Item 10:优先考虑限域枚举而非未限域枚举

条款10:优先考虑限域枚举而非未限域枚举

通常来说，在花括号中声明一个名字会限制它的作用域在花括号之内。但这对于C++98风格的enum中声明的枚举名是不成立的。这些在enum作用域中声明的枚举名所在的作用域也包括enum本身，也就是说这些枚举名和enum所在的作用域中声明的相同名字没有什么不同

```
enum Color { black, white, red };   // black, white, red 和
                                    // Color一样都在相同作用域
auto white = false;                 // 错误! white早已在这个作用
                                    // 域中存在
```

事实上这些枚举名泄漏进和它们所被定义的enum域一样的作用域。有一个官方的术语：未限域枚举(unscoped enum)在C++11中它们有一个相似物，限域枚举(scoped enum)，它不会导致枚举名泄漏：

```
enum class Color { black, white, red }; // black, white, red
                                        // 限制在Color域内
auto white = false;                     // 没问题，同样域内没有这个名字

Color c = white;                        //错误，这个域中没有white

Color c = Color::white;                 // 没问题
auto c = Color::white;                  // 也没问题（也符合条款5的建议）
```

因为限域枚举是通过**enum class**声明，所以它们有时候也被称为枚举类(enum classes)。

使用限域枚举减少命名空间污染是一个足够合理使用它而不是它的同胞未限域枚举的理由，其实限域枚举还有第二个吸引人的优点：在它的作用域中,枚举名是强类型。未限域枚举中的枚举名会隐式转换为整型（现在，也可以转换为浮点类型）。因此下面这种歪曲语义的做法也是完全有效的：

```
enum Color { black, white, red };       // 未限域枚举
std::vector<std::size_t>                // func返回x的质因子
primeFactors(std::size_t x);
Color c = red;
…
if (c < 14.5) {                         // Color与double比较 (!)
    auto factors =                      // 计算一个Color的质因子(!)
    primeFactors(c);
…
}
```

在**enum**后面写一个**class**就可以将非限域枚举转换为限域枚举，接下来就是完全不同的故事展开了。现在不存在任何隐式转换可以将限域枚举中的枚举名转化为任何其他类型。

```
enum class Color { black, white, red }; // Color现在是限域枚举
Color c = Color::red;                   // 和之前一样，只是
…                                       // 多了一个域修饰符
if (c < 14.5) {                         // 错误！不能比较
                                        // Color和double
    auto factors =                      // 错误！不能向参数为std::size_t的函数
        primeFactors(c);                // 传递Color参数
    …
}
```

如果你真的很想执行Color到其他类型的转换，和平常一样，使用正确的类型转换运算符扭曲类型系统：

```
if (static_cast<double>(c) < 14.5) { // 奇怪的代码，但是
                                     // 有效
    auto factors = // suspect, but
        primeFactors(static_cast<std::size_t>(c)); // 能通过编译
    …
}
```

似乎比起非限域枚举而言限域枚举有第三个好处，因为限域枚举可以前置声明。比如，它们可以不指定枚举名直接前向声明：

```
enum Color;         // 错误!
enum class Color;   // 没问题
```

其实这是一个误导。在C++11中，非限域枚举也可以被前置声明，但是只有在做一些其他工作后才能实现。这些工作来源于一个事实：
在C++中所有的枚举都有一个由编译器决定的整型的基础类型。对于非限域枚举比如`Color`，

```
enum Color { black, white, red };
```

编译器可能选择`char`作为基础类型，因为这里只需要表示三个值。然而，有些枚举中的枚举值范围可能会大些，比如：

```
enum Status { good = 0,
              failed = 1,
              incomplete = 100,
              corrupt = 200,
              indeterminate = 0xFFFFFFFF
              };
```

这里值的范围从**0**到**0xFFFFFFFF**。除了在不寻常的机器上（比如一个**char**至少有32bits的那种），编译器都会选择一个比**char**大的整型类型来表示**Status**。

为了高效使用内存，编译器通常在确保能包含所有枚举值的前提下为枚举选择一个最小的基础类型。在一些情况下，编译器
将会优化速度，舍弃大小，这种情况下它可能不会选择最小的基础类型，而是选择对优化大小有帮助的类型。为此，C++98
只支持枚举定义（所有枚举名全部列出来）；枚举声明是不被允许的。这使得编译器能为之前使用的每一个枚举选择一个基础类型。

但是不能前置声明枚举也是有缺点的。最大的缺点莫过于它可能增加编译依赖。再次考虑**Status**枚举：

```
enum Status { good = 0,
              failed = 1,
              incomplete = 100,
              corrupt = 200,
              indeterminate = 0xFFFFFFFF
              };
```

这种**enum**很有可能用于整个系统，因此系统中每个包含这个头文件的组件都会依赖它。如果引入一个新状态值，

```
enum Status { good = 0,
              failed = 1,
              incomplete = 100,
              corrupt = 200,
              audited = 500,
              indeterminate = 0xFFFFFFFF
              };
```

那么可能整个系统都得重新编译，即使只有一个子系统——或者一个函数使用了新添加的枚举名。这是大家都不希望看到的。C++11中的前置声明可以解决这个问题。
比如这里有一个完全有效的限域枚举声明和一个以该限域枚举作为形参的函数声明：

```
enum class Status; // forward declaration
void continueProcessing(Status s); // use of fwd-declared enum
```

