Contents

I 基本语言特性 2

1 结构化绑定 3

1.1 细说结构化绑定 4

1.2 结构化绑定的适用场景 8

1.2.1 结构体和类 9

1.2.2 原生数组 10

1.2.3 `std::pair`, `std::tuple` 和 `std::array` 10

1.3 为结构化绑定提供 Tuple-Like API 13

1.4 后记 22

2 带初始化的 `if` 和 `switch` 语句 23

II 模板特性 24

3 编译期 `if` 语句 25

III 新的标准库组件 26

IV 已有标准库的拓展和修改 26

4 子串和子序列搜索器 27

4.1 使用子串搜索器 27

4.1.1 通过 `search()` 使用搜索器 27

4.1.2 直接使用搜索器 27

V 专家的工具 28

VI 一些通用的提示 28

Part I

基本语言特性

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

专为模板编程设计的新的核心语言特性在Part II中介绍。

1 结构化绑定

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

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

MyStruct ms;
```

你可以通过如下的声明直接把该结构体的两个成员绑定到新的变量名：

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

这里，变量u和v的声明方式称为*结构化绑定*。某种程度上可以说它们解构了用来初始化的对象（有些观点称它们为*解构声明*）。

如下的每一种声明方式都是支持的：

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

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

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

你可以直接把返回的数据成员赋值给两个新的局部变量：

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

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

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

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

[1]感谢Zachary Turner指出这一点

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

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

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

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

上面的例子中我们可以使用准确体现语义的变量名直接访问每一个元素。

1.1 细说结构化绑定

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

绑定到一个匿名实体

如下代码的精确行为：

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

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

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

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

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

会打印出e.i和e.s（分别是ms.i和ms.s的拷贝）。

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

```
MyStruct ms{42, "hello"};
auto [u, v] = ms;
ms.i = 77;
std::cout << u;     // 打印出42
u = 99;
std::cout << ms.i;  // 打印出77
```

在这个例子中u和ms.i有不同的内存地址。

当使用结构化绑定来绑定返回值时，规则是相同的。如下初始化

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

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

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

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

使用修饰符

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

例如，我们可以把声明一个结构化绑定声明为const引用：

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

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

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

如果声明为非 const 引用，你甚至可以修改对象的成员：

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

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

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

修饰符并不是作用在结构化绑定引入的变量上

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

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

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

当声明对齐时也是类似：

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

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

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

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

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

那么如下声明之后：

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

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

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

move语义

move语义也遵循之前介绍的规则，如下声明：

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

这里v和n指向的匿名实体是ms的右值引用。同时ms的值仍然保持不变：

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

然而，你可以对指向ms.s的n进行移动赋值：

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

像通常一样值被移动走的对象处于一个值未定义但却有效的状态。因此打印它们的值是没有问题的，但不要对打印出的值做任何假设[3]。

上面的例子和直接用ms被移动走的值进行结构化绑定有些不同：

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

这里新的匿名实体是用ms被移动走的值来初始化的。因此，ms已经失去了值：

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

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

你可以继续用n进行移动赋值或者给n赋予新值，但已经不会再影响到ms.s了：

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

1.2 结构化绑定的适用场景

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

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

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

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

在任何情况下，结构化绑定中声明的变量名的数量必须和元素或数据成员的数量相同。你不能跳过某个元素，也不能重复使用变量名。然而，你可以使用非常短的名称例如`'_'`（有的程序员喜欢这个名字，有的讨厌它，

但注意全局命名空间不允许使用它），但这个名字在同一个作用域只能使用一次：

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

目前还不支持嵌套化的结构化绑定。

下一小节将详细讨论结构化绑定的使用。

1.2.1 结构体和类

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

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

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

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

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

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

联合还不支持使用结构化绑定。

1.2.2 原生数组

下面的代码用C风格数组的两个元素初始化了x和y：

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

注意这是C++中少数几种原生数组会按值拷贝的场景之一。

只有当数组的长度已知时才可以使用结构化绑定。对于传递进入的数组参数不能使用结构化绑定，因为数组会*退化 (decay)* 为相应的指针类型。

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

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

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

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

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

`std::array`

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

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

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

使用非临时变量的non-const引用进行绑定，还可以进行修改操作。例如：

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

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

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

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

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

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

std::tuple

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

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

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

std::pair

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

```
std::map<std::string, int> coll;
auto ret = coll.insert({"new", 42});
```

```
if (!ret.second) {
    // 如果插入失败，使用ret.first处理错误
    ...
}
```

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

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

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

为 pair 和 tuple 的结构化绑定赋予新值

在声明了一个结构化绑定之后，你通常不能一起修改所有绑定的变量。因为结构化绑定只能一起声明但不能一起使用。然而你可以使用 `std::tie()` 把值一起赋给所有变量。例如：

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

这种方法可以被用来处理返回多个值的循环，例如在循环中使用搜索器：

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

1.3 为结构化绑定提供 Tuple-Like API

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

支持只读结构化绑定

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

lang/customer1.hpp

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

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

我们可以用如下代码添加 tuple-like API：

lang/structbind1.hpp

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

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

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

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

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

- 顾客的姓是 `std::string` 类型
- 顾客的名是 `std::string` 类型
- 顾客的消费金额是 `long` 类型

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

```
template<>
struct std::tuple_size<Customer> {
    static constexpr int value = 3; // 我们有3个属性
};
```

属性的类型被定义为 std::tuple_element 的特化版本：

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

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

最后在和类 Customer 相同的命名空间定义了模板函数 get<>() 的重载版本作为 getter[4]：

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

在这种情况下，我们有一个主函数模板的声明和针对所有情况的全特化版本。

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

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

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

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

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

有了这个 API，我们就可以为类型 `Customer` 使用结构化绑定：

lang/structbind1.cpp

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

int main()
{
    Customer c{"Tim", "Starr", 42};

    auto [f, l, v] = c;

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

如下初始化之后：

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

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

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

使用结构化绑定等同于使用 `get<>()` 函数的返回值，因此：

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

只是简单的输出变量的值（并不会再次调用 getter 函数）。另外

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

这段代码修改了绑定变量的值。因此，这段程序总是有如下输出：

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

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

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

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

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

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

支持可写结构化绑定

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

lang/customer2.hpp

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

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

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

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

为了支持读写，我们需要为常量和非常量引用定义重载的 getter：

lang/structbind2.hpp

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

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

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

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

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

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

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

[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。

如：

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

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

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

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

lang/structbind2.cpp

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

int main()
{
    Customer c{"Tim", "Starr", 42};
    auto [f, l, v] = c;
    std::cout << "f/l/v:    " << f << ' '
              << l << ' ' << v << '\n';
```

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

程序的输出如下：

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

1.4 后记

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

2　带初始化的 `if` 和 `switch` 语句

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。

3 编译期 `if` 语句

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。

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

4 子串和子序列搜索器

4.1 使用子串搜索器

4.1.1 通过 `search()` 使用搜索器

4.1.2 直接使用搜索器

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）。

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。