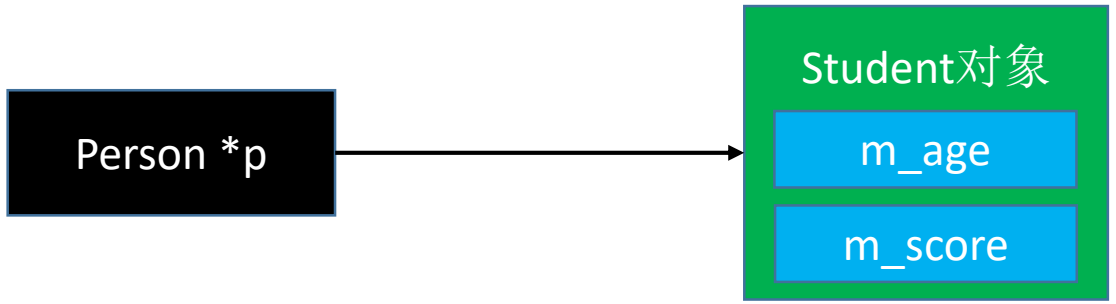
父类指针、子类指针

- 父类指针可以指向子类对象，是安全的，开发中经常用到（继承方式必须是public）

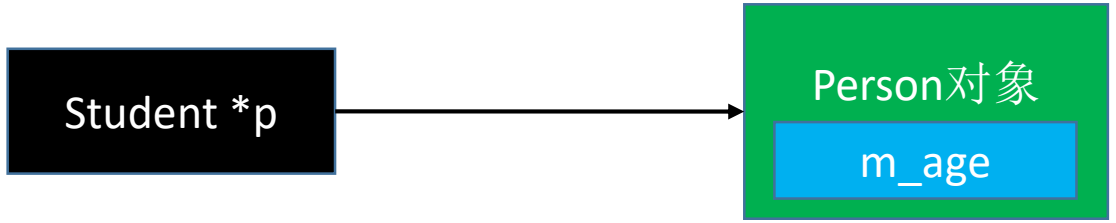
- 子类指针指向父类对象是不安全的

```
struct Person {
    int m_age;
};

struct Student : Person {
    int m_score;
};
```

```
Person *p = new Student();
p->m_age = 10;
```




```
Student *p = (Student *) new Person();
p->m_age = 10;
p->m_score = 100;
```



小码哥教育 SEEMYGO

多态

- 默认情况下，编译器只会根据指针类型调用对应的函数，不存在多态

- 多态是面向对象非常重要的一个特性
  - 同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果
  - 在运行时，可以识别出真正的对象类型，调用对应子类中的函数

- 多态的要素
  - 子类重写父类的成员函数（override）
  - 父类指针指向子类对象
  - 利用父类指针调用重写的成员函数

小码哥教育 SEEMYGO

虚函数

- C++中的多态通过虚函数（virtual function）来实现
- 虚函数：被virtual修饰的成员函数
- 只要在父类中声明为虚函数，子类中重写的函数也自动变成虚函数（也就是说子类中可以省略virtual关键字）

虚表

- 虚函数的实现原理是虚表，这个虚表里面存储着最终需要调用的虚函数地址，这个虚表也叫虚函数表

```
class Animal {
public:
    int m_age;
    virtual void speak() {
        cout << "Animal::speak()" << endl;
    }
    virtual void run() {
        cout << "Animal::run()" << endl;
    }
};

class Cat : public Animal {
public:
    int m_life;
    void speak() {
        cout << "Cat::speak()" << endl;
    }
    void run() {
        cout << "Cat::run()" << endl;
    }
};
```

```
Animal *cat = new Cat();
cat->m_age = 20;
cat->speak();
cat->run();
```

虚表（x86环境的图）

	内存地址	内存数据			内存地址	内存数据
cat	0x00E69B60	0x00B89B64		虚表	0x00B89B64	0x00B814E7
	0x00E69B61				0x00B89B65	
	0x00E69B62				0x00B89B66	
	0x00E69B63				0x00B89B67	
&m_age	0x00E69B64	20			0x00B89B68	0x00B814CE
	0x00E69B65				0x00B89B69	
	0x00E69B66				0x00B89B6A	
	0x00E69B67				0x00B89B6B	
&m_life	0x00E69B68	0				
	0x00E69B69			Cat::speak的调用地址：0x00B814E7		
	0x00E69B6A					
	0x00E69B6B			Cat::run的调用地址：0x00B814CE		

- 所有的Cat对象（不管在全局区、栈、堆）共用同一份虚表

小码哥教育 SEEMYGO

虚表汇编分析

```
// 调用Cat::speak
// 取出cat指针变量里面存储的地址值
// eax里面存放的是Cat对象的地址值
mov             eax,dword ptr [cat]
// 取出Cat对象的前面4个字节给edx
// edx里面存储的是虚表的地址
mov             edx,dword ptr [eax]
// 取出虚表中的前面4个字节给eax
// eax存放的就是Cat::speak的函数地址
mov             eax,dword ptr [edx]
call            eax
```

```
// 调用Cat::run
// 取出cat指针变量里面存储的地址值
// eax里面存放的是Cat对象的地址值
mov             eax,dword ptr [cat]
// 取出Cat对象的前面4个字节给edx
// edx里面存储的是虚表的地址
mov             edx,dword ptr [eax]
// 取出虚表中的后面4个字节给eax
// eax存放的就是Cat::run的函数地址
mov             eax,dword ptr [edx+4]
call            eax
```

虚表（x86环境的图）

```
class Animal {
public:
    int m_age;
    virtual void speak() {
        cout << "Animal::speak()" << endl;
    }
    virtual void run() {
        cout << "Animal::run()" << endl;
    }
};

class Cat : public Animal {
public:
    int m_life;
    void run() {
        cout << "Cat::run()" << endl;
    }
};
```

```
Animal *cat = new Cat();
cat->m_age = 20;
cat->speak();
cat->run();
```

	内存地址	内存数据
cat	0x00E69B60	0x00B89B64
	0x00E69B61	
	0x00E69B62	
	0x00E69B63	
&m_age	0x00E69B64	20
	0x00E69B65	
	0x00E69B66	
	0x00E69B67	
&m_life	0x00E69B68	0
	0x00E69B69	
	0x00E69B6A	
	0x00E69B6B	

