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（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能带来怎样的显著改进。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';
}
```

[1]感谢 Zachary Turner 指出这一点

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;
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 *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：

[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

```
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：

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]：

[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

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;
    }
};
```

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);
```

[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(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、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。

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。

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
```

[1] 感谢 Richard Smith 指出这一点。

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，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。

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：

```
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};
```

[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，此外在C++17之前，还要求没有继承

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)。

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
- 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

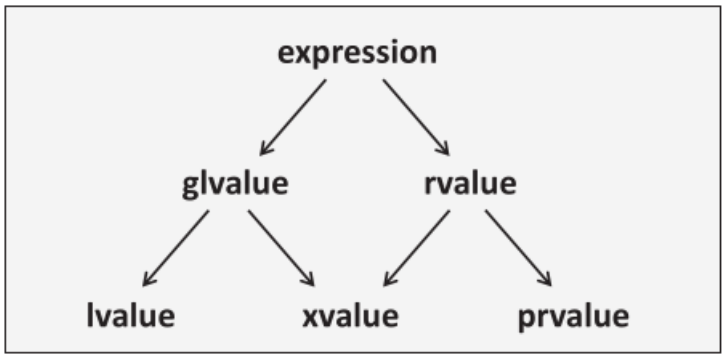

Figure 5.1: 从 C++11 起的值类型体系

- 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：

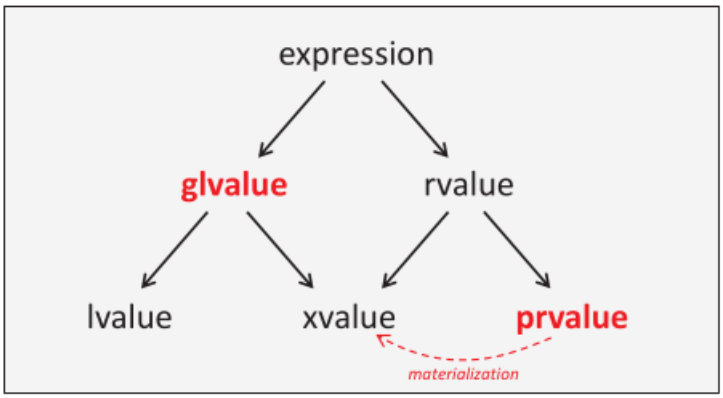

Figure 5.2: 自从C++17起的值类型体系

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{...}
}
```

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

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。

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;
};
```

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：

[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。

```
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")};
```

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：

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

```
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>
```

```cpp
#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 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。

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。

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

[1] 感谢 Roland Bock 指出这一点

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

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

[[fallthrough]] 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、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：

```
// 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
```

[1] 较旧版本的 GCC 或者 Visual C++ 的行为就是这样的。

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

[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。

- 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：

```
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。

[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。

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 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

嵌套命名空间定义最早由Jon Jagger于2003年在https://wg21.link/n1524中提出。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()`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

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。

[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。

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()

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 是 char[3]，参数 y 的类型是 const char(&)[6]，因此 T2 是 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：

```
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
```

[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。

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>
```

(Unordered) Map 推导

想让推导指引正常工作的复杂度可以通过给关联容器（map、multimap、unordered_map、unordered_multimap）定义推导指引来展示。

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：

[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）。

```
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` 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

类模板参数推导特性由 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。

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。

10.3 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。

[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。

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，

```
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) {
    ...
}
```

10.3.1 其他编译期`if`的示例

完美返回泛型值

编译期`if`的一个引用就是完美转发返回值，并在返回之前对返回值进行一些处理。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。

[3]感谢 Graham Haynes 和 Barry Revzin 指出这一点。

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。

10.4 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";
        ...
    }
    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`。

10.5 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]

10.6 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。

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), ...
}
```

[1] 感谢 Barry Revzin 指出这一点。

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)
{
    ...
    (... , (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：

[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。

```
    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 {
        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。

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。

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：

```
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）：

[1] 感谢 Bryce Adelstein Lelbach 提供这些例子。

```
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;
```

