Item10. 使用 scoped enum,而不是 unscoped enum

什么是 unscoped enum? Scoped enum?

在 C++11 之前，使用枚举的方式是这样的，

```
enum Color { black, white, red};
auto white = flase;  // 错误，因为 white 在作用域中已经定义。
```

枚举中的值的作用域不是在括号内，而是和 Color 的作用域是一样的。因此这些 enum 的成员已经泄露到了 enum 所在的作用域去了，官方称之为 unscoped.

在 C++11 之后,与之对应的版本为 scoped enum,如下

```
enum class Color { black, white, red};
auto white =false;  //可以的
Color c = Color::red;
```

因为 scoped enum 通过 enum class 声明，因此也叫做 enum class.

应该使用 unscoped enum 还是 scoped enum?

根据 modern effective c++ item10,根据这一节可以得出，倾向于选择 scoped enum, 相比于 unscoped enum，有以下优点：

- 上面所描述的一点是 scoped enum 中枚举中的变量不会泄露出来，从而较少对命名命名空间的污染.
- scoped enum 中的成员为强类类型，**不会隐式转换.**

```
enum Color { black, white, red};
Color c = red;
if(c > 3.4) { cout <<“c> 3.4”<<endl;}  // 隐式转换
```

// 上面的枚举变量可以隐式转换为 int，然后从数值类型转换为浮点类型，和 3.4 进行比较。

但是 scoped enum 如果需要转换到不同的类型，需要使用 cast 把 Color 转换为需要的类型。

```
enum Color { black, white, red};
double c = static_cast<double>(Color::red);
if (c > 3.4) { cout<< “c>3.4”<<endl;}  //需要强制转换。
```

Item 12. 把重写函数声明为 override

12.1. 成功重写的必要条件

重写的目的是为了通过基类的接口来调用派生类的函数。

```
class Base {
 public:
    virtual void doWork(); //  基类虚函数
};

class Derived: public Base {
public:
   virtual void doWork();    //virtual 是可选的
};

std::unique_ptr<Base> upb = std::make_unique<Derived>();
Upb->doWork();
```

为了能够重写成功，必要条件是：

- **基类函数必须是 virtual 的**。
- 基类函数和派生类函数的**名字必须完全一样**，除了析构函数
- 基类函数和派生类函数的**参数类型必须完全一样**
- 基类函数和派生类函数的 **const 属性必须完全一样**
- 基类函数和派生类函数的返回类型必须是兼容的[1]
- C++ 11 增加一条：函数的引用限定符必须完全一致[2]

[1] 基类函数和派生类函数的返回类型必须是兼容的？
[2] 函数的引用限定符是什么意思？

12.2. 使用 override