虚表 →

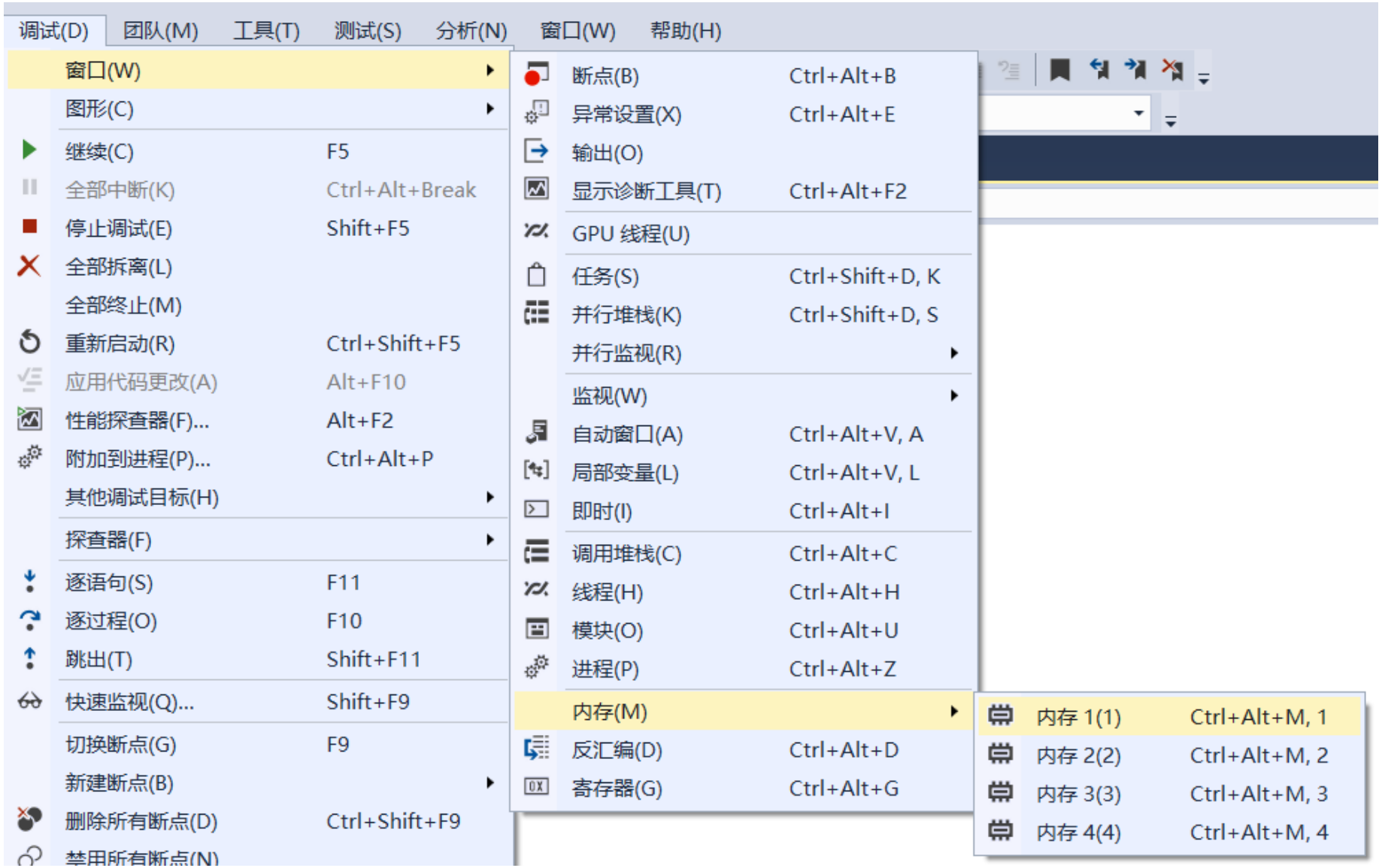
内存地址	内存数据
0x00B89B64	0x00DC14F1
0x00B89B65	
0x00B89B66	
0x00B89B67	
0x00B89B68	0x00DC14CE
0x00B89B69	
0x00B89B6A	
0x00B89B6B	

Animal::speak的调用地址：0x00DC14F1

Cat::run的调用地址：0x00DC14CE

小码哥教育 SEEMYGO

VS的内存窗口



小码哥教育 SEEMYGO

调用父类的成员函数实现

```
class Animal {
public:
    virtual void speak() {
        cout << "Animal::speak()" << endl;
    }
};

class Cat : public Animal {
public:
    void speak() {
        Animal::speak();
        cout << "Cat::speak()" << endl;
    }
};
```

虚析构函数

- 如果存在父类指针指向子类对象的情况，应该将析构函数声明为虚函数（虚析构函数）
  - delete父类指针时，才会调用子类的析构函数，保证析构的完整性

```
class Person {
public:
    virtual void run() {
        cout << "Person::run()" << endl;
    }
    virtual ~Person() {
        cout << "Person::~Person()" << endl;
    }
};

class Student : public Person {
public:
    void run() {
        cout << "Student::run()" << endl;
    }
    ~Student() {
        cout << "Student::~Student()" << endl;
    }
};
```

纯虚函数

- 纯虚函数：没有函数体且初始化为0的虚函数，用来定义接口规范

- 抽象类（Abstract Class）
  - 含有纯虚函数的类，不可以实例化（不可以创建对象）
  - 抽象类也可以包含非纯虚函数、成员变量
  - 如果父类是抽象类，子类没有完全重写纯虚函数，那么这个子类依然是抽象类

```
class Animal {
    virtual void speak() = 0;
    virtual void walk() = 0;
};
```

多继承

■ C++允许一个类可以有多个父类（不建议使用，会增加程序设计复杂度）

```
class Student {
public:
    int m_score;
    void study() {
        cout << "Student::study()" << endl;
    }
};

class Worker {
public:
    int m_salary;
    void work() {
        cout << "Worker::work()" << endl;
    }
};

class Undergraduate : public Student, public Worker {
public:
    int m_grade;
    void play() {
        cout << "Undergraduate::play()" << endl;
    }
};
```

```
Undergraduate ug;
ug.m_score = 100;
ug.m_salary = 2000;
ug.m_grade = 4;
ug.study();
ug.work();
ug.play();
```

		内存地址	内存数据
&ug	&m_score	0x00E69B60	100
		0x00E69B61	
		0x00E69B62	
		0x00E69B63	
	&m_salary	0x00E69B64	2000
		0x00E69B65	
		0x00E69B66	
		0x00E69B67	
	&m_grade	0x00E69B68	4
		0x00E69B69	
		0x00E69B6A	
		0x00E69B6B	

