Item30：熟悉完美转发的失败case

C++11最显眼的功能之一就是完美转发功能。完美转发，太棒了！哎，开始使用，你就发现“完美”，理想与现实还是有差距。C++11的完美转发是非常好用，但是只有当你愿意忽略一些失败情况，这个Item就是使你熟悉这些情形。

在我们开始epsilon探索之前，有必要回顾一下“完美转发”的含义。“转发”仅表示将一个函数的参数传递给另一个函数。对于被传递的第二个函数目标是收到与第一个函数完全相同的对象。这就排除了按值传递参数，因为它们是原始调用者传入内容的副本。我们希望被转发的函数能够可以与原始函数一起使用对象。指着参数也被排除在外，因为我们不想强迫调用者传入指针。关于通用转发，我们将处理引用参数。

完美转发意味着我们不仅转发对象，我们还转发显著的特征：它们的类型，是左值还是右值，是const还是volatile。结合到我们会处理引用参数，这意味着我们将使用通用引用（参见Item24），因为通用引用参数被传入参数时才确定是左值还是右值。

假定我们有一些函数f，然后想编写一个转发给它的函数（就使用一个函数模板）。我们需要的核心看起来像是这样：

```
template<typename T>
void fwd(T&& param)        // accept any argument
{
  f(std::forward<T>(param));  // forward it to f
}
```

从本质上说，转发功能是通用的。例如fwd模板，接受任何类型的采纳数，并转发得到的任何参数。这种通用性的逻辑扩展是转发函数不仅是模板，而且是可变模板，因此可以接受任何数量的参数。fwd的可变个是如下：

```
template<typename... Ts>
void fwd(Ts&&... params)    // accept any arguments
{
  f(std::forward<Ts>(params)...); // forward them to f
}
```

这种形式你会在标准化容器emplace中（参见Item42）和只能容器的工厂函数`std::make_unique`和`std::make_shared`中（参见Item21）看到。

给定我们的目标函数f和被转发的函数fwd，如果f使用特定参数做一件事，但是fwd使用相同的参数做另一件事，完美转发就会失败：

```
f(expression); // if this does one thing
fwd(expression); // but this does something else, fwd fails to perfectly forward
expression to f
```

导致这种失败的原因有很多。知道它们是什么以及如何解决它们很重要，因此让我们来看看那种参数无法做到完美转发。

Braced initializers（支撑初始化器）

假定f这样声明：

```
void f(const std::vector<int>& v);
```

在这个例子中，通过列表初始化器，

```
f({1,2,3});  // fine "{1,2,3}" implicitly converted to std::vector<int>
```

但是传递相同的列表初始化器给fwd不能编译

```
fwd({1,2,3}); // error! doesn't compile
```

这是因为这是完美转发失效的一种情况。

所有这种错误有相同的原因。在对f的直接调用（例如f({1,2,3})），编译器看到传入的参数是声明中的类型。如果类型不匹配，就会执行隐式转换操作使得调用成功。在上面的例子中，从`{1,2,3}`生成了临时变量`std::vector<int>`对象，因此f的参数会绑定到`std::vector<int>`对象上。

当通过调用函数模板fwd调用f时，编译器不再比较传入给fwd的参数和f的声明中参数的类型。代替的是，推导传入给fwd的参数类型，然后比较推导后的参数类型和f的声明类型。当下面情况任何一个发生时，完美转发就会失败：

- **编译器不能推导出一个或者多个fwd的参数类型**，编译器就会报错
- **编译器将一个或者多个fwd的参数类型推导错误**。在这里，“错误”可能意味着fwd将无法使用推导出的类型进行编译，但是也可能意味着调用者f使用fwd的推导类型对比直接传入参数类型表现出不一致的行为。这种不同行为的原因可能是因为f的函数重载定义，并且由于是“不正确的”类型推导，在fwd内部调用f和直接调用f将重载不同的函数。

在上面的`f({1,2,3})`例子中，问题在于，如标准所言，将括号初始化器传递给未声明为`std::initializer_list`的函数模板参数，该标准规定为“非推导上下文”。简单来讲，这意味着编译器在对fwd的调用中推导表达式`{1,2,3}`的类型，因为fwd的参数没有声明为`std::initializer_list`。对于fwd参数的推导类型被阻止，编译器只能拒绝该调用。

