

# Lambda表达式

## ■ Lambda表达式

□ 有点类似于JavaScript中的闭包、iOS中的Block，本质就是函数

□ 完整结构：[capture list] (params list) mutable exception-> return type { function body }

✓ capture list: 捕获外部变量列表

✓ params list: 形参列表，不能使用默认参数，不能省略参数名

✓ mutable: 用来说用是否可以修改捕获的变量

✓ exception: 异常设定

✓ return type: 返回值类型

✓ function body: 函数体

□ 有时可以省略部分结构

✓ [capture list] (params list) -> return type {function body}

✓ [capture list] (params list) {function body}

✓ [capture list] {function body}

# Lambda表达式 - 示例

```
int (*p1)(int, int) = [] (int v1, int v2) -> int {  
    return v1 + v2;  
};  
cout << p1(10, 20) << endl;
```

```
auto p3 = [](int v1, int v2) { return v1 - v2; } (20, 10);  
cout << p3 << endl;
```

```
int exec(int a, int b, int(*func)(int, int)) {  
    if (func == nullptr) return 0;  
    return func(a, b);  
}
```

```
cout << exec(20, 10, [] (int v1, int v2) { return v1 + v2; }) << endl;  
cout << exec(20, 10, [] (int v1, int v2) { return v1 - v2; }) << endl;  
cout << exec(20, 10, [] (int v1, int v2) { return v1 * v2; }) << endl;  
cout << exec(20, 10, [] (int v1, int v2) { return v1 / v2; }) << endl;
```

```
auto p2 = [] (int v1, int v2) {  
    return v1 + v2;  
};  
cout << p2(10, 20) << endl;
```

```
auto p4 = [] {  
    cout << "test" << endl;  
};  
p4();
```

# Lambda表达式 - 外部变量捕获

```
int a = 10;
int b = 20;
// 值捕获
auto func = [a, b] {
    cout << a << endl;
    cout << b << endl;
};
a = 11;
b = 22;
func(); // 10 20
```

```
int a = 10;
int b = 20;
// 隐式捕获 (值捕获)
auto func = [=] {
    cout << a << endl;
    cout << b << endl;
};
a = 11;
b = 22;
func(); // 10 20
```

```
int a = 10;
int b = 20;
// a是值捕获, 其余变量是地址捕获
auto func = [&, a] {
    cout << a << endl;
    cout << b << endl;
};
a = 11;
b = 22;
func(); // 10 22
```

```
int a = 10;
int b = 20;
// a是引用 (地址) 捕获, b是值捕获
auto func = [&a, b] {
    cout << a << endl;
    cout << b << endl;
};
a = 11;
b = 22;
func(); // 11 20
```

```
int a = 10;
int b = 20;
// 隐式捕获 (地址捕获)
auto func = [&] {
    cout << a << endl;
    cout << b << endl;
};
a = 11;
b = 22;
func(); // 11 22
```

```
int a = 10;
int b = 20;
// a是地址捕获, 其余变量是值捕获
auto func = [=, &a] {
    cout << a << endl;
    cout << b << endl;
};
a = 11;
b = 22;
func(); // 11 20
```

# Lambda表达式 - mutable

```
int a = 10;  
auto func = [a]() mutable {  
    cout << ++a << endl;  
};  
func(); // 11  
cout << a << endl; // 10
```