小码哥教育 SEEMYGO

多继承体系下的构造函数调用

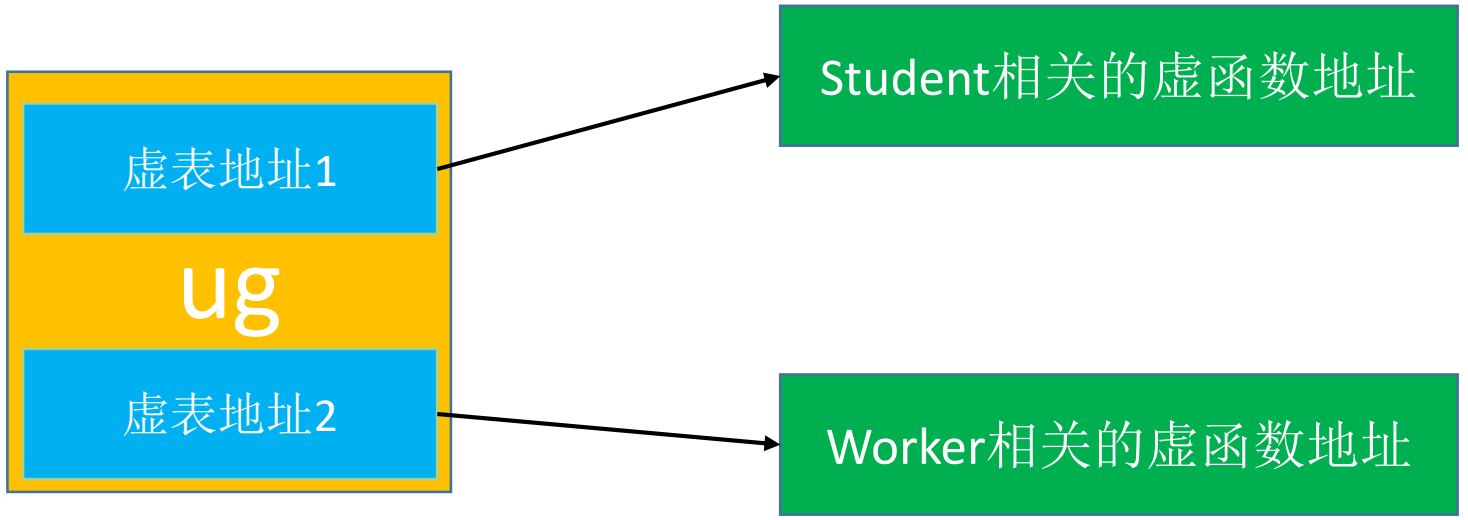
```
class Student {
    int m_score;
public:
    Student(int score) {
        this->m_score = score;
    }
};

class Worker {
    int m_salary;
public:
    Worker(int salary) {
        this->m_salary = salary;
    }
};

class Undergraduate : public Student, public Worker {
public:
    Undergraduate(int score, int salary) :Student(score), Worker(salary) {
    }
};
```

多继承-虚函数

■ 如果子类继承的多个父类都有虚函数，那么子类对象就会产生对应的多张虚表

```
class Student {
public:
    virtual void study() {
        cout << "Student::study()" << endl;
    }
};

class Worker {
public:
    virtual void work() {
        cout << "Worker::work()" << endl;
    }
};

class Undergraduate : public Student, public Worker {
public:
    void study() {
        cout << "Undergraduate::study()" << endl;
    }
    void work() {
        cout << "Undergraduate::work()" << endl;
    }
    void play() {
        cout << "Undergraduate::play()" << endl;
    }
};
```

```
Undergraduate ug;
```



小码哥教育 SEEMYGO

同名函数

```
class Student {
public:
    void eat() {
        cout << "Student::eat()" << endl;
    }
};

class Worker {
public:
    void eat() {
        cout << "Worker::eat()" << endl;
    }
};

class Undergraduate : public Student, public Worker {
public:
    void eat() {
        cout << "Undergraduate::eat()" << endl;
    }
};
```

```
Undergraduate ug;
ug.Student::eat(); // Student::eat()
ug.Worker::eat(); // Worker::eat()
ug.Undergraduate::eat(); // Undergraduate::eat()
ug.eat(); // Undergraduate::eat()
```

同名成员变量

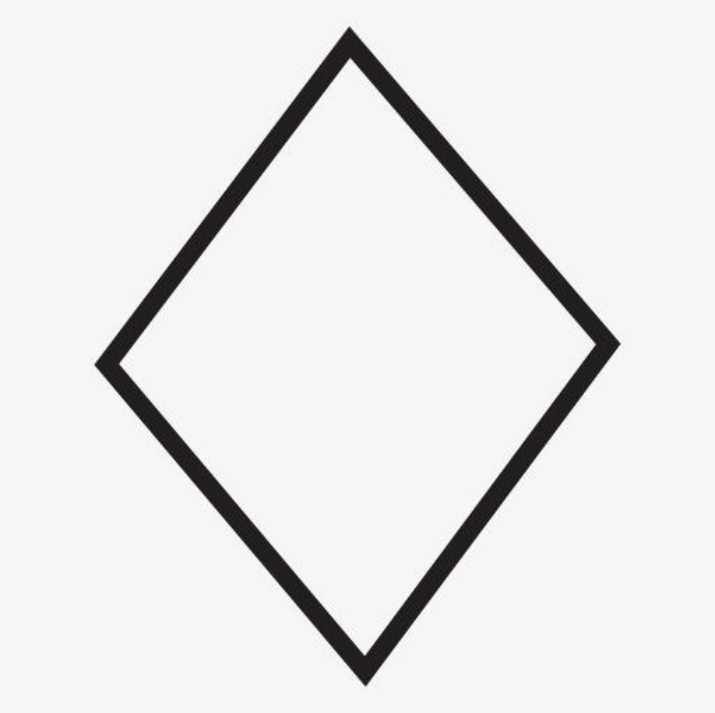
```
class Student {
public:
    int m_age;
};

class Worker {
public:
    int m_age;
};

class Undergraduate : public Student, public Worker {
public:
    int m_age;
};
```

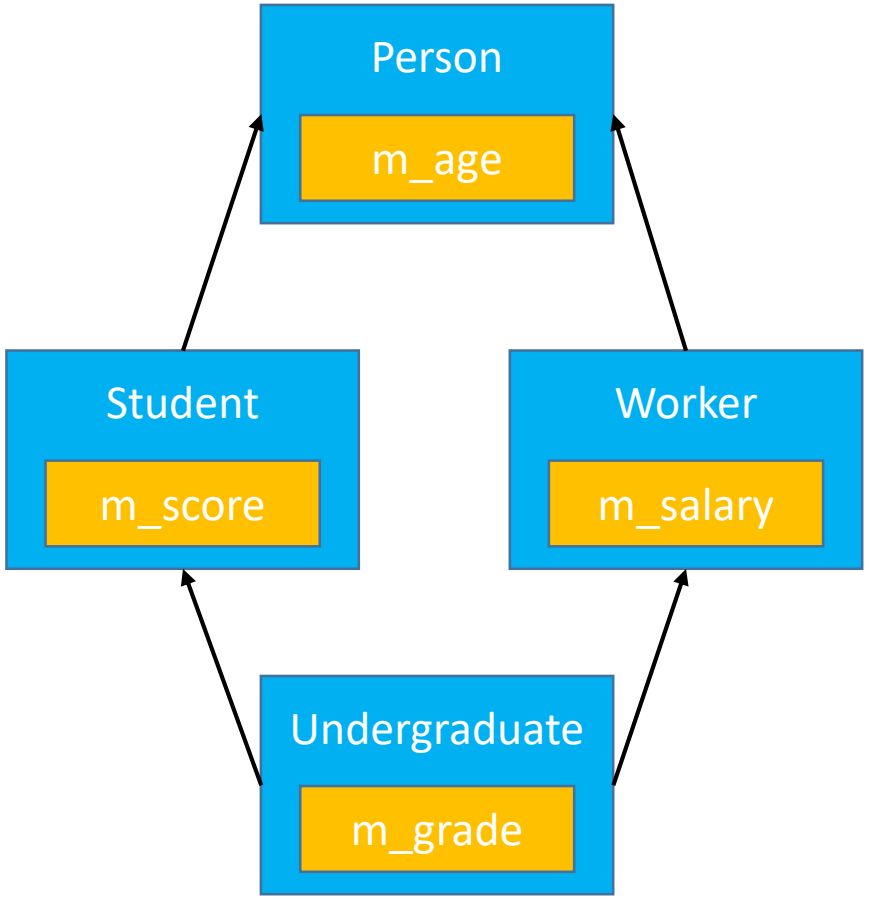
```
Undergraduate ug;
ug.m_age = 10;
ug.Student::m_age = 20;
ug.Worker::m_age = 30;

cout << ug.Undergraduate::m_age << endl; // 10
cout << ug.Student::m_age << endl; // 20
cout << ug.Worker::m_age << endl; // 30
```

