Contents

Part I

基本语言特性

这一部分介绍了 C++17 中新的核心语言特性，但不包括那些专为泛型编程（即 template）设计的特性。这些新增的特性对于应用程序员的日常编程非常有用，因此每一个使用 C++17 的 C++ 程序员都应该了解它们。

专为模板编程设计的新的核心语言特性在Part II中介绍。

Chapter 1

结构化绑定

结构化绑定允许你用一个对象的元素或成员同时实例化多个实体。例如，假设你定义了一个有两个不同成员的结构体：

```
struct MyStruct {
    int i = 0;
    std::string s;
};

MyStruct ms;
```

你可以通过如下的声明直接把该结构体的两个成员绑定到新的变量名：

```
auto [u, v] = ms;
```

这里，变量 u 和 v 的声明方式称为*结构化绑定*。某种程度上可以说它们解构了用来初始化的对象（在某些地方它们被称为*解构声明*）。

如下的每一种声明方式都是支持的：

```
auto [u2, v2] {ms};
auto [u3, v3] (ms);
```

结构化绑定对于返回结构体或者数组的函数来说非常有用。例如，考虑一个返回结构体的函数：

```
MyStruct getStruct() {
    return MyStruct{42, "hello"};
}
```

你可以直接把返回的数据成员赋值给两个新的局部变量：

```
auto [id, val] = getStruct();  // id和val分别是返回结构体的i和s成员
```

这个例子中，id 和 val 分别是返回结构体中的 i 和 s 成员。它们的类型分别是 int 和 std::string，可以被当作两个不同的对象来使用：

```
if (id > 30) {
    std::cout << val;
}
```

这么做的好处是可以直接访问成员，另外，把值绑定到能体现语义的变量名上，可以使代码的可读性更强。[1]

下面的代码演示了使用结构化绑定带来的显著改进。在不使用结构化绑定的情况下遍历 `std::map<>` 的元素需要这么写：

```
for (const auto& elem : mymap) {
    std::cout << elem.first << ": " << elem.second << '\n';
}
```

map 的元素的类型是键和值组成的 `std::pair` 类型，`std::pair` 的成员分别是 `first` 和 `second`，上边的例子中必须使用成员的名字来访问键和值。通过使用结构化绑定，代码的可读性大大提升：

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

上面的例子中我们可以使用准确体现语义的变量名直接访问每一个元素。

1.1 细说结构化绑定

为了理解结构化绑定，必须意识到这里面其实有一个隐藏的匿名对象。结构化绑定时新引入的局部变量名其实都指向这个匿名对象的成员/元素。

绑定到一个匿名实体

如下代码的精确行为：

```
auto [u, v] = ms;
```

等价于我们用 `ms` 初始化了一个新的实体 `e`，并且让结构化绑定中的 `u` 和 `v` 变成了 `e` 的成员的别名，类似于如下定义：

```
auto e = ms;
aliasname u = e.i;
aliasname v = e.s;
```

这意味着 `u` 和 `v` 仅仅是 `ms` 的一份本地拷贝的成员的别名。然而，我们没有为 `e` 声明一个名称，因此我们不能直接访问这个匿名对象。注意 `u` 和 `v` 并不是 `e.i` 和 `e.s` 的引用（而是它们的别名）。`decltype(u)` 的结果是成员 `i` 的类型，`declytpe(v)` 的结果是成员 `s` 的类型。因此：

```
std::cout << u << ' ' << v << '\n';
```

会打印出 `e.i` 和 `e.s`（分别是 `ms.i` 和 `ms.s` 的拷贝）。

`e` 的生命周期和结构化绑定的生命周期相同，当结构化绑定离开作用域时 `e` 也会被自动销毁。另外，除非使用了引用，否则修改结构化绑定的变量并不会影响被绑定的变量：

```
MyStruct ms{42, "hello"};
auto [u, v] = ms;
ms.i = 77;
```

[1] 感谢 Zachary Turner 指出这一点

```
std::cout << u;     // 打印出42
u = 99;
std::cout << ms.i;  // 打印出77
```

在这个例子中u和ms.i有不同的内存地址。

当使用结构化绑定来绑定返回值时，规则是相同的。如下初始化

```
auto [u, v] = getStruct();
```

的行为等价于我们用getStruct()的返回值初始化了一个新的实体e，之后结构化绑定的变量u和v变成了e的两个成员的别名，类似于如下定义：

```
auto e = getStruct();
aliasname u = e.i;
aliasname v = e.s;
```

也就是说，结构化绑定绑定到了一个新的实体e上，而不是直接绑定到了返回值上。匿名实体e同样遵循通常的内存对齐规则，结构化绑定的每一个变量都会根据相应成员的类型进行对齐。

使用修饰符

我们可以在结构化绑定中使用修饰符，例如const和引用，这些修饰符会作用在匿名实体e上。通常情况下，作用在匿名实体上和作用在结构化绑定的变量上的效果是一样的，但有些时候又是不同的（见下文）。

例如，我们可以把声明一个结构化绑定声明为const引用：

```
const auto& [u, v] = ms;    // 引用，因此u/v指向ms.i/ms.s
```

这里，匿名实体被声明为const引用，而u和v分别是这个引用的成员i和s的别名。因此，对ms的成员的修改会影响到u和v的值：

```
ms.i = 77;          // 影响u的值
std::cout << u;     // 打印出77
```

如果声明为非const引用，你甚至可以修改对象的成员：

```
MyStruct ms{42, "hello"};
auto& [u, v] = ms;      // 被初始化的实体是ms的引用
ms.i = 77;              // 影响到u的值
std::cout << u;         // 打印出77
u = 99;                 // 修改了ms.i
std::cout << ms.i;      // 打印出99
```

如果一个结构化绑定是引用类型，而且是对一个临时对象的引用，那么和往常一样，临时对象的生命周期会被延长到结构化绑定的生命周期：

```
MyStruct getStruct();
...
const auto& [a, b] = getStruct();
std::cout << "a: " << a << '\n';    // OK
```

修饰符并不是作用在结构化绑定引入的变量上

修饰符会作用在新的匿名实体上，而不是结构化绑定引入的新的变量名上。事实上，如下代码中：

```
const auto& [u, v] = ms;    // 引用，因此u/v指向ms.i/ms.s
```

u 和 v 都不是引用，只有匿名实体 e 是一个引用。u 和 v 分别是 ms 对应的成员的类型，只不过变成了 const 的。根据我们的推导，decltype(u) 是 const int，decltype(v) 是 const std::string。

当声明对齐时也是类似：

```
alignas(16) auto [u, v] = ms;   // 对齐匿名实体，而不是v
```

这里，我们对齐了匿名实体而不是 u 和 v。这意味着 u 作为第一个成员会按照 16 字节对齐，但 v 不会。

因此，即使使用了 auto 结构化绑定也不会发生类型退化 (*decay*) [2]。例如，如果我们有一个原生数组组成的结构体：

```
struct S {
    const char x[6];
    const char y[3];
};
```

那么如下声明之后：

```
S s1{};
auto [a, b] = s1;    // a和b的类型是结构体成员的精确类型
```

这里 a 的类型仍然是 char[6]。再次强调，auto 关键字应用在匿名实体上，这里匿名实体整体并不会发生类型退化。这和用 auto 初始化新对象不同，如下代码中会发生类型退化：

```
auto a2 = a;    // a2的类型是a的退化类型
```

move 语义

move 语义也遵循之前介绍的规则，如下声明：

```
MyStruct ms = { 42, "Jim" };
auto&& [v, n] = std::move(ms);     // 匿名实体是ms的右值引用
```

这里 v 和 n 指向的匿名实体是 ms 的右值引用。同时 ms 的值仍然保持不变：

```
std::cout << "ms.s: " << ms.s << '\n';  // 打印出"Jim"
```

然而，你可以对指向 ms.s 的 n 进行移动赋值：

```
std::string s = std::move(n);           // 把ms.s移动到s
std::cout << "ms.s: " << ms.s << '\n';  // 打印出未定义的值
std::cout << "n:    " << n << '\n';     // 打印出未定义的值
std::cout << "s:    " << s << '\n';     // 打印出"Jim"
```

像通常一样，值被移动走的对象处于一个值未定义但却有效的状态。因此可以打印它们的值，但不要对打印出的值有任何期望。[3]

上面的例子和直接用 ms 被移动走的值进行结构化绑定有些不同：

[2] 术语 *decay* 是指当参数按值传递时发生的类型转换，例如原生数组会转换为指针，顶层修饰符例如 const 和引用会被忽略

[3] 对于 string 来说，值被移动走之后一般是处于空字符串的状态，但并不保证这一点

```
MyStruct ms = {42, "Jim" };
auto [v, n] = std::move(ms);    // 新的匿名实体持有从ms处移动走的值
```

这里新的匿名实体是用 ms 被移动走的值来初始化的。因此，ms 已经失去了值：

```
std::cout << "ms.s: " << ms.s << '\n';  // 打印出未定义的值
std::cout << "n:    " << n << '\n';     // 打印出"Jim"
```

你可以继续用 n 进行移动赋值或者给 n 赋予新值，但已经不会再影响到 ms.s 了：

```
std::string s = std::move(n);   // 把n移动到s
n = "Lara";
std::cout << "ms.s: " << ms.s << '\n';  // 打印出未定义的值
std::cout << "n:    " << n << '\n';     // 打印出"Lara"
std::cout << "s:    " << s << '\n';     // 打印出"Jim"
```

1.2 结构化绑定的适用场景

原则上讲，结构化绑定适用于所有只有 public 数据成员的结构体、C 风格数组和类似元组 (tuple-like) 的对象：

- 对于所有非静态数据成员都是 public 的**结构体和类**，你可以把每一个成员绑定到一个新的变量名上。
- 对于**原生数组**，你可以把数组的每一个元素都绑定到新的变量名上。
- 对于任何类型，你可以使用 **tuple-like API** 来绑定新的名称，无论这套 API 是如何定义“元素”的。对于一个类型 *type* 这套 API 需要如下的组件：
  - std::tuple_size<type>::value 要返回元素的数量。
  - std::tuple_element<idx, type>::type 要返回第 idx 个元素的类型。
  - 一个全局或成员函数 get<idx>() 要返回第 idx 个元素的值。

  标准库类型 std::pair<>、std::tuple<>、std::array<> 就是提供了这些 API 的例子。

如果结构体和类提供了 tuple-like API，那么将会使用这些 API 进行绑定，而不是直接绑定数据成员。

在任何情况下，结构化绑定中声明的变量名的数量都必须和元素或数据成员的数量相同。你不能跳过某个元素，也不能重复使用变量名。然而，你可以使用非常短的名称例如 '_'（有的程序员喜欢这个名字，有的讨厌它，但注意全局命名空间不允许使用它），但这个名字在同一个作用域只能使用一次：

```
auto [_, val1] = getStruct();   // OK
auto [_, val2] = getStruct();   // ERROR:变量名_已经被使用过
```

目前还不支持嵌套化的结构化绑定。

下一小节将详细讨论结构化绑定的使用。

1.2.1 结构体和类

上面几节里已经介绍了对只有 public 成员的结构体和类使用结构化绑定的方法，一个典型的应用是直接对包含多个数据的返回值使用结构化绑定。然而有一些边缘情况需要注意。

注意要使用结构化绑定需要继承时遵循一定的规则。所有的非静态数据成员必须在同一个类中定义（也就是说，这些成员要么是全部直接来自于最终的类，要么是全部来自同一个父类）：

```
struct B {
    int a = 1;
    int b = 2;
};

struct D1 : B {
};
auto [x, y] = D1{};     // OK

struct D2 : B {
    int c = 3;
};
auto [i, j, k] = D2{};  // 编译期ERROR
```

注意只有当public成员的顺序保证是固定的时候你才应该使用结构化绑定。否则如果B中的int a和int b的顺序发生了变化，x和y的值也会随之变化。为了保证固定的顺序，C++17为一些标准库结构体（例如insert_return_type）定义了成员顺序。

联合还不支持使用结构化绑定。

1.2.2 原生数组

下面的代码用C风格数组的两个元素初始化了x和y：

```
int arr[] = { 47, 11 };
auto [x, y] = arr;  // x和y是arr中的int元素的拷贝
auto [z] = arr;     // ERROR：元素的数量不匹配
```

注意这是C++中少数几种原生数组会按值拷贝的场景之一。

只有当数组的长度已知时才可以使用结构化绑定。数组作为按值传入的参数时不能使用结构化绑定，因为数组会退化 *(decay)* 为相应的指针类型。

注意C++允许通过引用来返回带有大小信息的数组，结构化绑定可以应用于返回这种数组的函数：

```
auto getArr() -> int(&)[2]; // getArr()返回一个原生int数组的引用
...
auto [x, y] = getArr();     // x和y是返回的数组中的int元素的拷贝
```

你也可以对std::array使用结构化绑定，这是通过下一节要讲述的tuple-like API来实现的。

1.2.3 std::pair, std::tuple 和 std::array

结构化绑定的机制是可拓展的，你可以为任何类型添加对结构化绑定的支持。标准库中就为std::pair<>、std::tuple<>、std::array<>添加了支持。

std::array

例如，下面的代码为getArray()返回的std::array<>中的四个元素绑定了新的变量名a，b，c，d：

```
std::array<int, 4> getArray();
...
auto [a, b, c, d] = getArray(); // a,b,c,d是返回值的拷贝中的四个元素的别名
```

这里a，b，c，d被绑定到getArray()返回的std::array的元素上。

使用非临时变量的non-const引用进行绑定，还可以进行修改操作。例如：

```
std::array<int, 4> stdarr { 1, 2, 3, 4 };
...
auto& [a, b, c, d] = stdarr;
a += 10;    // OK:修改了stdarr[0]

const auto& [e, f, g, h] = stdarr;
e += 10;    // ERROR:引用指向常量对象

auto&& [i, j, k, l] = stdarr;
i += 10;    // OK:修改了stdarr[0]

auto [m, n, o, p] = stdarr;
m += 10;    // OK:但是修改的是stdarr[0]的拷贝
```

然而像往常一样，我们不能用临时对象(prvalue)初始化一个非const引用：

```
auto& [a, b, c, d] = getArray();    // ERROR
```

std::tuple

下面的代码将a，b，c初始化为getTuple()返回的std::tuple<>的拷贝的三个元素的别名：

```
std::tuple<char, float, std::string> getTuple();
...
auto [a, b, c] = getTuple();    // a,b,c的类型和值与返回的tuple中相应的成员相同
```

其中a的类型是char，b的类型是float，c的类型是std::string。

std::pair

作为另一个例子，考虑如下对关联/无序容器的insert()成员的返回值进行处理的代码：

```
std::map<std::string, int> coll;
auto ret = coll.insert({"new", 42});
if (!ret.second) {
    // 如果插入失败，使用ret.first处理错误
    ...
}
```

通过使用结构化绑定，而不是使用std::pair<>的first和second成员，代码的可读性大大增强：

```
auto [pos, ok] = coll.insert({"new", 42});
if (!ok) {
    // 如果插入失败，用pos处理错误
```

```
    ...
}
```

注意在这种场景中，C++17 中提供了一种使用带初始化的 `if` 语句 来进行改进的方法。

为 `pair` 和 `tuple` 的结构化绑定赋予新值

在声明了一个结构化绑定之后，你通常不能同时修改所有绑定的变量，因为结构化绑定只能一起声明但不能一起使用。然而，如果被赋的值可以赋给一个 `std::pair<>` 或 `std::tuple<>` ，你可以使用 `std::tie()` 把值一起赋给所有变量。例如：

```
std::tuple<char, float, std::string> getTuple();
...
auto [a, b, c] = getTuple();    // a,b,c的类型和值与返回的tuple相同
...
std::tie(a, b, c) = getTuple(); // a,b,c的值变为新返回的tuple的值
```

这种方法可以被用来处理返回多个值的循环，例如在循环中使用搜索器：

```
std::boyer_moore_searcher bmsearch{sub.begin(), sub.end()};
for (auto [beg, end] = bmsearch(text.begin(), text.end());
     beg != text.end();
     std::tie(beg, end) = bmsearch(end, text.end())) {
    ...
}
```

1.3 为结构化绑定提供 Tuple-Like API

你可以通过提供 *tuple-like API* 为任何类型添加对结构化绑定的支持，就像标准库为 `std::pair<>`、`std::tuple<>`、`std::array<>` 做的一样：

支持只读结构化绑定

下面的例子演示了怎么为一个类型 `Customer` 添加结构化绑定支持，类的定义如下：

lang/customer1.hpp

```
#include <string>
#include <utility>  // for std::move()

class Customer {
private:
    std::string first;
    std::string last;
    long val;
public:
    Customer (std::string f, std::string l, long v)
            : first{std::move(f)}, last{std::move(l)}, val{v} {
```

```
    }
    std::string getFirst() const {
        return first;
    }
    std::string getLast() const {
        return last;
    }
    long getValue() const {
        return val;
    }
};
```

我们可以用如下代码添加 tuple-like API：

lang/structbind1.hpp

```
#include "customer1.hpp"
#include <utility>  // for tuple-like API

// 为类Customer提供tulpe-like API：
template<>
struct std::tuple_size<Customer> {
    static constexpr int value = 3; // 有三个属性
};

template<>
struct std::tuple_element<2, Customer> {
    using type = long;  // 最后一个属性的类型是long
};
template<std::size_t Idx>
struct std::tuple_element<Idx, Customer> {
    using type = std::string;   // 其他的属性都是string
};

// 定义特化的getter：
template<std::size_t> auto get(const Customer& c);
template<> auto get<0>(const Customer& c) { return c.getFirst(); }
template<> auto get<1>(const Customer& c) { return c.getLast(); }
template<> auto get<2>(const Customer& c) { return c.getValue(); }
```

这里，我们为顾客的三个属性定义了 tuple-like API，并映射到三个 getter：

- 顾客的名（first name）是 `std::string` 类型
- 顾客的姓（last name）是 `std::string` 类型
- 顾客的消费金额是 `long` 类型

属性的数量被定义为 `std::tuple_size` 模板函数对类 `Customer` 的特化版本：

```
template<>
struct std::tuple_size<Customer> {
```

```
    static constexpr int value = 3; // 我们有3个属性
};
```

属性的类型被定义为`std::tuple_element`的特化版本：

```
template<>
struct std::tuple_element<2, Customer> {
    using type = long;  // 最后一个属性是long类型
};
template<std::size_t Idx>
struct std::tuple_element<Idx, Customer> {
    using type = std::string;   // 其他的属性是string
};
```

第三个属性是`long`，被定义为`Idx`为2时的完全特化版本。其他的属性类型都是`std::string`，被定义为部分特化版本（优先级比全特化版本低）。这里的类型就是结构化绑定时`decltype`返回的类型。

最后，在和类`Customer`同级的命名空间中定义了函数`get<>()`的重载版本作为getter[4]：

```
template<std::size_t> auto get(const Customer& c);
template<> auto get<0>(const Customer& c) { return c.getFirst(); }
template<> auto get<1>(const Customer& c) { return c.getLast(); }
template<> auto get<2>(const Customer& c) { return c.getValue(); }
```

在这种情况下，我们有一个主函数模板的声明和针对所有情况的全特化版本。

注意函数模板的全特化版本必须使用和声明时相同的类型（包括返回值类型都必须完全相同）。这是因为我们只是提供特化版本的实现，而不是新的声明。下面的代码将不能通过编译：

```
template<std::size_t> auto get(const Customer& c);
template<> std::string get<0>(const Customer& c) { return c.getFirst(); }
template<> std::string get<1>(const Customer& c) { return c.getLast(); }
template<> long get<2>(const Customer& c) { return c.getValue(); }
```

通过使用新的编译期`if`语句特性，我们可以把`get<>()`函数的实现 合并到一个函数里：

```
template<std::size_t I> auto get(const Customer& c) {
    static_assert(I < 3);
    if constexpr (I == 0) {
        return c.getFirst();
    }
    else if constexpr (I == 1) {
        return c.getLast();
    }
    else {  // I == 2
        return c.getValue();
    }
}
```

有了这个API，我们就可以为类型`Customer`使用结构化绑定：

[4]C++17标准也允许把`get<>()`函数定义为成员函数，但这可能只是一个疏忽，因此不应该这么用

lang/structbind1.cpp

```
#include "structbind1.hpp"
#include <iostream>

int main()
{
    Customer c{"Tim", "Starr", 42};
    auto [f, l, v] = c;

    std::cout << "f/l/v:    " << f << ' ' << l << ' ' << v << '\n';

    // 修改结构化绑定的变量
    std::string s{std::move(f)};
    l = "Waters";
    v += 10;
    std::cout << "f/l/v:    " << f << ' ' << l << ' ' << v << '\n';
    std::cout << "c:        " << c.getFirst() << ' '
              << c.getLast() << ' ' << c.getValue() << '\n';
    std::cout << "s:        " << s << '\n';
}
```

如下初始化之后：

```
auto [f, l, v] = c;
```

像之前的例子一样，`Customer c` 被拷贝到一个匿名实体。当结构化绑定离开作用域时匿名实体也被销毁。

另外，对于每一个绑定 `f`、`l`、`v`，它们对应的 `get<>()` 函数都会被调用。因为定义的 `get<>` 函数返回类型是 `auto`，所以这 3 个 getter 会返回成员的拷贝，这意味着结构化绑定的变量的地址不同于 `c` 中成员的地址。因此，修改 `c` 的值并不会影响绑定变量（反之亦然）。

使用结构化绑定等同于使用 `get<>()` 函数的返回值，因此：

```
std::cout << "f/l/v:    " << f << ' ' << l << ' ' << v << '\n';
```

只是简单的输出变量的值（并不会再次调用 getter 函数）。另外

```
std::string s{std::move(f)};
l = "Waters";
v += 10;
std::cout << "f/l/v:    " << f << ' ' << l << ' ' << v << '\n';
```

这段代码修改了绑定变量的值。因此，这段程序总是有如下输出：

```
f/l/v:    Tim Starr 42
f/l/v:     Waters 52
c:        Tim Starr 42
s:        Tim
```

第二行的输出依赖于被 move 的 `string` 的值，一般情况下是空值，但也有可能是其他有效的值。

你也可以在迭代一个元素类型是 `Customer` 的 `vector` 时使用结构化绑定：

```
std::vector<Customer> coll;
...
for (const auto& [first, last, val] : coll) {
    std::cout << first << ' ' << last << ": " << val << '\n';
}
```

在这个循环中，因为使用了 `const auto&` 所以不会有 `Customer` 被拷贝。然而，结构化绑定的变量初始化时会调用 `get<>()` 函数返回姓和名的拷贝。之后，循环体内的输出语句中再次使用了结构化绑定，不需要再次调用 getter。最后在每一次迭代结束的时候，拷贝的 `string` 会被销毁。

注意对绑定变量使用 `decltype` 会推导出变量自身的类型，不会受到匿名实体的类型修饰符的影响。也就是说这里 `decltype(first)` 的类型是 `const std::string` 而不是引用。

支持可写结构化绑定

实现 tuple-like API 时可以时候用返回 `non-const` 引用。这样结构化绑定就变得可写。设想类 `Customer` 提供了读写成员的 API[5]：

lang/customer2.hpp

```
#include <string>
#include <utility>  // for std::move()

class Customer {
private:
    std::string first;
    std::string last;
    long val;
public:
    Customer (std::string f, std::string l, long v)
            : first{std::move(f)}, last{std::move(l)}, val{v} {
    }
    const std::string& firstname() const {
        return first;
    }
    std::string& firstname() {
        return first;
    }
    const std::string& lastname() const {
        return last;
    }
    long value() const {
        return val;
    }
    long& value() {
        return val;
    }
```

[5]这个类的设计比较失败，因为通过成员函数可以直接访问私有成员。然而用来演示怎么支持可写结构化绑定已经足够了

```
};
```

为了支持读写，我们需要为常量和非常量引用定义重载的 getter：

lang/structbind2.hpp

```
#include "customer2.hpp"
#include <utility>  // for tuple-like API

// 为类Customer提供tuple-like API
template<>
struct std::tuple_size<Customer> {
    static constexpr int value = 3; // 有3个属性
};

template<>
struct std::tuple_element<2, Customer> {
    using type = long;  // 最后一个属性是long类型
}
template<std::size_t Idx>
struct std::tuple_element<Idx, Customer> {
    using type = std::string;   // 其他的属性是string
}

// 定义特化的getter：
template<std::size_t I> decltype(auto) get(Customer& c) {
    static_assert(I < 3);
    if constexpr (I == 0) {
        return c.firstname();
    }
    else if constexpr (I == 1) {
        return c.lastname();
    }
    else {  // I == 2
        return c.value();
    }
}
template<std::size_t I> decltype(auto) get(const Customer& c) {
    static_assert(I < 3);
    if constexpr (I == 0) {
        return c.firstname();
    }
    else if constexpr (I == 1) {
        return c.lastname();
    }
    else {  // I == 2
        return c.value();
    }
}
```

```
template<std::size_t I> decltype(auto) get(Customer&& c) {
    static_assert(I < 3);
    if constexpr (I == 0) {
        return std::move(c.firstname());
    }
    else if constexpr (I == 1) {
        return std::move(c.lastname());
    }
    else {  // I == 2
        return c.value();
    }
}
```

注意你必须提供这 3 个版本的特化来分别处理常量对象、非常量对象、可移动对象，[6]。为了实现返回引用，你应该使用 `decltype(auto)` [7]。

这里我们又一次使用了编译期 `if` 语句特性，这可以让我们的 getter 的实现变得更加简单。如果没有这个特性，我们必须写出所有的全特化版本，例如：

```
template<std::size_t> decltype(auto) get(const Customer& c);
template<std::size_t> decltype(auto) get(Customer& c);
template<std::size_t> decltype(auto) get(Customer&& c);
template<> decltype(auto) get<0>(const Customer& c) { return c.firstname(); }
template<> decltype(auto) get<0>(Customer& c) { return c.firstname(); }
template<> decltype(auto) get<0>(Customer&& c) { return c.firstname(); }
template<> decltype(auto) get<1>(const Customer& c) { return c.lastname(); }
template<> decltype(auto) get<1>(Customer& c) { return c.lastname(); }
...
```

再次强调，主函数模板声明必须和全特化版本拥有完全相同的签名（包括返回值）。下面的代码不能通过编译：

```
template<std::size_t> decltype(auto) get(Customer& c);
template<> std::string& get<0>(Customer& c) { return c.firstname(); }
template<> std::string& get<1>(Customer& c) { return c.lastname(); }
template<> long& get<2>(Customer& c) { return c.value(); }
```

你现在可以对 `Customer` 类使用结构化绑定了，并且还能通过绑定修改成员的值：

lang/structbind2.cpp

```
#include "structbind2.hpp"
#include <iostream>

int main()
{
    Customer c{"Tim", "Starr", 42};
    auto [f, l, v] = c;
```

[6]标准库中还为 `const&&` 实现了第 4 个版本的 `get<>()` 这么做是有原因的（见 https://wg21.link/lwg2485），但如果只是想支持结构化绑定则不是必须的。

[7]`decltype(auto)` 在 C++14 中引入，它可以根据表达式的值类别 (*value category*) 来推导（返回）类型。简单来说，将它设置为返回值类型之后引用会以引用返回，但临时值会以值返回。

```
    std::cout << "f/l/v:    " << f << ' ' << l << ' ' << v << '\n';

    // 通过引用修改结构化绑定
    auto&& [f2, l2, v2] = c;
    std::string s{std::move(f2)};
    f2 = "Ringo";
    v2 += 10;
    std::cout << "f2/l2/v2: " << f2 << ' ' << l2 << ' ' << v2 << '\n';
    std::cout << "c:        " << c.firstname() << ' '
              << c.lastname() << ' ' << c.value() << '\n';
    std::cout << "s:        " << s << '\n';
}
```

程序的输出如下：

```
f/l/v:    Tim Starr 42
f2/l2/v2: Ringo Starr 52
c:        Ringo Starr 52
s:        Tim
```

1.4 后记

结构化绑定最早由 Herb Sutter、Bjarne Stroustrup、Gabriel Dos Reis 在 https://wg21.link/p0144r0提出，当时提议使用花括号而不是方括号。最终被接受的提案由 Jens Maurer 发表于https://wg21.link/p0217r3。

Chapter 2

带初始化的 `if` 和 `switch` 语句

`if` 和 `switch` 语句现在允许在条件表达式里添加一条初始化语句。例如，你可以写出如下代码：

```
if (status s = check(); s != status::success) {
    return s;
}
```

其中的初始化语句是：

```
status s = check();
```

它初始化了 `s`，`s` 将在整个 `if` 语句中有效（包括 `else` 分支里）。

2.1 带初始化的 `if` 语句

在 `if` 语句的条件表达式里定义的变量将在整个 `if` 语句中有效（包括 *then* 部分和 *else* 部分）。例如：

```
if (std::ofstream strm = getLogStrm(); coll.empty()) {
    strm << "<no data>\n";
}
else {
    for (const auto& elem : coll) {
        strm << elem << '\n';
    }
}
// strm不再有效
```

在整个 `if` 语句结束时 `strm` 的析构函数会被调用。另一个例子是关于锁的使用，假设我们要在并发的环境中执行一个依赖某些条件的任务：

```
if (std::lock_guard<std::mutex> lg{collMutex}; !coll.empty()) {
    std::cout << coll.front() << '\n';
}
```

这个例子中，如果使用类模板参数推导，可以改写成如下代码：

```
if (std::lock_guard lg{collMutex}; !coll.empty()) {
    std::cout << coll.front() << '\n';
}
```

上面的代码等价于：

```
{
    std::lock_guard<std::mutex> lg{collMutex};
    if (!coll.empty()) {
        std::cout << coll.front() << '\n';
    }
}
```

细微的区别在于前者中 lg 在 if 语句的作用域之内定义，和条件语句在相同的作用域。

注意这个特性的效果和传统 for 循环里的初始化语句完全相同。上面的例子中为了让 lock_guard 生效，必须在初始化语句里明确声明一个变量名, 否则它就是一个临时变量，会在创建之后就立即销毁。因此，初始化一个没有变量名的临时 lock_guard 是一个逻辑错误，因为当执行到条件语句时锁就已经被释放了：

```
if (std::lock_guard<std::mutex>{collMutex};     // 运行时ERROR
    !coll.empty()) {                    // 锁已经被释放了
    std::cout << coll.front() << '\n';  // 锁已经被释放了
}
```

原则上讲，使用简单的 _ 作为变量名就已经足够了：

```
if (std::lock_guard<std::mutex> _{collMutex};   // OK，但是...
    !coll.empty()) {
    std::cout << coll.front() << '\n';
}
```

你也可以同时声明多个变量，并且可以在声明时初始化：

```
if (auto x = qqq1(), y = qqq2(); x != y) {
    std::cout << "return values " << x << " and " << y << "differ\n";
}
```

或者：

```
if (auto x{qqq1()}, y{qqq2()}; x != y) {
    std::cout << "return values " << x << " and " << y << "differ\n";
}
```

另一个例子是向 map 或者 unordered map 插入元素。你可以像下面这样检查是否成功：

```
std::map<std::string, int> coll;
...
if (auto [pos, ok] = coll.insert({"new", 42}); !ok) {
    // 如果插入失败，用pos处理错误
    const auto& [key, val] = *pos;
    std::cout << "already there: " << key << '\n';
}
```

这里，我们用了结构化绑定来给返回的 pos 指向的值声明了新的名称，而不是使用 first 和 second 成员。在 C++17 之前，相应的处理代码必须像下面这样写：

```
auto ret = coll.insert({"new", 42});
if (!ret.second) {
    // 如果插入失败，用ret.first处理错误
    const auto& elem = *(ret.first);
    std::cout << "already there: " << elem.first << '\n';
}
```

注意这个拓展也适用于编译期 if 语句特性。

2.2 带初始化的 switch 语句

通过使用带初始化的 switch 语句，我们可以在控制流之前初始化一个对象/实体。例如，我们可以先声明一个文件系统路径，然后再根据它的类别进行处理：

```
namespace fs = std::filesystem;
...
switch (fs::path p{name}; status(p).type()) {
    case fs::file_type::not_found:
        std::cout << p << " not found\n";
        break;
    case fs::file_type::directory:
        std::cout << p << ":\n";
        for (const auto& e : std::filesystem::directory_iterator{p}) {
            std::cout << "- " << e.path() << '\n';
        }
        break;
    default:
        std::cout << p << " exists\n";
        break;
}
```

这里，初始化的路径 p 可以在整个 switch 语句中使用。

2.3 后记

带初始化的 if 和 switch 语句最早由 Thomas Köppe 在https://wg21.link/p0305r0中提出，一开始只是提到了扩展 if 语句。最终被接受的提案由 Thomas Köpped 发表于https://wg21.link/p0305r1。

Chapter 3

内联变量

出于可移植性和易于整合的目的，提供包含类库声明的头文件是很重要的。然而，在C++17之前，只有当这个库既不提供也不需要全局对象的时候才可以直接在头文件中定义。

自从C++17开始，你可以在头文件中以`inline`的方式定义全局变量/对象：

```
class MyClass {
    inline static std::string msg{"OK"}; // OK（自C++17起）
    ...
};

inline MyClass myGlobalObj;  // 即使被多个CPP文件包含也OK
```

只要一个翻译单元内没有重复的定义即可。此例中的定义即使被多个翻译单元使用，也会指向同一个对象。

3.1 内联变量产生的动机

在C++里不允许在类里初始化非常量静态成员：

```
class MyClass {
    static std::string msg{"OK"};   // 编译期ERROR
    ...
};
```

可以在类定义的外部定义并初始化非常量静态成员，但如果被多个CPP文件同时包含的话又会引发新的错误：

```
class MyClass {
    static std::string msg;
    ...
};
std::string MyClass::msg{"OK"}; // 如果被多个CPP文件包含会导致链接ERROR
```

根据一次定义原则(ODR)，一个变量或实体的定义只能出现在一个翻译单元内——除非该变量或实体被定义为`inline`的。

即使使用预处理来进行保护也没有用：

```
#ifndef MYHEADER_HPP
#define MYHEADER_HPP

class MyClass {
    static std::string msg;
    ...
};
std::string MyClass::msg{"OK"}; // 如果被多个CPP文件包含会导致链接ERROR

#endif
```

问题并不在于头文件是否可能被重复包含多次，而是两个不同的CPP文件都包含了这个头文件，因而都定义了`MyClass::msg`。出于同样的原因，如果你在头文件中定义了一个类的实例对象也会出现相同的链接错误：

```
class MyClass {
    ...
};
MyClass myGlobalObject; // 如果被多个CPP文件包含会导致链接ERROR
```

解决方法

对于一些场景，这里有一些解决方法：

- 你可以在一个`class/struct`的定义中初始化数字或枚举类型的常量静态成员：

```
class MyClass {
    static const bool trace = false;    // OK，字面类型
    ...
};
```

然而，这种方法只能初始化字面类型，例如基本的整数、浮点数、指针类型或者用常量表达式初始化了所有内部非静态成员的类，并且该类不能有用户自定义的或虚的析构函数。另外，如果你需要获取这个静态常量成员的地址（例如你想定义一个指向它的引用）的话那么你必须在那个翻译单元内定义它并且不能在其他翻译单元内再次定义。

- 你可以定义一个返回`static`的局部变量的内联函数：

```
inline std::string& getMsg() {
    static std::string msg{"OK"};
    return msg;
}
```

- 你可以定义一个返回该值的`static`的成员函数：

```
class MyClass {
    static std::string& getMsg() {
        static std::string msg{"OK"};
        return msg;
    }
    ...
};
```

- 你可以使用变量模板（自 C++14 起）：

```
template<typename T = std::string>
T myGlobalMsg{"OK"};
```

- 你可以为静态数据成员定义一个模板类：

```
template<typename = void>
class MyClassStatics
{
    static std::string msg;
};

template<typename T>
std::string MyClassStatics<T>::msg{"OK"};
```

然后继承它：

```
class MyClass : public MyClassStatics<>
{
    ...
};
```

然而，所有这些方法都会导致签名重载，可读性也会变差，使用该变量的方式也变得不同。另外，全局变量的初始化可能会推迟到第一次使用时。所以那些假设变量一开始就已经初始化的写法是不可行的（例如使用一个对象来监控整个程序的过程）。

3.2 使用内联变量

现在，使用了 `inline` 修饰符之后，即使定义所在的头文件被多个 CPP 文件包含，也只会有一个全局对象：

```
class MyClass {
    inline static std::string msg{"OK"};    // 自从C++17起OK
    ...
};

inline MyClass myGlobalObj; // 即使被多个CPP文件包含也OK
```

这里使用的 `inline` 和函数声明时的 `inline` 有相同的语义：

- 它可以在多个翻译单元中定义，只要所有定义都是相同的。
- 它必须在每个使用它的翻译单元中定义

将变量定义在头文件里，然后多个 CPP 文件再都包含这个头文件，就可以满足上述两个要求。程序的行为就好像只有一个变量一样。你甚至可以利用它在头文件中定义原子类型：

```
inline std::atomic<bool> ready{fasle};
```

像通常一样，当你定义 `std::atomic` 类型的变量时必须进行初始化。

注意你仍然必须确保在你初始化内联变量之前它们的类型必须是完整的。例如，如果一个 `struct` 或者 `class` 有一个自身类型的 `static` 成员，那么这个成员只能在类型声明之后再进行定义：

```
struct MyType {
    int value;
    MyType(int i) : value{i} {
    }
    // 一个存储该类型最大值的静态对象
    static MyType max;  // 这里只能进行声明
    ...
};
inline MyType MyType::max{0};
```

另一个使用内联变量的例子参见追踪所有 new 调用的头文件。

3.3 constexpr static 成员现在隐含 inline

对于静态成员，constexpr 修饰符现在隐含着 inline。自从 C++17 起，如下声明定义了静态数据成员 n：

```
struct D {
    static constexpr int n = 5; // C++11/C++14: 声明
                                // 自从C++17起: 定义
}
```

和下边的代码等价：

```
struct D {
    inline static constexpr int n = 5;
};
```

注意在 C++17 之前，你就可以只有声明没有定义。考虑如下声明：

```
struct D {
    static constexpr int n = 5;
};
```

如果不需要 D::n 的定义的话只有上面的声明就够了，例如当 D::n 以值传递时：

```
std::cout << D::n; // OK, ostream::operator<<(int)只需要D::n的值
```

如果 D::n 以引用传递到一个非内联函数，并且该函数调用没有被优化掉的话，该调用将会导致错误。例如：

```
int twice(const int& i);

std::cout << twice(D::n);   // 通常情况下会导致ERROR
```

这段代码违反了一次定义原则 (ODR)。如果编译器进行了优化，那么这段代码可能会像预期一样工作也可能会因为缺少定义导致链接错误。如果不进行优化，那么几乎肯定会因为缺少 D::n 的定义而导致错误。[1] 如果创建一个 D::n 的指针那么更可能导致链接错误（但在某些编译模式下仍然可能正常编译）：

```
const int* p = &D::n;   // 通常会导致ERROR
```

因此在 C++17 之前，你必须在一个翻译单元内定义 D::n:

[1] 感谢 Richard Smith 指出这一点。

```
constexpr int D::n;      // C++11/C++14: 定义
                         // 自从C++17起: 多余的声明（已被废弃）
```

现在当使用C++17进行构建时，类中的声明本身就成了定义，因此即使没有正式的定义，上面的所有例子现在也都可以正常工作。正式的定义现在仍然有效但已经成了废弃的多余声明。

3.4 内联变量和 thread_local

通过使用 thread_local 你可以为每个线程创建一个内联变量：

```
struct ThreadData {
    inline static thread_local std::string name;    // 每个线程都有自己的name
    ...
};

inline thread_local std::vector<std::string> cache; // 每个线程都有一份cache
```

作为一个完整的例子，考虑如下头文件：

lang/inlinethreadlocal.hpp

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

struct MyData {
    inline static std::string gName = "global";     // 整个程序中只有一个
    inline static thread_local std::string tName = "tls";   // 每个线程有一个
    std::string lName = "local";    // 每个实例有一个
    ...
    void print(const std::string& msg) const {
        std::cout << msg << '\n';
        std::cout << "- gName: " << gName << '\n';
        std::cout << "- tName: " << tName << '\n';
        std::cout << "- lName: " << lName << '\n';
    }
};

inline thread_local MyData myThreadData;    // 每个线程一个对象
```

你可以在包含 main() 的翻译单元内使用它：

lang/inlinethreadlocal1.cpp

```
#include "inlinethreadlocal.hpp"
#include <thread>

void foo();

int main()
{
```

```
    myThreadData.print("main() begin:");

    myThreadData.gName = "thraed1 name";
    myThreadData.tName = "thread1 name";
    myThreadData.lName = "thread1 name";
    myThreadData.print("main() later:");

    std::thread t(foo);
    t.join();
    myThreadData.print("main() end:");
}
```

你也可以在另一个定义了 `foo()` 函数的翻译单元内使用这个头文件，这个函数会在另一个线程中被调用：

lang/inlinethreadlocal2.cpp

```
#include "inlinethreadlocal.hpp"

void foo()
{
    myThreadData.print("foo() begin:");

    myThreadData.gName = "thread2 name";
    myThreadData.tName = "thread2 name";
    myThreadData.lName = "thread2 name";
    myThreadData.print("foo() end:");
}
```

程序的输出如下：

```
main() begin:
- gName: global
- tName: tls
- lName: local
main() later:
- gName: thread1 name
- tName: thread1 name
- lName: thread1 name
foo() begin:
- gName: thread1 name
- tName: tls
- lName: local
foo() end:
- gName: thread2 name
- tName: thread2 name
- lName: thread2 name
main() end:
- gName: thread2 name
- tName: thread1 name
- lName: thread1 name
```

3.5 后记

内联变量的动机起源于David Krauss的https://wg21.link/n4147提案，之后由Hal Finkel和Richard Smith在https://wg21.link/n4424中第一次提出。最终被接受的提案由Hal Finkel和Richard Smith发表于https://wg21.link/p0386r2。

Chapter 4

聚合体扩展

C++ 有很多初始化对象的方法。其中之一叫做聚合体*初始化 (aggregate initialization)*，这是聚合体[1] 专有的一种初始化方法。从 C 语言引入的初始化方式是用花括号括起来的一组值来初始化类：

```
struct Data {
    std::string name;
    double value;
};

Data x = {"test1", 6.778};
```

自从 C++11 起，你可以忽略等号：

```
Data x{"test1", 6.778};
```

自从 C++17 起，聚合体可以拥有基类。也就是说像下面这种从其他类派生出的子类也可以使用这种初始化方法：

```
struct MoreData : Data {
    bool done;
}

MoreData y{{"test1", 6.778}, false};
```

如你所见，聚合体初始化时可以用一个子聚合体初始化来初始化类中来自基类的成员。

另外，你甚至可以省略子聚合体初始化的花括号：

```
MoreData y{"test1", 6.778, false};
```

这样写将遵循嵌套聚合体初始化时的通用规则，你传递的实参被用来初始化哪一个成员取决于它们的顺序。

4.1 扩展聚合体初始化的动机

如果没有这个特性，那么所有的派生类都不能使用聚合体初始化，这意味着你要像下面这样定义构造函数：

[1] 聚合体指数组或者 C 风格的简单类，简单类要求没有用户定义的构造函数、没有私有或保护的非静态数据成员、没有虚函数，另外在 C++17 之前，还要求没有继承。

```
struct Cpp14Data : Data {
    bool done;
    Cpp14Data (const std::string& s, double d, bool b) : Data{s, d}, done{b} {
    }
};

Cpp14Data y{"test1", 6.778, false};
```

现在我们不再需要定义任何构造函数就可以做到这一点。我们可以直接使用嵌套花括号的语法来实现初始化，如果给出了内层初始化需要的所有值就可以省略内层的花括号：

```
MoreData x{{"test1", 6.778}, false};    // 自从C++17起OK
MoreData y{"test1", 6.778, false};  // OK
```

注意因为现在派生类也可以是聚合体，所以其他的一些初始化方法也可以使用：

```
MoreData u;     // OOPS：value/done未初始化
MoreData z{};   // OK: value/done初始化为0/false
```

如果觉得这样很危险，可以使用成员初始值：

```
struct Data {
    std::string name;
    double value{0.0};
};

struct Cpp14Data : Data {
    bool done{false};
};
```

或者，提供一个默认构造函数。

4.2 使用聚合体扩展

聚合体初始化的一个典型应用场景是对一个派生自 C 风格结构体并且添加了新成员的类进行初始化。例如：

```
struct Data {
    const char* name;
    double value;
};

struct CppData : Data {
    bool critical;
    void print() const {
        std::cout << '[' << name << ',' << value << "]\n";
    }
};

CppData y{{"test1", 6.778}, false};
y.print();
```

这里，内层花括号里的参数被传递给基类Data。

注意你可以跳过初始化某些值。在这种情况下，跳过的成员将会进行默认初始化，例如：

```
CppData x1{};           // 所有成员默认初始化为0值
CppData x2{{"msg"}}     // 和{{"msg", 0.0}, false}等价
CppData x3{{}, true};   // 和{{nullptr, 0.0}, true}等价
CppData x4;             // 成员的值未定义
```

注意使用空花括号和不使用花括号完全不同：

- x1的定义会把所有成员默认初始化为0值，因此字符指针name被初始化为nullptr，double类型的value初始化为0.0，bool类型的flag初始化为false。
- x4的定义没有初始化任何成员。所有成员的值都是未定义的。

你也可以从非聚合体派生出聚合体。例如：

```
struct MyString : std::string {
    void print() const {
        if (empty()) {
            std::cout << "<undefined>\n";
        }
        else {
            std::cout << c_str() << '\n';
        }
    }
};

MyString x{{"hello"}};
MyString y{"world"};
```

注意这不是通常的具有多态性的public继承，因为std::string没有虚成员函数，你需要避免混淆这两种类型。

你甚至可以从多个基类和聚合体中派生出聚合体：

```
template<typename T>
struct D : std::string, std::complex<T>
{
    std::string data;
};
```

你可以像下面这样使用：

```
D<float> s{{"hello"}, {4.5, 6.7}, "world"}; // 自从C++17起OK
D<float> t{"hello", {4.5, 6.7}, "world"};   // 自从C++17起OK
std::cout << s.data;    // 输出："world"
std::cout << static_cast<std::string>(s);   // 输出："hello"
std::cout << static_cast<std::complex<float>>(s);   //输出：(4.5,6.7)
```

内部嵌套的初值列表将按照继承时基类声明的顺序传递给基类。

这个新的特性也可以帮助我们用很少的代码定义重载的lambda。

4.3 聚合体的定义

总的来说，在C++17中满足如下条件之一的对象被认为是聚合体：

- 是一个数组
- 或者是一个满足如下条件的类类型(class, struct, union)：
  - 没有用户定义的和`explicit`的构造函数
  - 没有使用`using`声明继承的构造函数
  - 没有`private`和`protected`的非静态数据成员
  - 没有`virtual`函数
  - 没有`virtual, private, protected`的基类

然而，要想使用聚合体初始化来初始化聚合体，那么还需要满足如下额外的约束：

- 基类中没有`private`或者`protected`的数据成员
- 没有`private`或者`protected`的构造函数

下一节就有一个因为不满足这些额外约束导致编译失败的例子。

C++17引入了一个新的类型trait `is_aggregate<>`来测试一个类型是否是聚合体：

```
template<typename T>
struct D : std::string, std::complex<T> {
    std::string data;
};
D<float> s{{"hello"}, {4.5, 6.7}, "world"}; // 自从C++17起OK
std::cout << std::is_aggregate<decltype(s)>::value; // 输出1(true)
```

4.4 向后的不兼容性

注意下面的例子不能再通过编译：

lang/aggr14.cpp

```
struct Derived;

struct Base {
    friend struct Derived;
private:
    Base() {
    }
};

struct Derived : Base {
};

int main()
{
    Derived d1{};   // 自从C++17起ERROR
```

```
    Derived d2;     // 仍然OK（但可能不会初始化）
}
```

在C++17之前，`Derived`不是聚合体。因此

```
Derived d1{};
```

会调用`Derived`隐式定义的默认构造函数，这个构造函数会调用基类`Base`的构造函数。尽管基类的默认构造函数是`private`的，但在派生类的构造函数里调用它也是有效的，因为派生类被声明为友元类。

自从C++17起，例子中的`Derived`是一个聚合体，所以它没有隐式的默认构造函数（构造函数没有使用`using`声明继承）。因此，`d1`的初始化将是一个聚合体初始化，如下表达式：

```
std::is_aggregate<Derived>::value
```

将返回`true`。

然而，因为基类有一个`private`的构造函数（见上一节）所以不能使用花括号来初始化。这和派生类是否是基类的友元无关。

4.5 后记

聚合体初始化扩展由Oleg Smolsky在https://wg21.link/n4404中第一次提到。最终被接受的正式提案由Oleg Smolsky发表于https://wg21.link/p0017r1。

类型特征`std::is_aggregate<>`作为美国国家机构对C++17标准的一个注释引入（见https://wg21.link/lwg2911）。

Chapter 5

强制省略拷贝或传递未实质化的对象 (Mandatory Copy Elision or Passing Unmaterialized Objects)

这一章的标题来自于以下两种视角：

- 从技术上讲，C++17 标准制定了一个规则：当以值传递或返回一个临时对象的时候必须省略对该临时对象的拷贝。
- 从效果上讲，我们实际上是传递了一个*未实质化的对象* (unmaterialized object)。

接下来首先从技术上介绍这个特性，之后再介绍实际效果和术语 *materialization*。

5.1 强制省略临时变量拷贝的动机

自从第一次标准开始，C++ 就允许在某些情况下*省略* (elision) 拷贝操作，即使这么做可能会影响程序的运行结果（例如，拷贝构造函数里的一条打印语句可能不会再执行）。当用临时对象初始化一个新对象时就很容易出现这种情况，尤其是当一个函数以值传递或返回临时对象的时候。例如：

```
class MyClass
{
    ...
};

void foo(MyClass param) {   // param用传递进入的实参初始化
    ...
}

MyClass bar() {
    return MyClass{};   // 返回临时对象
}
```

```
int main()
{
    foo(MyClass{});     // 传递临时对象来初始化param
    MyClass x = bar();  // 使用返回的临时对象初始化x
    foo(bar());         // 使用返回的临时对象初始化param
}
```

然而，因为这种优化并不是强制性的，所以例子中的情况要求该对象必须有隐式或显式的拷贝或构造函数。也就是说，尽管因为优化的原因大多数情况下并不会真的调用拷贝/移动函数，但它们必须存在。如果将上例中的`MyClass`定义换成如下定义则上例代码将不能通过编译：

```
class MyClass
{
  public:
    ...
    // 没有拷贝/移动构造函数的定义
    MyClass(const MyClass&) = delete;
    MyClass(MyClass&&) = delete;
    ...
};
```

只要没有拷贝构造函数就足以产生错误了，因为移动构造函数只有在没有用户声明的拷贝构造函数(或赋值操作符或析构函数)时才会隐式存在。（上例中只需要将拷贝构造函数定义为`delete`的就不会再有隐式定义的移动构造函数）

自从C++17起用临时变量初始化对象时省略拷贝变成了强制性的。事实上，之后我将会看到我们我们传递为参数或者作为返回值的临时变量将会被用来*实质化* (materialize)一个新的对象。这意味着即使上例中的`MyClass`完全不允许拷贝，示例代码也能成功编译。

然而，注意其他可选的省略拷贝的场景仍然是可选的。这种场景下仍然需要一个拷贝或者移动构造函数。例如：

```
MyClass foo()
{
    MyClass obj;
    ...
    return obj; // 仍然需要拷贝/移动构造函数的支持
}
```

这里，`foo()`中有一个具名的变量`obj`（当使用它时它是*左值*(lvalue)）。因此，*具名返回值优化* (named return value optimization)(NRVO)会生效，然而该优化仍然需要拷贝/移动支持。当`obj`是形参的时候也会出现这种情况：

```
MyClass bar(MyClass obj)    // 传递临时变量时会省略拷贝
{
    ...
    return obj;     // 仍然需要拷贝/移动支持
}
```

当向函数传递一个临时变量(也就是纯右值(prvalue))作为实参时不再需要拷贝/移动，但如果返回这个参数的话仍然需要拷贝/ 移动支持因为返回的对象是具名的。

作为变化的一部分，术语值类型体系的含义也做了很多修改和说明。

5.2 强制省略临时变量拷贝的好处

这个特性的一个显而易见的好处就是减少拷贝会带来更好的性能。尽管很多主流编译器之前就已经进行了这种优化，但现在这一行为有了标准的保证。尽管移动语义能显著的减少拷贝开销，但如果直接不拷贝还是能带来很大的性能提升（例如当对象有很多基本类型成员时移动语义还是要拷贝每个成员）。另外这个特性可以减少输出参数的使用，转而直接返回一个值（前提是这个值直接在返回语句里创建）。

另一个益处是可以定义一个总是可以工作的工厂函数因为现在它甚至可以返回不允许拷贝或移动的对象。例如，考虑如下泛型工厂函数：

lang/factory.hpp

