	对协程内未处理异常的反应
return_void()	处理 end 或不返回任何值的 co_return
return_value(val)	处理 co_return 的返回值
final_suspend()	最终挂起点 (让协程惰性结束)
await_transform(...)	将值从 co_await 映射到 awaiter
operator new(sz)	定义协程分配内存的方式
operator delete(ptr, sz)	定义协程释放内存的方式
get_return_object_on_... ...allocation_failure()	定义内存分配失败时的反应

表 15.1 协程 promise 的特殊成员函数

其中一些函数是强制性的，一些函数取决于协程是产生中间结果还是最终结果，还有一些函数是可选的。

15.3.1 强制性 promise 操作

对于协程 promise，需要以下成员函数。否则，代码就会无法编译，或者会出现未定义行为。

构造函数

协程 promise 的构造函数在协程初始化时由协程框架调用。

编译器可以使用构造函数用一些参数初始化协程状态，因此构造函数的签名必须与调用协程时传递给协程的参数相匹配。这种技术特别适用于协同程序特征。

get_return_object()

get_return_object() 由协程框架用于创建协程的返回值。

该函数通常必须返回协程接口，该接口由协程句柄初始化，而协程句柄通常 from_promise() 中的静态协程句柄成员函数初始化，该函数本身由 promise 初始化。

例如:

```
class CoroTask {
  public:
  // native coroutine handle and its promise type:
  struct promise_type;
  using CoroHdl = std::coroutine_handle<promise_type>;
  CoroHdl hdl;

```

```
  struct promise_type {
    auto get_return_object() { // init and return the coroutine interface
      return CoroTask{CoroHdl::from_promise(*this)};
    }
    ...
  }
  ...
};
```

原则上，get_return_object() 可以有不同的返回类型:

- get_return_object() 通过使用协程句柄显式初始化协程接口 (如上所述) 来返回协程接口。
- 相反，get_return_object() 也可以返回协程句柄，将隐式地创建协程接口。这要求接口构造函数不是显式的。
  目前还不清楚在这种情况下何时创建协程接口 (参见http://wg21.link/cwg2563)，所以在调用 initial_suspend() 时，协程接口对象可能存在，也可能不存在，很可能出现问题。
  因此，最好避免使用这种方法。
- 极少数情况下，甚至可以不返回任何东西，并将返回类型指定为 void。使用协程特征就是一个例子。

initial_suspend()

initial_suspend() 定义协程的初始挂起点，主要用于指定协程是在初始化后自动挂起 (惰性启动)，还是立即启动 (急切启动)。

协程的 promise prm 会使用该函数，如下所示:

```
co_await prm.initial_suspend();
```

因此，initial_suspend() 应该返回一个 awaiter。

通常，initial_suspend() 会返回

- std::suspend_always{}，若协程体较晚/惰性地启动，或者
- std::suspend_never{}，若协程体是立即/急切启动的

例如:

```
class CoroTask {
  public:
  ...
  struct promise_type {
    ...
    auto initial_suspend() { // suspend immediately
      return std::suspend_always{}; // - yes, always
    }
    ...
  }
```

```
  ...
};
```

然而，也可以让立即启动或延迟启动的决定取决于某些业务逻辑，或者使用此函数来执行协程主体范围的一些初始化。

final_suspend() noexcept

final_suspend() 定义协程的最终挂起点，并像 initial_suspend() 一样调用协程的 promise prm，如下所示:

```
co_await prm.final_suspend();
```

这个函数在调用 return_void()、return_value() 或 unhandled_exception() 后由 try 块外的协程框架调用，try 块包含了协程体。因为 final_suspend() 在 try 代码块之外，所以必须是 noexcept。

这个成员函数的名称有一点误导，因为它给人的印象是，也可以在这里返回 std::suspend_never{} 来强制在到达协程末尾后再次执行。然而，在最终挂起点恢复真正挂起的协程是未定义的行为。对于挂起的协程，唯一能做的就是销毁它。

因此，这个成员函数的真正目的是执行一些逻辑，例如发布结果、发出完成信号或在其他地方恢复执行。

相反，建议构建协程，以便在可能的情况下暂停。一个原因是，它使编译器更容易确定协程框架的生命周期何时嵌套在协程的调用者中，这使得编译器更有可能忽略协程框架的堆内存分配。

因此，除非有很好的理由不这样做，否则 final_suspend() 应该始终返回 std::suspend_always{}。例如:

```
class CoroTask {
  public:
  ...
  struct promise_type {
    ...
    auto final_suspend() noexcept { // suspend at the end
      ...
      return std::suspend_always{}; // - yes, always
    }
    ...
  }
  ...
};
```

unhandled_exception()

或 unhandled_exception() 定义协程抛出异常时的反应，该函数在协程框架的 catch 子句中调用。

对异常的可能反应如下所示:

- 忽略异常
- 本地处理异常
- 结束或终止程序 (例如，通过调用 std::terminate())
- 使用 std::current_exception() 存储异常以供以后使用

稍后将在单独的小节中讨论实现这些反应的方法。
在不结束程序的情况下调用此函数后，将直接调用 final_suspend()，并挂起协程。
若在 unhandled_exception() 中抛出或重新抛出异常，协程也会挂起。

15.3.2 promise 操作返回或生成值

根据是否以及如何使用 co_yield 或 co_return，还需要以下一些 promise 操作。

return_void() 或 return_value()

必须实现两个成员函数中的一个来处理 return 语句和协程的结束。

- **return_void()**
  协程到达终点时调用 (到达协程体的终点或到达不带参数的 co_return 语句)。
  有关详细信息，请参阅第一个协程示例。
- **return_value(Type)**
  当协程到达带有参数的 co_return 语句时调用，传递的参数必须或必须转换为指定的类型。
  有关详细信息，请参阅带有 co_return 的协程示例。

若协程的实现方式有时可能返回值，有时可能不返回值，这是未定义行为。考虑下面的例子:

```
ResultTask<int> coroUB( ... )
{
  if ( ... ) {
    co_return 42;
  }
}
```

这个协程无效，不允许同时拥有 return_void() 和 return_value()。[关于是否允许 promise 在未来的 C++ 标准中同时具有 return_void() 和 return_value() 的讨论正在进行。]

不幸的是，若只提供 return_value(int) 成员函数，这段代码甚至可能会编译并运行，编译器甚至可能不会对此发出警告，但这种情况有望很快改变。

这里可以重载不同类型的 return_value() (void 除外) 或使其泛型:

```
struct promise_type {
  ...
  void return_value(int val) { // reaction to co_yield for int
    ... // - process returned int
  }
  void return_value(std::string val) { // reaction to co_yield for string
```

```
    ... // - process returned string
  }
};
```

协程可以共同返回不同类型的值。

```
CoroGen coro()
{
  int value = 0;
  ...
  if ( ... ) {
    co_return "ERROR: can't compute value";
  }
  ...
  co_return value;
}
```

yield_value(Type)

若协程到达 co_yield 语句，则调用 yield_value(Type)。
有关基本细节，请参阅 co_yield 的协程示例。
可以为不同的类型重载 yield_value() 或其泛型:

```
struct promise_type {
  ...
  auto yield_value(int val) { // reaction to co_yield for int
    ... // - process yielded int
    return std::suspend_always{}; // - suspend coroutine
  }
  auto yield_value(std::string val) { // reaction to co_yield for string
    ... // - process yielded string
    return std::suspend_always{}; // - suspend coroutine
  }
};
```

协程可以 co_yield 不同类型的值:

```
CoroGen coro()
{
  while ( ... ) {
    if ( ... ) {
      co_yield "ERROR: can't compute value";
    }
    int value = 0;
    ...
    co_yield value;
```

```
10      }
11  }
```

15.3.3 可选的 promise 操作

promise 也可以用来定义一些可选的操作，这些操作定义了协程的特殊行为，通常会使用一些默认行为。

await_transform()

可以定义 await_transform() 将值从 co_await 映射到 awaiter。

new() 和 delete() 操作符

new() 和 delete() 操作符允许开发者为协程状态分配内存定义不同的方式。
这些函数还可以用于确保协程不会意外地使用堆内存。

get_return_object_on_allocation_failure()

get_return_object_on_allocation_failure() 允许开发者定义，如何对协程内存分配的无异常失败做出反应。

15.4. 协程处理的详情

类型 std::coroutine_handle<> 是协程句柄的泛型类型，可用于引用正在执行或挂起的协程。其模板参数是协程的 promise 类型，可用于放置额外的数据成员和行为。

“std::coroutine_handle<> 的 API”表列出了为协程句柄提供的所有操作。

操作	效果
coroutine_handle<PrmT>::from_promise(prm)	用 promise prm 创建一个句柄
CoroHandleType{}	创建无协程的句柄
CoroHandleType{nullptr}	创建无协程的句柄
CoroHandleType{hdl}	Copies 句柄 hdl(两者都引用相同的协程)
hdl = hdl2	分配句柄 hdl2(两者都引用相同的协程)
if (hdl)	Yields 句柄是否引用协程
==, !=	Checks 两个句柄是否引用同一个协程
<, <=, >, >=, <=>	创建协程句柄之间的顺序
hdl.resume()	恢复协程
hdl()	恢复协程
hdl.done()	Yields 挂起的协程是否已结束，并且不再允许使用 resume()

hdl.destroy()	销毁协程
hdl.promise()	Yields 协程的 promise
hdl.address()	Yields 协程数据的内部地址
coroutine_handle<PrmT>::from_address(addr)	Yields 句柄的地址 addr

表 15.2 std::coroutine_handle<> 的 API

静态成员函数 from_promise() 方法初始化一个协同程序处理提供了协同程序。该函数只是将 promise 的地址存储在句柄中，若有一个 promise prm:

```
auto hdl = std::coroutine_handle<decltype(prm)>::from_promise(prm);
```

from_promise() 通常在 get_return_object() 中调用，用于协程框架创建的 promise:

```
class CoroIf {
  public:
  struct promise_type {
    auto get_return_object() { // init and return the coroutine interface
      return CoroIf{std::coroutine_handle<promise_type>::from_promise(*this)};
    }
    ...
  };
...
};
```

当句柄被默认初始化或 nullptr 用作初始值或赋值时，协程句柄不引用任何协程。这种情况下，对布尔值的转换都会产生 false:

```
std::coroutine_handle<PrmType> hdl = nullptr;

if (hdl) ... // false
```

复制和分配协程句柄的成本很低，只是复制并赋值内部指针。协程句柄通常是按值传递的，所以多个协程句柄可以引用同一个协程。开发者必须确保当另一个协程句柄恢复或销毁该协程时，该句柄永远不会调用 resume() 或 destroy()。

address() 接口产生协程的内部指针为 void*。这允许开发者将协程句柄导出到某个地方，然后使用 from_address() 的静态函数重新创建句柄:

```
om_address():
auto hdl = std::coroutine_handle<decltype(prm)>::from_promise(prm);
...
void* hdlPtr = hdl.address();
...
auto hdl2 = std::coroutine_handle<decltype(prm)>::from_address(hdlPtr);
hdl == hdl2 // true
```

只要协程存在，就可以使用该地址。在一个协程销毁后，该地址可以让另一个协程句柄重用。

15.4.1 std::coroutine_handle<void>

所有协程句柄类型都有到类 std::coroutine<void> 的隐式类型转换 (可以跳过此声明中的 void)

[原始的 C++ 标准指定所有协程句柄类型都派生自 std::coroutine<void>，但http://wg21.link/lwg3460改变了这一点。感谢黄永祥指出这一点。]

```
namespace std {
  template<typename Promise>
  struct coroutine_handle {
    ...
    // implicit conversion to coroutine_handle<void>:
    constexpr operator coroutine_handle<>() const noexcept;
    ...
  };
}
```

若不需要 promise，可以使用类型 std::coroutine_handle<void> 或类型 std::coroutine_handle<>，而不需要模板参数。注意 std::coroutine_handle<> 没有提供 promise() 成员函数:

```
void callResume(std::coroutine_handle<> h)
{
  h.resume(); // OK
  h.promise(); // ERROR: no member promise() provided for h
}

auto hdl = std::coroutine_handle<decltype(prm)>::from_promise(prm);
...
callResume(hdl); // OK: hdl converts to std::coroutine_handle<void>
```

15.5. 协程中的异常

当在协程内部抛出异常并且该异常没有得到本地处理时，unhandled_exception() 将在协程主体后面的 catch 子句中调用 (参见图 15.1)。在出现 initial_suspend()、yield_value()、return_void()、return_value() 或协程中使用的任何可等待对象的异常情况时，也会调用该函数。

以下几种方式来处理这些异常:

- 忽略异常

  unhandled_exception() 只有一个空函数体:

```
void unhandled_exception() {
}
```

- 本地处理异常

  unhandled_exception() 只处理异常。在 catch 子句中，必须重新抛出异常并在本地处理:

```
void unhandled_exception() {
  try {
```

```
    throw; // rethrow caught exception
  }
  catch (const std::exception& e) {
    std::cerr << "EXCEPTION: " << e.what() << std::endl;
  }
  catch (...) {
    std::cerr << "UNKNOWN EXCEPTION" << std::endl;
  }
}
```

- 结束或终止程序 (例如，通过调用 std::terminate()):

```
void unhandled_exception() {
  ...
  std::terminate();
}
```

- 使用 std::current_exception() 存储异常以供以后使用

  需要在 promise 类型中设置一个异常指针:

```
struct promise_type {
  std::exception_ptr ePtr;
  ...
  void unhandled_exception() {
    ePtr = std::current_exception();
  }
};
```

必须提供在协程接口中处理异常的方法。例如:

```
class [[nodiscard]] CoroTask {
  ...
  bool resume() const {
    if (!hdl || hdl.done()) {
      return false;
    }
    hdl.promise().ePtr = nullptr; // no exception yet
    hdl.resume(); // RESUME
    if (hdl.promise().ePtr) { // RETHROW any exception from the coroutine
      std::rethrow_exception(hdl.promise().ePtr);
    }
    return !hdl.done();
  }
};
```

当然，这些方法也可以进行组合。

调用 unhandled_exception() 而不结束程序之后，协程就结束了。直接调用 final_suspend()，协程会挂起。

若在 unhandled_exception() 中抛出或重新抛出异常，协程也会挂起。unhandled_exception() 抛出的任何异常都会忽略。

15.6. 为协程帧分配内存

协程需要内存来保存它们的状态。因为协程可能会切换上下文，所以通常使用堆内存。本节将讨论如何做到这一点，以及如何更改。

15.6.1 协程如何分配内存

协程需要内存来存储它们从挂起到恢复的状态，但由于恢复可能发生在非常不同的上下文中，因此通常只能在堆上分配内存。事实上，C++ 标准规定:

> 实现可能需要为协程分配额外的存储空间。这种存储称为协程状态，通过调用非数组分配函数获得。

这里重要的词是“可能”，编译器可以优化掉对堆内存的需求。当然，编译器需要足够的信息，代码必须非常简单。事实上，最有可能优化的是

- 协程的生命周期保持在调用者的生命周期内。
- 使用内联函数使编译器至少可以看到计算帧大小的所有内容。
- final_suspend() 返回 std::suspend_always，否则生命周期管理将变得过于复杂。

然而，在撰写本章时 (2022 年 3 月)，只有 Clang 编译器为协程提供了相应的分配省略。[Visual C++ 提供了一个优化选项`/await:heapelide`，但目前似乎关闭了。]

例如，考虑以下协程:

```
CoroTask coro(int max)
{
  for (int val = 1; val <= max; ++val) {
    std::cout << "coro(" << max << "): " << val << '\n';
    co_await std::suspend_always{};
  }
}

CoroTask coroStr(int max, std::string s)
{
  for (int val = 1; val <= max; ++val) {
    std::cout << "coroStr(" << max << ", " << s << "): " << '\n';
    co_await std::suspend_always{};
  }
}
```

其不同之处在于第二个协程的附加字符串参数。假设只是将一些临时对象传递给每个参数:

```
coro(3); // create and destroy temporary coroutine
coroStr(3, "hello"); // create and destroy temporary coroutine
```

若没有优化和跟踪分配，可能会得到如下输出:

```
::new #1 (36 Bytes) => 0x8002ccb8
::delete (no size) at 0x8002ccb8

::new #2 (60 Bytes) => 0x8004cd28
::delete (no size) at 0x8004cd28
```

这里，第二个协程需要额外的字符串参数 24 个字节。

有关使用 coro/tracknews.hpp 查看何时分配或释放堆内存的完整示例，请参阅 coro/coromem.cpp。

15.6.2 避免堆内存分配

协程的 promise 类型允许开发者更改为协程分配内存的方式，只需要提供以下成员:

- `void* operator new(std::size_t sz)`
- `void operator delete(void* ptr, std::size_t sz)`

确保没有使用堆内存

首先，可以使用 new() 和 delete() 操作符来确定协程是否在堆上分配了内存。简单地声明 new 操作符而不实现:

```
class CoroTask {
  ...
  public:
  struct promise_type {
    ...
    // find out whether heap memory is allocated:
    void* operator new(std::size_t sz); // declared, but not implemented
  };
  ...
};
```

让协程不使用堆内存

这些操作符的另一个用途是改变协程分配内存的方式。例如，可以将对堆内存的调用，映射到堆栈上或程序的数据段中已经分配的一些内存。

下面是一个具体的例子:

coro/corotaskpmr.hpp

```
#include <coroutine>
#include <exception> // for terminate()
#include <cstddef> // for std::byte
#include <array>
```

```cpp
#include <memory_resource>

// coroutine interface to deal with a simple task
// - providing resume() to resume it
class [[nodiscard]] CoroTaskPmr {
  // provide 200k bytes as memory for all coroutines:
  inline static std::array<std::byte, 200'000> buf;
  inline static std::pmr::monotonic_buffer_resource
    monobuf{buf.data(), buf.size(), std::pmr::null_memory_resource()};
  inline static std::pmr::synchronized_pool_resource mempool{&monobuf};

public:
  struct promise_type;
  using CoroHdl = std::coroutine_handle<promise_type>;
private:
  CoroHdl hdl; // native coroutine handle

public:
  struct promise_type {
    auto get_return_object() { // init and return the coroutine interface
      return CoroTaskPmr{CoroHdl::from_promise(*this)};
    }
    auto initial_suspend() { // initial suspend point
      return std::suspend_always{}; // - suspend immediately
    }
    void unhandled_exception() { // deal with exceptions
      std::terminate(); // - terminate the program
    }
    void return_void() { // deal with the end or co_return;
    }
    auto final_suspend() noexcept { // final suspend point
      return std::suspend_always{}; // - suspend immediately
    }

    // define the way memory is allocated:
    void* operator new(std::size_t sz) {
      return mempool.allocate(sz);
    }
    void operator delete(void* ptr, std::size_t sz) {
      mempool.deallocate(ptr, sz);
    }
  };
  // constructor and destructor:
  CoroTaskPmr(auto h) : hdl{h} { }
  ~CoroTaskPmr() { if (hdl) hdl.destroy(); }

  // don't copy or move:
  CoroTaskPmr(const CoroTaskPmr&) = delete;
  CoroTaskPmr& operator=(const CoroTaskPmr&) = delete;
```

```
54
55    // API to resume the coroutine
56    // - returns whether there is still something to process
57    bool resume() const {
58      if (!hdl || hdl.done()) {
59        return false; // nothing (more) to process
60      }
61      hdl.resume(); // RESUME (blocks until suspended again or end)
62      return !hdl.done();
63    }
64  };
```

这里，使用多态内存资源，这是 C++17 引入的一个特性，通过提供标准化的内存池来简化内存管理。本例中，我们将 200kb 的数据段传递给内存池 monobuf，使用 null_memory_resource() 作为回退确保在内存不足时抛出 std::bad_alloc 异常。在最上面，放置了同步内存池 mempool，其用很少的碎片来管理这个缓冲区:[详细信息请参阅我的书《C++17——完整指南》]。

```
1   class [[nodiscard]] CoroTaskPmr {
2     // provide 200k bytes as memory for all coroutines:
3     inline static std::array<std::byte, 200'000> buf;
4     inline static std::pmr::monotonic_buffer_resource
5       monobuf{buf.data(), buf.size(), std::pmr::null_memory_resource()};
6     inline static std::pmr::synchronized_pool_resource mempool{&monobuf};
7     ...
8   public:
9     struct promise_type {
10      ...
11        // define the way memory is allocated:
12      void* operator new(std::size_t sz) {
13        return mempool.allocate(sz);
14      }
15      void operator delete(void* ptr, std::size_t sz) {
16        mempool.deallocate(ptr, sz);
17      }
18    };
19    ...
20  };
```

可以像往常一样使用这个协程接口:

coro/coromempmr.cpp

```
1   #include <iostream>
2   #include <string>
3   #include "corotaskpmr.hpp"
4   #include "tracknew.hpp"
5
6   CoroTaskPmr coro(int max)
7   {
```

```
  for (int val = 1; val <= max; ++val) {
    std::cout << "    coro(" << max << "): " << val << '\n';
    co_await std::suspend_always{};
  }
}

CoroTaskPmr coroStr(int max, std::string s)
{
  for (int val = 1; val <= max; ++val) {
    std::cout << "    coroStr(" << max << ", " << s << "): " << '\n';
    co_await std::suspend_always{};
  }
}

int main()
{
  TrackNew::trace();
  TrackNew::reset();
  coro(3); // initialize temporary coroutine
  coroStr(3, "hello"); // initialize temporary coroutine

  auto coroTask = coro(3); // initialize coroutine
  std::cout << "coro() started\n";
  while (coroTask.resume()) { // RESUME
    std::cout << "coro() suspended\n";
  }
  std::cout << "coro() done\n";
}
```

不再为这些协程分配堆内存。

还可以提供一个特定的操作符 new，在 size 参数之后接受协程参数。例如，对于接受 int 型和 string 型参数的协程，可以提供:

```
class CoroTaskPmr {
  public:
  struct promise_type {
    ...
    void* operator new(std::size_t sz, int, const std::string&) {
      return mempool.allocate(sz);
    }
  };
  ...
};
```

15.6.3 get_return_object_on_allocation_failure()

若 promise 有一个静态成员 get_return_object_on_allocation_failure()，则假定内存分配从不抛出。默认情况下，其效果是使用 new 操作符

```
::operator new(std::size_t sz, std::nothrow_t)
```

这种情况下，用户定义的操作符 new 必须为 noexcept，失败时返回 nullptr。

可以使用该函数来实现变通方法，例如创建不引用协程的协程句柄:

```
class CoroTask {
  ...
  public:
  struct promise_type {
    ...
    static auto get_return_object_on_allocation_failure() {
      return CoroTask{nullptr};
    }
  };
  ...
};
```

15.7. co_await 和 awaiter 的详情

之前已经介绍了 awaiter，让我们更详细的对其进行了解。

15.7.1 awaiter 接口的详细信息

“awaiter 的特殊成员函数”表再次列出了 awaiter 必须提供的关键操作。

操作	效果
Constructor	初始化 awaiter
await_ready()	是否 (当前) 禁用挂起
await_suspend(awaitHdl)	处理挂起
await_resume()	处理恢复

表 15.3 awaiter 的特殊成员函数

详细看看 awaiter 的接口:

- **构造函数**

  构造函数允许协程 (或任何创建 awaiter 的地方) 向 awaiter 传递参数，这些参数可用于影响或更改侍者处理挂起和恢复的方式。协程优先级请求的 awaiter 就是一个例子。

- **bool await_ready()**

  在调用此函数的协程挂起前调用。

  可以用来 (暂时) 完全关闭挂起。若它返回 true，就“准备好”立即从请求返回挂起并继续执行协程，而不挂起它。

  通常，这个函数只返回 false(“不，不要避免/阻止任何挂起”)，但可能会有条件地产生 true(例如，若挂起依赖于可用的某些数据)。

  通过其返回类型，await_suspend() 也可以发出不接受协程挂起的信号 (请注意，true 和 false 在这里具有相反的含义: 通过在 await_suspend() 中返回 true，接受挂起)。不使用 await_ready() 接受挂起的好处是，程序完全节省了启动协程挂起的成本。

  在这个函数内部，调用它的协程还没有挂起。不要在这里 (间接) 调用 resume() 或 destroy()，甚至可以在这里调用更复杂的业务逻辑，只要能够确保逻辑不会为这里挂起的协程调用 resume() 或 destroy()。

- **auto await_suspend(awaitHdl)**

  在调用此函数的协程挂起后立即调用。awaitHdl 是请求挂起的协程 (带有 co_await)，其具有等待协程句柄的类型 std::coroutine_handle<PromiseType>。

  可以指定下一步要做什么，包括立即恢复挂起的协程或等待的协程，并安排另一个延迟恢复，可以使用特殊的返回类型 (这将在下面讨论)。

  甚至可以在这里销毁协程，但必须确保协程没有在其他任何地方使用 (例如：调用协程接口中的 done())。

- **auto await_resume()**

  当调用此函数的协程在成功挂起后，恢复时调用 (即，若协程句柄调用 resume())。

  await_resume() 的返回类型是导致暂停的 co_await 或 co_yield 表达式产生的值的类型。若不是空的，协程的上下文可以返回一个值给恢复的协程。

await_suspend() 是这里的关键函数，其参数和返回值可以有如下变化:

- await_suspend() 的参数可以是:
  - 使用协程句柄的类型:

```
std::coroutine_handle<PrmType>
```

  - 使用可用于所有协程句柄的基本类型:

```
std::coroutine_handle<void> (or just std::coroutine_handle<>)
```

    这时，无法访问 promise。
  - 可以使用 auto 让编译器确定类型。
- await_suspend() 的返回类型可以是:
  - void，在执行 await_suspend() 中的语句后继续执行挂起，并返回到协程的调用者。
  - bool，来表明暂停是否真的应该发生，false 表示“不再挂起”(这与 await_ready() 的布尔返回值相反)。
  - std::coroutine_handle<> 用于恢复另一个协程。

    await_suspend() 的这种使用称为对称传输 (symmetric transfer)，稍后将详细描述。

    这时，可以使用 noop 协程来发出不恢复任何协程的信号 (与函数返回 false 的方式相同)。

此外，请注意以下事项:

- 成员函数通常是 const，除非 awaiter 有一个可修改的成员 (比如将协程句柄存储在 await_suspend() 中，以便在恢复时可用)。
- 成员函数通常是 noexcept(这是允许在 final_suspend() 中使用的必要条件)。
- 成员函数可以是 constexpr。

15.7.2 让 co_await 更新正在运行的协程

因为 awaiter 可以在挂起时执行代码，所以可以使用它们来改变处理协程的系统的行为。让我们看一个例子，让协程改变其优先级。

假设在调度器中管理所有的协程，并使用 co_await 更改其优先级。为此，首先定义一个默认优先级，并为 co_await 定义一些参数来改变优先级:

```
int CoroPrioDefVal = 10;
enum class CoroPrioRequest {same, less, more, def};
```

协程代码可能如下所示:

coro/coroprio.hpp

```
#include "coroprioshed.hpp"
#include <iostream>
CoroPrioTask coro(int max)
{
  std::cout << " coro(" << max << ")\n";
  for (int val = 1; val <= max; ++val) {
    std::cout << " coro(" << max << "): " << val << '\n';
    co_await CoroPrio{CoroPrioRequest::less}; // SUSPEND with lower prio
  }
  std::cout << " end coro(" << max << ")\n";
}
```

通过使用特殊的协程接口 CoroPrioTask，协程可以使用类型为 CoroPrio 的特殊 awaiter，允许协程传递更改优先级的请求。在挂起时，可以用它来改变请求的当前优先级:

```
co_await CoroPrio{CoroPrioRequest::less}; // SUSPEND with lower prio
```

为此，主程序创建一个调度器并调用 start() 将每个协程传递给调度器:

coro/coroprio.cpp

```
#include "coroprio.hpp"
#include <iostream>

int main()
{
  std::cout << "start main()\n";
```

```
  CoroPrioScheduler sched;

  std::cout << "schedule coroutines\n";
  sched.start(coro(5));
  sched.start(coro(1));
  sched.start(coro(4));

  std::cout << "loop until all are processed\n";
  while (sched.resumeNext()) {
  }
  std::cout << "end main()\n";
}
```

类 CoroPrioScheduler 和 CoroPrioTask 交互如下:

- 调度器按优先级顺序存储所有协程，并提供成员来存储新协程、恢复下一个协程和更改协程的优先级:

```
class CoroPrioScheduler
{
  std::multimap<int, CoroPrioTask> tasks; // all tasks sorted by priority
  ...
  public:
  void start(CoroPrioTask&& task);
  bool resumeNext();
  bool changePrio(CoroPrioTask::CoroHdl hdl, CoroPrioRequest pr);
};
```

- 调度器的成员函数 start() 将传递的协程存储在列表中，并将调度器存储在每个任务的 promise 中:

```
class CoroPrioScheduler
{
  ...
  public:
  void start(CoroPrioTask&& task) {
    // store scheduler in coroutine state:
    task.hdl.promise().schedPtr = this;
    // schedule coroutine with a default priority:
    tasks.emplace(CoroPrioDefVal, std::move(task));
  }
  ...
};
```

- 类 CoroPrioTask 赋予类 CoroPrioScheduler 对句柄的访问权，在 promise 类型中提供指向调度程序的成员，并允许协程使用 CoroPrioRequest 的 co_await:

```
class CoroPrioTask {
  ...
  friend class CoroPrioScheduler; // give access to the handle
```

```
  struct promise_type {
    ...
    CoroPrioScheduler* schedPtr = nullptr; // each task knows its scheduler:
    auto await_transform(CoroPrioRequest); // deal with co_await a
↪   CoroPrioRequest
  };
  ...
}
```

对于 start()，还可以以提供移动语义。

协程和调度器之间的接口是 CoroPrio 类型的 awaiter，其构造函数接受优先级请求，并在挂起时 (当我们获得协程句柄时) 将其存储在 promise 中，所以构造函数将请求存储在 await_suspend() 中:

```
class CoroPrio {
private:
  CoroPrioRequest prioRequest;
public:
  CoroPrio(CoroPrioRequest pr)
  : prioRequest{pr} { // deal with co_await a CoroPrioRequest
  }
  ...
  void await_suspend(CoroPrioTask::CoroHdl h) noexcept {
    h.promise().schedPtr->changePrio(h, prioRequest);
  }
  ...
};
```

其余部分只是通常的协同程序接口的样板代码加上优先级处理。完整的代码请参见 coro/coroprioschedule.hpp。

主程序调度三个协程并循环，直到所有协程处理完:

```
sched.start(coro(5));
sched.start(coro(1));
sched.start(coro(4));

while (sched.resumeNext()) {
}
```

输出结果如下所示:

```
start main()
schedule coroutines
loop until all are processed
  coro(5)
  coro(5): 1
  coro(1)
  coro(1): 1
```

```
  coro(4)
  coro(4): 1
  coro(5): 2
  end coro(1)
  coro(4): 2
  coro(5): 3
  coro(4): 3
  coro(5): 4
  coro(4): 4
  coro(5): 5
  end coro(4)
  end coro(5)
end main()
```

最初，启动的三个协程具有相同的优先级。由于协程的优先级在每次挂起时都会降低，因此其他协程会连续运行，直到第一个协程再次具有最高优先级。

15.7.3 具有等待器的对称传输

await_suspend() 的返回类型可以是协程句柄，通过立即恢复返回的协程来挂起协程，这种技术称为对称传输。

引入对称传输是为了提高性能并避免协程调用其他协程时堆栈溢出。通常，当使用 resume() 恢复协程时，程序需要为新协程提供一个新的堆栈帧。若要在 await_suspend() 中或在它被调用之后恢复协程，是要付出代价的。通过返回要调用的协程，当前协程的堆栈帧只是替换它的协程，因此不需要新的堆栈帧。

用协程实现对称传输

这种技术的典型应用是通过使用处理协程的 final await 来实现的。[这项技术在 Lewis Baker 和 Michael Eiler 的文章和电子邮件中得到了极大的帮助。http://lewissbaker.github.io/2020/05/11/understanding_symmetric_transfer特别提供了详细的动机和解释。]

假设有一个协程结束了，然后应该继续使用另一个后续协程，而不将控制权交还给调用者，会遇到以下问题: 在 final_suspend() 中，协程尚未处于挂起状态。必须等待，直到 await_suspend() 调用，以返回可等待对象来处理恢复。这可能看起来如下代码所示:

```
1  class CoroTask
2  {
3  public:
4    struct promise_type;
5    using CoroHdl = std::coroutine_handle<promise_type>;
6  private:
7    CoroHdl hdl; // native coroutine handle
8  public:
9    struct promise_type {
10     std::coroutine_handle<> contHdl = nullptr; // continuation (if there is one)
```

```
    ...
    auto final_suspend() noexcept {
      // the coroutine is not suspended yet, use awaiter for continuation
      return FinalAwaiter{};
    }
  };
  ...
};
```

返回一个 awaiter，可用于在协程暂停后处理协程。这里，返回的 awaiter 的类型是 FinalAwaiter，可能如下所示:

```
struct FinalAwaiter {
  bool await_ready() noexcept {
    return false;
  }
  std::coroutine_handle<> await_suspend(CoroTask::CoroHdl h) noexcept {
    // the coroutine is now suspended at the final suspend point
    // - resume its continuation if there is one
    if (h.promise().contHdl) {
      return h.promise().contHdl; // return the next coro to resume
    }
    else {
      return std::noop_coroutine(); // no next coro => return to caller
    }
  }
  void await_resume() noexcept {
  }
};
```

因为 await_suspend() 返回协程句柄，所以在挂起时自动恢复返回的协程，函数 std::noop_coroutine() 发出不恢复其他协程的信号，所以挂起的协程会返回给调用者。

std::noop_coroutine() 会返回一个 sstd::noop_coroutine_handle，是 std::coroutine_handle<std::noop_coroutine_promise> 的别名类型。该类型的协程在调用 resume() 或 destroy() 时不起作用，在调用 address() 时返回 nullptr，并且在调用 done() 时始终返回 false。

std::noop_coroutine() 及其返回类型是为 await_suspend() 可能可选地返回要继续的协程的情况提供的。通过返回类型 std::coroutine_handle<>，await_suspend() 可以返回一个 noop 协程，以表示不自动恢复另一个协程。

注意，std::noop_coroutine() 的两个不同的返回值并不一定相等。

因此，下面的代码不可移植:

```
std::coroutine_handle<> coro = std::noop_coroutine();
...
if (coro == std::noop_coroutine()) { // OOPS: does not check whether coro has initial
↪  value
  ...
```

```
  return coro;
}
```

