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：

```
class T5Base
{
public:
   T5Base() :value(55) {}
   virtual ~T5Base() {}
   void test1() { cout << "T5Base test1..." << endl; }
protected:
   int value;
};

class T5Derived : private T5Base {
public:
//using T5Base::test1;
//using T5Base::value;
void test2() { cout << "value is " << value << endl; }
};
```

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;

4. 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：

```
typedef std::string (Foo::* fooMemFnPtr) (const std::string&);
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。

```
template <typename T>
using Vec = MyVector<T, MyAlloc<T>>;

// usage
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：

```
template <typename T>
typedef MyVector<T, MyAlloc<T>> Vec;
```

```
// usage
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++

空间配置器作用：

1 空间的申请
2 对象的构造
3 对象的析构
4 空间的释放