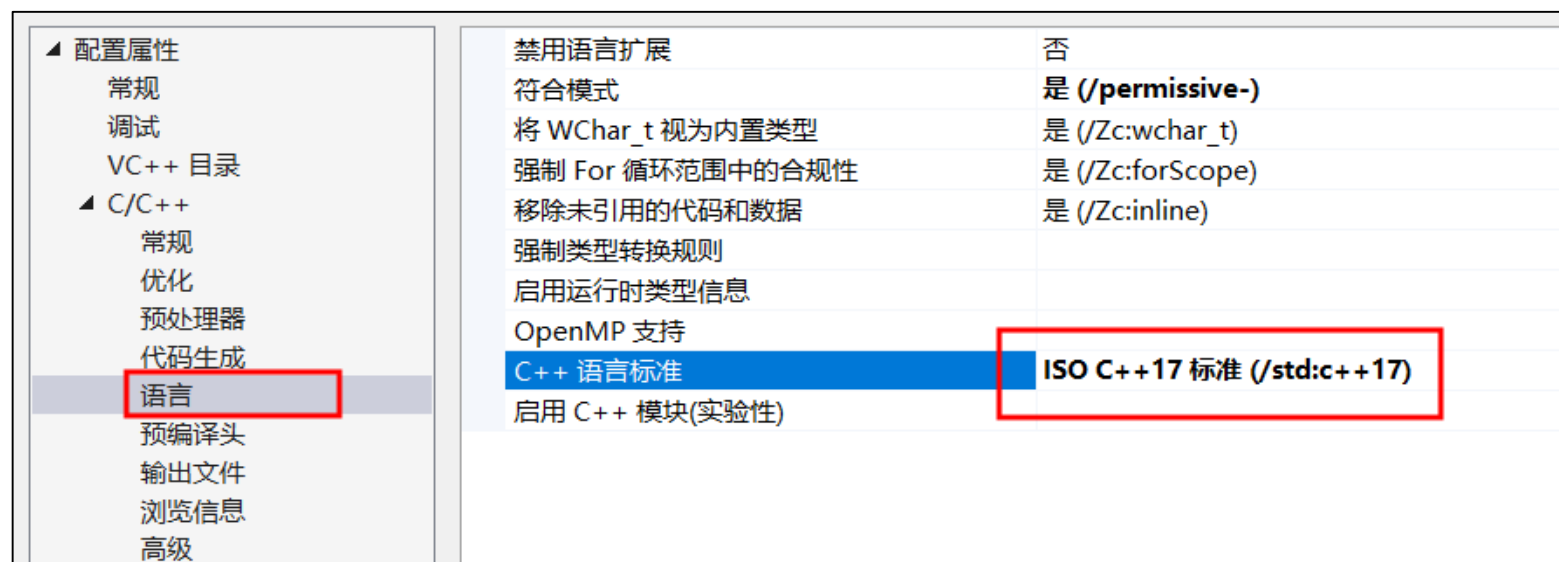
## ■ 泛型Lambda表达式

```
auto func = [](auto v1, auto v2) { return v1 + v2; };  
cout << func(10, 20.5) << endl;
```

## ■ 对捕获的变量进行初始化

```
int a;  
auto func = [a = 10]() {  
    cout << a << endl;  
};  
func();  
// 这里仍然是未初始化  
cout << a << endl;
```

## ■ 设置C++标准



## ■ 可以进行初始化的if、switch语句

```
// 变量a、b的作用域是它所在的if语句、以及其后面的if-else语句
if (int a = 10; a > 10) {
    a = 1;
} else if (int b = 20; a > 5 && b > 10) {
    b = 2;
    a = 2;
} else if (0) {
    b = 3;
    a = 3;
} else {
    b = 4;
    a = 4;
}
```

```
// 变量a的作用域是它所在switch语句
switch (int a = 10; a) {
    case 1:
        break;
    case 5:
        break;
    case 10:
        break;
    default:
        break;
}
```

- 编程过程中的常见错误类型
  - 语法错误
  - 逻辑错误
  - 异常
  - .....



- 异常是一种在程序运行过程中可能会发生的错误（比如内存不够）
- 异常没有被处理，会导致程序终止

```
int divide(int v1, int v2) {  
    if (v2 == 0) throw "不能除以0";  
    return v1 / v2;  
}
```

```
try {  
    int a = 10;  
    int b = 0;  
    int c = divide(a, b);  
} catch (const char *exception) {  
    cout << exception << endl;  
}  
  
cout << "runing..." << endl;
```

```
try {  
    // 被检测的代码  
} catch (异常类型 [变量名]) {  
    // 异常处理代码  
} catch (异常类型 [变量名]) {  
    // 异常处理代码  
} ...
```

- `throw`异常后，会在当前函数中查找匹配的`catch`，找不到就终止当前函数代码，去上一层函数中查找。如果最终都找不到匹配的`catch`，整个程序就会终止

# 异常的抛出声明

- 为了增强可读性和方便团队协作，如果函数内部可能会抛出异常，建议函数声明一下异常类型

```
// 抛出任意可能的异常
void func1() {
}

// 不抛出任何异常
void func2() throw() {
}

// 只抛出int、double类型的异常
void func3() throw(int, double) {
}
```