```
#include <utility>

template <typename T, typename... Args>
T create(Args&&... args)
{
    ...
    return T{std::forward<Args>(args)...};
}
```

这个工厂函数现在甚至可以用于 `std::atomic<>` 这种既没有拷贝又没有移动构造函数的类型：

lang/factory.cpp

```
#include "factory.hpp"
#include <memory>
#include <atomic>

int main()
{
    int i = create<int>(42);
    std::unique_ptr<int> up =
    create<std::unique_ptr<int>>(new int{42});
    std::atomic<int> ai = create<std::atomic<int>>(42);
}
```

另一个效果就是对于移动构造函数被显式删除的类，现在也可以返回临时对象来初始化新的对象：

```
class CopyOnly {
  public:
    CopyOnly() {
    }
    CopyOnly(int) {
```

```
    }
    CopyOnly(const CopyOnly&) = default;
    CopyOnly(CopyOnly&&) = delete;  // 显式delete
};

CopyOnly ret() {
    return CopyOnly{};  // 自从C++17起OK
}

CopyOnly x = 42;    // 自从C++17起OK
```

在 C++17 之前 x 的初始化是无效的，因为*拷贝初始化*（使用 = 初始化）需要把 42 转化为一个临时对象，然后要用这个临时对象初始化 x 原则上需要移动构造函数，尽管它可能不会被调用。（只有当移动构造函数不是用户自定义时拷贝构造函数才能作为移动构造函数的备选项）

5.3 更明确的值类型体系

用临时变量初始化新对象时强制省略临时变量拷贝的提议的一个副作用就是，为了支持这个提议，*值类型体系* (value category) 进行了很多修改。

5.3.1 值类型体系

C++ 中的每一个表达式都有值类型。这个类型描述了表达式的值可以用来做什么。

历史上的值类型体系

C++ 以前只有从 C 语言继承而来的*左值* (lvalue) 和*右值* (rvalue)，根据赋值语句划分：

```
x = 42;
```

这里表达式 x 是左值因为它可以出现在赋值等号的左边，42 是右值因为它只能出现在表达式的右边。然而，当 ANSI-C 出现之后事情就变得更加复杂了，因为如果 x 被声明为 `const int` 的话它将不能出现在赋值号左边，但它仍然是一个（不能修改的）左值。

之后，C++11 又引入了可移动的对象，从语义上分析，可移动对象只能出现在赋值号右侧但它却可以被修改因为赋值号能移走它们的值。出于这个原因，类型*到期值* (xvalue) 被引入，原来的右值被重命名为*纯右值* (prvalue)。

从 C++11 起的值类型体系

自从 C++11 起，值类型的关系见图 5.1：我们有了核心的值类型体系 *lvalue(左值)*，*prvalue(纯右值)*(”pure rvalue”) 和 *xvalue(到期值)*(”eXpiring value”)。复合的值类型体系有 *glvalue(广义左值)*（”generalized lvalue”，它是 *lvalue* 和 *xvalue* 的复合）和 *rvalue(右值)*（*xvalue* 和 *prvalue* 的复合）。

lvalue(左值) 的例子有：

- 只含有单个变量、函数或成员的表达式

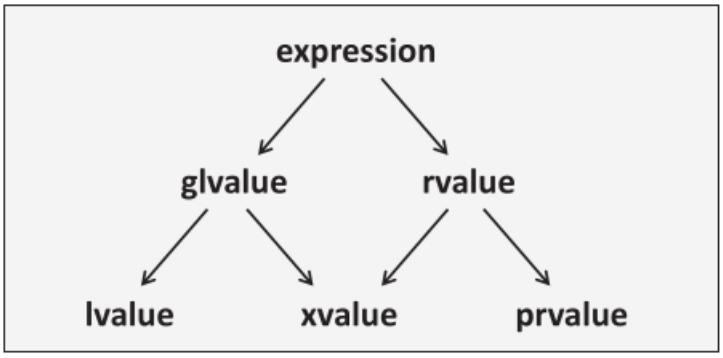

Figure 5.1: 从 C++11 起的值类型体系

- 只含有字符串字面量的表达式
- 内建的一元 * 运算符（解引用运算符）的结果
- 一个返回 *lvalue(左值)* 引用 (*type&*) 的函数的返回值

prvalue(纯右值) 的例子有：

- 除字符串字面量和用户自定义字面量之外的字面量组成的表达式
- 内建的一元 & 运算符（取地址运算符）的运算结果
- 内建的数学运算符的结果
- 一个返回值的函数的返回值
- lambda 返回值

xvalue(到期值) 的例子有：

- 一个返回右值引用 (*type&&*) 的函数的返回值（尤其是 `std::move()` 的返回值）
- 把一个对象转换为右值引用的操作的结果

简单来讲：

- 所有名称都是 *lvalue(左值)*。
- 所有用作表达式的字符串字面量是 *lvalue(左值)*。
- 所有其他的字面量（`4.2, true, nullptr`）是 *prvalue(纯右值)*。
- 所有临时对象（尤其是以值返回的对象）是 *prvalue(纯右值)*。
- `std::move()` 是一个 *xvalue(到期值)*

例如：

```
class X {
};
X v;
const X c;

void f(const X&);   // 接受任何值类型
void f(X&&);        // 只接受prvalue和xvalue，但是相比上边的版本是更好的匹配
```

```
f(v);                // 给第一个f()传递了一个可修改lvalue
f(c);                // 给第一个f()传递了不可修改的lvalue
f(X());              // 给第二个f()传递了一个prvalue
f(std::move(v));     // 给第二个f()传递了一个xvalue
```

值得强调的一点是严格来讲 glvalue(广义左值)、prvalue(纯右值)、xvalue(到期值) 是描述表达式的术语而不是描述值的术语（这意味着这些术语其实是误称）。例如，一个变量自身并不是左值，只含有这个变量的表达式才是左值：

```
int x = 3;  // 这里，x是一个变量，不是一个左值
int y = x;  // 这里，x是一个左值
```

在第一条语句中，3 是一个纯右值，用它初始化了变量（不是左值）x。在第二条语句中，x 是一个左值（该表达式的求值结果指向一个包含有数值 3 的对象）。左值 x 被转换为一个纯右值，然后用来初始化 y。

5.3.2 自从 C++17 起的值类型体系

C++17 再次明确了值类型体系，现在的值类型体系如图 5.2所示：

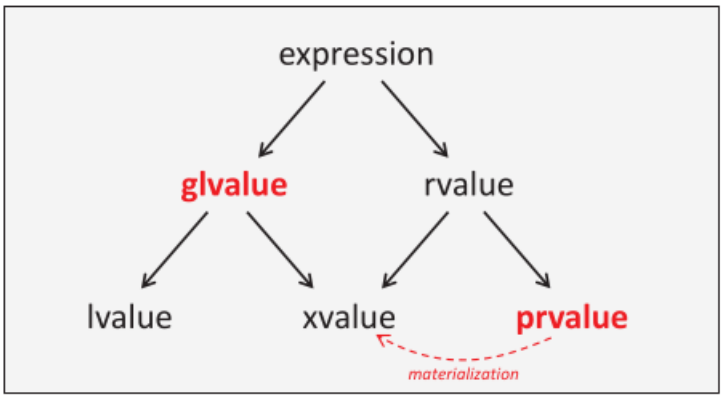

Figure 5.2: 自从 C++17 起的值类型体系

理解值类型体系的关键是现在广义上来说，我们只有两种类型的表达式：

- **glvaue:** 描述对象或函数位置的表达式
- **prvalue:** 用于初始化的表达式

而 **xvalue** 可以认为是一种特殊的位置，它代表一个资源可以被回收利用的对象（通常是因为该对象的生命周期即将结束）。

C++17 引入了一个新的术语：（临时对象的）实质化 (materialization)，目前 prvalue 就是一种临时对象。因此，临时对象实质化转换 (temporary materialization conversion) 是一种 prvalue 到 xvalue 的转换。

在任何情况下 prvalue 出现在需要 glvalue（lvalue 或者 xvalue）的地方都是有效的，此时会创建一个临时对象并用该 prvalue 来初始化（注意 prvalue 主要就是用来初始化的值）。然后该 prvalue 会被临时创建的 xvalue 类型的临时对象替换。因此上面的例子严格来讲是这样的：

```
void f(const X& p); // 接受一个任何值类型体系的表达式
                    // 但实际上需要一个glvalue
f(X());             // 传递了一个prvalue，该prvalue实质化为xvalue
```

因为这个例子中的 f() 的形参是一个引用，所以它需要 glvaue 类型的实参。然而，表达式 X() 是一个 prvalue。此时“临时变量实质化”规则会产生作用，表达式 X() 会“转换为”一个 xvalue 类型的临时对象。

注意实质化的过程中并没有创建新的/不同的对象。左值引用 p 仍然绑定到 xvalue 和 prvalue，尽管后者现在会转换为一个 xvalue。

因为 prvalue 不再是对象而是可以被用来初始化对象的表达式，所以当使用 prvalue 来初始化对象时不再需要 prvalue 是可移动的，进而省略临时变量拷贝的特性可以完美实现。我们现在只需要简单的传递初始值，然后它会被自动实质化来初始化新对象。[1]

5.4 未实质化的返回值传递

所有以值返回临时对象 (prvalue) 的过程都是在传递未实质化的返回值：

- 当我们返回一个非字符串字面量的字面量时：

```
int f1() {  // 以值返回int
    return 42;
}
```

- 当我们用 auto 或类型名作为返回类型并返回一个临时对象时：

```
auto f2() { // 以值返回退化的类型
    ...
    return MyType{...};
}
```

- 当使用 decltype(auto) 作为返回类型并返回临时对象时：

```
decltype(auto) f3() {   // 返回语句中以值返回临时对象
    ...
    return MyType{...}
}
```

注意当初始化表达式（此处是返回语句）是一个创建临时对象 (prvalue) 的表达式时 decltype(auto) 将会推导出值类型。因为我们在这些场景中都是以值返回一个 prvalue，所以我们完全不需要任何拷贝/移动。

5.5 后记

用临时变量初始化时强制省略拷贝由 Richard Smith 在https://wg21.link/p0135r0中首次提出。最终被接受的正式提案由 Richard Smith 发表于https://wg21.link/p0135r1。

[1] 感谢 Richard Smith 和 Graham Haynes 指出这一点

Chapter 6

lambda 表达式扩展

lambda 表达式是一个很大的成功，它最早在 C++11 中引入，在 C++14 中又引入了泛型 lambda。它允许我们将函数作为参数传递，这让我们能更轻易的指明一种行为。

C++17 扩展了 lambda 表达式的应用场景：

- 在常量表达式中使用（也就是在编译期间使用）
- 在需要当前对象的拷贝时使用（例如，当在不同的线程中调用 lambda 时）

6.1 `constexpr` lambda

自从 C++17 起，lambda 表达式会尽可能的隐式声明 `constexpr`。也就是说，任何只使用有效的编译期上下文（例如，只有字面量，没有静态变量，没有虚函数，没有 `try/catch`，没有 `new/delete` 的上下文）的 lambda 都可以被用于编译期。

例如，你可以使用一个 lambda 表达式计算参数的平方，并将计算结果用作 `std::array<>` 的大小，即使这是一个编译期的参数：

```
auto squared = [](auto val) {   // 自从C++17起隐式constexpr
    return val*val;
};
std::array<int, squared(5)> a;  // 自从C++17起OK => std::array<int, 25>
```

使用 `constexpr` 中不允许的特性将会使 lambda 失去成为 `constexpr` 的能力，不过你仍然可以在运行时上下文中使用 lambda：

```
auto squared2 = [](auto val) {  // 自从C++17起隐式constexpr
    static int calls = 0;   // OK，但会使该lambda不能成为constexpr
    ...
    return val*val;
};
std::array<int, squared2(5)> a;     // ERROR：在编译期上下文中使用了静态变量
std::cout << squared2(5) << '\n';   // OK
```

为了确定一个 lambda 是否能用于编译期，你可以将它声明为 `constexpr`：

```
auto squared3 = [](auto val) constexpr {  // 自从C++17起OK
    return val*val;
};
```

如果指明返回类型的话，看起来像下面这样：

```
auto squared3i = [](int val) constexpr -> int {  // 自从C++17起OK
    return val*val;
};
```

关于 constexpr 函数的规则也适用于 lambda：如果一个 lambda 在运行时上下文中使用，那么相应的函数体也会在运行时才会执行。

然而，如果在声明了 constexpr 的 lambda 内使用了编译期上下文中不允许出现的特性将会导致编译错误：[1]

```
auto squared4 = [](auto val) constexpr {
    static int calls = 0;  // ERROR：在编译期上下文中使用了静态变量
    ...
    return val*val;
};
```

一个隐式或显式的 constexpr lambda 的函数调用符也是 constexpr。也就是说，如下定义：

```
auto squared = [](auto val) {   // 自从C++17起隐式constexpr
    return val*val;
};
```

将会被转换为如下闭包类型 (closure type)：

```
class CompilerSpecificName {
  public:
    ...
    template<typename T>
    constexpr auto operator() (T val) const {
        return val*val;
    }
};
```

注意，这里自动生成的闭包类型的函数调用运算符自动声明为 constexpr。自从 C++17 起，如果 lambda 被显式或隐式地定义为 constexpr，那么生成的函数调用运算符将自动是 constexpr。注意如下定义：

```
auto squared1 = [](auto val) constexpr {  // 编译期lambda调用
    return val*val;
};
```

和如下定义：

```
constexpr auto squared2 = [](auto val) {  // 编译期初始化squared2
    return val*val;
};
```

[1] 不允许出现在编译期上下文中的特性有：静态变量、虚函数、try/catch、new/delete 等。

是不同的。

第一个例子中如果（只有）lambda 是 constexpr 那么它可以被用于编译期，但是 squared1 可能直到运行期才会被初始化，这意味着如果静态初始化顺序很重要那么可能导致问题。如果用 lambda 初始化的闭包对象是 constexpr，那么该对象将在程序开始时就初始化，但 lambda 可能还是只能在运行时使用。因此，可以考虑使用如下定义：

```
constexpr auto squared = [](auto val) constexpr {
    return val*val;
};
```

6.1.1 使用 constexpr lambda

这里有一个使用 constexpr lambda 的例子。假设我们有一个字符串的哈希函数，这个函数迭代字符串的每一个字符反复更新哈希值：[2]

```
auto hashed = [](const char* str) {
    std::size_t hash = 5381;    // 初始化哈希值
    while (*str != '\0') {
        hash = hash * 33 ^ *str++;  // 根据下一个字符更新哈希值
    }
    return hash;
};
```

使用这个 lambda，我们可以在编译期初始化不同字符串的哈希值，并定义为枚举：

```
enum Hashed { beer = hashed("beer"),
              wine = hashed("wine"),
              water = hashed("water"),
              ... };   // OK，编译期哈希
```

我们也可以在编译期计算 case 标签：

```
switch (hashed(argv[1])) {  // 运行时哈希
    case hashed("beer"):    // OK，编译期哈希
        ...
        break;
    case hashed("wine"):
        ...
        break;
    ...
}
```

注意，这里我们将在编译期调用 case 标签里的 hashed，而在运行期间调用 switch 表达式里的 hashed。

如果我们使用编译期 lambda 初始化一个容器，那么编译器优化时很可能在编译期就计算出容器的初始值（这里使用了 std::array 的类模板参数推导）：

```
std::array arr{ hashed("beer"),
                hashed("wine"),
                hashed(("water")};
```

[2] djb2 算法的源码见 http://www.cse.yorku.ca/~oz/hash.html。

你甚至可以在 hashed 函数里联合使用另一个 constexpr lambda。设想我们把 hashed 里根据当前哈希值和下一个字符值更新哈希值的逻辑定义为一个参数：

```
auto hashed = [](const char* str, auto combine) {
    std::size_t hash = 5381;
    while (*str != '\0') {
        hash = combine(hash, *str++);   // 用下一个字符更新哈希值
    }
    return hash;
};
```

这个 lambda 可以像下面这样使用：

```
constexpr std::size_t hv1{hashed("wine"), [](auto h, char c) {return h*33 + c;})};
constexpr std::size_t hv2{hashed("wine"), [](auto h, char c) {return h*33 ^ c;})};
```

这里，我们在编译期通过改变更新逻辑初始化了两个不同的”wine” 的哈希值。两个 hashed 都是在编译期调用。

6.2 向 lambda 传递 this 的拷贝

当在非静态成员函数里使用 lambda 时，你不能隐式获取对该对象成员的使用权。也就是说，如果你不捕获 this 的话你将不能在 lambda 里使用该对象的任何成员（即使你用 this-> 来访问也不行）：

```
class C {
  private:
    std::string name;
  public:
    ...
    void foo() {
        auto l1 = [] {std::cout << name << '\n';}; // ERROR
        auto l2 = [] {std::cout << this->name << '\n';}; // ERROR
        ...
    }
};
```

在 C++11 和 C++14 里，你可以通过值或引用捕获 this：

```
class C {
  private:
    std::string name;
  public:
    ...
    void foo() {
        auto l1 = [this] {std::cout << name << '\n';}; // OK
        auto l2 = [=] {std::cout << name << '\n';}; // OK
        auto l3 = [&] {std::cout << name << '\n';}; // OK
        ...
    }
};
```

然而，问题是即使是用拷贝的方式捕获 this 实质上获得的也是引用（因为只会拷贝 this 指针）。当 lambda 的生命周期比该对象的生命周期更长的时候，调用这样的函数就可能导致问题。比如一个极端的例子是在 lambda 中开启一个新的线程来完成某些任务，调用新线程时正确的做法是传递整个对象的拷贝来避免并发和生存周期的问题，而不是传递该对象的引用。另外有时候你可能只是简单的想向 lambda 传递当前对象的拷贝。

自从 C++14 起有了一个解决方案，但可读性和实际效果都比较差：

```
class C {
  private:
    std::string name;
  public:
    ...
    void foo() {
        auto l1 = [thisCopy=*this] { std::cout << thisCopy.name << '\n'; };
        ...
    }
};
```

例如，当使用了 = 或者 & 捕获了其他对象的时候你可能会在不经意间使用 this：

```
auto l1 = [&, thisCopy=*this] {
    thisCopy.name = "new name";
    std::cout << name << '\n'; // OOPS：仍然使用了原来的name
};
```

自从 C++17 起，你可以通过 *this 显式地捕获当前对象的拷贝：

```
class C {
  private:
    std::string name;
  public:
    ...
    void foo() {
        auto l1 = [*this] { std::cout << name << '\n'; };
        ...
    }
};
```

这里，捕获 *this 意味着该 lambda 生成的闭包将存储当前对象的一份拷贝。

你仍然可以在捕获 *this 的同时捕获其他对象，只要没有多个 this 的矛盾：

```
auto l2 = [&, *this] { ... };       // OK
auto l3 = [this, *this] { ... };    // ERROR
```

这里有一个完整的例子：

lang/lambdathis.cpp

```
#include <iostream>
#include <string>
#include <thread>
```

```
class Data {
private:
    std::string name;
public:
    Data(const std::string& s) : name(s) {
    }
    auto startThreadWithCopyOfThis() const {
        // 开启并返回新线程，新线程将在3秒后使用this
        using namespace std::literals;
        std::thread t([*this] {
            std::this_thread::sleep_for(3s);
            std::cout << name << '\n';
        });
        return t;
    }
};

int main()
{
    std::thread t;
    {
        Data d{"c1"};
        t = d.startThreadWithCopyOfThis();
    }   // d不再有效
    t.join();
}
```

lambda 里捕获了 `*this`，因此传递进 lambda 的是一份拷贝。因此，即使在 d 被销毁之后使用捕获的对象也没有问题。

如果我们使用 `[this]`、`[=]` 或者 `[&]` 捕获 `this`，那么新线程将会陷入未定义行为，因为当线程中打印 `name` 的时候将会使用一个已经销毁的对象的成员。

6.3 以常量引用捕获

通过使用一个新的库工具，现在也可以以常量引用捕获。

6.4 后记

`constexpr` lambda 由 Faisal Vali、Ville Voutilainen 和 Gabriel Dos Reis 在https://wg21.link/n4487中首次提出。最终被接受的正式提案由 Faisal Vali、Jens Maurer、Richard Smith 发表于https://wg21.link/p0170r1。

Chapter 7

新属性和属性特性

自从C++11起，就可以指明属性(attribute)（允许或者禁用某些警告的注解）。C++17引入了新的属性，另外现在属性也可以在其他一些地方使用，也许能带来一些便利。

7.1 [[nodiscard]]属性

新属性[[nodiscard]]可以鼓励编译器在某个函数的返回值未被使用时给出警告（这并不意味着编译器一定要给出警告）。[[nodiscard]]通常应该用于防止某些因为返回值未被使用导致的不当行为。这些不当行为可能是（译者注：请配合下边的例子理解这些不当行为）：

- **内存泄露**，例如返回值中含有动态分配的内存，但并未使用。
- **未知的或出乎意料的行为**，例如因为没有使用返回值而导致了一些奇怪的行为。
- **不必要的开销**，例如因为返回值没被使用而进行了一些无意义的行为。

这里有一些该属性发挥所用的例子：

- 申请资源但自身并不释放，而是将资源返回等待其他函数释放的函数应该被标记为[[nodiscard]]。一个典型的例子是申请内存的函数，例如malloc()函数或者分配器的allocate()成员函数。

  然而，注意有些函数可能会返回一个无需再处理的值。例如，程序员可能会用0字节调用C函数realloc()来释放内存，这种情况下的返回值无需之后调用free()函数释放。因此，如果把realloc()函数标记为[[nodiscard]]将会适得其反。
- 有时如果没有使用返回值将导致函数行为和预期不同，一个很好的例子是std::async()（C++11引入）。std::async()会在后台异步地执行一个任务并返回一个可以用来等待任务执行结束的句柄（也可以通过它获取返回值或者异常）。然而，如果返回值没有被使用的话该调用将变成同步的调用，因为在启动任务的语句结束之后未被使用的返回值的析构函数会立即执行，而析构函数会阻塞等待任务运行结束。因此，不使用返回值导致的结果与std::async()的目的完全矛盾。将std::async()标记为[[nodiscard]]可以让编译器给出警告。
- 另一个例子是成员函数empty()，它的作用是检查一个对象（容器/字符串）是否没有元素。程序员经常误用该函数来“清空”容器：

```
cont.empty();
```

这种对 empty() 的误用并没有使用返回值，所以 [[nodiscard]] 可以检查出这种误用：

```
class MyContainer {
    ...
  public:
    [[nodiscard]] bool empty() const noexcept;
    ...
};
```

这里的属性标记可以帮助检查这种逻辑错误。

如果因为某些原因你不想使用一个被标记为 [[nodiscard]] 的函数的返回值，你可以把返回值转换为 void：

```
(void)coll.empty(); // 禁止[[nodiscard]]警告
```

注意如果成员函数被覆盖或者隐藏时基类中标记的属性不会被继承：

```
struct B {
    [[nodiscard]] int* foo();
};

struct D : B {
    int* foo();
};

B b;
b.foo();        // 警告
(void)b.foo();  // 没有警告

D d;
d.foo();        // 没有警告
```

因此你需要给派生类里相应的成员函数再次标记 [[nodiscard]]（除非有某些原因导致你不想在派生类里确保返回值必须被使用）。

你可以把属性标记在函数前的所有修饰符之前，也可以标记在函数名之后：

```
class C {
    ...
    [[nodiscard]] friend bool operator== (const C&, const C&);
    friend bool operator!= [[nodiscard]] (const C&, const C&);
};
```

把属性放在 friend 和 bool 之间或者 bool 和 operator== 之间是错误的。

尽管这个特性从 C++17 起引入，但它还没有在标准库中使用。因为这个提案出现的太晚了，所以最应该需要它的 std::async() 也还没有使用它。不过这里讨论的所有例子，将在下一次 C++ 标准中实现（见 C++20 中通过的https://wg21.link/p0600r1提案）。

为了保证代码的可移植性，你应该使用 [[nodiscard]] 而不是一些不可移植的方案（例如 gcc 和 clang 的 [[gnu:warn_unused_result]] 或者 Visual C++ 的 _Check_return_）。

当定义 new() 运算符时，你应该用 [[nodiscard]] 对该函数进行标记，例如定义一个追踪所有 new 调用的头文件。

7.2 [[maybe_unused]] 属性

新的属性 [[maybe_unused]] 可以避免编译器在某个变量未被使用时发出警告。新的属性可以应用于类的声明、使用 typedef 或者 using 定义的类型、一个变量、一个非静态数据成员、一个函数、一个枚举类型、一个枚举值等场景。

例如其中一个作用是定义一个可能不会使用的参数：

```
void foo(int val, [[maybe_unused]] std::string msg)
{
#ifdef DEBUG
    log(msg);
#endif
    ...
}
```

另一个例子是定义一个可能不会使用的成员：

```
class MyStruct {
    char c;
    int i;
    [[maybe_unused]] char makeLargerSize[100];
    ...
};
```

注意你不能在一条语句上应用 [[maybe_unused]]。也就是说，你不能直接用 [[maybe_unused]] 来抵消 [[nodiscard]] 的作用：[1]

```
[[nodiscard]] void* foo();
int main()
{
    foo();  // 警告：返回值没有使用
    [[maybe_unused]] foo(); // 错误：maybe_unused不允许出现在此
    [[maybe_unused]] auto x = foo(); // OK
}
```

7.3 [[fallthrough]] 属性

新的属性 [[fallthrough]] 可以避免编译器在 switch 语句中某一个标签缺少 break 语句时发出警告。例如：

```
void commentPlace(int place)
{
    switch (place) {
        case 1:
            std::cout << "very ";
            [[fallthrough]];
```

[1] 感谢 Roland Bock 指出这一点

```
            case 2:
                std::cout << "well\n";
                break;
            default:
                std::cout << "OK\n";
                break;
        }
    }
```

这个例子中参数为1时将输出：

```
very well
```

`case 1`和`case 2`中的语句都会被执行。注意这个属性必须被用作单独的语句，还要有分号结尾。另外在`switch`语句的最后一个分支不能使用它。

7.4 通用的属性扩展

自从C++17起下列有关属性的通用特性变得可用：

1. 属性现在可以用来标记命名空间。例如，你可以像下面这样弃用一个命名空间：

```
namespace [[deprecated]] DraftAPI {
    ...
}
```

这也可以应用于内联的和匿名的命名空间。

2. 属性现在可以标记枚举子（枚举类型的值）。例如你可以像下面这样引入一个新的枚举值作为某个已有枚举值（并且现在已经被废弃）的替代：

```
enum class City { Berlin = 0,
    NewYork = 1,
    Mumbai = 2,
    Bombay [[deprecated]] = Mumbai,
    ... };
```

这里`Mumbai`和`Bombay`代表同一个城市的数字码，但使用`Bombay`已经被标记为废弃的。注意对于枚举值，属性被放置在标识符之后。

3. 用户自定义的属性一般应该定义在自定义的命名空间中。现在可以使用`using`前缀来避免为每一个属性重复输入命名空间。也就是说，如下代码：

```
[[MyLib::WebService, MyLib::RestService, MyLib::doc("html")]] void foo();
```

可以被替换为

```
[[using MyLib: WebService, RestService, doc("html")]] void foo();
```

注意在使用了`using`前缀时重复命名空间将导致错误：

```
[[using MyLib: MyLib::doc("html")]] void foo(); // ERROR
```

7.5 后记

三个新属性由 Andrew Tomazos 在https://wg21.link/p0068r0中首次提出。

[[nodiscard]] 属性最终被接受的提案由 Andrew Tomazos 发表于 https://wg21.link/p0189r1。

[[maybe_unused]] 属性最终被接受的提案由 Andrew Tomazos 发表于 https://wg21.link/p0212r1。

[[fallthrough]] 属性最终被接受的提案由 Andrew Tomazos 发表于 https://wg21.link/p0188r1。

允许为命名空间和枚举值标记属性由 Richard Smith 在https://wg21.link/n4196 中首次提出。该特性最终被接受的提案由 Richard Smith 发表于https://wg21.link/n4266。

属性的 using 前缀由 J. Daniel Garcia、Luis M. Sanchez、Massimo Torquati、Marco Danelutto、Peter Sommerlad 在https://wg21.link/p0028r0 中首次提出。最终被接受的提案由 J. Daniel Garcia 和 Daveed Vandevoorde 发表于 https://wg21.link/P0028R4。

Chapter 8

其他语言特性

在C++17中还有一些微小的核心语言特性的变更，将在这一章中介绍。

8.1 嵌套命名空间

自从2003年第一次提出，到现在C++标准委员会终于同意了以如下方式定义嵌套的命名空间：

```
namespace A::B::C {
    ...
}
```

等价于：

```
namespace A {
    namespace B {
        namespace C {
            ...
        }
    }
}
```

注意目前还没有对嵌套内联命名空间的支持。这是因为`inline`是作用于最内层还是整个命名空间还有歧义。（两种情况都很有用）

8.2 有定义的表达式求值顺序

许多C++书籍里的代码如果按照直觉来看似乎是正确的，但严格上讲它们有可能导致未定义的行为。一个简单的例子是在字符串中替换一个字串：

```
std::string s ="I heard it even works if you don't believe";
s.replace(0, 8, "").replace(s.find("even", 4, "sometimes")
                   .replace(s.find("you don't"), 9, "I");
```

通常的假设是前8个字符被空串替换，“even”被“sometimes”替换，“you don’t”被“I”替换，因此结果是：

```
it sometimes works if I believe
```

然而在C++17之前最后的结果实际上并没有任何保证。因为查找子串位置的`find()`函数可能在需要它们的返回值之前的任意时刻调用，而不是像直觉中的那样从左向右按顺序执行表达式。事实上，所有的`find()`调用可能在执行第一次替换之前就全部执行，因此结果变为：

```
it even worsometimesf youIlieve
```

其他的结果也是有可能的：

```
it sometimes workIdon’ t believe
it even worsometiIdon’ t believe
```

作为另一个例子，考虑使用输出运算符打印几个相互依赖的值：

```
std::cout << f() << g() << h();
```

通常的假设是依次调用`f()`、`g()`、`h()`函数。然而这个假设实际上是错误的。`f()`、`g()`、`h()`有可能以任意顺序调用，当这三个函数的调用顺序会影响返回值的时候可能就会出现奇怪的结果。

作为一个具体的例子，直到C++17之前，下面代码的行为都是未定义的：

```
i = 0;
std::cout << ++i << ' ' << --i << '\n';
```

在C++17之前，它可能会输出`1 0`，但也可能输出`0 -1`或者`0 0`，这和变量`i`是`int`还是用户自定义类型无关（不过对于基本类型，编译器一般会在这种情况下给出警告）。

为了解决这种未定义的问题，C++17标准重新定义了一些运算符的的求值顺序，因此这些运算符现在有了固定的求值顺序：

- 对于运算

  ```
  e1 [ e2 ]
  e1 . e2
  e1 .* e2
  e1 ->* e2
  e1 << e2
  e1 >> e2
  ```

  *e1*现在保证一定会在*e2*之前求值，因此求值顺序是从左向右。然而，注意同一个函数调用中的不同参数的计算顺序仍然是未定义的。也就是说：

  ```
  e1.f(a1, a2, a3);
  ```

  中的`e1.f`保证会在`a1`、`a2`、`a3`之前求值。但`a1`、`a2`、`a3`的求值顺序仍是未定义的。
- 所有的赋值运算

  ```
  e2 = e1
  e2 += e1
  e2 *= e1
  ...
  ```

  中右侧的`e1`现在保证一定会在左侧的`e2`之前求值。
- 最后，类似于如下的`new`表达式

```
        new Type(e)
```

中保证内存分配的操作在对 e 求值之前发生。新的对象的初始化操作保证在第一次使用该对象之前完成。所有这些保证适用于所有基本类型和自定义类型。

因此，自从 C++17 起

```
std::string s ="I heard it even works if you don't believe";
s.replace(0, 8, "").replace(s.find("even"), 4, "always")
    .replace(s.find("don't believe"), 13, "use C++17");
```

保证将会把 s 的值修改为：

```
it always works if you use C++17
```

因为现在每个 find() 之前的替换操作现在都保证会在 find() 调用之前完成。

另一个例子，如下语句：

```
i = 0;
std::cout << ++i << ' ' << --i << '\n';
```

对于任意类型的 i 都保证输出是 1 0。

然而，其他大多数运算符的运算顺序仍然是未知的。例如：

```
i = i++ + i;    // 仍然是未定义的行为
```

这里，最右侧的 i 可能在 i 自增之前求值也可能在自增之后求值。

新的表达式求值顺序的另一个应用是在参数之前插入空格的函数。

向后的不兼容性

新的有定义的求值顺序可能会影响现有程序的输出。例如，考虑如下程序：

lang/evalexcept.cpp

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

void print10elems(const std::vector<int>& v) {
    for (int i = 0; i < 10; ++i) {
        std::cout << "value: " << v.at(i) << '\n';
    }
}

int main()
{
    try {
        std::vector<int> vec{7, 14, 21, 28};
        print10elems(vec);
    }
    catch (const std::exception& e) {   // 处理标准异常
        std::cerr << "EXCEPTION: " << e.what() << '\n';
```

```
    }
    catch (...) {   // 处理任何其他异常
        std::cerr << "EXCEPTION of unknown type\n";
    }
}
```

因为这个程序中的 vector<> 只有 4 个元素，因此在 print10elems() 的循环中使用无效的索引调用 at() 时将会抛出异常：在 C++17 之前，输出可能是：

```
value: 7
value: 14
value: 21
value: 28
EXCEPTION: ...
```

因为 at() 允许在输出 value: 之前调用，所以当索引错误时可以跳过开头的 value: 输出。[1]

自从 C++17 以后，输出保证是：

```
value: 7
value: 14
value: 21
value: 28
value: EXCEPTION: ...
```

因为现在 value: 的输出保证在 at() 调用之前。

8.3 更宽松的用整型初始化枚举值的规则

对于一个有固定基础类型的枚举类型变量，自从 C++17 开始可以用一个整型值直接进行列表初始化。这可以用于带有明确类型的无作用域枚举和所有有作用域的枚举，因为它们都有默认的基础类型：

```
// 带有明确基础类型的无作用域枚举类型
enum MyInt : char { };
MyInt i1{42};       // 自从C++17起OK（C++17以前ERROR）
MyInt i2 = 42;      // 仍然ERROR
MyInt i3(42);       // 仍然ERROR
MyInt i4 = {42};    // 仍然ERROR

// 带有默认基础类型的有作用域枚举
enmu class Weekday { mon, tue, wed, thu, fri, sat, sun };
Weekday s1{0};      // 自从C++17起OK（C++17以前ERROR）
Weekday s2 = 0;     // 仍然ERROR
Weekday s3(0);      // 仍然ERROR
Weekday s4 = {0};   // 仍然ERROR
```

如果 Weekday 有明确的基础类型的话结果完全相同：

[1] 较旧版本的 GCC 或者 Visual C++ 的行为就是这样的。

```
// 带有明确基础类型的有作用域枚举
enum class Weekday : char { mon, tue, wed, thu, fri, sat, sun };
Weekday s1{0};      // 自从C++17起OK（C++17以前ERROR）
Weekday s2 = 0;     // 仍然ERROR
Weekday s3(0);      // 仍然ERROR
Weekday s4 = {0};   // 仍然ERROR
```

对于没有明确基础类型的无作用域枚举类型（没有 class 的 enum），你仍然不能使用列表初始化：

```
enum Flag { bit1=1, bit2=2, bit3=4 };
Flag f1{0};     // 仍然ERROR
```

注意列表初始化不允许窄化，所以你不能传递一个浮点数：

```
enum MyInt : char { };
MyInt i5{42.2}; // 仍然ERROR
```

一个定义新的整数类型的技巧是简单的定义一个以某个已有整数类型作为基础类型的枚举类型，就像上面例子中的 MyInt 一样。这个特性的动机之一就是为了支持这个技巧，如果没有这个特性，在不进行转换的情况下将无法初始化新的对象。

事实上自从 C++17 起标准库提供的 std::byte 就直接使用了这个特性。

8.4 修正 auto 类型的列表初始化

自从在 C++11 中引入了花括号统一初始化之后，每当使用 auto 代替明确类型进行列表初始化时就会出现一些意料之外的不一致的结果：

```
int x{42};       // 初始化一个int
int y{1, 2, 3};  // ERROR
auto a{42};      // 初始化一个std::initializer_list<int>
auto b{1, 2, 3}; // OK:初始化一个std::initializer_list<int>
```

这些直接使用列表初始化（没有使用 =）时的不一致行为现在已经被修复了。因此如下代码的行为变成了：

```
int x{42};       // 初始化一个int
int y{1, 2, 3};  // ERROR
auto a{42};      // 现在初始化一个int
auto b{1, 2, 3}; // 现在ERROR
```

注意这是一个**破坏性的更改 (breaking change)**，因为它可能导致很多代码的行为在无声无息中发生改变。因此，支持了这个变更的编译器现在即使在 C++11 模式下也会启用这个变更。对于主流编译器，接受这个变更的版本分别是 Visual Studio 2015，g++5，clang3.8。

注意当使用 auto 进行拷贝列表初始化（使用了 =）时仍然是初始化一个 std::initializer_list<>：

```
auto c = {42};      // 仍然初始化一个std::initializer_list<int>
auto d = {1, 2, 3}; // 仍然OK：初始化一个std::initializer_list<int>
```

因此，现在直接初始化（没有 =）和拷贝初始化（有 =）之间又有了显著的不同：

```
auto a{42};     // 现在初始化一个int
auto c = {42};  // 仍然初始化一个std::initializer_list<int>
```

这也是更推荐使用直接列表初始化（没有=的花括号初始化）的原因之一。

8.5 十六进制浮点数字面量

C++17 允许指定十六进制浮点数字面量（有些编译器甚至在 C++17 之前就已经支持）。当需要一个精确的浮点数表示时这个特性非常有用（如果直接用十进制的浮点数字面量不保证存储的实际精确值是多少）。

例如：

```
#include <iostream>
#include <iomanip>

int main()
{
    // 初始化浮点数
    std::initializer_list<double> values {
        0x1p4,          // 16
        0xA,            // 10
        0xAp2,          // 40
        5e0,            // 5
        0x1.4p+2,       // 5
        1e5,            // 100000
        0x1.86Ap+16,    // 100000
        0xC.68p+2,      // 49.625
    };

    // 分别以十进制和十六进制打印出值：
    for (double d : values) {
        std::cout << "dec: " << std::setw(6)
                  << std::defaultfloat << d << "  hex: "
                  << std::hexfloat << d << '\n';
    }
}
```

程序通过使用已有的和新增的十六进制浮点记号定义了不同的浮点数值。新的记号是一个以 2 为基数的科学记数法记号：

- 有效数字/尾数用十六进制书写
- 指数部分用十进制书写，表示乘以 2 的 n 次幂

例如 `0xAp2` 的值为 40（10×2^2）。这个值也可以被写作 `0x1.4p+5`，也就是 1.25×32（0.4 是十六进制的分数，等于十进制的 0.25，$2^5 = 32$）。

程序的输出如下：

```
dec:     16  hex: 0x1p+4
dec:     10  hex: 0x1.4p+3
dec:     40  hex: 0x1.4p+5
dec:      5  hex: 0x1.4p+2
dec:      5  hex: 0x1.4p+2
```

```
dec: 100000  hex: 0x1.86ap+16
dec: 100000  hex: 0x1.86ap+16
dec: 49.625  hex: 0x1.8dp+5
```

就像上例展示的一样，十六进制浮点数的记号很早就存在了，因为输出流使用的 `std::hexfloat` 操作符自从 C++11 起就已经存在了。

8.6 UTF-8 字符字面量

自从 C++11 起，C++ 就已经支持以 `u8` 为前缀的 UTF-8 字符串字面量。然而，这个前缀不能用于字符字面量。C++17 修复了这个问题，所以现在可以这么写：

```
auto c = u8'6'; // UTF-8编码的字符6
```

在 C++17 中，`u8'6'` 的类型是 `char`，在 C++20 中可能会变为 `char8_t`，因此这里使用 `auto` 会更好一些。

通过使用该前缀现在可以保证字符值是 UTF-8 编码。你可以使用所有的 7 位的 US-ASCII 字符，这些字符的 UTF-8 表示和 US-ASCII 表示完全相同。也就是说，`u8'6'` 也是有效的以 7 位 US-ASCII 表示的字符'6'（也是有效的 ISO Latin-1、ISO-8859-15、基本 Windows 字符集中的字符）。[2] 通常情况下你的源码字符被解释为 US-ASCII 或者 UTF-8 的结果是一样的，所以这个前缀并不是必须的。`c` 的值永远是 `54`（十六进制 `36`）。

这里给出一些背景知识来说明这个前缀的必要性：对于源码中的字符和字符串字面量，C++ 标准化了你可以使用的字符而不是这些字符的值。这些值取决于*源码字符值*。当编译器为源码生成可执行程序时它使用*运行字符集*。源码字符集几乎总是 7 位的 US-ASCII 编码，而运行字符集通常是相同的。这意味着在任何 C++ 程序中，所有相同的字符和字符串字面量（不管有没有 `u8` 前缀）总是有相同的值。

然而，在一些特别罕见的场景中并不是这样的。例如，在使用 EBCDIC 字符集的旧的 IBM 机器上，字符'6'的值将是 246（十六进制为 F6）。在一个使用 EBCDIC 字符集的程序中上面的字符 `c` 的值将是 246 而不是 54，如果在 UTF-8 编码的平台上运行这个程序可能会打印出字符 ö，这个字符在 ISO/IEC 8859-x 编码中的值为 246. 在这种情况下，这个前缀就是必须的。

注意 `u8` 只能用于单个字符，并且该字符的 UTF-8 编码必须只占一个字节。一个如下的初始化：

```
char c = u8'ö';
```

是不允许的，因为德语的曲音字符 ö 的 UTF-8 编码是两个字节的序列，分别是 195 和 182（十六进制为 C3 B6）。

因此，字符和字符串字面量现在接受如下前缀：

- u8 用于单字节 US-ASCII 和 UTF-8 编码
- u 用于两字节的 UTF-16 编码
- U 用于四字节的 UTF-32 编码
- L 用于没有指定编码，可能是两个或者四个字节的宽字符集

8.7 异常声明作为类型的一部分

自从 C++17 之后，异常处理声明变成了函数类型的一部分。也就是说，如下的两个函数的类型是不同的：

[2] ISO Latin-1 的正式命名为 ISO-8859-1，而为了包含欧元符号 € 引入的字符集 ISO-8859-15 也被命名为 ISO Latin-9。

```
void fMightThrow();
void fNoexcept() noexcept;  // 不同类型
```

在C++17之前这两个函数的类型是相同的。这样的一个问题就是如果把一个可能抛出异常的函数赋给一个保证不抛出异常的函数指针，那么调用时有可能会抛出异常：[3]

```
void (*fp)() noexcept;  // 指向不抛异常的函数的指针
fp = fNoexcept;         // OK
fp = fMightThrow;       // 自从C++17起ERROR
```

把一个不会抛出异常的函数赋给一个可能抛出异常的函数指针仍然是有效的：

```
void (*fp2)();      // 指向可能抛出异常的函数的指针
fp2 = fNoexcept;    // OK
fp2 = fMightThrow;  // OK
```

因此新的特性不会破坏使用了没有 noexcept 声明的函数指针的程序，但请注意现在不能再违反函数指针中的 noexcept 声明（这可能会善意的破坏现有的程序）。重载一个签名完全相同只有异常声明不同的函数是不允许的（就像不允许重载只有返回值不同的函数一样）：

```
void f3();
void f3(); noexcept;    // ERROR
```

注意其他的规则不受影响。例如，你仍然不能忽略基类中的 noexcept 声明：

```
class Base {
  public:
    virtual void foo() noexcept;
    ...
};

class Derived : public Base {
  public:
    void foo() override;    // ERROR：不能重载
    ...
};
```

这里，派生类中的成员函数 foo() 和基类中的 foo() 类型不同所以不能重载。这段代码不能通过编译，即使没有 override 修饰符代码也不能编译，因为我们不能用更宽松的异常声明重载。

使用传统的异常声明

当使用传统的 noexcept 声明时，函数的是否抛出异常取决于条件为 true 还是 false：

```
void f1();
void f2() noexcept;
void f3() noexcept(sizeof(int)<4);  // 和f1()或f2()的类型相同
void f4() noexcept(sizeof(int)>=4); // 和f3()的类型不同
```

这里 f3() 的类型取决于条件的值：

[3]这样看起来好像是一个错误，但至少之前 g++ 的确允许这种行为。

- 如果 `sizeof(int)` 返回 4 或者更大，签名等价于

```
void f3() noexcept(false); // 和f1()类型相同
```

- 如果 `sizeof(int)` 返回的值小于 4，签名等价于

```
void f3() noexcept(true);  // 和f2()类型相同
```

因为 `f4()` 的异常条件和 `f3()` 的恰好相反，所以 `f3()` 和 `f4()` 的类型总是不同（也就是说，它们中的一个肯定是可能抛异常，另一个不会抛异常）。

旧式的不抛异常的声明仍然有效但自从 C++11 起就被废弃：

```
void f5() throw();  // 和void f5() noexcept等价但已经被废弃
```

带参数的动态异常声明不再被支持（自从 C++11 起被废弃）：

```
void f6() throw(std::bad_alloc); // ERROR：自从C++17起无效
```

对泛型库的影响

将 `noexcept` 做为类型类型的一部分意味着会对泛型库产生一些影响。例如，下面的代码直到 C++14 是有效的，但从 C++17 起不能再通过编译：

lang/noexceptcalls.cpp

```
#include <iostream>

template<typename T>
void call(T op1, T op2)
{
    op1();
    op2();
}

void f1() {
    std::cout << "f1()\n";
}
void f2() noexcept {
    std::cout << "f2()\n";
}

int main()
{
    call(f1, f2);   // 自从C++17起ERROR
}
```

问题在于自从 C++17 起 `f1()` 和 `f2()` 的类型不再相同，因此在实例化模板函数 `call()` 时编译器无法推导出类型 `T`，

自从 C++17 起，你需要指定两个不同的模板参数来通过编译：

```
template<typename T1, typename T2>
void call(T1 op1, T2 op2)
{
    op1();
    op2();
}
```

现在如果你想重载所有可能的函数类型，那么你需要重载的数量将是原来的两倍。例如，对于标准库特征 `std::is_function<>` 的定义，主模板的定义如下，所以该模板匹配的参数类型 `T` 不可能是函数：

```
// 主模板（泛型类型T不是函数）：
template<typename T> struct is_function : std::false_type { };
```

模板从`std::false_type`派生，因此 `is_function<T>::value` 对任何类型 `T` 都会返回 `false`。

对于任何是函数的类型，存在从`std::true_type` 派生的部分特化版，因此成员 `value` 总是返回 `true`：

```
// 对所有函数类型的部分特化版
template<typename Ret, typename... Params>
struct is_function<Ret (Params...)> : std::true_type { };

template<typename Ret, typename... Params>
struct is_function<Ret (Params...) const> : std::true_type { };

template<typename Ret, typename... Params>
struct is_function<Ret (Params...) &> : std::true_type { };

template<typename Ret, typename... Params>
struct is_function<Ret (Params...) const &> : std::true_type { };
...
```

在 C++17 之前该特征总共有 24 个部分特化版本：因为函数类型可以用 `const` 和 `volatile` 修饰符修饰，另外还可能有左值引用 (&) 或右值引用 (&&) 修饰符，还需要重载可变参数列表的版本。

现在在 C++17 中部分特化版本的数量变为了两倍，因为还需要为所有版本添加一个带 `noexcept` 修饰符的版本：

```
...
// 对所有带有noexcept声明的函数类型的部分特化版本
template<typename Ret, typename... Params>
struct is_function<Ret (Params...) noexcept> : std::true_type { };

template<typename Ret, typename... Params>
struct is_function<Ret (Params...) const noexcept> : std::true_type { };

template<typename Ret, typename... Params>
struct is_function<Ret (Params...) & noexcept> : std::true_type { };

template<typename Ret, typename... Params>
struct is_function<Ret (Params...) const & noexcept> : std::true_type { };
...
```

那些没有实现 noexcept 重载的库可能在需要使用带有 noexcept 的函数的场景中不能编译通过了。

8.8 单参数 static_assert

自从 C++17 起，以前 static_assert() 需要的作为消息的参数变为可选的了。也就是说现在断言失败时输出的诊断信息完全依赖平台的实现。例如：

```
#include <type_traits>

template<typename t>
class C {
    // 自从C++11起OK
    static_assert(std::is_default_constructible<T>::value,
        "class C: elements must be default-constructible");

    // 自从C++17起OK
    static_assert(std::is_default_constructible_v<T>);
    ...
};
```

不带消息的新版本静态断言的示例也使用了类型特征后缀 _v。

8.9 预处理条件 __has_include

C++17 扩展了预处理，增加了一个检查某个头文件是否可以被包含的宏。例如：

```
#if __has_include(<filesystem>)
#  include <filesystem>
#  define HAS_FILESYSTEM 1
#elif __has_include(<experimental/filesystem>)
#  include <experimental/filesystem>
#  define HAS_FILESYSTEM 1
#  define FILESYSTEM_IS_EXPERIMENTAL 1
#elif __has_include("filesystem.hpp")
#  include "filesystem.hpp"
#  define HAS_FILESYSTEM 1
#  define FILESYSTEM_IS_EXPERIMENTAL 1
#else
#  define HAS_FILESYSTEM 0
#endif
```

当相应的 #include 指令有效时 __has_include(...) 会被求值为 1。其他的因素都不会影响结果（例如，相应的头文件是否已被包含过并不影响结果）。

另外，虽然求值为真可以说明相应的头文件确实存在但不能保证它的内容符号预期。它的内容可能是空的或者无效的。

__has_include 是一个纯粹的预处理指令。所以不能在源码里使用它：

```
if (__has_include(<filesystem>)) {  // ERROR
}
```

8.10 后记

嵌套命名空间定义由 Jon Jagger 于 2003 年在 https://wg21.link/n1524中首次提出。Robert Kawulak 于 2014 年在https://wg21.link/n4026中提出了新的提案。最终被接受的提案由 Robert Kawulak 和 Andrew Tomazos 发表于https://wg21.link/n4230。

有定义的表达式求值顺序由 Gabriel Dos Reis、Herb Sutter、Jonathan Caves 在 https://wg21.link/n4228中首次提出。最终被接受的提案由 Gabriel Dos Reis、Herb Sutter 和 Jonathan Caves 发表于https://wg21.link/p0145r3。

更宽松的用整型初始化枚举值的规则由 Gabriel Dos Reis 在https://wg21.link/p0138r0中首次提出。最终被接受的提案由 Gabriel Dos Reis 发表于 https://wg21.link/p0138r2。

修正 auto 类型的列表初始化由 Ville Voutilainen 在https://wg21.link/n3681以及 https://wg21.link/3912中首次提出。最终 auto 列表初始化的修正由 James Dennett 发表于https://wg21.link/n3681。

十六进制浮点数字面量最早由 Thomas Köppe 在https://wg21.link/p0245r0 中首次提出。最终被接受的提案由 Thomas Köppe 发表于https://wg21.link/p0245r1。

UTF-8 字符字面量是由 Richard Smith 在https://wg21.link/n4197中首次提出。最终被接受的提案由 Richard Smith 发表于https://wg21.link/n4267。

异常声明作为类型的一部分由 Jens Maurer 在https://wg21.link/n4320 中首次提出。最终被接受的提案由 Jens Maurer 发表于https://wg21.link/p0012r1。

单参数 static_assert的提案由 Walter E. Brown 发表于https://wg21.link/n3928。

预处理语句 __has_include()由 Clark Nelson 和 Richard Smith 在 https://wg21.link/p0061r0提案的某一部分中首次提出。最终被接受的提案由 Clark Nelson 和 Richard Smith 发表于https://wg21.link/p0061r1。