应该使用 nullptr:

```
std::coroutine_handle<> coro = nullptr;
...
if (coro) { // OK (checks whether coro has initial value)
  ...
  return std::noop_coroutine();
}
```

处理线程池中的协程演示了这种技术。

15.8. 处理 co_await 的其他方法

到目前为止，我们只给 co_await 传递了 awaiter。例如，通过使用标准的 awaiter:

```
co_await std::suspend_always{};
```

或者，另一个例子，co_await 接受用户定义的 awaiter:

```
co_await CoroPrio{CoroPrioRequest::less}; // SUSPEND with lower prio
```

然而，co_await expr 是一个操作符，可以作为更大表达式的一部分，并接受不具有 awaiter 类型的值。例如:

```
co_await 42;
co_await (x + y);
```

co_await 操作符与 sizeof 或 new 具有相同的优先级。由于这个原因，不能跳过上面例子中的圆括号而不改变其含义。该声明

```
co_await x + y;
```

将按以下方式进行计算:

```
(co_await x) + y;
```

注意，x + y，或者只是 x，在这里不一定是一个附加项。co_await 需要一个 awaitable 对象，而 awaiter 只是一种 (常规的) 实现。事实上，co_await 接受任何类型的值，只要有一个到 awaiter 的 API 的映射。对于映射，C++ 标准提供了两种方法:

- promise 的成员函数 await_transform()
- co_await() 操作符

这两种方法都允许协程将任何类型的值传递给 co_await，并隐式或间接地指定一个 awaiter。

15.8.1 await_transform()

若 co_await 表达式出现在协程中，编译器首先查找是否存在由协程的 promise 提供的成员函数 await_transform()。若是这样，则调用 await_transform() 并产生一个等待器，然后使用该等待器挂起协程。

这意味着:

```
class CoroTask {
  struct promise_type {
    ...
    auto await_transform(int val) {
      return MyAwaiter{val};
    }
  };
  ...
};

CoroTask coro()
{
  co_await 42;
}
```

与如下代码具有相同的效果:

```
class CoroTask {
  ...
};

CoroTask coro()
{
  co_await MyAwaiter{42};
}
```

也可以使用它来启用协程在使用 (标准)awaiter 之前向 promise 传递一个值:

```
class CoroTask {
  struct promise_type {
    ...
    auto await_transform(int val) {
      ... // process val
      return std::suspend_always{};
    }
  };
  ...
};

CoroTask coro()
{
```

```
  co_await 42; // let 42 be processed by the promise and suspend
}
```

使用值让 co_await 更新正在运行的协程

还记得使用 awaiter 来更改协程优先级的示例吗? 在这里，使用 CoroPrio 类型的 awaiter 使协程能够请求新的优先级:

```
co_await CoroPrio{CoroPrioRequest::less}; // SUSPEND with lower prio
```

也可以只允许传递新的优先级:

```
co_await CoroPrioRequest::less; // SUSPEND with lower prio
```

只需要让协程接口 promise 为这种类型的值提供 await_transform() 成员函数:

```
class CoroPrioTask {
public:
  struct promise_type;
  using CoroHdl = std::coroutine_handle<promise_type>;
private:
  CoroHdl hdl; // native coroutine handle
  friend class CoroPrioScheduler; // give access to the handle
public:
  struct promise_type {
    CoroPrioScheduler* schedPtr = nullptr; // each task knows its scheduler:
    ...
    auto await_transform(CoroPrioRequest); // deal with co_await CoroPrioRequest
  };
  ...
};
```

await_transform() 的实现如下:

```
inline auto CoroPrioTask::promise_type::await_transform(CoroPrioRequest pr) {
  auto hdl = CoroPrioTask::CoroHdl::from_promise(*this);
  schedPtr->changePrio(hdl, pr);
  return std::suspend_always{};
}
```

这里，再次使用协程句柄 from_promise() 中的静态成员函数来获取句柄，因为 changePrio() 需要该句柄作为其第一个参数。

这样的话，就可以跳过整个 awaiter CoroPrio 了。完整代码请参见 coro/coroprioscheduler2.hpp(以及 coro/coroprio2.cpp 和 coro/coroprio2.hpp)。

让 co_await 像 co_yield 一样运行

看一下 co_yield 的例子。要处理要产生的值，并做了以下的操作:

```
struct promise_type {
  int coroValue = 0; // last value from co_yield
  auto yield_value(int val) { // reaction to co_yield
    coroValue = val; // - store value locally
    return std::suspend_always{}; // - suspend coroutine
  }
  ...
};

co_yield val; // calls yield_value(val) on promise
```

可以用下面的方法得到同样的效果:

```
 can get the same effect with the following:
struct promise_type {
  int coroValue = 0; // last value from co_yield
  auto await_transform(int val) {
    coroValue = val; // - store value locally
    return std::suspend_always{};
  }
  ...
};

co_await val; // calls await_transform(val) on promise
```

事实上,

```
co_yield val;
```

等于

```
co_await prm.yield_value(val);
```

其中 prm 是封闭协程的 promise。

15.8.2 co_await() 操作符

让 co_await 处理 (几乎) 任何类型的值的另一个选择是为该类型实现操作符 co_await()，所以必须传递一个类的值。

假设已经为某些类型 MyType 实现了操作符 co_await():

```
class MyType {
  auto operator co_await() {
    return std::suspend_always{};
  }
};
```

然后，对该类型的对象调用 co_await:

```
CoroTask coro()
{
  ...
  co_await MyType{};
}
```

并使用返回的 awaiter 来处理暂停。这里，操作符 co_await() 产生 std::suspend_always，则可得到:

```
CoroTask coro()
{
  ...
  co_await std::suspend_always{};
}
```

也可以将参数传递给 MyType{}，并将其传递给一个 awaiter，以便将值传递给挂起的协程，立即或稍后恢复。

另一个例子是将协程传递给 co_await，然后可以调用，包括实现准备工作，例如：在线程池中进行更改线程或协程调度。

15.9. 协程的并发使用

原则上，将操作数传递给 co_await 可以执行任何代码。可以跳转到一个非常不同的上下文，或者执行一些操作，并在得到结果后继续执行。即使其他操作在不同的线程中运行，也不需要同步机制。

本节提供一个基本示例来进行演示，其一部分是处理协程的简单线程池，使用 std::jthread 和 C++20 的一些新的并发特性。

15.9.1 co_await 协程

假设有以下相互调用的协程:

coro/coroasync.hpp

```
#include "coropool.hpp"
#include <iostream>
#include <syncstream> // for std::osyncstream
```

```
4
5   inline auto syncOut(std::ostream& strm = std::cout) {
6     return std::osyncstream{strm};
7   }
8
9   CoroPoolTask print(std::string id, std::string msg)
10  {
11    syncOut() << "       > " << id << " print: " << msg
12          << "       on thread: " << std::this_thread::get_id() << std::endl;
13    co_return; // make it a coroutine
14  }
15
16  CoroPoolTask runAsync(std::string id)
17  {
18    syncOut() << "===== " << id << " start "
19          << "   on thread: " << std::this_thread::get_id() << std::endl;
20
21    co_await print(id + "a", "start");
22    syncOut() << "===== " << id << " resume "
23          << "   on thread " << std::this_thread::get_id() << std::endl;
24
25    co_await print(id + "b", "end ");
26    syncOut() << "===== " << id << " resume "
27          << "   on thread " << std::this_thread::get_id() << std::endl;
28
29    syncOut() << "===== " << id << " done" << std::endl;
30  }
```

这两个协程都使用 CoroPoolTask，一个在线程池中运行任务的协程接口。接口和池的实现将在后面讨论。

重要的部分是协程 runAsync() 使用 co_await 调用另一个协程 print():

```
1   CoroPoolTask runAsync(std::string id)
2   {
3     ...
4     co_await print( ... );
5     ...
6   }
```

这样做的效果是，协程 print() 将被安排在线程池中，在不同的线程中运行。此外，co_await 阻塞直到 print() 完成。

print() 需要一个 co_return 来确保它被编译器视为协程。若没有这个，就不会有任何 co_关键字，编译器 (假设这是一个普通的函数) 会抱怨有返回类型，但没有返回语句。

使用辅助函数 syncOut()，会产生一个 std::osyncstream，以确保来自不同线程的并发输出逐行同步。

假设按如下方式调用协程 runAsync():

coro/coroasync1.cpp

```cpp
#include "coroasync.hpp"
#include <iostream>

int main()
{
  // init pool of coroutine threads:
  syncOut() << "**** main() on thread " << std::this_thread::get_id()
        << std::endl;
  CoroPool pool{4};

  // start main coroutine and run it in coroutine pool:
  syncOut() << "runTask(runAsync(1))" << std::endl;
  CoroPoolTask t1 = runAsync("1");
  pool.runTask(std::move(t1));

  // wait until all coroutines are done:
  syncOut() << "\n**** waitUntilNoCoros()" << std::endl;
  pool.waitUntilNoCoros();

  syncOut() << "\n**** main() done" << std::endl;
}
```

首先使用 CoroPoolTask 接口为所有协程创建一个 CoroPool 类型的线程池:

```cpp
CoroPool pool{4};
```

然后，使用协程，惰性地启动协程并返回接口，并将接口交给池来控制协程:

```cpp
CoroPoolTask t1 = runAsync("1");
pool.runTask(std::move(t1));
```

为协程接口使用 (并且必须使用) 移动语义，因为 runTask() 需要传递一个右值，以便能够接管协程的所有权 (t1 不再拥有它)。

也可以用一条语句来实现:

```cpp
pool.runTask(runAsync("1"));
```

结束这个项目时，协程池会阻塞，直到所有调度的协程都处理完:

```cpp
pool.waitUntilNoCoros();
```

当运行这个程序时，会得到如下输出 (线程 id 不同):

```
**** main() on thread 0x80000008
runTask(runAsync(1))
```

```
**** waitUntilNoCoros()
===== 1 start on thread: 0x8002cd90
    > 1a print: start on thread: 0x8002ce68
===== 1 resume on thread 0x8002ce68
    > 1b print: end on thread: 0x8004d090
===== 1 resume on thread 0x8004d090
===== 1 done

**** main() done
```

这里使用不同的线程:

- 协程 runAsync() 在与 main() 不同的线程上启动。
- 调用的协程 print() 从第三个线程开始
- 第二次调用协程 print() 是从第四个线程开始的

然而,每次调用 print() 时,runAsync() 都会改变线程。通过使用 co_await,这些调用挂起 runAsync() 并在另一个线程上调用 print()(挂起调度池中调用的协程)。print() 结束时，调用协同程序 runAsync() 在 print() 运行的同一线程上恢复。

作为一种变体，还可以启动并调度四次协程 runAsync():

coro/coroasync2.cpp

```
#include "coroasync.hpp"
#include <iostream>

int main()
{
  // init pool of coroutine threads:
  syncOut() << "**** main() on thread " << std::this_thread::get_id()
        << std::endl;
  CoroPool pool{4};

  // start multiple coroutines and run them in coroutine pool:
  for (int i = 1; i <= 4; ++i) {
    syncOut() << "runTask(runAsync(" << i << "))" << std::endl;
    pool.runTask(runAsync(std::to_string(i)));
  }

  // wait until all coroutines are done:
  syncOut() << "\n**** waitUntilNoCoros()" << std::endl;
  pool.waitUntilNoCoros();

  syncOut() << "\n**** main() done" << std::endl;
}
```

这个程序可能有以下输出 (由于不同的平台使用不同的线程 id):

```
**** main() on thread 17308
runTask(runAsync(1))
```

```
runTask(runAsync(2))
runTask(runAsync(3))
runTask(runAsync(4))

**** waitUntilNoCoros()
===== 1 start on thread: 18016
===== 2 start on thread: 9004
===== 3 start on thread: 17008
===== 4 start on thread: 2816
    > 2a print: start on thread: 2816
    > 1a print: start on thread: 17008
===== 1 resume on thread 17008
    > 4a print: start on thread: 18016
===== 4 resume on thread 18016
    > 3a print: start on thread: 9004
===== 3 resume on thread 9004
===== 2 resume on thread 2816
    > 4b print: end on thread: 9004
    > 1b print: end on thread: 2816
===== 1 resume on thread 2816
===== 1 done
===== 4 resume on thread 9004
===== 4 done
    > 2b print: end on thread: 17008
===== 2 resume on thread 17008
===== 2 done
    > 3b print: end on thread: 18016
===== 3 resume on thread 18016
===== 3 done

**** main() done
```

输出现在变化更大，因为有使用下一个可用线程的并发协程。总的来说，其有相同的行为:

- 每当协程调用 print() 时，都会调度并可能在不同的线程上启动 (若没有其他线程可用，则可能是同一个线程)。
- 每当 print() 完成时，调用协同程序 runAsync() 直接在 print() 运行的同一个线程上恢复，所以每次调用 print() 时，runAsync() 都改变了运行线程。

15.9.2 协程任务的线程池

下面是协程接口 CoroPoolTask 和对应的线程池类 CoroPool 的实现:

coro/coropool.hpp

```
#include <iostream>
#include <list>
#include <utility> // for std::exchange()
#include <functional> // for std::function
```

```cpp
#include <coroutine>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <functional>

class CoroPool;

class [[nodiscard]] CoroPoolTask
{
  friend class CoroPool;
public:
  struct promise_type;
  using CoroHdl = std::coroutine_handle<promise_type>;
private:
  CoroHdl hdl;
public:
  struct promise_type {
    CoroPool* poolPtr = nullptr; // if not null, lifetime is controlled by pool
    CoroHdl contHdl = nullptr; // coro that awaits this coro

    CoroPoolTask get_return_object() noexcept {
      return CoroPoolTask{CoroHdl::from_promise(*this)};
    }
    auto initial_suspend() const noexcept { return std::suspend_always{}; }
    void unhandled_exception() noexcept { std::exit(1); }
    void return_void() noexcept { }

    auto final_suspend() const noexcept {
      struct FinalAwaiter {
        bool await_ready() const noexcept { return false; }
        std::coroutine_handle<> await_suspend(CoroHdl h) noexcept {
          if (h.promise().contHdl) {
            return h.promise().contHdl; // resume continuation
          }
          else {
            return std::noop_coroutine(); // no continuation
          }
        }
        void await_resume() noexcept { }
      };
      return FinalAwaiter{}; // AFTER suspended, resume continuation if there is one
    }
  };

  explicit CoroPoolTask(CoroHdl handle)
  : hdl{handle} {
  }
  ~CoroPoolTask() {
```

```
54      if (hdl && !hdl.promise().poolPtr) {
55        // task was not passed to pool:
56        hdl.destroy();
57      }
58    }
59    CoroPoolTask(const CoroPoolTask&) = delete;
60    CoroPoolTask& operator= (const CoroPoolTask&) = delete;
61    CoroPoolTask(CoroPoolTask&& t)
62      : hdl{t.hdl} {
63        t.hdl = nullptr;
64    }
65    CoroPoolTask& operator= (CoroPoolTask&&) = delete;
66
67    // Awaiter for: co_await task()
68    // - queues the new coro in the pool
69    // - sets the calling coro as continuation
70    struct CoAwaitAwaiter {
71      CoroHdl newHdl;
72      bool await_ready() const noexcept { return false; }
73      void await_suspend(CoroHdl awaitingHdl) noexcept; // see below
74      void await_resume() noexcept {}
75    };
76    auto operator co_await() noexcept {
77      return CoAwaitAwaiter{std::exchange(hdl, nullptr)}; // pool takes ownership of
    ↪ hdl
78    }
79  };
80
81  class CoroPool
82  {
83  private:
84    std::list<std::jthread> threads; // list of threads
85    std::list<CoroPoolTask::CoroHdl> coros; // queue of scheduled coros
86    std::mutex corosMx;
87    std::condition_variable_any corosCV;
88    std::atomic<int> numCoros = 0; // counter for all coros owned by the pool
89
90  public:
91    explicit CoroPool(int num) {
92      // start pool with num threads:
93      for (int i = 0; i < num; ++i) {
94        std::jthread worker_thread{[this](std::stop_token st) {
95            threadLoop(st);
96        }};
97        threads.push_back(std::move(worker_thread));
98      }
99    }
100
101   ~CoroPool() {
```

```
102     for (auto& t : threads) { // request stop for all threads
103       t.request_stop();
104     }
105     for (auto& t : threads) { // wait for end of all threads
106       t.join();
107     }
108     for (auto& c : coros) { // destroy remaining coros
109       c.destroy();
110     }
111   }
112
113   CoroPool(CoroPool&) = delete;
114   CoroPool& operator=(CoroPool&) = delete;
115
116   void runTask(CoroPoolTask&& coroTask) noexcept {
117     auto hdl = std::exchange(coroTask.hdl, nullptr); // pool takes ownership of hdl
118     if (coroTask.hdl.done()) {
119       coroTask.hdl.destroy(); // OOPS, a done() coroutine was passed
120     }
121     else {
122       schedule coroutine in the pool
123     }
124   }
125
126   // runCoro(): let pool run (and control lifetime of) coroutine
127   // called from:
128   // - pool.runTask(CoroPoolTask)
129   // - co_await task()
130   void runCoro(CoroPoolTask::CoroHdl coro) noexcept {
131     ++numCoros;
132     coro.promise().poolPtr = this; // disables destroy in CoroPoolTask
133     {
134       std::scoped_lock lock(corosMx);
135       coros.push_front(coro); // queue coro
136       corosCV.notify_one(); // and let one thread resume it
137     }
138   }
139
140   void threadLoop(std::stop_token st) {
141     while (!st.stop_requested()) {
142       // get next coro task from the queue:
143       CoroPoolTask::CoroHdl coro;
144       {
145         std::unique_lock lock(corosMx);
146         if (!corosCV.wait(lock, st, [&] {
147                         return !coros.empty();
148                       })) {
149           return; // stop requested
150         }
```

```
151         coro = coros.back();
152         coros.pop_back();
153       }
154
155       // resume it:
156       coro.resume(); // RESUME
157
158       // NOTE: The coro initially resumed on this thread might NOT be the coro
    ↪  finally called.
159       // If a main coro awaits a sub coro, then the thread that finally resumed the
    ↪  sub coro
160       // resumes the main coro as its continuation.
161       // => After this resumption, this coro and SOME continuations MIGHT be done
162       std::function<void(CoroPoolTask::CoroHdl)> destroyDone;
163       destroyDone = [&destroyDone, this](auto hdl) {
164               if (hdl && hdl.done()) {
165                 auto nextHdl = hdl.promise().contHdl;
166                 hdl.destroy(); // destroy handle done
167                 --numCoros; // adjust total number of coros
168                 destroyDone(nextHdl); // do it for all continuations done
169               }
170             };
171     destroyDone(coro); // recursively destroy coroutines done
172     numCoros.notify_all(); // wake up any waiting waitUntilNoCoros()
173     // sleep a little to force another thread to be used next:
174     std::this_thread::sleep_for(std::chrono::milliseconds{100});
175     }
176   }
177
178   void waitUntilNoCoros() {
179     int num = numCoros.load();
180     while (num > 0) {
181     numCoros.wait(num); // wait for notification that numCoros changed the value
182     num = numCoros.load();
183     }
184   }
185 };
186
187 // CoroPoolTask awaiter for: co_await task()
188 // - queues the new coro in the pool
189 // - sets the calling coro as continuation
190 void CoroPoolTask::CoAwaitAwaiter::await_suspend(CoroHdl awaitingHdl) noexcept
191 {
192   newHdl.promise().contHdl = awaitingHdl;
193   awaitingHdl.promise().poolPtr->runCoro(newHdl);
194 }
```

CoroPoolTask 和 CoroPool 两个类需要谨慎地使用:

- CoroPoolTask 用于将协程初始化为要调用的任务。任务应该在池中调度，然后由池控制它，直到销毁。
- CoroPool 实现了一个最小的线程池，可以运行调度的协程，还提供了一个非常小的 API
  - 阻塞，直到所有计划的协程完成
  - 关闭池 (由其析构函数自动完成)

详细地看一下代码。

类 CoroPoolTask

类 CoroPoolTask 提供了一个协程接口来运行具有以下特性的任务:

- 每个协程都知道有一个指向线程池的指针，这个指针在线程池控制协程时初始化:

```
class [[nodiscard]] CoroPoolTask
{
  ...
  struct promise_type {
    CoroPool* poolPtr = nullptr; // if not null, lifetime is controlled by pool
    ...
  };
  ...
};
```

- 每个协程都有一个可选持续的成员:

```
class [[nodiscard]] CoroPoolTask
{
  ...
  struct promise_type {
    ...
    CoroHdl contHdl = nullptr; // coro that awaits this coro
    ...
    auto final_suspend() const noexcept {
      struct FinalAwaiter {
        bool await_ready() const noexcept { return false; }
        std::coroutine_handle<> await_suspend(CoroHdl h) noexcept {
          if (h.promise().contHdl) {
            return h.promise().contHdl; // resume continuation
          }
          else {
            return std::noop_coroutine(); // no continuation
          }
        }
        void await_resume() noexcept {}
      };
      return FinalAwaiter{}; // AFTER suspended, resume continuation if there is
↪  one
    }
```

```
    };
    ...
};
```

- 每个协程都有一个 co_await() 运算符，则 co_await 任务 () 使用一个特殊的 awaiter，因此 co_await 在将传递的协程调度到池中，并以当前协程作为可持续成员后挂起当前协程。

下面是实现的操作符 co_await() 的工作原理:

```
class [[nodiscard]] CoroPoolTask
{
  ...
  struct CoAwaitAwaiter {
    CoroHdl newHdl;
    bool await_ready() const noexcept { return false; }
    void await_suspend(CoroHdl awaitingHdl) noexcept; // see below
    void await_resume() noexcept {}
  };
  auto operator co_await() noexcept {
    return CoAwaitAwaiter{std::exchange(hdl, nullptr)}; // pool takes ownership of
↪   hdl
  }
};

void CoroPoolTask::CoAwaitAwaiter::await_suspend(CoroHdl awaitingHdl) noexcept
{
  newHdl.promise().contHdl = awaitingHdl;
  awaitingHdl.promise().poolPtr->runCoro(newHdl);
}
```

当用 co_await 调用 print() 时:

```
CoroPoolTask runAsync(std::string id)
{
  ...
  co_await print( ... );
  ...
}
```

发生以下情况:

- print() 作为协程调用并初始化。
- 对于返回的 CoroPoolTask 类型的协程接口，调用操作符 co_await()。
- 操作符 co_await() 接受用于初始化 CoAwaitAwaiter 类型的 awaiter 的句柄，所以新的协程 print() 将成为该 await 而的成员 newwhdl。
- std::exchange() 可确保在初始化的协程接口中，句柄 HDL 变为 nullptr，这样析构函数就不会调用 destroy()。

- runAsync() 使用由操作符 co_await() 初始化的 awaiter 挂起，该操作符以协同程序 runAsync() 的句柄作为参数调用 await_suspend()。
- await_suspend() 中，将传递的挂起的 runAsync() 作为持续存储，并在池中调度 newHdl(print())，以便由其中一个线程恢复。

类 CoroPool

CoroPool 通过结合本文和本书其他部分介绍的一些特性来实现其最小线程池。

首先让看一下其成员:

- 与其他线程池一样，这个类有一个使用新线程类 std::jthread 的线程成员，这样我们就不必处理异常，可以发出停止信号:

```
std::list<std::jthread> threads; // list of threads
```

因为在初始化之后使用线程的唯一其他时刻是在 shutdown() 期间，所以不需要同步，其首先会连接所有线程 (信号停止之后)。不支持复制和移动池 (简单起见，可以很容易地提供移动支持)。

为了获得更好的性能，析构函数在开始 join() 之前首先请求所有线程停止。

- 然后，有一个具有相应同步成员的协程队列要处理:

```
std::list<CoroPoolTask::CoroHdl> coros; // queue of scheduled coros
std::mutex corosMx;
std::condition_variable_any corosCV;
```

- 为了避免过早停止池，池还跟踪池拥有的协程总数:

```
std::atomic<int> numCoros = 0; // counter for all coros owned by the pool
```

并提供成员函数 waitUntilNoCoros()。

成员函数 runCoro() 是调度协程恢复的关键函数，接受接口 CoroPoolTask 的协程句柄作为参数。有两种方法可以调度协程接口本身:

- 调用 runTask()
- 使用 co_await task()

这两种方法都将协程句柄移动到池中，这样池就有责任在不再使用该句柄时 destroy() 该句柄。

实现正确的时刻调用 destroy()(并调整协程的总数) 并不容易正确和安全地完成。协程应该最终挂起，这就排除了在任务的 final_suspend() 或 final awaiter 的 await_suspend() 中执行此操作，所以跟踪和销毁工作如下所示:

- 每次将协程句柄传递给池时，将增加协程的数量:

```
void runCoro(CoroPoolTask::CoroHdl coro) noexcept {
  ++numCoros;
  ...
}
```

- 线程的恢复完成后，将检查协程和可能的延续是否已经完成。根据任务的最后一个等待器的await_suspend()，协程帧会自动调用持续。

  因此，在每次恢复完成后，会递归地迭代所有的可持续对象以销毁所有已完成的协程:

```
std::function<void(CoroPoolTask::CoroHdl)> destroyDone;
destroyDone = [&destroyDone, this](auto hdl) {
          if (hdl && hdl.done()) {
            auto nextHdl = hdl.promise().contHdl;
            hdl.destroy(); // destroy handle done
            --numCoros; // adjust total number of coros
            destroyDone(nextHdl); // do it for all continuations done
          }
        };
destroyDone(coro); // recursively destroy coroutines done
```

  因为 Lambda 是递归使用的，所以必须将其转发声明为 std::function<>。

- 最后，用原子类型的新线程同步特性唤醒所有等待的 waitUntilNoCoros():

```
numCoros.notify_all(); // wake up any waiting waitUntilNoCoros()
...
void waitUntilNoCoros() {
  int num = numCoros.load();
  while (num > 0) {
    numCoros.wait(num); // wait for notification that numCoros changed the value
    num = numCoros.load();
  }
}
```

- 若池销毁了，也可在所有线程完成后销毁所有剩余的协程句柄。

同步等待异步协程

实际的协程池会更复杂，需要使用额外的技巧来使代码更健壮和安全。

例如，池可以提供一种调度任务并等待其结束的方法，对于 CoroPool 来说，看起来如下所示:

```
class CoroPool
{
  ...
  void syncWait(CoroPoolTask&& task) {
    std::binary_semaphore taskDone{0};
    auto makeWaitingTask = [&]() -> CoroPoolTask {
      co_await task;
      struct SignalDone {
        std::binary_semaphore& taskDoneRef;
        bool await_ready() { return false; }
        bool await_suspend(std::coroutine_handle<>) {
          taskDoneRef.release(); // signal task is done
          return false; // do not suspend at all
```

```
        }
        void await_resume() { }
      };
      co_await SignalDone{taskDone};
    };
    runTask(makeWaitingTask());
    taskDone.acquire();
  }
};
```

这里我们使用一个二值信号量，就可以发出传递任务结束的信号，并在另一个线程中等待它。

必须注意信号的确切内容，信号表明已经落后于执行了 co_await 调用的任务，所以也可以在 await_ready() 中发出信号:

```
bool await_ready() {
  taskDoneRef.release(); // signal task is done
  return true; // do not suspend at all
}
```

通常，必须考虑在 await_ready() 中，协程尚未挂起。因此，检查是否为 done()，甚至 destroy() 的信号都会导致致命的运行时错误。因为在这里可以使用并发代码，所以必须确保信号不会间接引起相应的调用。

15.9.3 C++20 之后的库将提供什么特性

在协程章节的这一节中，所看到的只是一个非常简单的代码示例，不够健壮、线程安全、完整或灵活，无法为协程的各种典型用例提供通用解决方案。

C++ 标准委员会正致力于在未来的库中提供更好的解决方案。

对于一个能够以安全和灵活的方式并发运行协程的程序，至少需要以下组件:

- 一种任务类型，允许将协程链接在一起
- 允许启动多个独立运行的协程，并在稍后汇入
- 比如 syncWait()，允许同步函数阻塞等待异步函数
- 允许多个协程在较少数量的线程上复用

类 CoroPool 将后三个方面或多或少地结合在一起，但更灵活的方法是使用可以以各种方式组合的单个构建块。此外，良好的线程安全性需要更好的设计和实现技术。

我们仍在学习和讨论如何做到最好。因此，即使在 C++23 中，也可能只有最低限度的支持 (若有的话)。

15.10. 协程的特征

所有示例中，协程接口都是由协程返回的。但也可以实现协程，以便它们使用在其他地方创建的接口。考虑以下示例 (第一个协程示例的修改版本):

```
void coro(int max, CoroTask&)
{
  std::cout << "CORO " << max << " start\n";

  for (int val = 1; val <= max; ++val) {
    // print next value:
    std::cout << "CORO " << val << '/' << max << '\n';

    co_await std::suspend_always{}; // SUSPEND
  }

  std::cout << "CORO " << max << " end\n";
}
```

协程将 CoroTask 类型的协程接口作为参数，但必须指定将参数用作协程接口。通过模板 std::coroutine_traits<> 的特化完成，模板参数必须指定签名 (返回类型和参数类型)，并且在内部，必须将类型成员 promise_type 映射到协程接口参数的类型成员:

```
template<>
struct std::coroutine_traits<void, int, CoroTask&>
{
  using promise_type = CoroTask::promise_type;
};
```

现在唯一需要确保的是，可以在没有协程的情况下创建协程接口，并在以后引用协程。为此，需要一个带有构造函数的 promise，该构造函数接受与协程相同的参数

```
class CoroTask {
  public:
  // required type for customization:
  struct promise_type {
    promise_type(int, CoroTask& ct) { // init passed coroutine interface
      ct.hdl = CoroHdl::from_promise(*this);
    }
    void get_return_object() { // nothing to do anymore
    }
    ...
  };

private:
  // handle to allocate state (can be private):
  using CoroHdl = std::coroutine_handle<promise_type>;
  CoroHdl hdl;

public:
  // constructor and destructor:
  CoroTask() : hdl{} { // enable coroutine interface without handle
```

```
  }
  ...
};
```

现在使用协程的代码可能如下所示:

```
CoroTask coroTask; // create coroutine interface without coroutine

// start coroutine:
coro(3, coroTask); // init and store coroutine in the interface created

// loop to resume the coroutine until it is done:
while (coroTask.resume()) { // resume
  std::this_thread::sleep_for(500ms);
}
```

可以在 coro/corotraits.cpp 中找到完整的示例。

promise 构造函数获取传递给协程的所有参数，由于 promise 通常只需要协程接口，可以将接口作为第一个参数传递，然后使用 auto&&…作为最后一个参数。

第 16 章 模块

本章介绍 C++20 新特性“模块”。模块提供了一种将来自多个文件的代码组合成一个逻辑实体 (模块、组件) 的方法。与类一样，数据封装有助于提供模块 API 的清晰定义。可确保模块代码不必多次编译，即使它放置在“头文件”中。

本章的撰写得到了 Daniela Engert 和 Hendrik Niemeyer 的大力帮助和支持，他们也分别在 2020 年会议 C++ 和 2021 年会议 ACCU 会议上对这个主题做了很好的介绍。

16.1. 用例子说明添加模块的动机

模块允许程序员为代码定义 API。代码可能由多个类、多个文件、几个函数和包括模板在内的各种辅助工具组成。通过关键字 export，可以指定导出的内容为模块的 API，该模块包装提供特定功能的所有代码，所以可以为在不同文件中实现组件，定义一个干净的 API。

让我们看几个简单的例子，其在一个文件中声明一个模块，然后在另一个文件中使用这个模块。

16.1.1 实现和导出模块

模块的 API 规范定义在其主接口 (正式名称为主模块接口单元) 中，每个模块只有一次:

modules/mod0.cppm

```
export module Square; // declare module Square

int square(int i);

export class Square {
private:
  int value;
public:
  Square(int i)
  : value{square(i)} {
  }
  int getValue() const {
    return value;
  }
};

export template<typename T>
Square toSquare(const T& x) {
  return Square{x};
}

int square(int i) {
  return i * i;
}
```

该文件使用了一个新的文件扩展名:.cppm。模块文件的扩展名还不清楚。稍后我们将讨论编译器对模块文件的处理。

主接口的关键是使用名称 Square 声明和导出模块的那行:

```
export module Square; // declare module Square
```

该名称仅用作导入模块的标识符，不会引入新的作用域或命名空间，模块导出的任何名称仍然在导出时所在的作用域中。

模块的名称可以包含句点，而句点在 C++ 中使用的任何其他类型的标识符中都是无效的，作为模块名称标识符的字符句点是有效的，并且没有特殊含义。例如:

```
export module Math.Square; // declare module “Math.Square”
```

除了使用句点具有视觉效果外，这除了用“MathDotSquare”命名模块之外没有其他效果。period 可以用来表示由组件或项目建立的模块之间的一些逻辑关系。使用它们不会产生句法或形式上的后果。

模块的公共 API 由使用关键字 export 显式导出的所有内容定义。本例中，导出了类 Square 和函数模板 toSquare<>():

```
export class Square {
  ...
};

export template<typename T>
Square toSquare(const T& x) {
  ...
}
```

其他所有内容都不能导出，也不能导入模块直接使用 (将在后面讨论如何可以访问未导出的模块符号，但不可见)，所以没有导出的函数 square() 不能导入该模块使用。

该文件看起来像一个头文件，但有以下区别:

- 模块声明。
- 符号、类型、函数 (甚至模板) 可以通过 export 导出。
- 不需要内联来定义函数。
- 不需要宏定义保护。

然而，模块文件不仅仅是一个改进的头文件。模块文件可以同时扮演头文件和源文件的角色，可以包含声明和定义。模块文件中，不必使用内联或预处理器保护来指定定义。当模块导出的实体在不同翻译单元导入时，不能违反同一定义规则。

每个模块必须有一个指定名称的主接口文件，模块的名称与模块内的任何符号都不冲突，也没有隐式地引入命名空间。模块可以具有其 (主要) 命名空间、类或函数的名称，模块名可能经常与导出符号的命名空间相匹配，但需要显式实现。

16.1.2 编译模块单元

模块文件可以同时具有声明和定义，所以可以看作是头文件和源文件的组合。可用它做两件事:

- **预编译声明** (包括所有泛型代码)，将声明转换为特定于编译器的格式
- **定义编译方式**，创建通常的目标文件

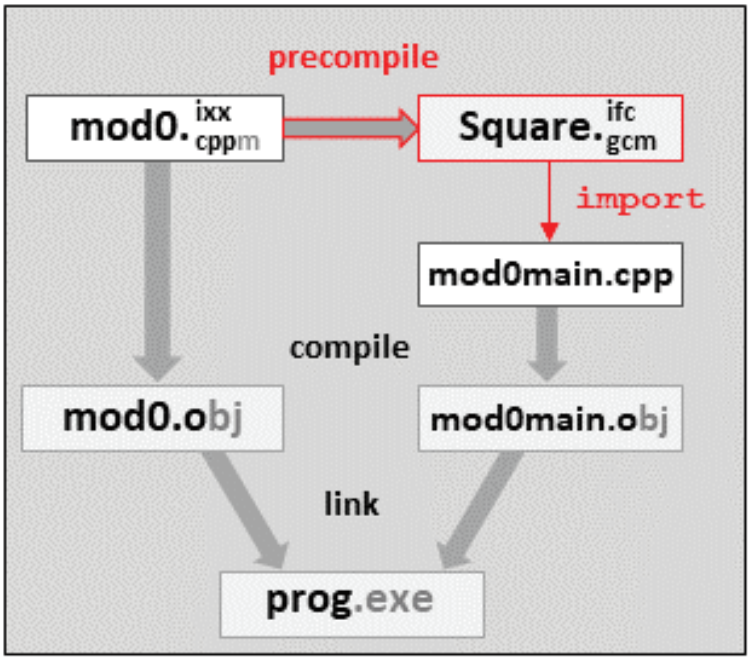

图 16.1 处理 C++ 的模块

假设有主模块接口 mod0，必须用两种方式处理它，如图 16.1 所示:

- 必须预编译 mod0.cppm，创建一个包含所有导出声明 (包括预编译的模板定义) 的预编译模块文件，由模块 Square 的名称标识，而不是源文件的名称。
- 必须编译 mod0.cppm，创建一个对象文件 mod0.o 或 mod0.obj 与汇编代码的所有定义，可以直接编译。

源模块文件不需要特定的文件扩展名，在这里使用.cppm。预编译模块文件也没有标准化后缀，这是由编译器决定的。通常，有以下情况:

- gcc/g++ 对预编译文件使用.gcm(并将它们放在 gcm.cache 子目录中)。
- Visual C++ 对预编译文件使用.ifc(并将它们放在本地目录中)。

我们将在后面详细讨论用于处理模块单元的文件后缀和选项。

要成功编译导入模块的源文件，需要模块的预编译工件可用，所以必须在预编译 mod0.cppm 前编译 mod0test.cpp。若没有遵循正确的顺序，可能会导入指定模块的非最新版本，所以不允许循环导入依赖关系。

与其他编程语言相比，C++ 不要求模块具有特殊的文件名或位于特殊目录中。C++ 文件都可以定义一个模块 (但只有一个)，模块的名称不必与文件名称或位置有关系。

当然，以某种方式保持文件名和模块名同步也很有意义，但这个决定最终取决于开发者的偏好，以及对所使用的配置管理和构建系统。

16.1.3 导入和使用模块

要在程序中使用模块的代码，必须以其名称导入该模块。下面是一个简单的程序示例，仅使用上面定义的模块 Square:

modules/mod0main.cpp

```
#include <iostream>

import Square; // import module “Square”

int main()
{
  Square x = toSquare(42);
  std::cout << x.getValue() << '\n';
}
```

有

```
import Square; // import module “Square”
```

从模块 Square 中导入所有导出的符号，则可以使用导出的类 Square 和函数模板 toSquare<>()。

使用未导出模块中的任何符号都会导致编译时错误:

```
import Square; // import module ” Square”

square(42) // ERROR: square() not exported
```

模块不会自动引入新的命名空间，在导出模块时所处的作用域中使用导出的模块符号。若希望将模块中的所有内容导出到其自己的命名空间中，则可以导出整个命名空间。

16.1.4 可及与可见

使用模块时，一个新的区别就出现了: 可达性与可见性。虽然可以间接地使用，但导出数据时，可能无法看到并直接使用模块的名称或符号。

当导出的 API 提供对未导出的类型的访问时，可能会出现可访问但不可见的符号。考虑下面的例子:

```
export module ModReach; // declare module ModReach

struct Data { // declare a type not exported
  int value;
};

export struct Customer { // declare an exported type
private:
  Data data;
```

```
public:
  Customer(int i)
  : data{i} {
  }
  Data getData() const { // yield a type not exported
    return data;
  }
};
```

导入该模块时，Data 类型不可见，因此不能直接使用:

```
import ModReach;
...

Data d{11}; // ERROR: type Data not exported
Customer c{42};
const Data& dr = c.getData(); // ERROR: type Data not exported
```

但类型 Data 是可访问的，因此可以间接使用:

```
import ModReach;
...

Customer c{42};
const auto& dr = c.getData(); // OK: type Data is used
auto d = c.getData(); // OK: d has type Data
std::cout << d.value << '\n'; // OK: type Data is used
```

甚至可以像下面这样声明一个 Data 类型对象:

```
decltype(std::declval<Customer>().getData()) d; // d has non-exported type Data
```

通过使用 std::declval<>()，对假定的 Customer 类型对象调用 getData()。若为 Customer 类型的对象调用 getData()，则使用 Data 类型声明它的返回类型。

私有模块段可以用来限制间接导出的类和函数的可达性。

稍后将详细讨论导出符号的可见性和可达性。

16.1.5 模块和命名空间

模块的符号导入到与导出时相同的作用域中。与其他一些编程语言不同，C++ 模块不会自动为模块引入命名空间。

因此，可以将模块中的所有内容，导出到具有其命名空间中。可以通过两种方式做到这:

- 命名空间中使用 export 指定要导出的组件:

```
export module Square; // declare module "Square"

namespace Square {
  int square(int i);

  export class Square {
    ...
  };

  export template<typename T>
  Square toSquare(const T& x) {
    ...
  }

  int square(int i) { // not exported
    ...
  }
}
```

- export 声明的命名空间中，指定想要导出的内容:

```
export module Square; // declare module "Square"

int square(int i);

export namespace Square {
  class Square {
    ...
  };

  template<typename T>
  Square toSquare(const T& x) {
    ...
  }
}

int square(int i) { // not exported
  ...
}
```

这两种情况下，模块都会导出类 Square::Square 和 Square::toSquare<>()(即使没有使用 export 标记，符号的命名空间也会导出)。

现在使用该模块的方式如下所示:

```
#include <iostream>

import Square; // import module "Square"

```

```
int main()
{
  Square::Square x = Square::toSquare(42);
  std::cout << v.getValue() << '\n';
}
```

16.2. 具有多个文件的模块

模块的目的是处理分布在多个文件上的大量代码。模块可用于包装由 2 个、10 个甚至 100 个文件组成的小型、中型和超大型组件的代码。这些文件甚至可能由多个开发者和团队提供和维护。

为了演示这种方法的可扩展性及其好处，现在来看一下如何使用多个文件来定义可以被其他代码使用/导入的模块。示例的代码大小仍然很小，通常不会将其分散到多个文件中。我们的目标是用非常简单的示例来演示这些特性。

16.2.1 模块单元

通常，模块由多个模块单元组成。模块单元是属于一个模块的翻译单元。

所有模块单元都必须以某种方式编译，只包含声明 (传统代码中的头文件)，也需要进行某种预编译。因此，这些文件总可转换成某种特定于平台的内部格式，以避免不得不一次又一次地 (预) 编译相同的代码。

除了主要的模块接口单元，C++ 还提供了另外三种单元类型来将模块的代码拆分为多个文件:

- 模块实现单元允许开发者在自己的文件中实现定义，这样就可以单独编译 (类似于传统的 C++ 源代码在.cpp 文件中)。
- 内部分区允许开发者在单独的文件中，提供仅在模块内可见的声明和定义。
- 接口分区甚至允许开发者将导出的模块 API 拆分为多个文件。

下一节将通过示例介绍这些额外的模块单元。

16.2.2 使用已实现的单元

第一个在多个文件中实现的模块示例演示了，如何分割定义 (例如函数实现)，以避免将它们放在一个文件中。这样做是为了能够分别编译定义。

可以通过使用模块实现 (正式名称是模块实现单元) 来完成，其处理方式与单独编译源文件类似。

让我们来看一个例子。

带有全局模块的主接口

首先需要定义导出内容的主接口:

modules/mod1/mod1.cppm

```
module; // start module unit with global module fragment

```

```
#include <string>
#include <vector>

export module Mod1; // module declaration

struct Order {
  int count;
  std::string name;
  double price;

  Order(int c, const std::string& n, double p)
  : count{c}, name{n}, price{p} {
  }
};

export class Customer {
private:
  std::string name;
  std::vector<Order> orders;
public:
  Customer(const std::string& n)
  : name{n} {
  }
  void buy(const std::string& ordername, double price) {
    orders.push_back(Order{1, ordername, price});
  }
  void buy(int num, const std::string& ordername, double price) {
    orders.push_back(Order{num, ordername, price});
  }
  double sumPrice() const;
  double averagePrice() const;
  void print() const;
};
```

这一次，模块从模块开始; 来表示我们有一个模块，并且可以在模块中使用一些预处理命令:

```
module; // start module unit with global module fragment

#include <iostream>
#include <string>
#include <vector>

export module Mod1; // module declaration
...
```

模块之间的区域; 模块可声明称为全局模块。可以使用它来放置预处理器命令，如 #define 和 #include，该区域中的内容都不会导出 (没有宏、没有声明、没有定义)。

用声明正式启动模块单元之前，不能做其他事情 (当然，注释除外):

```
export module mod1; // module declaration
```

这个模块中定义的是:

- 内部数据结构

```
struct Order {
  ...
};
```

此数据结构用于订单信息。每个信息保存有关订购了多少项、名称和价格的信息。构造函数确保初始化所有成员。

- 一个 customer 类，我们导出它:

```
export class Customer {
  ...
};
```

需要头文件和内部数据结构 Order 来定义 Customer 类，但由于不导出它们，导入该模块的代码就不能直接使用。

对于 Customer 类，只声明了成员函数 averagePrice()、sumPrice() 和 print()。这里，使用特性在模块实现单元中定义。

模块实现单元

一个模块可以有任意数量的实现单元。示例中，提供了其中的两个: 一个用于实现数值操作，另一个用于实现 I/O 操作。

数值运算的模块实现单元如下所示:

modules/mod1/mod1price.cpp

```
module Mod1; // implementation unit of module Mod1

double Customer::sumPrice() const
{
  double sum = 0.0;
  for (const Order& od : orders) {
    sum += od.count * od.price;
  }
  return sum;
}

double Customer::averagePrice() const
{
  if (orders.empty()) {
    return 0.0;
  }
```

```
  return sumPrice() / orders.size();
}
```

该文件是一个模块实现单元，以声明这是模块 Mod1 的文件开始:

```
module Mod1;
```

该声明导入了模块的主接口单元 (但没有其他内容)，所以 Order 和 Customer 类型的声明是已知的，可以直接提供它们的成员函数的实现。

注意，模块实现单元不导出任何东西。导出只允许在模块 (主接口或接口分区) 的接口文件中进行，这些文件是用 export module 声明 (记住，每个模块只允许一个主接口)。

同样，模块实现单元可以从全局模块开始，可以在 I/O 模块实现单元中看到:

modules/mod1/mod1io.cpp

```
module; // start module unit with global module fragment

#include <iostream>
#include <format>

module Mod1; // implementation unit of module Mod1

void Customer::print() const
{
  // print name:
  std::cout << name << ":\n";
  // print order entries:
  for (const auto& od : orders) {
    std::cout << std::format("{:3} {:14} {:6.2f} {:6.2f}\n",
                  od.count, od.name, od.price, od.count * od.price);
  }
  // print sum:
  std::cout << std::format("{:25} ------\n", ' ');
  std::cout << std::format("{:25} {:6.2f}\n", " Sum:", sumPrice());
}
```

这里，在实现单元中使用的头文件提供一个全局模块，<format> 是新格式库的头文件。

模块实现单元使用传统 C++ 翻译单元的文件扩展名 (大多数情况下是.cpp)，编译器就像处理其他非模块的 C++ 代码一样处理。

使用模块

使用该模块的代码如下所示:

modules/mod1/testmod1.cpp

```
#include <iostream>

```

```
import Mod1;

int main()
{
  Customer c1{"Kim"};

  c1.buy("table", 59.90);
  c1.buy(4, "chair", 9.20);

  c1.print();
  std::cout << " Average: " << c1.averagePrice() << '\n';
}
```

使用从主界面导出的 Customer 类来创建客户、下订单、输出带有所有订单的客户，以及输出订单的平均值。

该程序有以下输出:

```
Kim:
  1 table         59.90    59.90
  4 chair          9.20    36.80
                          ------
    Sum:                   96.70
Average: 48.35
```

导入模块的代码中使用 Order 类型的尝试都会导致编译时错误。

模块的使用并不取决于我们有多少实现单元。实现单元的数量之所以重要，只是因为链接器必须使用为其生成的目标文件。

16.2.3 内部分区

在前面的示例中，介绍了仅在模块内使用的数据结构 Order。看起来必须在主接口中声明，以使其对所有实现单元可用。当然，这在大型项目中是无法扩展的。使用内部分区，可以在单独的文件中声明和定义模块的内部类型和函数。分区还可以用于在单独的文件中定义导出接口的各个部分，将在后面讨论。

内部分区有时称为分区实现单元:C++20 标准中，其正式的名称为“模块分区的模块实现单元”，这听起来像是提供接口分区的实现，但并非如此。其和模块的内部头文件一样，可以提供声明和定义。

定义内部分区

使用内部分区，可以在自己的模块单元中定义本地类型 Order:

modules/mod2/mod2order.cppp

```
module; // start module unit with global module fragment

```

```
#include <string>

module Mod2:Order; // internal partition declaration

struct Order {
  int count;
  std::string name;
  double price;

  Order(int c, const std::string& n, double p)
    : count{c}, name{n}, price{p} {
  }
};
```

分区有模块名，然后是冒号，然后是分区名:

```
module Mod2:Order;
```

不支持 Mod2:Order:Main 的子分区。

还需要了解的是，该文件使用了另一个新的文件扩展名:.cppp，我们将在稍后查看其内容后再讨论。

主接口只能使用名称来导入这个分区:Order:

modules/mod2/mod2.cpp

```
module; // start module unit with global module fragment

#include <string>
#include <vector>

export module Mod2; // module declaration

import :Order; // import internal partition Order

export class Customer {
private:
  std::string name;
  std::vector<Order> orders;
public:
  Customer(const std::string& n)
  : name{n} {
  }
  void buy(const std::string& ordername, double price) {
    orders.push_back(Order{1, ordername, price});
  }
  void buy(int num, const std::string& ordername, double price) {
    orders.push_back(Order{num, ordername, price});
  }
  double sumPrice() const;
```

```
  double averagePrice() const;
};
```

主接口必须导入内部分区，因为它使用 Order 类型。通过导入，分区在模块的所有单元中都可用。若主接口不需要 Order 类型，也不导入内部分区，则所有需要 Order 类型的模块单元都必须直接导入内部分区。

分区只是模块的内部实现方面。对于代码的用户来说，代码是在主模块中、实现中还是在内部分区中都无关紧要，但不能导出内部分区中的代码。

16.2.4 分区的接口

还可以将模块的接口拆分为多个文件，可以声明接口分区，这些分区本身可以导出相应的内容。

若模块提供由不同开发者和/或团队维护的多个接口，接口分区特别有用。为简单起见，这里只使用当前的示例来演示，如何通过在单独的文件中只定义 Customer 接口来使用该特性。

为了只定义 Customer 接口，可以提供以下文件:

modules/mod3/mod3customer.cppm

```
module; // start module unit with global module fragment

#include <string>
#include <vector>

export module Mod3:Customer; // interface partition declaration

import :Order; // import internal partition to use Order

export class Customer {
private:
  std::string name;
  std::vector<Order> orders;
public:
  Customer(const std::string& n)
  : name{n} {
  }
  void buy(const std::string& ordername, double price) {
    orders.push_back(Order{1, ordername, price});
  }
  void buy(int num, const std::string& ordername, double price) {
    orders.push_back(Order{num, ordername, price});
  }
  double sumPrice() const;
  double averagePrice() const;
  void print() const;
};
```

分区或多或少是前主接口，但有一点不同:

- 作为一个分区，在模块名和冒号后面声明其名称:Mod3:Customer

像主接口一样:

- 导出这个模块分区:

```
export module Mod3:Customer;
```

- 使用新的文件扩展名.cppm，稍后将再讨论

主接口仍然是指定模块导出内容的唯一地方，但主模块可以将导出委托给接口分区。这样做的方法是将导入的接口分区作为一个整体直接导出:

modules/mod3/mod3.cppm

```
export module Mod3; // module declaration

export import :Customer; // import and export interface partition Customer
... // import and export other interface partitions
```

通过同时导入接口分区和导出接口分区 (两个关键字都要写)，主接口导出分区 Customer 的接口作为自己的接口:

```
export import :Customer; // import and export partition Customer
```

不允许导入接口分区，而不导出接口分区。

分区只是模块的内部实现方面，接口和实现是否在分区中提供并不重要，并且分区不会创建新的作用域。

对于 Customer 的成员函数的实现，将类的声明移动到分区中并不重要。作为模块 Mod3 的一部分，实现了类 Customer 的成员函数:

modules/mod3/mod3io.cppm

```
module; // start module unit with global module fragment

#include <iostream>
#include <vector>
#include <format>

module Mod3; // implementation unit of module Mod3

import :Order; // import internal partition to use Order

void Customer::print() const
{
  // print name:
  std::cout << name << ":\n";
  // print order entries:
  for (const Order& od : orders) {
    std::cout << std::format("{:3} {:14} {:6.2f} {:6.2f}\n",
                  od.count, od.name, od.price, od.count * od.price);
```

```
  }
  // print sum:
  std::cout << std::format("{:25} ------\n", ' ');
  std::cout << std::format("{:25} {:6.2f}\n", " Sum:", sumPrice());
}
```

这个实现单元中有一个不同之处: 因为主接口不再导入内部分区:Order。这个模块必须这样做，因为它使用了 Order 类型。

对于导入模块的代码，内部分发代码的方式也无关紧要，仍然可在全局作用域中导出 Customer 类:

modules/mod3/testmod3.cpp

```
#include <iostream>

import Mod3;

int main()
{
  Customer c1{"Kim"};

  c1.buy("table", 59.90);
  c1.buy(4, "chair", 9.20);

  c1.print();
  std::cout << " Average: " << c1.averagePrice() << '\n';
}
```

16.2.5 将模块拆分到不同文件的总结

本节中的示例演示了如何处理不断增加的代码大小的模块，以便拆分代码对“驯服野兽”有帮助，甚至是必要的:

- 模块实现单元允许项目将定义拆分为多个文件，这样源代码可以由不同的程序员维护，若局部情况发生变化，不必重新编译所有代码。
- 内部分区允许项目将模块局部声明和定义移到主接口之外。可以由主接口导入，也可以只由需要它们的模块单元导入。
- 接口分区允许项目在不同的文件中维护导出的接口。若导出的 API 变得如此之大，以至于有不同的文件 (以及团队) 来处理其中的一部分，这就是有意义的。

主接口将所有内容集合在一起，并指定导出给模块用户的内容 (通过直接导出符号或导出导入的接口分区)。

拥有的模块单元类型取决于 C++ 源文件中的模块声明 (可以在注释和预处理器命令的全局模块之后):

- export module name;

  主接口。对于每个模块，只能在 C++ 程序中存在一次。

- module name;
  仅提供定义 (可能使用局部声明) 的实现单元。想要多少提供多少。
- module name:partname;
  一个内部分区，声明和定义仅在模块内使用。可以有多个分区，但是对于每个 partname，只能有一个内部分区文件。
- export module name:partname;
  一个接口分区。可以有多个接口分区，但是对于每个 partname，只能有一个接口分区文件。

因为没有针对不同模块单元的标准后缀，所以工具必须解析 C++ 源文件的头部，以检测是否是模块单元，以及是哪种类型的模块单元。模块声明可能发生在注释和全局模块之后。请参阅http://github.com/josuttis/cppmodules中的 clmod.py，以获得一个 Python 脚本，该脚本演示处理模块时可能出现的情况。

16.3. 实践中处理模块

本节讨论在实践中使用模块的一些其他方面。

16.3.1 用不同的编译器处理模块文件

C++ 中，扩展是不标准化的。实践中，会使用不同的文件扩展名 (通常是.cpp 和.hpp，但也使用.cc、.cxx、.c、.hh、.hxx、.h，甚至是.h)。

所以也没有模块的标准后缀。更糟糕的是，甚至不同意是否有必要 (目前) 进行新的后缀。原则上，有两种判断方法:

- 编译器应该将各种模块文件视为普通的 C++ 源文件，并根据其内容找出如何处理方法。使用这种方法，所有文件的扩展名仍然是.cpp。gcc/g++ 遵循这个策略。
- 编译器对 (某些) 模块文件的处理方式不同，因为它们既可以是声明文件 (传统的头文件)，也可以是定义文件 (传统的源文件)。由于这个原因，使用不同的后缀非常有帮助，编译器甚至可能间接地要求使用不同的后缀，以避免对相同的扩展使用不同的命令行选项。Visual C++ 遵循这种方法。

实践中，不同的编译器会为模块文件推荐不同的文件扩展名 (.cppm、.ixx 和.cpp)，这就是为什么在实际使用模块仍然具有挑战性的原因之一。

我想了一会儿，尝试了一下，并与标准委员会的人讨论了当前的情况，但到目前为止，似乎没有令人信服的解决方案。在形式上，C++ 标准并没有标准化处理源代码的方式 (代码甚至可能没有存储在文件中)，所以没有一种简单的方法可以使用模块编写一个简单的可移植的第一个示例。

因此，我在这里提出一些建议，以便读者们可以在不同的平台上尝试使用模块:

为什么不同的文件扩展名似乎是必要的有多种原因:

- 编译器需要不同的命令行选项来处理不同类型的模块文件。
- 与头文件类似，必须向客户和第三方代码提供一些模块文件。
- 不同的模块文件创建不同的工件，可能必须处理这些工件 (例如，在删除生成的工件时)。

就个人而言，我还没有最终的决定和建议模块文件的文件扩展名。然而，鉴于目前的情况，我的建议如下:

- 对于接口文件 (主接口和接口分区)，使用文件扩展名.cppm。原因是:
  - 最好的自解释文件扩展名 (远好于 Visual C++ 目前推荐的.ixx)。
  - 这是 Clang 目前所需要的。
  - 也可以在 gcc 中使用。
  - Visual C++ 无论如何都需要特殊处理，除非使用扩展名.ixx。
- 对于模块实现文件 (但不是分区实现文件)，使用通常的文件扩展名.cpp。原因是:
  - 没有生成特殊的工件。
  - 不需要特殊的命令行选项。
- 对于内部分区文件 (分区实现文件)，使用文件扩展名.cppp。原因是:
  - Visual C++ 需要命令行选项/internalPartition 来存放这些文件。文件后缀不重要，所以必须使用特殊后缀，以便在不想解析文件内容的构建系统中有机会使用通用规则。
  - 也可以在 gcc 中使用。
  - 目前，Clang 根本不支持这些文件 (2021 年 9 月)

  微软处理内部分区的方式对模块的成功不利，我希望他们能尽快解决对特定后缀的需求。

因此，必须 (预) 编译模块文件如下:

- Visual C++:

  Visual C++ 需要特定的命令行扩展名，并且倾向使用与我建议的不同的文件扩展名.ixx。出于这个原因:
  - 编译一个接口文件 file.cppm 的命令行如下所示:

    ```
    cl /TP /interface /c file.cppm
    ```

    选项/TP 指定以下所有文件都包含 C++ 源代码。或者，可以使用/Tpfile.cppm。选项/interface 指定以下所有文件都是接口文件 (在一个命令行中同时拥有接口和非接口文件可能无法正常工作)。

    若使用文件扩展名.ixx，编译器会自动将该文件识别为接口文件。
  - 编译一个内部分区 file.cppp 的命令行如下所示:

    ```
    cl /Tp /internalPartition /c file.cppp
    ```

    选项/internalPartition 指定以下所有文件为内部分区，不支持在一个命令行中同时使用内部分区和接口文件。没有特定的后缀可以代替，内部分区需要这个选项。

  事实上，Visual C++ 目前推荐不同的文件后缀，并要求针对特定模块单元使用特定的命令行选项，这使得模块的使用既麻烦又不可移植。为了规避 Visual C++ 的限制 (至少在通过命令行编译时)，我提供了 Python 脚本 clmod.py，可以在http://github.com/josuttis/cppmodules上找到它。我希望微软能修复他们的缺陷，这样就不再需要这种变通的方法了。
- gcc/g++:

  gcc 根本不需要任何特殊的文件扩展名或命令行选项。通过使用特殊的文件扩展名，只需要使用命令行选项-xc++ 指定文件包含 C++ 代码:
  - 编译一个接口文件 file.cppm 的命令行如下所示:

```
g++ -xc++ -c file.cppm
```

– 编译一个内部分区文件 file.cppp 的命令行如下所示:

```
g++ -xc++ -c file.cppp
```

- Clang:

  Clang 目前只支持接口文件。由于建议的扩展名.cppm 无论如何都是必需的，因此应该可以工作。

  但不能使用内部分区文件。

16.3.2 处理头文件

虽然理论上模块可以取代所有传统的头文件，但实际上这永远不会发生。将会有一些不需要使用模块的头文件，其来自为 C++(和 C) 开发的代码和库。因为使用预编译器会使编译和链接 C++ 程序变得更加复杂，所以情况尤其如此，所以模块应该能够处理传统的头文件。

使用传统头文件的基本方法是使用全局模块。

- 用 module 开启模块;
- 然后，在进行模块声明之前放置所有必要的预处理器命令

这种情况下:

- 包含的头文件中未使用的所有内容都将丢弃。
- 使用的所有内容都将获得模块链接，所以只在整个模块单元中可见，而在其他模块单元和模块外部都不可见。
- 在 #include 之前使用 #define。

例如:

```
1  module;
2
3  #include <string>
4  #define NDEBUG
5  #include <cassert>
6
7  export module ModTest;
8  ...
9  void foo(std::string s) {
10   assert(s.empty()); // valid but not checked
11   ...
12 }
```

这个全局模块中，预处理器符号 NDEBUG 和宏 assert() 都是在这个模块单元中定义的。但由于 NDEBUG，使用 assert() 的运行时检查都是禁用的。

NDEBUG 和 assert() 在该模块或导入模块的其他单元中都不可见。

声明模块之后，不再支持 #include。可以使用其他预处理器命令，如 #define 和 #ifdef。

头文件的导入

未来的目标是使整个 C++ 标准库作为模块可用。然而，对于标准的 C++ 头文件，已经可以使用 import，然后可以在模块中使用。例如:

```
export module ModTest;

import <chrono>;
```

该命令是声明和导入模块的快捷方式，并导出相应头文件中的所有内容。通过此导入，甚至可以在该模块中看到宏 (对于所有其他导入，都不可见)。

但在使用 #define 导入之前定义的常量不会传递给导入的头文件。这样，就可以保证导入的头文件的内容总是相同的，这样头文件就可以预编译了。

注意，此功能只保证在标准 C++ 头文件上工作，不适用于 C++ 使用的标准 C 头文件:

```
export module ModTest;

import <chrono>; // OK
import <cassert>; // ERROR (or at least not portable)
```

平台也允许为其他头文件支持此功能，但使用此特性的代码不可移植。

标准模块

C++20 只是介绍了这种技术，没有引入任何标准模块 (原因是标准委员会希望重新组织不同模块上的符号，以清理头文件中一些历史上的混乱)。

看起来在 C++23，将有两个标准模块 (参见http://wg21.link/p2465)

- std 将提供来自 C++ 头文件的命名空间 std 中的所有内容，包括那些包装 C 的内容 (例如 std::sort()，std::ranges::sort()，std::fopen() 和::operator new)。
  不提供宏和特性测试宏。对于它们，必须手动包含 <cassert> 或 <version>。
- std.compat 将提供模块 std 中的所有内容，以及 C 头文件中的 C 符号对应 (例如，::fopen())。

std 和每个以 std 开头的模块名，都由 C++ 标准为其标准模块进行保留。

16.4. 模块的详情

本节描述使用模块的一些其他细节。

16.4.1 私有模块

若在主接口中声明一个模块，有时可能需要一个私有模块。这允许开发者在主接口中拥有声明和定义，这些声明和定义对任何其他模块或翻译单元都是不可见或不可达的。使用私有模块片段的

一种方法是禁用导出类或函数的定义，尽管声明是导出的。

例如，考虑下面的主接口:

```
export module MyMod;

export class C; // class C is exported
export void print(const C& c); // print() is exported

class C { // provides details of the exported class
private:
  int value;
public:
  void print() const;
};

void print(const C& c) { // provides details of the exported function
  c.print();
}
```

首先用 export 声明类 C 和函数 print():

```
export module MyMod;

export class C; // class C is exported
export void print(const C& c); // print() is exported
```

export 只能在命名空间中引入名称时指定一次，稍后还会导出详细信息。翻译单元都可以导入该模块，并使用 C 类型的对象:

```
import MyMod;
...

C c; // OK, definition of class C was exported
print(c); // OK (compiler can replace the function call with its body)
```

若想在模块中封装定义，以便导入代码只看到声明，并且仍然希望在主接口中拥有定义，必须将定义放在私有模块中:

```
export module MyMod;

export class C; // declaration is exported
export void print(const C& c); // declaration is exported

module :private; // following symbols are not even implicitly exported

class C { // complete class not exported
  private:
```

```
  int value;
  public:
  void print() const;
};

void print(const C& c) { // definition not exported
  c.print();
}
```

声明私有模块

```
module :private;
```

只能发生在主接口中，并且只能发生一次。有了这个声明，文件的其余部分不再隐式导出 (甚至不是隐式导出)。之后使用 export 来导出内容都是错误的。

通过将定义移动到私有模块中，导入代码就不能再使用其中的任何定义，只能使用类 C 的前向声明 (类 C 是不完整类型) 和 print()。

例如，不能创建 C 类型的对象:

```
import MyMod;
...

C c; // ERROR (C only declared, not defined)
print(c); // OK (compiler can replace the function call with its body)
```

声明还是可以使用 C 类型的引用和指针:

```
import MyMod;
...

void foo(const C& c) { // OK
  print(c); // OK
}
```

16.4.2 详细介绍模块的声明和导出

模块单元必须以下列之一开头 (初始注释和空格之后):

- module;
- export module name;
- module name;
- module name:partname;
- export module name:partname;

若一个模块单元以 module 开头; 要引入全局模块，其他模块声明中的一个必须遵循全局模块中的预处理器命令。

模块中，可以导出所有具有名称的符号:

- 可以导出具有名称的命名空间，将导出命名空间声明中定义的所有符号。例如:

```
export namespace MyMod {
  ... // exported symbols of namespace MyMod
}

namespace MyMod {
  ... // symbols of namespace MyMod not exported
}

export namespace MyMod {
  ... // more exported symbols of namespace MyMod
}
```

- 可以导出类型，这将导出所有成员 (若有的话)。例如:

```
export class MyClass;
export struct MyStruct;
export union MyUnion;
export enum class MyEnum;
export using MyString = std::string;
```

不必导出类成员或枚举值。若导出类型，则自动导出枚举类型的类成员和值。

- 可以导出对象。例如:

```
export std::string progname;

namespace MyStream {
  export using std::cout; // export std::cout as MyStream::cout
}

export auto myLambda = [] {};
```

- 可以导出函数。例如:

```
export friend std::ostream& operator<< (std::ostream&, const MyType&);
```

声明的实体应该导出，以导出作为首次声明。可以稍后再次指定它是导出的，但是不允许在没有导出的情况下声明实体，并在以后使用导出声明/定义实体。[然而，编译器已经接受了这一点。]

在未命名的命名空间、静态对象和私有模块片段中不允许导出。

导出都不需要内联。形式上，模块中的定义总是只存在一次。

即使对象也被另一个模块重新导出，这也适用。

16.4.3 伞形模块

模块可以导出所有内容。对于导入的接口分区，也需要导出。

要导出导入的符号，可以使用 using:

```
export module MyMod; // declare module MyMod

// export all symbols from OtherModule as a whole:
export import OtherModule;

// import LogModule to export parts of it:
import LogModule

// export Logger in the namespace LogModule as ::Logger:
export using LogModule::Logger;

// export Logger in the namespace LogModule as LogModule::Logger:
export namespace LogModule {
  using LogModule::Logger;
}

// export global symbol globalLogger:
export using ::globalLogger;

// export global symbol log (e.g., function log()):
export using ::log;
```

16.4.4 模块导入的详情

通过 import，C++ 源代码文件都可以导入一个模块来使用导出的函数、类型和对象。

import 不是一个普通的关键字，是一个上下文关键字。仍然可以使用标识符导入命名其他组件，尽管不建议这样做。其没有作为关键字引入，所以可能会破坏太多现有的代码。使用 import 的翻译单元或模块单元，必须在模块预编译之后进行编译。

否则，可能会得到模块未定义的错误，或者更糟的是，可能会针对旧版本的模块进行编译，所以不能循环导入。

16.4.5 可达符号与可见符号的详情

让我们看一些更多的详情，并举例说明导出和导入符号的可见性和可达性。

导入模块时，还可以间接导入导出 API 所使用的所有类型。若没有显式导出这些类型，则可以使用包含其所有成员函数的类型，但不能使用独立函数。

下面的模块将 getPerson() 导出为一个可见的符号，将 Person 导出为一个可访问的类:

modules/person1.cppm

```
module;
#include <iostream>
#include <string>

export module ModPerson; // THE module interface
```

```
6
7  class Person { // note: not exported
8    std::string name;
9  public:
10   Person(std::string n)
11   : name{std::move(n)} {
12   }
13   std::string getName() const {
14     return name;
15   }
16 };
17
18 std::ostream& operator<< (std::ostream& strm, const Person& p)
19 {
20   return strm << p.getName();
21 }
22
23 export Person getPerson(std::string s) {
24   return Person{s};
25 }
```

