

■ 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	

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) {  
    }  
};
```

- 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;



Student相关的虚函数地址

Worker相关的虚函数地址

```
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;

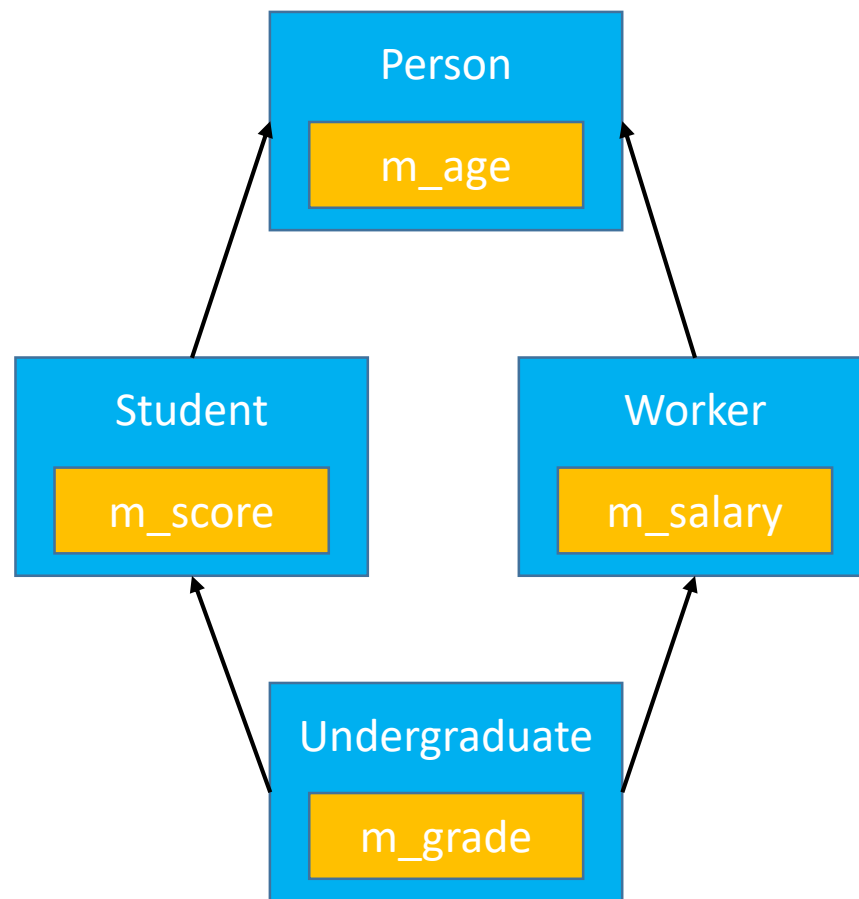
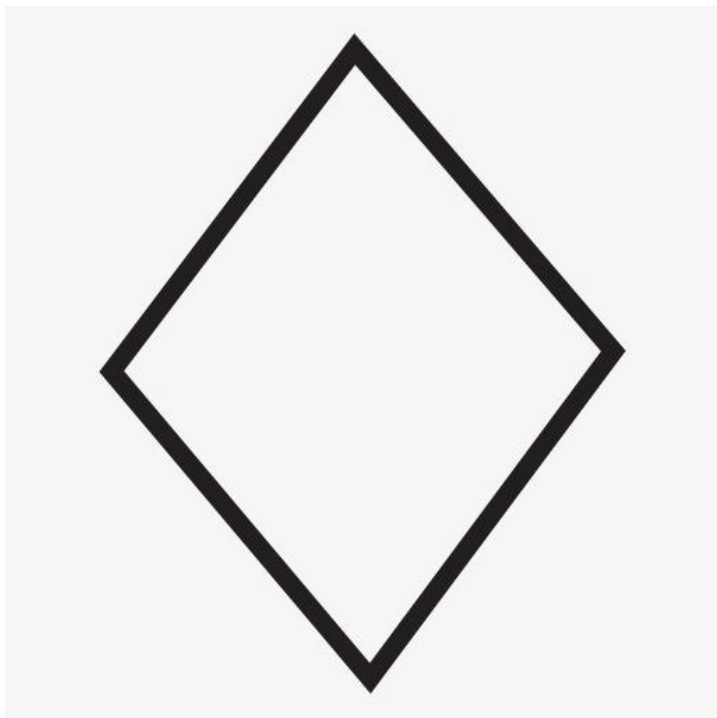
