Item 3: Understand decltype

条款三:理解decltype

decltype是一个奇怪的东西。给它一个名字或者表达式**decltype**就会告诉你名字或者表达式的类型。通常，它会精确的告诉你你想要的结果。但有时候它得出的结果也会让你挠头半天最后只能网上问答求助寻求解释。

我们将从一个简单的情况开始，没有任何令人惊讶的情况。相比模板类型推导和auto类型推导，decltype只是简单的返回名字或者表达式的类型：

```
const int i=0;                          //decltype(i)是const int

bool f(const Widget& w);      //decltype(w)是const Widget&
                                                //decltype(f)是bool(const
Widget&)

struct Point{
    int x;                                      //decltype(Point::x)是int
    int y;                                      //decltype(Point::y)是int
};

template<typename T>
class Vector{
    ...
    T& operator[](std::size_t index);
    ...
}
vector<int> v;                      //decltype(v)是vector<int>
...
if(v[0]==0)                             //decltype(v[0])是int&
```

看见了吧？没有任何奇怪的东西。

在C++11中，decltype最主要的用途就是用于函数模板返回类型，而这个返回类型依赖形参。举个例子，假定我们写一个函数，一个参数为容器，一个参数为索引值，这个函数支持使用方括号的方式访问容器中指定索引值的数据，然后在返回索引操作的结果前执行认证用户操作。函数的返回类型应该和索引操作返回的类型相同。

对一个T类型的容器使用**operator[]** 通常会返回一个T&对象，比如**std::deque**就是这样，但是**std::vector**有一个例外，对于**std::vector**，**operator[]**不会返回**bool&**，它会返回一个有名字的对象类型（译注：MSVC的STL实现中返回的是**std::_Vb_reference<std::Wrap_alloc<std::allocator>>**）。关于这个问题的详细讨论请参见Item6，这里重要的是我们可以看到对一个容器进行**operator[]**操作返回的类型取决于容器本身。

使用**decltype**使得我们很容易去实现它，这是我们写的第一个版本，使用**decltype**计算返回类型，这个模板需要改良，我们把这个推迟到后面：

```
template<typename Container,typename Index>
auto authAndAccess(Container& c,Index i)
->decltype(c[i])
{
    authenticateUser();
    return c[i];
}
```

函数名称前面的**auto**不会做任何的类型推导工作。相反的，他只是暗示使用了C++11的尾置返回类型语法，即在函数形参列表后面使用一个**->** 符号指出函数的返回类型，尾置返回类型的好处是我们可以在函数返回类型中使用函数参数相关的信息。在**authAndAccess**函数中，我们指定返回类型使用c和i。如果我们按照传统语法把函数返回类型放在函数名称之前， c和i就未被声明所以不能使用。

在这种声明中，**authAndAccess**函数返回**operator[]** 应用到容器中返回的对象的类型，这也正是我们期望的结果。

C++11允许自动推导单一语句的lambda表达式的返回类型， C++14扩展到允许自动推导所有的lambda表达式和函数，甚至它们内含多条语句。对于**authAndAccess**来说这意味着在C++14标准下我们可以忽略尾置返回类型，只留下一个**auto**。在这种形式下**auto**不再进行auto类型推导，取而代之的是它意味着编译器将会从函数实现中推导出函数的返回类型。

```
template<typename Container,typename Index> //C++ 14版本
auto authAndAccess(Container& c,Index i)
{
    authenticateUser();
    return c[i];
}
```

Item2解释了函数返回类型中使用**auto**编译器实际上是使用的模板类型推导的那套规则。如果那样的话就会这里就会有一些问题，正如我们之前讨论的，**operator[]** 对于大多数T类型的容器会返回一个**T&**，但是Item1解释了在模板类型推导期间，如果表达式是一个引用那么引用会被忽略。基于这样的规则，考虑它会对下面用户的代码有哪些影响：

```
std::deque<int> d;
...
authAndAccess(d,5)=10;          //认证用户，返回d[5],
                                                    //然后把10赋值给它
                                                     //无法通过编译器!
```

在这里**d[5]**本该返回一个**int&**，但是模板类型推导会剥去引用的部分，因此产生了**int**返回类型。函数返回的值是一个右值，上面的代码尝试把10赋值给右值，C++11禁止这样做，所以代码无法编译。

要想让**authAndAccess**像我们期待的那样工作，我们需要使用**decltype**类型推导来推导它的返回值，比如指定**authAndAccess**应该返回一个和**c[i]** 表达式类型一样的类型。C++期望在某些情况下当类型被暗示时需要使用**decltype**类型推导的规则，C++14通过使用**decltype(auto)** 说明符使得这成为可能。我们第一次看见**decltype(auto)** 可能觉得非常的矛盾，（到底是decltype还是auto？），实际上我们可以这样解释它的意义：**auto**说明符表示这个类型将会被推导，**decltype**说明**decltype**的规则将会引用到这个推导过程中。因此我们可以这样写**authAndAccess**：

```
template<typename Container,typename Index>
decltype(auto)
authAndAccess(Container& c,Index i)
{
    authenticateUser();
    return c[i];
}
```

现在authAndAccess将会真正的返回c[i]的类型。现在事情解决了，一般情况下c[i]返回**T&**，authAndAccess也会返回
T&,特殊情况下c[i]返回一个对象，authAndAccess也会返回一个对象。

decltype(auto) 的使用不仅仅局限于函数返回类型，当你想对初始化表达式使用decltype推导的规则，你也可以使用：

```
Widget w;

const Widget& cw = w;

auto myWidget1 = cw;             //auto类型推导
                                 //myWidget1的类型为Widget
decltype(auto) myWidget2 = cw;   //decltype类型推导
                                 //myWidget2的类型是const Widget&
```