	内存地址	内存数据
&Student::m_age	0x00E69B60	20
	0x00E69B61	
	0x00E69B62	
	0x00E69B63	
&Worker::m_age	0x00E69B64	30
	0x00E69B65	
	0x00E69B66	
	0x00E69B67	
&Undergraduate::m_age	0x00E69B68	10
	0x00E69B69	
	0x00E69B6A	
	0x00E69B6B	

菱形继承

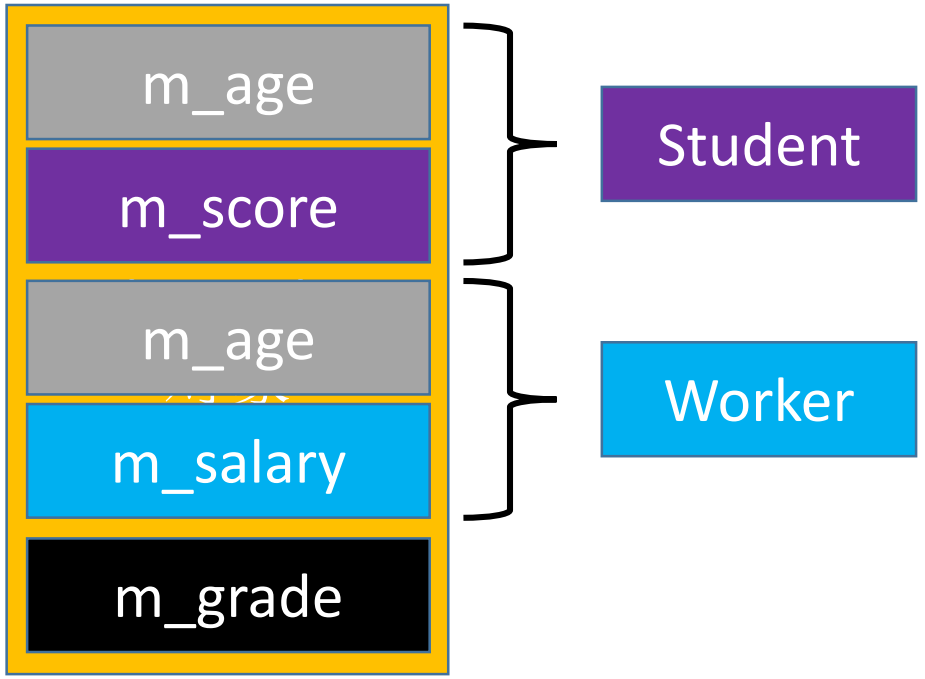
- 菱形继承带来的问题
  - 最底下子类从基类继承的成员变量冗余、重复
  - 最底下子类无法访问基类的成员，有二义性

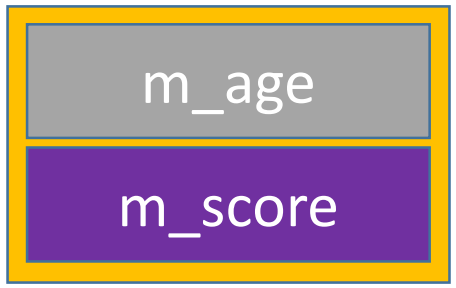


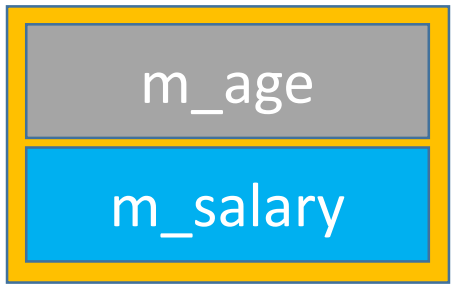


菱形继承

```
class Person {
    int m_age;
};

class Student : public Person {
    int m_score;
};

class Worker : public Person {
    int m_salary;
};

class Undergraduate : public Student, public Worker {
    int m_grade;
};
```

