



C++20

Get the Details

作者: Rainer Grimm

译者: 陈晓伟

本书概述

这本书既是 C++20 标准的教程，也是参考资料。会教你如何使用 C++20，并提供新 C++ 标准的细节。其中，最主要是 C++20 的四大特性。

概念 (Concept) 改变了模板的思考和编程方式，使用模板参数语义，可在类型系统中直接表达可以使用的类型。若出现错误，将出现一条明确的编译错误信息。

新的“范围”库，能够直接在容器上执行算法，用管道符号组合算法，并可应用到无限数据流上。

协程 (Coroutines) 的加入，使得异步编程成为 C++ 的主流。协程是协作任务、事件循环、无限数据流和管道的基础。

模块 (Modules) 克服了头文件的限制，例如：头文件和源文件的分离和预处理器一样过时。并且，其可以更快的构建代码和更简单的构建包。

将会了解

- 自动生成的比较运算符
- 日期和时区库
- 格式库
- 连续的内存块
- 加强版可中断线程
- 原子智能指针
- 信号量
- 协调原语，如锁存和栅栏

作者简介

Rainer Grimm 自 1999 年以来一直担任软件架构师、团队领导和讲师。2002 年，为公司组织了实习生会议。从 2002 年起，就开始开设培训课程，第一个教程关于专业管理软件，但不久之后开始教授 Python 和 C++。在业余时间，喜欢写关于 C++，Python 和 Haskell 的文章，也喜欢在会议上发言。每周都会在英语博客[Modernes Cpp](#)和由[German blog](#)主办的德语博客上发表文章。

自 2016 年以来，其一直作为一名独立讲师，在研讨会上讲授现代 C++ 和 Python。并用不同的语言出版了几本关于现代 C++ 的书，特别是关于并发性的技术书籍。由于其职业原因，他一直都在寻找教授现代 C++ 的最佳方法。

Cippi 简介

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



我是 Cippi，我好奇、聪明，并且“女人味”十足!!

Beatrix 绘制了 Cippi。

本书相关

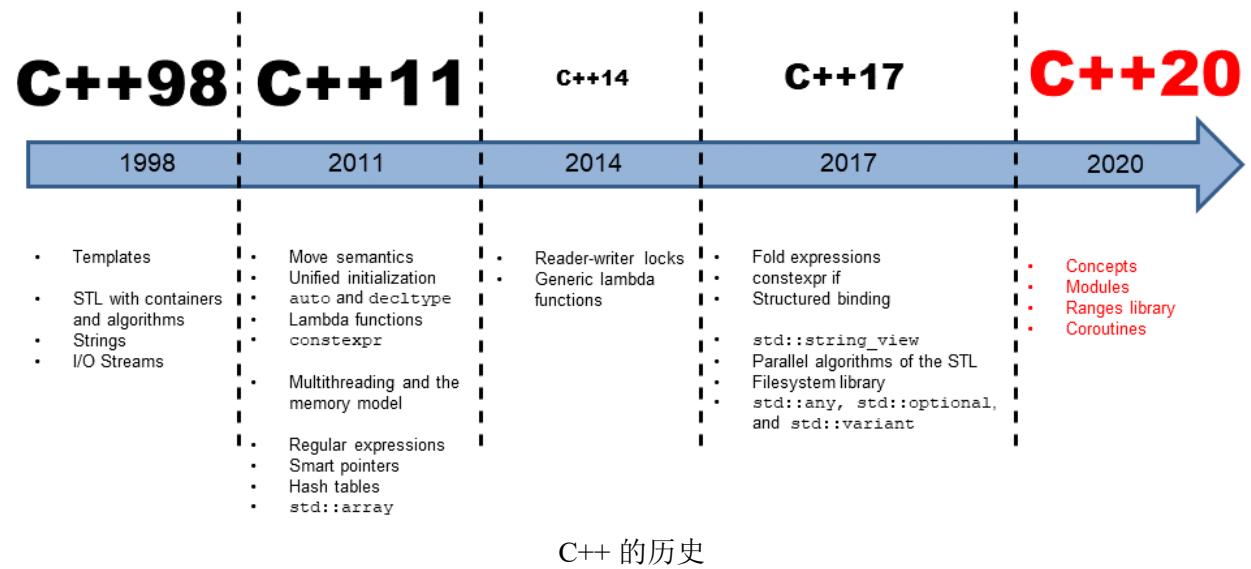
- Github 地址: <https://github.com/xiaoweichen/CXX20-Get-Details>

目录

第一部分：关于 C++

第 1 章 历史背景

C++20 是继 C++11 之后的一个重要的标准，也会改变我们使用现代 C++ 编程的方式。这种变化主要是因为概念、模块、范围和协程等特性。为了给进一步了解 C++ 发展打下良好的基础，先在这里简单介绍一下有 C++20 的历史背景。



C++ 出现已经有 40 年了，以下是对过去几年变化的简要概述。

1.1. C++98

80 年代末，Bjarne Stroustrup 和 Margaret A. Ellis 撰写了著名的[Annotated C++ 参考手册\(ARM\)](#)，其定义了 C++ 的功能，并为第一个 C++ 标准 C++98 (ISO/IEC 14882) 提供基础支持。C++98 支持一些基本特性，比如：模板、标准模板库 (STL) 及其容器、算法、字符串和 IO 流。

1.2. C++03

C++03(14882:2003) 中，对 C++98 进行了小幅度的技术修正，以至于没有出现在上面的时间轴中。社区中，C++03(包括 C++98) 也称为“过时”的 C++。

1.3. TR1

2005 年，令人兴奋的事情发生了，技术报告 1(TR1) 发布了。TR1 是迈向 C++11 的一大步，也是迈向现代 C++ 的一大步。TR1(TR 19768) 基于[Boost](#)，该项目由 C++ 标准化委员会的成员创建。TR1 有 13 个库，其注定会成为 C++11 标准的一部分。例如，正则表达式库、随机数库、智能指针和哈希表。只有一些特殊数学函数，需要等到 C++17 才能加入标准中。

1.4. C++11

我们把 C++11 标准为现代 C++ 开端。“现代 C++”也适用于 C++14 和 C++17。C++11 引入了许多特性，从根本上改变了如今的编程方式。C++11 增加了 TR1，也增加了移动语义、完美转发、可变参数模板和 `constexpr`，这还不是全部。C++11 中，作为线程的基本基础和线程 API 的标准化，第一次加入了内存模型。

1.5. C++14

C++14 是一个小型的 C++ 标准，带来了读写锁、广义 Lambda 表达式和扩展的 `constexpr` 函数。

1.6. C++17

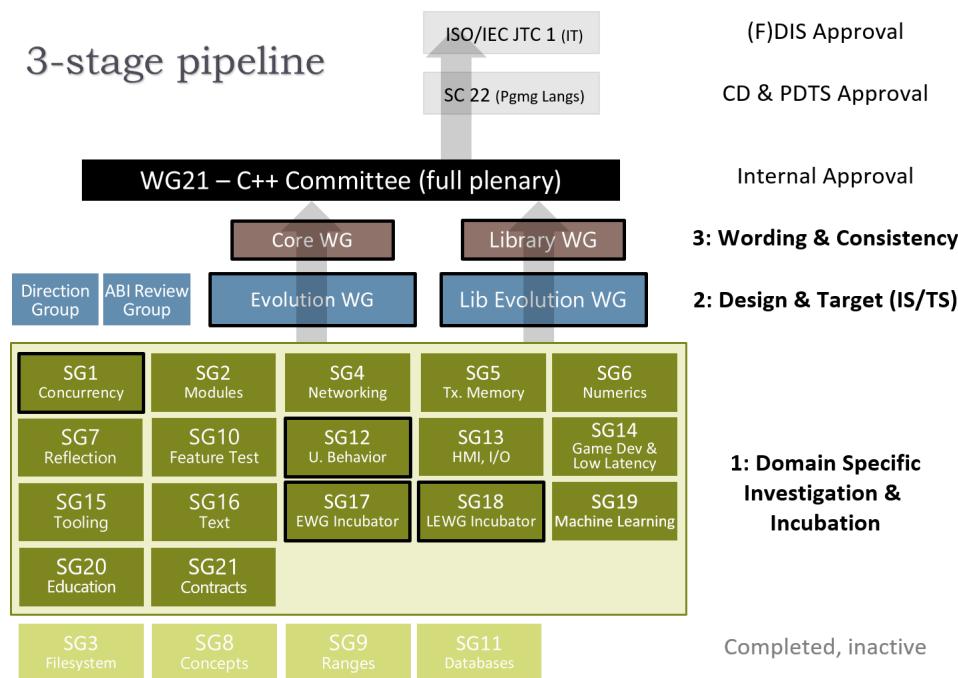
C++17 是一个不大不小的标准，有两个突出的特性：并行的 STL 和标准化的文件系统 API。标准模板库中大约有 80 种算法可以并行或向量化地执行。与 C++11 一样，boost 库对 C++17 的影响很大。Boost 提供了文件系统库和新的数据类型：`std::string_view`、`std::optional`、`std::variant` 和 `std::any`。

第 2 章 标准化

C++ 的标准化过程很民主，委员会称为 WG21(21 工作组)，成立于 1990-1991 年。21 工作小组的成员如下：

- 召集人: 主持 WG21 会议，制定会议日程，并任命研究小组
- 项目编辑: 对 C++ 标准的草案进行修改
- 秘书: 对 WG21 会议进行记录

该图展示了委员会的各个子小组和研究小组。



研究小组在 C++ 标准化过程中组成情况

委员会分为三个阶段，由几个小组组成。SG 代表研究组 (Study Group)。

2.1. 阶段 3

阶段 3 的用词和更改建议的一致性分为两组: 核心语言用词 (CWG) 和库用词 (LWG)。

2.2. 阶段 2

阶段 2 分为两组: 核心语言进化 (EWG) 和库进化 (LEWG)。EWG 和 LEWG 分别负责涉及语言和标准库扩展的新特性。

2.3. 阶段 1

阶段 1 的目标是针对特定领域的调查和孵化。研究小组的成员通过面对面的会议，也会在会议间隙，通过电话或视频会议进行交流。中央小组可审查各研究小组的工作，以确保一致性。

这些是领域特定的研究小组:

- SG1, 并发性 (Concurrency): 并发性和并行性主题, 包括内存模型
- SG2, 模块 (Module): 模块相关的主题
- SG3, 文件系统
- SG4, 网络 (Networking): 网络库开发
- SG5, 事务性内存 (Transactional Memory): 用于添加的事务性内存结构
- SG6, 数字相关 (Numerics): 数字主题, 如定点数、浮点数和分数
- SG7, 编译时编程 (Compile time programming): 泛型 (编译时) 编程
- SG8, 概念 (Concepts)
- SG9, 范围 (Ranges)
- SG10, 特性测试 (Feature Test): 用于测试 C++ 特性是否支持, 以及可移植性检查
- SG11, 数据库 (Database): 与数据库相关的库接口
- SG12, 未定义行为和漏洞 (UB&Vulnerabilities): 针对标准中的漏洞和未定义/未指定行为的改进
- SG13, HMI & I/O(人/机交互接口): 支持输出和输入设备
- SG14, 游戏开发和低延迟 (Game Development & Low Latency): 游戏开发人员和(其他)低延迟编程
- SG15, 工具 (Tooling): 开发人员工具, 包括模块和包
- SG16, Unicode: C++ 中的 Unicode 文本处理
- SG17, EWG 孵化器 (Incubator): 关于核心语言演化的早期讨论
- SG18, LEWG 孵化器 (Incubator): 关于库语言演化的早期讨论
- SG19, 机器学习: 人工智能 (AI) 的特定主题, 也包括线性代数
- SG20, 教育 (Education): C++ 教材指南
- SG21, 契约 (Contract): 契约式设计的语言支持
- SG22, C 与 C++ 的兼容 (C/C++ Liaison): 讨论 C 和 C++ 的协调兼容问题

本节简要介绍了 C++ 的标准化, 特别是 C++ 委员会的标准化。读者们可以在<https://isocpp.org/std>上找到更多关于标准化的信息。

第二部分：概述 C++20

第3章 C++20

深入研究 C++20 的细节之前，先简单介绍一下 C++20 的特性。

这有两个目的：先有个第一印象，这里会提供相关部分的链接，也可以直接了解细节。本章只有代码片段，不提供完整的程序。

本书开始就简要介绍了以前的 C++ 标准。比较 C++20 和以前的版本时，并通过提供历史背景说明了 C++20 的重要性。

C++20

2020

The Big Four

- Concepts
- Modules
- Ranges library
- Coroutines

Core Language

- Three-way comparison operator
- Designated initialization
- `constexpr` and `constinit`
- Template improvements
- Lambda improvements
- New attributes

Library

- `std::span`
- Container improvements
- Arithmetic utilities
- Calendar and time zone
- Formatting library

Concurrency

- Atomics
- Semaphores
- Latches and barriers
- Cooperative interruption
- `std::jthread`

C++20 有四个突出的特性：概念、范围、协程和模块。每个都有自己独立的章节。

3.1. 四大新特性

C++20

2020

The Big Four

- Concepts
- Modules
- Ranges library
- Coroutines

Core Language

- Three-way comparison operator
- Designated initialization
- `constexpr` and `constinit`
- Template improvements
- Lambda improvements
- New attributes

Library

- `std::span`
- Container improvements
- Arithmetic utilities
- Calendar and time zone
- Formatting library

Concurrency

- Atomics
- Semaphores
- Latches and barriers
- Cooperative interruption
- `std::jthread`

每一个特性都会改变使用 C++ 的方式。

3.1.1 概念

使用模板的泛型编程，使其能够定义可用于各种类型的函数和类。因此，使用错误类型实例化模板的情况并不少见，因此会看到长达数页的编译错误，而这个问题可以通过概念解决。概念可编写编译器检查模板参数的需求，并彻底改变开发者思考和编写泛型代码的方式：

- 模板参数的需求成为公共接口的一部分。
- 函数的重载或类模板的特化可以基于概念。

- 因为编译器会根据给定的模板参数检查已定义的模板形参需求，所以可以得到了更明确的错误消息。

这还没完。

- 开发者可以使用预定义的概念，也可以自定义概念。
- `auto` 和概念的用法统一，可以用概念来代替 `auto`。
- 若函数声明使用了概念，则自动成为函数模板。编写函数模板就像编写函数一样简单。

下面的代码片段演示了概念 `Integral` 的定义和使用：

```

1 template <typename T>
2 concept Integral = std::is_integral<T>::value;
3
4 Integral auto gcd(Integral auto a, Integral auto b) {
5     if( b == 0 ) return a;
6     else return gcd(b, a % b);
7 }
```

`Integral` 概念从其类型参数 `T` 中要求 `std::is_integral<T>::value` 为 `true`。`std::is_integral<T>::value` 来自[类型特征库](#)，在编译时检查 `T` 是否为整数。若 `std::is_integral<T>::value` 为 `true`，则一切正常；否则，将会看到相应的编译时错误。

`gcd` 算法基于[Euclidean](#)确定最大公约数，代码使用缩写函数模板语法来定义 `gcd`。这里，`gcd` 要求参数和返回类型支持 `Integral` 概念。换句话说，`gcd` 是一种对其参数和返回值类型有要求的函数模板。

下面是语义等效的 `gcd` 算法，使用了 `require` 子句。

```

1 template<typename T>
2 requires Integral<T>
3 T gcd(T a, T b) {
4     if( b == 0 ) return a;
5     else return gcd(b, a % b);
6 }
```

`require` 子句声明了对 `gcd` 参数类型的要求。

3.1.2 模块

模块的作用有很多：

- 更快的编译时间
- 减少宏定义
- 表达代码的逻辑结构
- 淘汰头文件
- 摆脱宏替换

这是一个简单的数学模块：

```

1 export module math;
2
```

```
3 export int add(int fir, int sec) {
4     return fir + sec;
5 }
```

表达式会导出模块 math(第 1 行) 是模块声明，将 export 放在函数 add 之前(第 3 行) 导出函数。现在，可以使用它了。

```
1 import math;
2
3 int main() {
4     add(2000, 20);
5 }
```

表达式 import math 导入 math 模块，并使导出名在当前作用域中可见。

3.1.3 范围库

范围库支持的算法有如下特征

- 可直接在容器上操作，不需要迭代器来指定范围
- 可以延迟求值
- 可以组合

简而言之：范围库支持函数模式。

下面的例子展示了如何组合管道操作符。

```
1 int main() {
2     std::vector<int> ints{0, 1, 2, 3, 4, 5};
3     auto even = [] (int i){ return i % 2 == 0; };
4     auto square = [] (int i) { return i * i; };
5
6     for (int i : ints | std::views::filter(even) |
7           std::views::transform(square)) {
8         std::cout << i << ' '; // 0 4 16
9     }
10 }
```

even(第 3 行) 是 Lambda 表达式，若参数 i 是偶数，则返回 true。Lambda 表达式 square(第 4 行) 将计算参数 i 为的平方。第 6 行和第 7 行演示了函数组合，必须从左到右阅读:for (int i: ints | std::views::filter(even) | std::views::transform(square))。对 int 类型的每个元素应用偶数过滤器，并将每个剩余元素计算其平方值。若熟悉函数式编程，那么这段代码看起来就很容易。

3.1.4 协程

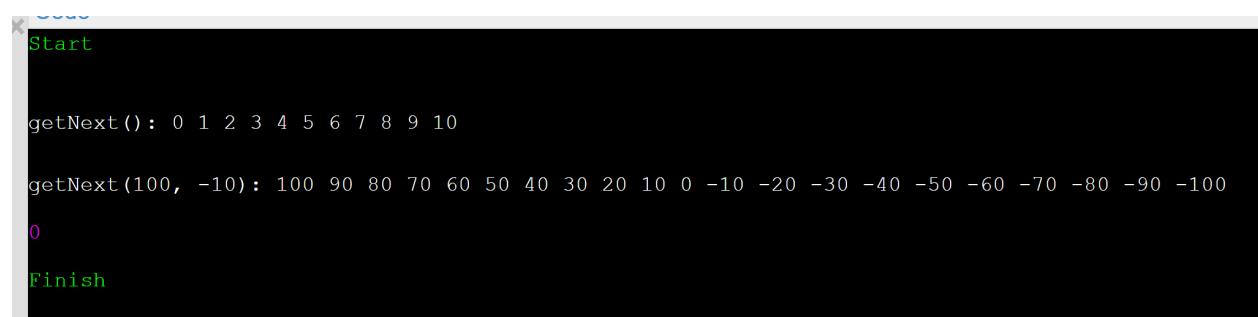
协程是可以在保持其状态的同时挂起并稍后恢复的函数，协程是编写事件驱动应用程序的一种方法，通常也用于协作多任务处理。事件驱动的应用程序可以是模拟、游戏、服务器、UI，甚至算法。

C++20 没有提供协程库，而提供了一个实现协程的框架。该框架由 20 多个函数组成，其中一些必须实现，一些可选实现。因此，可以根据自己的需要定制协程。

下面的代码片段使用生成器创建无限的数据流。本章协程提供了生成器实现的版本。

```
1 Generator<int> getNext(int start = 0, int step = 1) {
2     auto value = start;
3     while (true) {
4         co_yield value;
5         value += step;
6     }
7 }
8
9 int main() {
10
11     std::cout << '\n';
12
13     std::cout << "getNext():";
14     auto gen1 = getNext();
15     for (int i = 0; i <= 10; ++i) {
16         gen1.next();
17         std::cout << " " << gen1.getValue();
18     }
19
20     std::cout << "\n\n";
21
22     std::cout << "getNext(100, -10):";
23     auto gen2 = getNext(100, -10);
24     for (int i = 0; i <= 20; ++i) {
25         gen2.next();
26         std::cout << " " << gen2.getValue();
27     }
28
29     std::cout << "\n";
30 }
31 }
```

因为 `co_yield` 关键字，所以 `getNext` 是协程。这里有一个无限循环，会在 `co_yield` 处返回值（第 4 行），在 `next`（第 16 行和第 25 行）处恢复运行，然后调用 `getValue` 获取值。在 `getNext` 调用返回之后，协程再次暂停，直到下一次调用 `next`。在这个例子中有一个迷：`getNext` 函数的返回值 `Generator<int>`。这就开始复杂了，我将在协程部分中深入地进行讲解。



```
Start

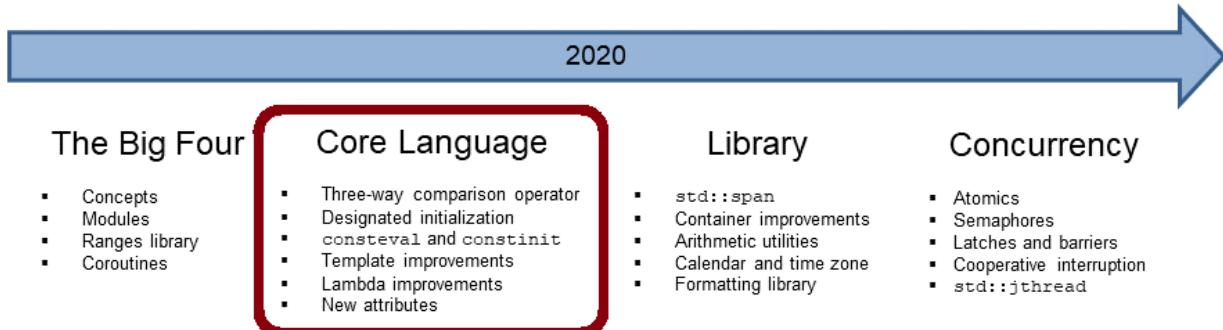
getNext(): 0 1 2 3 4 5 6 7 8 9 10

getNext(100, -10): 100 90 80 70 60 50 40 30 20 10 0 -10 -20 -30 -40 -50 -60 -70 -80 -90 -100
0

Finish
```

3.2. 核心语言特性

C++20



3.2.1 三向比较操作符

三向比较操作符 `<=>`, 或宇宙飞船操作符, 确定对于两个值 A 和 B 的大小关系, $A < B$, $A == B$, 或 $A > B$ 。

通过将三向比较操作符声明为 `default`, 编译器将尝试为该类自动生成关系操作符。在本例中, 将得到所有六个比较运算符: `==`、`!=`、`<`、`<=`、`>` 和 `>=`。

```
1 struct MyInt {
2     int value;
3     MyInt(int value) : value{value} { }
4     auto operator<=>(const MyInt&) const = default;
5 };
```

编译器生成的操作符 `<=>` 执行字典比较, 从基类开始, 按照声明顺序考虑所有非静态数据成员。下面的复杂例子, 来自于 Microsoft 的博客:[使用“航天学”简化代码:C++20 的宇宙飞船操作符](#)。

3.2.2 指定初始化

```
1 struct Basics {
2     int i;
3     char c;
4     float f;
5     double d;
6     auto operator<=>(const Basics&) const = default;
7 };
8
9 struct Arrays {
10    int ai[1];
11    char ac[2];
12    float af[3];
13    double ad[2][2];
14    auto operator<=>(const Arrays&) const = default;
15 };
```

```

16
17 struct Bases : Basics, Arrays {
18     auto operator<=>(const Bases&) const = default;
19 };
20
21 int main() {
22     constexpr Bases a = { { 0, 'c', 1.f, 1. },
23         { { 1 }, { 'a', 'b' }, { 1.f, 2.f, 3.f }, { { 1., 2. }, { 3., 4. } } } };
24     constexpr Bases b = { { 0, 'c', 1.f, 1. },
25         { { 1 }, { 'a', 'b' }, { 1.f, 2.f, 3.f }, { { 1., 2. }, { 3., 4. } } } };
26     static_assert(a == b);
27     static_assert(!(a != b));
28     static_assert(!(a < b));
29     static_assert(a <= b);
30     static_assert(!(a > b));
31     static_assert(a >= b);
32 }

```

假设这段代码中最复杂的不是太空船操作符，而是使用聚合初始化初始化 Base。聚合初始化意味着若所有成员都是 public，则可以直接初始化类类型 (class、struct 或 union) 的成员。所以，可以使用带括号的初始化列表，如示例所示。

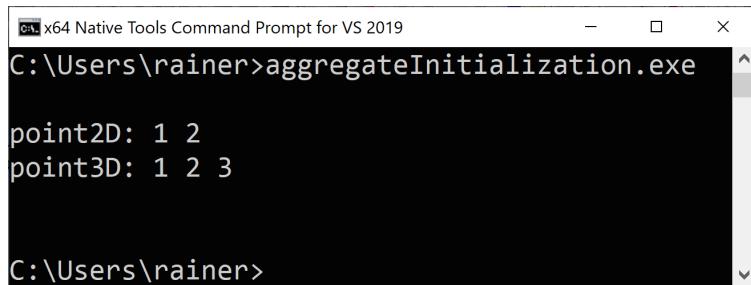
讨论指定初始化之前，先来详细介绍一下聚合初始化。先来看一个简单的例子。

```

1 struct Point2D{
2     int x;
3     int y;
4 };
5
6 class Point3D{
7     public:
8     int x;
9     int y;
10    int z;
11 };
12
13 int main(){
14     std::cout << "\n";
15     Point2D point2D {1, 2};
16     Point3D point3D {1, 2, 3};
17     std::cout << "point2D: " << point2D.x << " " << point2D.y << "\n";
18     std::cout << "point3D: " << point3D.x << " "
19     << point3D.y << " " << point3D.z << "\n";
20     std::cout << '\n';
21 }

```

输出为：



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

point2D: 1 2
point3D: 1 2 3

C:\Users\rainer>
```

因为可以交换构造函数参数，所以聚合初始化非常容易出错，而且不太会注意到，所以显式方式要好于隐式方式。再来看下[C99](#)(现在是 C++ 标准的一部分) 中的指定初始化：

```
1 struct Point2D{
2     int x;
3     int y;
4 };
5
6 class Point3D{
7 public:
8     int x;
9     int y;
10    int z;
11 };
12
13 int main(){
14
15     Point2D point2D {.x = 1, .y = 2};
16     // Point2D point2d {.y = 2, .x = 1}; // error
17     Point3D point3D {.x = 1, .y = 2, .z = 2};
18     // Point3D point3D {.x = 1, .z = 2} // {1, 0, 2}
19
20
21     std::cout << "point2D: " << point2D.x << " " << point2D.y << "\n";
22     std::cout << "point3D: " << point3D.x << " " << point3D.y << " " << point3D.z
23         << "\n";
24
25 }
```

Point2 和 Point3D 实例的参数显式命名，该程序的输出与前一个程序的输出完全相同。注释掉的第 16 行和第 18 行非常有趣。第 16 行会报错，因为指示符的顺序与数据成员的声明顺序不匹配。至于第 18 行，y 的指示符不见了，所以 y 会初始化为 0，等效于使用带括号的初始化列表 {1,0,2}。

3.2.3 consteval 和 constinit

C++20 中新增的 `consteval` 说明符创建了一个即时函数。所谓即时函数，每次调用该函数都会生成一个编译时常量表达式。即时函数是隐式 `constexpr` 函数，但不一定是 `constexpr` 函数。

```
1 consteval int sqr(int n) {
2     return n*n;
3 }
```

```

4 constexpr int r = sqr(100); // OK
5
6 int x = 100;
7 int r2 = sqr(x); // Error

```

因为 x 不是一个常量表达式，最后的赋值会出错，因此不能在编译时执行 `sqr(x)`。

`constinit`(详见 4.5.2 节) 确保静态存储时间段或线程存储时间段的变量，在编译时初始化。静态存储时间段意味着对象在程序开始时分配，在程序结束时释放。线程存储时间段，意味着对象的生存期与线程的生存期绑定。

`constinit` 确保在编译时对这类变量(静态存储时间段或线程存储时间段)进行初始化。不过，`constinit` 并不意味着常量。

3.2.4 模板的改进

C++20 对模板编程提供了各种改进。泛型构造函数是一种万能构造函数，可以用任何类型使用。

隐式和显式泛型构造函数

```

1 struct Implicit {
2     template <typename T>
3     Implicit(T t) {
4         std::cout << t << '\n';
5     }
6 };
7
8 struct Explicit {
9     template <typename T>
10    explicit Explicit(T t) {
11        std::cout << t << '\n';
12    }
13 };
14
15 Explicit exp1 = "implicit"; // Error
16 Explicit exp2{"explicit"};

```

隐式类的构造函数很泛型，通过将关键字 `explicit` 放在构造函数前面，就像显式构造函数一样，从而隐式转换不再有效。

3.2.5 Lambda 表达式的改进

Lambda 在 C++20 中进行了许多改进，可以有模板参数，可以在未求值的上下文中使用，无状态的 Lambda 也可以默认构造和复制赋值。此外，编译器现在可以检测何时隐式复制 `this` 指针。

若想定义一个只接受 `std::vector` 的 Lambda 表达式，模板参数可以这样写：

```

1 auto foo = []<typename T>(std::vector<T> const& vec) {
2     // do vector-specific stuff
3 };

```

3.2.6 新属性

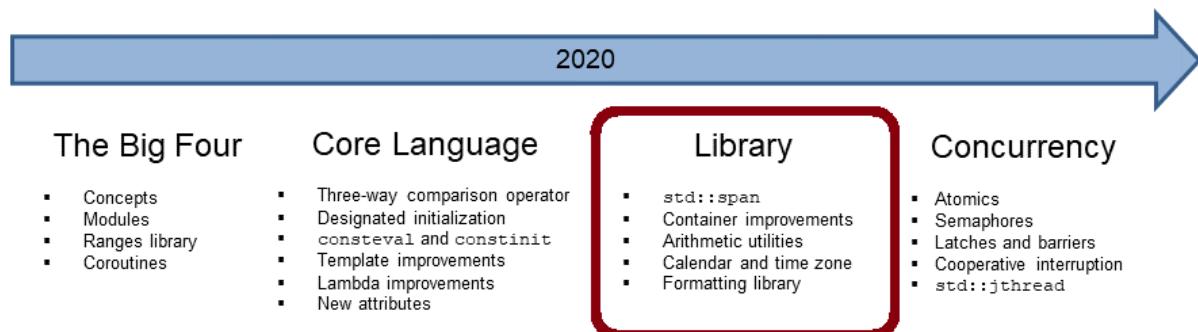
C++20 有了新的属性，包括 `[[likely]]` 和 `[[unlikely]]`。这两个属性都可以给优化器一些提示，指定哪个执行路径更可能或更不可能(可以更好的进行分支预测)。

属性 `[[likely]]`

```
1 for(size_t i=0; i < v.size(); ++i){  
2     if (v[i] < 0) [[likely]] sum -= sqrt(-v[i]);  
3     else sum += sqrt(v[i]);  
4 }
```

3.3. 标准库

C++20



3.3.1 `std::span`

`std::span` 表示引用一个连续的序列，有时也称为视图，并不是数据的所有者。视图可以是一个 C 风格数组，一个 `std::array`，一个有长度的指针，或者一个 `std::vector`。`std::span` 的实现需要指向其第一个元素的指针和长度。使用 `std::span` 的主要原因是，若将普通数组传递给函数，数组会衰退为指针，其长度信息会丢失。`std::span` 可以自动推导数组、`std::array` 或 `std::vector` 的长度。若使用指针初始化 `std::span`，则必须在构造函数中提供其长度。

`std::span` 作为函数参数

```
1 void copy_n(const int* src, int* des, int n) {}  
2  
3 void copy(std::span<const int> src, std::span<int> des) {}  
4  
5 int main(){  
6  
7     int arr1[] = {1, 2, 3};  
8     int arr2[] = {3, 4, 5};  
9  
10    copy_n(arr1, arr2, 3);
```

```
11     copy(arr1, arr2);
12
13 }
```

与函数 `copy_n` 相比，`copy` 不需要元素的数量。因此，`std::span<T>` 是导致错误的常见原因。

3.3.2 容器的改进

C++20 在标准模板库的容器方面有很多改进。首先，`std::vector` 和 `std::string` 具有 `constexpr` 构造函数，因此可以在编译时使用。所有标准库容器都支持擦除，关联容器支持 `contains` 成员函数。此外，`std::string` 允许检查前缀或后缀。

3.3.3 计算工具

有时，有符号整数和无符号整数的比较是导致意外行为和错误的原因，新的安全整数比较函数 `std::cmp_*` 可以避免这种问题。

安全的整数比较

```
1 int x = -3;
2 unsigned int y = 7;
3
4 if (x < y) std::cout << "expected";
5 else std::cout << "not expected"; // not expected
6
7 if (std::cmp_less(x, y)) std::cout << "expected"; // expected
8 else std::cout << "not expected";
```

此外，C++20 在命名空间 `std::numbers` 中包含了数学常量，包括 e 、 π 和 φ 。

新的位操作允许访问单个位和位序列，并重新解释它们。

访问单个位和位序列

```
1 std::uint8_t num= 0b10110010;
2
3 std::cout << std::has_single_bit(num) << '\n'; // false
4 std::cout << std::bit_width(unsigned(5)) << '\n'; // 3
5 std::cout << std::bitset<8>(std::rotl(num, 2)) << '\n'; // 11001010
6 std::cout << std::bitset<8>(std::rotr(num, 2)) << '\n'; // 10101100
```

3.3.4 日期和时区

C++11 中的 [chrono 库](#) 扩展了日期和时区功能。日期由表示年、月、周中的天和月中的第 n 个工作日组成。这些基本类型可以组合，形成复杂类型，例如：`year_month`、`year_month_day`、`year_month_day_last`、`year_month_weekday` 和 `year_month_weekday_last`。为了方便地指定时间点，操作符 “`/`” 可以重载。此外，可以用新字面量：`d` 表示天，`y` 表示年。

时间点可以显示在不同的时区。由于扩展了 `chrono` 库，下面的用例实现起来就很简单：

- 以特定格式表示日期
- 一个月的最后一天
- 获取两个日期之间的天数
- 输出不同时区的当前时间

下面的程序显示了不同时区的当前时间。

不同时区的当前时间

```

1 using namespace std::chrono;
2
3 auto time = floor<milliseconds>(system_clock::now());
4 auto localTime = zoned_time<milliseconds>(current_zone(), time);
5 auto berlinTime = zoned_time<milliseconds>("Europe/Berlin", time);
6 auto newYorkTime = zoned_time<milliseconds>("America/New_York", time);
7 auto tokyoTime = zoned_time<milliseconds>("Asia/Tokyo", time);
8
9 std::cout << time << '\n'; // 2020-05-23 19:07:20.290
10 std::cout << localTime << '\n'; // 2020-05-23 21:07:20.290 CEST
11 std::cout << berlinTime << '\n'; // 2020-05-23 21:07:20.290 CEST
12 std::cout << newYorkTime << '\n'; // 2020-05-23 15:07:20.290 EDT
13 std::cout << tokyoTime << '\n'; // 2020-05-24 04:07:20.290 JST

```

3.3.5 格式化库

新的格式化库提供了更安全且可扩展的替代方案，旨在补充现有的 I/O 流，并重用其一些基础结构，例如：用于定义类型的重载插入操作符。

```

1 std::string message = std::format("The answer is {}.", 42);

```

`std::format` 使用 Python 的语法进行格式化。下面的例子展示了一些用例：

- 格式和使用位置参数

```

1 std::string s = std::format("I'd rather be {1} than {0}.", "right", "happy");
2 // s == "I'd rather be happy than right."

```

- 以安全的方式将整数转换为字符串

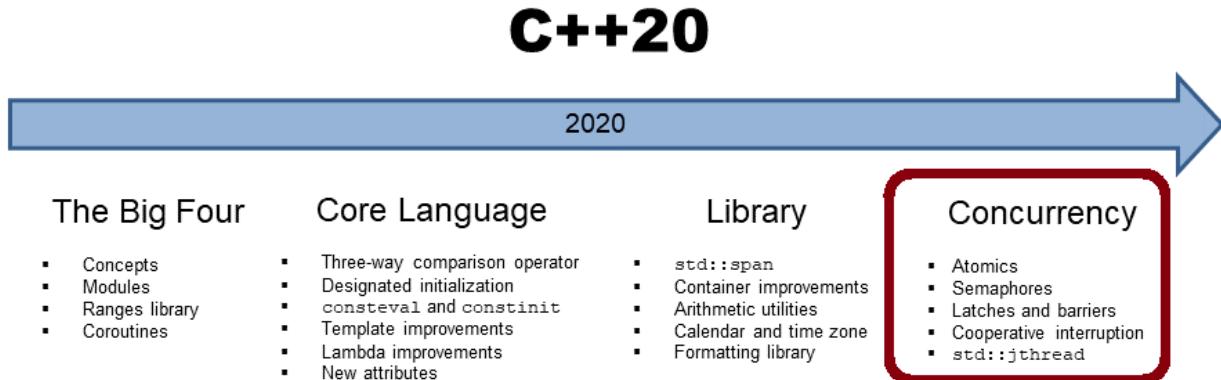
```

1 memory_buffer buf;
2 std::format_to(buf, "{}", 42); // replaces itoa(42, buffer, 10)
3 std::format_to(buf, "{:x}", 42); // replaces itoa(42, buffer, 16)

```

- 格式化用户定义的类型

3.4. 并发性



3.4.1 原子变量和操作

类模板 `std::atomic_ref` 对引用的非原子对象应用原子操作，可以对其进行并发的读写，不会出现数据竞争。引用对象的生存期必须超过 `std::atomic_ref` 的生命周期。使用 `std::atomic_ref` 访问引用对象的子对象时，不是线程安全的。

根据`std::atomic`, `std::atomic_ref` 可以特化，并支持内置数据类型的特化。

```
1 struct Counter {
2     int a;
3     int b;
4 };
5
6 Counter counter;
7
8 std::atomic_ref<Counter> cnt(counter);
```

C++20 中，有两个原子智能指针 (`std::atomic` 的偏特化)，分别是 `std::atomic<std::shared_ptr<T>` 和 `std::atomic<std::weak_ptr<T>`。这两个原子智能指针不仅保证控制块 (如`std::shared_ptr`) 的线程安全，而且还保证关联对象的线程安全。

`std::atomic` 有更多的扩展，C++20 为原子浮点类型提供了特化。当需要并发递增的浮点类型时，就可以直接使用了。

类型`std::atomic_flag`是一种原子布尔值，有一个清除和设置状态。简单起见，这里将 clear 状态称为 false，set 状态为 true。`clear()` 成员函数将其值设置为 false。使用 `test_and_set()` 成员函数，可以将值设置为 true，并获得之前的值，并且没有成员函数可以获取当前值。这将在 C++20 中改变，因为 `std::atomic_flag` 添加了一个 `test()` 方法。

此外，`std::atomic_flag` 可以通过成员函数 `notify_one()`、`notify_all()` 和 `wait()` 用于线程同步。C++20 中，通知和等待在 `std::atomic` 和 `std::atomic_ref` 的所有偏特化和全特化上都可用。特化可用于布尔、整型、浮点数和指针。

3.4.2 信号量

信号量是一种同步机制，用于控制对共享资源的并发访问。计数信号量，例如 C++20 中添加的计数信号量，是一种特殊的信号量，其初始计数器大于零。计数器在构造函数中初始化。获取信号量会减少计数器，释放信号量则会增加计数器。若线程试图在计数器为零时获取信号量，线程将阻塞，直到另一个线程通过释放信号量来增加计数器。

3.4.3 门闩和栅栏

门闩和栅栏是线程的同步机制，可以使一些线程阻塞，直到计数器为零。这两种线程同步的机制有什么不同？`std::latch` 只能使用一次，但 `std::barrier` 可以使用多次。`std::latch` 在多线程管理一个任务时很有用，`barrier` 对于管理多线程的重复任务。此外，`std::barrier` 可以在每次迭代中调整计数器。

以下是基于提案[N4204](#)的代码片段。我修正了一些错别字，重新进行了格式化。

使用 `std::latch` 进行线程同步

```
1 void DoWork(threadpool* pool) {
2
3     std::latch completion_latch(NTASKS);
4     for (int i = 0; i < NTASKS; ++i) {
5         pool->add_task([&] {
6             // perform work
7             ...
8             completion_latch.count_down();
9         });
10    }
11    // Block until work is done
12    completion_latch.wait();
13 }
```

`std::latch` 的 `completion_latch` 的计数器设置为 `NTASKS`(第 3 行)。线程池执行 `NTASKS` 作业(第 4-10 行)。任务结束时，计数器递减(第 8 行)。运行 `DoWork` 函数的线程在第 12 行阻塞，直到所有任务都完成。

3.4.4 中断协程

`std::stop_token` 可以中断 `std::jthread`。

中断 `std::jthread`

```
1 int main() {
2
3     std::cout << '\n';
4
5     std::jthread nonInterruptible([]{
6         int counter{0};
```

```

7   while (counter < 10) {
8     std::this_thread::sleep_for(0.2s);
9     std::cerr << "nonInterruptible: " << counter << '\n';
10    ++counter;
11  }
12 });
13
14 std::jthread interruptible([](std::stop_token stoken) {
15   int counter{0};
16   while (counter < 10) {
17     std::this_thread::sleep_for(0.2s);
18     if (stoken.stop_requested()) return;
19     std::cerr << "interruptible: " << counter << '\n';
20     ++counter;
21   }
22 });
23
24 std::this_thread::sleep_for(1s);
25
26 std::cerr << '\n';
27 std::cerr << "Main thread interrupts both jthreads" << std::endl;
28 nonInterruptible.request_stop();
29 interruptible.request_stop();
30
31 std::cout << '\n';
32
33 }

```

主程序启动两个线程，`nonInterruptible` 和 `interruptible`(第 5 行和第 14 行)。只有线程可中断得到 `std::stop_token`，在第 18 行中使用它来检查是否中断。在中断的情况下，Lambda 立即返回。调用 `interruptible.request_stop()` 会触发线程的取消，并且 `nonInterruptible.request_stop()` 没有任何效果。

```
C:\Users\seminar>interruptJthread.exe

nonInterruptible: 0
interruptible: 0
nonInterruptible: 1
interruptible: 1
nonInterruptible: 2
interruptible: 2
nonInterruptible: 3
interruptible: 3

Main thread interrupts both jthreads

nonInterruptible: 4
nonInterruptible: 5
nonInterruptible: 6
nonInterruptible: 7
nonInterruptible: 8
nonInterruptible: 9

C:\Users\seminar>
```

中断协程

3.4.5 std::jthread

std::jthread 通过自动加入已启动的线程扩展 std::thread，std::jthread 也可以中断。

因为 std::thread 的非直观行为，所以将 std::jthread 添加到 C++20 标准中。若 std::thread 仍可汇入，std::terminate 将在其析构函数中进行汇入。若既没有调用 thr.join()，也没有调用 thr.detach()，那么线程 thr 是可汇入的。

```
1 int main() {
2
3     std::cout << '\n';
4
5     std::cout << std::boolalpha;
6     std::thread thr[]{ std::cout << "Joinable std::thread" << '\n'; };
7     std::cout << "thr.joinable(): " << thr.joinable() << '\n';
8
9     std::cout << '\n';
10 }
```

```
File Edit View Bookmarks Settings Help
rainer@linux:~> threadJoinable
thr.joinable(): true
terminate called without an active exception
Aborted (core dumped)
rainer@linux:~> threadJoinable
thr.joinable(): true
terminate called without an active exception
Joinable std::thread
Aborted (core dumped)
rainer@linux:~> █
> rainer : bash
```

对可汇入线程使用 std::terminate

程序的两次执行都将终止。在第二次运行中，线程 thr 有足够的时间显示它的消息：“Joinable std::thread”。

修改后的示例中，我使用了 C++20 标准的 std::jthread。

```
1 int main() {
2     std::cout << '\n';
3
4     std::cout << std::boolalpha;
5     std::jthread thr[] { std::cout << "Joinable std::jthread" << '\n'; };
6     std::cout << "thr.joinable(): " << thr.joinable() << '\n';
7
8     std::cout << '\n';
9 }
```

现在，线程 thr 可在析构函数中进行汇入。

```
File Edit View Bookmarks Settings Help
rainer@linux:~> jthreadJoinable
thr.joinable(): true
Joinable std::jthread
rainer@linux:~> █
> rainer : bash
```

线程 thr 自动汇入

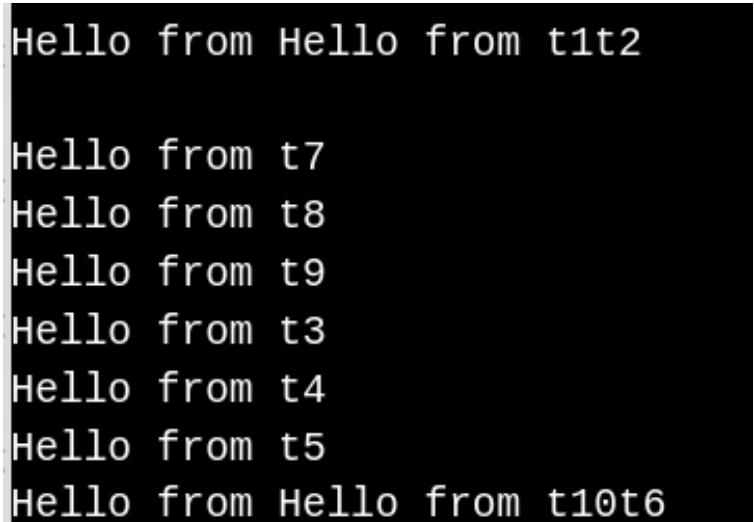
3.4.6 同步输出流

C++20 添加了同步的输出流。当更多线程并发地写入 std::cout，但没有同步时，会发生什么？

未同步写入 std::cout

```
1 void sayHello(std::string name) {
2     std::cout << "Hello from " << name << '\n';
3 }
4
5 int main() {
6     std::cout << "\n";
7
8     std::jthread t1(sayHello, "t1");
9     std::jthread t2(sayHello, "t2");
10    std::jthread t3(sayHello, "t3");
11    std::jthread t4(sayHello, "t4");
12    std::jthread t5(sayHello, "t5");
13    std::jthread t6(sayHello, "t6");
14    std::jthread t7(sayHello, "t7");
15    std::jthread t8(sayHello, "t8");
16    std::jthread t9(sayHello, "t9");
17    std::jthread t10(sayHello, "t10");
18
19    std::cout << '\n';
20 }
```

输出可能会弄得一团糟。



The terminal window displays the output of 10 threads. The threads are labeled t1 through t10. The output is interleaved and lacks synchronization, resulting in a chaotic sequence of "Hello from" messages.

```
Hello from Hello from t1t2

Hello from t7
Hello from t8
Hello from t9
Hello from t3
Hello from t4
Hello from t5
Hello from Hello from t10t6
```

未同步写入 std::cout

从函数 sayHello 中的 std::cout，切换到 std::osyncstream(std::cout) 可消除混乱。

同步写入 std::cout

```
1 void sayHello(std::string name) {
2     std::osyncstream(std::cout) << "Hello from " << name << '\n';
3 }
```

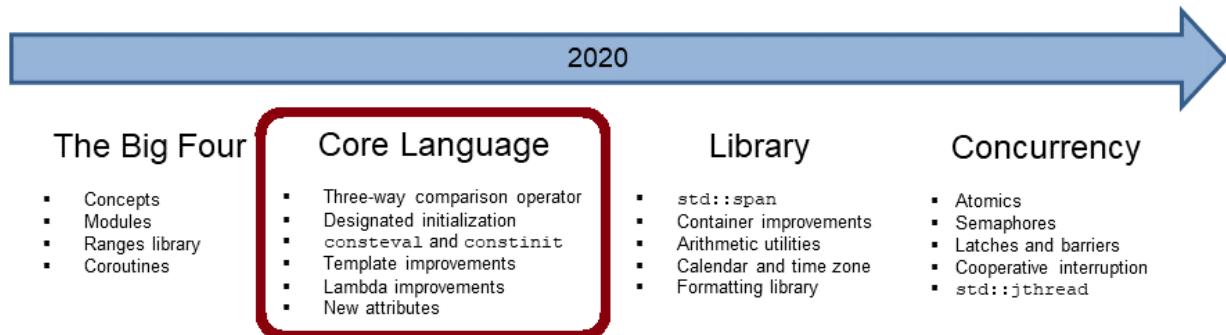
```
Hello from t1  
Hello from t2  
Hello from t3  
Hello from t4  
Hello from t5  
Hello from t6  
Hello from t7  
Hello from t8  
Hello from t9  
Hello from t10
```

同步写入 std::cout

第三部分：特性详情

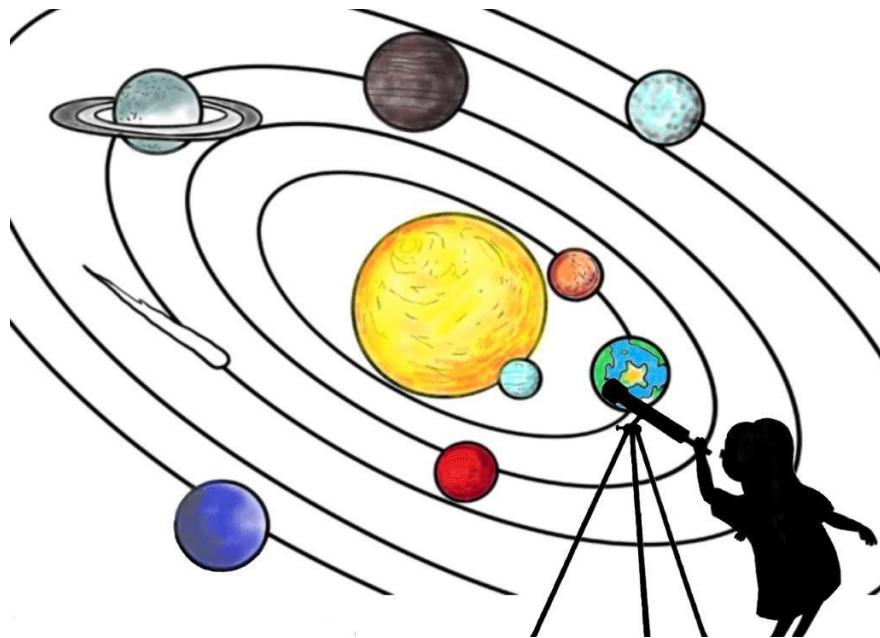
第4章 语言核心

C++20



概念是 C++20 的重要特性之一，所以这里将其作为展示 C++20 核心语言特性的起点。

4.1. 概念



为了充分理解概念，需要先了解概念出现的原因。

4.1.1 两种错误的方法

C++20 之前，可以有两种截然相反的方式来思考函数或类：为特定类型或泛型类型定义。后者，称之为函数模板或类模板。不过，这两种方法都有各自的问题：

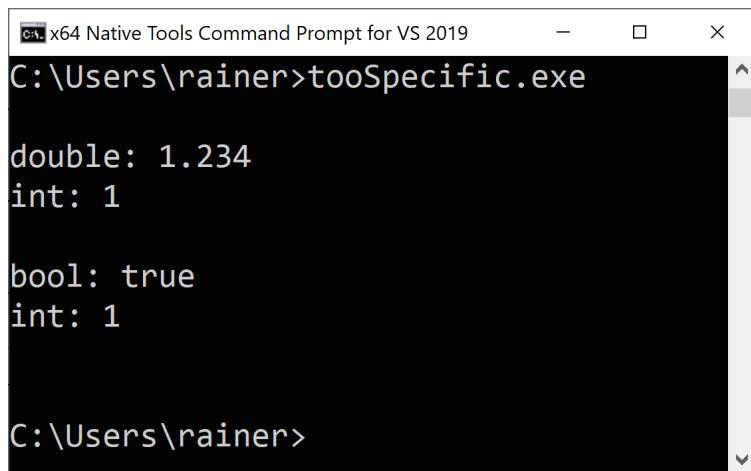
4.1.1.1 太具体

为每种类型的重载函数，或是重新进行类实现是一项乏味的工作。为了避免这种重复，通常会使用类型转换。不过，这种方式看似拯救，却又是诅咒。

类型的隐式转换

```
1 // tooSpecific.cpp
2
3 #include <iostream>
4
5 void needInt(int i){
6     std::cout << "int: " << i << '\n';
7 }
8
9 int main(){
10
11     std::cout << std::boolalpha << '\n';
12
13     double d{1.234};
14     std::cout << "double: " << d << '\n';
15     needInt(d);
16
17     std::cout << '\n';
18
19     bool b{true};
20     std::cout << "bool: " << b << '\n';
21     needInt(b);
22
23     std::cout << '\n';
24
25 }
```

例子(第13行)以double开始,以int结束(第15行)。第二次,以bool开始(第19行),并以int结束(第21行)。



The screenshot shows a terminal window titled 'x64 Native Tools Command Prompt for VS 2019'. The command 'C:\Users\rainer>tooSpecific.exe' is entered. The output consists of two pairs of lines, each starting with a type name ('double', 'bool') followed by its value ('1.234', 'true') and then an 'int' value ('1'). The terminal prompt 'C:\Users\rainer>' is visible at the bottom.

```
double: 1.234
int: 1

bool: true
int: 1
```

例子进行了两次类型隐式转换。

4.1.1.1.1 窄化转换

使用 double 类型调用 getInt(int a) 可以实现窄化转换。窄化转换是一种类型转换，包括精度损失。但有时，开发者并不想有任何损失。

4.1.1.1.2 类型提升

反过来会更好吗？

使用 bool 类型调用 getInt(int a)，可将 bool 类型提升为 int 类型。惊讶吗？许多 C++ 开发人员在添加两个 bool 时，并不知道他们会得到的是哪种数据类型。

对两个布尔值进行加法运算

```
1 template <typename T>
2 auto add(T first, T second) {
3     return first + second;
4 }
5
6 int main() {
7     add(true, false);
8 }
```

C++ Insights 在编译器在实例化中转换函数模板后，并将上面的源代码可视化。

```
1 template <typename T>
2 auto add(T first, T second) {
3     return first + second;
4 }
5
6 int main() {
7     add(true, false);
8 }
```

```

1 template<typename T>
2 auto add(T first, T second)
3 {
4     return first + second;
5 }
6
7
8 #ifdef INSIGHTS_USE_TEMPLATE
9 template<>
10 int add<bool>(bool first, bool second)
11 {
12     return static_cast<int>(first) + static_cast<int>(second);
13 }
14 #endif
15
16
17 int main()
18 {
19     add(true, false);
20     return 0;
21 }
22
23

```

bool 提升为 int

第 8-14 行是C++ Insights 截图中的关键行。函数模板 add 的模板实例化，使用返回类型 int 创建一个全特化函数，从而两个 bool 类型的参数都隐式提升为 int。

因为我们不想为每种类型重载函数或重新实现类，来尝试依赖于转换的魔力。

让我试试另一种方式，用一个泛型函数。

4.1.1.2 太通用

容器排序是一个很普遍的需求。若容器的元素支持排序，那么就应该适用于每个容器。下面的示例中，我将标准算法 std::sort 应用于标准容器 std::list。

对 std::list 排序

```

1 // tooGeneric.cpp
2
3 #include <algorithm>
4 #include <list>
5
6 int main() {
7

```

```

8 std::list<int> myList{1, 10, 3, 2, 5};
9
10 std::sort(myList.begin(), myList.end());
11
12 }

```

The screenshot shows a terminal window with a yellow background. At the top, there's a menu bar with File, Edit, View, Bookmarks, Settings, Help. Below the menu, the terminal prompt is "rainer@linux:~>". The main area of the terminal is filled with a long error message from the g++ compiler. The message is a stack trace of template argument deduction errors, primarily related to the std::sort function trying to deduce types for its parameters. It includes many lines starting with "/usr/include/c++/4.8/bits/stl_algobase.h:5461:22: note: candidate is:" and "In file included from /usr/include/c++/4.8/bits/stl_algobase.h:67:0, from sortlist.cpp:3". The error message is cut off at the bottom.

对 std::list 进行排序时，编译器报错

我不想解析这么长的信息，但还是要看出什么问题了。看一下本例中使用的std::sort特定重载的签名。

```

1 template< class RandomIt >
2 constexpr void sort( RandomIt first, RandomIt last );

```

std::sort 使用了奇怪的参数类型 RandomIt。RandomIt 代表随机访问迭代器，也就是编译错误提供的关键信息。std::list 只提供双向迭代器，但 std::sort 需要一个随机访问迭代器。下图显示了为什么 std::list 不支持随机访问迭代器。



去看一下 cppreference.com 上的 std::sort 文档，会发现一些模板参数对类型有要求。它们对类型提出了概念性的要求，这些要求形成了 C++20 的概念。

4.1.1.3 概念

概念对模板参数施加语义约束，`std::sort` 具有接受比较器的重载。

```
1 template< class RandomIt, class Compare >
2 constexpr void sort(RandomIt first, RandomIt last, Compare comp);
```

下面是 `std::sort` 重载的类型要求：

- `RandomIt` 必须满足 `ValueSwappable` 和 `LegacyRandomAccessIterator` 的要求
- 解引用 `RandomIt` 的类型必须满足 `MoveAssignable` 和 `MoveConstructible` 的要求。
- 可解引用的 `RandomIt` 的类型必须满足 `Compare` 的要求。

`ValueSwappable` 或 `LegacyRandomAccessIterator` 之类的要求就是类型需求，其中一些需求在 C++20 的概念中形式化了。

并且，`std::sort` 还需要一个 `LegacyRandomAccessIterator`，在 C++20 中称为 `random_access_iterator`(是 `<iterator>` 的一部分)：

`std::random_access_iterator`

```
1 template<class I>
2 concept random_access_iterator =
3     bidirectional_iterator<I> &&
4     derived_from<ITER_CONCEPT(I), random_access_iterator_tag> &&
5     totally_ordered<I> &&
6     sized_sentinel_for<I, I> &&
7     requires(I i, const I j, const iter_difference_t<I> n) {
8         { i += n } -> same_as<I&>;
9         { j + n } -> same_as<I>;
10        { n + j } -> same_as<I>;
11        { i -= n } -> same_as<I&>;
12        { j - n } -> same_as<I>;
13        { j[n] } -> same_as<iter_reference_t<I>>;
14    };
```

类型 `I` 支持概念 `random_access_iterator`，若支持概念 `bidirectional_iterator` 和以下所有需求。例如，`{i += n} -> same_as<I&>` 作为需求表达式的一部分，意味着对于类型为 `i` 的值，`{i += n}` 是一个有效的表达式，返回类型为 `I&`。对于链表，`std::list` 支持的是 `bidirectional_iterator`，而非 `std::sort` 要求的 `random_access_iterator`。

现在需要 `random_access_iterator` 的算法，接收到 `bidirectional_iterator` 时，就会显示一条简洁易懂的错误消息，明确的表示迭代器不满足 `random_access_iterator` 的要求。



标准模板库

泛型编程的本质

我想引用 Alexander Stepanov(标准模板库的创建者) 和 Daniel Rose(信息检索研究员) 写的书《From Mathematics to Generic Programming》中的一段话来开始这段简短的回顾: “泛型编程的本质在于概念的思想, 概念是描述一系列相关对象类型的一种方式。” 这些相关的对象类型可以是整型, 如 `bool`、`char` 或 `int`。概念是对相关类型的一组需求, 例如: 所支持的操作、语义, 以及时间和空间的复杂度。

标准模板库 (STL) 作为一个基于概念的通用库, 由三个部分组成: 容器, 运行在容器上的算法, 以及连接它们的迭代器。

每个容器都提供了符合其结构的迭代器, 算法可以对这些迭代器进行操作。容器 (如序列容器或关联容器) 基于半开放范围模型。迭代器提供对容器元素的访问, 并且可以遍历容器, 从而可对其进行比较。STL 的抽象基于半开放范围模型和迭代器等概念, 从而可以使用 STL 的容器和算法。

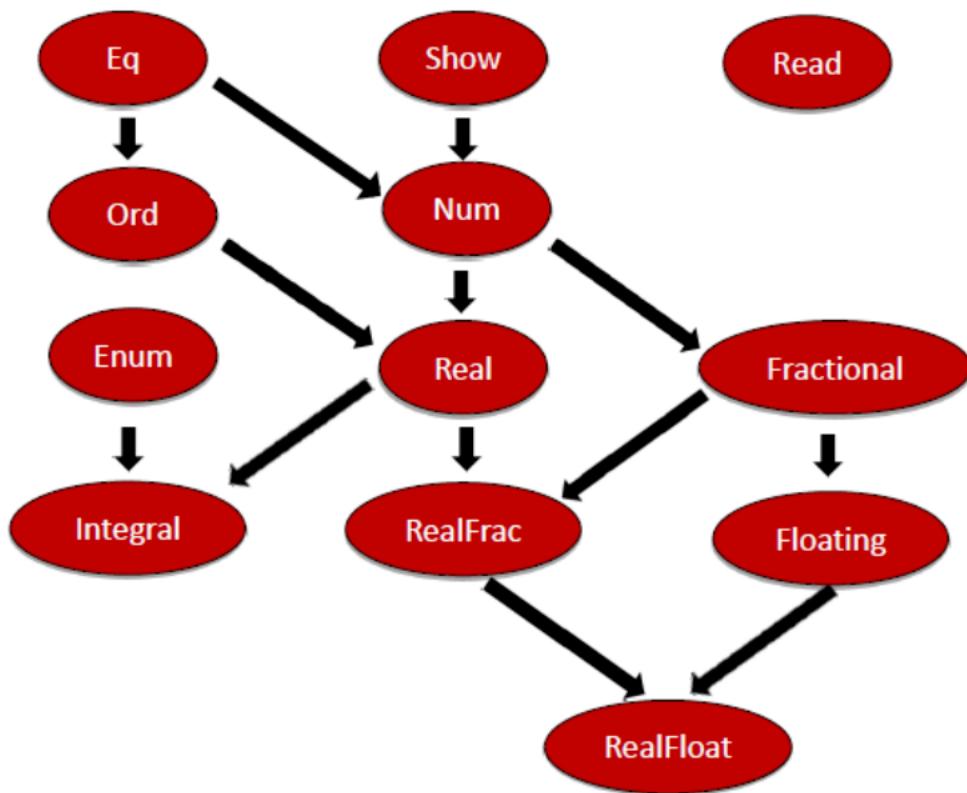
那么, 概念的优势是什么?

4.1.2 概念的优势

- 模板参数的需求描述是接口声明的一部分。
- 函数的重载和类模板的特化可以基于概念。
- 概念可用于函数模板、类模板和类或类模板的泛型成员函数。
- 因为编译器会将模板形参的要求与给定的模板实参进行比较, 所以可得到明确的错误消息。
- 可以使用预定义的概念, 也可以自定义概念。
- `auto` 和概念的用法统一, 可以用概念来代替 `auto`。
- 若函数声明使用了概念, 则它自动成为函数模板。因此, 编写函数模板就会如同编写函数一样简单。

4.1.3 漫长的历史

我第一次听说概念是在 2005 - 2006 年左右, 这让我想起 Haskell 的课程, Haskell 类型就具有类似的接口。下面是Haskell 的类型类层次结构的一部分。



Haskell 类型的层次结构

但 C++ 的概念不同。以下是我观察到的结果。

- Haskell 中，类型都必须是实例。C++20 中，类型必须满足概念的需求。
- C++ 中，概念可以用在模板的非类型参数上。例如，像值 5 这样的数字就是非类型参数。例如，当想要一个包含 5 个元素的整型数组 std::array 时，可以使用非类型参数 5:std::array<int, 5> myArray。
- 概念不会增加运行时的成本。

最初，概念是 C++11 的关键特性，但在 2009 年 7 月法兰克福的标准化会议上删除了。这里引用 Bjarne Stroustrup 对当时概念的看法：“[C++0x 的概念变成一个复杂的怪物](#)”。几年后，第二次尝试也没有成功：精简版概念从 C++17 标准中删除了。最终，成了 C++20 的一部分。

4.1.4 概念的使用

可以用四种方式来概念。

4.1.4.1 四种方式

我在 conceptsintegralvariables.cpp 中演示了这四种方式，使用了预定义的概念 std::integral。

使用 std::integral 概念的四种方式

```

1 // conceptsIntegralVariations.cpp
2

```

```

3 #include <concepts>
4 #include <iostream>
5
6 template<typename T>
7 requires std::integral<T>
8 auto gcd(T a, T b) {
9     if( b == 0 ) return a;
10    else return gcd(b, a % b);
11 }
12
13 template<typename T>
14 auto gcd1(T a, T b) requires std::integral<T> {
15     if( b == 0 ) return a;
16     else return gcd1(b, a % b);
17 }
18
19 template<std::integral T>
20 auto gcd2(T a, T b) {
21     if( b == 0 ) return a;
22     else return gcd2(b, a % b);
23 }
24
25 auto gcd3(std::integral auto a, std::integral auto b) {
26     if( b == 0 ) return a;
27     else return gcd3(b, a % b);
28 }
29
30 int main() {
31
32     std::cout << '\n';
33
34     std::cout << "gcd(100, 10)= " << gcd(100, 10) << '\n';
35     std::cout << "gcd1(100, 10)= " << gcd1(100, 10) << '\n';
36     std::cout << "gcd2(100, 10)= " << gcd2(100, 10) << '\n';
37     std::cout << "gcd3(100, 10)= " << gcd3(100, 10) << '\n';
38
39     std::cout << '\n';
40 }

```

因为第 3 行使用了 `<concepts>`, 所以可以在这里使用 `std::integral` 概念。若 `T` 是类型 `integral`, 则匹配了这个概念。`gcd` 函数表示基于 `Euclidean` 的最大公约数算法。

下面是使用概念的四种方式:

- `requires` 子句 (第 6 行)
- 尾部 `requires` 子句 (第 14 行)
- 约束模板参数 (第 19 行)
- 简化的函数模板 (第 25 行)

简单起见，每个函数模板只返回 auto。函数模板 gcd、gcd1、gcd2 和函数 gcd3 之间存在语义差异。对于 gcd、gcd1 或 gcd2，参数 a 和 b 必须具有相同的类型，而函数 gcd3 不同。参数 a 和 b 可以有不同的类型，但必须都满足 std::integral 概念。

```
gcd(100, 10) = 10
gcd1(100, 10) = 10
gcd2(100, 10) = 10
gcd3(100, 10) = 10
```

gcd 和 gcd1 使用的函数 requires 子句，其实很强大。

4.1.4.2 requires 子句

conceptsIntegralVariations.cpp 演示了如何使用概念来定义函数或函数模板。为了完整起见，我想补充一点：可以使用概念指定函数或函数模板的返回类型。

关键字 requires 引入了一个 requires 子句，指定了模板参数 (gcd) 或函数声明 (gcd1) 上的约束。requires 后面必须跟一个编译时谓词，例如概念 (gcd)、概念的连接/析取，或是 requires 表达式。

编译时谓词也可以是表达式：

requires 子句中使用编译时谓词

```
1 // requiresClause.cpp
2
3 #include <iostream>
4
5 template <unsigned int i>
6 requires (i <= 20)
7 int sum(int j) {
8     return i + j;
9 }
10
11
12 int main() {
13
14     std::cout << '\n';
15
16     std::cout << "sum<20>(2000): " << sum<20>(2000) << '\n',
17     // std::cout << "sum<23>(2000): " << sum<23>(2000) << '\n', // ERROR
18
19     std::cout << '\n';
20
21 }
```

第 6 行中使用的编译时谓词：需要用于非类型的 i，而不非普通的类型。

```
sum<20>(2000): 2020
```

当打开第 17 行的注释时，clang 编译器报告以下错误：

```
<source>:17:39: error: no matching function for call to 'sum'  
    std::cout << "sum<23>(2000): " << sum<23>(2000) << '\n', // ERROR  
                                         ^~~~~~  
<source>:7:5: note: candidate template ignored: constraints not satisfied [with i = 23]  
int sum(int j) {  
    ^  
<source>:6:11: note: because '23U <= 20' (23 <= 20) evaluated to false  
requires (i <= 20)  
    ^
```

requires 子句中使用编译时谓词失败

避免在 requires 子句中使用编译时谓词

当使用概念约束模板参数或函数模板时，应该使用命名概念或组合它们。概念属于语义范畴，而不是像 $i \leq 20$ 这样的语法约束。并且，为概念命名可以使其重用。

4.1.4.3 函数的返回类型为概念

下面是使用概念作为返回类型的函数模板 gcd 和函数 gcd1 的定义。

概念作为返回类型

```
1 template<typename T>  
2 requires std::integral<T>  
3 std::integral auto gcd(T a, T b) {  
4     if( b == 0 ) return a;  
5     else return gcd(b, a % b);  
6 }  
7  
8 std::integral auto gcd1(std::integral auto a, std::integral auto b) {  
9     if( b == 0 ) return a;  
10    else return gcd1(b, a % b);  
11 }
```

4.1.4.4 概念的用例

首先，概念是编译时谓词。编译时谓词是在编译时执行，并返回布尔值的函数。深入研究概念的各种用例前，我想先揭开概念的神秘面纱，并将它们简单地表示为在编译时返回布尔值的函数。

4.1.4.4.1 编译时谓词

概念可用于在运行时或编译时执行的控制结构中。

概念作为编译时谓词

```
1 // compileTimePredicate.cpp
2
3 #include <compare>
4 #include <iostream>
5 #include <string>
6 #include <vector>
7
8 struct Test{};
9
10 int main() {
11
12     std::cout << '\n';
13
14     std::cout << std::boolalpha;
15
16     std::cout << "std::three_way_comparable<int>: ";
17     << std::three_way_comparable<int> << "\n";
18
19     std::cout << "std::three_way_comparable<double>: ";
20     if (std::three_way_comparable<double>) std::cout << "True";
21     else std::cout << "False";
22
23     std::cout << "\n\n";
24
25     static_assert(std::three_way_comparable<std::string>);
26
27     std::cout << "std::three_way_comparable<Test>: ";
28     if constexpr(std::three_way_comparable<Test>) std::cout << "True";
29     else std::cout << "False";
30
31     std::cout << '\n';
32
33     std::cout << "std::three_way_comparable<std::vector<int>>: ";
34     if constexpr(std::three_way_comparable<std::vector<int>>) std::cout << "True";
35     else std::cout << "False";
36
37     std::cout << '\n';
38
39 }
```

例子中，我使用了 `std::three_way_comparable<T>` 这个概念，在比较时检查 `T` 是否支持六个比较运算符。作为编译时谓词 `std::three_way_comparable` 可以在运行时 (第 16 行和第 20 行) 或编译时使用。`static_assert`(第 25 行) 和 `constexpr if`(第 28 和 34 行) 可在编译时进行计算。

```
std::three_way_comparable<int>: True
std::three_way_comparable<double>: True

std::three_way_comparable<Test>: False
std::three_way_comparable<std::vector<int>>: True
```

对编译时谓词的概念进行简短的介绍之后，继续本节的概念的各种用例。这些概念的应用不是很详细，这里主要使用的是预定义概念，我将后续的章节中讨论这些概念。

4.1.4.4.2 类模板

类模板 MyVector 要求模板形参 T 是常规类型，所以 T 的行为类似于 int。[std::regular](#) 的定义在之前有提到。

类定义中使用概念

```
1 // conceptsClassTemplate.cpp
2
3 #include <concepts>
4 #include <iostream>
5
6 template <std::regular T>
7 class MyVector{};
8
9 int main() {
10
11     MyVector<int> myVec1;
12     MyVector<int&> myVec2; // ERROR because a reference is not regular
13
14 }
```

因为引用非常规类型，所以第 12 行会出现编译错误。下面是 GCC 编译器错误消息的重要部分：

```
<source>:13:18: error: template constraint failure for 'template<class T> requires regular<T> class MyVector'
13 |     MyVector<int&> myVec2;
```

引用非常规类型

4.1.4.4.3 泛型成员函数

这个例子中，向 MyVector 类添加了一个通用的 push_back 成员函数。push_back 要求参数可复制。

泛型成员函数中使用概念

```
1 // conceptMemberFunction.cpp
```

```

2
3 #include <concepts>
4 #include <iostream>
5
6 struct NotCopyable {
7     NotCopyable() = default;
8     NotCopyable(const NotCopyable&) = delete;
9 };
10
11 template <typename T>
12 struct MyVector{
13     void push_back(const T&) requires std::copyable<T> {};
14 };
15
16 int main() {
17
18     MyVector<int> myVec1;
19     myVec1.push_back(2020);
20
21     MyVector<NotCopyable> myVec2;
22     myVec2.push_back(NotCopyable()); // ERROR because not copyable
23
24 }

```

编译在第 22 行失败。因为复制构造函数声明为已删除，所以 NotCopyable 的实例不可复制。

4.1.4.4 可变模板

可变模板中可以使用概念。

```

1 // allAnyNone.cpp
2
3 #include <concepts>
4 #include <iostream>
5
6 template<std::integral... Args>
7 bool all(Args... args) { return (... && args); }
8
9 template<std::integral... Args>
10 bool any(Args... args) { return (... || args); }
11
12 template<std::integral... Args>
13 bool none(Args... args) { return not(... || args); }
14
15 int main(){
16
17     std::cout << std::boolalpha << '\n';
18
19     std::cout << "all(5, true, false): " << all(5, true, false) << '\n';
20
21     std::cout << "any(5, true, false): " << any(5, true, false) << '\n';

```

```
22
23     std::cout << "none(5, true, false): " << none(5, true, false) << '\n';
24
25 }
```

上述函数模板的定义基于折叠表达式。C++11 支持可变参数模板，可以接受任意数量的模板参数，任意数量的模板参数由一个所谓的参数包保存。C++17 中可以使用二进制操作符直接简化参数包，这种简化称为[折叠表达式](#)。

在本例中，逻辑和 `&&`(第 7 行)、逻辑或 `||(第 10 行)` 以及逻辑或的否定(第 13 行)作为二元运算符。此外，`all`、`any` 和 `none` 要求类型参数必须支持 `std::integral` 概念。

```
all(5, true, false): false
any(5, true, false): true
none(5, ture, false): false
```

4.1.4.4.5 重载

`std::advance`是标准模板库的算法，对给定的迭代器 `iter` 增加 `n` 个元素，根据给定迭代器的功能，可以使用不同的策略。例如，`std::forward_list` 支持只能朝一个方向前进的迭代器，而 `std::list` 支持双向迭代器，`std::vector` 支持随机访问迭代器。因此，对于 `std::forward_list` 或 `std::list` 提供的迭代器，对 `std::advance(iter, n)` 的调用必须加 `n` 倍(参见 `std::list` 的结构)耗时。这个时间复杂度不适用于 `std::vector` 提供的 `std::randomaccess_iterator`，数字 `n` 可以直接加到迭代器中。因此，线性时间复杂度 $O(n)$ 变成了 $O(1)$ 。为了区分迭代器类型，可以使用概念。`conceptsOverloadingFunctionTemplates.cpp` 展示了这样使用概念的方式。

```
1 // conceptsOverloadingFunctionTemplates.cpp
2
3 #include <concepts>
4 #include <iostream>
5 #include <forward_list>
6 #include <list>
7 #include <vector>
8
9 template<std::forward_iterator I>
10 void advance(I& iter, int n){
11     std::cout << "forward_iterator" << '\n';
12 }
13
14 template<std::bidirectional_iterator I>
15 void advance(I& iter, int n){
16     std::cout << "bidirectional_iterator" << '\n';
17 }
18
19 template<std::random_access_iterator I>
20 void advance(I& iter, int n){
```

```

21     std::cout << "random_access_iterator" << '\n';
22 }
23
24 int main() {
25
26     std::cout << '\n';
27
28     std::forward_list forwList{1, 2, 3};
29     std::forward_list<int>::iterator itFor = forwList.begin();
30     advance(itFor, 2);
31
32     std::list li{1, 2, 3};
33     std::list<int>::iterator itBi = li.begin();
34     advance(itBi, 2);
35
36     std::vector vec{1, 2, 3};
37     std::vector<int>::iterator itRa = vec.begin();
38     advance(itRa, 2);
39
40     std::cout << '\n';
41 }
```

函数 `advance` 的三种重载位于 `std::forward_iterator`(第 9 行)、`std::bidirectional_iterator`(第 14 行) 和 `std::random_access_iterator`(第 19 行) 这三个概念上，编译器会选择最合适的选择。所以，对于 `std::forward_list`(第 28 行)，对应的是基于 `std::forward_list` 概念的重载；对于 `std::list`(第 32 行)，对应的是基于 `std::bidirectional_iterator` 概念的重载；对于 `std::vector`(第 36 行)，对应的是基于 `std::random_access_iterator` 概念的重载。

forward_iterator
bidirectional_iterator
random_access_iterator

`std::random_access_iterator` 可以当作 `std::bidirectional_iterator` 看待，`std::bidirectional_iterator` 可以当作 `std::forward_iterator` 看待。

4.1.4.4.6 特化的模板

还可以使用概念特化的模板。

```

1 // conceptsSpecialization.cpp
2
3 #include <concepts>
4 #include <iostream>
5
6 template <typename T>
7 struct Vector {
8     Vector() {
```

```

9     std::cout << "Vector<T>" << '\n';
10    }
11 };
12
13 template <std::regular Reg>
14 struct Vector<Reg> {
15     Vector() {
16         std::cout << "Vector<std::regular>" << '\n';
17     }
18 };
19
20 int main() {
21
22     std::cout << '\n';
23
24     Vector<int> myVec1;
25     Vector<int&> myVec2;
26
27     std::cout << '\n';
28 }

```

实例化类模板时，编译器会选择最特化的一个。对于 `Vector<int> myVec`(第 24 行)，会选择 `std::regular`(第 13 行) 的偏特化模板。引用 `Vector<int&> myVec2`(第 25 行) 不是 `std::regular` 类型，因此选择主模板(第 6 行)。

```

Vector<std::regular>
Vector<T>

```

4.1.4.4.7 使用多个概念

目前，这些概念的使用都很简单，也可以同时使用多个概念。

```

1 template<typename Iter, typename Val>
2     requires std::input_iterator<Iter>
3         && std::equality_comparable<Value_type<Iter>, Val>
4 Iter find(Iter b, Iter e, Val v)

```

`find` 需要 `Iter` 迭代器与 `Val` 进行比较

- 迭代器必须是输入迭代器；
- 迭代器的值类型必须与 `Val` 相同。

对迭代器的限制也可以表示为受约束的模板形参。

```

1 template<std::input_iterator Iter, typename Val>
2     requires std::equality_comparable<Value_type<Iter>, Val>
3 Iter find(Iter b, Iter e, Val v)

```

4.1.5 约束和非约束占位符

首先，让我告诉你 C++14 中的一个不对称。

4.1.5.1 C++14 中的不对称

我经常在课堂上进行讨论。C++14 中，我们有泛型 Lambda，可以使用 auto。

泛型 Lambda 和函数模板的比较

```
1 // genericLambdaTemplate.cpp
2
3 #include <iostream>
4 #include <string>
5
6 auto addLambda = [] (auto fir, auto sec){ return fir + sec; };
7
8 template <typename T, typename T2>
9 auto addTemplate(T fir, T2 sec){ return fir + sec; }
10
11 int main(){
12
13     std::cout << std::boolalpha << '\n';
14
15     std::cout << addLambda(1, 5) << " " << addTemplate(1, 5) << '\n';
16     std::cout << addLambda(true, 5) << " " << addTemplate(true, 5) << '\n';
17     std::cout << addLambda(1, 5.5) << " " << addTemplate(1, 5.5) << '\n';
18
19     const std::string fir{"ge"};
20     const std::string sec{"neric"};
21     std::cout << addLambda(fir, sec) << " " << addTemplate(fir, sec) << '\n';
22
23     std::cout << '\n';
24
25 }
```

泛型 Lambda(第 6 行) 和函数模板 (第 8 行) 产生相同的结果。

```
File Edit View Bookmarks Settings Help
rainer@linux:~/genericLambdaTemplate
6 6
6 6
6.5 6.5
generic generic

rainer@linux:~/
```

泛型 Lambda 引入了一种定义函数模板的新方法。在课堂上，经常有人问：我们可以在函数中使用 auto 获取函数模板吗？C++14 不行，但 C++20 可以。

C++20 中，可以在函数声明中使用无约束占位符 (auto) 或有约束占位符 (概念) 来自动获取函数模板。规则非常简单，在每个可以使用无约束占位符 auto 的地方，都可以使用一个概念。我将在简写函数模板一节中进行详细说明。

4.1.5.2 占位符

使用约束占位符

```
1 // placeholders.cpp
2
3 #include <concepts>
4 #include <iostream>
5 #include <vector>
6
7 std::integral auto getIntegral(int val) {
8     return val;
9 }
10
11 int main() {
12
13     std::cout << std::boolalpha << '\n';
14
15     std::vector<int> vec{1, 2, 3, 4, 5};
16     for (std::integral auto i: vec) std::cout << i << " ";
17     std::cout << '\n';
18
19     std::integral auto b = true;
20     std::cout << b << '\n';
21
22     std::integral auto integ = getIntegral(10);
23     std::cout << integ << '\n';
24
25     auto integ1 = getIntegral(10);
26     std::cout << integ1 << '\n';
27
28     std::cout << '\n';
29 }
```

std::integral 概念可以用作返回类型 (第 7 行)、基于范围的 for 循环 (第 16 行)，或者用作变量 b (第 19 行) 或变量 integ (第 22 行) 的类型。为了查看 auto 和概念之间的对称性，第 25 行单独使用 auto，而不是在第 22 行使用的 std::integral auto。因此，integ1 可以接受任何类型的值。

```
1 2 3 4 5  
true  
10  
10
```

4.1.6 简写函数模板

C++20 中，可以在函数声明中使用不受约束的占位符 (auto) 或受约束的占位符 (concept)，这个函数声明会自动成为一个函数模板。

```
1 // abbreviatedFunctionTemplates.cpp  
2  
3 #include <concepts>  
4 #include <iostream>  
5  
6 template<typename T>  
7 requires std::integral<T>  
8 T gcd(T a, T b) {  
9     if( b == 0 ) return a;  
10    else return gcd(b, a % b);  
11 }  
12  
13 template<typename T>  
14 T gcd1(T a, T b) requires std::integral<T> {  
15     if( b == 0 ) return a;  
16     else return gcd1(b, a % b);  
17 }  
18  
19 template<std::integral T>  
20 T gcd2(T a, T b) {  
21     if( b == 0 ) return a;  
22     else return gcd2(b, a % b);  
23 }  
24  
25 std::integral auto gcd3(std::integral auto a, std::integral auto b) {  
26     if( b == 0 ) return a;  
27     else return gcd3(b, a % b);  
28 }  
29  
30 auto gcd4(auto a, auto b){  
31     if( b == 0 ) return a;  
32     return gcd4(b, a % b);  
33 }  
34  
35 int main() {  
36  
37     std::cout << '\n';
```

```

38 std::cout << "gcd(100, 10)= " << gcd(100, 10) << '\n';
39 std::cout << "gcd1(100, 10)= " << gcd1(100, 10) << '\n';
40 std::cout << "gcd2(100, 10)= " << gcd2(100, 10) << '\n';
41 std::cout << "gcd3(100, 10)= " << gcd3(100, 10) << '\n';
42 std::cout << "gcd4(100, 10)= " << gcd4(100, 10) << '\n';
43
44 std::cout << '\n';
45
46 }
47 }
```

函数模板 gcd(第 6 行)、gcd1(第 13 行) 和 gcd2(第 19 行) 在使用概念的四种方式时已经介绍过了。gcd 使用 requires 子句，gcd1 使用尾部 requires 子句，gcd2 使用约束模板参数。现在来看点新东西，函数模板 gcd3 有 std::integral 概念作为类型参数，因此是一个具有受限类型参数的函数模板。相比之下，gcd4 相当于对其类型参数，对函数模板没有限制。gcd3 和 gcd4 中用于创建函数模板的语法，称为缩写函数模板。