即使**Status**的定义发生改变，包含这些声明的头文件也不会重新编译。而且如果**Status**添加一个枚举名（比如添加一个*audited*），**continueProcessing**的行为不受影响（因为**continueProcessing**没有使用这个新添加的*audited*），**continueProcessing**也不需要重新编译。
但是如果编译器在使用它之前需要知晓该枚举的大小，该怎么声明才能让C++11做到C++98不能做到的事情呢？
答案很简单：限域枚举的基础类型总是已知的，而对于非限域枚举，你可以指定它。默认情况下，限域枚举的基础类型是`int`：

```
enum class Status; // 基础类型是int
```

如果默认的*int*不适用，你可以重写它：

```
enum class Status: std::uint32_t;   // Status的基础类型
                                    // 是std::uint32_t
                                    // (需要包含 <cstdint>)
```

不管怎样，编译器都知道限域枚举中的枚举名占用多少字节。要为非限域枚举指定基础类型，你可以同上，然后前向声明一下：

```
enum Color: std::uint8_t;   // 为非限域枚举Color指定
                            // 基础为
                            // std::uint8_t
```

基础类型说明也可以放到枚举定义处：

```
enum class Status: std::uint32_t { good = 0,
                                   failed = 1,
                                   incomplete = 100,
                                   corrupt = 200,
                                   audited = 500,
                                   indeterminate = 0xFFFFFFFF
                                   };
```

限域枚举避免命名空间污染而且不接受隐式类型转换，但它并非万事皆宜，你可能会很惊讶听到至少有一种情况下非限域枚举是很有用的。
那就是获取C++11 tuples中的字段的时候。比如在社交网站中，假设我们有一个`tuple`保存了用户的名字，email地址，声望点：

```
using UserInfo = // 类型别名，参见Item 9
    std::tuple<std::string, // 名字
    std::string, // email地址
    std::size_t> ; // 声望
```

虽然注释说明了tuple各个字段对应的意思，但当你在另文件遇到下面的代码那之前的注释就不是那么有用了：

```
UserInfo uInfo; // tuple对象
…
auto val = std::get<1>(uInfo); // 获取第一个字段
```

作为一个程序员，你有很多工作要持续跟进。你应该记住第一个字段代表用户的email地址吗？我认为不。
可以使用非限域枚举将名字和字段编号关联起来以避免上述需求：

```
enum UserInfoFields { uiName, uiEmail, uiReputation };
UserInfo uInfo;
…
auto val = std::get<uiEmail>(uInfo); // , 获取用户email
```

之所以它能正常工作是因为`UserInfoFields`中的枚举名隐式转换成**std::size_t**了,其中**std::size_t**是**std::get**模板实参所需的。
对应的限域枚举版本就很啰嗦了：

```
enum class UserInfoFields { uiName, uiEmail, uiReputation };
UserInfo uInfo; // as before
…
auto val =
std::get<static_cast<std::size_t>(UserInfoFields::uiEmail)>
(uInfo);
```

为避免这种冗长的表示，我们可以写一个函数传入枚举名并返回对应的**std::size_t**值，但这有一点技巧性。
std::get是一个模板（函数），需要你给出一个**std::size_t**值的模板实参（注意使用`<>`而不是`()`），因此将枚举名变换为**std::size_t**值会发生在编译期。
如Item 15提到的，那必须是一个**constexpr**模板函数。
事实上，它也的确该是一个**constexpr**函数，因为它应该能用于任何`enum`。
如果我们想让它更一般化，我们还要泛化它的返回类型。较之于返回**std::size_t**，我们更应该泛化枚举

的基础类型。
这可以通过**std::underlying_type**这个`type trait`获得。（参见Item 9关于type trait的内容）。
最终我们还要再加上**noexcept**修饰（参见Item 14），因为我们知道它肯定不会产生异常。
根据上述分析最终得到的**toUType**模板函数在编译期接受任意枚举名并返回它的值：

```
template<typename E>
constexpr typename std::underlying_type<E>::type
  toUType(E enumerator) noexcept
{
  return
    static_cast<typename
      std::underlying_type<E>::type>(enumerator);
}
```

在C++14中，**toUType**还可以进一步用`std::underlying_type_t`（参见Item 9）代替`typename std::underly ing_type<E>::type`打磨：

```
template<typename E> // C++14
constexpr std::underlying_type_t<E>
  toUType(E enumerator) noexcept
{
  return static_cast<std::underlying_type_t<E>>(enumerator);
}
```

还可以再用C++14 auto（参见Item 3）打磨一下代码：

```
template<typename E> // C++14
constexpr auto
  toUType(E enumerator) noexcept
{
  return static_cast<std::underlying_type_t<E>>(enumerator);
}
```

不管它怎么写，**toUType**现在允许这样访问tuple的字段了：

```
auto val = std::get<toUType(UserInfoFields::uiEmail)>(uInfo);
```

比起使用非限域枚举，限域有很多可圈可点的地方，它避免命名空间污染，防止不经意间使用隐式转换。
（下面这句我没看懂，保留原文。。（是什么典故吗。。。））
In many cases, you
may decide that typing a few extra characters is a reasonable price to pay for the ability
to avoid the pitfalls of an enum technology that dates to a time when the state of
the art in digital telecommunications was the 2400-baud modem.

记住

- C++98的枚举即非限域枚举
- 限域枚举的枚举名仅在enum内可见。要转换为其它类型只能使用cast。
- 非限域/限域枚举都支持基础类型说明语法，限域枚举基础类型默认是`int`。非限域枚举没有默认基础类型。
- 限域枚举总是可以前置声明。非限域枚举仅当指定它们的基础类型时才能前置。