Item 9:Prefer alias declarations to typedefs

条款九:优先考虑别名声明而非typedefs

我相信每个人都同意使用STL容器是个好主意，并且我希望Item18能说服你让你觉得使用**std:unique_ptr**也是个好主意，但我猜没有人喜欢写上几次`std::unique_ptr<std::unordered_map<std::string,std::string>>`这样的类型，它可能会让你患上腕管综合征的风险大大增加。

避免上述医疗悲剧也很简单，引入**typedef**即可:

```
typedef  std::unique_ptr<std::unordered_map<std::string, std::string>>
 UPtrMapSS;
```

但**typedef**是C++98的东西。虽然它可以在C++11中工作，但是C++11也提供了一个别名声明（alias declaration）:

```
using UPtrMapSS =  std::unique_ptr<std::unordered_map<std::string, std::string>>;
```

由于这里给出的**typedef**和别名声明做的都是完全一样的事情，我们有理由想知道会不会出于一些技术上的原因两者有一个更好。

这里，在说它们之前我想提醒一下很多人都发现当声明一个函数指针时别名声明更容易理解:

```
// FP是一个指向函数的指针的同义词，它指向的函数带有int和const std::string&形参，不返回任何东西
typedef void (*FP)(int, const std::string&);      // typedef

//同上
using FP = void (*)(int, const std::string&);     // 别名声明
```

当然，两个结构都不是非常让人满意，没有人喜欢花大量的时间处理函数指针类型的别名*[0]*，所以至少在这里，没有一个吸引人的理由让你觉得别名声明比**typedef**好。

不过有一个地方使用别名声明吸引人的理由是存在的：模板。特别的，别名声明可以被模板化但是**typedef**不能。
这使得C++11程序员可以很直接的表达一些C++98程序员只能把**typedef**嵌套进模板化的**struct**才能表达的东西，
考虑一个链表的别名，链表使用自定义的内存分配器，**MyAlloc**。

使用别名模板，这真是太容易了:

```
template<typename T>
using MyAllocList = std::list<T,MyAlloc<T>>;

MyAllocList<Widget> lw;
```

使用**typedef**，你就只能从头开始:

```
template<typename T>
struct MyAllocList {
    typedef std::list<T, MyAlloc<T>> type;
};
MyAllocList<Widget>::type lw;
```

更糟糕的是，如果你想使用在一个模板内使用**typedef**声明一个持有链表的对象，而这个对象又使用了模板参数，你就不得不在在**typedef**前面加上**typename**

```
template<typename T>
class Widget {
private:
    typename MyAllocList<T>::type list;
    …
};
```

这里**MyAllocList::type**使用了一个类型，这个类型依赖于模板参数**T**。
因此**MyAllocList::type**是一个依赖类型，在C++很多讨人喜欢的规则中的一个提到必须要在依赖类型名前加上**typename**。
如果使用别名声明定义一个**MyAllocList**，就不需要使用**typename**（同时省略麻烦的**::type**后缀），

```
template<typename T>
using MyAllocList = std::list<T, MyAlloc<T>>;  // as before
template<typename T>
class Widget {
private:
    MyAllocList<T> list;
     …
};
```

对你来说，**MyAllocList**（使用了模板别名声明的版本）可能看起来和**MyAllocList::type**（使用**typedef**的版本）一样都应该依赖模板参数**T**，但是你不是编译器。
当编译器处理**Widget**模板时遇到**MyAllocList**（使用模板别名声明的版本），它们知道**MyAllocList**是一个类型名，
因为**MyAllocList**是一个别名模板。它一定是一个类型名。因此**MyAllocList**就是一个非依赖类型，就不要求必须使用**typename**。

当编译器在**Widget**的模板中看到**MyAllocList::type**（使用**typedef**的版本），它不能确定那是一个类型的名称。
因为可能存在**MyAllocList**的一个特化版本没有**MyAllocList::type**。
那听起来很不可思议，但不要责备编译器穷尽考虑所有可能。
举个例子，一个误入歧途的人可能写出这样的代码：

```
class Wine { … };
template<>                          // 当T是Wine
class MyAllocList<Wine> {           // 特化MyAllocList
private:
    enum class WineType             // 参见Item10了解
    { White, Red, Rose };           // "enum class"
    WineType type;                  // 在这个类中，type是
    …                               // 一个数据成员!
};
```