导入该模块会产生以下结果:

```
1  #include <iostream>
2  import ModPerson; // import module ModPerson
3  ...
4  Person p1{"Cal"}; // ERROR: Person not visible
5  Person p2 = getPerson("Kim"); // ERROR: Person not visible
6  auto p3 = getPerson("Tana"); // OK
7  std::string s1 = p3.getName(); // ERROR (unless <iostream> includes <string>)
8  auto s2 = p3.getName(); // OK
9  std::cout << p3 << '\n'; // ERROR: free-standing operator<< not exported
10 std::cout << s2 << '\n'; // OK
```

若这段代码还包括字符串的头文件，则 s1 的声明编译为:

```
1  #include <iostream>
2  #include <string>
3  import ModPerson; // import module ModPerson
4  ...
5  Person p1{"Cal"}; // ERROR: Person not visible
6  Person p2 = getPerson("Kim"); // ERROR: Person not visible
7  auto p3 = getPerson("Tana"); // OK
8  std::string s1 = p3.getName(); // OK
9  auto s2 = p3.getName(); // OK
10 std::cout << p3 << '\n'; // ERROR: free-standing operator<< not exported
11 std::cout << s2 << '\n'; // OK
```

若在 Person 类中声明 << 操作符作为一个隐藏的友元 (应该这样做):

```
export module ModPerson;

class Person {
  ...
  friend std::ostream& operator<< (std::ostream& strm, const Person& p) {
    return strm << p.getName();
  }
};
```

成员操作符变为可访问的:

```
member operator becomes reachable:
auto p3 = getPerson("Tana"); // OK
std::cout << p3 << '\n'; // OK (operator<< is reachable now)
```

通过使用私有模块，可以限制间接导出符号的可达性。

未导出的符号不能冲突

间接导出的符号不可见，但可访问的行为允许开发者在其导出的接口中，使用相同符号名称的不同模块。例如，有一个模块定义了一个类 Person:

```
export module ModPerson1;

class Person {
  ...
  public:
  std::string getName() const {
    return name;
  }
};

export Person getPerson1(std::string s) {
  return Person{s};
}
```

另一个模块也定义了一个不同的类 Person:

```
export module ModPerson2;

class Person {
  ...
  public:
  std::string getName() const {
    return name;
```

```
  }
};

export Person getPerson2(std::string s) {
  return Person{s};
}
```

程序可以导入两个模块而不会产生任何冲突，因为唯一可见的符号是第一个模块的 getPerson1() 和第二个模块的 getPerson2()。下面的代码可以正常工作:

```
auto p1 = getPerson1("Tana");
auto s1 = p1.getName();

auto p2 = getPerson2("Tana");
auto s2 = p2.getName();
```

p1 和 p2 的类型名称相同，但实际类型不同:

```
std::same_as<decltype(p1), decltype(p2)> // yields false
```

16.5. 附注

C++ 中支持模块的想法是相当古老的。2004 年，Daveed Vandevoorde 在http://wg21.link/n1736上发表了关于这一问题的第一篇论文。

由于功能的复杂性，通过http://wg21.link/n4592建立了一个模块 TS(实验技术规范) 来处理细节。Gabriel Dos Reis 是本次 TS 内容的主要推动者。

最终将模块合并到 C++20 标准中的提案是 Richard Smith 在http://wg21.link/p1103r3中提出。在那之后，不同的作者在不同的论文中应用了一些修复和说明。

第 17 章 Lambda 的扩展

本章介绍 C++20 为 Lambda 引入的补充特性。

17.1. 带模板参数的泛型 Lambda

C++20 引入了一个扩展，允许对泛型 Lambda 使用模板形参。可以在捕获子句和调用参数 (若有的话) 之间指定这些模板参数:

```
auto foo = []<typename T>(const T& param) { // OK since C++20
        T tmp{}; // declare object with type of the template parameter
        ...
      };
```

Lambda 的模板形参有一个优点，即在声明泛型参数时为类型或类型的一部分提供名称。例如:

```
[]<typename T>(T* ptr) { // OK since C++20
  ... // can use T as type of the value that ptr points to
};
```

或:

```
[]<typename T, int N>(T (&arr)[N]) {
  ... // can use T as element type and N as size of the passed array
};
```

若想知道为什么在这些情况下不使用函数模板，Lambda 提供了一些函数无法提供的便利:

- 可以在函数内部定义。
- 可以捕获运行时的值，来指定在运行时的功能行为。
- 可以将它们作为参数传递，而无需指定参数类型。

17.1.1 使用泛型 Lambda 的模板参数

显式模板参数可用于特化 (或部分限制) 泛型 Lambda 的参数类型:

```
[]<typename T>(const std::vector<T>& vec) { // can only pass vectors
  ...
};
```

这个 Lambda 只接受 vector 作为参数。当使用 auto 时，将实参限制为 vector 并不容易，因为 C++(还) 不支持 std::vector<auto> 类型，但还可以使用类型约束来约束参数的类型 (例如：要求可随机访问，甚至特定类型)。

显式模板参数也有助于避免对 decltype 的需要。例如，要在 Lambda 中完美地转发泛型参数包:

```
[]<typename... Types>(Types&&... args) {
  foo(std::forward<Types>(args)...);
};
```

而非:

```
[] (auto&&... args) {
  foo(std::forward<decltype(args)>(args)...);
};
```

类似的例子是在访问 std::variant<> 时，会具有特定类型的行为 (C++17 中引入):

```
std::variant<int, std::string> var;
...
// call generic lambda with type-specific behavior:
std::visit([](const auto& val) {
        if constexpr(std::is_same_v<decltype(val), const std::string&>) {
          ... // string-specific processing
        }
        std::cout << val << '\n';
      },
      var);
```

必须使用 decltype() 来获取参数的类型，并将该类型作为 const& 进行比较 (或删除 const 和引用)。C++20 起，可以这样:

```
std::visit([]<typename T>(const T& val) { // since C++20
        if constexpr(std::is_same_v<T, std::string>) {
          ... // string-specific processing
        }
        std::cout << "value: " << val << '\n';
      },
      var);
```

还可以为 coneval 的 Lambda 声明模板参数，强制在编译时执行。

17.1.2 Lambda 模板参数的显式规范

Lambda 提供了一种方便的方式来定义函数对象 (函子)，对于泛型 lambda，函数调用函数操作符是一个模板。使用为参数指定名称，而不是使用 auto 的语法，可以在生成的函数调用操作符中为模板参数指定名称。

例如，若定义了下面的 Lambda:

```
auto primeNumbers = [] <int Num> () {
            std::array<int, Num> primes{};
```

```
            ... // compute and assign first Num prime numbers
            return primes;
          };
```

编译器定义了相应的闭包类型:

```
class NameChosenByCompiler {
public:
  ...
  template<int Num>
  auto operator() () const {
    std::array<int, Num> primes{};
    ... // compute and assign first Num prime numbers
    return primes;
  }
};
```

并创建该类的一个对象 (若没有捕获值则使用默认构造函数):

```
auto primeNumbers = NameChosenByCompiler{};
```

要显式指定模板形参，当将 Lambda 用作函数时，必须将其传递给函数操作符:

```
// initialize array with the first 20 prime numbers:
auto primes20 = primeNumbers.operator()<20>();
```

除非使用间接调用，否则在指函数操作符的模板形参时，无法避免指定函数操作符。

可以尝试将模板参数设置为编译时值，使其可推导，但最终的语法情况并没有好到哪里去:

```
auto primeNumbers = [] <int Num> (std::integral_constant<int, Num>) {
            std::array<int, Num> primes{};
            ... // compute and assign first Num prime numbers
            return primes;
          };

// initialize array with the first 20 prime numbers:
auto primes20 = primeNumbers(std::integral_constant<int,20>{});
```

或者，可以考虑使用变量模板，这是 C++14 中引入的一种技术。就可以将 primeNumbers 变量设置为泛型，而非将 Lambda 设置为泛型:

```
template<int Num>
auto primeNumbers = [] () {
            std::array<int, Num> primes{};
            ... // compute and assign first Num prime numbers
            return primes;
          };
```

```
...
// initialize array with the first 20 prime numbers:
auto primes20 = primeNumbers<20>();
```

这种情况下，不能在函数作用域中定义 Lambda。泛型 Lambda 可以在作用域内局部定义泛型功能。

17.2. 调用 Lambda 的默认构造函数

Lambda 提供了一种简单的方法来定义函数对象，若定义

```
auto cmp = [] (const auto& x, const auto& y) {
        return x > y;
      };
```

这相当于定义一个类 (闭包类型) 并创建一个该类的对象:

```
class NameChosenByCompiler {
  public:
  template<typename T1, T2>
  auto operator() (const T1& x, const T2& y) const {
    return x > y;
  }
};

auto cmp = NameChosenByCompiler{};
```

生成的闭包类型定义了函数操作符，则可以将 Lambda 对象 cmp 作为函数使用:

```
cmp(val1, val); // yields the result of 42 > obj2
```

C++20 之前，生成的闭包类型没有可调用的默认构造函数和赋值操作符。生成类的对象最初只能由编译器创建，只可复制:

```
auto cmp1 = [] (const auto& x, const auto& y) {
        return x > y;
      };

auto cmp2 = cmp1; // OK, copy constructor supported since C++11
decltype(cmp1) cmp3; // ERROR until C++20: no default constructor provided
cmp1 = cmp2; // ERROR until C++20: no assignment operator provided
```

因为容器需要辅助函数的类型，所以不好将 Lambda 作为排序标准或哈希函数传递给容器。考虑一个具有以下接口的 Customer 类:

```
class Customer
{
  public:
  ...
  std::string getName() const;
};
```

要使用 getName() 返回的名称作为哈希函数的排序标准或值，必须将类型和 Lambda 作为模板和调用参数传递:

```
// create balanced binary tree with user-defined ordering criterion:
auto lessName = [] (const Customer& c1, const Customer& c2) {
  return c1.getName() < c2.getName();
};
std::set<Customer, decltype(lessName)> coll1{lessName};

// create hash table with user-defined hash function:
auto hashName = [] (const Customer& c) {
  return std::hash<std::string>{}(c.getName());
};
std::unordered_set<Customer, decltype(hashName)> coll2{0, hashName};
```

容器在初始化时获得 Lambda，以便可以使用 Lambda 的内部副本 (对于无序容器，必须先传递最小桶大小)。要进行编译，容器的类型需要 Lambda 的类型。

自 C++20 起，没有捕获的 Lambda 有一个默认构造函数和一个赋值操作符:

```
auto cmp1 = [] (const auto& x, const auto& y) {
        return x > y;
      };

auto cmp2 = cmp1; // OK, copy constructor supported
decltype(cmp1) cmp3; // OK since C++20
cmp1 = cmp2; // OK since C++20
```

出于这个原因，可以分别为排序条件或哈希函数传递 Lambda 的类型:

```
// create balanced binary tree with user-defined ordering criterion:
auto lessName = [] (const Customer& c1, const Customer& c2) {
          return c1.getName() < c2.getName();
        };
std::set<Customer, decltype(lessName)> coll1; // OK since C++20

// create hash table with user-defined hash function:
auto hashName = [] (const Customer& c) {
          return std::hash<std::string>{}(c.getName());
        };
```

```
11
12 std::unordered_set<Customer, decltype(hashName)> coll2; // OK since C++20
```

这是有效的，因为排序条件或散列函数的参数有一个默认值，该值是排序条件或哈希列函数类型的默认构造对象。而且由于自 C++20 以来，没有捕获的 Lambda 有默认构造函数，因此使用 Lambda 类型的默认构造对象初始化排序条件现在可以编译了。

甚至可以在容器的声明中定义 Lambda，并使用 decltype 传递其类型。例如，可以声明一个关联容器，并在声明中定义排序:

```
1 // create balanced binary tree with user-defined ordering criterion:
2 std::set<Customer,
3     decltype([] (const Customer& c1, const Customer& c2) {
4       return c1.getName() < c2.getName();
5     })> coll3; // OK since C++20
```

以同样的方式，可以用哈希函数声明一个无序容器:

```
1 // create hash table with user-defined hash function:
2 std::unordered_set<Customer,
3           decltype([] (const Customer& c) {
4             return std::hash<std::string>{}(c.getName());
5           })> coll; // OK since C++20
```

请参阅 lang/lambdahash.cpp 获取完整示例。

17.3. Lambda 作为非类型模板参数

C++20 起，Lambda 可以用作非类型模板形参 (NTTPs):

```
1  template<std::invocable auto GetVat>
2  int addTax(int value)
3  {
4    return static_cast<int>(std::round(value * (1 + GetVat())));
5  }
6
7  auto defaultTax = [] { // OK
8    return 0.19;
9  };
10
11 std::cout << addTax<defaultTax>(100) << '\n';
```

这个特性有一个副作用，即仅将带有公共成员的文字类型用作非类型模板参数类型。有关详细的讨论和完整的示例，请参阅关于非类型模板参数扩展的章节

17.4. consteval 的 Lambda

通过对 Lambda 使用新的 consteval 关键字，现在可以要求 lambdas 成为直接函数，以便“函数调用”必须在编译时求值。例如:

```
auto hashed = [] (const char* str) consteval {
          ...
        };
auto hashWine = hashed("wine"); // hash() called at compile time
```

由于在 Lambda 的定义中使用了 consteval，调用都必须在编译时使用编译时已知的值进行。传递运行时值是错误的:

```
const char* s = "beer";
auto hashBeer = hashed(s); // ERROR

constexpr const char* cs = "water";
auto hashWater = hashed(cs); // OK
```

哈希本身并不一定是 constexpr，其是 Lambda 的运行时对象，在编译时对其执行“函数调用”。

关于新的 consteval 关键字一节中对 Lambda 的 consteval 进行更加详细的讨论。

还可以将新的模板语法用于具有 consteval 的泛型 Lambda，这使开发者能够在另一个函数中定义编译时函数的初始化。例如:

```
// local compile-time computation of Num prime numbers:
auto primeNumbers = [] <int Num> () consteval {
          std::array<int, Num> primes;
          int idx = 0;
          for (int val = 1; idx < Num; ++val) {
            if (isPrime(val)) {
              primes[idx++] = val;
            }
          }
          return primes;
        };
```

请参阅 lang/lambdaconsteval.cpp 获取使用该 Lambda 的完整示例。

这时，模板参数没有推导出来，所以显式指定模板参数的语法会有点难看:

```
auto primes = primeNumbers.operator()<100>();
```

必须在 consteval 之前提供参数列表 (指定 constexpr 时也适用)。即使没有声明参数，也不能没有括号。

17.5. 对捕获的修改

C++20 引入了几个新的扩展来捕获 Lambda 中的值和对象。

17.5.1 捕获 this 和 *this

若在成员函数中定义 Lambda，问题是如何访问调用该成员函数对象的数据。C++20 前，有以下规则:

```
class MyType {
  std::string name;
  ...
  void foo() {
    int val = 0;
    ...
    auto l0 = [val] { bar(val, name); }; // ERROR: member name not captured
    auto l1 = [val, name=name] { bar(val, name); }; // OK, capture val and name by
↪   value

    auto l2 = [&] { bar(val, name); }; // OK (val and name by reference)
    auto l3 = [&, this] { bar(val, name); }; // OK (val and name by reference)
    auto l4 = [&, *this] { bar(val, name); }; // OK (val by reference, name by value)

    auto l5 = [=] { bar(val, name); }; // OK (val by value, name by reference)
    auto l6 = [=, this] { bar(val, name); }; // ERROR before C++20
    auto l7 = [=, *this] { bar(val, name); }; // OK (val and name by value)
    ...
  }
};
```

自 C++20 起，可使用以下规则:

```
class MyType {
  std::string name;
  ...
  void foo() {
    int val = 0;
    ...
    auto l0 = [val] { bar(val, name); }; // ERROR: member name not captured
    auto l1 = [val, name=name] { bar(val, name); }; // OK, capture val and name by
↪   value

    auto l2 = [&] { bar(val, name); }; // deprecated (val and name by ref.)
    auto l3 = [&, this] { bar(val, name); }; // OK (val and name by reference)
    auto l4 = [&, *this] { bar(val, name); }; // OK (val by reference, name by value)

    auto l5 = [=] { bar(val, name); }; // deprecated (val by value, name by ref.)
    auto l6 = [=, this] { bar(val, name); }; // OK (val by value, name by reference)
    auto l7 = [=, *this] { bar(val, name); }; // OK (val and name by value)
    ...
  }
};
```

C++20 起，有如下的变化:

=, this 现在允许作为 Lambda 捕获 (一些编译器以前允许，尽管在形式上无效)。

- 不可隐式捕获 *this。

17.5.2 捕获结构化绑定

C++20 起，允许捕获结构化绑定 (C++17 引入):

```
std::map<int, std::string> mymap;
...

for (const auto& [key,val] : mymap) {
  auto l = [key, val] { // OK since C++20
         ...
       };
  ...
}
```

一些编译器以前确实允许捕获结构化绑定，尽管它在形式上无效。

17.5.3 捕获可变模板的参数包

若有一个可变的模板，可以像下面这样捕获参数包:

```
template<typename... Args>
void foo(Args... args)
{
  auto l1 = [&] {
          bar(args...); // OK
        };
  auto l2 = [args...] { // or [=]
          bar(args...); // OK
        };
  ...
}
```

若想返回为以后使用而创建的 Lambda，就会出现问题:

- 使用 [&]，将返回一个 lambda，该 Lambda 会引用已销毁的参数包。
- 使用 [args…] 或 [=]，则复制传递的参数包。

捕获对象时，可以使用 init-capturing 来使用移动语义:

```
template<typename T>
void foo(T arg)
{
  auto l3 = [arg = std::move(arg)] { // OK since C++14
```

```
        bar(arg); // OK
      };
  ...
}
```

但是，没有提供与参数包一起使用 init-capture 的语法。

C++20 引入了相应的语法:

```
template<typename... Args>
void foo(Args... args)
{
  auto l4 = [...args = std::move(args)] { // OK since C++20
        bar(args...); // OK
      };
  ...
}
```

还可以通过引用来初始化捕获参数包。例如，可以将 Lambda 的参数名更改为:

```
template<typename... Args>
void foo(Args... args)
{
  auto l4 = [&...fooArgs = args] { // OK since C++20
        bar(fooArgs...); // OK
      };
  ...
}
```

使用实例获取可变模板的参数包

创建并返回一个按值捕获可变数量参数的 Lambda 泛型函数:

```
template<typename Callable, typename... Args>
auto createToCall(Callable op, Args... args)
{
  return [op, ...args = std::move(args)] () -> decltype(auto) {
        return op(args...);
      };
}
```

使用简化函数模板的新语法:

```
auto createToCall(auto op, auto... args)
{
  return [op, ...args = std::move(args)] () -> decltype(auto) {
        return op(args...);
```

```
    };
}
```

下面是一个完整的例子:

lang/capturepack.cpp

```cpp
#include <iostream>
#include <string_view>

auto createToCall(auto op, auto... args)
{
  return [op, ...args = std::move(args)] () -> decltype(auto) {
    return op(args...);
  };
}

void printWithGAndNoG(std::string_view s)
{
  std::cout << s << "g " << s << '\n';
}

int main()
{
  auto printHero = createToCall(printWithGAndNoG, "Zhan");
  ...
  printHero();
}
```

17.5.4 Lambda 作为协程

Lambda 也可以是协程，这是在 C++20 引入的。但这种情况下，Lambda 不该捕获任何内容，因为协程可能比本地创建的 Lambda 对象存在的时间更长。

17.6. 附注

泛型 Lambda 的模板语法最初是由 Louis Dionne 在http://wg21.link/p0428r0中提出。最后接受的提案是由 Louis Dionne 在http://wg21.link/p0428r2中提出。

泛型 Lambda 的模板语法最初是由 Louis Dionne 在http://wg21.link/p0624r0中提出。最后接受的提案是由 Louis Dionne 在http://wg21.link/p0624r2中提出。

允许 Lambda 作为非类型模板参数是由 Jeff Snyder 和 Louis Dionne 在http://wg21.link/p0732r2中引入的。

对 consteval 的 Lambda 的支持是由 Richard Smith、Andrew Sutton 和 Daveed Vandevoorde 在http://wg21.link/p1073r3中最终提出。

对捕获 this 和 *this 的规则的修改最初是由 Thomas Köppe 在http://wg21.link/p0409r0和 http://wg21.link/p0806r0 中提出。最终接受的提案是由 Thomas Köppe 在http://wg21.link/p0409r2和http://wg21.link/p0806r2中制定。

捕获结构化绑定是 Nicolas Lesser 在http://wg21.link/p1091r3中最终提出。

Init-capturing 参数包由 Barry Revzin 在http://wg21.link/p0780r2和http://wg21.link/p2095r0中最终制定。

第 18 章 编译时计算

本章介绍了 C++ 的几个支持编译时计算的扩展。

本章介绍了两个新的关键字 constinit 和 consteval，允许开发者在编译时使用堆内存、vector 和字符串的扩展。

18.1. 关键字 constinit

C++20 引入的一个新关键字是 constinit，可用于强制并确保在编译时初始化可变静态或全局变量。大致效果为:

```
constinit = constexpr - const
```

是的，constinit 变量不是 const(最好将关键字命名为 compiletimeinit)。这个名称来源于这样一个事实，即这些初始化通常在编译时常量初始化时发生。

无论何时声明静态或全局变量，都可以使用 constinit。例如:

```
// outside any function:
constinit auto i = 42;
int getNextClassId() {
  static constinit int maxId = 0;
  return ++maxId;
}

class MyType {
  static constinit long max = sizeof(int) * 1000;
  ...
};

constexpr std::array<int, 5> getColl() {
  return {1, 2, 3, 4, 5};
}
constinit auto globalColl = getColl();
```

仍然可以修改声明的值。下面的代码是第一次使用上面的声明:

```
std::cout << i << " " << coll[0] << '\n'; // prints 42 1
i *= 2;
coll = {};
std::cout << i << " " << coll[0] << '\n'; // prints 84 0
```

输出如下:

```
42 1
84 0
```

使用 constinit 的效果是，只有当初始值是编译时已知的常数值时，初始化才会编译，与以下声明相反:

```
auto x = f(); // f() might be a runtime function
```

与 constinit 相对应的声明需要编译时初始化，则必须能够在编译时调用 f()(f() 必须是 constexpr 或 consteval)。

```
constinit auto x = f(); // f() must be a compile-time function
```

若用 constinit 初始化一个对象，必须能够在编译时使用构造函数:

```
constinit std::pair p{42, "ok"}; // OK constinit std::list l; // ERROR: default
↪  constructor not constexpr
```

使用 constinit 的原因如下所示:

- 可以在编译时要求初始化可变全局/静态对象。就可以避免在运行时进行初始化，当使用 thread_local 变量时，这可以提高性能。
- 可以确保全局/静态对象在使用时总是初始化。实际上，constinit 可以用来修复静态初始化顺序的错误，当一个静态/全局对象的初始值依赖于另一个静态/全局对象时，就会出现这种错误。

使用 constinit 永远不会改变程序的功能行为 (除非我们遇到静态初始化顺序的惨败)，只能导致代码不再可编译。

18.1.1 实践 constinit

使用 constinit 时有几件事需要注意。

首先，不能用另一个 constinit 值初始化一个 constinit 值:

```
constinit auto x = f(); // f() must be a compile-time function
constinit auto y = x; // ERROR: x is not a constant initializer
```

原因是初始值必须是编译时已知的常量值，但 constinit 不是常量。只编译以下代码:

```
constexpr auto x = f(); // f() must be a compile-time function
constinit auto y = x; // OK
```

初始化对象时，需要编译时构造函数，但编译时析构函数不是必需的。出于这个原因，可以对智能指针使用 constinit:

```
constinit std::unique_ptr<int> up; // OK
constinit std::shared_ptr<int> sp; // OK
```

constinit 并不意味着内联 (这与 constexpr 不同)。例如:

```
class Type {
  constinit static int val1 = 42; // ERROR
  inline static constinit int val2 = 42; // OK
  ...
};
```

可以和 extern 一起使用 constinit:

```
// header:
extern constinit int max;

// translation unit:
constinit int max = 42;
```

为了达到同样的效果，也可以跳过声明中的 constinit，但在定义中跳过 constinit，将不再强制进行编译时初始化。

也可以将 constinit 与 static 和 thread_local 一起使用:

```
static thread_local constinit int numCalls = 0;
```

constinit、static 和 thread_local 的顺序无所谓。

对于 thread_local 变量，使用 constinit 可能会提高性能，因为其可以避免生成的代码 (需要内部保护) 来指示变量是否初始化:

```
extern thread_local int x1 = 0;
extern thread_local constinit int x2 = 0; // better (might avoid an internal guard)
```

使用 constinit 声明引用是可能的，但是没有意义，因为引用引用的是一个常量对象，这里应该使用 constexpr。

18.1.2 constinit 如何解决静态初始化顺序的问题

C++ 中，有一个叫做静态初始化顺序的问题，constinit 可以解决这个问题。问题是没有定义不同翻译单元中静态和全局初始化的顺序。出于这个原因，下面的代码可能有问题:

- 假设有一个带构造函数的类型来初始化对象，并引入该类型的外部全局对象:

comptime/truth.hpp

```
#ifndef TRUTH_HPP
#define TRUTH_HPP

struct Truth {
  int value;
  Truth() : value{42} { // ensure all objects are initialized with 42
  }
```

```
};
extern Truth theTruth; // declare global object

#endif // TRUTH_HPP
```

- 在对象自己的转换单元中初始化该对象:

comptime/truth.cpp

```
#include "truth.hpp"

Truth theTruth; // define global object (should have value 42)
```

- 然后，在另一个转换单元中，用 theTruth 初始化另一个全局/静态对象:

comptime/fiasco.cpp

```
#include "truth.hpp"
#include <iostream>

int val = theTruth.value; // may be initialized before theTruth is initialized

int main()
{
  std::cout << val << '\n'; // OOPS: may be 0 or 42
  ++val;
  std::cout << val << '\n'; // OOPS: may be 1 or 43
}
```

有一个很好的机会，val 初始化在 theTruth 初始化之前。程序可能有以下输出:[例如，在使用 gcc 编译器时，若传递在 fiasco.o 前传递了 truth.o 给连接器，就会出现这个问题。]

```
0
1
```

当使用 constinit 声明 val 时，就不会出现这个问题。constinit 确保对象总是在使用之前进行初始化，因为初始化是在编译时进行的。若不能给出保证，代码将无法编译。若 val 用 constexpr 声明，初始化也会得到保证，但这种情况下，将无法再修改该值。

例子中，仅仅使用 constinit 首先会导致一个编译时错误 (表示在编译时无法保证初始化):

```
// truth.hpp:
struct Truth {
  int value;
  Truth() : value{42} {
  }
};

extern Truth theTruth;
// main translation unit:
constinit int val = theTruth.value ; // ERROR: no constant initializer
```

该错误消息表明，在编译时不能初始化 val。为了使初始化有效，必须修改类 Truth 和 theTruth 的声明，以便 theTruth 可以在编译时使用:

comptime/truthc.hpp

```
#ifndef TRUTH_HPP
#define TRUTH_HPP

struct Truth {
  int value;
  constexpr Truth() : value{42} { // enable compile-time initialization
  }
};

constexpr Truth theTruth; // force compile-time initialization

#endif // TRUTH_HPP
```

现在，编译程序，val 可保证用 theTruth 的初始值进行初始化:

comptime/constinit.hpp

```
#include "truthc.hpp"
#include <iostream>

constinit int val = theTruth.value ; // initialized after theTruth is initialized

int main()
{
  std::cout << val << '\n'; // guaranteed to be 42
  ++val;
  std::cout << val << '\n'; // guaranteed to be 43
}
```

程序的输出现在可以保证为:

```
42
43
```

还有其他方法可以解决静态初始化顺序的问题 (使用静态函数获取值或使用内联)。然而，若初始化不需要运行时值/特性，则可能需要仔细考虑遵循总是使用 constinit 声明全局变量和静态变量的编程风格。在这样的函数中使用，至少没有坏处:

```
long nextId()
{
  constinit static long id = 0;
  return ++id;
}
```

18.2. 关键字 consteval

C++11 起，有了关键字 constexpr, 支持在编译时对函数求值。若函数的所有方面在编译时已知，可以在编译时上下文中使用其结果。然而，constexpr 函数也可以作为“普通”运行时函数使用。

C++20 引入了一个类似的关键字 consteval，要求进行编译时计算。与用 constexpr 标记的函数相反，用 consteval 标记的函数不能在运行时调用; 相反，其需要在编译时调用。若这不可能，则程序处于病态。因为这些函数是立即调用的，所以当编译器看到对它们的调用时，这些函数也称为立即函数。

18.2.1 第一个 consteval 的示例

考虑下面的例子:

comptime/consteval1.cpp

```
#include <iostream>
#include <array>

constexpr
bool isPrime(int value)
{
  for (int i = 2; i <= value/2; ++i) {
    if (value % i == 0) {
      return false;
    }
  }
  return value > 1; // 0 and 1 are not prime numbers
}
template<
int Num>
consteval
std::array<int, Num> primeNumbers()
{
  std::array<int, Num> primes;
  int idx = 0;
  for (int val = 1; idx < Num; ++val) {
    if (isPrime(val)) {
      primes[idx++] = val;
    }
  }
  return primes;
}

int main()
{
  // init with prime numbers:
  auto primes = primeNumbers<100>();
  for (auto v : primes) {
```

```
    std::cout << v << '\n';
  }
}
```

这里，使用 consteval 定义了函数 primeNumbers<N>()，在编译时返回一个包含前 N 个素数的数组:

```
template<int Num>
consteval
std::array<int, Num> primeNumbers()
{
  std::array<int, Num> primes;
  ...
  return primes;
}
```

为了计算素数，primeNumbers() 使用一个辅助函数 isPrime()，该函数是用 constexpr 声明的，在运行时和编译时都可用 (也可以用 consteval 声明，但这样它在运行时就不可用了)。

然后使用 primeNumbers<>() 初始化一个包含 100 个素数的数组:

```
auto primes = primeNumbers<100>();
```

因为 primeNumbers<>() 是 consteval，所以这个初始化必须在编译时进行。若使用 constexpr 声明 primeNumbers<>()，并在编译时上下文中调用该函数 (例如，初始化 constexpr 或 constinit 数组 prime)，也会有同样的效果:

```
template<int Num>
constexpr
std::array<int, Num> primeNumbers()
{
  std::array<int, Num> primes;
  ...
  return primes;
}
...
constinit static auto primes = primeNumbers<100>();
```

这两种情况下，当要为初始化计算的素数数量显著增加时，编译时间明显变慢。

但请注意编译器在计算常量表达式时通常有限制，C++ 标准只保证在一个核心常量 exp 内对 1,048,576 个表达式求值

consteval 的 Lambda

现在也可以将 Lambda 声明为 consteval，从而要求在编译时对 Lambda 求值。

考虑下面的例子:

```cpp
int main(int argc, char* argv[])
{
  // compile-time function to compute hash value for string literals:
  // (for the algorithm, see http://www.cse.yorku.ca/~oz/hash.html )
  auto hashed = [] (const char* str) consteval {
          std::size_t hash = 5381;
          while (*str != '\0') {
            hash = hash * 33 ^ *str++;
          }
          return hash;
        };

  // OK (requires hashed() in compile-time context):
  enum class drinkHashes : long { beer = hashed("beer"), wine = hashed("wine"),
                  water = hashed("water"), ... };
  // OK (hashed() guaranteed to be called at compile time):
  std::array arr{hashed("beer"), hashed("wine"), hashed("water")};

  if (argc > 1) {
    switch (hashed(argv[1])) { // ERROR: argv is not known at compile time
      ...
    }
  }
}
```

这里，用一个只能在编译时使用的 Lambda 初始化 hash。因此，Lambda 的使用必须在具有编译时值的编译时上下文中进行。这些调用只有在接受字符串字面值或 constexpr const char* 类型的形参时才有效。

若使用 constexpr 声明 Lambda，则 switch 语句将生效。(没有必要使用 constexpr 声明，从 C++17 开始，所有的 Lambda 都是隐式的 constexpr)。然而，这样就不能保证在编译时对 arr 的初始值进行计算。

有关更多细节，请参阅有关 consteval Lambda 的讨论的章节中的另一个例子。

18.2.2 constexpr 和 consteval

使用 constexpr 和 consteval，现在有以下选项来影响何时可以调用/被调用函数:

- 既不是 constexpr 也不是 consteval:
  这些函数只能在运行时上下文中使用，但编译器仍然可以在编译时进行优化。
- constexpr:
  这些函数可以在编译时和运行时上下文中使用。即使在运行时上下文中，编译器仍然可以在编译时执行对函数求值的优化，编译器还可以在运行时对编译时上下文中的函数求值。
- consteval:
  这些函数只能在编译时使用，但结果可以在运行时上下文中使用。

例如，考虑在头文件中声明的以下三个函数 (因此，squareR() 是用 inline 声明的):

comptime/consteval2.hpp

```
// square() for runtime only:
inline int squareR(int x) {
  return x * x;
}

// square() for compile time and runtime:
constexpr int squareCR(int x) {
  return x * x;
}

// square() for compile time only:
consteval int squareC(int x) {
  return x * x;
}
```

可以这样使用这些函数:

comptime/consteval2.cpp

```
#include "consteval2.hpp"
#include <iostream>
#include <array>

int main()
{
  int i = 42;

  // using the square functions at runtime with runtime value:
  std::cout << squareR(i) << '\n'; // OK
  std::cout << squareCR(i) << '\n'; // OK
  // std::cout << squareC(i) << '' ; // ERROR
  // using the square functions at runtime with c
  ompile-time value:
  std::cout << squareR(42) << '\n'; // OK
  std::cout << squareCR(42) << '\n'; // OK
  std::cout << squareC(42) << '\n'; // OK: square computed at compile time

  // using the square functions at compile time:
  // std::array<int, squareR(42)> arr1; // ERROR
  std::array<int, squareCR(42)> arr2; // OK: square computed at compile time
  std::array<int, squareC(42)> arr3; // OK: square computed at compile time
  // std::array<int, squareC(i)> arr4; // ERROR
}
```

constexpr 和 consteval 的区别如下:

- 不允许 consteval 函数处理编译时未知的参数:

```
std::cout << squareCR(i) << '\n'; // OK
std::cout << squareC(i) << '\n'; // ERROR
std::array<int, squareC(i)> arr4; // ERROR
```