# 自定义异常类型

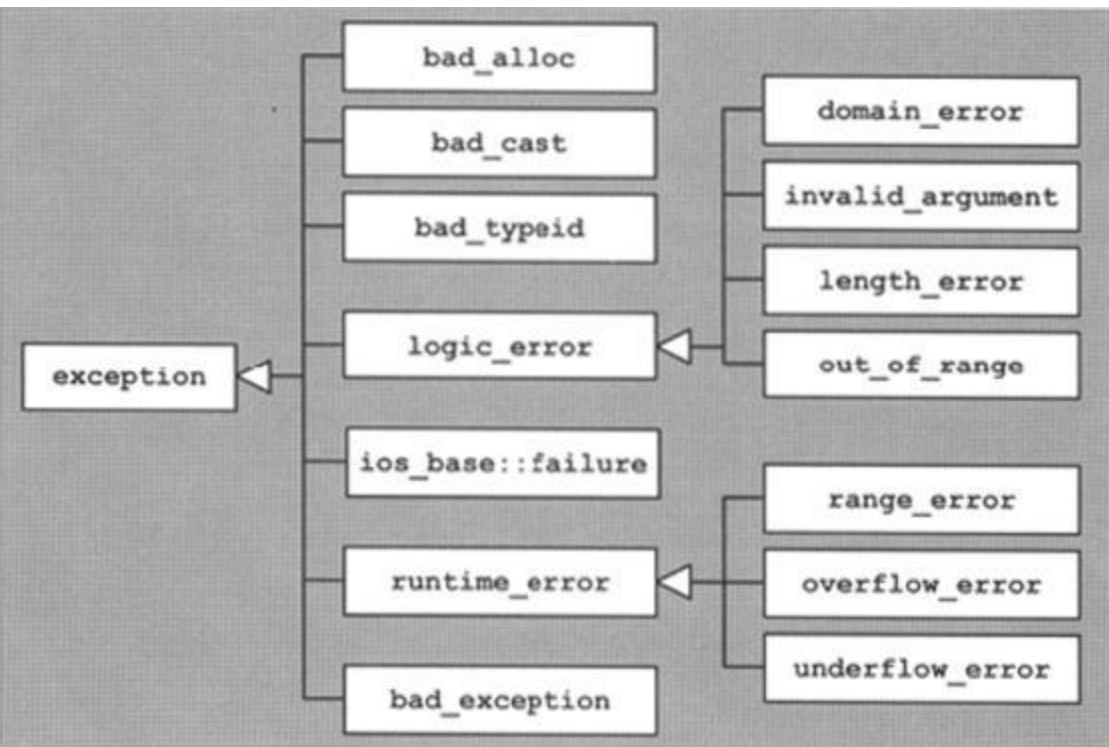
```
class Exception {  
public:  
    virtual string what() const = 0;  
};  
  
class DivideException : public Exception {  
public:  
    string what() const { return "不能除以0"; }  
};  
  
int divide(int v1, int v2) throw(Exception) {  
    if (v2 == 0) throw DivideException();  
    return v1 / v2;  
}
```

```
try {  
    int a = 10;  
    int b = 0;  
    int c = divide(a, b);  
} catch (const Exception &exception) {  
    cout << exception.what() << endl;  
}  
  
cout << "runing..." << endl;
```

# 拦截所有类型的异常

```
try {  
    int a = 10;  
    int b = 0;  
    int c = divide(a, b);  
} catch (...) {  
    cout << "出现异常" << endl;  
}
```

# 标准异常 (std)



异常	描述
<code>std::exception</code>	该异常是所有标准 C++ 异常的父类。
<code>std::bad_alloc</code>	该异常可以通过 <b>new</b> 抛出。
<code>std::bad_cast</code>	该异常可以通过 <b>dynamic_cast</b> 抛出。
<code>std::bad_exception</code>	这在处理 C++ 程序中无法预期的异常时非常有用。
<code>std::bad_typeid</code>	该异常可以通过 <b>typeid</b> 抛出。
<code>std::logic_error</code>	理论上可以通过读取代码来检测到的异常。
<code>std::domain_error</code>	当使用了一个无效的数学域时，会抛出该异常。
<code>std::invalid_argument</code>	当使用了无效的参数时，会抛出该异常。
<code>std::length_error</code>	当创建了太长的 <code>std::string</code> 时，会抛出该异常。
<code>std::out_of_range</code>	该异常可以通过方法抛出，例如 <code>std::vector</code> 和 <code>std::bitset&lt;&gt;::operator[]()</code> 。
<code>std::runtime_error</code>	理论上不可以通过读取代码来检测到的异常。
<code>std::overflow_error</code>	当发生数学上溢时，会抛出该异常。
<code>std::range_error</code>	当尝试存储超出范围的值时，会抛出该异常。
<code>std::underflow_error</code>	当发生数学下溢时，会抛出该异常。

# 标准异常 (std)

```
try {  
    int size = 999999;  
    for (size_t i = 0; i < size; i++) {  
        int *p = new int[size];  
    }  
} catch (std::bad_alloc exception) {  
    cout << exception.what() << endl;  
}  
  
cout << "runing..." << endl;
```