```

gcd(100, 10)= 10
gcd1(100, 10)= 10
gcd2(100, 10)= 10
gcd3(100, 10)= 10
gcd4(100, 10)= 10
```

通过下面的例子来强调这种对称性。

使用 auto 作为类型参数，函数 add 变成了一个函数模板，与同名的函数模板 add 等价。

```

1 template<typename T, typename T2>
2 auto add(T fir, T2 sec) {
3     return fir + sec;
4 }
5
6 auto add(auto fir, auto sec) {
7     return fir + sec;
8 }
```

相应地，由于 std::integral 概念的使用，sub 函数等价于函数模板 sub 函数。

```

1 template<std::integral T, std::integral T2>
2 std::integral auto sub(T fir, T2 sec) {
3     return fir - sec;
4 }
5
6 std::integral auto sub(std::integral auto fir, std::integral auto sec) {
7     return fir - sec;
8 }
```

函数和函数模板可以是任意类型，这两种类型可以是不同的，但必须是整型。例如，使用 `sub(100, 10)` 和 `sub(100, true)` 都可以。

缩写函数模板语法中，仍然缺少一个特性：可以重载 `auto` 或概念。

4.1.6.1 重载

以下函数会在 `auto`、`std::integral` 概念和 `long` 类型上进行重载。

```
1 // conceptsOverloading.cpp
2
3 #include <concepts>
4 #include <iostream>
5
6 void overload(auto t) {
7     std::cout << "auto : " << t << '\n';
8 }
9
10 void overload(std::integral auto t) {
11     std::cout << "Integral : " << t << '\n';
12 }
13
14 void overload(long t) {
15     std::cout << "long : " << t << '\n';
16 }
17
18 int main() {
19
20     std::cout << '\n';
21
22     overload(3.14);
23     overload(2010);
24     overload(2020L);
25
26     std::cout << '\n';
27
28 }
```

编译器选择 `auto`(第 6 行) 上的重载使用 `double`, `std::integral`(第 10 行) 上的重载使用 `int`, `long`(第 14 行) 上的重载使用 `long`。

```
auto : 3.14
Integral : 2010
long : 2020
```

遗漏的特性: 模板

也许在这一章的概念中遗漏了一个特性: 模板。将模板引入是概念技术规范的一部分, [TS ISO/IEC TS 19217:2015](#)是概念的实验性实现。[GCC 6](#)完全实现了概念 TS, 除了语法上与 C++20 中的概念不同之外, 概念 TS 支持一种简洁的定义模板的方式。

下面的例子中, 假设 Integral 是一个概念。

概念 TS 中的模板介绍

```
1 Integral{T}
2 Integral gcd(T a, T b) {
3     if( b == 0 ) { return a; }
4     else{
5         return gcd(b, a % b);
6     }
7 }
8
9 Integral{T}
10 class ConstrainedClass{};
```

上面的这个小代码片段, 以两种方式引入了模板。首先, 定义一个带有约束模板参数的函数模板; 其次, 定义带有约束模板参数的类模板。引入模板有一个限制, 只能将其用于有约束的模板参数 (concept), 而不能用于无约束的模板参数 (auto)。这种不对称性可以通过定义一个总是返回 true 的概念轻松搞定:

Generic 概念的实现

```
1 template<typename T>
2 concept bool Generic() {
3     return true;
4 }
```

不要着急, 我在示例中使用概念 TS 语法来定义泛型概念。C++20 的语法会简洁一些。在定义概念一节中可以了解更多的 C++20 语法细节。

4.1.7 预定义的概念

“不要白费力气”的黄金法则同样适用于概念。[C++ 核心指南](#)对这条规则的定义非常清楚:*T.11*: 只要可能, 就使用标准概念。因此, 我想给出一个重要的预定义概念的概述, 这里会有意地忽略特殊或辅助类型的概念。

所有预定义的概念在最新的 C++20 工作草案[N4860](#)中都有详细描述, 找到它们是一个相当大的挑战! 大部分概念都在第 18 章 (概念库) 和第 24 章 (范围库) 中。另外, 第 17 章 (语言支持库)、第 20 章 (通用实用程序库)、第 23 章 (迭代器库) 和第 26 章 (数字库) 中有一些概念。C++20 草案 N4860 还提供了所有库概念的索引, 并展示了如何实现这些概念。

4.1.7.1 语言支持库

本节讨论一个有趣的概念——`three_way_comparable`, 支持三向比较运算符, 在头文件`<compare>`中定义。

更正式地说, 设 `a` 和 `b` 是类型为 `t` 的值。只有在以下情况下, 这些值才支持 `three_way_comparable`:

- `(a <= b == 0) == bool(a == b) is true`
- `(a <= b != 0) == bool(a != b) is true`
- `((a <= b) <= 0) and (0 <= (b <= a)) are equal`
- `(a <= b < 0) == bool(a < b) is true`
- `(a <= b > 0) == bool(a > b) is true`
- `(a <= b <= 0) == bool(a <= b) is true`
- `(a <= b >= 0) == bool(a >= b) is true`

4.1.7.2 概念库

最常用的概念可以在概念库中找到, 在`<concepts>`头文件中定义。

4.1.7.2.1 语言相关的概念

本节有大约 15 个概念, 这些概念表示类型、类型分类和基本类型属性之间的关系。其实现通常直接基于[类型特性库](#)中的相应函数。若有必要, 我会提供额外的解释。

- `same_as`
- `derived_from`
- `convertible_to`
- `common_reference_with`: `common_reference_with<T, U>` 必须定义良好, `T` 和 `U` 必须可以转换为引用类型 `C`, 其中 `C` 与 `common_reference_t<T, U>` 相同
- `common_with`: 类似于 `common_reference_with`, 但是 `common` 类型 `C` 与 `common_type_t<T, U>` 相同, 并且可能不是引用类型
- `assignable_from`
- `swappable`

4.1.7.2.2 算术的概念

- `integral`
- `signed_integral`
- `unsigned_integral`
- `floating_point`

该标准对算术概念的定义很简单:

```

1 template<class T>
2 concept integral = is_integral_v<T>;
3
4 template<class T>
5 concept signed_integral = integral<T> && is_signed_v<T>;
6
7 template<class T>
8 concept unsigned_integral = integral<T> && !signed_integral<T>;
9
10 template<class T>
11 concept floating_point = is_floating_point_v<T>;

```

4.1.7.2.3 生命周期的概念

- destructible
- constructible_from
- default_constructible
- move_constructible
- copy_constructible

4.1.7.2.4 比较的概念

- equality_comparable
- totally_ordered

学习数学时会了解: 对于类型 T 的值 a、b 和 c, 若以下描述成立, 则 T 类型为 totally_ordered(全序关系)

- $\text{bool}(a < b)$ 、 $\text{bool}(a > b)$ 或 $\text{bool}(a == b)$ 中的一个为真
- $\text{bool}(a < b)$ 和 $\text{bool}(b < c)$, 则 $\text{bool}(a < c)$
- $\text{bool}(a > b) == \text{bool}(b < a)$
- $\text{bool}(a \leq b) == \neg \text{bool}(b < a)$
- $\text{bool}(a \geq b) == \neg \text{bool}(a < b)$

4.1.7.2.5 对象的概念

- movable
- copyable
- semiregular
- regular

以下是这四个概念的定义:

```

1 template<class T>
2 concept movable = is_object_v<T> && move_constructible<T> &&

```

```

3     assignable_from<T&, T> && swappable<T>;
4
5 template<class T>
6 concept copyable = copy_constructible<T> && movable<T> &&
7     assignable_from<T&, T&> &&
8     assignable_from<T&, const T&> && assignable_from<T&, const T>;
9
10 template<class T>
11 concept semiregular = copyable<T> && default_initializable<T>;
12
13 template<class T>
14 concept regular = semiregular<T> && equality_comparable<T>;

```

我必须补充几点。可移动的概念要求 T 的 `is_object_v<T>` 条件成立。根据类型特性 `is_object<T>` 的定义，T 可以是标量、数组、联合体或者类。

定义概念部分还实现了半常规和常规的概念。不太正式地说，半常规类型的行为类似于 `int` 类型；而常规类型的行为也类似于 `int` 类型，并且可以使用 `==` 操作符进行比较。

4.1.7.2.6 可调用的概念

- `invocable`
- `regular_invocable`: 类型建模为可调用且保持相等，并且不修改函数参数；保持相等，说明在给定相同的输入时会产生相同的输出
- `predicate`: 若类型对可调用对象建模并返回布尔值，则可对谓词建模

4.1.7.3 通用工具库

本章在标准中只有特殊的内存概念，因此在这里不提及它们。

4.1.7.4 迭代器库

迭代器库有许多重要的概念，在 `<iterator>` 头文件中定义。下面是迭代器的类别：

- `input_iterator`
- `output_iterator`
- `forward_iterator`
- `bidirectional_iterator`
- `random_access_iterator`
- `contiguous_iterator`

这六类迭代器对应于各自的迭代器概念。下表提供了两条有趣的信息。对于三个最突出的迭代器类别，该表显示了它们的属性和相关的标准库容器。

迭代器类别	属性	相应容器
std::forward_iterator	<code>++It, It++, *It</code>	<code>std::unordered_set</code>
	<code>It==It2, It!=It2</code>	<code>std::unordered_map</code> <code>std::unordered_multiset</code> <code>std::unordered_multimap</code> <code>std::forward_list</code>
std::bidirectional_iterator	<code>--It, It--</code>	<code>std::set</code> <code>std::map</code>
		<code>std::multiset</code> <code>std::multimap</code> <code>std::list</code>
std::random_access_iterator	<code>It[i]</code>	
	<code>It += n, It -= n</code>	<code>std::array</code>
	<code>It + n, It - n,</code>	<code>std::vector</code>
	<code>n + It,</code>	<code>std::deque</code>
	<code>It - It2,</code>	<code>std::string</code>
	<code>It < It2, It <= It2</code>	
	<code>It < It2, It >= It2</code>	

以下关系成立:

- 随机访问迭代器是双向迭代器，双向迭代器是前向迭代器。
- 连续迭代器是一种随机访问迭代器，要求容器的元素连续存储在内存中。

所以 `std::array`, `std::vector` 和 `std::string` 支持连续迭代器，但不包括 `std::deque`。

4.1.7.4.1 算法的概念

- `permutable`: 元素可直接重排序
- `mergeable`: 可以将已排序的序列合并到输出序列中
- `sortable`: 可以将一个序列排列成有序序列

4.1.7.5 范围库

范围库包含对范围和视图特性至关重要的概念，类似于迭代器库中的概念，定义在 `<ranges>` 头文件中。

4.1.7.5.1 范围

- `range`: 范围指定可以遍历的一组项，提供了一个开始迭代器和一个结束哨兵。当然，STL 容器也有范围。

对于 `std::ranges::range` 还可以进一步的细化。

- `input_range`: 指定一个范围，其迭代器类型满足 `input_iterator`(例如，可以从开始迭代到结束至少一次)
- `output_range`: 指定迭代器类型满足 `output_iterator` 的范围
- `forward_range`: 指定一个范围，其迭代器类型满足 `forward_iterator`(可以从开始到结束迭代多次)
- `bidirectional_range`: 指定迭代器类型满足 `bidirectional_iterator` 的范围(可以向前和向后迭代不止一次)
- `random_access_range`: 指定迭代器类型满足 `random_access_iterator` 的范围(可以在常量时间内使用索引操作符 [] 跳访问任意元素)
- `contiguous_range`: 指定一个范围，其迭代器类型满足 `contiguous_iterator`(元素连续存储在内存中)

标准模板库的每个容器都支持特定的范围，支持的范围指定了其迭代器的功能。

迭代器类型	属性	相应容器
<code>std::ranges::input_range</code>	<code>++It, It++, *It</code> <code>It == It2, It != It2</code>	<code>std::unordered_set</code> <code>std::unordered_map</code> <code>std::unordered_multiset</code> <code>std::unordered_multimap</code> <code>std::forward_list</code>
<code>std::ranges::bidirectional_range</code>	<code>--It, It--</code>	<code>std::set</code> <code>std::map</code> <code>std::multiset</code> <code>std::multimap</code> <code>std::list</code>
<code>std::ranges::random_access_range</code>	<code>It[i]</code> <code>It += n, It -= n</code> <code>It + n, It - n,</code> <code>n + It,</code> <code>It - It2,</code> <code>It < It2, It <= It2</code> <code>It < It2, It >= It2</code>	<code>std::deque</code>
<code>std::ranges::contiguous_range</code>	<code>It[i]</code> <code>It += n, It -= n</code> <code>It + n, It - n,</code> <code>n + It,</code> <code>It - It2,</code> <code>It < It2, It <= It2</code> <code>It < It2, It >= It2</code>	<code>std::array</code> <code>std::vector</code> <code>std::string</code>

若容器支持 `std::ranges::continuous_range` 概念，则支持表中提到的所有概念，如

`std::ranges::random_access_range`, `std::ranges::bidirectional_range` 和 `std::ranges::input_range`。对于其他范围同理。

4.1.7.5.2 视图

`std::ranges::view` 通常是应用在一个范围内，并用其执行一些操作。视图不拥有数据，视图用于复制、移动或赋值的时间恒定。下面引用 Eric Niebler 的 range-v3 实现 (是 C++20 范围的基础): “视图是范围的组合适配，随着视图的迭代，这种适配的功能会慢慢地发挥其功效。”

4.1.7.6 数值库

数值库提供了 `uniform_random_bit_generator` 的概念，该概念定义在头文件 `<random>` 中。类型 `G` 的 `uniform_random_bit_generator g` 必须返回均匀分布的无符号整数，类型为 `G` 的均匀随机位生成器 `g` 必须支持成员函数 `G::min` 和 `G::max`。

4.1.8 自定义概念

当在 C++20 中没有合适的预定义的概念时，可以自定义概念。在本节中，我将定义几个概念，使用 CamelCase 语法将它们与预定义的概念区别开来。因此，我的带符号整型的概念命名为 `signeintegral`，而 C++ 标准的概念为 `signed_integral`。

定义概念的语法很简单：

```
1 template <template-parameter-list>
2 concept concept-name = constraint-expression;
```

概念定义从关键字 `template` 开始，并有一个模板参数列表。第二行更有趣，使用关键字概念，后面跟着概念名称和约束表达式。

约束表达式可以是：

- 概念或编译时谓词的逻辑组合
 - 逻辑组合可以由连接 (`&&`)、析取 (`||`) 或否定 (`!`)
 - 编译时谓词是在编译时返回布尔值的可调用对象
- 需求表达式
 - 简单的需求
 - 类型的需求
 - 复合的需求
 - 嵌套的需求

接下来的两节中，将演示定义概念的各种方法。

4.1.8.1 其他概念和编译时谓词的组合

可以使用连接词 (`&&`) 和析取词 (`||`) 组合概念和编译时谓词。构建逻辑组合时，可以使用感叹号 (`!`) 否定。因为[类型特性库](#)有许多编译时谓词，所以具备了使用构建强大概念所需的工具。

不要递归地定义概念或尝试约束它们

概念的递归定义无效:

递归地定义概念

```
1 template<typename T>
2 concept Recursive = Recursive<T*>;
```

GCC 编译器在这种情况下抱怨'Recursive' 没有在此作用域中声明。

当尝试约束概念 (例如下面的代码片段) 时, GCC 编译器会明确地提示概念不能约束。

约束概念

```
1 template<typename T>
2 concept AlwaysTrue = true;
3
4 template<typename T>
5 requires AlwaysTrue<T>
6 concept Error = true;
```

我们先从 Integral、signeintegral 和 unsigneintegral 概念开始。

```
1 template <typename T>
2 concept Integral = std::is_integral<T>::value;
3
4 template <typename T>
5 concept SignedIntegral = Integral<T> && std::is_signed<T>::value;
6
7 template <typename T>
8 concept UnsignedIntegral = Integral<T> && !SignedIntegral<T>;
```

我使用类型特性函数std::is_integral来定义 Integral 概念 (第 2 行)。由于函数 std::is_signed, 将 Integral 概念改进为 SignedIntegral 概念 (第 4 行)。最后, 对 SignedIntegral 概念进行否定, 得到了 UnsignedIntegral 概念 (第 7 行)。

Okay, 我们来试试看。

```
1 // SignedUnsignedIntegers.cpp
2
3 #include <iostream>
4 #include <type_traits>
5
6 template <typename T>
7 concept Integral = std::is_integral<T>::value;
```

```

8
9 template <typename T>
10 concept SignedIntegral = Integral<T> && std::is_signed<T>::value;
11
12 template <typename T>
13 concept UnsignedIntegral = Integral<T> && !SignedIntegral<T>;
14
15 void func(SignedIntegral auto integ) {
16     std::cout << "SignedIntegral: " << integ << '\n';
17 }
18
19 void func(UnsignedIntegral auto integ) {
20     std::cout << "UnsignedIntegral: " << integ << '\n';
21 }
22
23 int main() {
24
25     std::cout << '\n';
26
27     func(-5);
28     func(5u);
29
30     std::cout << '\n';
31
32 }
```

使用简化的函数模板语法重载概念 SignedIntegral(第 15 行) 和 UnsignedIntegral(第 19 行) 上的函数 func。编译器选择预期的重载:

```

SignedIntegral: -5
UnsignedIntegral: 5
```

出于完整性的原因，会下面的算术概念中使用析取。

```

1 template <typename T>
2 concept Arithmetic = std::is_integral<T>::value || std::is_floating_point<T>::value;
```

4.1.8.2 需求表达式

因为需求表达式，现在可以定义功能强大的概念。需求表达式有如下形式:

```
1 requires (parameter-list(optional)) {requirement-seq}
```

- 参数列表: 以逗号分隔的参数列表，例如: 在函数声明中
- 需求序列: 由简单需求、类型需求、复合需求或嵌套需求组成

4.1.8.2.1 简单的需求

下面是概念 Addable 的简单需求:

```
1 template<typename T>
2 concept Addable = requires (T a, T b) {
3     a + b;
4 };
```

Addable 的概念要求两个相同类型 T 可以进行加法 $a + b$ 。

避免使用匿名概念

可以定义一个匿名概念并直接使用，但请避免这样做。这会使得代码难以阅读，并且无法重用相应的概念。

一个匿名概念，用于添加两个概念

```
1 template<typename T>
2     requires requires (T x) { x + x; }
3 T add1(T a, T b) { return a + b; }
```

函数模板自定义了概念，Add1 在 require 子句中使用需求表达式。匿名概念等同于前面定义的概念 Addable，下面使用命名概念 Addable 的函数模板 add2 也是如此。

使用 Addable 概念

```
1 template<Addable T>
2 T add2(T a, T b) { return a + b; }
```

概念应该对一般情况进行封装，并为它们提供一个自解释的名称以便重用。这对于维护代码非常重要。匿名概念读起来更像模板参数的语法约束。

4.1.8.2.2 类型的需求

类型的需求中，必须使用关键字 `typename` 和类型名。

```
1 template<typename T>
2 concept TypeRequirement = requires {
3     typename T::value_type;
4     typename Other<T>;
5 };
```

TypeRequirement 概念要求类型 T 有一个嵌套的成员 `value_type`，并且类模板 `Other` 可以用 T 实例化。让我们试试这个：

```
1 #include <iostream>
2 #include <vector>
3
4 template <typename>
5 struct Other;
```

```

7 template <>
8 struct Other<std::vector<int>> { };
9
10 template<typename T>
11 concept TypeRequirement = requires {
12     typename T::value_type;
13     typename Other<T>;
14 };
15
16 int main() {
17
18     TypeRequirement auto myVec= std::vector<int>{1, 2, 3};
19
20 }

```

表达式 TypeRequirement auto myVec = std::vector<int>{1, 2, 3}(第 18 行) 有效。std::vector 有一个内部成员 value_type(第 12 行), 类模板 Other 可以实例化 std::vector<int>(第 13 行)。

4.1.8.2.3 复合需求

复合需求有这样的形式

```
| {expression} noexcept(optional) return-type-requirement(optional);
```

除了简单的需求外, 复合需求还可以有**noexcept** 说明符和关于其返回类型的需求。

下面的示例中演示了在 Equal 概念中, 使用复合需求。

```

1 // conceptsDefinitionEqual.cpp
2
3 #include <concepts>
4 #include <iostream>
5
6 template<typename T>
7 concept Equal = requires(T a, T b) {
8     { a == b } -> std::convertible_to<bool>;
9     { a != b } -> std::convertible_to<bool>;
10 };
11
12 bool areEqual(Equal auto a, Equal auto b){
13     return a == b;
14 }
15
16 struct WithoutEqual{
17     bool operator==(const WithoutEqual& other) = delete;
18 };
19
20 struct WithoutUnequal{
21     bool operator!=(const WithoutUnequal& other) = delete;
22 };
23
24 int main() {

```

```

25
26     std::cout << std::boolalpha << '\n';
27     std::cout << "areEqual(1, 5): " << areEqual(1, 5) << '\n';
28
29 /*
30
31     bool res = areEqual(WithoutEqual(), WithoutEqual());
32     bool res2 = areEqual(WithoutUnequal(), WithoutUnequal());
33
34 */
35
36     std::cout << '\n';
37
38 }

```

Equal 概念(第 6 行)要求其类型参数 T 支持相等和不相等操作符。此外，两个操作符都必须返回一个可转换为布尔值的值。当然，int 支持 Equal 概念，但这并不适用于 WithoutEqual(第 16 行) 和 WithoutUnequal(第 20 行) 类型。因此，当使用 WithoutEqual 类型时(第 31 行)，在使用 GCC 编译器时，会得到以下错误消息。

```

<source>:6:17:  in requirements with 'T a', 'T b' [with T = WithoutEqual]
<source>:7:9: note: the required expression '(a == b)' is invalid
    7 |     { a == b } -> std::convertible_to<bool>;
          | ~~^~~~
<source>:8:9: note: the required expression '(a != b)' is invalid
    8 |     { a != b } -> std::convertible_to<bool>;
          | ~~^~~~

```

WithoutEqual 不匹配 Equal 概念

4.1.8.2.4 嵌套需求

嵌套需求的形式

```
1 requires constraint-expression;
```

嵌套需求用于指定类型参数上的需求。

下面是定义概念 `unsignedintegral` 的另一种方法(参见概念和谓词的逻辑组合):

```

1 // nestedRequirements.cpp
2
3 #include <type_traits>
4
5 template <typename T>
6 concept Integral = std::is_integral<T>::value;
7
8 template <typename T>
9 concept SignedIntegral = Integral<T> && std::is_signed<T>::value;
10
11 // template <typename T>
12 // concept UnsignedIntegral = Integral<T> && !SignedIntegral<T>;

```

```

13
14 template <typename T>
15 concept UnsignedIntegral = Integral<T> &&
16 requires(T) {
17     requires !SignedIntegral<T>;
18 };
19
20 int main() {
21
22 UnsignedIntegral auto n = 5u; // works
23 // UnsignedIntegral auto m = 5; // compile time error, 5 is a signed literal
24
25 }

```

第 14 行使用 `signeintegral` 概念，嵌套需求来细化 `Integral` 概念。老实说，第 11 行中注释掉的概念 `unsigneintegral` 阅读起来更方便。

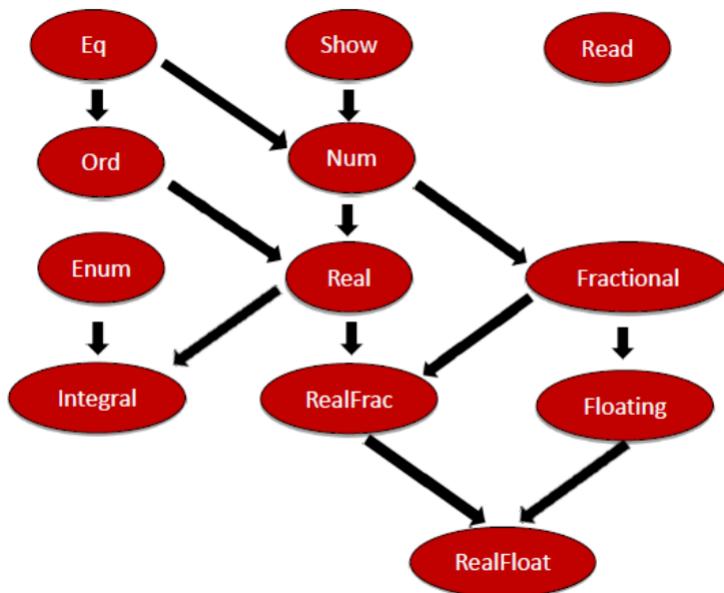
下一节中的有序概念演示了嵌套需求的使用。

4.1.9 应用

前面的章节中，回答了关于概念的两个基本问题：“如何使用概念？”和“如何定义你的概念？”本节中，将应用这些理论知识来定义更高级的概念，如 `Ordering`、`SemiRegular` 和 `Regular`。

4.1.9.1 Equal 和 Ordering 的概念

我已经在 Haskell 的类型的层次结构中介绍了概念的漫长历史：



Haskell 类型的层次结构

类层次结构表明类型 `Ord` 是类型 `Eq` 的细化，Haskell 优雅地表达了这一点。

Haskell 类型层次结构的一部分

```

1 class Eq a where
2   (==) :: a -> a -> Bool
3   (/=) :: a -> a -> Bool
4
5 class Eq a => Ord a where
6   compare :: a -> a -> Ordering
7   (<) :: a -> a -> Bool
8   (<=) :: a -> a -> Bool
9   (>) :: a -> a -> Bool
10  (>=) :: a -> a -> Bool
11  max :: a -> a -> a

```

每个类型 a 支持类型类 Eq(第 1 行), 必须支持等式(第 2 行)和不等式(第 3 行)。支持类型类 Ord 的每个类型 a 都必须支持类型类 Eq(在第 5 行中, (Eq a => Ord a))。此外, 类型 a 必须支持四个比较操作符, 以及 compare 和 max 函数(第 6-11 行)。

能否用 C++20 中的概念来表达 Haskell 在类型 Eq 和 Ord 之间的关系? 为了简单起见, 这里忽略 Haskell 的 compare 和 max 函数。

4.1.9.1.1 排序概念

有了需求表达式, 排序概念的定义看起来与 Haskell 中类型类 ord 的定义相似。

```

1 template <typename T>
2 concept Ordering =
3   Equal<T> &&
4   requires(T a, T b) {
5     { a <= b } -> std::convertible_to<bool>;
6     { a < b } -> std::convertible_to<bool>;
7     { a > b } -> std::convertible_to<bool>;
8     { a >= b } -> std::convertible_to<bool>;
9   };

```

排序概念在底层使用了嵌套的需求。类型 T 若支持 Equal 概念, 则支持排序概念, 此外还支持四个比较运算符。

排序概念的定义和使用

```

1 // conceptsDefinitionOrdering.cpp
2
3 #include <concepts>
4 #include <iostream>
5 #include <unordered_set>
6
7 template<typename T>
8 concept Equal =
9   requires(T a, T b) {
10     { a == b } -> std::convertible_to<bool>;
11     { a != b } -> std::convertible_to<bool>;
12   };

```

```

13
14
15 template <typename T>
16 concept Ordering =
17     Equal<T> &&
18     requires(T a, T b) {
19         { a <= b } -> std::convertible_to<bool>;
20         { a < b } -> std::convertible_to<bool>;
21         { a > b } -> std::convertible_to<bool>;
22         { a >= b } -> std::convertible_to<bool>;
23     };
24
25 template <Equal T>
26 bool areEqual(const T& a, const T& b) {
27     return a == b;
28 }
29
30 template <Ordering T>
31 T getSmaller(const T& a, const T& b) {
32     return (a < b) ? a : b;
33 }
34
35 int main() {
36
37     std::cout << std::boolalpha << '\n';
38
39     std::cout << "areEqual(1, 5): " << areEqual(1, 5) << '\n';
40
41     std::cout << "getSmaller(1, 5): " << getSmaller(1, 5) << '\n';
42
43     std::unordered_set<int> firSet{1, 2, 3, 4, 5};
44     std::unordered_set<int> secSet{5, 4, 3, 2, 1};
45
46     std::cout << "areEqual(firSet, secSet): " << areEqual(firSet, secSet) << '\n';
47
48 // auto smallerSet = getSmaller(firSet, secSet);
49
50     std::cout << '\n';
51
52 }

```

函数模板 `areEqual`(第 25 行) 要求实参 `a` 和 `b` 具有相同的类型并支持 `Equal` 概念，函数模板 `getsmall`(第 30 行) 要求两个参数都支持有序概念。当然，像 1 和 5 这样的整数可以同时匹配这两个概念，而 `std::unordered_set` 并不匹配有序概念。

因此，我将第 48 行注释掉。

```
areEqual(1, 5): false
getSmaller(1, 5): 1
areEqual(firstSet, secondSet): true
```

接下来，当编译第 48 行会发生什么？GCC 编译器明确指出 `std::unordered_set` 不是函数模板 `getsmall` 的有效参数。

```
<source>:48:48: required from here
<source>:16:9: required for the satisfaction of 'Ordering<T>' [with T = std::unordered_set<int, std::hash<int>, std::equal_to<int>, std::allocator<int> >]
<source>:18:5: in requirements with 'T a', 'T b' [with T = std::unordered_set<int, std::hash<int>, std::equal_to<int>, std::allocator<int> >]
<source>:19:13: note: the required expression '(a <= b)' is invalid
  19 |     { a <= b } -> std::convertible_to<bool>;
     |     ~~^~~
<source>:20:13: note: the required expression '(a < b)' is invalid
  20 |     { a < b } -> std::convertible_to<bool>;
     |     ~~^~~
<source>:21:13: note: the required expression '(a > b)' is invalid
  21 |     { a > b } -> std::convertible_to<bool>;
     |     ~~^~~
<source>:22:13: note: the required expression '(a >= b)' is invalid
  22 |     { a >= b } -> std::convertible_to<bool>;
     |     ~~^~~
```

函数模板 `getsmall` 的使用错误

排序概念已经是 C++20 标准的一部分。

- `std::three_way_comparable`: 等价于上面提出的排序概念
- `std::three_way_comparable_with`: 允许比较不同类型的值；例如: `1.0 < 1.0f`

C++20 中，可以使用三向比较操作符，也称为宇宙飞船操作符 `<=>`。我将在三向比较运算符的章节中介绍它。

4.1.9.2 半常规 (SemiRegular) 和常规 (Regular) 的概念

想要在 C++ 生态系统中定义一个工作良好的类型时，应该定义一个“行为像 `int` 型”的类型。形式上，具体类型应该是常规类型在本节中，来定义半常规和常规概念。

半常规和常规是 C++ 中的基本思想。抱歉，我应该说概念。例如，在《C++ 核心指南》中的解释 [T.46: 要求模板参数至少是半常规和常规类型](#)。现在，只剩下一个重要的问题需要回答：什么是常规类型或半常规类型？在讨论之前，先给出结论：

- 常规类型“行为类似于 `int` 型”，可以复制，并且复制操作的结果独立于原始操作，具有相同的值。

更正式一点。常规类型也是半常规类型，让我们开始吧。

常规类型

[Alexander Stepanov](#)，标准模板库的设计者，定义了术语常规类型和半常规类型。根据他的说法，若一个类型支持这些函数，那么它就是常规类型

- 复制构造
- 赋值
- 等式

- 析构
- 全序

复制构造意味着默认构造，等式构造意味着不相等。Stepanov 定义上述需求时，C++ 中还没有移动语义。Alexander Stepanov 和 Paul McJones 合著的书 [Elements of Programming](#) 专门介绍了常规类型。

4.1.9.2.1 半常规概念

一个半常规的 X 型必须支持六大函数，并且可交换。六大函数包括：

- 默认构造函数：

```
1 X()
```

- 复制构造函数：

```
1 X(const X&)
```

- 复制赋值操作符：

```
1 X& operator = (const X&)
```

- 移动构造函数：

```
1 X(X&&)
```

- 移动赋值操作符：

```
1 X& operator = (X&&)
```

- 析构函数：

```
1 ~X()
```

此外，X 必须是可交换的：swap(X&, X&)

[类型特性库](#) 中定义了相应概念。这里，我定义了类型特征 `isSemiRegular`，然后用它来定义概念 `SemiRegular`。

```

1 template<typename T>
2 struct isSemiRegular: std::integral_constant<bool,
3     std::is_default_constructible<T>::value &&
4     std::is_copy_constructible<T>::value &&
5     std::is_copy_assignable<T>::value &&
6     std::is_move_constructible<T>::value &&
7     std::is_move_assignable<T>::value &&
8     std::is_destructible<T>::value &&
9     std::is_swappable<T>::value >{};
10
11
12 template<typename T>
13 concept SemiRegular = isSemiRegular<T>::value;
```

类型特性 `isSemiRegular`(第 1 行) 是当所有的类型特性到六大函数(第 3-8 行) 和类型特质 `std::is_swappable`(第 9 行) 都满足时实现的。定义 `SemiRegular` 概念的剩余步骤，就是使用类型特征 `isSemiRegular`(第 13 行)。

我们继续解读 `Regular` 概念。

4.1.9.2.2 常规概念

我们已经完成了对 `Regular` 概念的定义。除了 `SemiRegular` 概念的需求外，`Regular` 概念还要求类型具有相等的可比性。我已经在需求表达式一节中定义了 `Equal` 概念。因此，已经完成了，只需要把 `Equal` 和 `SemiRegular` 这两个概念组合起来就可以了。

```
1 template<typename T>
2 concept Regular = Equal<T> &&
3     SemiRegular<T>;
```

现在，如何在 C++20 中定义相应的概念 `std::semiregular` 和 `std::regular`?

4.1.9.2.3 `std::semiregular` 和 `std::regular`

C++20 使用了现有类型特征和概念，定义了 `std::semiregular` 和 `std::regular` 概念。

```
1 template<class T>
2 concept movable = is_object_v<T> && move_constructible<T> &&
3     assignable_from<T&, T> && swappable<T>;
4
5 template<class T>
6 concept copyable = copy_constructible<T> && movable<T> &&
7     assignable_from<T&, T&> &&
8     assignable_from<T&, const T&> && assignable_from<T&, const T>;
9
10 template<class T>
11 concept semiregular = copyable<T> && default_initializable<T>;
12
13 template<class T>
14 concept regular = semiregular<T> && equality_comparable<T>;
```

`std::regular` 概念的定义类似于 `Regular` 概念，`std::semiregular` 概念可以与基本概念相结合，如 `std::copyable` 和 `std::moveable`。`std::movable` 概念基于类型特征函数 `std::is_object`。cppreference.com 还提供了编译时谓词的可能实现。

```
1 template< class T>
2 struct is_object : std::integral_constant<bool,
3     std::is_scalar<T>::value ||
4     std::is_array<T>::value ||
5     std::is_union<T>::value ||
6     std::is_class<T>::value> {};
```

若类型是标量、数组、联合体或类，则它是对象。

结束本节之前，我想使用用户定义的概念 `Regular` 和 C++20 的概念 `std::regular`。`regularSemiRegular.cpp` 完成了这项工作。

概念 Regular 和 SemiRegular 的应用

```
1 // regularSemiRegular.cpp
2
3 #include <concepts>
4 #include <vector>
5 #include <type_traits>
6
7 template<typename T>
8 struct isSemiRegular: std::integral_constant<bool,
9     std::is_default_constructible<T>::value &&
10    std::is_copy_constructible<T>::value &&
11    std::is_copy_assignable<T>::value &&
12    std::is_move_constructible<T>::value &&
13    std::is_move_assignable<T>::value &&
14    std::is_destructible<T>::value &&
15    std::is_swappable<T>::value >{};
16
17 template<typename T>
18 concept SemiRegular = isSemiRegular<T>::value;
19
20 template<typename T>
21 concept Equal =
22     requires(T a, T b) {
23         { a == b } -> std::convertible_to<bool>;
24         { a != b } -> std::convertible_to<bool>;
25     };
26
27 template<typename T>
28 concept Regular = Equal<T> &&
29     SemiRegular<T>;
30
31 template <Regular T>
32 void behavesLikeAnInt(T) {
33     // ...
34 }
35
36 template <std::regular T>
37 void behavesLikeAnInt2(T) {
38     // ...
39 }
40
41 struct EqualityComparable { };
42 bool operator == (EqualityComparable const&,
43                     EqualityComparable const&) {
44     return true;
45 }
46
47 struct NotEqualityComparable { };
```

```

48
49 int main() {
50
51     int myInt{};
52     behavesLikeAnInt(myInt);
53     behavesLikeAnInt2(myInt);
54
55     std::vector<int> myVec{};
56     behavesLikeAnInt(myVec);
57     behavesLikeAnInt2(myVec);
58
59     EqualityComparable equComp;
60     behavesLikeAnInt(equComp);
61     behavesLikeAnInt2(equComp);
62
63     NotEqualityComparable notEquComp;
64     behavesLikeAnInt(notEquComp);
65     behavesLikeAnInt2(notEquComp);
66
67 }

```

我将前面代码片段中的所有部分放在一起定义 Regular 概念(第 27 行)。函数模板 behavesLikeAnInt(第 31 行) 和 behavesLikeAnInt2(第 36 行) 检查参数是否“行为像 int”。使用用户自定义的概念 Regular 和 C++20 的概念 std::regular 来建立条件。所以，类型 EqualityComparable(第 41 行) 支持相 == 操作符，但类型 NotEqualityComparable(第 47 行) 不支持。在两个函数中(第 64 行和第 65 行) 使用 NotEqualityComparable 类型是这个程序中最有趣的部分。

虽然目前处于概念实现的早期阶段，这里就比较一下新的 GCC 和 MSVC 编译器的错误消息。

- GCC

我在[Compiler Explorer](#)上使用当前的 GCC 10.2 命令行参数`-std=c++20`。当使用用户自定义的 Regular(第 64 行) 概念时，会出现编译错误，这里只展示比较重要的错误消息：

```

<source>:23:13: note: the required expression '(a == b)' is invalid
  23 |         { a == b } -> std::convertible_to<bool>;
          | ~~~^~~~
<source>:24:13: note: the required expression '(a != b)' is invalid
  24 |         { a != b } -> std::convertible_to<bool>;
          | ~~~^~~~

```

C++20 概念 `std::regular` 更加全面。因此，第 65 行中的调用给出了一个更全面的错误消息：

```

/opt/compiler-explorer/gcc-10.2.0/include/c++/10.2.0/concepts:282:10: note: the required expression '(__t == __u)' is invalid
282 |     { __t == __u } -> __boolean_testable;
|     ~~~~~^~~~~~
/opt/compiler-explorer/gcc-10.2.0/include/c++/10.2.0/concepts:283:10: note: the required expression '(__t != __u)' is invalid
283 |     { __t != __u } -> __boolean_testable;
|     ~~~~~^~~~~~
/opt/compiler-explorer/gcc-10.2.0/include/c++/10.2.0/concepts:284:10: note: the required expression '(__u == __t)' is invalid
284 |     { __u == __t } -> __boolean_testable;
|     ~~~~~^~~~~~
/opt/compiler-explorer/gcc-10.2.0/include/c++/10.2.0/concepts:285:10: note: the required expression '(__u != __t)' is invalid
285 |     { __u != __t } -> __boolean_testable;
|     ~~~~~^~~~~~

```

- MSVC

MSVC 编译器给出的错误信息就不太具体了。

```

x64 Native Tools Command Prompt for VS 2019

C:\Users\seminar>cl.exe /EHsc /std:c++latest regularSemiRegular.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 19.27.29112 for x64
Copyright (C) Microsoft Corporation. All rights reserved.

/std:c++latest is provided as a preview of language features from the latest C++
working draft, and we're eager to hear about bugs and suggestions for improvements.
However, note that these features are provided as-is without support, and subject
to changes or removal as the working draft evolves. See
https://go.microsoft.com/fwlink/?LinkId=2045807 for details.

regularSemiRegular.cpp
regularSemiRegular.cpp(64): error C2672: 'behavesLikeAnInt': no matching overloaded function found
regularSemiRegular.cpp(64): error C7602: 'behavesLikeAnInt': the associated constraints are not satisfied
regularSemiRegular.cpp(32): note: see declaration of 'behavesLikeAnInt'
regularSemiRegular.cpp(65): error C2672: 'behavesLikeAnInt2': no matching overloaded function found
regularSemiRegular.cpp(65): error C7602: 'behavesLikeAnInt2': the associated constraints are not satisfied
regularSemiRegular.cpp(37): note: see declaration of 'behavesLikeAnInt2'

C:\Users\seminar>

```

使用 `std::regular` 概念时出现错误消息

从截图中看到的，编译器为 19.27.29112 的 x64 版本，并且使用了命令行参数/EHSC 和/std:c++latest。

常规类型

这里我想表达一下我自己的观点。首先，我陈述事实，然后得出结论。这些事实都是基于本章所述内容。那么，哪些论点支持改进，哪些论点支持革命呢？

改进派

- 概念促进在更高抽象级别上使用泛型代码。
- 当编译模板失败时，概念会给出可以理解的错误消息，提供了无法通过[type-trait 库](#)、[SFINAE](#)和[static_assert](#)实现的功能。
- `auto` 是一种不受约束的占位符。C++20 中，可以将概念用作有约束的占位符。
- C++14 中，可以使用泛型 Lambda 定义函数模板。

革命派

- 概念促进在更高抽象级别上使用泛型代码。
- 概念允许我们验证模板需求。当然，也可以通过组合[type-trait](#) 库、[SFINAE](#)和[static assert](#)来实现模板参数的验证，但是这种技术太高级了，不能将其视为通用解决方案。
- 有了简写的函数模板语法，从而模板定义得到了根本性的改进。
- 概念表示语义类别，而不是语法约束。我们不需要像 [Addable](#) 这样的概念，要求类型支持加法操作符，而应该考虑数字概念，其中数字是一个语义类别，例如：相等或有序。

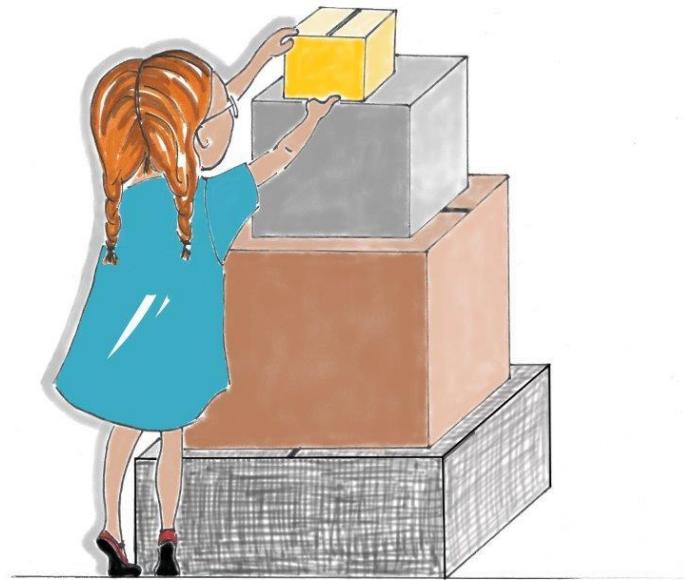
我的结论

关于概念是该改进稳步向前，还是进行革命性的飞跃，有很多争论。争端主要是因为语义分类，我是站在革命派这一边的。诸如“数”、“相等”或“排序”等概念让我想起了[Plato](#)的思想世界。若我们现在可以在这样的语义中进行编程，这将具有革命性意义。

总结

- 定义在特定类型或类型参数上的函数或类有一组问题，概念通过对类型参数施加语义约束来解决这些问题。
- 概念可以应用在 `requires` 子句中，约束的模板参数，或在缩写函数模板中。
- 概念是编译时谓词，可用于各种模板。也可以重载概念，使用概念特化模板，将概念用于成员函数或可变参数模板。
- 因为 C++20 和概念，不受约束占位符 (`auto`) 和受约束占位符 (`concept`) 的使用方式统一了。无论何时使用 `auto`，都可以使用 C++20 中的概念。
- 新的简化的函数模板语法，使得定义函数模板变得更加简单。
- 定义自己的概念之前，请先研究 C++20 标准中丰富的预定义概念集。定义概念时，可以使用两种技术：结合概念和编译时谓词，或者使用需求表达式。

4.2. 模块



Cippi 在准备包裹

模块是 C++20 的四大特性之一，模块的功能有很多：编译时间短，隔离宏，取消头文件，避免工作区。在介绍模块的优点之前，先来了解一下模块。

4.2.1 为什么需要模块？

从一个简单的可执行文件开始：

```
1 // helloWorld.cpp
2
3 #include <iostream>
4
5 int main() {
6     std::cout << "Hello World" << '\n';
7 }
```

使用[GCC](#)可将 `helloWorld.cpp` 编译成一个可执行的 `helloWorld`，可执行文件的大小是文本文件的 130 倍。

```
rainer@seminar:~> wc -c helloWorld.cpp
100 helloWorld.cpp
rainer@seminar:~> g++ helloWorld.cpp -o helloWorld
rainer@seminar:~> wc -c helloWorld
12928 helloWorld
rainer@seminar:~>
```

目标文件的大小

截图中的数字 100 和 12928 代表字节数。现在，我们再对底层发生的事情进行一下了解。

4.2.1.1 传统的构建过程

构建过程包括三个步骤：预处理、编译和链接。

4.2.1.1.1 预处理

预处理器以 #include 和 #define 的方式处理指令。预处理器用相应的头文件替换 #include 指令，并替换宏 (#define)。因为诸如 #if, #else, #elif, #ifdef, #ifndef 和 #endif 等指令，源代码的一部分可以包含或排除。

这个简单的文本替换过程(宏展开)可以通过在 GCC/Clang 上使用编译器标志-E, 或在 Windows 上使用/E, 就可以进行查看了。

```
rainer@seminar:~> g++ -E helloWorld.cpp | wc -c
659471
rainer@seminar:~>
```

预处理器的输出

哇！预处理器的输出超过 50 万个字节。我不想责怪 GCC，因为其他编译器也有同样的情况。预处理器的输出则是编译器的输入。

预处理步骤的结果就是翻译单元。

4.2.1.1.2 编译

编译器在预处理器的输出上执行编译，解析 C++ 源代码，并将其转换为汇编代码。生成的文件称为目标文件，包含二进制形式的编译代码。目标文件可以引用没有定义的符号，也可以放在存档文件中以供后续重用。这些存档文件称为静态库。

编译器生成的对象文件则是链接器的输入。

4.2.1.3 连接

链接器的输出可以是可执行文件、静态库或动态库。链接器的工作是解析对未定义符号的引用，符号在目标文件或库中定义。这个阶段常见的错误是符号没有定义或者定义了不止一次。

这个由三个步骤组成的构建过程继承自 C。若只有一个翻译单元，就能很好地工作。但若有不止一个翻译单元，就会出现很多问题。

4.2.1.2 构建中的问题

下面是经典构建过程中缺陷的不完整列表，这些缺陷可以通过模块来解决。

4.2.1.2.1 重复替换

预处理器用相应的头文件替换 `#include` 指令。修改一下 `helloWorld.cpp` 程序，重构了程序并添加了两个源文件 `hello.cpp` 和 `world.cpp`。源文件 `hello.cpp` 提供了 `hello` 函数，源文件 `world.cpp` 提供了 `world` 函数。

两个源文件都包含相应的头文件。重构意味着具有与前面的程序 `helloWorld.cpp` 相同的外部行为，但内部结构进行了改进。以下是新文件：

- `hello.cpp` 和 `hello.h`

```
1 // hello.cpp
2
3 #include "hello.h"
4
5 void hello() {
6     std::cout << "hello ";
7 }
```

```
1 // hello.h
2
3 #include <iostream>
4
5 void hello();
```

- `world.cpp` 和 `world.h`

```
1 // world.cpp
2
3 #include "world.h"
4
5 void world() {
6     std::cout << "world";
7 }
```

```
1 // world.h
2
3 #include <iostream>
4
```

```
5 void world();
```

- helloWorld2.cpp

```
1 // helloWorld2.cpp
2
3 #include <iostream>
4
5 #include "hello.h"
6 #include "world.h"
7
8 int main() {
9
10    hello();
11    world();
12    std::cout << '\n';
13 }
```

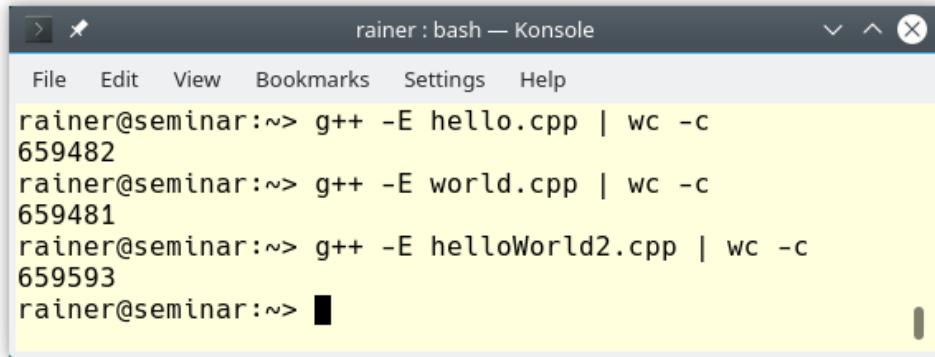
构建和执行如预期的一样:



```
rainer : bash — Konsole
File Edit View Bookmarks Settings Help
rainer@seminar:~> g++ -c hello.cpp -o hello.o
rainer@seminar:~> g++ -c world.cpp -o world.o
rainer@seminar:~> g++ helloWorld2.cpp -o helloWorld2 hello.o world.o
rainer@seminar:~> helloWorld2
hello world
rainer@seminar:~>
```

编译一个简单程序

问题是这样的。预处理器运行在每个源文件上，所以头文件 `<iostream>` 总共包含了三次。因此，每个源文件均多了 50 多万行代码。



```
rainer : bash — Konsole
File Edit View Bookmarks Settings Help
rainer@seminar:~> g++ -E hello.cpp | wc -c
659482
rainer@seminar:~> g++ -E world.cpp | wc -c
659481
rainer@seminar:~> g++ -E helloWorld2.cpp | wc -c
659593
rainer@seminar:~>
```

预处理源文件的大小

这是对编译时间的浪费。与头文件不同，模块只导入一次，所以不会有开销。

4.2.1.2.2 隔离预处理宏

若 C++ 社区有一个共识的话，那就是：应该去掉预处理器宏。为什么？使用宏只是简单的文本替换，不包括任何 C++ 语义。当然，这有许多负面后果：例如，这可能取决于包含宏的顺序，或者宏可能与应用程序中已经定义的宏或名称冲突。

假设有两个头文件 webcolors.h 和 productinfo.h。

先定义一个宏 RED

```
1 // webcolors.h
2 #define RED 0xFF0000
```

再定义一个宏 RED

```
1 // productinfo.h
2 #define RED 0
```

当源文件 client.cpp 包含两个头文件时，宏 RED 的值取决于所包含头文件的顺序。这种依赖关系非常容易出错。

模块则没有导入顺序的问题。

4.2.1.2.3 隔离宏

ODR 代表单一定义规则，对于函数：

- 函数在任何翻译单元中不能有多于一个的定义。
- 函数在程序中不能有多个定义。

可以在多个翻译单元中定义具有外部链接的内联函数，而宏必须满足每个定义的要求。

当链接违背单一定义规则的程序时，链接器会怎么样呢？下面的代码示例有两个头文件 header.h 和 header2.h。主程序包含头文件 header.h 两次，由于包含了 func 的两个定义，因此违背了单一定义规则。

函数 func 的定义

```
1 // header.h
2 void func() {}
```

将函数定义间接包含到 func 中

```
1 // header2.h
2 #include "header.h"
```

双重定义的函数 func

```
1 // main.cpp
2
3 #include "header.h"
4 #include "header2.h"
5
6 int main() {}
```

链接器会说，func 有多个定义：

rainer : bash — Konsole <3>

```
File Edit View Bookmarks Settings Help
rainer@seminar:~/> g++ main.cpp
In file included from header2.h:3:0,
                 from main.cpp:4:
header.h: In function 'void func()':
header.h:3:6: error: redefinition of 'void func()'
 void func(){}
 ^~~~~
In file included from main.cpp:3:0:
header.h:3:6: note: 'void func()' previously defined here
 void func(){}
 ^~~~~
rainer@seminar:~/>
```

违背了单一定义规则

我们已经习惯了一些丑陋的方法，比如：在头文件周围加一个包含守卫。在头文件 header.h 中添加包含守卫 FUNC_H 可以解决这个问题。

使用包含守卫来解决 ODR 问题

```
1 // header.h
2
3 #ifndef FUNC_H
4 #define FUNC_H
5
6 void func() {}
7
8#endif
```

对于模块，不太可能出现重复符号。

那么，现在我就来聊聊模块的优势。

4.2.2 优势

下面以简洁的形式介绍模块的优势：

- 模块只导入一次，编译开销很小。
- 导入模块没有顺序的区别。
- 模块不可能出现重复符号。
- 模块能够表达代码的逻辑结构。可以显式地指定应导出或不应导出的名称，还可以将几个模块捆绑到一个更大的模块中，并将它们作为逻辑包提供出去。
- 不需要再将源代码分离为接口和实现部分。

常规类型

C++ 中的模块可能比你想象的要更加古老。简短的历史回顾应该能让你了解，把如此有价值的东西纳入 C++ 标准需要多长时间。

2004 年, Daveed Vandevoorde 写了一份提案[N1736.pdf](#), 第一次描述了模块的思想。不过, 直到 2012 年才成立了专门的研究小组 (SG2, 模块)。2017 年, Clang 5.0 和 MSVC 19.1 提供了第一个实现。2018 年, 模块 TS(技术规范) 确定。大约在同一时间, Google 针对模块提出了 ATOM(Another Take On Modules) 提案 ([P0947](#))。2019 年, 模块 TS 和 ATOM 提案合并到 C++20 委员会草案中 ([N4842](#))。

4.2.3 举个例子

本节的目的很简单: 介绍模块。模块的更高级特性将在后面几节中介绍。

先从一个简单的数学模块开始吧。

简单的 math 模块

```
1 // math.ixx
2
3 export module math;
4
5 export int add(int fir, int sec){
6     return fir + sec;
7 }
```

表达式导出模块 math 是模块声明。通过将 export 放在函数 add 的定义之前, 将 add 导出。

使用 math 模块

```
1 // client.cpp
2
3 import math;
4
5 int main() {
6     add(2000, 20);
7 }
```

import math 导入 math 模块, 并使模块中导出的名称对 client.cpp 可见。

4.2.3.1 模块的声明文件

是否注意到模块的奇怪名称:math.ixx。

- Microsoft 编译器使用扩展名 ixx, 后缀 ixx 代表模块接口源。
- Clang 编译器最初使用扩展 cppm, 后缀中的 m 可能代表模块。这个约定在 Clang 的新版本中更改为 cpp 扩展(不使用后缀的方式来区分模块)。
- GCC 编译器则不使用特殊的扩展。

全局模块用于组合模块接口, 以关键字 module 开始, 以模块声明结束。全局模块可以出现在使用预处理器指令(如 #include)的地方, 以便模块接口可以编译。全局模块片段中的代码, 不会由模块接口导出。

math 模块的第二个版本支持 add 和 getProduct 两个函数。

全局模块的定义

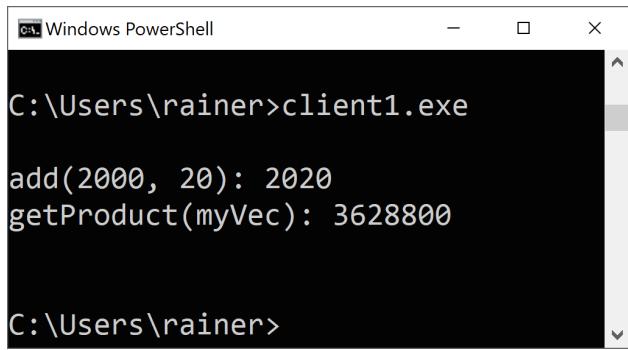
```
1 // math1.ixx
2
3 module;
4
5 #include <numeric>
6 #include <vector>
7
8 export module math;
9
10 export int add(int fir, int sec) {
11     return fir + sec;
12 }
13
14 export int getProduct(const std::vector<int>& vec) {
15     return std::accumulate(vec.begin(), vec.end(), 1, std::multiplies<int>());
16 }
```

我在全局模块(第 3 行)和模块声明(第 8 行)间包含了必要的头文件。

使用改进的 math 模块

```
1 // client1.cpp
2
3 #include <iostream>
4 #include <vector>
5
6 import math;
7
8 int main() {
9
10     std::cout << '\n';
11
12     std::cout << "add(2000, 20): " << add(2000, 20) << '\n';
13
14     std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
15
16     std::cout << "getProduct(myVec): " << getProduct(myVec) << '\n';
17
18     std::cout << '\n';
19 }
```

客户端导入 math 模块，并使用其功能：



```
C:\Users\rainer>client1.exe
add(2000, 20): 2020
getProduct(myVec): 3628800

C:\Users\rainer>
```

执行程序 client1.exe

现在，再了解一下细节。

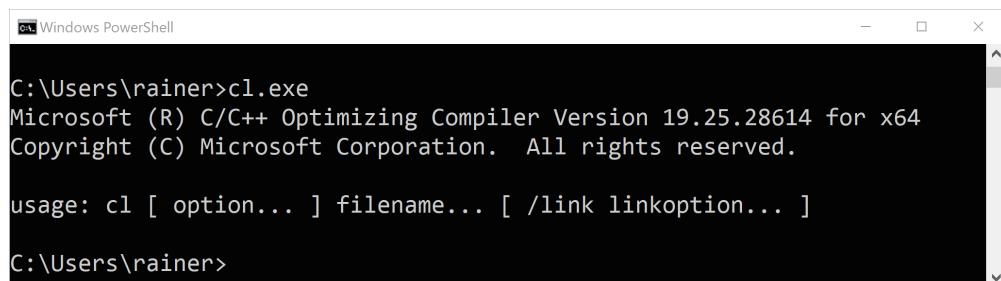
4.2.4 编译和使用

编译 math 模块。若客户端程序 client.cpp 使用 ixx，则必须使用最新的 Clang、GCC 或 Microsoft 编译器。

模块的编译还是具有挑战性，我将用 Microsoft 编译器和 Clang 编译器作为示例来演示模块的编译。

4.2.4.1 Microsoft Visual 编译器

首先，我使用 cl.exe 19.25.28614(x64) 编译器。



```
C:\Users\rainer>cl.exe
Microsoft (R) C/C++ Optimizing Compiler Version 19.25.28614 for x64
Copyright (C) Microsoft Corporation. All rights reserved.

usage: cl [ option... ] filename... [ /link linkoption... ]

C:\Users\rainer>
```

Microsoft 的编译器

这些是使用 Microsoft 编译器编译和使用模块的步骤，我只展示了最短的命令行示例。此外，对于旧的微软编译器，必须使用/std:cpplatest 编译标志。

使用 Microsoft 编译器构建可执行文件

```
cl.exe /experimental:module /c math.ixx
cl.exe /experimental:module client.cpp math.obj
```

- 第 1 行创建一个目标文件 math.obj 和一个 IFC 文件 math.ifc。IFC 文件包含模块接口的元数据描述。IFC 的二进制格式是模仿 Gabriel Dos Reis 和 Bjarne Stroustrup(2004/2005) 的[内部程序表示](#)。

- 第 2 行创建可执行文件 client.exe。若第一步没有隐式使用 math.ifc 文件，则链接器无法找到模块。

这里，就不展示程序执行的输出了。

4.2.4.2 Clang 编译器

在 Linux 上，我使用 Clang 10.0.0 编译器。

```
rainer@seminar:~> clang --version
clang version 10.0.0
Target: x86_64-unknown-linux-gnu
Thread model: posix
InstalledDir: /usr/local/clang_10.0.0/bin
rainer@seminar:~>
```

Clang 10.0.0 编译器

使用 clang 编译器，模块声明文件就是一个 cpp 文件，必须将 math.ixx 重命名为 math.cpp。

```
1 // math.cpp
2
3 export module math;
4
5 export int add(int fir, int sec) {
6     return fir + sec;
7 }
```

客户端文件 client.cpp 没有改变，这是创建可执行文件的必要步骤。

使用 Clang 编译器构建可执行文件

```
clang++ -std=c++2a -stdlib=libc++ -c math.cpp -Xclang \
-emit-module-interface -o math.pcm

clang++ -std=c++2a -stdlib=libc++ -fprebuilt-module-path=. \
client.cpp math.pcm -o client
```

- 第 1 行创建模块 math.pcm，后缀 pcm 代表预编译模块。标志 -std=c++2a 指定 C++20 标准的工作草案，-stdlib=libc++ 指定使用的 C++ 标准库。标记组合 -Xclang -emit-module-interface 是创建预编译模块所需的标志。

- 第 4 行创建可执行程序，使用模块 math.pcm，并用-fprebuilt-module-path 标志指定模块的路径。

4.2.4.3 使用过的编译器

我在这本书中使用的是 Microsoft 的 cl.exe。Microsoft 目前 (2020 年底)对模块的最佳支持的情况。Microsoft 博客提供了两个很好的模块介绍:概述 C++ 模块和C++ 模块的一致性改进与 MSVC 在 Visual Studio 2019 16.5。Clang 和 GCC 都没有提供类似的介绍，因此在这些编译器中使用模块也非常困难。

4.2.5 导出

有三种方法可以导出模块接口单元中的名称。

4.2.5.1 导出说明符

可以显式地导出每个名称。

```
1 export module math;
2 export int mult(int fir, int sec);
3 export void doTheMath();
```

4.2.5.2 导出组

导出组会导出所有名称。

```
1 export module math;
2 export {
3     int mult(int fir, int sec);
4     void doTheMath();
5 }
```

4.2.5.3 导出命名空间

可以使用导出的命名空间，代替导出组。

```
1 export module math;
2 export namespace math {
3     int mult(int fir, int sec);
4     void doTheMath();
5 }
```

当客户端使用来自导出的命名空间时，必须限定这些名称。

只有没有内部链接的名称才可以导出。

4.2.6 构造模块结构的指南

来了解一下构造模块的指导方针。

```
1 module; // global module fragment
2 #include <headers for libraries not modularized so far>
```

```
3 export module math; // module declaration; starts the module purview
4 import <importing of other modules>
5 <non-exported declarations> // names only visible inside the module
6 export namespace math {
7   <exported declarations> // exported names
8 }
```

这个指导原则有一个目的：提供简化的模块结构，以及将要书写的内容。那么，这个模块结构有什么新东西呢？

- 可选以关键字 `module` 开头的全局模块，之后和模块声明之前的位置，是包含头文件的正确位置。
- 模块声明导出模块 `math` 启动模块权限，该权限在翻译单元的末尾结束。
- 可以在模块权限的开始导入模块。导入的模块具有模块链接，在模块外部不可见。这一点也适用于非导出声明。

我将导出的名称放在命名空间 `math` 中，该名称与模块名称相同。

模块只有声明的名称。现在，一起来了解一下接口和模块的实现分离。

4.2.7 模块接口单元和模块实现单元

当模块变大时，应该将其组织成一个模块接口单元和一个或多个模块实现单元。按照前面提到的构造模块的指导原则重构 `math` 模块。

4.2.7.1 模块接口单元

```
1 // mathInterfaceUnit.ixx
2
3 module;
4
5 #include <vector>
6
7 export module math;
8
9 export namespace math {
10
11   int add(int fir, int sec);
12
13   int getProduct(const std::vector<int>& vec);
14
15 }
```

- 模块接口单元包含导出模块声明：`export module math`(第 7 行)。
- 导出 `add` 和 `getProduct`(第 11 和 13 行)。
- 一个模块只能有一个模块接口单元。

4.2.7.2 模块的实现单元

```

1 // mathImplementationUnit.cpp
2
3 module math;
4
5 #include <numeric>
6
7 namespace math {
8
9     int add(int fir, int sec) {
10         return fir + sec;
11     }
12
13     int getProduct(const std::vector<int>& vec) {
14         return std::accumulate(vec.begin(), vec.end(), 1, std::multiplies<int>());
15     }
16 }
```

- 模块实现单元包含非导出模块声明:module math;(第 3 行)。
- 一个模块可以有多个模块实现单元。

4.2.7.3 主程序

```

1 // client3.cpp
2
3 #include <iostream>
4 #include <vector>
5
6 import math;
7
8 int main() {
9
10     std::cout << '\n';
11
12     std::cout << "math::add(2000, 20): " << math::add(2000, 20) << '\n';
13
14     std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
15
16     std::cout << "math::getProduct(myVec): " << math::getProduct(myVec) << '\n';
17
18     std::cout << '\n';
19
20 }
```

从用户的角度来看，除了导入模块 math(第 6 行)，还需要添加命名空间 math。

解释依赖于编译器，我把它们放在一个单独的提示框中。若各位想尝试一下，可以参考这些信息。

使用 Microsoft 编译器构建可执行文件

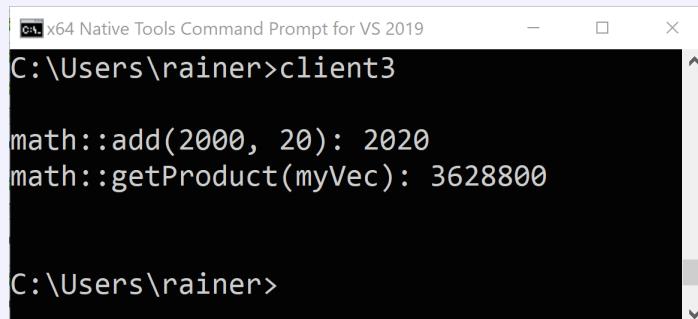
手动构建可执行文件包括以下几个步骤。

用模块接口单元和模块实现单元构建模块

```
cl.exe /c /experimental:module mathInterfaceUnit.ixx /EHsc  
cl.exe /c /experimental:module mathImplementationUnit.cpp /EHsc  
cl.exe /c /experimental:module client3.cpp /EHsc  
cl.exe client3.obj mathInterfaceUnit.obj mathImplementationUnit.obj
```

- 第 1 行：创建了对象文件 mathInterfaceUnit.obj 和模块接口文件 math.ifc。
- 第 2 行：创建了对象文件 mathImplementationUnit.obj。
- 第 3 行：创建对象文件 client3.obj。
- 第 4 行：创建可执行文件 client3.exe。

对于 Microsoft 编译器，必须指定异常处理模型 (/EHsc)，并启用模块支持:/experimental:module。
最后，是程序的输出：



```
C:\x64 Native Tools Command Prompt for VS 2019  
C:\Users\rainer>client3  
  
math::add(2000, 20): 2020  
math::getProduct(myVec): 3628800  
  
C:\Users\rainer>
```

执行程序 client2.exe

4.2.8 子模块和模块分区

当模块变大时，可以将其功能划分为可管理的组件。C++20 模块提供了两种方法：子模块和分区。

4.2.8.1 子模块

模块可以导入模块，然后重新导出它们。

下面的例子中，math 模块导入了子模块 math.math1 和 math.math2。

```
1 // mathModule.ixx  
2 export module math;  
3 export import math.math1;  
4 export import math.math2;
```

表达式 `export import math.math1` 将 math.math1 导入 math 模块，并将其作为 math 模块的一部分重新导出。

完整起见，下面是完整的 math.math1 和 math.math2。我使用单独的代码段，将 math 模块与其子模块分开。实际开发中，代码没必要分开。

子模块 math.math1

```
1 // mathModule1.ixx
2 export module math.math1;
3 export int add(int fir, int sec) {
4     return fir + sec;
5 }
```

子模块 math.math2

```
1 // mathModule2.ixx
2 export module math.math2;
3 export {
4     int mul(int fir, int sec) {
5         return fir * sec;
6     }
7 }
```

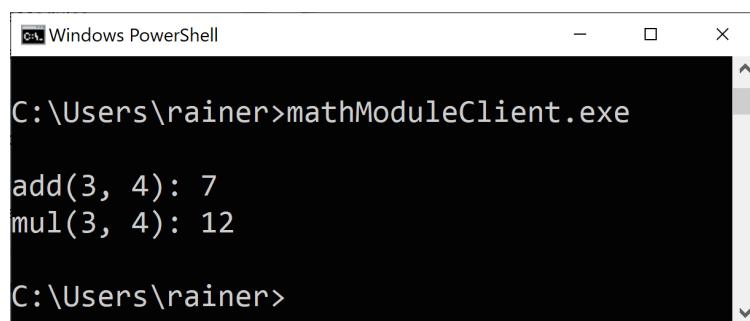
仔细观察会发现，math 模块中的 export 语句相较之前有所不同。math.math1 使用导出说明符，而 math.math2 则使用导出组或导出块。

从使用者的角度来看，使用 math 模块非常简单。

主程序

```
1 // mathModuleClient.cpp
2 #include <iostream>
3 import math;
4 int main() {
5     std::cout << '\n';
6     std::cout << "add(3, 4): " << add(3, 4) << '\n';
7     std::cout << "mul(3, 4): " << mul(3, 4) << '\n';
8 }
```

编译和执行程序可以得到预期输出。



```
C:\Users\rainer>mathModuleClient.exe
add(3, 4): 7
mul(3, 4): 12
C:\Users\rainer>
```

模块和子模块的用法

使用 Microsoft 编译器构建可执行文件

模块和其子模块一同构建可执行文件

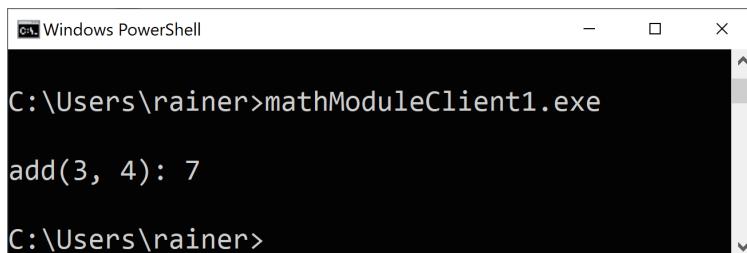
```
cl.exe /c /experimental:module mathModule1..hxx /EHsc
cl.exe /c /experimental:module mathModule2..hxx /EHsc
cl.exe /c /experimental:module mathModule.hxx /EHsc
cl.exe /EHsc /experimental:module mathModuleClient.cpp \
    mathModule1.obj mathModule2.obj mathModule.obj
```

这三个模块的每个编译过程都会创建两个工件:IFC 文件 (接口文件)*.ifc 在最后一行隐式使用, *.obj 文件在最后一行显式使用。

子模块也是模块, 每个子模块都有一个模块声明。用户可以只使用其中之一, 所以这里就创建一个只使用 math.math1 模块的主程序。

主程序只使用子模块 math.math1

```
1 // mathModuleClient1.cpp
2 #include <iostream>
3 import math.math1;
4 int main() {
5     std::cout << '\n';
6     std::cout << "add(3, 4): " << add(3, 4) << '\n';
7 }
```



子模块和模块的用法

将模块划分为模块和子模块, 是模块设计者让用户导入部分模块的一种方法, 但这个观察结果不适用于模块分区。

4.2.8.2 模块分区

一个模块可以划分为多个分区。每个分区由一个模块接口单元 (分区接口文件) 和零个或多个模块实现单元 (参见模块接口单元和模块实现单元) 组成。分区导出的名称由主模块接口单元 (主接口文件) 导入和重新导出。分区名称必须以模块名称开头, 并且分区不能单独存在。

模块分区的描述比其实现更难理解。下面的代码中, 我将为模块分区重写 math 模块, 以及其子模块 math.math1 和 math.math2(参见子模块)。这个过程中, 我会介绍模块分区的术语。

主接口文件

```
1 // mathPartition.ixx
2
3 export module math;
4
5 export import :math1;
6 export import :math2;
```

主接口文件由模块声明(第3行)组成, 使用冒号(第5行和第6行)导入和重新导出分区math1和math2。分区名称必须以模块名称开头, 所以不需要指定。

第一个模块分区

```
1 // mathPartition1.ixx
2
3 export module math:math1;
4
5 export int add(int fir, int sec) {
6     return fir + sec;
7 }
```

第二个模块分区

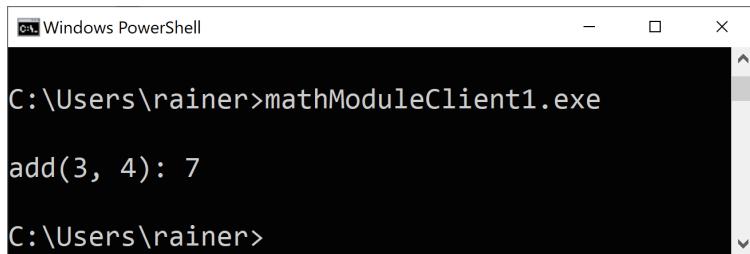
```
1 // mathPartition2.ixx
2
3 export module math:math2;
4
5 export {
6     int mul(int fir, int sec) {
7         return fir * sec;
8     }
9 }
```

与模块声明类似, 表达式导出模块math:math1和导出模块math:math2(第3行)声明了一个模块接口分区。模块接口分区也是模块接口单元。math代表模块, math1或math2代表分区。

导入模块分区

```
1 // mathModuleClient.cpp
2 import math;
3 int main() {
4     std::cout << '\n';
5     std::cout << "add(3, 4): " << add(3, 4) << '\n';
6     std::cout << "mul(3, 4): " << mul(3, 4) << '\n';
7 }
```

假设用户程序与以前在子模块中使用的程序相同, 同样的观察结果也适用于可执行文件的创建和程序的执行:



```
C:\Windows PowerShell
C:\Users\rainer>mathModuleClient1.exe
add(3, 4): 7
C:\Users\rainer>
```

4.2.9 模块中的模板

我经常听到这样的问题: 模块如何导出模板? 当实例化模板时, 其定义必须可用。这就是模板定义放在头文件中的原因。从概念上讲, 模板的使用具有以下结构:

4.2.9.1 无模块

- templateSum.h

```
1 // templateSum.h
2 template <typename T, typename T2>
3 auto sum(T fir, T2 sec) {
4     return fir + sec;
5 }
```

- sumMain.cpp

```
1 // sumMain.cpp
2 #include <templateSum.h>
3 int main() {
4     sum(1, 1.5);
5 }
```

主程序直接包含头文件 templateSum.h, sum(1,1.5) 触发模板实例化。编译器从函数模板 sum 中生成具体的函数 sum, 该函数以 int 型和 double 型作为参数。若想要可视化这个过程, 请使用[C++ Insights](#)上的示例。

4.2.9.2 有模块

C++20 中, 模板应该可以在模块中。模块有一个唯一的内部表示, 既不是源代码也不是程序集。这种表示是一种[抽象语法树](#)(AST)。由于有了这个 AST, 模板定义在模板实例化期间可用。

下面的例子中, 我在模块 math 中定义了函数模板 sum。

- mathModuleTemplate.ixx

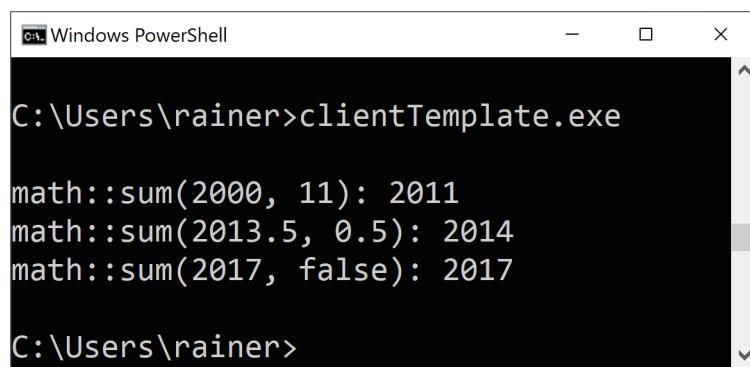
```
1 // mathModuleTemplate.ixx
2 export module math;
3 export namespace math {
4     template <typename T, typename T2>
5     auto sum(T fir, T2 sec) {
6         return fir + sec;
7     }
}
```

```
8 }
```

- clientTemplate.cpp

```
1 // clientTemplate.cpp
2 #include <iostream>
3 import math;
4 int main() {
5     std::cout << '\n';
6     std::cout << "math::sum(2000, 11): " << math::sum(2000, 11) << '\n';
7     std::cout << "math::sum(2013.5, 0.5): " << math::sum(2013.5, 0.5) << '\n';
8     std::cout << "math::sum(2017, false): " << math::sum(2017, false) << '\n';
9 }
```

编译程序的命令行与前面的没有什么不同，这里直接显示程序的输出：



```
Windows PowerShell
C:\Users\rainer>clientTemplate.exe

math::sum(2000, 11): 2011
math::sum(2013.5, 0.5): 2014
math::sum(2017, false): 2017

C:\Users\rainer>
```

通过模块，我们得到了一种新的连接方式。

4.2.10 模块链接

C++20 之前，C++ 支持两种类型的链接：内部链接和外部链接。

- 内部链接：翻译单元之外不能访问具有内部链接，内部链接主要包括声明为静态的命名空间作用域和匿名命名空间的成员。
- 外部链接：具有外部链接的名称可在翻译单元外部访问。外部链接包括声明为非静态的名称、类型及其成员、变量和模板。

C++20 引入模块连接：

- 模块链接：具有模块链接的名称只能在模块内部访问。若名称没有外部链接且不导出，则视为模块链接。

前面的模块声明 mathModuleTemplate.ixx 的变体说明了这个观点。假設想返回给函数模板 sum 的用户的不仅是加法的结果，还有编译器推导出的返回类型。

改进函数模板 sum 的定义

```
1 // mathModuleTemplate1.ixx
2
3 module;
```

```

4
5 #include <iostream>
6 #include <typeinfo>
7 #include <utility>
8
9 export module math;
10
11 template <typename T>
12 auto showType(T&& t) {
13     return typeid(std::forward<T>(t)).name();
14 }
15
16 export namespace math {
17
18     template <typename T, typename T2>
19     auto sum(T fir, T2 sec) {
20         auto res = fir + sec;
21         return std::make_pair(res, showType(res));
22     }
23
24 }
```

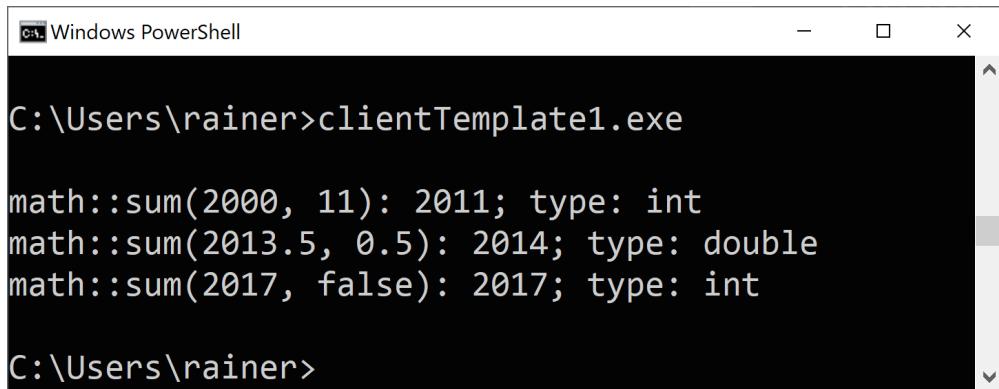
函数模板 `sum` 不返回数字的和，而是返回`std::pair`(第 21 行)，其中包含 `sum` 和 `res` 值类型的字符串表示。注意，我将函数模板 `showType`(第 11 行) 放在导出的 `math` 命名空间(第 16 行)之外。因此，不可能在模块 `math` 外使用。函数模板 `showType` 使用[完美转发](#)来保留函数参数 `t` 的类型。`typeid`运算符在运行时查询有关类型的信息([运行时类型标识 \(RTTI\)](#))。

使用改进的函数模板 `sum`

```

1 // clientTemplate1.cpp
2
3 #include <iostream>
4 import math;
5
6 int main() {
7
8     std::cout << '\n';
9
10    auto [val, message] = math::sum(2000, 11);
11    std::cout << "math::sum(2000, 11): " << val << "; type: " << message << '\n';
12
13    auto [val1, message1] = math::sum(2013.5, 0.5);
14    std::cout << "math::sum(2013.5, 0.5): " << val1 << "; type: " << message1
15        << '\n';
16    auto [val2, message2] = math::sum(2017, false);
17    std::cout << "math::sum(2017, false): " << val2 << "; type: " << message2
18        << '\n';
19
20 }
```

现在，程序显示 sum 的值和自动推导的类型的字符串表示形式。



```
C:\Users\rainer>clientTemplate1.exe

math::sum(2000, 11): 2011; type: int
math::sum(2013.5, 0.5): 2014; type: double
math::sum(2017, false): 2017; type: int

C:\Users\rainer>
```

4.2.11 头文件单元

到 2020 年底，还没有编译器支持头文件单元。头文件单元是一种从头文件转换到模块的平稳方式，只需要用新的 import 指令替换 #include 指令即可。

用 import 指令替换 #include 指令

```
1 #include <vector> => import <vector>;
2 #include "myHeader.h" => import "myHeader.h";
```

首先，import 遵循与 include 相同的查找规则。所以对于引号 ("myHeader.h") 方式来说，在系统搜索路径之前，会先在本地目录中进行搜索。

其次，这不仅仅是文本替换。编译器会根据 import 生成类似模块的东西，并将结果视为模块。导入模块语句从头中获取所有可导出的名称，包括宏。导入这些合成的头文件单元比包含头文件要快，并且在速度上与预编译的头文件相当。

4.2.11.1 一个缺点

头文件单元有一个缺点，并非所有的头文件都是可导入的。哪些头文件可导入已经定义好了，但是 C++ 标准保证所有标准库头文件都是可导入的头文件。导入功能排除了 C 头文件，它们只是简单包装在 std 名称空间中而已，例如 <cstring> 是 <string.h> 的 C++ 包装器。可以很容易地识别包装的 C 头文件，因为其模式是:xxx.h 包装为 cxxx。

使用 Microsoft 编译器构建可执行文件

- 模块克服了头文件和宏的缺陷。模块的导入是无开销的，与宏相比，也无所谓导入顺序。此外，还解决了命名冲突。
- 模块由模块接口单元和模块实现单元组成。必须有一个模块接口单元，并且具有导出模块声明和任意多个模块实现单元。未在模块接口中导出的名称具有模块链接，不能在模块外部使用。
- 模块可以有头文件，也可以导入和重新导出其他模块。
- C++20 中的标准库不是模块化的，用 C++20 构建模块也是一项具有挑战性的任务。
- 要构造大型软件系统，模块提供了两种方式：子模块和分区。与分区相反，子模块可以

独立存在。

- 有了头文件单元，就可以用直接使用 import 语句替换 include，编译器会自动生成一个模块。

4.3. 三向比较操作符



Cippi 在测量身高

三向比较操作符 `<=>` 通常称为“宇宙飞船操作符”。飞船操作符可以确定两个值 A 和 B 是 $A < B$, $A == B$, 还是 $A > B$ 。开发者可以定义飞船操作符, 或者让编译器自动生成。

为了理解三向比较运算符的优点, 先从经典的运算方法开始。

4.3.1 C++20 前比较排序

我使用 `MyInt` 简单的包装了 `int`。这里, 我想比较 `MyInt`。下面是我使用的解决方案——函数模板 `isLessThan`。

MyInt 支持的比较操作很少

```
1 // comparisonOperator.cpp
2 #include <iostream>
3 struct MyInt {
4     int value;
5     explicit MyInt(int val) : value{val} { }
6     bool operator < (const MyInt& rhs) const {
7         return value < rhs.value;
8     }
9 };
10
11 template <typename T>
12 constexpr bool isLessThan(const T& lhs, const T& rhs) {
13     return lhs < rhs;
14 }
15
16 int main() {
17     std::cout << std::boolalpha << '\n';
```