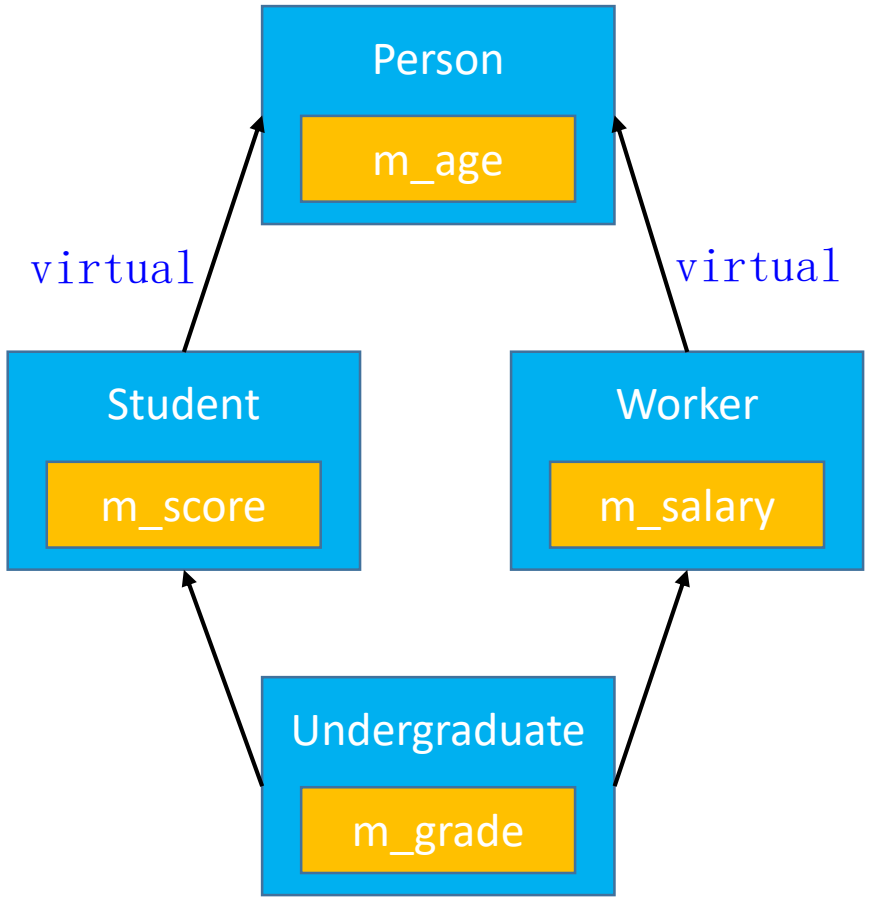






虚继承

- 虚继承可以解决菱形继承带来的问题
- Person类被称为虚基类

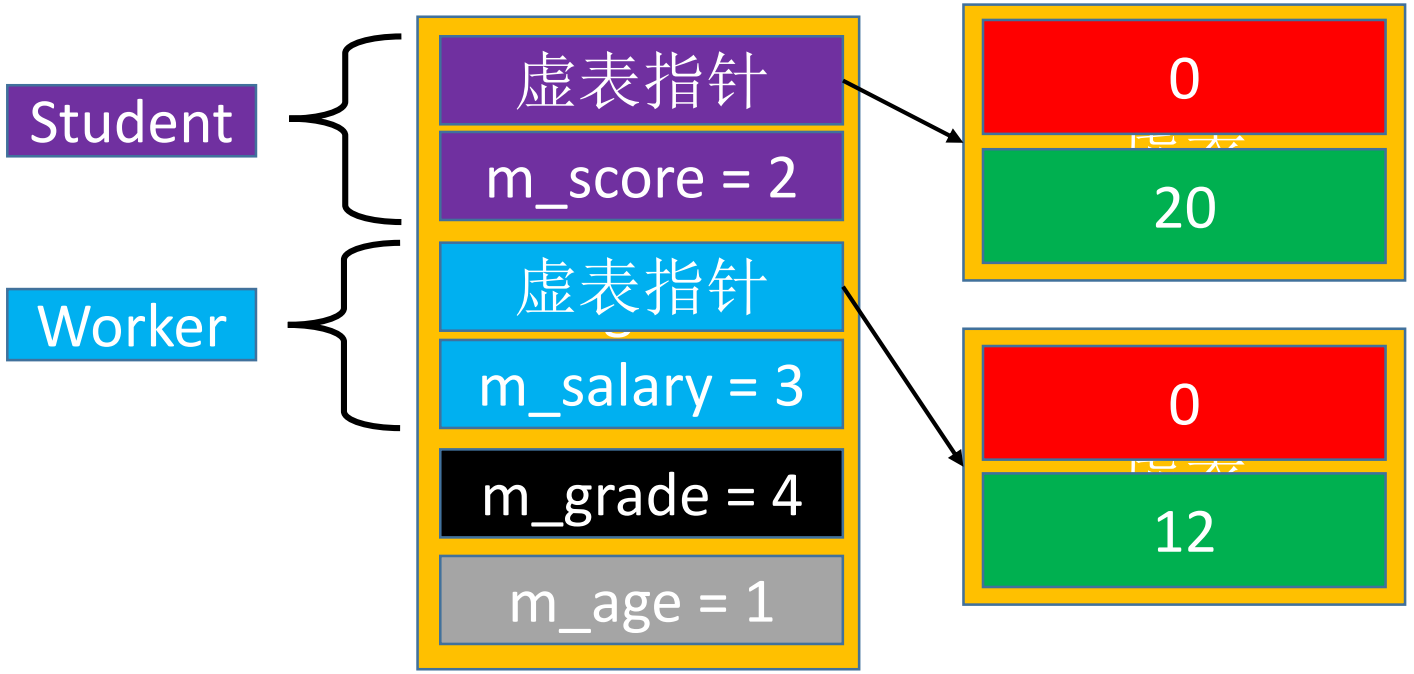


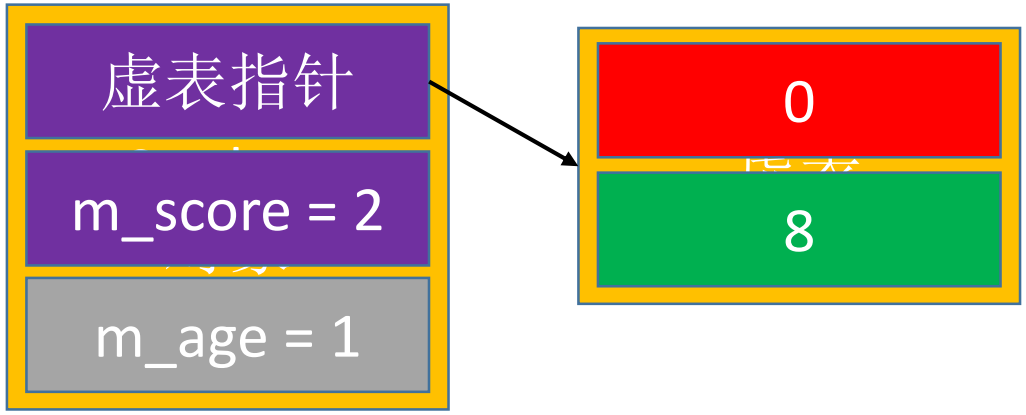
虚继承

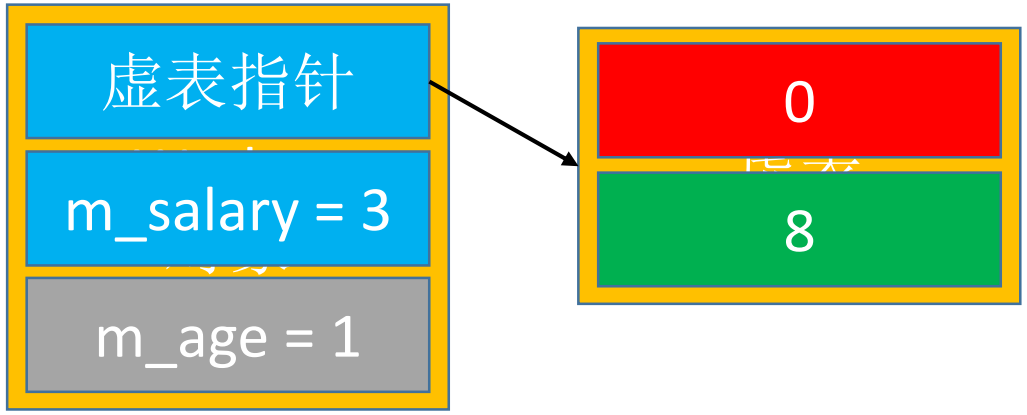
虚表指针与本类起始的偏移量（一般是0）

虚基类第一个成员变量与本类起始的偏移量

```
class Person {
    int m_age = 1;
};

class Student : virtual public Person {
    int m_score = 2;
};

class Worker : virtual public Person {
    int m_salary = 3;
};

class Undergraduate : public Student, public Worker {
    int m_grade = 4;
};
```







