35 The Mutable Keyword in C++

1. const

const语境下讨论mutable时，很明显我们是在讨论某种不可变，但是实际上有可以改变的，就像是*mutable*反转了*const*的意思

```C++
class Entity
{
private:
    std::string m_Name;
public:
    const std::string& GetName() const   // 承诺只读
    {
        return m_Name;
    }
};

int main()
{
    const Entity e;    // const对象不能调用非const的成员函数，因为后者存在修改对象的可能
    e.GetName();
    std::cin.get();
}
```

我们想修改对象中的变量，但又不想改变对象。

```C++
class Entity
{
private:
    std::string m_Name;
    int m_DebugCount = 0;
public:
    const std::string& GetName() //去掉const，使m_DebugCount可以修改
    {
        m_DebugCount++;
        return m_Name;
    }
};

int main()
{
    const Entity e;
    e.GetName();                    // 这里又没办法编译
    std::cin.get();
}
```

所以*mutable*的作用就体现了：

```C++
mutable int m_DebugCount = 0;
```

现在类中的const方法可以修改这个成员了。

2. lambda

基本只会在const方法中用到，但*lambda*表达式中也可能遇到
*lambda*就像一个一次性的小函数，你可以写出来并赋值给一个变量，就像这里做的。

```C++
auto f = []()
{
    std::cout << "Hello" << std::endl;
};
f();
```

如果想传参，会发现没办法修改通过值捕获的变量。

```C++
auto f = [=]()
    {
        x++;         // 无法修改
        std::cout << x << std::endl;
    };
```

可以通过局部变量的方式得到结果。

```C++
int x = 8;
auto f = [=]()
{
    int y = x;     // 创建局部变量
    y++;
    std::cout << y << std::endl;
};
f();
```

用mutable关键字

```cpp
int x = 8;
// 使用 mutable 关键字允许在 Lambda 表达式中修改通过值捕获的变量
auto f = [=]() mutable
{
    x++;
    std::cout << x << std::endl;
};
f();
```

在 Lambda 表达式的捕获列表中，可以使用不同的捕获方式来指定外部变量的访问方式。以下是不同捕获方式的作用：

1. `[=]`：值捕获（Capture by value）
   - 使用 `[=]` 表示 Lambda 表达式会通过值复制的方式捕获所有外部变量。
   - 在 Lambda 表达式内部，可以访问这些变量的副本，但对副本的修改不会影响原始变量。
   - 使用值捕获可以保证在 Lambda 表达式执行期间外部变量的值保持不变。
2. `[&]`：引用捕获（Capture by reference）
   - 使用 `[&]` 表示 Lambda 表达式会通过引用绑定的方式捕获所有外部变量。
   - 在 Lambda 表达式内部，可以直接访问和修改这些变量，修改会直接影响原始变量。
   - 使用引用捕获可以避免在 Lambda 表达式内部创建变量的副本，节省内存和复制开销。
3. `[x]`：捕获指定变量（Capture specific variable）
   - 使用 `[x]` 表示只捕获指定的变量 `x`。
   - 在 Lambda 表达式内部，可以访问和修改被指定的变量 `x`，修改会直接影响原始变量。
   - 使用特定变量的捕获可以控制 Lambda 表达式对外部变量的访问范围。