```

18 MyInt myInt2011(2011);
19 MyInt myInt2014(2014);
20 std::cout << "isLessThan(myInt2011, myInt2014): "
21     << isLessThan(myInt2011, myInt2014) << '\n';
22 std::cout << '\n';
23 }

```

该程序按预期工作:

```

rainer@seminar:~/comparisonOperator
isLessThan(myInt2011, myInt2014): true
rainer@seminar:~>

```

使用小于运算符

老实说，MyInt 是一个不直观的类型。当定义六个排序关系中的一个时，就应该定义所有的顺序关系。直观类型至少为半常规。现在，我必须编写大量的样板代码。

这是缺失的 5 个运算符实现:

```

1 bool operator == (const MyInt& rhs) const {
2     return value == rhs.value;
3 }
4 bool operator != (const MyInt& rhs) const {
5     return !(*this == rhs);
6 }
7 bool operator <= (const MyInt& rhs) const {
8     return !(rhs < *this);
9 }
10 bool operator > (const MyInt& rhs) const {
11     return rhs < *this;
12 }
13 bool operator >= (const MyInt& rhs) const {
14     return !(*this < rhs);
15 }

```

现在，来看看 C++20 的三向比较运算符。

4.3.2 C++20 的顺序关系

可以定义三向比较操作符，或者使用 `=default` 让编译器生成。这两种情况下，可以自动得到所有六个比较运算符:==、!=、<、<=、> 和 >=。

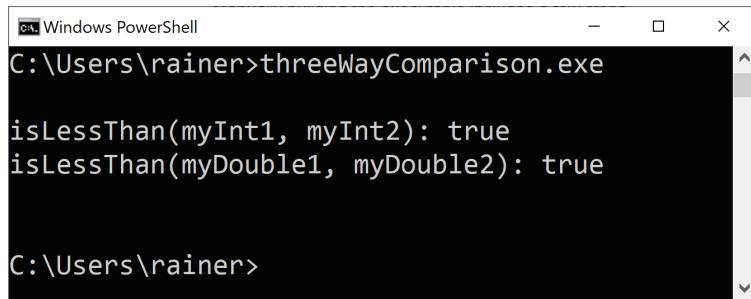
实现或生成的三向比较运算符

```

1 // threeWayComparison.cpp
2
3 #include <compare>
4 #include <iostream>
5
6 struct MyInt {
7     int value;
8     explicit MyInt(int val) : value{val} { }
9     auto operator<=>(const MyInt& rhs) const {
10         return value <=> rhs.value;
11     }
12 };
13
14 struct MyDouble {
15     double value;
16     explicit constexpr MyDouble(double val) : value{val} { }
17     auto operator<=>(const MyDouble&) const = default;
18 };
19
20 template <typename T>
21 constexpr bool isLessThan(const T& lhs, const T& rhs) {
22     return lhs < rhs;
23 }
24
25 int main() {
26
27     std::cout << std::boolalpha << '\n';
28
29     MyInt myInt1(2011);
30     MyInt myInt2(2014);
31
32     std::cout << "isLessThan(myInt1, myInt2): "
33             << isLessThan(myInt1, myInt2) << '\n';
34
35     MyDouble myDouble1(2011);
36     MyDouble myDouble2(2014);
37
38     std::cout << "isLessThan(myDouble1, myDouble2): "
39             << isLessThan(myDouble1, myDouble2) << '\n';
40
41     std::cout << '\n';
42 }

```

用户定义的(第 9 行)和编译器生成的(第 17 行)三向比较操作符如期工作。



```
C:\Windows PowerShell
C:\Users\rainer>threeWayComparison.exe

isLessThan(myInt1, myInt2): true
isLessThan(myDouble1, myDouble2): true

C:\Users\rainer>
```

这种情况下，用户定义的和编译器生成的三向比较操作符间有一些细微的区别。MyInt 的编译器推导的返回类型（第 9 行）支持强排序，MyDouble 的编译器推导的返回类型（第 17 行）支持部分排序。

比较指针

编译器生成的三向比较运算符可以比较指针，但不比较引用的对象。

```
1 // spaceshipPoiner.cpp
2
3 #include <iostream>
4 #include <compare>
5 #include <vector>
6
7 struct A {
8     std::vector<int>* pointerToVector;
9     auto operator <= > (const A&) const = default;
10 };
11
12 int main() {
13
14     std::cout << '\n';
15
16     std::cout << std::boolalpha;
17
18     A a1{new std::vector<int>()};
19     A a2{new std::vector<int>()};
20
21     std::cout << "(a1 == a2): " << (a1 == a2) << "\n\n";
22 }
```

令人惊讶的是，`a1 == a2`（第 21 行）的结果是 `false`，而不是 `true`，因为其比较了 `std::vector<int>*` 的地址。

`(a1 == a2): false`

并且，比较类别有三个。

4.3.3 比较类别

这三种比较类别的名称分别是强排序、弱排序和偏排序。对于 T 类型，以下三个属性区分三个比较类别。

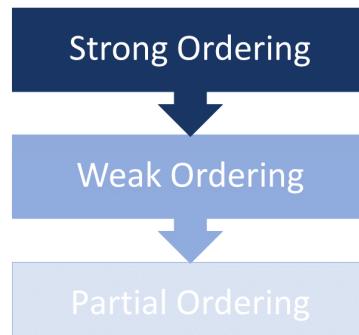
1. T 支持全部 6 种关系运算符:==、!=、<、<=、> 和 >=(简称: 关系运算符)
2. 所有相等的值都一样:(简称: 等价)
3. T 的所有值都具有可比性: 对于 T 的任意值 a 和 b, a < b、a == b 和 a > b 三个关系中的一个必须为真 (简称: 可比性)

当使用字符串的大小写不敏感表示作为排序标准时，等价值不必不同。此外，两个浮点值不需要具有可比性: 对于 a = 5.5, b = NaN(不是一个数字)，以下表达式都不返回 true: a < NaN, a == NaN, 或 a > NaN。

根据这三种性质，区分三种比较策略是很简单的:

比较类别	关系操作符	等价	可比较
强排序	yes	yes	yes
弱排序	yes		yes
偏排序	yes		

支持强排序的类型支持隐式弱排序和部分排序，这同样适用于弱排序。支持弱排序的类型也支持部分排序。其他方向则不适用。



强、弱、偏排序

若声明的返回类型是 auto，则实际返回类型是基对象和成员子对象，以及要比较的成员数组元素的公共比较类别。

关于这条规则，先来看个例子:

```
1 // strongWeakPartial.cpp
2
3 #include <compare>
4
5 struct Strong {
6     std::strong_ordering operator <=> (const Strong&) const = default;
7 };
8
9 struct Weak {
```

```

10    std::weak_ordering operator <=> (const Weak&) const = default;
11 }
12
13 struct Partial {
14     std::partial_ordering operator <=> (const Partial&) const = default;
15 };
16
17 struct StrongWeakPartial {
18
19     Strong s;
20     Weak w;
21     Partial p;
22
23     auto operator <=> (const StrongWeakPartial&) const = default;
24
25     // FINE
26     // std::partial_ordering operator <=> (const StrongWeakPartial&) const = default \
27 ;
28
29     // ERROR
30     // std::strong_ordering operator <=> (const StrongWeakPartial&) const = default; \
31
32     // std::weak_ordering operator <=> (const StrongWeakPartial&) const = default; \
33
34
35 };
36
37 int main() {
38
39     StrongWeakPartial a1, a2;
40
41     a1 < a2;
42
43 }

```

类型 StrongWeakPartial 具有支持强(第 6 行)、弱(第 10 行)和偏排序(第 14 行)的子类型。因此，StrongWeakPartial 类型(第 17 行)的常见比较类别是 std::partial_ordered。使用更强比较类别，例如强排序(第 29 行)或弱排序(第 30 行)，将导致编译时错误。

现在，我想把重点放在编译器生成的宇宙飞船操作符上。

4.3.4 编译器生成的宇宙飞船操作符

编译器生成的三向比较运算符需要头文件 <compare>，隐式地是 constexpr 和 noexcept，并执行字典顺序比较。

可以直接使用三向比较运算符。

4.3.4.1 直接使用三向比较运算符

`spaceship.cpp` 直接使用太空船操作符。

```
1 // spaceship.cpp
2
3 #include <compare>
4 #include <iostream>
5 #include <string>
6 #include <vector>
7
8 int main() {
9
10    std::cout << '\n';
11
12    int a(2011);
13    int b(2014);
14    auto res = a <= b;
15    if (res < 0) std::cout << "a < b" << '\n';
16    else if (res == 0) std::cout << "a == b" << '\n';
17    else if (res > 0) std::cout << "a > b" << '\n';
18
19    std::string str1("2014");
20    std::string str2("2011");
21    auto res2 = str1 <= str2;
22    if (res2 < 0) std::cout << "str1 < str2" << '\n';
23    else if (res2 == 0) std::cout << "str1 == str2" << '\n';
24    else if (res2 > 0) std::cout << "str1 > str2" << '\n';
25
26    std::vector<int> vec1{1, 2, 3};
27    std::vector<int> vec2{1, 2, 3};
28    auto res3 = vec1 <= vec2;
29    if (res3 < 0) std::cout << "vec1 < vec2" << '\n';
30    else if (res3 == 0) std::cout << "vec1 == vec2" << '\n';
31    else if (res3 > 0) std::cout << "vec1 > vec2" << '\n';
32
33    std::cout << '\n';
34
35 }
```

程序对 `int`(第 14 行)、`string`(第 21 行) 和 `vector`(第 28 行) 使用了太空船操作符。下面是程序的输出。

```
a < b
str1 > str2
vec1 == vec2
```

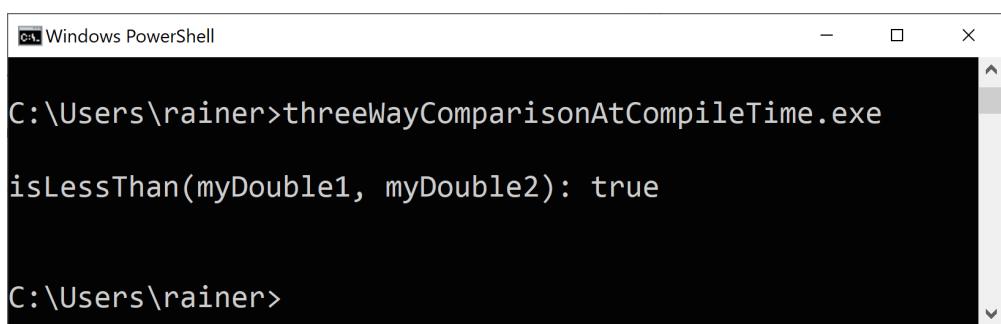
如前所述，这些比较是 `constexpr`，可以在编译时完成。

4.3.4.2 编译时的比较

三向比较操作符是隐式的 `constexpr`, 可以简化前面的程序 `threewaycompare.cpp`, 并在编译时比较下面程序中的 `MyDouble`。

```
1 // threeWayComparisonAtCompileTime.cpp
2
3 #include <compare>
4 #include <iostream>
5
6 struct MyDouble {
7     double value;
8     explicit constexpr MyDouble(double val) : value{val} { }
9     auto operator<=>(const MyDouble&) const = default;
10 };
11
12 template <typename T>
13 constexpr bool isLessThan(const T& lhs, const T& rhs) {
14     return lhs < rhs;
15 }
16
17 int main() {
18
19     std::cout << std::boolalpha << '\n';
20
21     constexpr MyDouble myDouble1(2011);
22     constexpr MyDouble myDouble2(2014);
23
24     constexpr bool res = isLessThan(myDouble1, myDouble2);
25
26     std::cout << "isLessThan(myDouble1, myDouble2): "
27     << res << '\n';
28
29     std::cout << '\n';
30
31 }
```

在编译时进行比较 (第 24 行), 并得到了比较结果。



使用 `constexpr` 编译器生成的太空船操作符

4.3.4.3 直接使用三向比较运算符

编译器生成的三向比较运算符执行字典顺序比较。本例中，字典比较是从左到右比较所有基类，以及类的所有非静态成员的声明顺序。所以，必须加以限定：出于性能原因，编译器生成的`==`和`!=`运算符在C++20中的表现不同。我将在优化的`==`和`!=`操作符一节中再来详谈。

来自Microsoft C++团队博客的帖子[“用火箭科学简化代码：C++20的宇宙飞船操作符”](#)提供了一个令人印象深刻的字典比较例子。为了可读性，我添加了一些注释。

字典序的比较

```
1 struct Basics {
2     int i;
3     char c;
4     float f;
5     double d;
6     auto operator<=>(const Basics&) const = default;
7 };
8
9 struct Arrays {
10    int ai[1];
11    char ac[2];
12    float af[3];
13    double ad[2][2];
14    auto operator<=>(const Arrays&) const = default;
15 };
16
17 struct Bases : Basics, Arrays {
18     auto operator<=>(const Bases&) const = default;
19 };
20
21 int main() {
22     constexpr Bases a = { { 0, 'c', 1.f, 1. }, // Basics
23                           { { 1 }, { 'a', 'b' }, { 1.f, 2.f, 3.f } }, // Arrays
24                           { { 1., 2. }, { 3., 4. } } };
25     constexpr Bases b = { { 0, 'c', 1.f, 1. }, // Basics
26                           { { 1 }, { 'a', 'b' }, { 1.f, 2.f, 3.f } }, // Arrays
27                           { { 1., 2. }, { 3., 4. } } };
28     static_assert(a == b);
29     static_assert(!(a != b));
30     static_assert(!(a < b));
31     static_assert(a <= b);
32     static_assert(!(a > b));
33     static_assert(a >= b);
34 }
```

我认为这个程序最具挑战性的不是宇宙飞船操作符，而是通过聚合初始化来初始化Bases(第22和25行)。当成员都是public时，聚合初始化使我们能够直接初始化类类型(class、struct、union)的成员，所以可以使用大括号初始化。聚合初始化将在C++20中指定初始化器一节中更详细地讨论。

优化的 == 和!= 操作符

对于类似字符串或类 vector 类型，存在优化潜力。这种情况下，== 和!= 可能比编译器生成的三向比较运算符更快。若比较的两个值的长度不同，== 和!= 操作符可以停止。否则，若一个值是另一个值的前缀，则将比较所有元素，直到较短的值结束为止。标准化委员会意识到了这个性能问题，并通过[P1185R2](#)解决了这个问题。因此，编译器生成的 == 和!= 操作符(对于类字符串或类 vector 类型)首先会比较其长度，必要时再比较其内容。

现在，是时候介绍 C++ 中的一些新东西了。C++20 引入了重写表达式的概念。

4.3.5 重写表达式

当编译器看到诸如 $a < b$ 之类的东西时，会使用太空船操作符将其重写为 $(a \leq b) < 0$ 。

当然，该规则适用于所有六个比较运算符: $a \text{ OP } b$ 变成 $(a \leq b) \text{ OP } 0$ 。若没有类型 (a) 到类型 (b) 的转换，编译器生成新的表达式为 $0 \text{ OP } (b \leq a)$ 。

例如，对于小于操作符，若 $(a \leq b) < 0$ 不起作用，编译器会生成 $0 < (b \leq a)$ 。本质上，编译器负责比较操作符的对称性。

下面是一些重写表达式的例子:

使用重写表达式的 MyInt

```
1 // rewritingExpressions.cpp
2
3 #include <compare>
4 #include <iostream>
5
6 class MyInt {
7 public:
8     constexpr MyInt(int val) : value{val} { }
9     auto operator<=(const MyInt& rhs) const = default;
10 private:
11     int value;
12 };
13
14 int main() {
15
16     std::cout << '\n';
17
18     constexpr MyInt myInt2011(2011);
19     constexpr MyInt myInt2014(2014);
20
21     constexpr int int2011(2011);
22     constexpr int int2014(2014);
23
24     if (myInt2011 < myInt2014) std::cout << "myInt2011 < myInt2014" << '\n';
25     if ((myInt2011 <= myInt2014) < 0) std::cout << "myInt2011 < myInt2014" << '\n';
26 }
```

```

27     std::cout << '\n';
28
29     if (myInt2011 < int2014) std:: cout << "myInt2011 < int2014" << '\n';
30     if ((myInt2011 <= int2014) < 0) std:: cout << "myInt2011 < int2014" << '\n';
31
32     std::cout << '\n';
33
34     if (int2011 < myInt2014) std::cout << "int2011 < myInt2014" << '\n';
35     if (0 < (myInt2014 <= int2011)) std:: cout << "int2011 < myInt2014" << '\n';
36
37     std::cout << '\n';
38
39 }

```

第 24 行、第 29 行和第 34 行中使用了小于操作符和相应的太空船表达式。第 35 行比较 (`int2011 < myInt2014`)，会生成飞船表达式 (`0 < (myInt2014 <= int2011)`)。

```

myInt2011 < myInt2014
myInt2011 < myInt2014

myInt2011 < int2014
myInt2011 < int2014

int2011 < myInt2014
int2011 < myInt2014

```

`MyInt` 有一个问题：构造函数接受一个参数时，应该声明为 `explicit`。接受一个参数的构造函数，如 `MyInt(int val)`(第 8 行) 是转换构造函数。所以 `MyInt` 的实例可以由任意整数或浮点值生成，因为每个整数或浮点值都可以隐式地转换为 `int`。

为了修复这个问题，需要使构造函数 `MyInt(int val)` 显式化。为了支持 `MyInt` 和 `int` 的比较，`MyInt` 需要用于 `int` 的三向比较操作符。

int 的另一个三向比较运算符

```

1 // threeWayComparisonForInt.cpp
2
3 #include <compare>
4 #include <iostream>
5
6 class MyInt {
7 public:
8     constexpr explicit MyInt(int val): value{val} { }
9
10    auto operator<=>(const MyInt& rhs) const = default;
11

```

```

12 constexpr auto operator<=>(const int& rhs) const {
13     return value <=> rhs;
14 }
15 private:
16     int value;
17 };
18
19 template <typename T, typename T2>
20 constexpr bool isLessThan(const T& lhs, const T2& rhs) {
21     return lhs < rhs;
22 }
23
24 int main() {
25
26     std::cout << std::boolalpha << '\n';
27
28     constexpr MyInt myInt2011(2011);
29     constexpr MyInt myInt2014(2014);
30
31     constexpr int int2011(2011);
32     constexpr int int2014(2014);
33
34     std::cout << "isLessThan(myInt2011, myInt2014): "
35     << isLessThan(myInt2011, myInt2014) << '\n';
36
37     std::cout << "isLessThan(int2011, myInt2014): "
38     << isLessThan(int2011, myInt2014) << '\n';
39
40     std::cout << "isLessThan(myInt2011, int2014): "
41     << isLessThan(myInt2011, int2014) << '\n';
42
43     constexpr auto res = isLessThan(myInt2011, int2014);
44
45     std::cout << '\n';
46 }
47 }
```

我在(第 10 行)中定义了三向比较运算符，并将其声明为 `constexpr`。用户定义的三向比较操作符不是隐式的 `constexpr`，这与编译器生成的三向比较操作符不同。每个组合中都可以比较 `MyInt` 和 `int`(第 34、37 和 40 行)。

```

isLessThan(MyInt2011, myInt2014): true
isLessThan(int2011, myInt2014): true
isLessThan(MyInt2011, int2014): true
```

各种三向比较运算符的实现非常优雅。编译器自动生成 `MyInt` 的比较，并且用户显式地定义了

与 int 的比较。此外，由于重新排序，需定义 2 个操作符就可以得到 $18 = 3 * 6$ 个比较操作符的组合。3 代表 int OP MyInt, MyInt OP MyInt 和 MyInt OP int 的组合，6 代表六个比较运算符。

4.3.6 用户定义和自动生成的比较操作符

可以定义六个比较操作符中的一个，并使用太空船操作符自动生成所有比较操作符时，有一个问题：哪个优先级更高？例如，MyInt 有一个用户定义的小于等于运算符，还有编译器生成的 6 个比较运算符。

看看会发生什么。

用户定义的操作符和自动生成的操作符间的互动

```
1 // userDefinedAutoGeneratedOperators.cpp
2
3 #include <compare>
4 #include <iostream>
5
6 class MyInt {
7 public:
8     constexpr explicit MyInt(int val) : value{val} { }
9     bool operator == (const MyInt& rhs) const {
10         std::cout << "==" << '\n';
11         return value == rhs.value;
12     }
13     bool operator < (const MyInt& rhs) const {
14         std::cout << "<" << '\n';
15         return value < rhs.value;
16     }
17     auto operator<=>(const MyInt& rhs) const = default;
18
19 private:
20     int value;
21 };
22
23 int main() {
24
25     MyInt myInt2011(2011);
26     MyInt myInt2014(2014);
27
28     myInt2011 == myInt2014;
29     myInt2011 != myInt2014;
30     myInt2011 < myInt2014;
31     myInt2011 <= myInt2014;
32     myInt2011 > myInt2014;
33     myInt2011 >= myInt2014;
34
35 }
```

为了查看用户定义的`==`和`<`操作符的作用，我将相应的消息写入`std::cout`。因为`std::cout`是一个运行时操作，所以这两个操作符都不能是`constexpr`。

看看会发生什么：

```
==  
==  
<
```

编译器使用用户定义的`==(第 29 和 30 行)`和`<`操作符(第 31 行)。此外，编译器使用`==`运算符生成了`!=`运算符(第 30 行)，而编译器不会使用`!=`操作符生成`==`操作符。

与 Python 的相似性

Python 3 中，编译器在必要时从`==`中生成`!=`，而不是反过来。Python 2 中，所谓的富比较(用户自定义的六个比较操作符)的优先级高于 Python 的三向比较操作符`__cmp__`。因为三向比较运算符`__cmp__`在 Python 3 中删除了，所以还是更像 Python 2。

总结

- 通过默认操作符`<=`，编译器自动生成 6 个比较操作符。编译器生成的比较运算符应用字典顺序的比较：所有基类从左到右比较，类的所有非静态成员按声明顺序比较。
- 当自动生成的比较操作符和用户定义的比较操作符同时存在时，用户定义的比较操作符具有更高的优先级。
- 编译器重写表达式以保证比较操作符的对称性。例如，若`(a <= b) < 0`不起作用，则编译器会生成`0 < (b <= a)`。

4.4. 指定初始化



Cippi 接受了神圣的触摸

指定初始化是聚合初始化的特例。

4.4.1 聚合初始化

首先: 什么是聚合? 聚合是数组和类型。类型可以是一个类、一个结构体或一个联合体。

C++20 中, 支持聚合初始化的类型必须满足以下条件:

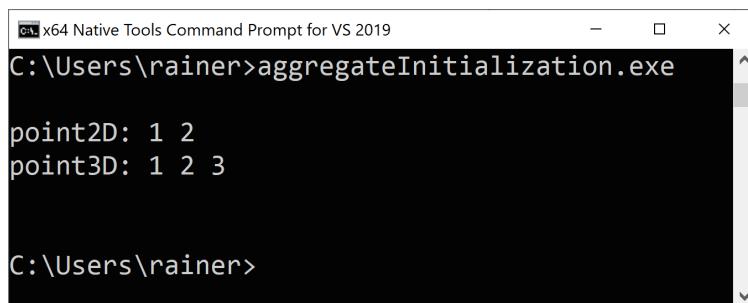
- 没有 `private` 或 `protected` 的非静态数据成员
- 没有用户声明的或继承的构造函数
- 没有虚基类、`private` 基类或 `protected` 基类
- 没有虚成员函数

下面的程序演示了聚合初始化。

```
1 // aggregateInitialization.cpp
2
3 #include <iostream>
4
5 struct Point2D{
6     int x;
7     int y;
8 };
9
10 class Point3D{
11 public:
12     int x;
13     int y;
14     int z;
15 };
16
17 int main(){
18
19     std::cout << '\n';
20
21     Point2D point2D{1, 2};
22     Point3D point3D{1, 2, 3};
```

```
23
24     std::cout << "point2D: " << point2D.x << " " << point2D.y << '\n';
25     std::cout << "point3D: " << point3D.x << " " << point3D.y << " "
26             << point3D.z << '\n';
27
28     std::cout << '\n';
29
30 }
```

第 21 和 22 行使用大括号直接初始化聚合，大括号中的初始化式的顺序必须与成员的声明顺序匹配。介绍三向比较操作符的部分中，有挺复杂的聚合初始化示例。



基于 C++11 中的聚合初始化，C++20 出现了设计好的初始化器。到 2020 年底，只有 Microsoft 编译器完全支持指定初始化。

4.4.2 类成员的命名初始化

指定初始化允许使用类型成员名直接初始化。对于联合体，只能提供一个初始化式。对于聚合初始化，大括号中的初始化式的顺序必须与成员的声明顺序匹配。

```
1 // designatedInitializer.cpp
2
3 #include <iostream>
4
5 struct Point2D{
6     int x;
7     int y;
8 };
9
10 class Point3D{
11 public:
12     int x;
13     int y;
14     int z;
15 };
16
17 int main(){
18
19     std::cout << '\n';
20 }
```

```

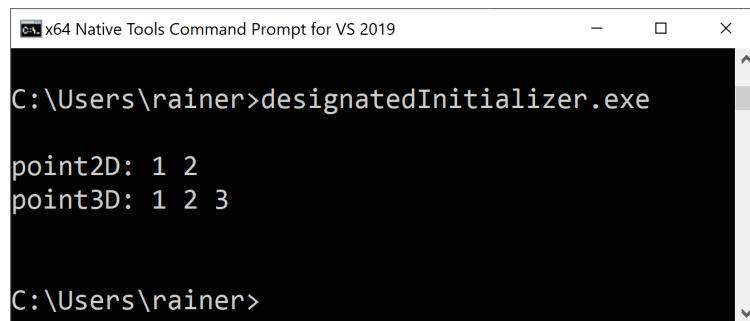
21 Point2D point2D{.x = 1, .y = 2};
22 Point3D point3D{.x = 1, .y = 2, .z = 3};

23 std::cout << "point2D: " << point2D.x << " " << point2D.y << '\n';
24 std::cout << "point3D: " << point3D.x << " " << point3D.y << " "
25             << point3D.z << '\n';

26 std::cout << '\n';
27
28 }

```

第 21 和 22 行使用指定的初始化器来初始化聚合，像.x 或.y 这样的初始化式称为指示符。



聚合成员可以已经有默认值，当缺少初始化式时使用此默认值，但这并不适用于联合体。

```

1 // designatedInitializersDefaults.cpp
2
3 #include <iostream>
4
5 class Point3D{
6 public:
7     int x;
8     int y = 1;
9     int z = 2;
10 };
11
12 void needPoint(Point3D p) {
13     std::cout << "p: " << p.x << " " << p.y << " " << p.z << '\n';
14 }
15
16 int main() {
17
18     std::cout << '\n';
19
20     Point3D point1{.x = 0, .y = 1, .z = 2}
21
22     std::cout << "point1: " << point3D.x << " " << point3D.y << " "
23             << point3D.z << '\n';
24
25     Point3D point2;
26     std::cout << "point2: " << point2.x << " " << point2.y << " "

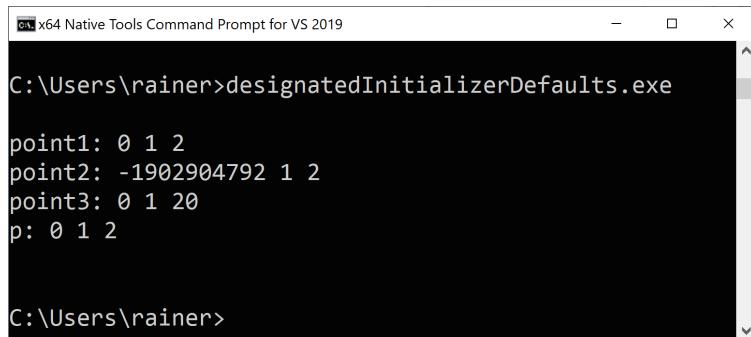
```

```

27     << point2.z << '\n';
28
29     Point3D point3{.x = 0, .z = 20};
30     std::cout << "point3: " << point3.x << " " << point3.y << " "
31             << point3.z << '\n';
32
33 // Point3D point4{.z = 20, .y = 1}; ERROR
34
35 needPoint({.x = 0});
36
37 std::cout << '\n';
38
39 }

```

第 20 行初始化了所有成员，但第 24 行没有为成员 x 提供值，x 没有初始化。若只初始化没有默认值的成员，比如在第 28 行或第 34 行中没问题，因为 z 和 y 的顺序是错误的，第 32 行中的表达式则无法编译。



指定初始化器检测窄化转换，导致精度降低。

```

1 // designatedInitializerNarrowingConversion.cpp
2
3 #include <iostream>
4
5 struct Point2D{
6     int x;
7     int y;
8 };
9
10 class Point3D{
11 public:
12     int x;
13     int y;
14     int z;
15 };
16
17 int main(){
18
19     std::cout << '\n';
20

```

```

21 Point2D point2D{.x = 1, .y = 2.5};
22 Point3D point3D{.x = 1, .y = 2, .z = 3.5f};

23 std::cout << "point2D: " << point2D.x << " " << point2D.y << '\n';
24 std::cout << "point3D: " << point3D.x << " " << point3D.y << " "
25             << point3D.z << '\n';

26 std::cout << '\n';
27
28 }

```

因为初始化.y=2.5 和.z=3.5f 会窄化为 int，所以第 21 行和第 22 行会产生编译时错误。

```

C:\Users\rainer>cl.exe /std:c++latest designatedInitializerNarrowingConversion.cpp /EHsc
Microsoft (R) C/C++ Optimizing Compiler Version 19.26.28806 for x64
Copyright (C) Microsoft Corporation. All rights reserved.

/std:c++latest is provided as a preview of language features from the latest C++
working draft, and we're eager to hear about bugs and suggestions for improvements.
However, note that these features are provided as-is without support, and subject
to changes or removal as the working draft evolves. See
https://go.microsoft.com/fwlink/?linkid=2045807 for details.

designatedInitializerNarrowingConversion.cpp
designatedInitializerNarrowingConversion.cpp(19): error C2397: conversion from 'double' to 'int' requires a narrowing conversion
designatedInitializerNarrowingConversion.cpp(20): error C2397: conversion from 'float' to 'int' requires a narrowing conversion

C:\Users\rainer>

```

有趣的是，指定初始化式在 C 和 C++ 的行为不太一样。

C 和 C++ 的区别

C 指定的初始化器支持 C++ 中不支持的用例。

C 允许：

- 乱序初始化聚合的成员
- 初始化嵌套聚合的成员
- 混合指定初始化式和常规初始化式
- 数组的指定初始化

提案[P0329R4](#)为这些用例提供了示例：

```

1 struct A { int x, y; };
2 struct B { struct A a; };
3 struct A a = {.y = 1, .x = 2}; // valid C, invalid C++ (out of order)
4 int arr[3] = {[1] = 5}; // valid C, invalid C++ (array)
5 struct B b = {.a.x = 0}; // valid C, invalid C++ (nested)
6 struct A a = {.x = 1, 2}; // valid C, invalid C++ (mixed)

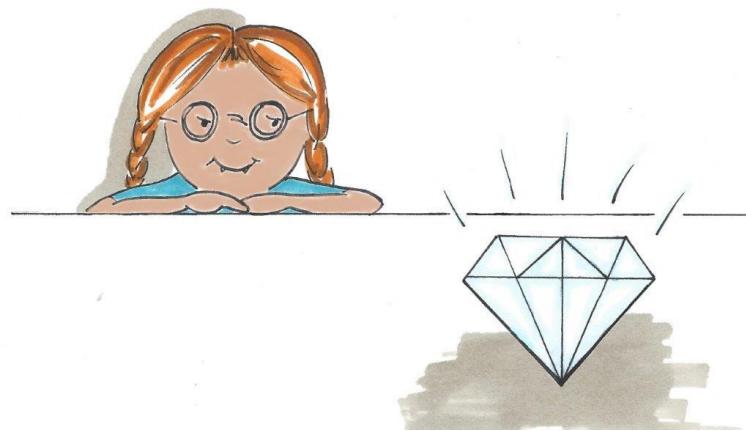
```

C 和 C++ 这种差异的原理，也是提案的一部分：“C++ 成员以反向构造顺序销毁，初始化式列表的元素以词法顺序求值，因此字段初始化式必须按顺序指定。数组指示符与 Lambda 表达式语法冲突。嵌套的指示符则很少使用。” 提案还认为，只有乱序初始化聚合会经常使用。

总结

指定初始化是聚合初始化的一种特殊情况，使其能够使用类成员名初始化类成员。初始化顺序必须与声明顺序匹配。

4.5. consteval 和 constinit



Cippi 在欣赏钻石

C++20 添加了两个新的关键字:consteval 和 constinit。关键字 consteval 生成一个在编译时执行的函数, constinit 保证变量在编译时初始化, 可能会觉得这两个说明符都与 constexpr 相似。在比较关键字 consteval、constinit、constexpr 和 const 之前, 先来介绍一下 consteval 和 constinit。

4.5.1 consteval

consteval 会创建了一个“立即函数”。

```
1 consteval int sqr(int n) {  
2     return n * n;  
3 }
```

每次调用立即函数都会创建一个编译时常数, consteval(立即) 函数在编译时执行。

consteval 不能应用于析构函数, 或具有分配或释放内存的函数。声明中最多只能使用 consteval、constexpr 或 constinit 说明符中的一种。立即函数 (consteval) 是隐式内联的, 必须满足 constexpr 函数的要求。

C++14 中 constexpr 函数, 以及 consteval 函数的要求:

- consteval(constexpr) 函数可以有
 - 条件跳转指令或循环指令。
 - 多条指令。
 - 调用 constexpr 函数。consteval 函数只能调用 constexpr 函数, 反之则不行。
 - 必须使用基本数据类型作为常量表达式初始化的变量。
- consteval (constexpr) 函数不能有
 - 静态或 thread_local 数据。
 - try 块或 goto。
 - 使用或使用非 consteval 函数。
 - 使用或使用非 constexpr 数据。

简而言之:consteval 函数的所有依赖项都必须在编译时可解析。

constevalSqr.cpp 使用了 consteval 函数 sqr。

```
1 // constevalSqr.cpp
2
3 #include <iostream>
4
5 consteval int sqr(int n) {
6     return n * n;
7 }
8
9 int main() {
10
11     std::cout << "sqr(5): " << sqr(5) << '\n';
12
13     const int a = 5;
14     std::cout << "sqr(a): " << sqr(a) << '\n';
15
16     int b = 5;
17     // std::cout << "sqr(b): " << sqr(b) << '\n'; ERROR
18
19 }
```

数字 5 是一个常量表达式，可以用作函数 sqr 的参数(第 11 行)，变量 a 也是如此(第 13 行)。常量变量(如 a)在用常量表达式初始化时，可在常量表达式中使用。变量 b(第 16 行)不是常量表达式。因此，sqr(b)(第 17 行)的调用无效。

下面是程序的输出:

```
sqr(5): 25
sqr(a): 25
```

4.5.2 constinit

constinit 可以应用于具有静态或线程存储的变量。

- 全局(命名空间)变量、静态变量或静态类成员具有静态存储的变量。这些对象在程序启动时分配，在程序结束时释放。
- thread_local 变量具有线程存储的变量。为使用此数据的每个线程创建线程本地数据。thread_local 数据专属于线程，在第一次使用时创建，其生存期与所属线程的生存期绑定。线程本地数据通常称为线程局部变量。

constinit 在编译时，可确保对这类变量(静态存储或线程存储的变量)进行初始化，但 constinit 并不意味着常量性(consteness)。

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

```

4
5 consteval int sqr(int n) {
6     return n * n;
7 }
8
9 constexpr auto res1 = sqr(5);
10 constinit auto res2 = sqr(5);
11
12 int main() {
13     std::cout << "sqr(5): " << res1 << '\n';
14     std::cout << "sqr(5): " << res2 << '\n';
15
16     constinit thread_local auto res3 = sqr(5);
17     std::cout << "sqr(5): " << res3 << '\n';
18 }
```

res1 和 res2 是具有静态存储的变量，res3 具有线程存储的变量。

```
sqr(5): 25
sqr(5): 25
sqr(5): 25
```

现在，来了解一下 const、constexpr、consteval 和 constinit 的区别。

首先，来讨论一下函数执行和变量的初始化。

4.5.3 函数执行

consteval.cpp 有三个版本的 square 函数。

```

1 // consteval.cpp
2
3 #include <iostream>
4
5 int sqrRunTime(int n) {
6     return n * n;
7 }
8
9 consteval int sqrCompileTime(int n) {
10    return n * n;
11 }
12
13 constexpr int sqrRunOrCompileTime(int n) {
14    return n * n;
15 }
16
17 int main() {
```

```

19 // constexpr int prod1 = sqrRunTime(100); ERROR
20 constexpr int prod2 = sqrCompileTime(100);
21 constexpr int prod3 = sqrRunOrCompileTime(100);
22
23 int x = 100;
24
25 int prod4 = sqrRunTime(x);
26 // int prod5 = sqrCompileTime(x); ERROR
27 int prod6 = sqrRunOrCompileTime(x);
28 }

```

普通函数 `sqrRunTime`(第 5 行) 在运行时运行, `constexpr` 函数 `sqrCompileTime` 在编译时运行(第 9 行), `constexpr` 函数 `sqrRunOrCompileTime` 可以在编译时或运行时运行。在编译时使用 `sqrRunTime`(第 19 行) 是一个错误, 所以使用非常量表达式作为 `sqrCompileTime`(第 26 行) 的参数也是一个错误。

`constexpr` 函数 `sqrRunOrCompileTime` 和 `constexpr` 函数 `sqrCompileTime` 之间的区别在于, 当上下文需要编译时求值时, `sqrRunOrCompileTime` 必须在编译时执行。

编译时和运行时执行

```

1 static_assert(sqrRunOrCompileTime(10) == 100); // compile time
2 int arrayNewWithConstExpressionFunction[sqrRunOrCompileTime(100)]; // compile time
3 constexpr int prod = sqrRunOrCompileTime(100); // compile time
4
5 int a = 100;
6 int runTime = sqrRunOrCompileTime(a); // run time
7
8 int runTimeOrCompiletime = sqrRunOrCompileTime(100); // run time or compile time
9
10 int alwaysCompileTime = sqrCompileTime(100); // compile time

```

第 1-3 行需要编译时计算。因为 `a` 不是常量表达式, 所以第 6 行只能在运行时求值。关键的是第 8 行, 函数可以在编译时或运行时执行, 它在编译时执行还是在运行时执行, 可能取决于编译器或优化级别。这一观察结果不适用于第 10 行, 因为 `constexpr` 函数总是在编译时执行。

4.5.4 变量的初始化

`constexprConstinit.cpp` 中比较一下 `const`、`constexpr` 和 `constinit`。

```

1 // constexprConstinit.cpp
2
3 #include <iostream>
4
5 constexpr int constexprVal = 1000;
6 constinit int constinitVal = 1000;
7
8 int incrementMe(int val) { return ++val; }
9
10 int main() {
11

```

```

12 auto val = 1000;
13 const auto res = incrementMe(val);
14 std::cout << "res: " << res << '\n';
15
16 // std::cout << "res: " << ++res << '\n'; ERROR
17 // std::cout << "++constexprVal: " << ++constexprVal << '\n'; ERROR
18 std::cout << "++constinitVal: " << ++constinitVal << '\n';
19
20 constexpr auto localConstexpr = 1000;
21 // constinit auto localConstinit = 1000; ERROR
22
23 }

```

只有 `const` 变量(第 13 行)在运行时初始化, `constexpr` 和 `constinit` 变量在编译时初始化。

`constinit`(第 18 行)并不意味着常量化,就像 `const`(第 16 行)或 `constexpr`(第 17 行)一样。声明了 `constexpr`(第 20 行)或 `const`(第 13 行)的变量可以创建为局部变量,但不能创建声明了 `constinit` 变量(第 21 行)。

```

res: 1001
++constinitVal: 1001

```

4.5.5 解决静态初始化顺序错误

根据[isocpp.org 的 FAQ](http://isocpp.org/faq), 静态初始化顺序错误是“一种导致程序崩溃的微妙方式”。问题解答写道:“C++ 静态初始化顺序是一个非常微妙,且经常被误解。”

在继续之前,我想做一个简短的免责声明。不同的翻译单元中,对具有静态存储(短静态)的变量的依赖通常是一种代码腐味,是需要重构的信号。因此,若愿意按照我的建议进行重构,可以跳过本节。

4.5.5.1 静态初始化顺序失败

翻译单元中的静态变量是根据定义顺序进行初始化的。

相反,在翻译单元之间初始化静态变量有一个严重的问题。当在一个翻译单元中定义了一个静态变量 `staticA`,而在另一个翻译单元中定义了另一个静态变量 `staticB`,并且 `staticB` 需要 `staticA` 来初始化自己时,这就会出现静态初始化顺序失败。程序呈现病态,因为不能保证哪个静态变量在(动态)运行时首先初始化。

介绍解决方案之前,来展示一下静态初始化顺序错误的例程。

4.5.5.1.1 对与错, 五五开

静态函数的初始化有什么独特之处?静态函数的初始化顺序分为两个步骤:静态和动态。

当一个静态对象不能在编译时进行常量初始化时,是不初始化的。在运行时,可以对这些不初始化的静态数据进行动态初始化。

```
1 // sourceSIOF1.cpp
2 int square(int n) {
3     return n * n;
4 }
5 auto staticA = square(5);
```

```
1 // mainSIOF1.cpp
2
3 #include <iostream>
4
5 extern int staticA;
6 auto staticB = staticA;
7
8 int main() {
9
10    std::cout << '\n';
11
12    std::cout << "staticB: " << staticB << '\n';
13
14    std::cout << '\n';
15
16 }
```

第 5 行声明静态变量 staticA，staticB 的初始化依赖于 staticA 的初始化。staticB 在编译时未初始化，在运行时动态初始化。因为 staticA 和 staticB 属于不同的翻译单元，所以问题在于不能保证初始化 staticA 或 staticB 的顺序。因此，staticB 为 0 或 25 的概率各为 50%。

为了演示这个问题，我可以改变目标文件的链接顺序。这也就改变了 staticB 的值！

```
rainer@seminar:~> g++ -c mainSIOF1.cpp
rainer@seminar:~> g++ -c sourceSIOF1.cpp
rainer@seminar:~> g++ mainSIOF1.o sourceSIOF1.o -o mainSource
rainer@seminar:~> g++ sourceSIOF1.o mainSIOF1.o -o sourceMain
rainer@seminar:~> mainSource

staticB: 0

rainer@seminar:~> sourceMain

staticB: 25

rainer@seminar:~>
```

实践静态初始化顺序错误

失败了吧！可执行文件的结果取决于目标文件的链接顺序。当没有 C++20 时，如何解决这个问题呢？

4.5.5.1.2 惰性初始化——局部作用域的静态对象

局部作用域的静态变量是在首次使用时创建的。局部作用域本质上是静态变量，以某种方式被大括号包围。这种惰性创建是 C++98 提供的保证。在 C++11 中，具有局部作用域的静态变量也以线程安全的方式初始化。线程安全的Meyers单例就是基于这个保证。

惰性初始化也可以用来避免静态初始化顺序错误。

```
1 // sourceSIOF2.cpp
2
3 int square(int n) {
4     return n * n;
5 }
6
7 int& staticA() {
8
9     static auto staticA = square(5);
10    return staticA;
11 }
12 }
```

```
1 // mainSIOF2.cpp
2
3 #include <iostream>
4
5 int& staticA();
6
7 auto staticB = staticA();
8
9 int main() {
10
11     std::cout << '\n';
12
13     std::cout << "staticB: " << staticB << '\n';
14
15     std::cout << '\n';
16 }
17 }
```

在本例中，`staticA`(`sourceSIOF2.cpp` 文件中的第 9 行) 是局部作用域中的静态函数。`mainSIOF2.cpp` 文件中的第 5 行声明了函数 `staticA`，该函数用于在下面的行 `staticB` 中初始化。这个 `staticA` 的局部作用域保证在运行时第一次使用 `staticA` 时创建并初始化它，从而改变链接顺序，但不会改变 `staticB` 的值。



```
rainer@seminar:~> g++ -c mainSIOF2.cpp
rainer@seminar:~> g++ -c sourceSIOF2.cpp
rainer@seminar:~> g++ mainSIOF2.o sourceSIOF2.o -o mainSource
rainer@seminar:~> g++ sourceSIOF2.o mainSIOF2.o -o sourceMain
rainer@seminar:~> mainSource

staticB: 25

rainer@seminar:~> sourceMain

staticB: 25

rainer@seminar:~>
```

使用本地静态变量解决静态初始化顺序问题

最后，我使用 C++20 解决了静态初始化顺序的问题。

4.5.5.1.3 静态对象的编译时初始化

对 staticA 应用 constinit，保证在编译时对 staticA 进行初始化。

```
// sourceSIOF3.cpp

constexpr int square(int n) {
    return n * n;
}

constinit auto staticA = square(5);
```

```
// mainSIOF3.cpp

#include <iostream>

extern constinit int staticA;

auto staticB = staticA;

int main() {

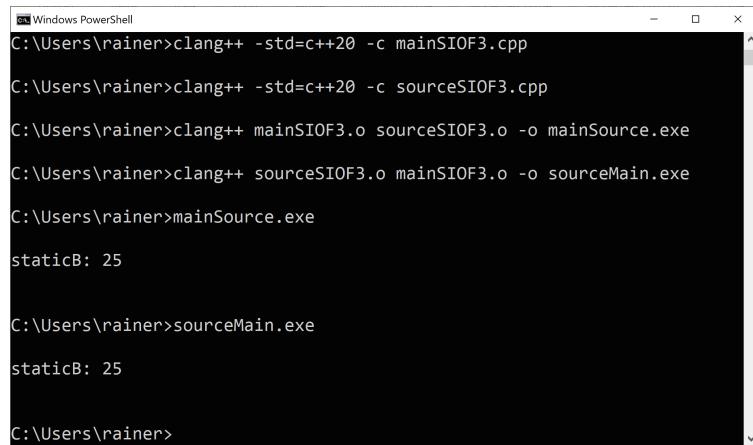
    std::cout << '\n';

    std::cout << "staticB: " << staticB << '\n';

    std::cout << '\n';

}
```

mainSIOF3.cpp 文件中的第 5 行声明变量 staticA，该变量在编译时初始化 (sourceSIOF3.cpp 文件中的第 7 行)。因为 `constexpr` 需要定义 (不仅是声明)，所以这里使用 `constexpr`(mainSIOF3.cpp 文件中的第 5 行) 是无效的。



```
C:\Users\rainer>clang++ -std=c++20 -c mainSIOF3.cpp
C:\Users\rainer>clang++ -std=c++20 -c sourceSIOF3.cpp
C:\Users\rainer>clang++ mainSIOF3.o sourceSIOF3.o -o mainSource.exe
C:\Users\rainer>clang++ sourceSIOF3.o mainSIOF3.o -o sourceMain.exe
C:\Users\rainer>mainSource.exe
staticB: 25

C:\Users\rainer>sourceMain.exe
staticB: 25

C:\Users\rainer>
```

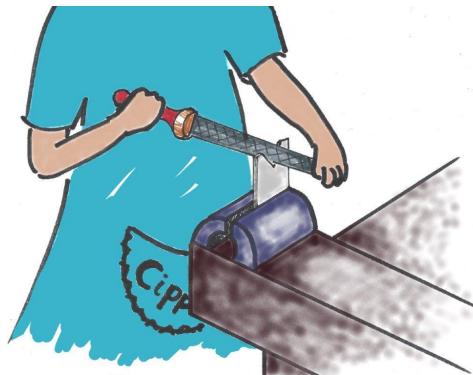
用 `constinit` 解决静态初始化顺序错误

与使用本地静态变量进行延迟初始化一样，`staticB` 的值恒为 25。

总结

- C++20 新增了两个新的关键字:`consteval` 和 `constinit`。`consteval` 生成一个在编译时执行的函数，`constinit` 保证变量在编译时初始化。
- 与 C++11 中的 `constexpr` 相比，`consteval` 保证函数在编译时执行。
- `const`、`constexpr` 和 `constinit` 之间是有区别的。`const` 和 `constexpr` 创建常量变量。`constexpr` 和 `constinit` 在编译时执行。

4.6. 模板的改进



Cippi 正在使用她的新工具

C++20 的改进使模板更加一致，在编写泛型程序时更不容易出错。

4.6.1 条件显式构造函数

有时需要一个类，需要有接受不同类型的构造函数。例如，一个 VariantWrapper 类，包含接受各种类型的 std::variant。

包含 std::variant 的 VariantWrapper 类

```
1 class VariantWrapper {
2     std::variant<bool, char, int, double, float, std::string> myVariant;
3 };
```

可以使用 bool、char、int、double、float 或 std::string 初始化 VariantWrapper，VariantWrapper 类需要为每个列出的类型提供相应构造函数。懒惰是一种美德——至少对程序员来说是这样——因此，可以使用泛型构造函数。

Implicit 类展示了一个泛型构造函数。

泛型构造函数

```
1 // implicitExplicitGenericConstructor.cpp
2
3 #include <iostream>
4 #include <string>
5
6 struct Implicit {
7     template <typename T>
8     Implicit(T t) {
9         std::cout << t << '\n';
10    }
11 };
12
13 struct Explicit {
14     template <typename T>
15     explicit Explicit(T t) {
```

```
16     std::cout << t << '\n';
17 }
18 };
19
20 int main() {
21
22     std::cout << '\n';
23
24     Implicit imp1 = "implicit";
25     Implicit imp2("explicit");
26     Implicit imp3 = 1998;
27     Implicit imp4(1998);
28
29     std::cout << '\n';
30
31 // Explicit exp1 = "implicit";
32 Explicit exp2{"explicit"};
33 // Explicit exp3 = 2011;
34 Explicit exp4{2011};
35
36     std::cout << '\n';
37 }
38 }
```

现在，问题来了。泛型构造函数(第7行)是一个全能构造函数，可以接受任何类型。但构造函数太贪婪了，可以通过在构造函数(第14行)前面放置 explicit 关键字，避免隐式转换(第31和33行)，只有显式调用(第32和34行)的方式是有效的。

```
implicit
implicit
1998
1998

explicit
2011
```

隐式和现实泛型构造函数

C++20 中，explicit 更有用。假设有一个 MyBool 类型，只支持从 bool 类型的隐式转换，而不支持其他隐式转换。这种情况下，explicit 可以有条件地使用。

允许从 **bool** 类型进行隐式转换的泛型构造函数

```
1 // conditionallyConstructor.cpp
2
```

```

3 #include <iostream>
4 #include <type_traits>
5 #include <typeinfo>
6
7 struct MyBool {
8     template <typename T>
9     explicit(!std::is_same<T, bool>::value) MyBool(T t) {
10         std::cout << typeid(t).name() << '\n';
11     }
12 };
13
14 void needBool(MyBool b) { }
15
16 int main() {
17
18     MyBool myBool1(true);
19     MyBool myBool2 = false;
20
21     needBool(myBool1);
22     needBool(true);
23     // needBool(5);
24     // needBool("true");
25
26 }

```

显式 (!std::is_same<T, bool>::value) 表达式保证 MyBool 只能由布尔值隐式创建。函数 std::is_same 是来自[类型特性库](#)的编译时谓词。因为是编译时谓词，所以 std::is_same 可在编译时进行计算，并返回一个布尔值。因此，可以对布尔类型进行隐式转换(第 19 和 22 行)，但不能从 int 和 C-string 类型进行转换(第 23 和 24 行，注释行)。

4.6.2 非类型模板参数

C++ 支持非类型作为模板参数。本质上非类型可能是

- 整数和枚举数
- 指向对象、函数和类属性的指针或引用
- std::nullptr_t

典型的非类型模板参数

当我问我班上的学生是否使用过非类型作为模板形参时，他们说：没有！

那只能由我回答这个棘手的问题了，并展示一个使用的非类型模板参数示例：

定义 std::array

```
1 std::array<int, 5> myVec;
```

常量 5 是一个非类型的模板参数。

从 C++98 开始, C++ 社区就一直在讨论支持浮点模板形参。现在, C++20 支持浮点数、文字类型和字符串文字作为非类型。

4.6.2.1 浮点和文字类型

文字类型有以下两个属性:

- 所有基类和非静态数据成员都是 public, 且不可变
- 所有基类和非静态数据成员的类型都是结构类型或数组

文字类型必须具有 `constexpr` 构造函数。下面的程序使用浮点类型和文字类型作为非类型模板参数:

使用浮点和文字类型作为非类型模板参数

```
1 // nonTypeTemplateParameter.cpp
2
3 struct ClassType {
4     constexpr ClassType(int) {}
5 };
6
7 template <ClassType cl>
8 auto getClassType() {
9     return cl;
10 }
11
12 template <double d>
13 auto getDouble() {
14     return d;
15 }
16
17 int main() {
18
19     auto c1 = getClassType<ClassType(2020)>();
20
21     auto d1 = getDouble<5.5>();
22     auto d2 = getDouble<6.5>();
23 }
```

`ClassType` 有一个 `constexpr` 构造函数 (第 4 行), 可以用作模板参数 (第 19 行)。函数模板 `getDouble`(第 13 行) 也是如此, 只能接受 `double` 类型。我想强调的是, 每次调用函数模板 `getDouble`(第 21 和 22 行) 都会创建一个新函数 `getDouble`, 这个函数是给定 `double` 值的全特化版本。

C++20 起, 字符串可以用作非类型模板形参。

4.6.2.2 字符串字面值

`StringLiteral` 类有一个 `constexpr` 构造函数。

```
1 // nonTypeTemplateParameterString.cpp
2
```

```

3 #include <algorithm>
4 #include <iostream>
5
6 template <int N>
7 class StringLiteral {
8 public:
9     constexpr StringLiteral(char const (&str) [N]) {
10         std::copy(str, str + N, data);
11     }
12     char data[N];
13 };
14
15 template <StringLiteral str>
16 class ClassTemplate {};
17
18 template <StringLiteral str>
19 void FunctionTemplate() {
20     std::cout << str.data << '\n';
21 }
22
23 int main() {
24
25     std::cout << '\n';
26
27     ClassTemplate<"string literal"> cls;
28     FunctionTemplate<"string literal">();
29
30     std::cout << '\n';
31
32 }

```

StringLiteral 是一个文字类型，因此可以用作 ClassTemplate(第 15 行) 和 FunctionTemplate(第 18 行) 的非类型模板参数。constexpr 构造函数(第 9 行)接受 C-string 作为参数。

string literal

读者们可能想知道，为什么需要字符串字面量作为非类型模板参数呢？

编译时的正则表达式

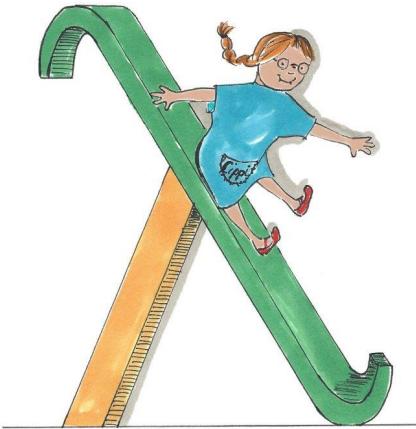
字符串字面量的一个经典用例就是[编译时正则表达式解析](#)。C++23 已经有了一个提案:[P1433R0: 编译时正则表达式](#)。Hana Dusíková 作为提案的作者，希望 C++ 在编译时可以使用正则表达式：“当前的 std::regex 设计和实现 [[正则表达式库](#)] 太慢了，主要是因为 RE[正则表达式] 模式是在运行时解析和编译的。用户通常不需要运行时 RE[正则表达式] 解析器引擎。许多常见的用例中，模式是在编译时就已知了。我认为这违背了 C++ 的承诺：『不使用就不支付』”。

若 RE[正则表达式] 在编译时已知，则应该在编译期间检查。但 `std::regex` 的设计不允许这种 [编译时求值] 的方式，因为 RE 输入是一个运行时字符串，语法错误会作为异常抛出。

总结

- 有条件显式构造函数，允许显式控制泛型构造函数中可以使用的类型。
- C++20 支持更多类型作为非类型模板参数：浮点数和字符串字面值。

4.7. Lambda 的改进



Cippi 滑下滑梯

C++20 中，Lambda 表达式支持模板参数和概念，可以有默认构造，并在没有状态时支持复制赋值。此外，Lambda 表达式可以在未计算的上下文中使用，还会检测何时隐式复制 this 指针。

先从 Lambda 的模板参数开始吧。

4.7.1 Lambda 的模板参数

C++20 中类型化 Lambda(C++11)、泛型 Lambda(C++14) 和模板 Lambda(Lambda 的模板形参)之间的差异很小。

```
1 // templateLambda.cpp
2
3 #include <iostream>
4 #include <string>
5 #include <vector>
6
7 auto sumInt = [](int fir, int sec) { return fir + sec; };
8 auto sumGen = [](auto fir, auto sec) { return fir + sec; };
9 auto sumDec = [](auto fir, decltype(fir) sec) { return fir + sec; };
10 auto sumTem = [<typename T>](T fir, T sec) { return fir + sec; };
11
12 int main() {
13
14     std::cout << '\n';
15
16     std::cout << "sumInt(2000, 11): " << sumInt(2000, 11) << '\n';
17     std::cout << "sumGen(2000, 11): " << sumGen(2000, 11) << '\n';
18     std::cout << "sumDec(2000, 11): " << sumDec(2000, 11) << '\n';
19     std::cout << "sumTem(2000, 11): " << sumTem(2000, 11) << '\n';
20
21     std::cout << '\n';
22
23     std::string hello = "Hello ";
```

```

24     std::string world = "world";
25     // std::cout << "sumInt(hello, world): " << sumInt(hello, world) << '\n';
26     std::cout << "sumGen(hello, world): " << sumGen(hello, world) << '\n';
27     std::cout << "sumDec(hello, world): " << sumDec(hello, world) << '\n';
28     std::cout << "sumTem(hello, world): " << sumTem(hello, world) << '\n';
29
30
31     std::cout << '\n';
32
33     std::cout << "sumInt(true, 2010): " << sumInt(true, 2010) << '\n';
34     std::cout << "sumGen(true, 2010): " << sumGen(true, 2010) << '\n';
35     std::cout << "sumDec(true, 2010): " << sumDec(true, 2010) << '\n';
36     // std::cout << "sumTem(true, 2010): " << sumTem(true, 2010) << '\n';
37
38     std::cout << '\n';
39
40 }

```

在展示程序输出之前，我想比较一下这四个 Lambda 表达式。

- sumInt
 - C++11
 - 类型化 Lambda
 - 只接受能转换为 int 的类型
- sumGen
 - C++14
 - 泛型 Lambda
 - 可接受所有类型
- sumDec
 - C++14
 - 泛型 Lambda
 - 第二个参数的类型必须可以转换为第一个参数的类型
- sumTem
 - C++20
 - 模板 Lambda
 - 两个参数类型必须相同

这对于不同类型的模板参数意味着什么？当然，每个 Lambda 传入的都是 int 型变量（第 16-19 行），而类型化的 Lambda sumInt 不接受字符串（第 25 行）。

将 bool true 和 int 2010 传递给 Lambda 可能会有出乎意料（第 33-36 行）的结果。

- sumInt 返回 2011，true 可以是一个整型值，可升格为 int。

- sumGen 返回 2011, true 可以是一个整型值, 可升格为 int。sumInt 和 sumGen 之间有一个细微的区别。
- sumDec 返回 2。为什么? 第二个形参 sec 的类型变成了第一个形参 fir 的类型: 因为 decltype(fir) sec, 编译器推导出了 fir 的类型, 其类型与 sec 类型相同。因此, 2010 转换为 true。在表达式 fir + sec 中, fir 提升为 1, 所以结果是 2。
- sumTem 是无效的。

```

sumInt(2000, 11): 2011
sumGen(2000, 11): 2011
sumDec(2000, 11): 2011
sumTem(2000, 11): 2011

sumGen(hello, world): Hello world
sumDec(hello, world): Hello world
sumTem(hello, world): Hello world

sumInt(true, 2010): 2011
sumGen(true, 2010): 2011
sumDec(true, 2010): 2

```

模板 Lambda 的典型的用例是在 Lambda 中使用容器。下面的代码给出了三个接受容器的 Lambda 表达式, 每个表达式都会返回容器的大小。

三个 Lambda 表达式都可以接受容器

```

1 // templateLambdaVector.cpp
2
3 #include <concepts>
4 #include <deque>
5 #include <iostream>
6 #include <string>
7 #include <vector>
8
9 auto lambdaGeneric = [](&const auto& container) { return container.size(); };
10 auto lambdaVector = [<typename T>(&const std::vector<T>& vec) { return vec.size(); };
11 auto lambdaVectorIntegral = [<std::integral T>(&const std::vector<T>& vec) {
12     return vec.size();
13 };
14
15 int main() {
16
17     std::cout << '\n';

```

```

19
20 std::deque deq{1, 2, 3};
21 std::vector vecDouble{1.1, 2.2, 3.3, 4.4};
22 std::vector vecInt{1, 2, 3, 4, 5};
23
24 std::cout << "lambdaGeneric(deq): " << lambdaGeneric(deq) << '\n';
25 // std::cout << "lambdaVector(deq): " << lambdaVector(deq) << '\n';
26 // std::cout << "lambdaVectorIntegral(deq): "
27 // << lambdaVectorIntegral(deq) << '\n';
28
29 std::cout << '\n';
30
31 std::cout << "lambdaGeneric(vecDouble): " << lambdaGeneric(vecDouble) << '\n';
32 std::cout << "lambdaVector(vecDouble): " << lambdaVector(vecDouble) << '\n';
33 // std::cout << "lambdaVectorIntegral(vecDouble): "
34 // << lambdaVectorIntegral(vecDouble) << '\n';
35
36 std::cout << '\n';
37
38 std::cout << "lambdaGeneric(vecInt): " << lambdaGeneric(vecInt) << '\n';
39 std::cout << "lambdaVector(vecInt): " << lambdaVector(vecInt) << '\n';
40 std::cout << "lambdaVectorIntegral(vecInt): "
41 << lambdaVectorIntegral(vecInt) << '\n';
42
43 std::cout << '\n';
44
45 }

```

函数 `lambdaGeneric`(第 9 行) 可以接受有成员函数 `size()` 的数据类型。函数 `lambdaVector`(第 10 行) 更具体: 只接受 `std::vector`。函数 `lambdaVectorIntegral`(第 11 行) 使用 C++20 概念 `std::integral`, 所以只接受使用整型(如 `int`)的 `std::vector`。为了使用 `std::integral` 概念, 必须包含头文件 `<concepts>`。

```

lambdaGeneric(deq): 3

lambdaGeneric(vecDouble): 4
lambdaVector(vecDouble): 4

lambdaGeneric(vecInt): 5
lambdaVector(vecInt): 5
lambdaVectorIntegral(vecInt): 5

```

类模板参数的推导

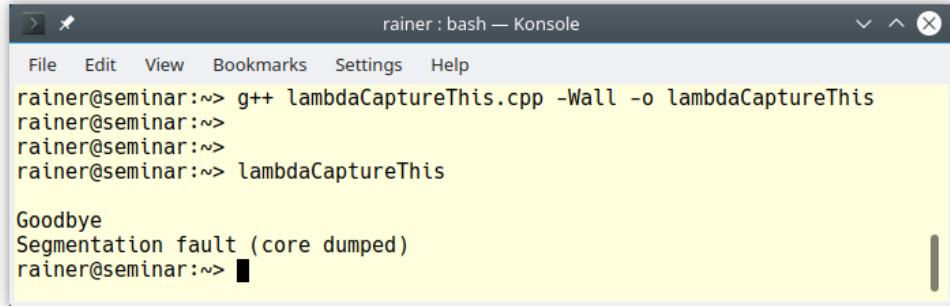
templateLambdaVector.cpp 中有个可能会错过的特性。从 C++17 开始，编译器就可以从类模板的实参推断类模板的类型(第 20-22 行)。初始化 vector 时，可以这样写 std::vector myVec1,2,3，而非冗长的 std::vector<int> myVec1,2,3。

4.7.2 检测并隐式复制 this 指针

C++20 编译器会检测何时隐式复制 this 指针。复制隐式捕获的 this 指针会导致未定义的行为，例如：

```
1 // lambdaCaptureThis.cpp
2
3 #include <iostream>
4 #include <string>
5
6 struct LambdaFactory {
7     auto foo() const {
8         return [=] { std::cout << s << '\n'; };
9     }
10    std::string s = "LambdaFactory";
11    ~LambdaFactory() {
12        std::cout << "Goodbye" << '\n';
13    }
14};
15
16 auto makeLambda() {
17     LambdaFactory lambdaFactory;
18
19     return lambdaFactory.foo();
20 }
21
22
23 int main() {
24
25     std::cout << '\n';
26
27     auto lam = makeLambda();
28     lam();
29
30     std::cout << '\n';
31
32 }
```

程序的编译没问题，但这并不说明程序执行起来没问题。



```
rainer@seminar:~/g&gt; g++ lambdaCaptureThis.cpp -Wall -o lambdaCaptureThis
rainer@seminar:~/g&gt;
rainer@seminar:~/g&gt;
rainer@seminar:~/g&gt; ./lambdaCaptureThis
Goodbye
Segmentation fault (core dumped)
rainer@seminar:~/g&gt;
```

由于未定义行为造成的段错误

发现 lambdaCaptureThis.cpp 中的问题了吗? 成员函数 foo(第 7 行) 返回表达式 [=] std::cout << s << '\n'; 具有 this 指针的隐式副本。这个隐式复制在(第 17 行)中没有问题,但是在作用域的末尾就成了问题。作用域的结束意味着局部 Lambda 生命周期的结束(第 19 行),所以使用 lam()(第 28 行)会触发未定义的行为。

这种情况下, C++20 编译器会发出警告。

```
<source>:8:16: warning: implicit capture of 'this' via '=[]' is deprecated in C++20 [-Wdeprecated]
  8 |         return [=] { std::cout << s << std::endl; };
     |         ^
<source>:8:16: note: add explicit 'this' or '*this' capture
Execution build compiler returned: 0
Program returned: 139

Goodbye
```

C++20 的最后两个 Lambda 特性结合起来非常好用:C++20 中的 Lambda 可以使用默认构造,并且在没有状态时支持复制赋值。此外, Lambda 表达式可以在未计算的上下文中使用。

4.7.3 未计算上下文中的 Lambda 和无状态的 Lambda 可以默认构造和复制赋值

本节的标题包含两个术语:未计算的上下文和无状态的 Lambda。先从“未计算的上下文”开始说起。

4.7.3.1 未计算的上下文

下面的代码段具有函数声明和函数定义。

```
1 int add1(int, int); // declaration
2 int add2(int a, int b) { return a + b; } // definition
```

函数 add1 是声明的,而 add2 是定义。若在已计算的上下文中使用 add1,例如:通过调用将得到一个链接时错误。关键是在于可以在未计算的上下文中使用 add1,例如 typeid 或 decltype,两个操作符都接受未计算的操作数。

```
1 // unevaluatedContext.cpp
2
3 #include <iostream>
4 #include <typeinfo> // typeid
```

```

5
6 int add1(int, int); // declaration
7 int add2(int a, int b) { return a + b; } // definition
8
9 int main() {
10
11    std::cout << '\n';
12
13    std::cout << "typeid(add1).name(): " << typeid(add1).name() << '\n';
14
15    decltype(*add1) add = add2;
16
17    std::cout << "add(2000, 20): " << add(2000, 20) << '\n';
18
19    std::cout << '\n';
20
21 }

```

typeid(add1).name()(第 13 行) 返回类型的字符串表示形式, decltype(第 15 行) 可以推导其参数的类型。



4.7.3.2 无状态的 Lambda

无状态是指不捕获任何内容的 Lambda。换句话说，无状态的 Lambda 是定义中初始中括号 [] 为空的 Lambda。例如，Lambda 表达式 auto add = [](int a, int b) {return a + b;}; 就是无状态的。

4.7.3.3 适配标准模板库中的关联容器

展示示例之前，先添加一些注释。容器 std::set 和其他来自标准模板库的有序关联容器 (std::map, std::multiset 和 std::multimap) 默认使用函数对象 std::less 对键进行排序。std::less 按字典升序对所有键进行排序。std::set 的声明，隐式使用了 std::less。

std::set 的声明

```

1 template<
2     class Key,

```

```
3 class Compare = std::less<Key>,
4 class Allocator = std::allocator<Key>
5 > class set;
```

再来看看排序。

Lambda 用于未计算的上下文中

```
43 } ) >;
44 setAbsolute set5 = {-10, 5, 3, 100, 0, -25};
45 printContainer(set5);
46
47 std::cout << "\n\n";
48
49 }
```

set1(第 19 行) 和 set4(第 38 行) 按升序排序键。set2(第 26 行)、set3(第 33 行) 和 set5(第 44 行) 都以相同的方式对其键进行排序，在未计算的上下文中使用 Lambda。使用 using 关键字(第 22 行) 声明了一个类型别名，下一行(第 26 行) 中使用它来定义 set。创建 std::set 会使用无状态 Lambda 表达式的默认构造函数。

下面是程序的输出。

```
Bjarne Dave Herb michael scott
scott michael Herb Dave Bjarne
Herb scott Bjarne michael

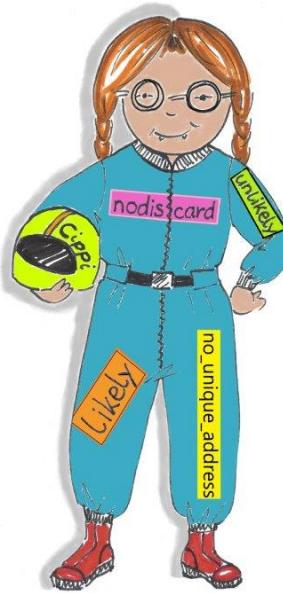
-25 -10 0 3 5 100
0 3 5 -10 -25 100
```

在研究该程序的输出时，可能会感到惊讶。set3 有些特殊，使用 `[](const auto& l, const auto& r){return l.size() < r.size();}` 作为谓词，并且忽略了 Dave。因为 Dave 和 Herb 字符串的长度一样，而 Herb 是先添加的，所以忽略 Dave。std::set 支持唯一键，所以使用特殊谓词的键相同。若使用 std::multiset，这就不会发生这种情况。

总结

C++20 中，Lambda 表达式可以有模板参数。此外，Lambda 会检测 this 指针何时隐式引用。

4.8. 新属性



Cippi 已经为比赛做好了准备

C++20 对属性进行了改进，并添加了新的属性，如 `[[nodiscard("reason")]]`、`[[likely]]`、`[[unlikely]]` 和 `[[no_unique_address]]`。并且，`[[nodiscard("reason")]]` 可以用来显式地表达接口的意图。

属性

属性允许开发者对源代码进行约束，或者为编译器提供优化可能性。可以为类型、变量、函数、名称和代码块使用属性。使用多个属性时，可以依次应用 (func1)，或者在一个属性中一起应用，用逗号 (func2) 分隔：

使用属性

```
1 [[attribute1]] [[attribute2]] [[attribute3]]
2 int func1();
3
4 [[attribute1, attribute2, attribute3]]
5 int func2();
```

属性可以是实现定义的语言扩展或标准属性，下面是 C++11 至 C++17 的已有属性列表。

- `[[noreturn]]` (C++11): 表示函数不返回
- `[[carries_dependency]]` (C++11): 表示 release-consume 内存序 依赖链中的依赖项
- `[[deprecated]]` (C++14): 表示不再使用的名称
- `[[fallthrough]]` (C++17): 表示有意跳过一个分支
- `[[maybe_unused]]` (C++17): 抑制编译器对未使用名称的警告

4.8.1 [[nodiscard("reason")]]

C++17 毫无理由地引入了新的属性 [[nodiscard]]。C++20 增加了向属性中添加信息的可能性。

丢弃对象和错误码

```
1 // withoutNodiscard.cpp
2
3 #include <utility>
4
5 struct MyType {
6
7     MyType(int, bool) {}
8
9 };
10
11 template <typename T, typename ... Args>
12 T* create(Args&& ... args) {
13     return new T(std::forward<Args>(args)...);
14 }
15
16 enum class ErrorCode {
17     Okay,
18     Warning,
19     Critical,
20     Fatal
21 };
22
23 ErrorCode errorProneFunction() { return ErrorCode::Fatal; }
24
25 int main() {
26
27     int* val = create<int>(5);
28     delete val;
29
30     create<int>(5);
31
32     errorProneFunction();
33
34     MyType(5, true);
35 }
36 }
```

因为转发和参数包，工厂函数 `create`(第 11 行) 可以调用构造函数，并返回一个在堆上分配的对象。

这段代码有很多问题。首先，第 30 行存在内存泄漏，堆上创建的 `int` 永远不会删除。其次，没有检查函数 `errorProneFunction`(第 32 行) 的错误代码。最后，构造函数调用 `MyType(5, true)`(第 34 行) 创建一个临时对象，创建后立即销毁，这至少是一种资源浪费。现在，让 [[nodiscard]] 发挥作用吧。

[[nodiscard]] 可以在函数声明、枚举声明或类声明中使用。若丢弃了声明为 [[nodiscard]] 的函

数的返回值，编译器应该发出警告。通过复制枚举或声明为 [[nodiscard]] 的类返回的函数也是如此。若仍然想忽略返回值，可以将其强制转换为 void。

下面的例子中，使用了 C++17 的属性 [[nodiscard]]。

```
1 // nodiscard.cpp
2
3 #include <utility>
4
5 struct MyType {
6
7     MyType(int, bool) {}
8
9 };
10
11 template <typename T, typename ... Args>
12 [[nodiscard]]
13 T* create(Args&& ... args) {
14     return new T(std::forward<Args>(args)...);
15 }
16
17 enum class [[nodiscard]] ErrorCode {
18     Okay,
19     Warning,
20     Critical,
21     Fatal
22 };
23
24 ErrorCode errorProneFunction() { return ErrorCode::Fatal; }
25
26 int main() {
27
28     int* val = create<int>(5);
29     delete val;
30
31     create<int>(5);
32
33     errorProneFunction();
34
35     MyType(5, true);
36
37 }
```

工厂函数 create(第 13 行) 和 enum ErrorCode(第 17 行) 声明为 [[nodiscard]]。因此，第 31 行和第 33 行中在使用时，触发了警告。

```

rainer@seminar:~/> g++ nodiscard.cpp -fno-nodiscard
nodiscard.cpp: In function 'int main()':
nodiscard.cpp:31:16: warning: ignoring return value of 'T* create(Args&& ...)' [with T = int; Args = {int}], declared with attribute nodiscard [-Wunreachable-code]
    create<int>(5); // (1)
                                         ^
nodiscard.cpp:13:4: note: declared here
T* create(Args&& ... args);
                                         ^
nodiscard.cpp:33:23: warning: ignoring returned value of type 'ErrorCode', declared with attribute nodiscard [-Wunreachable-code]
    ErrorCode errorProneFunction(); // (2)
                                         ^
nodiscard.cpp:24:11: note: in call to 'ErrorCode errorProneFunction()', declared here
    ErrorCode errorProneFunction() { return ErrorCode::Fatal; }
                                         ^
nodiscard.cpp:17:26: note: 'ErrorCode' declared here
enum class [[nodiscard]] ErrorCode {
                                         ^
rainer@seminar:~/>

```

C++17 编译器会报错

这样就好多了，但这段代码还有一些问题。`[[nodiscard]]` 不能用于构造函数等不返回值的函数，所以临时的 `MyType(5, true)`(第 35 行)仍会创建，并且没有警告。第二，错误消息太一般。作为函数的使用者，我想知道为什么丢弃结果这一行为是有问题的。

这两个问题都可以用 C++20 解决。构造函数可以声明为 `[[nodiscard]]`，警告可以包含其他信息。

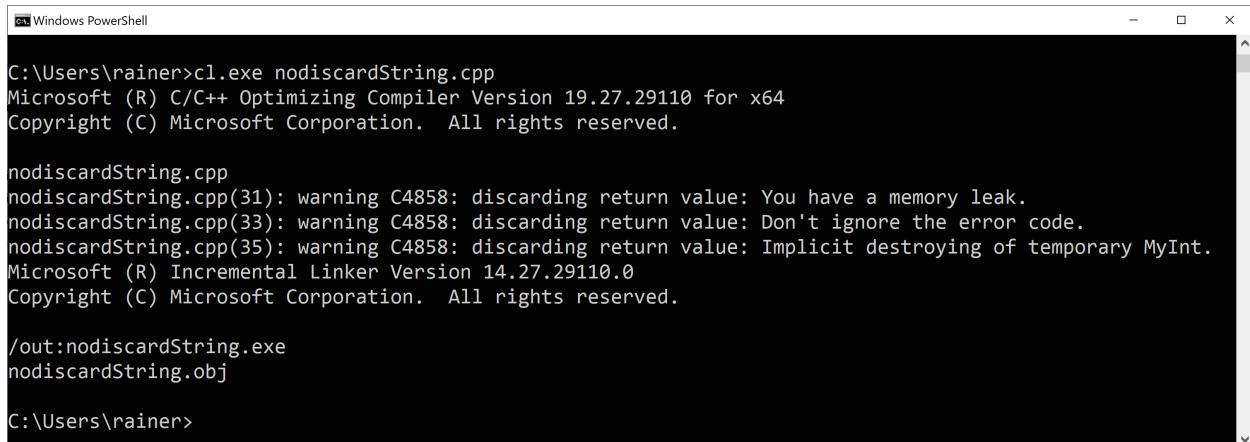
C++20 中使用 `[[nodiscard]]` 属性

```

1 // nodiscardString.cpp
2
3 #include <utility>
4
5 struct MyType {
6
7     [[nodiscard("Implicit destroying of temporary MyInt.")]] MyType(int, bool) {}
8
9 };
10
11 template <typename T, typename ... Args>
12 [[nodiscard("You have a memory leak.")]]
13 T* create(Args&& ... args){
14     return new T(std::forward<Args>(args)...);
15 }
16
17 enum class [[nodiscard("Don't ignore the error code.")]] ErrorCode {
18     Okay,
19     Warning,
20     Critical,
21     Fatal
22 };
23
24 ErrorCode errorProneFunction() { return ErrorCode::Fatal; }
25
26 int main() {
27
28     int* val = create<int>(5);
29     delete val;
30
31     create<int>(5);
32 }
```

```
33     errorProneFunction();
34
35     MyType(5, true);
36
37 }
```

现在，将输出特定的消息。下面是 Microsoft 编译器的输出。



```
C:\Users\rainer>cl.exe nodiscardString.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 19.27.29110 for x64
Copyright (C) Microsoft Corporation. All rights reserved.

nodiscardString.cpp
nodiscardString.cpp(31): warning C4858: discarding return value: You have a memory leak.
nodiscardString.cpp(33): warning C4858: discarding return value: Don't ignore the error code.
nodiscardString.cpp(35): warning C4858: discarding return value: Implicit destroying of temporary MyInt.
Microsoft (R) Incremental Linker Version 14.27.29110.0
Copyright (C) Microsoft Corporation. All rights reserved.