cout << ug.Undergraduate::m_age << endl; // 10
cout << ug.Student::m_age << endl; // 20
cout << ug.Worker::m_age << endl; // 30
```

	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

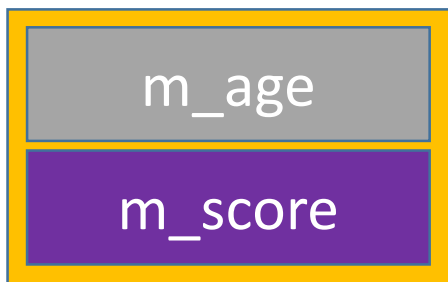
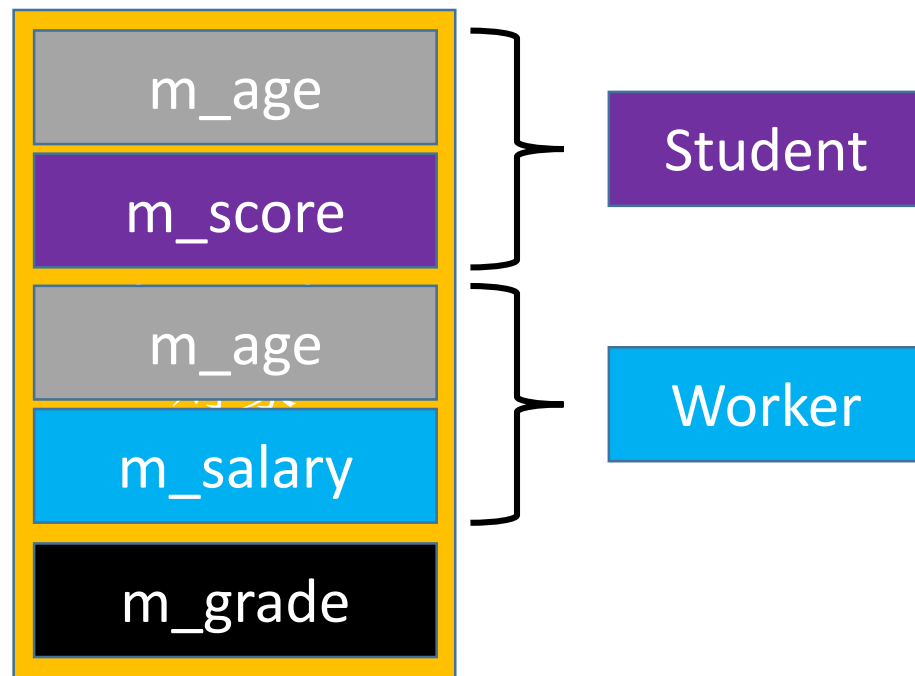
■ 菱形继承带来的问题

- 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



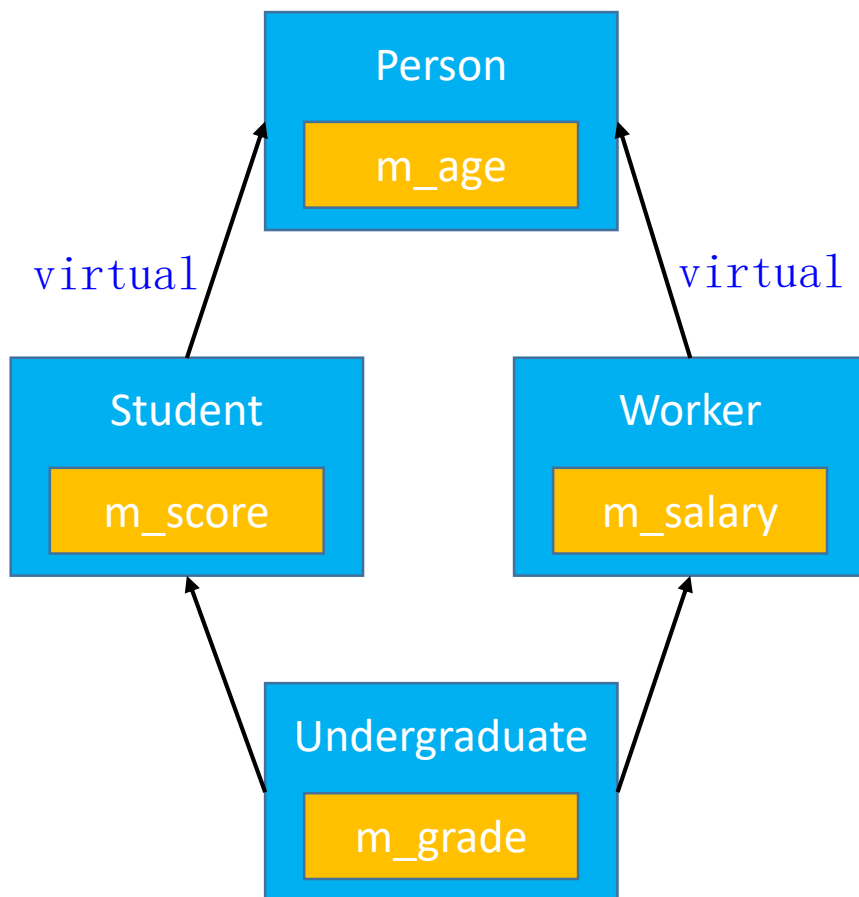
菱形继承