小码哥教育 SEEMYGO

静态成员（static）

- 静态成员：被static修饰的成员变量\函数
  - 可以通过对象（对象.静态成员）、对象指针（对象指针->静态成员）、类访问（类名::静态成员）
- 静态成员变量
  - 存储在数据段（全局区，类似于全局变量），整个程序运行过程中只有一份内存
  - 对比全局变量，它可以设定访问权限（public、protected、private），达到局部共享的目的
  - 必须初始化，必须在类外面初始化，初始化时不能带static，如果类的声明和实现分离（在实现.cpp中初始化）
- 静态成员函数
  - 内部不能使用this指针（this指针只能用在非静态成员函数内部）
  - 不能是虚函数（虚函数只能是非静态成员函数）
  - 内部不能访问非静态成员变量\函数，只能访问静态成员变量\函数
  - 非静态成员函数内部可以访问静态成员变量\函数
  - 构造函数、析构函数不能是静态
  - 当声明和实现分离时，实现部分不能带static

小码哥教育 SEEMYGO

静态成员

```
class Car {
    int m_price;
    static int ms_count;
public:
    static int getCount() {
        return ms_count;
    }
    Car(int price = 0) :m_price(price) {
        ms_count++;
    }
    ~Car() {
        ms_count--;
    }
};

int Car::ms_count = 0;
```

小码哥教育 SEEMYGO

静态成员经典应用 – 单例模式

```
class Rocket {
private:
    Rocket() {}
    ~Rocket() {}
    static Rocket *ms_rocket;
public:
    static Rocket *sharedRocket() {
        // 这里要考虑多线程安全
        if (ms_rocket == NULL) {
            ms_rocket = new Rocket();
        }
        return ms_rocket;
    }

    static void deleteRocket() {
        // 这里要考虑多线程安全
        if (ms_rocket != NULL) {
            delete ms_rocket;
            ms_rocket = NULL;
        }
    }
};
```

const成员

- const成员：被const修饰的成员变量、非静态成员函数

- const成员变量
  - 必须初始化（类内部初始化），可以在声明的时候直接初始化赋值
  - 非static的const成员变量还可以在初始化列表中初始化

- const成员函数（非静态）
  - const关键字写在参数列表后面，函数的声明和实现都必须带const
  - 内部不能修改非static成员变量
  - 内部只能调用const成员函数、static成员函数
  - 非const成员函数可以调用const成员函数
  - const成员函数和非const成员函数构成重载
  - 非const对象（指针）优先调用非const成员函数
  - const对象（指针）只能调用const成员函数、static成员函数

```
class Car {
    const int mc_wheelsCount = 20;
public:
    Car() :mc_wheelsCount(10) { }

    void run() const {
        cout << " run()" << endl;
    }
};
```

小码哥教育 SEEMYGO

引用类型成员

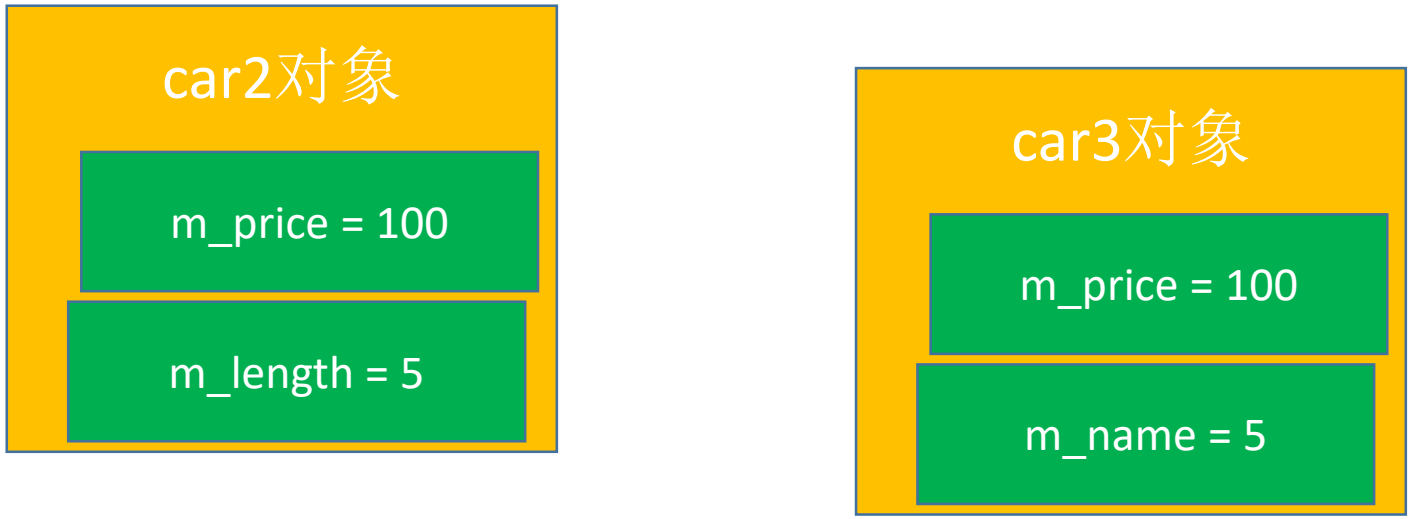
- 引用类型成员变量必须初始化（不考虑static情况）
  - 在声明的时候直接初始化
  - 通过初始化列表初始化

```
class Car {
    int age;
    int &m_price = age;
public:
    Car(int &price) :m_price(price) { }
};
```

拷贝构造函数（Copy Constructor）

- 拷贝构造函数是构造函数的一种
- 当利用已存在的对象创建一个新对象时（类似于拷贝），就会调用新对象的拷贝构造函数进行初始化
- 拷贝构造函数的格式是固定的，接收一个const引用作为参数

```
class Car {
    int m_price;
public:
    Car(int price = 0) :m_price(price) { }
    Car(const Car &car) {
        this->m_price = car.m_price;
    }
};
```


car2对象
m_price = 100
m_length = 5
car3对象
m_price = 100
m_name = 5

小码哥教育 SEEMYGO

调用父类的拷贝构造函数

```
class Person {
    int m_age;
public:
    Person(int age) :m_age(age) { }
    Person(const Person &person) :m_age(person.m_age) { }
};

class Student : public Person {
    int m_score;
public:
    Student(int age, int score) :Person(age), m_score(score) { }
    Student(const Student &student) :Person(student), m_score(student.m_score) { }
};
```

小码哥教育 SEEMYGO

拷贝构造函数

- car2、car3都是通过拷贝构造函数初始化的，car、car4是通过非拷贝构造函数初始化

```
Car car(100, "BMW 730Li");
Car car2 = car;
Car car3(car2);
Car car4;
car4 = car3;
```