有趣的是，Item2 说明了使用**braced initializer**的auto的变量初始化的类型推导是成功的。这种变量被视为`std::initializer_list`对象，在转发函数应推导为`std::initializer_list`类型的情况，这提供了一种简单的解决方法----使用auto声明一个局部变量，然后将局部变量转发：

```
auto il = {1,2,3}; // il's type deduced to be std::initializer_list<int>
fwd(il); // fine, perfect-forwards il to f
```

0或者NULL作为空指针

Item8说明当你试图传递0或者NULL作为空指针给模板时，类型推导会出错，推导为一个整数类型而不是指针类型。结果就是不管是0还是NULL都不能被完美转发为空指针。解决方法非常简单，使用nullptr就可以了，具体的细节，参考Item 8.

仅声明的整数静态const数据成员

通常，无需在类中定义整数静态const数据成员；声明就可以了。这是因为编译器会对此类成员

```
class Widget {
public:
  static const std::size_t MinVals = 28; // MinVal's declaration
  ...
};
...      // no defn. for MinVals
std::vector<int> widgetData;
widgetData.reserve(Widget::MinVals); // use of MinVals
```

这里，我们使用`Widget::MinVals`（或者简单点MinVals）来确定`widgetData`的初始容量，即使`MinVals`缺少定义。编译器通过将值28放入所有位置来补充缺少的定义。没有为`MinVals`的值留存储空间是没有问题的。如果要使用`MinVals`的地址（例如，有人创建了`MinVals`的指针），则`MinVals`需要存储（因为指针总要有一个地址），尽管上面的代码仍然可以编译，但是链接时就会报错，直到为`MinVals`提供定义。

按照这个思路，想象下f（转发参数给fwd的函数）这样声明：

```
void f(std::size_t val);
```

使用`MinVals`调用f是可以的，因为编译器直接将值28代替`MinVals`：

```
f(Widget::MinVals); // fine, treated as "28"
```

同样的，如果尝试通过fwd来调用f

```
fwd(Widget::MinVals); // error! shouldn't link
```

代码可以编译，但是不能链接。就像使用`MinVals`地址表现一样，确实，底层的问题是一样的。

尽管代码中没有使用`MinVals`的地址，但是fwd的参数是通用引用，而引用，在编译器生成的代码中，通常被视作指针。在程序的二进制底层代码中指针和引用是一样的。在这个水平下，引用只是可以自动取消引用的指针。在这种情况下，通过引用传递`MinVals`实际上与通过指针传递`MinVals`是一样的，因此，必须有内存使得指针可以指向。通过引用传递整型static const数据成员，必须定义它们，这个要求可能会造成完美转发失败，即使等效不使用完美转发的代码成功。（译者注：这里意思应该是没有定义，完美转发就会失败）

可能你也注意到了在上述讨论中我使用了一些模棱两可的词。代码“不应该”链接，引用“通常”被看做指针。传递整型static const数据成员“通常”要求定义。看起来就像有些事情我没有告诉你.....

确实，根据标准，通过引用传递`MinVals`要求有定义。但不是所有的实现都强制要求这一点。所以，取决于你的编译器和链接器，你可能发现你可以在未定义的情况使用完美转发，恭喜你，但是这不是那样做的理由。为了具有可移植性，只要给整型static const提供一个定义，比如这样：

```
const std::size_t Widget::MinVals; // in Widget's .cpp file
```

注意定义中不要重复初始化（这个例子中就是赋值28）。不要忽略这个细节，否则，编译器就会报错，提醒你只初始化一次。

重载的函数名称和模板名称

假定我们的函数f（通过fwd完美转发参数给f）可以通过向其传递执行某些功能的函数来定义其行为。假设这个函数参数和返回值都是整数，f声明就像这样：

```
void f(int (*pf)(int)); // pf = "process function"
```

值得注意的是，也可以使用更简单的非指针语法声明。这种声明就像这样，含义与上面是一样的：

```
void f(int pf(int)); // declares same f as above
```

无论哪种写法，我们都有了一个重载函数，processVal:

```
int processVal(int value);
int processVal(int value, int priority);
```