Part II

模板特性

这一部分介绍了 C++17 为泛型编程（即 template）提供的新的语言特性。

我们首先从类模板参数推导开始，这一特性只影响模板的使用。之后的章节会介绍为编写泛型代码（函数模板，类模板，泛型库等）的程序员提供的新特性。

Chapter 9

类模板参数推导

在C++17之前，你必须明确指出类模板的所有参数。例如，你不可以省略下面的`double`：

```
std::complex<double> c{5.1, 3.3};
```

也不可以省略下面的`std::mutex`：

```
std::mutex mx;
std::lock_guard<std::mutex> lg(mx);
```

自从C++17起必须指明类模板参数的限制被放宽了。通过使用*类模板参数推导*(class template argument deduction)(CTAD)，只要编译器能根据初始值*推导出*所有模板参数，那么就可以不指明参数。

例如：

- 你现在可以这么声明：

```
std::complex c{5.1, 3.3};   // OK：推导出std::complex<double>
```

- 你现在可以这么写：

```
std::mutex mx;
std::lock_guard lg{mx}; // OK：推导出std::lock_guard<std::mutex>
```

- 你现在甚至可以让容器来推导元素类型：

```
std::vector v1{1, 2, 3}; // OK：推导出std::vector<int>
std::vector v2{"hello", "world"}; // OK：推导出std::vector<const char*>
```

9.1 使用类模板参数推导

只要能根据初始值推导出所有模板参数就可以使用类模板参数推导。推导过程支持所有方式的初始化（只要保证初始化是有效的）：

```
std::complex c1{1.1, 2.2};  // 推导出std::complex<double>
std::complex c2(2.2, 3.3);  // 推导出std::complex<double>
std::complex c3 = 3.3;      // 推导出std::complex<double>
std::complex c4 = {4.4};    // 推导出std::complex<double>
```

因为 `std::complex` 只需要一个参数就可以初始化并推导出模板参数 `T`：

```
namespace std {
    template<typename T>
    class complex {
        constexpr complex(const T&re = T(), const T& im = T());
        ...
    }
};
```

所以 `c3` 和 `c4` 可以正确初始化。对于如下声明：

```
std::complex c1{1.1, 2.2};
```

编译器会查找到构造函数：

```
constexpr complex(const T& re = T(), const T& im = T());
```

并调用。因为两个参数都是 `double` 类型，因此编译器会推导出 `T` 的就是 `double` 并生成如下代码：

```
complex<double>::complex(const double& re = double(), const double& im = double());
```

注意推导的过程中模板参数必须没有歧义。也就是说，如下初始化代码不能通过编译：

```
std::complex c5{5, 3.3};    // ERROR：尝试将T推导为int和double
```

推导模板参数时不会使用隐式类型转换。

也可以对可变参数模板使用类模板参数推导。例如，对于一个如下定义的 `std::tuple`：

```
namespace std {
    template<typename... Types>
    class tuple {
      public:
        constexpr tuple(const Types&...);
        ...
    };
};
```

如下声明：

```
std::tuple t{42, 'x', nullptr};
```

将推导出类型 `std::tuple<int, char, std::nullptr_t>`。

你也可以推导非类型模板参数。例如，我们可以根据传入的参数同时推导数组的元素类型和元素数量：

```
template<typename T, int SZ>
class MyClass {
  public:
    MyClass (T(&)[SZ]) {
        ...
    }
};

MyClass mc("hello");    // 推导出T为const char，SZ为6
```

这里我们推导出 SZ 为 6 因为传入的字符串字面量有 6 个字符。[1]

你甚至可以推导用作基类的 lambda来实现重载或者推导auto 模板参数。

9.1.1 默认以拷贝方式推导

类模板参数推导过程中会首先尝试以拷贝的方式初始化。例如，首先初始化一个只有一个元素的 std::vector：

```
std::vector v1{42}; // 一个元素的vector<int>
```

然后使用这个 vector 初始化另一个 vector，推导时会解释为创建一个拷贝：

```
std::vector v2{v1}; // v2也是一个std::vector<int>
```

而不是创建一个只有一个元素的 vector<vector<int>>。

这个规则适用于所有形式的初始化：

```
std::vector v2{v1};          // v2也是vector<int>
std::vector v3(v1);          // v3也是vector<int>
std::vector v4 = {v1};       // v4也是vector<int>
auto v5 = std::vector{v1};   // v5也是vector<int>
```

注意这是花括号初始化总是把列表中的参数作为元素这一规则的一个例外。如果你传递一个只有一个 vector 的初值列来初始化另一个 vector，你将得到一个传入的 vector 的拷贝。然而，如果用多于一个元素的初值列来初始化的话就会把传入的参数作为元素并推导出其类型作为模板参数（因为这种情况下无法解释为创建拷贝）：

```
std::vector vv{v1, v2}; // vv是一个vector<vector<int>>
```

这引出了一个问题就是对可变参数模板使用类模板参数推导时会发生什么：

```
template<typename... Args>
auto make_vector(const Args&... elems) {
    return std::vector{elem...};
}

std::vector<int> v{1, 2, 3};
auto x1 = make_vector(v, v); // vector<vector<int>>
auto x2 = make_vector(v);    // vector<int>还是vector<vector<int>>?
```

目前不同的编译器会有不同的行为，这个问题还在讨论之中。

9.1.2 推导 lambda 的类型

通过使用类模板参数推导，我们可以用 lambda 的类型（确切的说是 lambda 生成的闭包的类型）作为模板参数来实例化类模板。例如我们可以提供一个泛型类，对一个任意回调函数进行包装并统计调用次数：

tmpl/classarglambda.hpp

```
#include <utility>  // for std::forward()
```

[1]注意构造函数里以引用作为参数是必须的。否则根据语言规则传入的字符数组将会退化为指针，然后将无法推导出 SZ。

```
template<typename CB>
class CountCalls
{
private:
    CB callback;    // 要调用的回调函数
    long calls = 0; // 调用的次数
public:
    CountCalls(CB cb) : callback(cb) {
    }
    template<typename... Args>
    decltype(auto) operator() (Args&&... args) {
        ++calls;
        return callback(std::forward<Args>(args)...);
    }
    long count() const {
        return calls;
    }
};
```

这里构造函数获取一个回调函数并进行包装，这样在初始化时会把参数的类型推导为 CB。例如，我们可以使用一个 lambda 作为参数来初始化一个对象：

```
CountCalls sc{[](auto x, auto y) { return x > y; }};
```

这意味着排序准则 sc 的类型将被推导为 CountCalls<*TypeOfTheLambda*>。这样，我们可以统计出排序准则被调用的次数：

```
std::sort(v.begin(), v.end(),   // 排序区间
          std::ref(sc));        // 排序准则
std::cout << "sorted with " << sc.count() << " calls\n";
```

这里包装过后的 lambda 被用作排序准则。注意这里必须要传递引用，否则 std::sort() 将会获取 sc 的拷贝作为参数，计数时只会修改该拷贝内的计数器。

然而，我们可以直接把包转后的 lambda 传递给 std::for_each()，因为该算法（非并行版本）最后会返回传入的回调函数，以便于获取回调函数最终的状态：

```
auto fo = std::for_each(v.begin(), v.end(), CountCalls{[](auto i) {
                                                std::cout << "elem: " << i << '\n';
                                            }});
std::cout << "output with " << fo.count() << " calls\n";
```

输出将会如下（排序准则调用次数可能会不同，因为 sort() 的实现可能会不同）：

```
sorted with 39 calls
elem: 19
elem: 17
elem: 13
elem: 11
elem: 9
```

```
elem: 7
elem: 5
elem: 3
elem: 2
output with 9 calls
```

如果计数器是原子的，你也可以使用并行算法:

```
std::sort(std::execution::par, v.begin(), v.end(), std::ref(sc));
```

9.1.3 没有类模板部分参数推导

注意，不像函数模板，类模板不能只指明一部分模板参数，然后指望编译器去推导剩余的部分参数。甚至使用 <> 指明空模板参数列表也是不允许的。例如：

```
template<typename T1, typename T2, typename T3 = T2>
class C
{
  public:
    C (T1 x = {}, T2 y = {}, T3 z = {}) {
        ...
    }
    ...
};

// 推导所有参数
C c1(22, 44.3, "hi");   // OK：T1是int，T2是double，T3是const char*
C c2(22, 44.3);         // OK：T1是int，T2和3是double
C c3("hi", "guy");      // OK：T1、T2、T3都是const char*

// 推导部分参数
C<string> c4("hi", "my");   // ERROR：只有T1显式指明
C<> c5(22, 44.3);           // ERROR：T1和T2都没有指明
C<> c6(22, 44.3, 42);       // ERROR：T1和T2都没有指明

// 指明所有参数
C<string, string, int> c7;   // OK：T1、T2是string，T3是int
C<int, string> c8(52, "my"); // OK：T1是int，T2、T3是string
C<string, string> c9("a", "b", "c");    // OK：T1、T2、T3都是string
```

注意第三个模板参数有默认值，因此只要指明了第二个参数就不需要再指明第三个参数。

如果你想知道为什么不支持部分参数推导，这里有一个导致这个决定的例子：

```
std::tuple<int> t(42, 43);  // 仍然ERROR
```

`std::tuple` 是一个可变参数模板，因此你可以指明任意数量的模板参数。在这个例子中，并不能判断出只指明一个参数是一个错误还是故意的。

不幸的是，不支持部分参数推导意味着一个常见的编码需求并没有得到解决。我们仍然不能简单的使用一个 lambda 作为关联容器的排序准则或者无序容器的 hash 函数：

```
std::set<Cust> coll([] (const Cust& x, const Cust& y) { // 仍然ERROR
                        return x.getName() > y.getName();
                    });
```

我们仍然不得不指明 lambda 的类型。例如：

```
auto sortcrit = [](const Cust& x, const Cust& y) {
                    return x.getName() > y.getName();
                };
std::set<Cust, decltype(sortcrit)> coll(sortcrit); // OK
```

仅仅指明类型是不行的，因为容器初始化时会尝试用给出的 lambda 类型创建一个 lambda。但这在 C++17 中是不允许的，因为默认构造函数只有编译器才能调用。在 C++20 中如果 lambda 不需要捕获任何东西的话这将成为可能。

9.1.4 使用类模板参数推导代替快捷函数

原则上讲，通过使用类模板参数推导，我们可以摆脱已有的几个快捷函数模板，这些快捷函数的作用其实就是根据传入的参数实例化相应的类模板。

一个明显的例子是 `std::make_pair()`，它可以帮助我们避免指明传递进入的参数的类型。例如，在如下声明之后：

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

我们可以这样：

```
auto p = std::make_pair(v.begin(), v.end());
```

而不需要写：

```
std::pair<typename std::vector<int>::iterator, typename std::vector<int>::iterator>
    p(v.begin(), v.end());
```

现在这种场景已经不再需要 `std::make_pair()` 了，我们可以简单的写为：

```
std::pair p(v.begin(), v.end());
```

或者：

```
std::pair p{v.begin(), v.end());
```

然而，从另一个角度来看 `std::make_pair()` 也是一个很好的例子，它演示了有时便捷函数的作用不仅仅是推导模板参数。事实上 `std::make_pair()` 会使传入的参数退化（在 C++03 中以值传递，自从 C++11 起使用特征）。这样会导致字符串字面量的类型（字符数组）被推导为 `const char*`：

```
auto q = std::make_pair("hi", "world"); // 推导为指针的pair
```

这个例子中，q 的类型为 `std::pair<const char*, const char*>`。

使用类模板参数推导可能会让事情变得更加复杂。考虑如下这个类似于 `std::pair` 的简单的类的声明：

```
template<typename T1, typename T2>
struct Pair1 {
    T1 first;
```

```
    T2 second;
    Pair1(const T1& x, const T2& y) : first{x}, second{y} { }
};
```

这里元素以引用传入，根据语言规则，当以引用传递参数时模板参数的类型不会退化。因此，当调用：

```
Pair1 p1{"hi", "world"}; // 推导为不同大小的数组的pair，但是……
```

T1 被推导为 char[3]，T2 被推导为 char[6]。原则上讲这样的推导是有效的。然而，我们使用了 T1 和 T2 来声明成员 first 和 second，因此它们被声明为：

```
char first[3];
char second[6];
```

然而使用一个左值数组来初始化另一个数组是不允许的。它类似于尝试编译如下代码：

```
const char x[3] = "hi";
const char y[6] = "world";
char first[3] {x};  // ERROR
char second[6] {y}; // ERROR
```

注意如果我们声明参数时以值传参就不会再有这个问题：

```
tempalte<typename T1, typename T2>
struct Pair2 {
    T1 first;
    T2 second;
    Pair2(T1 x, T2 y) : first{x}, second{y} { }
};
```

如果我们像下面这样创建新对象：

```
Pair2 p2{"hi", "world"}; // 推导为指针的pair
```

T1 和 T2 都会被推导为 const char*。

然而，因为 std::pair<> 的构造函数以引用传参，所以下面的初始化正常情况下应该不能通过编译：

```
std::pair p{"hi", "world"}; // 看似会推导出不同大小的数组的pair，但是……
```

然而你，事实上它能通过编译，因为 std::pair<> 定义推导指引，我们将在下一小节讨论它。

9.2 推导指引

你可以定义特定的推导指引来给类模板参数添加新的推导或者修正构造函数定义的推导。例如，你可以定义无论何时推导 Pair3 的模板参数，推导的行为都好像参数是以值传递的：

```
template<typename T1, typename T2>
struct Pair3 {
    T1 first;
    T2 second;
    Pair3(const T1& x, const T2& y) : first{x}, second{y} { }
};
```

```
// 为构造函数定义的推导指引
tempalte<typename T1, typename T2>
Pair3(T1, T2) -> Pair3<T1, T2>;
```

在->的左侧我们声明了我们想要推导什么。这里我们声明的是使用两个以值传递且类型分别为T1和T2的对象创建一个Pair3对象。在->的右侧，我们定义了推导的结果。在这个例子中，Pair3以类型T1和T2实例化。

你可能会说这是构造函数已经做到的事情。然而，构造函数是以引用传参，两者是不同的。一般来说，不仅是模板，所有以值传递的参数都会退化，而以引用传递的参数不会退化。退化意味着原生数组会转换为指针，并且顶层的修饰符例如const或者引用将会被忽略。

如果没有推导指引，对于如下声明：

```
Pair3 p3{"hi", "world"};
```

参数x的类型是const char(&)[3]，因此T1被推导为char[3]，参数y的类型是const char(&)[6]，因此T2被推导为char[6]。

有了推导指引饮后，模板参数就会退化。这意味着传入的数组或者字符串字面量会退化为相应的指针类型。现在，如下声明：

```
Pair3 p3{"hi", "world"};
```

推导指引会发挥作用，因此会以值传参。因此，两个类型都会退化为const char*，然后被用作模板参数推导的结果。上面的声明和如下声明等价：

```
Pair3<const char*, const char*> p3{"hi", "world"};
```

注意构造函数仍然以引用传参。推导指引只和模板参数的推导相关，它与推导出T1和T2之后实际调用的构造函数无关。

9.2.1 使用推导指引强制类型退化

就像上一个例子展示的那样，这种重载的推导规则的一个非常有用的用途就是确保模板参数T在推导时发生退化。考虑如下的一个经典的类模板：

```
template<typename T>
struct C {
    C(const T&) {
    }
    ...
};
```

这里，如果我们传递一个字符串字面量"hello"，传递的类型将是const char(&)[5]，因此T被推导为char[6]：

```
C x{"hello"};   // T被推导为char[6]
```

原因是当参数以引用传递时模板参数不会退化为相应的指针类型。

通过使用一个简单的推导指引：

```
template<typename T> C(T) -> C<T>;
```

我们就可以修正这个问题：

```
C x{"hello"};   // T被推导为const char*
```

推导指引以值传递参数因此"hello"的类型T会退化为const char*。

因为这一点，任何构造函数里传递引用作为参数的模板类都需要一个相应的推导指引。C++标准库中为pair和tuple提供了相应的推导指引。

9.2.2 非模板推导指引

推导指引并不一定是模板，也不一定应用于构造函数。例如，为下面的结构体添加的推导指引也是有效的：

```
template<typename T>
struct S {
    T val;
};

S(const char*) -> S<std::string>;   // 把S<字符串字面量>映射为S<std::string>
```

这里我们创建了一个没有相应构造函数的推导指引。推导指引被用来推导参数T，然后结构体的模板参数就相当于已经被指明了。

因此，下面所有初始化代码都是正确的，并且都会把模板参数T推导为std::string：

```
S s1{"hello"};      // OK，等同于S<std::string> s1{"hello"};
S s2 = {"hello"};   // OK，等同于S<std::string> s2 = {"hello"};
S s3 = S{"hello"};  // OK，两个S都被推导为S<std::string>
```

因为传入的字符串字面量能隐式转换为std::string，所以上面的初始化都是有效的。

注意聚合体需要列表初始化。下面的代码中参数推导能正常工作，但会因为没有使用花括号导致初始化错误：

```
S s4 = "hello"; // ERROR：不能不使用花括号初始化聚合体
S s5("hello");  // ERROR：不能不使用花括号初始化聚合体
```

9.2.3 推导指引与构造函数冲突

推导指引会和类的构造函数产生竞争。类模板参数推导时会根据重载情况选择最佳匹配的构造函数/推导指引。如果一个构造函数和一个推导指引匹配程度相同，那么将会优先使用推导指引。

考虑如下定义：

```
template<typename T>
struct C1 {
    C1(const T&) {
    }
};
C1(int)->C1<long>;
```

当传递一个int时将会使用推导指引，因为根据重载规则它的匹配度更高。[2] 因此，T被推导为long：

[2]非模板函数的匹配度比模板函数更高，除非其他因素的影响更大。

```
C1 x1{42};  // T被推导为long
```

然而，如果我们传递一个char，那么构造函数的匹配度更高（因为不需要类型转换），这意味着T会被推导为char：

```
C1 x3{'x'}; // T被推导为char
```

在重载规则中，以值传参和以引用传参的匹配度相同的。然而在相同匹配度的情况下将优先使用推导指引。因此，通常会把推导指引定义为以值传参（这样做还有类型退化的优点）。

9.2.4 显式推导指引

推导指引可以用explicit声明。当出现explicit不允许的初始化或转换时这一条推导指引就会被忽略。例如：

```
template<typename T>
struct S {
    T val;
};

explicit S(const char*) -> S<std::string>;
```

如果用拷贝初始化（使用=）将会忽略这一条推导指引。这意味着下面的初始化是无效的：

```
S s1 = {"hello"};   // ERROR（推导指引被忽略，因此是无效的）
```

直接初始化或者右侧显式推导的方式仍然有效：

```
S s2{"hello"};  // OK，等同于S<std::string> s2{"hello"};
S s3 = S{"hello"};   // OK
S s4 = {S{"hello"}}; // OK
```

另一个例子如下：

```
template<typename T>
struct Ptr
{
    Ptr(T) { std::cout << "Ptr(T)\n"; }
    template<typename U>
    Ptr(U) { std::cout << "Ptr(U)\n"; }
}

template<typename T>
explicit Ptr(T) -> Ptr<T*>;
```

上面的代码会产生如下结果：

```
Ptr p1{42};     // 根据推导指引推导出Ptr<int*>
Ptr p2 = 42;    // 根据构造函数推导出Ptr<int>
int i = 42;
Ptr p3{&i};     // 根据推导指引推导出Ptr<int**>
Ptr p4 = &i;    // 根据构造函数推导出Ptr<int*>
```

9.2.5 聚合体的推导指引

泛型聚合体中也可以使用推导指引，这样才能支持类模板参数推导。例如，对于：

```
template<typename T>
struct A {
    T val;
};
```

在没有推导指引的情况下尝试使用类模板参数推导会导致错误：

```
A i1{42};       // ERROR
A s1("hi");     // ERROR
A s2{"hi"};     // ERROR
A s3 = "hi";    // ERROR
A s4 = {"hi"};  // ERROR
```

你必须显式指明参数的类型 T：

```
A<int> i2{42};
A<std::string> s5 = {"hi"};
```

然而，如果有如下推导指引的话：

```
A(const char*) -> A<std::string>;
```

你就可以像下面这样初始化聚合体：

```
A s2{"hi"};     // OK
A s4 = {"hi"};  // OK
```

注意你仍然需要使用花括号（像通常的聚合体初始化一样）。否则，类型 T 能成功推导出来，但初始化会错误：

```
A s1("hi");     // ERROR：T是string，但聚合体不能初始化
A s3 = "hi";    // ERROR：T是string，但聚合体不能初始化
```

std::array 的推导指引是一个有关聚合体推导指引的进一步的例子。

9.2.6 标准推导指引

C++17 标准在标准库中引入了很多推导指引。

pair 和 tuple 的推导指引

正如在推导指引的动机中介绍的一样，std::pair 需要推导指引来确保类模板参数推导时会推导出参数的退化类型：

```
namespace std {
    template<typename T1, typename T2>
    struct pair {
        ...
        constexpr pair(const T1& x, const T2& y);   // 以引用传参
        ...
```

```
    };
    template<typename T1, typename T2>
    pair(T1, T2) -> pair<T1, T2>;   // 以值推导类型
}
```

因此，如下声明：

```
std::pair p{"hi", "wrold"}; // 参数类型分别为const char[3]和const char[6]
```

等价于：

```
std::pair<const char*, const char*> p{"hi", "world"};
```

可变参数类模板 `std::tuple` 也使用了相同的方法：

```
namespace std {
    template<typename... Types>
    class tuple {
      public:
        constexpr tuple(const Types&...);   // 以引用传参
        template<typename... UTypes> constexpr tuple(UTypes&&...);
        ...
    };

    template<typename... Types>
    tuple(Types...) -> tuple<Types...>; // 以值推导类型
}
```

因此，如下声明：

```
std::tuple t{42, "hello", nullptr};
```

将会推导出 `t` 的类型为 `std::tuple<int, const char*, std::nullptr_t>`。

从迭代器推导

为了能够从表示范围的两个迭代器推导出元素的类型，所有的容器类例如 `std::vector<>` 都有类似于如下的推导指引：

```
// 使std::vector<>能根据初始的迭代器推导出元素类型
namespace std {
    template<typename Iterator>
    vector(Iterator, Iterator) -> vector<typename iterator_traits<Iterator>::value_type>;
}
```

下面的例子展示了它的作用：

```
std::set<float> s;
std::vector v1(s.begin(), s.end()); // OK，推导出std::vector<float>
```

注意这里使用圆括号来初始化是必须的。如果你使用花括号：

```
std::vector v2{s.begin(), s.end()}; // 注意：并不会推导出std::vector<float>
```

那么这两个参数将会被看作一个初值列的两个元素（根据重载规则初值列的优先级更高）。因此，它等价于：

```
std::vector<std::set<float>::iterator> v2{s.begin(), s.end()};
```

这意味着我们初始化了一个两个元素的 vector，第一个元素是一个指向首元素的迭代器，第二个元素是指向尾后元素的迭代器。

另一方面，考虑：

```
std::vector v3{"hi", "world"};  // OK，推导为std::vector<const char*>
std::vector v4("hi", "world");  // OOPS：运行时错误
```

v3 的声明会初始化一个拥有两个元素的 vector（两个元素都是字符串字面量），v4 的初始化会导致运行时错误，很可能会导致 core dump。问题在于字符串字面量被转换成为字符指针，也算是有效的迭代器。因此，我们传递了两个不是指向同一个对象的迭代器。换句话说，我们指定了一个无效的区间。我们推导出了一个 `std::vector<const char>`，但是根据这两个字符串字面量在内存中的位置关系，我们可能会得到一个 `bad_alloc` 异常，也可能会因为没有距离而得到一个 core dump，还有可能得到两个位置之间的未定义范围内的字符。

总而言之，使用花括号是最佳的初始化 vector 的**元素**的方法。唯一的例外是传递单独一个 vector（这时会优先进行拷贝）。当传递别的含义的参数时，使用圆括号会更好。

在任何情况下，对于像 `std::vector<>` 或其他 STL 容器一样拥有复杂的构造函数的类模板，**强烈建议**不要**使用类模板参数推导**，而是显式指明类型。

`std::array<>` 推导

有一个更有趣的例子是关于 `std::array<>` 的。为了能够同时推导出元素的类型和数量：

```
std::array a{42, 45, 77};   // OK，推导出std::array<int, 3>
```

而定义了下面的推导指引（间接的）：

```
// 让std::array<>推导出元素的数量（元素的类型必须相同）：
namespace std {
    template<typename T, typename... U>
    array(T, U...) -> array<enable_if_t<(is_same_v<T, U> && ...), T>, (1 + sizeof...(U))>;
}
```

这个推导指引使用了折叠表达式

```
(is_same_v<T, U> && ...)
```

来保证所有参数的类型相同。[3] 因此，下面的代码是错误的：

```
std::array a{42, 45, 77.7}; // ERROR：元素类型不同
```

注意类模板参数推导的初始化甚至可以在编译期上下文中生效：

```
constexpr std::array arr{0, 8, 15}; // OK，推导出std::array<int, 3>
```

[3] C++ 标准委员会讨论过这个地方是否应该允许隐式类型转换，最后决定采用保守的策略（不允许隐式类型转换）。

(Unordered) Map 推导

想让推导指引正常工作是非常困难的。可以通过给关联容器（map、multimap、unordered_map、unordered_multimap）定义推导指引来展示其复杂程度。

这些容器里元素的类型是 std::pair<const *keytype*, *valuetype*>。这里 const 是必需的，因为元素的位置取决于 key 的值，这意味着如果能修改 key 的值的话会导致容器内部陷入不一致的状态。

在 C++17 标准中为 std::map：

```
namespace std {
    template<typename Key, typename T, typename Compare = less<Key>,
        typename Allocator = allocator<pair<const Key, T>>>
    class map {
        ...
    };
}
```

想出的第一个解决方案是，为如下构造函数：

```
map(initializer_list<pair<const Key, T>>, const Compare& = Compare(),
    const Allocator& = Allocator());
```

定义了如下的推导指引：

```
namespace std {
    template<typename Key, typename T, typename Compare = less<Key>,
        typename Allocator = allocator<pair<const Key, T>>>
    map(initializer_list<pair<const Key, T>>, Compare = Compare(), Allocator = Allocator())
    -> map<Key, T, Compare, Allocator>;
}
```

所有的参数都以值传递，因此这个推导指引允许传递的比较器和分配器 像之前讨论的一样发生退化。然而，我们在推导指引中直接使用了和构造函数中完全相同的元素类型，这意味着初值列的 key 的类型必须是 const 的。因此，下面的代码不能工作（如同 Ville Voutilainen 在https://wg21.link/lwg3025中指出的一样）：

```
std::pair elem1{1, 2};
std::pair elem2{3, 4};
...
std::map m1{elem1, elem2};  // 原来的C++17推导指引会ERROR
```

这是因为 elem1 和 elem2 会被推导为 std::pair<int, int>，而推导指引需要 pair 中的第一个元素是 const 的类型，所以不能成功匹配。因此，你仍然要像下面这么写：

```
std::map<int, int> m1{elem1, elem2};  // OK
```

因此，推导指引中的 const 必须被删掉：

```
namespace std {
    template<typename Key, typename T, typename Compare = less<Key>,
        typename Allocator = allocator<pair<const Key, T>>>
    map(initializer_list<pair<Key, T>>, Compare = Compare(), Allocator = Allocator())
    -> map<Key, T, Compare, Allocator>;
}
```

然而，为了继续支持比较器和分配器的退化，我们还需要为 const key 类型的 pair 定义一个重载版本。否则当传递一个 const key 类型的参数时将会使用构造函数来推导类型，这样会导致传递 const key 和非 const key 参数时推导的结果会有细微的不同。

智能指针没有推导指引

注意 C++ 标准库中某些你觉得应该有推导指引的地方实际上没有推导指引。

你可能会希望共享指针和独占指针有推导指引，这样你就不用写：

```
std::shared_ptr<int> sp{new int(7)};
```

而是直接写：

```
std::shared_ptr sp{new int(y)}; // 不支持
```

上边的写法是错误的，因为相应的构造函数是一个模板，这意味着没有隐式的推导指引：

```
namespace std {
    template<typename T> class shared_ptr {
      public:
        ...
        template<typename Y> explicit shared_ptr(Y* p);
        ...
    };
}
```

这里 Y 和 T 是不同的模板参数，这意味着虽然能从构造函数推导出 Y，但不能推导出 T。这是一个为了支持如下写法的特性：

```
std::shared_ptr<Base> sp{new Derived(...)};
```

假如我们要提供推导指引的话，那么相应的推导指引可以简单的写为：

```
namespace std {
    template<typename Y> shared_ptr(Y*) -> shared_ptr<Y>;
}
```

然而，这可能导致当分配数组时也会应用这个推导指引：

```
std::shared_ptr sp{new int[10]}; // OOPS：推导出shared_ptr<int>
```

就像经常在 C++ 遇到的一样，我们陷入了一个讨厌的 C 问题：就是一个对象的指针和一个对象的数组拥有或者退化以后拥有相同的类型。

这个问题看起来很危险，因此 C++ 标准委员会决定不支持这么写。对于单个对象，你仍然必须这样调用：

```
std::shared_ptr<int> sp1{new int};  // OK
```

对于数组则要：

```
std::shared_ptr<std::string> p(new std::string[10],
                               [](std::string* p) {
                                   delete[] p;
                               });
```

或者，使用实例化原生数组的智能指针的新特性，只需要：

```
std::shared_ptr<std::string[]> p{new std::string[10]};
```

9.3 后记

类模板参数推导由 Michael Spertus 于 2007 年在https://wg21.link/n2332中首次提出。2013 年 Michael Spertus 和 David Vandevoorde 在https://wg21.link/n3602中再次提出。最终被接受的提案由 Michael Spertus、Faisal Vali 和 Richard Smith 发表于 https://wg21.link/p0091r3，之后 Michael Spertus、Faisal Vali 和 Richard Smith 在https://wg21.link/p0512r0中、Jason Merrill 在https://wg21.link/p0620r0中、Michael Spertus 和 Jason Merrill 在https://wg21.link/p702r1（作为 C++17 的缺陷）中提出修改。

标准库中对类模板参数推导特性的支持由 Michael Spertus、Walter E.Brown、Stephan T.Lavavej 在https://wg21.link/p0433r2和 https://wg21.link/p0739r0（作为 C++17 的缺陷）中添加。

Chapter 10

编译期 `if` 语句

通过使用语法 `if constexpr(...)`，编译器可以计算编译期的条件表达式来在编译期决定使用一个 `if` 语句的 *then* 的部分还是 *else* 的部分。其余部分的代码将会被丢弃，这意味着它们甚至不会被生成。然而这并不意味着被丢弃的部分完全被忽略，这些部分中的代码也会像没使用的模板一样进行语法检查。

例如：

tmpl/ifcomptime.hpp

```
#include <string>

template <typename T>
std::string asString(T x)
{
    if constexpr(std::is_same_v<T, std::string>) {
        return x;   // 如果T不能自动转换为string该语句将无效
    }
    else if constexpr(std::is_arithmetic_v<T>) {
        return std::to_string(x); // 如果T不是数字类型该语句将无效
    }
    else {
        return std::string(x);  // 如果不能转换为string该语句将无效
    }
}
```

通过使用 `if constexpr` 我们在编译期就可以决定我们是简单返回传入的字符串、对传入的数字调用 `to_string()` 还是使用构造函数来把传入的参数转换为 `std::string`。无效的调用将被*丢弃*，因此下面的代码能够通过编译（如果使用运行时 `if` 语句则不能通过编译）：

tmpl/ifcomptime.cpp

```
#include "ifcomptime.hpp"
#include <iostream>

int main()
```

```
{
    std::cout << asString(42) << '\n';
    std::cout << asString(std::string("hello")) << '\n';
    std::cout << asString("hello") << '\n';
}
```

10.1 编译期 if 语句的动机

如果我们在上面的例子中使用运行时 if，下面的代码将永远不能通过编译：

tmpl/ifruntime.hpp

```
#include <string>

template <typename T>
std::string asString(T x)
{
    if (std::is_same_v<T, std::string>) {
        return x;   // 如果不能自动转换为string会导致ERROR
    }
    else if (std::is_numeric_v<T>) {
        return std::to_string(x); // 如果不是数字将导致ERROR
    }
    else {
        return std::string(x);  // 如果不能转换为string将导致ERROR
    }
}
```

这是因为模板在实例化时整个模板会作为一个整体进行编译。然而 if 语句的条件表达式的检查是运行时特性。即使在编译期就能确定条件表达式的值一定是 false，*then* 的部分也必须能通过编译。因此，当传递一个 std::string 或者字符串字面量时，会因为 std::to_string() 无效而导致编译失败。此外，当传递一个数字值时，将会因为第一个和第三个返回语句无效而导致编译失败。

使用编译期 if 语句时，*then* 部分和 *else* 部分中不可能被用到的部分将成为 *丢弃的语句*：

- 当传递一个 std::string 时，第一个 if 语句的 *else* 部分将被丢弃。
- 当传递一个数字时，第一个 if 语句的 *then* 部分和最后的 *else* 部分将被丢弃。
- 当传递一个字符串字面量（类型为 const char*）时，第一和第二个 if 语句的 *then* 部分将被丢弃。

因此，在每一个实例化中，无效的分支都会在编译时被丢弃，所以代码能成功编译。

注意被丢弃的语句并不是被忽略了。即使是被忽略的语句也必须符合正确的语法，并且所有和模板参数无关的调用也必须正确。事实上，模板翻译的第一个阶段（定义期间）将会检查语法和所有与模板无关的名称是否有效。所有的 static_asserts 也必须有效，即使所在的分支没有被编译。

例如：

```
template<typename T>
void foo(T t)
```

```
{
    if constexpr(std::is_integral_v<T>) {
        if (t > 0) {
            foo(t-1);    // OK
        }
    }
    else {
        undeclared(t); // 如果未被声明且未被丢弃将导致错误
        undeclared();  // 如果未声明将导致错误（即使被丢弃也一样）
        static_assert(false, "no integral"); // 总是会进行断言（即使被丢弃也一样）
    }
}
```

对于一个符合标准的编译器来说，上面的例子永远不能通过编译的原因有两个：

- 即使 T 是一个整数类型，如下调用：

```
undeclared(); // 如果未声明将导致错误（即使被丢弃也一样）
```

  如果该函数未定义时即使处于被丢弃也会导致错误，因为这个调用并不依赖于模板参数。

- 如下断言：

```
static_assert(false, "no integral"); // 总是会进行断言（即使被丢弃也一样）
```

  即使被丢弃也总是会断言失败，因为它也不依赖于模板参数。一个如下的编译期条件的静态断言将正常生效：

```
static_assert(!std::is_integral_v<T>, "no integral");
```

注意有一些编译器（例如 Visual C++ 2013 和 2015）并没有正确实现模板翻译的两个阶段。它们把第一个阶段（定义期间）的大部分工作推迟到了第二个阶段（实例化期间），因此有些无效的函数调用甚至一些错误的语法都可能通过编译。[1]

10.2 使用编译期 if 语句

原则上讲，只要条件表达式是编译期的表达式你就可以像使用运行期 if 一样使用编译期 if。你也可以混合使用编译期和运行期的 if：

```
if constexpr (std::is_integral_v<std::remove_reference_t<T>>) {
    if (val > 10) {
        if constexpr (std::numeric_limits<char>::is_signed) {
            ...
        }
        else {
            ...
        }
    }
    else {
        ...
```

[1] Visual C++ 正在一步步的修复这个错误的行为，然而可能需要指定编译选项例如 /permissive-，因为这个修复可能会破坏现有的代码。

```
    }
}
else {
    ...
}
```

注意你不能在函数体之外使用 `if constexpr`。因此，你不能使用它来替换预处理器的条件编译。

10.2.1 编译期 if 的注意事项

使用编译期 `if` 时可能会导致一些并不明显的后果。这将在接下来的小节中讨论。[2]

编译期 if 影响返回值类型

编译期 `if` 可能会影响函数的返回值类型。例如，下面的代码总能通过编译，但返回值的类型可能会不同：

```
auto foo()
{
    if constexpr (sizeof(int) > 4) {
        return 42;
    }
    else [
        return 42u;
    }
}
```

这里，因为我们使用了 `auto`，返回值的类型将依赖于返回语句，而执行哪条返回语句又依赖于 `int` 的字节数：

- 如果大于 4 字节，返回 `42` 的返回语句将会生效，因此返回值类型是 `int`。
- 否则，返回 `42u` 的返回语句将生效，因此返回值类型是 `unsigned int`。

通过这个方法，`if constexpr` 可以返回完全不同的类型。例如，如果我们不写 *else* 部分，返回值将会是 `int` 或者 `void`：

```
auto foo()  // 返回值类型可能是int或者void
{
    if constexpr (sizeof(int) > 4) {
        return 42;
    }
}
```

注意这里如果使用运行期 `if` 那么代码将永远不能通过编译，因为推导返回值类型时会考虑到所有可能的返回值类型，因此推导会有歧义。

即使在 *then* 部分返回也要考虑 *else* 部分

运行期 `if` 有一个模式不能应用于编译期 `if`：如果代码在 *then* 和 *else* 部分都会返回，那么在运行期 `if` 中你可以跳过 `else` 部分。也就是说，

[2] 感谢 Graham Haynes、Paul Reilly 和 Barry Revzin 提醒了我编译期 `if` 对这些方面的影响。

```
if (...) {
    return a;
}
else {
    return b;
}
```

可以写成：

```
if (...) {
    return a;
}
return b;
```

但这个模式不能应用于编译期 if，因为在第二种写法里，返回值类型将同时依赖于两个返回语句而不是依赖其中一个，这会导致行为发生改变。例如，如果按照上面的示例修改代码，那么也许能也许不能通过编译：

```
auto foo()
{
    if constexpr (sizeof(int) > 4) {
        return 42;
    }
    return 42u;
}
```

如果条件表达式为 true（int 大于 4 字节），编译器将会推导出两个不同的返回值类型，这会导致错误。否则，将只会有一条有效的返回语句，因此代码能通过编译。

编译期短路求值

考虑如下代码：

```
template<typename T>
constexpr auto foo(const T& val)
{
    if constexpr(std::is_integral<T>::value) {
        if constexpr (T{} < 10) {
            return val * 2;
        }
    }
    return val;
}
```

这里我们使用了两个编译期条件来决定是直接返回传入的值还是返回传入值的两倍。

下面的代码都能编译：

```
constexpr auto x1 = foo(42);    // 返回84
constexpr auto x2 = foo("hi");  // OK，返回"hi"
```

运行时 if 的条件表达式会进行短路求值（当 && 左侧为 false 时停止求值，当 || 左侧为 true 时停止求值）。这可能会导致你希望编译期 if 也会短路求值：

```
template<typename T>
constexpr auto bar(const T& val)
{
    if constexpr (std::is_integral<T>::value && T{} < 10) {
        return val * 2;
    }
    return val;
}
```

然而，编译期 if 的条件表达式总是作为整体实例化并且必须整体有效，这意味着如果传递一个不能进行 <10 运算的类型将不能通过编译：

```
constexpr auto x2 = bar("hi");  // 编译期ERROR
```

因此，编译期 if 在实例化时并不短路求值。如果后边的条件的有效性依赖于前边的条件，那你需要把条件进行嵌套。例如，你必须写成如下形式：

```
if constexpr (std::is_same_v<MyType, T>) {
    if constepxr (T::i == 42) {
        ...
    }
}
```

而不是写成：

```
if constexpr (std::is_same_v<MyType, T> && T::i == 42) {
    ...
}
```

10.2.2 其他编译期 if 的示例

完美返回泛型值

编译期 if 的一个应用就是返回值的完美转发，这些返回值在返回之前必须先进行一些处理。因为 decltype(auto) 不能推导为 void（因为 void 是不完全的类型），所以你必须像下面这么写：

tmpl/perfectreturn.hpp

```
#include <function>     // for std::forward()
#include <type_traits>  // for std::is_same<> and std::invoke_result<>

template<typename Callable, typename... Args>
decltype(auto) call(Callable op, Args&... args)
{
    if constexpr(std::is_void_v<std::invoke_result_t<Callable, Args...>>) {
        // 返回值类型是void：
        op(std::forward<Args>(args)...);
        ... // 在返回前进行一些处理
        return;
    }
```

```
    else {
        // 返回值类型不是void:
        decltype(auto) ret{op(std::forward<Args>(args)...)};
        ... // 在返回前用ret进行一些处理
        return ret;
    }
}
```

函数的返回值类型可以推导为 void，但 ret 的声明不能推导为 void，因此必须把 op 返回 void 的情况单独处理。

使用编译期 if 进行类型分发

编译期 if 的一个典型应用是类型分发。在 C++17 之前，你必须为每一个想处理的类型重载一个单独的函数。现在，有了编译期 if，你可以把所有的逻辑放在一个函数里。

例如，如下的重载版本的 std::advance() 算法：

```
template<typename Iterator, typename Distance>
void advance(Iterator& pos, Distance n) {
    using cat = std::iterator_traits<Iterator>::iterator_category;
    advanceImpl(pos, n, cat{}); // 根据迭代器类型进行分发
}

template<typename Iterator, typename Distance>
void advanceImpl(Iterator& pos, Distance n, std::random_access_iterator_tag) {
    pos += n;
}

template<typename Iterator, typename Distance>
void advanceImpl(Iterator& pos, Distance n, std::bidirectional_iterator_tag) {
    if (n >= 0) {
        while (n--) {
            ++pos;
        }
    }
    else {
        while (n++) {
            --pos;
        }
    }
}

template<typename Iterator, typename Distance>
void advanceImpl(Iterator& pos, Distance n, std::input_iterator_tag) {
    while (n--) {
        ++pos;
    }
}
```

现在可以把所有实现都放在同一个函数中：

```
template<typename Iterator, typename Distance>
void advance(Iterator& pos, Distance n) {
    using cat = std::iterator_traits<Iterator>::iterator_category;
    if constexpr (std::is_convertible_v<cat, std::random_access_iterator_tag>) {
        pos += n;
    }
    else if constexpr (std::is_convertible_v<cat, std::bidirectional_access_iterator_tag>) {
        if (n >= 0) {
            while (n--) {
                ++pos;
            }
        }
        else {
            while (n++) {
                --pos;
            }
        }
    }
    else {  // input_iterator_tag
        while (n--) {
            ++pos;
        }
    }
}
```

这里我们就像是有了一个编译期 switch，每一个 if constexpr 语句就像是一个 case。然而，注意例子中的两种实现还是有一处不同的：[3]

- 重载函数的版本遵循**最佳匹配**语义。
- 编译期 if 的版本遵循**最先匹配**语义。

另一个类型分发的例子是使用编译期 if 实现 get<> 重载 来实现结构化绑定接口。

第三个例子是在用作 std::variant<> 访问器 的泛型 lambda 中处理不同的类型。

10.3 带初始化的编译期 if 语句

注意编译期 if 语句也可以使用新的带初始化的形式。例如，如果有一个 constexpr 函数 foo()，你可以这样写：

```
template<typename T>
void bar(const T x)
{
    if constexpr (auto obj = foo(x); std::is_same_v<decltype(obj), T>) {
        std::cout << "foo(x) yields same type\n";
        ...
```

[3] 感谢 Graham Haynes 和 Barry Revzin 指出这一点。

```
    }
    else {
        std::cout << "foo(x) yields different type\n";
    }
}
```

如果有一个接受参数类型的 constexpr 函数 foo()，你就可以根据 foo(x) 是否返回与 x 相同的类型来进行不同的处理。

如果要根据 foo(x) 返回的值来进行判定，那么可以写：

```
constexpr auto c = ...;
if constexpr (constexpr auto obj = foo(c); obj == 0) {
    std::cout << "foo() == 0\n";
    ...
}
```

注意如果想在条件语句中使用 obj 的值，那么 obj 必须要声明为 constexpr。

10.4 在模板之外使用编译期 if

if constexpr 可以在任何函数中使用，而并非仅限于模板。只要条件表达式是编译期的，并且可以转换成 bool 类型。然而，在普通函数里使用时 *then* 和 *else* 部分的所有语句都必须有效，即使有可能被丢弃。

例如，下面的代码不能通过编译，因为 undeclared() 的调用必须是有效的，即使 char 是有符号数导致 *else* 部分被丢弃也一样：

```
#include <limits>

template<typename T>
void foo(T t);

int main()
{
    if constexpr(std::numeric_limits<char>::is_signed) {
        foo(42);    // OK
    }
    else {
        undeclared(42); // 未声明时总是ERROR（即使被丢弃）
    }
}
```

下面的代码也永远不能通过编译，因为总有一个静态断言会失败：

```
if constexpr(std::numeric_limits<char>::is_signed) {
    static_assert(std::numeric_limits<char>::is_signed);
}
else {
    static_assert(!std::numeric_limits<char>::is_signed);
}
```

在泛型代码之外使用编译期 if 的唯一好处是被丢弃的部分不会成为最终程序的一部分，这将减小生成的可执行程序的大小。例如，在如下程序中：

```
#include <limits>
#include <string>
#include <array>

int main()
{
    if (!std::numeric_limits<char>::is_signed) {
        static std::array<std::string, 1000> arr1;
        ...
    }
    else {
        static std::array<std::string, 1000> arr2;
        ...
    }
}
```

要么 arr1 要么 arr2 会成为最终可执行程序的一部分，但不可能两者都是。[4]

10.5 后记

编译期 if 语句最初的灵感来自于 Walter Bright、Herb Sutter、Andrei Alexandrescu 发表的https://wg21.link/n3329以及 Ville Voutilainen 在https://wg21.link/n4461中提出的 static if 特性。

在https://wg21.link/p0128r0中，Ville Voutilainen 提出了这个特性并命名为 constexpr_if（本章的特性由此得名）。最终被接受的提案由 Jens Maurer 发表于https://wg21.link/p0292r2。

[4]即使不使用 constexpr 也可能实现相同的效果，因为编译器可以优化掉永远不会被使用的代码。然而，如果使用了 constexpr，那么这一行为将得到保证。

Chapter 11

折叠表达式

自从C++17起，有一个新的特性可以计算对参数包中的所有参数应用一个二元运算符的结果。例如，下面的函数将会返回所有参数的总和：

```
template<typename... T>
auto foldSum (T... args) {
    return (... + args);    //((arg1 + arg2) + arg3)...
}
```

注意返回语句中的括号是折叠表达式的一部分，不能被省略。

如下调用：

```
foldSum(47, 11, val, -1);
```

会把模板实例化为：

```
return 47 + 11 + val + -1;
```

如下调用：

```
foldsum(std::string("hello"), "world", "!");
```

会把模板实例化为：

```
return std::string("hello") + "world" + "!";
```

注意折叠表达式里参数的位置很重要（可能看起来还有些反直觉）。如下写法：

```
(... + args)
```

会展开为：

```
((arg1 + arg2) + arg3) ...
```

这意味着折叠表达式会以后递增式重复展开。你也可以写：

```
(args + ...)
```

这样就会前递增式展开，因此结果会变为：

```
(arg1 + (arg2 + arg3)) ...
```

11.1 折叠表达式的动机

折叠表达式的出现让我们不必再用递归实例化模板的方式来处理参数包。在C++17之前，你必须实现为：

```
template<typename T>
auto foldSumRec (T arg) {
    return arg;
}
template<typename T1, typename... Ts>
auto foldSumRec (T1 arg1, Ts... otherArgs) {
    return arg1 + foldSumRec(otherArgs...);
}
```

这样的实现不仅写起来麻烦，对C++编译器来说也很难处理。使用如下写法：

```
template<typename... T>
auto foldSum (T... args) {
    return (... + args);    // arg1 + arg2 + arg3
}
```

能显著的减少程序员和编译器的工作量。

11.2 使用折叠表达式

给定一个参数 *args* 和一个操作符 *op*，C++17允许我们这么写：

- 一元左折叠 (… ***op*** *args*) 将会展开为：*((arg1* ***op*** *arg2)* ***op*** *arg3)* ***op*** *…*
- 一元右折叠 (*args* ***op*** …) 将会展开为：*arg1* ***op*** *(arg2* ***op*** *… (argN-1* ***op*** *argN))*

括号是必须的，然而，括号和省略号(…)之间并不需要用空格分隔。

左折叠和右折叠的不同比想象中更大。例如，当你使用+时可能会产生不同的效果。使用左折叠时：

```
template<typename... T>
auto foldSumL(T... args) {
    return (... + args);    // ((arg1 + arg2) + arg3)...
}
```

如下调用

```
foldSumL(1, 2, 3);
```

会求值为

```
((1 + 2) + 3)
```

这意味着下面的例子能够通过编译：

```
std::cout << foldSumL(std::string("hello"), "world", "!")
          << '\n';  // OK
```

记住对字符串而言只有两侧至少有一个是`std::string`时才能使用+。使用作折叠式，会首先计算

```
std::string("hello") + "world"
```

这将返回一个 `std::string`，因此再加上字符串字面量`"!"`是有效的。

然而，如下调用

```
std::cout << foldSumL("hello", "world", std::string("!"))
          << '\n';  // ERROR
```

将不能通过编译，因为它会求值为

```
("hello" + "world") + std::string("!");
```

然而把两个字符串字面量相加是错误的。

然而如果我们把实现修改为：

```
template<typename... T>
auto foldSumR(T... args) {
    return (args + ...);    // (arg1 + (arg2 + arg3))...
```

那么如下调用

```
foldSumR(1, 2, 3)
```

将求值为

```
(1 + (2 + 3))
```

这意味着下面的例子不能再通过编译：

```
std::cout << foldSumR(std::string("hello"), "world", "!")
          << '\n';  // ERROR
```

然而如下调用现在反而可以编译了：

```
std::cout << foldSumR("hello", "world", std::string("!"))
          << '\n';  // OK
```

在任何情况下，从左向右求值都是符合直觉的。因此，更推荐使用左折叠的语法：

```
(... + args);       // 推荐的折叠表达式语法
```

11.2.1 处理空参数包

当使用折叠表达式处理空参数包时，将遵循如下规则：

- 如果使用了 `&&` 运算符，值为 `true`。
- 如果使用了 `||` 运算符，值为 `false`。
- 如果使用了逗号运算符，值为 `void()`。
- 使用所有其他的运算符，都会引发格式错误

对于所有其他的情况，你可以添加一个初始值：给定一个参数包 *args*，一个初始值 *value*，一个操作符 *op*，C++17 允许我们这么写：

- 二元左折叠 (*value* ***op*** ... ***op*** *args*) 将会展开为：*(((value* ***op*** *arg1)* ***op*** *arg2)* ***op*** *arg3)* ***op*** *...*
- 二元右折叠 (*args* ***op*** ... ***op*** *value*) 将会展开为：*arg1* ***op*** *(arg2* ***op*** *... (argN* ***op*** *value))*

省略号两侧的 *op* 必须相同。

例如，下面的定义在进行加法时允许传递一个空参数包：

```
template<typename... T>
auto foldSum (T... s) {
    return (0 + ... + s);   // 即使sizeof...(s)==0也能工作
}
```

从概念上讲，不管 `0` 是第一个还是最后一个操作数应该和结果无关：

```
template<typename... T>
auto foldSum (T... s) {
    return (s + ... + 0);   // 即使sizeof...(s)==0也能工作
}
```

然而，对于一元折叠表达式来说，不同的求值顺序比想象中的更重要。对于二元表达式来说，也更推荐左折叠的方式：

```
(val + ... + args);         // 推荐的二元折叠表达式语法
```

有时候第一个操作数是特殊的，比如下面的例子：

```
template<typename... T>
void print(const T&... args)
{
    (std::cout << ... << args) << '\n';
}
```

这里，传递给 `print()` 的第一个参数输出之后将返回输出流，所以后面的参数可以继续输出。其他的实现可能不能编译或者产生一些意料之外的结果。例如，

```
std::cout << (args << ... << '\n');
```

类似 `print(1)` 的调用可以编译，但会打印出 1 左移 `'\n'` 位之后的值，`'\n'` 的值通常是 10，所以结果是 1024。

注意在这个 `print()` 的例子中，两个参数之间没有输出空格字符。因此，如下调用 `print("hello", 42, "world")` 将会打印出：

```
hello42world
```

为了用空格分隔传入的参数，你需要一个辅助函数来确保除了第一个参数之外的剩余参数输出前都先输出一个空格。例如，使用如下的模板 `spaceBefore()` 可以做到这一点：

tmpl/addspace.hpp

```
template<typename T>
const T& spaceBefore(const T& arg) {
    std::cout << ' ';
    return arg;
}

template <typename First, typename... Args>
void print (const First& firstarg, const Args&... args) {
```

```
    std::cout << firstarg;
    (std::cout << ... << spaceBefore(args)) << '\n';
}
```

