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。

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（有些观点称它们为*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分别是返回结构体中的i和s成员
```

这里，id和val分别是返回结构体中的i和s成员。它们的类型分别对应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，另外通过把值绑定到能体现语义的变量名可以使代码的可读性更强。[1]

[1]感谢Zachary Turner指出这一点

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';
}
```

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;
```

其实等价于我们用 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;
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
```

无论是 u 还是 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];
};
```

[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

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。因此打印它们的值是没有问题的，但不要对打印出的值做任何假设。[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：

```
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"
```

[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

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的示例。

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。对于传递进入的数组参数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non-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，你通常不能一起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 *tuple-like API* 为任何类型添加对结构化绑定的支持，就像标准库中为 `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：

- 顾客的姓是 `std::string` 类型
- 顾客的名是 `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 相同的命名空间定义了模板函数 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(); }
```

[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

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：

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;
    }
```

[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

```
    long value() const {
        return val;
    }
    long& value() {
        return val;
    }
};
```

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：

[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。

```
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;
    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和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。最终被接受的正式提案由Jens Maurer发表于https://wg21.link/p0217r3。

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：

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

这里初始化语句：

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

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 语句中生效，包括 *then* 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";
```

```
}
```

作为另一个例子，考虑向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

带初始化语句的 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。最终被接受的正式提案由 Thomas Köpped 发表于https://wg21.link/p0305r1。

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"};    // 自从C++17起OK
    ...
};

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
```

问题不在于头文件可能被重复包含多次，问题在于两个不同的CPP文件都包含了这个头文件，因此都定义了`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（自从C++14）：

```
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。
- 它必须在使用它的每个翻译单元中定义

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)只需要获取
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。[8] 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`:

```
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`你可以为每个县城创建一个内联变量：

```
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
```

[8] 感谢 Richard Smith 指出这一点。

```
    inline static thread_local std::string tName = "tls";
// 每个线程有一个
    std::string lName = "local";    // 每一个实例有一个
    ...
    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，由 Hal Finkel 和 Richard Smith 在https://wg21.link/n4424中第一次提出。最终被接受的正式提案由 Hal Finkel 和 Richard Smith 发表于https://wg21.link/p0386r2。

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[9]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：

[9]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，此外在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trait `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）。

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)。

我将会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（译者：???）或成员的表达式
- 只含有字符串字面量的表达式（译者：???）
- 内建的一元 * 运算符（解引用运算符）的结果

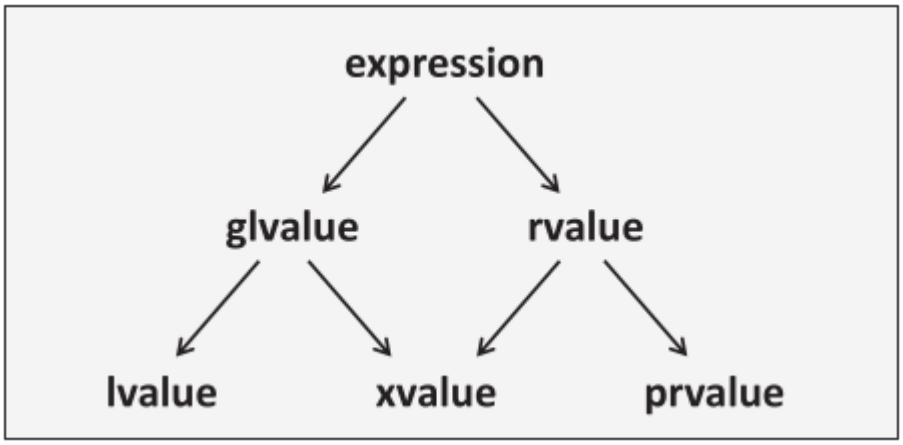

Figure 5.1: 从 C++11 起的值类型体系

- 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：

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

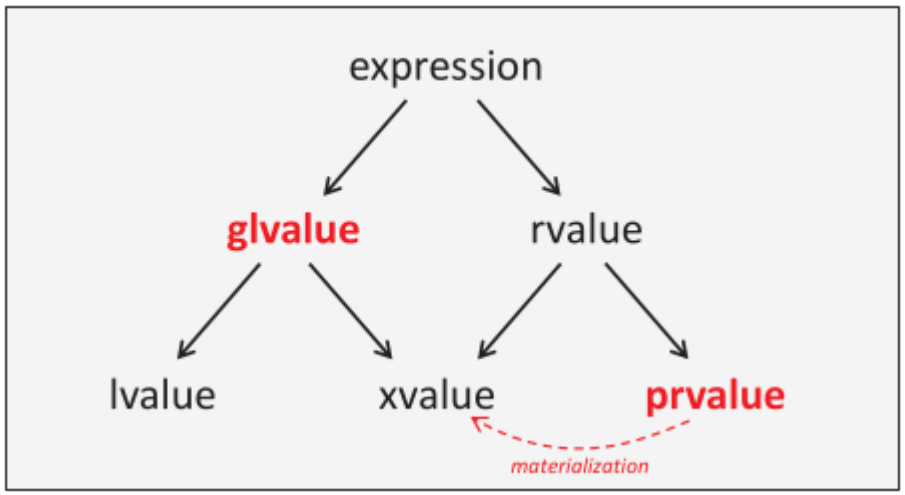

Figure 5.2: 自从 C++17 起的值类型体系

- **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。[10]

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{...}
}
```

