Item 6:Use the explicitly typed initializer idiom when auto deduces undesired types.

条款六:auto推导若非己愿，使用显式类型初始化惯用法

在Item5中解释了比起显式指定类型使用auto声明变量有若干技术优势，但是有时当你想向左转auto却向右转。举个例子，假如我有一个函数，参数为Widget，返回一个`std::vector<bool>`，这里的bool表示Widget是否提供一个独有的特性。

```
std::vector<bool> features(const Widget& w);
```

更进一步假设5表示是否Widget具有高优先级，我们可以写这样的代码:

```
bool highPriority = features(w)[5];
...
processWidget(w,highPriority);
```

这个代码没有任何问题。它会正常工作，但是如果我们使用auto代替显式指定类型做一些看起来很无害的改变:

```
auto highPriority = features(w)[5];
...
processWidget(w,highPriority);      //未定义行为!
```

就像注释说的，这个processWidget是一个未定义行为。为什么呢？答案有可能让你很惊讶，使用auto后highPriority不再是bool类型。虽然从概念上来说`std::vector<bool>`意味着存放bool，但是`std::vector<bool>`的`operator[]`不会返回容器中元素的引用，取而代之它返回一个`std::vector<bool>::reference`的对象（一个嵌套于`std::vector<bool>`中的类）。`std::vector<bool>::reference`之所以存在是因为`std::vector<bool>`指定了它作为代理类。`operator[]`返回一个代理类来扮演bool&。要想成功扮演这个角色，bool&适用的上下文`std::vector<bool>::reference`也必须一样能适用。基于这个特性`std::vector<bool>::reference`可以隐式的转化为bool（不是bool&，是bool！要想完整的解释`std::vector<bool>::reference`能模拟bool&的行为所使用的一堆技术可能扯得太远了，所以这里简单地说隐式类型转换只是这个大型马赛克的一小块）

有了这些信息，我们再来看看原始代码的一部分:

```
bool highPriority = features(w)[5];     //显式的声明highPriority的类型
```

这里，feature返回一个`std::vector<bool>`对象后再调用`operator[]`，`operator[]`将会返回一个`std::vector<bool>::reference`对象，然后再通过隐式转换赋值给bool变量highPriority。highPriority因此表示的是features返回的vector中的第五个bit，这也正如我们所期待的那样。
然后再对照一下当使用auto时发生了什么:

```
auto highPriority = features(w)[5];     //推导highPriority的类型
```

同样的，feature返回一个`std::vector<bool>`对象，再调用`operator[]`，`operator[]`将会返回一个`std::vector<bool>::reference`对象，但是现在这里有一点变化了，auto推导highPriority的类型为`std::vector<bool>::reference`，但是highPriority对象没有第五bit的值。

这个值取决于`std::vector<bool>::reference`的具体实现。其中的一种实现是这样的（`std::vector<bool>::reference`）对象包含一个指向word的指针，然后加上方括号中的偏移实现被引用bit这样的行为。然后再来考虑highPriority初始化表达的意思，注意这里假设`std::vector<bool>::reference`就是刚提到的实现方式。

调用feature将返回一个std::vector，这个对象没有名字，为了方便我们的讨论，我这里叫他temp，`operator[]`被temp调用，然后然后的`std::vector<bool>::reference`包含一个指针，这个指针指向一个temp里面的word，加上相应的偏移,。highPriority是一个`std::vector<bool>::reference`的拷贝，所以highPriority也包含一个指针，指向temp中的一个word，加上合适的偏移，这里是5.在这个语句解释的时候temp将会被销毁，因为它是一个临时变量。因此highPriority包含一个悬置的指针，如果用于processWidget调用中将会造成未定义行为：

```
processWidget(w,highPriority);      //未定义行为!
                                    //highPriority包含一个悬置指针
```

`std::vector<bool>::reference`是一个代理类的例子：所谓代理类就是以模仿和增强一些类型的行为为目的而存在的类。很多情况下都会使用代理类，`std::vector<bool>::reference`展示了对`std::vector<bool>`使用`operator[]`来实现引用bit这样的行为。另外，C++标准模板库中的智能指针也是用代理类实现了对原始指针的资源管理行为。代理类的功能已被大家广泛接受。事实上，“Proxy”设计模式是软件设计这座万神庙中一直都存在的高级会员。

一些代理类被设计于用以对客户可见。比如`std::shared_ptr`和`std::unique_ptr`。其他的代理类则与之相反，比如`std::vector<bool>::reference`和`std::bitset::reference`。

在后者的阵营里一些C++库也是用了表达式模板的黑科技。这些库通常被用于提高数值运算的效率。给出一个矩阵类Matrix和矩阵对象m1，m2，m3，m4，举个例子，这个表达式

```
Matrix sum = m1 + m2 + m3 + m4;
```