这里，折叠表达式

```
(std::cout << ... << spaceBefore(args))
```

将会展开为：

```
std::cout << spaceBefore(arg1) << spaceBefore(arg2) << ...
```

因此，对于参数包中的每一个参数 `args`，都会调用辅助函数，在输出参数之前先输出一个空格到 `std::cout`。为了确保不会对第一个参数调用辅助函数，我们添加了额外的模板参数对第一个参数进行单独处理。

注意要想让参数包正确输出需要确保对每个参数调用 `spaceBefore()` 之前左侧的所有输出都已经完成。得益于操作符 `<<` 的有定义的表达式求值顺序，自从 C++17 起将保证行为正确：

我们也可以使用 lambda 来在 `print()` 内定义 `spaceBefore()`：

```
template<typename First, typename... Args>
void print (const First& firstarg, const Args&... args) {
    std::cout << firstarg;
    auto spaceBefore = [](const auto& arg) {
        std::cout << ' ';
        return arg;
    };
    (std::cout << ... << spaceBefore(args)) < '\n';
}
```

然而，注意默认情况下 lambda 以值返回对象，这意味着会创建参数的不必要的拷贝。解决方法是显式指明返回类型为 `const auto&` 或者 `decltype(auto)`：

```
template<typename First, typename... Args>
void print (const First& firstarg, const Args&... args) {
    std::cout << firstarg;
    auto spaceBefore = [](const auto& arg) -> const auto& {
        std::cout << ' ';
        return arg;
    };
    (std::cout << ... << spaceBefore(args)) << '\n';
}
```

如果你不能把他们写在一个表达式里那么 C++ 就不是 C++ 了：

```
template<typename First, typename... Args>
void print (const First& firstarg, const Args&... args) {
    std::cout << firstarg;
    (std::cout << ... << [](const auto& arg) -> decltype(auto) {
                             std::cout << ' ';
                             return arg;
                         }(args)) << '\n';
}
```

不过，一个更简单的实现 `print()` 的方法是使用一个 lambda 输出空格和参数，然后在一元折叠表达式里使用它：[1]

```
template<typename First, typename... Args>
void print(First first, const Args&... args) {
    std::cout << first;
    auto outWithSpace = [](const auto& arg) {
                            std::cout << ' ' << arg;
                        };
    (... , outWithSpace(args));
    std::cout << '\n';
}
```

通过使用新的auto 模板参数，我们可以使 `print()` 变得更加灵活：可以把间隔符定义为一个参数，这个参数可以是一个字符、一个字符串或者其它任何可打印的类型。

11.2.2 支持的运算符

你可以对除了 `.`、`->`、`[]` 之外的所有二元运算符使用折叠表达式。

折叠函数调用

折叠表达式可以用于逗号运算符，这样就可以在一条语句里进行多次函数调用。也就是说，你现在可以简单写出如下实现：

```
template<typename... Types>
void callFoo(const Types&... args)
{
    ...
    (... , foo(args));  // 调用foo(arg1), foo(arg2), foo(arg3), ...
}
```

来对所有参数调用函数 `foo()`。

另外，如果需要支持移动语义：

```
template<typename... Types>
void callFoo(Types&&... args)
{
    ...
    (... , foo(std::forward<Types>(args))); // 调用foo(arg1), foo(arg2), ...
}
```

如果 `foo()` 函数返回的类型重载了逗号运算符，那么代码行为可能会改变。为了保证这种情况下代码依然安全，你需要把返回值转换为 `void`：

```
template<typename... Types>
void callFoo(const Types&... args)
{
```

[1] 感谢 Barry Revzin 指出这一点。

```
    ...
    (... , (void)foo(std::forward<Types>(args))); // 调用foo(arg1), foo(arg2), ...
}
```

注意自然情况下，对于逗号运算符不管我们是左折叠还是右折叠都是一样的。函数调用们总是会从左向右执行。如下写法：

```
(foo(args) , ...);
```

中的括号只是把后边的调用括在一起，因此首先是第一个 foo() 调用，之后是被括起来的两个 foo() 调用：

```
foo(arg1) , (foo(arg2) , foo(arg3));
```

然而，因为逗号表达式的求值顺序通常是自左向右，所以第一个调用通常发生在括号里的的后两个调用之前，并且括号里左侧的调用在右侧的调用之前。[2]

不过，因为左折叠更符合自然的求值顺序，因此在使用折叠表达式进行多次函数调用时还是推荐使用左折叠。

组合哈希函数

另一个使用逗号折叠表达式的例子是组合哈希函数。可以用如下的方法完成：

```
template<typename T>
void hashCombine (std::size_t& seed, const T& val)
{
    seed ^= std::hash<T>()(val) + 0x9e3779b9 + (seed<<6) + (seed>>2);
}

template<typename... Types>
std::size_t combinedHashValue (const Types&... args)
{
    std::size_t seed = 0;               // 初始化seed
    (... , hashCombine(seed, args));    // 链式调用hashCombine()
    return seed;
}
```

如下调用

```
combinedHashValue ("Hi", "World", 42);
```

函数中的折叠表达式将被展开为：

```
hashCombine(seed, "Hi"), (hashCombine(seed, "World"), hashCombine(seed, 42));
```

有了这些定义，我们现在可以轻易的定义出一个新的哈希函数，并将这个函数用于某一个类型例如 Customer 的无序 set 或无序 map：

```
sturct CustomerHash
{
    std::size_t operator() (const Customer& c) const {
```

[2]如果重载逗号运算符，你可以改变它的求值顺序，这可能影响左折叠和右折叠的求值顺序。

```
        return combinedHashValue(c.getFirstname(), c.getLastname(), c.getValue());
    }
};

std::unordered_set<Customer, CustomerHash> coll;
std::unordered_map<Customer, std::string, CustomerHash> map;
```

折叠基类的函数调用

折叠表达式可以在更复杂的场景中使用。例如，你可以折叠逗号表达式来调用可变数量基类的成员函数：

tmpl/foldcalls.cpp

```
#include <iostream>

// 可变数量基类的模板
template<typename... Bases>
class MultiBase : private Bases...
{
public:
    void print() {
        // 调用所有基类的print()函数
        (... , Bases::print());
    }
};

struct A {
    void print() { std::cout << "A::print()\n"; }
}

struct B {
    void print() { std::cout << "B::print()\n"; }
}

struct C {
    void print() { std::cout << "C::print()\n"; }
}

int main()
{
    MultiBase<A, B, C> mb;
    mb.print();
}
```

这里

```
template<typename... Bases>
class MultiBase : private Bases...
{
```

```
    ...
};
```

允许我们用可变数量的基类初始化对象：

```
MultiBase<A, B, C> mb;
```

进一步通过

```
(... , Bases::print());
```

这个折叠表达式将展开为调用每一个基类中的`print`。也就是说，这条语句会被展开为如下代码：

```
(A::print(), B::print()), C::print();
```

折叠路径遍历

你也可以使用折叠表达式通过运算符`->*`遍历一个二叉树中的路径。考虑下面的递归数据结构：

tmpl/foldtraverse.hpp

```
// 定义二叉树结构和遍历帮助函数
struct Node {
    int value;
    Node *subLeft{nullptr};
    Node *subRight{nullptr};

    Node(int i = 0) : value{i} {}

    int getValue() const {
        return value;
    }

    ...
    // 遍历帮助函数
    static constexpr auto left = &Node::subLeft;
    static constexpr auto right = &Node::subRight;

    // 使用折叠表达式遍历树
    template<typename T, typename... TP>
    static Node *traverse(T np, TP... paths) {
        return (np ->* ... ->* paths);  // np ->* paths1 ->* paths2
    }
};
```

这里，

```
(np ->* ... ->* paths)
```

使用了折叠表达式以`np`为起点遍历可变长度的路径，可以像下面这样使用这个函数：

tmpl/foldtraverse.cpp

```
#include "foldtraverse.hpp"
#include <iostream>

int main()
{
    // 初始化二叉树结构:
    Node* root = new Node{0};
    root->subLeft = new Node{1};
    root->subLeft->subRight = new Node{2};
    ...
    // 遍历二叉树
    Node* node = Node::traverse(root, Node::left, Node::right);
    std::cout << node->getValue() << '\n';
    node = root ->* Node::left ->* Node::right;
    std::cout << node->getValue() << '\n';
    node = root -> subLeft -> subRight;
    std::cout << node->getValue() << '\n';
}
```

当调用

```
Node::traverse(root, Node::left, Node::right);
```

时折叠表达式将展开为：

```
root ->* Node::left ->* Node::right
```

结果等价于

```
root -> subLeft -> subRight
```

11.2.3 使用折叠表达式处理类型

通过使用类型特征，我们也可以使用折叠表达式来处理模板参数包（任意数量的模板类型参数）。例如，你可以使用折叠表达式来判断一些类型是否相同：

tmpl/ishomogeneous.hpp

```
#include <type_traits>

// 检查是否所有类型都相同
template<typename T1, typename... TN>
struct IsHomogeneous {
    static constexpr bool value = (std::is_same_v<T1, TN> && ...);
};

// 检查是否所有传入的参数类型相同
template<typename T1, typename... TN>
constexpr bool isHomogeneous(T1, TN...)
```

```
{
    return (std::is_same_v<T1, TN> && ...);
}
```

类型特征 `IsHomogeneous<>` 可以像下面这样使用：

```
IsHomogeneous<int, MyType, decltype(42)>::value
```

这种情况下，折叠表达式将会展开为：

```
std::is_same_v<int, MyType> && std::is_same_v<int, decltype(42)>
```

函数模板 `isHomogeneous<>()` 可以像下面这样使用：

```
isHomogeneous(43, -1, "hello", nullptr)
```

在这种情况下，折叠表达式将会展开为：

```
std::is_same_v<int, int> && std::is_same_v<int, const char*> && is_same_v<int, std::nullptr_t>
```

像通常一样，运算符 `&&` 会短路求值（出现第一个 `false` 时就会停止运算）。

标准库里`std::array<>` 的推导指引就使用了这个特性。

11.3 后记

折叠表达式特性最早由 Andrew Sutton 和 Richard Smith 在https://wg21.link/n4191中提出。最终被接受的提案由 Andrew Sutton 和 Richard Smith 发表于https://wg21.link/n4295。之后对运算符 *、+、&、| 处理空参数包的支持由 Thibaut Le Jehan 在 https://wg21.link/p0036中移除。

Chapter 12

处理字符串字面量模板参数

一直以来，不同版本的C++标准一直在放宽模板参数的标准，C++17也是如此。另外现在非类型模板参数的实参不需要再定义在使用处的外层作用域。

12.1 在模板中使用字符串

非类型模板参数只能是常量整数值（包括枚举）、对象/函数/成员的指针、对象或函数的左值引用、`std::nullptr_t`（`nullptr`的类型）。

对于指针，在C++17之前需要外部或者内部链接。然而，自从C++17起，可以使用无链接的指针。然而，你仍然不能直接使用字符串字面量。例如：

```
template<const char* str>
class Message {
    ...
};

extern const char hello[] = "Hello World!"; // 外部链接
const char hello11[] = "Hello World!";      // 内部链接

void foo()
{
    Message<hello> msg;     // OK（所有C++标准）
    Message<hello11> msg11; // OK（自从C++11起）

    static const char hello17[] = "Hello World!";   // 无链接
    Message<hello17> msg17; // OK（自从C++17起）

    Message<"hi"> msgError; // ERROR
}
```

也就是说自从C++17起你仍然需要至少两行才能把字符串字面量传给一个模板参数。然而，你现在可以把第一行写在和实例化代码相同的作用域内。

这个特性还解决了一个不行的约束：自从C++11起可以把一个指针作为模板实参：

```
template<int* p> struct A {
};

int num;
A<&num> a;  // OK（自从C++11起）
```

但不能用一个返回指针的编译期函数作为模板实参，然而现在可以了：

```
int num;
...
constexpr int* pNum() {
    return &num;
}
A<pNum()> b;    // C++17之前ERROR，现在OK
```

12.2 后记

允许编译期常量求值结果作为非类型模板实参由Richard Smith在https://wg21.link/n4198中首次提出。最终被接受的提案由Richard Smith发表于https://wg21.link/n4268。

Chapter 13

占位符类型作为模板参数（例如 auto）

自从 C++17 起，你可以使用占位符类型（auto 和 decltype(auto)）作为非类型模板参数的类型。这意味着我们可以写出泛型代码来处理不同类型的非类型模板参数。

13.1 使用 auto 模板参数

自从 C++17 起，你可以使用 auto 来声明非类型模板参数。例如：

```
template<auto N> class S {
    ...
};
```

这允许我们为不同类型实例化非类型模板参数 N：

```
S<42> s1;   // OK：S中N的类型是int
S<'a'> s2;  // OK：S中N的类型是char
```

然而，你不能使用这个特性来实例化一些不允许作为模板参数的类型：

```
S<2.5> s3;  // ERROR：模板参数的类型不能是double
```

我们甚至还可以用指明类型的版本作为部分特化版的模板：

```
template<int N> class S<N> {
    ...
};
```

甚至还支持类模板参数推导。例如：

```
template<typename T, auto N>
class A {
  public:
    A(const std::array<T, N>&) {
    }
    A(T(&)[N]) {
    }
```

```
    ...
};
```

这个类可以推导出T的类型、N的类型、N的值：

```
A a2{"hello"};  // OK，推导为A<const char, 6>，N的类型是std::size_t

std::array<double, 10> sa1;
A a1{sa1};      // OK，推导为A<double, 10>，N的类型是std::size_t
```

你也可以修饰auto，例如，可以确保参数类型必须是个指针：

```
template<const auto* P> struct S;
```

另外，通过使用可变参数模板，你可以使用多个不同类型的模板参数来实例化模板：

```
template<auto... VS> class HeteroValueList {
};
```

也可以用多个相同类型的参数：

```
template<auto V1, decltype(V1)... VS> class HomoValueList {
};
```

例如：

```
HeteroValueList<1, 2, 3> vals1;         // OK
HeteroValueList<1, 'a', true> vals2;    // OK
HomoValueList<1, 2, 3> vals3;           // OK
HomoValueList<1, 'a', true> vals4;      // ERROR
```

13.1.1 字符和字符串模板参数

这个特性的一个应用就是你可以定义一个既可能是字符也可能是字符串的模板参数。例如，我们可以像下面这样改进用折叠表达式输出任意数量参数的方式：

tmpl/printauto.hpp

```
#include <iostream>

template<auto Sep = ' ', typename First, typename... Args>
void print(const First& first, const Args&... args) {
    std::cout << first;
    auto outWithSep = [](const auto& arg) {
                        std::cout << Sep << arg;
                    };
    (... , outWithSep(args));
    std::cout << '\n';
}
```

将默认的参数分隔符Sep设置为空格，我们可以实现和之前的效果：

```
template<auto Sep = ' ', typename First, typename... Args>
void print (const First& firstarg, const Args&... args) {
    ...
}
```

我们仍然可以像之前一样调用：

```
std::string s{"world"};
print(7.5, "hello", s};      // 打印出：7.5 hello world
```

然而，通过把分隔符 Sep 参数化，我们也可以显示指明另一个字符作为分隔符：

```
print<'-'>(7.5, "hello", s); // 打印出：7.5-hello-world
```

甚至，因为使用了 auto，我们甚至可以传递被声明为无链接的字符串字面量作为分隔符：

```
static const char sep[] = ", ";
print<sep>(7.5, "hello", s); // 打印出：7.5, hello, world
```

另外，我们也可以传递任何其他可以用作模板参数的类型：

```
print<-11>(7.5, "hello", s); // 打印出：7.5-11hello-11world
```

13.1.2 定义元编程常量

auto 模板参数特性的另一个应用是可以让我们更轻易的定义编译期常量。[1] 原本的下列代码：

```
template<typename T, T v>
struct constant
{
    static constexpr T value = v;
};

using i = constant<int, 42>;
using c = constant<char, 'x'>;
using b = constant<bool, true>;
```

现在可以简单的实现为：

```
template<auto v>
struct constant
{
    static constexpr auto value = v;
};

using i = constant<42>;
using c = constant<'x'>;
using b = constant<true>;
```

同样，原本的下列代码：

[1] 感谢 Bryce Adelstein Lelbach 提供这些例子。

```
template<typename T, T... Elements>
struct sequence {
};

using indexes = sequence<int, 0, 3, 4>;
```

现在可以简单的实现为：

```
template<auto... Elements>
struct sequence {
};

using indexes = sequence<0, 3, 4>;
```

你现在甚至可以定义一个持有若干不同类型的值的编译期对象（类似于一个简单的 tuple）：

```
using tuple = sequence<0, 'h', true>;
```

13.2 使用 auto 作为变量模板的参数

你也可以使用 auto 作为模板参数来实现变量模板 *(variable templates)*。[2] 例如，下面的声明定义了一个变量模板 arr，元素的类型和数量作为参数：

```
template<typename T, auto N> std::array<T, N> arr;
```

在每个编译单元中，所有对 arr<int, 10> 的引用将指向同一个全局对象。而 arr<long, 10> 和 arr<int, 10u> 将指向其他对象（每一个都可以在所有编译单元中使用）。

作为一个完整的例子，考虑如下的头文件：

tmpl/vartmplauto.hpp

```
#ifndef VARTMPLAUTO_HPP
#define VARTMPLAUTO_HPP

#include <array>
template<typename T, auto N> std::array<T, N> arr{};

void printArr();

#endif // VARTMPLAUTO_HPP
```

这里，我们可以在一个编译单元内修改两个变量模板的不同实例：

tmpl/vartmplauto1.cpp

```
#include "vartmplauto.hpp"

int main()
```

[2] 不要混淆了变量模板 *(variable templates)* 和可变参数模板 *(variadic templates)*，前者是模板化的变量，后者是有任意数量参数的模板。

```
{
    arr<int, 5>[0] = 17;
    arr<int, 5>[3] = 42;
    arr<int, 5u>[1] = 11;
    arr<int, 5u>[3] = 33;
    printArr();
}
```

另一个编译单元内可以打印这两个变量模板：

tmpl/vartmplauto2.cpp

```
#include "vartmplauto.hpp"
#include <iostream>

void printArr()
{
    std::cout << "arr<int, 5>:  ";
    for (const auto& elem : arr<int, 5>) {
        std::cout << elem << ' ';
    }
    std::cout << "\narr<int, 5u>: ";
    for (const auto& elem : arr<int, 5>) {
        std::cout << elem << ' ';
    }
    std::cout << '\n';
}
```

程序的输出将是：[3]

```
arr<int, 5>:  17 0 0 42 0
arr<int, 5u>: 0 11 0 33 0
```

用同样的方式你可以声明一个任意类型的常量变量模板，类型可以通过初始值推导出来：

```
template<auto N> constexpr auto val = N; // 自从C++17起OK
```

之后可以像下面这样使用：

```
auto v1 = val<5>;       // v1 == 5，v1的类型为int
auto v2 = val<true>;    // v2 == true，v2的类型为bool
auto v3 = val<'a'>;     // v3 == 'a'，v3的类型为char
```

这里阐述了发生了什么：

```
std::is_same_v<decltype(val<5>), int>       // 返回false
std::is_same_v<decltype(val<5>), const int> // 返回true
std::is_same_v<decltype(v1), int>   // 返回true（因为auto会退化）
```

[3] g++7 有一个 bug 显示它的两个变量模板实质上是同一对象，这个 bug 在 g++8 里修复了。

13.3 使用 decltype(auto) 模板参数

你现在也可以使用另一个占位类型 `decltype(auto)`（C++14 引入）作为模板参数。注意，这个占位类型的推导有非常特殊的规则。根据 `decltype` 的规则，如果使用 `decltype(auto)` 来推导*表达式 (expressions)* 而不是变量名，那么推导的结果将依赖于表达式的值类型：

- prvalue（例如临时变量）推导出 *type*
- lvalue（例如有名字的对象）推导出 *type&*
- xvalue（例如 `std::move()` 的对象）推导出 *type&&*

这意味着你很容易就会把模板参数推导为引用，这可能导致一些令人惊奇的效果。

例如：

tmpl/decltypeauto.cpp

```
#include <iostream>

template<decltype(auto) N>
struct S {
    void printN() const {
        std::cout << "N: " << N << '\n';
    }
};

static const int c = 42;
static int v = 42;

int main()
{
    S<c> s1;        // N的类型推导为const int 42
    S<(c)> s2;      // N的类型推导为const int&，N是c的引用
    s1.printN();
    s2.printN();

    S<(v)> s3;      // N的类型推导为int&，N是v的引用
    v = 77;
    s3.printN();    // 打印出：N: 77
}
```

13.4 后记

非类型模板参数的占位符类型最早由 James Touton 和 Michael Spertus 作为 https://wg21.link/n4469的一部分提出。最终被接受的提案由 James Touton 和 Michael Spertus 发表于 https://wg21.link/p0127r2。

Chapter 14

扩展的 using 声明

using 声明扩展之后可以支持逗号分隔的名称，也可以支持参数包。
例如，你现在可以这么写：

```
class Base {
  public:
    void a();
    void b();
    void c();
};

class Derived : private Base {
  public:
    using Base::a, Base::b, Base::c;
};
```

在 C++17 之前，你需要使用 3 个 using 声明分别进行声明。

14.1 使用变长的 using 声明

逗号分隔的 using 声明允许你用泛型代码从可变数量的所有基类中派生同一种运算。
这项技术的一个很酷的应用是创建一个重载的 lambda 的集合。通过如下定义：

tmpl/overload.hpp

```
// 继承所有基类里的函数调用运算符
template<typename... Ts>
struct overload : Ts...
{
    using Ts::operator()...;
};

// 基类的类型从传入的参数中推导
template<typename... Ts>
```

```
overload(Ts...) -> overload<Ts...>;
```

你可以像下面这样重载两个 lambda：

```
auto twice = overload {
                [](std::string& s) { s += s; },
                [](auto& v) { v *= 2; }
             };
```

这里，我们创建了一个 `overload` 类型的对象，并且提供了推导指引 来根据 lambda 的类型推导出 `overload` 的基类的类型。并且我们使用了聚合体初始化 来调用每个 lambda 生成的闭包类型的拷贝构造函数来初始化基类子对象。

上例中的 using 声明使得 `overload` 类型可以同时访问所有子类中的函数调用运算符。如果没有这个 using 声明，两个基类会产生同一个成员函数 `operator()` 的重载，这将会导致歧义。[1]

最后，如果你传递一个字符串参数将会调用第一个重载，其他类型（操作符 `*=` 有效的类型）将会调用第二个重载：

```
int i = 42;
twice(i);
std::cout << "i: " << i << '\n';    // 打印出：84
std::string s = "hi";
twice(s);
std::cout << "s: " << s << '\n';    // 打印出：hihi
```

这项技术的另一个应用是`std::variant` 访问器。

14.2 使用变长 using 声明继承构造函数

除了逐个声明继承构造函数之外，现在还支持如下的方式：你可以声明一个可变参数类模板 `Multi`，让它继承每一个参数类型的基类：

tmpl/using2.hpp

```
template<typename T>
class Base {
    T value{};
  public:
    Base() {
        ...
    }
    Base(T v) : value{v} {
        ...
    }
    ...
};
```

[1] clang 和 Visual C++ 都不会把不同基类中不同类型的同名函数当作歧义处理，所以这个例子中其实不需要 `using`。然而，这段代码如果没有 using 声明的将不具备可移植性。

```
template<typename... Types>
class Multi : private Base<Types>...
{
  public:
    // 继承所有构造函数：
    using Base<Types>::Base...;
    ...
};
```

有了所有基类构造函数的 using 声明，你可以继承每个类型对应的构造函数。

现在，当使用不同类型声明 `Multi<>` 时：

```
using MultiISB = Multi<int, std::string, bool>;
```

你可以使用每一个相应的构造函数来声明对象：

```
MultiISB m1 = 42;
MultiISB m2 = std::string("hello");
MultiISB m3 = true;
```

根据新的语言规则，每一个初始化会调用匹配基类的相应构造函数和所有其他基类的默认构造函数。因此：

```
MultiISB m2 = std::string("hello");
```

会调用 `Base<int>` 的默认构造函数，`Base<std::string>` 的字符串构造函数，`Base<bool>` 的默认构造函数。

原则上讲，你也可以通过如下声明来支持 `Multi<>` 进行赋值操作：

```
template<typename... Types>
class Multi : private Base<Types>...
{
    ...
    // 派生所有赋值运算符
    using Base<Types>::operator=...;
};
```

14.3 后记

逗号分隔的 using 声明列表由 Robert Haberlach 在https://wg21.link/p0195r0中首次提出。最终被接受的提案由 Robert Haberlach 和 Richard Smith 发表于https://wg21.link/p0195r2。

关于继承构造函数有一些核心的问题。最终修复这些问题的提案由 Richard Smith 发表于 https://wg21.link/n4429。

还有一个由 Vicente J.Botet Escriba 提出的提案。除了 lambda 之外，它还支持重载普通函数、成员函数来实现泛型的 `overload` 函数。然而，这个提议并没有进入 C++17 标准。详情请见https://wg21.link/p0051r1。

Part III

新的标准库组件

这一部分介绍 C++17 中新的标准库组件。

Chapter 15

std::optional<>

在编程时，我们经常会遇到我们可能会返回/传递/使用一个确定类型的对象。也就是说，这个对象可能有一个确定类型的值也可能没有任何值。因此，我们需要一种方法来模拟类似指针的语义：指针可以通过 nullptr 来表示没有值。解决方法是定义该对象的同时再定义一个附加的 bool 类型的值作为标志来表示该对象是否有值。std::optional<> 提供了一种类型安全的方式来实现这种对象。

可选对象所需的内存等于内含对象的大小加上一个布尔类型的大小。因此，可选对象一般比内含对象大一个字节（可能还要加上内存对齐的空间开销）。可选对象不需要分配对内存，并且对齐方式和内含对象相同。

然而，可选对象并不是简单的等价于附加了 bool 标志的内含对象。例如，在没有值的情况下，将不会调用内含对象的构造函数（通过这种方式，没有默认构造函数的内含类型也可以处于有效的默认状态）。

和 std::variant<> 和 std::any 一样，可选对象有值语义。也就是说，拷贝操作会被实现为深拷贝：将创建一个新的独立对象，新对象在自己的内存空间内拥有原对象的标记和内含值（如果有的话）的拷贝。拷贝一个无内含值的 std::optional<> 的开销很小，但拷贝有内含值的 std::optional<> 的开销约等于拷贝内含值的开销。另外，std::optional<> 对象也支持 move 语义。

15.1 使用 std::optional<>

std::optional<> 模拟了一个可以为空的任意类型的实例。他可以被用作成员、参数、返回值等。

15.1.1 可选的返回值

下面的示例程序展示了将 std::optional<> 用作返回值的一些功能：

lib/optional.cpp

```
#include <optional>
#include <string>
#include <iostream>

// 如果可能的话把string转换为int：
std::optional<int> asInt(const std::string& s)
```

```
{
    try {
        return std::stoi(s);
    }
    catch (...) {
        return std::nullopt;
    }
}

int main()
{
    for (auto s : {"42", "  077", "hello", "0x33"}) {
        // 尝试把s转换为int，并打印结果
        std::optional<int> oi = asInt(s);
        if (oi) {
            std::cout << "convert '" << s << "' to int: " << *oi << "\n";
        }
        else {
            std::cout << "can't convert '" << s << "' to int\n";
        }
    }
}
```

这段程序包含了一个 `asInt()` 函数来把传入的字符串转换为整数。然而这个操作有可能会失败，因此把返回值定义为 `std::optional<>`，这样我们可以返回“无整数值”而不是约定一个特殊的 `int` 值，或者向调用者抛出异常来表示失败。

因此，当转换成功时我们用 `stoi()` 返回的 `int` 初始化返回值并返回，否则会返回 `std::nullopt` 来表明没有 `int` 值。

我们也可以像下面这样实现相同的行为：

```
std::optional<int> asInt(const std::string& s)
{
    std::optional<int> ret; // 初始化为无值
    try {
        ret = std::stoi(s);
    }
    catch (...) {
    }
    return ret;
}
```

在 `main()` 中，我们用不同的字符串调用了这个函数：

```
for (auto s : {"42", "  077", "hello", "0x33"} ) {
    // 尝试把s转换为int，并打印结果
    std::optional<int> oi = asInt(s);
    ...
}
```

对于每一个返回的 std::optional<int> 类型的 oi，我们可以判断它是否含有值（将该对象用作布尔表达式）并通过“解引用”的方式访问了该可选对象的值：

```
if (oi) {
    std::cout << "convert '" << s << "' to int: " << *oi << "\n";
}
```

注意用字符串 "0x33" 调用 asInt() 将会返回 0，因为 stoi() 不会以十六进制的方式来解析字符串。

还有一些别的方式来处理返回值，例如：

```
std::optional<int> oi = asInt(s);
if (oi.has_value()) {
    std::cout << "convert '" << s << "' to int: " << oi.value() << "\n";
}
```

这里使用了 has_value() 来检查是否返回了一个值，使用了 value() 来访问值。value() 比运算符 * 更安全：当没有值时它会抛出一个异常。运算符 * 应该只用于已经确定含有值的场景，否则程序将可能有未定义的行为。[1]

注意，我们现在可以使用新的类型 std::string_view 和新的快捷函数 std::from_chars() 来改进 asInt()。

15.1.2 可选的参数和数据成员

另一个使用 std::optional<> 的例子是传递可选的参数和设置可选的数据成员：

lib/optionalmember.cpp

```
#include <string>
#include <optional>
#include <iostream>

class Name
{
  private:
    std::string first;
    std::optional<std::string> middle;
    std::string last;
  public:
    Name (std::string f, std::optional<std::string> m, std::string l)
          : first{std::move(f)}, middle{std::move(m)}, last{std::move(l)} {
    }
    friend std::ostream& operator << (std::ostream& strm, const Name& n) {
        strm << n.first << ' ';
        if (n.middle) {
            strm << *n.middle << ' ';
        }
        return strm << n.last;
```

[1] 注意你可能不会注意到这个未定义的行为，因为运算符 * 将会返回某个内存位置的值，这个值可能是有意义的。

```
    }
};

int main()
{
    Name n{"Jim", std::nullopt, "Knopf"};
    std::cout << n << '\n';

    Name m{"Donald", "Ervin", "Knuth"};
    std::cout << m << '\n';
}
```

类 Name 代表了一个由名、可选的中间名、姓组成的姓名。成员 middle 被定义为可选的，当没有中间名是你可以传递一个 std::nullopt。这和中间名是空字符串是不同的。

注意和通常值语义的类型一样，最佳的定义构造函数的方式是以值传参，然后把参数的值移动到成员里。

注意 std::optional<> 改变了成员 middle 的值的使用方式。直接使用 n.middle 将是一个布尔表达式，标记是否有中间名。使用 *n.middle 可以访问当前的值（如果有值的话）。

另一个访问值的方法是使用成员函数 value_or()，当没有值的时候可以指定一个备选值。例如，在类 Name 里我们可以实现为：

```
std::cout << middle.value_or("");   // 打印中间名或空
```

然而，这种方式下，当没有值时名和姓之间将有两个空格而不是一个。

15.2 std::optional<> 类型和操作

这一小节详细描述 std::optional 类型和支持的操作。

15.2.1 std::optional<> 类型

标准库在头文件 <optional> 中以如下方式定义了 std::optional<> 类：

```
namespace std {
    template<typename T> class optional;
}
```

另外还定义了下面这些类型和对象：

- std::nullopt_t 类型的 std::nullopt，作为可选对象无值时候的“值”。
- 从 std::exception 派生的 std::bad_optional_access 异常类，当无值时候访问值将会抛出该异常。

可选对象也使用了 <utility> 头文件中定义的 std::in_place 对象（类型是 std::in_place_t）来初始化多个参数的可选对象（见下文）。

15.2.2 std::optional<> 的操作

表std::optional 的操作列出了 std::optional<> 的所有操作：

操作符	效果
构造函数	创建一个可选对象（可能会调用内含类型的构造函数也可能不会）
make_optional<>()	创建一个用参数初始化的可选对象
析构函数	销毁一个可选对象
=	赋予一个新值
emplace()	给内含类型赋予一个新值
reset()	销毁值（使对象变为无值状态）
has_value()	返回可选对象是否含有值
转换为bool	返回可选对象是否含有值
*	访问内部的值（如果无值将会产生未定义行为）
->	访问内部值的成员（如果无值将会产生未定义行为）
value()	访问内部值（如果无值将会抛出异常）
value_or()	访问内部值（如果无值将返回参数的值）
swap()	交换两个对象的值
==、!=、<、<=、>、>=	比较可选对象
hash<>	计算哈希值的函数对象的类型

Table 15.1: std::optional<> 的操作

构造函数

特殊的构造函数允许你直接传递内含类型的值作为参数。

- 你可以创建一个不含有值的可选对象。这种情况下，你必须指明内含的类型：

```
std::optional<int> o1;
std::optional<int> o2(std::nullopt);
```

这种情况下将不会调用内含类型的任何构造函数。

- 你可以传递一个值来初始化内含类型。得益于推导指引，你不需要再指明内含类型：

```
std::optional o3{42};       // 推导出optional<int>
std::optional o4{"hello"};  // 推导出optional<const char*>
using namespace std::string_literals;
std::optional o5{"hello"s}; // 推导出optional<string>
```

- 为了用多个参数初始化可选对象，你必须传递一个构造好的对象或者添加 std::in_place 作为第一个参数（内含类型不能再被推导出来）：

```
std::optional o6{std::complex{3.0, 4.0}};
std::optional<std::complex<double>> o7{std::in_place, 3.0, 4.0};
```

注意第二种方式避免了创建临时变量。通过使用这种方式，你甚至可以传递一个初值列加上其他参数：

```
// 用lambda作为排序准则初始化set
auto sc = [] (int x, int y) {
            return std::abs(x) < std::abs(y);
```

```
            };
    std::optional<std::set<int, decltype(sc)>> o8{std::in_place, {4, 8, -7, -2, 0, 5}, sc};
```

然而，只有当素有的初始值都和容器里元素的类型匹配时才可以这么写。否则，你必须显式传递一个std::initializer_list<>：

```
    // 用lambda作为排序准则初始化set
    auto sc = [] (int x, int y) {
                return std::abs(x) < std::abs(y);
              };
    std::optional<std::set<int, decltype(sc)>>
        o8{std::in_place, std::initializer_list<int>{4, 5L}, sc};
```

- 如果底层类型支持拷贝的话可选对象也可以拷贝（支持类型转换）：

```
    std::optional o9{"hello"};          // 推导出optional<const char*>
    std::optional<std::string> o10{o9}; // OK
```

然而，注意如果内含类型本身可以用一个可选对象来构造，那么将会优先用可选对象构造内含对象，而不是拷贝：[2]

```
    std::optional<int> o11;
    std::optional<std::any> o12{o11}; // o12内含了一个any对象，该对象的值是一个空的optional<int>
```

注意还有一个快捷函数make_optional<>()，他可以用单个或多个参数初始化一个可选对象（不用使用in_place）。像通常的make...函数一样，它的参数也会退化：

```
auto o13 = std::make_optional(3.0);     // optional<double>
auto o14 = std::make_optional("hello"); // optional<const char*>
auto o15 = std::make_optional<std::complex<double>>(3.0, 4.0);
```

然而，注意没有一个构造函数可以根据参数的值来判断是应该用某个值还是用nullopt来初始化可选对象。这种情形下，只能使用运算符?:。[3] 例如：

```
std::multimap<std::string, std::string> englishToGerman;
...
auto pos = englishToGerman.find("wisdom");
auto o16 = pos != englishToGerman.end() ? std::optional{pos->second} : std::nullopt;
```

这个例子中，根据类模板参数推导，通过表达式std::optional{pos->second}能推导出o16的类型是std::optional<std::string>。类模板参数推导不能对单独的std::nullopt生效，但通过使用运算符?:，std::nullopt也会转换成optional<string>类型，这是因为?:运算符的两个表达式必须有相同的类型。

访问值

为了检查一个可选对象是否有值，你可以调用has_value()或者在bool表达式中使用它：

[2] 感谢 Tim Song 指出这一点。
[3] 感谢 Roland Bock 指出这一点。

```
std::optional o{42};

if (o) ...              // true
if (!o) ...             // false
if (o.has_value())...   // true
```

没有为可选对象定义 I/O 运算符，因为当可选对象无值时不确定应该输出什么：

```
std::cout << o;         // ERROR
```

要访问内部值可以使用指针语法。也就是说，通过运算符 *，你可以直接访问可选对象的底层值，也可以使用 -> 访问内部值的成员：

```
std::optional o{std::pair{42, "hello"}};
auto p = *o;            // 初始化p为pair<int, string>
std::cout << o->first;  // 打印出42
```

注意这些操作符都需要可选对象包含有值。在没有值的情况下这样使用会导致未定义行为：

```
std::optional<std::string> o{"hello"};

std::cout << *o;    // OK：打印出"hello"
o = std::nullopt;
std::cout << *o;    // 未定义行为
```

注意在实践中第二个输出语句仍能正常编译并可能再次打印出"hello"，因为可选对象里底层值的内存并没有被修改。然而，你绝不应该依赖这一点。如果你不知道是否一个可选对象含有值，你必须像下面这样调用：

```
if (o) std::cout << *o;     // OK（可能输出为空字符串）
```

或者你可以使用 value() 成员函数来访问值，当内部没有值时将抛出一个 std::bad_optional_access 异常：

```
std::cout << o.value();     // OK（无值时会抛出异常）
```

std::bad_optional_access 直接派生自 std::exception。

请注意 operator* 和 value() 都是返回内含对象的引用。因此，当直接使用这些操作返回的临时对象时要小心。例如，对于一个返回可选字符串的函数：

```
std::optional<std::string> getString();
```

把它返回的可选对象的值赋给新对象总是安全的：

```
auto a = getString().value();   // OK：内含对象的拷贝或抛出异常
```

然而，直接使用返回值（或者作为参数传递）是麻烦的根源：

```
auto b = *getString();  // ERROR：如果返回std::nullopt将会有未定义行为
const auto& r1 = getString().value(); // ERROR：引用销毁的内含对象
auto&& r2 = getString().value();      // ERROR：引用销毁的内含对象
```

使用引用的问题是：根据规则，引用会延长 value() 的返回值的生命周期，而不是 getString() 返回的可选对象的生命周期。因此，r1 和 r2 会引用不存在的值，使用它们将会导致未定义行为。

注意当使用范围 for 循环时很容易出现这个问题：

```
std::optional<std::vector<int>> getVector();
...
for (int i : getVector().value()) { // ERROR：迭代一个销毁的vector
    std::cout << i << '\n';
}
```

注意迭代一个 non-optional 的 `vector<int>` 类型的返回值是可以的。因此，不要盲目的把函数返回值替换为相应的可选对象类型。（译者注：有点看不懂原文这里想表达什么意思，暂且就这么翻译。）

最后，你可以在获取值时针对无值的情况设置一个 fallback 值。这通常是把一个可选对象写入到输出流的最简单的方式：

```
std::cout << o.value_or("NO VALUE");  // OK（没有值时写入NO VALUE）
```

然而，`value()` 和 `value_or()` 之间有一个需要考虑的差异：[4] `value_or()` 返回值，而 `value()` 返回引用。这意味着如下调用：

```
std::cout << middle.value_or("");
```

和：

```
std::cout << o.value_or("fallback");
```

都会暗中分配内存，而 `value()` 永远不会。

然而，当在临时对象 (rvalue) 上调用 `value_or()` 时，将会移动走内含对象的值并以值返回，而不是调用拷贝函数构造。这是唯一一种能让 `value_or()` 适用于 move-only 的类型的方法，因为在左值 (lvalue) 上调用的 `value_or()` 的重载版本需要内含对象可以拷贝。

因此，上面例子中效率最高的实现方式是：

```
std::cout << o ? o->c_str() : "fallback";
```

而不是：

```
std::cout << o.value_or("fallback");
```

注意 `value_or()` 是一个能够更清晰地表达意图的接口，但开销可能会更大一点。

比较

你可以使用通常的比较运算符。操作数可以是可选对象、内含类型的对象、`std::nullopt`。

- 如果两个操作数都是有值的对象，将会调用内含类型的相应操作符。
- 如果两个操作数都是没有值的对象，那么它们相等（`==`、`<=`、`>=` 返回 `true`，其他比较返回 `false`）。
- 如果恰有一个操作数有值，那么无值的操作数小于有值的操作数。

例如：

```
std::optional<int> o0;
std::optional<int> o1{42};

o0 == std::nullopt  // 返回true
o0 == 42            // 返回false
```

[4] 感谢 Alexander Brockmöller 指出这一点。

```
o0 < 42             // 返回true
o0 > 42             // 返回false
o1 == 42            // 返回true
o0 < o1             // 返回true
```

这意味着 unsigned int 的可选对象，甚至可能小于 0：

```
std::optional<unsigned> uo;

uo < 0              // 返回true
uo < -42            // 返回true
```

对于 bool 类型的可选对象，也可能小于 false：

```
std::optional<bool> bo;
bo < false          // 返回true
```

为了让代码可读性更高，应该使用

```
if (!uo.has_value())
```

而不是

```
if (uo <0)
```

可选对象和底层类型之间的混合比较也是支持的，前提是底层类型支持这种比较：

```
std::optional<int> o1{42};
std::optional<double> o2{42.0};

o2 == 42            // 返回true
o1 == o2            // 返回true
```

如果底层类型支持隐式类型转换，那么相应的可选对象类型也会进行隐式类型转换。

注意可选的 bool 或原生指针类型可能会导致一些奇怪的行为。

修改值

赋值运算和 emplace() 操作可以用来修改值：

```
std::optional<std::complex<double>> o;  // 没有值
std::optional ox{77};   // optional<int>, 值为77

o = 42;                 // 值变为complex(42.0, 0.0)
o = {9.9, 4.4};         // 值变为complex(9.9, 4.4)
o = ox;                 // OK, 因为int转换为complex<double>
o = std::nullopt;       // o不再有值
o.emplace(5.5, 7.7);    // 值变为complex(5.5, 7.7)
```

赋值为 std::nullopt 会移除内含值，如果之前有值的话将会调用内含类型的析构函数。你也可以通过调用 reset() 实现相同的效果：

```
o.reset();      // o不再有值
```

或者赋值为空的花括号：

```
o = {};          // o不再有值
```

最后，我们也可以使用operator*来修改值，因为它返回的是引用。然而，注意这种方式要求值必须存在：

```
std::optional<std::complex<double>> o;
*o = 42;         // 未定义行为
...
if (o) {
    *o = 88;          // OK：值变为complex(88.0, 0.0)
    *o = {1.2, 3.4}; // OK：值变为complex(1.2, 3.4)
}
```

move语义

std::optional<>也支持move语义。如果你move了整个可选对象，那么内部的状态会被拷贝，值会被move。因此，被move的可选对象仍保持原来的状态，但值变为未定义。

然而，你也可以单独把内含的值移进或移出。例如：

```
std::optional<std::string> os;
std::string s = "a very very very long string";
os = std::move(s);                  // OK，move
std::string s2 = *os;               // OK，拷贝
std::string s3 = std::move(*os);    // OK，move
```

注意在最后一次调用之后，os仍然含有一个字符串值，但就像值被移走的对象一样，这个值是未定义的。因此，你可以使用它，但不要对它的值有任何假设。你也可以给它赋一个新的字符串。

另外注意有些重载版本会保证临时的可选对象被move。[5] 考虑下面这个返回一个可选字符串的函数：

```
std::optional<std::string> func();
```

在这种情况下，下面的代码将会move临时可选对象的值：

```
std::string s4 = func().value();    // OK，move
std::string s5 = *func();           // OK，move
```

可以通过重载相应成员函数的右值版本来保证上述的行为：

```
namespace std {
    template<typename T>
    class optional {
        ...
        constexpr T& operator*() &;
        constexpr const T& operator*() const&;
        constexpr T&& operaotr*() &&;
        constexpr const T&& operator*() const&&;
        constexpr T& value() &;
        constexpr const T& value() const&;
        constexpr T&& value() &&;
```

[5] 感谢Alexander Brockmöller指出这一点。

```
            constexpr const T&& value() const&&;
        };
    }
```

换句话说，你也可以像下面这样写：

```
std::optional<std::string> os;
std::string s6 = std::move(os).value(); // OK，move
```

哈希

可选对象的哈希值就等于内含值的哈希值（如果有值的话）。

无值的可选对象的哈希值未定义。

15.3 特殊情况

一些特定的可选类型可能会导致特殊或意料之外的行为。

15.3.1 bool 类型或原生指针的可选对象

将可选对象用作 bool 值时使用比较运算符会有特殊的语义。如果内含类型是 bool 或者指针类型的话这可能导致令人迷惑的行为。例如：

```
std::optional<bool> ob{false};  // 值为false
if (!ob) ...                    // 返回false
if (ob == false) ...            // 返回true

std::optional<int*> op{nullptr};
if (!op) ...                    // 返回false
if (op == nullptr) ...          // 返回true
```

15.3.2 可选对象的可选对象

原则上讲，你可以定义可选对象的可选对象：

```
std::optional<std::optional<std::string>> oos1;
std::optional<std::optional<std::string>> oos2 = "hello";
std::optional<std::optional<std::string>> oos3{std::in_place, std::in_place, "hello"};

std::optional<std::optional<std::complex<double>>> ooc{std::in_place, std::in_place, 4.2, 5.3};
```

你甚至可以通过隐式类型转换直接赋值：

```
oos1 = "hello";         // OK：赋新值
ooc.emplace(std::in_place, 7.2, 8.3);
```

因为两层可选对象都可能没有值，可选对象的可选对象允许你在内层无值或者在外层无值，这可能会导致不同的语义：

```
*oos1 = std::nullopt;    // 内层可选对象无值
oos1 = std::nullopt;     // 外层可选对象无值
```

这意味着在处理这种可选对象的时候你必须特别小心：

```
if (!oos1) std::cout << "no value\n";
if (oos1 && !*oos1) std::cout << "no inner value\n";
if (oos1 && *oos1) std::cout << "value: " << **oos1 << '\n';
```

然而，从语义上来看，这只是一个有两种状态都代表无值的类型而已。因此，带有两个 bool 值或monostate的std::variant<>将是一个更好的替代。

15.4 后记

可选对象由 Fernando Cacciola 于 2005 年在https://wg21.link/n1878中首次提出，并指定 Boost.Optional 作为参考实现。这个类因为 Fernando Cacciola 和 Andrzej Krzemienski 在https://wg21.link/n3793中的提案被 Library Fundamentals TS 采纳。

这个类因 Beman Dawes 和 Alisdair Meredith 发表于https://wg21.link/p0220r1 的提案而和其他组件一起被 C++17 标准采纳。

Tony van Eerd 在发表于https://wg21.link/n3765和 https://wg21.link/p0307r2的提案中极大的改进了可选对象的比较运算的语义。Vicente J. Botet Escriba 在发表于https://wg21.link/p0032r3的提案中统一了 std::optional 和 std::variant<> 以及 std::any 类的 API。Jonathan Wakely 在https://wg21.link/p0504r0修正了 in_place 标记类型的行为。

Chapter 16

std::variant<>

通过 std::variant<>，C++ 标准库提供了一个新的联合类型，它最大的优势是提供了一种新的具有多态性的处理异质集合的方法。也就是说，它可以帮助我们处理不同类型的数据，并且不需要公共基类和指针。

16.1 std::variant<> 的动机

起源于 C 语言，C++ 也提供对 union 的支持，它的作用是持有一个值，这个值的类型可能是指定的若干类型中的任意一个。然而，这项语言特性有一些缺陷：

- 对象并不知道它们现在持有的值的类型。
- 因此，你不能持有非平凡类型，例如 std::string（没有进行特殊处理的话）。[1]
- 你不能从 union 派生。

通过 std::variant<>，C++ 标准库提供了一种可辨识的联合（这意味着要指明一个可能的类型列表）

- 当前值的类型已知
- 可以持有任何类型的值
- 可以从它派生

事实上，一个 std::variant<> 持有的值有若干*候选项 (alternative)*，这些选项通常有不同的类型。然而，两个不同选项的类型也有可能相同，这在多个类型相同的选项分别代表不同含义的时候很有用（例如，可能有两个选项类型都是字符串，分别代表数据库中不同列的名称，你可以知道当前的值代表哪一个列）。

variant 所占的内存大小等于所有可能的底层类型中最大的再加上一个记录当前选项的固定内存开销。不会分配堆内存。[2]

一般情况下，除非你指定了一个侯选项来表示为空，否则 variant 不可能为空。然而，在非常罕见的情况下（例如赋予一个不同类型新值时发生了异常），variant 可能会变为没有值的状态。

和 std::optional<>、std::any 一样，variant 对象是值语义。也就是说，拷贝被实现为深拷贝，将会创建一个在自己独立的内存空间内存储有当前选项的值的新对象。然而，拷贝 std::variant<> 的开销要比

[1] 自从 C++11 起，原则上 union 可以拥有非平凡类型的成员，但你必须实现几个特定的成员函数例如拷贝构造函数和析构函数，因为你只能通过程序的逻辑来判断当前哪个成员是有效的。

[2] 这一点和 Boost.Variant 不同，后者必须在堆里分配内存来确保当值改变时如果发生异常可以恢复。

拷贝当前选项的开销稍微大一点，这是因为variant必须找出要拷贝哪个值。另外，variant也支持move语义。

16.2 使用std::variant<>

下面的代码展示了std::variant<>的核心功能：

lib/variant.cpp

```
#include <variant>
#include <iostream>

int main()
{
    std::variant<int, std::string> var{"hi"};   // 初始化为string选项
    std::cout << var.index() << '\n';           // 打印出1
    var = 42;                                   // 现在持有int选项
    std::cout << var.index() << '\n';           // 打印出0
    ...
    try {
        int i = std::get<0>(var);               // 通过索引访问
        std::string s = std::get<std::string>(var); // 通过类型访问（这里会抛出异常）
        ...
    }
    catch (const std::bad_variant_access& e) {  // 当索引/类型错误时进行处理
        std::cerr << "EXCEPTION: " << e.what() << '\n';
        ...
    }
}
```

成员函数index()可以用来指出当前选项的索引（第一个选项的索引是0）。

初始化和赋值操作都会查找最匹配的选项。如果类型不能精确匹配，可能会发生奇怪的事情。

注意空variant、有引用成员的variant、有C风格数组成员的variant、有不完全类型（例如void）的variant都是不允许的。[3]

variant没有空的状态。这意味着每一个构造好的variant对象，至少调用了一次构造函数。默认构造函数会调用第一个选项类型的默认构造函数：

```
std::variant<std::string, int> var; // => var.index()==0, 值==""
```

如果第一个类型没有默认构造函数，那么调用variant的默认构造函数将会导致编译期错误：

```
struct NoDefConstr {
    NoDefConstr(int i) {
        std::cout << "NoDefConstr::NoDefConstr(int) called\n";
    }
};
```

[3]这些特性可能会在之后添加，但直到C++17还没有足够的经验来支持它们。

```
std::variant<NoDefConstr, int> v1; // ERROR：不能默认构造第一个选项
```

辅助类型 std::monostate 提供了处理这种情况的能力，还可以用来模拟空值的状态。

std::monostate

为了支持第一个类型没有默认构造函数的 variant，C++ 标准库提供了一个特殊的辅助类：std::monostate。std::monostate 类型的对象总是处于相同的状态。因此，比较它们的结果总是相等。它的作用是可以作为一个选项，当 variant 处于这个选项时表示此 variant 没有其他任何类型的值。

因此，类 std::monostate 可以作为第一个选项类型来保证 variant 能默认构造。例如：

```
std::variant<std::monostate, NoDefConstr, int> v2;  // OK
std::cout << "index: " << v2.index() << '\n';       // 打印出0
```

某种意义上，你可以把这种状态解释为 variant 为空的信号。[4]

下面的代码展示了几种检测 monostate 的方法，也同时展示了 variant 的其他一些操作：

```
if (v2.index() == 0) {
    std::cout << "has monostate\n";
}
if (!v2.index()) {
    std::cout << "has monostate\n";
}
if (std::holds_alternative<std::monostate>(v2)) {
    std::cout << "has monostate\n";
}
if (std::get_if<0>(&v2)) {
    std::cout << "has monostate\n";
}
if (std::get_if<std::monostate>(&v2)) {
    std::cout << "has monostate\n";
}
```

get_if<>() 的参数是一个指针，并在当前选项为 T 时返回一个指向当前选项的指针，否则返回 nullptr。这和 get<T>() 不同，后者获取 variant 的引用作为参数并在提供的索引或类型正确时以值返回当前选项，否则抛出异常。