- consteval 函数必须在编译时执行计算:

```
std::cout << squareCR(42) << '\n'; // may be computed at compile time or runtime
std::cout << squareC(42) << '\n'; // computed at compile time
```

在两种情况下使用 consteval 是有意义的:

- 希望强制执行编译时计算。
- 希望禁用在运行时使用的函数。

例如，函数 square() 或 hashed() 是 constexpr 还是 consteval 在编译时上下文中没有区别:

```
enum class Drink = { water = hashed("water"), wine = hashed("wine") };

switch (value) {
  case square(42):
  ...
}

if constexpr(hashed("wine") > hashed("water")) {
  ...
}
```

运行时上下文中，consteval 可能会有所不同，因为那时不需要编译时计算:

```
std::array drinks = { hashed("water"), hashed("wine") };

std::cout << hashed("water");

if (hashed("wine") > hashed("water")) {
  ...
}
```

18.2.3 实践 consteval

对于 consteval 函数，实际使用时有一些限制。

consteval 的限制

consteval 函数与 constexpr 函数几乎相同 (注意，C++20 放宽了这些限制):

- 参数和返回类型 (若不是 void) 必须是文字类型。
- 函数体只能包含既不是静态，也不是 thread_local 的字面量类型的变量。

- 不允许使用 goto 和标签。
- 函数只能是构造函数或析构函数，类没有虚基类。
- consteval 函数隐式内联。
- consteval 函数不能用作协程。

使用 consteval 函数的调用链

带有 consteval 标记的函数可以调用其他带有 constexpr 或 consteval 标记的函数:

```
constexpr int funcConstExpr(int i) {
  return i;
}

consteval int funcConstEval(int i) {
  return i;
}

consteval int foo(int i) {
  return funcConstExpr(i) + funcConstEval(i); // OK
}
```

然而，constexpr 函数不能为变量调用 consteval 函数:

```
consteval int funcConstEval(int i) {
  return i;
}

constexpr int foo(int i) {
  return funcConstEval(i); // ERROR
}
```

函数 foo() 无法编译，原因是 i 仍然不是编译时值，可以在运行时调用。foo() 只能用编译时变量调用 funcConstEval():

```
constexpr int foo(int i) {
  return funcConstEval(42); // OK
}
```

即使 if(std::is_constant_evaluate()) 在这里也没有太大的作用。

带有 consteval 标记的函数也不允许调用纯运行时函数 (既不带有 constexpr，也不带有 consteval 标记的函数)，但只有在真正执行调用时才会检查这一点。对于 consteval 函数，只要没有到达运行时函数，包含调用运行时函数的语句就不是错误 (这条规则已经适用于编译时调用的 constexpr 函数)。

考虑下面的例子:

```cpp
void compileTimeError()
{
}

consteval int nextTwoDigitValue(int val)
{
  if (val < 0 || val >= 99) {
    compileTimeError(); // call something not valid to call at compile time
  }
  return ++val;
}
```

这个编译时函数有一个有趣的效果，只能用于一些值在 0 到 98 之间的参数:

```cpp
constexpr int i1 = nextTwoDigitValue(0); // OK (initializes i1 with 1)
constexpr int i2 = nextTwoDigitValue(98); // OK (initializes i2 with 99)
constexpr int i3 = nextTwoDigitValue(99); // compile-time ERROR
constexpr int i4 = nextTwoDigitValue(-1); // compile-time ERROR
```

使用 static_assert() 在这里不起作用，因为只能为编译时已知的值调用，并且 consteval 不会使 val 成为函数内部的编译时值:

```cpp
consteval int nextTwoDigitValue(int val)
{
  static_assert(val >= 0 && val < 99); // always ERROR: val is not a compile-time
↪  value
  ...
}
```

通过使用这个技巧，可以将编译时函数约束为某些值，就可以在编译时对解析的字符串发出无效格式的信号。

18.2.4 编译时值与编译时上下文

可以假设编译时函数中的每个值都是编译时值，但事实并非如此。在编译时函数中，代码的静态类型和值的动态计算之间也存在差异。

考虑下面的例子:

```cpp
consteval void process()
{
  constexpr std::array a1{0, 8, 15};
  constexpr auto n1 = std::ranges::count(a1, 0); // OK
  std::array<int, n1> a1b; // OK

  std::array a2{0, 8, 15};
```

```
  constexpr auto n2 = std::ranges::count(a2, 0); // ERROR

  std::array a3{0, 8, 15};
  auto n3 = std::ranges::count(a3, 0); // OK
  std::array<int, n3> a3b; // ERROR
}
```

虽然这是一个只能在编译时使用的函数，但 a2 不是编译时值，所以不能用于通常需要编译时值的地方，例如：初始化 constexpr 变量 n2。

出于同样的原因，只能使用 n1 来声明 std::array 的大小，使用非 constexpr 的 n3 会失败 (即使将 n3 声明为 const)。

若 process() 声明为 constexpr 函数，同样适用。但是否在编译时上下文中调用，则并不重要。

18.3. constexpr 函数的宽松约束

从 C++11 到 C++20 的每个 V++ 版本都放宽了对 constexpr 函数的限制。

基本上，constexpr 和 consteval 函数的限制现在如下:

- 参数和返回类型 (若有的话) 必须是文字类型。
- 函数体只能定义文字类型的变量。这些变量既不是静态的，也不是 thread_local 的。
- 不允许使用 goto 和标签。
- 只有当类没有虚基类时，构造函数和析构函数才可以是编译时函数。
- 这些函数不能用作协程。

这些函数为隐式内联。

18.4. std::is_constant_evaluated()

C++20 提供了一个新的辅助函数，允许开发者为编译时和运行时计算实现不同的代码:std::is_constant_evaluate()。其在头文件 <type_traits> 中定义 (实际上不是一个类型函数)。

其允许代码只能在运行时调用的辅助函数，再切换到可以在编译时使用的代码。例如:

comptime/isconsteval.hpp

```
#include <type_traits>
#include <cstring>

constexpr int len(const char* s)
{
  if (std::is_constant_evaluated()) {
    int idx = 0;
    while (s[idx] != '\0') { // compile-time friendly code
      ++idx;
    }
    return idx;
  }
  else {
```

```
    return std::strlen(s); // function called at runtime
  }
}
```

函数 len() 中，计算原始字符串或字符串字面值的长度。若在运行时调用，则使用标准的 C 函数 strlen()。但为了在编译时使用该函数，若在编译时上下文中，则提供不同的实现。

下面是这两个分支的调用方式:

```
constexpr int l1 = len("hello"); // uses then branch
int l2 = len("hello"); // uses else branch (no required compile-time context)
```

len() 的第一次调用发生在编译时上下文中，is_constant_evaluate() 产生 true，因此可使用 then 分支。len() 的第二次调用发生在运行时上下文中，因此_constant_evaluate() 产生 false 并调用 strlen()。即使编译器决定在编译时对调用求值，后一种情况也会发生。并且，这些调用是否需要在编译时发生。

下面是一个相反的函数: 在编译时和运行时都将整型值转换为字符串:

comptime/asstring.hpp

```
#include <string>
#include <format>

// convert an integral value to a std::string
// - can be called at compile time or runtime
constexpr std::string asString(long long value)
{
  if (std::is_constant_evaluated()) {
    // compile-time version:
    if (value == 0) {
      return "0";
    }
    if (value < 0) {
      return "-" + asString(-value);
    }
    std::string s = asString(value / 10) + std::string(1, value % 10 + '0');
    if (s.size() > 1 && s[0] == '0') { // skip leading 0 if there is one
      s.erase(0, 1);
    }
    return s;
  }
  else {
    // runtime version:
    return std::format("{}", value);
  }
}
```

运行时，只需使用 std::format()。编译时，可手动创建一个由可选的负号和所有数字组成的字

符串 (使用递归方法将它们按正确的顺序排列)。在关于将编译时字符串导出到运行时字符串的部分中，可以找到一个使用它的示例。

18.4.1 std::is_constant_evaluated() 的详情

根据 C++20 标准，std::is_constant_evaluate() 在明显为常量求值的表达式或转换中调用时产生 true。称其为:

- 在常量表达式中
- 在常量上下文中 (在 if constexpr、consteval 函数或常量初始化中)
- 用于可在编译时使用的变量的初始化器

例如:

```
constexpr bool isConstEval() {
  return std::is_constant_evaluated();
}

bool g1 = isConstEval(); // true
const bool g2 = isConstEval(); // true
static bool g3 = isConstEval(); // true
static int g4 = g1 + isConstEval(); // false

int main()
{
  bool l1 = isConstEval(); // false
  const bool l2 = isConstEval(); // true
  static bool l3 = isConstEval(); // true
  int l4 = g1 + isConstEval(); // false
  const int l5 = g1 + isConstEval(); // false
  static int l6 = g1 + isConstEval(); // false
  int l7 = isConstEval() + isConstEval(); // false
  const auto l8 = isConstEval() + 42 + isConstEval(); // true
}
```

通过 constexpr 函数间接调用 isConstEval()，也会得到相同的效果。

使用 constexpr 和 consteval 的 std::is_constant_evaluated()

例如，假设定义了以下函数:

```
bool runtimeFunc() {
  return std::is_constant_evaluated(); // always false
}

constexpr bool constexprFunc() {
  return std::is_constant_evaluated(); // may be false or true
```

```
}

consteval bool constevalFunc() {
  return std::is_constant_evaluated(); // always true
}
```

然后有以下行为:

```
void foo()
{
  bool b1 = runtimeFunc(); // false
  bool b2 = constexprFunc(); // false
  bool b3 = constevalFunc(); // true

  static bool sb1 = runtimeFunc(); // false
  static bool sb2 = constexprFunc(); // true
  static bool ab3 = constevalFunc(); // true
  const bool cb1 = runtimeFunc(); // ERROR
  const bool cb2 = constexprFunc(); // true
  const bool cb3 = constevalFunc(); // true

  int y = 42;
  static bool sb4 = y + runtimeFunc(); // function yields false
  static bool sb5 = y + constexprFunc(); // function yields false
  static bool ab6 = y + constevalFunc(); // function yields true
  const bool cb4 = y + runtimeFunc(); // function yields false
  const bool cb5 = y + constexprFunc(); // function yields false
  const bool cb6 = y + constevalFunc(); // function yields true
}
```

通过使用 constexpr 或静态 constinit(而非 const)，不带 y 的初始化将具有与 cb1、cb2 和 cb3 相同的效果。但若涉及到 y，因为涉及到运行时的值，所以使用 constexpr 或静态 constinit 总是会导致错误。

通常，在以下情况使用 std::is_constant_evaluate() 没有意义。

- 作为编译时 if 的条件，总是产生 true:

```
if constexpr (std::is_constant_evaluated()) { // always true
  ...
}
```

if constexpr 中，有一个编译时上下文，所以是否处于编译时上下文的问题的答案总是为 true(不管整个函数是从哪个上下文调用的)。

- 纯运行时函数中，这通常会产生 false。

唯一的例外是 if is_constant_evaluated() 用于局部常量求值:

```
void foo() {
  if (std::is_constant_evaluated()) { // always false
```

```
    ...
  }
  const bool b = std::is_constant_evaluated(); // true
}
```

- consteval 函数中，总为 true:

```
consteval void foo() {
  if (std::is_constant_evaluated()) { // always true
    ...
  }
}
```

使用 std::is_constant_evaluate() 通常只在 constexpr 函数中有意义。在 constexpr 函数中使用 std::is_constant_evaluate() 来调用 consteval 函数也是没有意义的，通常不允许从 constexpr 函数调用 consteval 函数:[C++23 可能会使用 if consteval 启用这样的代码。]

```
consteval int funcConstEval(int i) {
  return i;
}

constexpr int foo(int i) {
  if (std::is_constant_evaluated()) {
    return funcConstEval(i); // ERROR
  }
  else {
    return funcRuntime(i);
  }
}
```

std::is_constant_evaluated() 和三元操作符?:

C++20 标准中，有一个有趣的例子来说明如何使用 std::is_constant_evaluate()。稍作修改，如下所示:

```
int sz = 10;
constexpr bool sz1 = std::is_constant_evaluated() ? 20 : sz; // true, so 20
constexpr bool sz2 = std::is_constant_evaluated() ? sz : 20; // false, so ERROR
```

这种行为的原因如下所示:

- sz1 和 sz2 的初始化要么是静态初始化，要么是动态初始化。
- 对于静态初始化，初始化项必须是常量，所以编译器尝试用 std::is_constant_evaluate() 作为值为 true 的常量来计算初始化式。
  - 对于 sz1，这是成功的。结果是 1，这是一个常数，所以 sz1 是一个用 20 初始化的常数。

– 对于 sz2，结果是 sz，不是常数。因此，sz2(理论上) 是动态初始化的，先前的结果将丢弃，并使用 std::is_constant_evaluate() 对初始化器进行计算，结果反而产生 false，则初始化 sz2 的表达式也是 20。
但 sz2 不一定是常量，因为 std::is_constant_evaluate() 在此求值期间不一定是常量表达式，所以用这个 20 初始化 sz2 不能编译。

使用 const(而不是 constexpr) 会使情况更加棘手:

```
int sz = 10;
const bool sz1 = std::is_constant_evaluated() ? 20 : sz; // true, so 20
const bool sz2 = std::is_constant_evaluated() ? sz : 20; // false, so also 20

double arr1[sz1]; // OK
double arr2[sz2]; // may or may not compile
```

只有 sz1 是编译时常数，并且总是可以用来初始化数组，sz2 也可初始化为 20。但由于初始值不一定是常量，因此 arr2 的初始化可能会编译，也可能不会编译 (取决于所使用的编译器和优化)。

18.5. 在编译时使用堆内存、vector 和字符串

C++20 开始，编译时函数可以分配内存，前提是这些内存在编译时也释放。由于这个原因，现在可以在编译时使用字符串或 vector，但有一个重要的限制: 在编译时创建的字符串或 vector 不能在运行时使用，原因是在编译时分配的内存也必须在编译时释放。

18.5.1 在编译时使用 vector

下面是第一个例子，在编译时使用 std::vector<>:

comptime/vector.hpp

```
#include <vector>
#include <ranges>
#include <algorithm>
#include <numeric>

template<std::ranges::input_range T>
constexpr auto modifiedAvg(const T& rg)
{
  using elemType = std::ranges::range_value_t<T>;

  // initialize compile-time vector with passed elements:
  std::vector<elemType> v{std::ranges::begin(rg),
              std::ranges::end(rg)};

  // perform several modifications:
  v.push_back(elemType{});
  std::ranges::sort(v);
```

```
  auto newEnd = std::unique(v.begin(), v.end());
  ...

  // return average of modified vector:
  auto sum = std::accumulate(v.begin(), newEnd,
                             elemType{});
  return sum / static_cast<double>(v.size());
}
```

这里，用 constexpr 定义 modifiedAvg()，以便在编译时调用。在内部，使用 std::vector<> v，用传递的范围的元素初始化，可以使用 vector 的全部 API(特别是插入和删除元素)。例如，插入一个元素，对元素排序，并使用 unique() 删除连续的重复项。所有这些算法现在都是 constexpr 的，因此可以在编译时使用。

但在最后，只返回编译时 vector 下计算出来的值。

可以在编译时调用这个函数:

comptime/vector.cpp

```
#include "vector.hpp"
#include <iostream>
#include <array>

int main()
{
  constexpr std::array orig{0, 8, 15, 132, 4, 77};

  constexpr auto avg = modifiedAvg(orig);
  std::cout << "average: " << avg << '\n';
}
```

由于 avg 是用 constexpr 声明的，所以 modiedavg() 在编译时求值。

也可以用 consteval 声明 modifiedAvg()，因为不会在运行时复制元素，所以可以按值传递参数:

```
template<std::ranges::input_range T>
consteval auto modifiedAvg(T rg)
{
  using elemType = std::ranges::range_value_t<T>;

  // initialize compile-time vector with passed elements:
  std::vector<elemType> v{std::ranges::begin(rg),
              std::ranges::end(rg)};
  ...
}
```

然而，若 vector 可以在运行时使用，则仍然不能在编译时声明和初始化:

```
int main()
{
```

```
  constexpr std::vector orig{0, 8, 15, 132, 4, 77}; // ERROR
  ...
}
```

出于同样的原因，编译时函数只能在编译时使用返回值时返回一个 vector:

comptime/returnvector.cpp

```
#include <vector>

constexpr auto returnVector()
{
  std::vector<int> v{0, 8, 15};
  v.push_back(42);
  ...

  return v;
}

constexpr auto returnVectorSize()
{
  auto coll = returnVector();
  return coll.size();
}

int main()
{
  // constexpr auto coll = returnVector(); // ERROR
  constexpr auto tmp = returnVectorSize(); // OK
  ...
}
```

18.5.2 编译时返回集合

虽然不能返回编译时的 vector，以便在运行时使用，但有一种方法可以返回编译时计算的元素集合: 可以返回 std::array<>。唯一的问题是需要知道数组的大小，因为大小不能通过 vector 的大小初始化:

```
std::vector v;
...
std::array<int, v.size()> arr; // ERROR
```

这样的代码永远不会编译，因为 size() 是一个运行时值，而 arr 的声明需要一个编译时值。这段代码是否在编译时求值并不重要，所以必须返回一个固定大小的数组。例如:

comptime/mergevalues.hpp

```
#include <vector>
#include <ranges>
```

```cpp
#include <algorithm>
#include <array>

template<std::ranges::input_range T>
consteval auto mergeValues(T rg, auto... vals)
{
  // create compile-time vector from passed range:
  std::vector<std::ranges::range_value_t<T>> v{std::ranges::begin(rg),
                          std::ranges::end(rg)};
  (... , v.push_back(vals)); // merge all passed parameters
  std::ranges::sort(v); // sort all elements

  // return extended collection as array:
  constexpr auto sz = std::ranges::size(rg) + sizeof...(vals);
  std::array<std::ranges::range_value_t<T>, sz> arr{};
  std::ranges::copy(v, arr.begin());
  return arr;
}
```

在 consteval 函数中使用 vector 将传入的可变数量的实参与传入范围的元素合并，并对其全排序。但可将结果集合作为 std::array 返回，以便将其传递给运行时上下文。例如，下面的程序可以很好地使用这个函数:

comptime/mergevalues.cpp

```cpp
#include "mergevalues.hpp"
#include <iostream>
#include <array>

int main()
{
  // compile-time initialization of array:
  constexpr std::array orig{0, 8, 15, 132, 4, 77, 3};

  // initialization of sorted extended array:
  auto merged = mergeValues(orig, 42, 4);

  // print elements:
  for(const auto& i : merged) {
    std::cout << i << ' ';
  }
}
```

其输出如下:

```
0 3 4 4 8 15 42 77 132
```

若在编译时不知道数组的结果大小，则必须声明具有最大大小的返回数组，并返回结果大小。合并这些值的函数现在可能如下所示:

comptime/mergevaluessz.hpp

```cpp
#include <vector>
#include <ranges>
#include <algorithm>
#include <array>

template<std::ranges::input_range T>
consteval auto mergeValuesSz(T rg, auto... vals)
{
  // create compile-time vector from passed range:
  std::vector<std::ranges::range_value_t<T>> v{std::ranges::begin(rg),
                          std::ranges::end(rg)};

  (... , v.push_back(vals)); // merge all passed parameters

  std::ranges::sort(v); // sort all elements

  // return extended collection as array and its size:
  constexpr auto maxSz = std::ranges::size(rg) + sizeof...(vals);
  std::array<std::ranges::range_value_t<T>, maxSz> arr{};
  auto res = std::ranges::unique_copy(v, arr.begin());
  return std::pair{arr, res.out - arr.begin()};
}
```

此外，可使用 std::ranges::unique_copy() 来删除排序后的连续重复项，并返回数组和结果元素的数量。

注意，应该用 {} 声明 arr，以确保数组中的所有值都初始化，编译时函数不允许产生未初始化的内存。

现在可以这样使用返回值:

comptime/mergevaluessz.cpp

```cpp
#include "mergevaluessz.hpp"
#include <iostream>
#include <array>
#include <ranges>

int main()
{
  // compile-time initialization of array:
  constexpr std::array orig{0, 8, 15, 132, 4, 77, 3};

  // initialization of sorted extended array:
  auto tmp = mergeValuesSz(orig, 42, 4);
  auto merged = std::views::counted(tmp.first.begin(), tmp.second);

  // print elements:
  for(const auto& i : merged) {
```

```
    std::cout << i << ' ';
  }
}
```

通过使用视图适配器 std::views::counted()，可以将返回的数组和返回的元素大小结合起来，作为一个范围在数组中使用。

现在程序的输出为:

```
0 3 4 8 15 42 77 132
```

18.5.3 编译时使用字符串

对于编译时字符串，现在所有的操作都是 constexpr，所以现在可以在编译时使用 std::string，也可以使用其他字符串类型，例如 std::u8string。

但这也有一个限制，即不能在运行时使用编译时字符串。例如:

```
consteval std::string returnString()
{
  std::string s = "Some string from compile time";
  ...
  return s;
}

void useString()
{
  constexpr auto s = returnString(); // ERROR
  ...
}

constexpr void useStringInConstexpr()
{
  std::string s = returnString(); // ERROR
  ...
}

consteval void useStringInConsteval()
{
  std::string s = returnString(); // OK
  ...
}
```

不能通过使用 data() 或 c_str() 或 std::string_view 返回编译时字符串来解决这个问题，可将返回在编译时分配的内存地址。若发生这种情况，编译器会引发编译时错误。

然而，可以使用与上面描述的 vector 相同的技巧。可以将 string 对象转换为固定大小的数组，并返回数组和 vector 对象的大小。

下面是一个完整的例子:

comptime/comptimestring.cpp

```cpp
#include <iostream>
#include <string>
#include <array>
#include <cassert>
#include "asstring.hpp"

// function template to export a compile-time string to runtime:
template<int MaxSize>
consteval auto toRuntimeString(std::string s)
{
  // ensure the size of the exported array is large enough:
  assert(s.size() <= MaxSize);

  // create a compile-time array and copy all characters into it:
  std::array<char, MaxSize+1> arr{}; // ensure all elems are initialized
  for (int i = 0; i < s.size(); ++i) {
    arr[i] = s[i];
  }

  // return the compile-time array and the string size:
  return std::pair{arr, s.size()};
}

// function to import an exported compile-time string at runtime:
std::string fromComptimeString(const auto& dataAndSize)
{
  // init string with exported array of chars and size:
  return std::string{dataAndSize.first.data(),
              dataAndSize.second};
}

// test the functions:
consteval auto comptimeMaxStr()
{
  std::string s = "max int is " + asString(std::numeric_limits<int>::max())
          + " (" + asString(std::numeric_limits<int>::digits + 1)
          + " bits)";
  return toRuntimeString<100>(s);
}

int main()
{
  std::string s = fromComptimeString(comptimeMaxStr());
  std::cout << s << '\n';
}
```

同样，这里定义了两个辅助函数来将编译时字符串导出为运行时字符串:

- 编译时函数 toRuntimeString() 将字符串转换为 std::array<>，并返回该数组和字符串的大小:

```
template<int MaxSize>
consteval auto toRuntimeString(std::string s)
{
  assert(s.size() <= MaxSize); // ensure array size fits

  // create a compile-time array and copy all characters into it:
  std::array<char, MaxSize+1> arr{}; // ensure all elems are initialized
  for (int i = 0; i < s.size(); ++i) {
    arr[i] = s[i];
  }

  return std::pair{arr, s.size()}; // return array and size
}
```

使用 assert()，再次检查数组的大小是否足够大。以同样的方式，可以在编译时确定有没有浪费太多的内存。

- 运行时函数 fromComptimeString() 接受返回的数组和大小，初始化运行时字符串并返回:

```
std::string fromComptimeString(const auto& dataAndSize)
{
  return std::string{dataAndSize.first.data(),
                     dataAndSize.second};
}
```

该函数的测试用例使用辅助函数 asString()，可以在编译时和运行时使用将整型值转换为字符串。

该程序有以下输出:

```
max int is 2147483647 (32 bits)
```

18.6. 其他 constexpr 扩展

除了在编译时使用堆内存的能力之外，编译时函数 (无论是用 constexpr 还是 consteval 声明) 自 C++20 以来还可以使用一些额外的语言特性，所以现在还可以在编译时使用一些库特性。

18.6.1 constexpr 的语言扩展

自 C++20 起，以下语言特性可以在编译时函数中使用 (无论是用 constexpr 还是 consteval 声明):

- 现在可以在编译时使用堆内存。
- 支持运行时多态性:
  - 可以使用虚函数。
  - 可以使用 dynamic_cast。
  - 可以使用 typeid。

- 现在可以使用 try-catch 语句块了 (但仍然不允许抛出异常)。
- 现在可以更改联合的活动成员。

注意，仍然不允许在 constexpr 或 consteval 函数中使用 static。

18.6.2 constexpr 的库扩展

C++ 标准库扩展了可以在编译时使用的工具。

constexpr 算法和工具

<algorithm>、<numeric> 和 <utility> 中的大多数算法现在都是 constexpr。所以，现在可以在编译时对元素进行排序和累加 (参见在编译时使用 vector 的示例)。

然而，并行算法 (具有执行策略参数的算法) 仍然只能在运行时使用。

constexpr 的库类型

几种类型的 C++ 标准库现在更好地支持 constexpr，以便在编译时可以 (更好地) 使用对象:

- vector 和字符串现在可以在编译时使用。
- 一些 std::complex<> 操作变成了 constexpr。
- std::optional<> 和 std::variant<> 中添加了两个缺失的 constexpr。
- constexpr 添加到 std::invoke()，std::ref()，std::cref()，mem_fn()，not_fn()，std::bind() 和 std::bind_front() 中，
- pointer_traits 中，用于原始指针的 pointer_to() 现在可以在编译时使用。

18.7. 附注

关键字 constinit 最初是由 Eric Fiselier 在http://wg21.link/p1143r0中作为属性提出。最终接受的提案是由 Eric Fiselier 在http://wg21.link/p1143r2中提出。

关键词 consteval 最早是由 Richard Smith, Andrew Sutton 和 Daveed Vandevoorde 在http://wg21.link/p1073r0上提出。最终接受的提案是由 Richard Smith、Andrew Sutton 和 Daveed Vandevoorde 在http://wg21.link/p1073r3上制定。David Stone 后来在http://wg21.link/p1937r2中进行了一个小的修复。

std::is_constant_evaluate() 首先由 Daveed Vandevoorde 在http://wg21.link/p0595r0中作为 constexpr 操作符提出。最终接受的提案是由 Richard Smith、Andrew Sutton 和 Daveed Vandevoorde 在http://wg21.link/p0595r2上制定。

编译时容器 (如 vector 和 string) 最早是由 Daveed Vandevoorde 在http://wg21.link/p0597中提出。最终接受的语言特征是由 Peter Dimov、Louis Dionne、Nina Ranns、Richard Smith 和 Daveed Vandevoorde 在http://wg21.link/p0784r7上描述。最终被接受的库特征是由 Antony Polukhin(http://wg21.link/p0858r0) 和 Louis Dionne(http://wg21.link/p1004r2) 提出，当然还有http://wg21.link/00980r1。

第 19 章 非类型模板参数 (NTTP) 扩展

C++ 模板形参不必仅仅是类型，也可以是值 (非类型模板参数 (NTTP))，但这些值的类型有限制。C++20 增加了更多可用于非类型模板形参的类型。浮点值、数据结构对象 (如 std::pair<> 和 std::array<>) 和简单类，甚至 Lambda 现在都可以作为模板参数传递。本章将介绍这些特性的类型和相关的应用。

注意，非类型模板参数还有另一个新特性: 从 C++20 开始，可以使用概念来约束 NTTP 的值。

19.1. 非类型模板参数的新类型

自 C++20 起，可以为非类型模板形参使用新类型

- 浮点类型 (如 double)
- 结构和简单类 (如 std::pair<>)，这也间接地允许使用字符串字面值作为模板参数
- Lambda

事实上，非类型模板形参现在可能都是结构类型。结构类型是一种类型

- 可以是 (const 或 volatile 限定的) 算术类型、枚举类型或指针类型
- 或为左值引用类型
- 或为文字类型 (要么是聚合类型，要么有 constexpr 构造函数，没有复制/移动构造函数，没有析构函数，没有复制/移动构造函数或析构函数，并且每个数据成员的初始化都是常量表达式)，其中:
  - 所有非静态成员都是 public 且不可变的，并且只使用结构类型或其数组
  - 所有基类 (若有的话) 都是 public 继承的，结构类型也是如此

来看看这个新类型如何使用。

19.1.1 浮点值作为非类型模板形参

考虑下面的例子:

lang/nttpdouble.cpp

```
#include <iostream>
#include <cmath>

template<double Vat>
int addTax(int value)
{
  return static_cast<int>(std::round(value * (1 + Vat)));
}

int main()
{
  std::cout << addTax<0.19>(100) << '\n';
  std::cout << addTax<0.19>(4199) << '\n';
```

```
  std::cout << addTax<0.07>(1950) << '\n';
}
```

程序输出如下:

```
119
4997
2087
```

addTax() 的声明如下:

```
template<double Vat>
int addTax(int value)
```

函数模板 addTax() 接受一个 double 作为模板参数，然后将其用作增值税，将其添加到整数值中。

当使用 auto 声明非类型模板形参时，现在也允许传递浮点值:

```
template<auto Vat>
int addTax(int value)
{
  ...
}

std::cout << addTax<0>(1950) << '\n'; // Vat is the int value 0
std::cout << addTax<0.07>(1950) << '\n'; // Vat is the double value 0.07
```

同样，现在可以在类模板中使用浮点值 (声明为 double 或 auto):

```
template<double Vat>
class Tax {
  ...
};
```

处理不精确的浮点值

由于舍入误差，浮点类型的值最终会略微不精确，处理浮点值作为模板参数时会产生影响，当模板的两个实例化具有相同的类型时会有问题。

考虑下面的例子:

lang/nttpdouble2.cpp

```
#include <iostream>
#include <limits>
#include <type_traits>

```

```cpp
template<double Val>
class MyClass {
};

int main()
{
  std::cout << std::boolalpha;
  std::cout << std::is_same_v<MyClass<42.0>, MyClass<17.7>> // always false
        << '\n';
  std::cout << std::is_same_v<MyClass<42.0>, MyClass<126.0 / 3>> // true or false
        << '\n';
  std::cout << std::is_same_v<MyClass<42.7>, MyClass<128.1/ 3>> // true or false
        << "\n\n";
  std::cout << std::is_same_v<MyClass<0.1 + 0.3 + 0.00001>,
  MyClass<0.3 + 0.1 + 0.00001>> // true or false
        << '\n';
  std::cout << std::is_same_v<MyClass<0.1 + 0.3 + 0.00001>,
  MyClass<0.00001 + 0.3 + 0.1>> // true or false
        << "\n\n";
  constexpr double NaN = std::numeric_limits<double>::quiet_NaN();
  std::cout << std::is_same_v<MyClass<NaN>, MyClass<NaN>> // always true
        << '\n';
}
```

该程序的输出取决于平台。通常是这样的:

```
false
true
false
---
true
false
---
true
```

为 NaN 实例化的模板总是具有相同的类型，即使 NaN == NaN 为 false。

19.1.2 对象作为非类型模板形参

C++20 起，可以使用数据结构或类的对象/值作为非类型模板形参，前提是所有成员都是 public 且类型为字面类型。

考虑下面的例子:

lang/nttpstruct.cpp

```cpp
#include <iostream>
#include <cmath>
#include <cassert>
```

```
4
5  struct Tax {
6    double value;
7
8    constexpr Tax(double v)
9    : value{v} {
10     assert(v >= 0 && v < 1);
11   }
12
13   friend std::ostream& operator<< (std::ostream& strm, const Tax& t) {
14     return strm << t.value;
15   }
16 };
17
18 template<Tax Vat>
19 int addTax(int value)
20 {
21   return static_cast<int>(std::round(value * (1 + Vat.value)));
22 }
23
24 int main()
25 {
26   constexpr Tax tax{0.19};
27   std::cout << "tax: " << tax << '\n';
28
29   std::cout << addTax<tax>(100) << '\n';
30   std::cout << addTax<tax>(4199) << '\n';
31   std::cout << addTax<Tax{0.07}>(1950) << '\n';
32 }
```

这里，声明了一个具有 public 成员的文本数据结构 Tax，一个 constexpr 构造函数和一个额外的成员函数:

```
1  struct Tax {
2    double value;
3    constexpr Tax(double v) {
4      ...
5    }
6    friend std::ostream& operator<< (std::ostream& strm, const Tax& t) {
7      ...
8    }
9  };
```

这允许将这种类型的对象作为模板参数传递:

```
1  constexpr Tax tax{0.19};
2  std::cout << "tax: " << tax << '\n';
3  std::cout << addTax<tax>(100) << '\n'; // pass Tax object as a template argument
```

若数据结构或类是结构类型，则此操作有效:

- 所有非静态成员都是 public 且不可变的，并且只使用结构类型或其数组
- 所有基类 (若有的话) 都是 public 继承的，结构类型也是如此
- 该类型是文字类型 (要么是聚合类型，要么有 constexpr 构造函数，没有复制/移动构造函数，没有析构函数，没有复制/移动构造函数或析构函数，其中每个数据成员的初始化都是常量表达式)。

例如:

lang/nttpstruct2.cpp

```
#include <iostream>
#include <array>

constexpr int foo()
{
  return 42;
}

struct Lit {
  int x = foo(); // OK because foo() is constexpr
  int y;
  constexpr Lit(int i) // OK because constexpr
    : y{i} {
  }
};

struct Data {
  int i;
  std::array<double,5> vals;
  Lit lit;
};

template<auto Obj>
void func()
{
  std::cout << typeid(Obj).name() << '\n';
}

int main()
{
  func<Data{42, {1, 2, 3}, 42}>(); // OK

  constexpr Data d2{1, {2}, 3};
  func<d2>();
}
```

若 Type 的构造函数或 foo() 不是 constexpr，或者使用 std::string 成员，则不能使用 Type。

std::pair<> 和 std::array<> 作为非类型模板形参

可以使用 std::pair<> 和 std::array<> 类型的编译时对象作为模板形参:

[C++ 必须添加额外的要求，std::pair<> 和 std::array<> 不能用 private 基类实现，这是 C++20 之前的一些实现者所做的 (参见http://wg21.link/lwg3382)。]

```
template<auto Val>
class MyClass {
  ...
};

MyClass<std::pair{47,11}> mcp; // OK since C++20
MyClass<std::array{0, 8, 15}> mca; // OK since C++20
```

字符串作为非类型模板形参

将字符数组作为公共成员的数据结构是结构类型，就可以很容易地将字符串字面量作为模板参数传递。

例如:

lang/nttpstring.cpp

```
#include <iostream>
#include <string_view>

template<auto Prefix>
class Logger {
  ...
public:
  void log(std::string_view msg) const {
    std::cout << Prefix << msg << '\n';
  }
};

template<std::size_t N>
struct Str {
  char chars[N];
  const char* value() {
    return chars;
  }
  friend std::ostream& operator<< (std::ostream& strm, const Str& s) {
    return strm << s.chars;
  }
};
template<std::size_t N> Str(const char(&)[N]) -> Str<N>; // deduction guide

```

```
int main()
{
  Logger<Str{"> "}> logger;
  logger.log("hello");
}
```

该程序有以下输出:

```
> hello
```

19.1.3 Lambda 作为非类型模板形参

因为 Lambda 只是函数对象的引用，现在只要 Lambda 能在编译时使用，也可以用作非类型模板参数。

考虑下面的例子:

lang/nttplambda.cpp

```
#include <iostream>
#include <cmath>

template<std::invocable auto GetVat>
int addTax(int value)
{
  return static_cast<int>(std::round(value * (1 + GetVat())));
}

int main()
{
  auto getDefaultTax = [] {
    return 0.19;
  };
  std::cout << addTax<getDefaultTax>(100) << '\n';
  std::cout << addTax<getDefaultTax>(4199) << '\n';
  std::cout << addTax<getDefaultTax>(1950) << '\n';
}
```

函数模板 addTax() 使用了一个辅助函数，也可以是 Lambda:

```
template<std::invocable auto GetVat>
int addTax(int value)
{
  return static_cast<int>(std::round(value * (1 + GetVat())));
}
```

现在可以将 Lambda 传递给这个函数模板:

```
auto getDefaultTax = [] {
  return 0.19;
};

addTax<getDefaultTax>(100) // passes lambda as template argument
```

甚至可以在调用函数模板时直接定义 Lambda:

```
addTax<[]{ return 0.19; }>(100) // passes lambda as template argument
```

用 std::invocable 或 std::regular_invocable 这两个概念约束模板参数是个好主意，就可以记录并确保可以使用指定类型的参数调用传递的可调用对象。

通过使用 std::invocable auto，要求可调用对象不接受任何参数。若传递的可调用对象应该接受参数，则需要:

```
template<std::invocable<std::string> auto GetVat>
int addTax(int value, const std::string& name)
{
  double vat = GetVat(name); // get VAT according to the passed name
  ...
}
```

函数模板的声明中不能跳过 auto，使用 std::invocable 作为传递的值/对象/回调具有类型约束:

```
template<std::invocable auto GetVat> // GetVat is a callable with constrained type
```

若没有 auto，将声明一个具有普通类型形参的函数模板，并对其进行约束:

```
template<std::invocable GetVat> // GetVat is a constrained type
```

若用 Lambda 的具体类型声明函数模板也可以 (必须首先定义 Lambda):

```
auto getDefaultTax = [] {
  return 0.19;
};

template<decltype(getDefaultTax) GetVat>
int addTax(int value)
{
  return static_cast<int>(std::round(value * (1 + GetVat())));
}
```

以下使用 Lambda 作为非类型模板形参的约束:

- Lambda 可能没有捕获任何东西。

- 必须能够在编译时使用 Lambda。

幸运的是，从 C++17 开始，Lambda 都是隐式的 constexpr，只要它使用对编译时计算有效的特性。或者，可以使用 constexpr 或 consteval 声明 Lambda，以便在 Lambda 内部捕获错误，而不是在使用无效编译时特性时将其用作模板参数。

19.2. 附注

对更多非类型模板形参的要求从第一个 C++ 标准开始就存在。我们已经在 C++ 模板-完整指南的第一版中讨论过 (参见http://tmplbook.com)。

允许非类型模板参数的任意文字类型最早是由 Jens Maurer 在http://wg21.link/n3413中提出。允许类对象作为非类型模板参数是由 Jeff Snyder 在http://wg21.link/p0732r0中为 C++20 提出。随后，Jorg Brown 在http://wg21.link/p1714r0中提出了允许浮点值作为非类型模板参数的 C++20。

最终接受的提案是由 Jeff Snyder 和 Louis Dionne 在http://wg21.link/p0732r2和 Jens Maurer 在http://wg21.link/p1907r1提出。

第 20 章 新类型特征

本章将介绍 C++20 在其标准库中引入的几个类型特征 (以及两个用于类型的底层函数)。“新类型特征”表列出了 C++20 的这些新类型特征 (都定义在命名空间 std 中)。

特征	效果
is_bounded_array_v<T>	若类型 T 是已知范围的数组类型，则返回 true
is_unbounded_array_v<T>	若类型 T 是范围未知的数组类型，则返回 true
is_nothrow_convertible_v<T, T2>	若类型 T 可转换为类型 T2 而不抛出，则返回 true
is_layout_compatible_v<T1, T2>	若 T1 与类型 T2 的布局兼容，则返回 true
is_pointer_interconvertible... _base_of_v<BaseT, DerT>	若指向 DerT 的指针可以安全地转换为指向其基类型 BaseT 的指针，则返回 true
remove_cvref_t<T>	产生类型 T，没有引用，const, volatile
unwrap_reference_t<T>	若类型 T 是 std::reference_wrapper<>，则返回 T 的包装类型，否则为 T
unwrap_ref_decay_t<T>	若类型 T 是 std::reference_wrapper<>，则返回 T 的包装类型，否则返会 T 的衰退类型
common_reference_t<T...>	产生通用类型 pf 所有类型 T…可以为其赋值
type_identity_t<T>	生成类型 T
iter_difference_t<T>	生成可增量/迭代器类型的差异类型 T
iter_value_t<T>	返回指针/迭代器类型 T 的值/元素类型
iter_reference_t<T>	生成指针/迭代器类型的引用类型 T
iter_rvalue_reference_t<T>	生成指针/迭代器类型 T 的右值引用类型
is_clock_v<T>	若 T 是时钟类型，则返回 true
compare_three_way_result_t<T>	使用操作符 <=> 比较两个值的类型。

表 20.1 新类型特征

以下章节将详细讨论这些特征，除了以下特征:

- std::is_clock_v<> 将在关于新计时特性的部分中讨论。
- std::compare_three_way_result_t<> 将在有关新三路比较的一节中讨论。

另外请注意，类型特性 std::is_pod<> 在 C++20 中已弃用。

20.1. 用于类型分类的新类型特征

20.1.1 is_bounded_array_v<> and is_unbounded_array_v

std::is_bounded_array_v<T> std::is_unbounded_array_v<T>

确定类型 T 是有界/无界数组 (已知范围/未知范围)。

例如:

```
int a[5];
std::is_bounded_array_v<decltype(a)> // true
std::is_unbounded_array_v<decltype(a)> // false

extern int b[];
std::is_bounded_array_v<decltype(b)> // false
std::is_unbounded_array_v<decltype(b)> // true
```

20.2. 用于类型检查的新类型特征

20.2.1 is_nothrow_convertible_v<>

std::is_nothrow_convertible_v<From, To>

生成类型 From 是否可转换为类型 To，并保证不会抛出任何异常。

例如:

```
// char* to std::string:
std::is_convertible_v<char*, std::string> // true
std::is_nothrow_convertible_v<char*, std::string> // false

// std::string to std::string_view:
std::is_convertible_v<std::string, std::string_view> // true
std::is_nothrow_convertible_v<std::string, std::string_view> // true
```

20.3. 用于类型转换的新类型特征

20.3.1 remove_cvref_t<>

std::remove_cvref_t<T>

生成类型 T，而不是引用、const 或 volatile。

表达式为

```
std::remove_cvref_t<T>
```

等价于：

```
std::remove_cv_t<remove_reference_t<T>>
```

例如:

```
std::remove_cvref_t<const std::string&> // std::string
std::remove_cvref_t<const char* const> // const char*
```

20.3.2 unwrap_reference<> and unwrap_ref_decay_t

std::unwrap_reference_t<T>

若 T 是 std::reference_wrapper<>(使用 std::ref() 或 std::cref() 创建)，则返回 T 的包装类型，否则为 T。

std::unwrap_ref_decay_t<T>

若 T 是 std::reference_wrapper<>(使用 std::ref() 或 std::cref() 创建)，则返回已封装的 T 类型，否则返回已封装的 T 类型。

例如:

```
std::unwrap_reference_t<decltype(std::ref(s))> // std::string&
std::unwrap_reference_t<decltype(std::cref(s))> // const std::string&
std::unwrap_reference_t<decltype(s)> // std::string
std::unwrap_reference_t<decltype(s)&> // std::string&
std::unwrap_reference_t<int[4]> // int[4]

std::unwrap_ref_decay_t<decltype(std::ref(s))> // std::string&
std::unwrap_ref_decay_t<decltype(std::cref(s))> // const std::string&
std::unwrap_ref_decay_t<decltype(s)> // std::string
std::unwrap_ref_decay_t<decltype(s)&> // std::string
std::unwrap_ref_decay_t<int[4]> // int*
```

20.3.3 common_reference<>_t

std::common_reference_t<T...>

生成所有类型 T…的通用类型 (若有的话)，可以给它赋值。因此，给定类型 T1、T2 和 T3, 特性生成的类型可以同时赋值这三种类型。理想情况下，是引用类型。但若涉及到类型转换并创建临时对象，则为值类型。

例如:

```
std::common_reference_t<int&, int> // int
std::common_reference_t<int&, int&> // int&
std::common_reference_t<int&, int&&> // const int&
```

```
std::common_reference_t<int&&, int&&> // int&&
std::common_reference_t<int&, double> // double
std::common_reference_t<int&, double&&> // double
std::common_reference_t<char*, std::string, std::string_view> // std::string_view
std::common_reference_t<char, std::string> // ERROR
```

20.3.4 type_identity_t<>

std::type_identity_t<T>

只生成 T 类型。

这个类型特性有大量的用例:

- 可以禁用使用参数来推断模板参数。例如:

```
template<typename T>
void insert(std::vector<T>& coll, const std::type_identity_t<T>& value)
{
  coll.push_back(value);
}

std::vector<double> coll;
...
insert(coll, 42); // OK: type of 42 not used to deduce type T
```

若只使用 const T& 声明形参值，则编译器会引发错误，其会为类型 T 推断出两种不同的类型。

- 可以使用它作为构建块来定义生成类型的类型特征，可以简单地定义一个类型特性来删除常量化:[可参见 CppCon 2014 (http://www.youtube.com/watch?v=Am2is2QCvxY) 上 Walter E. Brown 的演讲现代模板元编程]。

```
template<typename T>
struct remove_const : std::type_identity<T> {
};

template<typename T>
struct remove_const<const T> : std::type_identity<T> {
};
```

20.4. 新迭代器的类型特征

本节列出了迭代器的特征，这些特征在关于迭代器的基本类型函数一节中列出。

20.4.1 iter_difference_t<>

std::iter_difference_t<T>

生成与可递增类型/迭代器类型 T 对应的差值类型。

该特性特别用于处理间接可读类型的两个对象的值类型，与传统的迭代器特征 (std::iterator_traits<>) 相比，这个特征可以正确地处理新的迭代器类别。

与名为 type 的成员没有相应的数据结构 std::iter_difference。相反，类型特性是通过尝试使用新辅助类型 std::incrementable_traits<> 的成员 difference_type 来定义的，该成员 difference_type 已经定义如下:

- 若特化，则使用 std::incrementable_traits<T>::difference_type。
- 对于原始指针，使用 std::ptrdiff_t。
- 否则，若定义了，则使用 T::difference_type。
- 否则，使用两个 T 之间的有符号整数差分类型。
- 对于 const T，使用 T 的不同类型。

例如:

```
using T1 = std::iter_difference_t<int*>; // std::ptrdiff_t
using T2 = std::iter_difference_t<std::string>; // std::ptrdiff_t
using T3 = std::iter_difference_t<std::vector<long>>; // std::ptrdiff_t
using T4 = std::iter_difference_t<int>; // int
using T5 = std::iter_difference_t<std::chrono::sys_seconds>; // ERROR
```

20.4.2 iter_value_t<>

std::iter_value_t<T>

生成非 const 值/元素类型，对应指针/迭代器类型 T。

该特性特别用于处理间接可读类型的值类型，与传统的迭代器特性 (std::iterator_traits<>) 相比，这个特性可以正确处理新迭代器的类别。

名为 type 的成员中没有相应的数据结构 std::iter_value。相反，类型特性是通过尝试使用新的辅助类型 std::indirectly_readable_traits<> 的成员 value_type 来定义的，该成员值已经定义如下:

- 若特化，则使用 std::indirectly_readable_traits<T>::value_type。
- 对于原始指针，使用引用的非 const/volatile 类型。
- 否则，若定义了，则使用 remove_cv_t<T::value_type>。
- 否则，若定义了，则使用 remove_cv_t<T::element_type>。
- 对于 const T，使用 T 的值类型。

例如:

```cpp
using T1 = std::iter_value_t<int*>; // int
using T2 = std::iter_value_t<const int* const>; // int
using T3 = std::iter_value_t<std::string>; // char
using T4 = std::iter_value_t<std::vector<long>>; // long
using T5 = std::iter_value_t<int>; // ERROR
```

20.4.3 iter_reference_t<> and iter_rvalue_reference_t<>

std::iter_reference_t<T>

生成对应于可解引用指针/迭代器类型 T 的左值引用类型。等价于：

```cpp
decltype(*declval<T&>())
```

std::iter_rvalue_reference_t<T>

产生对应于可解引用指针/迭代器类型 T 的右值类型

```cpp
decltype(std::ranges::iter_move(declval<T&>()))
```

这些特性是专门用来处理间接可写类型的值类型的，与传统的迭代器特征 (std::iterator_traits<>) 不同，这些特征可以正确处理新的迭代器类别。

没有带有类型成员的相应数据结构 std::iter_value。相反，这些特征是按照上面描述的那样直接定义。

例如:

```cpp
using T1 = std::iter_reference_t<int*>; // int&
using T2 = std::iter_reference_t<const int* const>; // const int&
using T3 = std::iter_reference_t<std::string>; // ERROR
using T4 = std::iter_reference_t<std::vector<long>>; // ERROR
using T5 = std::iter_reference_t<int>; // ERROR

using T6 = std::iter_rvalue_reference_t<int*>; // int&&
using T7 = std::iter_rvalue_reference_t<const int* const>; // const int&&
using T8 = std::iter_rvalue_reference_t<std::string>; // ERROR
```

20.5. 用于布局兼容性的类型特征和函数

一些情况下，了解两个类型或指向类型的指针是否可以安全地相互转换很重要。C++ 标准使用术语“布局兼容”来表示这一点。

为了检查类成员之间的布局兼容关系，C++20 还引入了两个新的普通函数，可以在运行时上下文中使用。

20.5.1 is_layout_compatible_v<>

std::is_layout_compatible_v<T1, T2>

生成类型 T1 和 T2 是否与布局兼容，以便可以使用 reinterpret_cast 安全地转换指向它们的指针。

例如:

```
struct Data {
  int i;
  const std::string s;
};

class Type {
  private:
  const int id = nextId();
  std::string name;
  public:
  ...
};

std::is_layout_compatible_v<Data, Type> // true
```

对于布局兼容性来说，仅仅是类型和位的粗略匹配完全不够。根据语言规则:

- 有符号类型永远不会与无符号类型的布局兼容。
- char 甚至不兼容有符号 char 和无符号 char 的布局。
- 引用与非引用的布局永不兼容。
- 不同 (甚至是布局兼容) 类型的数组永远不与布局兼容。
- 枚举类型永远不会与其基础类型的布局兼容。

例如:

```
enum class E {};
enum class F : int {};

std::is_layout_compatible_v<E, F> // true
std::is_layout_compatible_v<E[2], F[2]> // false
std::is_layout_compatible_v<E, int> // false

std::is_layout_compatible_v<char, char> // true
std::is_layout_compatible_v<char, signed char> // false
std::is_layout_compatible_v<char, unsigned char> // false
std::is_layout_compatible_v<char, char&> // false
```

20.5.2 is_pointer_interconvertible_base_of_v<>

std::is_pointer_interconvertible_base_of<Base, Der>

若使用 reinterpret_cast 可以安全地将指向 Der 类型的指针，并转换为指向其基类型 base 的指针，则返回 true。

若两者具有相同的类型，则该特性返回 true。

例如:

```
struct B1 { };
struct D1 : B1 { int x; };

struct B2 { int x; };
struct D2 : B2 { int y; }; // no standard-layout type

std::is_pointer_interconvertible_base_of_v<B1, D1> // true
std::is_pointer_interconvertible_base_of_v<B2, D2> // false
```

指向 D2 的指针不能安全地转换为指向 B2 的指针，原因是它不是标准布局类型，因为并非所有成员都定义在具有相同访问权限的同一类中。

20.5.3 is_corresponding_member()

```
template<typename S1, typename S2, typename M1, typename M2>
constexpr bool is_corresponding_member(M1 S1::*m1, M2 S2::*m2) noexcept;
```

返回 m1 和 m2 是否指向 S1 和 S2 的布局兼容成员，所以这些成员和这些成员前面的所有成员必须布局兼容。当且仅当 S1 和 S2 是标准布局类型，M1 和 M2 是对象类型，且 M1 和 M2 不为空时，函数返回 true。

例如:

```
struct Point2D { int a; int b; };
struct Point3D { int x; int y; int z; };
struct Type1 { const int id; int val; std::string name; };
struct Type2 { unsigned int id; int val; };

std::is_corresponding_member(&Point2D::b, &Point3D::y) // true
std::is_corresponding_member(&Point2D::b, &Point3D::z) // false (2nd versus 3rd int)
std::is_corresponding_member(&Point2D::b, &Type1::val) // true
std::is_corresponding_member(&Point2D::b, &Type2::val) // false (signed vs. unsigned)
```

20.5.4 is_pointer_interconvertible_with_class()

```
template<typename S, typename M>
constexpr bool is_pointer_interconvertible_with_class(M S::*m) noexcept;
```

返回类型 s 的每个对象，s 是否与其子对象的 s 指针可互换。当且仅当 S 是标准布局类型，M 是对象类型，且 M 不为空时，该函数返回 true。

例如:

```
struct B1 { int x; };
struct B2 { int y; };

struct DB1 : B1 {};
struct DB1B2 : B1, B2 {}; // not a standard-layout type

std::is_pointer_interconvertible_with_class<B1, int>(&DB1::x) // true
std::is_pointer_interconvertible_with_class<DB1, int>(&B1::x) // true
std::is_pointer_interconvertible_with_class<DB1, int>(&DB1::x) // true

std::is_pointer_interconvertible_with_class<B1, int>(&DB1B2::x) // true
std::is_pointer_interconvertible_with_class<DB1B2, int>(&B1::x) // false
std::is_pointer_interconvertible_with_class<DB1B2, int>(&DB1B2::x) // false
```

C++20 标准在注释中解释了后两个语句为 false 的原因:

指针到成员表达式 &C::b 的类型，并不总是指向 C 成员的指针。当将这些函数与继承结合使用时，会导致令人惊讶的结果:

```
struct A { int a; }; // a standard-layout class
struct B { int b; }; // a standard-layout class
struct C: public A, public B { }; // not a standard-layout class

std::is_pointer_interconvertible_with_class(&C::b) // true
// true because, despite its appearance, &C::b has type
// “pointer to member of B of type int”

std::is_pointer_interconvertible_with_class<C>(&C::b) // false
// false because it forces the use of class C and fails
```

20.6. 附注

类型特性 is_bounded_array<> 和 is_unbounded_array<> 是 Walter E. Brown 和 Glen J. Fernandes 在http://wg21.link/p1357r1中提出。

类型特性 is_nothrow_convertible<> 是 Daniel Krügler 在http://wg21.link/p0758r1中提出。

Eric Niebler、Casey Carter 和 Christopher Di Bella 在http://wg21.link/p0896r4中描述了类型特征 common_eference<>，从而使其成为 ranges 库的一部分。

类型特征 unwrap_eference<> 和 unwrap_ref_decay<> 是由 Vicente J. Botet Escriba 在http://wg21.link/p0318r1中提出。

类型特征 remove_cvref<> 是 Walter E. Brown 在http://wg21.link/p0550r2中提出。

类型特征 type_identity<> 是 Timur Doumler 在http://wg21.link/p0887r1中提出。

迭代器类型特征接受为采用由 Eric Niebler、Casey Carter 和 Christopher Di Bella 在http://wg21.link/p0896r4中制定的提案，从而成为 ranges 库的一部分。

类型特征 is_layout_compatible<> 和 is_pointer_interconvertible_base_of<>，以及函数 is_corresponding_member() 和 is_pointer_interconvertible_with_class() 由 Lisa Lippincott 在http://wg21.link/p0466r5中提出。

第 21 章 核心语言的小改进

本章将介绍 C++20 为其核心语言引入的额外特性和扩展，这些特性和扩展在本书中尚未涉及。泛型编程的其他小特性将在下一章中进行描述。

21.1. 基于范围的 for 循环的初始化

C++17 引入了 if 和 switch 控制结构的可选初始化，C++20 现在为基于范围的 for 循环引入了这样一个可选的初始化。

例如，下面的代码在增加计数器的同时，遍历集合的元素:

```
for (int i = 1; const auto& elem : coll) {
  std::cout << std::format("{:3}: {}\n", i, elem);
  ++i;
}
```

考虑下面的代码，遍历目录 dirname 的目录名:

```
for (std::filesystem::path p{dirname};
const auto& e : std::filesystem::directory_iterator{p}) {
  std::cout << " " << e.path().lexically_normal().string() << '\n';
}
```

下面的代码在迭代一个集合时锁定一个互斥锁:

```
for (std::lock_guard lg{collMx}; const auto& elem : coll) {
  std::cout << elem: << elem << '\n';
}
```

控制结构中初始化器通常需要注意的是: 初始化器需要声明一个有名称的变量。否则，初始化本身就是一个表达式，创建并立即销毁临时对象。初始化未命名的锁保护是一个逻辑错误，因为当迭代发生时，保护将不再锁定:

```
for (std::lock_guard{collMx}; const auto& elem : coll) { // runtime ERROR
  std::cout << elem: << elem << '\n'; // - no longer locked
}
```

带有初始化的基于范围的 for 循环，也可以用作解决基于范围的 for 循环中的错误的方法。根据其规范，在迭代对临时对象的引用时，使用基于范围的 for 循环可能会导致 (致命的) 运行时错误。这个问题早在 2009 年就发现了 (见http://wg21.link/cwg900)。然而，C++ 标准委员会至今还不愿意像http://wg21.link/p2012中提议的那样修复这个 bug。

例如:

```cpp
std::optional<std::vector<int>> getValues(); // forward declaration

for (int i : getValues().value()) { // fatal runtime ERROR
  ...
}
```

使用基于范围的 for 循环和初始化可以避免这个问题:

```cpp
std::optional<std::vector<int>> getValues(); // forward declaration

for (auto&& optColl = getValues(); int i : optColl) { // OK
  ...
}
```

以同样的方式，可以使用 span 修复中断的迭代:

```cpp
for (auto elem : std::span{getCollOfConst()}) ... // fatal runtime error

for (auto&& coll = getCollOfConst(); auto elem : std::span{coll}) ... // OK
```

21.2. using 用于枚举值

假设有一个限定范围的枚举类型 (用枚举类声明):

```cpp
enum class Status{open, progress, done = 9};
```

与未限定作用域的枚举类型 (不带类的枚举) 不同，此类型的值需要带有类型名的限定符:

```cpp
auto x = Status::open; // OK
auto x = open; // ERROR
```

然而，在某些明显没有冲突的上下文中，一直限定每个值可能会变得有点乏味。为了更方便地使用作用域枚举类型，现在可以使用 using 枚举声明。

一个典型的例子是切换所有可能的枚举值，可以这样实现:

```cpp
void print(Status s)
{
  switch (s) {
  using enum Status; // make enum values available in current scope
  case open:
    std::cout << "open";
    break;
  case progress:
    std::cout << "in progress";
    break;
```

```
  case done:
    std::cout << "done";
    break;
  }
}
```

只要在 print() 的作用域中没有声明其他名为 open、progress 或 done 的符号，这段代码就可以正常工作。

也可以为特定的枚举值使用多个 using 声明:

```
void print(Status s)
{
  switch (s) {
  using Status::open, Status::progress, Status::done;
  case open:
    std::cout << "open";
    break;
  case progress:
    std::cout << "in progress";
    break;
  case done:
    std::cout << "done";
    break;
  }
}
```

这样，就确切地知道哪些名称在当前作用域中可用。

也可以对无作用域的枚举类型使用 using 声明。这不是必需的，但就不必知道枚举类型是如何定义的了:

```
enum Status{open, progress, done = 9}; // unscoped enum

auto s1 = open; // OK
auto s2 = Status::open; // OK

using enum Status; // OK, but no effect
auto s3 = open; // OK
auto s4 = Status::open; // OK
```

21.3. 将枚举类型委托给不同的作用域

使用枚举声明还可以将枚举值委托给不同的作用域。例如:

```
namespace MyProject {
  class Task {
    public:
```

```
    enum class Status{open, progress, done = 9};
    Task();
    ...
  };
  using enum Task::Status; // expose the values of Status to MyProject
}

auto x = MyProject::open; // OK: x has value MyProject::Task::open
auto y = MyProject::done; // OK: y has value MyProject::Task::done
```

请注意 using enum 声明只公开值，而不公开类型:

```
MyProject::Status s; // ERROR
```

要公开类型和它的值，还需要一个普通的 using 声明 (类型别名):

```
namespace MyProject {
  using Status = Task::Status; // expose the type Task::Status
  using enum Task::Status; // expose the values of Task::Status
}

MyProject::Status s = MyProject::done; // OK
```

对于公开的枚举值，甚至依赖于参数的查找 (ADL) 也可以正常工作。可以将上面的例子扩展为如下例子:

```
namespace MyProject {
  void foo(MyProject::Task::Status) {
  }
}

namespace MyScope {
  using enum MyProject::Task::Status; // OK
}

foo(MyProject::done); // OK: calls MyProject::foo() with
↪  MyProject::Task::Status::done

foo(MyScope::done); // OK: calls MyProject::foo() with MyProject::Task::Status::done
```

ADL 通常不使用类型别名:

```
namespace MyScope {
  void bar(MyProject::Task::Status) {
  }
  using MyProject::Task::Status; // expose enum type to MyScope
  using enum MyProject::Task::Status; // expose the enum values to MyScope
```

```
}

MyScope::Status s = MyScope::open; // OK
bar(MyScope::done); // ERROR
MyScope::bar(MyScope::done); // OK
```

21.4. 新字符类型 char8_t

为了更好地支持 UTF-8，C++20 引入了新的字符类型 char8_t 和新的对应字符串类型 std::u8string。

类型 char8_t 是一个新关键字，作为保存 UTF-8 字符和字符序列的类型。例如:

```
char8_t c = u8'@'; // character with UTF-8 encoding for character @
const char8_t* s = u8"K\u00F6ln"; // character sequence with UTF-8 encoding for Köln
```

这种类型的引入是一个突破性的变化:

- u8 字符字面量现在使用 char8_t，而非 char
- 对于 UTF-8 字符串，现在使用新的类型 std::u8string 和 std::u8string_view

考虑下面的例子:

```
auto c = u8'@'; // character with UTF-8 encoding for @
auto s1 = u8"K\u00F6ln"; // character sequence with UTF-8 encoding for K¨oln
using namespace std::literals;
auto s2 = u8"K\u00F6ln"s; // string with UTF-8 encoding for Köln
auto sv = u8"K\u00F6ln"sv; // string view with UTF-8 encoding for Köln
```

c 和 s 的类型在这里发生了变化:

- C++20 之前，这相当于:

```
char c = u8'@'; // UTF-8 character type before C++20
const char* s1 = u8"K\u00F6ln"; // UTF-8 character sequence type before C++20
using namespace std::literals;
std::string s2 = u8"K\u00F6ln"s; // UTF-8 string type before C++20
std::string_view sv = u8"K\u00F6ln"s; // UTF-8 string view type before C++20
```

- 自 C++20 起，这相当于:

```
char8_t c = u8'@'; // UTF-8 character type since C++20
const char8_t* s1 = u8"K\u00F6ln"; // UTF-8 character sequence type since C++20
using namespace std::literals;
std::u8string s2 = u8"K\u00F6ln"s; // UTF-8 string type since C++20
std::u8string_view sv = u8"K\u00F6ln"sv; // UTF-8 string view type since C++20
```

这个改变的原因很简单: 现在可以为 UTF-8 字符和字符串实现特殊的行为:

- 可以重载 UTF-8 字符序列:

```
void store(const char* s)
{
  storeInFile(convertToUTF8(s)); // store after converting to UTF-8 encoding
}

void store(const char8_t* s)
{
  storeInFile(s); // store as is because it already has UTF-8 encoding
}
```

- 可以在泛型代码中实现特殊行为:

```
void store(const CharT* s)
{
  if constexpr(std::same_as<CharT, char8_t>) {
    storeInFile(s); // store as is because it already has UTF-8 encoding
  }
  else {
    storeInFile(convertToUTF8(s)); // store after converting to UTF-8 encoding
  }
}
```

只能在 char8_t 类型的对象中存储一个字节的 UTF-8 字符 (记住，UTF-8 字符具有可变宽度):

- 符号 @ 的十进制值为 64(十六进制 0x40)，可以将其值存储在 char8_t 中。因此，u8 字符字面量定义良好:

```
char8_t c = u8'@'; // OK (c has value 64)
```

- 欧洲货币 Euro(€) 由三个编码单位组成:226 130 172(十六进制:0xE2 0x82 0xAC)，所以不能将其值存储在 char8_t 中:

```
char8_t cEuro = u8'€'; // ERROR: invalid character literal
```

必须初始化一个字符序列或 UTF-8 字符串:

```
const char8_t* cEuro = u8"\u20AC"; // OK
std::u8string sEuro = u8"\u20AC"; // OK
```

使用 Unicode 符号来指定 UTF-8 字符的值，这会创建一个包含四个 const char8_t(包括末尾的 null 字符) 的数组，然后来初始化 euro 和 sEuro。
若编译器在源代码中接受字符 e(必须支持带有字符集的源文件编码，如 UTF-8 或 ISO-8859-15)，甚至可以直接在字面量中使用这个符号:

```
const char8_t* cEuro = u8"€"; // OK if valid character for the compiler
std::u8string sEuro = u8"€"; // OK if valid character for the compiler
```

由于该源代码不可移植，应该使用 Unicode 字符 (例如使用\u20AC 表示 € 符号)。

C++ 标准库中的一些内容已相应更改:

- 添加了新字符类型 char8_t 的重载
- 使用或返回 UTF-8 字符串的函数使用新的 UTF-8 字符串类型

另外，注意 char8_t 不能保证有 8 位。定义为在内部使用 unsigned char，通常有 8 位，但可能更多。像往常一样，可以使用 std::numeric_limits<> 来检查位数:

```
std::cout << "char8_t has "
      << std::numeric_limits<char8_t>::digits << " bits\n";
```

21.4.1 C++ 标准库中对 char8_t 的更改

C++ 标准库更改了以下内容以支持 char8_t:

- 现在提供了 u8string(定义为 std::basic_string<char8_t>)
- 现在提供了 u8string_view(定义为 std::basic_string_view<char8_t>)
- 现在定义了 Std::numeric_limits<char8_t>
- 现在定义了 std::char_traits<char8_t>
- 现在提供了 char8_t 字符串和字符串视图的哈希函数
- 现在提供了 std::mbrtoc8() 和 c8rtomb()
- 现在提供了用于 char8_t 和 char16_t 或 char32_t 之间转换的编解码 facet
- 对于文件系统路径，u8string() 现在返回 std::u8string，而非 std::string
- 现在提供了 std::atomic<char8_t>

当使用 C++20 编译现有代码时，这些更改可能会破坏现有代码，这将在后面讨论。

21.4.2 向后兼容性的破坏

由于 C++20 更改了返回 UTF-8 字符串的 UTF-8 字面值和签名的类型，使用 UTF-8 字符的代码可能不再编译。

使用字符类型的错误代码

例如:

```
std::string s0 = u8"text"; // OK in C++17, ERROR since C++20
auto s = u8"K\u00F6ln"; // s is const char* until C++17, but const char8_t* since
↪  C++20
const char* s2 = s; // OK in C++17, ERROR since C++20
std::cout << s << '\n'; // OK in C++17, ERROR since C++20
auto c = u8'c'; // c1 is char in C++17, but char8_t since C++20
char c2 = c; // OK (even if char8_t)
char* cp = &c; // OK in C++17, ERROR since C++20
std::cout << c; // OK in C++17, ERROR since C++20
```

使用新字符串类型的错误代码

特别是，返回 UTF-8 字符串的代码现在可能会导致问题。

下面的代码不再编译，因为 u8string() 现在返回 std::u8string，而非 std::string:

```
// iterate over directory entries:
for (const auto& entry : fs::directory_iterator(path)) {
  std::string name = entry.path().u8string(); // OK in C++17, ERROR since C++20
  ...
}
```

必须使用不同的类型 auto 来调整代码，或者同时支持这两种类型:

```
// iterate over directory entries (C++17 and C++20):
for (const auto& entry : fs::directory_iterator(path)) {
#ifdef __cpp_char8_t
  std::u8string name = entry.path().u8string(); // OK since C++20
#else
  std::string name = entry.path().u8string(); // OK in C++17
#endif
  ...
}
```

对 UTF-8 字符串的错误代码 I/O

也可以不再输出 UTF-8 字符或字符串到 std::cout(或其他标准输出流):

```
std::cout << u8"text"; // OK in C++17, ERROR since C++20
std::cout << u8'X'; // OK in C++17, ERROR since C++20
```