但是这里有两个问题困惑着你。一个是我之前提到的authAndAccess的改良至今都没有描述。让我们现在加上它。

再看看C++14版本的authAndAccess：

```
template<typename Container,typename Index>
decltype(auto) authAndAccess(Container& c,Index i);
```

容器通过传引用的方式传递非常量左值引用，因为返回一个引用允许用户可以修改容器。但是这意味着在这个函数里面不能传值调用，右值不能被绑定到左值引用上（除非这个左值引用是一个const，但是这里明显不是）。

公认的向**authAndAccess**传递一个右值是一个edge case。一个右值容器，是一个临时对象，通常会在authAndAccess调用结束被销毁，这意味着authAndAccess返回的引用将会成为一个悬置的(dangle)引用。但是使用向authAndAccess传递一个临时变量也并不是没有意义，有时候用户可能只是想简单的获得临时容器中的一个元素的拷贝，比如这样：

```
std::deque<std::string> makeStringDeque();       //工厂函数

//从makeStringDeque中或得第五个元素的拷贝并返回
auto s = authAndAccess(makeStringDeque(),5);
```

要想支持这样使用authAndAccess我们就得修改一下当前的声明使得它支持左值和右值。重载是一个不错的选择（一个函数重载声明为左值引用，另一个声明为右值引用），但是我们就不得不维护两个重载函数。另一个方法是使authAndAccess的引用可以绑定左值和右值，Item24解释了那正是通用引用能做的，所以我们这里可以使用通用引用进行声明：

```
template<typename Containter,typename Index>
decltype(auto) authAndAccess(Container&& c,Index i);
```

在这个模板中，我们不知道我们操纵的容器的类型是什么，那意味着我们相当于忽略了索引对象的可能，对一个未知类型的对象使用传值是通常对程序的性能有极大的影响在这个例子中还会造成不必要的拷贝，还会造成对象切片行为，以及给同事落下笑柄。但是就容器索引来说，我们遵照标准模板库对于对于索引的处理是有理由的，所以我们坚持传值调用。

然而，我们还需要更新一下模板的实现让它能听从Item25的告诫应用**std::forward**实现通用引用：

```
template<typename Container,typename Index>     //最终的C++14版本
decltype(auto)
authAndAccess(Container&& c,Index i){
    authenticateUser();
    return std::forward<Container>(c)[i];
}
```

这样就能对我们的期望交上一份满意的答卷，但是这要求编译器支持C++14。如果你没有这样的编译器，你还需要使用C++11版本的模板，它看起来和C++14版本的极为相似，除了你不得不指定函数返回类型之外：

```
template<typename Container,typename Index>     //最终的C++11版本
auto
authAndAccess(Container&& c,Index i)
->decltype(std::forward<Container>(c)[i])
{
    authenticateUser();
    return std::forward<Container>(c)[i];
}
```

另一个问题是就像我在条款的开始唠叨的那样，decltype通常会产生你期望的结果，但并不总是这样。在极少数情况下它产生的结果可能让你很惊讶。老实说如果你不是一个大型库的实现者你不太可能会遇到这些异常情况。

为了完全理解decltype的行为，你需要熟悉一些特殊情况。它们大多数都太过晦涩以至于几乎没有书进行有过权威的讨论，这本书也不例外，但是其中的一个会让我们更加理解decltype的使用。

对一个名字使用decltype将会产生这个名字被声明的类型。名字是左值表达式，但那不影响decltype的行为，decltype确保产生的类型总是左值引用。换句话说，如果一个左值表达式除了名字外还有类型，那么decltype将会产生**T&**LEIX .这几乎没有什么太大影响，因为大多数左值表达式的类型天生具备一个左值引用修饰符。举个例子，函数返回左值，几乎也返回了左值引用。

这个行为暗含的意义值得我们注意，在：

```
int x =0;
```

中，x是一个变量的名字，所以**decltype(x)** 是**int**。但是如果用一个小括号包覆这个名字，比如这样**(x)**，就会产生一个比名字更复杂的表达式。对于名字来说，**x**是一个左值，C++11定义了表达式**(x)** 也是一个左值。因此**decltype((x))** 是**int&** 。用小括号覆盖一个名字可以改变decltype对于名字产生的结果。

在C++11中这稍微有点奇怪，但是由于C++14允许了**decltype(auto)** 的使用，这意味着你在函数返回语句中细微的改变就可以影响类型的推导：

```
decltype(auto) f1()
{
    int x = 0;
    ...
    return x;           //decltype(x) 是int，所以f1返回int
}

decltype(auto) f2()
{
    int x =0l;
    return (x);         //decltype((x))是int&，所以f2返回int&
}
```

注意不仅f2的返回类型不同于f1，而且它还引用了一个局部变量！这样的代码将会把你送上未定义行为的特快列车，一辆你绝对不想上第二次的车。

当使用**decltype(auto)** 的时候一定要加倍的小心，在表达式中看起来无足轻重的细节将会影响到类型的推导。为了确认类型推导是否产出了你想要的结果，请参见Item4描述的那些技术。

同时你也不应该忽略decltype这块大蛋糕。没错，decltype可能会偶尔产生一些令人惊讶的结果，但那毕竟是少数情况。通常，decltype都会产生你想要的结果，尤其是当你对一个名字使用decltype时，因为在这种情况下，decltype只是做一件本分之事：它产出名字的声明类型。

记住

- **decltype**总是不加修改的产生变量或者表达式的类型。
- 对于T类型的左值表达式，**decltype**总是产出T的引用即**T&**。
- C++14支持**decltype(auto)**，就像auto一样，推导出类型，但是它使用自己的独特规则进行推导。