通常情况下，你可以赋予 variant 一个和当前选项类型不同的其他选项的值，甚至可以赋值为 monostate 来表示为空：

```
v2 = 42;
std::cout << "index: " << v2.index() << '\n';   // index：1

v2 = std::monostate{};
std::cout << "index: " << v2.index() << '\n';   // index: 0
```

[4] 理论上讲，std::monostate 可以作为任意选项，而不是必须作为第一个选项。然而，如果不是第一个选项的话就不能帮助 variant 进行默认构造。

从 variant 派生

你可以从 variant 派生。例如，你可以定义如下派生自 std::variant<> 的 聚合体：

```
class Derived : public std::variant<int, std::string> {
};

Derived d = {{"hello"}};
std::cout << d.index() << '\n';          // 打印出：1
std::cout << std::get<1>(d) << '\n';     // 打印出：hello
d.emplace<0>(77);                        // 初始化为int，销毁string
std::cout << std::get<0>(d) << '\n';     // 打印出：77
```

16.3 std::variant<> 类型和操作

这一节详细描述了 std::variant<> 类型和操作。

16.3.1 std::variant<> 类型

在头文件 variant，C++ 标准库以如下方式定义了类 std::variant<>：

```
namespace std {
    template<typename... Types> class variant;
    // 译者注：此处原文的定义是
    // template<typename Types...> class variant;
    // 应是作者笔误
}
```

也就是说，std::variant<> 是一个可变参数 *(variadic)* 类模板（C++11 引入的处理任意数量参数的特性）。

另外，还定义了下面的类型和对象：

- 类模板 std::variant_size
- 类模板 std::variant_alternative
- 值 std::variant_npos
- 类型 std::monostate
- 异常类 std::bad_variant，派生自 std::exception

还有两个为 variant 定义的变量模板：std::in_place_type<> 和 std:: in_place_index<>。它们的类型分别是 std::in_place_type_t 和 std::in_place_index_t，在头文件 <utility> 中定义。

16.3.2 std::variant<> 的操作

表std::variant 的操作列出了 std::variant<> 的所有操作。

构造函数

默认情况下，variant 的默认构造函数会调用第一个选项的默认构造函数：

操作符	效果
构造函数	创建一个 variant 对象（可能会调用底层类型的构造函数）
析构函数	销毁一个 variant 对象
=	赋新值
emplace<T>()	销毁旧值并赋一个 T 类型选项的新值
emplace<Idx>()	销毁旧值并赋一个索引为 Idx 的选项的新值
valueless_by_exception	返回变量是否因为异常而没有值
index()	返回当前选项的索引
swap()	交换两个对象的值
==、!=、<、<=、>、>=	比较 variant 对象
hash<>	计算哈希值的函数对象
holds_alternative<T>()	返回是否持有类型 T 的值
get<T>()	返回类型为 T 的选项的值
get<Idx>()	返回索引为 Idx 的选项的值
get_if<T>()	返回指向类型为 T 的选项的指针或 nullptr
get_if<Idx>()	返回指向索引为 Idx 的选项的指针或 nullptr
visit()	对当前选项进行操作

Table 16.1: std::variant<> 的操作

```
std::variant<int, int, std::string> v1; // 第一个int初始化为0，index()==0
```

选项被默认初始化，意味着基本类型会初始化为 0、false、nullptr。

如果传递一个值来初始化，将会使用最佳匹配的类型：

```
std::variant<long, int> v2{42};
std::cout << v2.index() << '\n';    // 打印出1
```

然而，如果有两个类型同等匹配会导致歧义：

```
std::variant<long, long> v3{42};          // ERROR：歧义
std::variant<int, float> v4{42.3};        // ERROR：歧义
std::variant<int, double> v5{42.3};       // OK
std::variant<int, long double> v6{42.3};  // ERROR：歧义

std::variant<std::string, std::string_view> v7{"hello"}; // ERROR：歧义
std::variant<std::string, std::string_view,
    const char*> v8{"hello"};       // OK
std::cout << v8.index() << '\n';    // 打印出2
```

为了传递多个值来调用构造函数初始化，你必须使用 in_place_type 或者 in_place_index 标记：

```
std::variant<std::complex<double>> v9{3.0, 4.0};    // ERROR
std::variant<std::complex<double>> v10{{3.0, 4.0}}; // ERROR
std::variant<std::complex<double>> v11{std::in_place_type<std::complex<double>>, 3.0, 4.0};
std::variant<std::complex<double>> v12{std::in_place_index<0>, 3.0, 4.0};
```

你也可以使用 in_place_index 在初始化时解决歧义问题或者打破匹配优先级：

```
std::variant<int, int> v13{std::in_place_index<1>, 77};    // 初始化第二个int
std::variant<int, long> v14{std::in_place_index<1>, 77};    // 初始化long，而不是int
std::cout << v14.index() << '\n';       // 打印出1
```

你甚至可以传递一个带有其他参数的初值列：

```
// 用一个lambda作为排序准则初始化一个set的variant
auto sc = [] (int x, int y) {
            return std::abs(x) < std::abs(y);
          };
std::variant<std::vector<int>, std::set<int, decltype(sc)>>
    v15{std::in_place_index<1>, {4, 8, -7, -2, 0, 5}, sc};
```

然而，只有当所有初始值都和容器里元素类型匹配时才可以这么做。否则你必须显式传递一个 std::initializer_list<>：

```
// 用一个lambda作为排序准则初始化一个set的variant
auto sc = [] (int x, int y) {
            return std::abs(x) < std::abs(y);
          };
std::variant<std::vector<int>, std::set<int, decltype(sc)>>
v15{std::in_place_index<1>, std::initializer_list<int>{4, 5L}, sc};
```

std::variant<> 不支持类模板参数推导，也没有 make_variant<>() 快捷函数（不像 std::optional<> 和 std::any）。这样做也没有意义，因为 variant 的目标是处理多个侯选项。

如果所有的侯选项都支持拷贝，那么就可以拷贝 variant 对象：

```
struct NoCopy {
    NoCopy() = default;
    NoCopy(const NoCopy&) = delete;
};

std::variant<int, NoCopy> v1;
std::variant<int, NoCopy> v2{v1};   // ERROR
```

访问值

通常调用 get<>() 或 get_if<> 来获取当前选项的值。你可以传递一个索引、或者传递一个类型（该类型的选项只能有一个）。使用一个无效的索引和无效/歧义的类型将会导致编译错误。如果访问的索引或者类型不是当前的选项，将会抛出一个 std::bad_variant_exception。

例如：

```
std::variant<int, int, std::string> var; // 第一个int设为0，index()==0

auto a = std::get<double>(var); // 编译期ERROR：没有double类型的选项
auto b = std::get<4>(var);      // 编译期ERROR：没有第五个选项
auto c = std::get<int>(var);    // 编译期ERROR：有两个int类型的选项
```

```
try {
    auto s = std::get<std::string>(var); // 抛出异常（当前选项是第一个int）
    auto i = std::get<0>(var);  // OK，i==0
    auto j = std::get<1>(var);  // 抛出异常（当前选项是另一个int）
}
catch (const std::bad_variant_access& e) {
    std::cout << "Exception: " << e.what() << '\n';
}
```

还有另一个 API 可以在访问值之前先检查给定的选项是否是当前选项。你需要给 `get_if<>()` 传递一个指针，如果访问成功则返回指向当前选项的指针，否则返回 `nullptr`。

```
if (auto ip = std::get_if<1>(&var); ip != nullptr) {
    std::cout << *ip << '\n';
}
else {
    std::cout << "alternative with index 1 not set\n";
}
```

这里还使用了带初始化的 `if` 语句，把初始化过程和条件检查分成了两条语句。你也可以直接把初始化语句用作条件语句：

```
if (auto ip = std::get_if<1>(&var)) {
    std::cout << *ip << '\n';
}
else {
    std::cout << "alternative with index 1 not set\n";
}
```

另一种访问不同选项的值的方法是使用variant 访问器。

修改值

赋值操作和 `emplace()` 函数可以修改值：

```
std::variant<int, int, std::string> var; // 设置第一个int为0，index()==0
var == "hello";          // 设置string选项，index()==2
var.emplace<1>(42);      // 设置第二个int，index()==1
```

注意 `operator=` 将会直接赋予 variant 一个新值，只要有和新值类型对应的选项。`emplace()` 在赋予新值之前会先销毁旧的值。

你也可以使用 `get<>()` 或者 `get_if<>()` 来给当前选项赋予新值：

```
std::variant<int, int, std::string> var; // 设置第一个int为0，index()==0
std::get<0>(var) = 77;  // OK，但当前选项仍是第一个int
std::get<1>(var) = 99;  // 抛出异常（因为当前选项是另一个int）
```

另一个修改不同选项的值的方法是variant 访问器。

比较

对两个类型相同的 variant（也就是说，它们每个选项的类型和顺序都相同），你可以使用通常的比较运算符。比较运算将遵循如下规则：

- 当前选项索引较小的小于当前选项索引较大的。
- 如果两个 variant 当前的选项相同，将调用当前选项类型的比较运算符进行比较。
  注意所有的 `std::monostate` 类型的对象都相等。
- 两个 variant 都处于特殊状态（`valueless_by_exception()`为真的状态）时相等。否则，`valueless_by_exception()` 返回 ture 的 variant 小于另一个。

例如：

```
std::variant<std::monostate, int, std::string> v1, v2{"hello"}, v3{42};
std::variant<std::monostate, std::string, int> v4;
    v1 == v4    // 编译期错误
    v1 == v2    // 返回false
    v1 < v2     // 返回true
    v1 < v3     // 返回true
    v2 < v3     // 返回false

v1 = "hello";
    v1 == v2    // 返回true

v2 = 41;
    v2 < v3     // 返回true
```

move 语义

只要所有的选项都支持 move 语义，那么 `std::variant<>` 也支持 move 语义。

如果你 move 了 variant 对象，那么当前状态会被拷贝，而当前选项的值会被 move。因此，被 move 的 variant 对象仍然保持之前的选项，但值会变为未定义。

你也可以把值一进或移出 variant 对象。

哈希

如果所有的选项类型都能计算哈希值，那么 variant 对象也能计算哈希值。注意 variant 对象的哈希值不保证是当前选项的哈希值。在某些平台上它是，有些平台上不是。

16.3.3 访问器

另一个处理 variant 对象的值的方法是使用访问器 (visitor)。访问器是为明确的每一个可能的类型都提供一个函数调用运算符的对象。当这些对象“visit”一个 variant 时，它们会调用和当前选项类型最匹配的函数。

使用函数对象作为访问器

例如：

lib/variantvisit.cpp

```
#include <variant>
#include <string>
#include <iostream>

struct MyVisitor
{
    void operator() (int i) const {
        std::cout << "int:    " << i << '\n';
    }
    void operator() (std::string s) const {
        std::cout << "string: " << s << '\n';
    }
    void operator() (long double d) const {
        std::cotu << "double: " << d << '\n';
    }
};

int main()
{
    std::variant<int, std::string, double> var(42);
    std::visit(MyVisitor(), var);   // 调用int的operator()
    var = "hello";
    std::visit(MyVisitor(), var);   // 调用string的operator()
    var = 42.7;
    std::visit(MyVisitor(), var);   // 调用long double的operator()
}
```

如果访问器没有某一个可能的类型的 operator() 重载，那么调用 visit() 将会导致编译期错误，如果调用有歧义的话也会导致编译期错误。这里的示例能正常工作是因为 long double 比 int 更匹配 double。

你也可以使用访问器来修改当前选项的值（但不能赋予一个其他选项的新值）。例如：

```
struct Twice
{
    void operator()(double& d) const {
        d *= 2;
    }
    void operator()(int& i) const {
        i *= 2;
    }
    void operator()(std::string& s) const {
        s = s + s;
    }
};
std::visit(Twice(), var);   // 调用匹配类型的operator()
```

访问器调用时只根据类型判断，你不能对类型相同的不同选项做不同的处理。

注意上面例子中的函数调用运算符都应该标记为 const，因为它们是 无状态的 *(stateless)*（它们的行为只受

参数的影响）。

使用泛型 lambda 作为访问器

最简单的使用访问器的方式是使用泛型 lambda，它是一个可以处理任意类型的函数对象：

```
auot printvariant = [](const auto& val) {
                        std::cout << val << '\n';
                    };
...
std::visit(printvariant, var);
```

这里，泛型 lambda 生成的闭包类型中会将函数调用运算符定义为模板：

```
class ComplierSpecificClosureTypeName {
  public:
    template<typename T>
    auto operator() (const T& val) const {
        std::cout << val << '\n';
    }
};
```

因此，只要调用时生成的函数内的语句有效（这个例子中就是输出运算符要有效），那么把 lambda 传递给 `std::visit()` 就可以正常编译。

你可以可以使用 lambda 来修改当前选项的值：

```
// 将当前选项的值变为两倍
std::visit([](auto& val) {
                val = val + val;
            }, var);
```

或者：

```
// 将当前选项的值设为默认值
std::visit([](auto& val) {
                val = std::remove_reference_t<decltype(val)>{};
            }, var);
```

你甚至可以使用编译期 `if` 语句来对不同的选项类型进行不同的处理。例如：

```
auto dblvar = [](auto& val) {
    if constexpr(std::is_convertible_v<decltype(val), std::string>) {
        val = val + val;
    }
    else {
        val *= 2;
    }
};
...
std::visit(dblvar, var);
```

这里，对于 `std::string` 类型的选项，泛型 lambda 会把函数调用模板实例化为计算：

```
    val = val + val;
```

而对于其他类型的选项，例如 int 或 double，lambda 函数调用模板会实例化为计算：

```
    val *= 2;
```

注意检查 val 的类型时必须小心。这里，我们检查了 val 的类型是否能转换为 std::string。如下检查：

```
    if constexpr(std::is_same_v<decltype(val), std::string>) {
```

将不能正确工作，因为 val 的类型只可能是 int&、std::string&、long double& 这样的引用类型。

在访问器中返回值

访问器中的函数调用可以返回值，但所有返回值类型必须相同。例如：

```
using IntOrDouble = std::variant<int, double>;

std::vector<IntOrDouble> coll { 42, 7.7, 0, -0.7 };

double sum{0};
for (const auto& elem : coll) {
    sum += std::visit([] (const auto& val) -> double {
                        return val;
                      }, elem);
}
```

上面的代码会把所有选项的值加到 sum 上。如果 lambda 中没有显式指明返回类型将不能通过编译，因为自动推导的话返回类型会不同。

使用重载的 lambda 作为访问器

通过使用函数对象和 lambda 的重载器 *(overloader)*，可以定义一系列 lambda，其中最佳的匹配将会被用作访问器。

假设有一个如下定义的 overload 重载器：

tmpl/overload.hpp

```
// 继承所有基类里的函数调用运算符
template<typename... Ts>
struct overload : Ts...
{
    using Ts::operator()...;
};

// 基类的类型从传入的参数中推导
template<typename... Ts>
overload(Ts...) -> overload<Ts...>;
```

你可以为每个可能的选项提供一个 lambda，之后使用 overload 来访问 variant：

```
std::variant<int, std::string> var(42);
...
std::visit(overload { // 为当前选项调用最佳匹配的lambda
                []{int i) { std::cout << "int: " << i << '\n'; },
                [](const std::string& s) {
                    std::cout << "string: " << s << '\n';
                },
            }, var);
```

你也可以使用泛型 lambda，它可以匹配所有情况。例如，要想修改一个 variant 当前选项的值，你可以使用重载实现字符串和其他类型值翻倍：

```
auto twice = overload {
                [] (std::string& s) { s += s; },
                [] (auto& i) { i *= 2; },
             };
```

使用这个重载，对于字符串类型的选项，值将变为原本的字符串再重复一遍；而对于其他类型，将会把值乘以 2。下面展示了怎么应用于 variant：

```
std::variant<int, std::string> var(42);
std::visit(twice, var); // 值42变为84
...
var = "hi";
std::visit(twice, var); // 值"hi"变为"hihi"
```

16.3.4 异常造成的无值

如果你赋给一个 variant 新值时发生了异常，那么这个 variant 可能会进入一个非常特殊的状态：它已经失去了旧的值但还没有获得新的值。例如：

```
struct S {
    operator int() { throw "EXCEPTION"; } // 转换为int时会抛出异常
};
std::variant<double, int> var{12.2};      // 初始化为double
var.emplace<1>(S{});    // OOPS：当设为int时抛出异常
```

如果这种情况发生了，那么：

- `var.valueless_by_exception()` 会返回 `true`
- `var.index()` 会返回 `std::variant_npos`

这些都标志该 variant 当前没有值。

这种情况下有如下保证：

- 如果 `emplace()` 抛出异常，那么 `valueless_by_exception()` 可能会返回 `true`。
- 如果 `operator=()` 抛出异常且这次修改没有改变当前选项，那么 `index()` 和 `value_by_exception()` 的状态将保持不变。值的状态依赖于值类型的异常保证。

- 如果 operator=() 抛出异常且新值是新的选项，那么 variant 可能会没有值（valueless_by_exception() 可能会返回 true）。具体情况依赖于异常抛出的时机。如果发生在实际修改值之前的类型转换期间，那么 variant 将依然持有旧值。

通常情况下，如果你不再使用这种情况下的 variant 那么这些保证就足够了。如果你仍然想使用抛出了异常的 variant，你需要检查它的状态。例如：

```
std::variant<double, int> var{12.2};  // 初始化为double
try {
    var.emplace<1>(S{});              // OOPS：设置为int时抛出异常
}
catch (...) {
    if (!var.valueless_by_exception()) {
        ...
    }
}
```

16.4 使用 std::variant 实现多态的异质集合

std::variant 允许一种新式的多态性，可以用来实现异质集合。这是一种带有闭类型集合的运行时多态性。

关键在于 variant<> 可以持有多种选项类型的值。可以将元素类型定义为 variant 来实现异质的集合，这样的集合可以持有不同类型的值。因为每一个 variant 知道当前的选项，并且有了访问器接口，我们可以定义在运行时根据不同类型进行不同操作的函数/方法。同时因为 variant 有值语义，所以我们不需要指针（和相应的内存管理）或者虚函数。

16.4.1 使用 std::variant 实现几何对象

例如，假设我们要负责编写表示几何对象的库：

lib/variantpoly1.cpp

```
#include <iostream>
#include <variant>
#include <vector>
#include "coord.hpp"
#include "line.hpp"
#include "circle.hpp"
#include "rectangle.hpp"

// 所有几何类型的公共类型
using GeoObj = std::variant<Line, Circle, Rectangle>;

// 创建并初始化一个几何体对象的集合
std::vector<GeoObj> createFigure()
{
```

```
    std::vector<GeoObj> f;
    f.push_back(Line{Coord{1, 2}, Coord{3, 4}});
    f.push_back(Circle{Coord{5, 5}, 2});
    f.push_back(Rectangle{Coord{3, 3}, Coord{6, 4}});
    return f;
}

int main()
{
    std::vector<GeoObj> figure = createFigure();
    for (const GeoObj& geoobj : figure) {
        std::visit([] (const auto& obj) {
            obj.draw(); // 多态性调用draw()
        }, geoobj);
    }
}
```

首先，我们为所有可能的类型定义了一个公共类型：

```
using GeoObj = std::variant<Line, Circle, Rectangle>;
```

这三个类型不需要有任何特殊的关系。事实上它们甚至没有一个公共的基类、没有任何虚函数、接口也可能不同。例如：

lib/circle.hpp

```
#ifndef CIRCLE_HPP
#define CIRCLE_HPP

#include "coord.hpp"
#include <iostream>

class Circle {
  private:
    Coord center;
    int rad;
  public:
    Circle (Coord c, int r) : center{c}, rad{r} { }

    void move(const Coord& c) {
        center += c;
    }

    void draw() const {
        std::cout << "circle at " << center << " with radius " << rad << '\n';
    }
};

#endif
```

我们现在可以创建相应的对象并把它们以值传递给容器，最后可以得到这些类型的元素的集合：

```
std::vector<GeoObj> createFigure()
{
    std::vector<GeoObj> f;
    f.push_back(Line{Coord{1, 2}, Coord{3, 4}});
    f.push_back(Circle{Coord{5, 5}, 2});
    f.push_back(Rectangle{Coord{3, 3}, Coord{6, 4}});
    return f;
}
```

以前如果没有使用继承和多态的话是不可能写出这些代码的。以前要想实现这样的异构集合，所有的类型都必须继承自 GeoObj，并且最后将得到一个元素类型为 GeoObj 的指针的 vector。为了使用指针，必须用 new 创建新对象，这导致最后还要追踪什么时候调用 delete，或者要使用智能指针来完成（unique_ptr 或者 shared_ptr）。

现在，通过使用访问器，我们可以迭代每一个元素，并依据元素的类型“做正确的事情”：

```
std::vector<GeoObj> figure = createFigure();
for (const GeoObj& geoobj : figure) {
    std::visit([] (const auto& obj) {
                    obj.draw(); // 多态调用draw()
                }, geoobj);
}
```

这里，visit() 使用了泛型 lambda 来为每一个可能的 GeoObj 类型实例化。也就是说，当编译 visit() 调用时，lambda 将会被实例化并编译为 3 个函数：

- 为类型 Line 编译代码：

```
[] (const Line& obj) {
    obj.draw(); // 调用Line::draw()
}
```

- 为类型 Circle 编译代码：

```
[] (const Circle& obj) {
    obj.draw(); // 调用Circle::draw()
}
```

- 为类型 Rectangle 编译代码：

```
[] (const Rectangle& obj) {
    obj.draw(); // 调用Rectangle::draw()
}
```

如果这些实例中有一个不能编译，那么对 visit() 的调用也不能编译。如果所有实例都能编译，那么将保证会对所有元素类型调用相应的函数。注意生成的代码并不是 *if-else* 链。C++ 标准保证这些调用的性能不会依赖于 variant 选项的数量。

也就是说，从效率上讲，这种方式和虚函数表的方式的行为相同（通过类似于为所有 visit() 创建局部虚函数表的方式）。注意，draw() 函数不需要是虚函数。

如果对不同类型的操作不同，我们可以使用编译期 if 语句或者重载访问器 来处理不同的情况（见上边的第二个例子）。

16.4.2 使用 std::variant 实现其他异质集合

考虑如下另一个使用 std::variant<> 实现异质集合的例子：

lib/variantpoly2.cpp

```
#include <iostream>
#include <string>
#include <variant>
#include <vector>
#include <type_traits>

int main()
{
    using Var = std::variant<int, double, std::string>;

    std::vector<Var> values {42, 0.19, "hello world", 0.815};

    for (const Var& val : values) {
        std::visit([] (const auto& v) {
            if constexpr(std::is_same_v<decltype(v), const std::string&>) {
                std::cout << '"' << v << "\" ";
            }
            else {
                std::cout << v << ' ';
            }
        }, val);
    }
}
```

我们又一次定义了自己的类型来表示若干可能类型中的一个：

```
using Var = std::variant<int, double, std::string>;
```

我们可以用它创建并初始化一个异质的集合：

```
std::vector<Var> values {42, 0.19, "hello world", 0.815};
```

注意我们可以用若干异质的元素来实例化 vector，因为它们都能自动转换为 variant 类型。然而，如果我们还传递了一个 long 类型的初值，上面的初始化将不能编译，因为编译器不能决定将它转换为 int 还是 double。

当我们第迭代元素时，我们使用了访问器来调用相应的函数。这里使用了一个泛型 lambda。lambda 为 3 种可能的类型分别实例化了一个函数调用。为了对字符串进行特殊的处理（在输出值时用双引号包括起来），我们使用了编译期 if 语句：

```
for (const Var& val : values) {
    std::visit([] (const auto& v) {
        if constexpr(std::is_same_v<decltype(v), const std::string&>) {
```

```
            std::cout << '"' << v << "\" ";
        }
        else {
            std::cout << v << ' ';
        }
    }, val);
}
```

这意味着输出将是：

```
42 0.19 "hello world" 0.815
```

通过使用重载访问器，我们可以像下面这样实现：

```
for (const auto& val : values) {
    std::visit(overload {
        [] (const auto& v) {
            std::cout << v << ' ';
        },
        [] (const std::string& v) {
            std::cout << '"' << v "\" ";
        }
    }, val);
}
```

然而，注意这样可能会陷入重载匹配的问题。有的情况下泛型 lambda（即函数模板）匹配度比隐式类型更高，这意味着可能会调用错误的类型。

16.4.3 比较多态的 variant

让我们来总结一下使用 std::variant 实现多态的异构集合的优点和缺点：

优点有：

- 你可以使用任意类型并且这些类型不需要有公共的基类（这种方法是非侵入性的）
- 你不需要使用指针来实现异质集合
- 不需要 virtual 成员函数
- 值语义（不会出现访问已释放内存或内存泄露等问题）
- vector 中的元素是连续存放在一起的（原本指针的方式所有元素是散乱分布在堆内存中的）

缺点有：

- 闭类型集合（你必须在编译期指定所有可能的类型）
- 每个元素的大小都是所有可能的类型中最大的（当不同类型大小差距很大时这是个问题）
- 拷贝元素的开销可能会更大

一般来说，我并不确定是否要推荐默认使用 std::variant<> 来实现多态。一方面这种方法很安全（没有指针，意味着没有 new 和 delete），也不需要虚函数。然而另一方面，使用访问器有一些笨拙，有时你可能会需要引用语义（在多个地方使用同一个对象），还有在某些情形下并不能在编译期确定所有的类型。

性能开销也有很大不同。没有了 new 和 delete 可能会减少很大开销。但另一方面，以值传递对象又可能会增大很多开销。在实践中，你必须自己测试对你的代码来说哪种方法效率更高。在不同的平台上，我已经观测

到性能上的显著差异了。

16.5 std::variant<> 的特殊情况

特定类型的 variant 可能导致特殊或着出乎意料的行为。

16.5.1 同时有 bool 和 std::string 选项

如果一个 std::variant<> 同时有 bool 和 std::string 选项，赋予一个字符串字面量可能会导致令人惊奇的事，因为字符串字面量会优先转换为 bool，而不是 std::string。例如：

```
std::variant<bool, std::string> v;
v = "hi";   // OOPS：设置bool选项
std::cout << "index: " << v.index() << '\n';
std::visit([](const auto& val) {
                std::cout << "value: " << val << '\n';
            }, v);
```

这段代码片段将会有如下输出：

```
index: 0
value: true
```

可以看出，字符串字面量会被解释为初始化 variant 的 bool 选项为 true （因为指针不是 0 所以是 true）。

这里有一下修复这个赋值问题的方法：

```
v.emplace<1>("hello");           // 显式赋值给第二个选项

v.emplace<std::string>("hello"); // 显式赋值给string选项

v = std::string{"hello"};        // 确保用string赋值

using namespace std::literals;   // 确保用string赋值
v = "hello"s;
```

参见https://wg21.link/p0608进一步了解关于这个问题的讨论。

16.6 后记

variant 对象由 Axel Naumann 于 2005 年https://wg21.link/n4218中首次提出，并指定 Boost.Variant 作为参考实现。最终被接受的提案由 Axel Naumann 发表于 https://wg21.link/p0088r3。

Tony van Eerd 在https://wg21.link/p0393r3中显著改进了比较运算符的语义。Vicente J. Botet Escriba 在https://wg21.link/p0032r3中统一了 std::variant 和 std::optional<> 以及 std::any 的 API。Jonathan Wakely 在https://wg21.link/p0504r0中修正了 in_place 标记类型的行为。禁止使用引用、不完全类型、数组作为选项类型和禁止空 variant 的限制由 Erich Keane 在https://wg21.link/p0510r0中提出。C++17 标准发布之后，Mike Spertus、Walter E、Brown 和 Stephan T. Lavavej 在https://wg21.link/p0739r0中修复了一些小缺陷。

Chapter 17

std::any

一般来说，C++ 是一门类型绑定和类型安全的语言。值对象被声明为确定的类型，这个类型定义了所有可能的操作、也定义了对象的行为。而且，对象不能改变自身的类型。

std::any 是一种在保证类型安全的基础上还能改变自身类型的值类型。也就是说，它可以持有任意类型的值，并且它知道自己当前持有的值是什么类型的。当声明一个这种类型的对象时不需要指明所有可能的类型。

关键在于 std::any 对象同时包含了值和值的类型。因为内含的值可以有任意的大小，所以可能会在堆上分配内存。然而，实现应该尽量避免为小类型的值例如 int 在堆上分配内存。

对于一个 std::any 对象，如果你赋值为一个字符串，它将会分配内存并拷贝字符串，并存储记录当前的值是一个字符串。之后，可以使用运行时检查来判断当前的值的类型。为了将当前的值转换为真实的类型，必须要使用 any_cast<>。

和 std::optional<>、std::variant<> 一样，std::any 对象有值语义。也就是说，拷贝被实现为深拷贝，会创建一个在自己的内存中持有当前值的独立对象。因为可能会使用堆内存，所以拷贝 std::any 的开销一般都很大。更推荐以引用传递对象，或者 move 值。std::any 支持部分 move 语义。

17.1 使用 std::any

下面的示例演示了 std::any 的核心能力：

```
std::any a;             // a为空
std::any b = 4.3;       // b有类型为double的值4.3
a = 42;                 // a有类型为int的值42
b = std::string{"hi"};  // b有类型为std::string的值"hi"

if (a.type() == typeid(std::string)) {
    std::string s = std::any_cast<std::string>(a);
    useString(s);
}
else if (a.type() == typeid(int)) {
    useInt(std::any_cast<int>(a));
}
```

你可以声明一个空的 std::any，也可以用一个指定类型的值初始化。如果传递了初始值，std::any 内含的值的类型将变为初始值的类型。

通过使用成员函数 type()，你可以检查内含值的类型和某一个类型的 ID 是否相同。如果对象是空的，类型 ID 将等于 typeid(void)。

为了访问内部的值，你必须使用 std::any_cast<> 将它转换为真正的类型：

```
auto s = std::any_cast<std::string>(a);
```

如果转换失败，可能是因为对象为空或者与内部值的类型不匹配，这是会抛出一个 std::bad_any_cast 异常。因此，如果不进行检查并且不知道当前类型，你最好像下面这样使用：

```
try {
    auto s = std::any_cast<std::string>(a);
    ...
}
catch (std::bad_any_cast& e) {
    std::cerr << "EXCEPTION: " << e.what() << '\n';
}
```

注意 std::any_cast<> 会创建一个指定类型的对象。例如这个例子中，如果你用 std::string 作为 std::any_cast<> 的模板参数，它将创建一个临时的字符串 (prvalue)，然后使用临时字符串初始化新的对象 s。如果不需要初始化其他变量，更推荐转换为引用类型来避免创建临时对象：

```
std::cout << std::any_cast<const std::string&>(a);
```

如果想要修改当前的值，也需要转换为相应的引用类型：

```
std::any_cast<std::string&>(a) = "world";
```

你也可以对一个 std::any 对象的地址调用 std::any_cast。这种情况下，如果类型匹配那么转换结果将会是一个相应类型的指针，否则将返回 nullptr：

```
auto p = std::any_cast<std::string>(&a);
if (p) {
    ...
}
```

或者，可以使用新的带初始化的 if 语句：

```
if (auto p = std::any_cast<std::string>(&a); p != nullptr) {
    ...
}
```

或者：

```
if (auto p = std::any_cast<std::string>(&a)) {
    ...
}
```

为了清空一个 std::any 对象，你可以调用：

```
a.reset();   // 清空对象
```

或者：

```
a = std::any{};
```

或者：

```
a = {};
```

你也可以检查对象是否有值：

```
if (a.has_value()) {
    ...
}
```

注意存储值时类型会退化（数组转换为指针，顶层引用和 const 被忽略）。对于字符串字面量，值类型将是 const char*。为了使用 std::any_cast<> 进行转换，你必须显式指明这个类型：

```
std::any a = "hello";   // type()是const char*
if (a.type() == typeid(const char*)) {  // true
    ...
}
if (a.type() == typeid(std::string)) {  // false
    ...
}
std::cout << std::any_cast<const char*>(a) << '\n'; // OK
std::cout << std::any_cast<std::string>(a) << '\n'; // EXCEPTION
```

这基本就是 std::any 支持的所有操作了。没有定义比较运算符（这意味着你不能比较或者排序对象）。没有定义哈希函数，也没有定义 value() 成员函数。而且，因为类型只有在运行时才能获取，所以也不能使用泛型 lambda 来独立于类型处理当前的值。你只能使用运行时的 std::any_cast<> 函数来处理当前的值，这意味着处理当前值时你需要指定类型来重入 C++ 的类型系统。

然而你，你可以把 std::any 对象放置在容器中。例如：

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

v.push_back(42);
std::string s = "hello";
v.push_back(s);

for (const auto& a : v) {
    if (auto pa = std::any_cast<const std::string>(&a); pa != nullptr) {
        std::cout << "string: " << *pa << '\n';
    }
    else if (auto pa = std::any_cast<const int>(&a); pa != nullptr) {
        std::cout << "int: " << *pa << '\n';
    }
}
```

注意你应该总是使用这样的 if-else 链。这里不能使用 switch 语句。

17.2 std::any 类型和操作

这一节详细描述 std::any 的类型和操作。

17.2.1 Any 类型

在头文件 <any> 中，C++ 标准库以如下方式定义了类 std::any：

```
namespace std {
    class any;
}
```

也就是说，std::any 根本就不是模板类。

另外，还定义了下面的类型和对象：

- 异常类 std::bad_any_cast，当转换失败时会抛出这种异常。这个类派生自 std::bad_cast，后者又派生自 std::exception。。

std::any 类也使用了定义在头文件 <utility> 中的 std::in_place_type 对象（类型是 std::in_place_type_t）。

17.2.2 Any 操作

表std::any 的操作列出了 std::any 的所有操作：

操作	效果
构造函数	创建一个 any 对象（可能会调用底层类型的构造函数）
make_any()	创建一个 any 对象（传递参数来初始化）
析构函数	销毁 any 对象
=	赋予新值
emplace<T>()	赋予一个类型 T 的新值
reset()	销毁 any 类型（使对象变为空）
has_value()	返回对象是否持有值
type()	以 std::type_info 对象返回当前类型
any_cast<T>()	将当前值转换为类型 T 的值（如果类型不正确将抛出异常/返回 nullptr）
swap()	交换两个对象的值

Table 17.1: std::any 的操作

构造函数

默认情况下，std::any 被初始化为空。

```
std::any a1;        // a1是空的
```

如果传递值来初始化，内含值的类型将是它退化后的类型：

```
std::any a2 = 42;       // a2包含int类型的值
std::any a3 = "hello";  // a2包含const char*类型的值
```

为了使内部值的类型和初始值的类型不同，你需要使用 in_place_type 标记：

```
std::any a4{std::in_place_type<long>, 42};
std::any a5{std::in_place_type<std::string>, "hello"};
```

传给 in_place_type 的类型也可能退化。下面的代码声明了一个持有 const char* 的对象：

```
std::any a5b{std::in_place_type<const char[6]>, "hello"};
```

为了用多个参数初始化 std::any 对象，你必须手动创建对象或者你可以添加 std::in_place_type 作为第一个参数（因为内含类型不能直接从多个初始值推导出来）：

```
std::any a6{std::complex{3.0, 4.0}};
std::any a7{std::in_place_type<std::complex<double>>, 3.0, 4.0};
```

你甚至可以传递初值列和其他参数：

```
// 用一个以lambda为排序准则的set初始化std::any对象
auto sc = [] (int x, int y) {
                return std::abs(x) < std::abs(y);
          };
std::any a8{std::in_place_type<std::set<int, decltype(sc)>>,
            {4, 8, -7, -2, 0, 5}, sc};
```

注意还有一个快捷函数 make_any()，它可以接受一个或多个参数（不需要使用 in_place_type 参数）。你必须显式指明初始化的类型（即使只有一个参数它也不会自动推导类型）：

```
auto a10 = std::make_any<float>(3.0);
auto a11 = std::make_any<std::string>("hello");
auto a13 = std::make_any<std::complex<double>>(3.0, 4.0);
auto a14 = std::make_any<std::set<int, decltype(sc)>> ({4, 8, -7, -2, 0, 5}, sc);
```

修改值

可以使用赋值操作和 emplace() 来修改值。例如：

```
std::any a;

a = 42;         // a含有int类型的值
a = "hello";    // a含有const char*类型的值
a.emplace<std::string>("hello"); // a含有std::string类型的值
a.emplace<std::complex<double>>(4.4, 5.5); // a含有std::complex<double>类型的值
```

访问值

为了访问内含的值，你必须使用 **std::any_cast<>** 将它转换为真实的类型。例如为了把值转换成 string，你有以下几种方法：

```
std::any_cast<std::string>(a);        // 返回值的拷贝

std::any_cast<std::string&>(a);       // 通过引用获取写权限

std::any_cast<const std::string&>(a); // 通过引用获取读权限
```

当移除掉顶层引用之和const之后如果有相同的类型ID说明类型匹配。如果转换失败，将会抛出一个std::bad_any_cast异常。

为了避免处理异常，你可以传递any对象的地址。当因为类型不匹配导致转换失败时它会返回nullptr：

```
if (auto sp{std::any_cast<std::string>(&a)}; sp != nullptr) {
    ... // 使用*sp获取a的值的写权限
}

if (auto sp{std::any_cast<const std::string>(&a)}; sp != nullptr) {
    ... // 使用*sp获取a的值的读权限
}
```

注意这里，转换为应用将导致运行时错误：

```
std::any_cast<std::string&>(&a);    // 运行时错误
```

move语义

std::any也支持move语义。然而，注意前提是底层类型要支持拷贝语义。这是因为，**move-only类型**不支持作为内含的值类型。

处理move语义的最佳方式可能不是很明显，你可以这样做：

```
std::string s("hello, world!");

std::any a;
a = std::move(s);   // 把s移进a

s = std::move(std::any_cast<std::string&>(a));  // 把a中的string移动到s
```

注意像通常一样，值被移走的对象处于仍然有效但是值未定义的状态。因此，你可以继续将a用作字符串，只要不对他的值做任何假设。下面的语句将不会输出"NIL"，值被移走的字符串一般是空字符串（但也可能有其他的值）：

```
std::cout << (a.has_value() ? std::any_cast<std::string>(a) : std::string("NIL"));
```

注意：

```
s = std::any_cast<std::string>(std::move(a));
```

也能生效，但需要一次额外的move。

直接转换成右值引用将不能通过编译：

```
s = std::any_cast<std::string&&>(a);    // 编译期error
```

注意和如下调用

```
a = std::move(s);   // 把s移进a
```

相比，下面的代码有可能不能工作（即使这是C++标准里的一个例子）：

```
std::any_cast<string&>(a) = std::move(s); // OOPS：a必须持有string
```

只有当a已经包含有一个std::string类型的值时这段代码才能工作。否则，在我们赋予新值之前，这个转换就会抛出std::bad_any_cast异常。

17.3 后记

Any对象是由Kevlin Henney和Beman Dawes于2006年在https://wg21.link/n1939中首次提出，并指定Boost.Any作为参考实现。这个类因Beman Dawes、Kevlin Henney、Daniel Krügler在https://wg21.link/n3804中的提案被Library Fundamentals TS采纳。

后来这个类因Beman Dawes和Alisdair Meredith在https://wg21.link/p0220r1发表的提案而和其他组件一起被C++17标准采纳。

Vicente J.Botet Escriba在https://wg21.link/p0032r3中统一了std:: any和std::variant<>以及std::optional<>的API。Jonathan Wakely在https://wg21.link/p0504r0中修正了in_place标记类型的行为。

Chapter 18

std::byte

通过 std::byte，C++17 引入了一个类型来代表内存的最小单位：字节。std::byte 本质上代表一个字节的值，但并不能进行数字或字符的操作，也不对每一位进行解释。对于不需要数字计算和字符序列的场景，这样会更加类型安全。

然而你，注意 std::byte 实现的大小和 unsigned char 一样，这意味着它并不保证是 8 位，可能会更多。

18.1 使用 std::byte

下面的代码展示了 std::byte 的核心能力：

```
#include <cstddef>  // for std::byte

std::byte b1{0x3F};
std::byte b2{0b1111'0000};

std::byte b3[4] {b1, b2, std::byte{1}}; // 4个字节（最后一个是0）

if (b1 == b3[0]) {
    b1 <<= 1;
}

std::cout << std::to_integer<int>(b1) << '\n';  // 输出：126
```

这里，我们定义了两个初始值不同的字节。b2 的初始化使用了两个 C++14 引入的特性：

- 前缀 0b 允许定义二进制字面量
- 数字分隔符 ' 可以增强数字字面量的可读性（它可以被放置在数字字面量中任意两个数字之间）。

注意列表初始化（使用花括号初始化）是唯一可以直接初始化 std::byte 对象的方法。所有其他的形式都不能编译：

```
std::byte b1{42};       // OK（因为自从C++17起所有枚举都有固定的底层类型）
std::byte b2(42);       // ERROR
std::byte b3 = 42;      // ERROR
```

```
std::byte b4 = {42};    // ERROR
```

这是将 std::byte 实现为枚举类型的一个直接后果。花括号初始化使用了新的用整数值初始化有作用域的枚举特性。

这里不能使用隐式类型转换，这意味着你必须显式对整数值进行转换才能初始化字节数组：

```
std::byte b5[] {1};             // ERROR
std::byte b6[] {std::byte{1}};  // OK
```

如果没有初始化，std::byte 将是栈上的值未定义的对象：

```
std::byte b;    // 值未定义
```

像通常一样（除了原子类型），你可以使用花括号强制初始化为每一位为 0：

```
std::byte b{};  // 等价于b{0}
```

std::to_integer<> 允许你将 std::byte 对象转换为整数值（包括 bool 和 char 类型）。如果没有转换，将不能使用输出运算符。注意因为这个转换函数是模板，所以你需要使用带有 std:: 的完整名称：

```
std::cout << b1;    // ERROR
std::cout << to_integer<int>(b1);       // ERROR（ADL在这里不起作用）
std::cout << std::to_integer<int>(b1);  // OK
```

也可以使用 using 声明（但请只在局部作用域中这么做）：

```
using std::to_integer;
...
std::cout << to_integer<int>(b1);       // OK
```

如果要将 std::byte 用作布尔值也需要这样的转换。例如：

```
if (b2) ...                         // ERROR
if (b2 != std::byte{0}) ...         // OK
if (to_integer<bool>(b2)) ...       // ERROR（ADL在这里不起作用）
if (std::to_integer<bool>(b2)) ...  // OK
```

因为 std::byte 被实现为底层类型是 unsigned char 的枚举类型，所以它的大小总是 1：

```
std::cout << sizeof(b);             // 总是1
```

它的位数依赖于底层类型 unsigned char 的位数，你可以通过标准数字限制来获取位数：

```
std::cout << std::numeric_limits<unsigned char>::digits; // std::byte的位数
```

这等价于：

```
std::cout << std::numeric_limits<std::underlying_type_t<std::byte>>::digits;
```

大多数时候结果是 8，但在有些平台上可能不是。

18.2 std::byte 类型和操作

这一节详细描述 std::byte 类型和操作。

18.2.1 std::byte 类型

在头文件 <cstddef> 中，C++ 标准库以如下方式定义了 std::byte：

```
namespace std {
    enum class byte : unsigned char {
    };
}
```

也就是说，std::byte 不是别的，只是一个带有一些位运算符操作的有作用域的枚举类型：

```
namespace std {
    ...
    template<typename IntType>
    constexpr byte  operator<<  (byte  b, IntType shift) noexcept;
    template<typename IntType>
    constexpr byte& operator<<= (byte& b, IntType shift) noexcept;
    template<typename IntType>
    constexpr byte  operator>>  (byte  b, IntType shift) noexcept;
    template<typename IntType>
    constexpr byte& operator>>= (byte& b, IntType shift) noexcept;

    constexpr byte& operator|= (byte& l, byte r) noexcept;
    constexpr byte  operator|  (byte  l, byte r) noexcept;
    constexpr byte& operator&= (byte& l, byte r) noexcept;
    constexpr byte  operator&  (byte  l, byte r) noexcept;
    constexpr byte& operator^= (byte& l, byte r) noexcept;
    constexpr byte  operator^  (byte  l, byte r) noexcept;
    constexpr byte  operator~  (byte b) noexcept;

    template<typename IntType>
    constexpr IntType to_integer (byte b) noexcept;
}
```

18.2.2 std::byte 操作

表std::byte 的操作列出了 std::byte 的所有操作。

转换为整数类型

用 to_integer<>() 可以把 std::byte 转换为任意基本整数类型（bool、字符类型或者整数类型）。这也是必须的，例如为了将 std::byte 和整数值比较或者将它用作条件：

```
if (b2) ...                         // ERROR
if (b2 != std::byte{0}) ...         // OK
if (to_integer<bool>(b2)) ...       // ERROR（ADL 在这里不生效）
if (std::to_integer<bool>(b2)) ...  // OK
```

另一个例子使用它的例子是std::byte I/O。

to_integer<>() 使用 static_cast 来把 unsigned char 转换为目标类型。例如：

操作	效果
构造函数	创建一个字节对象（调用默认构造函数时值未定义）
析构函数	销毁一个字节对象（什么也不做）
=	赋予新值
==、!=、<、<=、>、>=	比较字节对象
<<、>>、|、& 、^、~	二元位运算符
<<=、>>=、|=、& =、^=	修改自身的位运算符
to_integer<T>()	把字节对象转换为整数类型 T
sizeof()	返回 1

Table 18.1: std::byte 的操作

```
std::byte ff{0xFF};
std::cout << std::to_integer<unsigned int>(ff); // 255
std::cout << std::to_integer<int>(ff);          // 也是255
std::cout << static_cast<int>(std::to_integer<signed char>(ff)); // -1
```

std::byte 的 I/O

std::byte 没有定义输入和输出运算符，因此不得不把它转换为整数类型再进行 I/O：

```
std::byte b;
...
std::cout << std::to_integer<int>(b);   // 以十进制值打印出值
std::cout << std::hex << std::to_integer<int>(b); // 以十六进制打印出值
```

通过使用 std::bitset<>，你可以以二进制输出值（一串位序列）：

```
#include <cstddef>  // for std::byte
#include <bitset>   // for std::bitset
#include <limits>   // for std::numeric_limits

std::byte b1{42};
using ByteBitset = std::bitset<std::numeric_limits<unsigned char>::digits>;
std::cout << ByteBitset{std::to_integer<unsigned>(b1)};
```

上例中 using 声明定义了一个位数和 std::byte 相同的 bitset 类型，之后把字节对象转换为整数来初始化一个这种类型的对象，最后输出了该对象。最后值 42 将会有如下输出（假设一个 char 是 8 位）：

```
00101010
```

另外，你可以使用 std::underlying_type_t<std::byte> 代替 unsigned char，这样 using 声明的目的将更明显。

你也可以使用这种方法把 std::byte 的二进制表示写入一个字符串：

```
std::string s = ByteBitset{std::to_integer<unsigned>(b1)}.to_string();
```

如果你已经有了一个字符序列，你也可以像下面这样使用to_chars()：[1]

```
#include <charconv>
#include <cstddef>

std::byte b1{42};
// 译者注：此处原文写的是
// int value = 42;
// 应是作者笔误

char str[100];
std::to_chars_result res = std::to_chars(str, str+99, std::to_integer<int>(b1), 2);
*res.ptr = '\0';    // 确保最后有一个空字符结尾
```

注意这种形式将不会写入前导0，这意味着对于值42，最后的结果是（假设一个char有8位）：

```
101010
// 译者注：此处原文写的是
// 1111110
// 应是作者笔误
```

可以使用相似的方式进行输入：以整数、字符串或bitset类型读入并进行转换。例如，你可以像下面这样实现读入字节对象的二进制表示的输入运算符：

```
std::istream& operator>> (std::istream& strm, std::byte& b)
{
    // 读入一个bitset
    std::bitset<std::numeric_limits<unsigned char>::digits> bs;
    strm >> bs;
    // 如果没有失败就转换为std::byte
    if (!std::cin.fail()) {
        b = static_cast<std::byte>(bs.to_ulong());  // OK
    }
    return strm;
}
```

注意我们必须使用static_cast<>()来把bitset转换成的unsigned long转换为std::byte，列表初始化将不能工作，因为会发生窄化：[2]

```
b = std::byte{bs.to_ulong()};   // ERROR：发生窄化
```

并且我们也没有其他的初始化方法了。

另外，你也可以使用from_chars来从给定的字符序列读取：

```
#include <charconv>

const char* str = "101001";
int value;
std::from_chars_result res =
```

[1]感谢Daniel Krügler指出这一点。

[2]使用gcc/g++时，如果没有编译选项-pedantic-errors，窄化初始化也可能通过编译。

```
    std::from_chars(str, str+6, // 要读取的字符范围
                    value,      // 读取后存入的对象
                    2);         // 2进制
```

18.3 后记

Neil MacIntosh 在发表于https://wg21.link/p0298r0的提案中首次提出 std::byte。最终被接受的提案由 Neil MacIntosh 发表于https://wg21.link/p0298r3。

Chapter 19

字符串视图 (String Views)

在 C++17 中，C++ 标准库引入了一个特殊的字符串类：`std::string_view`，它能让我们像处理字符串一样处理字符序列，而不需要为它们分配内存空间。也就是说，`std::string_view` 类型的对象只是引用一个外部的字符序列，而不需要持有它们。因此，一个字符串视图对象可以被看作字符串序列的引用。

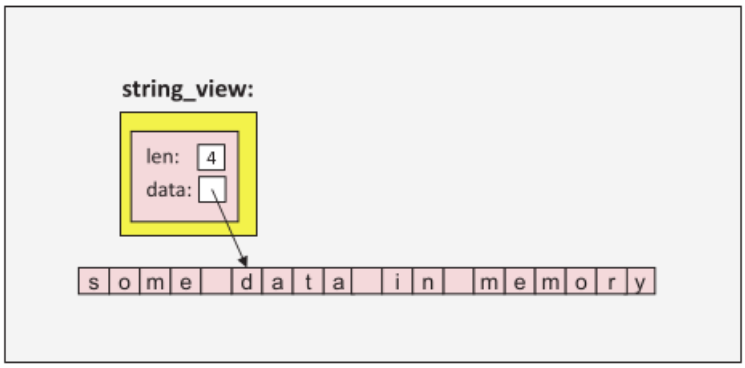

Figure 19.1: 字符串视图对象

使用字符串视图的开销很小，速度却很快（以值传递一个 `string_view` 的开销总是很小）。然而，它也有一些潜在的危险，就和原生指针一样，在使用 `string_view` 时也必须程序员自己来保证引用的字符串序列是有效的。

19.1 和 `std::string` 的不同之处

和 `std::string` 相比，`std::string_view` 对象有以下特点：

- 底层的字符序列是只读的。没有操作可以修改底层的字符。你只能赋予一个新值、交换值、把视图缩小为字符序列的子序列。
- 字符序列不保证有空字符终止。因此，字符串视图并不是一个空字符终止的字节流 *(NTBS)*。

- `data()` 返回的值可能是 `nullptr`。例如，当用默认构造函数初始化一个字符串视图之后，调用 `data()` 将返回 `nullptr`。
- 没有分配器支持。

因为可能返回 `nullptr`，并且可能不以空字符结尾，所以你使用 `operator[]` 或者 `data()` 之前应该总是使用 `size()` 获取长度（除非你已经知道了长度）。

19.2 使用字符串视图

字符串视图有两个主要的应用：

1. 你可能已经分配或者映射了字符序列或者字符串的数据，并且想在不分配更多内存的情况下使用这些数据。典型的例子是内存映射文件或者处理大文本文件中的子串。
2. 你可能想提升接收字符串为参数并以只读方式使用它们的函数/操作的性能，且这些函数/操作不需要结尾的空字符。

   这种情况的一种特殊形式是想以类似于 `string` 的 API 来处理字符串字面量对象：

```
std::string_view hello{"hello world"};
```

第一个应用通常意味着只需要传递字符串视图，然而程序逻辑必须保证底层的字符序列仍然有效（即内存映射文件不会中途取消映射）。你也可以在任何时候使用一个字符串视图来初始化或赋值给 `std::string`。

注意不要把字符串视图当作“更好的 string”来使用。这样可能导致性能问题和一些运行时错误。请仔细的阅读下面的小节。

19.3 使用字符串视图作为参数

下面是使用字符串视图作为只读字符串的第一个例子，这个例子定义了一个函数将传入的字符串视图作为前缀，之后打印一个集合中的元素：