/out:nodiscardString.exe
nodiscardString.obj

C:\Users\rainer>
```

C++20 编译器会对丢弃对象和错误代码的行为报错

std::async 的问题

C++ 中的许多函数都可以使用 `[[nodiscard]]` 属性，并从中获益，其中就包括 `std::async`。当不使用 `std::async` 的返回值时，想要 `std::async` 异步使用隐式地转变为同步使用。应该在单独的线程中运行的内容，但却以阻塞的方式使用。更多关于 `std::async` 的反直觉行为，请参阅我的帖子 “[the-special-futures](#)”。

研究cppreference.com/nodiscard上的 `[[nodiscard]]` 语法时，我注意到 `std::async` 的声明在 C++20 中发生了变化：

std::async 在 C++20 中使用属性 `[[nodiscard]]`

```
1 template<class Function, class... Args>
2 [[nodiscard]]
3 std::future<std::invoke_result_t<std::decay_t<Function>,
4             std::decay_t<Args>...>>
5     async( Function&& f, Args&&... args );
```

`std::async` 的返回类型在 C++20 中声明为 `[[nodiscard]]`。

接下来的两个属性 `[[likely]]` 和 `[[unlikely]]` 是关于优化的。

4.8.2 `[[likely]]` 和 `[[unlikely]]`

`[[likely]]` 和 `[[unlikely]]` 属性的提案[P0479R5](#)据我所知是最短的。为了说明动机，提案中添加了有趣的注释。“可能属性的使用旨在允许实现针对以下情况进行优化：语句或标签上，包含该属性的

执行路径比不包含该属性的执行路径的可能性都要大。使用 `unlikely` 属性是为了允许实现针对以下情况进行优化：语句或标签上，包含该属性的执行路径比不包含该属性的执行路径更不可能。当执行路径包含到该标签的跳转时，包含该标签。不过，过度使用这些属性都可能导致性能的下降。”总而言之，这两个属性都会向优化器提供与预期执行路径相关的提示。

[[likely]]：给优化器一个提示

```
1 for(size_t i=0; i < v.size(); ++i){  
2     if (v[i] < 0) [[likely]] sum -= sqrt(-v[i]);  
3     else sum += sqrt(v[i]);  
4 }
```

优化的过程继续使用新的属性 `[[no_unique_address]]`，这个优化处理的是空间上的，而不是执行时间上的。

4.8.3 [[no_unique_address]]

`[[no_unique_address]]` 表示类中的成员变量不需要有内存地址。若成员为空类型，编译器可以将其优化为不占用内存。

下面的程序演示了新属性的用法。

使用 `[[no_unique_address]]` 属性

```
1 // uniqueAddress.cpp  
2  
3 #include <iostream>  
4  
5 struct Empty {};  
6  
7 struct NoUniqueAddress {  
8     int d{};  
9     [[no_unique_address]] Empty e{};  
10 };  
11  
12 struct UniqueAddress {  
13     int d{};  
14     Empty e{};  
15 };  
16  
17 int main() {  
18  
19     std::cout << '\n';  
20  
21     std::cout << std::boolalpha;  
22  
23     std::cout << "sizeof(int) == sizeof(NoUniqueAddress): "  
24         << (sizeof(int) == sizeof(NoUniqueAddress)) << '\n';  
25 }
```

```

26     std::cout << "sizeof(int) == sizeof(UniqueAddress): "
27         << (sizeof(int) == sizeof(UniqueAddress)) << '\n';
28
29     std::cout << '\n';
30
31     NoUniqueAddress NoUnique;
32
33     std::cout << "&NoUnique.d: " << &NoUnique.d << '\n';
34     std::cout << "&NoUnique.e: " << &NoUnique.e << '\n';
35
36     std::cout << '\n';
37
38     UniqueAddress unique;
39
40     std::cout << "&unique.d: " << &unique.d << '\n';
41     std::cout << "&unique.e: " << &unique.e << '\n';
42
43     std::cout << '\n';
44
45 }

```

NoUniqueAddress 类的大小等于 int(第 7 行), 但 UniqueAddress 类的大小不等于 int(第 12 行)。UniqueAddress 的成员 d 和 e(第 40 行和 41 行)有不同的地址, 但 UniqueAddress 类的成员没有(第 33 行和 34 行)。

```

sizeof(int) == sizeof(NoUniqueAddress): true
sizeof(int) == sizeof(UniqueAddress): false

&NoUnique.d: 0x7fff44f8fd0c
&NoUnique.e: 0x7fff44f8fd0c

&unique.d: 0x7fff44f8fd04
&unique.e: 0x7fff44f8fd08

```

总结

C++20 添加了一些新的属性:

- `[[nodiscard("reason")]]` 可以在上下文中检查函数的返回值是否忽略。
- `[[likely]]` 和 `[[unlikely]]` 允许开发者给编译器一个提示, 哪个代码路径更有可能执行到。
- 使用 `[[no_unique_address]]` 属性, 类中不同的成员变量可以拥有相同的地址。

4.9. 进一步的改善



Cippi 在上楼梯

本节介绍 C++20 核心语言中其余的改进。

4.9.1 volatile

P1152R0 提案摘要概述了 volatile 所经历的变化：“建议弃用 volatile 的部分用法，并删除不明确的部分的用法”[知乎上的讨论](#)。这篇文章的目的在于，在运行时或编译器更新时已经出现了微妙损坏的代码，使用 volatile 的不明确行为时，会使编译时代码遭到破坏。”深入讨论 volatile 之前，我想回答一个关键问题：什么时候应该使用 volatile？

C++ 标准的某个注释：“volatile 是对实现的一个提示，以避免进行激进地优化，因为对象的值可能会通过实现无法检测到的方式改变。”所以对于单线程，编译器必须在可执行文件中执行加载或存储操作，就像源码中一样频繁，所以 volatile 不能消除或重新排序。因此，可以使用 volatile 对象与处理程序通信，但不能用于与另一个执行线程通信。

展示 volatile 保留了哪些语义之前，我想先从已弃用的特性开始说起：

1. 弃用 volatile 的复合赋值，以及前后递增/递减
2. 弃用函数参数或返回类型的 volatile 限定
3. 结构化绑定声明中弃用 volatile 限定符

若想知道背后复杂的细节，我建议你观看一下 CppCon 2019 的演讲[“废除 volatile”](#)。下面是他演讲中的几个例子，我修复了源码中的一些拼写错误。下面代码片段中的数字代表前面列出的三个弃用语义。

弃用的 volatile 用例

```
1 // (1)
2 int neck, tail;
3 volatile int brachiosaur;
4 brachiosaur = neck; // OK, a volatile store
5 tail = brachiosaur; // OK, a volatile load
6
7 // deprecated: does this access brachiosaur once or twice
8 tail = brachiosaur = neck;
```

```

9
10 // deprecated: does this access brachiosaur once or twice
11 brachiosaur += neck;
12
13 // OK, a volatile load, an addition, a volatile store
14 brachiosaur = brachiosaur + neck;
15
16 ######
17 // (2)
18 // deprecated: a volatile return type has no meaning
19 volatile struct amber jurassic();
20
21 // deprecated: volatile parameters aren't meaningful to the
22 // caller, volatile only applies within the function
23 void trex(volatile short left_arm, volatile short right_arm);
24
25 // OK, the pointer isn't volatile, the data it points to is
26 void fly(volatile struct pterosaur* pterandon);
27
28 #####
29 (3)
30 struct linhenykus { volatile short forelimb; };
31 void park(linhenykus alvarezsauroid) {
32     // deprecated: does the binding copy the forelimbs?
33     auto [what_is_this] = alvarezsauroid; // structured binding
34     // ...
35 }
```

volatile 和多线程语义

volatile 通常用于表示，独立于常规程序流，并可以进行修改的对象。例如，这些是嵌入式编程中表示外部设备（内存映射 I/O）的对象。因为这些对象可以独立于常规程序流进行修改，并且它值直接写入主存，因此缓存中不会进行优化存储。简单来说，volatile 避免了主动优化，并且没有多线程语义。

4.9.2 带有初始化式的范围 for 循环

C++20 可以直接使用带有初始化式的范围 for 循环。

```

1 // rangeBasedForLoopInitializer.cpp
2
3 #include <iostream>
4 #include <string>
5 #include <vector>
6
7 int main() {
8
9     for (auto vec = std::vector{1, 2, 3}; auto v : vec) {
```

```

10    std::cout << v << " ";
11 }
12
13 std::cout << "\n\n";
14
15 for (auto initList = {1, 2, 3}; auto e : initList) {
16     e *= e;
17     std::cout << e << " ";
18 }
19
20 std::cout << "\n\n";
21
22 using namespace std::string_literals;
23 for (auto str = "Hello World"s; auto c: str) {
24     std::cout << c << " ";
25 }
26
27 std::cout << '\n';
28
29 }

```

基于范围的 for 循环在第 9 行使用 std::vector，第 15 行使用 std::initializer_list，第 23 行使用 std::string。此外，第 9 行和第 15 行中，类模板使用了 C++17 的自动类型推断(使用 std::vector，而非 std::vector<int>)。

```

1 2 3

1 4 9

H e l l o W o r l d

```

4.9.3 virtual constexpr 函数

constexpr 函数有可能在编译时执行，但也可以在运行时执行。因此，可以使用 C++20 的 virtual 关键字创建 constexpr 函数。virtual 的 constexpr 函数可以重写为非 constexpr 函数，virtual 的非 constexpr 函数可以重写 virtual 的 constexpr 函数。在这里，重写意味着基类的相关函数是虚函数。

virtualconstexpr.cpp 展示了这两种组合：

```

1 // virtualConstexpr.cpp
2
3 #include <iostream>
4
5 struct X1 {
6     virtual int f() const = 0;
7 };

```

```

9 struct X2: public X1 {
10    constexpr int f() const override { return 2; }
11 };
12
13 struct X3: public X2 {
14    int f() const override { return 3; }
15 };
16
17 struct X4: public X3 {
18    constexpr int f() const override { return 4; }
19 };
20
21 int main() {
22
23    X1* x1 = new X4;
24    std::cout << "x1->f(): " << x1->f() << '\n';
25
26    X4 x4;
27    X1& x2 = x4;
28    std::cout << "x2.f(): " << x2.f() << '\n';
29
30 }
```

第 24 行通过指针使用虚函数(后期绑定), 第 28 行通过引用使用虚函数。

```
x1->f(): 4
x2.f(): 4
```

4.9.4 UTF-8 字符串的新字符类型:char8_t

除了 C++11 中的字符类型 `char16_t` 和 `char32_t` 外, C++20 添加了新的字符类型 `char8_t`。类型 `char8_t` 足够大, 可以表示任何 UTF-8 单位(8 位)。其具有与 `unsigned char` 相同的大小、符号和对齐方式, 但这是两个不同的类型。

char 和 `char8_t`

与 `char8_t` 相反, `char` 有一个字节, 一个字节的比特数, 并且 `char` 的比特数没有明确定义。不过, 几乎所有的实现都使用 8 位作为一个字节。`char` 类型的 `std::basic_string`, 别名为 `std::string`。

`std::string` 和 `std::string` 字面值

```

1 std::string std::basic_string<char>
2 "Hello World"s
```

因此, C++20 有一个新的字符类型 `char8_t`(第 1 行)和新的 UTF-8 字符串字面值(第 2 行)。

新的 `char8_t` 字符类型和 UTF-8 字面值

```
1 std::u8string std::basic_string<char8_t>
2 u8"Hello World"
```

`char8Str.cpp` 展示了新字符类型 `char8_t` 的用法。

```
1 // char8Str.cpp
2
3 #include <iostream>
4 #include <string>
5
6 int main() {
7
8     const char8_t* char8Str = u8"Hello world";
9     std::basic_string<char8_t> char8String = u8"helloWorld";
10    std::u8string char8String2 = u8"helloWorld";
11
12    char8String2 += u8".";
13
14    std::cout << "char8String.size(): " << char8String.size() << '\n';
15    std::cout << "char8String2.size(): " << char8String2.size() << '\n';
16
17    char8String2.replace(0, 5, u8"Hello ");
18
19    std::cout << "char8String2.size(): " << char8String2.size() << '\n';
20
21 }
```

下面是程序的输出:

```
char8String.size(): 10
char8String2.size(): 11
char8String2.size(): 12
```

4.9.5 本地作用域中的 `using enum`

`using enum` 声明在局部作用域中引入已命名枚举的枚举值。

本地作用域内引入枚举值

```
1 // enumUsing.cpp
2
3 #include <iostream>
4 #include <string_view>
5
6 enum class Color {
```

```

7     red,
8     green,
9     blue
10 }
11
12 std::string_view toString(Color col) {
13     switch (col) {
14         using enum Color;
15         case red: return "red";
16         case green: return "green";
17         case blue: return "blue";
18     }
19     return "unknown";
20 }
21
22 int main() {
23
24     std::cout << '\n';
25
26     std::cout << "toString(Color::red): " << toString(Color::red) << '\n';
27
28     using enum Color;
29
30     std::cout << "toString(green): " << toString(green) << '\n';
31
32     std::cout << '\n';
33
34 }

```

using enum 声明 (第 14 行) 将作用域 enumeration Color 的枚举值引入到本地作用域，从而枚举值可以无作用域的进行使用 (第 15-17 行)。

```

Windows PowerShell
C:\Users\rainer>enumUsing.exe
toString(Color::red): red
toString(green): green

C:\Users\rainer>

```

使用 using enum

4.9.6 位域成员变量的默认初始化器

首先，什么是位域？下面是来自[Wikipedia](#)：“位域是计算机编程中的数据结构，由许多相邻的计算机内存位置组成，这些位置可用来保存一组位值，存储的目的是使该集合中的单个位或一组位都可以寻址。位域常用来表示固定宽度的整型。”

C++20 可以默认初始化位域成员变量：

```

1 // bitField.cpp

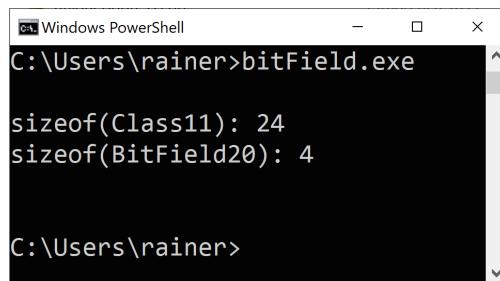
```

```

2
3 #include <iostream>
4
5 struct Class11 {
6     int i = 1;
7     int j = 2;
8     int k = 3;
9     int l = 4;
10    int m = 5;
11    int n = 6;
12 };
13
14 struct BitField20 {
15     int i : 3 = 1;
16     int j : 4 = 2;
17     int k : 5 = 3;
18     int l : 6 = 4;
19     int m : 7 = 5;
20     int n : 7 = 6;
21 };
22
23 int main () {
24
25     std::cout << '\n';
26
27     std::cout << "sizeof(Class11): " << sizeof(Class11) << '\n';
28     std::cout << "sizeof(BitField20): " << sizeof(BitField20) << '\n';
29
30     std::cout << '\n';
31
32 }

```

C++11 中，根据类的成员(第 6-11 行)，位域成员可以有默认的初始化器(第 15-20 行)。当把 3、4、5、6、7、7 这些数字加起来时，会得到 32。因此，32 位或 4 字节恰好是 BitField20 的大小：



位域的大小

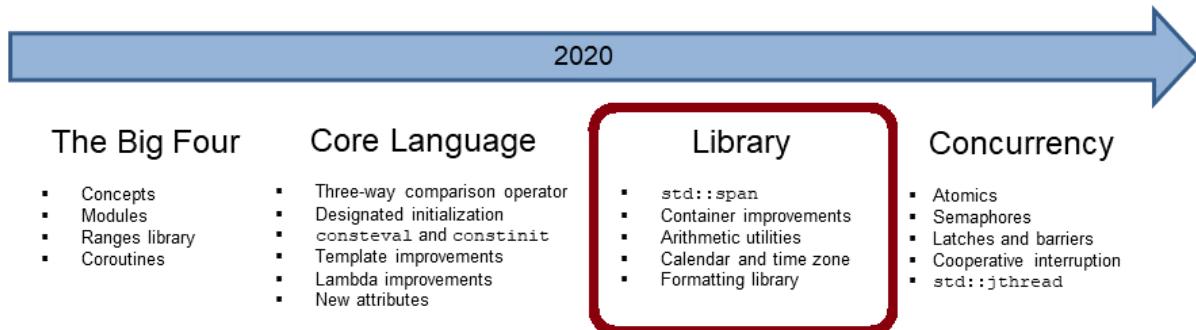
总结

- C++20 中明确了 volatile 的行为。volatile 没有多线程语义，应仅用于避免激进优化，因为对象可能独立于常规程序流，并且可以进行修改。

- 基于范围的 for 循环可以使用初始化式。
- 新的字符类型 `char8_t` 足够大，明确可以表示 8 位。
- `using enum` 声明在本地作用域中引入了命名枚举的枚举值。
- 位域的成员可以默认初始化。
- `constexpr` 函数可以是虚函数。

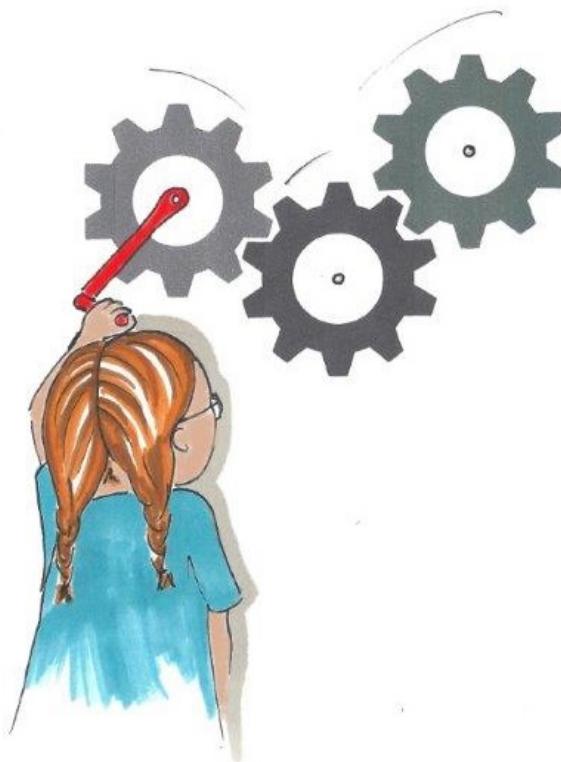
第 5 章 标准库

C++20



除了范围库之外，C++20 标准库还提供了许多新特性，`std::span` 作为对连续内存区域的非所有引用，改进的字符串和容器实现，以及改进的算法。此外，为 chrono 库扩展了日历和时区功能，还可以安全且高效地对文本进行格式化。

5.1. 范围库



Cippi 开启了流水作业

由于 C++20 中的范围库，使用标准模板库 (STL) 变得更加好用和强大。范围库使用惰性算法，可以直接在容器上工作，并且可以很容易地组合。简而言之：范围库的简单和强大基于其思想。

深入讨论细节之前，先来看一个范围库的例子：

结合 transform 和 filter 函数

```
1 // rangesFilterTransform.cpp
2 #include <iostream>
3 #include <ranges>
4 #include <vector>
5 int main() {
6     std::vector<int> numbers = {1, 2, 3, 4, 5, 6};
7     auto results = numbers | std::views::filter([](int n){ return n % 2 == 0; })
8         | std::views::transform([](int n){ return n * 2; });
9     for (auto v: results) std::cout << v << " ";
10    // 4 8 12
11 }
```

必须从左到右读这个表达式。管道符号代表函数组合：首先，所有偶数都可以通过 (std::views::filter([](int n){return n % 2 == 0;}))。之后，每个剩余的数字会映射到双倍功能上 (std::views::transform([](int n){return n * 2;}))。这个示例展示了范围库的两个新特性：可以将函数组合，应用于单个容器。

接下来，来了解一下范围 (range) 和视图 (view)。

5.1.1 范围和视图

在关于概念的章节中，我已经介绍了范围和视图，这里简单复习一下。

- 范围 (range) 是一组可以迭代的项，提供了一个开始迭代器和一个结束哨兵，STL 容器也属于范围。

视图是在范围上执行的一些操作。视图不拥有数据，其复制、移动或赋值的时间复杂度恒定。

在范围内操作的视图

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

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

C++20 的视图有助于函数式编程。视图可以组合，并且是惰性的。我已经介绍了两种视图，但 C++20 提供了更多的视图。

视图	描述
std::views::all_t	将范围转换为视图。
std::views::all	
std::ranges::ref_view	获取另一个范围中的所有元素。
std::ranges::filter_view	获取满足谓词的元素。
std::views::filter	
std::ranges::transform_view	变换每个元素。
std::views::transform	
std::ranges::take_view	获取另一个视图的前 n 个元素。
std::views::take	
std::ranges::take_while_view	若谓词返回 true，就获取另一个视图的元素。
std::views::take_view	
std::ranges::drop_view	跳过另一个视图的前 n 个元素。
std::views::drop	
std::ranges::drop_while_view	跳过另一个视图的初始元素，直到谓词返回 false。
std::views::drop_while	
std::ranges::split_view	使用分隔符拆分视图。
std::views::split	
std::ranges::common_view	将一个视图转换为 std::ranges::common_range。
std::views::common	
std::ranges::reverse_view	倒序迭代视图。
std::views::reverse	
std::ranges::basic_istream_view	输入流上应用操作符 >>。
std::ranges::istream_view	
std::ranges::elements_view	对元组的第 n 个元素创建视图。
std::views::elements	
std::ranges::keys_view	对类似 pair 的第一个元素创建视图。
std::views::keys	
std::ranges::values_view	对类似 pair 的第二个元素创建视图。
std::views::values	

通常，可以直接使用 std::views::transform，并将其命名为 std::ranges::transform_view。

5.1.2 在容器上使用

标准模板库 (STL) 的算法有时有点不方便，同时需要开始和结束迭代器。

```

1 // sortClassical.cpp
2
3 #include <algorithm>
4 #include <iostream>
5 #include <vector>
```

```

6
7 int main() {
8     std::vector<int> myVec{-3, 5, 0, 7, -4};
9     std::sort(myVec.begin(), myVec.end());
10    for (auto v: myVec) std::cout << v << " "; // -4, -3, 0, 5, 7
11 }

```

若 `std::sort` 可以在整个容器上执行，这不是很好吗？因为范围库，这在 C++20 中是可能的。

范围库的算法直接对容器进行操作

```

1 // sortRanges.cpp
2
3 #include <algorithm>
4 #include <iostream>
5 #include <vector>
6
7 int main() {
8     std::vector<int> myVec{-3, 5, 0, 7, -4};
9     std::ranges::sort(myVec);
10    for (auto v: myVec) std::cout << v << " "; // -4, -3, 0, 5, 7
11 }

```

算法库的算法包含在 `<algorithm>` 头文件中，例如 `std::sort`，其有一个范围版本 `std::ranges::sort`。当了解 `std::ranges::sort` 的重载时，会注意到它们支持投影。

5.1.2.1 投影

`std::ranges::sort` 有两个重载：

```

1 template< std::random_access_iterator I, std::sentinel_for<I> S,
2           class Comp = ranges::less, class Proj = std::identity >
3 requires std::sortable<I, Comp, Proj>
4 constexpr I sort( I first, S last, Comp comp = {}, Proj proj = {} );
5
6 template< ranges::random_access_range R, class Comp = ranges::less,
7           class Proj = std::identity >
8 requires std::sortable<ranges::iterator_t<R>, Comp, Proj>
9 constexpr ranges::borrowed_iterator_t<R> sort( R& r, Comp comp = {}, Proj proj = {} \ )
10

```

第二个重载需要一个可排序范围 `R`、一个谓词 `Comp` 和一个投影 `Proj`。谓词 `Comp` 默认使用 `ranges::less`，投影 `Proj` 用于特征式。投影是集合到子集的映射：

```

1 // rangeProjection.cpp
2
3 #include <algorithm>
4 #include <functional>
5 #include <iostream>
6 #include <vector>
7

```

```

8 struct PhoneBookEntry{
9     std::string name;
10    int number;
11};
12
13 void printPhoneBook(const std::vector<PhoneBookEntry>& phoneBook) {
14     for (const auto& entry: phoneBook) std::cout << "(" << entry.name << ", "
15                                         << entry.number << ")";
16     std::cout << "\n\n";
17 }
18
19 int main() {
20     std::cout << '\n';
21
22     std::vector<PhoneBookEntry> phoneBook{ {"Brown", 111}, {"Smith", 444},
23         {"Grimm", 666}, {"Butcher", 222}, {"Taylor", 555}, {"Wilson", 333} };
24
25     std::ranges::sort(phoneBook, {}, &PhoneBookEntry::name); // ascending by name
26     printPhoneBook(phoneBook);
27
28     std::ranges::sort(phoneBook, std::ranges::greater(),
29                         &PhoneBookEntry::name); // descending by name
30     printPhoneBook(phoneBook);
31
32     std::ranges::sort(phoneBook, {}, &PhoneBookEntry::number); // ascending by number
33     printPhoneBook(phoneBook);
34
35     std::ranges::sort(phoneBook, std::ranges::greater(),
36                         &PhoneBookEntry::number); // descending by number
37     printPhoneBook(phoneBook);
38
39     std::cout << '\n';
40 }

```

phoneBook(第 23 行) 由 PhoneBookEntry 类型的结构体(第 8 行)组成。PhoneBookEntry 由名称和一个数字组成。由于有了投影, phoneBook 可以按照姓名升序(第 26 行)、姓名降序(第 29 行)、数字升序(第 33 行)和数字降序(第 36 行)排序。

```
(Brown, 111) (Butcher, 222) (Grimm, 666) (Smith, 444) (Taylor, 555) (Wilson, 333)

(Wilson, 333) (Taylor, 555) (Smith, 444) (Grimm, 666) (Butcher, 222) (Brown, 111)

(Brown, 111) (Butcher, 222) (Wilson, 333) (Smith, 444) (Taylor, 555) (Grimm, 666)

(Grimm, 666) (Taylor, 555) (Smith, 444) (Wilson, 333) (Butcher, 222) (Brown, 111)
```

大多数范围算法都支持投影。

5.1.2.2 键和值的视图

此外，还可以在 std::unordered_map 的键(第 16 行) 和值(第 24 行) 上创建视图。

```
1 // rangesEntireContainer.cpp
2
3 #include <iostream>
4 #include <ranges>
5 #include <string>
6 #include <unordered_map>
7
8 int main() {
9
10 std::unordered_map<std::string, int> freqWord{ {"witch", 25}, {"wizard", 33},
11                 {"tale", 45}, {"dog", 4},
12                 {"cat", 34}, {"fish", 23} };
13
14 std::cout << "Keys:" << '\n';
15 auto names = std::views::keys(freqWord);
16 for (const auto& name : names){ std::cout << name << " "; }
17 std::cout << '\n';
18 for (const auto& name : std::views::keys(freqWord)){ std::cout << name << " "; }
19
20 std::cout << "\n\n";
21
22 std::cout << "Values: " << '\n';
23 auto values = std::views::values(freqWord);
24 for (const auto& value : values){ std::cout << value << " "; }
25 std::cout << '\n';
26 for (const auto& value : std::views::values(freqWord)) {
27     std::cout << value << " ";
28 }
29 }
```

当然，键和值可以直接显示(第 19 和 27 行)，输出是相同的。

```
Keys:
fish cat dog tale wizard witch
fish cat dog tale wizard witch
```

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

直接在容器上工作可能没什么新奇，但支持函数组合和惰性计算会让人眼前一亮。

5.1.3 函数组合

rangesComposition.cpp 中，使用的是 std::map，所以键的顺序至关重要。

```
1 // rangesComposition.cpp
2
3 #include <iostream>
4 #include <ranges>
5 #include <string>
6 #include <map>
7
8 int main() {
9
10    std::map<std::string, int> freqWord{ {"witch", 25}, {"wizard", 33},
11                                {"tale", 45}, {"dog", 4},
12                                {"cat", 34}, {"fish", 23} };
13
14    std::cout << "All words: ";
15    for (const auto& name : std::views::keys(freqWord)) { std::cout << name << " "; }
16
17    std::cout << '\n';
18
19    std::cout << "All words, reverses: ";
20    for (const auto& name : std::views::keys(freqWord)
21         | std::views::reverse) { std::cout << name << " "; }
22
23    std::cout << '\n';
24
25    std::cout << "The first 4 words: ";
26    for (const auto& name : std::views::keys(freqWord)
27         | std::views::take(4)) { std::cout << name << " "; }
28
29    std::cout << '\n';
30
31    std::cout << "All words starting with w: ";
32    auto firstw = [] (const std::string& name) { return name[0] == 'w'; };
33    for (const auto& name : std::views::keys(freqWord)
34         | std::views::filter(firstw)) { std::cout << name << " "; }
35
36    std::cout << '\n';
37
38 }
```

这里只对键感兴趣，所以这里显示了所有的键(第 15 行)、反向的键(第 20 行)、前四行(第 26 行)，以及以字母“w”开头的键(第 32 行)。

下面是程序的输出。

```
All words: cat dog fish tale witch wizard
All words, reversed: wizard witch tale fish dog cat
The first 4 words: cat dog fish tale
All words starting with w: witch wizard
```

管道符号 | 属于语法糖，用于组合函数。组合的方式不仅可以是 C(R)，也可以是 R | C。因此，下面的三行代码等效。

函数组合的三种句法形式

```
1 auto rev1 = std::views::reverse(std::views::keys(freqWord));
2 auto rev2 = std::views::keys(freqWord) | std::views::reverse;
3 auto rev3 = freqWord | std::views::keys | std::views::reverse;
```

5.1.4 惰性计算

std::views::iota 是一个范围工厂，通过连续递增一个初始值来创建一个元素序列。这个序列可以有限，也可以无限。rangesIota.cpp 用 10 个 int 填充 std::vector，从 0 开始。

```
1 // rangesIota.cpp
2
3 #include <iostream>
4 #include <numeric>
5 #include <ranges>
6 #include <vector>
7
8 int main() {
9
10    std::cout << std::boolalpha;
11
12    std::vector<int> vec;
13    std::vector<int> vec2;
14
15    for (int i: std::views::iota(0, 10)) vec.push_back(i);
16
17    for (int i: std::views::iota(0) | std::views::take(10)) vec2.push_back(i);
18
19    std::cout << "vec == vec2: " << (vec == vec2) << '\n';
20
21    for (int i: vec) std::cout << i << " ";
22
23 }
```

第一个 iota 调用 (第 15 行) 创建从 0 到 9 的所有数字，加 1，第二个 iota 调用 (第 17 行) 创建一个无限数据流，从 0 开始，加 1。std::views::iota(0) 是惰性的。只有当请求时，才会得到一个新值。我请求了十次，所以两个 vector 相同。

```
vec == vec2: true
0 1 2 3 4 5 6 7 8 9
```

现在，我想来个小挑战：找出以 1,000,000 以内，前 20 个质数。

```
// rangesLazy.cpp

#include <iostream>
#include <ranges>

bool isPrime(int i) {
    for (int j=2; j*j <= i; ++j){
        if (i % j == 0) return false;
    }
    return true;
}

int main() {

    std::cout << "Numbers from 1'000'000 to 1'001'000 (displayed each 100th): "
          << '\n';
    for (int i: std::views::iota(1'000'000, 1'001'000)) {
        if (i % 100 == 0) std::cout << i << " ";
    }

    std::cout << "\n\n";

    auto odd = [] (int i){ return i % 2 == 1; };
    std::cout << "Odd numbers from 1'000'000 to 1'001'000 (displayed each 100th): "
          << '\n';
    for (int i: std::views::iota(1'000'000, 1'001'000) | std::views::filter(odd)) {
        if (i % 100 == 1) std::cout << i << " ";
    }

    std::cout << "\n\n";
    std::cout << "Prime numbers from 1'000'000 to 1'001'000: " << '\n';
    for (int i: std::views::iota(1'000'000, 1'001'000) | std::views::filter(odd)
                                                 | std::views::filter(isPrime)) {
        std::cout << i << " ";
    }

    std::cout << "\n\n";

    std::cout << "20 prime numbers starting with 1'000'000: " << '\n';
    for (int i: std::views::iota(1'000'000) | std::views::filter(odd)
                                 | std::views::filter(isPrime)
                                 | std::views::take(20)) {
        std::cout << i << " ";
```

```
44     }
45
46     std::cout << '\n';
47
48 }
```

这是我的迭代策略：

- 第 18 行：我不知道什么时候有 20 个大于 1000000 的质数。安全起见，我创建了 1000 个数字。为了方便观察，每次只展示 100 个。
- 第 27 行：我只对奇数感兴趣，所以去掉偶数。
- 第 34 行：现在，是时候使用下一个 filter 了，谓词 isPrime(第 7 行) 返回一个数字是否为素数。正如下面的截图所示，获得了 75 个质数。
- 第 42 行：懒惰是一种美德。我使用 std::iota 作为无限大的数字工厂，从 1000000 开始，精确地要求 20 个质数。

```
Numbers from 1'000'000 to 1'001'000 (displayed each 100th):
1000000 1000100 1000200 1000300 1000400 1000500 1000600 1000700 1000800 1000900

Odd numbers from 1'000'000 to 1'001'000 (displayed each 100th):
1000001 1000101 1000201 1000301 1000401 1000501 1000601 1000701 1000801 1000901

Prime numbers from 1'000'000 to 1'001'000:
1000003 1000033 1000037 1000039 1000081 1000099 1000117 1000121 1000133 1000151
1000159 1000171 1000183 1000187 1000193 1000199 1000211 1000213 1000231 1000249
1000253 1000273 1000289 1000291 1000303 1000313 1000333 1000357 1000367 1000381
1000393 1000397 1000403 1000409 1000423 1000427 1000429 1000453 1000457 1000507
1000537 1000541 1000547 1000577 1000579 1000589 1000609 1000619 1000621 1000639
1000651 1000667 1000669 1000679 1000691 1000697 1000721 1000723 1000763 1000777
1000793 1000829 1000847 1000849 1000859 1000861 1000889 1000907 1000919 1000921
1000931 1000969 1000973 1000981 1000999

20 prime numbers starting with 1'000'000:
1000003 1000033 1000037 1000039 1000081 1000099 1000117 1000121 1000133 1000151
1000159 1000171 1000183 1000187 1000193 1000199 1000211 1000213 1000231 1000249
```

找出以 1,000,000 以内，前 20 个质数

5.1.5 自定义视图

5.1.5.1 std::ranges::view_interface

有了 `std::ranges::view_interface` 辅助类，可以更容易对视图进行定义。为了实现视图，自定义视图至少需要一个默认构造函数，以及成员函数 `begin()` 和 `end()`：

```
1 class MyView : public std::ranges::view_interface<MyView> {
2 public:
3     auto begin() const { /*...*/ }
```

```
4     auto end() const { /*...*/ }
5 }
```

通过从辅助类 `std::ranges::view_interface` 中 `public` 派生的 `MyView`，并将其作为模板参数，`MyView` 变成了一个视图。这种将类模板本身作为模板参数的技术称为[奇异的循环模板模式](#)(简称 CRTP)。

下一个示例中，我将使用这种技术和标准模板库的容器创建视图。

5.1.5.2 容器视图

视图 `ContainerView` 可以在任意容器上创建视图。

```
1 // containerView.cpp
2
3 #include <iostream>
4 #include <ranges>
5 #include <string>
6 #include <vector>
7
8 template<std::ranges::input_range Range>
9 requires std::ranges::view<Range>
10 class ContainerView : public std::ranges::view_interface<ContainerView<Range>> {
11 private:
12     Range range_{};
13     std::ranges::iterator_t<Range> begin_{ std::begin(range_) };
14     std::ranges::iterator_t<Range> end_{ std::end(range_) };
15
16 public:
17     ContainerView() = default;
18
19     constexpr ContainerView(Range r) : range_(std::move(r)) ,
20                                         begin_(std::begin(r)), end_(std::end(r)) {}
21
22     constexpr auto begin() const {
23         return begin_;
24     }
25     constexpr auto end() const {
26         return end_;
27     }
28 };
29
30 template<typename Range>
31 ContainerView(Range&& range) -> ContainerView<std::ranges::views::all_t<Range>>;
32
33 int main() {
34
35     std::vector<int> myVec{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
36
37     auto myContainerView = ContainerView(myVec);
38     for (auto c : myContainerView) std::cout << c << " ";

```

```

39
40     std::cout << '\n';
41
42     for (auto i : std::views::reverse(ContainerView(myVec))) std::cout << i << ' ';
43     std::cout << '\n';
44
45     for (auto i : ContainerView(myVec) | std::views::reverse) std::cout << i << ' ';
46     std::cout << '\n';
47
48     std::cout << std::endl;
49
50     std::string myStr = "Only for testing purpose.";
51
52     auto myContainerView2 = ContainerView(myStr);
53     for (auto c: myContainerView2) std::cout << c << " ";
54     std::cout << '\n';
55
56     for (auto i : std::views::reverse(ContainerView(myStr))) std::cout << i << ' ';
57     std::cout << '\n';
58
59     for (auto i : ContainerView(myStr) | std::views::reverse) std::cout << i << ' ';
60     std::cout << '\n';
61
62 }

```

类模板 ContainerView(第 8 行) 派生自辅助类 std::ranges::view_interface，并且要求容器支持 std::ranges::view(第 9 行)。剩下的最小的实现很简单，ContainerView 有一个默认构造函数(第 17 行)，两个必需的成员函数 begin()(第 22 行) 和 end()(第 25 行)。方便起见，我添加了一个用户定义的类模板参数推断指南(第 31 行)。

main 函数中，我将在 std::vector(第 37 行) 和 std::string(第 49 行) 上使用 ContainerView，并向前和向后遍历它们。

```

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

only for testing purpose.
.esoprup gnitset rof ylnO
.esoprup gnitset rof ylnO

```

这里提到了“类模板参数推断指南”，对于这个概念容我多说几句。

类模板参数推断指南

C++17 开始，编译器可以从模板实参推导出模板形参。模板推导指南是编译器用于推导模板参数的模式。

当使用 ContainerView(myVec) 时，编译器应用以下用户定义的推导指南：

ContainerView 中用户自定义的推导指南

```
1 template<class Range>
2 ContainerView(Range&& range) -> ContainerView<std::ranges::views::all_t<Range>>;
```

本质上，使用 Container(myVec) 会导致编译器实例化箭头-> 右侧的代码：

应用容器推导指南 (myVec)

```
1 ContainerView<std::ranges::views::all_t<std::vector<int>>>(myVec);
```

cppreference.com 提供了类模板的用户定义推导指南的更多信息。

关于范围库的下一节中，我想进行一个小实验。可以在 C++ 中加入 Python 吗？

5.1.6 Python 的味道

编程语言[Python](#)具有方便的过滤器和映射函数。

- 过滤器 (filter): 对可迭代对象的所有元素应用谓词，并返回谓词返回 `true` 的那些元素
- 映射 (map): 对可迭代对象的所有元素使用一个函数，并返回一个包含转换后元素的新可迭代对象

C++ 中的可迭代对象，也可以在基于范围的 `for` 循环中使用。

此外，Python 允许列表解析中组合这两个函数。

- 列表解析: 将过滤器和映射应用于可迭代对象，并返回一个新的可迭代对象

这就是我的挑战：尝试在 C++20 中使用范围库实，现类似 Python 的函数过滤器、映射，以及列表解析。

5.1.6.1 filter

Python 的 `filter` 可以直接映射到相应的范围函数。

C++ 中的 filter

```
1 / filterRanges.cpp
2
3 #include <iostream>
4 #include <numeric>
5 #include <ranges>
6 #include <string>
7 #include <vector>
8
9 template <typename Func, typename Seq>
10 auto filter(Func func, const Seq& seq) {
11
12     typedef typename Seq::value_type value_type;
```

```

14     std::vector<value_type> result{};
15     for (auto i : seq | std::views::filter(func)) result.push_back(i);
16
17     return result;
18 }
19
20
21 int main() {
22
23     std::cout << '\n';
24
25     std::vector<int> myInts(50);
26     std::iota(myInts.begin(), myInts.end(), 1);
27     auto res = filter([](int i){ return (i % 3) == 0; }, myInts);
28     for (auto v: res) std::cout << v << " ";
29
30
31     std::vector<std::string> myStrings{"Only", "for", "testing", "purposes"};
32     auto res2 = filter([](const std::string& s){ return std::isupper(s[0]); },
33                         myStrings);
34
35     std::cout << "\n\n";
36
37     for (auto word: res2) std::cout << word << '\n';
38
39     std::cout << '\n';
40
41 }
```

解释程序之前，先展示输出效果。

```
3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48
```

```
Only
```

`filter`(第 9 行) 的可读性还不错，第 12 行检测底层元素的类型。我只是将 `func` 应用到序列的每个元素，并返回 `std::vector` 中的元素。第 27 行选择从 1 到 50 的所有数字 `i`，`i` 需要满足条件 `(i % 3) == 0`。只有以大写字母开头的字符串，才能通过第 32 行中的过滤器。

5.1.6.2 map

`map` 对输入序列的每个元素，使用同一个可调用对象。

C++ 中的 `map`

```

1 // mapRanges.cpp
2
3 #include <iostream>
4 #include <list>
5 #include <ranges>
6 #include <string>
7 #include <vector>
8 #include <utility>
9
10
11 template <typename Func, typename Seq>
12 auto map(Func func, const Seq& seq) {
13
14     typedef typename Seq::value_type value_type;
15     using return_type = decltype(func(std::declval<value_type>()));
16
17     std::vector<return_type> result{};
18     for (auto i : seq | std::views::transform(func)) result.push_back(i);
19
20     return result;
21 }
22
23 int main() {
24
25     std::cout << '\n';
26
27     std::list<int> myInts{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
28     auto res = map([](int i){ return i * i; }, myInts);
29
30     for (auto v: res) std::cout << v << " ";
31
32     std::cout << "\n\n";
33
34     std::vector<std::string> myStrings{"Only", "for", "testing", "purposes"};
35     auto res2 = map([](const std::string& s){ return std::make_pair(s.size(), s); },
36                           myStrings);
37
38     for (auto p: res2) std::cout << "(" << p.first << ", " << p.second << ")";
39
40     std::cout << "\n\n";
41
42 }
```

map 函数定义中的第 15 行非常有趣。表达式 `decltype(func(std::declval<value_type>()))` 推导出 `return_type`。若函数 `func` 应用于输入序列的所有元素，则输入类型将转换为 `return_type`。`std::declval<value_type>()` 返回一个右值引用，`decltype` 可以使用它来推断类型。使用 `map([](int i){return i * i;}, myInts)(第 28 行)` 将 `myInt` 的每个元素映射为其平方值，并且使用 `map([](const std::string& s){return std::make_pair(s.size(), s);}, myStrings)` 将 `myStrings` 中的每个字符串映射为

pair，每个 pair 的第一个元素是字符串的长度。

```
1 4 9 16 25 36 49 64 81 100  
  
(4, Only) (3, for) (7, testing) (8, purposes)
```

5.1.6.3 列表解析

listComprehensionRanges.cpp 是对 Python 列表解析算法实现的简化版本。

map 对输入序列的每个元素使用同一个可调用对象。

```
1 // listComprehensionRanges.cpp  
2  
3 #include <algorithm>  
4 #include <cctype>  
5 #include <functional>  
6 #include <iostream>  
7 #include <ranges>  
8 #include <string>  
9 #include <vector>  
10 #include <utility>  
11  
12 template <typename T>  
13 struct AlwaysTrue {  
14     constexpr bool operator()(const T&) const {  
15         return true;  
16     }  
17 } ;  
18  
19 template <typename Map, typename Seq, typename Filt = AlwaysTrue<  
20                         typename Seq::value_type>>  
21 auto mapFilter(Map map, Seq seq, Filt filt = Filt()) {  
22  
23     typedef typename Seq::value_type value_type;  
24     using return_type = decltype(map(std::declval<value_type>()));  
25  
26     std::vector<return_type> result{};  
27     for (auto i : seq | std::views::filter(filt)  
28          | std::views::transform(map)) result.push_back(i);  
29     return result;  
30 }  
31  
32 int main() {  
33  
34     std::cout << '\n';  
35  
36     std::vector myInts{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

```

37
38 auto res = mapFilter([](int i){ return i * i; }, myInts);
39 for (auto v: res) std::cout << v << " ";
40
41 std::cout << "\n\n";
42
43 res = mapFilter([](int i){ return i * i; }, myInts,
44 [](auto i){ return i % 2 == 1; });
45 for (auto v: res) std::cout << v << " ";
46
47 std::cout << "\n\n";
48
49 std::vector<std::string> myStrings{"Only", "for", "testing", "purposes"};
50 auto res2 = mapFilter([](const std::string& s){
51             return std::make_pair(s.size(), s);
52         }, myStrings);
53 for (auto p: res2) std::cout << "(" << p.first << ", " << p.second << ")" " ;
54
55 std::cout << "\n\n";
56
57 myStrings = {"Only", "for", "testing", "purposes"};
58 res2 = mapFilter([](const std::string& s){
59             return std::make_pair(s.size(), s);
60         }, myStrings,
61         [](const std::string& word){ return std::isupper(word[0]); });
62
63 for (auto p: res2) std::cout << "(" << p.first << ", " << p.second << ")" " ;
64
65 std::cout << "\n\n";
66
67 }

```

过滤器函数使用的默认谓词(第 19 行)总返回 true(第 12 行), 所以 mapFilter 函数在默认情况下只是作为一个映射函数, 所以 mapFilter 函数在第 38 行和第 50 行中的行为与前面的 map 函数相同。第 42 和 55 行在一次调用中同时应用 map 和 filter 函数。

```

1 4 9 16 25 36 49 64 81 100

1 9 25 49 81

(4, Only) (3, for) (7, testing) (8, purposes)

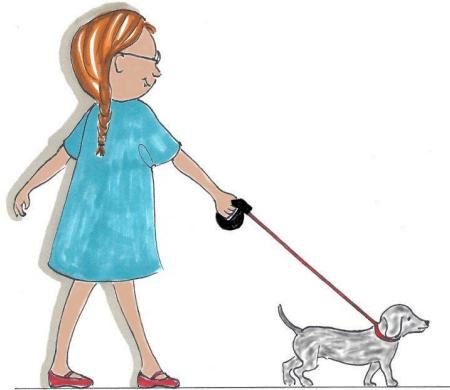
(4, Only)

```

总结

- 范围库为我们提供了 STL 算法的特殊版本。范围库算法是惰性的，可以直接工作在容器上，并且可以组合。
- 范围库的算法
 - 惰性，因此可以在无限的数据流上调用。
 - 可以直接操作容器，不需要由两个迭代器定义的范围。
 - 可以使用管道 ($\|$) 符号进行组合。

5.2. std::span



Cippi 在遛狗

std::span 表示一个对象，该对象引用一个连续的对象序列。std::span 有时也称为视图，其不是底层数据的所有者。这个连续的对象序列可以是一个普通的 C 数组、有长度信息的指针、std::array、std::vector 或 std::string。

std::span 可以具有静态范围或动态范围。默认情况下，std::span 有一个动态范围：

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

5.2.1 静态范围与动态范围

std::span 有一个静态范围时，其大小在编译时已知，并且是类型 std::span<T, size> 的一部分。因此，其实现只需要一个指向连续对象序列的第一个元素的指针。

具有动态范围的 std::span，由指向第一个元素的指针和连续对象序列的长度组成，但长度信息不是类型 std::span<T> 的一部分。

staticDynamicExtentSpan.cpp 展示了这两种视图之间的区别。

```
1 // staticDynamicExtentSpan.cpp
2
3 #include <iostream>
4 #include <span>
5 #include <vector>
6
7 void printMe(std::span<int> container) {
8
9     std::cout << "container.size(): " << container.size() << '\n';
10    for (auto e : container) std::cout << e << ' ';
11    std::cout << "\n\n";
12 }
13
14 int main() {
```

```

16 std::cout << '\n';
17
18 std::vector myVec1{1, 2, 3, 4, 5};
19 std::vector myVec2{6, 7, 8, 9};
20
21 std::span<int> dynamicSpan(myVec1);
22 std::span<int, 4> staticSpan(myVec2);
23
24 printMe(dynamicSpan);
25 printMe(staticSpan); // implicitly converted into a dynamic span
26
27 // staticSpan = dynamicSpan; ERROR
28 dynamicSpan = staticSpan;
29
30 printMe(staticSpan);
31
32 std::cout << '\n';
33
34 }

```

dynamicSpan(第 21 行)有一个动态范围，而 staticSpan(第 22 行)有一个静态范围。两个 std::span 都在 printMe 函数中返回它们的长度(第 9 行)。具有动态范围的 std::span 可以赋值给具有静态范围的 std::span，但不能反过来。第 27 行会导致错误，而第 7、25 和 28 行是可以工作的。

std::span 的静态范围与动态范围

使用 std::span<T> 的重要原因是，若将普通 C 数组退化成指针，那么长度信息就会丢失。这种退化是 C/C++ 中出现错误的原因之一。

5.2.2 自行推断连续对象序列的长度

与 C 数组相反，std::span<T> 可以自行推断对象连续序列的长度。

```

1 // printSpan.cpp
2

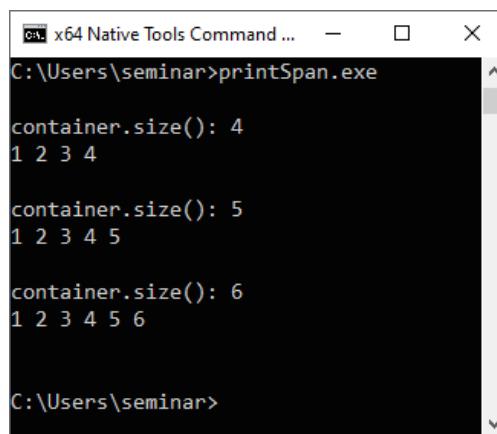
```

```

3 #include <iostream>
4 #include <vector>
5 #include <array>
6 #include <span>
7
8 void printMe(std::span<int> container) {
9
10    std::cout << "container.size(): " << container.size() << '\n';
11    for (auto e : container) std::cout << e << ' ';
12    std::cout << "\n\n";
13}
14
15 int main() {
16
17    std::cout << '\n';
18
19    int arr[] {1, 2, 3, 4};
20    printMe(arr);
21
22    std::vector vec{1, 2, 3, 4, 5};
23    printMe(vec);
24
25    std::array arr2{1, 2, 3, 4, 5, 6};
26    printMe(arr2);
27
28}

```

C-array(第 19 行)、std::vector(第 22 行) 和 std::array(第 25 行) 元素类型都是 int，所以 std::span 使用 int 类型的特化。这个简单的例子中，还有一些更有趣的东西。对于每个容器，std::span 可以推断其长度(第 10 行)。



创建 std::span 的方法有很多。

5.2.3 用指针和长度来创建 std::span

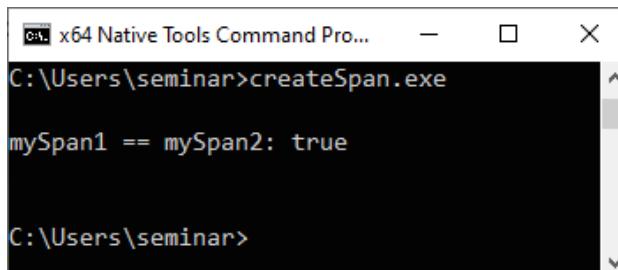
可以用指针和内存长度来创建 std::span。

```

1 // createSpan.cpp
2
3 #include <algorithm>
4 #include <iostream>
5 #include <span>
6 #include <vector>
7
8 int main() {
9
10    std::cout << '\n';
11    std::cout << std::boolalpha;
12
13    std::vector myVec{1, 2, 3, 4, 5};
14
15    std::span mySpan1{myVec};
16    std::span mySpan2{myVec.data(), myVec.size()};
17
18    bool spansEqual = std::equal(mySpan1.begin(), mySpan1.end(),
19                                mySpan2.begin(), mySpan2.end());
20
21    std::cout << "mySpan1 == mySpan2: " << spansEqual << '\n';
22
23    std::cout << '\n';
24
25 }

```

由 `std::vector` 创建的 `mySpan1`(第 15 行) 和由指针和长度创建的 `mySpan2`(第 16 行) 相同(第 21 行)。



用指针和长度值来创建 `std::span`

`std::span` 既不是 `std::string_view` 也不是视图

您可能还记得 `std::span` 有时会认为是视图，但不要将 `std::span` 与范围库中的视图或`std::string_view`混淆。

范围库中的视图可以对特定范围内的元素执行一些操作。而视图不拥有数据，其复制、移动和赋值的时间成本固定。`std::span` 和 `std::string_view` 虽不是视图，但也可以处理字符串。

`std::span` 和 `std::string_view` 的主要区别是，`std::span` 可以修改其引用的对象。

5.2.4 修改引用对象

可以修改整个 span，也可以只修改子 span。修改 span 时，也会修改引用的对象。

下面的代码展示了如何使用子 span 来修改 std::vector 中引用的对象。

```
1 // spanTransform.cpp
2
3 #include <algorithm>
4 #include <iostream>
5 #include <vector>
6 #include <span>
7
8 void printMe(std::span<int> container) {
9
10    std::cout << "container.size(): " << container.size() << '\n';
11    for (auto e : container) std::cout << e << ' ';
12    std::cout << "\n\n";
13 }
14
15 int main() {
16
17    std::cout << '\n';
18
19    std::vector vec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
20    printMe(vec);
21
22    std::span span1(vec);
23    std::span span2{span1.subspan(1, span1.size() - 2)};
24
25
26    std::transform(span2.begin(), span2.end(),
27                  span2.begin(),
28                  [] (int i){ return i * i; });
29
30
31    printMe(vec);
32    printMe(span1);
33
34 }
```

span1 引用 std::vector vec(第 22 行)。相反，span2 只引用底层 vec 的元素，不包括第一个和最后一个元素(第 23 行)。所以，只对这些元素进行平方的映射(第 26 行)。

```
C:\Users\seminar>spanTransform.exe
container.size(): 10
1 2 3 4 5 6 7 8 9 10

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

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

C:\Users\seminar>
```

修改 std::span 的引用对象

其实，有很多方便的函数可以处理 std::span 的元素。

5.2.5 处理 std::span 的元素

下表给出了用于引用 std::span 元素的接口。

std::span spse 的接口

接口	描述
sp.front()	访问第一个元素。
sp.back()	访问最后一个元素。
sp[i]	访问第 i 个元素。
sp.data()	返回一个指向序列开头的指针。
sp.size()	返回元素的个数。
sp.size_bytes()	以字节为单位返回序列的大小。
sp.empty()	若为空，则返回 true。
sp.first<count>() sp.frist(count)	返回由序列前 count 个元素组成的子 span。
sp.last<count>() sp.last(count)	返回由序列后 count 个元素组成的子 span。
sp.subspan<first, count>() sp.subspan(first, count)	返回由 first 之后的 count 个元素组成的子 span。

subspan.cpp 展示了成员函数 subspan 的用法。

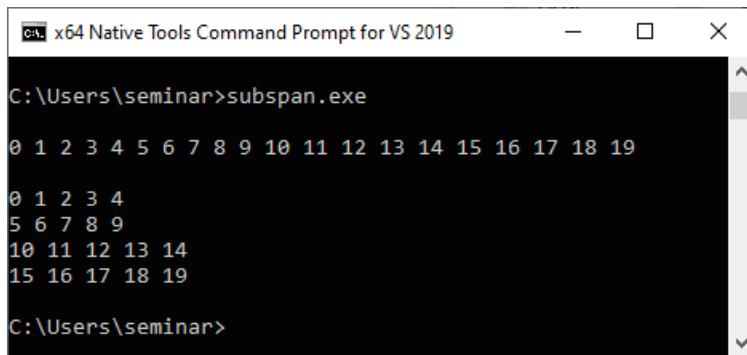
```
1 // subspan.cpp
2
3 #include <iostream>
4 #include <numeric>
5 #include <span>
6 #include <vector>
```

```

7
8 int main() {
9
10    std::cout << '\n';
11
12    std::vector<int> myVec(20);
13    std::iota(myVec.begin(), myVec.end(), 0);
14    for (auto v: myVec) std::cout << v << " ";
15
16    std::cout << "\n\n";
17
18    std::span<int> mySpan(myVec);
19    auto length = mySpan.size();
20
21    std::size_t count = 5;
22    for (std::size_t first = 0; first <= (length - count); first += count) {
23        for (auto ele: mySpan.subspan(first, count)) std::cout << ele << " ";
24        std::cout << '\n';
25    }
26
27 }

```

第 13 行使用算法`std::iota`用从 0 到 19 的所有数字填充 vector(第 13 行), 这个 vector 用于初始化`std::span`(第 18 行)。最后, for 循环(第 22 行) 使用`subspan`函数创建子 span, 从第一个开始直到使用`mySpan`为止, 子 span 都包含 count 个元素。



使用成员函数 `subspan`

Kilian Henneberger 给我说了一个`std::span`的特殊用例, 可修改元素的常量范围。

5.2.6 可修改元素的常量范围

简单起见, 我将`std::vector`和`std::span`命名为一个范围。与`std::string`类似, `std::vector`对可修改元素的可修改范围进行建模:`std::vector<T>`。当把`std::vector`声明为`const`时, 范围模型就表示常量对象的常量范围: 不能对可修改元素的常量范围进行建模。`std::span`对一个常量范围的可修改对象建模:`std::span<T>`。下表强调了(常量/可修改)范围和(常量/可修改)元素的变化。

(常量/可修改)元素的(常量/可修改)范围

	可修改的元素	常量元素
可修改的范围	std::vector<T>	
常量范围	std::span<T>	const std::vector<T> std::span<const T>

constRangeModifiableElements.cpp 展示了每种组合。

```

1 // constRangeModifiableElements.cpp
2
3 #include <iostream>
4 #include <span>
5 #include <vector>
6
7 void printMe(std::span<int> container) {
8
9     std::cout << "container.size(): " << container.size() << '\n';
10    for (auto e : container) std::cout << e << ' ';
11    std::cout << "\n\n";
12 }
13
14 int main() {
15
16     std::cout << '\n';
17
18     std::vector<int> origVec{1, 2, 2, 4, 5};
19
20     // Modifiable range of modifiable elements
21     std::vector<int> dynamVec = origVec;
22     dynamVec[2] = 3;
23     dynamVec.push_back(6);
24     printMe(dynamVec);
25
26     // Constant range of constant elements
27     const std::vector<int> constVec = origVec;
28     // constVec[2] = 3; ERROR
29     // constVec.push_back(6); ERROR
30     std::span<const int> constSpan(origVec);
31     // constSpan[2] = 3; ERROR
32
33     // Constant range of modifiable elements
34     std::span<int> dynamSpan{origVec};
35     dynamSpan[2] = 3;
36     printMe(dynamSpan);
37
38     std::cout << '\n';
39
40 }
```

vector dynamVec(第 21 行) 是可修改元素的可修改范围。这个观察结果不适用于 vector

`constVec`(第 27 行), `constVec` 不能改变元素的多少, `constSpan`(第 30 行) 的行为也一样。`dynamSpan` 为常量范围的可修改元素提供了一种方式。

```
x64 Native Tools Command Prompt for VS 2019
C:\Users\seminar>constRangeModifiableElements.exe

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

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

C:\Users\seminar>
```

(常量/可修改) 元素的 (常量/可修改) 范围

总结

- `std::span` 是一个引用连续对象序列的对象。`std::span` 也称为视图, 从来不是数据的所有者, 因此不分配内存。对象的连续序列可以是一个 C 数组、带内存长度的指针、`std::array`、`std::vector` 或 `std::string`。
- 与 C 数组相反, `std::span` 可以自动推断其引用对象序列的长度。
- `std::span` 修改其元素时, 也会修改其引用的对象。

5.3. 容器的改进



Cippi 在检查集装箱

C++20 对标准模板库的容器还进行了很多改进。首先，`std::vector` 和 `std::string` 添加了 `constexpr` 构造函数，从而可以在编译时使用。所有容器都支持统一的 `erase` 成员函数，以及关联容器 (`map`/`set`) 添加了成员函数 `contains`。此外，`std::string` 可以对前缀或后缀进行检查。

5.3.1 `constexpr` 的容器和算法

C++20 支持 `constexpr` 容器 `std::vector` 和 `std::string`，其中 `constexpr` 表明这两个容器的成员函数都可以在编译时使用。此外，标准模板库中，声明为 `constexpr` 的算法超过了 100 个。

因此，可以在编译时对 `int` 类型的 `std::vector` 进行排序。

```
1 // constexprVector.cpp
2
3 #include <algorithm>
4 #include <iostream>
5 #include <vector>
6
7 constexpr int maxElement() {
8     std::vector myVec = {1, 2, 4, 3};
9     std::sort(myVec.begin(), myVec.end());
10    return myVec.back();
11 }
12 int main() {
13
14     std::cout << '\n';
15
16     constexpr int maxValue = maxElement();
17     std::cout << "maxValue: " << maxValue << '\n';
18 }
```

```

19 constexpr int maxValue2 = [] {
20     std::vector myVec = {1, 2, 4, 3};
21     std::sort(myVec.begin(), myVec.end());
22     return myVec.back();
23 }();
24
25 std::cout << "maxValue2: " << maxValue2 << '\n';
26
27 std::cout << '\n';
28
29 }

```

两个 `std::vector`(第 8 行和第 20 行) 在编译时使用 `constexpr` 声明的函数进行排序。第一种情况下，函数 `maxElement` 返回 `myVec` 的最后一个元素，也就是它的最大值。第二种情况下，使用声明为 `constexpr` 的 Lambda 表达式。

```

maxValue: 4
maxValue2: 4

```

5.3.2 `std::array`

C++20 提供了两种创建数组的方法。`std::to_array` 可以创建 `std::array`，而 `std::make_shared` 现在可以创建 `std::shared_ptr` 数组了。

5.3.2.1 `std::to_array`

`std::to_array` 从现有一维数组创建 `std::array`，其中的元素是从现有一维数组复制过来的。

一维数组可以是 C-string、`std::initializer_list` 或 `std::pair` 的一维数组。下面的例子来自cppreference.com/to_array。

```

1 // toArray.cpp
2
3 #include <iostream>
4 #include <utility>
5 #include <array>
6 #include <memory>
7
8 int main() {
9
10    std::cout << '\n';
11
12    auto arr1 = std::to_array("A simple test");
13    for (auto a: arr1) std::cout << a;
14    std::cout << "\n\n";
15
16    auto arr2 = std::to_array({1, 2, 3, 4, 5});
17    for (auto a: arr2) std::cout << a;

```

```

18 std::cout << "\n\n";
19
20 auto arr3 = std::to_array<double>({0, 1, 3});
21 for (auto a: arr3) std::cout << a;
22 std::cout << '\n';
23 std::cout << "typeid(arr3[0]).name(): " << typeid(arr3[0]).name() << '\n';
24 std::cout << '\n';
25
26 auto arr4 = std::to_array<std::pair<int, double>>({{1, 0.0}, {2, 5.1},
27 {3, 5.1}});
28 for (auto p: arr4) {
29     std::cout << "(" << p.first << ", " << p.second << ")" << '\n';
30 }
31
32 std::cout << "\n\n";
33
34 }

```

使用 C-string(第 12 行), std::initializer_list(第 16 行和第 20 行), std::pair 的 std::initializer_list(第 26 行)都可以创建 std::array。通常, 编译器可以推断出 std::array 的类型, 还可以对类型进行指定(第 20 和 26 行)。

A simple **test**

12345

013

typeid(arr3[0]).name(): d

(1, 0)

(2, 5.1)

(3, 5.1)

5.3.2.2 std::make_shared

C++11 的工厂函数 **std::make_shared** 用于创建 std::shared_ptr。C++20 中, 这个工厂函数终于支持创建 std::shared_ptr 数组了。

- std::shared_ptr<double[]> sha = std::make_shared<double[]>(1024); 创建一个 shared_ptr, 默认初始化 1024 个 double 变量
- std::shared_ptr<double[]> shar = std::make_shared<double[]>(1024, 1.0); 创建了一个 shared_ptr, 其中 1024 个 double 初始化为 1.0

5.3.3 擦除功能

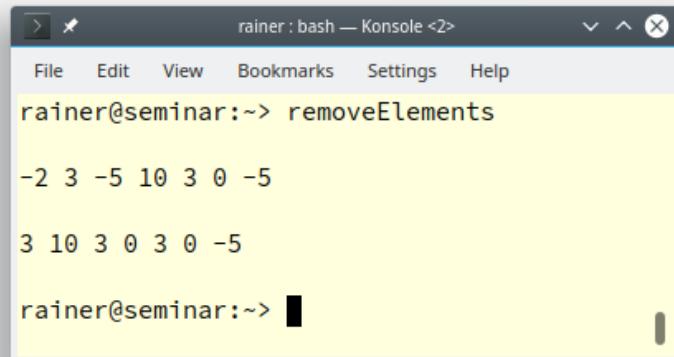
C++20 之前，从容器中删除元素太麻烦了。

5.3.3.1 erase-remove 习语

从容器中删除元素好像很容易。对于 std::vector，可以使用 std::remove_if 函数。

```
1 // removeElements.cpp
2
3 #include <algorithm>
4 #include <iostream>
5 #include <vector>
6
7 int main() {
8
9     std::cout << '\n';
10
11     std::vector myVec{-2, 3, -5, 10, 3, 0, -5};
12
13     for (auto ele: myVec) std::cout << ele << " ";
14     std::cout << "\n\n";
15
16     std::remove_if(myVec.begin(), myVec.end(), [](int ele){ return ele < 0; });
17     for (auto ele: myVec) std::cout << ele << " ";
18
19     std::cout << "\n\n";
20
21 }
```

removeElements.cpp 从 std::vector 中删除所有小于零的元素。很容易，对吧？或许也并不容易；现在，您可能陷入了资深 C++ 程序员所熟知的陷阱。



std::remove_if(第 16 行) 不会删除任何东西，std::vector 的元素数量不变。算法只是返回修改后容器的新(逻辑)末端。

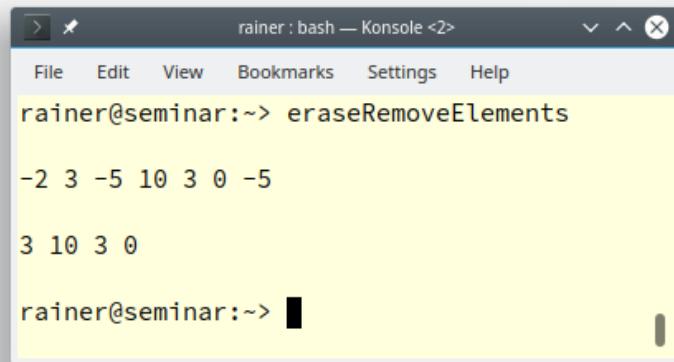
为了修改容器内容，必须将新末端应用于容器。

```

1 // eraseRemoveElements.cpp
2
3 #include <algorithm>
4 #include <iostream>
5 #include <vector>
6
7 int main() {
8
9     std::cout << '\n';
10
11    std::vector myVec{-2, 3, -5, 10, 3, 0, -5};
12
13    for (auto ele: myVec) std::cout << ele << " ";
14    std::cout << "\n\n";
15
16    auto newEnd = std::remove_if(myVec.begin(), myVec.end(),
17        [] (int ele){ return ele < 0; });
18    myVec.erase(newEnd, myVec.end());
19    // myVec.erase(std::remove_if(myVec.begin(), myVec.end(),
20    //                 [] (int ele){ return ele < 0; }), myVec.end());
21    for (auto ele: myVec) std::cout << ele << " ";
22
23    std::cout << "\n\n";
24
25 }

```

第 16 行返回容器 myVec 的新末端 newEnd。这个新末端应用于第 18 行，并从 newEnd 开始删除 myVec 中的元素。当在表达式中应用函数 remove 和 erase(第 19 行)时，就会清楚地了解为什么这种方式称为“erase-remove 习语”了。



因为 C++20 中新增了 `erase` 和 `erase_if`，从容器中擦除元素就方便了许多。

5.3.3.2 C++20 的 `erase` 和 `erase_if`

使用 `erase` 和 `erase_if`, 可以直接对容器进行操作。相比之下, 前面介绍的 `erase-remove` 习语看上去就相当麻烦了: 迭代了两次。

来一起看看新函数 `erase` 和 `erase_if` 在实践中怎么使用。下面的代码, 是从容器中删除一些元素。

```
1 // eraseCpp20.cpp
2
3 #include <iostream>
4 #include <numeric>
5 #include <deque>
6 #include <list>
7 #include <string>
8 #include <vector>
9
10 template <typename Cont>
11 void eraseVal(Cont& cont, int val) {
12     std::erase(cont, val);
13 }
14
15 template <typename Cont, typename Pred>
16 void erasePredicate(Cont& cont, Pred pred) {
17     std::erase_if(cont, pred);
18 }
19
20 template <typename Cont>
21 void printContainer(Cont& cont) {
22     for (auto c: cont) std::cout << c << " ";
23     std::cout << '\n';
24 }
25
26 template <typename Cont>
27 void doAll(Cont& cont) {
28     printContainer(cont);
29     eraseVal(cont, 5);
30     printContainer(cont);
31     erasePredicate(cont, [](auto i) { return i >= 3; });
32     printContainer(cont);
33 }
34
35 int main() {
36
37     std::cout << '\n';
38
39     std::string str{"A Sentence with an E."};
40     std::cout << "str: " << str << '\n';
41     std::erase(str, 'e');
42     std::cout << "str: " << str << '\n';
43     std::erase_if(str, [](char c){ return std::isupper(c); });
44     std::cout << "str: " << str << '\n';
```

```

45 std::cout << "\nstd::vector " << '\n';
46 std::vector vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
47 doAll(vec);
48
49 std::cout << "\nstd::deque " << '\n';
50 std::deque deq{1, 2, 3, 4, 5, 6, 7, 8, 9};
51 doAll(deq);
52
53 std::cout << "\nstd::list" << '\n';
54 std::list lst{1, 2, 3, 4, 5, 6, 7, 8, 9};
55 doAll(lst);
56
57
58 }

```

第 41 行删除给定字符串 str 中的所有'c' 字符。第 43 行将 Lambda 表达式应用于同一字符串，并删除了所有大写字母。

代码的其余部分，删除了 std::vector(第 47 行)、 std::deque(第 51 行) 和 std::list(第 55 行) 的相应元素。对每个容器上应用函数模板 doAll(第 26 行)， doAll 删除元素 5 和所有大于或等于 3 的元素。函数模板 eraseVal(第 10 行) 使用新的 erase 函数，函数模板 erasePredicate(第 15 行) 使用新的函数 erase_if。

```

C:\Users\seminar>eraseCpp20.exe

str: A Sentence with an E.
str: A Sntnc with an E.
str: ntnc with an .

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

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

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

C:\Users\seminar>

```

新函数 erase 和 erase_if 可以应用于标准模板库的所有容器。但这并不适用于另一个函数 contains，其需要对关联容器进行操作。

5.3.4 关联容器的 contains

由于 `contains` 函数，就可以轻松地检查关联容器中是否存在元素。你可能会说，我们已经可以用 `find` 或 `count` 来完成这个任务了。

非也非也，这两个函数都不适合初学者，而且有各自的缺点。

```
1 // checkExistence.cpp
2
3 #include <set>
4 #include <iostream>
5
6 int main() {
7
8     std::cout << '\n';
9
10    std::set mySet{3, 2, 1};
11
12    if (mySet.find(2) != mySet.end()) {
13        std::cout << "2 inside" << '\n';
14    }
15
16    std::multiset myMultiSet{3, 2, 1, 2};
17
18    if (myMultiSet.count(2)) {
19        std::cout << "2 inside" << '\n';
20    }
21
22 }
```

函数产生预期的结果。

```
2 inside
2 inside
```

使用 `find` 和 `count` 检查容器中是否有给定的元素

两个方式都有问题。`find`(第 11 行) 太冗长了，`count`(第 16 行) 也是，`count` 还存在性能上的问题。想知道某个元素是否在容器中时，应该在找到时停止，而不是继续查找直到结束。上面的代码中，`myMultiSet.count(2)` 会返回 2。

与 `find` 和 `count` 不同，C++20 中的 `contains` 成员函数使用起来非常方便。

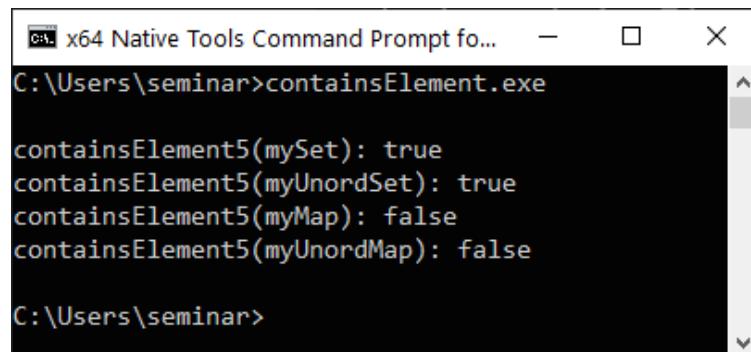
```
1 // containsElement.cpp
2
3 #include <iostream>
4 #include <set>
5 #include <map>
6 #include <unordered_set>
```

```

7 #include <unordered_map>
8
9 template <typename AssocCont>
10 bool containsElement5(const AssocCont& assocCont) {
11     return assocCont.contains(5);
12 }
13
14 int main() {
15
16     std::cout << std::boolalpha;
17
18     std::cout << '\n';
19
20     std::set<int> mySet{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
21     std::cout << "containsElement5(mySet): " << containsElement5(mySet);
22
23     std::cout << '\n';
24
25     std::unordered_set<int> myUnordSet{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
26     std::cout << "containsElement5(myUnordSet): " << containsElement5(myUnordSet);
27
28     std::cout << '\n';
29
30     std::map<int, std::string> myMap{ {1, "red"}, {2, "blue"}, {3, "green"} };
31     std::cout << "containsElement5(myMap): " << containsElement5(myMap);
32
33     std::cout << '\n';
34
35     std::unordered_map<int, std::string> myUnordMap{ {1, "red"}, {2, "blue"}, {3, "green"} };
36     std::cout << "containsElement5(myUnordMap): " << containsElement5(myUnordMap);
37
38     std::cout << '\n';
39 }

```

若关联容器包含键 5，则函数模板 `containsElement5` 返回 `true`。例子中，我只使用了关联容器 `std::set`、`std::unordered_set`、`std::map` 和 `std::unordered_map`，它们都不能多次保存给定的键。



5.3.5 字符串检查前缀和后缀

`std::string` 获取新的成员函数 `starts_with` 和 `ends_with`, 可以检查 `std::string` 是否以指定的子字符串开始或结束。

```
1 // stringStartsWithEndsWith.cpp
2
3 #include <iostream>
4 #include <string_view>
5 #include <string>
6
7 template <typename PrefixType>
8 void startsWith(const std::string& str, PrefixType prefix) {
9     std::cout << " starts with " << prefix << ":" "
10    << str.starts_with(prefix) << '\n';
11 }
12
13 template <typename SuffixType>
14 void endsWith(const std::string& str, SuffixType suffix) {
15     std::cout << " ends with " << suffix << ":" "
16    << str.ends_with(suffix) << '\n';
17 }
18
19 int main() {
20
21     std::cout << '\n';
22
23     std::cout << std::boolalpha;
24
25     std::string helloWorld("Hello World");
26
27     std::cout << helloWorld << '\n';
28
29     startsWith(helloWorld, helloWorld);
30
31     startsWith(helloWorld, std::string_view("Hello"));
32
33     startsWith(helloWorld, 'H');
34
35     std::cout << "\n\n";
36
37     std::cout << helloWorld << '\n';
38
39     endsWith(helloWorld, helloWorld);
40
41     endsWith(helloWorld, std::string_view("World"));
42
43     endsWith(helloWorld, 'd');
44
45 }
```

`starts_with` 和 `ends_with` 的成员函数都是谓词，返回布尔值。可以用 `std::string`(第 29 行和 39 行), `std::string_view`(第 31 行和 41 行) 和 `char`(第 33 行和 43 行) 使用新的成员函数。

```
Hello World
    starts with Hello World: true
    starts with Hello: true
    starts with H: true
```

```
Hello World
    ends with Hello World: true
    ends with World: true
    ends with d: true
```

总结

- `std::vector` 和 `std::string` 具有 `constexpr` 构造函数，可以在编译时实例化。由于标准模板库 (STL) 提供 `constexpr` 算法，可以在编译时执行。
- C++20 提供了两种创建数组的方法。`std::to_array` 创建 `std::array`, `std::make_shared` 可以创建包含数组的 `std::shared_ptr`。
- 新的算法 `std::erase` 和 `std::erase_if` 用于从 STL 容器中删除特定的元素 (`erase`) 或满足谓词 (`erase_if`) 的元素。
- 有了新成员函数 `contains`，就可以直接检查关联容器是否具有所请求的键。
- `std::string` 支持新的成员函数 `start_with` 和 `end_with`，可用来检查容器是否具有特定的前缀或后缀。

5.4. 算术工具



Cippi 在研究算盘

有符号整数和无符号整数的比较，有时是导致意外行为和错误的原因。C++20 添加了新的类型安全的比较函数 `std::cmp_*`，相应的错误就能完全避免。此外，C++20 还包含了一些数学常数，比如 `e`、 π 或 φ ；使用函数 `std::midpoint` 和 `std::lerp`，可以计算两个数的中点或它们的线性插值。新的位操作可以访问和修改单个位或位序列。

5.4.1 整型数的安全比较

比较有符号整数和无符号整数时，可能得不到预期的结果。C++20 添加了六个 `std::cmp_*` 函数，使得比较更加安全。为了更好的了解添加整数安全比较意义，这里先从不安全比较开始说起。

整数与整型

术语 `integral`(整数) 和 `integer`(整型) 是 C++ 中的同义词。这是基本类型标准的表述：“类型 `bool`, `char`, `char8_t`, `char16_t`, `char32_t`, `wchar_t`，以及有符号整型和无符号整型统称为整数类型。整数是某一种的整型具象表达”。本书里，我更喜欢用整型这个词。

5.4.1.1 不安全的比较

当然，下面这段代码的名称为 `unsafecompare.cpp`，命名成这样是有原因的。

```
1 // unsafeComparison.cpp
2
3 #include <iostream>
4
5 int main() {
6
7     std::cout << '\n';
8
9     std::cout << std::boolalpha;
```

```

10
11 int x = -3;
12 unsigned int y = 7;
13
14 std::cout << "-3 < 7: " << (x < y) << '\n';
15 std::cout << "-3 <= 7: " << (x <= y) << '\n';
16 std::cout << "-3 > 7: " << (x > y) << '\n';
17 std::cout << "-3 => 7: " << (x >= y) << '\n';
18
19 std::cout << '\n';
20
21 }

```

当执行时，输出可能不符合期望。

```

-3 < 7: false
-3 <= 7: false
-3 > 7: true
-3 => 7: true

```

不安全的比较结果令人惊讶

看到程序的输出时，会发现-3 比 7 大。我想你大概知道为什么会这样。我比较了有符号的 x(第 11 行) 和无符号的 y(第 12 行)。这背后发生了什么？

解决不安全的整数比较

```

1 // unsafeComparison2.cpp
2
3 int main() {
4     int x = -3;
5     unsigned int y = 7;
6
7     bool val = x < y;
8     static_assert(static_cast<unsigned int>(-3) == 4'294'967'293);
9 }

```

本例中，主要关注小于操作符。[C++ Insights](#)输出如下：

```

1 int main()
2 {
3     int x = -3;
4     unsigned int y = 7;
5     bool val = static_cast<unsigned int>(x) < y;
6     /* PASSED: static_assert(static_cast<long>(static_cast<unsigned int>(-3)) == 4294967293L); */
7     return 0;
8 }

```

事情是这样的：

- 编译器将表达式 $x < y$ (第 7 行) 转换为 `static_cast<unsigned int>(x) < y`。特别地，有符号 x 转换为无符号 int 类型。
- 因为这个转换， -3 变成了 $4'294'967'293$ 。
- $4'294'967'293$ 等于 2^{32} 加 -3
- 32 是 C++ Insights 中无符号整型的位宽。

由于 C++20 的出现，现在可以安全地对整数进行比较了。

5.4.1.2 安全的整数比较

C++20 支持 6 个整数比较函数：

六种安全比较函数

比较函数	描述
<code>std::cmp_equal</code>	<code>==</code>
<code>std::cmp_not_equal</code>	<code>!=</code>
<code>std::cmp_less</code>	<code><</code>
<code>std::cmp_less_equal</code>	<code><=</code>
<code>std::cmp_greater</code>	<code>></code>
<code>std::cmp_greater_equal</code>	<code>>=</code>

由于这六个比较函数的出现，现在可以轻松地将之前的程序 `unsafecomparis.cpp` 改写为 `safecomparis.cpp`。新的比较函数需要包含头文件 `<utility>`。

```
1 // safeComparison.cpp
2
3 #include <iostream>
4 #include <utility>
5
6 int main() {
7
8     std::cout << '\n';
9
10    std::cout << std::boolalpha;
11
12    int x = -3;
13    unsigned int y = 7;
14
15    std::cout << "-3 == 7: " << std::cmp_equal(x, y) << '\n';
16    std::cout << "-3 != 7: " << std::cmp_not_equal(x, y) << '\n';
17    std::cout << "-3 < 7: " << std::cmp_less(x, y) << '\n';
18    std::cout << "-3 <= 7: " << std::cmp_less_equal(x, y) << '\n';
19    std::cout << "-3 > 7: " << std::cmp_greater(x, y) << '\n';
20    std::cout << "-3 >= 7: " << std::cmp_greater_equal(x, y) << '\n';
```

```
21
22     std::cout << '\n';
23
24 }
```

另外，这里使用了等号和不等号运算符。

```
-3 == 7: false
-3 != 7: true
-3 < 7: true
-3 <= 7: true
-3 > 7: false
-3 => 7: false
```

使用非整数 (如 double) 调用安全比较函数会导致编译时错误。

尝试安全比较 uint 和 double

```
1 // safeComparison2.cpp
2
3 #include <iostream>
4 #include <utility>
5
6 int main() {
7
8     double x = -3.5;
9     unsigned int y = 7;
10
11    std::cout << "-3.5 < 7: " << std::cmp_less(x, y); // ERROR
12
13 }
```

另外，还是可以用经典的方法比较 double 型和 unsigned 型的数值。classicalcompare.cpp 就使用了 double 类型和 unsigned int 类型的经典比较方式。

```
1 // classicalComparison.cpp
2
3 int main() {
4
5     double x = -3.5;
6     unsigned int y = 7;
7
8     auto res = x < y; // true
9
10 }
```

没毛病！unsigned int 将升格为浮点，再提升为 double。[C++ Insights](#)展示了实际发生了什么：

```

int main()
{
    double x = -3.5;
    unsigned int y = 7;
    bool res = x < static_cast<double>(y);
}

```

浮点提升为 double

5.4.2 数学常数

首先，数学常数要求包含头文件 `<numbers>` 和命名空间 `std::numbers`，下表提供了一个概述。

Mathematical Constant	Description
<code>std::numbers::e</code>	e
<code>std::numbers::log2e</code>	$\log_2 e$
<code>std::numbers::log10e</code>	$\log_{10} e$
<code>std::numbers::pi</code>	π
<code>std::numbers::inv_pi</code>	$\frac{1}{\pi}$
<code>std::numbers::inv_sqrtpi</code>	$\frac{1}{\sqrt{\pi}}$
<code>std::numbers::ln2</code>	$\ln 2$
<code>std::numbers::ln10</code>	$\ln 10$
<code>std::numbers::sqrt2</code>	$\sqrt{2}$
<code>std::numbers::sqrt3</code>	$\sqrt{3}$
<code>std::numbers::inv_sqrt3</code>	$\frac{1}{\sqrt{3}}$
<code>std::numbers::egamma</code>	欧拉常数 (Euler-Mascheroni 常数)
<code>std::numbers::phi</code>	ϕ

`mathematicConstants.cpp` 使用了一些数学常数。

```

1 // mathematicConstants.cpp
2
3 #include <iomanip>
4 #include <iostream>
5 #include <numbers>
6
7 int main() {
8     std::cout << '\n';
9
10    std::cout << std::setprecision(10);
11
12    std::cout << "std::numbers::e: " << std::numbers::e << '\n';
13    std::cout << "std::numbers::log2e: " << std::numbers::log2e << '\n';
14    std::cout << "std::numbers::log10e: " << std::numbers::log10e << '\n';
15    std::cout << "std::numbers::pi: " << std::numbers::pi << '\n';

```

```
16 std::cout << "std::numbers::inv_pi: " << std::numbers::inv_pi << '\n';
17 std::cout << "std::numbers::inv_sqrt(pi): " << std::numbers::inv_sqrt(pi) << '\n';
18 std::cout << "std::numbers::ln2: " << std::numbers::ln2 << '\n';
19 std::cout << "std::numbers::sqrt2: " << std::numbers::sqrt2 << '\n';
20 std::cout << "std::numbers::sqrt3: " << std::numbers::sqrt3 << '\n';
21 std::cout << "std::numbers::inv_sqrt3: " << std::numbers::inv_sqrt3 << '\n';
22 std::cout << "std::numbers::egamma: " << std::numbers::egamma << '\n';
23 std::cout << "std::numbers::phi: " << std::numbers::phi << '\n';
24
25 std::cout << '\n';
26 }
```

下面是使用 MSVC 编译器生成程序的输出。

```
std::numbers::e: 2.718281828
std::numbers::log2e: 1.442695041
std::numbers::log10e: 0.4342944819
std::numbers::pi: 3.141592654
std::numbers::inv_pi: 0.3183098862
std::numbers::inv_sqrt(pi): 0.5641895835
std::numbers::ln2: 0.6931471806
std::numbers::sqrt2: 1.414213562
std::numbers::sqrt3: 1.732050808
std::numbers::inv_sqrt3: 0.5773502692
std::numbers::egamma: 0.5772156649
std::numbers::phi: 1.618033989
```

数学常数可用于 float、double 和 long double。默认情况下，使用 double，也可以指定为 float (std::numbers::pi_v<float>) 或 long double (std::numbers::pi_v<long double>)。

5.4.3 中间值与线性插值

- std::midpoint(a, b): 计算整数、浮点数或指针的中点 ($a + (b - a) / 2$)。若 a 和 b 是指针，则必须指向同一个数组对象。该函数需要包含头文件 <numeric>。
- std::lerp(a, b, t): 计算线性插值 ($a + t(b - a)$)。当 t 在范围 [0,1] 之外时，则计算线性外推。函数需要包含头文件 <cmath>。

midpointLerp.cpp 使用这两个函数。

计算中点和数字的线性插值

```
1 // midpointLerp.cpp
2
3 #include <cmath>
```

```
4 #include <numeric>
5 #include <iostream>
6
7 int main() {
8
9     std::cout << '\n';
10
11    std::cout << "std::midpoint(10, 20): " << std::midpoint(10, 20) << '\n';
12
13    std::cout << '\n';
14
15    for (auto v: {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}) {
16        std::cout << "std::lerp(10, 20, " << v << ") : " << std::lerp(10, 20, v)
17            << '\n';
18    }
19
20    std::cout << '\n';
21
22 }
```

输出就没什么好说的了：

```
std::midpoint(10, 20): 15

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

5.4.4 位操作

头文件 `<bit>` 声明了访问和操作单个位或位序列的函数。

5.4.4.1 std::endian

新类型 `std::endian`, 可以获得标量类型的字节序。端序可以是大端序或小端序。大端表示高位字节在最左边, 小端表示低位字节在最左边。标量类型可以是算术类型、枚举、指针、成员指针或

std::nullptr_t。

endian 类提供了所有标量类型的端序:

