**前言：**

今天在看vector.h的时候，碰到一个using的奇怪用法，才疏学浅之前没有碰到过，整理一下。

来看下source code：

1. template<class \_Ty,
2. class \_Alloc = allocator<\_Ty>>
3. class vector
4. : public \_Vector\_alloc<\_Vec\_base\_types<\_Ty, \_Alloc>>
5. { *// varying size array of values*
6. private:
7. using \_Mybase = \_Vector\_alloc<\_Vec\_base\_types<\_Ty, \_Alloc>>;
8. using \_Alty = typename \_Mybase::\_Alty;
9. using \_Alty\_traits = typename \_Mybase::\_Alty\_traits;
11. public:
12. static\_assert(!\_ENFORCE\_MATCHING\_ALLOCATORS || is\_same<\_Ty, typename \_Alloc::value\_type>::value,
13. \_MISMATCHED\_ALLOCATOR\_MESSAGE("vector<T, Allocator>", "T"));
15. using value\_type = \_Ty;
16. using allocator\_type = \_Alloc;
17. using pointer = typename \_Mybase::pointer;
18. using const\_pointer = typename \_Mybase::const\_pointer;
19. using reference = \_Ty&;
20. using const\_reference = const \_Ty&;
21. using size\_type = typename \_Mybase::size\_type;
22. using difference\_type = typename \_Mybase::difference\_type;
23. using iterator = typename \_Mybase::iterator;
24. using const\_iterator = typename \_Mybase::const\_iterator;
25. using reverse\_iterator = \_STD reverse\_iterator<iterator>;
26. using const\_reverse\_iterator = \_STD reverse\_iterator<const\_iterator>;

下面来整理using的三种用法。

**1、命名空间的使用**

一般为了代码的冲突，都会用命名空间。例如，对于Android代码会使用Android作为命名空间。

namespace android;

在code中使用的时候可以用**android::**加具体的类方法。也可以直接使用using namespace android;

具体的命名空间使用方法不做过多说明。

**2、在子类中引用基类的成员**

来看下source code：

1. class T5Base {
2. public:
3. T5Base() :value(55) {}
4. virtual ~T5Base() {}
5. void test1() { cout << "T5Base test1..." << endl; }
6. protected:
7. int value;
8. };
10. class T5Derived : private T5Base {
11. public:
12. *//using T5Base::test1;*
13. *//using T5Base::value;*
14. void test2() { cout << "value is " << value << endl; }
15. };

基类中成员变量value是protected，在private继承之后，对于外界这个值为private，也就是说T5Derived的对象无法使用这个value。

如果想要通过对象使用，需要在public下通过using T5Base::value来引用，这样T5Derived的对象就可以直接使用。

同样的，对于基类中的成员函数test1()，在private继承后变为private，T5Derived的对象同样无法访问，通过using T5Base::test1 就可以使用了。

注意，using只是引用，不参与形参的指定。

**3、别名指定**

这点就是最开始看到的source code。在C++11中提出了通过using指定别名。

例如上面source code 中：

using value\_type = \_Ty

以后使用value\_type value; 就代表\_Ty value；

这个让我们想起了typedef，using 跟typedef有什么区别呢？哪个更好用些呢？

例如：

typedef std::unique\_ptr<std::unordered\_map<std::string, std::string>> UPtrMapSS;

而C++11中：

using UPtrMapSS = std::unique\_ptr<std::unordered\_map<std::string, std::string>>;

或许从这个例子中，我们是看不出来明显的好处的（而于我来说，以一个第三者的角度，这个例子也难以说服我一定要用C++11的using）。  
再来看下：

typedef void (\*FP) (int, const std::string&);

若不是特别熟悉函数指针与typedef的童鞋，我相信第一眼还是很难指出FP其实是一个别名，代表着的是一个函数指针，而指向的这个函数返回类型是void，接受参数是int, const std::string&。那么，让我们换做C++11的写法：

using FP = void (\*) (int, const std::string&);

我想，即使第一次读到这样代码，并且知道C++11 using的童鞋也能很容易知道FP是一个别名，using的写法把别名的名字强制分离到了左边，而把别名指向的放在了右边，比较清晰。  
  
  
而针对这样的例子，我想我可以再补充一个例子：

1. typedef std::string (Foo::\* fooMemFnPtr) (const std::string&);
3. using fooMemFnPtr = std::string (Foo::\*) (const std::string&);

从可读性来看，using也是要好于typedef的。  
  
那么，若是从可读性的理由支持using，力度也是稍微不足的。来看第二个理由，那就是举出了一个typedef做不到，而using可以做到的例子：alias templates, 模板别名。

1. template <typename T>
2. using Vec = MyVector<T, MyAlloc<T>>;
4. *// usage*
5. Vec<int> vec;

这一切都会非常的自然。  
  
  
那么，若你使用typedef来做这一切：

1. template <typename T>
2. typedef MyVector<T, MyAlloc<T>> Vec;
4. *// usage*
5. Vec<int> vec;

当你使用编译器编译的时候，将会得到类似：error: a typedef cannot be a template的错误信息。

那么，为什么typedef不可以呢？在 n1449 中提到过这样的话："we specifically avoid the term “typedef template” and introduce the new syntax involving the pair “using” and “=” to help avoid confusion: we are not defining any types here, we are introducing a synonym (i.e. alias) for an abstraction of a type-id (i.e. type expression) involving template parameters." 所以，我认为这其实是标准委员会他们的观点与选择，在C++11中，也是完全鼓励用using，而不用typedef的。

具体的可以看下Effective Modern C++