# 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

# 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）

# 思考下面代码有没有问题

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

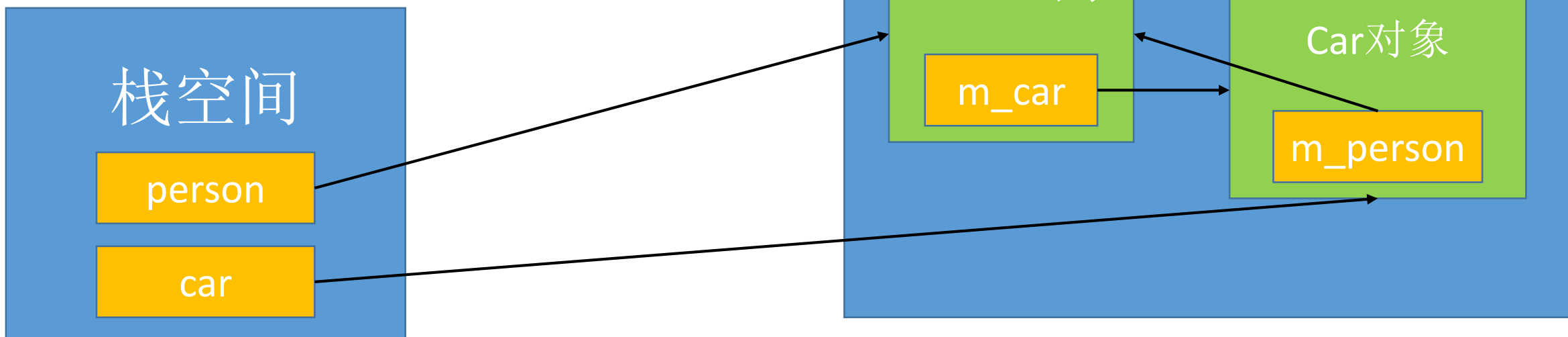
{
    shared_ptr<Person> p1(p);
} // ~Person()

{
    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

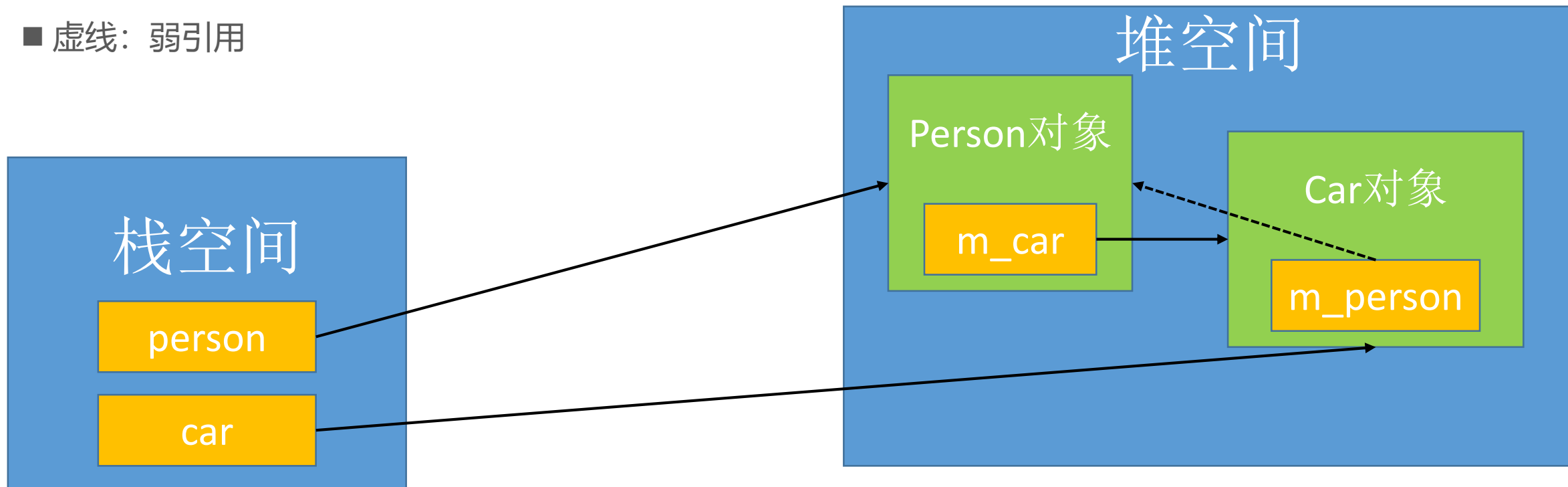
■ weak\_ptr 可以指向对象解决 shared\_ptr 的循环引用问题

```
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);
```