```
1 enum class endian
2 {
3     little = /*implementation-defined*/,
4     big = /*implementation-defined*/,
5     native = /*implementation-defined*/
6 };
```

- 若所有标量类型都是小端，则 std::endian::native 等于 std::endian::little。
- 若所有标量类型都是大端，则 std::endian::native 等于 std::endian::big。

也支持极端情况:

- 若所有的标量类型都具有 sizeof 1，端序就不重要了，枚举数 std::endian::little、std::endian::big 和 std::endian::native 的值相同。
- 若平台使用混合字节序，std::endian::native 既不等于 std::endian::big，也不等于 std::endian::little。

当在 x86 架构上执行下面的 getEndianness.cpp 时，我得到的结果是 little-endian。

```
1 // getEndianness.cpp
2
3 #include <bit>
4 #include <iostream>
5
6 int main() {
7
8     if constexpr (std::endian::native == std::endian::big) {
9         std::cout << "big-endian" << '\n';
10    }
11    else if constexpr (std::endian::native == std::endian::little) {
12        std::cout << "little-endian" << '\n'; // little-endian
13    }
14
15 }
```

constexpr if 使编译器能够有条件地编译源代码。所以，编译依赖于架构的字节序。

5.4.4.2 访问，位操作，位序列

下表概述了所有函数，可以在头文件 <bit> 中找到相应的函数。

函数	位操作描述
std::bit_cast	重新解释对象的表示
std::has_single_bit	检查数字是否为 2 的次幂
std::bit_ceil	求不小于给定值的 2 的最小整数次幂
std::bit_floor	求不大于给定值的 2 的最大整数次幂
std::bit_width	找到表示给定值的最小位数
std::rotl	计算按位向左移
std::rotr	计算按位向右移
std::countl_zero	从最高位开始，计数连续 0 的个数
std::countl_one	从最高位开始，计算连续 1 的个数
std::countr_zero	从最低位开始，计数连续 0 的个数
std::countr_one	从最低有效位，计算连续 1 的个数
std::popcount	统计无符号整数中位值为 1 的个数

除了 std::bit_cast 外，所有函数都要求是无符号整数类型 (unsigned char、unsigned short、unsigned int、unsigned long 或 unsigned long)。

bit.cpp 展示了如何使用这些函数。

```

1 // bit.cpp
2
3 #include <bit>
4 #include <bitset>
5 #include <iostream>
6
7 int main() {
8
9     std::uint8_t num= 0b00110010;
10
11    std::cout << std::boolalpha;
12
13    std::cout << "std::has_single_bit(0b00110010): " << std::has_single_bit(num)
14        << '\n';
15
16    std::cout << "std::bit_ceil(0b00110010): " << std::bitset<8>(std::bit_ceil(num))
17        << '\n';
18    std::cout << "std::bit_floor(0b00110010): "
19        << std::bitset<8>(std::bit_floor(num)) << '\n';
20
21    std::cout << "std::bit_width(5u): " << std::bit_width(5u) << '\n';
22
23    std::cout << "std::rotl(0b00110010, 2): " << std::bitset<8>(std::rotl(num, 2))
24        << '\n';
25
26    std::cout << "std::rotr(0b00110010, 2): " << std::bitset<8>(std::rotr(num, 2))
27        << '\n';
28

```

```
29     std::cout << "std::countl_zero(0b00110010): " << std::countl_zero(num) << '\n';
30     std::cout << "std::countl_one(0b00110010): " << std::countl_one(num) << '\n';
31     std::cout << "std::countr_zero(0b00110010): " << std::countr_zero(num) << '\n';
32     std::cout << "std::countr_one(0b00110010): " << std::countr_one(num) << '\n';
33     std::cout << "std::popcount(0b00110010): " << std::popcount(num) << '\n';
34 }
```

下面是程序的输出：

```
std::has_single_bit(0b00110010): false
std::bit_ceil(0b00110010): 01000000
std::bit_floor(0b00110010): 00100000
std::bit_width(5u): 3
std::rotl(0b00110010, 2): 11001000
std::rotr(0b00110010, 2): 10001100
std::countl_zero(0b00110010): 2
std::countl_one(0b00110010): 0
std::countr_zero(0b00110010): 1
std::countr_one(0b00110010): 0
std::popcount(0b00110010): 3
```

下面的程序展示了，将 std::bit_floor, std::bit_ceil, std::bit_width 和 std::bit_popcount 用于数字 2 到 7 的效果。

```
1 // bitFloorCeil.cpp
2
3 #include <bit>
4 #include <bitset>
5 #include <iostream>
6
7 int main() {
8
9     std::cout << '\n';
10
11    std::cout << std::boolalpha;
12
13    for (auto i = 2u; i < 8u; ++i) {
14        std::cout << "bit_floor(" << std::bitset<8>(i) << ") = "
15                    << std::bit_floor(i) << '\n';
16
17        std::cout << "bit_ceil(" << std::bitset<8>(i) << ") = "
18                    << std::bit_ceil(i) << '\n';
19
20        std::cout << "bit_width(" << std::bitset<8>(i) << ") = "
21                    << std::bit_width(i) << '\n';
```

```
22
23     std::cout << "popcount(" << std::bitset<8>(i) << ")" = "
24         << std::popcount(i) << '\n';
25
26     std::cout << '\n';
27 }
28
29 std::cout << '\n';
30 }
```

```
bit_floor(00000010) = 2
bit_ceil(00000010) = 2
bit_width(00000010) = 2
popcount(00000010) = 1
```

```
bit_floor(00000011) = 2
bit_ceil(00000011) = 4
bit_width(00000011) = 2
popcount(00000011) = 2
```

```
bit_floor(00000100) = 4
bit_ceil(00000100) = 4
bit_width(00000100) = 3
popcount(00000100) = 1
```

```
bit_floor(00000101) = 4
bit_ceil(00000101) = 8
bit_width(00000101) = 3
popcount(00000101) = 2
```

```
bit_floor(00000110) = 4
bit_ceil(00000110) = 8
bit_width(00000110) = 3
popcount(00000110) = 2
```

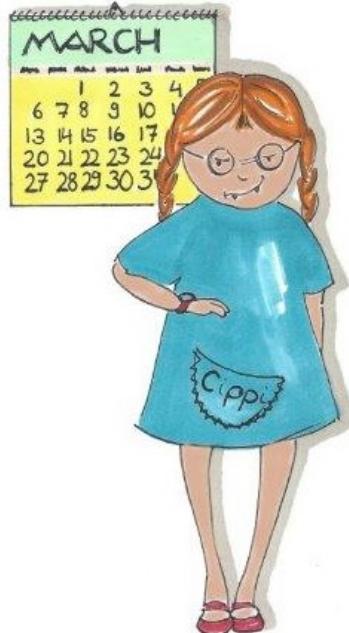
```
bit_floor(00000111) = 4
bit_ceil(00000111) = 8
bit_width(00000111) = 3
popcount(00000111) = 3
```

使用 `std::bit_floor`, `std::bit_ceil`, `std::bit_width` 和 `std::popcount`

总结

- C++20 中的 `cmp_*` 函数支持安全的整数比较，其可以检测有符号整数和无符号整数的比较。在不安全比较的情况下，编译会失败。
- 并且，定义了许多数学常数，如 e 、 $\log_2 e$ 或 π 。
- C++20 提供了计算两个值的中点或线性插值的工具函数。
- 新函数可以访问和操作位或位序列。

5.5. 日期和时区



Cippi 在研究日历

缺少编译器支持

截止到 2020 年底，还没有 C++ 编译器支持 chrono 扩展。感谢 HowardHinnant 的原型库[date](#)，它本质上是 C++20 中时间功能扩展的超集，我用它做了一些实验。该库托管在 GitHub 上，使用 date 原型的方法有很多：

- 可以在 [Wandbox](#) 上试试。Howard 已经上传了 `date.h` 头文件，这足以使用新的类型 `std::time_of_day` 和日历。下面是 Howard 给出的链接：[在 Wandbox 上试一试吧！](#)。
- 下载项目并构建它，GitHub 上[date](#)提供了更多信息。想要尝试新的时区特性时，需要进行此步骤。

本章的例子使用 Howard Hinnant 的库。不过，我的解释是基于 C++20 术语的。当 C++ 编译器支持扩展的 chrono 功能时，我将使示例适配 C++20 的语法。

- 一天中的时间是指从午夜开始的时间长度，分为小时、分钟、秒和分秒。
- `Calendar` 表示各种日历日期，如年、月、工作日或一周的第 n 天。
- 时区表示特定于地理区域的时间。

时区功能 (C++20) 是基于日历功能 (C++20)，而日历功能 (C++20) 是基于时间功能 (C++11)。

C++11 的时间库

为了充分利用本节的内容，必须先对 chrono 库有基本的了解。C++11 引入了三个主要组件来处理时间：

- 时间点由时间起点 (即所谓的 epoch) 和时间段定义。
- 时间间隔由两个时间点之间的差值，使用时钟周期数表示。

- 时钟由时间起点 (epoch) 和时钟周期数组成，可以计算出当前的时间点。

老实说，时间对我来说是个谜。一方面，每个人对时间都有一个直观的概念；另一方面，正式定义它极具挑战性。例如，时间点 (time point)、时间段 (time duration) 和时钟 (clock) 三者相互依赖。若想了解更多 C++11 中有关时间的功能，请阅读我在[time](#)上发布的关于时间的文章。

5.5.1 日期

`std::chrono::hh_mm_ss` 是自午夜开始的时间段，分为小时、分钟、秒和分秒，这种类型通常用作格式化工具。下表提供了 `std::chrono::hh_mm_ss` 实例 `tOfDay` 的简明概述。

日期

函数	描述
<code>tOfDay.hours()</code>	返回从午夜开始累计的小时数
<code>tOfDay.minutes()</code>	返回从午夜开始累计的分钟数
<code>tOfDay.seconds()</code>	返回从午夜开始累计的秒数
<code>tOfDay.subseconds()</code>	返回从午夜开始累计的分秒数
<code>tOfDay.to_duration()</code>	返回从午夜以来的时间段
<code>std::chrono::make12(hour)</code>	返回 12 小时的等价 24 小时的时间格式
<code>std::chrono::make24(hour)</code>	返回 24 小时的等价 12 小时的时间格式
<code>std::chrono::is_am(hour)</code>	检测 24 小时时间格式是否为 a.m.
<code>std::chrono::is_pm(hour)</code>	检测 24 小时时间格式是否为 p.m.

这些函数的使用起来很简单。

```

1 // timeOfDay.cpp
2
3 #include <chrono>
4 #include <iostream>
5
6 int main() {
7     using namespace std::chrono;
8
9     using namespace std::chrono_literals;
10
11    std::cout << std::boolalpha << '\n';
12    auto timeOfDay = std::chrono::hh_mm_ss(10.5h + 98min + 2020s + 0.5s);
13
14    std::cout << "timeOfDay: " << timeOfDay << '\n';
15
16    std::cout << '\n';
17
18    std::cout << "timeOfDay.hours(): " << timeOfDay.hours() << '\n';

```

```

19 std::cout << "timeOfDay.minutes(): " << timeOfDay.minutes() << '\n';
20 std::cout << "timeOfDay.seconds(): " << timeOfDay.seconds() << '\n';
21 std::cout << "timeOfDay.subseconds(): " << timeOfDay.subseconds() << '\n';
22 std::cout << "timeOfDay.to_duration(): " << timeOfDay.to_duration() << '\n';
23 std::cout << '\n';

24
25 std::cout << "date::hh_mm_ss(45700.5s): " << std::chrono::hh_mm_ss(45700.5s) << '\n';
26
27 std::cout << '\n';

28
29 std::cout << "date::is_am(5h): " << std::chrono::is_am(5h) << '\n';
30 std::cout << "date::is_am(15h): " << std::chrono::is_am(15h) << '\n';
31
32 std::cout << '\n';

33
34 std::cout << "date::make12(5h): " << std::chrono::make12(5h) << '\n';
35 std::cout << "date::make12(15h): " << std::chrono::make12(15h) << '\n';
36
37 }

```

首先，我在第 12 行创建了 `std::chrono::hh_mm_ss`: `timeOfDay` 的一个新实例。C++14 添加了 `chrono` 字面值，就可以添加一些时间时间段来初始化一天中的时间对象。使用 C++20，可以直接输出 `timeOfDay`(第 14 行)，这就是在第 7 行中引入命名空间 `date` 的原因，其余部分应该很容易阅读。第 18-21 行，以小时、分钟、秒和分数秒为单位显示了从午夜开始的时间分量。第 22 行返回自午夜以来的时间时间段，以秒为单位。第 26 行更有趣：给定的秒对应于第 15 行中显示的时间。若给定的时间是 a.m.，则返回第 30 和 32 行。第 35 行和 36 行返回与给定小时等价 12 小时的时间格式。

下面是程序的输出：

```

timeOfDay: 12:41:40.500000

timeOfDay.hours(): 12h
timeOfDay.minutes(): 41min
timeOfDay.seconds(): 40s
timeOfDay.subseconds(): 0.500000s
timeOfDay.to_duration(): 45700.500000s

date::hh_mm_ss(45700.5s): 12:41:40.500000

date::is_am(5h): true
date::is_am(15h): false

date::make12(5h): 5h
date::make12(15h): 3h

```

5.5.2 日历日期

C++20 中一个新的 chrono 扩展类型是日历日期。C++20 支持多种方式来创建日历日期，并可与之交互。那什么是日历日期呢？

日历日期是由一年、一个月和一天组成的日期，所以 C++20 有一个特定的数据类型 std::chrono::year_month_day。C++20 提供了很多相关功能。在下面的表中，将了解日历日期类型，然后再展示各种方式的用例。

各种日历日期类型

类型	描述
std::chrono::last_spec	一个月的最后一天或工作日
std::chrono::day	一个月中的一天
std::chrono::month	一年中的一个月
std::chrono::year	表示公历中的一年
std::chrono::weekday	表示公历中的一周中的一天
std::chrono::weekday_indexed	表示每月的第 n 个工作日
std::chrono::weekday_last	表示一个月的最后一个工作日
std::chrono::month_day	表示特定月份中的特定日期
std::chrono::month_day_last	表示特定月份的最后一天
std::chrono::month_weekday	表示特定月份的第 n 个工作日
std::chrono::month_weekday_last	表示特定月份的最后一个工作日
std::chrono::year_month	表示特定年份中的特定月份
std::chrono::year_month_day	表示特定的年、月和日
std::chrono::year_month_day_last	表示特定年份和月份的最后一天
std::chrono::year_month_weekday	表示特定年份和月份的第 n 个工作日
std::chrono::year_month_day_weekday_last	表示特定年份和月份的最后一个工作日
std::chrono::operator /	创建公历日期

先创建几个日历日期。

5.5.2.1 创建日历日期

createCalendar.cpp 展示了创建日历相关日期的各种方法。

```
1 // createCalendar.cpp
2
3 #include <iostream>
4 #include "date.h"
5
6 int main() {
7
8     std::cout << '\n';
9 }
```

```
10 using namespace date;
11
12 constexpr auto yearMonthDay{year(1940)/month(6)/day(26)};
13 std::cout << yearMonthDay << " ";
14 std::cout << date::year_month_day(1940_y, June, 26_d) << '\n';
15
16 std::cout << '\n';
17
18 constexpr auto yearMonthDayLast{year(2010)/March/last};
19 std::cout << yearMonthDayLast << " ";
20 std::cout << date::year_month_day_last(2010_y, month_day_last(month(3))) << '\n';
21
22 constexpr auto yearMonthWeekday{year(2020)/March/Thursday[2]};
23 std::cout << yearMonthWeekday << " ";
24 std::cout << date::year_month_weekday(2020_y, month(March), Thursday[2]) << '\n';
25
26 constexpr auto yearMonthWeekdayLast{year(2010)/March/Monday[last]};
27 std::cout << yearMonthWeekdayLast << " ";
28 std::cout << date::year_month_weekday_last(2010_y, month(March),
29 weekday_last(Monday)) << '\n';
30
31 std::cout << '\n';
32
33 constexpr auto day_{day(19)};
34 std::cout << day_ << " ";
35 std::cout << date::day(19) << '\n';
36
37 constexpr auto month_{month(1)};
38 std::cout << month_ << " ";
39 std::cout << date::month(1) << '\n';
40
41 constexpr auto year_{year(1988)};
42 std::cout << year_ << " ";
43 std::cout << date::year(1988) << '\n';
44
45 constexpr auto weekday_{weekday(5)};
46 std::cout << weekday_ << " ";
47 std::cout << date::weekday(5) << '\n';
48
49 constexpr auto yearMonth{year(1988)/1};
50 std::cout << yearMonth << " ";
51 std::cout << date::year_month(year(1988), January) << '\n';
52
53 constexpr auto monthDay{10/day(22)};
54 std::cout << monthDay << " ";
55 std::cout << date::month_day(October, day(22)) << '\n';
56
57 constexpr auto monthDayLast{June/last};
58 std::cout << monthDayLast << " ";
```

```

59 std::cout << date::month_day_last(month(6)) << '\n';
60
61 constexpr auto monthWeekday{2/Monday[3]};
62 std::cout << monthWeekday << " ";
63 std::cout << date::month_weekday(February, Monday[3]) << '\n';
64
65 constexpr auto monthWeekDayLast{June/Sunday[last]};
66 std::cout << monthWeekDayLast << " ";
67 std::cout << date::month_weekday_last(June, weekday_last(Sunday)) << '\n';
68
69 std::cout << '\n';
70
71 }

```

有两种方法可以创建日历日期。可以使用 `yearMonthDay{year(1940)/month(6)/day(26)}`(第 12 行), 或直接使用显式类型 `date::year_month_day(1940y, June, 26d)`(第 14 行)。使用显式类型非常有趣, 因为其使用了日期-时间的字面值 1940y、26d 和常量 June。这里, 我把对第一种好用的语法的解释推迟到下一节。

第 18 行、第 22 行和第 26 行提供了更多创建日历日期的方法。

1. 第 18 行:2010 年 3 月的最后一天

```
1 {year(2010)/March/last}
```

或

```
1 year_month_day_last(2010y, month_day_last(month(3)))
```

2. 第 22 行:2020 年 3 月第二个星期四

```
1 {year(2020)/March/Thursday[2]}
```

或

```
1 year_month_weekday(2020y, month(March), Thursday[2])
```

3. 第 26 行:2010 年 3 月的第一个星期一

```
1 {year(2010)/March/Monday[last]}
```

或

```
1 year_month_weekday_last(2010y, month(March), weekday_last(Monday))
```

其余日历类型代表一天(第 33 行)、一个月(第 37 行)或一年(第 41 行)。可以将它们组合使用, 并将其用作完全指定的日历日期的基本构建块, 例如第 18、22 或 26 行。

这是程序的输出:

```
1940-06-26 1940-06-26
```

```
2010/Mar/last 2021/Mar/last
2020/Mar/Thu[2] 2020/Mar/Thu[2]
2010/Mar/mon[last] 2010/Mar/mon[last]

19 19
Jan Jan
1988 1988
Fri Fri
1988/Jan 1988/Jan
Oct/22 Oct/22
Jun/last Jun/last
Feb/Mon[3] Feb/Mon[3]
Jun/Sun[last] Jun/Sun[last]
```

5.5.2.2 “好用的”语法解释

这种语法由重载除法运算符组成，用于指定日历日期。重载操作符支持时间字面量(例如:2020y, 31d)和常量(January, February, March, April, May, June, July, August, September, October, November, December)。

使用这种语法时，可以使用以下年、月和日的三种组合。

```
1 year/month/day
2 day/month/year
3 month/day/year
```

这些组合并不是随意选择的，是可以在世界范围内使用的。但不能使用其他的组合方式。

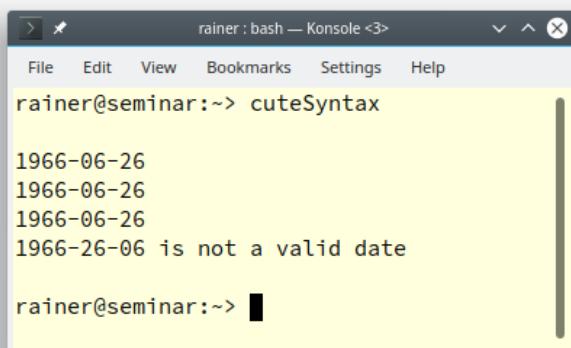
当为第一个参数选择类型年、月或日时，其余两个参数的类型就不再需要了，用数字就可以了。

```
1 // cuteSyntax.cpp
2
3 #include <iostream>
4 #include "date.h"
5
6 int main() {
7
8     std::cout << '\n';
9
10    using namespace date;
11
12    constexpr auto yearMonthDay{year(1966)/6/26};
13    std::cout << yearMonthDay << '\n';
14
15    constexpr auto dayMonthYear{day(26)/1966};
16    std::cout << dayMonthYear << '\n';
```

```
17
18 constexpr auto monthDayYear{month(6)/26/1966};
19 std::cout << monthDayYear << '\n';
20
21 constexpr auto yearDayMonth{year(1966)/month(26)/6};
22 std::cout << yearDayMonth << '\n';
23
24 std::cout << '\n';
25
26 }
```

年/日/月的组合 (第 21 行) 是非法的，这会导致一条运行时的错误。

```
1966-06-26
1966-06-26
1966-06-26
1966-26-06 is not a valid date
```



这里，假设希望以可读的形式显示日历日期 {year(2010)/March/last}，例如 2020-03-31。这是 local_days 或 sys_days 操作符的任务。

5.5.2.3 显示日历日期

使用 std::chrono::local_days 或 std::chrono::sys_days，可以将日历日期转换为 std::chrono::time_point。例子中，我使用的是 std::chrono::sys_days，其基于 std::chrono::system_clock。这里，我将对前一个程序 createCalendar.cpp 中的日历日期进行转换 (第 18、22 和 26 行)。

```
1 // sysDays.cpp
2
3 #include <iostream>
4 #include "date.h"
5
6 int main() {
```

```

7     std::cout << '\n';
8
9
10    using namespace date;
11
12    constexpr auto yearMonthDayLast{year(2010)/March/last};
13    std::cout << "sys_days(yearMonthDayLast): "
14    << sys_days(yearMonthDayLast) << '\n';
15
16    constexpr auto yearMonthWeekday{year(2020)/March/Thursday[2]};
17    std::cout << "sys_days(yearMonthWeekday): "
18    << sys_days(yearMonthWeekday) << '\n';
19
20    constexpr auto yearMonthWeekdayLast{year(2010)/March/Monday[last]};
21    std::cout << "sys_days(yearMonthWeekdayLast): "
22    << sys_days(yearMonthWeekdayLast) << '\n';
23
24    std::cout << '\n';
25
26    constexpr auto leapDate{year(2012)/February/last};
27    std::cout << "sys_days(leapDate): " << sys_days(leapDate) << '\n';
28
29    constexpr auto noLeapDate{year(2013)/February/last};
30    std::cout << "sys_day(noLeapDate): " << sys_days(noLeapDate) << '\n';
31
32    std::cout << '\n';
33
34}

```

std::chrono::last 常量 (第 11 行) 让可以轻松确定一个月有多少天。输出显示，2012 是闰年 (第 26 行)，而 2013 年不是 (第 29 行)。

```

sys_days(yearMonthDayLast): 2010-03-31
sys_days(yearMonthWeekday): 2020-03-12
sys_days(yearMonthWeekdayLast): 2010-03-29

sys_days(leapDate): 2012-02-29
sys_days(noLeapDate): 2013-02-28

```

假设有一个日历日期，例如：year(2100)/2/29。第一个问题是：这个日期有效吗？

5.5.2.4 检查日期是否有效

C++20 中的各种日历类型都有一个 `ok` 函数。若日期有效，该函数返回 `true`。

```

1 // leapYear.cpp
2

```

```

3 #include <iostream>
4 #include <chrono>
5
6 int main() {
7
8     std::cout << std::boolalpha << '\n';
9
10    using namespace std::chrono;
11
12    std::cout << "Valid days" << '\n';
13    day day31(31);
14    day day32 = day31 + days(1);
15    std::cout << " day31: " << static_cast<unsigned>(day31) << "; ";
16    std::cout << "day31.ok(): " << day31.ok() << '\n';
17    std::cout << " day32: " << static_cast<unsigned>(day32) << "; ";
18    std::cout << "day32.ok(): " << day32.ok() << '\n';
19
20
21    std::cout << '\n';
22
23    std::cout << "Valid months" << '\n';
24    month month1(1);
25    month month0(0);
26    std::cout << " month1: " << static_cast<unsigned>(month1) << "; ";
27    std::cout << "month1.ok(): " << month1.ok() << '\n';
28    std::cout << " month0: " << static_cast<unsigned>(month0) << "; ";
29    std::cout << "month0.ok(): " << month0.ok() << '\n';
30
31    std::cout << '\n';
32
33    std::cout << "Valid years" << '\n';
34    year year2020(2020);
35    year year32768(-32768);
36    std::cout << " year2020: " << static_cast<int>(year2020) << "; ";
37    std::cout << "year2020.ok(): " << year2020.ok() << '\n';
38    std::cout << " year32768: " << static_cast<int>(year32768) << "; ";
39    std::cout << "year32768.ok(): " << year32768.ok() << '\n';
40
41    std::cout << '\n';
42
43    std::cout << "Leap Years" << '\n';
44
45    constexpr auto leapYear2016{year(2016)/2/29};
46    constexpr auto leapYear2020{year(2020)/2/29};
47    constexpr auto leapYear2024{year(2024)/2/29};
48
49    std::cout << " leapYear2016.ok(): " << leapYear2016.ok() << '\n';
50    std::cout << " leapYear2020.ok(): " << leapYear2020.ok() << '\n';
51    std::cout << " leapYear2024.ok(): " << leapYear2024.ok() << '\n';

```

```

52     std::cout << '\n';
53
54     std::cout << "No Leap Years" << '\n';
55
56
57     constexpr auto leapYear2100{year(2100)/2/29};
58     constexpr auto leapYear2200{year(2200)/2/29};
59     constexpr auto leapYear2300{year(2300)/2/29};
60
61     std::cout << " leapYear2100.ok(): " << leapYear2100.ok() << '\n';
62     std::cout << " leapYear2200.ok(): " << leapYear2200.ok() << '\n';
63     std::cout << " leapYear2300.ok(): " << leapYear2300.ok() << '\n';
64
65     std::cout << '\n';
66
67     std::cout << "Leap Years" << '\n';
68
69
70     constexpr auto leapYear2000{year(2000)/2/29};
71     constexpr auto leapYear2400{year(2400)/2/29};
72     constexpr auto leapYear2800{year(2800)/2/29};
73
74     std::cout << " leapYear2000.ok(): " << leapYear2000.ok() << '\n';
75     std::cout << " leapYear2400.ok(): " << leapYear2400.ok() << '\n';
76     std::cout << " leapYear2800.ok(): " << leapYear2800.ok() << '\n';
77
78     std::cout << '\n';
79 }

```

程序中检查给定的日期 (第 12 行)、给定的月份 (第 23 行) 或给定的年份 (第 33 行) 是否有效。天的范围是 [1,31]，月的范围是 [1,12]，年的范围是 [-32767,32767]，所以 ok() 将返回 false。当我显示不同的值时，若值无效，则输出显示：“不是有效的日期”、“不是有效的月份”、“不是有效的年份” [译者注：标准库不会输出这些信息]。其次，月份值会以字符串形式显示。

Valid days

```

day31: 31; day31.ok(): true
day32: 32; day32.ok(): false

```

Valid months

```

month1: 1; month1.ok(): true
month0: 0; month0.ok(): false

```

Valid years

```

year2020: 2020; year2020.ok(): true

```

```
year32768: -32768; year32768.ok(): false
```

Leap Years

```
leapYear2016.ok(): true
leapYear2020.ok(): true
leapYear2024.ok(): true
```

No Leap Years

```
leapYear2100.ok(): false
leapYear2200.ok(): false
leapYear2300.ok(): false
```

Leap Years

```
leapYear2000.ok(): true
leapYear2400.ok(): true
leapYear2800.ok(): true
```

可以在日历日期上使用 `ok`, 所以检查一个特定的日历日期是否为闰日就很容易了, 从而就可以判断相应的年份是否为闰年。全球使用的[格里高利历](#)中, 适用以下规则:

非整世纪年能被 4 整除的年份是闰年。

整世纪年能被 100 整除的年份是不是闰年, 能被 400 整除的年份才是闰年。

有点复杂哈? `leapYears.cpp` 用代码表达了这个规则。

扩展的 `chrono` 库可以很容易地计算日历日期之间的时间时间段。

5.5.2.5 查询日历日期

下面的 `queryCalendarDates.cpp` 就是在查询一些日历日期。

```
1 // queryCalendarDates.cpp
2
3 #include "date.h"
4 #include <iostream>
5
6 int main() {
7
8     using namespace date;
9
10    std::cout << '\n';
11
12    auto now = std::chrono::system_clock::now();
13    std::cout << "The current time is: " << now << " UTC\n";
14    std::cout << "The current date is: " << floor<days>(now) << '\n';
15    std::cout << "The current date is: " << year_month_day{floor<days>(now)}
```

```

16    << '\n';
17    std::cout << "The current date is: " << year_month_weekday{floor<days>(now)}
18    << '\n';
19
20    std::cout << '\n';
21
22
23    auto currentDate = year_month_day(floor<days>(now));
24    auto currentYear = currentDate.year();
25    std::cout << "The current year is " << currentYear << '\n';
26    auto currentMonth = currentDate.month();
27    std::cout << "The current month is " << currentMonth << '\n';
28    auto currentDay = currentDate.day();
29    std::cout << "The current day is " << currentDay << '\n';
30
31    std::cout << '\n';
32
33    auto hAfter = floor<std::chrono::hours>(now) - sys_days(January/1/currentYear);
34    std::cout << "It has been " << hAfter << " since New Year!\n";
35
36    auto nextYear = currentDate.year() + years(1);
37    auto nextNewYear = sys_days(January/1/nextYear);
38    auto hBefore = sys_days(January/1/nextYear) - floor<std::chrono::hours>(now);
39    std::cout << "It is " << hBefore << " before New Year!\n";
40
41    std::cout << '\n';
42
43    std::cout << "It has been " << floor<days>(hAfter) << " since New Year!\n";
44    std::cout << "It is " << floor<days>(hBefore) << " before New Year!\n";
45
46    std::cout << '\n';
47
48 }

```

使用 C++20 扩展，可以直接显示时间点，例如：now(第 12 行)。atd::chrono::floor 将时间点转换为 atd::chrono::sys_days，这个值可以用来初始化日历类型 std::chrono::year_month_day。将值放入 std::chrono::year_month_weekday 日历类型中时，得到的答案是这个特定的日子是 10 月的第 3 个星期二。当然，还可以基于月份计算日历日期，例如当前的年、月或日(第 23 行)。

第 33 行是最有趣的一行。当使用基于小时的日期，从当前日期减去当前年份的 1 月 1 日时，得到了自新的一年来的小时数。相反，当从明年的 1 月 1 日减去当前日期(第 37 行)时，使用小时解析后，得到了到新的一年的小数小时数(也许你不喜欢基于小时的方式，第 42 和 43 行使用基于天的方式显示值)。

```
The current time is: 2020-10-20 06:08:01.516990636 UTC
```

```
The current date is: 2020-10-20
```

```
The current date is: 2020-10-20  
The current data is: 2020/Oct/Tue[3]
```

```
The current year is 2020  
The current month is Oct  
The current day is 20
```

```
It has been 7038h sine New Year!  
It is 1746h before New Year!
```

```
It has been 293d since New Year!  
It is 72d before New Year!
```

现在，我想知道我的生日是星期几。

5.5.2.6 查询星期几

由于扩展的 chrono 库，可以很容易地获得给定日期是星期几。

```
1 // weekdaysOfBirthdays.cpp  
2  
3 #include <cstdlib>  
4 #include <iostream>  
5 #include "date.h"  
6  
7 int main() {  
8  
9     std::cout << '\n';  
10  
11    using namespace date;  
12  
13    int y;  
14    int m;  
15    int d;  
16  
17    std::cout << "Year: ";  
18    std::cin >> y;  
19    std::cout << "Month: ";  
20    std::cin >> m;  
21    std::cout << "Day: ";  
22    std::cin >> d;  
23  
24    std::cout << '\n';  
25  
26    auto birthday = year(y)/month(m)/day(d);  
27
```

```

28 if (not birthday.ok()) {
29     std::cout << birthday << '\n';
30     std::exit(EXIT_FAILURE);
31 }
32
33 std::cout << "Birthday: " << birthday << '\n';
34 auto birthdayWeekday = year_month_weekday(birthday);
35 std::cout << "Weekday of birthday: " << birthdayWeekday.weekday() << '\n';
36
37 auto currentDate = year_month_day(floor<days>(
38     std::chrono::system_clock::now()));
39 auto currentYear = currentDate.year();
40
41 auto age = (int)currentDate.year() - (int)birthday.year();
42 std::cout << "Your age: " << age << '\n';
43
44 std::cout << '\n';
45
46 std::cout << "Weekdays for your next 10 birthdays" << '\n';
47
48 for (int i = 1, newYear = (int)currentYear; i <= 10; ++i) {
49     std::cout << " Age " << +age << '\n';
50     auto newBirthday = year(++newYear)/month(m)/day(d);
51     std::cout << " Birthday: " << newBirthday << '\n';
52     std::cout << " Weekday of birthday: "
53         << year_month_weekday(newBirthday).weekday() << '\n';
54 }
55
56 std::cout << '\n';
57
58 }

```

首先，程序询问我的出生的年、月和日(第 17 行)信息。根据输入，创建一个日历日期(第 26 行)并检查它是否有效(第 28 行)。现在，显示了生日的星期名称。我使用日历日期来填充 std::chrono::year_month_weekday(第 34 行)。为了获得日历类型 year 的 int 表示形式，必须将其转换为 int(第 41 行)。现在，可以显示年龄了。最后，for 循环为接下来的十个生日(第 46 行)显示以下信息：年龄、日历日期和工作日。这里，只需要增加 age 和 newYear 就好。

以我生日作为输入时，程序运行的后的输出。

```

Year: 1966
Month: 6
Day: 26

Birthday: 1966-06-26
Weekday of birthday: Sun

```

Your age: 57

Weekdays **for** your next 10 birthdays

Age 58

Birthday: 2024-06-26

Weekday of birthday: Wed

Age 59

Birthday: 2025-06-26

Weekday of birthday: Thu

Age 60

Birthday: 2026-06-26

Weekday of birthday: Fri

Age 61

Birthday: 2027-06-26

Weekday of birthday: Sat

Age 62

Birthday: 2028-06-26

Weekday of birthday: Mon

Age 63

Birthday: 2029-06-26

Weekday of birthday: Tue

Age 64

Birthday: 2030-06-26

Weekday of birthday: Wed

Age 65

Birthday: 2031-06-26

Weekday of birthday: Thu

Age 66

Birthday: 2032-06-26

Weekday of birthday: Sat

Age 67

Birthday: 2033-06-26

Weekday of birthday: Sun

我生日那天的星期名称

5.5.2.7 计算日期序数

这是最后一个日历功能的例子，我想介绍 HowardHinnant 提供的[示例](#)，其中有大约 40 个计时

功能的示例。据推测，C++20 中的 chrono 扩展并不容易获得，这么些示例就显得尤为宝贵了。读者们应该尝试这些例子，并将其作为进一步实验的起点，从而加强对这块内容的理解。

这里，介绍一个由[Roland Bock](#)编写的计算日期顺序的程序。

“日期序数由一年和该年中的某一天组成（1月1日是第1天，12月31日是第365天或第366天）。年份可以直接从 `year_month_day` 获取。计算起来非常简单。在下面的代码中，我们知道 `year_month_day` 可以处理像 1月0日这样的无效日期：”我在 Roland(Roland Bock) 的程序中添加了必要的头文件。

```
1 // ordinalDate.cpp
2
3 #include <chrono>
4 #include <iomanip>
5 #include <iostream>
6 #include <cassert>
7
8 int main()
9 {
10     using namespace std::chrono;
11
12     const auto time = std::chrono::system_clock::now();
13     const auto daypoint = floor<days>(time);
14     const auto ymd = year_month_day{daypoint};
15
16     // calculating the year and the day of the year
17     const auto year = ymd.year();
18     const auto year_day = daypoint - sys_days{year/January/0};
19
20     std::cout << (int)year << '-' << std::setfill('0') << std::setw(3)
21     << year_day.count() << '\n';
22
23     // inverse calculation and check
24     assert(ymd == year_month_day{sys_days{year/January/0} + year_day});
25 }
```

第13行截断当前时间点，该值在下一行中用于初始化日历日期。第18行计算两个时间点之间的时间时间段。两个时间点都以天为单位。最后，第21行中的 `year_day.count()` 返回以天为单位的时间时间段。[译者注：输出表示该日为2020年的第298天]

2020-298

5.5.3 时区

时区是一个地区，具有日期全部的历史，如夏令时或闰秒。C++20 中的时区库是[IANA 时区数据库](#)的完整解析器。下表将带您初步了解添加的新功能。

类型	时区数据类型描述
std::chrono::tzdb	描述 IANA 时区数据库的副本
std::chrono::tdz_list	表示 tzdb 链表
std::chrono::get_tzdb	
std::chrono::get_tzdb_list	访问和控制全局时区数据库
std::chrono::reload_tzdb	
std::chrono::remote_version	
std::chrono::locate_zone	根据名称定位时区
std::chrono::current_zone	返回当前时区
std::chrono::time_zone	表示时区
std::chrono::sys_info	表示特定时间点上的时区信息
std::chrono::local_info	表示本地时间到 UNIX 时间转换的信息
std::chrono::zoned_traits	用于表示时区的类指针
std::chrono::zoned_time	表示时区和时间点
std::chrono::leap_second	包含有关闰秒的信息
std::chrono::time_zone_link	表示时区的别名
std::chrono::nonexistent_local_time	若本地时间不存在将引发该异常

例子中，使用了 std::chrono::zones_time 函数，其本质上是一个时区和一个时间点的组合。

编译示例

要使用时区库编译程序，必须从[date](#)库编译 tz.cpp 文件，并将其链接到[curl](#)库。curl 库是获取当前[IANA 时区数据库](#)所必需的。下面的 g++ 编译命令应该能让你明白所有的事情：

使用原型库日期进行编译

```
g++ localTime.cpp -I <Path to data/tz.h> tz.cpp -std=c++17 \
-lcurl -o localTime
```

第一个例子很简单，显示 UTC 时间和本地时间。

5.5.3.1 UTC 时间和本地时间

[UTC 时间或协调世界时](#)是世界范围内的主要时间标准。计算机使用[Unix time](#)，非常接近 UTC。UNIX 时间是从 UNIX 纪元开始的秒数。Unix 纪元是 UTC 时间 1970 年 1 月 1 日 00:00:00。

std::chrono::system_clock::now() 在程序中返回 localTime.cpp 的 Unix 时间。

```
1 // localTime.cpp
2
3 #include "date/tz.h"
4 #include <iostream>
```

```

5
6 int main() {
7
8     std::cout << '\n';
9
10    using namespace date;
11
12    std::cout << "UTC time" << '\n';
13    auto utcTime = std::chrono::system_clock::now();
14    std::cout << " " << utcTime << '\n';
15    std::cout << " " << date::floor<std::chrono::seconds>(utcTime) << '\n';
16
17    std::cout << '\n';
18
19    std::cout << "Local time" << '\n';
20    auto localTime = date::make_zoned(date::current_zone(), utcTime);
21    std::cout << " " << localTime << '\n';
22    std::cout << " " << date::floor<std::chrono::seconds>(localTime.get_local_time())
23                                << '\n';
24
25    auto offset = localTime.get_info().offset;
26    std::cout << " UTC offset: " << offset << '\n';
27
28    std::cout << '\n';
29
30}

```

从第 12 行开始的代码块获取当前时间点，将其截断为秒并显示它。调用 `make_zoned`(第 20 行) 创建 `std::chrono::zoned_time` `localTime`，使用 `localTime.get_local_time()` 将存储的时间点作为本地时间返回，这个时间点也截断为秒。`localTime`(第 25 行) 还可以用于获取关于时区的信息。本例中，我感兴趣的是 UTC 时间的偏移。

```

UTC time
2020-10-23 21:23:26.128743011
2020-10-23 21:23:26

Local time
2020-10-23 21:23:26.128743011 CEST
2020-10-23 21:23:26
UTC offset: 7200s

```

下面的例子回答了我在教授不同时区时的一个关键问题：应该什么时候开始线上课程？

5.5.3.2 线上课程的不同时区

onlineClass.cpp 程序回答以下问题: 当我在当地时间(德国)7 小时、13 小时或 17 小时开始线上课程时, 在相应的时区是几点?

线上课程将于 2021 年 2 月 1 日开始, 时长为 4 小时。由于夏令时, 日历日期对于得到正确答案至关重要。

```
1 // onlineClass.cpp
2
3 #include "date/tz.h"
4 #include <algorithm>
5 #include <iomanip>
6 #include <iostream>
7
8 template <typename ZonedTime>
9 auto getMinutes(const ZonedTime& zonedTime) {
10     return date::floor<std::chrono::minutes>(zonedTime.get_local_time());
11 }
12
13 void printStartEndTimes(const date::local_days& localDay,
14                         const std::chrono::hours& h,
15                         const std::chrono::hours& durationClass,
16                         const std::initializer_list<std::string>& timeZones ) {
17
18     date::zoned_time startDate{date::current_zone(), localDay + h};
19     date::zoned_time endDate{date::current_zone(), localDay + h + durationClass};
20     std::cout << "Local time: [" << getMinutes(startDate) << ", "
21             << getMinutes(endDate) << "]" << '\n';
22
23     longestStringSize = std::max(timeZones, [] (const std::string& a,
24                                         const std::string& b) { return a.size() < b.size(); }).size();
25     for (auto timeZone: timeZones) {
26         std::cout << " " << std::setw(longestStringSize + 1) << std::left
27                     << timeZone
28                     << "[" << getMinutes(date::zoned_time(timeZone, startDate))
29                     << ", " << getMinutes(date::zoned_time(timeZone, endDate))
30                     << "]" << '\n';
31     }
32 }
33
34 int main() {
35
36     using namespace std::string_literals;
37     using namespace std::chrono;
38
39     std::cout << '\n';
40
41     constexpr auto classDay{date::year(2021)/2/1};
42     constexpr auto durationClass = 4h;
43     auto timeZones = {"America/Los_Angeles"s, "America/Denver"s,
```

```
45     "America/New_York"s, "Europe/London"s,
46     "Europe/Minsk"s, "Europe/Moscow"s,
47     "Asia/Kolkata"s, "Asia/Novosibirsk"s,
48     "Asia/Singapore"s, "Australia/Perth"s,
49     "Australia/Sydney"s};

50
51 for (auto startTime: {7h, 13h, 17h}) {
52     printStartEndTimes(date::local_days{classDay}, startTime,
53                         durationClass, timeZones);
54     std::cout << '\n';
55 }
56
57 }
```

深入 `getMinutes`(第 8 行) 和 `printStartEndTimes`(第 13 行) 函数之前，先来看看 `main` 函数。`main` 函数定义上课的日期、上课的时间段和所有时区。最后，使用基于范围的 `for` 循环(第 51 行)，遍历所有时间段的起始时间。因为 `printStartEndTimes` 函数(第 13 行)的帮助，所有必要的信息都显示出来了。

从第 18 行开始，通过将课程的开始时间和时间段添加到日历日期，来计算培训的 `startDate` 和 `endDate`。这两个值都在函数 `getMinutes`(第 8 行) 的帮助下显示出来了。`floor<std::chrono::minutes>(zonedDateTime.get_local_time())` 从 `std::chrono::zoned_time` 中获取存储的时间点，并将值以分钟为单位表示。为了正确对齐程序的输出，第 23 行确定所有时区名称中最长的一个的大小。第 25 行遍历所有时区，并显示时区名称，以及每个时间段的开始和结束。一些地方的时间甚至都在 0 点(半夜)之后了。

```

rainer@seminar:~> onlineClass

Local time: [2021-02-01 07:00:00, 2021-02-01 11:00:00]
    America/Los_Angeles [2021-01-31 22:00:00, 2021-02-01 02:00:00]
    America/Denver      [2021-01-31 23:00:00, 2021-02-01 03:00:00]
    America/New_York     [2021-02-01 01:00:00, 2021-02-01 05:00:00]
    Europe/London        [2021-02-01 06:00:00, 2021-02-01 10:00:00]
    Europe/Minsk         [2021-02-01 09:00:00, 2021-02-01 13:00:00]
    Europe/Moscow        [2021-02-01 09:00:00, 2021-02-01 13:00:00]
    Asia/Kolkata         [2021-02-01 11:30:00, 2021-02-01 15:30:00]
    Asia/Novosibirsk    [2021-02-01 13:00:00, 2021-02-01 17:00:00]
    Asia/Singapore       [2021-02-01 14:00:00, 2021-02-01 18:00:00]
    Australia/Perth      [2021-02-01 14:00:00, 2021-02-01 18:00:00]
    Australia/Sydney     [2021-02-01 17:00:00, 2021-02-01 21:00:00]

Local time: [2021-02-01 13:00:00, 2021-02-01 17:00:00]
    America/Los_Angeles [2021-02-01 04:00:00, 2021-02-01 08:00:00]
    America/Denver      [2021-02-01 05:00:00, 2021-02-01 09:00:00]
    America/New_York     [2021-02-01 07:00:00, 2021-02-01 11:00:00]
    Europe/London        [2021-02-01 12:00:00, 2021-02-01 16:00:00]
    Europe/Minsk         [2021-02-01 15:00:00, 2021-02-01 19:00:00]
    Europe/Moscow        [2021-02-01 15:00:00, 2021-02-01 19:00:00]
    Asia/Kolkata         [2021-02-01 17:30:00, 2021-02-01 21:30:00]
    Asia/Novosibirsk    [2021-02-01 19:00:00, 2021-02-01 23:00:00]
    Asia/Singapore       [2021-02-01 20:00:00, 2021-02-02 00:00:00]
    Australia/Perth      [2021-02-01 20:00:00, 2021-02-02 00:00:00]
    Australia/Sydney     [2021-02-01 23:00:00, 2021-02-02 03:00:00]

Local time: [2021-02-01 17:00:00, 2021-02-01 21:00:00]
    America/Los_Angeles [2021-02-01 08:00:00, 2021-02-01 12:00:00]
    America/Denver      [2021-02-01 09:00:00, 2021-02-01 13:00:00]
    America/New_York     [2021-02-01 11:00:00, 2021-02-01 15:00:00]
    Europe/London        [2021-02-01 16:00:00, 2021-02-01 20:00:00]
    Europe/Minsk         [2021-02-01 19:00:00, 2021-02-01 23:00:00]
    Europe/Moscow        [2021-02-01 19:00:00, 2021-02-01 23:00:00]
    Asia/Kolkata         [2021-02-01 21:30:00, 2021-02-02 01:30:00]
    Asia/Novosibirsk    [2021-02-01 23:00:00, 2021-02-02 03:00:00]
    Asia/Singapore       [2021-02-02 00:00:00, 2021-02-02 04:00:00]
    Australia/Perth      [2021-02-02 00:00:00, 2021-02-02 04:00:00]
    Australia/Sydney     [2021-02-02 03:00:00, 2021-02-02 07:00:00]

rainer@seminar:~>

```

5.5.3.3 新时钟

除了 C++11 中的墙壁时钟 `std::system_clock`、稳定时钟 `std::steady_clock` 和精确时钟 `std::high_resolution_clock` 之外，C++20 还添加了 5 个时钟。

- `std::utc_clock`: 协调世界时 (UTC) 的时钟。度量自 UTC 时间 1970 年 1 月 1 日 00:00:00 开始的时间，包括闰秒。
- `std::tai_clock`: 国际原子时 (TAI)。测量 1958 年 1 月 1 日 00:00:00 以来的时间，并将该日期的 UTC 时间往前偏移 10 秒，不包括闰秒。
- `std::gps_clock`: GPS 时间时钟，表示 全球定位系统 (GPS) 的时间。测量的是从 1980 年 1 月 6 日 00:00:00 UTC 开始的时间，不包括闰秒。
- `std::file_clock`: 文件时间时钟，是 `std::filesystem::file_time_type` 的别名。
- `std::local_t`: 表示本地时间的伪时钟。

5.5.3.4 时间的 I/O

格式化库中的 `std::chrono::parse` 函数和 `std::formatter` 函数，可以对 `chrono` 对象进行读写。

- `std::chrono::parse`: 从流中解析 `chrono` 对象。cppreference.com/parse 提供了关于格式字符串的详细信息。
- `std::formatter`: 定义各种时间类型的特化。阅读关于 `std::formatter` 格式规范的详细信息，cppreference.com/formatter。

总结

- C++20 在 chrono 库中添加了新的组件: 当日时间、日历时间和时区。
- 当日时间是指从午夜开始的时间长度, 分为小时、分钟、秒和分秒。
- 日历表示各种日历日期, 如年、月、工作日或一周的第 n 天。
- 时区表示特定于地理区域的时间。

5.6. 格式库



Cippi 在做杯子

缺少编译器支持

截止到 2020 年底，没有 C++ 编译器支持格式化库。感谢 Victor Zverovich 的原型库[fmt](#)，我可以用它进行实验。该库托管在[Compiler Explorer](#)上。当三大编译器 GCC、Clang 或 MSVC 中的一个支持 C++20 格式库，我将修改本章中的示例，使其使用标准库。

格式化库为替代[printf](#)系列函数提供了一种安全且可扩展的方案，并扩展了 I/O 流。标准库为头文件 `<format>`。格式规范遵循[Python 语法](#)，并允许指定填充字母和文本对齐、设置符号、指定数字的宽度和精度以及指定数据类型。

5.6.0.1 格式化函数

C++20 支持三个格式化函数：

函数	描述
<code>std::format</code>	返回格式化的字符串
<code>std::format_to</code>	将结果写入输出迭代器
<code>std::format_to_n</code>	向输出迭代器写入最多 n 个字符

格式化函数接受任意数量的参数。format.cpp 就来使用一下这三个函数。

```
1 // format.cpp
2
3 #include <fmt/core.h>
4 #include <fmt/format.h>
5 #include <iostream>
6 #include <iterator>
7 #include <string>
8
```

```

9 int main() {
10
11 std::cout << '\n';
12
13 std::cout << fmt::format("Hello, C++{}!\n", "20") << '\n';
14
15 std::string buffer;
16
17 fmt::format_to(
18     std::back_inserter(buffer),
19     "Hello, C++{}!\n",
20     "20");
21
22 std::cout << buffer << '\n';
23
24 buffer.clear();
25
26 fmt::format_to_n(
27     std::back_inserter(buffer), 5,
28     "Hello, C++{}!\n",
29     "20");
30 std::cout << buffer << '\n';
31
32
33 std::cout << '\n';
34
35 }

```

第 13 行的程序直接显示格式化的字符串，第 17 行和第 26 行的调用使用字符串作为缓冲区。此外，`std::format_to_n` 只将 5 个字符压入缓冲区。

Hello, C++20!

Hello, C++20!

Hello

这三个格式化函数中最有趣的部分是格式化字符串的部分 ("Hello, C++!\\n")。

5.6.1 格式化字符串

格式化字符串语法与格式化函数 `std::format`、`std::format_to` 和 `std::format_to_n` 相同。我在示例中使用了 `std::format`。

- 语法: `std::format(FormatString, Args)`

格式字符串 `FormatString` 由

- 普通字符 ({和} 除外)
- 使用 {和} 替换转义序列 {{和}}
- 替换字段

替换字段的格式为 {}

- 可以在替换字段中使用一个参数 id 和一个冒号，后面跟着一个指定的格式。这两个组件都是可选的。

参数 id 允许你指定参数在 Args 中的索引，id 以 0 开头。当不提供参数 id 时，字段将按照给定参数的相同顺序填充。要么所有替换字段都必须使用参数 id，要么不使用。

```
1 std::format("{} {}, {}", "Hello", "World")
```

等价于

```
1 std::format("{1}, {0}", "World", "Hello")
```

但下面这样不行

```
1 std::format("{1}, {}", "World", "Hello")
```

std::formatter 及其特化定义了参数类型的具体格式。

- 基于 Python 的格式规范的基本类型和 std::string: 标准格式规范
- chrono 类型:Chrono 格式规范
- 其他可格式化的类型: 用户定义的 std::formatter 特化

我将在下一节中用实践来充实理论。先从参数 id 开始，然后再讨论格式规范。

5.6.1.1 参数 id

有了参数 id，就可以对参数重新排序或处理特定的参数。

```
1 // formatArgumentID.cpp
2
3 #include <fmt/core.h>
4 #include <iostream>
5 #include <string>
6
7 int main() {
8
9     std::cout << '\n';
10
11    std::cout << fmt::format("{} {}:{} {}\n", "Hello", "World", 2020);
12
13    std::cout << fmt::format("{1} {0}: {2} {}\n", "World", "Hello", 2020);
14
15    std::cout << fmt::format("{0} {0} {1}: {2} {}\n", "Hello", "World", 2020);
16
17    std::cout << fmt::format("{0}: {2} {}\n", "Hello", "World", 2020);
18
19    std::cout << '\n';
```

```
20  
21 }
```

第 11 行按给定顺序显示参数，而第 13 行重新排序了第一个和第二个参数，第 15 行显示了第一个参数两次，第 17 行忽略了第二个参数。

下面是程序的输出：

```
Hello World: 2020!  
Hello World: 2020!  
Hello Hello World: 2020!  
Hello: 2020!
```

格式规范中使用参数 id 使得 C++20 中的文本格式化功能更加强大了。

5.6.1.2 格式规范

我不打算介绍基本类型、字符串类型或计时类型的正式格式规范。对于基本类型和 std::string，请阅读此处的完整细节:[标准格式规范](#)。可以在这里找到 chrono 类型的详细信息:[chrono 格式规范](#)。

相反，我会给出了基本类型和字符串类型的简化格式规范。

```
1 fill_align(opt) sign(opt) #(opt) 0(opt) width(opt) precision(opt) type(opt)
```

所有部分都是可选的。接下来的几节将介绍该格式规范的各个部分。

5.6.1.2.1 填充和对齐

填充字符是可选的(除 { 或 } 外的其他字符)，后面跟着一个对齐规范。

- 填充字符：默认使用空格
- 对齐：
 - <：左(默认为非数字)
 - >：右(默认为数字)
 - ^：中间

```
1 // formatFillAlign.cpp  
2  
3 #include <fmt/core.h>  
4 #include <iostream>  
5  
6 int main() {  
7  
    std::cout << '\n';  
8  
    int num = 2020;  
9  
    std::cout << fmt::format("{:6}", num) << '\n';  
10  
11  
12 }
```

```

13 std::cout << fmt::format("{:6}", 'x') << '\n';
14 std::cout << fmt::format("{:*<6}", 'x') << '\n';
15 std::cout << fmt::format("{:*>6}", 'x') << '\n';
16 std::cout << fmt::format("{:*^6}", 'x') << '\n';
17 std::cout << fmt::format("{:6d}", num) << '\n';
18 std::cout << fmt::format("{:6}", true) << '\n';
19
20 std::cout << '\n';
21
22 }

```

2020

x

*****x
x*
2020
true

5.6.1.2.2 正负号, # 和 0

正负号、# 和 0 字符仅在使用整数或浮点类型时有效。

正负号可以有以下值：

- +: 用于零和正数
- : 仅用于负数(默认)
- 空格: 填充空格表示非负数, 负号表示负数

```

1 // formatSign.cpp
2
3 #include <fmt/core.h>
4 #include <iostream>
5
6 int main() {
7
8     std::cout << '\n';
9
10    std::cout << std::format("{0:},{0:+},{0:-},{0: }", 0) << '\n';
11    std::cout << std::format("{0:},{0:+},{0:-},{0: }", -0) << '\n';
12    std::cout << std::format("{0:},{0:+},{0:-},{0: }", 1) << '\n';
13    std::cout << std::format("{0:},{0:+},{0:-},{0: }", -1) << '\n';
14
15    std::cout << '\n';
16
17 }

```

```
0,+0,0, 0  
0,+0,0, 0  
1,+1,1, 1  
-1,-1,-1,-1
```

会形成另一种形式:

- 对于整数类型，前缀 0b、0 或 0x 用于二进制、八进制或十六进制表示的类型
- 对于浮点类型，总是使用小数点
- 0: 使用 0 填充

```
1 // formatAlternate.cpp  
2  
3 #include <fmt/core.h>  
4 #include <iostream>  
5  
6 int main() {  
7  
7   std::cout << '\n';  
9  
10  std::cout << fmt::format("{:#015}", 0x78) << '\n';  
11  std::cout << fmt::format("{:#015b}", 0x78) << '\n';  
12  std::cout << fmt::format("{:#015x}", 0x78) << '\n';  
13  
14  std::cout << '\n';  
15  
16  std::cout << fmt::format("{:g}", 120.0) << '\n';  
17  std::cout << fmt::format("{:#g}", 120.0) << '\n';  
18  
19  
20  std::cout << '\n';  
21  
22 }
```

```
000000000000120  
0b0000001111000  
0x00000000000078  
  
120  
120.000
```

5.6.1.2.3 宽度和精度

可以指定类型的宽度和精度。宽度说明符可应用于数字，精度可应用于浮点数和字符串。对于浮点类型，精度指定为格式化精度；对于字符串，精度指定使用多少字符，从而最终修整字符串。若精度大于字符串的长度，则不会影响字符串。

- 宽度：可以使用正十进制数或替换字段（{} 或 {n}），n 指定的是最小宽度。
- 精度：可以使用小数点（.）后跟非负十进制数或替换字段。

以下几个例子有助于了解这些知识：

```
1 // formatWidthPrecision.cpp
2
3 #include <fmt/core.h>
4 #include <iostream>
5 #include <string>
6
7 int main() {
8
9     int i = 123456789;
10    double d = 123.456789;
11
12    std::cout << "---" << fmt::format("{}", i) << "---\n";
13    std::cout << "---" << fmt::format("{:15}", i) << "---\n"; // (w = 15)
14    std::cout << "---" << fmt::format("{:}", i, 15) << "---\n"; // (w = 15)
15
16    std::cout << '\n';
17
18    std::cout << "---" << fmt::format("{}", d) << "---\n";
19    std::cout << "---" << fmt::format("{:15}", d) << "---\n"; // (w = 15)
20    std::cout << "---" << fmt::format("{:}", d, 15) << "---\n"; // (w = 15)
21
22    std::cout << '\n';
23
24    std::string s= "Only a test";
25
26    std::cout << "---" << fmt::format("{:10.50}", d) << "---\n"; // (w = 50, p = 50)
27    std::cout << "---" << fmt::format("{:{}.{}}", d, 10, 50) << "---\n"; // (w = 50,
28                                         // p = 50)
29    std::cout << "---" << fmt::format("{:10.5}", d) << "---\n"; // (w = 10, p = 5)
30    std::cout << "---" << fmt::format("{:{}.{}}", d, 10, 5) << "---\n"; // (w = 10,
31                                         // p = 5)
32
33    std::cout << '\n';
34
35    std::cout << "---" << fmt::format("{:.500}", s) << "---\n"; // (p = 500)
36    std::cout << "---" << fmt::format("{:.{}}", s, 500) << "---\n"; // (p = 500)
37    std::cout << "---" << fmt::format("{:.5}", s) << "---\n"; // (p = 5)
38
39 }
```

源码中的 w 字符代表宽度，p 字符表示精度。当使用替换字段指定宽度时（第 14 行），不会添加额外的空格。当指定的精度高于 double 的长度（第 26 行和第 27 行）时，显示值的长度会反映精度，但这个观察结果并不适用于字符串（第 35 和 36 行）。

```
---123456789---
---      123456789---
---123456789---

---123.456789---
---      123.456789---
---123.456789---

---123.45678900000000055570126278325915336608886718750---
---123.45678900000000055570126278325915336608886718750---
---      123.46---
---      123.46---

---Only a test---
---Only a test---
---Only ---
```

5.6.1.2.4 类型

通常，编译器会推断所使用值的类型，但有时需要指定类型。以下是最重要的类型规范：

- 字符串: s
- 整型：
 - b: 二进制格式
 - B: 与 b 相同，但前缀是 0B
 - d: 十进制格式
 - o: 八进制格式
 - x: 十六进制格式
 - X: 与 x 相同，但前缀是 0X
- char 和 wchar_t：
 - b, B, d, o, x, X: 和整型一样
- bool：
 - s: true 或 false
 - b, B, d, o, x, X: 和整型一样
- 浮点：

- e: 指数格式
- E: 和 e 一样，但是指数用 E
- f, F: 定点格式，精度为 6
- g, G: 精度 6，指数用 E

若不指定类型，则显示如下所示。字符串显示为字符串，十进制格式的整数，字符显示为字符，浮点值显示为[std::to_chars](#)。

有了类型说明符，就可以在不同的数字系统中轻松显示 int 了。

```

1 // formatType.cpp
2
3 #include <fmt/core.h>
4 #include <iostream>
5
6 int main() {
7
8     int num{2020};
9
10    std::cout << "default: " << fmt::format("{}", num) << '\n';
11    std::cout << "decimal: " << fmt::format("{:d}", num) << '\n';
12    std::cout << "binary: " << fmt::format("{:b}", num) << '\n';
13    std::cout << "octal: " << fmt::format("{:o}", num) << '\n';
14    std::cout << "hexadecimal: " << fmt::format("{:x}", num) << '\n';
15
16 }
```

```

default: 2020
decimal: 2020
binary: 11111100100
octal: 3744
hexadecimal: 7e4
```

目前为止，我已经格式化了基本类型和字符串。此外，还可以格式化自定义的类型。

5.6.2 自定义的类型

要格式化自定义类型，必须为自定义类型特化类[std::formatter](#)。所以，必须实现成员函数的 parse 和 format。

- parse:
 - 接受 parse 上下文
 - 解析 parse 上下文
 - 返回格式规范末尾的迭代器

- 若发生错误，抛出 std::format_error

- format:

- 获取应该格式化的值 t 和格式上下文 fc
- 根据格式上下文进行格式化
- 将输出写入到 fc.out()
- 返回一个表示输出结束的迭代器

来把理论应用到实践中吧，先来格式化 std::vector。

5.6.2.1 格式化 std::vector

对 std::formatter 类的第一个特化非常简单，为容器的每个元素指定了格式规范。

```
1 // formatVector.cpp
2
3 #include <iostream>
4 #include <fmt/format.h>
5 #include <string>
6 #include <vector>
7
8 template <typename T>
9 struct fmt::formatter<std::vector<T>> {
10
11     std::string formatString;
12
13     auto constexpr parse(format_parse_context& ctx) {
14         formatString = "{:";  
15         std::string parseContext(std::begin(ctx), std::end(ctx));
16         formatString += parseContext;
17         return std::end(ctx) - 1;
18     }
19
20     template <typename FormatContext>
21     auto format(const std::vector<T>& v, FormatContext& ctx) {
22         auto out= ctx.out();
23         fmt::format_to(out, "[");
24         if (v.size() > 0) fmt::format_to(out, formatString, v[0]);
25         for (int i= 1; i < v.size(); ++i) fmt::format_to(out, ", " + formatString, v[i]);
26         fmt::format_to(out, "]");
27         return fmt::format_to(out, "\n");
28     }
29
30 };
31
32
33 int main() {
34
35     std::vector<int> myInts{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

```

36     std::cout << fmt::format("{:}", myInts);
37     std::cout << fmt::format("{:+}", myInts);
38     std::cout << fmt::format("{:03d}", myInts);
39     std::cout << fmt::format("{:b}", myInts);
40
41     std::cout << '\n';
42
43     std::vector<std::string> myStrings{"Only", "for", "testing", "purpose"};
44     std::cout << fmt::format("{:}", myStrings);
45     std::cout << fmt::format("{:.3}", myStrings);
46
47 }

```

`std::vector`(第 8 行) 的特化有成员函数 `parse`(第 13 行) 和 `format`(第 20 行)。`parse` 实际上创建了 `formatString`, 应用于 `std::vector` 的每个元素(第 24 和 25 行), 解析上下文 `ctx`(第 13 行) 包含冒号(:) 和右大括号({}) 之间的字符。最后, 函数返回一个指向右大括号({}) 的迭代器。成员函数 `format` 的工作方式更有趣, 格式上下文会返回输出迭代器。有了输出迭代器和函数 `std::format_to`, `std::vector` 的元素就能很好地显示出来了。

`vector`(第 35 行) 的元素有几种格式化方式。第 36 行显示数字, 第 37 行在每个数字之前写一个符号, 第 38 行将其以 3 个字符的长度对齐, 并使用 0 填充。第 39 行以二进制格式显示。剩下的两行输出 `std::vector` 的每个字符串。最后, 第 45 行将每个字符串截断为三个字符。

```

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
 [+1, +2, +3, +4, +5, +6, +7, +8, +9, +10]
 [001, 002, 003, 004, 005, 006, 007, 008, 009, 010]
 [1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010]

[Only, for, testing, purpose]
[Onl, for, tes, pur]

```

当 `std::vector` 元素越来越多的时候, 我想添加一个换行符。对于这个用例, 我扩展了格式规范的语法。

```

1 // formatVectorLinebreak.cpp
2
3 #include <algorithm>
4 #include <iostream>
5 #include <limits>
6 #include <numeric>
7 #include <fmt/format.h>
8 #include <string>
9 #include <vector>
10
11 template <typename T>
12 struct fmt::formatter<std::vector<T>> {
13

```

```

14 std::string systemFormatString;
15 std::string userFormatString;
16 int lineBreak{std::numeric_limits<int>::max()};
17
18 auto constexpr parse(format_parse_context& ctx) {
19     std::string startFormatString = "{:";  

20     std::string parseContext(std::begin(ctx), std::end(ctx));
21     auto posCurly = parseContext.find_last_of("}");
22     auto posTab = parseContext.find_last_of("|");
23     if (posTab == std::string::npos) {
24         systemFormatString = startFormatString + parseContext.substr(0, posCurly + 1);
25     }
26     else {
27         systemFormatString = startFormatString + parseContext.substr(0, posTab) + "}";
28         userFormatString = parseContext.substr(posTab + 1, posCurly - posTab - 1);
29         lineBreak = std::stoi(userFormatString);
30     }
31     return std::begin(ctx) + posCurly;
32 }
33
34 template <typename FormatContext>
35 auto format(const std::vector<T>& v, FormatContext& ctx) {
36     auto out = ctx.out();
37     auto vectorSize = v.size();
38     if (vectorSize == 0) return fmt::format_to(out, "\n");
39     for (int i = 1; i < vectorSize + 1; ++i) {
40         fmt::format_to(out, systemFormatString, v[i-1]);
41         if ((i % lineBreak) == 0) fmt::format_to(out, "\n");
42     }
43     return fmt::format_to(out, "\n");
44 }
45
46 };
47
48 int main() {
49
50     std::vector<int> myInts(100);
51     std::iota(myInts.begin(), myInts.end(), 1);
52
53     std::cout << fmt::format("{:|20}", myInts);
54     std::cout << '\n';
55     std::cout << fmt::format("{: |20}", myInts);
56     std::cout << '\n';
57     std::cout << fmt::format("{:4d|20}", myInts);
58     std::cout << '\n';
59     std::cout << fmt::format("{:10b|8}", myInts);
60
61 }

```

下面是其工作原理。我支持一个可选的 | 符号，后面跟着一个数字的格式规范。这个数字表示

是否应该引入换行符。我搜索可选的 | 符号和右大括号}。出于对程序鲁棒性的考虑，第 21 和 22 行查找最后一个符号。有了 | 和} 的索引，就可以创建字符串 systemFormatString 和 useFormatString(第 24 至 29 行)。成员函数格式使用 systemFormatString，并将其应用于 vector 的每个元素。当满足 (i % lineBreak == 0) 条件(第 41 行)时，就进行换行。

第 53 行显示一行中有 20 个元素，并使用换行符，当然可以做得更好。格式规范为 {:|20}(第 55 行)在每个数字前加一个空格。另外，第 57 行将每个元素对齐为四个字符。最后一行每行显示 8 个数字，将元素以 8 个字符对齐，并显示{:|10b|8}。

屏幕截图显示了 std::vector 的可读格式化元素。

```
1234567891011121314151617181920
2122232425262728293031323334353637383940
4142434445464748495051525354555657585960
6162636465666768697071727374757677787980
81828384858687888990919293949596979899100

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 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80
81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100

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  32  33  34  35  36  37  38  39  40
41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60
61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80
81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100

      1          10          11          100         101         110         111         1000
     1001        1010        1011        1100        1101        1110        1111        10000
    10001       10010       10011       10100       10101       10110       10111       11000
   11001       11010       11011       11100       11101       11110       11111       100000
  100001      100010      100011      100100      100101      100110      100111      101000
  101001      101010      101011      101100      101101      101110      101111      110000
   110001     110010     110011     110100     110101     110110     110111     111000
   111001     111010     111011     111100     111101     111110     111111     1000000
  1000001    1000010    1000011    1000100    1000101    1000110    1000111    1001000
  1001001    1001010    1001011    1001100    1001101    1001110    1001111    1010000
  1010001    1010010    1010011    1010100    1010101    1010110    1010111    1011000
  1011001    1011010    1011011    1011100    1011101    1011110    1011111    1100000
  1100001    1100010    1100011    1100100    1100101    1100110    1100111    1110000
```

总结

- 格式化库提供了替换 printf 类函数的安全且可扩展的方案，并扩展了 I/O 流。
- 格式规范允许定填充字母和文本对齐、设置符号、指定数字的宽度和精度，以及指定数

据类型。

- 有了 `parse` 和 `format` 函数，就可以根据需要调整自定义类型的格式。

5.7. 进一步的改善



Cippi 在上楼

5.7.1 std::bind_front

std::bind_front (Func&& Func, Args&&…Args) 会为 func 可调用对象创建包装器。std::bind_front 可以有任意数量的参数，并将其参数绑定到前面。

std::bind_front 与 std::bind

C++11 时就有了 std::bind 和 Lambda 表达式。C++20 中，添加了 std::bind_front。这可能会大伙感到疑惑。std::bind 自技术报告 1 (TR1) 开始可用，std::bind 和 Lambda 表达式可以用来替换 std::bind_front。此外，std::bind_front 看起来像是 std::bind 的姊妹，因为 std::bind 支持参数的重排。当然，使用 std::bind_front 是有原因的：与 std::bind 相反，std::bind_front 会传播底层调用操作符的异常规范。

比较 std::bind_front、std::bind 和 Lambda 表达式

```
1 // bindFront.cpp
2
3 #include <functional>
4 #include <iostream>
5
6 int plusFunction(int a, int b) {
7     return a + b;
8 }
9
10 auto plusLambda = [](int a, int b) {
11     return a + b;
12 };
13
14 int main() {
```

```

16     std::cout << '\n';
17
18     auto twoThousandPlus1 = std::bind_front(plusFunction, 2000);
19     std::cout << "twoThousandPlus1(20): " << twoThousandPlus1(20) << '\n';
20
21     auto twoThousandPlus2 = std::bind_front(plusLambda, 2000);
22     std::cout << "twoThousandPlus2(20): " << twoThousandPlus2(20) << '\n';
23
24     auto twoThousandPlus3 = std::bind_front(std::plus<int>(), 2000);
25     std::cout << "twoThousandPlus3(20): " << twoThousandPlus3(20) << '\n';
26
27     std::cout << "\n\n";
28
29     using namespace std::placeholders;
30
31     auto twoThousandPlus4 = std::bind(plusFunction, 2000, _1);
32     std::cout << "twoThousandPlus4(20): " << twoThousandPlus4(20) << '\n';
33
34     auto twoThousandPlus5 = [] (int b) { return plusLambda(2000, b); };
35     std::cout << "twoThousandPlus5(20): " << twoThousandPlus5(20) << '\n';
36
37     std::cout << '\n';
38
39 }

```

每个调用 (第 18、21、24、31 和 34 行) 获得一个带有两个参数的可调用对象，并返回一个只带有一个参数的可调用对象，并且第一个参数绑定到 2000。可调用对象是一个函数 (第 18 行)、一个 Lambda 表达式 (第 21 行) 和一个预定义的函数对象 (第 24 行)。`_1` 就是占位符 (第 31 行)，代表缺失的参数。使用 Lambda 表达式 (第 34 行)，可以直接应用一个参数，并为缺失的形参提供参数 `b`。从可读性的角度来看，`std::bind_front` 可能比 `std::bind` 或 Lambda 表达式更容易阅读。

```

twoThousandPlus1(20): 2020
twoThousandPlus2(20): 2020
twoThousandPlus3(20): 2020

twoThousandPlus4(20): 2020
twoThousandPlus5(20): 2020

```

5.7.2 std::is_constant_evaluated

`std::is_constant_evaluated` 决定函数是在编译时执行，还是在运行时执行。为什么需要类型特征库中的这个函数呢？C++20 中，我们大致介绍了三种函数：

- `constexpr` 声明函数在编译时运行：`constexpr int alwaysCompiletime();`
- `constexpr` 声明的函数可以在编译时或运行时运行；`constexpr int itDepends();`

- 一般的函数在运行时运行:int alwaysRuntime();

现在，我要写一个复杂的例子:constexpr 函数可以在编译时或运行时运行。有时这些函数的行为应该不同，这取决于函数是在编译时执行，还是在运行时执行。像 getSum 这样的 constexpr 函数有可能就在编译时运行。

使用 constexpr 的函数声明

```
1 constexpr int getSum(int l, int r) {
2     return l + r;
3 }
```

如何确定函数在编译时执行? 有三种可能。

1. constexpr 函数在编译时执行:

- 该函数用于常量求值上下文中，常量求值上下文可以在 constexpr 函数或 static_assert 中使用。
- 函数显式地希望在编译时得到结果:constexpr auto res = getSum(2000,11)。现在，getSum() 必须在编译时运行.

