智能指针（Smart Pointer）

- 传统指针存在的问题
  - 需要手动管理内存
  - 容易发生内存泄露（忘记释放、出现异常等）
  - 释放之后产生野指针

- 智能指针就是为了解决传统指针存在的问题
  - auto_ptr：属于C++98标准，在C++11中已经不推荐使用（有缺陷，比如不能用于数组）
  - shared_ptr：属于C++11标准
  - unique_ptr：属于C++11标准

小码哥教育 SEEMYGO

智能指针的简单自实现

```
template<class T>
class SmartPointer {
    T *m_pointer;
public:
    SmartPointer(T *pointer) :m_pointer(pointer) { }

    ~SmartPointer() {
        if (m_pointer == nullptr) return;
        delete m_pointer;
    }

    T *operator->() {
        return m_pointer;
    }
};
```

shared_ptr

- shared_ptr的设计理念
  - 多个shared_ptr可以指向同一个对象，当最后一个shared_ptr在作用域范围内结束时，对象才会被自动释放
- 可以通过一个已存在的智能指针初始化一个新的智能指针

```
shared_ptr<Person> p1(new Person());
shared_ptr<Person> p2(p1);
```

- 针对数组的用法

```
shared_ptr<Person> ptr1(new Person[5]{}, [](Person *p) { delete[] p; });
```

```
shared_ptr<Person[]> persons(new Person[5]{});
```

shared_ptr的原理

- 一个shared_ptr会对一个对象产生强引用（strong reference）

- 每个对象都有个与之对应的强引用计数，记录着当前对象被多少个shared_ptr强引用着
  - 可以通过shared_ptr的use_count函数获得强引用计数

- 当有一个新的shared_ptr指向对象时，对象的强引用计数就会+1

- 当有一个shared_ptr销毁时（比如作用域结束），对象的强引用计数就会-1

- 当一个对象的强引用计数为0时（没有任何shared_ptr指向对象时），对象就会自动销毁（析构）

小码哥教育 SEEMYGO

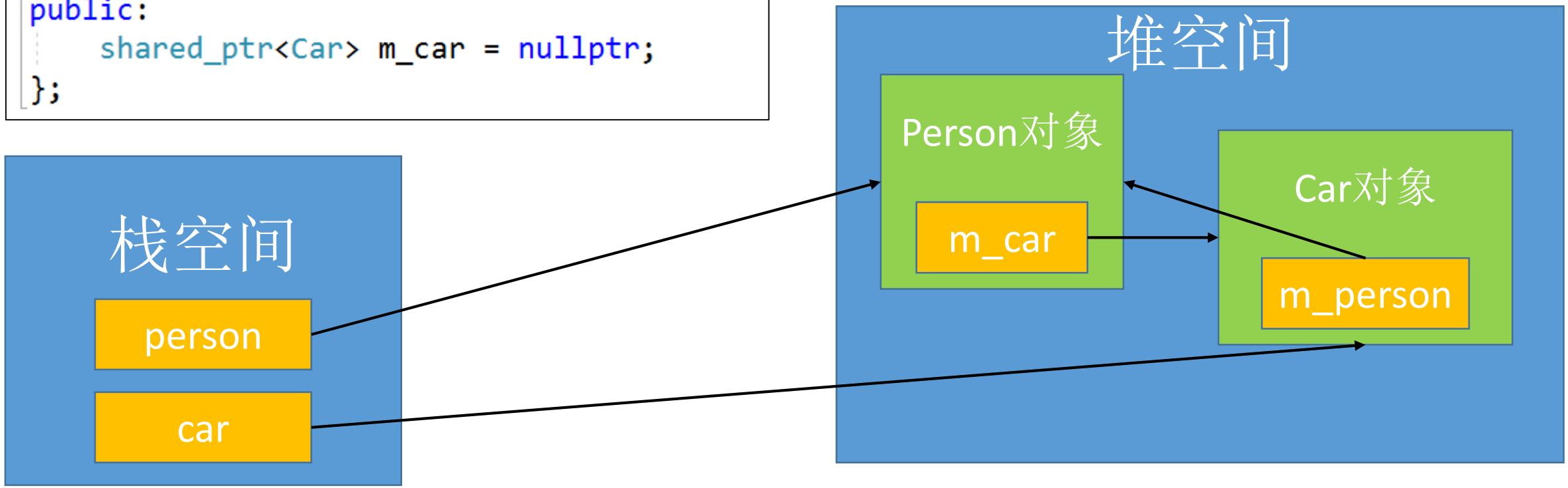
思考下面代码有没有问题

```
Person *p = new Person(); // Person()

{
    shared_ptr<Person> p1(p);
} // ~Person()

{
    shared_ptr<Person> p2(p);
} // ~Person()
```