注意当用来初始化的表达式（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。

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

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。

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：[11]

```
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;
    }
};
```

[11]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。

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;
};
```

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：[12]

```
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;
```

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

```
};
```

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")};
```

你甚至可以在hashed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 最早由 Faisal Vali、Ville Voutilainen、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。

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，如果对realloc()标记[[nodiscard]]将会适得其反。

- 一个因为没有使用返回值而导致函数行为和预期不同的很好的例子是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：[13]

```
[[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]];
        case 2:
            std::cout << "well\n";
            break;
```

[13] 感谢 Roland Bock 指出这一点

```
            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的特性由 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 和 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。

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`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。[14]

[14]较旧版本的GCC或者Visual C++的行为就是这样的。

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：

```
// 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）。[15] 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：

[15] 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：[16]

```
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
    ...
};
```

[16]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。

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：

- 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();
}
```

现在如果你想要重载全部可能的函数类型，你需要重载原来两倍的数量。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之前该特征共共有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 trait 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和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和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是由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()最早由Clark Nelson和Richard Smith作为https://wg21.link/p0061r0 的部分内容提出。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。

9 类模板参数推导

9.1 使用类模板参数推导

9.1.1 默认以拷贝方式推导

9.1.2 推导 lambda 的类型

9.1.3 没有类模板部分参数推导

9.1.4 使用类模板参数推导代替快捷函数

9.2 推导指引

9.2.1 使用推导指引强制类型退化

9.2.2 非模板推导指引

9.2.3 推导指引与构造函数冲突

9.2.4 `explicit` 推导指引

9.2.5 聚合体的推导指引

9.2.6 标准推导指引

pair 和 tuple 的推导指引

从迭代器推导

`std::array<>` 推导

(Unordered) Map 推导

智能指针没有推导指引

9.3 后记

10　编译期`if`语句

11 折叠表达式

11.1 折叠表达式的动机

11.2 使用折叠表达式

11.2.1 处理空参数包

12 扩展的 using 声明

12.1 使用变长的 using 声明

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。

13 `std::byte`

14 文件系统库

14.1 基本的例子

14.1.1 打印文件系统路径类的属性

14.1.2 用 `switch` 语句处理不同的文件系统类型

14.1.3 创建不同类型的文件

14.1.4 使用并行算法处理文件系统

14.2 原则和术语

14.2.1 通用的可移植性路径分隔符

14.2.2 命名空间

14.2.3 文件系统路径

15 类型 trait 扩展

15.1 类型 trait 后缀 `_v`

15.2 新的类型 trait

类型 trait `is_aggregate<>`

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

16 子串和子序列搜索器

16.1 使用子串搜索器

16.1.1 通过 `search()` 使用搜索器

16.1.2 直接使用搜索器

17 其他工具函数和算法

17.1 size()，empty()，data()

17.1.1 泛型size()函数

17.1.2 泛型empty()函数

17.1.3 泛型data()函数

17.2 as_const()

17.2.1 以常量引用捕获

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）。

18 使用 new 和 delete 管理超对齐数据

18.1 使用带有对齐的 new 运算符

18.2 实现内存对齐分配的 new() 运算符

18.2.1 在 C++17 之前实现对齐的内存分配

18.2.2 实现类型特化的 new() 运算符

18.3 实现全局的 new() 运算符

18.4 追踪所有 ::new 调用

18.5 后记

19 编写泛型代码的改进

19.1 `std::invoke<>()`

19.2 `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。