可以使计算更加高效，只需要使让`operator+`返回一个代理类代理结果而不是返回结果本身。也就是说，对两个Matrix对象使用`operator+`将会返回如`Sum<Matrix,Matrix>`这样的代理类作为结果而不是直接返回一个Matrix对象。在`std::vector<bool>::reference`和bool中存在一个隐式转换，同样对于Matrix来说也可以存在一个隐式转换允许Matrix的代理类转换为Matrix，这让表达式等号右边能产生代理对象来初始化Sum。客户应该避免看到实际的类型。

作为一个通则，不可见的代理类通常不适用于auto。这样类型的对象的生命期通常不会设计为能活过一条语句，所以创建那样的对象你基本上就走向了违反程序库设计基本假设的道路.`std::vector<bool>::reference`就是这种情况，我们看到违反这个基本假设将导致未定义行为。

因此你想避开这种形式的代码：

```
auto someVar = expression of "invisible" proxy class type;
```

但是你怎么能意识到你正在使用代理类？它们被设计为不可见，至少概念上说是这样！每当你发现它们，你真的应该舍弃Item5演示的auto所具有的诸多好处吗？

让我们首先回到如何找到它们的问题上。虽然代理类都在程序员日常使用的雷达下方飞行，但是很多库都证明它们可以上方飞行。当你越熟悉你使用的库的基本设计理念，你的思维就会越活跃，不至于思维僵化认为代理类只能在这些库中使用。

当缺少文档的时候，可以去看看头文件。很少会出现源代码全都用代理对象，它们通常用于一些函数的返回类型，所以通常能从函数签名中看出它们的存在。这里有一份来自C++ STANDARD的说明书：

```
namespace std{
    template<class Allocator>
    class vector<bool,Allocator>{
        public:
        class reference{...};

        reference operator[](size_type n);
    };
}
```

假设你知道对std::vector使用`operator[]`通常会返回一个**T&**,在这里operator[]不寻常的返回类型提示你它使用了代理类。多关注你使用的接口可以暴露代理类的存在。

实际上，很多开发者都是在跟踪一些令人困惑的复杂问题或在单元测试出错进行调试时才看到代理类的使用。不管你怎么发现它们的，当你不知道这个类型有没有被代理还想使用auto时你就不能单单只用一个auto。auto本身没什么问题，问题是auto不会推导出你想要的类型。解决方案是强制使用一个不同的类型推导形式，这种方法我通常称之为显式类型初始器惯用法（*the explicitly typed initialized idiom*）

显式类型初始器惯用法使用auto声明一个变量，然后对表达式强制类型转换得出你期望的推导结果。举个例子，我们该怎么将这个惯用法施加到highPriority上？

```
auto highPriority = static_cast<bool>(features(w)[5]);
```

这里，feature(w)[5]还是返回一个`std::vector<bool>::reference`对象，就像之前那样，但是这个转型使得表达式类型为bool，然后auto才被用于推导highPriority。在运行时，对std::vector使用operator[]将返回一个std::vector::reference，然后强制类型转换使得它执行向bool的转型，在这个过程中指向`std::vector<bool>`的指针已经被解引用。这就避开了我们之前的未定义行为。然后5将被用于指向bit的指针，bool值被用于初始化highPriority。

对于Matrix来说，显式类型初始器惯用法是这样的：

```
auto sum = static_cast<Matrix>(m1+m2+m3+m4);
```

应用这个惯用法不限制初始化表达式产生一个代理类。它也可以用于强调你声明了一个变量类型，它的类型不同于初始化表达式的类型。举个例子，假设你有这样一个表达式计算公差值：

```
double calEpsilon();
```

calEpsilon清楚的表明它返回一个double，但是假设你知道对于这个程序来说使用float的精度已经足够了，而且你很关心double和float的大小。你可以声明一个float变量储存**calEpsilon**的计算结果。

```
float ep = calEpsilon();
```

但是这几乎没有表明“我确实要减少函数返回值的精度”。使用显式类型初始器惯用法我们可以这样：

```
auto ep = static_cast<float>(calEpsilon());
```

处于同样的原因，如果你故意想用int类型存储一个表达式返回的float类型的结果，你也可以使用这个方法。假如你需要计算一个随机访问迭代器（比如std::vector,std::deque,std::array）中某元素的下标，你给它一个0.0到1.0的值表明这个元素离容器的头部有多远（0.5意味着位于容器中间）。进一步假设你很自信结果下标是int。如果容器是c，d是double类型变量，你可以用这样的方法计算容器下标：

```
int index = d * c.size();
```

但是这种写法并没有明确表明你想将右侧的double类型转换成int类型，显式类型初始器可以帮助你正确表意：

```
auto index = static_cast<int>(d * size());
```

记住

- 不可见的代理类可能会使auto从表达式中推导出“错误的”类型
- 显式类型初始器惯用法强制auto推导出你想要的结果