shared_ptr的循环引用

```
class Car {
public:
    shared_ptr<Person> m_person = nullptr;
};

class Person {
public:
    shared_ptr<Car> m_car = nullptr;
};
```

```
shared_ptr<Person> person(new Person());
shared_ptr<Car> car(new Car());
person->m_car = car;
car->m_person = person;
```



weak_ptr

- weak_ptr会对一个对象产生弱引用

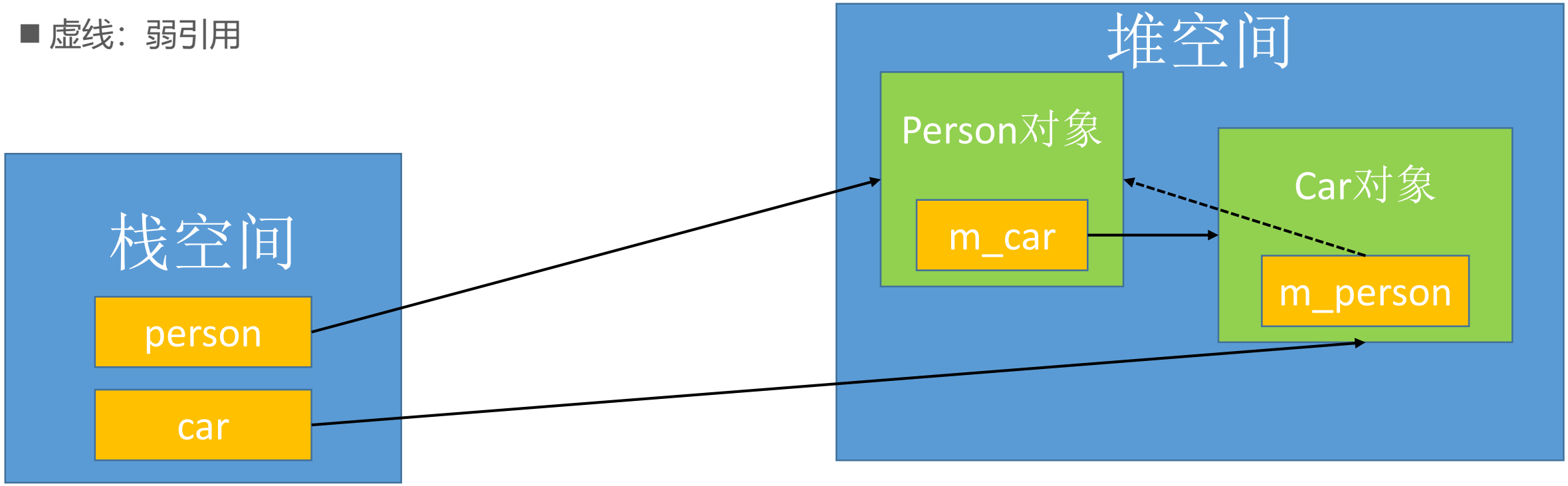
- weak_ptr可以指向对象解决shared_ptr的循环引用问题

```
class Car {
public:
    weak_ptr<Person> m_person;
};

class Person {
public:
    shared_ptr<Car> m_car = nullptr;
};
```

```
shared_ptr<Person> person(new Person());
shared_ptr<Car> car(new Car());
person->m_car = car;
car->m_person = person;
```

weak_ptr解决循环引用

- 实线：强引用
- 虚线：弱引用



unique_ptr

- unique_ptr也会对一个对象产生强引用，它可以确保同一时间只有1个指针指向对象
- 当unique_ptr销毁时（作用域结束时），其指向的对象也就自动销毁了
- 可以使用std::move函数转移unique_ptr的所有权

```
// ptr1强引用着Person对象
unique_ptr<Person> ptr1(new Person());
// 转移之后，ptr2强引用着Person对象
unique_ptr<Person> ptr2 = std::move(ptr1);
```