在每一个翻译单元中，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翻译单元中使用）。

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，我们可以在一个翻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()
{
    arr<int, 5>[0] = 17;
    arr<int, 5>[3] = 42;
    arr<int, 5u>[1] = 11;
    arr<int, 5u>[3] = 33;
    printArr();
}
```

另一个翻译单元内可以打印这两个变量模板：

tmpl/vartmplauto2.cpp

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

[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。

```
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）
```

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：

[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。

tmpl/decltypeauto.cpp

```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。

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} {
        ...
    }
    ...
};

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>;
```

[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。

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。

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，我们可能会用 `stoi()` 调用的结果也就是一个 `int` 来初始化返回值并返回，也可能会返回 `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<> 操作

表std::optional<> 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<> 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：

[2] 感谢 Tim Song 指出这一点。
[3] 感谢 Roland Bock 指出这一点。

```
std::multimap<std::string, std::string> englishToGerman;
...
auto pos = englishToGerman.find("wisdom");
auto o16 = pos != englishToGerman.end()
            ? std::optional{pos->second}
            : std::nullopt;
```

这里，根据类模板参数推导，依据 std::optionalpos->second 表达式能推导出 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，因为?: 运算符的两种可能必须有相同的类型。

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：

```
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，当没有内含值时将会抛出一个 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

[4] 感谢 Alexander Brockmöller 指出这一点。

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
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() &&;
        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。

[5] 感谢 Alexander Brockmöller 指出这一点。

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，并引用了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。

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构集合的方法。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。

[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。

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";
    }
};

std::variant<NoDefConstr, int> v1; // ERROR：不能默认构造第一个选项
```

[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。

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型：`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

16.3.2 `std::variant<>` 操作

构造函数

获取值

修改值

比较

move 语义

哈希

16.3.3 访问器

使用函数对象作为访问器

使用泛型 lambda 作为访问器

在访问器中返回值

使用重载的 lambda 作为访问器

17 std::byte

18 字符串视图 (String Views)

18.1 和 `std::string` 的不同之处

18.2 使用字符串视图

18.3 使用字符串视图作为参数

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";
        }
    }
}
```

19 文件系统库

19.1 基本的例子

19.1.1 打印文件系统路径类的属性

19.1.2 用 `switch` 语句处理不同的文件系统类型

19.1.3 创建不同类型的文件

19.1.4 使用并行算法处理文件系统

19.2 原则和术语

19.2.1 通用的可移植性路径分隔符

19.2.2 命名空间

19.2.3 文件系统路径

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

20 类型 trait 扩展

20.1 类型 trait 后缀 `_v`

20.2 新的类型 trait

类型 trait `is_aggregate<>`

21　并行 STL 算法

22 子串和子序列搜索器

22.1 使用子串搜索器

22.1.1 通过 `search()` 使用搜索器

22.1.2 直接使用搜索器

23 其他工具函数和算法

23.1 size()，empty()，data()

23.1.1 泛型size()函数

23.1.2 泛型empty()函数

23.1.3 泛型data()函数

23.2 as_const()

23.2.1 以常量引用捕获

24 标准库的其他微小的特性和修改

24.1 `std::uncaught_exceptions()`

24.2 共享指针改进

24.2.1 对原始C数组的共享指针的特殊处理

24.2.2 共享指针的`reinterpret_pointer_cast`

24.2.3 共享指针的`weak_type`

24.2.4 共享指针的`weak_from_this`

24.3 数学扩展

24.3.1 最大公约数和最小公倍数

24.3.2 `std::hypot()`的三参数重载

24.3.3 数学领域的特殊函数

24.3.4 `chrono`扩展

24.3.5 `constexpr`扩展和修正

24.3.6 `noexcept`扩展和修正

24.3.7 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）。

25 使用 new 和 delete 管理超对齐数据

25.1 使用带有对齐的 new 运算符

25.2 实现内存对齐分配的 new() 运算符

25.2.1 在 C++17 之前实现对齐的内存分配

25.2.2 实现类型特化的 new() 运算符

25.3 实现全局的 new() 运算符

25.4 追踪所有 ::new 调用

25.5 后记

26 `std::to_chars()`和`std::from_chars()`

26.1 字符序列和数字值之间的底层转换的动机

26.2 使用示例

26.2.1 `from_chars`

27 编写泛型代码的改进

27.1 `std::invoke<>()`

27.2 `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。