```
#include <string_view>

template<typename T>
void printElems(const T& coll, std::string_view prefix = {})
{
    for (const auto& elem : coll) {
        if (prefix.data()) {    // 排除nullptr
            std::cout << prefix << ' ';
        }
        std::cout << elem << '\n';
    }
}
```

这里，把函数参数声明为 `std::string_view`，与声明为 `std::string` 比较起来，可能会减少一次分配堆内存的调用。具体的情况依赖于是否传递的是短字符串和是否使用了短字符串优化 (SSO)。例如，如果我们像下面这么声明：

```
template<typename T>
void printElems(const T& coll, const std::string& prefix = {});
```

然后传递了一个字符串字面量，那么这个调用会创建一个临时的string，这将会在堆上分配一次内存，除非使用了短字符串优化。通过使用字符串视图，将不会分配内存因为字符串视图只指向字符串字面量。

注意在使用值未知的字符串视图前应该检查`data()`来排除`nullptr`。这里为了避免写入额外的空格分隔符，必须检查`nullptr`。值为`nullptr`的字符串视图可以用来写入到输出流，但不能用来写入到字符序列。

另一个例子是使用字符串视图作为只读的字符串来改进`std::optional`章节的`asInt()`示例，改进的方法就是把参数声明为字符串视图：

lib/asint.cpp

```
#include <optional>
#include <string_view>
#include <charconv> // for from_chars()
#include <iostream>

// 尝试将string转换为int
std::optional<int> asInt(std::string_view sv)
{
    int val;
    // 把字符串序列读入int：
    auto [ptr, ec] = std::from_chars(sv.data(), sv.data() + sv.size(), val);
    // 如果有错误码，就返回空值：
    if (ec != std::errc{}) {
        return std::nullopt;
    }
    return val;
}

int main()
{
    for (auto s : {"42", "  077", "hello", "0x33"}) {
        // 尝试把s转换为int，并打印结果
        std::optional<int> oi = asInt(s);
        if (oi) {
            std::cout << "convert '" << s << "' to int: " << *oi << "\n";
        }
        else {
            std::cout << "can't convert '" << s << "' to int\n";
        }
    }
}
```

将`asInt()`的参数改为字符串视图之后进行很多修改。首先，没有必要再使用`std::stoi()`来转换为整数，因为`stoi()`的参数是string，而根据string view创建string的开销相对较高。

作为代替，我们向新的标准库函数`from_chars`传递了字符范围。这个函数需要两个字符指针为参数，分

别代表字符序列的起点和终点，并进行转换。注意这意味着我们可以避免单独处理空字符串视图，这种情况下 `data()` 返回 `nullptr`，`size()` 返回 0，因为从 `nullptr` 到 `nullptr+0` 是一个有效的空范围（任何指针类型都支持与 0 相加，并且不会有任何效果）。

`std::from_chars()` 返回一个 `std::from_chars_result`，这个结构体有两个成员，一个指针 `ptr` 指向未被处理的第一个字符，另一个 `sc` 的类型是 `std:errc`，`std::errc` 代表没有错误。因此，使用返回值中的 `ec` 成员初始化 `ec` 之后（使用了结构化绑定），下面的检查将在转换失败时返回`nullopt`：

```
if (ec != std::errc{}) {
    return std::nullopt;
}
```

使用字符串视图还可以显著提升子字符串排序的性能。

19.3.1 字符串视图有害的一面

通常“智能对象”例如智能指针会比相应的语言特性更安全（至少不会更危险）。因此，你可能会有一种印象：字符串视图是一种字符串的引用，应该比字符串引用更安全或者至少一样安全。然而不幸的是，事实并不是这样的，字符串视图远比字符串引用或者智能指针更危险。它们的行为更近似于原生字符指针。

不要把临时字符串赋值给字符串视图

考虑声明一个返回新字符串的函数：

```
std::string retString();
```

使用返回值总是安全的：

- 用返回值来初始化一个 string 或者用 `auto` 声明的对象是安全的：

```
std::string s1 = retString();   // 安全
```

- 用返回值初始化 string 引用，如果可行的话，只要只在局部使用将是安全的。因为引用会延长返回值的生命周期：

```
std::string& s2 = retString();  // 编译期ERROR（缺少const）

const std::string& s3 = retString();  // s3延长了返回的string的生命周期
std::cout << s3 << '\n';        // OK
auto&& s4 = retString();        // s4延长了返回的string的生命周期
std::cout << s4 << '\n';        // OK
```

字符串视图没有这么安全，它既不拷贝也不延长返回值的生命周期：

```
std::string_view sv = retString(); // sv不延长返回值的生命周期
std::cout << sv << '\n';           // 运行时ERROR：返回值已经被销毁
```

这里，在第一条语句结束时返回的字符串已经被销毁了，所以使用指向它的字符串视图 `sv` 将会导致未定义的运行时错误。

这个问题类似于如下调用：

```
const char* p = retString().c_str();
```

或者：

```
auto p = retString().c_str();
```

因此，当使用返回的字符串视图时必须非常小心：[1]

```
// 非常危险：
std::string_view substring(const std::string& s, std::size_t idx = 0);

// 因为：
auto sub = substring("very nice", 5); // 返回临时string的视图
                                      // 但是临时string已经被销毁了
std::cout << sub << '\n';     // 运行时ERROR：临时字符串s已经被销毁
```

返回值类型是字符串视图时不要返回字符串

返回值类型是字符串视图时返回字符串是非常危险的。因此，你不应该像下面这样写：

```
class Person {
    std::string name;
  public:
    ...
    std::string_view getName() const {  // 不要这么做
        return name;
    }
};
```

这是因为，下面的代码将会产生运行时错误并导致未定义行为：

```
Person createPerson();
auto n = createPerson().getName();  // OOPS：delete临时字符串
std::cout << "name: " << n << '\n'; // 运行时错误
```

如果把 `getName()` 改为返回一个值或者引用就不会有这个问题了，因为 `n` 将会变为返回值的拷贝。

函数模板应该使用 `auto` 作为返回值类型

注意无意中把字符串作为字符串视图返回是很常见的。例如，下面的两个函数单独看起来都很有用：

```
// 为字符串视图定义+，返回string
std::string operator+ (std::string_view sv1, std::string_view sv2) {
    return std::string(sv1) + std::string(sv2);
}

// 泛型连接函数
template<typename T>
T concat (const T& x, const T& y) {
    return x + y;
}
```

[1] 可以在这里找到关于这个例子的讨论：https://groups.google.com/a/isocpp.org/forum/#!topic/std-discussion/Gj5gt5E-po8

然而，如果把它们一起使用就很容易导致运行时错误：

```
std::string_view hi = "hi";
auto xy = concat(hi, hi);   // xy是std::string_view
std::cout << xy << '\n';    // 运行时错误：指向的string已经被销毁了
```

这样的代码很可能在无意中被写出来。真正的问题在于`concat()`的返回类型。如果你把返回类型交给编译期自动推导，上面的例子将会把`xy`初始化为`std::string`：

```
// 改进的泛型连接函数
template<typename T>
auto concat (const T& x, const T& y) {
    return x + y;
}
```

不要在调用链中使用字符串视图来初始化字符串

在一个中途或者最后需要字符串的调用链中使用字符串视图可能会适得其反。例如，如果你定义了一个有如下构造函数的类`Person`：

```
class Person {
    std::string name;
  public:
    Person (std::string_view n) // 不要这样做
        : name {n} {
    }
    ...
};
```

传递一个字符串字面量或者之后还会使用的 string 是没问题的：

```
Person p1{"Jim"};       // 没有性能开销
std::string s = "Joe";
Person p2{s};           // 没有性能开销
```

然而，使用 move 的 string 将会导致不必要的开销。因为传入的 string 首先要隐式转换为字符串视图，之后会再用它创建一个新的 string，会再次分配内存：

```
Person p3{std::move(s)};  // 性能开销：move被破坏
```

不要在这里使用`std::string_view`。以值传参然后把值 move 到成员里仍然暗示最佳的方案（除非你想要双倍的开销）：

```
class Person {
    std::string name;
  public:
    Person (std::string n) : name{std::move(n)} {
    }
    ...
};
```

如果我们必须创建/初始化一个 string，直接作为参数创建可以让我们在传参时享受所有可能的优化。知道我们简单的把它 move 到成员，这个操作开销很小：

```
std::string newName()
{
    ...
    return std::string{...};
}

Person p{newName()};
```

强制省略拷贝特性将把新 string 的实质化过程推迟到值被传递给构造函数。构造函数里我们有了一个叫 n 的 string，这是一个有内存地址的对象（一个广义左值 *(glvalue)*）。之后把这个对象的值 move 到成员 `name` 里。

安全使用字符串视图的总结

总结起来就是**小心的使用 `std::string_view`**，也就是说你应该按下面这样调整你的编码风格：

- 不要在把参数传递给 string 的 API 中使用 string view。
  - 不要用 string view 形参来初始化 string 成员。
  - 不要把 string 设为 string view 调用链的终点。
- 不要返回 string view。
  - 除非它只是转发输入的参数，或者你可以标记它很危险，例如，通过命名来体现危险性。
- **函数模板**永远不应该返回传入的泛型参数的类型 **T**。
  - 作为替代，返回 `auto` 类型。
- 永远不要用返回之类初始化为 string view。
- **不要**把返回泛型类型的函数模板的返回值赋给 **`auto`**。
  - 这意味着 AAA(*总是 auto(Almost Always Auto)*) 原则不适用于 string view。

如果因为这些规则太过复杂或者太困难而不能遵守，那就完全不要使用 `std:: string_view`（除非你知道自己在做什么）。

19.4 字符串视图类型和操作

这一节详细描述字符串视图类型和操作

19.4.1 字符串视图的具体类型

在头文件 `<string_view>` 中，C++ 标准库为 `basic_string_view<>` 提供了很多特化版本：

- 类 `std::string_view` 是预定义的字符为 `char` 类型的特化模板：

```
namespace std {
    using string_view = basic_string_view<char>;
}
```

- 对于使用宽字符集、例如Unicode或者某些亚洲字符集，还定义了另外三个类型：

```
namespace std {
    using u16string_view = basic_string_view<char16_t>;
    using u32string_view = basic_string_view<char32_t>;
    using wstring_view = basic_string_view<wchar_t>;
}
```

上面的小节中，如果使用这三种字符串视图效果也一样。这几种字符串视图类的用法和问题都是一样的，因为它们都有相同的接口。因此，“string view”意味着任何string view类型：`string_view`、`u16string_view`、`u32string_view`、`wstring_view`。这本书中的例子通常使用类型`string_view`，因为欧洲和英美的环境是大部分软件开发的普遍环境。

19.4.2 字符串视图的操作

表字符串视图的操作列出了字符串视图的所有操作。

除了`remove_prefix`和`remove_suffix()`之外，所有字符串视图的操作`std::string`也都有。然而，相应的操作的保证可能有些许不同，`data()`的返回值可能是`nullptr`或者没有空字符终止。

构造

你可以使用很多种方法来创建字符串视图：用默认构造函数创建、用拷贝函数构造创建、从原生字符数组创建（空字符终止或者指明长度）、从`std::string`创建或者从带有`sv`后缀的字面量创建。然而，注意以下几点：

- 默认构造函数创建的字符串视图对象调用`data()`会返回`nullptr`。因此，`operator[]`调用将无效。

```
std::string_view sv;
auto p = sv.data();     // 返回nullptr
std::cout << sv[0];     // ERROR：没有有效的字符
```

- 当使用空字符终止的字节流初始化字符串视图时，最终的大小是不包括`'\0'`在内的字符的数量，另外索引空字符所在的位置是无效的：

```
std::string_view sv{"hello"};
std::cout << sv;        // OK
std::cout << sv.size(); // 5
std::cout << sv.at(5);  // 抛出std::out_of_range异常
std::cout << sv[5];     // 未定义行为
std::cout << sv.data(); // OOPS：恰好sv后边还有个'\0'，所以能直接输出字符指针
```

你可以指定传递的字符数量来把空字符初始化为字符串视图的一部分：

```
std::string_view sv{"hello", 6};    // NOTE：包含'\0'的6个字符
std::cout << sv.size();             // 6
std::cout << sv.at(5);              // OK，打印出'\0'的值
std::cout << sv[5];                 // OK，打印出'\0'的值
std::cout << sv.data();             // OK
```

操作	效果
构造函数	创建或拷贝一个字符串视图
析构函数	销毁一个字符串视图
=	赋予新值
swap()	交换两个字符串视图的值
==、!=、<、<=、>，>=、compare()	比较字符串视图
empty()	返回字符串视图是否为空
size()、length()	返回字符的数量
max_size()	返回可能的最大字符数
[]、at()	访问一个字符
front()、back()	访问第一个或最后一个字符
<<	将值写入输出流
copy()	把内容拷贝或写入到字符数组
data()	返回 nullptr 或常量字符数组（没有空字符终止）
查找函数	查找子字符串或字符
begin()、end()	提供普通迭代器支持
cbegin()、cend()	提供常量迭代器支持
rbegin()、rend()	提供反向迭代器支持
crbegin()、crend()	提供常量反向迭代器支持
substr()	返回子字符串
remove_prefix()	移除开头的若干字符
remove_suffix()	移除结尾的若干字符
hash<>	计算哈希值的函数对象

Table 19.1: 字符串视图的操作

- 为了从一个 string 创建一个字符串视图，有一个为 std::string 定义的隐式转换运算符。再强调一次，string 保证在最后一个字符之后有一个空字符，字符串视图没有这个保证：

```
std::string s = "hello";
std::cout << s.size();      // 5
std::cout << s.at(5);       // 抛出std::out_of_range异常
std::cout << s[5];          // OK，打印出'\0'的值
std::cout << s.data();      // OK

std::string_view sv{s};
std::cout << sv.size();     // 5
std::cout << sv.at(5);      // 抛出std::out_of_range异常
std::cout << sv[5];         // 未定义行为
std::cout << sv.data();     // OOPS：只有当sv后有'\0'时才能正常工作
```

- 因为后缀 sv 定义了字面量运算符，所以可以像下面这样创建一个字符串视图：

```
using namespace std::literals;
auto s = "hello"sv;
```

注意 `std::char_traits` 成员被改为了 `constexpr`，所以你可以在编译期用一个字符串字面量初始化字符串视图：

```
constexpr string_view hello = "Hello World!";
```

结尾的空字符

创建字符串视图的不同方式展示了字符串视图优越的一点，理解这一点是很重要的：一般情况下，字符串视图的值不以空字符结尾，甚至可能是 `nullptr`。因此，你应该总是在访问字符串视图的字符之前检查 `size()`（除非你已经明确知道值）。然而，你可能会遇到两个特殊的场景，这两个场景会让人很迷惑：

1. 你可以确保字符串视图的值以空字符结尾，尽管空字符并不是值的一部分。当你用字符串字面量初始化字符串视图时就会遇到这种情况：

   ```
   std::string_view sv1{"hello"};      // sv1的结尾之后有一个'\0'
   ```

   这里，字符串视图的状态可能让人很困惑。这种状态有明确的定义所以可以将它用作空字符结尾的字符序列。然而，只有当我们明确知道这个字符串视图后有一个不属于自身的空字符时才有明确的定义。
2. 你可以确保'\0'成为字符串视图的一部分。例如：

   ```
   std::string_view sv2{"hello", 6};   // 参数6使'\0'变为值的一部分
   ```

   这里，字符串视图的状态可能让人困惑：打印它的话看起来像是只有5个字符，但它的实际状态是持有6个字符（空字符成为了值的一部分，使它变得更像一个两段的字符串 (视图)(binary string(view))）。

问题在于要想确保字符串视图后有一个空字符的话，这两种方式哪一种更好。我倾向于不要用这两种方式中的任何一种，但截止目前，C++还没有更好的实现方式。看起来我们似乎还需要一个既保证以空字符结尾又不需要拷贝字符的字符串视图类型（就像 `std::string` 一样）。在没有更好的替代的情况下，字符串视图就只能这么用了。事实上，我们已经可以看到很多提案建议C++标准：当返回空字符结尾的字符串时用 `string_view` 作为返回类型（https://wg21.link/P0555r0有一个例子）。

哈希

C++标准库保证值相同的字符串和字符串视图的哈希值相同。

修改字符串视图

这里有几个修改字符串视图的操作：

- 你可以赋予新值或者交换两个字符串视图的值：

  ```
  std::string_view sv1 = "hey";
  std::string_view sv2 = "world";
  sv1.swap(sv2);
  sv2 = sv1;
  ```

- 你可以跳过开头或结尾的字符（即把起始位置移动到第一个字符之后或者把结尾位置移动到最后一个字符之前）。

```
std::string_view sv = "I like my kindergarten";
sv.remove_prefix(2);
sv.remove_suffix(8);
std::cout << sv;    // 打印出：like my kind
```

注意没有对 operator+ 的支持。因此：

```
std::string_view sv1 = "hello";
std::string_view sv2 = "world";
auto s1 = sv1 + sv2;    // ERROR
```

一个操作数必须是 string：

```
auto s2 = std::string(sv1) + sv2;   // OK
```

注意字符串视图没有到 string 的隐式类型转换，因为这个操作会分配内存所以开销很大。因此，只能使用显式的转换：[2]

19.4.3 其他类型对字符串视图的支持

原则上讲，任何需要传递字符串值的地方都可以传递字符串视图，前提是接收者只读取值且不需要空字符结尾。

然而，到目前为止，C++ 标准只为大多数重要的场景添加了支持：

- 使用字符串时可以联合使用字符串视图：
  - 你可以从一个字符串视图创建一个 string（构造函数是 explict 的）。如果字符串视图没有值（data() 返回 nullptr），字符串将被初始化为空。
  - 你可以把字符串视图用作字符串的赋值、扩展、插入、替换、比较或查找操作的参数。
  - 存在从 string 到 string view 的隐式类型转换。
- 你可以把字符串视图传给 std::quoted，它把参数用双引号括起来输出。例如：

```
using namespace std::literals;

auto s = R"(some\value)"sv;     // raw string view
std::cout << std::quoted(s);    // 输出："some\value"
```

- 你可以使用字符串视图初始化、扩展或比较文件系统路径。

其他对字符串视图的支持，例如 C++ 标准库中的正则表达式库的支持，仍然缺失。

19.5 在 API 中使用字符串视图

字符串视图开销很小并且每一个 std::string 都可以用作字符串视图。因此，看起来好像 std::string_view 是更好的用作字符串参数的类型。然而，有一些细节很重要...

首先，只有当函数按照如下约束使用参数时，使用 std::string_view 才有意义：

[2] 理论上讲，我们可以标准化一个操作来把两个字符串视图连接起来并返回新的 string，但直到目前为止还没有实现。

- 它并不需要结尾有空字符。给一个以单个 `const char*` 为参数而没有长度参数的 C 函数传递参数时就不属于这种情况。
- 它不会违反传入参数的生命周期。通常，这意味着接收函数只会在传入值的生命周期结束之前使用它。
- 调用者函数不应该更改底层字符的所有权（例如销毁它、改变它的值或者释放它的内存）。
- 它可以处理参数值为 `nullptr` 的情况。

注意同时有 `std::string` 和 `std::string_view` 重载的函数可能会导致歧义：

```
void foo(const std::string&);
void foo(std::string_view);

foo("hello");   // ERROR：歧义
```

最后，记住上文提到的警告：

- 不要把临时字符串赋给字符串视图。（此处原文是 Do not assign temporary string views to strings. 应是作者笔误）
- 不要返回字符串视图。
- 不要在调用链中使用字符串视图来初始化或重设字符串的值。

带着这些考虑，让我们来看一些使用字符串视图进行改进的例子。

19.5.1 使用字符串视图代替 string

考虑下列代码：

```
// 带前缀输出时间点
void print (const std::string& prefix, const std::chrono::system_clock::time_point& tp)
{
    // 转换为日历时间
    auto rawtime{std::chrono::system_clock::to_time_t(tp)};
    std::string ts{std::ctime(&rawtime)};   // 注意：不是线程安全的

    ts.resize(ts.size()-1); // 跳过末尾的换行符

    std::cout << prefix << ts;
}
```

可以被替换为下列代码：

```
void print (std::string_view prefix, const std::chrono::system_clock::time_point& tp)
{
    auto rawtime{std::chrono::system_clock::to_time_t(tp)};
    std::string_view ts{std::ctime(&rawtime)};  // 注意：不是线程安全的

    ts.remove_suffix(1);    // 跳过末尾的换行符

    std::cout << prefix << ts;
}
```

最先想到也是最简单的改进就是把只读字符串引用 `prefix` 换成字符串视图，只要我们不使用会因为没有值或者没有空终止符而失败的操作就可以。这种情况下，我们只是打印字符串视图的值，这是没问题的。如果字符

串视图没有值（data()返回nullptr）将不会输出任何字符。注意字符串视图是以值传参的，因为拷贝字符串视图的开销很小。

我们也对内部ctime()返回的值使用了字符串视图。然而，我们必须小心保证当我们在字符串视图中使用它时它的值还存在。也就是说，这个值只有在下一次ctime()或者asctime()调用之前有效。因此，在多线程环境下，这个函数将导致问题（使用string时也有一样的问题）。

如果函数返回把前缀和时间点连接起来的字符串，代码可能会像下面这样：

```
std::string toString (std::string_view prefix, const std::chrono::system_clock::time_point& tp)
{
    auto rawtime{std::chrono::system_clock_to_time_t(tp)};
    std::string_view ts{std::ctime(&rawtime)};  // 注意：不是线程安全的

    ts.remove_suffix(1);    // 跳过末尾的换行符
    return std::string{prefix} + ts; // 很不幸没有两个字符串视图的+运算符
}
```

注意我们不能简单地用operator+连接两个字符串视图。我们必须把其中一个转换为std::string（很不幸这个操作会分配不必要的内存）。如果字符串视图没有值（data()返回nullptr），字符串将为空。

另一个使用字符串视图的例子是使用字符串视图和并行算法来排序子字符串：

```
sort(std::execution::par, coll.begin(), coll.end(),
    // 译者注：此处原文是
    // sort(coll.begin(), coll.end(),
    // 应是作者笔误

    [] (const auto& a, const auto& b) {
       return std::string_view{a}.substr(2)
              < std::string_view{b}.substr(2);
    });
```

要比使用string的子字符串快得多：

```
sort(std::execution::par,
    coll.begin(), coll.end(),
    [] (const auto& a, const auto& b) {
        return a.substr(2) < b.substr(2);
    });
```

这是因为string的substr是一个会分配自己内存的新字符串。

19.6 后记

首个引用语义的字符串类由Jeffrey Yasskin在https://wg21.link/n3334中提出（命名为string_ref）。这个类因Jeffrey Yasskin在https://wg21.link/n3921中的提议而被Library Fundamentals TS采纳。

这个类因Beman Dawes和Alisdair Meredith发表于https://wg21.link/p0220r1的提议而和其他组件一起被C++17标准采纳。之后为了更好的集成，Marshall Clow在https://wg21.link/p0254r2和https://wg21.link/p0403r1中、Nicolai Josuttis在https://wg21.link/p0392r0中添加了一些修改。

Antony Polukhin 在发表于https://wg21.link/p0426r1提案中添加了对 `constexpr` 的支持。

Daniel Krügler 发表的https://wg21.link/lwg2946中包含了一些修复（作为C++17的缺陷被C++20采纳）。

Chapter 20

文件系统库

直到C++17，Boost.Filesystem库终于被C++标准采纳。在这个过程中，这个库用新的语言特性进行了很多调整、改进了和其他库的一致性、进行了精简、还扩展了很多确实的功能（例如计算两个文件系统路径之间的相对路径）。

20.1 基本的示例

让我们以一些基本的示例开始。

20.1.1 打印文件系统路径类的属性

下面的程序允许我们传递一个字符串作为文件系统路径，然后根据给定路径的文件类型打印出一些信息：

filesystem/checkpath1.cpp

```
#include <iostream>
#include <filesystem>
#include <cstdlib> // for EXIT_FAILURE

int main(int argc, char* argv[])
{
    if (argc < 2) {
        std::cout << "Usage: " << argv[0] << " <path> \n";
        return EXIT_FAILURE;
    }

    std::filesystem::path p{argv[1]}; // p代表一个文件系统路径（可能不存在）
    if (is_regular_file(p)) {   // 路径p是普通文件吗？
        std::cout << p << " exists with " << file_size(p) << " bytes\n";
    }
    else if (is_directory(p)) { // 路径p是目录吗？
        std::cout << p << " is a directory containing:\n";
        for (const auto& e : std::filesystem::directory_iterator{p}) {
```

```
            std::cout << "   " << e.path() << '\n';
        }
    }
    else if (exists(p)) {       // 路径p存在吗？
        std::cout << p << " is a special file\n";
    }
    else {
        std::cout << "path " << p << " does not exist\n";
    }
}
```

我们首先把传入的命令行参数转换为了一个文件系统路径：

```
std::filesystem::path p{argv[1]};   // p代表一个文件系统路径（有可能不存在）
```

然后，我们进行了下列检查：

- 如果改路径代表一个普通文件，我们打印出它的大小：

  ```
  if (is_regular_file(p)) {   // 路径p是普通文件？
      std::cout << p << " exists with " << file_size(p) << " bytes\n";
  }
  ```

  像下面这样调用程序：

  ```
  checkpath checkpath.cpp
  ```

  将会有如下输出：

  ```
  "checkpath.cpp" exists with 907 bytes
  ```

  注意输出路径时会自动把路径名用双引号括起来输出（把路径用双引号括起来、反斜杠用另一个反斜杠转义，对 Windows 路径来说是一个问题）。

- 如果路径是一个目录，我们遍历这个目录中的所有文件并打印出这些文件的路径：

  ```
  if (is_directory(p)) {      // 路径p是目录？
      std::cout << p << " is a directory containing:\n";
      for (auto& e : std::filesystem::directory_iterator{p}) {
          std::cout << "   " << e.path() << '\n';
      }
  }
  ```

  这里，我们使用了 `directory_iterator`，它提供了 `begin()` 和 `end()`，所以我们可以使用范围 for 循环来遍历 `directory_entry` 元素。在这里，我们使用了 `directory_entry` 的成员函数 `path()`，返回该目录项的文件系统路径。像下面这样调用程序：

  ```
  checkpath .
  ```

  输出将是：

  ```
  "." is a directory containing:
    "./checkpath.cpp"
    "./checkpath.exe"
    ...
  ```

- 最后，我们检查传入文件系统路径是否不存在：

```
if (exists(p)) {        // 路径p存在吗？
    ...
}
```

注意根据参数依赖查找 *(argument dependent lookup)(ADL)*，你不需要使用完全限定的名称来调用 `is_regular_file()`、`file_size()`、`is_directory()`、`exists()` 等函数。它们都属于命名空间 `std::filesystem`，但是因为它们的参数也属于这个命名空间，所以调用它们时会自动在这个命名空间中进行查找。

在 Windows 下处理路径

（译者注：可能是水平有限，完全看不懂作者在这一小节的逻辑，所以只能按照自己的理解胡乱翻译，如有错误请见谅。）

默认情况下，输出路径时用双引号括起来并用反斜杠转义反斜杠在 Windows 下会导致一个问题。在 Windows 下以如下方式调用程序：

```
checkpath C:\
```

将会有如下输出：

```
"C:" is a directory containing:
  ...
  "C:Users"
  "C:Windows"
```

用双引号括起来输出路径可以确保输出的文件名可以被直接复制粘贴到其他程序里，并且经过转义之后还会恢复为原本的文件名。然而，终端通常不接受这样的路径。

因此，一个在 Windows 下的可移植版本应该使用成员函数 `string()`，这样可以在向标准输出写入路径时避免输出双引号：

filesystem/checkpath2.cpp

```cpp
#include <iostream>
#include <filesystem>
#include <cstdlib>  // for EXIT_FAILURE

int main(int argc, char* argv[])
{
    if (argc < 2) {
        std::cout << "Usage: " << argv[0] << " <path> \n";
        return EXIT_FAILURE;
    }
    std::filesystem::path p{argv[1]};   // p代表一个文件系统路径（可能不存在）
    if (is_regular_file(p)) {   // 路径p是普通文件吗？
        std::cout << '"' << p.string() << "\" existes with " << file_size(p) << " bytes\n";
    }
    else if (is_directory(p)) { // 路径p是目录吗？
        std::cout << '"' << p.string() << "\" is a directory containing:\n";
```

```
        for (const auto& e : std::filesystem::directory_iterator{p}) {
            std::cout << "  \"" << e.path().string() << "\"\n";
        }
    }
    else if (exists(p)) {       // 路径p存在吗？
        std::cout << '"' << p.string() << "\" is a special file\n";
    }
    else {
        std::cout << "path \"" << p.string() << "\" does not exist\n";
    }
}
```

现在，在 Windows 上以下面方式调用程序：

```
checkpath C:\
```

将会有如下输出：

```
"C:\" is a directory containing:
  ...
  "C:\Users"
  "C:\Windows"
```

通用字符串格式还提供了其他转换来把 string 转换为本地编码。

20.1.2 用 switch 语句处理不同的文件系统类型

我们可以像下面这样修改并改进上面的例子：

filesystem/checkpath3.cpp

```
#include <iostream>
#include <filesystem>
#include <cstdlib>  // for EXIT_FAILURE

int main(int argc, char* argv[])
{
    if (argc < 2) {
        std::cout << "Usage: " << argv[0] << " <path> \n";
        return EXIT_FAILURE;
    }
    namespace fs = std::filesystem;

    switch (fs::path p{argv[1]}; status(p).type()) {
        case fs::file_type::not_found:
            std::cout << "path \"" << p.string() << "\" does not exist\n";
            break;
        case fs::file_type::regular:
            std::cout << '"' << p.string() << "\" exists with " << file_size(p) << " bytes\n";
            break;
```

```
        case fs::file_type::directory:
            std::cout << '"' << p.string() << "\" is a directory containing:\n";
            for (const auto& e : std::filesystem::directory_iterator{p}) {
                std::cout << "  " << e.path().string() << '\n';
            }
            break;
        default:
            std::cout << '"' << p.string() << "\" is a special file\n";
            break;
    }
}
```

命名空间 fs

首先，我们做了一个非常普遍的操作：我们定义了 fs 作为命名空间 std::filesystem 的缩写：

```
namespace fs = std::filesystem;
```

使用新初始化的命名空间的一个例子是下面 switch 语句中的路径 p：

```
fs::path p{argv[1]};
```

这里的 switch 语句使用了新的带初始化的 switch 语句特性，初始化路径的同时把路径的类型作为分支条件：

```
switch (fs::path p{argv[1]}; status(p).type()) {
    ...
}
```

表达式 status(p).type() 首先创建了一个 file_status 对象，然后该对象的 type() 方法返回了一个 file_type 类型的值。通过这种方式我们可以直接处理不同的类型，而不需要使用 is_regular_file()、is_directory() 等函数构成的 if-else 链。我们通过多个步骤（先调用 status() 再调用 type() 才得到了最后的类型，因此我们不需要为不感兴趣的其他信息付出多余的系统调用开销。

注意可能已经有特定实现的 file_type 存在。例如，Windows 就提供了特殊的文件类型 junction。然而，使用了它的代码是不可移植的。

20.1.3 创建不同类型的文件

在介绍了文件系统的只读操作之后，让我们给出首个进行修改的例子。下面的程序在一个子目录 tmp 中创建了不同类型的文件：

filesystem/createfiles.cpp

```
#include <iostream>
#include <fstream>
#include <filesystem>
#include <cstdlib> // for std::exit()和EXIT_FAILURE

int main()
{
```

```
    namespace fs = std::filesystem;
    try {
        // 创建目录tmp/test/（如果不存在的话）
        fs::path testDir{"tmp/test"};
        create_directories(testDir);

        // 创建数据文件tmp/test/data.txt：
        auto testFile = testDir / "data.txt";
        std::ofstream  dataFile{testFile};
        if (!dataFile) {
            std::cerr << "OOPS, can't open\"" << testFile.string() << "\"\n";
            std::exit(EXIT_FAILURE);    // 以失败方式结束程序
        }
        dataFile << "The answer is 42\n";

        // 创建符号链接tmp/slink/，指向tmp/test/：
        create_directory_symlink("test", testDir.parent_path() / "slink");
    }
    catch (const fs::filesystem_error& e) {
        std::cerr << "EXCEPTION: " << e.what() << '\n';
        std::cerr << "    path1: \"" << e.path1().string() << "\"\n";
    }

    // 递归列出所有文件（同时遍历符号链接）
    std::cout << fs::current_path().string() << ":\n";
    auto iterOpts{fs::directory_options::follow_directory_symlink};
    for (const auto& e : fs::recursive_directory_iterator(".", iterOpts)) {
        std::cout << "  " << e.path().lexically_normal().string() << '\n';
    }
}
```

让我们一步步来分析这段程序。

命名空间 `fs`

首先，我们又一次定义了 `fs` 作为命名空间 `std::filesystem` 的缩写：

```
namespace fs = std::filesystem;
```

之后我们使用这个命名空间为临时文件创建了一个基本的子目录：

```
fs::path testDir{"tmp/test"};
```

创建目录

当我们尝试创建子目录时：

```
create_directories(testDir);
```

通过使用 create_directories() 我们可以递归创建整个路径中所有缺少的目录（还有一个 create_directory() 只在已存在的目录中创建目录）。

当目标目录已经存在时这个调用并不会返回错误。然而，其他的问题会导致错误病抛出一个相应的异常。

如果 testDis 已经存在，create_directories() 会返回 false。因此，你可以这么写：

```
if (!create_directories(testDir)) {
    std::cout << "\"" << testDir.string() << "\" already exists\n";
}
```

创建普通文件

之后我们用一些内容创建了一个新文件 tmp/test/data.txt：

```
auto testFile = testDir / "data.txt";
std::ofstream dataFile{testFile};
if (!dataFile) {
    std::cerr << "OOPS, can't open \"" << testFile.string() << "\"\n";
    std::exit(EXIT_FAILURE);  // 失败退出程序
}
dataFile << "The answer is 42\n";
```

这里，我们使用了运算符/来扩展路径，然后传递给文件流的构造函数。如你所见，普通文件的创建可以使用现有的I/O流库来实现。然而，I/O流的构造函数多了一个以文件系统路径为参数的版本（一些函数例如open()也添加了这种重载版本）。

注意你仍然应该总是检查创建/打开文件的操作是否成功了。这里有很多种可能发生的错误（见下文）。

创建符号链接

接下来的语句尝试尝试创建符号链接 tmp/slink 指向目录 tmp/test：

```
create_directory_symlink("test", testDir.parent_path() / "slink");
```

注意第一个参数的路径是以即将创建的符号链接所在的目录为起点的相对路径。因此，你必须传递"test"而不是"tmp/test"来高效的创建链接 tmp/slink 指向 tmp/test。如果你调用：

```
std::filesystem::create_directory_symlink("tmp/test", "tmp/slink");
```

你将会高效的创建符号链接 tmp/slink，然而它会指向 tmp/tmp/test。

注意通常情况下，也可以调用 create_symlink() 代替 create_ directory_symlink() 来创建目录的符号链接。然而，一些操作系统可能对目录的符号链接有特殊处理或者当知道要创建的符号链接指向目录时会有优化，因此，当你想创建指向目录的符号链接时你应该使用 create_directory_symlink()。

最后，注意这个调用在Windows上可能会失败并导致错误处理，因为创建符号链接可能需要管理员权限。

递归遍历目录

最后，我们递归的遍历了当前目录：

```
auto iterOpts = fs::directory_options::follow_directory_symlink;
for (auto& e : fs::recursive_directory_iterator(".", iterOpts)) {
    std::cout << "  " << e.path().lexically_normal().string() << '\n';
}
```

注意我们使用了一个递归目录迭代器并传递了选项`follow_directory_ symlink`来遍历符号链接。因此，我们在 POSIX 兼容系统上可能会得到类似于如下输出：

```
/home/nico:
  ...
  tmp
  tmp/slink
  tmp/slink/data.txt
  tmp/test
  tmp/test/data.txt
  ...
```

在 Windows 系统上有类似如下输出：

```
C:\Users\nico:
  ...
  tmp
  tmp\slink
  tmp\slink\data.txt
  tmp\test
  tmp\test\data.txt
  ...
```

注意在我们打印目录项之前调用了`lexically_normal()`。如果略过这一步，目录项的路径可能会包含一个前缀，这个前缀是创建目录迭代器时传递的实参。因此，在循环内直接打印路径：

```
auto iterOpts = fs::directory_options::follow_directory_symlink;
for (auto& e : fs::recursive_directory_iterator(".", iterOpts)) {
    std::cout << "  " << e.path() << '\n';
}
```

将会在 POSIX 兼容系统上有如下输出：

```
all files:
  ...
  "./testdir"
  "./testdir/data.txt"
  "./tmp"
  "./tmp/test"
  "./tmp/test/data.txt"
```

在 Windows 上，输出将是：

```
all files:
  ...
  ".testdir"
  ".testdirdata.txt"
```

```
".tmp"
".tmptest"
".tmptestdata.txt"
```

通过调用 `lexically_normal()` 我们可以得到正规化的路径，它移除了前导的代表当前路径的点。还有，如上文所述，通过调用 `string()` 我们避免了输出路径时用双引号括起来。这里没有调用 `string()` 的输出结果在 POSIX 兼容的系统上看起来 OK（只是路径两端有双引号），但在 Windows 上的结果看起来就很奇怪（因为每一个反斜杠都需要反斜杠转义）。

错误处理

文件系统往往是麻烦的根源。你可能因为在文件名中使用了无效的字符而导致操作失败，或者当你正在访问文件系统时它已经被其他程序修改了。因此，根据平台和权限的不同，这个程序中可能会有很多问题。

对于那些没有被返回值覆盖的情况（例如当目录已经存在时），我们捕获了相应的异常并打印了一般的信息和第一个路径：

```
try {
    ...
}
catch (const fs::filesystem_error& e) {
    std::cerr << "EXCEPTION: " << e.what() << '\n';
    std::cerr << "    path1: \"" << e.path1().string() << "\"\n";
}
```

例如，如果我们不能创建目录，将会打印出类似于如下消息：

```
EXCEPTION: filesystem error: cannot create directory: [tmp/test]
    path1: "tmp/test"
```

如果我们不能创建符号链接可能是因为它已经存在了，或者我们可能需要特殊权限，这些情况下你可能会得到如下消息：

```
EXCEPTION: create_directory_symlink: Can't create a file when it
                                     already exists: "tmp\test\
                                     data.txt", "testdir"
    path1: "tmp\test\data.txt"
```

或者：

```
EXCEPTION: create_directory_symlink: A requied privilege is not
                                     held by the client.: "test
                                     ", "tmp\slink"
    path1: "test"
```

在每一种情况下，都要注意在多用户/多进程操作系统中情况可能会在任何时候改变，这意味着你刚刚创建的目录甚至可能已经被删除、重命名、或已经被同名的文件覆盖。因此，很显然不能只根据当前的情况就保证一个预期操作一定是有效的。最好的方式就是尝试做想做的操作（例如，创建目录、打开文件）并处理抛出的异常和错误，或者验证预期的行为。

然而，有些时候文件系统操作能正常执行但不是按你预想的结果。例如，如果你想在指定目录中创建一个文件并且已经有了一个和目录同名的指向另一个目录的符号链接，那么这个文件可能在一个预料之外的地方创建或者覆写。（译者注：比如你想在当前目录下递归创建”a/b”，但已经有了一个”a” 是一个指向”c” 目录的符号链接，这时你实际会在目录”c” 下创建”b”）这种情况是有可能的（用户完全有可能会创建目录的符号链接），但是如果你想检测这种情况，在创建文件之前你需要检查文件是否存在（这可能比你一开始想的要复杂很多）。

再强调一次：文件系统并不保证进行处理之前的检查的结果直到你进行处理时仍然有效。

20.1.4 使用并行算法处理文件系统

参见dirsize.cpp 查看另一个使用并行算法计算目录树中所有文件大小之和的例子。

20.2 原则和术语

在讨论文件系统库的细节之前，我们不得不继续介绍一些设计原则和术语。这是必须的，因为标准库要覆盖不同的操作系统并把系统提供的接口映射为公共的 API。

20.2.1 通用的可移植的分隔符

C++ 标准库不仅标准化了所有操作系统的文件系统中公共的部分，在很多情况下，C++ 标准还尽可能的遵循 POSIX 标准的要求来实现。对于一些操作，只要是合理的就应该能正确执行，如果操作是不合理的，实现应该报错。这些错误可能是：

- 特殊的字符不能被用作文件名
- 文件系统的某些元素不支持创建（例如，符号链接）

不同文件系统的差异也应该纳入考虑：

- 大小写敏感：`"hello.txt"` 和`"Hello.txt"` 和`"hello.TXT"` 可能指向同一个文件（Windows 上）也可能指向三个不同的文件（POSIX 兼容系统）。
- 绝对路径和相对路径：
- 在某些系统上，”/bin” 是一个绝对路径（POSIX 兼容系统），然而在某些系统上不是（Windows）。

20.2.2 命名空间

文件系统库在 `std` 里有自己的子命名空间 `filesystem`。一个很常见的操作是定义缩写 `fs`：

```
namespace fs = std::filesystem;
```

这允许我们使用 `fs::current_path()` 代替 `std::filesystem:: current_path()`。

这一章的示例代码中将经常使用 `fs` 作为缩写。

注意你应该总是使用完全限定的函数调用，尽管不指明命名空间时通过 *参数依赖查找 (argument dependent lookup)(ADL)* 也能够工作。但如果不用命名空间限定有时可能导致意外的行为。

20.2.3 文件系统路径

文件系统库的一个关键元素是 path。它代表文件系统中某一个文件的名字。它由可选的根名称、可选的根目录、和一些以目录分隔符分隔的文件名组成。路径可以是相对的（此时文件的位置依赖于当前的工作目录）或者是绝对的。

路径可能有不同的格式：

- 通过格式，这是可移植的
- 本地格式，这是底层文件系统特定的

在 POSIX 兼容系统上通用格式和本地格式没有什么区别。在 Windows 上，通用格式 /tmp/test.txt 也是有效的本地格式，另外 \tmp\test.txt 也是有效的（/tmp/test.txt 和 \tmp\test.txt 是同一个路径的两种本地版本）。在 OPenVMS 上，相应的本地格式将是 [tmp]test.txt。

也有一些特殊的文件名：

- ”.” 代表当前 mulu
- ”..” 代表父目录

通用的路径格式如下：

```
[rootname] [rootdir] [relativepath]
```

这里：

- 可选的根名称是实现特定的（例如，在 POSIX 系统上可以是 //host，而在 Windows 上可以是 C:）
- 可选的根目录是一个目录分隔符
- 相对路径是若干目录分隔符分隔的文件名

目录分隔符由一个或多个 '/' 组成或者是实现特定的。

可移植的通用路径的例子有：

```
//host1/bin/hello.txt
.
tmp/
/a/b//.../c
```

注意在 POSIX 系统上最后一个路径和 /a/c 指向同一个位置，并且都是绝对路径。而在 Windows 上则是相对路径（因为没有指定驱动器/分区（盘））。

另一方面，C:/bin 在 Windows 上是绝对路径（在”C” 盘上的根目录 "bin"），但在 POSIX 系统上是一个相对路径（目录”C:” 下的子目录”bin”）。

在 Windows 系统上，反斜杠是实现特定的目录分隔符，因此上面的路径也可以使用反斜杠作为目录分隔符：

```
host1\bin\hello.txt
.
tmp\
\a\b\..\c
```

文件系统库提供了在本地格式和通用格式之间转换的函数。

一个 path 可能为空，这意味着没有定义路径。这种状态的含义不需要和 "." 一样。它的含义依赖于上下文。

20.2.4 正规化

路径可以进行正规化，在正规化的路径中：

- 文件名由单个推荐的目录分隔符分隔。
- 除非整个路径就是"."（代表当前目录），否则路径中不会使用"."。
- 路径中除了开头以外的地方不会包含".."（不能在路径中上下徘徊）。
- 除非整个路径就是"."或者".."，否则当路径结尾的文件名是目录时要在最后加上目录分隔符。

注意正规化之后以目录分隔符结尾的路径和不以目录分隔符结尾的路径是不同的。这是因为在某些操作系统中，当它们知道目标路径是一个目录时行为可能会发生改变（例如，有尾部的分隔符时符号链接将被解析）。

表路径正规化的效果列举了一些在 POSIX 系统和 Windows 系统上对路径进行正规化的例子。注意再重复一次，在 POSIX 系统上，`C:bar` 和 `C:` 只是把冒号作为文件名的一部分的单个文件名，并没有特殊的含义，而在 Windows 上，它们指定了一个分区。

路径	POSIX 正规化	Windows 正规化
`foo/.///bar/../`	`foo/`	`foo\`
`//host/../foo.txt`	`//host/foo.txt`	`\\host\foo.txt`
`./f/../.f/`	`.f/`	`.f\`
`C:bar/../`	`.`	`C:`
`C:/bar/..`	`C:/`	`C:\`
`C:\bar\..`	`C:\bar\..`	`C:\`
`/./../data.txt`	`/data.txt`	`\data.txt`
`./././`	`.`	`.`

Table 20.1: 路径正规化的效果

文件系统库同时提供了词法正规化（不访问文件系统）和依赖文件系统的正规化两种方式的相关函数。

20.2.5 成员函数 VS 独立函数

文件系统库提供了一些函数，有些是成员函数有些是独立函数。这么做的目的是：

- **成员函数开销较小**。这是因为它们是纯词法的操作，并不会访问实际的文件系统，这意味着它们不需要进行操作系统调用。例如：

```
mypath.is_absolute()        // 检查路径是否是绝对的
```

- **独立函数开销较大**。因为它们通常会访问实际的文件系统，这意味着需要进行操作系统调用。例如：

```
equivalent(path1, path2);   // 如果两个路径指向同一个文件则返回true
```

有时，文件系统库甚至为同一个功能既提供根据词法的版本又提供访问实际文件系统的版本：

```
std::filesystem::path fromP, toP;
...
toP.lexically_relative(fromP);  // 返回从fromP到toP的词法路径
relative(toP, fromP);           // 返回从fromP到toP的实际路径
```

得益于参数依赖查找*(ADL)*，很多情况下当调用独立函数时你不需要指明完整命名空间 `std::filesystem`，只要参数是文件系统库里定义的类型。只有当用其它类型隐式转换为参数时你才需要给出完全限定的函数名。例如：

```
create_directory(std::filesystem::path{"tmpdir"});  // OK
remove(std::filesystem::path{"tmpdir"});            // OK
std::filesystem::create_directory("tmpdir");        // OK
std::filesystem::remove("tmpdir");                  // OK
create_directory("tmpdir");                         // ERROR
```

最后一个调用将会编译失败，因为我们并没有传递文件系统命名空间里的类型作为参数，因此也不会在该命名空间里查找符号 `create_directory`。

然而，这里有一个著名的陷阱：

```
remove("tmpdir");   // OOPS：调用C函数remove()
```

根据你包含的头文件，这个调用可能会找到C函数 `remove()`，它的行为有一些不同：它也会删除指定的文件但不会删除空目录。

因此，强烈推荐使用完全限定的文件系统库里的函数名。例如：

```
namespace fs = std::filesystem;
...
fs::remove("tmpdir");   // OK：调用C++文件系统库函数remove()
```

20.2.6 错误处理

如上文所述，文件系统是错误的根据。你必须考虑相应的文件是否存在、文件的操作是否被允许、该操作是否会违背资源限制。另外，当程序运行时其它进程可能创建、修改、或者移除了某些文件，这意味着事先检查并不能保证没有错误。

问题在于从理论上讲，你不能提前保证下一次文件系统操作能够成功。任何事先检查的结果都可能在你实际进行处理时失效。因此，最好的方法是在进行一个或多个文件系统操作时处理好相应的异常或者错误。

注意，当读写普通文件时，默认情况下I/O流并不会抛出异常或错误。当操作遇到错误时它只会什么也不做。因此，建议至少检查一下文件是否被成功打开。

因为并不是所有情况下都适合抛出异常（例如当一个文件系统调用失败时你想直接处理），所以文件系统库使用了混合的异常处理方式：

- 默认情况下，文件系统错误会作为异常处理。
- 然而，如果你想的话可以在本地处理具体的某一个错误。

因此，文件系统库通常为每个操作提供两个重载版本：

1. 默认情况下（没有额外的错误处理参数），出现错误时抛出 `filesystem_error` 异常。
2. 传递额外的输出参数时，可以得到一个错误码或错误信息，而不是异常。

注意在第二种情况下，你可能会得到一个特殊的返回值来表示特定的错误。

使用 `filesystem_error` 异常

例如，你可以尝试像下面这样创建一个目录：

```
if (!create_directory(p)) { // 发生错误时抛出异常（除非错误是该路径已经存在）
    std::cout << p << " already exists\n";  // 该路径已经存在
}
```

这里没有传递错误码参数，因此错误时通常会抛出异常。然而，注意当目录已存在时这种特殊情况是直接返回 `false`。因此，只有当其他错误例如没有权限创建目录、路径 `p` 无效、违反了文件系统限制（例如路径长度超过上限）时才会抛出异常。

可以直接或间接的用 `try-catch` 包含这段代码，然后处理 `std::filesystem::filesystem_error` 异常：

```
try {
    ...
    if (!create_directory(p)) { // 错误时抛出异常（除非错误是该路径已经存在）
        std::cout << p << " already exists\n"; // 该路径已经存在
    }
    ...
}
catch (const std::filesystem::filesystem_error& e) { // 派生自std::exception
    std::cout << "EXCEPTION: " << e.what() << '\n';
    std::cout << "     path: " << e.path1() << '\n';
}
```

如你所见，文件系统异常提供了标准异常的 `what()` 函数 API 来返回一个实现特定的错误信息。然而，API 还提供了 `path1()` 来获取错误相关的第一个路径，和 `path2()` 来获取相关的第二个路径。

使用 `error_code` 参数

另一种创建目录的方式如下所示：

```
std::error_code ec;
create_directory(p, ec);    // 发生错误时设置错误码
if (ec) {                   // 如果设置了错误码（因为发生了错误）
    std::cout << "ERROR: " << ec.message() << "\n";
}
```

之后，我们还可以检查特定的错误码：

```
if (ec == std::errc::read_only_file_system) {   // 如果设置了特定的错误码
    std::cout << "ERROR: " << p << " is read-only\n";
}
```

注意这种情况下，我们仍然必须检查 `create_directory()` 的返回值：

```
std::error_code ec;
if (!create_directory(p, ec)) { // 发生错误时设置错误码
    // 发生任何错误时
    std::cout << "can't create directory " << p << "\n";
    std::cout << "error: " << ec.message() << "\n";
}
```

然而，并不是所有的文件系统操作都提供这种能力（因为它们在正常情况下会返回一些值）。

类型 `error_code` 由 C++11 引入，它包含了一系列可移植的错误条件例如 `std::errc::read_only_filesystem`。在 POSIX 兼容的系统上这些被映射为 `errno` 的值。

20.2.7 文件类型

不同的操作系统支持不同的文件类型。标准文件系统库中也考虑到了这一点，它定义了一个枚举类型 `file_type`，标准中定义了如下的值：

```
namespace std::filesystem {
    enum class file_type {
        regular, directory, symlink,
        block, character, fifo, socket,
        ...
        none, not_found, unknown,
    };
}
```

表file_type 的值列出了这些值的含义。

值	含义
`regular`	普通文件
`directory`	目录文件
`symlink`	符号链接文件
`character`	字符特殊文件
`block`	块特殊文件
`fifo`	FIFO 或者管道文件
`socket`	套接字文件
…	附加的实现定义的文件类型
`none`	文件的类型未知
`unknown`	文件存在但推断不出类型
`not_found`	虚拟的表示文件不存在的类型

Table 20.2: 文件系统类型的值

操作系统平台可能会提供附加的文件类型值。然而，使用它们是不可移植的。例如，Windows 就提供了文件类型值 `junction`，它被用于 NTFS 文件系统中的 *NTFS junctions*（也被称为软链接）。它们被用作链接来访问同一台电脑上不同的子卷 (盘)。

除了普通文件和目录之外，最常见的类型是符号链接，它是一种指向另一个位置的文件。指向的为之可能有一个文件也可能没有。注意有些操作系统和/或文件系统（例如 FAT 文件系统）完全不支持符号链接。有些操作系统只支持普通文件的符号链接。注意在 Windows 上需要特殊的权限才能创建符号链接，可以用 `mklink` 命令创建。

字符特殊文件、块特殊文件、FIFO、套接字都来自于 UNIX 文件系统。目前，Visual C++ 并没有使用这四种类型中的任何一个。[1]

如你所见，有一些特殊的值来表示文件不存在或者类型未知或者无法探测出类型。

在这一章的剩余部分我将使用两种广义的类型来代表相应的若干文件类型：

- *其他文件*：除了普通文件、目录、符号链接之外的所有类型的文件。库函数 `is_other()` 和这个术语相匹配。
- *特殊文件*：下列类型的文件：字符特殊文件、块特殊文件、FIFO、套接字。

另外，*特殊*文件类型加上实现定义的文件类型就构成了*其他*文件类型。

20.3 路径操作

有很多处理文件系统的操作。这些操作的关键是一个类型 `std::filesystem::path`，它表示一个可能存在也可能不存在的文件的绝对或相对的路径。

你可以创建路径、检查路径、修改路径、比较路径。因为这些操作一般都不会访问实际的文件系统（例如不会检查文件是否存在，也不会解析符号链接），所以它们的开销很小。因此，它们通常被定义为成员函数（如果这些操作既不是构造函数也不是运算符的话）。

20.3.1 创建路径

表创建路径列出了创建新的路径对象的方法。

调用	效果
`path{charseq}`	用一个字符序列初始化路径
`path{beg, end}`	用一个范围初始化路径
`u8path{u8string}`	用一个 UTF-8 字符串初始化路径
`current_path()`	返回当前工作目录的路径
`temp_directory_path()`	返回临时文件的路径

Table 20.3: 创建路径

第一个构造函数以字符序列为参数，这里的字符序列代表一系列有效的方式：

- 一个 `string`
- 一个 `string_view`
- 一个以空字符结尾的字符数组
- 一个以空字符结尾的字符输入迭代器（指针）

注意 `current_path()` 和 `temp_directory_path()` 都是开销较大的操作，因为它们依赖于系统调用。如果给 `current_path()` 传递一个参数，它也可以用来 修改当前工作目录。

通过 `u8path()` 你可以使用 UTF-8 字符串创建可移植的路径。例如：

```
// 将路径p初始化为"Köln"（Cologne的德语名）
std::filesystem::path p{std::filesystem::u8path(u8"K\u00F6ln")};
```

[1] Windows 管道的行为有些不同，也不被识别为 `fifo`。

```
...