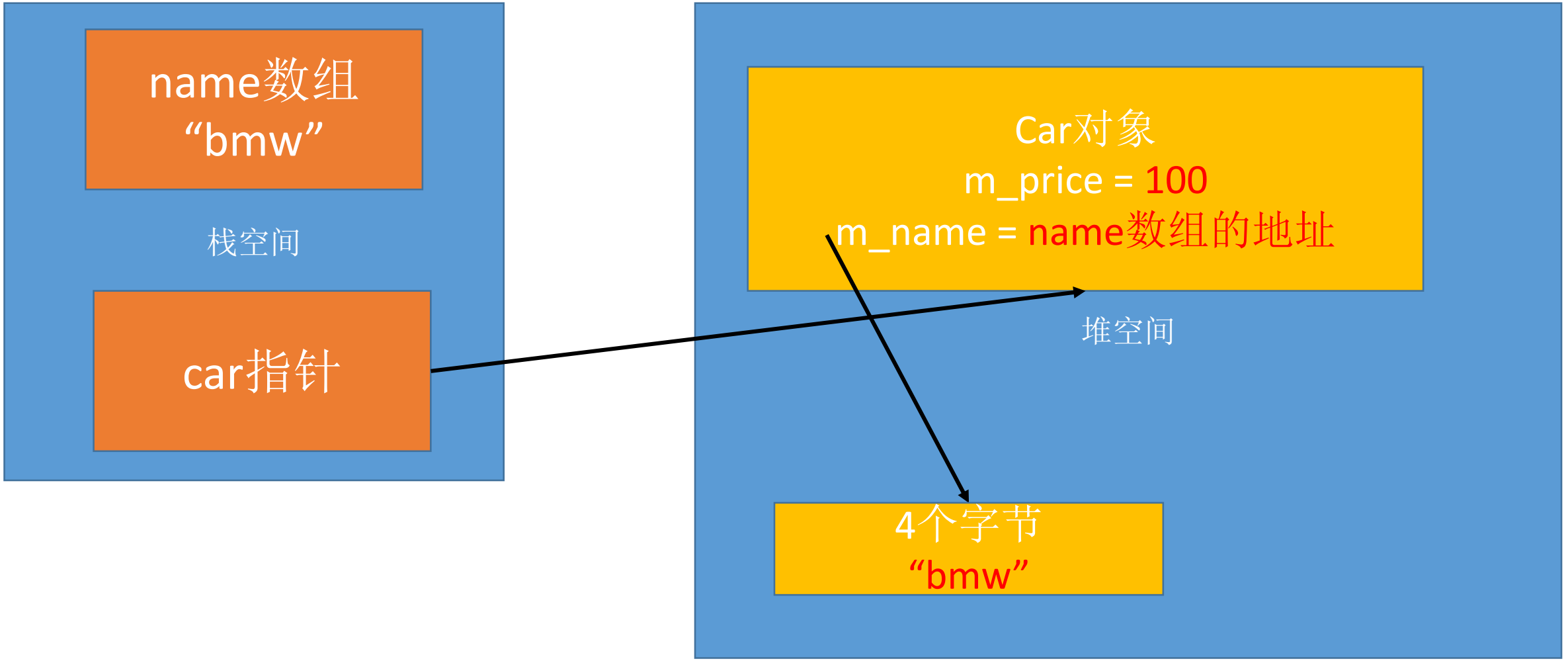
- car4 = car3是一个赋值操作（默认是浅复制），并不会调用拷贝构造函数

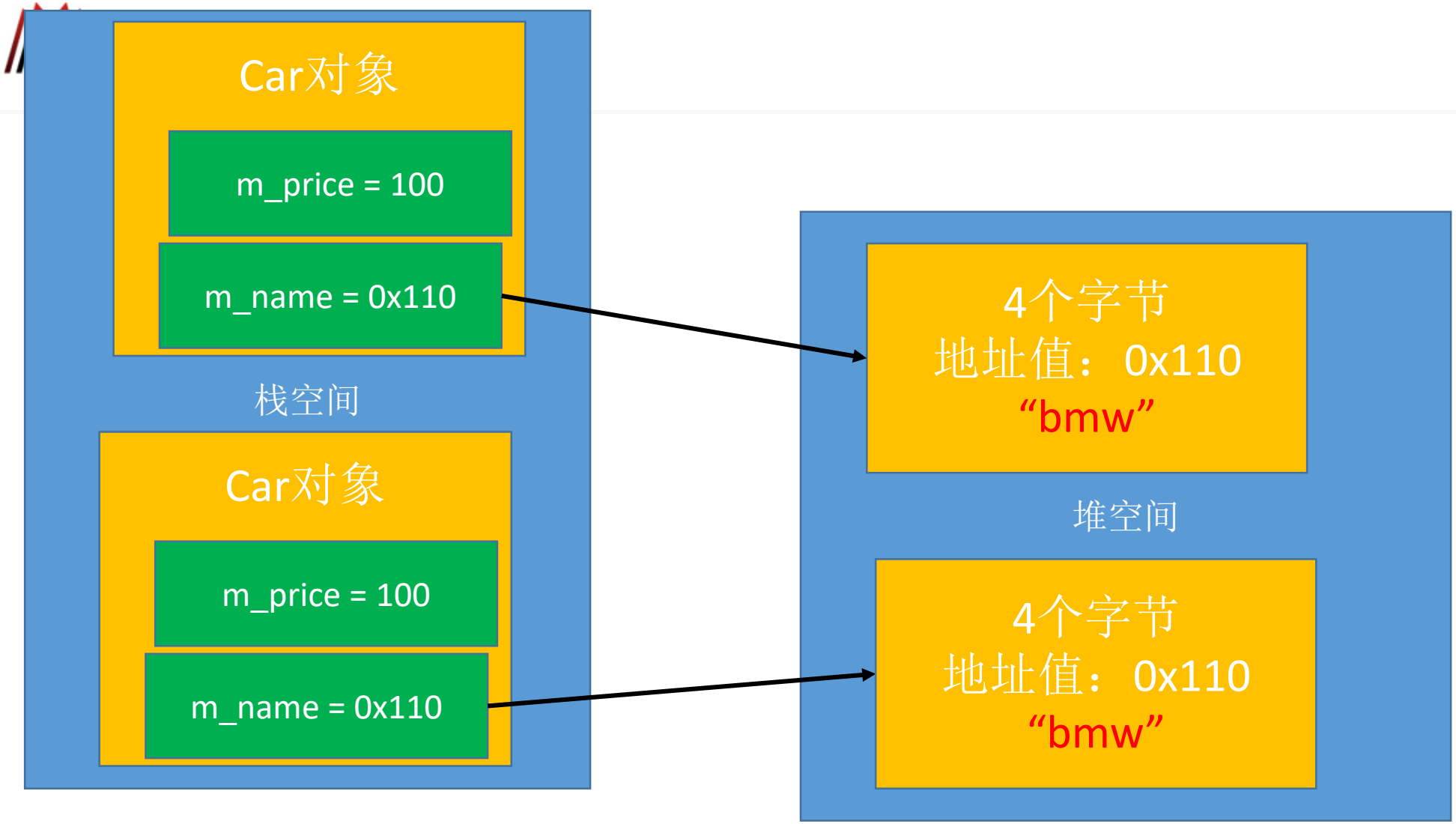
小码哥教育 SEEMYGO

浅拷贝、深拷贝

- 编译器默认的提供的拷贝是浅拷贝（shallow copy）
  - 将一个对象中所有成员变量的值拷贝到另一个对象
  - 如果某个成员变量是个指针，只会拷贝指针中存储的地址值，并不会拷贝指针指向的内存空间
  - 可能会导致堆空间多次free的问题