```
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;  
};
```



■ 虚继承可以解决菱形继承带来的问题

■ Person类被称为**虚基类**



虚表指针与本类起始的偏移量（一般是0）

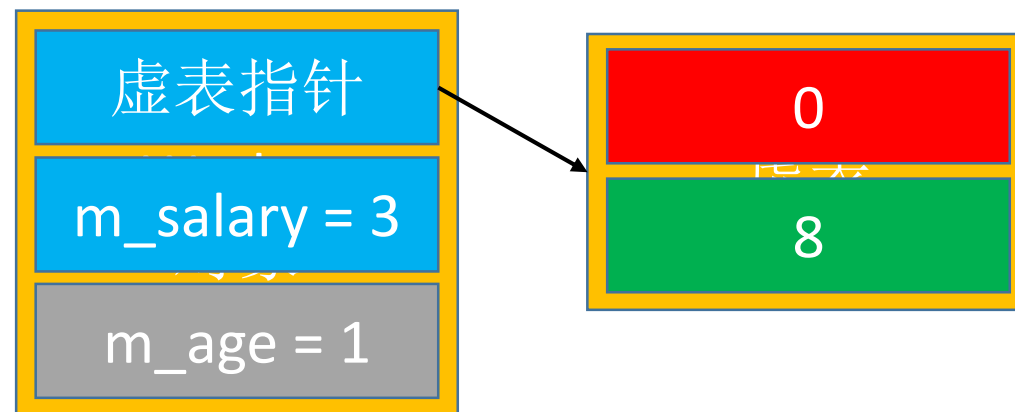
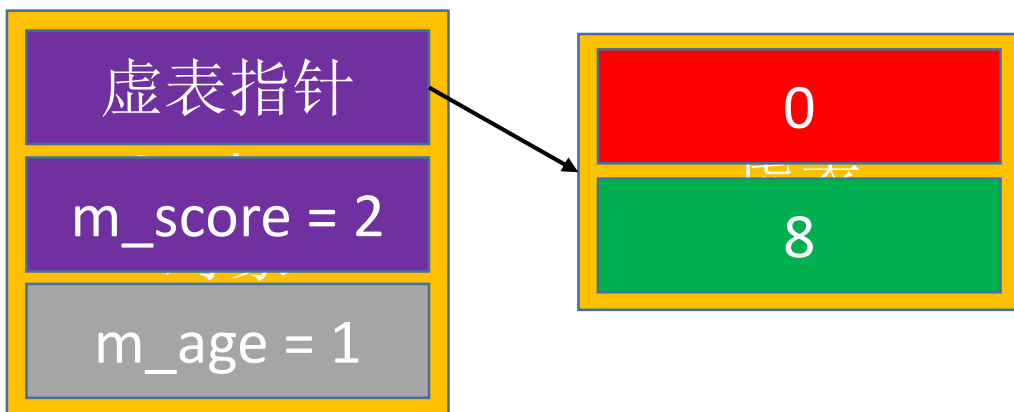
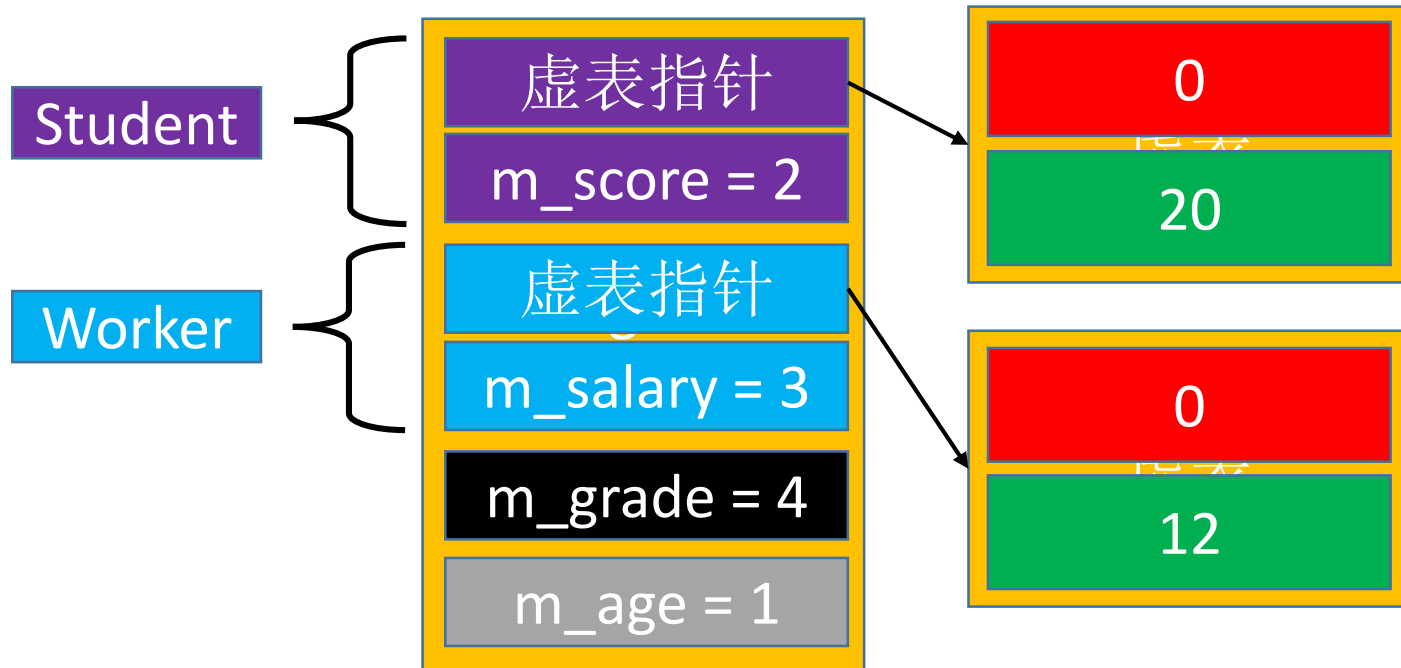
虚基类第一个成员变量与本类起始的偏移量