就像你看到的，**MyAllocList::type**不是一个类型。
如果**Widget**使用**Wine**实例化，在**Widget**模板中的**MyAllocList::type**将会是一个数据成员，不是一个类型。
在**Widget**模板内，如果**MyAllocList::type**表示的类型依赖于**T**，编译器就会坚持要求你在前面加上**typename**。

如果你尝试过模板元编程（TMP），你一定会碰到取模板类型参数然后基于它创建另一种类型的情况。
举个例子，给一个类型**T**，如果你想去掉**T**的常量修饰和引用修饰，比如你想把**const std::string&**变成**const std::string**。
又或者你想给一个类型加上**const**或左值引用，比如把**Widget**变成**const Widget**或**Widget&**。
（如果你没有用过玩过模板元编程，太遗憾了，因为如果你真的想成为一个高效C++程序员*[1]*，至少你需要熟悉C++的基础。你可以看看我在Item23，27提到的类型转换）。
C++11在*type traits*中给了你一系列工具去实现类型转换，如果要使用这些模板请包含头文件**<type_traits>**。
里面不全是类型转换的工具，也包含一些`predictable`接口的工具。给一个类型**T**，你想将它应用于转换中，结果类型就是**std::transformation<T>::type**，比如：

```
std::remove_const<T>::type           // 从const T中产出T
std::remove_reference<T>::type       // 从T&和T&&中产出T
std::add_lvalue_reference<T>::type   // 从T中产出T&
```

注释仅仅简单的总结了类型转换做了什么，所以不要太随便的使用。
在你的项目使用它们之前，你最好看看它们的详细说明书。
尽管写了一些，但我这里不是想给你一个关于type traits使用的教程。注意类型转换尾部的**::type**。
如果你在一个模板内部使用类型参数，你也需要在它们前面加上**typename**。
至于为什么要这么做是因为这些type traits是通过在**struct**内嵌套**typedef**来实现的。
是的，它们使用类型别名*[2]*技术实现，而正如我之前所说这比别名声明要差。

关于为什么这么实现是有历史原因的，但是我们跳过它（我认为太无聊了），因为标准委员会没有及时认识到别名声明是更好的选择，所以直到C++14它们才提供了使用别名声明的版本。
这些别名声明有一个通用形式：对于C++11的类型转换**std::transformation::type**在C++14中变成了**std::transformation_t**。
举个例子或许更容易理解：

```
std::remove_const<T>::type           // C++11: const T → T
std::remove_const_t<T>               // C++14 等价形式

std::remove_reference<T>::type       // C++11: T&/T&& → T
std::remove_reference_t<T>           // C++14 等价形式

std::add_lvalue_reference<T>::type   // C++11: T → T&
std::add_lvalue_reference_t<T>       // C++14 等价形式
```

C++11的的形式在C++14中也有效，但是我不能理解为什么你要去用它们。
就算你没有使用C++14，使用别名模板也是小儿科
只需要C++11，甚至每个小孩都能仿写。
对吧？如果你有一份C++14标准，就更简单了，只需要复制粘贴：

```
template <class T>
using remove_const_t = typename remove_const<T>::type;

template <class T>
using remove_reference_t = typename remove_reference<T>::type;

template <class T>
using add_lvalue_reference_t =  typename add_lvalue_reference<T>::type;
```

看见了吧？不能再简单了。

记住

- typedef不支持模板化，但是别名声明支持。
- 别名模板避免了使用"**::type**"后缀，而且在模板中使用**typedef**还需要在前面加上**typename**
- C++14提供了C++11所有类型转换的别名声明版本

译注

[0] 即FP

[1] 哈，这大概是作为《Modern C++ Design -Generic Programming and Design Pattern Applied》的作者的Scott Meyes才能说出的话。

[2] 作者所言的类型别名是泛指typedef和using语法进行的别名操作，根据上下文这里的类型别名指的是使用typedef