我们可以传递processVal给f

```
f(processVal); // fine
```

但是有一点要注意，f要求一个函数指针，但是`processVal`不是一个函数指针或者一个函数，它是两个同名的函数。但是，编译器可以知道它需要哪个：通过参数类型和数量来匹配。因此选择了一个int参数的`processVal`地址传递给f

工作的基本机制是让编译器帮选择f的声明选择一个需要的`processVal`。但是，fwd是一个函数模板，没有需要的类型信息，使得编译器不可能帮助自动匹配一个合适的函数：

```
fwd(processVal); // error! which processVal?
```

`processVal`没有类型信息，就不能类型推导，完美转发失败。

同样的问题会发生在如果我们试图使用函数模板代替重载的函数名。一个函数模板是未实例化的函数，表示一个函数族：

```
template<typename T>
T workOnVal(T param) { ... } // template for processing values
fwd(workOnVal); // error! which workOnVal instantiation ?
```

获得像fwd的完美转发接受一个重载函数名或者模板函数名的方式是指定转发的类型。比如，你可以创造与f相同参数类型的函数指针，通过processVal或者workOnVal实例化这个函数指针（可以引导生成代码时正确选择函数实例），然后传递指针给f:

```
using ProcessFuncType = int (*)(int); // make typedef; see Item 9
PorcessFuncType processValPtr = processVal; // specify needed signature for
processVal
fwd(processValPtr); // fine
fwd(static_cast<ProcessFuncType>(workOnVal)); // alse fine
```

当然，这要求你知道fwd转发的函数指针的类型。对于完美转发来说这一点并不合理，毕竟，完美转发被设计为转发任何内容，如果没有文档告诉你转发的类型，你如何知道？（译者注：这里应该想表达，这是解决重载函数名或者函数模板的解决方案，但是这是完美转发本身的问题）

位域

完美转发最后一种失败的情况是函数参数使用位域这种类型。为了更直观的解释，IPv4的头部可以如下定义：

```
struct IPv4Header {
  std::uint32_t version:4,
                IHL:4,
                DSCP:6,
                ECN:2,
                totalLength:16;
  ...
};
```

如果声明我们的函数f（转发函数fwd的目标）为接收一个`std::size_t`的参数，则使用IPv4Header对象的totalLength字段进行调用没有问题：

```
void f(std::size_t sz);
IPv4Header h;
...
f(h.totalLength);// fine
```

如果通过fwd转发h.totalLength给f呢，那就是一个不同的情况了：

```
fwd(h.totalLength); // error!
```

问题在于fwd的参数是引用，而h.totalLength是非常量位域。听起来并不是那么糟糕，但是C++标准非常清楚地谴责了这种组合：非常量引用不应该绑定到位域。禁止的理由很充分。位域可能包含了机器字节的任意部分（比如32位int的3-5位），但是无法直接定位。我之前提到了在硬件层面引用和指针时一样的，所以没有办法创建一个指向任意bit的指针（C++规定你可以指向的最小单位是char），所以就没有办法绑定引用到任意bit上。

一旦意识到接收位域作为参数的函数都将接收位域的副本，就可以轻松解决位域不能完美转发的问题。毕竟，没有函数可以绑定引用到位域，也没有函数可以接受指向位域的指针（不存在这种指针）。这种位域类型的参数只能按值传递，或者有趣的事，常量引用也可以。在按值传递时，被调用的函数接受了一个位域的副本，而且事实表明，位域的常量引用也是将其“复制”到普通对象再传递。

传递位域给完美转发的关键就是利用接收参数函数接受的是一个副本的事实。你可以自己创建副本然后利用副本调用完美转发。在IPv4Header的例子中，可以如下写法：

```
// copy bitfield value; see Item6 for info on init. form
auto length = static_cast<std::uint16_t>(h.totalLength);
fwd(length); // forward the copy
```

总结

在大多数情况下，完美转发工作的很好。你基本不用考虑其他问题。但是当其不工作时，当看起来合理的代码无法编译，或者更糟的是，无法按照预期运行时，了解完美转发的缺陷就很重要了。同样重要的是如何解决它们。在大多数情况下，都很简单

需要记住的事

- 完美转发会失败当模板类型推导失败或者推导类型错误
- 导致完美转发失败的类型有braced initializers，作为空指针的0或者NULL，只声明的整型static const数据成员，模板和重载的函数名，位域