```
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;
};
```



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`

```
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;
```

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;  
    }  
};
```

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

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) { }  
};
```

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

浅拷贝、深拷贝

- 编译器默认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

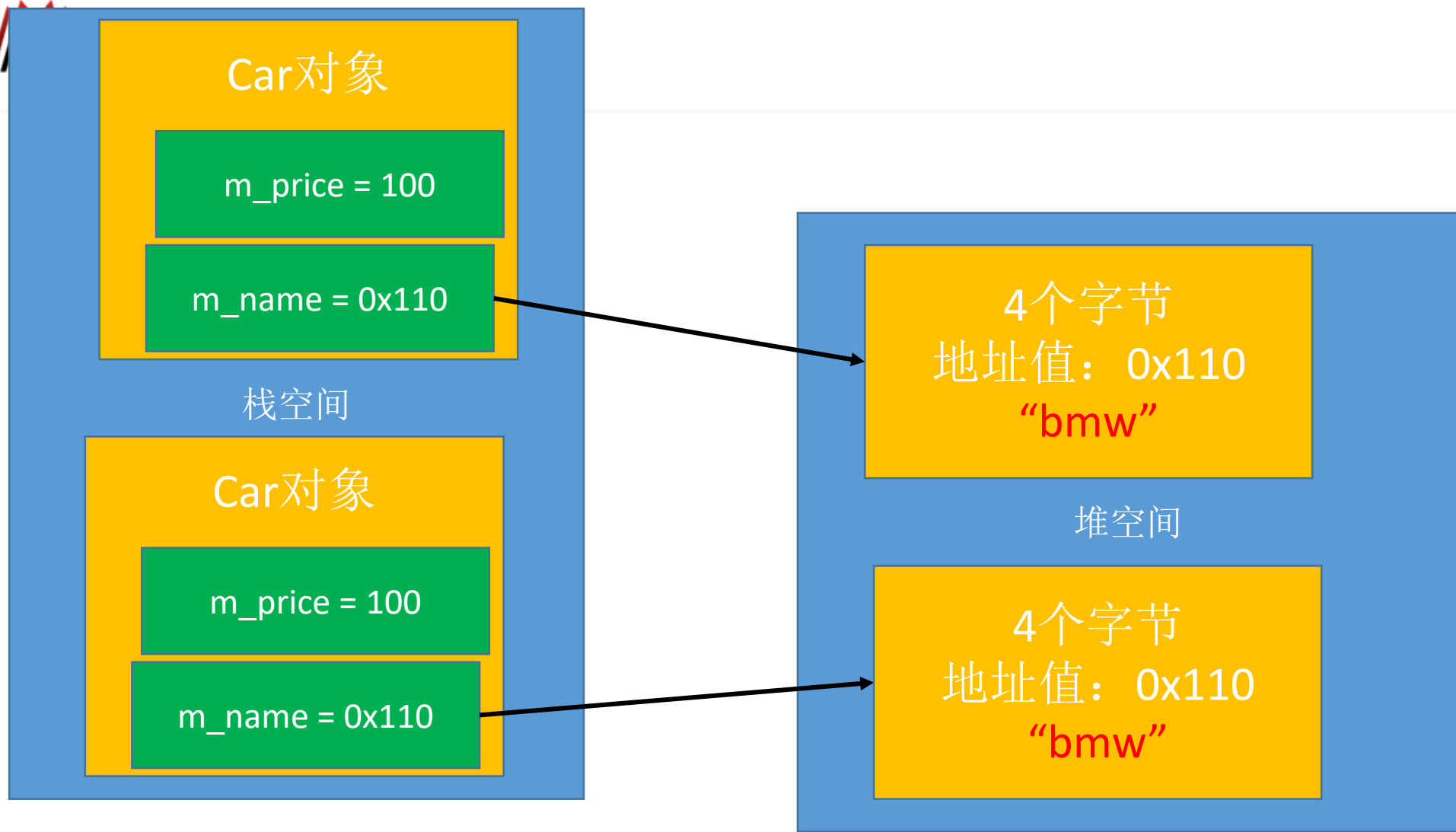
Car对象

m_price = 100

m_name = name数组的地址

堆空间

4个字节
"bmw"



```
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) { }  
};
```

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

- 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;  
    }  
};
```

- 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;  
        }  
    };  
};
```

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() {  
  
}
```

- 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();
}
```