// 用UTF-8字符串创建目录
std::string utf8String = readUTF8String(...);
create_directory(std::filesystem::u8path(utf8String));
```

20.3.2 检查路径

表检查路径列出了检查路径p时可以调用的函数。注意这些操作都不会访问底层的操作系统，因此都是path类的成员函数。

调用	效果
p.empty()	返回路径是否为空
p.is_absolute()	返回路径是否是绝对的
p.is_relative()	返回路径是否是相对的
p.has_filename()	返回路径是否既不是目录也不是根名称
p.has_stem()	和has_filename()一样
p.has_extension()	返回路径是否有扩展名
p.has_root_name()	返回路径是否包含根名称
p.has_root_directory()	返回路径是否包含根目录
p.has_root_path()	返回路径是否包含根名称或根目录
p.has_parent_path()	返回路径是否包含父路径
p.has_relative_path()	返回路径是否不止包含根元素
p.filename()	返回文件名（或者空路径）
p.stem()	返回没有扩展名的文件名（或者空路径）
p.extension()	返回扩展名（或者空路径）
p.root_name()	返回根名称（或者空路径）
p.root_directory()	返回根目录（或者空路径）
p.root_path()	返回根元素（或者空路径）
p.parent_path()	返回父路径（或者空路径）
p.relative_path()	返回不带根元素的路径（或者空路径）
p.begin()	返回路径元素的起点
p.end()	返回路径元素的终点

Table 20.4: 检查路径

每一个路径要么是绝对的要么是相对的。如果没有根目录那么路径就是相对的（相对路径也可能包含根名称；例如，C:hello.txt就是Windows下的一个相对路径）。

has_...()函数等价于检查相应的没有has_前缀的函数返回值是否为空路径。

注意下面几点：

- 如果路径含有根元素或者目录分隔符那么就包含父路径。如果路径只由根元素组成（也就是说相对路径为空），`parent_path()` 的返回值就是整个路径。也就是说，路径`"/"` 的父路径还是`"/"`。只有纯文件名的路径例如 `"hello.txt"` 的父路径是空。
- 如果一个路径包含文件名那么一定包含 stem（文件名中不带扩展名的部分）。[2]
- 空路径是相对路径（除了 `is_empty()` 和 `is_relative()` 之外的操作都返回 `false` 或者空路径）。

这些操作的结果可能会依赖于操作系统。例如，路径 `C:/hello.txt`

- 在 Unix 系统上
  - 是相对路径
  - 没有根元素（既没有根名称也没有根目录），因为`C:` 只是一个文件名
  - 有父路径`C:`
  - 有相对路径`C:/hello.txt`
- 在 Windows 系统上
  - 是绝对的
  - 有根名称`C:` 和根目录`/`
  - 沒有父路径
  - 有相对路径`hello.txt`

遍历路径

你可以遍历一个路径，这将会返回路径的所有元素：根名称（如果有的话）、根目录（如果有的话）、所有的文件名。如果路径以目录分隔符结尾，最后的元素将是空文件名。[3]

路径迭代器是双向迭代器，所以你可以递减它。迭代器的值的类型是`path`。然而，两个在同一个路径上路径上迭代的迭代器可能不指向同一个`path`对象，即使它们迭代到了相同的路径元素。

例如，考虑：

```
void printPath(const std::filesystem::path& p)
{
    std::cout << "path elements of \"" << p.string() << "\":\n";
    for (std::filesystem::path elem : p) {
        std::cout << "  \"" << elem.string() << '"';
    }
    std::cout << '\n';
}
```

和如下代码效果相同：

```
void printPath(const std::filesystem::path& p)
{
    std::cout << "path elements of \"" << p.string() << "\":\n";
    for (auto pos = p.begin(); pos != p.end(); ++pos) {
        std::filesystem::path elem = *pos;
```

[2]从C++17才开始这样，因为以前文件名可以只包含扩展名。

[3]在C++17之前，文件系统库使用 . 来表示结尾的目录分隔符。更改这个行为是为了区分以路径分隔符结尾的路径和以路径分隔符后还有一个点结尾的路径。

```
            std::cout << "  \"" << elem.string() << '"';
        }
        std::cout << '\n';
    }
```

如果像下面这样调用这个函数：

```
printPath("../sub/file.txt");
printPath("/usr/tmp/test/dir/");
printPath("C:\\usr\\tmp\\test\\dir\\");
```

在 POSIX 兼容系统上的输出将会是：

```
path elements of "../sub/file.txt":
  ".."  "sub"  "file.txt"
path elements of "/usr/tmp/test/dir/":
  "/"  "usr"  "tmp"  "test"  "dir"  ""
path elements of "C:\\usr\\tmp\\test\\dir\\":
  "C:\\usr\\tmp\\test\\dir\\"
```

注意最后一个路径只是一个文件名，因为在 POSIX 兼容系统上 `C:` 不是有效的根名称，反斜杠也不是有效的目录分隔符。

在 Windows 上的输出将是：

```
path elements of "../sub/file.txt":
  ".."  "sub"  "file.txt"
path elements of "/usr/tmp/test/dir/":
  "/"  "usr"  "tmp"  "test"  "dir"  ""
path elements of "C:\usr\tmp\test\dir\":
  "C:"  "\"  "usr"  "tmp"  "test"  "dir"  ""
```

为了检查路径 `p` 是否以目录分隔符结尾，你可以这么写：

```
if (!p.empty() && (--p.end())->empty()) {
    std::cout << p << " has a trailing separator\n";
}
```

20.3.3 路径 I/O 和转换

表路径 I/O 和转换列出了路径的读写操作和转换操作。这些函数也不会访问实际的文件系统。如果你必须要处理符号链接，你可能要使用 依赖文件系统的路径转换。

`lexically_...()` 函数会返回一个新的路径，而其他的转换函数将返回相应的字符串类型。所有这些函数都不会修改调用者的路径。

例如，下面的代码：

```
std::filesystem::path p{"/dir/./sub//sub1/../sub2"};
std::cout <<  "path:                " << p << '\n';
std::cout <<  "string():            " << p.string() << '\n';
std::wcout << "wstring():           " << p.wstring() << '\n';
std::cout <<  "lexically_normal(): " << p.lexically_normal() << '\n';
```

调用	效果
`strm << p`	用双引号括起来输出路径
`strm >> p`	读取用双引号括起来的路径
`p.string()`	以 `std::string` 返回路径
`p.wstring()`	以 `std::wstring` 返回路径
`p.u8string()`	以类型为 `std::u8string` 的 UTF-8 字符串返回路径
`p.u16string()`	以类型为 `std::u16string` 的 UTF-16 字符串返回路径
`p.u32string()`	以类型为 `std::u32string` 的 UTF-32 字符串返回路径
`p.string<...>()`	以 `std::basic_string` 返回路径
`p.lexically_normal()`	返回正规化的路径
`p.lexically_relative(p2)`	返回从 `p2` 到 `p` 的相对路径（如果没有则返回空路径）
`p.lexically_proximate(p2)`	返回从 `p2` 到 `p` 的路径（如果没有则返回 `p`）

Table 20.5: 路径 I/O 和转换

前三行的输出是相同的：

```
path                "/dir/./sub//sub1/../sub2"
string():           /dir/./sub//sub1/../sub2
wstring():          /dir/./sub//sub1/../sub2
```

但最后一行的输出就依赖于目录分隔符了。在 POSIX 兼容系统上输出是：

```
lexically_normal(): "/dir/sub/sub2"
```

而在 Windows 上输出是：

```
lexically_normal(): "\\dir\\sub\\sub2"
```

路径 I/O

首先，注意 I/O 运算符以双引号括起来的字符串方式读写路径。你可以把它们转换为字符串来避免双引号：

```
std::filesystem::path file{"test.txt"};
std::cout << file << '\n';          // 输出："test.txt"
std::cout << file.string() << '\n'; // 输出：test.txt
```

在 Windows 上，情况可能会更糟糕。下面的代码：

```
std::filesystem::path tmp{"C:\\Windows\\Temp"};
std::cout << tmp << '\n';
std::cout << tmp.string() << '\n';
std::cout << '"' << tmp.string() << "\"\n";
```

将会有如下输出：

```
"C:\\Windows\\Temp"
C:\Windows\Temp
"C:\Windows\Temp"
```

注意读取路径时既支持带双引号的字符串也支持不带双引号的字符串。因此，所有的输出形式都能使用输入运算符再读取回来：

```
std::filesystem::path tmp;
std::cin >> tmp;    // 读取有双引号和无双引号的路径
```

正规化

当你处理可移植代码时正规化可能会导致更多令人惊奇的结果。例如：

```
std::filesystem::path p2{"//host\\dir/sub\\/./\\"};
// 译者注：此处原文是
// std::filesystem::path p2{"//dir\\subdir/subsubdir\\/./\\"};
// 应是作者笔误

std::cout << "p2:                 " << p2 << '\n';
std::cout << "lexically_normal(): " << p2.lexically_normal() << '\n';
```

在 Windows 系统上可能会有如下输出：

```
p2:                 "//host\\dir/sub\\/./\\"
lexically_normal(): "\\\\host\\dir\\sub\\"
```

然而，在 POSIX 兼容系统上，输出将是：

```
p2:                 "//host\\dir/sub\\/./\\"
lexically_normal(): "/host\\dir/sub\\/\\"
```

原因是对于 POSIX 兼容系统来说反斜杠既不是路径分隔符也不是有效的根名称，这意味着我们得到了一个有三个文件名的绝对路径，三个文件名分别是 `host\dir`、`sub\`、`\`。在 POSIX 兼容系统上，没有办法把反斜杠作为目录分隔符处理（`generic_string()` 和 `make_preferred()` 也没有用）。因此，对于可移植的代码，当处理路径时你应该总是使用通用路径格式。

但是，当遍历当前目录时使用 `lexically_normal()` 移除开头的点是个好方法。

相对路径

`lexically_relative()` 和 `lexically_proximate()` 都可以被用来计算两个路径间的相对路径。不同之处在于如果没有相对路径时的行为，只有当一个是相对路径一个是绝对路径或者两个路径的根名称不同时才会发生这种情况。这种情况下：

- 对于 `p.lexically_relative(p2)`，如果没有从 `p2` 到 `p` 的相对路径，会返回空路径。
- 对于 `p.lexically_proximate(p2)`，如果没有从 `p2` 到 `p` 的相对路径，会返回 `p`。

因为这两个操作都是词法操作，所以不会考虑实际的文件系统（可能会有符号链接）和 `current_path()`。如果两个路径相同，相对路径将是`"."`。例如：

```
fs::path{"/a/d"}.lexically_relative("/a/b/c");      // "../../d"
fs::path{"/a/b/c"}.lexically_relative("/a/d");      // "../b/c"
fs::path{"/a/b"}.lexically_relative("/a/b");        // "."
fs::path{"/a/b"}.lexically_relative("/a/b/");       // "."
```

```
fs::path{"/a/b"}.lexically_relative("/a/b\\");      // "."
fs::path{"/a/b"}.lexically_relative("/a/d/../c");   // "../b"
fs::path{"a/d/../b"}.lexically_relative("a/c");     // "../d/../b"
fs::path{"a//d/..//b"}.lexically_relative("a/c");   // "../d/../b"
```

在 Windows 平台上，将会有：

```
fs::path{"C:/a/b"}.lexically_relative("c:/c/d");    // ""
fs::path{"C:/a/b"}.lexically_relative("D:/c/d");    // ""
fs::path{"C:/a/b"}.lexically_proximate("D:/c/d");   // "C:/a/b"
```

转换为字符串

通过 `u8string()` 你可以将路径用作 UTF-8 字符串，这是当前存储数据的通用格式。例如：

```
// 把路径存储为UTF-8字符串：
std::vector<std::string> utf8paths; // 自从C++20起将改为std::u8string
for (const auto& entry : fs::directory_iterator(p)) {
    utf8paths.push_back(entry.path().u8string());
}
```

注意自从 C++20 起 `u8string()` 的返回值可能会从 `std::string` 改为 `std::u8string()`（新的 UTF-8 字符串类型和存储 UTF-8 字符的 `char8_t` 类型的提案在https://wg21.link/p0482）。[4]

成员模板 `string<>()` 可以用来转换成特殊的字符串类型，例如一个大小写无关的字符串类型：

```
struct ignoreCaseTraits : public std::char_traits<char> {
    // 大小写不敏感的比较两个字符：
    static bool eq(const char& c1, const char& c2) {
        return std::toupper(c1) == std::toupper(c2);
    }
    static bool lt(const char& c1, const char& c2) {
        return std::toupper(c1) < std::toupper(c2);
    }
    // 比较s1和s2的至多前n个字符：
    static int compare(const char* s1, const char* s2, std::size_t n);
    // 搜索s中的字符c：
    static const char* find(const char* s, std::size_t n, const char& c);
};

// 定义一个这种类型的字符串：
using icstring = std::basic_string<char, ignoreCaseTraits>;

std::filesystem::path p{"/dir\\subdir/subsubdir\\/./\\"};
icstring s2 = p.string<char, ignoreCaseTraits>();
```

注意你不应该使用函数 `c_str()`，因为它会转换为本地字符串格式，例如可能会使用 `wchar_t`，因此你需要使用 `std::wcout` 代替 `std::cout` 来输出到输出流。

[4] 感谢 Tom Honermann 指出这一点，并做出这个改进（C++ 开始提供真正的 UTF-8 支持是非常重要的）。

20.3.4 本地和通用格式的转换

表本地和通用格式的转换列出了在通用路径格式 和实际平台特定实现的格式之间转换的方法。

调用	效果
`p.generic_string()`	返回 `std::string` 类型的通用路径
`p.generic_wstring()`	返回 `std::wstring` 类型的通用路径
`p.generic_u8string()`	返回 `std::u8string` 类型的通用路径
`p.generic_u16string()`	返回 `std::u16string` 类型的通用路径
`p.generic_u32string()`	返回 `std::u32string` 类型的通用路径
`p.generic_string<...>()`	返回 `std::basic_string<...>()` 类型的通用路径
`p.native()`	返回 `path::string_type` 类型的本地路径格式
到本地路径的转换	到本地字符类型的隐式转换
`p.c_str()`	返回本地字符串格式的字符序列形式的路径
`p.make_preferred()`	把 `p` 中的目录分隔符替换为本地格式的分隔符并返回修改后的 `p`

Table 20.6: 本地和通用格式的转换

这些函数在 POSIX 兼容系统上没有效果，因为这些系统的本地格式和通用格式没有区别。在其他平台上调用这些函数可能会有效果：

- `generic...()` 函数返回转换为通用格式之后的相应类型的字符串。
- `native()` 返回用本地字符串编码的路径，其类型为 `std::filesystem::path::string_type`。这个类型在 Windows 下是 `std::wstring`，这意味着你需要使用 `std::wcout` 代替 `std::cout` 来输出。新的重载允许我们向文件流传递本地字符串。
- `c_str()` 以空字符结尾的字符序列形式返回结果。注意使用这个函数是不可移植的，因为使用 `std::cout` 打印字符序列在 Windows 上的输出不正确。你应该使用 `std::wcout`。
- `make_preferred()` 会使用本地的目录分隔符替换除了根名称之外的所有的目录分隔符。注意这是唯一一个会修改调用者的函数。因此，严格来讲，这个函数应该属于下一节的修改路径的函数，但是因为它是处理本地格式的转换，所以也在这里列出。

例如，在 Windows 上，下列代码：

```
std::filesystem::path p{"/dir\\subdir/subsubdir\\/./\\"};
std::cout <<  "p:                  " << p << '\n';
std::cout <<  "string():           " << p.string() << '\n';
std::wcout << "wstring():          " << p.wstring() << '\n';
std::cout <<  "lexically_normal(): " << p.lexically_normal() << '\n';
std::cout <<  "generic_string():   " << p.generic_string() << '\n';
std::wcout << "generic_wstring():  " << p.generic_wstring() << '\n';
// 因为这是在Windows下，相应的本地字符串类型是wstring：
std::wcout << "native():          " << p.native() << '\n'; // Windows!
std::wcout << "c_str():           " << p.c_str() << '\n';
std::cout <<  "make_preferred():   " << p.make_preferred() << '\n';
std::cout <<  "p:                  " << p << '\n';
```

将会有如下输出：

```
p:                  "/dir\\subdir/subsubdir\\/./\\"
string():           /dir\subdir/subsubdir\/./\
wstring():          /dir\subdir/subsubdir\/./\
lexically_normal(): "\\dir\\subdir\\subsubdir\\"
generic_string():   /dir/subdir/subsubdir//.//
generic_wstring():  /dir/subdir/subsubdir//.//
native():           /dir\subdir/subsubdir\/./\
c_str():            /dir\subdir/subsubdir\/./\
make_preferred():   "\\dir\\subdir\\subsubdir\\\\.\\\\"
p:                  "\\dir\\subdir\\subsubdir\\\\.\\\\"
```

再次注意：

- 本地字符串格式是不可移植的。在 Windows 上是 `wstring`，而在 POSIX 兼容系统上是 `string`，这意味着你要使用 `cout` 而不是 `wcout` 来打印 `native()` 的结果。
- 只有 `make_preferred()` 会修改调用者。其他的调用都会保持 `p` 不变。

20.3.5 修改路径

表修改路径列出了可以直接修改路径的操作。

注意 `+=` 和 `concat()` 简单的把字符添加到路径后，`/`、`/=`、`append()` 则是在路径后用目录分隔符添加一个子路径：

```
std::filesystem::path p{"myfile"};
p += ".git";        // p:myfile.git
p /= ".git";        // p:myfile.git/.git
p.concat("1");      // p:myfile.git/.git1
p.append("1");      // P:myfile.git/.git1/1
std::cout << p << '\n';
std::cout << p / p << '\n';
```

在 POSIX 兼容系统上输出将是：

```
"myfile.git/.git1/1"
"myfile.git/.git1/1/myfile.git/.git1/1"
```

在 Windows 系统上输出将是：

```
"myfile.git\\.git1\\1"
"myfile.git\\.git1\\1\\myfile.git\\.git1\\1"
```

注意如果添加一个绝对路径子路径意味着替换原本的路径。例如，如下操作之后：

```
namespace fs = std::filesystem;
auto p1 = fs::path("/usr") / "tmp";     // 路径是/usr/tmp或者/usr\tmp
auto p2 = fs::path("/usr/") / "tmp";    // 路径是/usr/tmp
auto p3 = fs::path("/usr") / "/tmp";    // 路径是/tmp
auto p4 = fs::path("/usr/") / "/tmp";   // 路径是/tmp
```

我们有了四个指向两个不同文件的路径：

调用	效果
`p = p2`	赋予一个新路径
`p = sv`	赋予一个字符串（视图）作为新路径
`p.assign(p2)`	赋予一个新路径
`p.assign(sv)`	赋予一个字符串（视图）作为新路径
`p.assign(beg, end)`	赋予从 beg 到 end 的元素组成的路径
`p1 / p2`	返回把 p2 作为子路径附加到 p1 之后的结果
`p /= sub`	把 sub 作为子路径附加到路径 p 之后
`p.append(sub)`	把 sub 作为子路径附加到路径 p 之后
`p.append(beg, end)`	把从 beg 到 end 之间的元素作为子路径附加到路径 p 之后
`p += str`	把 str 里的字符添加到路径 p 之后
`p.concat(str)`	把 str 里的字符添加到路径 p 之后
`p.concat(beg, end)`	把从 beg 到 end 之间的元素附加到路径 p 之后
`p.remove_filename()`	移除路径末尾的文件名
`p.replace_filename(repl)`	替换末尾的文件名（如果有的话）
`p.replace_extension()`	移除末尾的文件的扩展名
`p.replace_extension(repl)`	替换末尾的文件的扩展名（如果有的话）
`p.clear()`	清空路径
`p.swap(p2)`	交换两个路径
`swap(p1, p2)`	交换两个路径
`p.make_preferred()`	把 p 中的目录分隔符替换为本地格式的分隔符并返回修改后的 p

Table 20.7: 修改路径

- p1 和 p2 相等，都指向文件 /usr/tmp（注意在 Windows 上它们相等，但 p1 将是 /usr\tmp。
- p3 和 p4 相等，指向文件 /tmp，因为附加的子路径是绝对路径。

对于根元素，是否赋予新的根元素的结果将不同。例如，在 Windows 上，结果将是：

```
auto p1 = fs::path("usr") / "C:/tmp";   // 路径是C:/tmp
auto p2 = fs::path("usr") / "C:";       // 路径是C:
auto p3 = fs::path("C:") / "":          // 路径是C:
auto p4 = fs::path("C:usr") / "/tmp";   // 路径是C:/tmp
auto p5 = fs::path("C:usr") / "C:tmp";  // 路径是C:usr\tmp
auto p6 = fs::path("C:usr") / "c:tmp";  // 路径是c:tmp
auto p7 = fs::path("C:usr") / "D:tmp";  // 路径是D:tmp
```

函数 make_preferred() 在一个 path 内部把目录分隔符替换成本地格式。例如：

```
std::filesystem::path p{"//server/dir//subdir///file.txt"};
p.make_preferred();
std::cout << p << '\n';
```

在 POSIX 兼容系统上输出将是：

```
"//server/dir/subdir/file.txt"
```

在 Windows 系统上输出将是：

```
"\\\\server\\dir\\\\subdir\\\\\\file.txt"
```

注意开头的根名称没有被修改，因为它由两个斜杠或者反斜杠组成。也注意这个函数在 POSIX 兼容系统上也不会把反斜杠转换成斜杠，因为反斜杠不被识别为目录分隔符。`replace_extension()` 可以替换、添加、或者删除扩展名：

- 如果文件已经有扩展名了，它会进行替换。
- 如果文件没有扩展名，会添加新的扩展名。
- 如果你跳过了新的扩展名参数或者新扩展名参数为空，它会移除已有的扩展名。

替换时是否有前导的点没有影响。函数会确保文件名和扩展名之间有且只有一个点。例如：

```
fs::path{"file.txt"}.replace_extension("tmp")   // file.tmp
fs::path{"file.txt"}.replace_extension(".tmp")  // file.tmp
fs::path{"file.txt"}.replace_extension("")      // file
fs::path{"file.txt"}.replace_extension()        // file
fs::path{"dir"}.replace_extension("tmp")        // dir.tmp
fs::path{".git"}.replace_extension("tmp")       // .git.tmp
```

注意“纯扩展名”的文件名（例如 `.git`）不会被当作扩展名。[5]

20.3.6 比较路径

表比较路径列出了可以比较两个路径的操作。

调用	效果
`p1 == p2`	返回两个路径是否相等
`p1 != p2`	返回两个路径是否不相等
`p1 < p2`	返回一个路径是否小于另一个
`p1 <= p2`	返回一个路径是否小于等于另一个
`p1 >= p2`	返回一个路径是否大于等于另一个
`p1 > p2`	返回一个路径是否大于另一个
`p.compare(p2)`	返回 `p` 是小于、等于还是大于 `p2`
`p.compare(sv)`	返回 `p` 是小于、等于还是大于字符串（视图）`sv` 转换成的路径
`equivalent(p1, p2)`	访问实际文件系统的开销较大的比较操作

Table 20.8: 比较路径

注意这些比较操作大部分都不会访问实际的文件系统，这意味着它们只是以词法的方式比较，这样开销很小但可能会导致令人惊奇的结果：

- 使用 `==`、`!=`、`compare()` 的结果是下面的路径都不相同：

[5] 自从 C++17 起才改成这样。在 C++17 之前，最后一条语句的结果将是 `.tmp`。

```
tmp1/f
./tmp1/f
tmp1/./f
tmp1/tmp11/../f
```

- 只有分隔符不同时才会检测为相同。因此，下面的路径是相等的（假设反斜杠也是有效的目录分隔符）：

```
tmp1/f
tmp1//f
tmp1\f
tmp1/\/f
```

如果你使用了`lexically_normal()`那么上面的所有路径都相等（假设反斜杠也是有效的目录分隔符）。例如：

```
std::filesystem::path p1{"tmp1/f"};
std::filesystem::path p2{"./tmp1/f"};

p1 == p2                                                // false
p1.compare(p2)                                          // 非0
p1.lexically_normal() == p2.lexically_normal()          // true
p1.lexically_normal().compare(p2.lexically_normal())    // 0
```

如果你想要访问实际的文件系统来正确处理包含符号链接的情况，你需要使用`equivalent()`。然而，注意这个函数要求两个路径都代表已经存在的文件。因此，一个通用的尽可能精确（但没有最佳的性能）的比较两个路径的方法是：

```
bool pathsAreEqual(const std::filesystem::path& p1,
                   const std::filesystem::path& p2)
{
    return exists(p1) && exists(p2) ? equivalent(p1, p2)
        : p1.lexically_normal() == p2.lexically_normal();
}
```

20.3.7 其他路径操作

表其他路径操作列出了剩余的路径操作。

调用	效果
`p.hash_value()`	返回一个路径的哈希值

Table 20.9: 其他路径操作

注意只有相等的路径才保证有相同的哈希值，下面的路径返回不同的哈希值：

```
tmp1/f
./tmp1/f
tmp1/./f
tmp1/tmp11/../f
```

因此，在把它们放入哈希表之前，你可能需要先对路径进行正规化。

20.4 文件系统操作

这一节介绍开销更大的会访问实际文件系统的操作。

因为这些操作通常会访问文件系统（要确定是否文件存在、解析符号链接等等），它们比纯路径操作的开销要大的多。因此，他们通常是独立函数。

20.4.1 文件属性

对于一个路径你可以查询很多文件属性。首先，表文件类型的操作 列出了你可以用来检查路径 p 是否指向特定的文件并查询它的类型（如果有的话）。注意这些操作都会访问实际的文件系统，也都是独立函数。

调用	效果
`exists(p)`	返回是否存在一个可访问到的文件
`is_symlink(p)`	返回是否文件 p 并且是符号链接
`is_regular_file(p)`	返回是否文件 p 存在并且是普通文件
`is_directory(p)`	返回是否文件 p 存在并且是目录
`is_other(p)`	返回是否文件 p 存在并且不是普通文件或目录或符号链接
`is_block_file(p)`	返回是否文件 p 存在并且是块特殊文件
`is_character_file(p)`	返回是否文件 p 存在并且是字符特殊文件
`is_fifo(p)`	返回是否文件 p 存在并且是 FIFO 或者管道文件
`is_socket(p)`	返回是否文件 p 存在并且是套接字

Table 20.10: 文件类型操作

文件系统类型的函数和相应的`file_type`值相对应。然而，注意这些函数（除了`is_symlink()`之外）都会解析符号链接。也就是说，对于一个指向目录的符号链接，`is_symlink()`和`is_directory()`都返回`true`。

还要注意对于特殊文件（非普通文件、非目录、非符号链接）的检查，根据其他文件类型的定义，`is_other()`会返回`true`。

对于实现特定的文件类型没有明确的便捷函数，这意味着只有`is_other()`会返回`true`（如果是一个指向这种文件的符号链接，那么`is_symlink()`也会返回`true`）。你可以使用文件状态 API来检测这些特定的类型。

如果不想解析符号链接，可以像下面将要讨论的`exists()`一样使用`symlink_status()`，并对返回的`file_status`调用这些函数。

检查文件是否存在

`exists()`检查是否有一个可以打开的文件。像之前讨论的一样，它会解析符号链接。因此，对于指向不存在文件的符号链接它会返回`false`。

因此，下面的代码不能像预期中工作：

```
// 如果不存在文件p，创建一个符号链接p指向file：
```

```
if (!exists(p)) {    // OOPS：检查文件p指向的目标是否不存在
    std::filesystem::create_symlink(file, p);
}
```

如果 p 已经存在但是是一个指向不存在文件的符号链接，`create_symlink()` 将会尝试在 p 位置处创建新的符号链接并抛出一个相应的异常。

因为多用户/多进程操作系统中文件系统的状况可能会在任意时刻改变，最佳的方法就是尝试进行操作并在失败时处理错误。因此，我们可以直接进行操作并处理相应的异常或者传递额外的参数来处理错误码。

然而，有时你必须检测文件是否存在（在进行文件系统操作之前）。例如，如果你想在指定位置创建一个文件并且该位置处已经有了一个符号链接，那么新创建的文件（可能）会在一个意料之外的地方创建或者覆盖。这种情况下，你应该像下面这样检查文件是否存在：[6]

```
if (!exists(symlink_status(p))) {    // OK：检查p是否还不存在（作为符号链接）
    ...
}
```

这里，我们使用了`symlink_status()`，它会在不解析符号链接的情况下返回文件状态，以检查位置 p 处是否有任何文件存在。

其他文件属性

表文件属性操作列出了一些检查额外文件属性的独立函数。

调用	效果
`is_empty()`	返回文件是否为空
`file_size()`	返回文件大小
`hard_link_count(p)`	返回硬链接数量
`last_write_time(p)`	返回最后一次修改文件的时间

Table 20.11: 文件系统属性

注意路径是否为空和路径指向的文件是否为空是不同的：

```
p.empty()       // 如果路径p为空返回true（开销很小）
is_empty(p)     // 如果路径p处的文件为空返回true（文件系统操作）
```

如果文件存在并且是普通文件，那么 `file_size(p)` 以字节为单位返回文件 p 的大小（在 POSIX 系统上，这个值和 `stat()` 返回的 `st_size` 成员的值相同）。对于所有其他文件，结果是实现特定的，因此不可移植。

`hard_link_count(p)` 返回文件系统中某一个文件存在的次数。通常情况下，这个数字是 1，但是在某些文件系统上同一个文件可以出现在不同的位置（也就是有多个不同的路径）。这和软链接指向其他文件不同，这里的路径可以直接访问文件。只有当最后一个硬链接被删除时文件才会被删除。

[6] 感谢 Billy O’Neal 指出这一点。

处理最后修改的时间

`last_write_time(p)` 返回文件最后一次修改或者写入的时间。返回的类型是标准 `chrono` 库里的时间点类型 `time_point`：

```
namespace std::filesystem {
    using file_time_type = chrono::time_point<travialClock>;
}
```

时钟类型 *trivialClock* 是一个实现特定的时钟类型，它能反应时钟的精度和范围。例如，你可以这么使用它：

```
void printFileTime(const std::filesystem::path& p)
{
    auto filetime = last_write_time(p);
    auto diff = std::filesystem::file_time_type::clock::now() - filetime;
    std::cout << p << " is "
              << std::chrono::duration_cast<std::chrono::seconds>(diff).count()
              << " Seconds old.\n";
}
```

输出可能是：

```
"fileattr.cpp" is 4 Seconds old.
```

这个例子中，你可以使用：

```
decltype(filetime)::clock::now()
```

来代替

```
std::filesystem::file_time_type::clock::now()
```

注意文件系统时间点使用的时钟不保证是标准的 `system_clock`。因此，现在还没有把文件系统时间点转换为 `time_t` 然后在字符串或者输出中用作绝对时间的标准化支持。[7] 然后还是有一些解决方法的，下面的代码“粗略地”把任何时钟的时间点转换为 `time_t` 对象：

```
template<typename TimePoint>
std::time_t toTimeT(TimePoint tp)
{
    using system_clock = std::chrono::system_clock;
    return system_clock::to_time_t(system_clock::now() + (tp - decltype(tp)::clock::now()));
}
```

技巧是计算出文件系统时间点和现在的差值，类型是 `duration`，然后加上现在的系统时钟的时间就可以得到用系统时钟表示的文件系统时间点。这个函数不是很精确，因为不同的时钟可能有不同的精度，而且两次 `now()` 调用之间可能时间差。然而，一般来说，这个函数工作得很好。

例如，对一个路径 `p` 我们可以调用：

```
auto ftime = last_write_time(p);
std::time_t t = toTimeT(ftime);
// 转换为日历时间（跳过末尾的换行符）：
```

[7] C++20 中引入的 `clock_cast` 将休复这个问题。

```
std::string ts = ctime(&t);
ts.resize(ts.size()-1);
std::cout << "last access of " << p << ": " << ts << '\n';
```

输出可能是：

```
last access of "fileattr.exe": Sun Jun 24 10:41:12 2018
```

为了把时间格式化为我们想要的格式，我们可以这样调用：

```
std::time_t t = toTimeT(ftime);
char mbstr[100];
if (std::strftime(mbstr, sizeof(mbstr), "last access: %B %d, %Y at %H:%M\n",
                    std::localtime(&t))) {
    std::cout << mbstr;
}
```

输出可能是：

```
last access: June 24, 2018 at 10:41
```

把任意文件系统时间点转换为字符串的一个有用的辅助函数可以是：

filesystem/ftimeAsString.hpp

```
#include <string>
#include <chrono>
#include <filesystem>

std::string asString(const std::filesystem::file_time_type &ft)
{
    using system_clock = std::chrono::system_clock;
    auto t = system_clock::to_time_t(system_clock::now()
                                     + (ft - std::filesystem::file_time_type::clock::now()));
    // 转换为日历时间(跳过末尾的换行符)：
    std::string ts = ctime(&t);
    ts.resize(ts.size()-1);
    return ts;
}
```

注意 `ctime()` 和 `strftime()` 是非线程安全的，不能在并发环境中使用。

参见修改已存在的文件小节查看相应的修改最后访问的 API。

20.4.2 文件状态

为了避免文件系统访问，有一个特殊的类型 `file_status` 可以被用来存储并修改被缓存的文件类型和权限。发生以下情况时可以设置状态：

- 当使用表文件状态的操作中的方法访问某个指定路径的文件状态时
- 当遍历目录时

调用	效果
`status(p)`	返回文件 p 的 file_status（解析符号链接）
`symlink_status(p)`	返回文件 p 的 file_status(解析符号链接)

Table 20.12: 文件状态的操作

不同之处在于路径 `p` 是否解析符号链接，`status()` 将会解析符号链接并返回指向的文件的属性（文件状态也可能是没有文件），而 `symlink_status(p)` 将会返回符号链接自身的状态。

表file_status 的操作列出了 `file_status` 类型的对象 `fs` 的所有操作。

调用	效果
`exists(fs)`	返回是否有文件存在
`is_regular_file(fs)`	返回是否有文件存在并且是普通文件
`is_directory(fs)`	返回是否有文件存在并且是目录
`is_symlink(fs)`	返回是否有文件存在并且是符号链接
`is_other(fs)`	返回是否有文件存在并且不是普通文件或目录或符号链接
`is_character_file(fs)`	返回是否有文件存在并且是字符特殊文件
`is_block_file(fs)`	返回是否有文件存在并且是块特殊文件
`is_fifo(fs)`	返回是否有文件存在并且是 FIFO 或者管道文件
`is_socket(fs)`	返回是否有文件存在并且是套接字
`fs.type()`	返回文件的 `file_type`
`fs.permissions()`	返回文件的权限

Table 20.13: `file_status` 的操作

使用状态操作的一个好处是可以节省同一个文件的多次操作系统调用。例如，原本的如下代码：

```
if (!is_directory(path)) {
    if (is_character_file(path) || is_block_file(path)) {
        ...
    }
    ...
}
```

你可以以如下方式更高效的实现：

```
auto pathStatus{status(path)};
if (!is_directory(pathStatus)) {
    if (is_character_file(pathStatus) || is_block_file(pathStatus)) {
        ...
    }
    ...
}
```

另一个关键的好处是通过使用 `symlink_status()`，你可以在不解析任何符号链接的情况下检查文件状态。这很有用，例如当你想检测指定路径处是否有文件存在时。

因为文件状态不使用操作系统，所以没有提供相应的返回错误码的版本。

以路径作为参数的 `exists()` 和 `is_...()` 函数是调用并检查文件状态的 `type()` 的缩写。例如，

```
is_regular_file(mypath);
```

是如下代码的缩写：

```
is_regular_file(status(mypath))
```

上面的代码又是下面代码的缩写：

```
status(mypath).type() == file_type::regular
```

20.4.3 权限

处理权限的模型来自于 UNIX/POSIX 的世界。用若干位来表示文件所有者、同组其他用户、其他用户的读、写、执行/搜索的权限。另外，还有特殊的位表示“运行时设置用户 ID”、“运行时设置组 ID”和粘贴位（或者其他操作系统特定的含义）。

表权限位列出了位域枚举类型 `perms` 的值，该类型定义在 `std::filesystem` 中，表示一个或多个权限位。

你可以查询当前的权限并检查返回的 `perms` 对象。为了组合标记，你需要使用位运算。例如：

```
// 如果可写
if ((fileStatus.permissions() &
    (fs::perms::owner_write | fs::perms::group_write | fs::perms::others_write))
    != fs::perms::none) {
    ...
}
```

一个较短（但可读性较差）的初始化位掩码的方法是直接使用相应的 8 进制值和 更宽松的枚举初始化特性：

```
// 如果可写：
if ((fileStatus.permissions() & fs::perms{0222}) != fs::perms::none) {
    ...
}
```

注意你必须把 `&` 表达式放在括号里，因为它的优先级低于比较运算符。还要注意不能跳过比较，因为没有从位域枚举类型到 `bool` 的隐式类型转换。

作为另一个例子，为了像 UNIX 命令 `ls -l` 一样把权限位转换为字符串，你可以使用如下的辅助函数：

filesystem/permAsString.hpp

```
#include <string>
#include <chrono>
#include <filesystem>

std::string asString(const std::filesystem::perms &pm) {
    using perms = std::filesystem::perms;
```

枚举	8 进制值	POSIX	含义
none	0		没有权限集
owner_read	0400	S_IRUSR	所属用户有读权限
owner_write	0200	S_IWUSR	所属用户有写权限
owner_exec	0100	S_IXUSR	所属用户有执行/搜索权限
owner_all	0700	S_IRWXU	所属用户有所有权限
group_read	040	S_IRGRP	同组用户有读权限
group_write	020	S_IWGRP	同组用户有写权限
group_exec	010	S_IXGRP	同组用户有执行/搜索权限
group_all	070	S_IRWXG	同组用户有所有权限
others_read	04	S_IROTH	其他用户有读权限
others_write	02	S_IWOTH	其他用户有写权限
others_exec	01	S_IXOTH	其他用户有执行/搜索权限
others_all	07	S_IRWXO	其他用户有所有权限
all	0777		所有用户有所有权限
set_uid	04000	S_ISUID	运行时设置用户 ID
set_gid	02000	S_ISGID	运行时设置组 ID
sticky_bit	01000	S_ISVTX	操作系统特定
mask	07777		所有可能位的掩码
unknown	0xFFFF		未知权限

Table 20.14: 权限位

```
    std::string s;
    s.resize(9);
    s[0] = (pm & perms::owner_read)   != perms::none ? 'r' : '-';
    s[1] = (pm & perms::owner_write)  != perms::none ? 'w' : '-';
    s[2] = (pm & perms::owner_exec)   != perms::none ? 'x' : '-';
    s[3] = (pm & perms::group_read)   != perms::none ? 'r' : '-';
    s[4] = (pm & perms::group_write)  != perms::none ? 'w' : '-';
    s[5] = (pm & perms::group_exec)   != perms::none ? 'x' : '-';
    s[6] = (pm & perms::others_read)  != perms::none ? 'r' : '-';
    s[7] = (pm & perms::others_write) != perms::none ? 'w' : '-';
    s[8] = (pm & perms::others_exec)  != perms::none ? 'x' : '-';
    return s;
}
```

这允许你使用标准输出打印出文件的权限：

```
std::cout << "permissions: " << asString(stasus(mypath).permissions()) << '\n';
```

对于一个所有者拥有所有权限、其他用户拥有读和执行权限的文件，输出可能是：

```
permissions: rwxr-xr-x
```

然而，注意 Windows 的 ACL（访问控制列表）并不是完全按照这套方案来划分权限的。因此，当使用 Visual C++ 时，可写的文件总是同时有读、写、执行位被设置（即使它不是可执行文件），带有只写标志的文件通常有读和执行位设置。这也影响了可移植的修改权限。的 API。

20.4.4 修改文件系统

你可以通过创建、删除或者修改已有文件来修改文件系统。

创建和删除文件

表创建和删除文件列出了通过路径 p 创建和删除文件的操作。

调用	效果
`create_directory(p)`	创建目录
`create_directory(p, attrPath)`	创建属性为 `attrPath` 的目录
`create_directories(p)`	创建目录和该路径中所有不存在的目录
`create_hard_link(to, new)`	为已存在文件 `to` 创建另一个文件系统项 `new`
`create_symlink(to, new)`	创建指向 `to` 的符号链接 `new`
`create_directory_symlink(to, new)`	创建指向目录 `to` 的符号链接 `new`
`copy(from, to)`	拷贝任意类型的文件
`copy(from, to, options)`	以选项 `options` 拷贝任意类型的文件
`copy_file(from, to)`	拷贝文件（不能是目录或者符号链接）
`copy_file(from, to, options)`	以选项 `options` 拷贝文件
`copy_symlink(from, to)`	拷贝符号链接（`to` 也指向 `from` 指向的文件）
`remove(p)`	删除一个文件或者空目录
`remove_all(p)`	删除路径 p 并递归删除所有子文件（如果有的话）

Table 20.15: 创建和删除文件

没有函数创建普通文件。这可以通过标准 I/O 流库做到。例如，下面的语句会创建一个新的空文件（如果不存在的话）：

```
std::ofstream("log.txt");
```

创建一个或多个目录的函数返回指定目录是否被创建。因此，如果发现指定路径已经有目录的话并不会返回错误。[8] `copy...()` 函数对特殊文件类型无效。默认情况下它们会：

- 如果已经存在文件时报错
- 不递归操作
- 解析符号链接

[8] 一开始，C++17 还指明如果已经有非目录的文件存在时也不返回错误。之后这被作为 C++17 的缺陷修复了（见https://wg21.link/p1164r1）。

这些默认行为可以通过参数 options 覆盖，该参数的类型是位域枚举类型 copy_options，定义在命名空间 std::filesystem 中。

表拷贝选项列出了所有可能的值。当创建目录的符号链接时，使用 create_directory_symlink()

copy_options	效果
none	默认值（值为 0）
skip_existing	跳过覆盖已有文件
overwrite_existing	覆盖已有文件
update_existing	如果新文件更新的话覆盖已有文件
recursive	递归拷贝子目录和内容
copy_symlinks	拷贝符号链接为符号链接
skip_symlinks	忽略符号链接
directories_only	只拷贝目录
create_hard_links	创建新的硬链接而不是拷贝文件
create_symlinks	创建符号链接而不是拷贝文件（源路径必须是绝对路径，除非目标路径在当前目录下）

Table 20.16: 拷贝选项

比 create_symlink() 更好，因为有些操作系统需要显式指明目标是一个目录。注意第一个参数是相对于要创建的符号链接所在的目录的相对路径。因此，要创建一个指向 sub/file.txt 的符号链接 sub/slink，你必须调用：

```
std::filesystem::create_symlink("file.txt", "sub/slink");
```

语句

```
std::filesystem::create_symlink("sub/file.txt", "sub/slink");
```

将会创建一个指向 sub/sub/file.txt 的符号链接 sub/slink。

删除文件的函数有以下行为：

- remove() 删除一个文件或者空目录。如果没有要删除的文件/目录或者不能被删除时返回 false 而不会抛出异常。
- remove_all() 递归删除一个文件或者目录。它返回一个 uintmax_t 类型的值表示删除的文件数量。如果没有文件时返回 0，如果出错时返回 uintmax_t(-1) 而不会抛出异常。

在这两种情况下，都会删除符号链接本身而不是它们指向的文件。

注意当传递字符串字面量作为参数时你应该总是正确地使用完全限定的 remove()，否则可能会调用 C 函数 remove()。

修改已存在的文件

表修改文件列出了修改已存在文件的操作。

调用	效果
`rename(old, new)`	重命名并/或移动文件
`last_write_time(p, newtime)`	修改最后修改时间
`permissions(p, prms)`	用 `prms` 替换文件权限
`permissions(p, prms, mode)`	根据 `mode` 修改文件权限
`resize_file(p, newSize)`	修改普通文件的大小

Table 20.17: 修改文件

`rename()` 可以除了任何类型的文件（包括目录和符号链接）。对于符号链接，它会重命名符号链接本身，而不是指向的文件。注意 `rename()` 需要完整的包含文件名的目标路径：

```
// 移动"tmp/sub/x"到"tmp/x":
std::filesystem::rename("tmp/sub/x", "tmp");    // ERROR
std::filesystem::rename("tmp/sub/x", "tmp/x");  // OK
```

`last_write_time()` 使用在处理最后修改的时间小节介绍的时间点格式。例如：

```
// 创建文件p（更新最后修改时间）
last_write_time(p, std::filesystem::file_time_type::clock::now());
```

`permissions()` 使用了在权限小节描述的 API。可选的 `mode` 参数是一个位域枚举类型，定义在命名空间 `std::filesystem` 中。一方面，它允许你在 `replace`、`add`、`remove` 之间选择。另一方面，使用 `nofollow` 可以让你修改符号链接本身的权限而不是它们指向的文件的权限。例如：

```
// 移除同组用户的写权限和其他用户的所有权限
permissions(mypath, std::filesystem::perms::group_write | std::filesystem::perms::others_all,
                    std::filesystem::perm_options::remove);
```

再次注意 Windows 支持的权限概念有些不同。它的ACL 权限概念只支持两种方式：

- 所有用户可读、写、执行/搜索（`rwxrwxrwx`
- 所有用户可读、执行/搜索（`r-xr-xr-x`

为了在两种模式之间可移植的进行切换，你必须同时启用或者禁用三个写权限（只删除一个没有用）：

```
// 可移植的启用/禁用写权限选项值：
auto allWrite = std::filesystem::perms::owner_write
                | std::filesystem::perms::group_write
                | std::filesystem::perms::others_write;
// 可移植的移除写权限：
permissions(file, allWrite, std::filesystem::perm_options::remove);
```

一个更短（但可读性较差）的方法是以如下方式初始化 `allWrite` （使用更宽松的枚举初始化特性）：

```
std::filesystem::perms allWrite{0222};
```

`resize_file()` 可以用来缩减或者扩展普通文件的大小。例如：

```
// 清空文件
resize_file(file, 0);
```

20.4.5 符号链接和依赖文件系统的路径转换

表文件系统路径转换列出了所有访问实际文件系统的处理路径的操作。当你想处理符号链接时这些操作尤其重要。使用纯路径转换开销更小但不会访问实际的文件系统。

调用	效果
`read_symlink(symlink)`	返回已存在的符号链接指向的文件
`absolute(p)`	返回已存在的 `p` 的绝对路径（不解析符号链接）
`canonical(p)`	返回已存在的 `p` 的绝对路径（解析符号链接）
`weakly_canonical(p)`	返回 `p` 的绝对路径（解析符号链接）
`relative(p)`	返回从当前目录到 `p` 的相对（或空）路径
`relative(p, base)`	返回从 `base` 到 `p` 的相对（或空）路径
`proximate(p)`	返回从当前目录到 `p` 的相对（或绝对）路径
`proximate(p, base)`	返回从 `base` 到 `p` 的相对（或绝对）路径

Table 20.18: 文件系统路径转换

注意根据文件是否必须存在、路径是否正规化、是否解析符号链接这些函数有不同的行为。表文件系统路径转换属性列出了这些函数的需求和行为。

调用	必须存在	正规化	解析符号链接
`read_symlink()`	Yes	Yes	Once
`absolute()`	No	Yes	No
`canonical()`	Yes	Yes	All
`weakly_canonical()`	No	Yes	All
`relative()`	No	Yes	All
`proximate()`	No	Yes	All

Table 20.19: 文件系统路径转换属性

下面的函数演示了处理符号链接时这些函数的使用方法和效果：

filesystem/symlink.hpp

```
#include <filesystem>
#include <iostream>

void testSymLink(std::filesystem::path top)
{
    top = absolute(top);     // 转换为绝对路径，方便切换当前目录时
    create_directory(top);   // 确保top存在
    current_path(top);       // 然后切换当前目录
    std::cout << std::filesystem::current_path() << '\n'; // 打印top的路径
```

```
    // 定义我们自己的子目录（还没有创建）：
    std::filesystem::path px{top / "a/x"};
    std::filesystem::path py{top / "a/y"};
    std::filesystem::path ps{top / "a/s"};

    // 打印出一些相对路径（这些文件还不存在）：
    std::cout << px.relative_path() << '\n';         // px的相对路径
    std::cout << px.lexically_relative(py) << '\n'; // 从py到px的路径："../x"
    std::cout << relative(px, py) << '\n';           // 从py到px的路径："../x"
    std::cout << relative(px) << '\n';               // 从当前路径到px的路径："a/x"

    std::cout << px.lexically_relative(ps) << '\n'; // 从ps到px的路径："../x"
    std::cout << relative(px, ps) << '\n';           // 从ps到px的路径："../x"

    // 现在创建子目录和符号链接
    create_directories(px);                          // 创建"top/a/x"
    create_directories(py);                          // 创建"top/a/y"
    if (!is_symlink(ps)) {
        create_directory_symlink(top, ps)            // 创建"top/a/s" -> "top"
    }
    std::cout << "ps: " << ps << '\n' << " -> " << read_symlink(ps) << '\n';

    // 观察词法处理和文件系统处理的相对路径的不同：
    std::cout << px.lexically_relative(ps) << '\n'; // ps到px的路径："../x"
    std::cout << relative(px, ps) << '\n';           // ps到px的路径："a/x"
}
```

注意我们首先把一个可能是相对路径的路径转换成了绝对路径，否则切换当前路径时当前路径可能会影响切换到的位置。`relative_path()`和`lexically_relative()`都是开销很小的成员函数，不会访问实际的文件系统。因此，它们会忽略符号链接。

独立函数`relative()`会访问实际的文件系统。当文件不存在时它的行为和`lexically_relative()`一样。然而，在创建了符号链接`ps`（指向`top`）之后，它会解析符号链接并给出一个不同的结果。

在POSIX系统上，在当前路径`"/tmp"`以参数`"top"`调用上述函数将有如下输出：

```
"/tmp/top"
"tmp/top/a/x"
"../x"
"../x"
"a/x"
"../x"
"../x"
ps: "tmp/top/a/s" -> "/tmp/top"
"../x"
"a/x"
```

在Windows系统上，在当前路径`"C:/temp"`以参数`"top"`调用上述函数将会由如下输出：