- 若参数不是 constexpr，则只能在运行时执行 constexpr 函数。函数 getSum(a, 11) 用一个变量调用，若这个变量没有声明为 constexpr: int a = 2000，就会出现这种情况。
- 规则 1 和规则 2 都不适用时，就可以在编译时或运行时执行 constexpr 函数。所以两个选项都是有效的，而这个决策则交由编译器完成。

正是第 3 点，使得 std::is_constant_evaluated 的功能开始发挥作用。可以检测程序是在编译时运行，还是在运行时运行，并执行不同的操作。cppreference.com/is_constant_evaluated显示了一个不过的用例。在编译时，手动计算两个数字的幂；在运行时，使用 std::pow。

编译时和运行时执行不同的代码

```
// constantEvaluated.cpp

#include <type_traits>
#include <cmath>
#include <iostream>

constexpr double power(double b, int x) {
    if (std::is_constant_evaluated() && !(b == 0.0 && x < 0)) {

        if (x == 0)
            return 1.0;
        double r = 1.0, p = x > 0 ? b : 1.0 / b;
        auto u = unsigned(x > 0 ? x : -x);
        while (u != 0) {
            if (u & 1) r *= p;
            u /= 2;
            p *= p;
        }
    }
    return std::pow(b, x);
}
```

```

18     }
19     return r;
20 }
21 else {
22     return std::pow(b, double(x));
23 }
24 }
25
26 int main() {
27
28     std::cout << '\n';
29
30     constexpr double kilo1 = power(10.0, 3);
31     std::cout << "kilo1: " << kilo1 << '\n';
32
33     int n = 3;
34     double kilo2 = power(10.0, n);
35     std::cout << "kilo2: " << kilo2 << '\n';
36
37     std::cout << '\n';
38 }
```

我想分享一个有趣的观察结果。可以在 `consteval` 声明的函数，或只能在运行时运行的函数中使用 `std::is_constant_evaluated`，这些调用的结果总是 true 或 false。

5.7.3 `std::source_location`

`std::source_location` 表示源代码的信息，包括文件名、行号和函数名。当需要关于调用站点的信息时，例如：用于调试、日志记录或测试目的时，这些信息非常有价值。类 `std::source_location` 是比 C++11 宏 `_FILE_` 和 `_LINE_` 更好的选择，推荐使用。

`std::source_location` 可以给你以下信息。

`std::source_location src`

函数	描述
<code>std::source_location::current()</code>	创建一个 <code>source_location</code> 对象 <code>src</code>
<code>src.line()</code>	返回行号
<code>src.column()</code>	返回列号
<code>src.file_name()</code>	返回文件名
<code>src.function_name()</code>	返回函数名

使用 `std::source_location::current()` 创建一个新的源位置对象 `src`，该对象表示相应调用点的信息。2020 年底，没有 C++ 编译器支持 `std::source_location`，所以下面的程序 `sourcelocation.cpp` 来自 cppreference.com/source_location。

使用 `std::source_location` 显示有关调用点的信息

```
1 // sourceLocation.cpp
2 // from cppreference.com
3
4 #include <iostream>
5 #include <string_view>
6 #include <source_location>
7
8 void log(std::string_view message,
9         const std::source_location& location = std::source_location::current())
10 {
11     std::cout << "info:"
12         << location.file_name() << ':'
13         << location.line() << ' '
14         << message << '\n';
15 }
16
17 int main()
18 {
19     log("Hello world!"); // info:main.cpp:19 Hello world!
20 }
```

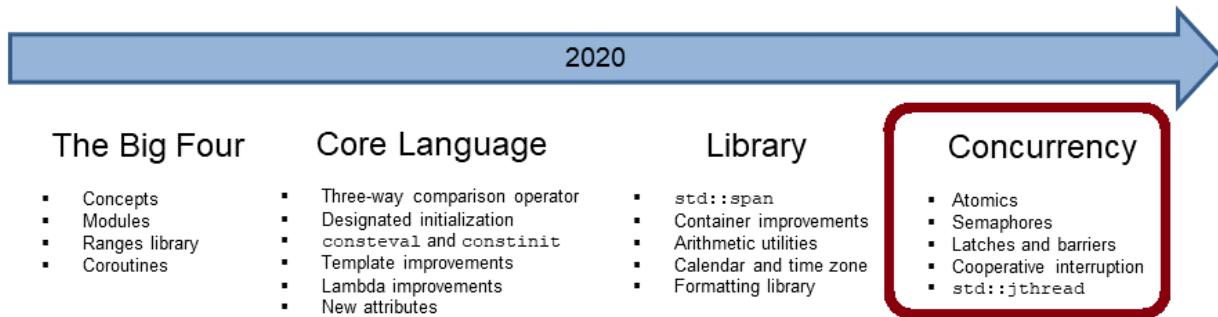
程序的输出是其源码的一部分。

总结

- `std::bind_front` 是 `std::bind(C++11)` 变体，可能比 `std::bind` 更容易使用。与 `std::bind` 不同，`std::bind_front` 不允许对其参数进行重新排列。
- `std::is_constant_evaluated` 决定函数是在编译时执行，还是在运行时执行。
- `std::source_location` 表示源代码的信息。此信息包括文件名、行号和函数名，对于调试、日志记录或测试非常有用。

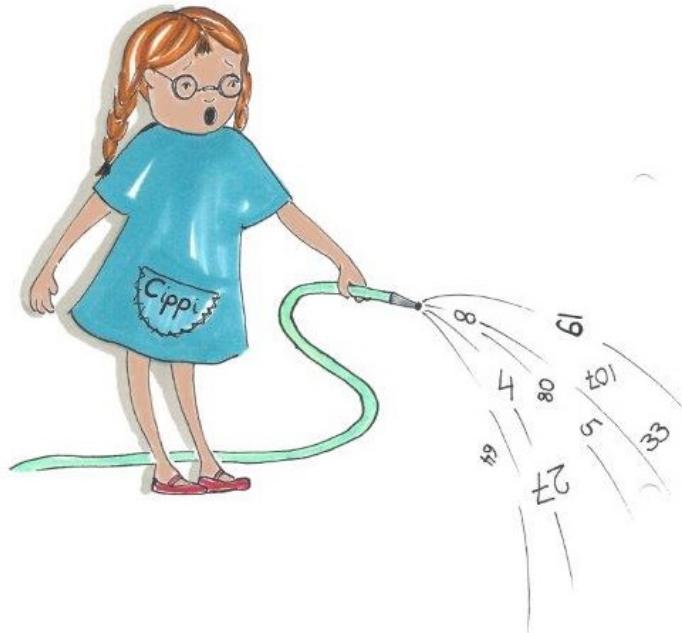
第6章 并发性

C++20



C++11 标准发布后，C++ 有了标准多线程库和内存模型，其基本构建块有，原子变量、线程、锁和条件变量。这是 C++ 标准 (如 C++20) 可以建立更高级别抽象的基础。

6.1. 协程



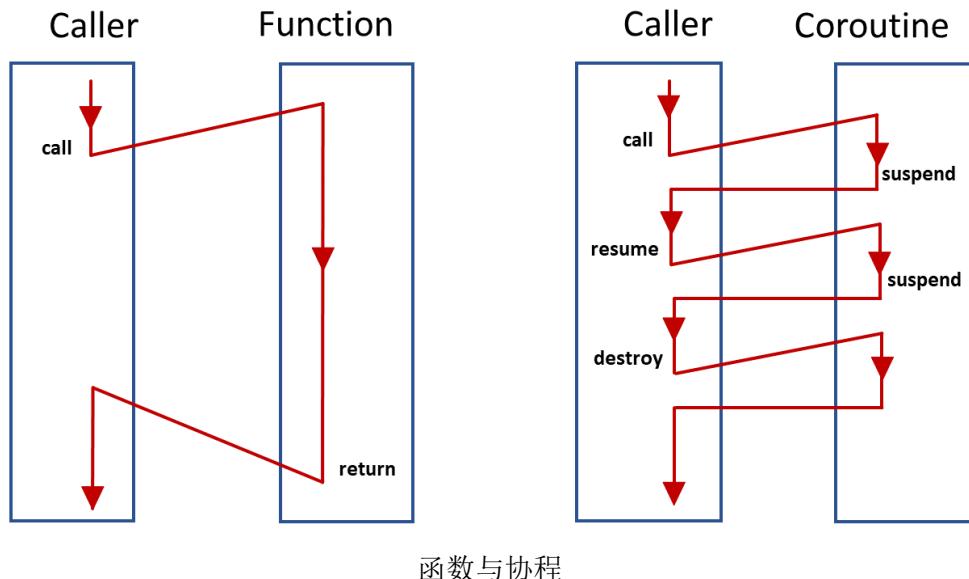
Cippi 在浇花

协程是可以在保持其状态的同时，暂停和恢复其执行的函数。

挑战：理解协程

对我来说，理解协程是一个相当大的挑战，所以强烈建议不要按顺序阅读本章后续的章节。在第一次阅读中跳过“6.1.3 框架”和“6.1.5 工作流”部分，直接阅读“7.2 Future 的变化”，“7.3 生成器的修改和泛化”和“7.4 不同的工作流”。阅读、研究和使用所提供的示例，应该会让您对协程的细节和工作流有初步和直观的了解。

这一节中提到的新思想，其实相当古老。协程这个术语是由Melvin Conway创建，他在 1963 年关于编译器构造的出版物中使用了这个词。Donald Knuth将程序称为协程的一种特殊情况。有时，需要一段时间来接受这样的设定。



(单线程中) 调用函数只能等待其返回，但自使用协程时，可以将其挂起，之后再恢复它，并且可以销毁一个处于挂起状态的协程。

C++20 使用新的关键字 `co_await` 和 `co_yield`, 扩展了 C++ 函数的执行方式。

`co_await` 表达式，可能暂停和恢复表达式的执行。使用 `co_await` 表达式时，若函数的结果不可用，调用 `auto getResult = func()` 时不会阻塞。这里的阻塞，不是消耗资源的忙等，而是资源友好的等待。

`co_yield` 表达式支持生成器函数。每次调用生成器函数时，都会返回一个新值。生成器函数是一种数据流，可以从中选择值，数据流可以是无限的，这就是 C++ 惰性求值的基础。

6.1.1 生成器函数

下面的程序非常简单。函数 `getNumbers` 会对范围内的整数加上对应的 `inc`，并放在 `vector` 中返回。`begin` 值必须小于 `end` 值，`inc` 值必须为正。

```
1 // greedyGenerator.cpp
2
3 #include <iostream>
4 #include <vector>
5
6 std::vector<int> getNumbers(int begin, int end, int inc = 1) {
7
8     std::vector<int> numbers;
9     for (int i = begin; i < end; i += inc) {
10         numbers.push_back(i);
11     }
12
13     return numbers;
```

```

14
15 }
16
17 int main() {
18
19     std::cout << '\n';
20
21     const auto numbers= getNumbers(-10, 11);
22
23     for (auto n: numbers) std::cout << n << " ";
24
25     std::cout << "\n\n";
26
27     for (auto n: getNumbers(0, 101, 5)) std::cout << n << " ";
28
29     std::cout << "\n\n";
30
31 }

```

这里，我用 `getNumbers` 重新发明了轮子，这项工作也可以用 `std::iota` 完成。

下面是输出。

```

Datei  Bearbeiten  Ansicht  Lesezeichen  Einstellungen  Hilfe
rainer@suse:~> greedyGenerator
-10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
rainer@suse:~> ■
rainer@rainer: bash

```

我对 `greedyGenerator.cpp` 有两个观察。一是，第 8 行中的 `vector` 总会得到所有值。即使只对一个有 1000 个元素容器的前 5 个元素感兴趣，这也是成立的。二是，将函数 `getNumbers` 转换为惰性生成器很容易。以下程序是有意不完整显示的，这里生成器的定义仍然处于缺失状态。

惰性生成器函数

```

1 // lazyGenerator.cpp
2
3 #include <iostream>
4
5 generator<int> generatorForNumbers(int begin, int inc = 1) {
6
7     for (int i = begin;; i += inc) {
8         co_yield i;
9     }
10
11 }
12
13 int main() {

```

```

14
15     std::cout << '\n';
16
17     const auto numbers = generatorForNumbers(-10);
18
19     for (int i= 1; i <= 20; ++i) std::cout << numbers() << " ";
20
21     std::cout << "\n\n";
22
23     for (auto n: generatorForNumbers(0, 5)) std::cout << n << " ";
24
25     std::cout << "\n\n";
26
27 }

```

getNumbers 函数返回一个 `std::vector<int>`(greedyGenerator.cpp)，而协程 `generatorForNumbers` 会返回一个生成器 (lazyGenerator.cpp)。第 17 行中的生成器数字或第 23 行中的 `generatorForNumbers(0,5)` 在请求时返回一个新数字。基于范围的 for 循环触发查询，协程的查询通过 `co_yield i` 返回值 `i`，并立即暂停其执行。若请求了一个新值，协程将在该位置恢复执行。

第 23 行中的表达式 `generatorForNumbers(0,5)` 是生成器的立即使用方式。我想明确强调，因为第 8 行中的 for 循环没有结束条件，所以协程生成器 `for numbers` 可以创建无限的数据流。若只要求有限数量的值，但这对于第 23 行并不适用，因为没有结束条件，所以表达式会持续运行。

6.1.2 特征

协程有一些独有的特征。

6.1.2.1 典型用例

协程是编写[事件驱动应用](#)的常用方法，可以是模拟、游戏、服务器、用户界面，甚至算法。协程通常也用于[协作多任务处理](#)。协作多任务处理的关键是，每个任务都需要尽可能多的时间，不休眠或等待，而是允许其他任务运行。合作多任务与抢占式多任务相反，抢占式多任务有一个调度程序，需要决定每个任务占用 CPU 的时间。

当然，也有各种类型的协程。

6.1.2.2 基本概念

C++20 中的协程是不对称的、头等的、无堆栈的。

非对称协程的工作流程会直接返回至调用方，这并不适用于对称协程。对称协程可以将其工作流委托给另一个协程。

头等协程类似于头等函数，因为协程的行为类似于数据，所以可以将它们用作函数的参数或从函数返回值，或将它们存储在变量中。

无堆栈协程可以挂起和恢复顶层协程。协程的执行和协程的产物返回给调用方，协程将其恢复状态与堆栈分开存储。无堆栈协程通常称为可恢复函数。

6.1.2.3 设计目的

Gor Nishanov 在提案[N4402](#)中描述了协程的设计目的。

协程应该是

- 高度可扩展 (可并发数十亿个协程)
- 具有高效的恢复和暂停操作，并且成本与功能的开销相当
- 与现有设施无缝交互，没有开销
- 开放的协程机制允许库设计者开发协程库，公开各种高级语义，如生成器、[goroutines](#)、任务等
- 可用于禁止异常不可用的环境

因为可扩展性和与现有设施的无缝交互的设计目标，所以协程是无堆栈的。相反，堆栈协程在 Windows 上会有 1MB 大小的默认堆栈，在 Linux 上有 2MB 大小的默认堆栈。

一个函数有四种方式可以成为协程。

6.1.2.4 成为协程

若函数使用了协程，就成为了协程

- `co_return`, 或
- `co_await`, 或
- `co_yield`, 或
- 基于范围的 `for` 循环中的 `co_await` 表达式。

协程工厂和协程对象的区别

协程这个术语通常用于两个不同方面：调用 `co_return`、`co_await` 或 `co_yield` 的函数，以及协程对象。用一个术语表示两个不同的协程方面，可能会让读者感到困惑（就像我一样）。先来说一下这两个术语。

生成 2021 的协程

```
1 MyFuture<int> createFuture() {
2     co_return 2021;
3 }
4
5 int main() {
6
7     auto fut = createFuture();
8     std::cout << "fut.get(): " << fut.get() << '\n';
9 }
10 }
```

例子中有一个 `createFuture` 函数，返回一个 `MyFuture<int>` 类型的对象，两者都称为协程。`createFuture` 函数是一个返回协程对象的协程工厂。协程对象是一个可恢复的对象，是对特定行为建模的框架。我在 `co_return` 一节中会介绍这个简单的协程的实现和使用。

6.1.2.4.1 限制

协程不能有返回语句或占位符返回类型，这适用于无约束占位符 (auto) 和有约束占位符 (concepts)。

此外，具有[可变参数](#)、`constexpr` 函数、`consteval` 函数、构造函数、析构函数和 `main` 函数的函数不能是协程。

6.1.3 架构

实现协程的框架由 20 多个函数组成，其中一些必须实现，一些可以重载。因此，可以根据需要定制协程。

协程有三个重要部分：`promise` 对象、协程句柄和协程帧。用户获得的协程句柄，可以与 `promise` 对象交互，`promise` 对象将其状态保存在协程帧中。

Promise 对象

成员函数	描述
默认构造函数	<code>Promise</code> 必须是默认可构造的。
<code>initial_suspend()</code>	确定协程在运行前是否挂起。
<code>final_suspend noexcept()</code>	确定协程在结束前是否挂起。
<code>unhandled_exception()</code>	发生异常时使用。
<code>get_return_object()</code>	返回协程对象 (可恢复对象)。
<code>return_value(val)</code>	等价于 <code>co_return val</code> 。
<code>return_void()</code>	等价于 <code>co_return</code> 。
<code>yield_value(val)</code>	等价于 <code>co_yield val</code> 。

编译器在执行协程期间会自动调用这些函数。

`get_return_object` 返回调用端，用来与协程交互的可恢复对象。`promise` 至少需要一个成员函数 `return_value`、`return_void` 或 `yield_value`。若协程永不结束，则不需要定义成员函数 `return_value` 或 `return_void`。

这三个函数 `yield_value`、`initial_suspend` 和 `final_suspend` 返回可等待对象。可等待对象表示需要对其行为进行等待的对象，且可等待属性决定协程是否可以暂停。

6.1.3.2 协程句柄

协程句柄是一个非自有句柄，用于从外部恢复或销毁协程帧。协程句柄是可恢复函数的一部分。

下面的代码段展示了一个具有协程句柄 `coro` 的简单 `Generator` 对象。

```
1 template<typename T>
2 struct Generator {
3
4     struct promise_type;
5     using handle_type = std::coroutine_handle<promise_type>;
6 }
```

```

7 Generator(handle_type h): coro(h) {}
8 handle_type coro;
9
10 ~Generator() {
11     if ( coro ) coro.destroy();
12 }
13 T getValue() {
14     return coro.promise().current_value;
15 }
16 bool next() {
17     coro.resume();
18     return not coro.done();
19 }
20 ...
21 }
```

构造函数(第 7 行)获取类型为`std::coroutine_handle<promise_type>`的 promise 的协程句柄。成员函数 next(第 16 行) 和 getValue(第 13 行) 允许调用端使用协程句柄恢复 promise(`gen.next()`) 或获取其值(`gen.getValue()`)。

调用协程

```

1 Generator<int> coroutineFactory(); // function that returns a coroutine object
2
3 auto gen = coroutineFactory();
4 gen.next();
5 auto result = gen.getValue();
```

在内部，两个函数都会使用协程句柄 `coro`(第 8 行)

- 恢复协程:`coro.resume()`(第 17 行) 或 `coro()`;
- 销毁协程:`coro.destroy()`(第 11 行);
- 检查协程的状态:`coro`(第 11 行)。

协程在其函数体结束时自动销毁，调用 `coro` 会在其最终挂起点返回 `true`。

可恢复对象需要内部类型 `promise_type`

可恢复对象(例如: `Generator`) 必须有一个内部类型 `promise_type`。或者，可以特化 `Generator` 上的`std::coroutine_traits`，并在其中定义一个公共成员 `promise_type: std::coroutine_traits<Generator>`。

6.1.3.3 协程帧

协程帧在内部使用，通常由堆分配。它由前面提到的 `promise` 对象、协程复制的参数、挂起点、当前挂起点之前生命周期结束的局部变量，以及生命周期超过当前挂起点的局部变量组成。

优化协程的分配有两个必要条件：

1. 协程的生存期必须嵌套在调用者的生存期内
2. 协程的调用者知道协程帧的大小。

6.1.4 可等待类型和具有等待模式的类型

promise 对象的三个函数分别 `yield_value`、`initial_suspend` 和 `final_suspend` 返回可等待对象。

6.1.4.1 可等待类型

可等待对象表示需要对其进行等待的对象。并且，可等待属性决定协程是否可以暂停。实际上，编译器使用 `promise` `prom` 和 `co_await` 操作符生成了三个函数调用。

编译器生成的函数调用

函数调用	编译器生成的调用
<code>yield_value</code>	<code>co_await prom.yield_value(value)</code>
<code>prom.initial_suspend()</code>	<code>co_await prom.initial_suspend()</code>
<code>prom.final_suspend()</code>	<code>co_await prom.final_suspend()</code>

`co_await` 操作符需要一个可等待对象 `as` 参数。

6.1.4.2 可等待的概念

可等待的概念需要三个功能。

可等待的概念

函数	描述
<code>await_ready</code>	指示结果是否已准备好。当它返回 <code>false</code> 时，将调用 <code>await_suspend</code> 。
<code>await_suspend</code>	将协程调度状态改为恢复或销毁。
<code>await_resume</code>	提供 <code>co_await</code> 表达式的结果。

C++20 标准定义了两个基本的可等待对象:`std::suspend_always` 和 `std::suspend_never`。

6.1.4.3 `std::suspend_always` 和 `std::suspend_never`

顾名思义，`std::suspend_always` 总是挂起，调用 `await_ready` 会返回 `false`。

```

1 struct suspend_always {
2     constexpr bool await_ready() const noexcept { return false; }
3     constexpr void await_suspend(std::coroutine_handle<>) const noexcept {}
4     constexpr void await_resume() const noexcept {}
5 };

```

相反的情况是 `std::suspend_never`，调用 `await_ready` 会返回 `true`。

```

1 struct suspend_never {
2     constexpr bool await_ready() const noexcept { return true; }
3     constexpr void await_suspend(std::coroutine_handle<>) const noexcept {}
4     constexpr void await_resume() const noexcept {}
5 };

```

可等待对象 std::suspend_always 和 std::suspend_never 是函数的基本构建块, 例如: initial_suspend 和 final_suspend。这两个函数在协程执行时自动执行:initial_suspend 在协程的开始执行, final_suspend 在协程的结束执行。

6.1.4.4 initial_suspend

成员函数 initial_suspend 返回 std::suspend_always 时, 协程将从其开始处挂起。当返回 std::suspend_never 时, 协程不会暂停。

- 立即暂停的惰性协程

```

1 std::suspend_always initial_suspend() {
2     return {};
3 }

```

- 立即运行的立即协程

```

1 std::suspend_never initial_suspend() {
2     return {};
3 }

```

6.1.4.5 final_suspend

成员函数 final_suspend 返回 std::suspend_always 时, 协程将在结束时挂起。当返回 std::suspend_never 时, 协程不会暂停。

- 结束时暂停的惰性协程

```

1 std::suspend_always final_suspend() noexcept {
2     return {};
3 }

```

- 立即协程不会在结束时暂停

```

1 std::suspend_never final_suspend() noexcept {
2     return {};
3 }

```

现在, 我们只有可等待对象, 但我们需要一些东西来进行等待。让我来填补空缺, 就是写一篇关于 Awaiters 的文章。

6.1.4.6 待等待类型 (Awaiter, 具有等待模式的类型)

有两种方法获取待等待对象。

- 定义 co_await 操作符。
- 将可等待对象变为待等待对象。

当使用 co_await 表达式时，该表达式是一个可等待表达式。此外，表达式是对 promise 对象(可等待)的调用:prom.yield_value(value)、prom.initial_suspend() 或 prom.final_suspend()。为了可读性，我在以下几行中将 promise 对象 prom 重命名为 awaitable。

现在，编译器执行以下查找规则来获取待等待对象:

1. 在 promise 对象上查找 co_await 操作符，并返回一个 awaier:

```
1 awaier = awaitable.operator co_await();
```

2. 寻找一个独立的 co_wait 操作符，并返回一个 awaier:

```
1 awaier = operator co_await();
```

3. 若没有定义 co_wait 操作符，awaitable 就成为 awaier:

```
1 awaier = awaitable;
```

awaier = awaitable

本章学习协程实现时，可能会注意大部分示例都将使 awaitable 隐式地变成 awaier。只有线程同步的例子使用了 co_await 操作符来获取待等待对象。

了解了协程的静态特性后，继续来看看其动态特性。

6.1.5 工作流

编译器转换协程，并运行两个工作流:外部 promise 工作流和内部 awaier 工作流。

6.1.5.1 Promise 工作流

在一个函数中使用 co_yield、co_await 或 co_return 时，这个函数就变成了协程，编译器就会把其主体转换成类似于下面这行代码的东西。

转换后的协程

```
1 {
2     Promise prom;
3     co_await prom.initial_suspend();
4     try {
5         <function body having co_return, co_yield, or co_wait>
6     }
7     catch (...) {
8         prom.unhandled_exception();
9     }
10    FinalSuspend:
11        co_await prom.final_suspend();
12 }
```

编译器使用 promise 对象的函数自动运行转换后的代码，我把这个工作流称为 promise 工作流。下面是这个工作流的主要步骤：

- 协程开始执行
 - 若需要，可以分配协程帧
 - 将所有函数参数复制到协程帧
 - 创建 prom(第 2 行)
 - 调用 `prom.get_return_object()` 创建协程句柄，将其保存在局部变量中。协程第一次挂起时，调用的结果将返回给调用方。
 - 调用 `prom.initial_suspend()`，并 `co_await` 其结果。promise 类型通常对急于启动的协程返回 `suspend_never`，对惰性启动的协程则返回 `suspend_always`。(第 3 行)
 - 协程的主体在 `co_await prom.initial_suspend()` 恢复时执行
- 协程到达一个暂停点
 - 返回对象 (`prom.get_return_object()`) 返回给恢复协程的调用者
- 协程达到 `co_return`
 - 为 `co_return` 或 `co_return` 表达式调用 `prom.return_void()`，其中表达式类型为 `void`
 - 为 `co_return` 表达式调用 `prom.return_value(expression)`，其中表达式具有非 `void` 类型。
 - 销毁所有在堆栈上创建的变量
 - 调用 `prom.final_suspend()`，并 `co_await` 其结果
- 销毁协程 (通过 `co_return` 一个未捕获的异常终止，或者通过协程句柄终止)
 - 调用 `promise` 对象的销毁
 - 调用函数参数的析构函数
 - 释放协程帧使用的内存
 - 将控制权转回调用方

当协程以未捕获的异常结束时，会发生以下情况：

- 捕获异常并从捕获块调用 `prom.unhandled_exception()`
- 调用 `prom.final_suspend()` 和 `co_await`(第 11 行)

在协程中使用 `co_await expr` 时，或者编译器隐式调用 `co_await prom.initial_suspend()`、`co_await prom.final_suspend()` 或 `co_await prom.yield_value(value)` 时，第二个内部 awainer 工作流将启动。

6.1.5.2 awainer 工作流

使用 `co_await expr` 会使编译器基于 `await_ready`、`await_suspend` 和 `await_resume` 函数来转换代码，所以我将转换后的代码的执行称为 awainer 工作流。

编译器使用可等待方式生成大致如下的代码，我忽略了异常处理，并使用注释描述了工作流。

生成的 awainer 工作流

```

1 awaitable.await_ready() returns false:
2
3     suspend coroutine
4
5 awaitable.await_suspend(coroHandle) returns:
6     void:
7         awaitable.await_suspend(coroHandle);
8         coro keeps suspended
9         return to caller
10
11    bool:
12        bool result = awaitable.await_suspend(coroHandle);
13        if result:
14            coro keep suspended
15            return to caller
16        else:
17            go to resumptionPoint
18
19    another coro handle:
20        auto anotherCoroHandle = awaitable.await_suspend(coroHandle);
21        anotherCoroHandle.resume();
22        return to caller
23
24 resumptionPoint:
25
26 return awaitable.await_resume();

```

只有当 `awaitable.await_ready()` 返回 `false`(第 1 行) 时，工作流才会执行。若返回 `true`，协程已经准备就绪，并返回调用 `awaitable.await_resume()` 的结果(第 26 行)。

假设 `awaitable.await_ready()` 返回 `false`。首先，将协程挂起(第 3 行)，并立即计算 `awaitable.await_suspend()` 的返回值。返回类型可以是 `void`(第 6 行)、布尔值(第 11 行)或另一个协程句柄(第 19 行)，例如：`anotherCoroHandle`。根据返回类型，程序流返回或执行另一个协程。

返回 `awaitable.await_suspend()` 的值

类型	描述
<code>void</code>	协程保持挂起，并将运行权返回给调用者。
<code>bool</code>	<code>bool == true</code> : 协程保持挂起，并将运行权返回给调用者。 <code>bool == false</code> : 协程恢复，运行权不返回给调用方。
<code>anotherCoroHandle</code>	恢复另一个协程，并将运行权返回给调用方。

若抛出异常会发什么？若异常发生在 `await_ready`、`await_suspend` 或 `await_resume` 中，情况会有所不同。

- `await_ready`: 协程不会挂起，也不会计算 `await_suspend` 或 `await_resume`。
- `await_suspend`: 异常捕获，协程恢复，异常重新抛出。不调用 `await_resume`。

- `await_resume`: `await_ready` 和 `await_suspend` 计算并返回所有值。当然，`await_resume` 的调用并不返回结果。

好了，来把理论付诸实践吧。

6.1.6 co_return

协程使用 `co_return` 作为返回语句。

6.1.6.1 future

不可否认，下面的 `eagerFuture.cpp` 中的协程是我能想象到的最简单的协程。就算这样，它还是可以做一些有意义的事情：自动存储调用结果。

```

1 // eagerFuture.cpp
2
3 #include <coroutine>
4 #include <iostream>
5 #include <memory>
6
7 template<typename T>
8 struct MyFuture {
9     std::shared_ptr<T> value;
10    MyFuture(std::shared_ptr<T> p) : value(p) {}
11    ~MyFuture() {}
12    T get() {
13        return *value;
14    }
15
16    struct promise_type {
17        std::shared_ptr<T> ptr = std::make_shared<T>();
18        ~promise_type() {}
19        MyFuture<T> get_return_object() {
20            return ptr;
21        }
22        void return_value(T v) {
23            *ptr = v;
24        }
25        std::suspend_never initial_suspend() {
26            return {};
27        }
28        std::suspend_never final_suspend() noexcept {
29            return {};
30        }
31        void unhandled_exception() {
32            std::exit(1);
33        }
34    };
35};

```

```

37 MyFuture<int> createFuture() {
38     co_return 2021;
39 }
40
41 int main() {
42
43     std::cout << '\n';
44
45     auto fut = createFuture();
46     std::cout << "fut.get(): " << fut.get() << '\n';
47
48     std::cout << '\n';
49
50 }
```

MyFuture 的行为和 future 很像，会立即运行。调用协程 createFuture(第 45 行) 返回 future，调用 fut.get(第 46 行) 获取相关 promise 中的结果。

与 future 有一个微妙的区别，协程 createFuture 的返回值在调用后可用。由于生命周期问题，返回值由 std::shared_ptr 管理(第 9 行和第 17 行)。协程总是使用 std::suspend_never(第 25 行和第 28 行)，因为在运行前后都不会挂起，所以在调用 createFuture 函数时可以执行协程。成员函数 get_return_object(第 19 行) 创建并存储协程对象的句柄，return_value(第 22 行) 存储协程的结果，该结果由 co_return 2021(第 38 行) 提供。调用端使用 fut.get(第 46 行)，并使用 future 作为 promise 的句柄。成员函数 get 将结果返回给调用端(第 13 行)。

```
fut.get(): 2021
```

有读者可能认为，不值得花费精力实现一个行为就像函数一样的协程。你说得没错！然而，这个简单的协程是编写各种 future 实现的起点。第 7 章中，可以看到更多 future 的变体。

6.1.7 co_yield

使用 co_yield，可以实现一个生成器，生成无限数据流，可以连续查询值。生成器 generatorForNumbers(int begin, int inc= 1) 的返回类型是 generator<int>，其中 generator 内部持有一个特殊的 promise p，使得使用 co_yield i 等价于使用 co_await p.yield_value(i)。语句 co_yield i 可以使用任意次。每次使用后，协程会立即挂起。

6.1.7.1 无限的数据流

infiniteDataStream.cpp 产生一个无限数据流。协程 getNext 使用 co_yield 创建一个数据流，该数据流从 start 开始，并在请求时给出按步递增的下一个值。

```

1 // infiniteDataStream.cpp
2
3 #include <coroutine>
4 #include <memory>
```

```
5 #include <iostream>
6
7 template<typename T>
8 struct Generator {
9
10     struct promise_type;
11     using handle_type = std::coroutine_handle<promise_type>;
12
13     Generator(handle_type h) : coro(h) {} // (3)
14     handle_type coro;
15
16     ~Generator() {
17         if ( coro ) coro.destroy();
18     }
19
20     Generator(const Generator&) = delete;
21     Generator& operator = (const Generator&) = delete;
22     Generator(Generator&& oth) noexcept : coro(oth.coro) {
23         oth.coro = nullptr;
24     }
25     Generator& operator = (Generator&& oth) noexcept {
26         coro = oth.coro;
27         oth.coro = nullptr;
28         return *this;
29     }
30     T getValue() {
31         return coro.promise().current_value;
32     }
33     bool next() { // (5)
34         coro.resume();
35         return not coro.done();
36     }
37     struct promise_type {
38         promise_type() = default; // (1)
39
40         ~promise_type() = default;
41
42         auto initial_suspend() { // (4)
43             return std::suspend_always{};
44         }
45         auto final_suspend() noexcept {
46             return std::suspend_always{};
47         }
48         auto get_return_object() { // (2)
49             return Generator{handle_type::from_promise(*this)};
50         }
51         auto return_void() {
52             return std::suspend_never{};
53         }
54 }
```

```

54
55     auto yield_value(const T value) { // (6)
56         current_value = value;
57         return std::suspend_always{};
58     }
59     void unhandled_exception() {
60         std::exit(1);
61     }
62     T current_value;
63 };
64
65 };
66
67 Generator<int> getNext(int start = 0, int step = 1) {
68     auto value = start;
69     while (true) {
70         co_yield value;
71         value += step;
72     }
73 }
74
75 int main() {
76
77     std::cout << '\n';
78
79     std::cout << "getNext():";
80     auto gen = getNext();
81     for (int i = 0; i <= 10; ++i) {
82         gen.next();
83         std::cout << " " << gen.getValue(); // (7)
84     }
85
86     std::cout << "\n\n";
87
88     std::cout << "getNext(100, -10):";
89     auto gen2 = getNext(100, -10);
90     for (int i = 0; i <= 20; ++i) {
91         gen2.next();
92         std::cout << " " << gen2.getValue();
93     }
94
95     std::cout << '\n';
96
97 }
```

主程序创建了两个协程。第一个 `gen`(第 80 行) 返回从 0 到 10 的值，第二个 `gen2`(第 89 行) 返回从 100 到-100 的值。在深入了解工作流程之前，可以在在线编译器[Wandbox](#)上尝试这段程序，下面是程序的输出。

```
Start
getNext(): 0 1 2 3 4 5 6 7 8 9 10
getNext(100, -10): 100 90 80 70 60 50 40 30 20 10 0 -10 -20 -30 -40 -50 -60 -70 -80 -90 -100
0
Finish
```

infinitedatstream.cpp 中的数字代表工作流第一次迭代中的步骤。

1. 创建 promise
2. 调用 `promise.get_return_object()`, 并将结果保存在一个局部变量中
3. 创建生成器
4. 调用 `promise.initial_suspend()`。生成器是惰性的, 因此总是挂起。
5. 请求下一个值, 若生成器已用完, 则直接返回。
6. 由 `co_yield` 触发调用, 下一个值在此之后可用。
7. 获取下一个值

其他迭代中, 只执行步骤 5、6 和 7。

第 7 章中的线程的修改和泛化讨论了生成器 `infinitedatstream.cpp` 的进一步改进和修改

6.1.8 co_await

`co_await` 会使协程的执行暂停或恢复。`co_await exp` 中的表达式 `exp` 必须是可等待的表达式, 即必须实现一个特定的接口, 由 `await_ready`、`await_suspend` 和 `await_resume` 三个函数组成。

`co_await` 的典型用例是等待事件的服务器。

阻塞式服务器

```
1 Acceptor acceptor{443};
2 while (true) {
3     Socket socket = acceptor.accept(); // blocking
4     auto request = socket.read(); // blocking
5     auto response = handleRequest(request);
6     socket.write(response); // blocking
7 }
```

服务器非常简单, 因为在同一个线程中依次回答每个请求。服务器监听 443 端口 (第 1 行), 接受连接 (第 3 行), 从客户端读取传入数据 (第 4 行), 并将其答案写入客户端 (第 6 行)。第 3、4 和 6 行的调用是阻塞的。

因为 `co_await`, 所以阻塞调用现在可以挂起并恢复。

等待式服务器

```
1 Acceptor acceptor{443};
2 while (true) {
3     Socket socket = co_await acceptor.accept();
```

```
4 auto request = co_await socket.read();
5 auto response = handleRequest(request);
6 co_await socket.write(response);
7 }
```

介绍使用协程进行线程同步的示例之前，我想先从一些简单的事情开始：根据请求进行启动。

6.1.8.1 根据请求进行启动

下面示例中的协程非常简单，等待预定义的 std::suspend_never()。

根据请求进行启动

```
1 // startJob.cpp
2
3 #include <coroutine>
4 #include <iostream>
5
6 struct Job {
7     struct promise_type;
8     using handle_type = std::coroutine_handle<promise_type>;
9     handle_type coro;
10    Job(handle_type h) : coro(h) {}
11    ~Job() {
12        if (coro) coro.destroy();
13    }
14    void start() {
15        coro.resume();
16    }
17
18    struct promise_type {
19        auto get_return_object() {
20            return Job{handle_type::from_promise(*this)};
21        }
22        std::suspend_always initial_suspend() {
23            std::cout << " Preparing job" << '\n';
24            return {};
25        }
26        std::suspend_always final_suspend() noexcept {
27            std::cout << " Performing job" << '\n';
28            return {};
29        }
30        void return_void() {}
31        void unhandled_exception() {}
32    };
33
34 };
35 }
36
37 Job prepareJob() {
```

```

38     co_await std::suspend_never();
39 }
40
41 int main() {
42
43     std::cout << "Before job" << '\n';
44
45     auto job = prepareJob();
46     job.start();
47
48     std::cout << "After job" << '\n';
49
50 }
```

有读者可能认为协程 `prepareJob`(第 37 行) 没有意义, 因为可等待对象总是挂起。不! 函数 `prepareJob` 至少是一个使用 `co_await`(第 38 行), 并返回一个协程对象的协程工厂。第 45 行中的函数调用 `prepareJob()` 创建了 `Job` 类型的协程对象。了解数据类型 `Job` 时, 会发现协程立即挂起, 因为 `promise` 的成员函数返回 `std::suspend_always`(第 23 行), 这正是函数启动作业的原因。`start`(第 46 行) 是恢复协程(第 15 行)所必需的, 成员函数 `final_suspend` 也返回 `std::suspend_always`(第 27 行)。

```

Before job
Preparing job
Performing job
After job
```

第 7 章的作业流部分, 我将使用 `startJob` 程序作为进一步实验的案例。

6.1.8.2 线程同步

这是典型的线程同步, 一个线程准备另一个线程等待的工作包。[条件变量](#), [promises](#) 和 [future](#), 以及[原子布尔类型](#)都可以用来创建发送方-接收方工作流。因为协程可用, 所以线程同步变得非常容易, 避免了条件变量的风险, 比如: 伪唤醒和未唤醒。

```

1 // senderReceiver.cpp
2
3 #include <coroutine>
4 #include <chrono>
5 #include <iostream>
6 #include <functional>
7 #include <string>
8 #include <stdexcept>
9 #include <atomic>
10 #include <thread>
11
12 class Event {
13 public:
```

```
14 Event() = default;
15
16 Event(const Event&) = delete;
17 Event(Event&&) = delete;
18 Event& operator=(const Event&) = delete;
19 Event& operator=(Event&&) = delete;
20
21 class Awaite;
22 Awaite operator co_await() const noexcept;
23
24 void notify() noexcept;
25
26 private:
27
28 friend class Awaite;
29
30 mutable std::atomic<void*> suspendedWaiter{nullptr};
31 mutable std::atomic<bool> notified{false};
32
33 };
34
35 class Event::Awaite {
36 public:
37     Awaite(const Event& eve) : event(eve) {}
38
39     bool await_ready() const;
40     bool await_suspend(std::coroutine_handle<> corHandle) noexcept;
41     void await_resume() noexcept {}
42
43 private:
44     friend class Event;
45
46     const Event& event;
47     std::coroutine_handle<> coroutineHandle;
48 };
49
50 bool Event::Awaite::await_ready() const {
51
52     // allow at most one waiter
53     if (event.suspendedWaiter.load() != nullptr) {
54         throw std::runtime_error("More than one waiter is not valid");
55     }
56
57     // event.notified == false; suspends the coroutine
58     // event.notified == true; the coroutine is executed like a normal function
59     return event.notified;
60 }
61
62 bool Event::Awaite::await_suspend(std::coroutine_handle<> corHandle) noexcept {
```

```

63
64     coroutineHandle = corHandle;
65
66     if (event.notified) return false;
67
68     // store the waiter for later notification
69     event.suspendedWaiter.store(this);
70
71     return true;
72 }
73
74 void Event::notify() noexcept {
75     notified = true;
76
77     // try to load the waiter
78     auto* waiter = static_cast<Awaiter*>(suspendedWaiter.load());
79
80     // check if a waiter is available
81     if (waiter != nullptr) {
82         // resume the coroutine => await_resume
83         waiter->coroutineHandle.resume();
84     }
85 }
86
87 Event::Awaiter Event::operator co_await() const noexcept {
88     return Awaiter{ *this };
89 }
90
91 struct Task {
92     struct promise_type {
93         Task get_return_object() { return {}; }
94         std::suspend_never initial_suspend() { return {}; }
95         std::suspend_never final_suspend() noexcept { return {}; }
96         void return_void() {}
97         void unhandled_exception() {}
98     };
99 }
100
101 Task receiver(Event& event) {
102     auto start = std::chrono::high_resolution_clock::now();
103     co_await event;
104     std::cout << "Got the notification! " << '\n';
105     auto end = std::chrono::high_resolution_clock::now();
106     std::chrono::duration<double> elapsed = end - start;
107     std::cout << "Waited " << elapsed.count() << " seconds." << '\n';
108 }
109
110 using namespace std::chrono_literals;
111

```

```

112 int main() {
113
114     std::cout << '\n';
115
116     std::cout << "Notification before waiting" << '\n';
117     Event event1{};
118     auto senderThread1 = std::thread([&event1]{ event1.notify(); }); // Notification
119     auto receiverThread1 = std::thread(receiver, std::ref(event1));
120
121     receiverThread1.join();
122     senderThread1.join();
123
124     std::cout << '\n';
125
126     std::cout << "Notification after 2 seconds waiting" << '\n';
127     Event event2{};
128     auto receiverThread2 = std::thread(receiver, std::ref(event2));
129     auto senderThread2 = std::thread([&event2]{
130         std::this_thread::sleep_for(2s);
131         event2.notify(); // Notification
132     });
133
134     receiverThread2.join();
135     senderThread2.join();
136
137     std::cout << '\n';
138
139 }

```

让我们看一下 `senderReceiver.cpp`。线程 `senderThread1`(第 119 行) 和 `senderThread2`(第 130 行) 分别在第 119 行和第 132 行中使用一个事件发送通知。第 102-109 行中的函数 `receiver` 是协程，在线程 `receiverThread1`(第 122 行) 和 `receiverThread2`(第 135 行) 中执行。我统计了协程开始和结束之间的时间，并将其显示出来。这个数字显示了协程等待的时间，下面的屏幕截图显示了程序的输出。

```

Start
Notification before waiting
Got the notification!
Waited 1.5738e-05 seconds.

Notification after 2 seconds waiting
Got the notification!
Waited 2.00019 seconds.

0
Finish

```

线程同步

若将无限数据流中的类 `Generator` 与本例中的类 `Event` 进行比较，就会发现细微的差异。第一种

情况下，生成器是可等待的和待等待的；第二种情况下，事件使用操作符 `co_await` 返回待等待对象。这种将关注点分离为可等待对象和待等待对象的方法，改善了代码结构。

输出显示第二个协程的执行大约需要两秒钟。原因是 `event1` 在协程挂起之前发送了通知（第 119 行），而 `event2` 在 2 秒（第 132 行）后才发送的通知。

现在，切换为实现者角色。协程的工作流程确实很难掌控。类 `Event` 有两个有趣的成员：`suspendedWaiter` 和 `notified`。第 31 行中的变量 `suspendedWaiter` 保存了信号的等待器，第 32 行中的 `notify` 具有通知的状态。

在对这两个工作流的解释中，假设在第一种情况下（第一个工作流），事件通知发生在协程等待事件之前。对于第二种情况（第二个工作流），假设正好相反。

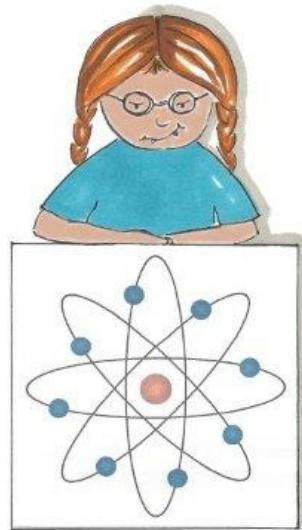
先来看看 `event1` 和第一个工作流，`event1` 在启动 `receiverThread1` 之前发送通知。调用 `event1`（第 118 行）触发 `notify`（第 75 行）。设置通知标志，然后调用 `static_cast<Awaiter*>(suspendedWaiter.load())`；装载 `waiter`。本例中，`waiter` 是 `nullptr`，因为之前没有设置，所以不会执行第 84 行中对 `waiter` 的以下恢复调用。随后执行函数 `await_ready`（第 51 行）首先检查是否有多个 `waiter`，若是的话抛出 `std::runtime_error` 异常，该方法的关键部分是返回值。`event.notification` 已经在 `notify` 方法中设置为 `true`，所以协程不会挂起，而是会像正常函数一样执行。

第二个工作流中，`co_await event2` 发生在 `event2` 发送通知之前。`co_wait event2` 会触发 `await_ready`（第 51 行），与第一个工作流不同就是该事件。`event.notified` 为 `false`，从而使协程挂起。在这里，执行 `await_suspend`（第 63 行），获取协程句柄 `corHandle`，并将其存储在变量 `coroutineHandle` 中供后续使用（第 65 行）。当然，稍后的使用意味着协程恢复执行。其次，`waiter` 存储在变量 `suspendedWaiter` 中。在之后 `event2.notify` 触发通知，执行 `notify`（第 75 行）。与第一个工作流的不同之处在于条件 `waiter != nullptr` 的计算结果为 `true`，从而 `waiter` 可以使用 `coroutineHandle` 来恢复协程。

总结

- 协程是通用函数，可以暂停和恢复其执行，同时保持其状态。
- C++20 中，没有具体的协程，但有实现协程的框架。该框架由 20 多个函数组成，部分函数必须实现，部分函数可以重载。
- 添加了新的关键字 `co_await` 和 `co_yield`，C++20 用两个新概念扩展了 C++ 函数的执行方式。
- 有了 `co_await` 表达式，才有可能暂停和恢复表达式的执行。函数 `func` 中使用 `co_await` 表达式时，若函数的结果不可用，使用 `auto getResult = func()` 不会阻塞。并且，等待也不是消耗资源的忙等，而是资源友好的等待。
- `co_yield` 能够创建无限的数据流。

6.2. 原子



Cippi 在研究原子

C++20 中，原子操作和类型得到了一些重要的扩展，最重要的可能就是原子引用和原子智能指针。

6.2.1 std::atomic_ref

类模板 `std::atomic_ref` 可对引用的对象进行原子操作。

原子对象的并发写入和读取可避免数据竞争。引用对象的生命周期必须超过 `atomic_ref` 的生命周期。当 `atomic_ref` 正在访问一个对象时，对该对象的所有其他访问必须使用 `atomic_ref`。此外，`atomic_ref` 访问对象的子对象不能被另一个 `atomic_ref` 访问。

6.2.1.1 动机

等一下！您可能认为在原子中使用引用就可以完成这项工作，但事实并不是这样。

下面的程序中，实现了一个 `ExpensiveToCopy` 类，它包括一个计数器。计数器并发递增几个线程，所以 `counter` 必须得到保护。

使用原子引用

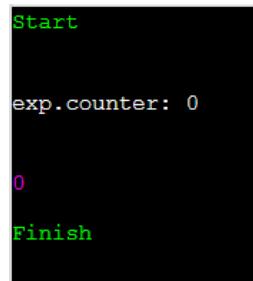
```
1 // atomicReference.cpp
2
3 #include <atomic>
4 #include <iostream>
5 #include <random>
6 #include <thread>
7 #include <vector>
8
9 struct ExpensiveToCopy {
10     int counter{};
11 };
```

```

12
13 int getRandom(int begin, int end) {
14
15     std::random_device seed; // initial seed
16     std::mt19937 engine(seed()); // generator
17     std::uniform_int_distribution<> uniformDist(begin, end);
18
19     return uniformDist(engine);
20 }
21
22 void count(ExpensiveToCopy& exp) {
23
24     std::vector<std::thread> v;
25     std::atomic<int> counter{exp.counter};
26
27     for (int n = 0; n < 10; ++n) {
28         v.emplace_back([&counter] {
29             auto randomNumber = getRandom(100, 200);
30             for (int i = 0; i < randomNumber; ++i) { ++counter; }
31         });
32     }
33
34     for (auto& t : v) t.join();
35
36 }
37
38 int main() {
39
40     std::cout << '\n';
41
42     ExpensiveToCopy exp;
43     count(exp);
44     std::cout << "exp.counter: " << exp.counter << '\n';
45
46     std::cout << '\n';
47
48 }
```

变量 `exp`(第 42 行) 是复制成本较高的对象。出于对性能的考虑, `count`(第 22 行) 通过引用获取 `exp`。函数 `count` 使用 `exp.counter` 初始化 `std::atomic<int>`(第 25 行)。下面的代码行创建了 10 个线程(第 27 行), 每个线程执行 Lambda 表达式, 该表达式通过引用接受 `counter`。Lambda 表达式获得 100 到 200 之间的随机数(第 29 行), 并以相同的频率递增计数器。函数 `getRandom`(第 13 行) 从初始种子开始, 并通过随机数生成器[马特赛特旋转演算法](#)创建一个在 100 到 200 之间的均匀分布数。

最后, `exp.counter`(第 44 行) 的近似值应该是 1500, 因为 10 个线程平均增加 150 倍。[Wandbox 在线编译器](#)上执行程序会给我一个令人惊讶的结果。

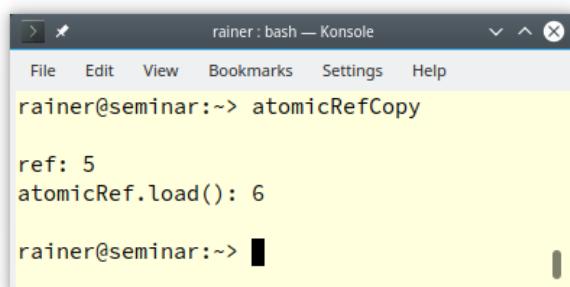


计数器为 0。发生了什么？问题在第 25 行。表达式 `std::atomic<exp. counter>` 创建一个副本。下面的小程序说明了这个问题。

复制引用

```
1 // atomicRefCopy.cpp
2
3 #include <atomic>
4 #include <iostream>
5
6 int main() {
7
8     std::cout << '\n';
9
10    int val{5};
11    int& ref = val;
12    std::atomic<int> atomicRef(ref);
13    ++atomicRef;
14    std::cout << "ref: " << ref << '\n';
15    std::cout << "atomicRef.load(): " << atomicRef.load() << '\n';
16
17    std::cout << '\n';
18 }
```

第 13 行中的增量操作没有处理引用引用（第 11 行），`ref` 的值没有改变。



将 `std::atomic<int>` 替换为 `std::atomic_ref<int>` 可以解决这个问题。

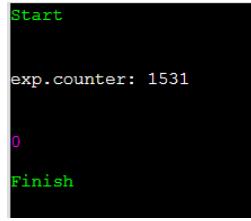
```
1 // atomicRef.cpp
2
```

```

3 #include <atomic>
4 #include <iostream>
5 #include <random>
6 #include <thread>
7 #include <vector>
8
9 struct ExpensiveToCopy {
10     int counter{};
11 };
12
13 int getRandom(int begin, int end) {
14     std::random_device seed; // initial randomness
15     std::mt19937 engine(seed()); // generator
16     std::uniform_int_distribution<> uniformDist(begin, end);
17
18     return uniformDist(engine);
19 }
20
21 void count(ExpensiveToCopy& exp) {
22
23     std::vector<std::thread> v;
24     std::atomic_ref<int> counter{exp.counter};
25
26     for (int n = 0; n < 10; ++n) {
27         v.emplace_back([&counter] {
28             auto randomNumber = getRandom(100, 200);
29             for (int i = 0; i < randomNumber; ++i) { ++counter; }
30         });
31     }
32
33     for (auto& t : v) t.join();
34 }
35
36 int main() {
37
38     std::cout << '\n';
39
40     ExpensiveToCopy exp;
41     count(exp);
42     std::cout << "exp.counter: " << exp.counter << '\n';
43
44     std::cout << '\n';
45 }

```

现在，counter 的值和预期的一样：



为了与 `std::atomic` 保持一致，类型 `std::atomic_ref` 也可以特化，并支持内置数据类型的特化。

6.2.1.2 `std::atomic_ref` 的特化

可以将 `std::atomic_ref` 特化为自定义的类型，对指针类型使用偏特化，对算术类型（如整型或浮点型）使用全特化。

6.2.1.2.1 主模板

主模板 `std::atomic_ref` 可以用 [TriviallyCopyable](#) 类型 T 进行实例化。

```
1 struct Counters {
2     int a;
3     int b;
4 };
5
6 Counter counter;
7 std::atomic_ref<Counters> cnt(counter);
```

6.2.1.2.2 指针类型的偏特化

标准为指针类型提供了偏特化: `std::atomic_ref<T*>`。

6.2.1.2.3 算术类型的特化

该标准为整型和浮点型提供了特化：

- 字符类型: `char`、`char8_t`(C++20)、`char16_t`、`char32_t` 和 `wchar_t`
- 标准有符号整型类型: 有符号 `char`、`short`、`int`、`long` 和 `long long`
- 标准的无符号整型类型: `unsigned char`、`unsigned short`、`unsigned int`、`unsigned long` 和 `unsigned long long`
- 头文件[`<cstdint>`](#)中定义的其他整数类型：
 - `int8_t`, `int16_t`, `int32_t` 和 `int64_t` (8,16,32 和 64 位有符号整数)
 - `uint8_t`, `uint16_t`, `uint32_t` 和 `uint64_t` (8,16,32 和 64 位无符号整数)
 - `int_fast8_t`, `int_fast16_t`, `int_fast32_t` 和 `int_fast64_t` (8,16,32 和 64 位的快速有符号整数)
 - `uint_fast8_t`, `uint_fast16_t`, `uint_fast32_t` 和 `uint_fast64_t` (8,16,32 和 64 位快速无符号整数)
 - `int_least8_t`, `int_least16_t`, `int_least32_t` 和 `int_least64_t` (8,16,32 和 64 位的最小有符号整数)
 - `uint_least8_t`, `uint_least16_t`, `uint_least32_t` 和 `uint_least64_t` (8,16,32 和 64 位的最小无符号整数)

- `intmax_t` 和 `uintmax_t` (最大有符号整数和无符号整数)
- `intptr_t` 和 `uintptr_t` (有符号和无符号整数, 用于保存指针)
- 标准浮点类型: 浮点型、双精度浮点型和长双精度浮点型

6.2.1.2.4 原子操作

下面是 `atomic_ref` 的操作列表。

函数	描述
<code>is_lock_free</code>	检查 <code>atomic_ref</code> 对象是否无锁。
<code>load</code>	原子地返回引用对象的值。
<code>store</code>	原子地用非原子值替换引用对象的值。
<code>exchange</code>	原子地用新值替换引用对象的值。
<code>compare_exchange_strong</code>	原子地比较并最终交换引用对象的值。
<code>compare_exchange_weak</code>	
<code>fetch_add, +=</code>	原子地对引用的对象进行加(减)法。
<code>fetch_sub, -=</code>	
<code>fetch_or, =</code>	
<code>fetch_and, &=</code>	对引用的对象原子地执行按位(AND、OR 和 XOR)操作。
<code>fetch_xor, ^=</code>	
<code>++, -</code>	递增或递减(递增前或递增后)引用对象。
<code>notify_one</code>	解除阻塞所有原子等待操作。
<code>notify_all</code>	解除单个原子等待操作。
	阻塞直到得到通知。
<code>wait</code>	将自身与旧值进行比较, 以防止为唤醒和未唤醒。 若该值与旧值不同, 则直接返回。

复合赋值运算符 (`+=`, `-=`, `|=`, `&=`, 或 `^=`) 返回新值, `fetch` 方式返回旧值。

每个函数都支持一个内存序参数, 其默认值是 `std::memory_order_seq_cst`, 也可以使用 `std::memory_order_relax`, `std::memory_order_consume`, `std::memory_order_acquire`, `std::memory_order_release` 或 `std::memory_order_acq_rel`。 `compare_exchange_strong` 和 `compare_exchange_weak` 方法有两个内存序参数, 一个用于成功情况, 另一个用于失败情况。两个调用若相等则执行原子交换, 不相等则执行原子加载。在成功的情况下, 返回 `true`, 否则返回 `false`。若只显式地提供一种内存序, 则成功和失败的情况都使用这个内存序。下面是[内存序](#)的详细信息。

当然, 并不是所有操作都适用于 `std::atomic_ref` 引用的所有类型。该表显示了所有原子操作的列表, 具体取决于 `std::atomic_ref` 引用的类型。

所有原子操作, 取决于 `std::atomic_ref` 引用的类型

函数	<code>atomic_ref<T></code>	<code>atomic_ref<integral></code>	<code>atomic_ref<floating></code>	<code>atomic_ref<T*></code>
<code>is_lock_free</code>	yes	yes	yes	yes
<code>load</code>	yes	yes	yes	yes
<code>store</code>	yes	yes	yes	yes
<code>exchange</code>	yes	yes	yes	yes
<code>compare_exchange_strong</code>	yes	yes	yes	yes
<code>compare_exchange_weak</code>	yes	yes	yes	yes
<code>fetch_add, +=</code>		yes	yes	yes
<code>fetch_sub, -=</code>		yes	yes	yes
<code>fetch_or, =</code>		yes		
<code>fetch_and, &=</code>		yes		
<code>fetch_xor, ^=</code>		yes		
<code>++, -</code>		yes		yes
<code>notify_one</code>	yes	yes	yes	yes
<code>notify_all</code>	yes	yes	yes	yes
<code>wait</code>	yes	yes	yes	yes

6.2.2 原子智能指针

`std::shared_ptr`是由一个控制块及其资源组成。控制块是线程安全的，但对资源的访问不是。所以修改引用计数器是一个原子操作，从而可以保证资源只删除一次。

线程安全的重要性

先来强调一下，`std::shared_ptr`具有定义良好的多线程语义非常重要。乍一看，对于多线程代码来说，使用`std::shared_ptr`似乎不是一个明智的选择。根据定义，它是共享和可变的，是非同步读写操作的理想候选对象，所以是未定义行为的理想候选对象。另一方面，现代C++中有一条准则：不要使用原始指针。因此，开发者应该在多线程程序中使用智能指针。

提案[N4162](#)直接解决了当前所面临的问题。这些问题可以归结为三点：一致性、正确性和性能。

- **一致性**:`std::shared_ptr`的原子操作是非原子数据类型的唯一原子操作。
- **正确性**: 全局原子操作的使用很容易出错，正确的使用方式是基于规则的。很容易忘记使用原子操作——例如：使用`ptr = localPtr`，而不是`std::atomic_store(&ptr, localPtr)`。由于数据竞争，从而导致未定义的行为。若使用原子智能指针，类型系统将不允许这样做。
- **性能**: 原子智能指针与自`atomic_*`函数相比有很大的优势。原子版本是为特殊用例设计的，可以在内部有一个`std::atomic_flag`作为一种廉价的**自旋锁**。没必要将指针函数的非原子版本设计为线程安全的，在单线程场景中使用原子操作没必要，并且还会有性能惩罚。

正确性可能是最重要的一个。为什么？答案就在提案中。该建议提供了线程安全的单链表，支持元素的插入、删除和搜索。并且，这个单链表以无锁的方式实现。

6.2.2.1 线程安全的单链表

```

template<typename T> class concurrent_stack {
    struct Node { T t; shared_ptr<Node> next; };
    atomic_shared_ptr<Node> head;
        // in C++11: remove "atomic_" and remember to use the special
        // functions every time you touch the variable
    concurrent_stack( concurrent_stack &) =delete;
    void operator=(concurrent_stack&) =delete;

public:
    concurrent_stack() =default;
    ~concurrent_stack() =default;
    class reference {
        shared_ptr<Node> p;
    public:
        reference(shared_ptr<Node> p_) : p{p_{}} {}
        T& operator* () { return p->t; }
        T* operator->() { return &p->t; }
    };

    auto find( T t ) const {
        auto p = head.load(); // in C++11: atomic_load(&head)
        while( p && p->t != t )
            p = p->next;
        return reference(move(p));
    }
    auto front() const {
        return reference(head); // in C++11: atomic_load(&head)
    }
    void push_front( T t ) {
        auto p = make_shared<Node>();
        p->t = t;
        p->next = head; // in C++11: atomic_load(&head)
        while( !head.compare_exchange_weak(p->next, p) ){}
        // in C++11: atomic_compare_exchange_weak(&head, &p->next, p);
    }
    void pop_front() {
        auto p = head.load();
        while( p && !head.compare_exchange_weak(p, p->next) ){}
        // in C++11: atomic_compare_exchange_weak(&head, &p, p->next);
    }
};

```

C++11 编译器编译程序所需的所有更改，都用红色标记。使用原子智能指针的实现要容易得多，并且不容易出错。C++20 的类型系统，不允许在原子智能指针上使用非原子操作。

提案N4162提出了新的类型 `std::atomic_shared_ptr` 和 `std::atomic_weak_ptr`。将它们合并到 ISO C++ 标准中，表示它们成为 `std::atomic` 的模板偏特化，即 `std::atomic<std::shared_ptr<T>>` 和 `std::atomic<std::weak_ptr<T>>`。

因此，`std::shared_ptr` 的原子操作在 C++20 中弃用了。

6.2.3 `std::atomic_flag` 扩展

介绍 C++20 中的 `std::atomic_flag` 扩展前，我想简单地提一下 C++11 中的 `std::atomic_flag`。若想了解更多细节，请阅读我关于 C++11 中 [std::atomic_flag](#) 的文章。

6.2.3.1 C++11

`std::atomic_flag` 是一种原子布尔值，具有清态和定态的功能。简单起见，我称 clear 状态为 false，

set 状态为 true。clear 成员函数可以将其值设置为 false。使用 test_and_set 方法，可以将值设置为 true 并返回之前的值。ATOMIC_FLAG_INIT 允许将 std::atomic_flag 初始化为 false。

std::atomic_flag 有两个重要的属性

- 唯一的无锁原子类型。
- 更高的线程构建块抽象。

C++11 中，没有成员函数要求 std::atomic_flag 的当前值而不更改它，而这在 C++20 中有所改变。

6.2.3.2 C++20 的扩展

下表展示了在 C++20 中 std::atomic_flag 具有更强大的接口。

操作	描述
atomicFlag.clear()	清除原子 flag。
atomicFlag.test_and_set()	设置原子 flag，并返回旧值。
atomicFlag.test() (C++20)	返回 flag 的值。
atomicFlag.notify_one() (C++20)	通知一个等待原子 flag 的线程。
atomicFlag.notify_all() (C++20)	通知所有等待原子 flag 的线程。
atomicFlag.wait(bool bo) (C++20)	阻塞线程，直到得到通知，将原子值改变为止。

调用 atomicFlag.test() 返回 atomicFlag 值而不更改它，使用 std::atomic_flag 进行线程同步：atomicFlag.wait()、atomicFlag.notify_one() 和 atomicFlag.notify_all()。成员函数 notify_one 或 notify_all 通知一个或所有处于等待的原子 flag，atomicFlag.wait(bo) 需要一个布尔值 bo，使用 atomicFlag.wait(bo) 将其阻塞，直到下一次通知或伪唤醒。然后，会检查 atomicFlag 的值是否等于 bo，若不等于则解除阻塞。值 bo 用作防止伪唤醒的谓词，而伪唤醒是一种错误的通知。

与 C++11 相比，std::atomic_flag 的默认构造初始化为 false。

根据 C++ 标准，更强大的原子功能，可以通过互斥量来提供。因此这些原子有一个成员函数 is_lock_free，可以用来检查原子内部是否使用了互斥量。在主流的平台上，我总是得到 false 的回答，但各位需要了解这一点。

6.2.3.3 线程的一次性同步

发送-接收工作流对于线程来说非常常见。这样的工作流中，接收者在 Future 继续工作之前等待发送者的通知。有多种方法可以实现这些工作流。C++11 可以使用条件变量或 promise/future 对，C++20 可以使用 std::atomic_flag。每种方法都有其利弊，所以我想对它们进行比较。我假设读者们并不了解条件变量，promise 和 future 的细节。因此，这里会提供一些简短的复习内容。

6.2.3.3.1 条件变量

条件变量可以扮演发送者或接收者的角色。作为发送方，可以通知一个或多个接收方。

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

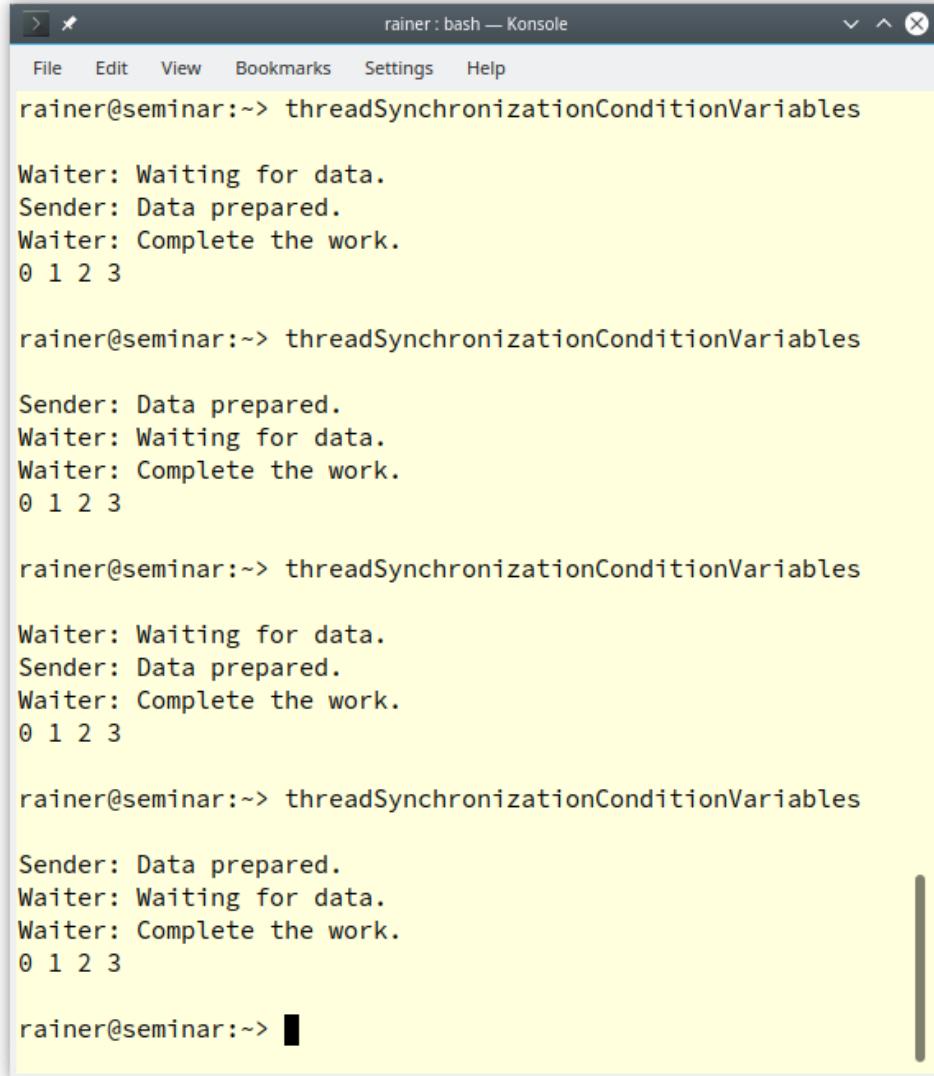
```

4 #include <condition_variable>
5 #include <mutex>
6 #include <thread>
7 #include <vector>
8
9 std::mutex mut;
10 std::condition_variable condVar;
11
12 std::vector<int> myVec{};
13
14 void prepareWork() {
15
16 {
17     std::lock_guard<std::mutex> lck(mut);
18     myVec.insert(myVec.end(), {0, 1, 0, 3});
19 }
20 std::cout << "Sender: Data prepared." << '\n';
21 condVar.notify_one();
22}
23
24 void completeWork() {
25
26 std::cout << "Waiter: Waiting for data." << '\n';
27 std::unique_lock<std::mutex> lck(mut);
28 condVar.wait(lck, []{ return not myVec.empty(); });
29 myVec[2] = 2;
30 std::cout << "Waiter: Complete the work." << '\n';
31 for (auto i: myVec) std::cout << i << " ";
32 std::cout << '\n';
33 }
34}
35
36 int main() {
37
38 std::cout << '\n';
39
40 std::thread t1(prepareWork);
41 std::thread t2(completeWork);
42
43 t1.join();
44 t2.join();
45
46 std::cout << '\n';
47
48}

```

程序有两个子线程:t1 和 t2，在第 40 行和第 41 行中获得负载 prepareWork 和 completeWork。函数 prepareWork(第 14 行) 通知它已经完成了工作的准备:condVar.notify_one()。当持有锁时，线程 t2 正在等待它的通知:condVar.wait(lck, []{return not myVec.empty();})。等待线程总是执行相同的步骤。

唤醒时，在持有锁的同时检查谓词 ([]{return not myVec.empty();})。若谓词不成立，重新回到休眠状态；否则，将继续其工作。在具体的工作流中，发送线程将初始值放入 std::vector(第 18 行)，接收线程完成(第 29 行)。



```
rainer@seminar:~> threadSynchronizationConditionVariables
Waiter: Waiting for data.
Sender: Data prepared.
Waiter: Complete the work.
0 1 2 3

rainer@seminar:~> threadSynchronizationConditionVariables
Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

rainer@seminar:~> threadSynchronizationConditionVariables
Waiter: Waiting for data.
Sender: Data prepared.
Waiter: Complete the work.
0 1 2 3

rainer@seminar:~> threadSynchronizationConditionVariables
Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

rainer@seminar:~>
```

条件变量有许多已知的问题。例如，接收者可能在没有通知的情况下唤醒，或者可能丢失通知。第一个问题称为伪唤醒，第二个问题称为未唤醒。谓词可以杜绝这两种问题。若发送方在接收方处于等待状态且未使用谓词之前发送通知，则通知可能丢失，所以接收者等待的事情永远不会发生。这是一个死锁。在研究程序的输出时会发现，若不使用谓词，那么每次运行一秒就会死锁。当然，可以使用没有谓词的条件变量。

若想知道发送-接收工作流程和条件变量陷阱的细节，请阅读我的帖子[“C++ 核心指南: 小心条件变量的陷阱”](#)。

接下来，让我使用 future/promise 来实现相同的工作流。

6.2.3.3.2 future 和 promise

promise 可以关联 future 的发送值、异常或通知。下面是使用 promise 和 future 的相应工作流。

```

1 // threadSynchronizationPromiseFuture.cpp
2
3 #include <iostream>
4 #include <future>
5 #include <thread>
6 #include <vector>
7
8 std::vector<int> myVec{};
9
10 void prepareWork(std::promise<void> prom) {
11
12     myVec.insert(myVec.end(), {0, 1, 0, 3});
13     std::cout << "Sender: Data prepared." << '\n';
14     prom.set_value();
15
16 }
17
18 void completeWork(std::future<void> fut) {
19
20     std::cout << "Waiter: Waiting for data." << '\n';
21     fut.wait();
22     myVec[2] = 2;
23     std::cout << "Waiter: Complete the work." << '\n';
24     for (auto i: myVec) std::cout << i << " ";
25     std::cout << '\n';
26
27 }
28
29 int main() {
30
31     std::cout << '\n';
32
33     std::promise<void> sendNotification;
34     auto waitForNotification = sendNotification.get_future();
35
36     std::thread t1(prepareWork, std::move(sendNotification));
37     std::thread t2(completeWork, std::move(waitForNotification));
38     t1.join();
39     t2.join();
40
41     std::cout << '\n';
42
43 }

```

研究这个工作流时，会发现同步简化为几个基本部分:prom.set_value()(第 14 行) 和 fut.wait()(第 21 行)。我跳过了这次运行的截图，结果应该与前面条件变量的运行结果相同。

这里有更多关于 promise 和 future 的信息，通常称为task。

6.2.3.3.3 std::atomic_flag

现在，我直接从 C++11 跳到 C++20。

```
1 // threadSynchronizationAtomicFlag.cpp
2
3 #include <atomic>
4 #include <iostream>
5 #include <thread>
6 #include <vector>
7
8 std::vector<int> myVec{};
9
10 std::atomic_flag atomicFlag{};
11
12 void prepareWork() {
13
14     myVec.insert(myVec.end(), {0, 1, 0, 3});
15     std::cout << "Sender: Data prepared." << '\n';
16     atomicFlag.test_and_set();
17     atomicFlag.notify_one();
18 }
19
20 void completeWork() {
21
22     std::cout << "Waiter: Waiting for data." << '\n';
23     atomicFlag.wait(false);
24     myVec[2] = 2;
25     std::cout << "Waiter: Complete the work." << '\n';
26     for (auto i: myVec) std::cout << i << " ";
27     std::cout << '\n';
28 }
29
30 }
31
32 int main() {
33
34     std::cout << '\n';
35
36     std::thread t1(prepareWork);
37     std::thread t2(completeWork);
38
39     t1.join();
40     t2.join();
41
42     std::cout << '\n';
43 }
```

准备工作的线程 (第 16 行) 将 atomicFlag 设置为 true 并发送通知。完成工作的线程等待通知。只有当 atomicFlag 等于 true 时，才会解除阻塞。

下面是使用 Microsoft 编译器运行该程序，并运行几次的截图。

```
C:\Users\seminar>threadSynchronizationAtomicFlag.exe
Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>threadSynchronizationAtomicFlag.exe
Waiter: Waiting for data.
Sender: Data prepared.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>threadSynchronizationAtomicFlag.exe
Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>threadSynchronizationAtomicFlag.exe
Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>
```

6.2.4 std::atomic 扩展

C++20 中，`std::atomic` 与 `std::atomic_ref` 一样可以用浮点类型实例化，如 `float`、`double` 和 `long double`。此外，`std::atomic_flag` 和 `std::atomic` 可以通过成员函数 `notify_one`、`notify_all` 和 `wait` 进行线程同步。通知和等待在 `std::atomic`(布尔，整型，浮点和指针) 和 `std::atomic_ref` 的所有偏和全特化上都可用。

有了 `atomic<bool>`，前面 `threadSynchronizationAtomicFlag.cpp` 可以重新进行实现。

```
1 // threadSynchronizationAtomicBool.cpp
2
3 #include <atomic>
4 #include <iostream>
5 #include <thread>
6 #include <vector>
7
```

```

8 std::vector<int> myVec{ };
9
10 std::atomic<bool> atomicBool{false};
11
12 void prepareWork() {
13
14     myVec.insert(myVec.end(), {0, 1, 0, 3});
15     std::cout << "Sender: Data prepared." << '\n';
16     atomicBool.store(true);
17     atomicBool.notify_one();
18 }
19
20 void completeWork() {
21
22     std::cout << "Waiter: Waiting for data." << '\n';
23     atomicBool.wait(false);
24     myVec[2] = 2;
25     std::cout << "Waiter: Complete the work." << '\n';
26     for (auto i: myVec) std::cout << i << " ";
27     std::cout << '\n';
28 }
29
30 }
31
32 int main() {
33
34     std::cout << '\n';
35
36     std::thread t1(prepareWork);
37     std::thread t2(completeWork);
38
39     t1.join();
40     t2.join();
41
42     std::cout << '\n';
43 }
44 }
```

若 `atomicBool == false` 成立，则调用 `atomicBool.wait(false)` 将其阻塞，所以调用 `atomicBool.store(true)`(第 16 行)将 `atomicBool` 设置为 `true` 并发送通知。

和前面一样，下面是使用 Microsoft 编译器编译，并运行四次运行。

```
cmd x64 Native Tools Command Prompt for VS 2019
C:\Users\seminar>threadSynchronizationAtomicBool.exe
Waiter: Waiting for data.
Sender: Data prepared.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>threadSynchronizationAtomicBool.exe
Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>threadSynchronizationAtomicBool.exe
Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>threadSynchronizationAtomicBool.exe
Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>
```

条件变量、promise/future 对和 std::atomic_flag

当只需要一次性通知时 (threadSynchronizationConditionVariable.cpp)，promise/future 对是比条件变量更好的选择，可以避免伪唤醒和未唤醒。此外，这种方式不需要使用锁或互斥量，也不需要使用谓词来防止伪唤醒或未唤醒。不过，其只有一个缺点：只能使用一次。

不确定我是否会使用 promise/future 对或原子类型（如 std::atomic_flag 或 std::atomic<bool>），来实现这样一个简单的线程同步工作流。从设计上讲，这些方式都是线程安全的，目前还不需要额外的保护机制。promise/future 对更容易使用，不过原子类型可能运行的更快。但有一点是确定的，尽可能不要去使用条件变量。

总结

- std::atomic_ref 对引用的对象应用原子操作。并发写和读对于引用的对象来说是原子的，没有数据竞争。引用对象的生命周期必须超过 std::atomic_ref 的生命周期。

- `std::shared_ptr` 由一个控制块及其资源组成。控制块是线程安全的，但对资源的访问不是。C++20 中，添加了原子共享指针:`std::atomic<std::shared_ptr<T>>` 和 `std::atomic<std::weak_ptr<T>>`。
- `std::atomic_flag` 作为一种原子布尔值，是 C++ 中唯一保证无锁的数据结构。其有限的接口在 C++20 中进行了扩展，现在可以返回它的值，并且可以将其用于线程同步。
- `std::atomic`，在 C++11 中引入，并在 C++20 中得到了改进。可以为浮点值特化 `std::atomic`，并且可以将其用于线程同步。

6.3. 信号量

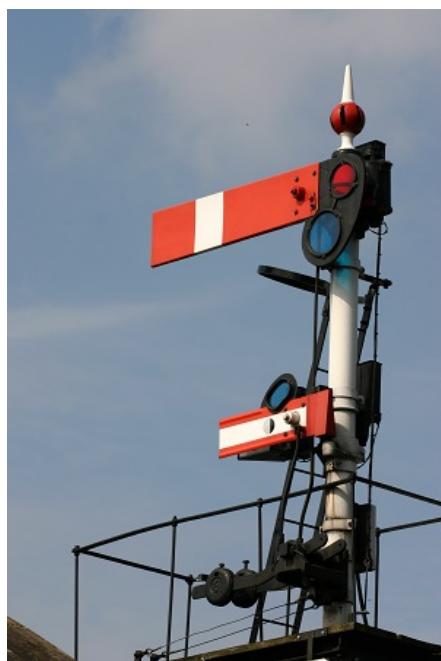


Cippi 在指挥着火车

信号量是一种同步机制，用于控制对共享资源的并发访问。计数信号量是一种特殊的信号量，其计数器的值大于零，计数器在构造函数中初始化。获取信号量会减少计数，释放信号量会增加计数器。若线程试图在计数器为零时获取信号量，则该线程将阻塞，直到另一个线程通过释放信号量来增加计数。

Edsger W. Dijkstra 发明了信号量

荷兰计算机科学家[Edsger W. Dijkstra](#)在 1965 年提出了信号量的概念，是一种具有队列和计数器的数据结构。计数器初始化为一个等于或大于零的值，支持等待和信号两种操作，操作等待获取信号量并减少计数。若计数器为零，将阻止线程获取信号量。操作信号释放信号量并增加计数。阻塞的线程会添加到队列中以避免[饥饿](#)。而最初，信号量是用在铁路上的。



信号量

相关的英文维基百科最初由 AmosWolfe 上传 - [Transferred from en.wikipedia to Commons., CC](#)

BY 2.0,

C++20 支持 `std::binary_semaphore`, 其是 `std::counting_semaphore<1>` 的别名, 最大值是 1。`std::binary_semaphores` 可以用来实现锁。

```
1 using binary_semaphore = std::counting_semaphore<1>;
```

与 `std::mutex` 相反, `std::counting_semaphore` 不绑定到线程, 所以信号量调用的获取和释放可以发生在不同的线程上。下表给出了 `std::counting_semaphore` 的接口。

成员函数	<code>std::counting_semaphore</code> 成员函数的描述
<code>sem.max()</code> (static)	返回计数器的最大值。
<code>sem.release(upd = 1)</code>	增加计数器 <code>upd</code> , 随后解锁获取信号量 <code>sem</code> 的线程。
<code>sem.acquire()</code>	将计数器值减 1 或阻塞, 直到计数器值大于 0。
<code>sem.try_acquire()</code>	若计数器大于 0, 则尝试将计数器值减 1。
<code>sem.try_acquire_for(relTime)</code>	若计数器值为 0, 则尝试将计数器值减 1 或最多阻塞时长为 <code>relTime</code> 。
<code>sem.try_acquire_until(absTime)</code>	尝试将计数器值减 1, 若计数器值为 0, 则最多阻塞时长为 <code>absTime</code> 。

构造函数调用 `std::counting_semaphore<10> sem(5)` 创建信号量 `sem`, 其最大值至少为 10, 计数器为 5。调用 `sem.max()` 返回最小的最大值。`try_acquire_for(relTime)` 需要一个时间段; 成员函数 `sem.try_acquire_until(absTime)` 需要一个时间点。使用 `sem.try_acquire`, `sem.try_acquire_for` 和 `sem.try_acquire_until` 时, 会返回一个指示调用成功与否的布尔值。

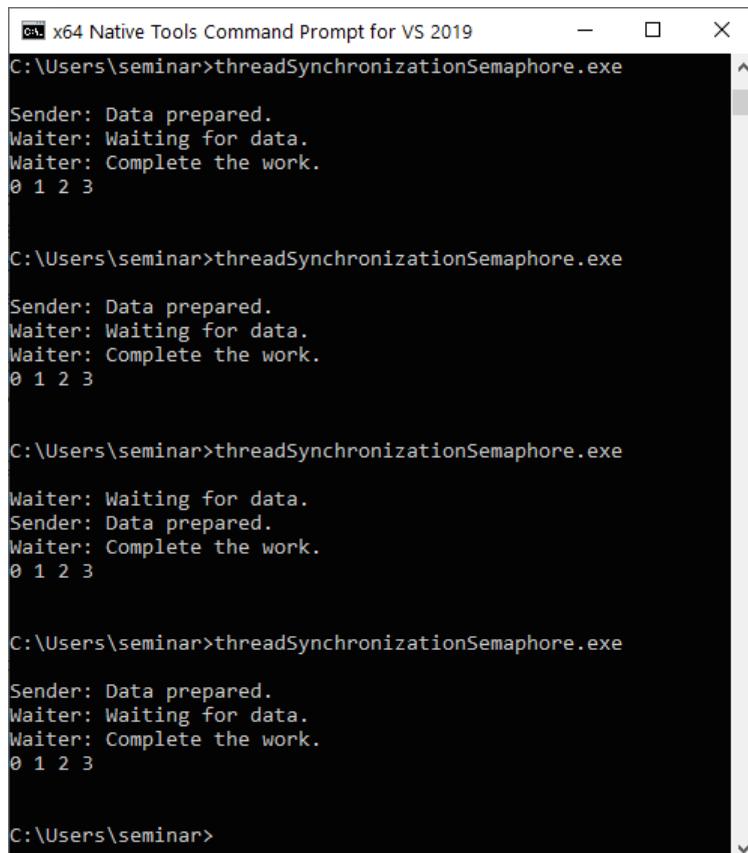
信号量通常用于发送-接收工作流。用 0 初始化信号量 `sem` 将阻塞接收方的 `sem.acquire()` 调用, 直到发送方调用 `sem.release()`。因此, 接收方等待发送方的通知。

可以对前面的线程一次性同步程序, 使用信号量进行重新实现。

```
1 // threadSynchronizationSemaphore.cpp
2
3 #include <iostream>
4 #include <semaphore>
5 #include <thread>
6 #include <vector>
7
8 std::vector<int> myVec{};
9
10 std::counting_semaphore<1> prepareSignal(0);
11
12 void prepareWork() {
13
14     myVec.insert(myVec.end(), {0, 1, 0, 3});
15     std::cout << "Sender: Data prepared." << '\n';
16     prepareSignal.release();
17 }
18
19 void completeWork() {
20
21     std::cout << "Waiter: Waiting for data." << '\n';
22     prepareSignal.acquire();
23 }
```

```
23     myVec[2] = 2;
24     std::cout << "Waiter: Complete the work." << '\n';
25     for (auto i: myVec) std::cout << i << " ";
26     std::cout << '\n';
27 }
28 }
29
30 int main() {
31
32     std::cout << '\n';
33
34     std::thread t1(prepareWork);
35     std::thread t2(completeWork);
36
37     t1.join();
38     t2.join();
39
40     std::cout << '\n';
41
42 }
```

std::counting_semaphore prepareSignal(第 10 行) 的值可以是 0 和 1。示例中，初始化为 0(第 10 行)，可以使用 prepareSignal.release() 将值设置为 1(第 16 行)，并解除 prepareSignal.acquire()(第 22 行) 的阻塞。



```
C:\x64 Native Tools Command Prompt for VS 2019
C:\Users\seminar>threadSynchronizationSemaphore.exe

Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>threadSynchronizationSemaphore.exe

Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>threadSynchronizationSemaphore.exe

Waiter: Waiting for data.
Sender: Data prepared.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>threadSynchronizationSemaphore.exe

Sender: Data prepared.
Waiter: Waiting for data.
Waiter: Complete the work.
0 1 2 3

C:\Users\seminar>
```

总结

- 信号量是一种同步机制，用于控制对共享资源的并发访问。
- C++20 中的计数信号量有一个计数器。获取信号量会减少计数器，释放信号量会增加计数器。若线程试图在计数器为零时获取信号量，则该线程将阻塞，直到另一个线程通过释放信号量来增加计数器。

6.4. 门闩和栅栏



Cippi 在栅栏外等待

锁存器(门闩)和栅栏属于协调类型，可以阻塞线程，直到计数器变为零。C++20 中有两种类型的门闩和栅栏:std::latch 和 std::barrier。并发调用 std::latch 或 std::barrier 的成员函数不会产生数据竞争。

首先，有两个问题：

- 这两种协调线程的机制之间有什么区别?std::latch 只能使用一次，但 std::barrier 可以使用多次。std::latch 在多线程管理一个任务时很有用，std::barrier 可以帮助管理多个线程的重复任务。此外，std::barrier 可在完成步骤中执行函数，这里的“完成步骤”指的是计数器变为零时的状态。
- 在 C++11 和 C++14 中用 future、线程或条件变量与锁结合的情况下，门闩和栅栏支持哪些用例？门闩和栅栏没有解决新的用例，但是它们的使用要容易得多。因为它们经常在内部使用无锁机制，所以性能会更好。

6.4.1 std::latch

现在，来了解一下 std::latch 的接口。

成员函数	std::latch 的成员函数的描述
lat.count_down(upd = 1)	按 upd 原子地递减计数器，而不阻塞调用者。
lat.try_wait()	若 counter == 0 则返回 true。
lat.wait()	若 counter == 0 立即返回。若不阻塞，直到 counter == 0 再返回。
lat.arrive_and_wait(upd = 1)	等价于 count_down(upd); wait();

upd 的默认值为 1。当 upd 大于计数器或为负数时，行为未定义。使用 lat.try_wait() 从不会真正等待，正如其名称一样。

下面的 bossWorkers.cpp 中，使用两个 std::latch 来构建一个 boss-workers 工作流。我使用 synchronizedOut 函数将输出同步到 std::cout(第 13 行)，使工作流更容易使用这种同步方式。

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

```
4 #include <mutex>
5 #include <latch>
6 #include <thread>
7
8 std::latch workDone(6);
9 std::latch goHome(1);
10
11 std::mutex coutMutex;
12
13 void synchronizedOut(const std::string& s) {
14     std::lock_guard<std::mutex> lo(coutMutex);
15     std::cout << s;
16 }
17
18 class Worker {
19 public:
20     Worker(std::string n): name(n) { }
21
22     void operator() () {
23         // notify the boss when work is done
24         synchronizedOut(name + ": " + "Work done!\n");
25         workDone.count_down();
26
27         // waiting before going home
28         goHome.wait();
29         synchronizedOut(name + ": " + "Good bye!\n");
30     }
31 private:
32     std::string name;
33 };
34
35 int main() {
36
37     std::cout << '\n';
38
39     std::cout << "BOSS: START WORKING! " << '\n';
40
41     Worker herb(" Herb");
42     std::thread herbWork(herb);
43
44     Worker scott(" Scott");
45     std::thread scottWork(scott);
46
47     Worker bjarne(" Bjarne");
48     std::thread bjarneWork(bjarne);
49
50     Worker andrei(" Andrei");
51     std::thread andreiWork(andrei);
52 }
```

```

53 Worker andrew(" Andrew");
54 std::thread andrewWork(andrew);
55
56 Worker david(" David");
57 std::thread davidWork(david);
58
59 workDone.wait();
60
61 std::cout << '\n';
62
63 goHome.count_down();
64
65 std::cout << "BOSS: GO HOME!" << '\n';
66
67 herbWork.join();
68 scottWork.join();
69 bjarneWork.join();
70 andreiWork.join();
71 andrewWork.join();
72 davidWork.join();
73
74 }

```

六个工人 herb, scott, bjarne, andrei, andrew 和 david(第 41 - 57 行) 必须完成他们的工作。当每个人都完成了他的工作时, 开始倒数 std::latch workDone(第 25 行)。boss(主线程) 在第 59 行阻塞, 直到计数器变为 0。当计数器为 0 时, boss 使用第二个 std::latch goHome 向其员工发出回家的信号。所以, 初始计数器是 1(第 9 行), goHome.wait() 会阻塞直到计数器变为 0。

```

C:\Users\seminar>bossWorkers.exe

BOSS: START WORKING!
    Herb: Work done!
        Andrei: Work done!
        Bjarne: Work done!
        Andrew: Work done!
        David: Work done!
        Scott: Work done!

BOSS: GO HOME!
        David: Good bye!
        Andrew: Good bye!
        Scott: Good bye!
        Andrei: Good bye!
        Bjarne: Good bye!
        Herb: Good bye!

C:\Users\seminar>

```

研究这个工作流时, 可能会注意到其可以在没有老板的情况下执行。

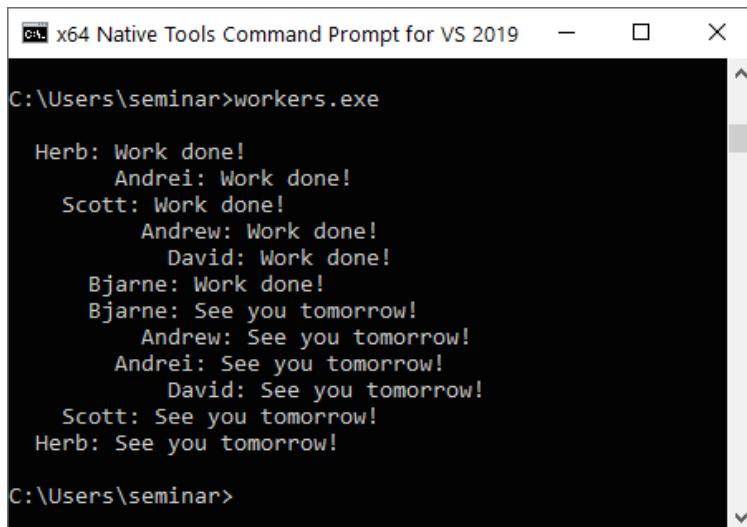
```
1 // workers.cpp
2
3 #include <iostream>
4 #include <barrier>
5 #include <mutex>
6 #include <thread>
7
8 std::latch workDone(6);
9 std::mutex coutMutex;
10
11 void synchronizedOut(const std::string& s) {
12     std::lock_guard<std::mutex> lo(coutMutex);
13     std::cout << s;
14 }
15
16 class Worker {
17 public:
18     Worker(std::string n): name(n) { }
19
20     void operator() () {
21         synchronizedOut(name + ": " + "Work done!\n");
22         workDone.arrive_and_wait(); // wait until all work is done
23         synchronizedOut(name + ": " + "See you tomorrow!\n");
24     }
25 private:
26     std::string name;
27 };
28
29 int main() {
30
31     std::cout << '\n';
32
33     Worker herb(" Herb");
34     std::thread herbWork(herb);
35
36     Worker scott(" Scott");
37     std::thread scottWork(scott);
38
39     Worker bjarne(" Bjarne");
40     std::thread bjarneWork(bjarne);
41
42     Worker andrei(" Andrei");
43     std::thread andreiWork(andrei);
44
45     Worker andrew(" Andrew");
46     std::thread andrewWork(andrew);
47
48     Worker david(" David");
49     std::thread davidWork(david);
```

```

50
51     herbWork.join();
52     scottWork.join();
53     bjarneWork.join();
54     andreiWork.join();
55     andrewWork.join();
56     davidWork.join();
57
58 }