- 如果需要实现深拷贝（deep copy），就需要自定义拷贝构造函数
  - 将指针类型的成员变量所指向的内存空间，拷贝到新的内存空间


name数组
“bmw”
栈空间
car指针
Car对象
m_price = 100
m_name = name数组的地址
堆空间
4个字节
“bmw”


Car对象
m_price = 100
m_name = 0x110
栈空间
Car对象
m_price = 100
m_name = 0x110
4个字节
地址值：0x110
“bmw”
堆空间
4个字节
地址值：0x110
“bmw”

小码哥教育 SEEMYGO

深拷贝示例

```
class Car {
    int m_price;
    char *m_name;
    void copyName(const char *name) {
        if (name == NULL) return;

        this->m_name = new char[strlen(name) + 1]{};
        strcpy(this->m_name, name);
    }
public:
    Car(int price = 0, const char *name = NULL) :m_price(price) {
        copyName(name);
    }
    Car(const Car &car) {
        this->m_price = car.m_price;

        copyName(car.m_name);
    }
    ~Car() {
        if (this->m_name != NULL) {
            delete[] this->m_name;
        }
    }
};
```

对象型参数和返回值

- 使用对象类型作为函数的参数或者返回值，可能会产生一些不必要的中间对象

```
class Car {
    int m_price;
public:
    Car() { }
    Car(int price) :m_price(price) { }
    Car(const Car &car) :m_price(car.m_price) { }
};

void test1(Car car) {

}

Car test2() {
    Car car(20); // Car(int price)
    return car;
}
```

```
Car car1(10); // Car(int price)
test1(car1); // Car(const Car &car)

Car car2 = test2(); // Car(const Car &car)

Car car3(30); // Car(int price)
car3 = test2(); // Car(const Car &car)
```

匿名对象（临时对象）

- 匿名对象：没有变量名、没有被指针指向的对象，用完后马上调用析构

```
void test1(Car car) {

}

Car test2() {
    return Car(60);
}
```

```
Car(10); // Car(int price)
Car(20).display(); // Car(int price)

Car car1 = Car(30); // Car(int price)

test1(Car(40)); // Car(int price)

Car car3(50); // Car(int price)
car3 = test2(); // Car(int price)
```

隐式构造

- C++中存在隐式构造的现象：某些情况下，会隐式调用单参数的构造函数

```
void test1(Car car) {

}

Car test2() {
    return 70;
}
```

```
Car car1 = 10; // Car(int price)

Car car2(20); // Car(int price)
car2 = 30; // Car(int price)，匿名对象

test1(40); // Car(int price)

Car car3 = test2(); // Car(int price)
```

- 可以通过关键字explicit禁止掉隐式构造

```
class Car {
    int m_price;
public:
    Car() { }
    explicit Car(int price) :m_price(price) { }
    Car(const Car &car) :m_price(car.m_price) { }
};
```

小码哥教育 SEEMYGO

编译器自动生成的构造函数

- C++的编译器在某些特定的情况下，会给类自动生成无参的构造函数，比如
  - 成员变量在声明的同时进行了初始化
  - 有定义虚函数
  - 虚继承了其他类
  - 包含了对象类型的成员，且这个成员有构造函数（编译器生成或自定义）
  - 父类有构造函数（编译器生成或自定义）

- 总结一下
  - 对象创建后，需要做一些额外操作时（比如内存操作、函数调用），编译器一般都会为其自动生成无参的构造函数

小码哥教育 SEEMYGO

友元

- 友元包括友元函数和友元类
- 如果将函数A（非成员函数）声明为类C的友元函数，那么函数A就能直接访问类C对象的所有成员
- 如果将类A声明为类C的友元类，那么类A的所有成员函数都能直接访问类C对象的所有成员
- 友元破坏了面向对象的封装性，但在某些频繁访问成员变量的地方可以提高性能

```
class Point {
    friend Point add(const Point &, const Point &);
    friend class Math;
private:
    int m_x;
    int m_y;
public:
    Point() { }
    Point(int x, int y) :m_x(x), m_y(y) { }
};
```

```
Point add(const Point &p1, const Point &p2) {
    return Point(p1.m_x + p2.m_x, p1.m_y + p2.m_y);
}

class Math {
    void test() {
        Point point;
        point.m_x = 10;
        point.m_y = 20;
    }

    static void test2() {
        Point point;
        point.m_x = 10;
        point.m_y = 20;
    }
};
```

内部类

- 如果将类A定义在类C的内部，那么类A就是一个内部类（嵌套类）
- 内部类的特点
  - 支持public、protected、private权限
  - 成员函数可以直接访问其外部类对象的所有成员（反过来则不行）
  - 成员函数可以直接不带类名、对象名访问其外部类的static成员
  - 不会影响外部类的内存布局
  - 可以在外部类内部声明，在外部类外面进行定义

```
class Point {
    static void test1() {
        cout << "Point::test1()" << endl;
    }
    static int ms_test2;
    int m_x;
    int m_y;
public:
    class Math {
    public:
        void test3() {
            cout << "Point::Math::test3()" << endl;
            test1();
            ms_test2 = 10;

            Point point;
            point.m_x = 10;
            point.m_y = 20;
        }
    };
};
```

小码哥教育 SEEMYGO

内部类 – 声明和实现分离

```
class Point {
    class Math {
        void test();
    };
};

void Point::Math::test() {

}
```

```
class Point {
    class Math;
};

class Point::Math {
    void test() {

    }
};
```

```
class Point {
    class Math;
};

class Point::Math {
    void test();
};

void Point::Math::test() {

}
```

局部类

- 在一个函数内部定义的类，称为局部类
- 局部类的特点
  - 作用域仅限于所在的函数内部
  - 其所有的成员必须定义在类内部，不允许定义static成员变量
  - 成员函数不能直接访问函数的局部变量（static变量除外）

```
int m_age1 = 0;

void test() {
    static int s_age2 = 0;
    int age3 = 0;

    class Point {
        int m_x;
        int m_y;
    public:
        static void display() {
            m_age1 = 10;
            s_age2 = 20;
            age3 = 30;
        }
    };

    Point::display();
}
```