```
"C:\\temp\\top"
```

```
"temp\\top\\a/x"
"..\\x"
"..\\x"
"a\\x"
"..\\x"
"..\\x"
ps: "C:\\temp\\top\\a/s" -> "C:\\temp\\top"
"..\\x"
"a\\x"
```

再次注意在 Windows 上你需要管理员权限才能创建符号链接。

20.4.6 其他文件系统操作

表其他操作列出了所有还未提到的其他文件系统操作。

调用	效果
`equivalent(p1, p2)`	返回是否 p1 和 p2 指向同一个文件
`space(p)`	返回路径 p 的磁盘空间信息
`current_path(p)`	将当前工作目录设置为 p

Table 20.20: 其他操作

`equivalent()` 函数在关于路径比较的小节介绍。

`space()` 的返回值是如下的结构体：

```
namespace std::filesystem {
    struct space_info {
        uintmax_t capacity;
        uintmax_t free;
        uintmax_t available;
    };
}
```

因此，使用结构化绑定，你可以像下面这样打印出可用的磁盘空间：

```
auto [cap, _, avail] = std::filesystem::space("/");
std::cout << std::fixed << std::precision(2)
          << avail/1.0e6 << " of " << cap/1.0e6 << " MB available\n\n";
```

输出可能是：

```
43019.82 of 150365.79 MB available
```

`current_path()` 调用将整个程序（也会应用于所有线程）的当前目录设置为参数指示的路径。通过下面的代码，你可以切换到另一个工作目录并在离开作用域时恢复旧的工作目录：

```
// 保存当前路径：
auto currentDir{std::filesystem::current_path();
```

```
try {
    // 临时切换当前路径：
    std::filesystem::current_path(subdir);
    ...     // 执行一些操作
}
catch (...) {
    // 发生异常时恢复当前路径：
    std::filesystem::current_path(currentDir);
    throw;  // 重新抛出
}
// 没有异常时恢复当前路径：
std::filesystem::current_path(currentDir);
```

20.5 遍历目录

文件系统库的一个关键作用就是遍历一个文件系统（子）树中的所有文件。

能做到这一点的最快捷的方法就是使用范围 `for` 循环。你可以遍历一个目录中的所有文件：

```
for (const auto& e : std::filesystem::directory_iterator(dir)) {
    std::cout << e.path() << '\n';
}
```

或者递归遍历一个文件系统（子）树：

```
for (const auto& e : std::filesystem::recursive_directory_iterator(dir)) {
    std::cout << e.path() << '\n';
}
```

传入的参数 `dir` 可以是一个 `path` 也可以是其他可以隐式转换为路径的类型（特指各种形式的字符串）。

注意 `e.path()` 返回包含遍历起点目录的文件名。因此，如果我们遍历 `"."` 里的一个文件 `file.txt`，文件名将是 `./file.txt` 或者 `.\file.txt`。

另外，路径被写入输出流时会带有双引号，所以输出时将变为 `"./file.txt"` 或者 `".\\file.txt"`。因此，就像之前的最开始的例子，下面的循环可移植性更强：

```
for (const auto& e : std::filesystem::directory_iterator(dir)) {
    std::cout << e.path().lexically_normal().string() << '\n';
}
```

为了遍历当前目录，传递 `"."` 作为当前目录而不是 `""`。在 Windows 上传递一个空路径是可以的但不可移植。

目录迭代器表示一个范围

你可能会很惊讶你可以把一个迭代器传递给范围 `for` 循环，因为正常情况下应该传递一个范围。

技巧在于 `directory_iterator` 和 `recursive_directory_iterator` 都有提供全局的 `begin()` 和 `end()` 函数重载：

- `begin()` 返回迭代器自身。
- `end()` 返回尾后迭代器，可以使用默认构造函数创建。

因此，你可以像下面这样遍历：

```
std::filesystem::directory_iterator di{p};
for (auto pos = begin(di); pos != end(di); ++pos) {
    std::cout << pos->path() << '\n';
}
```

或者像下面这样：

```
for (std::filesystem::directory_iterator pos{p};
     pos != std::filesystem::directory_iterator{};
     ++pos) {
    std::cout << pos->path() << '\n';
}
```

目录迭代器选项

当遍历目录时，你可以传递 `directory_options` 类型的值，这些值在表 目录迭代器选项中列出。这个类型是一个位域的枚举类型，定义在命名空间 `std::filesystem` 中。

directory_options	效果
none	默认情况（值为 0）
follow_directory_symlink	解析符号链接（而不是跳过它们）
skip_permission_denied	当权限不足时跳过目录

Table 20.21: 目录迭代器选项

默认行为是不解析符号链接并在没有权限访问（子）目录时抛出异常。使用 `skip_permission_denied` 选项可以忽略那些没有权限访问的目录。

filesystem/createfiles.cpp 展示了 `follow_directory_symlink` 选项的一个应用。

20.5.1 目录项

目录迭代器的元素类型是 `std::filesystem::directory_entry`。因此，如果目录迭代器有效的话，使用 `operator*()` 将会返回这个类型。这意味着范围 `for` 循环的正确类型如下所示：

```
for (const std::filesystem::directory_entry& e : std::filesystem::directory_iterator(p)) {
    std::cout << e.path() << '\n';
}
```

目录项包含一个 `path` 对象和一些附加的属性例如硬连接数量、文件状态、文件大小、上次修改时间、是否是符号链接、如果是的话它指向的实际路径。

注意这个迭代器是输入迭代器。因为目录项随时都可能变化，所以重复遍历一个目录可能会得到不同的结果，在并行算法中使用目录迭代器时必须考虑到这一点。

表目录项操作列出了目录项 `e` 的所有操作。他们基本都是一些查询文件属性、获取文件状态、检查权限、比较路径的操作。

`assign()` 和 `replace_filename()` 会调用相应的修改路径操作 但不会修改底层文件系统中的文件。

目录项缓存

鼓励实现缓存额外的文件属性来避免使用目录项时额外的文件系统访问开销。然而，实现并不是必须缓存数据，当然这样开销会变大。[9]

因为所有的值通常会被缓存，因此这些调用通常开销很小：

```
for (const auto& e : std::filesystem::directory_iterator{"."}) {
    auto t = e.last_write_time();   // 通常开销很小
    ...
}
```

不管是否有缓存，但在多用户或者多进程的操作系统中，所有这些迭代都可能返回不再有效的数据。文件的大小和内容可能改变、文件可能被删除或者替换（因此，文件的类型也有可能改变）、权限也可能被修改。

在这种情况下，你可以要求更新目录项持有的数据：

```
for (const auto& e : std::filesystem::directory_iterator{"."}) {
    ...             // 数据已经失效
    e.refresh();    // 刷新文件的缓存数据
    if (e.exists()) {
        auto t = e.last_write_time();
        ...
    }
}
```

或者，你可以查询当前的状态：

```
for (const auto& e : std::filesystem::directory_iterator{"."}) {
    ...             // 数据已经失效
    if (exists(e.path()) {
        auto t = last_write_time(e.path());
        ...
    }
}
```

20.6 后记

文件系统库一开始作为Boost库在Beman Dawes的带领下开发了很多年。在2014年，它第一次成为了正式的测试标准：*File System Technical Specification*（见https://wg21.link/n4100)。

Beman Dawes的https://wg21.link/p0218r0提案使得*File System Technical Specification*被标准库采纳。计算相对路径的支持由Beman Dawes、Nicolai Josuttis、Jamie Allsop在https://wg21.link/p0219r1中添加。Beman Dawes在https://wg21.link/p0317r1中、Nicolai Josuttis在https://wg21.link/p0392r0中、Jason Liu和Hubert Tong在https://wg21.link/p0430r2中、尤其是文件系统小组（Beman Dawes、S. Davis Herring、Nicolai Josuttis、Jason Liu、Billy O'Neal、P.J. Plauger、Jonathan Wakely）在https://wg21.link/p0492r2中修正了很多小错误。

[9]事实上，C++17文件系统库在g++ v9中的测试实现只缓存文件类型，而不缓存文件大小（等到库正式发布时这一点可能会改变。）

C++17 标准发布以后，Nicolai Josuttis 在https://wg21.link/p1164r1提案中作为 C++17 的缺陷修正了`make_directory()`的行为。

调用	效果
`e.path()`	返回当前目录项的文件系统路径
`e.exists()`	返回文件是否存在
`e.is_regular_file()`	返回是否文件存在并且是普通文件
`e.is_directory()`	返回是否文件存在并且是目录
`e.is_symlink()`	返回是否文件存在并且是符号链接
`e.is_other()`	返回是否文件存在并且不是普通文件或目录或符号链接
`e.is_block_file()`	返回是否文件存在并且是块特殊文件
`e.is_character_file()`	返回是否文件存在并且是字符特殊文件
`e.is_fifo()`	返回是否文件存在并且是 FIFO 或者管道文件
`e.is_socket()`	返回是否文件存在并且是套接字
`e.file_size()`	返回文件大小
`e.hard_link_count()`	返回硬链接数量
`e.last_write_time()`	返回最后一次修改时间
`e.status()`	返回文件 p 的状态
`e.symlink_status()`	返回文件 p 的文件状态（解析符号链接）
`e1 == e2`	返回是否两个目录项的路径相等
`e1 != e2`	返回是否两个目录项的路径不相等
`e1 < e2`	返回是否一个目录项的路径小于另一个
`e1 <= e2`	返回是否一个目录项的路径小于等于另一个
`e1 >= e2`	返回是否一个目录项的路径大于等于另一个
`e1 > e2`	返回是否一个目录项的路径大于另一个
`e.assign(p)`	用 p 替换 e 的路径并更新所有属性
`e.replace_filename(p)`	用 p 替换 e 的路径中的文件名并更新所有属性
`e.refresh()`	更新该目录项所有缓存的属性

Table 20.22: 目录项操作

Part IV

已有标准库的拓展和修改

这一部分介绍 C++17 对已有标准库组件的拓展和修改。

Chapter 21

类型特征扩展

C++17扩展了类型特征（标准类型函数）的通用能力并引入了一些新的类型特征。

21.1 类型特征后缀 _v

自从C++17起，你可以对所有返回值的类型特征使用后缀 `_v`（就像你可以为所有返回类型的类型特征使用 `_t` 一样）。例如，对于任何类型 `T`，你现在可以写：

```
std::is_const_v<T>      // 自从C++17起
```

来代替：

```
std::is_const<T>::value // 自从C++11起
```

这适用于所有返回值的类型特征。方法是为每一个标准类型特征定义一个相应的新的变量模板。例如：

```
namespace std {
    template<typename T>
    constexpr bool is_const_v = is_const<T>::value;
}
```

这样一个类型特征可以也用做运行时的条件表达式：

```
if (std::is_signed_v<char>) {
    ...
}
```

然而，因为这些类型特征是在编译期求值，所以你也可以在编译期使用它们（例如，在一个编译期 `if` 语句中）：

```
if constexpr (std::is_signed_v<char) {
    ...
}
```

另一个应用是用来支持不同的模板实例化：

```
// 类C<T>的主模板
template<typename T, bool = std::is_pointer_v<T>>
```

```
class C {
    ...
};

// 指针类型的偏特化版本：
template<typename T>
class C<T, true> {
    ...
};
```

这里，类 C 为指针类型提供了一个偏特化版本。

后缀 _v 也可以用于返回非 bool 类型的类型特征，例如 std::extent<>，返回原生数组的某一个维度的大小：

```
int a[5][7];
std::cout << std::extent_v<decltype(a)> << '\n';    // 打印出5
std::cout << std::extent_v<decltepe(a), 1> << '\n'; // 打印出7
```

21.2 新的类型特征

表新类型特征列出了 C++17 引入的新类型特征。

另外，is_litera_type<> 和 result_of<> 自从 C++17 起被废弃。下面的段落将详细讨论这些特征。

类型特征 is_aggregate<>

std::**is_aggregate<T>**::value

返回 *T* 是否是聚合体类型：

```
template<typename T>
struct D : std::string, std::complex<T> {
    std::string data;
};

D<float> s{{"hello"}, {4.5, 6.7}, "world"};         // 自从C++17起OK
std::cout << std::is_aggregate<decltype(s)>::value; // 输出：1（true）
```

类型特征 is_swappable<> 和 is_swappable_with<>

std::**is_swappable<T>**::value

std::**is_nothrow_swappable<T>**::value

std::**is_swappable_with<T1, T2>**::value

std::**is_nothrow_swappable_with<T1, T2>**::value

在以下情况下返回 true：

- 类型 T 的左值可以被交换，或者
- 类型 T1 和 T2 的值类型可以交换

特征	效果
`is_aggregate<T>`	是否是聚合体类型
`is_swappable<T>`	该类型是否能调用 `swap()`
`is_nothrow_swappable<T>`	该类型是否能调用 `swap()` 并且该操作不会抛出异常
`is_swappable_with<T,T2>`	特定值类型的这两种类型是否能调用 `swap()`
`is_nothrow_swappable_with<T, T2>`	特定值类型的这两种类型是否能调用 `swap()` 并且该操作不会抛出异常
`has_unique_object_representations<T>`	是否该类型的两个值相等的对象在内存中的表示也一样
`is_invocable<T, Args...>`	该类型是否可以用 *Args...* 调用
`is_nothrow_invocable<T, Args...>`	该类型是否可以用 *Args...* 调用，并且该操作不会抛出异常
`is_invocable_r<RT, T, Args...>`	该类型是否可以用 *Args...* 调用并返回 *RT* 类型
`is_nothrow_invocable_r<RT, T, Args...>`	该类型是否可以用 *Args...* 调用并返回 *RT* 类型且不会抛出异常
`invoke_result<T, Args...>`	用 *Args* 作为实参进行调用会返回的类型
`conjunction<B...>`	对 bool 特征 *B...* 进行逻辑与运算
`disjunction<B...>`	对 bool 特征 *B...* 进行逻辑或运算
`negation<B>`	对 bool 特征 B 进行非运算
`is_execution_policy<T>`	是否是执行策略类型

Table 21.1: 新类型特征

（使用 `nothrow` 形式时还要保证不会抛出异常）。

注意 `is_swappable<>` 或 `is_nothrow_swappable<>` 检查你是否可以交换某个指定类型的值（检查该类型的左值）。相反，`is_swappable_with<>` 和 `is_nothrow_swappable_with<>` 还会考虑值类型。也就是说：

```
is_swappable_v<int>             // 返回true
```

等价于

```
is_swappable_with_v<int&, int&> // 返回true（和上边等价）
```

而：

```
is_swappable_with_v<int, int>   // 返回false
```

将会返回 false，因为你不能调用 `std::swap(42, 77)`。

例如：

```
is_swappable_v<std::string>     // 返回true
is_swappable_v<std::string&>    // 返回true
```

```
is_swappable_v<std::string&&>   // 返回true
is_swappable_v<void>            // 返回false
is_swappable_v<void*>           // 返回true
is_swappable_v<char[]>          // 返回false

is_swappable_with_v<std::string, std::string>       // 返回false
is_swappable_with_v<std::string&, std::string&>     // 返回true
is_swappable_with_v<std::string&&, std::string&&>   // 返回false
```

类型特征 has_unique_object_representations<>

`std::has_unique_object_representations<T>::value`

当任意两个值相同的类型 T 的对象在内存中的表示都相同时返回 true。也就是说，两个相同的值在内存中总是有相同的字节序列。

有这种属性的对象可以通过对字节序列哈希来得到对象的哈希值（不用担心某些不参与比较的位可能不同的情况）。

类型特征 is_invocable<> 和 is_invocable_r<>

`std::is_invocable<T, Args...>::value`
`std::is_nothrow_invocable<T, Args...>::value`
`std::is_invocable_r<Ret, T, Args...>::value`
`std::is_nothrow_invocable_r<Ret, T, Args...>::value`

当你能以 `Args...` 为实参调用 T 类型的对象并且返回值可以转换为 `Ret` 类型（并且保证不抛出异常）时返回 true。也就是说，我们可以使用这些特征来测试我们是否可以用 `Args...` 为实参调用（直接调用或者通过 `std::invoke()`）T 类型的可调用对象并返回 `Ret` 类型的值。

例如，如下定义：

```
struct C {
    bool operator() (int) const {
        return true;
    }
};
```

会有如下结果：

```
std::is_invocable<C>::value                         // false
std::is_invocable<C, int>::value                    // true
std::is_invocable<int*>::value                      // false
std::is_invocable<int(*)()>::value                  // true

std::is_invocable_r<bool, C, int>::value            // true
std::is_invocable_r<int, C, long>::value            // true
std::is_invocable_r<void, C, int>::value            // true
std::is_invocable_r<char*, C, int>::value           // false
std::is_invocable_r<long, int(*)(int)>::value       // false
```

```
std::is_invocable_r<long, int(*)(int), int>::value      // true
std::is_invocable_r<long, int(*)(int), double>::value   // true
```

类型特征 invoke_result<>

`std::invoke_result<T, Args...>::type`

返回当使用实参 `Args...` 调用 `T` 类型的对象时会返回的类型。也就是说，我们可以使用这个特征来获知如果使用 `Args...` 调用 `T` 类型的对象时将会返回的类型。

这个类型特征替换了 `result_of<>`，后者现在不应该再使用。

例如：

```
std::string foo(int);

using T1 = std::invoke_result_t<decltype(foo), int>;   // T1是std::string

struct ABC {
    virtual ~ABC() = 0;
    void operator() (int) const {
    }
};

using T2 = typename std::invoke_result<ABC, int>::type; // T2是void
```

bool 类型特征的逻辑操作

表组合其他类型特征的类型特征列出了对 bool 类型类征（几乎所有的返回 bool 类型值的标准类型特征）进行逻辑组合的类型特征。

特征	效果
`conjunction<B...>`	对 bool 特征 *B...* 进行逻辑与运算
`disjunction<B...>`	对 bool 特征 *B...* 进行逻辑或运算
`negation<B>`	对 bool 特征 B 进行非运算

Table 21.2: 组合其他类型特征的类型特征

它们的一大应用就是通过组合现有类型特征来定义新的类型特征。例如，你可以轻松的定义一个检查某个类型是否是“指针”（原生指针，成员函数指针，或者空指针）的特征：

```
template<typename T>
struct IsPtr : std::disjunction<std::is_null_pointer<T>,
                                std::is_member_pointer<T>,
                                std::is_pointer<T>> {
};
```

现在我们在一个编译期 `if` 语句中使用这个新的特征：

```
template<typename T>
void foo(T x)
{
    if constexpr(IsPtr<T>) {
        ... // 处理是指针的情况
    }
    else {
        ... // 处理不是指针的情况
    }
}
```

作为另一个例子，我们可以定义一个检查指定类型是否是整数或者枚举但又不是bool的类型特征：

```
template<typename T>
struct IsIntegralOrEnum : std::conjunction<std::disjunction<std::is_integral<T>,
                                                            std::is_enum<T>>,
                          std::negation<std::is_same<T, bool>>> {
};
```

这里，类似于计算

```
(is_integral<T> || is_enum<T>) && !is_same<T, bool>
```

这么写的一个好处是`std::conjunction<>`和`std::disjunction<>`会*短路求值*bool表达式，这意味着当*conjunction*出现第一个`false`或者*disjunction*出现第一个`true`时就会停止计算。这节省了编译时间，甚至因为短路求值可以在某些情况下使原本无效的代码变得有效。[1] 例如，如果像下面这样使用不完全类型：

```
struct X {
    X(int);     // 从int转换而来
};

struct Y;       // 不完全类型
```

下面的静态断言会失败，因为对于不完全类型`is_constructible`会陷入未定义行为（尽管有些编译器接受这样的代码）：

```
// 未定义行为
static_assert(std::is_constructible<X, int>{} || std::is_constructible<Y, int>{},
              "can't init X or Y from int");
```

下面使用`std::disjunction`的替代版保证不会失败，因为当`is_constructible<X, int>`返回`true`后求值就会停止：

```
// OK：
static_assert(std::disjunction<std::is_constructible<X, int>,
                               std::is_constructible<Y, int>>{},
              "can't init X or Y from int");
```

[1] 感谢Howard Hinnant指出这一点。

21.3 后记

标准类型特征的变量模板由 Stephan T. Lavavej 于 2014 年在https://wg21.link/n3854 中首次提出。它们最终因 Alisdair Meredith 发表于https://wg21.link/p0006r0 的提案而被 Library Fundamentals TS 采纳。

类型特征 `std::is_aggregate<>` 在 C++17 标准中作为美国国家机构对的注释引入（见https://wg21.link/lwg2911）。

`is_swappable` 特征家族由 Andrew Morrow 在https://wg21.link/n3619 中首次提出。最终被接受的提案由 Daniel Krügler 发表于https://wg21.link/p0185r1。

`std::has_unique_object_representations<>` 由 Michael Spencer 在https://wg21.link/p0258r0中首次提出，当时的名称为`is_contiguous_layout`。最终被接受的提案由 Michael Spencer 发表于https://wg21.link/p0258r2。

`is_invocable` 特征家族由 Agustin Berge 在https://wg21.link/n4446中首次提出。和 `std::invoke_result_t<>` 一起被接受的提案由 Daniel Krügler、Pablo Halpern、Jonathan Wakely 发表于https://wg21.link/p0604r0。

bool 类型特征的逻辑操作由 Jonathan Wakely 在https://wg21.link/p0013r0中首次提出。最终被接受的提案由 Jonathan Wakely 发表于https://wg21.link/p0013r1。

Chapter 22

并行 STL 算法

本章的并行算法如果在 *linux* 上使用 *gcc* 或者 *clang* 进行编译的话，会有以下几种情况：

- 没有安装 *tbb* 库，链接时没有指定 *-ltbb* 选项：正常编译，但并行算法会串行化，变得和串行算法一样。
- 安装了 *tbb* 库，链接时没有指定 *-ltbb* 选项：编译错误。
- 安装了 *tbb* 库，链接时指定了 *-ltbb* 选项：正常编译，并行算法并行执行。

显然第三种情况才能正常使用并行算法，遇到第一种情况的读者不要奇怪为什么并行算法和串行算法的性能一样。正确编译选项示例：

```
g++ -std=c++17 -ltbb lib/parreduce.cpp -o prog
```

——译者注

为了从现代的多核体系中受益，C++17 标准库引入了并行 STL 算法来使用多个线程并行处理元素。

许多算法扩展了一个新的参数来指明是否要并行运行算法（当然，没有这个参数的以前的版本仍然受支持）。另外，还多了一些专为并行编程补充的新算法。

一个简单的计时器辅助类

对于这一章中的示例，有时我们需要一个计时器来测量算法的速度。因此，我们使用了一个简单的辅助类，它会初始化一个计时器并提供 `printDiff()` 来打印出消耗的毫秒数并重新初始化计时器：

lib/timer.hpp

```
#ifndef TIMER_HPP
#define TIMER_HPP

#include <iostream>
#include <string>
#include <chrono>


/*****************************************
* timer to print elapsed time
```

```
*****************************************/

class Timer {
  private:
    std::chrono::steady_clock::time_point last;
  public:
    Timer() : last{std::chrono::steady_clock::now()} {
    }
    void printDiff(const std::string& msg = "Timer diff: ") {
        auto now{std::chrono::steady_clock::now()};
        std::chrono::duration<double, std::milli> diff{now - last};
        std::cout << msg << diff.count() << "ms\n";
        last = std::chrono::steady_clock::now();
    }
};

#endif // TIMER_HPP
```

22.1 使用并行算法

让我们从一些示例算法开始，这些算法演示了怎么让现有算法并行运行和使用新的并行算法。

22.1.1 使用并行 `for_each()`

这是第一个例子，简单地演示了并行运行标准算法 `for_each()`：

lib/parforeach.cpp

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <numeric>
#include <execution>    // for执行策略
#include <cmath>        // for math()
#include "timer.hpp"

int main()
{
    int numElems = 1000;

    struct Data {
        double value;   // 初始值
        double sqrt;    // 并行计算平方根
    };

    // 初始化numElems个还没有计算平方根的值
    std::vector<Data> coll;
```

```
    coll.reserve(numElems);
    for (int i=0; i<numElems; ++i) {
        coll.push_back(Data{i * 4.37, 0});
    }

    // 并行计算平方根
    for_each(std::execution::par,
             coll.begin(), coll.end(),
             [](auto& val) {
                val.sqrt = std::sqrt(val.value);
            });
}
```

如你所见，使用并行算法是很简单的：

- 包含头文件 `<execution>`
- 像通常调用算法一样进行调用，只不过需要添加一个第一个参数 `std::execution::par`，这样我们就可以要求算法在并行模式下运行：

```
#include <algorithm>
#include <execution>
...
for_each(std::execution::par,
         coll.begin(), coll.end(),
         [](auto& val) {
             val.sqrt = std::sqrt(val.value);
         });
```

像通常一样，这里的 `coll` 可以是任何范围。然而，注意所有的并行算法要求迭代器至少是前向迭代器（我们会在不同的线程中迭代范围内的元素，如果两个迭代器不能迭代相同的值这将毫无意义）（译者注：括号里的内容怪怪的，实际上是针对输入迭代器和前向迭代器的不同来进行说明）。

并行算法实际运行的方式是实现特定的。当然，使用多线程不一定能加快速度，因为启动和控制多线程也会消耗时间。

性能提升

为了查明是否应该使用和应该在什么时候使用并行算法，让我们把示例修改为下面这样：

lib/parforeachloop.cpp

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <numeric>
#include <execution>    // for执行策略
#include <cstdlib>      // for atoi()
#include "timer.hpp"

int main(int argc, char* argv[])
```

```
{
    // 从命令行读取numElems（默认值：1000）
    int numElems = 1000;
    if (argc > 1) {
        numElems = std::atoi(argv[1]);
    }

    struct Data {
        double value;   // 初始值
        double sqrt;    // 并行计算平方根
    };

    // 初始化numElems个还没有计算平方根的值：
    std::vector<Data> coll;
    coll.reserve(numElems);
    for (int i = 0; i < numElems; ++i) {
        coll.push_back(Data{i * 4.37, 0});
    }

    // 循环来重复测量
    for (int i{0}; i < 5; ++i) {
        Timer t;
        // 顺序执行：
        for_each(std::execution::seq,
                 coll.begin(), coll.end(),
                 [](auto& val) {
                    val.sqrt = std::sqrt(val.value);
                 });
        t.printDiff("sequential: ");

        // 并行执行
        for_each(std::execution::par,
                 coll.begin(), coll.end(),
                 [](auto& val) {
                    val.sqrt = std::sqrt(val.value);
                 });
        t.printDiff("parallel: ");
        std::cout << '\n';
    }
}
```

关键的修改点在于：

- 通过命令行参数，我们可以传递要操作的元素的数量。
- 我们使用了类 Timer 来测量算法运行的时间。
- 我们在循环中重复了多次测量，可以让结果更准确。

结果主要依赖于硬件、使用的 C++ 编译器、使用的 C++ 库。在我的笔记本上（使用 Visual C++，CPU 是 2 核心超线程的 Intel i7），可以得到如下结果：

- 只有 100 个元素时，串行算法明显快的多（快 10 倍以上）。这是因为启动和管理线程占用了太多时间，对于这么少的元素来说完全不值得。
- 10,000 个元素时基本相当。
- 1,000,000 个元素时并行执行快了接近 3 倍。

再强调一次，没有通用的方法来判断什么场景什么时间值得使用并行算法，但这个示例演示了即使只是非平凡的数字操作，也可能值得使用它们。

值得使用并行算法的关键在于：

- 操作很长（复杂）
- 有很多很多元素

例如，使用算法 `count_if()` 的并行版本来统计 vector 中偶数的数量永远都不值得，即使有 1,000,000,000 个元素：

```
auto num = std::count_if(std::execution::par,       // 执行策略
                         coll.begin(), coll.end(),  // 范围
                         [](int elem) {             // 判断准则
                            return elem % 2 == 0;
                         });
```

事实上，对于一个只有快速的判断式的简单算法（比如这个例子），并行执行永远会更慢。适合使用并行算法的场景应该是：对每个元素的处理需要消耗很多的时间并且处理过程需要独立于其他元素的处理。

然而，你并不能事先想好一切，因为什么时候如何使用并行线程取决于 C++ 标准库的实现。事实上，你不能控制使用多少个线程，具体的实现也可能只对一定数量的元素使用多线程。

请在你的目标平台上自行测试适合的场景。

22.1.2 使用并行 `sort()`

排序是另一个使用并行算法的例子。因为排序准则对每个元素都会调用不止一次，所以你可以节省很多时间。（译者注：没看明白这句话什么意思）

考虑像下面这样初始化一个 string 的 vector：

```
std::vector<std::string> coll;
for (int i = 0; i < numElems / 2; ++i) {
    coll.emplace_back("id" + std::to_string(i));
    coll.emplace_back("ID" + std::to_string(i));
}
```

也就是说，我们创建了一个 vector，其元素为以 `"id"` 或 `"ID"` 开头之后紧跟一个整数的字符串：

```
id0 ID0 id1 ID1 id2 ID2 id3 ... id99 ID99 id100 ID100 ...
```

像通常一样，我们可以以如下方式顺序排序元素：

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

现在也可以显式传递一个“顺序”执行策略：

```
sort(std::execution::seq,
     coll.begin(), coll.end());
```

传递顺序执行策略的好处是，如果要在运行时决定是顺序执行还是并行执行的话你不需要再修改调用方式。

想要并行排序也很简单：

```
sort(std::execution::par,
     coll.begin(), coll.end());
```

注意还有另一个并行执行策略：

```
sort(std::execution::par_unseq,
     coll.begin(), coll.end());
```

我会在后文解释其中的不同。

因此，问题又一次变成了（什么时候）使用并行排序会更好？在我的笔记本上 10,000 个字符串时并行排序只需要顺序排序一半的时间。即使只排序 1000 个字符串时并行排序也稍微更快一点。

结合其他的改进

注意还可以进一步修改来提升性能。例如，如果我们只根据数字进行排序，那么我们可以在排序准则中获取不包含前两个字母的子字符串来进行比较，这样可以再一次看到使用并行算法的双重改进：

```
sort(std::execution::par,
     coll.begin(), coll.end(),
     [] (const auto& a, const auto& b) {
        return a.substr(2) < b.substr(2);
     });
```

然而，`substr()` 是一个开销很大的成员函数，因为它会创建并返回一个新的临时字符串。通过使用字符串视图，即使顺序执行也能看到三重改进：

```
sort(coll.begin(), coll.end(),
     [] (const auto& a, const auto& b) {
        return std::string_view{a}.substr(2) < std::string_view{b}.substr(2);
     });
```

我们也可以轻松的把并行算法和字符串视图结合起来：

```
sort(std::execution::par,
     coll.begin(), coll.end(),
     [] (const auto& a, const auto& b) {
        return std::string_view{a}.substr(2) < std::string_view{b}.substr(2);
     });
```

最后，这种方式可能比直接使用 `string` 的 `substr()` 成员然后顺序执行快 10 倍：

```
sort(coll.begin(), coll.end(),
     [] (const auto& a, const auto& b) {
        return a.substr(2) < b.substr(2);
     });
```

22.2 执行策略

你也可以向并行STL算法传递不同的执行策略*(execution policies)*作为第一个参数。它们定义在头文件<execution>中。表执行策略列出了所有标准的执行策略。

策略	含义
std::execution::seq	顺序执行
std::execution::par	并行化顺序执行
std::execution::par_unseq	并行化乱序（矢量化）执行

Table 22.1: 执行策略

让我们仔细讨论这些执行策略：

- 使用seq指定**顺序执行**

  这意味着，像非并行化算法一样，当前线程会一个一个的对所有元素执行操作。使用这个选项和使用不接受执行策略参数的非并行化版本的效果类似，然而，这种形式将比非并行化版本多一些约束条件，例如for_each()不能返回值，所有的迭代器必须至少是前向迭代器。

  提供这个策略的目的是让你可以通过修改一个参数来要求顺序执行，而不是修改调用的函数来做到这一点。然而，注意使用这个策略的并行算法的行为和相应的非并行版本可能有些细微的不同。

- 使用par指定**并行化顺序执行**

  这意味着多个线程将会顺序的对元素执行操作。当某个线程对一个新的元素进行操作之前，它会先处理完它之前处理过的其他元素。

  与par_unseq不同，这个策略可以保证在以下情况下不会出现问题或者死锁：执行了某个元素的第一步处理后必须在执行另一个元素的第一步处理之前执行这个元素接下来的处理步骤。

- 使用par_unseq指定**并行化乱序执行**

  这意味着多个线程执行时不需要保证某一个线程在执行完某一个元素的处理之前不会切换到其他的元素。特别地，这允许矢量化执行，一个线程可以先执行完多个元素的第一步处理，然后再执行下一步处理。

  并行化乱序执行需要编译器/硬件的特殊支持来检测哪些操作如何矢量化。[1]

所有的执行策略都是定义在命名空间std中的新类（sequenced_policy、parallel_policy、parallel_unsequenced_policy）的constexpr对象。还提供了新的类型特征std::is_execution_policy<>来在泛型编程中检查模板参数是否是执行策略。

22.3 异常处理

当处理元素的函数因为未捕获的异常而退出时所有的并行算法会调用std::terminate()。

注意即使选择了顺序执行策略也会这样。如果觉得这样不能接受，使用非并行化版本的算法将是更好的选择。

[1]例如，如果一个CPU支持512位的寄存器，编译器可能同时加载8个64的值或者4个128位的值到寄存器里然后对这些值同时进行计算。

注意并行算法本身也可能抛出异常。如果它们申请并行执行所需的临时内存资源时失败了，可能会抛出 `std::bad_alloc` 异常。然而，不会有别的异常被抛出。

22.4 不使用并行算法的好处

既然已经有了并行算法并且还可以给并行算法指定顺序执行的策略，那么我们是否还需要原本的非并行算法呢？

事实上，除了向后的兼容性之外，使用非并行算法还可以提供以下好处：

- 可以使用输入和输出迭代器。
- 算法不会在遇到异常时 `terminate()`。
- 算法可以避免因为意外使用元素导致的副作用。
- 算法可以提供额外的功能，例如 `for_each()` 会返回传入的可调用对象，我们可能会需要该对象最终的状态。

22.5 并行算法概述

表无限制的并行 STL 算法列出了标准中支持的所有不需要修改就可以并行运行的算法。

无限制的并行算法
`find_end()`, `adjacent_find()`
`search()`, `search_n()`（和“搜索器”一起使用时除外）
`swap_ranges()`
`replace()`, `replace_if()`
`fill()`
`generate()`
`remove()`, `remove_if()`, `unique()`
`reverse()`, `rotate()`
`partition()`, `stable_partition()`
`sort()`, `stable_sort()`, `partial_sort()`
`is_sorted()`, `is_sorted_until()`
`nth_element()`
`inplace_merge()`
`is_heap()`, `is_heap_until()`
`min_element()`, `max_element()`, `min_max_element()`

Table 22.2: 无限制的并行 STL 算法

表无并行版本的 STL 算法列出了还不支持并行运行的算法：

注意对于 `accumulate()`、`partial_sum()`、`inner_product()`，提供了新的要求更宽松的并行算法来代替（如下所示）：

无并行版本的算法
accumulate()
partial_sum()
inner_product()
search()（和“搜索器”一起使用时）
copy_backward() move_backward()
sample(), shuffle()
partition_point()
lower_bound(), upper_bound(), equal_range()
binary_search()
is_permutation()
next_permutation(), prev_permutation()
push_heap(), pop_heap(), make_heap(), sort_heap()

Table 22.3: 无并行版本的 STL 算法

- 为了并行地运行 accumulate()，使用reduce() 或者transform_reduce()。
- 为了并行地运行 partial_sum()，使用...scan() 算法。
- 为了并行地运行 inner_product()，使用transform_reduce()。

表有限制的并行 STL 算法列出了标准支持的所有只需要进行一些修改就可以并行运行的算法。

22.6 并行编程的新算法的动机

C++17 还引入了一些补充的算法来实现那些从 C++98 就可用的标准算法的并行执行。

22.6.1 reduce()

例如，reduce() 作为 accumulate() 的并行版本引入，后者会“累积”所有的元素（你可以自定义“累积”的具体行为）。例如，考虑下面对 accumulate() 的使用：

lib/accumulate.cpp

```
#include <iostream>
#include <vector>
#include <numeric>  // for accumulate()

void printSum(long num)
{
    // 用数字序列1 2 3 4创建coll：
    std::vector<long> coll;
    coll.reserve(num * 4);
    for (long i = 0; i < num; ++i) {
```

```
        coll.insert(coll.end(), {1, 2, 3, 4});
    }

    auto sum = std::accumulate(coll.begin(), coll.end(), 0L);
    std::cout << "accumulate(): " << sum << '\n';
}

int main()
{
    printSum(1);
    printSum(1000);
    printSum(1000000);
    printSum(10000000);
}
```

我们计算出了所有元素的和，输出为：

```
accumulate(): 10
accumulate(): 10000
accumulate(): 10000000
accumulate(): 100000000
```

可结合可交换操作的并行化

上文的示例程序可以替换成 `reduce()` 来实现并行化：

lib/parreduce.cpp

```
#include <iostream>
#include <vector>
#include <numeric>  // for reduce()
#include <execution>

void printSum(long num)
{
    // 用数字序列1 2 3 4创建coll：
    std::vector<long> coll;
    coll.reserve(num * 4);
    for (long i = 0; i < num; ++i) {
        coll.insert(coll.end(), {1, 2, 3, 4});
    }

    auto sum = std::reduce(std::execution::par, coll.begin(), coll.end(), 0L);
    std::cout << "reduce():     " << sum << '\n';
}

int main()
{
    printSum(1);
```

```
    printSum(1000);
    printSum(1000000);
    printSum(10000000);
}
```

输出基本相同，程序可能会变得更快也可能会变得更慢（根据是否支持启动多个线程、启动线程的时间大于还是小于并行运行节省的时间）。

这里使用的操作是 +，这个操作是可结合可交换的，因为整型元素相加的顺序并不会影响结果。

不可交换操作的并行化

然而，对于浮点数来说相加的顺序不同结果也可能不同，像下面的程序演示的一样：

lib/parreducefloat.cpp

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

void printSum(long num)
{
    // 用数字序列0.1 0.3 0.0001创建coll：
    std::vector<double> coll;
    coll.reserve(num * 4);
    for (long i = 0; i < num; ++i) {
        coll.insert(coll.end(), {0.1, 0.3, 0.00001});
    }

    auto sum1 = std::accumulate(coll.begin(), coll.end(), 0.0);
    std::cout << "accumulate(): " << sum1 << '\n';
    auto sum2 = std::reduce(std::execution::par, coll.begin(), coll.end(), 0.0);
    std::cout << "reduce():     " << sum2 << '\n';
    std::cout << (sum1==sum2 ? "equal\n" : "differ\n");
}

#include <iomanip>

int main()
{
    std::cout << std::setprecision(5);
    // 译者注：此处原文是
    // std::cout << std::setprecision(20);
    // 应是作者笔误

    printSum(1);
    printSum(1000);
    printSum(1000000);
```

```
    printSum(10000000);
}
```

这里我们同时使用了 `accumulate()` 和 `reduce()` 来计算结果。一个可能的输出是：

```
accumulate(): 0.40001
reduce():     0.40001
equal
accumulate(): 400.01
reduce():     400.01
differ
accumulate(): 400010
reduce():     400010
differ
accumulate(): 4.0001e+06
reduce():     4.0001e+06
differ
```

尽管结果看起来一样，但实际上可能是不同的。这是用不同顺序相加浮点数的可能导致的结果。

如果我们改变输出的浮点数精度：

```
std::cout << std::setprecision(20);
```

我们可以看到下面的结果有一些不同：

```
accumulate(): 0.40001000000000003221
reduce():     0.40001000000000003221
equal
accumulate(): 400.01000000000533419
reduce():     400.01000000000010459
differ
accumulate(): 400009.99999085225863
reduce():     400009.9999999878346
differ
accumulate(): 4000100.0004483023658
reduce():     4000100.0000019222498
```

因为没有是否使用、何时使用、怎么使用并行算法的保证，某些平台上的结果可能看起来是相同的（当元素数量不超过一定值时）。

要想了解更多关于 `reduce()` 的细节，可以参见它的参考小节。

不可结合操作的并行化

让我们把操作改为累积时加上每个值的平方：

lib/accumulate2.cpp

```
#include <iostream>
#include <vector>
#include <numeric>  // for accumulate()
```

```
void printSum(long num)
{
    // 用数字序列1 2 3 4创建coll
    std::vector<long> coll;
    coll.reserve(num * 4);
    for (long i = 0; i < num; ++i) {
        coll.insert(coll.end(), {1, 2, 3, 4});
    }

    auto squaredSum = [] (auto sum, auto val) {
                        return sum + val * val;
                      };

    auto sum = std::accumulate(coll.begin(), coll.end(), 0L, squaredSum);
    std::cout << "accumulate(): " << sum << '\n';
}

int main()
{
    printSum(1);
    printSum(1000);
    printSum(1000000);
    printSum(10000000);
}
```

这里，我们传递了一个 lambda，把没新的元素的值的平方加到当前的和上：

```
auto squaredSum = [] (auto sum, auto val) {
                    return sum + val * val;
                  };
```

使用 `accumulate()` 的输出将是：

```
accumulate(): 30
accumulate(): 30000
accumulate(): 30000000
accumulate(): 300000000
```

然而，让我们切换到 `reduce()` 并行执行：

lib/parreduce2.cpp

```
#include <iostream>
#include <vector>
#include <numeric>  // for reduce()
#include <execution>

void printSum(long num)
{
    // 用数字序列1 2 3 4创建coll：
    std::vector<long> coll;
```

```
    coll.reserve(num * 4);
    for (long i = 0; i < num; ++i) {
        coll.insert(coll.end(), {1, 2, 3, 4});
    }

    auto squaredSum = [] (auto sum, auto val) {
                        return sum + val * val;
                      };

    auto sum = std::reduce(std::execution::par, coll.begin(), coll.end(), 0L, squaredSum);
    std::cout << "reduce():     " << sum << '\n';
}

int main()
{
    printSum(1);
    printSum(1000);
    printSum(1000000);
    printSum(10000000);
}
```

输出可能是这样的：

```
reduce():     30
reduce():     30000
reduce():     -425251612
reduce():     705991074
```

没错，结果有时有可能是错的。问题在于这个操作是不可结合的。（译者注：此处可结合指的是最后的结果与结合的方式无关（即满足结合律的），即如果`op(arg1, op(arg2, arg3))`和`op(op(arg1, arg2), arg3)`的结果相同，则`op`是可结合的。）如果我们并行的将这个操作应用于元素 1、2、3，那么我们可能会首先计算`0+1*1`和`2+3*3`，但是当我们组合中间结果时，我们把后者作为了第二个参数，实质上是计算了：

```
(0+1*1) + (2+3*3) * (2+3*3)
```

但是为什么这里的结果有时候是正确的呢？好吧，看起来是在这个平台上，只有当元素达到一定数量时`reduce()`才会并行运行。而这个行为是完全符合标准的。因此，需要像这里一样使用足够多的元素作为测试用例。

解决这个问题的方法是使用另一个新算法`transform_reduce()`。它把我们对每一个元素的操作（这个过程可以并行化）和可交换的结果的累加（这个过程也可以并行化）分离开来。

lib/partransformreduce.cpp

```
#include <iostream>
#include <vector>
#include <numeric>  // for transform_reduce()
#include <execution>
#include <functional>
```

```
void printSum(long num)
{
   // 用数字序列1 2 3 4创建coll：
   std::vector<long> coll;
   coll.reserve(num * 4);
   for (long i = 0; i < num; ++i) {
       coll.insert(coll.end(), {1, 2, 3, 4});
   }

   auto sum = std::transform_reduce(std::execution::par, coll.begin(), coll.end(),
                                    0L, std::plus{},
                                    [] (auto val) {
                                        return val * val;
                                    });
   std::cout << "transform_reduce(): " << sum << '\n';
}

int main()
{
    printSum(1);
    printSum(1000);
    printSum(1000000);
    printSum(10000000);
}
```

当调用 `transform_reduce()` 时，我们传递了

- 允许并行执行的执行策略
- 要处理的值范围
- 外层累积的初始值 `0L`
- 外层累积的操作 `+`
- 在累加之前处理每个值的 lambda

`transform_reduce()` 可能是到目前为止最重要的并行算法，因为我们经常在组合值之前先修改它们（也被称为 *map reduce* 原理）。

更多关于 `transform_reduce()` 的细节，参见它的参考小节。

使用 `transform_reduce` 进行文件系统操作

这里有另一个并行运行 `transform_reduce()` 的例子：

lib/dirsize.cpp

```
#include <vector>
#include <iostream>
#include <numeric>      // for transform_reduce()
#include <execution>    // for执行策略
#include <filesystem>   // 文件系统库
#include <cstdlib>      // for EXIT_FAILURE
```

```
int main(int argc, char *argv[]) {
    // 根目录作为命令行参数传递
    if (argc < 2) {
        std::cout << "Usage: " << argv[0] << " <path> \n";
        return EXIT_FAILURE;
    }
    std::filesystem::path root{argv[1]};

    // 初始化文件树中所有文件路径的列表：
    std::vector <std::filesystem::path> paths;
    try {
        std::filesystem::recursive_directory_iterator dirpos{root};
        std::copy(begin(dirpos), end(dirpos), std::back_inserter(paths));
    }
    catch (const std::exception &e) {
        std::cerr << "EXCEPTION: " << e.what() << std::endl;
        return EXIT_FAILURE;
    }

    // 累积所有普通文件的大小：
    auto sz = std::transform_reduce(
                    std::execution::par,                    // 并行执行
                    paths.cbegin(), paths.cend(),           // 范围
                    std::uintmax_t{0},                      // 初始值
                    std::plus<>,                            // 累加...
                    [](const std::filesystem::path& p) {    // 如果是普通文件返回文件大小
                        return is_regular_file(p) ? file_size(p) : std::uintmax_t{0};
                    });
    std::cout << "size of all " << paths.size()
              << " regular files: " << sz << '\n';
}
```

首先，我们递归收集了命令行参数中给出的目录中的所有文件的文件系统路径：

```
std::filesystem::path root{argv[1]};

std::vector<std::filesystem::path> paths;
std::filesystem::recursive_directory_iterator dirpos{root};
std::copy(begin(dirpos), end(dirpos), std::back_inserter(paths));
```

注意因为我们可能会传递无效的路径，可能会有（文件系统）异常抛出并被处理。

之后，我们遍历文件系统路径的集合来累加普通文件的大小：

```
auto sz = std::transform_reduce(
                std::execution::par,                    // 并行执行
                paths.cbegin(), paths.cend(),           // 范围
                std::uintmax_t{0},                      // 初始值
                std::plus<>,                            // 累加...
```

```
[](const std::filesystem::path& p) {    // 如果是普通文件返回文件大小
    return is_regular_file(p) ? file_size(p) : std::uintmax_t{0};
});
```

新的标准算法`transform_reduce()`以如下方式执行：

- 最后一个参数会作用于每一个元素。这里，传入的lambda会对每个元素调用，如果是常规文件的话会返回它的大小。
- 倒数第二个参数用来把所有大小组合起来。因为我们想要累加大小，所以使用了标准函数对象`std::plus<>`。
- 倒数第三个参数是组合操作的初始值。这个值的类型也是整个算法的返回值类型。我们把`0`转换为`file_size()`的返回值类型`std::uintmax_t`。否则，如果我们简单的传递一个`0`，我们会得到窄化为`int`类型的结果。因此，如果路径集合是空的，那么整个调用会返回`0`。

注意查询一个文件的大小是一个开销很大的操作，因为它需要操作系统调用。因此，使用并行算法来通过多个线程并行调用这个转换函数（从路径到大小的转换）并计算大小的总和会快很多。之前第一个测试结果就已经显示出并行算法的明显优势了（最多能快两倍）。

注意你不能直接向并行算法传递指定目录的迭代器，因为目录迭代器是输入迭代器，而并行算法要求至少是前向迭代器。

最后，注意`transform_reduce()`定义在头文件`numeric`而不是`algorithm`（和`accumulate()`一样，后者作为数值算法来进行计算）。

22.7 后记

并行STL算法由Jared Hoberock、Michael Garland、Olivier Giroux、Vinod Grover、Ujval Kapasi、Jaydeep Marathe于2012年在https://wg21.link/n3408中首次提出。当时它成为了一个正式的测试标准：*Technical Specification for C++ Extensions for Parallelism*（见https://wg21.link/n3850）。Jared Hoberock、Grant Mercer、Agustin Berge、Harmut Kaiser又在https://wg21.link/n4276中添加了额外的算法。最终因Jared Hoberock在https://wg21.link/p0024r2中的提案，*Technical Specification for C++ Extensions for Parallelism*被标准库接纳。关于异常处理，JF Bastien和Bryce Adelstein Lelbach在https://wg21.link/p0394r4中的提案最终被接受。

算法	限制
for_each()	返回类型 void、前向迭代器
for_each_n()	前向迭代器（新）
all_of(), and_of(), none_of()	前向迭代器
find(), find_if(), find_if_not()	前向迭代器
find_first_of()	前向迭代器
count(), count_if()	前向迭代器
mismatch()	前向迭代器
equal()	前向迭代器
is_partitioned()	前向迭代器
partial_sort_copy()	前向迭代器
includes()	前向迭代器
lexicographical_compare()	前向迭代器
fill_n()	前向迭代器
generate_n()	前向迭代器
reverse_copy()	前向迭代器
rotate_copy()	前向迭代器
copy(), copy_n(), copy_if()	前向迭代器
move()	前向迭代器
transform()	前向迭代器
replace_copy(), replace_copy_if()	前向迭代器
remove_copy(), remove_copy_if()	前向迭代器
unique_copy()	前向迭代器
partition_copy()	前向迭代器
merge()	前向迭代器
set_union(), set_intersection()	前向迭代器
set_differrnce(), set_symmetric_difference()	前向迭代器
inclusive_scan(), exclusive_scan()	前向迭代器（新）
transform_inclusive_scan(), transform_exclusive_scan()	前向迭代器（新）

Table 22.4: 有限制的并行 STL 算法

Chapter 23

新的 STL 算法详解

23.1 `std::for_each_n()`

23.2 新的数学 STL 算法

23.2.1 `std::reduce()`

23.2.2 `std::transform_reduce()`

单范围的 `std::transform_reduce()`

双范围的 `std::transform_reduce()`

23.2.3 `std::inclusive_scan()` 和 `std::exclusive_scan()`

23.2.4 `std::transform_inclusive_scan()` 和 `std::transform_exclusive_scan()`

23.3 后记

Chapter 24

子串和子序列搜索器

24.1 使用子串搜索器

24.1.1 通过 `search()` 使用搜索器

24.1.2 直接使用搜索器

Chapter 25

其他工具函数和算法

25.1 `size()`，`empty()`，`data()`

25.1.1 泛型`size()`函数

25.1.2 泛型`empty()`函数

25.1.3 泛型`data()`函数

25.2 `as_const()`

25.2.1 以常量引用捕获

Chapter 26

容器和字符串扩展

Chapter 27

多线程和并发

Chapter 28

标准库的其他微小特性和修改

28.1 `std::uncaught_exceptions()`

28.2 共享指针改进

28.2.1 对原始 C 数组的共享指针的特殊处理

28.2.2 共享指针的 `reinterpret_pointer_cast`

28.2.3 共享指针的 `weak_type`

28.2.4 共享指针的 `weak_from_this`

28.3 数学扩展

28.3.1 最大公约数和最小公倍数

28.3.2 `std::hypot()` 的三参数重载

28.3.3 数学领域的特殊函数

28.3.4 `chrono` 扩展

28.3.5 `constexpr` 扩展和修正

28.3.6 `noexcept` 扩展和修正

28.3.7 后记

Part V

专家的工具

这一部分介绍了普通应用程序员通常不需要知道的新的语言特性和库。它主要包括了为编写基础库和语言特性的程序员准备的用来解决特殊问题语言特性（例如修改了堆内存的管理方式）。

Chapter 29

多态内存资源 (PMR)

Chapter 30

使用 new 和 delete 管理超对齐数据

Chapter 31

`std::to_chars()`和 `std::from_chars()`

31.1 字符序列和数字值之间的底层转换的动机

31.2 使用示例

31.2.1 `from_chars`

31.2.2 `to_chars()`

Chapter 32

`std::launder()`

Chapter 33

编写泛型代码的改进

33.1 `std::invoke<>()`

33.2 `std::bool_constant<>`

Part VI

一些通用的提示

这一部分介绍了一些有关C++17的通用的提示，例如对C语言和废弃特性的兼容性更改。

Chapter 34

常见的 C++17 事项

Chapter 35

废弃和移除的特性