事实上，C++20 删除了所有扩展字符类型 (wchar_t, char8_t, char16_t, char32_t) 的输出操作符，除非输出流支持相同的字符类型:

```
std::cout << "text"; // OK
std::cout << L"text"; // wchar_t string: OOPS in C++17, ERROR since C++20
std::cout << u8"text"; // UTF-8 string: OK in C++17, ERROR since C++20
std::cout << u"text"; // UTF-16 string: OOPS in C++17, ERROR since C++20
std::cout << U"text"; // UTF-32 string: OOPS in C++17, ERROR since C++20

std::wcout << "text"; // OK
std::wcout << L"text"; // OK
std::wcout << u8"text"; // UTF-8 string: OK in C++17, ERROR since C++20
std::wcout << u"text"; // UTF-16 string: OOPS in C++17, ERROR since C++20
std::wcout << U"text"; // UTF-32 string: OOPS in C++17, ERROR since C++20
```

注意带有 OOPS 标记的语句: 都可 C++17 中编译，但其输出的是字符串的地址，而非值。因此，C++20 不仅禁用了 UTF-8 字符的输出，还禁用了根本不起作用的输出。

处理错误代码

现在想知道如何处理以前使用 UTF-8 字符的代码，最简单的处理方法是使用 reinterpret_cast<>:

```
auto s = u8"text"; // s is const char8_t* since C++20
std::cout << s; // ERROR since C++20
std::cout << reinterpret_cast<const char*>(s); // OK
```

对于单个字符，使用 static_cast 就足够了:

```
auto c = u8'x'; // c is char8_t since C++20
std::cout << c; // ERROR since C++20
std::cout << static_cast<char>(c); // OK
```

可以在 char8_t 字符特性的特性测试宏上绑定其使用或使用其他解决方案:[对于 C++20 指定的最终行为，__cpp_char8_t 应该 (至少) 具有值 201907。]

```
auto s = u8"text"; // s is const char8_t* since C++20
#ifdef __cpp_char8_t
std::cout << reinterpret_cast<const char *>(s); // OK
#else
std::cout << s; // OK in C++17, ERROR since C++20
#endif
```

若想知道为什么 C++20 没有为 UTF-8 字符提供工作输出操作符，这是一个相当复杂的问题，我们根本没有足够的时间来解决。可以在这里阅读更多内容:http://stackoverflow.com/a/58895428。

因为当大量使用 UTF-8 字符时，使用 reinterpret_cast<> 可能无法扩展，Tom Honermann 编写了一个指南，关于如何处理 C++20 之前和之后的 UTF-8 字符代码: “P1423R3 char8_t 向后兼容性修复”。若要处理 UTF-8 字符和字符串，一定要阅读，可以从http://wg21.link/p1423下载到。

21.5. 聚合的改进

C++20 为聚合提供了一些改进，这些改进将在本节中介绍:

- (部分) 支持指定初始化式 (特定成员的初始值)
- 可以用圆括号初始化聚合
- 固定聚合的定义和 std::is_default_constructible<>

本书的其他部分描述了聚合在泛型代码中的使用:

- 对聚合使用类模板参数推导 (CTAD)
- 聚合可以用作非类型模板参数 (NTTP)。

21.5.1 指定初始化器

对于聚合，C++20 提供了一种方法来指定应该用传递的初始值初始化哪个成员，但只能使用它来跳过参数。

假设有以下聚合类型:

```
struct Value {
  double amount = 0;
  int precision = 2;
  std::string unit = "Dollar";
};
```

那么，现在支持以下方式来初始化该类型的值:

```
Value v1{100}; // OK (not designated initializers)
Value v2{.amount = 100, .unit = "Euro"}; // OK (second member has default value)
Value v3{.precision = 8, .unit = "$"}; // OK (first member has default value)
```

完整的示例请参见 lang/designated.cpp。

注意以下限制:

- 必须使用 = 或 {} 传递初始值。
- 可以跳过会员，但必须遵循其顺序。按名称初始化的成员的顺序必须与声明中的顺序匹配。
- 必须对所有参数使用指定初始化式，或者不使用指定初始化式。不允许混合初始化。
- 不支持对数组使用指定初始化。
- 可以使用指定初始化式嵌套初始化，但不能直接使用.mem.mem。
- 初始化带有圆括号的聚合时，不能使用指定初始化器。
- 指定初始化式也可以用在联合体中。

例如:

```
Value v4{100, .unit = "Euro"}; // ERROR: all or none designated
Value v5{.unit = "$", .amount = 20}; // ERROR: invalid order
Value v6(.amount = 29.9, .unit = "Euro"); // ERROR: only supported for curly braces
```

与编程语言 C 相比，指定初始化器遵循成员顺序的限制，可以用于所有参数，也可以不用于参数，不支持直接嵌套，不支持数组。尊重成员顺序的原因是确保初始化反映了调用构造函数的顺序(与调用析构函数的顺序相反)。

作为使用 =、{} 和联合体嵌套初始化的示例:

```
union Sub {
  double x = 0;
  int y = 0;
};

struct Data {
```

```
  std::string name;
  Sub val;
};

Data d1{.val{.y=42}}; // OK
Data d2{.val = {.y{42}}}; // OK
```

不能直接嵌套指定初始化式:

```
Data d2{.val.y = 42}; // ERROR
```

21.5.2 用圆括号聚合初始化

假设已经声明了以下集合:

```
struct Aggr {
  std::string msg;
  int val;
};
```

C++20 之前，只能使用花括号来初始化带有值的聚合:

```
Aggr a0; // OK, but no initialization
Aggr a1{}; // OK, value initialized with "" and 0
Aggr a2{"hi"}; // OK, initialized with "hi" and 0
Aggr a3{"hi", 42}; // OK, initialized with "hi" and 42
Aggr a4 = {}; // OK, initialized with "" and 0
Aggr a5 = {"hi"}; // OK, initialized with "hi" and 0
Aggr a6 = {"hi", 42}; // OK, initialized with "hi" and 42
```

自 C++20 起，还可以使用圆括号作为外部字符来直接初始化，不带 =:

```
Aggr a7("hi"); // OK since C++20: initialized with "hi" and 0
Aggr a8("hi", 42); // OK since C++20: initialized with "hi" and 42
Aggr a9({"hi", 42}); // OK since C++20: initialized with "hi" and 42
```

使用 = 或内括号仍然不起作用:

```
Aggr a10 = "hi"; // ERROR
Aggr a11 = ("hi", 42); // ERROR
Aggr a12(("hi", 42)); // ERROR
```

使用内括号甚至可以编译，其用作使用逗号操作符的表达式周围的圆括号。

现在可以使用圆括号来初始化边界未知的数组:

```cpp
int a1[]{1, 2, 3}; // OK since C++11
int a2[](1, 2, 3); // OK since C++20
int a3[] = {1, 2, 3}; // OK
int a4[] = (1, 2, 3); // still ERROR
```

然而，不支持” 省略大括号”(没有嵌套的大括号可以省略):

```cpp
struct Arr {
  int elem[10];
};

Arr arr1{1, 2, 3}; // OK
Arr arr2(1, 2, 3); // ERROR
Arr arr3{{1, 2, 3}}; // OK
Arr arr4({1, 2, 3}); // OK (even before C++20)
```

要初始化 std::arrays，仍须使用大括号:

```cpp
std::array<int,3> a1{1, 2, 3}; // OK: shortcut for std::array{{1, 2, 3}}
std::array<int,3> a2(1, 2, 3); // still ERROR
```

用圆括号初始化聚合的原因

支持带圆括号的聚合初始化的原因是，可以用圆括号调用 new 操作符:

```cpp
struct Aggr {
  std::string msg;
  int val;
};

auto p1 = new Aggr{"Rome", 200}; // OK since C++11
auto p2 = new Aggr("Rome", 200); // OK since C++20 (ERROR before C++20)
```

这有助于支持在类型中使用聚合，这些类型在内部调用 new 时使用括号将值存储在现有内存中，就像容器和智能指针的情况一样。自 C++20 起，以下是可能的情况:

- 现在可以对聚合使用 std::make_unique<>() 和 std::make_shared<>():

```cpp
auto up = std::make_unique<Aggr>("Rome", 200); // OK since C++20
auto sp = std::make_shared<Aggr>("Rome", 200); // OK since C++20
```

C++20 之前，无法对聚合使用这些辅助函数。

- 现在可以将新值放置到聚合容器中:

```cpp
std::vector<Aggr> cont;
cont.emplace_back("Rome", 200); // OK since C++20
```

仍然有一些类型不能用括号初始化，但可以用花括号初始化: 作用域枚举 (枚举类类型)。类型 std::byte (C++17 引入) 就是一个例子:

```
std::byte b1{0}; // OK
std::byte b2(0); // still ERROR
auto upb2 = std::make_unique<std::byte>(0); // still ERROR
auto upb3 = std::make_unique<std::byte>(std::byte{0}); // OK
```

对于 std::array，仍然需要大括号 (如上所述):

```
std::vector<std::array<int, 3>> ca;
ca.emplace_back(1, 2, 3); // ERROR
ca.emplace_back({1, 2, 3}); // ERROR
ca.push_back({1, 2, 3}); // still OK
```

详细地用圆括号聚合初始化

引入圆括号初始化的建议列出了以下设计准则:

- Type(val) 的现有含义都不应该改变。
- 括号初始化和括号初始化应尽可能相似，但也应尽可能不同，以符合现有的括号列表和括号列表的模型。

实际上，带花括号的聚合初始化和带圆括号的聚合初始化有以下区别:

- 带圆括号的初始化不会检测窄化转换。
- 用圆括号初始化允许所有隐式转换 (不仅是从派生类到基类)。
- 当使用括号时，引用成员不会延长传递的临时对象的生命周期.
- 使用圆括号不支持大括号省略 (使用它们就像向形参传递实参一样)。
- 带空括号的初始化甚至适用于显式成员。
- 使用圆括号不支持指定初始化式。

缺少检测窄化的例子如下:

```
struct Aggr {
  std::string msg;
  int val;
};

Aggr a1{"hi", 1.9}; // ERROR: narrowing
Aggr a2("hi", 1.9); // OK, but initializes with 1

std::vector<Aggr> cont;
cont.emplace_back("Rome", 1.9); // initializes with 1
```

emplace 函数永远不会检测窄化。

处理隐式转换时的区别示例如下:

```
example of the difference when dealing with implicit conversions is this:
struct Other {
  ...
  operator Aggr(); // defines implicit conversion to Aggr
};

Other o;
Aggr a7{o}; // ERROR: no implicit conversion supported
Aggr a8(o); // OK, implicit conversion possible
```

聚合本身不能定义转换，因为聚合不能有用户定义的构造函数。

缺少大括号省略和大括号初始化的复杂规则，会导致以下行为:

```
Aggr a01{"x", 65}; // init string with "x" and int with 65
Aggr a02("x", 65); // OK since C++20 (same effect)

Aggr a11{{"x", 65}}; // runtime ERROR: "x" doesn't have 65 characters
Aggr a12({"x", 65}); // OK even before C++20: init string with "x" and int with 65

Aggr a21{{{"x", 65}}}; // ERROR: cannot initialize string with initializer list of
↪  "x" and 65
Aggr a22({{"x", 65}}); // runtime ERROR: "x" doesn't have 65 characters

Aggr a31{'x', 65}; // ERROR: cannot initialize string with 'x'
Aggr a32('x', 65); // ERROR: cannot initialize string with 'x'

Aggr a41{{'x', 65}}; // init string with 'x' and char(65)
Aggr a42({'x', 65}); // OK since C++20 (same effect)

Aggr a51{{{'x', 65}}}; // init string with 'x' and char(65)
Aggr a52({{'x', 65}}); // OK even before C++20: init string with 'x' and 65

Aggr a61{{{{'x', 65}}}}; // ERROR
Aggr a62({{{{'x', 65}}}}); // OK even before C++20: init string with 'x' and 65
```

当执行复制初始化 (用 = 初始化) 时，显式很重要，使用空括号可能会产生不同的效果:

```
struct C {
  explicit C() = default;
};

struct A { // aggregate
  int i;
  C c;
};

auto a1 = A{42, C{}}; // OK: explicit initialization
```

```
auto a2 = A(42, C()); // OK since C++20: explicit initialization

auto a3 = A{42}; // ERROR: cannot call explicit constructor
auto a4 = A(42); // ERROR: cannot call explicit constructor

auto a5 = A{}; // ERROR: cannot call explicit constructor
auto a6 = A(); // OK: can call explicit constructor
```

这在 C++20 中并不新鲜，在 C++20 之前就已经支持 a6 的初始化了。

最后，若对具有右值引用成员的聚合使用圆括号进行初始化，则初始值的生存期不会延长，所以不应该传递临时对象 (右值):

```
struct A {
  int a;
  int&& r;
};

int f();
int n = 10;

A a1{1, f()}; // OK, lifetime is extended
std::cout << a1.r << '\n'; // OK
A a2(1, f()); // OOPS: dangling reference
std::cout << a2.r << '\n'; // runtime ERROR
A a3(1, std::move(n)); // OK as long as a3 is used only while n exists
std::cout << a3.r << '\n'; // OK
```

由于这些复杂的规则和陷阱，只有在必要时才应该使用带圆括号的聚合初始化，比如在使用 std::make_unique<>()、std::make_shared<>() 或 emplace 函数时。

21.5.3 聚合体的定义

聚合的定义在 C++20 中发生了变化，修改逆转了 C++11 中出现的扩展，结果证明这是一个错误。从 C++11 到 C++17，不允许使用用户提供的构造函数。基于这个原因，下面是聚合的合法定义:

```
struct A { // aggregate from C++11 until C++17
  ...
  A() = delete; // user-declared, but not user-provided constructor
};
```

有了这个定义，即使类型有一个删除的默认构造函数，聚合初始化也是有效的:

```
A a1; // ERROR
A a2{}; // OK from C++11 until C++17
```

这不仅适用于默认构造函数。例如:

```cpp
struct D { // aggregate from C++11 until C++17
  int i = 0;
  D(int) = delete; // user-declared, but not user-provided constructor
};

D d1(3); // ERROR
D d2{3}; // OK from C++11 until C++17
```

这种特殊的行为是在一个非常特殊的情况下引入的，但完全违反直觉。[该特性是作为“hack”引入的，以在初始化原子类型时支持 C 兼容性。] 然而，就连几名委员会成员也不知道这种行为，他们感到非常惊讶，甚至发现这种支持根本不需要。C++20 修正了这个问题，重新要求聚合没有用户声明的构造函数 (就像 C++11 之前的情况一样):

```cpp
struct A { // NO aggregate since C++20
  ...
  A() = delete; // user-declared, but not user-provided constructor
};

A a1; // ERROR
A a2{}; // ERROR since C++20
```

如上所述，对于类型 A 和 D，类型特性 std::is_default_constructible_v<> 将不再为 true。

一些开发者在创建聚合类型的对象时确实使用此特性来强制 (可能为空) 花括号 (这确实可确保始终对成员进行初始化)。对于它们来说，获得相同行为的解决方法是从基类型派生:

```cpp
struct MustInit {
  MustInit(MustInit&&) = default;
};

struct A : MustInit {
  ...
};

A a1; // ERROR
A a2{}; // OK
```

总而言之，自 C++20 起，聚合的定义如下:

- 一个数组
- 或者一个类型 (类、结构或联合):
  - 没有用户声明的构造函数
  - using 声明没有继承构造函数
  - 没有 private 或 proctected 的非静态数据成员
  - 没有虚函数
  - 没有 virtual、private 或 proctected 的基类

为了初始化聚合，需要应用以下附加限制:

- 没有 private 或 proctected 的基类成员
- 没有 private 或 proctected 的构造函数

21.6. 新增属性和属性特性

自 C++11 起，可以指定属性 (启用或禁用警告的正式注释)。在 C++20 中，又引入了新的属性，并扩展了现有的属性。

21.6.1 属性 [[likely]] 和 [unlikely]]

C++20 引入了新属性 [[likely]] 和 [[unlikely]]，以帮助编译器执行分支优化。

当代码中有多条路径时，可以使用这些属性向编译器提示哪条路径最有可能出现，哪条路径最不可能出现。

例如:

```
int f(int n)
{
  if (n <= 0) [[unlikely]] { // n <= 0 is considered to be arbitrarily unlikely
    return n;
  }
  else {
    return n * n;
  }
}
```

这可能会迫使编译器生成直接处理 else 分支的汇编代码，同时跳转到后面的 then 情况的汇编命令。

但不保证有相同的效果:

```
int f(int n)
{
  if (n <= 0) {
    return n;
  }
  else [[likely]] { // n > 0 is considered to be arbitrarily likely
    return n * n;
  }
}
```

下面是另一个例子:

```
int g(int n)
{
  switch (n) {
```

```
    case 1:
    ...
    break;
    [[likely]] case 2: // n == 2 is considered to be arbitrarily most likely
    ...
    break;
  }
  ...
}
```

这些属性的效果特定于编译器，不能保证这些属性的影响。

在使用这些属性时应该谨慎，并仔细检查它们的效果。通常，编译器更了解如何优化代码，若过度使用这些属性可能会适得其反。

21.6.2 属性 [[no_unique_address]]

类通常具有影响行为但不提供状态的成员，例如：无序容器的哈希函数、std::unique_ptr 的 delete 函数，或者容器或字符串的标准分配器。它们提供的只是成员函数 (和静态成员)，而不是非静态数据成员。

然而，成员通常需要内存，即使他们不存储。例如，下面的代码:

```
struct Empty {}; // empty class: size is usually 1

struct I { // size is same as sizeof(int)
  int i;
};

struct EandI { // size is sum of size of members with alignment
  Empty e;
  int i;
};

std::cout << "sizeof(Empty): " << sizeof(Empty) << '\n';
std::cout << "sizeof(I): " << sizeof(I) << '\n'
std::cout << "sizeof(EandI): " << sizeof(EandI) << '\n';
```

根据 int 类型的大小，输出可能如下所示:

```
sizeof(E):     1
sizeof(I):     4
sizeof(EandI): 8
```

这是不必要的空间浪费。在 C++20 之前，可以使用空基类优化 (EBCO) 来避免不必要的开销。通过从没有数据成员的类派生，编译器可以节省相应的空间:

```
struct EbasedI : Empty { // using EBCO
  int i;
};

std::cout << "sizeof(EbasedI): " << sizeof(EbasedI) << '\n';
```

在具有上述情况的平台上，输出如下:

```
sizeof(EbasedI): 4
```

这种解决方法有点笨拙，可能并不总是有效。例如，若空基类是 final，则不能使用 EBCO。

自 C++20 起，有一种不同的方法可以获得相同的效果。只需要用属性 [[no_unique_address]] 声明不提供状态的成员:

```
struct EattrI { // same effect as EBCO
  [[no_unique_address]] Empty e;
  int i;
};

struct IattrE { // same effect as EBCO
  int i;
  [[no_unique_address]] Empty e;
};

std::cout << "sizeof(EattrI): " << sizeof(EattrI) << '\n';
std::cout << "sizeof(IattrE): " << sizeof(IattrE) << '\n';
```

在上面支持此属性的平台上，输出会变为:

```
sizeof(EattrI): 4
sizeof(IattrE): 4
```

编译器不需要遵循此属性。

标记为 [[no_unique_address]] 的成员在初始化时仍会视为成员:

```
EattrI ei = {42}; // ERROR: can't initialize member e with 42
EattrI ei = {{},42}; // OK
```

这种优化还意味着成员 e 的地址与同一对象的成员 i 的地址相同，则两个不同对象的成员 e 有不同的地址。

若数据类型只有具有此属性的数据成员，则类型 trait std::is_empty_v<> 是否产生 true 是由实现定义的:

```cpp
struct OnlyEmpty {
  [[no_unique_address]] Empty e;
};

std::is_empty_v<OnlyEmpty> // might yield true or false
```

最后，请注意 Visual C++ 目前忽略了这个属性。原因是 Visual C++ 最初允许不遵守这个属性，现在遵守它变成了一个 ABI 中断。不过，Visual C++ 可能会在未来的 ABI 版本中支持。在此之前，可以使用 [[msvc::no_unique_address]] 代替:

```cpp
struct EattrI { // also works for Visual C++
  [[no_unique_address]] [[msvc::no_unique_address]] Empty e;
  Type i;
};
```

21.6.3 带参数的属性 [[nodiscard]]

C++17 引入了属性 [[nodiscard]]，若没有使用函数的返回值，可以使用该属性鼓励编译器发出警告。

当返回值未被使用时，[[nodiscard]] 通常用来表示错误行为。不当行为可能是:

- 内存泄漏，例如：不使用返回的已分配内存。
- 意外或非直观的行为，例如：在不使用返回值时获得不同/意外的行为
- 不必要的开销，例如：若返回值未被使用，则使用 no-op 的行为

但是，没有办法指定一条消息来解释为什么开发者应该使用返回值。C++20 为此引入了一个可选参数。

例如:

```cpp
class MyType {
  public:
  ...
  [[nodiscard("Possible memory leak")]] // OK, since C++20
  char* release();

  void clear();

  [[nodiscard("Did you mean clear()?")]] // OK, since C++20
  bool empty() const;
};
```

当 empty() 的返回值未使用时，第二条声明要求编译器打印警告消息“Did you mean clear()?”。事实上，C++20 为所有标准容器的成员函数 empty() 引入了这个特性。根据反馈，我们知道编译器确实发现了一些错误，开发者认为他们要求将集合变为空。

21.7. 特性测试宏

每个 C++ 版本都会逐步引入编译器支持的各种语言和库特性。出于这个原因，知道编译器通常支持哪个 C++ 版本通常是不够的; 对于可移植代码，了解某个特定特性是否可用很重要。

为此，C++20 正式引入了特性测试宏。对于每个新语言和库特性，都可以使用一个宏来表示该特性是否可用。宏甚至可以提供关于支持哪个版本的特性的信息。

例如，以下源代码将根据是否可用 (以及以何种形式) 使用不同的代码:

```
#ifdef __cpp_generic_lambdas
#if __cpp_generic_lambdas >= 201707
... // generic lambdas with template parameters can be used
#else
... // generic lambdas can be used
#else
... // no generic lambdas can be used
#endif
```

所有用于语言特性的特性测试宏都以__cpp 开头。

作为另一个例子，若 std::as_const() 还不可用，下面的代码提供并使用了一个解决方案:

```
#ifndef __cpp_lib_as_const
template<typename T>
const T& asConst(T& t) {
  return t;
}
#endif

#ifdef __cpp_lib_as_const
  auto printColl = [&coll = std::as_const(coll)] {
#else
  auto printColl = [&coll = asConst(coll)] {
#endif
  ...
  };
```

库特性的所有特性测试宏都以__cpp_lib 开头。

语言特性的特性测试宏由编译器定义，库特性的特性测试宏由新头文件 <version> 提供。

参见__cpp_char8_t 的用法，这是另一个使代码在 C++20 之前和之后都可移植 UTF-8 字符的例子。

21.8. 附注

初始化的基于范围的 for 循环首先由 Thomas Köppe 在http://wg21.link/p0614r0中提出。最终接受的提案是由 Thomas Köppe 在http://wg21.link/p0614r1中制定。

用于枚举值的 using 最早是由 Gasper Azman 和 Jonathan Müller 在http://wg21.link/p1099r0中提出。最终接受的提案是由 Gasper Azman 和 Jonathan Müller 在http://wg21.link/p1099r5中制定。

为 UTF-8 字符提供不同类型的请求最早是由 Tom Honermann 在http://wg21.link/p0482r0中提出。最后接受的提案是由 Tom Honermann 在http://wg21.link/p0482r6中提出的。对输出运算符的相应修复，是 Tom Honermann 在http://wg21.link/p1423r3中提出。

支持指定初始化式的请求是由 Tim Shen、Richard Smith、Zhihao Yuan 和 Chandler Carruth 在http://wg21.link/p0329r0中首先提出。最终接受的提案是由 Tim Shen 和 Richard Smith 在http://wg21.link/p0329r4上制定。

带括号的聚合初始化最初是由 Ville Voutilainen 在http://wg21.link/p0960r0中提出。最终接受的提案是由 Ville Voutilainen 和 Thomas Köppe 在http://wg21.link/p0960r3中制定。

由 Timur Doumler、Arthur O’dwyer、Richard Smith、Howard E. Hinnant 和 Nicolai Josuttis 在http://wg21.link/p1008r1中提出了改变聚合的定义，并接受。

属性 [[likely]] 和 [[unlikely]] 是由 Clay Trychta 在http://wg21.link/p0479r5中提出。

属性 [[no_unique_address]] 由 Richard Smith 在http://wg21.link/p0840r2中提出。

JeanHeyd Meneide 和 Isabella Muerte 在http://wg21.link/p1301r4中提出允许属性 [[nodiscard]] 有一个参数。

特性测试宏是由 Ville Voutilainen 和 Jonathan Wakely 在http://wg21.link/p0941r2中提出。

第 22 章 泛型编程的小改进

本章介绍了 C++20 在泛型编程方面的额外特性和扩展，这些在本书中还没有涉及到。

22.1. 模板形参的类型成员的隐式类型名

当使用模板形参的类型成员时，通常必须使用关键字 typename 来限定这种用法:

```
template<typename T>
typename T::value_type getElem(const T& cont, typename T::iterator pos)
{
  using Itor = typename T::iterator;
  typename T::value_type elem;
  ...
  return elem;
}
```

C++20 之前，类型成员 value_type 和 T 的迭代器的所有限定条件都是必需的。

自 C++20 起，可以在明确传递类型的上下文中跳过 typename，这适用于返回类型和别名声明中使用的类型的规范 (其中 using 为类型引入了一个新名称):

```
template<typename T>
T::value_type getElem(const T& cont, typename T::iterator pos)
{
  using Itor = T::iterator;
  typename T::value_type elem;
  ...
  return elem;
}
```

参数 pos 和变量 elem 仍然需要 typename，可以跳过 typename 的地方是:

- 当声明返回类型时 (局部前向声明除外)
- 在类模板中声明成员时
- 在类模板中声明成员函数或友元函数的形参时
- 声明 require 表达式的参数时
- 对于别名声明中的类型

当声明一个类模板时，可以在大多数情况下跳过 typename:

```
template<typename T>
class MyClass {
  T::value_type val; // no need for typename since C++20
public:
  ...
  T::iterator begin() const; // no need for typename since C++20
```

```
  T::iterator end() const; // no need for typename since C++20
  void print(T::iterator) const; // no need for typename since C++20
};
```

由于隐式 typename 的规则在某种程度上非常微妙，因此在使用模板形参的类型成员时 (至少在类模板之外)，可能仍然使用 typename。

22.1.1 隐式类型名的详情

自 C++20 起，以下情况为模板形参使用类型成员时，可以跳过 typename:

- 在别名声明中 (即，使用 using 声明类型名称时); 注意，带 typedef 的类型声明仍然需要 typename
- 当定义或声明函数的返回类型时 (除非声明发生在函数或块范围内)
- 声明尾步返回类型时
- 当指定 static_cast、const_cast、reinterpret_cast 或 dynamic_cast 的目标类型时
- 指定类型时
- 在类中
  - 声明数据成员时
  - 声明成员函数的返回类型时
  - 声明成员函数或友元函数或 Lambda 的形参 (默认实参可能仍然需要) 时
- 在 require 表达式中声明参数类型时
- 为模板的类型参数声明默认值时
- 声明非类型模板形参的类型时

注意，C++20 之前，typename 在一些其他情况下是不必要的:

- 指定继承类的基类型时
- 在构造函数中将初始值传递给基类时
- 在类声明中使用类型成员时

下面的例子演示了上面的大多数情况 (TYPENAME 表示自 C++20 以来可选填的地方):

```
template<typename T,
auto ValT = typename T::value_type{}> // typename required
class MyClass {
  TYPENAME T::value_type val; // typename optional
  public:
  using iterator = TYPENAME T::iterator; // typename optional
  TYPENAME T::iterator begin() const; // typename optional
  TYPENAME T::iterator end() const; // typename optional
  void print(TYPENAME T::iterator) const; // typename optional
  template<typename T2 = TYPENAME T::value_type> // second typename optional
  void assign(T2);
};

template<typename T>
```

```
TYPENAME T::value_type // typename optional
foo(const T& cont, typename T::value_type arg) // typename required
{
  typedef typename T::value_type ValT2; // typename required
  using ValT1 = TYPENAME T::value_type; // typename optional
  typename T::value_type val; // typename required
  typename T::value_type other1(void); // typename required
  auto other2(void) -> TYPENAME T::value_type; // typename optional
  auto l1 = [] (TYPENAME T::value_type) { // typename optional
  };
  auto p = new TYPENAME T::value_type; // typename optional
  val = static_cast<TYPENAME T::value_type>(0); // typename optional
  ...
}
```

22.2. 泛型代码中聚合的改进

C++20 为聚合提供了一些改进。对于泛型代码，现在有:

- 对聚合使用类模板参数推导 (CTAD)
- 聚合可以用作非类型模板参数 (NTTP)。

本节将介绍前者。

聚合还有其他新特性:

- (部分) 支持指定初始化式 (特定成员的初始值)
- 可以用圆括号初始化聚合
- 聚合的固定定义和 std::is_default_constructible<>

22.2.1 聚合的类模板参数推导 (CTAD)

从 C++17 开始，构造函数可用于推断类模板的模板形参。例如:

```
template<typename T>
class Type {
  T value;
public:
  Type(T val)
  : value{val} {
  }
  ...
};

Type<int> t1{42};
Type t2{42}; // deduces Type<int> since C++17
```

即使对于简单的聚合，也不支持根据对象初始化方式进行类似的推导:

```cpp
template<typename T>
struct Aggr {
  T value;
};

Aggr<int> a1{42}; // OK
Aggr a2{42}; // ERROR before C++20
```

必须提供一个推导指南:

```cpp
template<typename T>
struct Aggr {
  T value;
};

template<typename T>
Aggr(T) -> Aggr<T>;

Aggr<int> a1{42}; // OK
Aggr a2{42}; // OK since C++17
```

自 C++20 起，不再需要推导指南，则以下内容就足够了:

```cpp
template<typename T>
struct Aggr {
  T value;
};

Aggr<int> a1{42}; // OK
Aggr a2{42}; // OK since C++20 even without deduction guide
```

当使用括号初始化聚合时，此功能也有效:

```cpp
Aggr a3(42); // OK since C++20 even without deduction guide
```

推导规则可以很微妙:

```cpp
template<typename T>
struct S {
  T x;
  T y;
};

template<typename T>
struct C {
  S<T> s;
  T x;
```

```
  T y;
};

C c1 = {{1, 2}, 3, 4}; // OK, C<int> deduced
C c2 = {{1, 2}, 3}; // OK, C<int> deduced (y is 0)
C c3 = {{1, 2}, 3.3, 4.4}; // OK, C<double> deduced

C c4 = {{1, 2}, 3, 4.4}; // ERROR: int and double deduced for T
C c5 = {{1, 2}}; // ERROR: T only indirectly deduced
C c6 = {1, 2, 3}; // ERROR: don't know how many values S<T> needs
C<int> c7 = {1, 2, 3}; // OK
```

类模板参数推导甚至适用于具有可变数量元素的聚合:

```
// aggregate with variadic number of base types:
template<typename... T>
struct C : T... {
};

struct Base1 {
};
struct Base2 {
};

// aggregate initialized with two elements of types Base1 and Base2:
C c1{Base1{}, Base2{}};

// aggregate initialized with three lambda elements:
C c2{[] { f1(); },
  [] { f2(); },
  [] { f3(); }
};
```

22.3. 显式条件

要禁用隐式类型转换，可以将构造函数声明为显式。但对于泛型代码，当且仅当类型参数具有显式构造函数时，可能希望显式地使用构造函数。

这样，就可以完美地将类型转换支持委托给包装器类型，指定显式条件的方法类似于指定条件noexcept。直接在 explicit 之后，就可以在括号中指定布尔编译时表达式。下面是一个例子:

lang/wrapper.hpp

```
#include <type_traits> // for std::is_convertible_v<>

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

```
  template<typename U>
  explicit(!std::is_convertible_v<U, T>)
  Wrapper(const U& val)
  :value{val} {
  }
  ...
};
```

保存 T 类型值的类模板 Wrapper<> 有一个泛型构造函数，则可以用可以隐式转换为 T 的类型初始化该值。若没有从 U 到 T 的隐式转换，则构造函数是显式的。

只有在启用了隐式类型转换的情况下，才能初始化类型的 Wrapper。例如:

lang/explicitwrapper.cpp

```
#include "wrapper.hpp"
#include <string>
#include <vector>

void printStringWrapper(Wrapper<std::string>) {
}
void printVectorWrapper(Wrapper<std::vector<std::string>>) {
}

int main()
{
  // implicit conversion from string literal to string:
  std::string s1{"hello"};
  std::string s2 = "hello"; // OK
  Wrapper<std::string> ws1{"hello"};
  Wrapper<std::string> ws2 = "hello"; // OK
  printStringWrapper("hello"); // OK

  // NO implicit conversion from size to vector<string>:
  std::vector<std::string> v1{42u};
  std::vector<std::string> v2 = 42u; // ERROR: explicit
  Wrapper<std::vector<std::string>> wv1{42u};
  Wrapper<std::vector<std::string>> wv2 = 4u2; // ERROR: explicit
  printVectorWrapper(42u); // ERROR: explicit
}
```

为了演示条件显式的效果，对字符串和字符串的 vector 使用 Wrapper<>:

- 对于 std::string 类型，构造函数支持从字符串字面值进行隐式转换，所以 Wrapper<> 类型的构造函数不是显式的，其作用是传递字符串字面值来复制初始化字符串包装器，或者将字符串字面值传递给接受字符串包装器的函数。[记住，复制初始化 (initialization with =) 和参数传递需要隐式转换。]
- 对于 std::vector<std::string> 类型，接受无符号大小的构造函数在 C++ 标准库中声明为显式，所以 std::is_convertible<> 从大小转换为向量为 false，Wrapper<> 构造函数变为显式。也不能传

递 size 来初始化 string 类型 vector 的包装器，也不能将 size 传递给接受该包装器类型的函数。

可以使用 explicit 的地方都可以使用条件显式，也可以使用它使转换操作符显式条件化:

```
template<typename T>
class MyType {
  public:
  ...
  explicit(!std::is_convertible_v<T, bool>) operator bool();
};
```

然而，到 bool 的转换是迄今为止最有用的转换操作符的应用，应该始终是显式的，这样就不会意外地将 MyType 对象传递给期望布尔值的函数。

22.3.1 标准库中的条件显式

C++ 标准库在几个地方使用了条件显式。例如，std::pair<> 和 std::tuple<> 用它来支持稍微不同类型的对和元组的赋值，前提是存在可用的隐式转换。

例如:

```
std::pair<int, int> p1{11, 11};

std::pair<long, long> p2{};
p2 = p1; // OK: implicit conversion from int to long

std::pair<std::chrono::day, std::chrono::month> p3{};
p3 = p1; // ERROR
p3 = std::pair<std::chrono::day, std::chrono::month>{p1}; // OK
```

因为接受整数值的 chrono 日期类型的构造函数是显式的，所以将一对 int 赋值给一对 day 和 month 会失败，所以必须使用显式转换。

这也适用于在映射中插入元素时 (因为元素是键/值对):

```
std::map<std::string, std::string> coll1;
coll1.insert({"hi", "ho"}); // OK: uses implicit conversions to strings

std::map<std::string, std::chrono::month> coll2;
coll2.insert({"XI", 11}); // ERROR: no implicit conversion that fits
coll2.insert({"XI", std::chrono::month{11}}); // OK (inserts elem with string and
↪  month)
```

std::pair<> 的这种行为并不新鲜。在 C++20 前，标准库的实现必须使用 SFINAE 来实现 explicit 的条件行为 (声明两个构造函数，若不满足条件则禁用其中一个)。

另一个例子，std::span<> 的构造函数是条件显式的:

```cpp
namespace std {
  template<typename ElementType, size_t Extent = dynamic_extent>
  class span {
  public:
    static constexpr size_type extent = Extent;
    ...
    constexpr span() noexcept;
    template<typename It>
      constexpr explicit(extent != dynamic_extent)
      span(It first, size_type count);
    template<typename It, typename End>
      constexpr explicit(extent != dynamic_extent)
      span(It first, End last);
    ...
  };
}
```

因此，只有当 span 具有动态范围时，才允许 span 的隐式类型转换。

22.4. 附注

某些情况下使 typename 成为可选的是由 Daveed Vandevoorde 在http://wg21.link/p0634r0中首先提出。最终接受的措辞是由 Nina Ranns 和 Daveed Vandevoorde 在http://wg21.link/p0634r3上制定。

聚合的类模板参数推导首先由 Mike Spertus 在http://wg21.link/p1021r0中提出。最终接受的提案是由 Timur Doumler 在http://wg21.link/p1816r0和http://wg21.link/p2082r1中提出。

条件显式特征最早是由 Barry Revzin 和 Stephan T. Lavavej 在http://wg21.link/p0892r0中提出。最终接受的提案是由 Barry Revzin 和 Stephan T. Lavavej 在http://wg21.link/p0892r2上制定。

第 23 章 C++ 标准库的小修改

本章介绍了本书中尚未涉及的 C++20 标准库的附加特性和扩展。

23.1. 字符串类型的更新

C++20 中，字符串类型的某些方面发生了变化。这些更改影响字符串 (类型 std::basic_string<> 及其实例化，如 std::string)，字符串视图 (std::basic_string_view<> 及其实例化，如 std::string_view)，或两者都有。

实际上，C++20 为字符串类型引入了以下改进:

- 所有字符串类型现在都支持太空船操作符 <=>，则现在只声明 == 和 <=> 操作符，不再声明!=、<、<=、> 和 >=。
- 所有字符串类型现在都提供了新的成员函数，ends_with()，ends_with()。
- 对于字符串，成员函数 reserve() 不能再用于请求缩小字符串的容量 (为值分配的内存)。由于这个原因，不能再向 reserve() 传递参数。
- 对于 UTF-8 字符，C++ 现在提供了字符串类型 std::u8string 和 std::u8string_view。它们定义为 std::basic_string<> 和 std::basic_string_view<>，用于新的 UTF-8 字符类型 char8_t，返回 UTF-8 字符串的库函数现在具有返回类型 std::u8string。当切换到 C++20 时，此更改可能会破坏现有的代码。
- 字符串 (std::string 和 std::basic_string<> 的其他实例) 现在是 constexpr，则可以在编译时使用字符串。

  不能在编译时和运行时同时使用 std::string，但有几种方法可以将编译时字符串导出到运行时。
- 字符串视图现在标记为视图和租借范围。
- 为类型 std::u8string 和 std::u8string_view 以及 std::pmr::string、std::pmr::u8string、std::pmr::u16string、std::pmr::u32string 和 std::pmr::wstring 添加了标准哈希函数。

下面几节将解释其他章节中没有介绍和解释的重要改进。

23.1.1 字符串成员 starts_with() 和 ends_with()

字符串和字符串视图现在都有新的成员函数，starts_with() 和 ends_with()。它们提供了一种简单的方法，根据特定的字符序列检查字符串的开头和结尾字符。可以对单个字符、字符数组、字符串或字符串视图进行比较。

例如:

```
void foo(const std::string& s, std::string_view suffix)
{
  if (s.starts_with('.')) {
    ...
  }
  if (s.ends_with(".tmp")) {
    ...
```

```
  }
  if (s.ends_with(suffix)) {
    ...
  }
}
```

23.1.2 受限字符串成员 reserve()

对于字符串，成员函数 reserve() 不能再用于请求缩小字符串的容量 (为值分配的内存):

```
void modifyString(std::string& s)
{
  if ( ... ) {
    s.clear();
    s.reserve(0); // may no longer shrink memory (as before on some platforms)
    return;
  }
  ...
}
```

这样做的原因是释放内存可能需要一些时间，则在移植代码时，此调用的性能可能会有很大差异。

reserve() 需要传递参数:

```
s.reserve(); // ERROR since C++20
```

可以使用 shrink_to_fit() 代替:

```
s.shrink_to_fit(); // still OK
```

23.2. std::source_location

有时，让程序处理当前正在处理的源代码的位置很重要。这尤其用于记录、测试和检查不变量。到目前为止，开发者需要使用 C 预处理器宏__FILE__、__LINE__和__func__。C++20 为此引入了一个类型安全特性，这样就可以用当前源位置初始化对象，并且可以像传递其他对象一样传递该信息。

用法很简单:

```
#include <source_location>

void foo()
{
  auto sl = std::source_location::current();
  ...
  std::cout << "file: " << sl.file_name() << '\n';
  std::cout << "function: " << sl.function_name() << '\n';
```

```
  std::cout << "line/col: " << sl.line() << '/' << sl.column() << '\n';
}
```

静态 consteval 函数 std::source_location::current() 为 std::source_location 类型的当前源位置生成一个对象，其接口如下所示:

- file_name() 可产生文件的名称。
- function_name() 生成函数的名称 (若在函数之外调用则为空)。
- line() 生成行号 (当行号未知时可能为 0)。
- column() 生成行中的列 (当列号未知时可能为 0)。

函数名的确切格式和列的确切位置等细节可能会有所不同。例如，使用 gcc 库，输出可能如下所示:

```
file:     sourceloc.cpp
function: void foo()
line/col: 8/42
```

而 Visual C++ 的输出可能如下所示:

```
file:     sourceloc.cpp
function: foo
line/col: 8/35
```

通过在形参声明中使用 std::source_location::current() 作为默认实参，将获得函数调用的位置。例如:

```
void bar(std::source_location sl = std::source_location::current())
{
  ...
  std::cout << "file: " << sl.file_name() << '\n';
  std::cout << "function: " << sl.function_name() << '\n';
  std::cout << "line/col: " << sl.line() << '/' << sl.column() << '\n';
}

int main()
{
  ...
  bar();
  ...
}
```

输出可能是这样的:

```
file:     sourceloc.cpp
function: int main()
line/col: 34/6
```

或:

```
file:     sourceloc.cpp
function: main
line/col: 34/3
```

因为 std::source_location 是一个对象，所以可以将它存储在容器中并四处传递:

```
std::source_location myfunc()
{
  auto sl = std::source_location::current();
  ...
  return sl;
}

int main()
{
  std::vector<std::source_location> locs;
  ...
  locs.push_back(myfunc());
  ...
  for (const auto& loc : locs) {
    std::cout << "called: " << loc.function_name() << '\n';
  }
}
```

输出可能是:

```
called: ’ std::source_location myfunc()’
```

或:

```
called: ’ myfunc’
```

完整的示例请参见 lib/sourceloc.cpp。

23.3. 整型值和大小的安全比较

比较不同类型的整数值比预期的要复杂。本节介绍 C++20 处理此问题的两个新特性:

- 比较整型的工具
- std::ssize()

23.3.1 整数值的安全比较

比较和转换数字，即使是不同的数字类型，也应该是一项微不足道的任务，但事实并非如此。几乎所有的开发者都看到过关于代码这样做的警告。这些警告有一个很好的理由: 由于隐式转换，可能会编写不安全的代码而没有注意到它。

例如，大多数时候我们期望一个简单的 x < y 就可以了。考虑下面的例子:

```
int x = -7;
unsigned y = 42;

if (x < y) ... // OOPS: false
```

这种行为的原因是，当根据规则 (来自编程语言 C) 比较有符号值和无符号值时，有符号值被转换为无符号类型，这使其成为一个大的正整数值。

要修复此代码，必须编写以下代码:

```
if (x < static_cast<int>(y)) ... // true
```

若将小整数值与大整数值进行比较，可能会出现其他问题。

C++20 在 <utility> 中提供了安全的整数比较函数。“安全整数比较函数”表列出了这些函数，可以这样使用:

```
if (std::cmp_less(x, y)) ... // true
```

这种整数比较总是安全的。

常数	模板
std::cmp_equal(x, y)	表示 x 是否等于 y
std::cmp_not_equal(x, y)	表示 x 是否等于 y
std::cmp_less(x, y)	表示 x 是否小于 y
std::cmp_less_equal(x, y)	表示 x 是否小于等于 y
std::cmp_greater(x, y)	表示 x 是否大于 y
std::cmp_greater_equal(x, y)	表示 x 是否大于等于 y
std::in_range<T>(x)	生成 x 是否是类型 T 的有效值

表 23.1 安全整数比较函数

若传递的值是可以由传递的类型表示的值，则函数 std:: 在_range() 中产生 true:

```
bool b = std::in_range<int>(x); // true if x has a valid int value
```

只是生成 x 是否大于或等于所传递类型的最小值，是否小于或等于最大值。

这些函数不能用于比较 bool、字符类型或 std::byte 的值。

23.3.2 ssize()

我们经常需要将集合、数组或范围的大小作为带符号值。为了避免在这里出现警告:

```
for (int i = 0; i < coll.size(); ++coll) { // possible warning
  ...
}
```

问题是 size() 产生一个无符号值，若值非常大，有符号值和无符号值之间的比较可能会失败。
虽然可以将 i 声明为 unsigned int 或 std::size_t，但可能有一个很好的理由将其用作 int。
引入了一个辅助函数 std::ssize()，允许以下用法:

```
for (int i = 0; i < std::ssize(coll); ++coll) { // usually no warning
  ...
}
```

多亏了 ADL，当使用标准容器或其他标准类型时，编写以下代码就足够了:

```
for (int i = 0; i < ssize(coll); ++coll) { // OK for std types
  ...
}
```

这有时甚至是避免编译时错误所必需的，例子是锁存器、栅栏和信号量的初始化。
范围库在命名空间 std::ranges 中提供了 ssize()。

23.4. 数学常数

C++20 为数学浮点常量引入了常量，“数学常量”表列出了它们。
这些常量在头文件 <numbers> 的命名空间 std::numbers 中提供。

常量	模板
std::number::e	std::number::e_v<>
std::number::pi	std::number::pi_v<>
std::number::inv_pi	std::number::inv_pi_v<>
std::number::inv_sqrtpi	std::number::inv_sqrtpi_v<>
std::number::sqrt2	std::number::sqrt2_v<>
std::number::sqrt3	std::number::sqrt3_v<>
std::number::inv_sqrt3	std::number::inv_sqrt3_v<>
std::number::log2e	std::number::log2e_v<>
std::number::log10e	std::number::log10e
std::number::ln2	std::number::ln2_v<>
std::number::ln10	std::number::ln10_v
std::number::egamma	std::number::egamma_v<>

std::number::phi	std::number::phi_v<>

表 23.2 数学常量

这些常量是后缀为_v 的相应变量模板的 double 类型的特化，这些值是对应类型中最接近的表示值。例如:

```
namespace std::number {
  template<std::floating_point T> inline constexpr T pi_v<T> = ... ;
  inline constexpr double pi = pi_v<double>;
}
```

这些定义使用 std::floating_point 概念 (为此引入该概念)。

可以这样使用:

```
#include <numbers>
...

double area1 = rad * rad * std::numbers::pi;

long double area2 = rad * rad * std::numbers::pi_v<long double>;
```

23.5. 处理位的工具

C++20 为处理位提供了更好、更清晰的支持:

- 缺少底层位操作
- 位的强制转换
- 检查平台的端序

所有这些工具都在头文件 <bit> 中定义。

23.5.1 位操作

硬件通常对位操作有特殊的支持，比如“左移”或“右移”。C++20 之前，C++ 开发者不能直接访问这些指令，新引入的位操作为底层 CPU 的位指令提供了一个直接的 API。

“位操作”表列出了 C++20 引入的所有标准化的位操作，其在头文件 <bit> 中作为命名空间 std 中的独立函数提供。

操作	意义
rotl(val, n)	生成向左移 n 位的 val
rotr(val, n)	生成向右移 n 位的 val
countl_zero(val)	生成前置位中 (从最高有效位起)0 的个数

countl_one(val)	产生前置位中 (从最高有效位起)1 的个数
countr_zero(val)	产生后置位中 (从最低有效位起)0 的个数
countl_one(val)	产生后置位中 (从最低有效位起)1 的个数
popcount(val)	输出值中 1 位的个数
has_single_bit(val)	返回 val 是否为 2 的幂 (二进制集合)
bit_floor(val)	得到之前的 2 次幂值
bit_ceil(val)	得到下一个 2 次幂值
bit_width(val)	生成存储该值所需的位数

表 23.3 位操作

考虑下面的程序:

lib/bitops8.cpp

```
#include <iostream>
#include <format>
#include <bitset>
#include <bit>

int main()
{
  std::uint8_t i8 = 0b0000'1101;
  std::cout
    << std::format("{0:08b} {0:3}\n", i8) // 00001101
    << std::format("{0:08b} {0:3}\n", std::rotl(i8, 2)) // 00110100
    << std::format("{0:08b} {0:3}\n", std::rotr(i8, 1)) // 10000110
    << std::format("{0:08b} {0:3}\n", std::rotr(i8, -1)) // 00011010
    << std::format("{}\n", std::countl_zero(i8)) // four leading zeros
    << std::format("{}\n", std::countr_one(i8)) // one trailing one
    << std::format("{}\n", std::popcount(i8)) // three ones
    << std::format("{}\n", std::has_single_bit(i8)) // false
    << std::format("{0:08b} {0:3}\n", std::bit_floor(i8)) // 00001000
    << std::format("{0:08b} {0:3}\n", std::bit_ceil(i8)) // 00010000
    << std::format("{}\n", std::bit_width(i8)); // 4
}
```

该程序有以下输出:

```
00001101  13
00110100  52
10000110 134
00011010  26
4
1
3
false
```

```
00001000  8
00010000 16
4
```

注意事项如下:

- 只有当传递的类型是无符号整型时，才提供所有这些函数。
- 位移函数也取负 n，则改变了方向。
- 所有返回计数的函数的返回类型都是 int。唯一的例外是 bit_width()，会返回传递类型的值，这是标准中的不一致 (我认为是错误)。当将其作为 int 类型使用或直接打印时，可能必须使用静态强制转换。

若为 std::uint16_t 运行相应的程序 (参见 lib/bitops16.cpp)，会得到以下输出:

```
0000000000001101    13
0000000000110100    52
1000000000000110 32774
0000000000011010    26
12
1
3
false
0000000000001000     8
0000000000010000    16
4
```

这些函数仅为无符号整型定义。所以:

- 不能对有符号整型使用位操作:

```
int b1 = ... ;
auto b2 = std::rotl(b1, 2); // ERROR
```

- 不能对 char 类型使用位操作:

```
char b1 = ... ;
auto b2 = std::rotl(b1, 2); // ERROR
```

unsigned char 类型可以。

- 不能对 std::byte 类型使用位操作:

```
std::byte b1{ ... };
auto b2 = std::rotl(b1, 2); // ERROR
```

“位操作的硬件支持”表，摘自http://wg21.link/p0553r4，列出了一些新的位操作到现有硬件的可能映射。

操作	Intel/AMD	ARM	PowerPC
rotl()	ROL	-	rldicl
rotr()	ROR	ROR, EXTR	-
popcount()	POPCNT	-	popcntb
countl_zero()	BSR, LZCNT	CLZ	cntlzd
countl_one()	-	CLS	-
countr_zero()	BSF, TZCNT	-	-
countr_ont()	-	-	-

表 23.4 位操作的硬件支持

23.5.2 std::bit_cast<>()

C++20 提供了一个新的强制转换操作，用于更改位序列的类型。与使用 reinterpret_cast<> 或联合相比，操作符 std::bit_cast<> 确保位数合适，使用标准布局，并且不使用指针类型。

例如:

```
std::uint8_t b8 = 0b0000'1101;

auto bc = std::bit_cast<char>(b8); // OK
auto by = std::bit_cast<std::byte>(b8); // OK
auto bi = std::bit_cast<int>(b8); // ERROR: wrong number of bits
```

23.5.2 std::endian

C++20 引入了一个新的实用程序枚举类型 std::endian，可用于检查执行环境的端序。其引入了三个枚举值:

- std::endian::big，一个代表“大端”的值 (标量类型存储时，最重要的字节放在第一位，其余字节按降序排列)。
- std::endian::little，一个代表“1 小端”的值 (标量类型将最低有效位字节放在首位，其余字节按升序存储)。
- std::endian::native，用于指定执行环境的端序

若所有标量类型都是大端字节数，std::endian::native 等于 std::endian::big。若所有标量类型都是 little-endian, std::endian::native 等于 std::endian::little。std::endian::native 的值既不是 std::endian::big，也不是 std::endian::little。

若所有标量类型的大小都为 1，则 std::endian::little、std::endian::big 和 std::endian::native 都具有相同的值。

该类型在头文件 <bit> 中定义。

作为枚举值，这些值可以在编译时使用。例如:

```cpp
#include <bit>
...

if constexpr (std::endian::native == std::endian::big) {
  ... // handle big-endian
}
else if constexpr (std::endian::native == std::endian::little) {
  ... // handle little-endian
}
else {
  ... // handle mixed endian
}
```

23.6. <version>

C++20 引入了一个新的头文件 <version>，这个头文件不提供任何功能。相反，它提供了有关所使用的 C++ 标准库的所有特定于实现的通用信息。

例如，<version> 可能包含:

- C++ 标准库的版本和发布日期
- 版权信息

另外，<version> 定义了 C++ 标准库的特性测试宏 (也在其应用的头文件中定义)。

这个头文件很短，加载速度也很快，所以工具可以包含这个头文件来获取所有必要的信息，以便根据提供的特性集做出决策，或者找到处理所使用库的一般信息。

23.7. 算法的扩展

对于算法，C++20 提供了一些扩展 (其中一些已经在本书的其他章节中描述过了)。

23.7.1 支持范围

对于许多算法，现在都支持:

- 用于将整个范围 (容器，视图) 作为单个参数传递
- 用于将投影参数作为单个参数传递

这种支持要求在命名空间 std::ranges 中调用算法。

对于以下算法，目前还不支持范围:

- 数值算法 (如 accumulate())
- 并行执行算法
- 算法 lexicographical_compare_three_way()

有关所有标准算法的范围支持的更多详细信息，请参阅算法概述。

23.7.2 新算法

“新标准算法” 表列出了 C++20 中引入的标准算法。

名称	效果
min()	生成传递的范围的最小值
max()	生成传递的范围的最大值
minmax()	生成传递范围的最小值和最大值
shift_left()	将所有元素移到前面
shift_right()	将所有元素移到后面
lexicographical_compare_three_way()	使用运算符对两个范围排序 <=>

表 23.5 新标准算法

支持范围的 min(), max() 和 minmax()

范围库中引入了 std::ranges::min()、std::ranges::max() 和 std::ranges::minmax() 算法，以产生传递的范围的最小值和/或最大值。可以选择传递比较条件和投影。

例如:

lib/minmax.cpp

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

int main()
{
  std::vector coll{0, 8, 15, 47, 11};

  std::cout << std::ranges::min(coll) << '\n';
  std::cout << std::ranges::max(coll) << '\n';
  auto [min, max] = std::ranges::minmax(coll);
  std::cout << min << ' ' << max << '\n';
}
```

该程序有以下输出:

```
0
47
0 47
```

std 中没有接受两个迭代器的相应算法，只有函数接受两个值或一个 std::initializer_list<>(和一个比较条件) 作为参数。与往常一样，范围库还为其提供了传递投影的选项.

shift_left() 和 shift_right()

使用新的标准算法 shift_left() 和 shift_right，可以分别将元素移动到前面或后面。这些算法分别返回新的结束或开始。

例如:

lib/shift.cpp

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

void print (const auto& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

int main()
{
  std::vector coll{1, 2, 3, 4, 5, 6, 7, 8};

  print(coll);

  // shift one element to the front (returns new end):
  std::shift_left(coll.begin(), coll.end(), 1);
  print(coll);

  // shift three elements to the back (returns new begin):
  auto newbeg = std::shift_right(coll.begin(), coll.end(), 3);
  print(coll);
  print(std::ranges::subrange{newbeg, coll.end()});
}
```

该程序有以下输出:

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

可以通过执行策略来允许使用多个线程，但支持传递单参数范围和/或项目的范围 (可能是疏忽)。

lexicographical_compare_three_way()

为了以返回一种新的比较类别类型的方式比较两个不同容器中的元素，C++20 引入了 std::lexicographical_compare_three_way() 算法，在 <=> 操作符的章节中进行了描述。

对于这个算法，还不支持范围。既不能将范围作为单个参数传递，也不能传递投影参数 (这可能是一个疏忽)。

23.7.2 unseq 算法执行策略

C++17 为新引入的并行算法引入了各种执行策略，可以启用并行计算并允许线程并行地操作多个数据项 (称为向量化或 SIMD 处理)。

但不能允许算法在多个数据项上操作，而是限制算法仅使用一个线程，因此 C++20 现在提供了执行策略 std::execution::unseq。与往常一样，新执行策略是命名空间 std::execution 中对应的惟一类 unsequenced_policy 的 constexpr 对象。“执行策略”表列出了现在支持的所有标准化执行策略。

策略	意义
std::execution::seq	用一个线程顺序计算单个值
std::execution::par	用多个线程并行计算单个值
std::execution::unseq	用一个线程并行计算多个值 (C++20 起)
std::execution::par_unseq	用多个线程并行计算多个值

表 23.6 执行策略

非顺序执行策略允许在单个执行线程中交错执行传递给访问元素的操作，不应该将此策略用于阻塞同步 (例如使用互斥锁)，这可能会导致死锁。

下面是一个使用的例子:

lib/unseq.cpp

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

int main (int argc, char** argv)
{
  // init number of argument from command line (default: 1000):
  int numElems = 1000;
  if (argc > 1) {
    numElems = std::atoi(argv[1]);
  }

  // init vector of different double values:
  std::vector<double> coll;
```

```
  coll.reserve(numElems);
  for (int i=0; i<numElems; ++i) {
    coll.push_back(i * 4.37);
  }

  // process square roots:
  // - allow SIMD processing but only one thread
  std::for_each(std::execution::unseq, // since C++20
          coll.begin(), coll.end(),
          [](auto& val) {
          val = std::sqrt(val);
          });

  for (double value : coll) {
    std::cout << value << '\n';
  }
}
```

与通常的执行策略一样，无法影响何时以及如何使用策略。使用此策略，可以启用向量化或 SIMD 计算，但不能强制这样做。在不支持此策略的硬件上，或若实现在运行时决定不使用 (例如，因为 CPU 负载太高)，则 unseq 策略将会串行执行。

23.8. 附注

字符串成员函数 starts_with() 和 ends_with() 由 Mikhail Maltsev 在http://wg21.link/p0457r2中提出。

字符串的 reserve() 函数的限制最早是由 Andrew Luo 在http://wg21.link/lwg2968中提出，最终接受的提案是由 Mark Zeren 和 Andrew Luo 在http://wg21.link/p0966r1上制定。

访问源位置最早是由 Robert Douglas 在http://wg21.link/n3972中提出，最终接受的提案是由 Robert Douglas、Corentin Jabot、Daniel Krügler 和 Peter Sommerlad 在http://wg21.link/p1208r6上制定。

采用 Federico Kircheis 在http://wg21.link/p0586r2中提出的安全整型比较函数。

ssize() 函数作为比较有符号索引和无符号索引大小的问题已经讨论了很长时间。我们希望让所有 size() 函数产生一个带符号的值，但存在向后兼容性问题。作为对 std::span 建议使用带符号大小类型的回应，ssize() 首先由 Robert Douglas、Nevin Liber 和 Marshall Clow 在http://wg21.link/p1089r0中提出。最终接受的提案由 Jorg Brown 在http://wg21.link/p1227r2中阐述。std::ranges::ssize() 后来由 Hannes Hauswedell、Jorg Brown 和 Casey Carter 在http://wg21.link/p1970r2中添加。

数学常数最初是由 Lev Minkovsky 在http://wg21.link/p0631r0中提出，最终接受的提案是由 Lev Minkovsky 和 John McFarlane 在http://wg21.link/p0631r8上提出。

位操作最早是由 Matthew Fioravante 在http://wg21.link/n3864上提出，最终接受的提案由 Jens Maurer 在http://wg21.link/p0553r4和http://wg21.link/p0556r3中制定。这些名字后来由 Vincent Reverdy 在http://wg21.link/p1956r1上提出修改。

std::bit __ cast<>() 是 JF Bastien 在http://wg21.link/p0476r2中提出。

std::endian 最早由 Howard Hinnant 在http://wg21.link/p0463r0中提出，最终接受的提案是 Howard Hinnant 在http://wg21.link/p0463r1中提出。头文件后来由 Walter E. Brown 和 Arthur O 'Dwyer 在http://wg21.link/p1612r1中修正。

头文件 <version> 是由 Alan Talbot 在http://wg21.link/p0754r2中提出。

新算法 shift_left() 和 shift_right() 由 Dan Raviv 在http://wg21.link/p0769r2中提出，在http://wg21.link/p0896r4中提出的范围库中引入了新的范围最小/最大算法。

unseq 执行策略首先由 Arch D. Robison、Pablo Halpern、Robert Geva 和 Clark Nelson 在http://wg21.link/p0076r0上提出，最终接受的提案是由 Alisdair Meredith 和 Pablo Halpern 在http://wg21.link/p1001r2上制定。

第 24 章 已弃用和已删除的功能

C++20 中有一些特性已经弃用或移除。

实现可能仍然提供已删除的功能，但不能依赖于此。若使用不推荐的特性，实现可能会发出警告。

24.1. 已弃用和删除的核心语言功能

- 不推荐使用 *this 的隐式捕获。
- 聚合可能不再有任何用户声明 (但不是用户提供) 的构造函数。

24.2. 已弃用和已删除的库功能

24.2.1 已弃用的库功能

以下库特性自 C++20 以来已弃用，不应再使用:

- 类型特征 is_pod<>。
  使用 is_trivial<> 或相似的类型特征替代。
- 普通 shared 指针的原子操作现在已经弃用，使用原子 shared 指针代替。

24.2.2 删除的库功能

- 对于字符串，成员函数 reserve() 不能再在没有参数的情况下调用，它也不再缩减容量。
- 不能再将 UTF-8 字符串写入标准输出流。

24.3. 附注

Jens Maurer 在http://wg21.link/p0767r1中提议弃用 is_pod<>。

术语表

本术语表提供了本书中术语的简短定义。

A

聚合，aggregate

聚合是简单的类或结构类型，具有限制，因此重点在于存储数据，而不是提供复杂的行为。其来自编程语言 C，历史上是原始数组、结构体以及这两个特性的组合，可以用花括号对其进行初始化。C++ 中，关键的要求是聚合只有公共数据成员，没有构造函数，也没有虚函数。这些要求在 C++20 中略有变化，现在可以用括号初始化聚合。

参数相关查找，argument-dependent lookup (ADL)

当一个参数在函数的命名空间中时，允许开发者跳过函数的命名空间限定的特性，所以会在传递的参数的所有命名空间中查找函数。

C

类模板参数推导，class template argument deduction (CTAD)

从使用类模板的上下文中隐式确定模板参数的过程，这是在 C++17 中引入的，当模板形参可以从构造函数中推导出来时，还可跳过对象模板实参的说明。

F

转发引用，forwarding reference

C++ 标准用于通用引用的术语。

全特化，full specialization

不再依赖于模板参数 (主) 模板的替代定义。

函数对象，function object (函子，functor)

可以用作函数的对象，operator() 在类中定义。所有 Lambda 都是函数对象。

G

glvalue

生成存储值 (广义可本地化值) 的位置的表达式的值类别。glvalue 可以是左值或 xvalue。

I

不完整的类型，incomplete type

已声明但未定义的类、结构或未限定作用域的枚举类型、未知大小的数组、void(可选择带有 const 和/或 volatile) 或不完整元素类型的数组。

L

lvalue

一种表达式的值类别，为不可移动的存储值 (即，不是 xvalue 的 glvalues) 生成位置。例子有:

- 命名对象 (变量)
- 字符串字面值
- 返回左值引用
- 函数和所有对函数的引用
- 左值的数据成员

P

偏特化，partial specialization

仍然依赖于一个或多个模板参数的 (主) 模板的替代定义。

谓词，predicate

一个可调用对象 (函数、函数对象或 Lambda)，用于检查某个标准是否适用于一个或多个参数，会返回一个布尔值，是只读的，无状态的。

纯右值，prvalue

执行初始化的表达式的值类别。可以假定值指定纯数学值 (如 1 或 true) 和没有名称的临时对象。例子有:

- 除字符串外的所有字面值 (42、true、nullptr 等)
- 返回值 (不是通过引用返回的值)
- 返回左值引用
- 构造函数调用的结果
- Lambda
- this

在 C++11 之前称为 rvalue，而从 C++11 开始称为 prvalue。

R

资源获取即初始化，resource acquisition is initialization (RAII)

一种编程模式，将结束使用资源所需的清理委托给析构函数，以便在表示资源的对象离开其作用域或结束其生命周期时自动进行清理。

普通类型，regular type

与内置值类型 (如 int) 的语义匹配的类型。根据http://stepanovpapers.com/DeSt98.pdf中的定义，正则类型提供以下基本操作:

- 默认构造 (T x;)
- 复制 (T y = x;) 和可在 C++ 中移动
- 赋值 (x = y;) 和可在 C++ 中移动赋值
- 支持等式和不等式 (x == y 和 x != y)
- 可排序 (x < y 等)

这些操作遵循“普通”朴素规则:

- 若一个对象是另一个对象的副本，则两个对象相等。
- 对象的副本等于用指定源值的默认构造函数创建的对象。
- 对象具有值语义。若两个对象相等，修改其中一个，就不再相等了。

右值，rvalue

非左值表达式的值类别。右值可以是右值 (没有名称的临时对象) 或 xvalue(用 std::move() 标记的左值)。在 C++11 之前称为右值的东西，从 C++11 后称为 prvalue。

S

半普通类型，semiregular type

普通类型，但不提供比较操作符:

- 若一个对象是另一个对象的副本，则两个对象相等。
- 对象的副本等于用指定源值的默认构造函数创建的对象。
- 对象具有值语义。若两个对象相等，修改其中一个，就不再相等了。

替换失败不是错误，substitution failure is not an error (SFINAE)

一种机制，当模板形参的实参声明格式错误时，该机制静默地丢弃模板，而不是触发编译错误。可以使用该机制，通过强制格式错误的声明来禁用某些参数的模板。

小字符串优化，small string optimization (SSO)

通过始终为一定数量的字符保留内存来节省为短字符串分配内存的方法。标准库实现中的典型值是始终保留 16 或 24 字节的内存，以便字符串可以有 15 或 23 个字符 (加上 null 终止符的 1 字节)，而无需分配内存。这使得所有的字符串对象都变大，但通常节省了大量的运行时间。实践中，字符串通常短于 16 或 24 个字符，并且在堆上分配内存是一个相当昂贵的操作。

无状态，stateless

若函数或操作不因调用而改变其状态，则是无状态的。这样做的效果是，不会随着时间的推移而改变其行为，并且总是为相同的参数产生相同的结果。

标准模板库，standard template library (STL)

STL 是 C++ 标准库的一部分，用于处理容器 (数据结构)、算法以及作为粘合剂的迭代器。第一个 C++ 标准采用了这种方法，其目标是开发者可以从各种标准数据结构和算法中受益，而不必实现。作为泛型代码，当尝试组合不受支持的内容时，仍然会得到编译时错误消息。

U

通用引用，universal reference

可以通用引用任何对象，而不使其为 const 的引用。还可以延长返回值的生命周期，可声明为模板参数的右值引用 (T&&) 或 auto&&。通用引用对于使用 std::forward<>() 完美地转发参数非常有用 (因此，C++ 标准将其命名为转发引用)。

然而，也是唯一一种可以引用表达式 (左值和右值)，而不使值为 const 的引用，所以对于声明接受视图的泛型函数参数是必要的。

V

值类别，value category

表达式的分类，传统的值类别左值和右值继承自编程语言 C。C++11 引入了额外的类别，所以有以下主要的值类别:

- prvalues (pure rvalues)，用于初始化对象 (包括参数) 的值
- lvalues (localizable values)，可以查询地址的对象
- xvalues (eXpiring values)，是不再需要值的对象 (通常是用 std::move() 标记的对象)。

此外，C++ 有以下组合值类别:

- glvalues (generalized lvalues)，表示“要么是 lvalue，要么是 xvalue”。
- rvalues (readable values)，表示“要么是 rvalue，要么是 xvalue”。

变量模板，variable template

一个通用变量，允许使用特定类型或值替换模板形参来定义变量或静态成员。

可变模板，variadic template

具有模板参数的模板，该模板参数表示任意数量的类型或值。

X

xvalue

表达式的值类别，可以假定不再需要的存储对象生成位置。例子有:

- 用 std::move() 标记的值
- 返回的 rvalue 引用
- 将对象 (不是函数) 强制转换为右值引用
- 值成员的 rvalue