```

这个简化的工作流程中，不需要添加太多的内容。`wordDone.arrival_and_wait()`(第 22 行) 等价于 `count_down(upd);wait();`。所以，工人间可以相互协调，不再需要老板，就和 `bossWorkers.cpp` 一样。



`std::barrier` 与 `std::latch` 类似。

6.4.2 std::barrier

`std::latch` 和 `std::barrier` 之间有两个区别。首先，可以多次使用 `std::barrier`；其次，可以为下一步(迭代)设置计数器。计数器变为零之后，“完成步骤”立即开始。“完成步骤”中，调用一个可调用对象。`std::barrier` 在它的构造函数中，可以获得这个可调用的对象。

“完成步骤”执行以下操作：

1. 阻塞所有线程。
2. 任意线程解除阻塞并执行可调用对象。
3. 若完成了“完成步骤”，则所有线程都将解除阻塞。

成员函数	<code>std::barrier bar</code> 成员函数的描述
<code>bar.arrive(upd)</code>	按 <code>upd</code> 自动递减计数器。
<code>bar.wait()</code>	在同步点上阻塞，直到完成步骤完成。
<code>bar.arrive_and_wait()</code>	等价于 <code>wait(arrive())</code>
<code>bar.arrive_and_drop()</code>	将当前阶段和后续阶段的计数器减 1。
<code>std::barrier::max</code>	获取实现支持的最大值。

bar.arrival_and_drop() 会让计数器在下一阶段减 1, fullTimePartTimeWorkers.cpp 将第二阶段的工作人员数量减半。

全职和兼职员工

```
1 // fullTimePartTimeWorkers.cpp
2
3 #include <iostream>
4 #include <barrier>
5 #include <mutex>
6 #include <string>
7 #include <thread>
8
9 std::barrier workDone(6);
10 std::mutex coutMutex;
11
12 void synchronizedOut(const std::string& s) {
13     std::lock_guard<std::mutex> lo(coutMutex);
14     std::cout << s;
15 }
16
17 class FullTimeWorker {
18 public:
19     FullTimeWorker(std::string n): name(n) { }
20
21     void operator() () {
22         synchronizedOut(name + ": " + "Morning work done!\n");
23         workDone.arrive_and_wait(); // Wait until morning work is done
24         synchronizedOut(name + ": " + "Afternoon work done!\n");
25         workDone.arrive_and_wait(); // Wait until afternoon work is done
26
27     }
28 private:
29     std::string name;
30 };
31
32 class PartTimeWorker {
33 public:
34     PartTimeWorker(std::string n): name(n) { }
35
36     void operator() () {
37         synchronizedOut(name + ": " + "Morning work done!\n");
38         workDone.arrive_and_drop(); // Wait until morning work is done
39     }
40 private:
41     std::string name;
42 };
43
44 int main() {
```

```

45
46     std::cout << '\n';
47
48     FullTimeWorker herb(" Herb");
49     std::thread herbWork(herb);
50
51     FullTimeWorker scott(" Scott");
52     std::thread scottWork(scott);
53
54     FullTimeWorker bjarne(" Bjarne");
55     std::thread bjarneWork(bjarne);
56     PartTimeWorker andrei(" Andrei");
57     std::thread andreiWork(andrei);
58
59     PartTimeWorker andrew(" Andrew");
60     std::thread andrewWork(andrew);
61
62     PartTimeWorker david(" David");
63     std::thread davidWork(david);
64
65     herbWork.join();
66     scottWork.join();
67     bjarneWork.join();
68     andreiWork.join();
69     andrewWork.join();
70     davidWork.join();
71
72 }

```

这个工作流由两种工人组成: 全职工人 (第 17 行) 和兼职工人 (第 32 行)。兼职工人在上午工作, 全职工人在上午和下午工作, 所以全职工人使用 `workDone.arrive_and_wait()` 两次 (第 23 行和第 25 行), 而兼职工人只使用 `workDone.arrive_and_drop()` (第 38 行) 一次。`workDone.arrive_and_drop()` 会让兼职工人跳过下午的工作。相应地, 计数器在第一阶段 (上午) 的值为 6, 在第二阶段 (下午) 的值会变为 3。

```

C:\x64 Native Tools Command Prompt for...
C:\Users\seminar>fullTimePartTimeWorkers.exe

Herb: Morning work done!
Bjarne: Morning work done!
Andrei: Morning work done!
Andrew: Morning work done!
David: Morning work done!
Scott: Morning work done!
Scott: Afternoon work done!
Bjarne: Afternoon work done!
Herb: Afternoon work done!

C:\Users\seminar>

```

全职和兼职工作者

总结

- 闩门和栅栏都是协调类型，可以阻塞线程，直到计数器变为零。`std::latch` 只能使用一次，而 `std::barrier` 可以使用多次。
- `std::latch` 多用于多线程管理一次性任务，`std::barrier` 多用于多线程管理重复的任务。

6.5. 中断协程



Cippi 在停车标志前停了下来

中断线程的功能是基于 `std::stop_token`、`std::stop_callback` 和 `std::stop_source` 实现。

首先，杀死线程为什么不是一个好主意？

杀死线程很危险

因为不知道线程当前的状态，可能会出现以下是两种的恶性结果。

- 线程的工作只完成了一半。因为不知道它的作业的状态，所以也不知道程序的状态，从而以未定义的行为结束。
- 线程可能处于临界区，并且已经锁定了一个互斥锁。在线程锁定互斥锁时杀死线程，很可能导致死锁。

6.5.1 `std::stop_token`, `std::stop_callback` 和 `std::stop_source`

`std::stop_token`、`std::stop_callback` 或 `std::stop_source` 允许线程异步请求执行停止或查询一个执行是否得到了停止信号。`std::stop_token` 可以传递一个操作，然后用于主动轮询令牌，以获得停止请求或通过 `std::stop_callback` 注册回调。停止请求由 `std::stop_source` 发送，这个信号影响所有相关的 `std::stop_token`。`std::stop_source`、`std::stop_token` 和 `std::stop_callback` 三个类共享一个相关停止状态的所有权，`request_stop()`、`stop_requested()` 和 `stop_possible()` 是原子的。

可以用两种方法构造 `std::stop_source`:

```
1 stop_source();
2 explicit stop_source(std::nostopstate_t) noexcept;
```

默认构造函数(第 1 行)构造了一个 `std::stop_source`，其中包含一个新的停止状态。构造函数采用 `std::nostopstate_t`(第 2 行) 构造一个空的 `std::stop_source`，没有相关的停止状态。

`std::stop_source` `src` 组件提供了以下成员函数来处理停止请求。

成员函数	std::stop_source src 成员函数的描述
src.get_token()	若 stop_possible() 为 true, 为相关的停止状态返回一个 stop_token。 否则, 返回一个默认构造的(空)stop_token。
src.stop_possible()	若 src 可以请求停止, 则为 true。
src.stop_requested()	若使用了 stop_possible() 和 request_stop() 中其中一个, 则为 true。
src.request_stop()	若 stop_possible() 和 !stop_requested(), 则调用停止请求。 否则, 调用无效。

src.stop_possible() 表示 src 有一个相关的停止状态。src.stop_requested() 在 src 有一个相关的停止状态并且之前没有要求停止时, 返回 true。使用 src.request_stop() 是成功的, 若 src 有一个相关的停止状态, 并且之前没有请求停止, 则返回 true。

src.get_token() 返回停止令牌 stoken。可以通过 stoken 检查停止请求是否已经发出, 或可以为其相关的停止源 src 发出。停止标记 stoken 可以对停止源 src 进行观察。

成员函数	std::stop_token stoken 成员函数的描述
stoken.stop_possible()	若 stoken 有关联的停止状态, 则返回 true。
stoken.stop_requested()	若在相关的 std::stop_source src 上调用 request_stop(), 则为 true, 否则为 false。

没有关联停止状态的默认构造令牌。若已经发出停止请求, stoken.stop_possible 也返回 true。在停止令牌具有相关的停止状态, 并且已经接收到停止请求时, stoken.stop_requested() 会返回 true。

若 std::stop_token 需要暂时禁用, 可以用默认构造的令牌替换。默认构造的令牌没有关联的停止状态。下面的代码片段展示了如何禁用和启用线程接受停止请求的能力。

暂时禁用停止令牌

```

1 std::jthread jthr([](std::stop_token stoken) {
2     ...
3     std::stop_token interruptDisabled;
4     std::swap(stoken, interruptDisabled);
5     ...
6     std::swap(stoken, interruptDisabled);
7     ...
8 }
```

std::stop_token interruptDisabled 没有关联的停止状态, 所以线程 jthr 可以接受除第 4 和第 5 行以外的所有行中的停止请求。

下一个例子展示了如何使用回调。

```

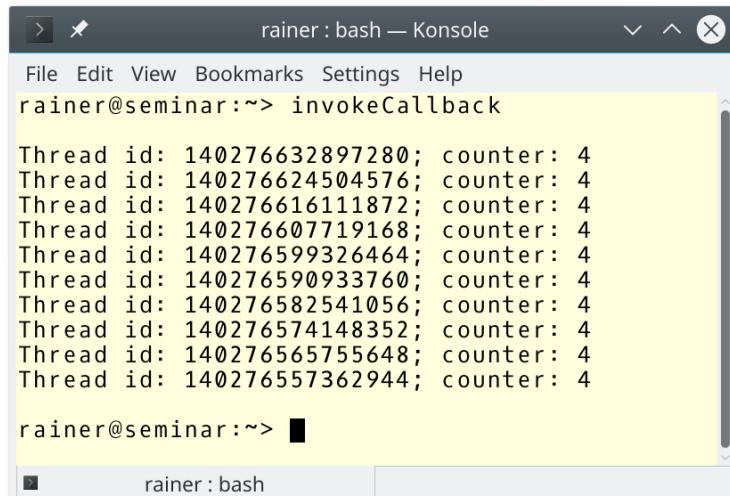
1 // invokeCallback.cpp
2
3 #include <atomic>
4 #include <chrono>
5 #include <iostream>
6 #include <thread>
```

```

7 #include <vector>
8
9 using namespace std::literals;
10
11 auto func = [](std::stop_token stoken) {
12     std::atomic<int> counter{0};
13     auto thread_id = std::this_thread::get_id();
14     std::stop_callback callBack(stoken, [&counter, thread_id] {
15         std::cout << "Thread id: " << thread_id
16             << "; counter: " << counter << '\n';
17     });
18     while (counter < 10) {
19         std::this_thread::sleep_for(0.2s);
20         ++counter;
21     }
22 };
23
24 int main() {
25
26     std::cout << '\n';
27
28     std::vector<std::jthread> vecThreads(10);
29     for(auto& thr: vecThreads) thr = std::jthread(func);
30
31     std::this_thread::sleep_for(1s);
32
33     for(auto& thr: vecThreads) thr.request_stop();
34
35     std::cout << '\n';
36
37 }

```

这十个线程中的每一个都调用 Lambda 函数 func(第 11-22 行), 第 14-17 行中的回调显示线程 id 和计数器。由于主线程的休眠时间为 1 秒, 子线程的休眠时间为 0.2 秒, 因此调用回调时计数器为 4。thr.request_stop() 在每个线程上都会触发回调。



6.5.1.1 汇入线程

std::jthread 是一个 std::thread，其功能就是可以发出中断信号和自动 join()。为了支持这个功能，其具有一个 std::stop_token。

成员函数	std::jthread jthr 用于停止令牌处理成员函数的描述
t.get_stop_source()	返回与共享停止状态关联的 std::stop_source 对象。
t.get_stop_token()	返回与共享停止状态关联的 std::stop_token 对象。
t.request_stop()	通过共享停止状态停止执行请求。

6.5.1.2 condition_variable_any 的新重载

std::condition_variable_any 的三个 wait 变量的 wait、wait_for 和 wait_until 得到新的重载，都有一个 std::stop_token。

```
1 template <class Predicate>
2 bool wait(Lock& lock,
3            stop_token stoken,
4            Predicate pred);
5
6 template <class Rep, class Period, class Predicate>
7 bool wait_for(Lock& lock,
8               stop_token stoken,
9               const chrono::duration<Rep, Period>& rel_time,
10              Predicate pred);
11
12 template <class Clock, class Duration, class Predicate>
13 bool wait_until(Lock& lock,
14                  stop_token stoken,
15                  const chrono::time_point<Clock, Duration>& abs_time,
16                  Predicate pred);
```

这些新的重载需要一个谓词，所提供的版本需要确保线程在对传递的 std::stop_token stoken 的停止请求发出信号时得到通知。其会返回一个布尔值，指示谓词的结果是否为 true。返回的布尔值与是否请求停止或是否触发超时无关。这三个重载等价于以下表达式：

```
1 // wait in lines 1 - 4
2 while (!stoken.stop_requested()) {
3     if (pred()) return true;
4     wait(lock);
5 }
6 return pred();
7
8 // wait_for in lines 6 - 10
9 return wait_until(lock,
10                   std::move(stoken),
11                   chrono::steady_clock::now() + rel_time,
12                   std::move(pred))
```

```

13     );
14
15 // wait_until in lines 12 - 16
16 while (!stoken.stop_requested()) {
17     if (pred()) return true;
18     if (wait_until(lock, timeout_time) == std::cv_status::timeout) return pred();
19 }
20 return pred();

```

等待调用之后，可以检查是否有停止请求。

```

1 cv.wait(lock, stoken, predicate);
2 if (stoken.stop_requested()){
3     // interrupt occurred
4 }

```

下面的示例展示了条件变量与停止请求的使用方式。

```

1 // conditionVariableAny.cpp
2
3 #include <condition_variable>
4 #include <thread>
5 #include <iostream>
6 #include <chrono>
7 #include <mutex>
8 #include <thread>
9
10 using namespace std::literals;
11
12 std::mutex mut;
13 std::condition_variable_any condVar;
14
15 bool dataReady;
16
17 void receiver(std::stop_token stopToken) {
18
19     std::cout << "Waiting" << '\n';
20
21     std::unique_lock<std::mutex> lck(mut);
22     bool ret = condVar.wait(lck, stopToken, []{return dataReady;});
23     if (ret) {
24         std::cout << "Notification received: " << '\n';
25     }
26     else{
27         std::cout << "Stop request received" << '\n';
28     }
29 }
30
31 void sender() {
32
33     std::this_thread::sleep_for(5ms);

```

```

34     {
35         std::lock_guard<std::mutex> lck(mut);
36         dataReady = true;
37         std::cout << "Send notification" << '\n';
38     }
39     condVar.notify_one();
40 }
41 }
42
43 int main() {
44
45     std::cout << '\n';
46
47     std::jthread t1(receiver);
48     std::jthread t2(sender);
49
50     t1.request_stop();
51
52     t1.join();
53     t2.join();
54
55     std::cout << '\n';
56 }
57 }
```

接收线程(第 17-29 行)正在等待发送线程(第 31-41 行)的通知。发送方线程在第 39 行发送通知之前，主线程在第 50 行触发了一个停止请求。程序的输出显示，停止请求发生在通知之前。

```

Waiting
Stop request received
Send notification
```

总结

- 可以使用 `std::stop_token`、`std::stop_source` 和 `std::stop_callback`，对线程和条件变量进行协作中断。协作中断意味着线程获得一个可以接受或忽略的停止请求。
- `std::stop_token` 可以传递给一个操作，用来主动轮询令牌以获得停止请求，或者使用 `std::stop_callback` 注册一个回调。
- 除了 `std::jthread` 之外，`std::condition_variable_any` 也可以接受停止请求。

6.6. std::jthread



Cippi 在扎辫子

std::jthread 代表连接线程。除了 C++11 中的 std::thread 之外，std::jthread 会在析构函数中自动汇入，并且可以协作中断。

下表简要概述了 std::jthread t 功能。更多详情请参考 cppreference.com。

函数	std::jthread t 的函数描述
t.join()	等待线程 t 完成执行。
t.detach()	独立执行创建的线程 t。
t.joinable()	若线程 t 仍然是可汇入的，则返回 true。
t.get_id() and std::this_thread::get_id()	返回线程的 id。
std::jthread::hardware_concurrency()	表示可以并发运行的线程数。
std::this_thread::sleep_until(absTime)	线程 t 进入睡眠状态，直到时间点 absTime。
std::this_thread::sleep_for(relTime)	线程 t 进入睡眠状态，时间为 relTime。
std::this_thread::yield()	允许系统运行另一个线程。
t.swap(t2) and std::swap(t1, t2)	交换线程。
t.get_stop_source()	返回与共享停止状态关联的 std::stop_source 对象。
t.get_stop_token()	返回与共享停止状态关联的 std::stop_token 对象。
t.request_stop()	通过共享停止状态停止执行请求。

6.6.1 自动汇入

这是 std::thread 的一种非直观行为。若 std::thread 仍然可汇入，std::terminate 将在其析构函数中调用。若既没有调用 thr.join()，也没有调用 thr.detach()，那么线程 thr 是可汇入的。

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

```

4 #include <thread>
5
6 int main() {
7     std::cout << '\n';
8     std::cout << std::boolalpha;
9
10    std::thread thr{}{ std::cout << "Joinable std::thread" << '\n'; };
11
12    std::cout << "thr.joinable(): " << thr.joinable() << '\n';
13
14    std::cout << '\n';
15
16 }

```

执行时，程序终止。

```

File Edit View Bookmarks Settings Help
rainer@linux:~/threadJoinable
thr.joinable(): true
terminate called without an active exception
Aborted (core dumped)
rainer@linux:~/threadJoinable
thr.joinable(): true
terminate called without an active exception
Joinable std::thread
Aborted (core dumped)
rainer@linux:~> █
rainer:bash

```

std::thread 的两次执行都终止。在第二次运行中，线程 thr 有足够的时间显示它的消息：“Joinable std::thread”。

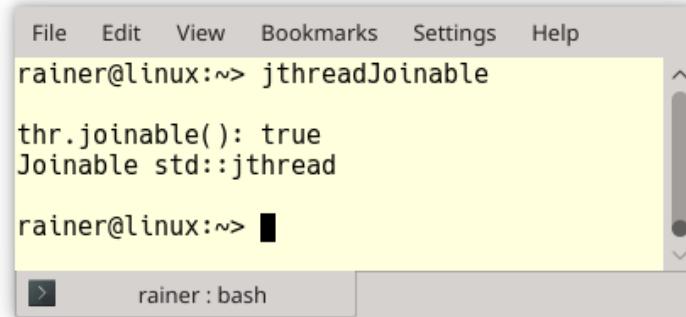
下一个例子中，使用来自 C++20 标准的 std::jthread。

```

1 // jthreadJoinable.cpp
2
3 #include <iostream>
4 #include <thread>
5
6 int main() {
7     std::cout << '\n';
8     std::cout << std::boolalpha;
9
10    std::jthread thr{}{ std::cout << "Joinable std::thread" << '\n'; };
11
12    std::cout << "thr.joinable(): " << thr.joinable() << '\n';
13
14    std::cout << '\n';
15 }

```

现在，若线程 thr 是可汇入的，则会在析构函数中使用自动汇入。



A screenshot of a terminal window titled "rainer@linux:~>". The window contains the command "jthreadJoinable" followed by its output: "thr.joinable(): true" and "Joinable std::jthread". The terminal has a standard menu bar at the top and a scroll bar on the right.

```
rainer@linux:~> jthreadJoinable
thr.joinable(): true
Joinable std::jthread
rainer@linux:~>
```

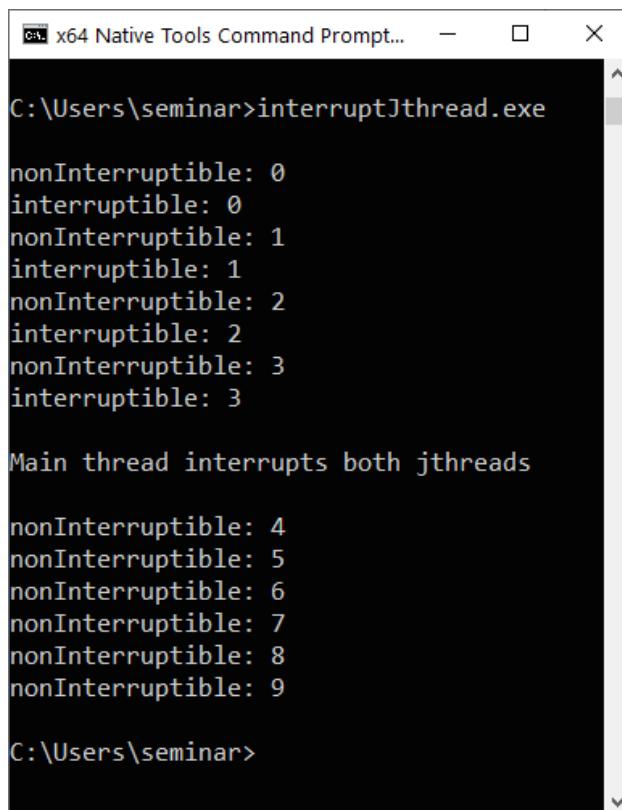
6.6.2 std::jthread 的协作中断

举一个简单的例子来了解大体的情况。

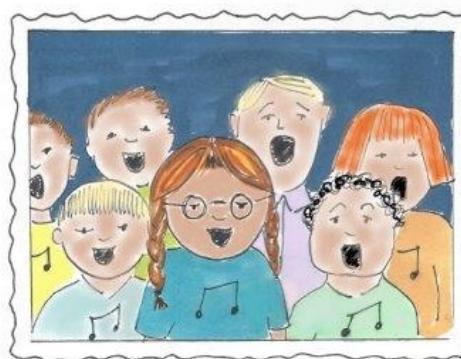
```
1 // interruptJthread.cpp
2
3 #include <chrono>
4 #include <iostream>
5 #include <thread>
6
7 using namespace::std::literals;
8
9 int main() {
10
11     std::cout << '\n';
12
13     std::jthread nonInterruptible([]{
14         int counter{0};
15         while (counter < 10){
16             std::this_thread::sleep_for(0.2s);
17             std::cerr << "nonInterruptible: " << counter << '\n';
18             ++counter;
19         }
20     });
21
22     std::jthread interruptible([](std::stop_token stoken) {
23         int counter{0};
24         while (counter < 10){
25             std::this_thread::sleep_for(0.2s);
26             if (stoken.stop_requested()) return;
27             std::cerr << "interruptible: " << counter << '\n';
28             ++counter;
29         }
30     });
31
32     std::this_thread::sleep_for(1s);
33 }
```

```
34     std::cerr << '\n';
35     std::cerr << "Main thread interrupts both jthreads" << '\n';
36     nonInterruptible.request_stop();
37     interruptible.request_stop();
38
39     std::cout << '\n';
40
41 }
```

主程序中，我启动了 `nonInterruptible` 和 `interruptible` 两个线程（第 13 行和第 22 行）。与不可中断线程不同，可中断线程获得 `std::stop_token`，并在第 26 行中使用它来检查是否中断`stop_requested()`。停止请求的情况下，Lambda 函数返回，因此线程结束，调用 `interruptible.request_stop()`（第 37 行）停止请求。但这并不适用于之前的调用 `nonInterruptible.request_stop()`，所以调用无效。



6.7. 同步输出流



Cippi 在唱诗班唱歌

编译器对同步输出流的支持

到 2020 年底，只有 GCC 11 支持同步输出流。

若不同步地写入 std::cout 会发生什么？

```
1 // coutUnsynchronized.cpp
2
3 #include <chrono>
4 #include <iostream>
5 #include <thread>
6
7 class Worker{
8 public:
9     Worker(std::string n):name(n) {};
10    void operator() () {
11        for (int i = 1; i <= 3; ++i) {
12            // begin work
13            std::this_thread::sleep_for(std::chrono::milliseconds(200));
14            // end work
15            std::cout << name << ":" << "Work " << i << " done !!!" << '\n';
16        }
17    }
18 private:
19     std::string name;
20 };
21
22
23 int main() {
24
25     std::cout << '\n';
26
27     std::cout << "Boss: Let's start working.\n\n";
28
29     std::thread herb= std::thread(Worker("Herb"));
30     std::thread andrei= std::thread(Worker(" Andrei"));
```

```

31 std::thread scott= std::thread(Worker(" Scott"));
32 std::thread bjarne= std::thread(Worker(" Bjarne"));
33 std::thread bart= std::thread(Worker(" Bart"));
34 std::thread jenne= std::thread(Worker(" Jenne"));

35

36 herb.join();
37 andrei.join();
38 scott.join();
39 bjarne.join();
40 bart.join();
41 jenne.join();

42 std::cout << "\n" << "Boss: Let's go home." << '\n';

43 std::cout << '\n';
44 }

45
46
47
48 }

```

老板有六个工人 (第 29-34 行)。每个工作人员必须处理三个工作包，每个工作包花费 1/5 秒 (第 13 行)。工人完成了他的工作包后，他大声地向老板报告 (第 15 行)。当老板收到所有员工的通知，老板就会把工人们送回家 (第 44 行)。

这么简单的工作流程真是一团糟! 每个员工都大声喊出自己的信息，无视其他同事!

```

rainer@seminar:~> coutUnsynchronized
Boss: Let's start working.

Andrei: Work 1 done !!!
Bart: Work 1      Bjarne: Work 1 done !!!
Herb: Work 1 done !!!
done !!!
Scott: Work 1 done !!!
Jenne: Work 1 done !!!
Scott: Work 2 done !!!
Andrei: Work 2 done !!!
Bjarne: Work          Jenne: Work 2 done !!!
Herb: Work done !!!
2 done !!!
Bart: Work 2 done !!!
Scott: Work 3 done !!!
Andrei: Work 3 done !!!
Jenne: Work 3 done !!!
Bjarne: Work 3 done !!!
Herb: Work 3 done !!!
Bart: Work 3 done !!!
Boss: Let's go home.

rainer@seminar:~>

```

std::cout 是线程安全的

C++11 标准确定不需要对 std::cout 进行额外的保护，每个字符都是原子地“输出”。类似于示例中的输出语句可能交织在一起的情况，只是一个视觉问题；而程序定义良好，没有问题。此注释对所有全局流对象有效。插入和提取全局流对象 (std::cout, std::cin, std::cerr 和 std::clog) 是线程安全的。更准确地说：写入 std::cout 并没有参与数据竞争，而是创建了条件竞争，所以由于线程的交错，从而导致了屏幕输出的错乱。

如何解决这个问题？C++11 中，答案很简单：可以使用 `lock_guard` 这样的锁来同步对 std::cout 的访问。

```
1 // coutSynchronized.cpp
2
3 #include <chrono>
4 #include <iostream>
5 #include <mutex>
6 #include <thread>
7
8 std::mutex coutMutex;
9
10 class Worker{
11 public:
12     Worker(std::string n):name(n) {};
13
14     void operator() () {
15         for (int i = 1; i <= 3; ++i) {
16             // begin work
17             std::this_thread::sleep_for(std::chrono::milliseconds(200));
18             // end work
19             std::lock_guard<std::mutex> coutLock(coutMutex);
20             std::cout << name << ":" << "Work " << i << " done !!!\n";
21         }
22     }
23 private:
24     std::string name;
25 };
26
27
28 int main() {
29
30     std::cout << '\n';
31
32     std::cout << "Boss: Let's start working." << "\n\n";
33
34     std::thread herb= std::thread(Worker("Herb"));
35     std::thread andrei= std::thread(Worker(" Andrei"));
36     std::thread scott= std::thread(Worker(" Scott"));
37     std::thread bjarne= std::thread(Worker(" Bjarne"));
38     std::thread bart= std::thread(Worker(" Bart"));
```

```

39 std::thread jenne= std::thread(Worker(" Jenne"));
40
41 herb.join();
42 andrei.join();
43 scott.join();
44 bjarne.join();
45 bart.join();
46 jenne.join();
47
48 std::cout << "\n" << "Boss: Let's go home." << '\n';
49
50 std::cout << '\n';
51
52 }

```

第 8 行中的 coutMutex 保护共享对象 std::cout。将 coutMutex 放入 std::lock_guard 中可以保证 coutMutex 在 std::lock_guard 的构造函数 (第 19 行) 中被锁定，在 std::lock_guard 的析构函数 (第 21 行) 中解锁。因为 coutLock 守护的 coutMutex，混乱的输出才会变得和谐。

```

rainer@seminar:~> coutSynchronized

Boss: Let's start working.

    Scott: Work 1 done !!!
    Bjarne: Work 1 done !!!
    Andrei: Work 1 done !!!
    Herb: Work 1 done !!!
        Jenne: Work 1 done !!!
        Bart: Work 1 done !!!
    Scott: Work 2 done !!!
    Andrei: Work 2 done !!!
    Bjarne: Work 2 done !!!
    Herb: Work 2 done !!!
        Bart: Work 2 done !!!
        Jenne: Work 2 done !!!
    Andrei: Work 3 done !!!
    Scott: Work 3 done !!!
    Bjarne: Work 3 done !!!
    Herb: Work 3 done !!!
        Bart: Work 3 done !!!
        Jenne: Work 3 done !!!

Boss: Let's go home.

rainer@seminar:~>

```

C++20 编写同步的 std::cout 更是小菜一碟。std::basic_syncbuf 是 std::basic_streambuf 的包装器，可以在缓冲区中累积输出。包装器在销毁时将其内容设置为包装的缓冲区，所以内容会显示为连续

的字符序列，不会发生字符的交错。

通过 std::basic_ostream，可以直接同步写入 std::cout。

可以创建命名同步输出流。并且，可以对前面的 coutUnsynchronized.cpp 进行重构，将 synchronized 写入到 std::cout 中。

```
1 // synchronizedOutput.cpp
2
3 #include <chrono>
4 #include <iostream>
5 #include <syncstream>
6 #include <thread>
7
8 class Worker{
9 public:
10    Worker(std::string n): name(n) {};
11    void operator() () {
12        for (int i = 1; i <= 3; ++i) {
13            // begin work
14            std::this_thread::sleep_for(std::chrono::milliseconds(200));
15            // end work
16            std::osyncstream syncStream(std::cout);
17            syncStream << name << ":" << "Work " << i << " done !!!" << '\n';
18        }
19    }
20 private:
21    std::string name;
22 };
23
24
25 int main() {
26
27     std::cout << '\n';
28
29     std::cout << "Boss: Let's start working.\n\n";
30
31     std::thread herb= std::thread(Worker("Herb"));
32     std::thread andrei= std::thread(Worker(" Andrei"));
33     std::thread scott= std::thread(Worker(" Scott"));
34     std::thread bjarne= std::thread(Worker(" Bjarne"));
35     std::thread bart= std::thread(Worker(" Bart"));
36     std::thread jenne= std::thread(Worker(" Jenne"));
37
38
39     herb.join();
40     andrei.join();
41     scott.join();
42     bjarne.join();
43     bart.join();
44     jenne.join();
```

```
45     std::cout << "\n" << "Boss: Let's go home." << '\n';
46
47     std::cout << '\n';
48 }
49 }
```

对 coutUnsynchronized.cpp 的唯一更改，是将 std::cout 包装在 std::osyncstream 中（第 16 行）。当 std::osyncstream 在第 18 行中超出范围时，将传输字符并刷新 std::cout。值得一提的是，主程序中的 std::cout 调用不会引入数据竞争，因此不需要同步。

因为只在第 17 行声明了一次 syncStream，所以使用临时对象可能更合适。下面的代码片段展示了修改后的调用运算符。

```
1 void operator() () {
2     for (int i = 1; i <= 3; ++i) {
3         // begin work
4         std::this_thread::sleep_for(std::chrono::milliseconds(200));
5         // end work
6         std::osyncstream(std::cout) << name << ":" << "Work " << i << " done !!!"
7                         << '\n';
8     }
9 }
```

std::basic_osyncstream syncStream 提供了两个有趣的成员函数。

1. syncStream.emit() 发出所有缓冲输出并对所有挂起进行刷新。
2. syncStream.get_wrapped() 返回一个指向包装缓冲区的指针。

cppreference.com 展示了如何使用 get_wrapped 成员函数对不同输出流的输出进行排序。

```
1 // sequenceOutput.cpp
2
3 #include <syncstream>
4 #include <iostream>
5
6 int main() {
7
8     std::osyncstream bout1(std::cout);
9     bout1 << "Hello, ";
10    {
11        std::osyncstream(bout1.get_wrapped()) << "Goodbye, " << "Planet!" << '\n';
12    } // emits the contents of the temporary buffer
13
14    bout1 << "World!" << '\n';
15
16 } // emits the contents of bout1
```

Goodbye, Planet!

Hello, World!

总结

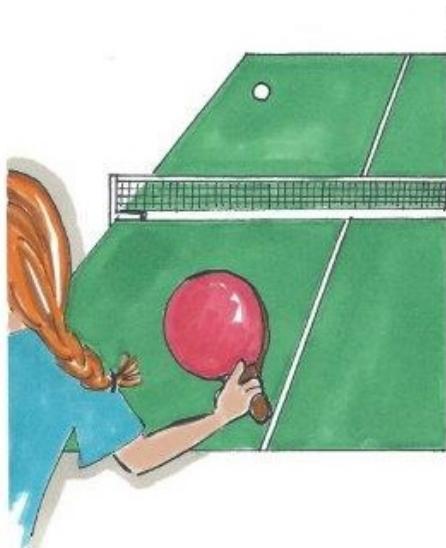
- `std::cout` 是线程安全的，但是当线程并发地写入 `std::cout` 时，可能会出现输出操作的交叉。这只是一个视觉问题，而不是数据竞争。
- C++20 支持同步输出流，其在内部缓冲区中积累输出，并在原子步骤中写入内容，所以不会发生输出操作交错的情况。

第7章 相关案例

在为 C++20 提供理论之后，我现在将理论应用于实践，并为您提供一些研究案例。

想要多次同步线程时，可以使用条件变量、`std::atomic_flag`、`std::atomic<bool>` 或信号量。在“线程的快速同步”部分，我想回答哪种方式是最快的？关于协程的部分，给出了三个基于 `co_return`、`co_yield` 和 `co_await` 的协程。我使用这些协程作为进一步实验的起点，以加深对协程控制流的理解。在“Future 的变体”章节中，我在使用 `co_return` 时，实现了惰性 `future` 和基于 `future` 的 `future`。线程的章节中，修改和泛化改进了使用 `co_return` 的生成器。最后，“不同的工作流”章节中从 `co_await` 开始，讨论了一些作业工作流。

7.1. 线程的快速同步



Cippi 在打乒乓球

具体设备的性能参考

对性能数据应该持保留态度，我对 Linux 和 Windows 上每种算法变体的性能数字不感兴趣。我更感兴趣的是一种直觉，哪些算法可能有效或无效。我并不是在比较我的 Linux 桌面和 Windows 笔记本电脑上的绝对数字，但我想知道某些算法在 Linux 或 Windows 上可以运行得更好。

当要多次同步线程时，可以使用条件变量、`std::atomic_flag`、`std::atomic<bool>` 或信号量。本节中，我要回答的问题是：哪种方式最快？

为了得到类似的数字，我实现了一个乒乓球游戏。一个线程执行 `ping` 函数（或简称 `ping` 线程），另一个线程执行 `pong` 函数（或简称 `pong` 线程）。`ping` 线程等待 `pong` 线程通知，并将通知发送回 `pong` 线程，100 万次“打击”后比赛结束。我将每款游戏执行 5 次，以获得可比较的性能数据。

关于数字

我在 2020 年底用全新的 Visual Studio 编译器 19.28 进行了性能测试，该编译器支持与原子 (std::atomic_flag 和 std::atomic) 和信号量的同步。此外，我用最大优化 (/Ox) 编译了这些示例。性能值只能大致说明同步线程的各种方法的相对性能。若想在平台上获得准确的数字，必须重复进行测试。

先与 C++11 进行比较。

7.1.1 条件变量

```
1 // pingPongConditionVariable.cpp
2
3 #include <condition_variable>
4 #include <iostream>
5 #include <atomic>
6 #include <thread>
7
8 bool dataReady{false};
9
10 std::mutex mutex_;
11 std::condition_variable condVar1;
12 std::condition_variable condVar2;
13
14 std::atomic<int> counter{};
15 constexpr int countlimit = 1'000'000;
16
17 void ping() {
18
19     while(counter <= countlimit) {
20
21         std::unique_lock<std::mutex> lck(mutex_);
22         condVar1.wait(lck, []{return dataReady == false;});
23         dataReady = true;
24     }
25     ++counter;
26     condVar2.notify_one();
27 }
28
29
30 void pong() {
31
32     while(counter < countlimit) {
33
34         std::unique_lock<std::mutex> lck(mutex_);
35         condVar2.wait(lck, []{return dataReady == true;});
36         dataReady = false;
37     }
38 }
```

```

38     condVar1.notify_one();
39 }
40
41 }
42
43 int main() {
44
45     auto start = std::chrono::system_clock::now();
46
47     std::thread t1(ping);
48     std::thread t2(pong);
49
50     t1.join();
51     t2.join();
52
53     std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
54     std::cout << "Duration: " << dur.count() << " seconds" << '\n';
55 }
```

程序中使用了两个条件变量:condVar1 和 condVar2。ping 线程等待 condVar1 的通知，并将其通知与 condVar2 一起发送，dataReady 可以防止伪和未唤醒。当计数器达到计数限时，乒乓球比赛结束。调用 notify_one(第 26 行和第 38 行) 和计数器是线程安全的，因此不在临界区内。

下面就是数据。

```

x64 Native Tools Command Prompt for VS 2...
C:\Users\rainer>pingPongConditionVariable.exe
Duration: 0.527351 seconds

C:\Users\rainer>pingPongConditionVariable.exe
Duration: 0.527036 seconds

C:\Users\rainer>pingPongConditionVariable.exe
Duration: 0.467337 seconds

C:\Users\rainer>pingPongConditionVariable.exe
Duration: 0.463415 seconds

C:\Users\rainer>pingPongConditionVariable.exe
Duration: 0.631053 seconds

C:\Users\rainer>
```

平均执行时间为 0.52 秒。

C++20 中，将这个工作流移植到 std::atomic_flag 很简单。

7.1.2 std::atomic_flag

下面是使用两个原子标志和一个原子标志的相同工作流。

7.1.2.1 两个原子标志

下面的程序中，我将条件变量的等待替换为对原子标志的等待，并将条件变量的通知替换为原子标志设置后的通知。

```
1 // pingPongAtomicFlags.cpp
2
3 #include <iostream>
4 #include <atomic>
5 #include <thread>
6
7 std::atomic_flag condAtomicFlag1{};
8 std::atomic_flag condAtomicFlag2{};
9
10 std::atomic<int> counter{};
11 constexpr int countlimit = 1'000'000;
12 void ping() {
13     while(counter <= countlimit) {
14         condAtomicFlag1.wait(false);
15         condAtomicFlag1.clear();
16
17         ++counter;
18
19         condAtomicFlag2.test_and_set();
20         condAtomicFlag2.notify_one();
21     }
22 }
23
24 void pong() {
25     while(counter < countlimit) {
26         condAtomicFlag2.wait(false);
27         condAtomicFlag2.clear();
28
29         condAtomicFlag1.test_and_set();
30         condAtomicFlag1.notify_one();
31     }
32 }
33
34 int main() {
35
36     auto start = std::chrono::system_clock::now();
37
38     condAtomicFlag1.test_and_set();
39     std::thread t1(ping);
40     std::thread t2(pong);
41
42     t1.join();
43     t2.join();
44
45     std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
46     std::cout << "Duration: " << dur.count() << " seconds" << '\n';
47 }
```

若原子标志的值为 false, condAtomicFlag1.wait(false)(第 15 行) 将阻塞; 若 condAtomicFlag1 的值为 true, 则返回。布尔值作为一种谓词, 因此设置为 false(第 15 行)。将通知(第 21 行)发送到 pong 线程之前, condAtomicFlag1 设置为 true(第 20 行)。condAtomicFlag1(第 39 行)的初始设置为 true, 开始游戏。

因为使用了 std::atomic_flag, 游戏结束得更快了。

```
C:\Users\rainer>pingPongAtomicFlags.exe
Duration: 0.326716 seconds

C:\Users\rainer>pingPongAtomicFlags.exe
Duration: 0.327883 seconds

C:\Users\rainer>pingPongAtomicFlags.exe
Duration: 0.317234 seconds

C:\Users\rainer>pingPongAtomicFlags.exe
Duration: 0.305536 seconds

C:\Users\rainer>pingPongAtomicFlags.exe
Duration: 0.343247 seconds

C:\Users\rainer>
```

一场游戏平均需要 0.32 秒。

当分析程序时, 可能会觉得一个原子标志对于这个工作流来说足够了。

7.1.2.2 一个原子标志

使用一个原子标志使工作流更容易理解。

```
// pingPongAtomicFlag.cpp

#include <iostream>
#include <atomic>
#include <thread>

std::atomic_flag condAtomicFlag{};

std::atomic<int> counter{};
constexpr int countlimit = 1'000'000;

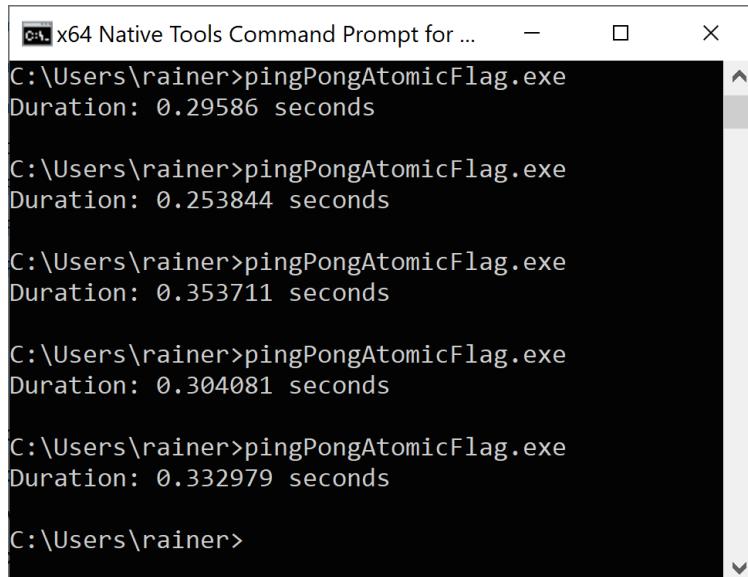
void ping() {
    while(counter <= countlimit) {
        condAtomicFlag.wait(true);
        condAtomicFlag.test_and_set();
        ++counter;
    }
}
```

```

19     condAtomicFlag.notify_one();
20 }
21 }
22
23 void pong() {
24     while(counter < countlimit) {
25         condAtomicFlag.wait(false);
26         condAtomicFlag.clear();
27         condAtomicFlag.notify_one();
28     }
29 }
30
31 int main() {
32
33     auto start = std::chrono::system_clock::now();
34
35     condAtomicFlag.test_and_set();
36     std::thread t1(ping);
37     std::thread t2(pong);
38
39     t1.join();
40     t2.join();
41
42     std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
43     std::cout << "Duration: " << dur.count() << " seconds" << '\n';
44
45 }

```

ping 线程阻塞在 true 上，而 pong 线程阻塞在 false 上。从性能的角度来看，使用一个或两个原子标志没有区别。



执行的平均时间为 0.31 秒。

这个例子中，我使用 `std::atomic_flag` 作为原子布尔值。让我们用 `std::atomic<bool>` 再试一次。

7.1.3 std::atomic<bool>

下面的 C++20 基于 std::atomic 的实现。

```
1 // pingPongAtomicBool.cpp
2
3 #include <iostream>
4 #include <atomic>
5 #include <thread>
6
7 std::atomic<bool> atomicBool{};
8
9 std::atomic<int> counter{};
10 constexpr int countlimit = 1'000'000;
11
12 void ping() {
13     while(counter <= countlimit) {
14         atomicBool.wait(true);
15         atomicBool.store(true);
16
17         ++counter;
18
19         atomicBool.notify_one();
20     }
21 }
22
23 void pong() {
24     while(counter < countlimit) {
25         atomicBool.wait(false);
26         atomicBool.store(false);
27         atomicBool.notify_one();
28     }
29 }
30
31 int main() {
32
33     std::cout << std::boolalpha << '\n';
34
35     std::cout << "atomicBool.is_lock_free(): "
36     << atomicBool.is_lock_free() << '\n';
37
38     std::cout << '\n';
39
40     auto start = std::chrono::system_clock::now();
41
42     atomicBool.store(true);
43     std::thread t1(ping);
44     std::thread t2(pong);
45
46     t1.join();
```

```
47     t2.join();
48
49     std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
50     std::cout << "Duration: " << dur.count() << " seconds" << '\n';
51
52 }
```

std::atomic<bool> 可以在内部使用锁，例如：互斥锁。Windows 在运行时是无锁的。

```
C:\x64 Native Tools Command Prompt for V...
C:\Users\rainer>pingPongAtomicBool.exe
atomicBool.is_lock_free(): true
Duration: 0.424524 seconds

C:\Users\rainer>pingPongAtomicBool.exe
atomicBool.is_lock_free(): true
Duration: 0.357399 seconds

C:\Users\rainer>pingPongAtomicBool.exe
atomicBool.is_lock_free(): true
Duration: 0.38501 seconds

C:\Users\rainer>pingPongAtomicBool.exe
atomicBool.is_lock_free(): true
Duration: 0.370447 seconds

C:\Users\rainer>pingPongAtomicBool.exe
atomicBool.is_lock_free(): true
Duration: 0.400319 seconds

C:\Users\rainer>
```

平均执行时间为 0.38 秒。

从可读性的角度来看，这个基于 std::atomic 的实现很容易理解。这个观察结果也适用于基于信号量实现的乒乓游戏。

7.1.4 信号量

信号量比条件变量更快，来看看这是不是真的。

```
1 // pingPongSemaphore.cpp
2
3 #include <iostream>
4 #include <semaphore>
5 #include <thread>
```

```

6
7 std::counting_semaphore<1> signal2Ping(0);
8 std::counting_semaphore<1> signal2Pong(0);
9
10 std::atomic<int> counter{};
11 constexpr int countlimit = 1'000'000;
12
13 void ping() {
14     while(counter <= countlimit) {
15         signal2Ping.acquire();
16         ++counter;
17         signal2Pong.release();
18     }
19 }
20
21 void pong() {
22     while(counter < countlimit) {
23         signal2Pong.acquire();
24         signal2Ping.release();
25     }
26 }
27
28 int main() {
29
30     auto start = std::chrono::system_clock::now();
31
32     signal2Ping.release();
33     std::thread t1(ping);
34     std::thread t2(pong);
35
36     t1.join();
37     t2.join();
38
39     std::chrono::duration<double> dur = std::chrono::system_clock::now() - start;
40     std::cout << "Duration: " << dur.count() << " seconds" << '\n';
41
42 }

```

pingpongsemaphore.cpp 使用了两个信号量:signal2Ping 和 signal2Pong(第 7 行和第 8 行)。这两个信号量都可以有 0 或 1，并都初始化为 0。当信号量 signal2Ping 的值为 0 时，signal2Ping.release()(第 24 和 32 行) 将该值设置为 1，因此这是一个通知。signal2Ping.acquire()(第 15 行) 进行阻塞，直到值变为 1。同样的方式，也适用于第二个信号量 signal2Pong。

```
C:\Users\rainer>pingPongSemaphore.exe
Duration: 0.367456 seconds

C:\Users\rainer>pingPongSemaphore.exe
Duration: 0.359944 seconds

C:\Users\rainer>pingPongSemaphore.exe
Duration: 0.339582 seconds

C:\Users\rainer>pingPongSemaphore.exe
Duration: 0.308024 seconds

C:\Users\rainer>pingPongSemaphore.exe
Duration: 0.319354 seconds

C:\Users\rainer>
```

平均执行时间为 0.33 秒。

7.1.5 所有的数据

如预期的那样，条件变量是最慢的方式，原子标记是同步线程的最快方式。`std::atomic<bool>`的性能介于两者之间。`std::atomic<bool>`有一个缺点，是唯一无锁的原子数据类型。信号量给我的印象最深，因为它几乎与原子标志一样快。

	条件 变量	两个原子 标志	一个原子 标志	原子 布尔	信号量
执行 时间 (秒)		0.52	0.32	0.31	0.38

7.2. Future 的变体



Cippi 开启工作流

在使用 `co_return` 的章节中创建 `future` 之前，我们应该理解其控制流。注释使控制流透明。此外，我还提供了在线编译器上提供的程序的链接。

```
// eagerFutureWithComments.cpp

#include <coroutine>
#include <iostream>
#include <memory>

template<typename T>
struct MyFuture {
    std::shared_ptr<T> value;
    MyFuture(std::shared_ptr<T> p): value(p) {
        std::cout << " MyFuture::MyFuture" << '\n';
    }
    ~MyFuture() {
        std::cout << " MyFuture::~MyFuture" << '\n';
    }
    T get() {
        std::cout << " MyFuture::get" << '\n';
        return *value;
    }
    struct promise_type {
        std::shared_ptr<T> ptr = std::make_shared<T>();
        promise_type() {
            std::cout << " promise_type::promise_type" << '\n';
        }
        ~promise_type() {
            std::cout << " promise_type::~promise_type" << '\n';
        }
        MyFuture<T> get_return_object() {
            std::cout << " promise_type::get_return_object" << '\n';
            return ptr;
        }
        void return_value(T v) {
            std::cout << " promise_type::return_value" << '\n';
            *ptr = v;
        }
    };
};
```

```

36     }
37     std::suspend_never initial_suspend() {
38         std::cout << " promise_type::initial_suspend" << '\n';
39         return {};
40     }
41     std::suspend_never final_suspend() noexcept {
42         std::cout << " promise_type::final_suspend" << '\n';
43         return {};
44     }
45     void unhandled_exception() {
46         std::exit(1);
47     }
48 };
49 }
50
51 MyFuture<int> createFuture() {
52     std::cout << "createFuture" << '\n';
53     co_return 2021;
54 }
55
56 int main() {
57
58     std::cout << '\n';
59
60     auto fut = createFuture();
61     auto res = fut.get();
62     std::cout << "res: " << res << '\n';
63
64     std::cout << '\n';
65 }

```

调用 `createFuture`(第 60 行) 创建 `MyFuture` 实例。`MyFuture` 的构造函数调用(第 10 行)完成之前, `promise` `promise_type` 就会创建、执行和销毁(第 21-49 行)。`promise` 在其控制流的每一步中都使用可等待的 `std::suspend_never`(第 37 和 41 行), 因此从不暂停。为了为后面的 `fut.get()` 保存 `promise` 的结果(第 60 行), 所以必须对它进行内存分配。此外, 使用的 `std::shared_ptr` 的(第 9 行和第 22 行)程序不会出现内存泄漏。作为局部变量, `fut` 在第 65 行超出了范围, C++ 运行时届时会调用其析构函数。

您可以在[Compiler Explorer](#)上尝试编译运行该程序。

```

promise_type::promise_type
promise_type::get_return_object
promise_type::initial_suspend
createFuture
    promise_type::return_value
    promise_type::final_suspend
    promise_type::~promise_type
MyFuture::MyFuture
MyFuture::get
res: 2021

MyFuture::~MyFuture

```

所呈现的协程立即运行。此外，协程会在调用者的线程中运行。

现在，来让协程变懒惰。

7.2.1 惰性 future

惰性 future 是指仅在请求值时才运行的 future，来看看需要对 eagerFutureWithComments.cpp 中的立即协程中进行哪些修改，才能使其变得懒惰。

```
1 // lazyFuture.cpp
2
3 #include <coroutine>
4 #include <iostream>
5 #include <memory>
6
7 template<typename T>
8 struct MyFuture {
9     struct promise_type;
10    using handle_type = std::coroutine_handle<promise_type>;
11
12    handle_type coro;
13
14    MyFuture(handle_type h) : coro(h) {
15        std::cout << " MyFuture::MyFuture" << '\n';
16    }
17
18    ~MyFuture() {
19        std::cout << " MyFuture::~MyFuture" << '\n';
20        if ( coro ) coro.destroy();
21    }
22
23    T get() {
24        std::cout << " MyFuture::get" << '\n';
25        coro.resume();
26        return coro.promise().result;
27    }
28
29    struct promise_type {
30        T result;
31        promise_type() {
32            std::cout << " promise_type::promise_type" << '\n';
33        }
34        ~promise_type() {
35            std::cout << " promise_type::~promise_type" << '\n';
36        }
37        auto get_return_object() {
38            std::cout << " promise_type::get_return_object" << '\n';
39            return MyFuture{handle_type::from_promise(*this)};
40        }
41        void return_value(T v) {
42            std::cout << " promise_type::return_value" << '\n';
43            result = v;
44        }
45    };
46}
```

```

43     }
44     std::suspend_always initial_suspend() {
45         std::cout << " promise_type::initial_suspend" << '\n';
46         return {};
47     }
48     std::suspend_always final_suspend() noexcept {
49         std::cout << " promise_type::final_suspend" << '\n';
50         return {};
51     }
52     void unhandled_exception() {
53         std::exit(1);
54     }
55 };
56 };
57
58 MyFuture<int> createFuture() {
59     std::cout << "createFuture" << '\n';
60     co_return 2021;
61 }
62
63 int main() {
64
65     std::cout << '\n';
66
67     auto fut = createFuture();
68     auto res = fut.get();
69     std::cout << "res: " << res << '\n';
70
71     std::cout << '\n';
72 }
73 }
```

我们先来研究一下 promise。promise 总是在开头(第 44 行)和结尾(第 48 行)处挂起。此外，成员函数 get_return_object(第 36 行)会创建返回对象，返回给协程 createFuture 的调用者(第 58 行)。future MyFuture 更有趣，它有一个用来处理 promise 的句柄 coro(第 12 行)。MyFuture 使用这个句柄来管理 promise。可以恢复 promise(第 24 行)执行，向 promise 请求结果(第 25 行)，最后销毁它(第 19 行)。恢复协程是必要的，因为它不会自动运行(第 44 行)。当调用端使用 fut.get()(第 68 行)来请求 future 的结果时，就会隐式地恢复 promise(第 24 行)的运行。

可以在[Compiler Explorer](#)上尝试编译运行该程序。

```
promise_type::promise_type
promise_type::get_return_object
MyFuture::MyFuture
promise_type::initial_suspend
MyFuture::get
createFuture
promise_type::return_value
promise_type::final_suspend
res: 2021

MyFuture::~MyFuture
promise_type::~promise_type
```

若调用端对 future 的结果不感兴趣怎么办? 让我们模拟一下。

```
1 int main() {
2
3     std::cout << '\n';
4
5     auto fut = createFuture();
6     // auto res = fut.get();
7     // std::cout << "res: " << res << '\n';
8
9     std::cout << '\n';
10 }
```

正如读者们可能猜到的那样, promise 永远不会运行, 成员函数 return_value 和 final_suspend 也不会执行。

```
promise_type::promise_type
promise_type::get_return_object
MyFuture::MyFuture
promise_type::initial_suspend

MyFuture::~MyFuture
promise_type::~promise_type
```

关于数据

处理协程的挑战之一是处理协程的生命周期。在前面的 `eagerFutureWithComments.cpp` 中, 我将协程结果存储在 `std::shared_ptr` 中。这很重要, 因为协程会立即执行。

`lazyFuture.cpp` 中, 使用 `final_suspend` 总是挂起(第 48 行):`std::suspend_always final_suspend()`。因此, `promise` 比调用端的生命周期更长, 不再需要 `std::shared_ptr`。所以, 从函数 `final_suspend` 返回 `std::suspend_never` 将导致未定义行为。因此, 结果的生命周期在调用端请求它之前就结束了。

让我们进一步修改协程, 并在单独的线程中运行 `promise`。

7.2.2 在另一个线程上执行协程

进入协程 `createFuture` 之前，因为成员函数 `initial_suspend` 返回 `std::suspend_always`(第 51 行)，所以协程处于挂起(第 66 行)状态。因此，`promise` 可以在另一个线程上执行。

```
1 // lazyFutureOnOtherThread.cpp
2
3 #include <coroutine>
4 #include <iostream>
5 #include <memory>
6 #include <thread>
7
8 template<typename T>
9 struct MyFuture {
10     struct promise_type;
11     using handle_type = std::coroutine_handle<promise_type>;
12     handle_type coro;
13
14     MyFuture(handle_type h) : coro(h) {}
15     ~MyFuture() {
16         if (coro) coro.destroy();
17     }
18
19     T get() {
20         std::cout << " MyFuture::get: "
21             << "std::this_thread::get_id(): "
22             << std::this_thread::get_id() << '\n';
23         std::thread t([this] { coro.resume(); });
24         t.join();
25         return coro.promise().result();
26     }
27
28     struct promise_type {
29         promise_type() {
30             std::cout << " promise_type::promise_type: "
31                 << "std::this_thread::get_id(): "
32                 << std::this_thread::get_id() << '\n';
33         }
34         ~promise_type() {
35             std::cout << " promise_type::~promise_type: "
36                 << "std::this_thread::get_id(): "
37                 << std::this_thread::get_id() << '\n';
38         }
39
40         T result;
41         auto get_return_object() {
42             return MyFuture(handle_type::from_promise(*this));
43         }
44         void return_value(T v) {
45             std::cout << " promise_type::return_value: "
```

```

46         << "std::this_thread::get_id(): "
47         << std::this_thread::get_id() << '\n';
48     std::cout << v << std::endl;
49     result = v;
50 }
51 std::suspend_always initial_suspend() {
52     return {};
53 }
54 std::suspend_always final_suspend() noexcept {
55     std::cout << " promise_type::final_suspend: "
56         << "std::this_thread::get_id(): "
57         << std::this_thread::get_id() << '\n';
58     return {};
59 }
60 void unhandled_exception() {
61     std::exit(1);
62 }
63 };
64 };
65
66 MyFuture<int> createFuture() {
67     co_return 2021;
68 }
69
70 int main() {
71
72     std::cout << '\n';
73
74     std::cout << "main: "
75         << "std::this_thread::get_id(): "
76         << std::this_thread::get_id() << '\n';
77
78     auto fut = createFuture();
79     auto res = fut.get();
80     std::cout << "res: " << res << '\n';
81
82     std::cout << '\n';
83 }
84 }
```

我在代码中添加了一些注释，显示正在运行的线程的 id。lazyFutureOnOtherThread.cpp 与前面的程序 lazyFuture.cpp 非常相似。主要区别在于成员函数 get(第 19 行)。使用 std::thread t([this]{coro.resume();});(第 23 行)，将在另一个线程上恢复协程的执行。

你可以在[Wandbox](#)在线编译器上尝试这个程序。

```
main: std::this_thread::get_id(): 139819561723776
promise_type::promise_type: std::this_thread::get_id(): 139819561723776
MyFuture::get: std::this_thread::get_id(): 139819561723776
promise_type::return_value: std::this_thread::get_id(): 139819456755456
promise_type::final_suspend: std::this_thread::get_id(): 139819456755456
res: 2021

promise_type::~promise_type: std::this_thread::get_id(): 139819561723776
```

我想补充一些关于成员函数 `get` 的内容。在另一个线程中恢复的 `promise`, 必须在返回 `coro.promise().result` 之前完成, 这一点至关重要。

成员函数获取 `std::thread`

```
1 T get() {
2     std::thread t([this] { coro.resume(); });
3     t.join();
4     return coro.promise().result;
5 }
```

函数在调用 `return coro.promise()` 后将线程 `t` 汇入, 程序将具有未定义行为。下面函数 `get` 的实现中, 我使用了 `std::jthread`。`std::jthread` 在超出作用域时可以自动汇入, 但这太迟了。

成员函数获取 `std::jthread`

```
1 T get() {
2     std::jthread t([this] { coro.resume(); });
3     return coro.promise().result;
4 }
```

这种情况下, 调用端很可能在 `promise` 使用成员函数 `return_value` 准备 `promise` 之前, 就得到了结果。`result` 有一个任意值, `res` 也一样。

```
main: std::this_thread::get_id(): 139913381070720
promise_type::promise_type: std::this_thread::get_id(): 139913381070720
MyFuture::get: std::this_thread::get_id(): 139913381070720
promise_type::return_value: std::this_thread::get_id(): 139913276102400
promise_type::final_suspend: std::this_thread::get_id(): 139913276102400
res: -1

promise_type::~promise_type: std::this_thread::get_id(): 139913381070720
```

还有其他方法可以确保线程在返回调用之前完成。

- 在其作用域中创建 `std::jthread`。

```
1 T get() {
2     {
3         std::jthread t([this] { coro.resume(); });
4     }
5     return coro.promise().result;
6 }
```

- 将 std::jthread 设置为临时对象

```
1 T get() {
2     std::jthread([this] { coro.resume(); });
3     return coro.promise().result;
4 }
```

特别说明，我不喜欢最后一个解决方案，因为它可能需要几秒钟，才能意识到刚刚调用了 std::jthread 的构造函数。

7.3. 生成器的修改和泛化



Cippi 在处理数据流

在为无限数据流修改和泛化生成器之前，我想把它作为我们旅程的起点。我有意在源代码中放置了许多输出操作，并且只要求三个值。这种简化和可视化应该有助于理解控制流。

```
1 // infiniteDataStreamComments.cpp
2
3 #include <coroutine>
4 #include <memory>
5 #include <iostream>
6
7 template<typename T>
8 struct Generator {
9
10     struct promise_type;
11     using handle_type = std::coroutine_handle<promise_type>;
12
13     Generator(handle_type h) : coro(h) {
14         std::cout << " Generator::Generator" << '\n';
15     }
16     handle_type coro;
17
18     ~Generator() {
19         std::cout << " Generator::~Generator" << '\n';
20         if ( coro ) coro.destroy();
21     }
22     Generator(const Generator&) = delete;
23     Generator& operator = (const Generator&) = delete;
24     Generator(Generator&& oth) : coro(oth.coro) {
25         oth.coro = nullptr;
26     }
27     Generator& operator = (Generator&& oth) {
28         coro = oth.coro;
29         oth.coro = nullptr;
```

```

30     return *this;
31 }
32 int getNextValue() {
33     std::cout << " Generator::getNextValue" << '\n';
34     coro.resume();
35     return coro.promise().current_value;
36 }
37 struct promise_type {
38     promise_type() {
39         std::cout << " promise_type::promise_type" << '\n';
40     }
41
42     ~promise_type() {
43         std::cout << " promise_type::~promise_type" << '\n';
44     }
45
46     std::suspend_always initial_suspend() {
47         std::cout << " promise_type::initial_suspend" << '\n'; \
48
49         return {};
50     }
51     std::suspend_always final_suspend() noexcept {
52         std::cout << " promise_type::final_suspend" << '\n';
53         return {};
54     }
55     auto get_return_object() {
56         std::cout << " promise_type::get_return_object" << '\n'; \
57
58         return Generator<handle_type>::from_promise(*this);
59     }
60
61     std::suspend_always yield_value(int value) {
62         std::cout << " promise_type::yield_value" << '\n'; \
63
64         current_value = value;
65         return {};
66     }
67     void return_void() {}
68     void unhandled_exception() {
69         std::exit(1);
70     }
71
72     T current_value;
73 };
74
75 };
76
77 Generator<int> getNext(int start = 10, int step = 10) {
78     std::cout << " getNext: start" << '\n';

```

```

79    auto value = start;
80    while (true) {
81        std::cout << " getNext: before co_yield" << '\n';
82        co_yield value;
83        std::cout << " getNext: after co_yield" << '\n';
84        value += step;
85    }
86 }
87
88 int main() {
89     auto gen = getNext();
90     for (int i = 0; i <= 2; ++i) {
91         auto val = gen.getNextValue();
92         std::cout << "main: " << val << '\n';
93     }
94 }
95 }
```

可以在[Compiler Explorer](#)上执行程序，使控制流更加明显。

```

promise_type::promise_type
promise_type::get_return_object
Generator::Generator
promise_type::initial_suspend
Generator::getNextValue
getNext: start
getNext: before co_yield
promise_type::yield_value
main: 10
    Generator::getNextValue
getNext: after co_yield
getNext: before co_yield
promise_type::yield_value
main: 20
    Generator::getNextValue
getNext: after co_yield
getNext: before co_yield
promise_type::yield_value
main: 30
    Generator::~Generator
promise_type::~promise_type
```

来分析一下控制流。

调用 `getNext()`(第 89 行) 创建 `Generator<int>`，同时创建 `promise_type`(第 38 行)，然后下面的 `get_return_object`(第 55 行) 创建生成器(第 58 行)，并将其存储在一个局部变量中。当协程第一次挂起时，此调用的结果返回给调用方，初始暂停立即发生(第 46 行)。因为成员函数调用 `initial_suspend` 返回可等待 `std::suspend_always`(第 46 行)，控制流继续使用协程 `getNext`，直到指令 `co_yield value`(第 82 行)。该调用会映射到 `yield_value(int value)`(第 61 行)，当前值为 `current_value = value`(第 64 行)，成员函数 `yield_value(int value)` 返回可等待的 `std::suspend_always`(第 58 行)。因此，协程的执行暂停，控制流返回到主函数，`for` 循环开始(第 90 行)。`gen.getNextValue()`(第 91 行) 通过使用 `coro.resume()`(第 34 行) 恢复协程来开始执行协程。此外，函数 `getNextValue()` 返回使用前面调用的成员函数 `yield_value(int`

value) 准备的当前值(第 61 行)。最后，生成的数字显示在第 92 行，for 循环继续。最后，销毁生成器和 promise。

详细分析之后，我想对控制流进行第一次修改。

7.3.1 修改

代码段和行号都基于前面的 infiniteDataStreamComments.cpp，这里只展示修改部分。

7.3.3.1 协程不会恢复

当我禁用协程恢复(第 91 行中的 gen.getNextValue()) 和其值的显示(第 92 行)时，协程会立即暂停。

```
1 int main() {
2
3     auto gen = getNext();
4     for (int i = 0; i <= 2; ++i) {
5         // auto val = gen.getNextValue();
6         // std::cout << "main: " << val << '\n';
7     }
8
9 }
```

协程永远不会运行。因此，生成器和 promise 创建后，直接销毁。

```
promise_type::promise_type
promise_type::get_return_object
Generator::Generator
promise_type::initial_suspend
Generator::~Generator
promise_type::~promise_type
```

7.3.3.2 initial_suspend 从不挂起

在程序中，成员函数 initial_suspend 返回可等待的 std::suspend_always(第 46 行)，所以 std::suspends_always 会导致协程立即暂停。所以返回 std::suspend_never，而非 std::suspend_always。

```
1 std::suspend_never initial_suspend() {
2     std::cout << " promise_type::initial_suspend" << '\n';
3     return {};
4 }
```

从而，协程立即运行，并在函数 yield_value 调用时暂停。后续 gen.getNextValue() 恢复协程，并再次触发成员函数 yield_value 的执行。结果会忽略起始值 10，所以协程的返回值为 20、30 和 40。

```
promise_type::promise_type
promise_type::get_return_object
Generator::Generator
promise_type::initial_suspend
getNext: start
getNext: before co_yield
promise_type::yield_value
Generator::getNextValue
getNext: after co_yield
getNext: before co_yield
promise_type::yield_value
main: 20
Generator::getNextValue
getNext: after co_yield
getNext: before co_yield
promise_type::yield_value
main: 30
Generator::getNextValue
getNext: after co_yield
getNext: before co_yield
promise_type::yield_value
main: 40
Generator::~Generator
promise_type::~promise_type
```

7.3.3.3 yield_value 从不挂起

成员函数 `yield_value` 由 `co_yield value` 触发，并准备 `current_value`。该函数返回 `std::suspend_always`，因此暂停协程，后续 `gen.getNextValue` 必须恢复协程。当我改变成员函数 `yield_value` 的返回值为 `std::suspend_never` 时，来看看会发生什么。

```
1 std::suspend_never yield_value(int value) {
2     std::cout << " promise_type::yield_value" << '\n';
3     current_value = value;
4     return {};
5 }
```

正如各位猜到的那样，`while` 循环会永远运行，协程不返回任何东西。

```
promise_type::promise_type
promise_type::get_return_object
Generator::Generator
promise_type::initial_suspend
Generator::getNextValue
getNext: start
getNext: before co_yield
promise_type::yield_value
getNext: after co_yield
```

重构 infiniteDataStreamComments.cpp 中的生成器很简单，这样它就能产生有限数量的值了。

7.3.2 泛化

各位可能会好奇，为什么我从来没有使用 Generator 的全泛化。现在，我将调整它的实现，以生成标准模板库任意容器的连续元素。

```
1 // coroutineGetElements.cpp
2
3 #include <coroutine>
4 #include <memory>
5 #include <iostream>
6 #include <string>
7 #include <vector>
8
9 template<typename T>
10 struct Generator {
11
12     struct promise_type;
13     using handle_type = std::coroutine_handle<promise_type>;
14
15     Generator(handle_type h) : coro(h) {}
16
17     handle_type coro;
18
19     ~Generator() {
20         if ( coro ) coro.destroy();
21     }
22     Generator(const Generator&) = delete;
```

```

23 Generator& operator = (const Generator&) = delete;
24 Generator(Generator&& oth): coro(oth.coro) {
25     oth.coro = nullptr;
26 }
27 Generator& operator = (Generator&& oth) {
28     coro = oth.coro;
29     oth.coro = nullptr;
30     return *this;
31 }
32 T getNextValue() {
33     coro.resume();
34     return coro.promise().current_value;
35 }
36 struct promise_type {
37     promise_type() {}
38
39     ~promise_type() {}
40
41     std::suspend_always initial_suspend() {
42         return {};
43     }
44     std::suspend_always final_suspend() noexcept {
45         return {};
46     }
47     auto get_return_object() {
48         return Generator{handle_type::from_promise(*this)};
49     }
50
51     std::suspend_always yield_value(const T value) {
52         current_value = value;
53         return {};
54     }
55     void return_void() {}
56     void unhandled_exception() {
57         std::exit(1);
58     }
59
60     T current_value;
61 };
62
63 };
64
65 template <typename Cont>
66 Generator<typename Cont::value_type> getNext(Cont cont) {
67     for (auto c: cont) co_yield c;
68 }
69
70 int main() {
71 }
```

```

72     std::cout << '\n';
73
74     std::string helloWorld = "Hello world";
75     auto gen = getNext(helloWorld);
76     for (int i = 0; i < helloWorld.size(); ++i) {
77         std::cout << gen.getNextValue() << " ";
78     }
79
80     std::cout << "\n\n";
81
82     auto gen2 = getNext(helloWorld);
83     for (int i = 0; i < 5 ; ++i) {
84         std::cout << gen2.getNextValue() << " ";
85     }
86
87     std::cout << "\n\n";
88
89     std::vector myVec{1, 2, 3, 4 ,5};
90     auto gen3 = getNext(myVec);
91     for (int i = 0; i < myVec.size() ; ++i) {
92         std::cout << gen3.getNextValue() << " ";
93     }
94
95     std::cout << '\n';
96
97 }
```

本例中，生成器进行了实例化，并使用了三次。前两种情况下，`gen`(第 75 行) 和 `gen2`(第 82 行) 使用 `std::string helloWorld` 进行了初始化，而 `gen3` 使用 `std::vector<int>`(第 90 行) 进行初始化。程序的输出应该没啥好说的。第 77 行依次返回字符串 `helloWorld` 的所有字符，第 84 行只返回前 5 个字符，第 92 行返回 `std::vector<int>` 的元素。

您可以在[Compiler Explorer](#)上尝试编译运行该程序。



```
H e l l o w o r l d
H e l l o
1 2 3 4 5
```

简而言之，`Generator<T>` 的实现几乎与前一个示例完全相同。与前一个示例的关键区别是协程 `getNext`。

```

1 template <typename Cont>
2 Generator<typename Cont::value_type> getNext(Cont cont) {
3     for (auto c: cont) co_yield c;
4 }
```

`getNext` 是一个函数模板，接受容器作为参数，并在基于范围的 `for` 循环中遍历容器的所有元素。每次迭代后，函数模板都会暂停。生成器的返回类型可能会让你感到惊讶，`Cont::value_type` 是一个依赖的模板参数，解析器需要对此进行提示。默认情况下，若可以解释为类型或非类型，则编译器假定其是一个非类型。出于这个原因，必须把 `typename` 放在 `Cont::value_type` 的前面。

7.4. 不同的工作流



Cippi 在挖地

在修改 `co_await` 章节的工作流之前，我想让待等待工作流更加透明。

7.4.1 透明的待等待工作流

我在 `startJob.cpp` 中添加了一些输出。

```
1 // startJobWithComments.cpp
2
3 #include <coroutine>
4 #include <iostream>
5
6 struct MySuspendAlways {
7     bool await_ready() const noexcept {
8         std::cout << " MySuspendAlways::await_ready" << '\n';
9         return false;
10    }
11    void await_suspend(std::coroutine_handle<>) const noexcept {
12        std::cout << " MySuspendAlways::await_suspend" << '\n';
13    }
14    void await_resume() const noexcept {
15        std::cout << " MySuspendAlways::await_resume" << '\n';
16    }
17 };
18
19 struct MySuspendNever {
```

```

21 bool await_ready() const noexcept {
22     std::cout << " MySuspendNever::await_ready" << '\n';
23     return true;
24 }
25 void await_suspend(std::coroutine_handle<>) const noexcept {
26     std::cout << " MySuspendNever::await_suspend" << '\n';
27 }
28 }
29 void await_resume() const noexcept {
30     std::cout << " MySuspendNever::await_resume" << '\n';
31 }
32 };
33
34 struct Job {
35     struct promise_type;
36     using handle_type = std::coroutine_handle<promise_type>;
37     handle_type coro;
38     Job(handle_type h) : coro(h) {}
39     ~Job() {
40         if ( coro ) coro.destroy();
41     }
42     void start() {
43         coro.resume();
44     }
45
46
47     struct promise_type {
48         auto get_return_object() {
49             return Job{handle_type::from_promise(*this)};
50         }
51         MySuspendAlways initial_suspend() {
52             std::cout << " Job prepared" << '\n';
53             return {};
54         }
55         MySuspendAlways final_suspend() noexcept {
56             std::cout << " Job finished" << '\n';
57             return {};
58         }
59         void return_void() {}
60         void unhandled_exception() {}
61
62     };
63 };
64
65 Job prepareJob() {
66     co_await MySuspendNever();
67 }
68
69 int main() {

```

```

70
71     std::cout << "Before job" << '\n';
72
73     auto job = prepareJob();
74     job.start();
75
76     std::cout << "After job" << '\n';
77
78 }

```

首先，我用可等待对象 `MySuspendAlways`(第 6 行) 和 `MySuspendNever`(第 20 行) 替换了预定义的可等待对象 `std::suspend_always` 和 `std::suspend_never`，在第 51、55 和 66 行使用了它们。这些可等待对象模仿预定义的可等待对象的行为，但还要加点注解。由于使用 `std::cout`，所以成员函数 `await_ready`、`await_suspend` 和 `await_resume` 不能声明为 `constexpr`。

程序执行的屏幕截图很好地显示了控制流，可以直接在[Compiler Explorer](#)上观察到。

```

Before job
    Job prepared
        MySuspendAlways::await_ready
        MySuspendAlways::await_suspend
        MySuspendAlways::await_resume
        MySuspendNever::await_ready
        MySuspendNever::await_resume
    Job finished
        MySuspendAlways::await_ready
        MySuspendAlways::await_suspend
After job

```

根据请求开启工作 (包括注解)

函数 `initial_suspend`(第 51 行) 在协程的开头执行，`final_suspend` 在协程的末尾执行(第 55 行)。`prepareJob()`(第 73 行) 触发了协程对象的创建，函数调用 `job.start()` 恢复了协程对象，因此完成了协程对象的创建(第 74 行)。因此，`MySuspendAlways` 的 `await_ready`、`await_suspend` 和 `await_resume` 成员将执行。在不恢复成员函数 `final_suspend` 返回的可等待对象(例如：协程对象)的情况下，不会处理 `await_resume`。相反，`MySuspendNever` 函数是立即执行的，因为 `await_ready` 返回 `true`，所以不会挂起。通过这些注解，读者们应该对待等待的工作流有了基本的了解。

现在，来修改它吧。

7.4.2 自动恢复的待等待流

前面的工作流中，我显式地启动了该任务。

```

1 int main() {
2
3     std::cout << "Before job" << '\n';
4
5     auto job = prepareJob();
6     job.start();
7

```

```
8     std::cout << "After job" << '\n';
9
10 }
```

显式调用 `job.start()` 是必要的，因为 `MySuspendAlways` 中的 `await_ready` 总是返回 `false`。现在假设 `await_ready` 可以返回 `true` 或 `false`，并且任务没有显式地启动。一个简短的提醒：当 `await_ready` 返回 `true` 时，函数 `await_resume`(不是 `await_suspend`) 会直接调用。

```
1 // startJobWithAutomaticResumption.cpp
2
3 #include <coroutine>
4 #include <functional>
5 #include <iostream>
6 #include <random>
7
8 std::random_device seed;
9 auto gen = std::bind_front(std::uniform_int_distribution<>(0,1),
10                           std::default_random_engine(seed()));
11
12 struct MySuspendAlways {
13     bool await_ready() const noexcept {
14         std::cout << " MySuspendAlways::await_ready" << '\n';
15         return gen();
16     }
17     bool await_suspend(std::coroutine_handle<> handle) const noexcept {
18         std::cout << " MySuspendAlways::await_suspend" << '\n';
19         handle.resume();
20         return true;
21     }
22     void await_resume() const noexcept {
23         std::cout << " MySuspendAlways::await_resume" << '\n';
24     }
25 };
26
27 struct Job {
28     struct promise_type;
29     using handle_type = std::coroutine_handle<promise_type>;
30     handle_type coro;
31     Job(handle_type h) : coro(h){}
32     ~Job() {
33         if (coro) coro.destroy();
34     }
35
36     struct promise_type {
37         auto get_return_object() {
38             return Job{handle_type::from_promise(*this)};
39         }
40         MySuspendAlways initial_suspend() {
41             std::cout << " Job prepared" << '\n';
42         }
43     };
44 };
45
46 int main() {
47     Job job;
48     job.start();
49     std::cout << " Job started" << '\n';
50     job.await_resume();
51     std::cout << " Job resumed" << '\n';
52 }
```

```

43     return {};
44 }
45 std::suspend_always final_suspend() noexcept {
46     std::cout << " Job finished" << '\n';
47     return {};
48 }
49 void return_void() {}
50 void unhandled_exception() {}

51 };
52 };
53 }

55 Job performJob() {
56     co_await std::suspend_never();
57 }

59 int main() {
60
61     std::cout << "Before jobs" << '\n';
62
63     performJob();
64     performJob();
65     performJob();
66     performJob();
67
68     std::cout << "After jobs" << '\n';
69
70 }

```

现在，协程称为 `performJob` 并自动运行。`gen`(第 9 行) 是数字 0 或 1 的随机数生成器，其使用默认的随机引擎，用种子初始化。因为 `std::bind_front`，所以可以将它与 `std::uniform_int_distribution` 绑定在一起，获得一个可调用对象。使用时，会生成出一个随机数 0 或 1。

例子中，我从 C++ 标准中删除了带有预定义可等待对象的可等待对象，除了作为成员函数 `initial_suspend` 返回类型的可等待对象 `MySuspendAlways`(第 41 行)。`await_ready`(第 13 行) 返回一个布尔值，当为 `true` 时，控制流直接跳转到成员函数 `await_resume`(第 23 行)；当为 `false` 时，协程立即挂起。因此，函数 `await_suspend` 运行(第 17 行)。函数 `await_suspend` 获取协程句柄，并使用它来恢复协程(第 19 行)。而不是返回值 `true`, `await_suspend` 也可以返回 `void`。

下面的截图显示：当 `await_ready` 返回 `true` 时，调用 `await_resume` 函数，当 `await_ready` 返回 `false` 时，调用 `await_suspend` 函数。

读者们可以在[Compiler Explorer](#)上尝试编译和运行该程序。

```

Before jobs
    Job prepared
        MySuspendAlways::await_ready
        MySuspendAlways::await_suspend
        MySuspendAlways::await_resume
    Job finished
    Job prepared
        MySuspendAlways::await_ready
        MySuspendAlways::await_resume
    Job finished
    Job prepared
        MySuspendAlways::await_ready
        MySuspendAlways::await_resume
    Job finished
    Job prepared
        MySuspendAlways::await_ready
        MySuspendAlways::await_resume
    Job finished
After jobs

```

进一步修改程序，并在单独的线程上恢复待等待对象。

7.4.3 自动恢复单独线程上的待等待流

下面的程序是在前一个程序的基础上编写的。

```

1 // startJobWithAutomaticResumptionOnThread.cpp
2
3 #include <coroutine>
4 #include <functional>
5 #include <iostream>
6 #include <random>
7 #include <thread>
8 #include <vector>
9
10 std::random_device seed;
11 auto gen = std::bind_front(std::uniform_int_distribution<>(0,1),
12                           std::default_random_engine(seed()));
13
14 struct MyAwaitable {
15     std::jthread& outerThread;
16     bool await_ready() const noexcept {
17         auto res = gen();
18         if (res) std::cout << " (executed)" << '\n';
19         else std::cout << " (suspended)" << '\n';
20         return res;
21     }
22     void await_suspend(std::coroutine_handle<> h) {
23         outerThread = std::jthread([h] { h.resume(); });
24     }

```

```

25     void await_resume() {}
26 };
27
28
29 struct Job{
30     static inline int JobCounter{1};
31     Job() {
32         ++JobCounter;
33     }
34
35     struct promise_type {
36         int JobNumber{JobCounter};
37         Job get_return_object() { return {}; }
38         std::suspend_never initial_suspend() {
39             std::cout << " Job " << JobNumber << " prepared on thread "
40                 << std::this_thread::get_id();
41             return {};
42         }
43         std::suspend_never final_suspend() noexcept {
44             std::cout << " Job " << JobNumber << " finished on thread "
45                 << std::this_thread::get_id() << '\n';
46             return {};
47         }
48         void return_void() {}
49         void unhandled_exception() {}
50     };
51 };
52
53 Job performJob(std::jthread& out) {
54     co_await MyAwaitable{out};
55 }
56
57 int main() {
58
59     std::vector<std::jthread> threads(8);
60     for (auto& thr: threads) performJob(thr);
61
62 }
```

与前一个程序的区别在于，协程 `performJob` 中使用了新的可等待 `MyAwaitable`(第 54 行)。相反，从协程 `performJob` 返回的协程对象很简单，成员函数 `initial_suspend`(第 38 行) 和 `final_suspend`(第 43 行) 返回预定义的可等待 `std::suspend_never`。此外，这两个函数都显示所执行作业的 `JobNumber` 和所运行的线程 ID。屏幕截图显示了哪个协程立即运行，哪个挂起。有了线程 id，才可以观察到挂起的协程在另一个线程上恢复了。

