Item 13:优先考虑const_iterator而非iterator

条款 13:优先考虑const_iterator而非iterator

STL **const_iterator**等价于指向常量的指针。它们都指向不能被修改的值。标准实践是能加上**const**就加上，这也指示我们对待**const_iterator**应该如出一辙。

上面的说法对C++11和C++98都是正确的，但是在C++98中，标准库对**const_iterator**的支持不是很完整。首先不容易创建它们，其次就算你有了它，它的使用也是受限的。
假如你想在`std::vector<int>`中查找第一次出现**1983**(C++代替C with classes的那一年)的位置，然后插入**1998**（第一个ISO C++标准被接纳的那一年）。如果vector中没有1983，那么就在vector尾部插入。在C++98中使用**iterator**可以很容易做到：

```
std::vector<int> values;
…
std::vector<int>::iterator it =
std::find(values.begin(),values.end(), 1983);
values.insert(it, 1998);
```

但是这里**iterator**真的不是一个好的选择，因为这段代码不修改**iterator**指向的内容。用**const_iterator**重写这段代码是很平常的，但是在C++98中就不是了。下面是一种概念上可行但是不正确的方法：

```
typedef std::vector<int>::iterator IterT;   // typetypedef
std::vector<int>::const_iterator ConstIterT; // defs
std::vector<int> values;
…
ConstIterT ci =
    std::find(static_cast<ConstIterT>(values.begin()), // cast
        static_cast<ConstIterT>(values.end()), // cast
        1983);
values.insert(static_cast<IterT>(ci), 1998); // 可能无法通过编译，原因见下
```

typedef不是强制的，但是可以让类型转换更好写。（你可能想知道为什么我使用**typedef**而不是Item 9提到的别名声明，因为这段代码在演示C++98做法，别名声明是C++11加入的特性）

之所以`std::find`的调用会出现类型转换是因为在C++98中**values**是非常量容器，没办法简简单单的从非常量容器中获取**const_iterator**。严格来说类型转换不是必须的，因为用其他方法获取**const_iterator**也是可以的（比如你可以把values绑定到常量引用上，然后再用这个变量代替values），但不管怎么说，从非常量容器中获取**const_iterator**的做法都有点别扭。

当你费劲地获得了**const_iterator**，事情可能会变得更糟，因为C++98中，插入操作的位置只能由**iterator**指定，**const_iterator**是不被接受的。这也是我在上面的代码中，将**const_iterator**转换为**iterat**的原因，因为向**insert**传入**const_iterator**不能通过编译。

老实说，上面的代码也可能无法编译，因为没有一个可移植的从**const_iterator**到**iterator**的方法，即使使用`static_cast`也不行。甚至传说中的牛刀**reinterpret_cast**也杀不了这条鸡。（它C++98的限制，也不是C++11的限制，只是**const_iterator**就是不能转换为**iterator**，不管看起来对它们施以转换是有多么合理。）不过有办法生成一个**iterator**，使其指向和**const_iterator**指向相同，但是看起来不明显，也没有广泛应用，在这本书也不值得讨论。除此之外，我希望目前我陈述的观点是清晰的：**const_iterator**在C++98中会有很多问题。这一天结束时，开发者们不再相信能加**const**就加它的教条，而是只在实用的地方加它，C++98的**const_iterator**不是那么实用。

所有的这些都在C++11中改变了，现在**const_iterator**即容易获取又容易使用。容器的成员函数`cbegin`和`cend`产出**const_iterator**，甚至对于非常量容器，那些之前只使用**iterator**指示位置的STL成员函数也可以使用**const_iterator**了。使用C++11 **const_iterator**重写C++98使用**iterator**的代码也稀松平常：

```
std::vector<int> values; // 和之前一样
…
auto it =                                           // 使用cbegin
    std::find(values.cbegin(),values.cend(), 1983); // 和cend
values.insert(it, 1998);
```

现在使用**const_iterator**的代码就很实用了！

唯一一个C++11对于**const_iterator**支持不足（译注：C++14支持但是C++11的时候还没）的情况是：当你想写最大程度通用的库，并且这些库代码为一些容器和类似容器的数据结构提供非成员函数**begin**、**end**（以及**cbegin**，**cend**，**rbegin**，**rend**）而不是成员函数（其中一种情况就是原生数组）。最大程度通用的库会考虑使用非成员函数而不是假设成员函数版本存在。

举个例子，我们可以泛化下面的`findAndInsert`：

```
template<typename C, typename V>
void findAndInsert(C& container,     // 在容器中查找第一次
const V& targetVal,                  // 出现targetVal的位置，
const V& insertVal)                  // 然后插入insertVal
{
    using std::cbegin; // there
    using std::cend;
    auto it = std::find(cbegin(container),  // 非成员函数cbegin
                        cend(container),    // 非成员函数cend
                        targetVal);
    container.insert(it, insertVal);
}
```

它可以在C++14工作良好，但是很遗憾，C++11不在良好之列。由于标准化的疏漏，C++11只添加了非成员函数**begin**和**end**，但是没有添加**cbegin**，**cend**，**rbegin**，**rend**，**crbegin**，**crend**。C++14修订了这个疏漏，如果你使用C++11，并且想写一个最大程度通用的代码，而你使用的STL没有提供缺失的非成员函数**cbegin**和它的朋友们，你可以简单的抛出你自己的实现。比如，下面就是非成员函数**cbegin**的实现：

```
template <class C>
auto cbegin(const C& container)->decltype(std::begin(container))
{
    return std::begin(container); // 解释见下
}
```

你可能很惊讶非成员函数**cbegin**没有调用成员函数**cbegin**吧？但是请跟逻辑走。这个**cbegin**模板接受任何容器或者类似容器的数据结构`C`，并且通过`const`引用访问第一个实参**container**。如果`C`是一个普通的容器类型（如`std::vector<int>`），**container**将会引用一个常量版本的容器（即`const std::vector<int>&`）。对const容器调用非成员函数**begin**（由C++11提供)将产出**const_iterator**，这个迭代器也是模板要返回的。用这种方法实现的好处是就算容器只提供**begin**不提供**cbegin**也没问题。那么现在你可以将这个非成员函数**cbegin**施于只支持**begin**的容器。

如果**C**是原生数组，这个模板也能工作。这时，**container**成为一个const数组。C++11为数组提供特化版本的非成员函数**begin**，它返回指向数组第一个元素的指针。一个const数组的元素也是const，所以对于const数组，非成员函数**begin**返回指向const的指针。在数组的上下文中，所谓指向const的指针，也就是**const_iterator**了。

回到最开始，本条款的中心是鼓励你只要能就使用**const_iterator**。最原始的动机是——只要它有意义就加上const——C++98就有的思想。但是在C++98，它（译注：const_iterator）只是一般有用，到了C++11,它就是极其有用了，C++14在其基础上做了些修补工作。

记住

- 优先考虑const_iterator而非iterator
- 在最大程度通用的代码中，优先考虑非成员函数版本的**begin**，**end**，**rbegin**等，而非同名成员函数