读者们可以在[Wandbox](#)上尝试该程序。

```
Job 1 prepared on thread 140434982274944 (executed)
Job 1 finished on thread 140434982274944
Job 2 prepared on thread 140434982274944 (suspended)
Job 3 prepared on thread 140434982274944 (suspended)
Job 4 prepared on thread 140434982274944 (suspended)
Job 2 finished on thread 140434877310720
Job 5 prepared on thread 140434982274944 (executed)
Job 5 finished on thread 140434982274944
Job 6 prepared on thread 140434982274944 (suspended)
Job 7 prepared on thread 140434982274944 (suspended)
Job 3 finished on thread 140434868918016
Job 8 prepared on thread 140434982274944 (executed)
Job 8 finished on thread 140434982274944
Job 4 finished on thread 140434860525312
Job 6 finished on thread 140434852132608
Job 7 finished on thread 140434843739904
```

来讨论一下这个程序的有趣的控制流程。第 59 行创建了 8 个默认构造的线程，协程 performJob(第 53 行) 通过引用接受这些线程。此外，引用成为创建 MyAwaitable{out} 的参数(第 54 行)。根据 res 的值(第 17 行)，以及函数 await_ready 的返回值，协程将继续(res 为 true) 运行或挂起(res 为 false)。若 MyAwaitable 挂起，函数 await_suspend(第 22 行) 将执行。由于分配了 outerThread(第 23 行)，其变成了一个正在运行的线程，从而生命周期必须比协程的长，所以线程具有 main 函数的作用域。

总结

- 想要多次同步线程时，有许多选项，比如：条件变量、`std::atomic_flag`、`std::atomic<bool>` 或信号量。这个案例研究回答了一个问题：哪个变体是最快的？数字表明，条件变量是最慢的方式，原子标记是同步线程的最快方式。`std::atomic<bool>` 的性能介于两者之间，信号量几乎和原子标志一样快。
- 使用 `co_return`，协程的章节中介绍了一个立即执行的 `future`。这个 `future` 是一个理想的起点，可以让它变得懒惰，最后让它在自己的线程上运行。
- 对无限数据流的生成器的修改了解了其本质。当成员函数 `initial_suspend` 返回 `std::suspend_never` 时，协程立即启动并忽略第一个值。相反，从函数 `yield_value` 返回 `std::suspend_never` 将以无限循环结束。当忘记恢复协程时，其将永远不会运行。
- 泛型 `Generator<T>`，可以连续返回标准模板库中任意容器的元素，而不是无限的数据流。
- 实现自己的可等待的 `MySuspendNever` 和 `MySuspendAlways` 使等待工作流透明。调整可等待的 `MySuspendAlways` 使其能够创建一个在必要时恢复自身的待等待对象。
- 修改可等待对象，可使协程在其他线程上恢复运行。

第四部分：后记

恭喜你！当你阅读这些文字时，已经掌握了具有挑战性和令人激动的 C++20 标准。C++20 是一个 C++ 标准，其他 C++ 标准一样具有影响力，例如：C++98 和 C++11。由于 C++11，C++ 社区使用了以下 C++ 标准的名称。

- 旧 C++: C++98 和 C++03
- 现代 C++: C++11, C++14 和 C++17
- < 占位符 >: C++20

我不确定将来 C++20 会用什么名字。我只确定 C++20 开创了新的 C++ 领域，特别是四大特性改变了我们使用 C++ 的方式。

- 概念：彻底改变了我们思考和编写泛型代码的方式。从而，可以第一次在语义类别（如数字或顺序）中对我们的程序进行推理。
- 模块：模块是软件组件的起点，有助于克服遗留头文件和宏的不足。
- 范围：范围库用功能思维扩展了标准模板库。算法可以直接在容器上操作，可以延迟计算，也可以进行组合。
- 协程：使得异步编程成为 C++ 中的一等公民。协程转换等待中的阻塞函数调用，在模拟、服务器或用户界面等事件驱动系统中非常有用。

C++20 只是一个起点。C++23 中，还有很多工作要做，以充分集成和利用 C++ 中四大特性的潜力。这里，我来说一些对 C++ 未来的想法。

- 标准模板库是由 [Alexander Stephanov](#) 设计的。不过，C++20 中缺少对概念的集成。
- 可以期待一个模块化的标准模板库，并希望 C++ 有一个打包系统。
- 许多函数式编程中已知的算法在范围库中仍然没有，未来的 C++ 标准应该改进范围算法和标准容器之间的互动。
- 还没有真正“好用的”协程，目前只有一个用来构建协程强大框架，而协程库很有可能在 C++23 中出现。

在关于 C++23 及以后标准的章节中，我给出了关于 C++ 不久的将来的更多细节。简而言之：C++ 具有光明的未来。

第五部分：更多信息

第 8 章 C++23 和之后的标准

大家都会认为一个重要的 C++ 标准之后是一个小的 C++ 标准更新，这是错误的。C++23 将提供和 C++20 一样强大的扩展。Ville Voutilainen 的建议[P0592R4](#)“大胆地建议 C++23 的总体计划”，让人们对即将到来的 C++23 标准有了初步的了解。Ville 命名了七个特性。

- C++23
 - 支持协程序
 - 模块化标准库
 - 执行器
 - 网络
- C++23 或之后的标准
 - 反射
 - 模式匹配
 - 契约

前四个特性是针对 C++23 的，其余三个没有具体的时间表。反射、模式匹配和契约很可能会相继添加到 C++ 标准中。

“预测非常困难，尤其是关于未来的预测。”——([Niels Bohr](#))。所以，可以把这一章作为我预测 C++ 未来的一种尝试。

8.1. C++23

协程序、模块化标准库和执行器都是 C++23 的一部分。

8.1.1 协程序

C++20 中的协程只不过是实现具体协程的框架，所以需要软件开发人员来实现协同程序。Lewis Baker 的[cppcoro](#)库给出了协程序的第一个概念，该库提供了 C++20 没有的“高级协程”。

使用 cppcoro

cppcoro 库是基于协程 TS 的。TS 代表技术规范，是 C++20 所提供的协程功能的初步版本。Lewis大概会将 cppcoro 库从协程 TS 移植为 C++20 中定义的协程。该库可以在 Windows (Visual Studio 2017) 或 Linux (Clang 5.0/6.0 和 libc++) 上使用。在我的实验中，对所有示例都使用了以下命令行：

cppcoro 的命令行

```
clang++ -std=c++17 -fcoroutines-ts -Iinclude -stdlib=libc++ libcppcoro.a  
cppcoroTask.cpp -pthread
```

- `-std=c++17`: 支持 C++17
- `-fcoroutines-ts` : 支持 C++ 协程 TS
- `-Iinclude` : `cppcoro` 头文件
- `-stdlib=libc++`: LLVM 标准库的实现
- `libcppcoro.a`: `cppcoro` 库

当 `cppcoro` 基于 C++20 的协程时，就可以将它们用于支持 C++20 的编译器。此外，它们还让你对 C++23 中的具体协程实现有了初步了解。

本节关于协程库的其余部分中，我想演示几个示例，以展示协程的强大。我的演示从协程的类型开始。

8.1.1.1 协程的类型

`cppcoro` 具有很多生成器，可以执行有各种各样的任务 (task)。

8.1.1.1.1 task<T>

什么是任务 (task)? `cppcoro` 中使用的定义是:

- 任务表示延迟执行的异步计算，其中协程的执行直到等待任务时才开始。

任务是一个协程。下面的程序中，函数 `main` 首先等待函数，第一个等待第二个，第二个等待第三个。

```

1 // cppcoroTask.cpp
2
3 #include <chrono>
4 #include <iostream>
5 #include <string>
6 #include <thread>
7
8 #include <cppcoro/sync_wait.hpp>
9 #include <cppcoro/task.hpp>
10
11 using std::chrono::high_resolution_clock;
12 using std::chrono::time_point;
13 using std::chrono::duration;
14
15 using namespace std::chrono_literals;
16
17 auto getTimeSince(const time_point<high_resolution_clock>& start) {
18
19     auto end = high_resolution_clock::now();
20     duration<double> elapsed = end - start;
21     return elapsed.count();
22 }
23 }
```

```

25    cppcoro::task<> third(const time_point<high_resolution_clock>& start) {
26
27        std::this_thread::sleep_for(1s);
28        std::cout << "Third waited " << getTimeSince(start) << " seconds." << '\n';
29
30        co_return;
31    }
32
33
34    cppcoro::task<> second(const time_point<high_resolution_clock>& start) {
35
36        auto thi = third(start);
37        std::this_thread::sleep_for(1s);
38        co_await thi;
39
40        std::cout << "Second waited " << getTimeSince(start) << " seconds." << '\n';
41    }
42
43
44    cppcoro::task<> first(const time_point<high_resolution_clock>& start) {
45
46        auto sec = second(start);
47        std::this_thread::sleep_for(1s);
48        co_await sec;
49
50        std::cout << "First waited " << getTimeSince(start) << " seconds." << '\n';
51
52    }
53
54    int main() {
55
56        std::cout << '\n';
57
58        auto start = high_resolution_clock::now();
59        cppcoro::sync_wait(first(start));
60
61        std::cout << "Main waited " << getTimeSince(start) << " seconds." << '\n';
62
63        std::cout << '\n';
64    }
65

```

诚然，这个程序没有做什么有意义的事情，但它有助于理解协同程序的工作流程。

首先，主函数不能是协程。Cppcoro::sync_wait(第 59 行)通常用作启动顶级任务，并等待任务完成。与其他协程类似，协程首先获取开始时间作为参数，并显示其执行时间。协程首先做什么？第二次启动协程(第 36 和 46 行)，立即暂停，休眠一秒，然后通过句柄 sec 恢复协程(第 38 和 48 行)。第二个协程执行相同的工作流，但第三个协程不执行。至于第三个，它是一个协程，不返回任何东西，也不等待另一个协程。当第三个完成时，所有其他协程都会执行，所以每个协程需要 3 秒。

```
rainer@seminar:~/> cppcoroTask
Third waited 3.00058 seconds.
Second waited 3.00065 seconds.
First waited 3.00068 seconds.
Main waited 3.00072 seconds.
```

稍微改变一下程序。若协程在 `co_await` 调用后休眠会发生什么？

```
// cppcoroTask2.cpp

#include <chrono>
#include <iostream>
#include <string>
#include <thread>

#include <cppcoro/sync_wait.hpp>
#include <cppcoro/task.hpp>

using std::chrono::high_resolution_clock;
using std::chrono::time_point;
using std::chrono::duration;

using namespace std::chrono_literals;

auto getTimeSince(const time_point<::high_resolution_clock>& start) {
    auto end = high_resolution_clock::now();
    duration<double> elapsed = end - start;
    return elapsed.count();
}

cppcoro::task<> third(const time_point<high_resolution_clock>& start) {
    std::cout << "Third waited " << getTimeSince(start) << " seconds." << '\n';
    std::this_thread::sleep_for(1s);
    co_return;
}

cppcoro::task<> second(const time_point<high_resolution_clock>& start) {
    auto thi = third(start);
    co_await thi;
```

```

37     std::cout << "Second waited " << getTimeSince(start) << " seconds." << '\n';
38     std::this_thread::sleep_for(1s);
39
40 }
41
42 cppcoro::task<> first(const time_point<high_resolution_clock>& start) {
43
44     auto sec = second(start);
45     co_await sec;
46
47     std::cout << "First waited " << getTimeSince(start) << " seconds." << '\n';
48     std::this_thread::sleep_for(1s);
49
50 }
51
52 int main() {
53
54     std::cout << '\n';
55
56     auto start = ::high_resolution_clock::now();
57
58     cppcoro::sync_wait(first(start));
59
60     std::cout << "Main waited " << getTimeSince(start) << " seconds." << '\n';
61
62     std::cout << '\n';
63
64 }
```

主函数等待 3 秒，但每个迭代调用的协程要少等待 1 秒。

cppcoro 提供的下一个协程是 generator<T>。

8.1.1.2 generator<T>

以下是 cppcoro 对生成器的定义：

- 生成器表示生成 T 类型值序列的协程类型，其中的值是惰性和同步生成的。

话不多说，cppcoroGenerator.cpp 程序演示了两个实际运行的生成器。

```

1 // cppcoroGenerator.cpp
2
3 #include <iostream>
4 #include <cppcoro/generator.hpp>
5
6 cppcoro::generator<char> hello() {
7     co_yield 'h';
8     co_yield 'e';
9     co_yield 'l';
10    co_yield 'l';
11    co_yield 'o';
```

```

12 }
13
14 cppcoro::generator<const long long> fibonacci() {
15     long long a = 0;
16     long long b = 1;
17     while (true) {
18         co_yield b;
19         auto tmp = a;
20         a = b;
21         b += tmp;
22     }
23 }
24
25 int main() {
26
27     std::cout << '\n';
28     for (auto c: hello()) std::cout << c;
29
30     std::cout << "\n\n";
31
32     for (auto i: fibonacci()) {
33         if (i > 1'000'000) break;
34         std::cout << i << " ";
35     }
36
37     std::cout << "\n\n";
38
39 }

```

第一个协程 `hello` 应请求返回下一个字符，协程 `fibonacci` 返回下一个 `fibonacci` 数字。斐波那契数列可以创建无限的数据流。第 33 行发生了什么呢？基于范围的 `for` 循环触发协程的执行。第一次迭代启动协程，返回 `co_yield b` 处的值（第 18 行），然后暂停。后续基于范围的 `for` 循环调用恢复协程 `fibonacci` 并返回下一个 `fibonacci` 数字。

```

rainer : bash — Konsole
File Edit View Bookmarks Settings Help
rainer@seminar:~> cppcoroGenerator
hello
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 832040
rainer@seminar:~> █

```

`cppcoro` 中提供了更多可等待的类型。

8.1.1.2 可等待类型

`cppcoro` 支持各种可等待类型：

- `single_consumer_event`
- `single_consumer_async_auto_reset_event`

- `async_mutex`
- `async_manual_reset_event`
- `async_auto_reset_event`
- `async_latch`
- `sequence_barrier`
- `multi_producer_sequencer`
- `single_producer_sequencer`

我想仔细聊聊可等待的 `single_consumer_event` 和 `async_mutex`。

8.1.1.2.1 `single_consumer_event`

根据文档，`single_consumer_event` 是一个手动重置事件类型，一次只支持单个协程的等待。`single_consumer_event` 为线程的一次性同步提供了一种新方法。

```

1 // cppcoroProducerConsumer.cpp
2
3 #include <cppcoro/single_consumer.hpp>
4 #include <cppcoro/sync_wait.hpp>
5 #include <cppcoro/task.hpp>
6
7 #include <future>
8 #include <iostream>
9 #include <string>
10 #include <thread>
11 #include <chrono>
12
13 cppcoro::single_consumer_event event;
14
15 cppcoro::task<> consumer() {
16
17     auto start = std::chrono::high_resolution_clock::now();
18
19     co_await event; // suspended until some thread calls event.set()
20
21     auto end = std::chrono::high_resolution_clock::now();
22     std::chrono::duration<double> elapsed = end - start;
23     std::cout << "Consumer waited " << elapsed.count() << " seconds." << '\n';
24
25     co_return;
26 }
27
28 void producer() {
29
30     using namespace std::chrono_literals;
31     std::this_thread::sleep_for(2s);
32
33     event.set(); // resumes the consumer
34 }
```

```

35 }
36
37 int main() {
38
39     std::cout << '\n';
40
41     auto con = std::async([]{ cppcoro::sync_wait(consumer()); });
42     auto prod = std::async(producer);
43
44     con.get(), prod.get();
45
46     std::cout << '\n';
47
48 }

```

消费者 (第 41 行) 和生产者 (第 42 行) 在各自的线程中运行。因为主函数不能是协程，所以 `cppcoro::sync_wait(consumer())` 调用 (第 41 行) 作为顶级任务。调用一直等待，直到协程使用者完成。协程消费者在调用 `co_await` 事件 (第 19 行) 中等待，直到调用 `event.set()`(第 33 行)。函数生产者在休眠两秒后发送它的事件。



`cppcoro` 还支持[mutex](#)。

8.1.1.2.2 `async_mutex`

像 `cppcoro::async_mutex` 这样的互斥锁是一种同步机制，多个线程中只有一个能够访问保护的共享数据。

```

1 // cppcoroMutex.cpp
2
3 #include <cppcoro/async_mutex.hpp>
4 #include <cppcoro/sync_wait.hpp>
5 #include <cppcoro/task.hpp>
6
7 #include <iostream>
8 #include <thread>
9 #include <vector>
10
11
12 cppcoro::async_mutex mutex;
13
14 int sum{};

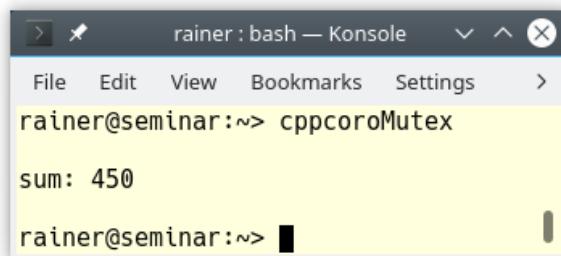
```

```

15
16   /cppcoro::task<> addToSum(int num) {
17        cppcoro::async_mutex_lock lockSum = co_await mutex.scoped_lock_async();
18        sum += num;
19    }
20
21
22 int main() {
23
24     std::cout << '\n';
25
26     std::vector<std::thread> vec(10);
27
28     for(auto& thr: vec) {
29         thr = std::thread([]{
30             for(int n = 0; n < 10; ++n) cppcoro::sync_wait(addToSum(n));
31         });
32
33     for(auto& thr: vec) thr.join();
34
35     std::cout << "sum: " << sum << '\n';
36
37     std::cout << '\n';
38
39 }

```

第 26 行创建 10 个线程，每个线程将数字 0 到 9 添加到共享 sum 变量（第 14 行）。addToSum 函数是协程，在表达式 co_await mutex.scoped_lock_async()（第 17 行）中等待，直到获得互斥量。等待互斥锁的协程不是阻塞状态，而是挂起状态。前一个锁持有者在其解锁调用中恢复等待协程，所以互斥锁一直保持锁定状态，直到作用域结束（第 20 行）。



8.1.1.3 函数

还有更有趣的函数可以用来处理可等待对象。

- sync_wait()
- when_all()
- when_all_ready()
- fmap()

- `schedule_on()`
- `resume_on()`

`when_all` 函数创建一个等待所有输入可等待对象的可等待对象，并返回它们各自结果的聚合。

看看下面的代码：

```

1 // cppcoroWhenAll.cpp
2
3 #include <chrono>
4 #include <iostream>
5 #include <thread>
6
7 #include <cppcoro/sync_wait.hpp>
8 #include <cppcoro/task.hpp>
9 #include <cppcoro/when_all.hpp>
10
11 using namespace std::chrono_literals;
12
13 cppcoro::task<std::string> getFirst() {
14     std::this_thread::sleep_for(1s);
15     co_return "First";
16 }
17
18 cppcoro::task<std::string> getSecond() {
19     std::this_thread::sleep_for(1s);
20     co_return "Second";
21 }
22
23 cppcoro::task<std::string> getThird() {
24     std::this_thread::sleep_for(1s);
25     co_return "Third";
26 }
27
28
29 cppcoro::task<> runAll() {
30
31     auto[fir, sec, thi] = co_await cppcoro::when_all(getFirst(), getSecond(),
32     getThird());
33
34     std::cout << fir << " " << sec << " " << thi << '\n';
35
36 }
37
38 int main() {
39
40     std::cout << '\n';
41
42     auto start = std::chrono::steady_clock::now();
43
44     cppcoro::sync_wait(runAll());

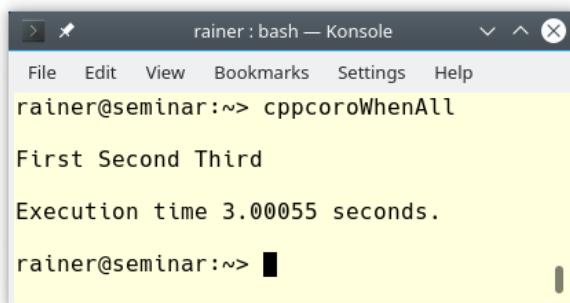
```

```

45
46     std::cout << '\n';
47
48     auto end = std::chrono::high_resolution_clock::now();
49     std::chrono::duration<double> elapsed = end - start;
50     std::cout << "Execution time " << elapsed.count() << " seconds." << '\n';
51
52     std::cout << '\n';
53
54 }

```

顶层任务 `cppcoro::sync_wait(runAll())`(第 44 行) 等待可等待的 `runAll`, 后者等待可等待的 `getFirst`、`getSecond` 和 `getThird`(第 31 行)。可等待的 `runAll`、`getFirst`、`getSecond` 和 `getThird` 是协程。每个 `get` 函数休眠一秒钟(第 14、19 和 24 行), 三乘以一秒等于三秒。这是调用 `cppcoro::sync_wait(runAll())` 等待协程的时间, 第 49 行显示时间持续时间。



可以将 cpp 中的 `when_all` 和线程池结合起来

8.1.1.4 static_thread_pool

`static_thread_pool` 在固定大小的线程池上进行调度工作。

可以调用 `cppcoro::static_thread_pool`, 也可以不调用。这个数字表示创建的线程数。若不指定数字, 则使用 C++11 中加入的 `std::thread::hardware_concurrency()`。`std::thread::hardware_concurrency` 给出了系统支持的硬件线程数, 这可能是处理器或内核的数量。

下面的示例基于 `cppcoroWhenAllOnThreadPool.cpp`, 使用了可等待的 `when_all`。这一次, 协程并发执行。

```

1 // cppcoroWhenAllOnThreadPool.cpp
2
3 #include <chrono>
4 #include <iostream>
5 #include <thread>
6
7 #include <cppcoro/sync_wait.hpp>
8 #include <cppcoro/task.hpp>
9 #include <cppcoro/static_thread_pool.hpp>
10 #include <cppcoro/when_all.hpp>
11
12

```

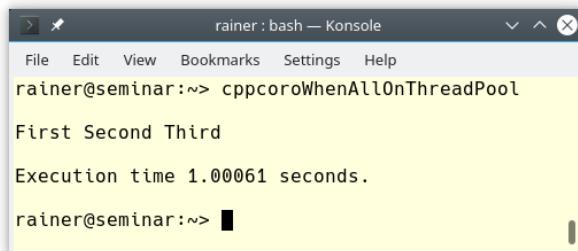
```

13 using namespace std::chrono_literals;
14
15 cppcoro::task<std::string> getFirst() {
16     std::this_thread::sleep_for(1s);
17     co_return "First";
18 }
19
20 cppcoro::task<std::string> getSecond() {
21     std::this_thread::sleep_for(1s);
22     co_return "Second";
23 }
24
25 cppcoro::task<std::string> getThird() {
26     std::this_thread::sleep_for(1s);
27     co_return "Third";
28 }
29
30 template <typename Func>
31 cppcoro::task<std::string> runOnThreadPool(cppcoro::static_thread_pool& tp,
32                                         Func func) {
33     co_await tp.schedule();
34     auto res = co_await func();
35     co_return res;
36 }
37
38 cppcoro::task<> runAll(cppcoro::static_thread_pool& tp) {
39
40     auto[fir, sec, thi] = co_await cppcoro::when_all(
41         runOnThreadPool(tp, getFirst),
42         runOnThreadPool(tp, getSecond),
43         runOnThreadPool(tp, getThird));
44
45     std::cout << fir << " " << sec << " " << thi << '\n';
46
47 }
48
49 int main() {
50
51     std::cout << '\n';
52
53     auto start = std::chrono::steady_clock::now();
54
55     cppcoro::static_thread_pool tp;
56     cppcoro::sync_wait(runAll(tp));
57
58     std::cout << '\n';
59
60     auto end = std::chrono::high_resolution_clock::now();
61     std::chrono::duration<double> elapsed = end - start;

```

```
62     std::cout << "Execution time " << elapsed.count() << " seconds." << '\n';
63
64     std::cout << '\n';
65
66 }
```

这是与之前的程序 `cppcoroWhenAll.cpp` 的关键区别。第 55 行，创建了一个线程池 `tp`，并将其用作函数 `runAll(tp)` 的参数（第 56 行）。函数 `runAll` 使用线程池并发地启动协程，因为有了结构化绑定（第 40 行），每个协程的值可以很容易地聚合并分配给一个变量。最后，`main` 函数只需要 1 秒而不是 3 秒。



8.1.2 模块化的标准库

也许您想停止使用标准库头文件？Microsoft 根据 C++ 提案[P0541](#)支持所有 STL 头文件的模块。Microsoft 的实现可以让开发者初步了解模块化的标准库是什么样的。以下是我 Microsoft C++ 团队博客的帖子[Visual Studio 2017 中使用 C++ 模块](#)中找到的内容。

8.1.2.1 Visual Studio 2017 中使用 C++ 模块

- `std.regex` 提供了头文件 `<regex>` 的内容
- `std.filesystem` 提供了头文件 `<experimental/filesystem>` 的内容
- `std.memory` 提供了头文件 `<memory>` 的内容
- `std.threading` 提供了头文件 `<atomic>`, `<condition_variable>`, `<future>`, `<mutex>`, `<shared_mutex>` 和 `<thread>` 的内容
- `std.core` 提供了 C++ 标准库中的所有其他内容

要使用 Microsoft 标准库模块，必须指定异常处理模型 (`/EHsc`) 和多线程库 (`/MD`)。此外，必须使用 `/std:c++latest` 和 `/experimental:module`。

在模块部分，我使用了以下模块定义。

具有全局模块的模块定义

```
1 // math1..hxx
2
3 module;
4
5 #include <numeric>
6 #include <vector>
```

```

7
8 export module math;
9
10 export int add(int fir, int sec) {
11     return fir + sec;
12 }
13
14 export int getProduct(const std::vector<int>& vec) {
15     return std::accumulate(vec.begin(), vec.end(), 1, std::multiplies<int>());
16 }
```

可以使用模块化的标准库直接重构此模块定义，必须用模块 std.core 替换头文件 <numeric> 和 <vector>。

将模块 std.core 导入到接口文件

```

1 // math2.ixx
2 module;
3
4 export module math;
5
6 import std.core;
7
8 export int add(int fir, int sec) {
9     return fir + sec;
10 }
11
12 export int getProduct(const std::vector<int>& vec) {
13     return std::accumulate(vec.begin(), vec.end(), 1, std::multiplies<int>());
14 }
```

此外，必须使用模块 std.core，而不是标准头文件：

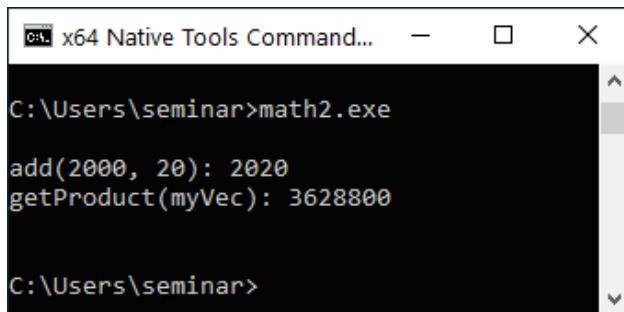
将模块 std.core 导入

```

1 // client2.cpp
2
3 import math;
4 import std.core;
5
6 int main() {
7
8     std::cout << '\n';
9
10    std::cout << "add(2000, 20): " << add(2000, 20) << '\n';
11
12    std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
13
14    std::cout << "getProduct(myVec): " << getProduct(myVec) << '\n';
15 }
```

```
16     std::cout << '\n';
17 }
```

该程序产生预期的输出:



```
C:\Users\seminar>math2.exe
add(2000, 20): 2020
getProduct(myVec): 3628800
C:\Users\seminar>
```

在 Windows 上使用模块 std.core

8.1.3 执行器

执行器在 C++ 中的历史相当悠久，有关讨论早在 2010 年就开始了。关于细节，Detlef Vollmann 在他的演讲[Finally Executors For C++](#)中给出了一个很好的概述。

我对执行器的介绍主要基于对执行器[P0761](#)的设计建议，以及其正式描述[P0443](#)。我还提到了相对较新的[保守执行器提案 P1055](#)。

首先，啥是执行器？

执行器是 C++ 中执行的基本构建块，并在执行中扮演类似的角色，例如：C++ 中容器的分配器。许多关于执行器的建议已经发布，但还有很多设计决策仍然悬而未决。它们应该是 C++23 的一部分，但是可以更早地用于 C++ 扩展标准。

执行器由关于在何处、何时，以及如何运行可调用对象的规则组成。

- Where: 可调用对象可以在内部或外部处理器上运行，结果可以从内部或外部处理器读取。
- When: 可调用对象可以立即运行，也可以只是调度。
- How: 可调用对象可以在 CPU 或 GPU 上运行，甚至可以以向量化的方式执行。

C++ 的并发性和并行性特性，很大程度依赖于执行器的构建块。这种依赖关系适用于现有的并发特性，例如：[并行算法的标准模板库](#)，但也适用于新的并发特性，例如：[闩门和栅栏](#)，[协程](#)，[网络库](#)，[扩展的 future](#)，[事务性内存](#)或[任务块](#)。

8.1.3.1 一个例子

下面的代码应该会给展示一个关于执行器的使用。

8.1.3.1.1 使用执行器

- 使用 std::async

```
1 // get an executor through some means
2 my_executor_type my_executor = ...
3
4 // launch an async using my executor
```

```
5 auto future = std::async(my_executor, [] {
6     std::cout << "Hello world, from a new execution agent!" < '\n';
7});
```

- STL 算法 std::for_each

```
1 // get an executor through some means
2 my_executor_type my_executor = ...
3
4 // execute a parallel for_each "on" my executor
5 std::for_each(std::execution::par.on(my_executor),
6               data.begin(), data.end(), func);
```

8.1.3.1.2 获取执行器

有多种可获取执行器。

- 从 static_thread_pool 的执行上下文中获取

```
1 // create a thread pool with 4 threads
2 static_thread_pool pool(4);
3
4 // get an executor from the thread pool
5 auto exec = pool.executor();
6
7 // use the executor on some long-running task
8 auto task1 = long_running_task(exec);
```

- 从系统执行器中获取

若没有指定，系统执行器是默认执行器。

- 从一个执行器适配器中获取

```
1 // get an executor from a thread pool
2 auto exec = pool.executor();
3
4 // wrap the thread pool's executor in a logging_executor
5 logging_executor<decltype(exec)> logging_exec(exec);
6
7 // use the logging executor in a parallel sort
8 std::sort(std::execution::par.on(logging_exec), my_data.begin(), my_data.end());
```

logging_executor 是池执行器的包装器。

8.1.3.2 执行器的目标

根据提案[P1055](#)，执行器的目标是什么呢？

- **批处理:** 控制可调用对象转换的成本与其体积之间的权衡。
- **异构:** 允许可调用对象在异构上下文中运行并返回结果。

- **有序:** 指定调用可调用对象的顺序。目标包括顺序保证，如后进先出、FIFO 先入先出执行、优先级或时间限制，甚至是顺序执行。
- **可控:** 可调用对象必须是针对特定计算资源的，可以延迟，甚至取消。
- **可持续:** 对于非阻塞提交的工作单元，需要来自工作单元的信号。这些信号必须指示结果是否可用、是否发生错误、可调用对象何时完成，或被调用者是否想要取消可调用对象。也应该可以显式地启动可调用对象或停止启动。
- **可分层:** 层次结构允许在不增加简单用例复杂性的情况下添加新功能。
- **可用:** 实现人员和用户的易用性应该是主要目标。
- **可组合:** 允许用户为不属于标准的特性扩展执行器。
- **最小化:** 执行器概念上不需要其他东西。

8.1.3.3 执行函数

执行器提供一个或多个执行函数，用于从可调用对象创建执行代理。执行程序必须至少支持以下六个函数中的一个。

执行器的函数

成员函数	基数	方向
execute	单个	oneway
twoway_execute	单个	twoway
then_execute	单个	then
bulk_execute	批量	oneway
bulk_twoway_execute	批量	twoway
bulk_then_execute	批量	then

每个执行函数都有两个属性: 基数和方向。

- **基数:**
 - 单个: 创建一个执行代理
 - 批量: 创建一组执行代理
- **方向:**
 - oneway: 创建执行代理，但不返回结果
 - twoway: 创建执行代理并返回可用于等待执行完成的 future
 - then: 创建执行代理并返回可用于等待执行完成的 future，执行代理在给定的 future 准备就绪后开始执行。

下面几行对执行函数进行了更正式的解释。

首先，引用单基数的情况:

- oneway 执行函数是即发即忘的任务，非常类似于“即发即弃”，但它不会自动阻塞future的析构函数。

- `twoway` 执行函数返回一个 `future`, 可以用它来获取结果。其行为类似于 `std::promise`, 将返回相关 `std::future` 的句柄。
- `then` 执行函数是一个延续, 返回一个 `future`, 但是只有在提供的 `future` 准备就绪时, 执行代理才会运行。

其次, 批量基数的情况更复杂。这些函数创建一组执行代理, 每个执行代理调用给定的可调用对象。它们返回工厂的结果, 而不是执行代理调用的单个可调用 `f` 的结果。用户负责通过该工厂, 消除正确结果的歧义。

8.1.3.3.1 `execution::require`

如何确保执行程序支持特定的执行函数?

使用执行函数 `execute` 的执行器

```
1 void concrete_context(const my_oneway_single_executor& ex)
2 {
3     auto task = ...;
4     ex.execute(task);
5 }
```

通常, 可以使用函数 `execute::require` 来请求。

需要单一和 `twoway` 执行函数的执行器

```
1 template <typename Executor>
2 void generic_context(const Executor& ex)
3 {
4     auto task = ...;
5
6     // ensure .twoway_execute() is available with execution::require()
7     execution::require(ex, execution::single, execution::twoway).twoway_execute(task)\n
8 };
9 }
```

所以, 执行器 `ex` 必须支持单基数和 `twoway` 执行。

8.1.4 网络库

C++23 中的网络库基于 Christopher M. Kohlhoff 的 `boost::asio` 库。该库的目标是网络和底层 I/O 编程。

下面的组件是网络库的一部分

- TCP、UDP 和多路广播
- 客户机/服务器应用
- 更多并发连接的可扩展性
- IPv4 和 IPv6
- 域名解析 (DNS)

- 时钟

但是，以下组件不是网络库的一部分：

- 网络协议的实现，如 HTTP、SMTP 或 FTP
- 加密 (SSL 或 TLS)
- 操作特定的多路复用接口，如 select 或轮询
- 支持实时
- TCP/IP 协议，如 ICMP

由于支持了网络库，就可以直接实现一个回显服务器。

```

1 template <typename Iterator>
2 void uppercase(Iterator begin, Iterator end) {
3     std::locale loc("en");
4     for (Iterator iter = begin; iter != end; ++iter)
5         *iter = std::toupper(*iter, loc);
6 }
7
8 void sync_connection(tcp::socket& socket) {
9     try {
10         std::vector<char> buffer_space(1024);
11         while (true) {
12             std::size_t length = socket.read_some(buffer(buffer_space));
13             uppercase(buffer_space.begin(), buffer_space.begin() + length);
14             write(socket, buffer(buffer_space, length));
15         }
16     }
17     catch (std::system_error& e) {
18         // ...
19     }
20 }
```

服务器获取客户端套接字 (第 8 行)，读取文本 (第 12 行)，将文本转换为大写字母 (第 13 行)，并将文本发送回客户端 (第 14 行)。

boost 库有更多聊天或 HTTP 服务器的示例。此外，服务器可以同步运行 (如程序中所示)，也可以异步运行。

8.2. C++23 或之后的标准

下面的三个特性，契约、反射和模式匹配，不确定是否会成为 C++23 的一部分。总的想法是，它们应该是即将到来的 C++ 标准的一部分。C++23 可能会部分支持它们。

8.2.1 契约

契约本来是 C++20 的第五大特性。由于设计问题，在 2019 年 7 月在科隆举行的标准化委员会上移除。同时，创建了研究契约的[第 21 研究组](#)。

- 什么是契约？

契约以精确和可检查的方式，为软件组件指定接口。这些软件组件通常是函数和成员函数，必须满足前置条件、后置条件或不变式(断言)。以下是这三个术语的简化定义：

- 前置条件：假定在函数输入时持有的谓词
- 后置条件：假定在函数退出时保持的谓词
- 断言：在计算中假定在其点上保持不变的谓词

前置条件和后置条件放置在函数定义之外，但不变式(断言)放置在函数定义内部。谓词是一个返回布尔值的函数。

这里是一个例子：

```
1 int push(queue& q, int val)
2   [[ expects: !q.full() ]]
3   [[ ensures !q.empty() ]] {
4     ...
5     [[ assert: q.is_ok() ]]
6     ...
7 }
```

属性 expect 是先决条件，属性 ensure 后置条件，属性 assert 是断言。函数 push 的契约是，在添加元素之前队列不满，在添加元素之后队列不空，并且断言 q.is_ok() 成立。

前置条件和后置条件是函数接口的一部分，所以不能访问函数的局部成员或类的私有或受保护成员。然而，断言是实现的一部分，因此可以访问类的私有成员或受保护成员函数的局部变量：

```
1 class X {
2 public:
3   void f(int n)
4   [[ expects: n < m ]] // error; m is private
5   {
6     [[ assert: n < m ]]; // OK
7     // ...
8   }
9 private:
10   int m;
11 };
```

成员 m 是私有的，因此不能成为前提条件的一部分。默认情况下，违反契约将终止程序。

并且，可以调整属性的行为。

8.2.1.1 调整属性

适配属性的语法比较复杂：[[契约属性修饰符：条件表达式]]。

- 契约属性：期望、保证和断言
- 修饰符：指定契约级别或契约的强制执行，其值包括 default、audit 和 axiom
 - default：运行时检查的成本应该很小，是默认的修饰符
 - audit：假定运行时检查的成本很大
 - axiom：运行时不检查谓词

- 条件表达式: 契约的谓词

对于保证属性，还有一个额外的标识符可用: [[保证修饰符标识符: 条件表达式]]
标识符允许引用函数的返回值。

```

1 int mul(int x, int y)
2   [[expects: x > 0]] // implicit default
3   [[expects default: y > 0]]
4   [[ensures audit res: res > 0]] {
5     return x * y;
6 }
```

res 作为标识符是名称。如示例所示，可以使用更多相同类型的契约。

现在，来看看违反契约的处理方式。

8.2.1.2 违约的处理

编译有三个断言构建级别:

- off: 不检查契约
- default: 默认检查契约
- audit: 默认检查契约和审核契约

当发生违约时，由于谓词返回 false，将调用违约处理程序。违约规处理程序会得到一个 std::contract_violation 类型的值，此值提供关于违约的详细信息。

```

1 namespace std {
2   class contractViolation{
3     public:
4       uint_least32_t line_number() const noexcept;
5       string_view file_name() const noexcept;
6       string_view function_name() const noexcept;
7       string_view comment() const noexcept;
8       string_view assertion_level() const noexcept;
9   };
10 }
```

- line_number: 违反契约的行号
- file_name: 违反契约的文件名称
- function_name: 违反契约的函数名
- comment: 契约的谓词
- assertion_level: 契约的断言级别

8.2.1.3 契约的声明

契约可以放在函数的声明中，包括虚函数或函数模板的声明。

- 函数的契约声明必须相同，与第一个声明不同的声明都可以省略契约。

```

1 int f(int x)
```

```

1  [[expects: x > 0]]
2  [[ensures r: r > 0]];
3
4
5 int f(int x); // OK. No contract.
6
7 int f(int x)
8  [[expects: x >= 0]]; // Error missing ensures and different expects condition

```

- 重载函数不能修改契约。

```

1 struct B {
2     virtual void f(int x) [[expects: x > 0]];
3     virtual void g(int x);
4 };
5
6 struct D: B{
7     void f(int x) [[expects: x >= 0]]; // error
8     void g(int x) [[expects: x != 0]]; // error
9 };

```

D类契约的两个定义都是错误的。成员函数D::f的契约与B::f的契约不同，而成员函数D::g向B::g添加了一个契约。

结束语

契约是C++20的一部分，但目前至少推迟到了C++23。Herb Sutter关于[Sutter的话](#)可以让你了解了它们的重要性：“契约是迄今为止C++20中最有影响力特性，可以说是自C++11以来，最有影响力特性。”

8.2.2 反射

反射是程序分析和修改自身的可能性。反射发生在编译时，遵循C++的元规则：“不为不用的东西付费”。[type-trait库](#)是一个用于反射的强大工具，但是用于静态反射的提案[P0385](#)走得更远。

下面的代码片段应该会给你一个关于反射的直观印象：

反射操作符

```

1 template <typename T>
2 T min(const T& a, const T& b) {
3     log() << "function: min<" 
4         << get_base_name_v<get_aliased_t<$reflect(T)>>
5         << ">(" 
6         << get_base_name_v<$reflect(a)> << ":" 
7         << get_base_name_v<get_aliased_t<get_type_t<$reflect(a)>>>
8         << " = " << a << ", " 
9         << get_base_name_v<$reflect(b)> << ":" 
10        << get_base_name_v<get_aliased_t<get_type_t<$reflect(b)>>>
11        << " = " << b 
12        << ")" << '\n';

```

```
13     return a < b ? a : b;
14 }
```

反射操作符 \$reflect 是示例中的关键表达式。首先，new 操作符创建一个特殊的数据类型，它提供了关于模板参数 T(第 4 行) 和值 a(第 6 行) 和 c(第 9 行) 的元信息。由于函数组合，元信息可以用来提供更多信息: get_base_name_v<get_aliased_t...>(第 7 和第 10 行)。

当用参数 min(12.34, 23.45) 调用函数 min 时，会得到以下输出：

```
function: min<double>(a: double = 12.34, b: double = 23.45)
```

读者们可能会好奇：通过反射可以获得哪些元信息？以下几点可以给你答案：

- 对象：源代码行和列，以及文件名
- 类：私有和公共数据成员和成员函数
- 别名：已解析的别名

提案 P0385 的下一个示例展示了，反射如何帮助确定类的私有和公共成员。

```
1 #include <reflect>
2 #include <iostream>
3
4 struct foo {
5     private:
6     int _i, _j;
7     public:
8     static constexpr const bool b = true;
9     float x, y, z;
10    private:
11    static double d;
12 };
13
14 template <typename ... T>
15 void eat(T ... ) { }
16
17 template <typename Metaobjects, std::size_t I>
18 int do_print_data_member(void) {
19     using namespace std;
20     typedef reflect::get_element_t<Metaobjects, I> metaobj;
21     cout << I << ":" << (reflect::is_public_v<metaobj>?"public":"non-public")
22     << " "
23     << (reflect::is_static_v<metaobj>?"static":"")
24     << " "
25     << reflect::get_base_name_v<reflect::get_type_t<metaobj>>
26     << " "
27     << reflect::get_base_name_v<metaobj>
28     << '\n';
29 }
```

```

31    return 0;
32
33 template <typename Metaobjects, std::size_t ... I>
34 void do_print_data_members(std::index_sequence<I...>) {
35     eat(do_print_data_member<Metaobjects, I>() ...);
36 }
37
38 template <typename Metaobjects>
39 void do_print_data_members(void) {
40     using namespace std;
41
42     do_print_data_members<Metaobjects>(
43         make_index_sequence<
44             reflect::get_size_v<Metaobjects>
45         >()
46     );
47 }
48
49 template <typename MetaClass>
50 void print_data_members(void) {
51     using namespace std;
52     cout << "Public data members of " << reflect::get_base_name_v<MetaClass>
53         << '\n';
54
55     do_print_data_members<reflect::get_public_data_members_t<MetaClass>>();
56 }
57
58 template <typename MetaClass>
59 void print_all_data_members(void) {
60     using namespace std;
61
62     cout << "All data members of " << reflect::get_base_name_v<MetaClass>
63         << '\n';
64     do_print_data_members<reflect::get_data_members_t<MetaClass>>();
65 }
66
67 int main(void) {
68     print_data_members<$reflect(foo)>();
69     print_all_data_members<$reflect(foo)>();
70     return 0;
71 }
```

该程序产生以下输出:

```

    Public data members of foo
    0: public static bool b
    1: public float x
    2: public float y
    3: public float z
    All data members of foo
    0: non-public int _i
    1: non-public int _j
    2: public static bool b
    3: public float x
    4: public float y
    5: public float z
    6: non-public static double d

```

8.2.3 模式匹配

诸如`std::tuple`或`std::variant`这样的新数据类型需要新的方法来处理它们的元素。简单的`if`或`switch`条件或函数，如`std::apply`或`std::visit`只能提供基本功能。函数式编程中，大量使用的模式匹配以支持更强大的新数据类型处理。

下面的代码片段来自关于模式匹配的提案[P1371R2](#)，比较了经典的控制结构和模式匹配。模式匹配使用关键字`inspect`和`_`作为占位符。

- `switch`语句

```

1 switch (x) {
2     case 0: std::cout << "got zero"; break;
3     case 1: std::cout << "got one"; break;
4     default: std::cout << "don't care";
5 }
6
7 inspect (x) {
8     0: std::cout << "got zero";
9     1: std::cout << "got one";
10    _: std::cout << "don't care";
11 }

```

- `if`条件语句

```

1 if (s == "foo") {
2     std::cout << "got foo";
3 } else if (s == "bar") {
4     std::cout << "got bar";
5 } else {
6     std::cout << "don't care";
7 }
8
9 inspect (s) {
10    "foo": std::cout << "got foo";
11    "bar": std::cout << "got bar";

```

```
12     __: std::cout << "don't care";
13 }
```

模式匹配在 std::tuple、std::variant 或 polymorphism 上的应用证明了它的强大。

- std::tuple

```
1 auto&& [x, y] = p;
2 if (x == 0 && y == 0) {
3     std::cout << "on origin";
4 } else if (x == 0) {
5     std::cout << "on y-axis";
6 } else if (y == 0) {
7     std::cout << "on x-axis";
8 } else {
9     std::cout << x << ',' << y;
10 }
11
12 inspect (p) {
13     [0, 0]: std::cout << "on origin";
14     [0, y]: std::cout << "on y-axis";
15     [x, 0]: std::cout << "on x-axis";
16     [x, y]: std::cout << x << ',' << y;
17 }
```

- std::variant

```
1 struct visitor {
2     void operator()(int i) const {
3         os << "got int: " << i;
4     }
5     void operator()(float f) const {
6         os << "got float: " << f;
7     }
8     std::ostream& os;
9 };
10 std::visit(visitor{strm}, v);
11
12 inspect (v) {
13     <int> i: strm << "got int: " << i;
14     <float> f: strm << "got float: " << f;
15 }
```

- 多态数据类型

```
1 struct Shape { virtual ~Shape() = default; };
2 struct Circle : Shape { int radius; };
3 struct Rectangle : Shape { int width, height; };
4
5 virtual int Shape::get_area() const = 0;
6
7 int Circle::get_area() const override {
```

```
8     return 3.14 * radius * radius;
9 }
10 int Rectangle::get_area() const override {
11     return width * height;
12 }
13
14 int get_area(const Shape& shape) {
15     return inspect(shape) {
16         <Circle> [r] => 3.14 * r * r,
17         <Rectangle> [w, h] => w * h
18     }
19 }
```

关于模式匹配的提案 P1371R2 提供了更高级的用例。例如，模式匹配可用于遍历[表达式树](#)。

8.3. C++23 的更多信息

提案[P0592R4](#)只给出了 C++23 的一个粗略的概念，并集中在主要特性上。诸如[任务块](#)、[统一的 future](#)、[事务型内存](#)或[数据并行向量库](#)等支持[SIMD](#)的特性甚至都没有提及。当想更深入地了解 C++20 的未来时，必须去看看cppreference.com/compiler_support或阅读与 C++23 相关的[标准化委员会文件](#)。

第 9 章 功能测试

头文件 `<version>` 允许请求编译器提供 C++11 或更高版本的支持。可以查询核心语言或库的属性、特性。`<version>` 定义了大约 200 个宏，这些宏在实现特性时扩展为一个数字。这个数字代表该特性添加到 C++ 标准中的年份和月份。这些是 `static_assert`、`Lambda` 和概念的数字。

用于表示 `static_assert`、`Lambda` 和概念的宏

```
1 __cpp_static_assert 200410L
2 __cpp_lambdas 200907L
3 __cpp_concepts 201907L
```

支持的功能

当尝试全新的 C++ 特性时，我会检查哪个编译器实现了我感兴趣的特性。这是我访问 cppreference.com/compiler_support 的时候，搜索我想尝试的功能，并希望三大编译器 (GCC, Clang, MSVC) 中至少有一个实现了新功能。

只得到部分答案并不令人满意。最后，当一个全新功能的编译失败时，我也不知道该去找谁。

C++20 feature	Paper(s)	GCC	Clang	MSVC	Apple Clang	EDG eC++	Intel C++	IBM XLC++	Sun/Oracle C++	Embarcadero C++ Builder	Portland Group (PGI)	Cray	Nvidia nvcc	[Collapse]
Allow lambda-capture [=, this]	P0409R2	8	6	19.22*	10.0.0*	5.1								
<code>__VA_OPT__</code>	P0306R4 P1042R1	8 (partial)* 10 (partial)*	9	19.25*	11.0.3*	5.1								
Designated initializers	P0329R4	4.7 (partial)* 8 10	3.0 (partial)* 10	19.21*	(partial)*	5.1								
template-parameter-list for generic lambdas	P0428R2	8	9	19.22*	11.0.0*	5.1								
Default member initializers for bit-fields	P0683R1	8	6	19.25*	10.0.0*	5.1								
Initializer list constructors in class template argument deduction	P0702R1	8	6	19.14*	Yes	5.0								
const&-qualified pointers to members	P0704R1	8	6	19.0*	10.0.0*	5.1								
Concepts	P0734R0	6 (TS only) 10	10	19.23* (partial)*		6.1								
Lambdas in unevaluated contexts	P0315R4	9		19.28*										

支持 C++20 核心语言的特性

cppreference.com 的[功能测试](#)页面，在一个很长的源文件中使用了所有宏。

```
// featureTest.cpp
```

```

2 // from cppreference.com
3
4 #if __cplusplus < 201100
5 # error "C++11 or better is required"
6 #endif
7
8 #include <algorithm>
9 #include <cstring>
10 #include <iomanip>
11 #include <iostream>
12 #include <string>
13
14 #ifdef __has_include
15 # if __has_include(<version>)
16 # include <version>
17 # endif
18 #endif
19
20 #define COMPILER_FEATURE_VALUE(value) #value
21 #define COMPILER_FEATURE_ENTRY(name) { #name, COMPILER_FEATURE_VALUE(name) },
22
23 #ifdef __has_cpp_attribute
24 # define COMPILER_ATTRIBUTE_VALUE_AS_STRING(s) #s
25 # define COMPILER_ATTRIBUTE_AS_NUMBER(x) COMPILER_ATTRIBUTE_VALUE_AS_STRING(x)
26 # define COMPILER_ATTRIBUTE_ENTRY(attr) \
27     { #attr, COMPILER_ATTRIBUTE_AS_NUMBER(__has_cpp_attribute(attr)) },
28 #else
29 # define COMPILER_ATTRIBUTE_ENTRY(attr) { #attr, "_" },
30 #endif
31
32 // Change these options to print out only necessary info.
33 static struct PrintOptions {
34     constexpr static bool titles = 1;
35     constexpr static bool attributes = 1;
36     constexpr static bool general_features = 1;
37     constexpr static bool core_features = 1;
38     constexpr static bool lib_features = 1;
39     constexpr static bool supported_features = 1;
40     constexpr static bool unsupported_features = 1;
41     constexpr static bool sorted_by_value = 0;
42     constexpr static bool cxx11 = 1;
43     constexpr static bool cxx14 = 1;
44     constexpr static bool cxx17 = 1;
45     constexpr static bool cxx20 = 1;
46     constexpr static bool cxx23 = 0;
47 } print;
48
49 struct CompilerFeature {
50     CompilerFeature(const char* name = nullptr, const char* value = nullptr)

```

```

51     : name(name), value(value) {}
52     const char* name; const char* value;
53 };
54
55 static CompilerFeature cxx[] = {
56 COMPILER_FEATURE_ENTRY(__cplusplus)
57 COMPILER_FEATURE_ENTRY(__cpp_exceptions)
58 COMPILER_FEATURE_ENTRY(__cpp_rtti)
59 #if 0
60 COMPILER_FEATURE_ENTRY(__GNUC__)
61 COMPILER_FEATURE_ENTRY(__GNUC_MINOR__)
62 COMPILER_FEATURE_ENTRY(__GNUC_PATCHLEVEL__)
63 COMPILER_FEATURE_ENTRY(__GNUG__)
64 COMPILER_FEATURE_ENTRY(__clang__)
65 COMPILER_FEATURE_ENTRY(__clang_major__)
66 COMPILER_FEATURE_ENTRY(__clang_minor__)
67 COMPILER_FEATURE_ENTRY(__clang_patchlevel__)
68 #endif
69 };
70 static CompilerFeature cxx11[] = {
71 COMPILER_FEATURE_ENTRY(__cpp_alias_templates)
72 COMPILER_FEATURE_ENTRY(__cpp_attributes)
73 COMPILER_FEATURE_ENTRY(__cpp_constexpr)
74 COMPILER_FEATURE_ENTRY(__cpp_decltype)
75 COMPILER_FEATURE_ENTRY(__cpp_delegating_constructors)
76 COMPILER_FEATURE_ENTRY(__cpp_inheriting_constructors)
77 COMPILER_FEATURE_ENTRY(__cpp_initializer_lists)
78 COMPILER_FEATURE_ENTRY(__cpp_lambdas)
79 COMPILER_FEATURE_ENTRY(__cpp_nsdimi)
80 COMPILER_FEATURE_ENTRY(__cpp_range_based_for)
81 COMPILER_FEATURE_ENTRY(__cpp_raw_strings)
82 COMPILER_FEATURE_ENTRY(__cpp_ref_qualifiers)
83 COMPILER_FEATURE_ENTRY(__cpp_rvalue_references)
84 COMPILER_FEATURE_ENTRY(__cpp_static_assert)
85 COMPILER_FEATURE_ENTRY(__cpp_threadsafe_static_init)
86 COMPILER_FEATURE_ENTRY(__cpp_unicode_characters)
87 COMPILER_FEATURE_ENTRY(__cpp_unicode_literals)
88 COMPILER_FEATURE_ENTRY(__cpp_user_defined_literals)
89 COMPILER_FEATURE_ENTRY(__cpp_variadic_templates)
90 };
91 static CompilerFeature cxx14[] = {
92 COMPILER_FEATURE_ENTRY(__cpp_aggregate_nsdimi)
93 COMPILER_FEATURE_ENTRY(__cpp_binary_literals)
94 COMPILER_FEATURE_ENTRY(__cpp_constexpr)
95 COMPILER_FEATURE_ENTRY(__cpp_decltype_auto)
96 COMPILER_FEATURE_ENTRY(__cpp_generic_lambdas)
97 COMPILER_FEATURE_ENTRY(__cpp_init_captures)
98 COMPILER_FEATURE_ENTRY(__cpp_return_type_deduction)
99 COMPILER_FEATURE_ENTRY(__cpp_sized_deallocation)

```

```

100 COMPILER_FEATURE_ENTRY(__cpp_variable_templates)
101 };
102 static CompilerFeature cxx14lib[] = {
103     COMPILER_FEATURE_ENTRY(__cpp_lib_chrono_udls)
104     COMPILER_FEATURE_ENTRY(__cpp_lib_complex_udls)
105     COMPILER_FEATURE_ENTRY(__cpp_lib_exchange_function)
106     COMPILER_FEATURE_ENTRY(__cpp_lib_generic_associative_lookup)
107     COMPILER_FEATURE_ENTRY(__cpp_lib_integer_sequence)
108     COMPILER_FEATURE_ENTRY(__cpp_lib_integral_constant_callable)
109     COMPILER_FEATURE_ENTRY(__cpp_lib_is_final)
110     COMPILER_FEATURE_ENTRY(__cpp_lib_is_null_pointer)
111     COMPILER_FEATURE_ENTRY(__cpp_lib_make_reverse_iterator)
112     COMPILER_FEATURE_ENTRY(__cpp_lib_make_unique)
113     COMPILER_FEATURE_ENTRY(__cpp_lib_null_iterators)
114     COMPILER_FEATURE_ENTRY(__cpp_lib_quoted_string_io)
115     COMPILER_FEATURE_ENTRY(__cpp_lib_result_of_sfinae)
116     COMPILER_FEATURE_ENTRY(__cpp_lib_robust_nonmodifying_seq_ops)
117     COMPILER_FEATURE_ENTRY(__cpp_lib_shared_timed_mutex)
118     COMPILER_FEATURE_ENTRY(__cpp_lib_string_udls)
119     COMPILER_FEATURE_ENTRY(__cpp_lib_transformation_trait_aliases)
120     COMPILER_FEATURE_ENTRY(__cpp_lib_transparent_operators)
121     COMPILER_FEATURE_ENTRY(__cpp_lib_tuple_element_t)
122     COMPILER_FEATURE_ENTRY(__cpp_lib_tuples_by_type)
123 };
124
125 static CompilerFeature cxx17[] = {
126     COMPILER_FEATURE_ENTRY(__cpp_aggregate_bases)
127     COMPILER_FEATURE_ENTRY(__cpp_aligned_new)
128     COMPILER_FEATURE_ENTRY(__cpp_capture_star_this)
129     COMPILER_FEATURE_ENTRY(__cpp_constexpr)
130     COMPILER_FEATURE_ENTRY(__cpp_deduction_guides)
131     COMPILER_FEATURE_ENTRY(__cpp_enumerator_attributes)
132     COMPILER_FEATURE_ENTRY(__cpp_fold_expressions)
133     COMPILER_FEATURE_ENTRY(__cpp_guaranteed_copy_elision)
134     COMPILER_FEATURE_ENTRY(__cpp_hex_float)
135     COMPILER_FEATURE_ENTRY(__cpp_if_constexpr)
136     COMPILER_FEATURE_ENTRY(__cpp_inheriting_constructors)
137     COMPILER_FEATURE_ENTRY(__cpp_inline_variables)
138     COMPILER_FEATURE_ENTRY(__cpp_namespace_attributes)
139     COMPILER_FEATURE_ENTRY(__cpp_noexcept_function_type)
140     COMPILER_FEATURE_ENTRY(__cpp_nontype_template_args)
141     COMPILER_FEATURE_ENTRY(__cpp_nontype_template_parameter_auto)
142     COMPILER_FEATURE_ENTRY(__cpp_range_based_for)
143     COMPILER_FEATURE_ENTRY(__cpp_static_assert)
144     COMPILER_FEATURE_ENTRY(__cpp_structured_bindings)
145     COMPILER_FEATURE_ENTRY(__cpp_template_template_args)
146     COMPILER_FEATURE_ENTRY(__cpp_variadic_using)
147 };
148 static CompilerFeature cxx17lib[] = {

```

```
149 COMPILER_FEATURE_ENTRY(__cpp_lib_addressof_constexpr)
150 COMPILER_FEATURE_ENTRY(__cpp_lib_allocator_traits_is_always_equal)
151 COMPILER_FEATURE_ENTRY(__cpp_lib_any)
152 COMPILER_FEATURE_ENTRY(__cpp_lib_apply)
153 COMPILER_FEATURE_ENTRY(__cpp_lib_array_constexpr)
154 COMPILER_FEATURE_ENTRY(__cpp_lib_as_const)
155 COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_is_always_lock_free)
156 COMPILER_FEATURE_ENTRY(__cpp_lib_bool_constant)
157 COMPILER_FEATURE_ENTRY(__cpp_lib_boyer_moore_searcher)
158 COMPILER_FEATURE_ENTRY(__cpp_lib_byte)
159 COMPILER_FEATURE_ENTRY(__cpp_lib_chrono)
160 COMPILER_FEATURE_ENTRY(__cpp_lib_clamp)
161 COMPILER_FEATURE_ENTRY(__cpp_lib_enable_shared_from_this)
162 COMPILER_FEATURE_ENTRY(__cpp_lib_execution)
163 COMPILER_FEATURE_ENTRY(__cpp_lib_filesystem)
164 COMPILER_FEATURE_ENTRY(__cpp_lib_gcd_lcm)
165 COMPILER_FEATURE_ENTRY(__cpp_lib_hardware_interference_size)
166 COMPILER_FEATURE_ENTRY(__cpp_lib_has_unique_object_representations)
167 COMPILER_FEATURE_ENTRY(__cpp_lib_hypot)
168 COMPILER_FEATURE_ENTRY(__cpp_lib_incomplete_container_elements)
169 COMPILER_FEATURE_ENTRY(__cpp_lib_invoke)
170 COMPILER_FEATURE_ENTRY(__cpp_lib_is_aggregate)
171 COMPILER_FEATURE_ENTRY(__cpp_lib_is_invocable)
172 COMPILER_FEATURE_ENTRY(__cpp_lib_is_swappable)
173 COMPILER_FEATURE_ENTRY(__cpp_lib_launder)
174 COMPILER_FEATURE_ENTRY(__cpp_lib_logical_traits)
175 COMPILER_FEATURE_ENTRY(__cpp_lib_make_from_tuple)
176 COMPILER_FEATURE_ENTRY(__cpp_lib_map_try_emplace)
177 COMPILER_FEATURE_ENTRY(__cpp_lib_math_special_functions)
178 COMPILER_FEATURE_ENTRY(__cpp_lib_memory_resource)
179 COMPILER_FEATURE_ENTRY(__cpp_lib_node_extract)
180 COMPILER_FEATURE_ENTRY(__cpp_lib_nonmember_container_access)
181 COMPILER_FEATURE_ENTRY(__cpp_lib_not_fn)
182 COMPILER_FEATURE_ENTRY(__cpp_lib_optional)
183 COMPILER_FEATURE_ENTRY(__cpp_lib_parallel_algorithm)
184 COMPILER_FEATURE_ENTRY(__cpp_lib_raw_memory_algorithms)
185 COMPILER_FEATURE_ENTRY(__cpp_lib_sample)
186 COMPILER_FEATURE_ENTRY(__cpp_lib_scoped_lock)
187 COMPILER_FEATURE_ENTRY(__cpp_lib_shared_mutex)
188 COMPILER_FEATURE_ENTRY(__cpp_lib_shared_ptr_arrays)
189 COMPILER_FEATURE_ENTRY(__cpp_lib_shared_ptr_weak_type)
190 COMPILER_FEATURE_ENTRY(__cpp_lib_string_view)
191 COMPILER_FEATURE_ENTRY(__cpp_lib_to_chars)
192 COMPILER_FEATURE_ENTRY(__cpp_lib_transparent_operators)
193 COMPILER_FEATURE_ENTRY(__cpp_lib_type_trait_variable_templates)
194 COMPILER_FEATURE_ENTRY(__cpp_lib_uncaught_exceptions)
195 COMPILER_FEATURE_ENTRY(__cpp_lib_unordered_map_try_emplace)
196 COMPILER_FEATURE_ENTRY(__cpp_lib_variant)
197 COMPILER_FEATURE_ENTRY(__cpp_lib_void_t)
```

```
198 } ;
199
200 static CompilerFeature cxx20[] = {
201 COMPILER_FEATURE_ENTRY(__cpp_aggregate_paren_init)
202 COMPILER_FEATURE_ENTRY(__cpp_char8_t)
203 COMPILER_FEATURE_ENTRY(__cpp_concepts)
204 COMPILER_FEATURE_ENTRY(__cpp_conditional_explicit)
205 COMPILER_FEATURE_ENTRY(__cpp_consteval)
206 COMPILER_FEATURE_ENTRY(__cpp_constexpr)
207 COMPILER_FEATURE_ENTRY(__cpp_constexpr_dynamic_alloc)
208 COMPILER_FEATURE_ENTRY(__cpp_constexpr_in_decltype)
209 COMPILER_FEATURE_ENTRY(__cpp_constinit)
210 COMPILER_FEATURE_ENTRY(__cpp_deduction_guides)
211 COMPILER_FEATURE_ENTRY(__cpp_designated_initializers)
212 COMPILER_FEATURE_ENTRY(__cpp_generic_lambdas)
213 COMPILER_FEATURE_ENTRY(__cpp_impl_coroutine)
214 COMPILER_FEATURE_ENTRY(__cpp_impl_destroying_delete)
215 COMPILER_FEATURE_ENTRY(__cpp_impl_three_way_comparison)
216 COMPILER_FEATURE_ENTRY(__cpp_init_captures)
217 COMPILER_FEATURE_ENTRY(__cpp_modules)
218 COMPILER_FEATURE_ENTRY(__cpp_nontype_template_args)
219 COMPILER_FEATURE_ENTRY(__cpp_using_enum)
220 };
221 static CompilerFeature cxx20lib[] = {
222 COMPILER_FEATURE_ENTRY(__cpp_lib_array_constexpr)
223 COMPILER_FEATURE_ENTRY(__cpp_lib_assume_aligned)
224 COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_flag_test)
225 COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_float)
226 COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_lock_free_type_aliases)
227 COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_ref)
228 COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_shared_ptr)
229 COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_value_initialization)
230 COMPILER_FEATURE_ENTRY(__cpp_lib_atomic_wait)
231 COMPILER_FEATURE_ENTRY(__cpp_lib_barrier)
232 COMPILER_FEATURE_ENTRY(__cpp_lib_bind_front)
233 COMPILER_FEATURE_ENTRY(__cpp_lib_bit_cast)
234 COMPILER_FEATURE_ENTRY(__cpp_lib_bitops)
235 COMPILER_FEATURE_ENTRY(__cpp_lib_bounded_array_traits)
236 COMPILER_FEATURE_ENTRY(__cpp_lib_char8_t)
237 COMPILER_FEATURE_ENTRY(__cpp_lib_chrono)
238 COMPILER_FEATURE_ENTRY(__cpp_lib_concepts)
239 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_algorithms)
240 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_complex)
241 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_dynamic_alloc)
242 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_functional)
243 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_iterator)
244 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_memory)
245 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_numeric)
246 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_string)
```

```

247 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_string_view)
248 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_tuple)
249 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_utility)
250 COMPILER_FEATURE_ENTRY(__cpp_lib_constexpr_vector)
251 COMPILER_FEATURE_ENTRY(__cpp_lib_coroutine)
252 COMPILER_FEATURE_ENTRY(__cpp_lib_destroying_delete)
253 COMPILER_FEATURE_ENTRY(__cpp_lib_endian)
254 COMPILER_FEATURE_ENTRY(__cpp_lib_erase_if)
255 COMPILER_FEATURE_ENTRY(__cpp_lib_execution)
256 COMPILER_FEATURE_ENTRY(__cpp_lib_format)
257 COMPILER_FEATURE_ENTRY(__cpp_lib_generic_unordered_lookup)
258 COMPILER_FEATURE_ENTRY(__cpp_lib_int_pow2)
259 COMPILER_FEATURE_ENTRY(__cpp_lib_integer_comparison_functions)
260 COMPILER_FEATURE_ENTRY(__cpp_lib_interpolate)
261 COMPILER_FEATURE_ENTRY(__cpp_lib_is_constant_evaluated)
262 COMPILER_FEATURE_ENTRY(__cpp_lib_is_layout_compatible)
263 COMPILER_FEATURE_ENTRY(__cpp_lib_is_nothrow_convertible)
264 COMPILER_FEATURE_ENTRY(__cpp_lib_is_pointer_interconvertible)
265 COMPILER_FEATURE_ENTRY(__cpp_lib_jthread)
266 COMPILER_FEATURE_ENTRY(__cpp_lib_latch)
267 COMPILER_FEATURE_ENTRY(__cpp_lib_list_remove_return_type)
268 COMPILER_FEATURE_ENTRY(__cpp_lib_math_constants)
269 COMPILER_FEATURE_ENTRY(__cpp_lib_polymorphic_allocator)
270 COMPILER_FEATURE_ENTRY(__cpp_lib_ranges)
271 COMPILER_FEATURE_ENTRY(__cpp_lib_remove_cvref)
272 COMPILER_FEATURE_ENTRY(__cpp_lib_semaphore)
273 COMPILER_FEATURE_ENTRY(__cpp_lib_shared_ptr_arrays)
274 COMPILER_FEATURE_ENTRY(__cpp_lib_shift)
275 COMPILER_FEATURE_ENTRY(__cpp_lib_smart_ptr_for_overwrite)
276 COMPILER_FEATURE_ENTRY(__cpp_lib_source_location)
277 COMPILER_FEATURE_ENTRY(__cpp_lib_span)
278 COMPILER_FEATURE_ENTRY(__cpp_lib_ssize)
279 COMPILER_FEATURE_ENTRY(__cpp_lib_starts_ends_with)
280 COMPILER_FEATURE_ENTRY(__cpp_lib_string_view)
281 COMPILER_FEATURE_ENTRY(__cpp_lib_syncbuf)
282 COMPILER_FEATURE_ENTRY(__cpp_lib_three_way_comparison)
283 COMPILER_FEATURE_ENTRY(__cpp_lib_to_address)
284 COMPILER_FEATURE_ENTRY(__cpp_lib_to_array)
285 COMPILER_FEATURE_ENTRY(__cpp_lib_type_identity)
286 COMPILER_FEATURE_ENTRY(__cpp_lib_unwrap_ref)
287 };
288
289 static CompilerFeature cxx23[] = {
290 COMPILER_FEATURE_ENTRY(__cpp_cxx23_stub) //< Populate eventually
291 };
292 static CompilerFeature cxx23lib[] = {
293 COMPILER_FEATURE_ENTRY(__cpp_lib_cxx23_stub) //< Populate eventually
294 };
295

```

```

296 static CompilerFeature attributes[] = {
297 COMPILER_ATTRIBUTE_ENTRY(carries_dependency)
298 COMPILER_ATTRIBUTE_ENTRY(deprecated)
299 COMPILER_ATTRIBUTE_ENTRY(fallthrough)
300 COMPILER_ATTRIBUTE_ENTRY(likely)
301 COMPILER_ATTRIBUTE_ENTRY(maybe_unused)
302 COMPILER_ATTRIBUTE_ENTRY(nodiscard)
303 COMPILER_ATTRIBUTE_ENTRY(noreturn)
304 COMPILER_ATTRIBUTE_ENTRY(no_unique_address)
305 COMPILER_ATTRIBUTE_ENTRY(unlikely)
306 };
307
308 constexpr bool is_feature_supported(const CompilerFeature& x) {
309     return x.value[0] != '_' && x.value[0] != '0' ;
310 }
311
312 inline void print_compiler_feature(const CompilerFeature& x) {
313     constexpr static int max_name_length = 44; //< Update if necessary
314     std::string value{ is_feature_supported(x) ? x.value : "-----" };
315     if (value.back() == 'L') value.pop_back(); //~ 201603L -> 201603
316     // value.insert(4, 1, '-'); //~ 201603 -> 2016-03
317     if ( (print.supported_features && is_feature_supported(x))
318         || (print.unsupported_features && !is_feature_supported(x))) {
319         std::cout << std::left << std::setw(max_name_length)
320             << x.name << " " << value << '\n';
321     }
322 }
323
324 template<size_t N>
325 inline void show(char const* title, CompilerFeature (&features)[N]) {
326     if (print.titles) {
327         std::cout << '\n' << std::left << title << '\n';
328     }
329     if (print.sorted_by_value) {
330         std::sort(std::begin(features), std::end(features),
331                 [] (CompilerFeature const& lhs, CompilerFeature const& rhs) {
332                     return std::strcmp(lhs.value, rhs.value) < 0;
333                 });
334     }
335     for (const CompilerFeature& x : features) {
336         print_compiler_feature(x);
337     }
338 }
339
340 int main() {
341     if (print.general_features) show("C++ GENERAL", cxx);
342     if (print.cxx11 && print.core_features) show("C++11 CORE", cxx11);
343     if (print.cxx14 && print.core_features) show("C++14 CORE", cxx14);
344     if (print.cxx14 && print.lib_features ) show("C++14 LIB" , cxx14lib);

```

```

345     if (print.cxx17 && print.core_features) show("C++17 CORE", cxx17);
346     if (print.cxx17 && print.lib_features ) show("C++17 LIB" , cxx17lib);
347     if (print.cxx20 && print.core_features) show("C++20 CORE", cxx20);
348     if (print.cxx20 && print.lib_features ) show("C++20 LIB" , cxx20lib);
349     if (print.cxx23 && print.core_features) show("C++23 CORE", cxx23);
350     if (print.cxx23 && print.lib_features ) show("C++23 LIB" , cxx23lib);
351     if (print.attributes) show("ATTRIBUTES", attributes);
352 }

```

当然，源文件太长了。若想了解更多关于每个宏的信息，请访问[特性测试](#)页面。特别是，该页面为每个宏提供了一个链接，以便您可以获得关于某个特性的更多信息。下面是属性表：

<i>attribute-token</i>	Attribute	Value	Standard
<code>carries_dependency</code>	<code>[[carries_dependency]]</code>	<code>200809L</code>	(C++11)
<code>deprecated</code>	<code>[[deprecated]]</code>	<code>201309L</code>	(C++14)
<code>fallthrough</code>	<code>[[fallthrough]]</code>	<code>201603L</code>	(C++17)
<code>likely</code>	<code>[[likely]]</code>	<code>201803L</code>	(C++20)
<code>maybe_unused</code>	<code>[[maybe_unused]]</code>	<code>201603L</code>	(C++17)
<code>no_unique_address</code>	<code>[[no_unique_address]]</code>	<code>201803L</code>	(C++20)
<code>nodiscard</code>	<code>[[nodiscard]]</code>	<code>201603L</code>	(C++17)
		<code>201907L</code>	(C++20)
<code>noreturn</code>	<code>[[noreturn]]</code>	<code>200809L</code>	(C++11)
<code>unlikely</code>	<code>[[unlikely]]</code>	<code>201803L</code>	(C++20)

属性的宏

下面是<version>标头及其宏的演示。我在全新的GCC、Clang和MSVC编译器上执行了这个程序，为GCC和Clang编译器使用了编译器资源管理器。`/Zc:_cplusplus`标志允许`_cplusplus`宏报告最新的C++语言标准支持。此外，我在所有三个平台上启用了C++20支持。这里，我只展示了对C++20核心语言的支持。

GCC 10.2

C++20 CORE	
__cpp_aggregate_paren_init	201902
__cpp_char8_t	201811
__cpp_concepts	201907
__cpp_conditional_explicit	201806
__cpp_consteval	-----
__cpp_constexpr	201907
__cpp_constexpr_dynamic_alloc	201907
__cpp_constexpr_in_decltype	201711
__cpp_constinit	201907
__cpp_deduction_guides	201907
__cpp_designated_initializers	201707
__cpp_generic_lambdas	201707
__cpp_implementation_coroutine	-----
__cpp_implementation_destroying_delete	201806
__cpp_implementation_three_way_comparison	201907
__cpp_init_captures	201803
__cpp_modules	-----
__cpp_nontype_template_args	201411
__cpp_using_enum	-----

Clang 11.0

C++20 CORE	
__cpp_aggregate_paren_init	-----
__cpp_char8_t	201811
__cpp_concepts	201907
__cpp_conditional_explicit	201806
__cpp_consteval	-----
__cpp_constexpr	201907
__cpp_constexpr_dynamic_alloc	201907
__cpp_constexpr_in_decltype	201711
__cpp_constinit	201907
__cpp_deduction_guides	201703
__cpp_designated_initializers	201707
__cpp_generic_lambdas	201707
__cpp_implementation_coroutine	-----
__cpp_implementation_destroying_delete	201806
__cpp_implementation_three_way_comparison	201907
__cpp_init_captures	201803
__cpp_modules	-----
__cpp_nontype_template_args	201411
__cpp_using_enum	-----

MSVC 19.27

The screenshot shows a command prompt window titled "x64 Native Tools Command Prompt for VS 2019". The window displays a list of C++20 features and their corresponding implementation dates. The list is as follows:

C++20 CORE	
__cpp_aggregate_paren_init	-----
__cpp_char8_t	201811
__cpp_concepts	201811
__cpp_conditional_explicit	201806
__cpp_consteval	-----
__cpp_constexpr	201603
__cpp_constexpr_dynamic_alloc	-----
__cpp_constexpr_in_decltype	-----
__cpp_constinit	-----
__cpp_deduction_guides	201907
__cpp_designated_initializers	201707
__cpp_generic_lambdas	201707
__cpp_impl_coroutine	-----
__cpp_impl_destroying_delete	201806
__cpp_impl_three_way_comparison	201907
__cpp_init_captures	201803
__cpp_modules	-----
__cpp_nontype_template_args	201911
__cpp_using_enum	201907

这三张截图清楚地传达了这三巨头的信息: 他们的 C++20 的核心语言支持, 在 2020 年底时还是相当不错的。

第 10 章 术语表

本术语表只作为参考。

10.1. 可调用

参见可调用单元

10.2. 可调用单元

可调用单元(简称可调用)的行为类似于函数。这些不仅可命名为函数，还可命名为函数对象或 Lambda 表达式。若可调用对象接受一个参数，称为一元可调用对象；若有两个参数，称为二元可调用对象。

谓词是会返回布尔值作为结果的特殊可调用对象(单元)。

10.3. 并发性

并发性指同时执行多个任务的能力，并发性是并行性的超集。

10.4. 临界区

临界区是包含共享变量的代码段，必须对其进行保护以避免数据竞争。在同一时间点，只有一个线程可以进入临界区。

10.5. 数据竞争

数据竞争是指至少有两个线程同时访问一个共享变量的情况。至少有一个线程尝试修改变量，而另一个线程尝试读取或修改变量。若的程序存在数据竞争，则该程序具有未定义的行为，所以任何结果都是可能的。

10.6. 死锁

死锁是一种状态，在这种状态下，至少有一个线程永远阻塞，因为它等待无法释放的资源。

造成死锁的主要原因有两个：

1. 互斥对象未解锁。
2. 以错误的顺序锁定互斥对象。

10.7. 立即求值

立即求值的情况下，表达式将进行阻塞求值。这种求值策略与惰性求值相反。立即求值也称为贪婪求值。

10.8. 执行器

执行器是与特定执行上下文相关联的对象，提供了一个或多个执行函数，用于从可调用函数对象创建执行代理。

10.9. 函数对象

首先，它们不叫[函数](#)。这是一个定义明确的术语，来自数学的一个分支，叫做[category 理论](#)。

函数对象是行为类似于函数的对象，通过实现函数调用操作符来实现。由于函数对象是对象，若依具有属性，也可以具有状态。

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

实例化函数对象

算法中使用函数对象 (Square) 的名称而不是函数对象 (Square()) 本身实例是一个常见的错误:std::for_each(myVec.begin(), myVec.end(), Square)。必须使用实例:std::for_each(myVec.begin(), myVec.end(), Square())

10.10. Lambda 表达式

Lambda 表达式就地提供功能，编译器获取所有必要的信息以优化代码。Lambda 函数可以通过值或引用接收参数，也可以通过值或引用捕获其定义环境的变量。

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

10.11. 延迟(惰性)求值

对于[延迟\(惰性\)求值](#)，只在需要时计算表达式。这种求值策略与立即求值是相反的。延迟(惰性)求值常称为按需调用。

10.12. 无锁

若能在保证系统范围内的正确运行，则非阻塞算法是无锁的。

10.13. 错过唤醒

错过唤醒是指线程由于竞态条件，而错过唤醒通知的情况。

10.14. 数学规律

某个集合 X 上的二元操作 (*) 是

- 结合律，若满足 x 中所有 x, y, z 的结合律: $(x * y) * z = x * (y * z)$
- 交换律，若对 x 中的所有 x, y 都满足交换律: $x * y = y * x$
- 分配律，若满足 x 中所有 x, y, z 的分配律: $x(y + z) = xy + xz$

10.15. 内存位置

内存位置的信息: cppreference.com

- 标量类型的对象 (算术类型、指针类型、枚举类型或 `std::nullptr_t`)，
- 或长度非零的最大连续位域序列。

10.16. 内存模型

内存模型定义对象和内存位置间的关系，并处理以下问题：若两个线程访问相同的内存位置会发生什么？

10.17. 非阻塞

若线程的失败或挂起不会导致另一个线程的失败或挂起，则该算法称为非阻塞算法。这个定义来自[Java 并发实践](#)。

10.18. 对象

若类型是标量、数组、联合或类，则它是对象。

10.19. 并行性

并行性指同时执行多个任务，并行是并发的一个子集。与并发性相反，平行性需要多核。

10.20. 谓词

谓词是返回布尔值的可调用单元。若谓词有一个参数，称为一元谓词。若谓词有两个参数，称为二元谓词。

10.21. RAI(资源获取即初始化)

资源获取即初始化，简称 RAI，是 C++ 中的一种技术，资源获取和释放与对象的生命周期绑定在一起。所以对于锁，互斥锁将在构造函数中锁定，在析构函数中解锁。

C++ 中的常见用例是处理底层互斥锁的生命周期的锁，处理其资源（内存）的生命周期的智能指针，或者处理其元素生命周期的[标准模板库容器](#)。

10.22. 竞态条件

竞态条件是一种情况，其操作的结果取决于某些单独操作的交错（操作的顺序）。

竞态条件很难发现，是否发生取决于线程的交错。内核数量、系统利用率或可执行文件的优化级别，都可能是竞态条件出现或不出现的原因。

10.23. 常规类型

除了半常规类型概念的要求外，常规概念还要求类型具有相等的可比性。

10.24. 标量

标量类型可以是算术类型 ([std::is_arithmetic](#))、枚举、指针、成员指针或 std::nullptr_t。

10.25. 半常规类型

半常规型对象 X 必须支持六大函数，并且必须是可交换的: swap(X&, X&)

10.26. 伪唤醒

伪唤醒是一种错误的通知。条件变量或原子标志的等待组件可以获得通知，但通知组件没有发送信号。

10.27. 四大特性

C++20 的四个关键特性: 概念、模块、范围库和协程。

- 概念改变了我们思考和使用模板编程的方式，是模板参数的语义类别。能够在类型系统中直接表达开发者意图。若出现错误，编译器会给出一个明确的错误消息。
- 模块克服了头文件的限制。例如，头文件和源文件的分离就像预处理程序一样过时了。最后，有了更快的构建时间和更简单的构建包的方法。
- 范围库支持直接在容器上执行算法，使用管道符号组合算法，以及可在无限数据流上惰性地应用算法。
- 协程的出现使得异步编程成为 C++ 的主流。协程是协作任务、事件循环、无限数据流或管道的基础。

10.28. 六大函数

六大函数:

- 默认构造函数:

```
1 X()
```

- 复制构造函数:

```
1 X(const X&)
```

- 复制赋值操作符:

```
1 X& operator = (const X&)
```

- 移动构造函数:

```
1 X(X&&)
```

- 移动赋值操作符:

```
1 X& operator = (X&&)
```

- 析构函数:

```
1 ~X()
```

10.29. 线程

在计算机科学中，执行线程是调度程序可以独立管理的编程指令的最小序列，通常是操作系统的一部分。线程和进程的实现因操作系统而异，但在大多数情况下，线程是进程组件。一个进程中可以存在多个线程，并发执行并共享内存等资源，而不同的进程不共享这些资源。有关详细信息，请阅读维基百科关于[线程](#)的文章。

10.30. 时间复杂度

$O(i)$ 表示操作的时间复杂度(运行时间)。对于 $O(1)$ ，容器上操作的运行时间是常数，因此与容器的大小无关。相反， $O(n)$ 表示运行时间线性地依赖于容器元素的数量。

10.31. 翻译单元

翻译单元是经过 C 预处理器处理后的源文件。C 预处理器使用 `#include` 指令包含头文件，使用 `#ifdef` 或 `#ifndef` 等指令执行条件包含，并展开宏。编译器使用翻译单元创建一个目标文件。

10.32. 未定义行为

未定义行为在程序中，可能产生正确的结果，也可能产生错误的结果，可能在运行时崩溃，甚至可能无法编译。当移植到一个新的平台，升级到一个新的编译器，或者由于不相关的代码更改，行为可能会发生变化。