

# 面向对象

@M了个J

<https://weibo.com/exceptions>

<https://github.com/CoderMJLee>



实力IT教育 [www.520it.com](http://www.520it.com)

# 回想一下面向对象的常见知识点

- 类

- 对象

- 成员变量、成员函数

- 封装、继承、多态

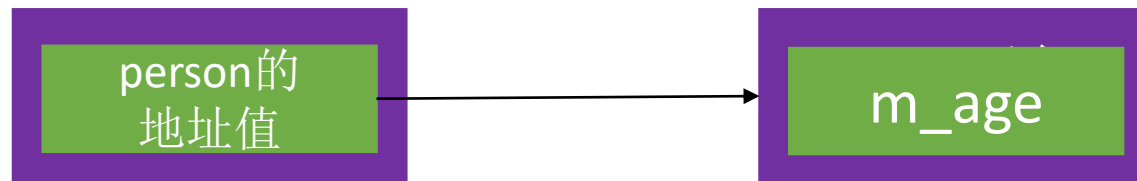
- .....

■ C++中可以使用struct、class来定义一个类

■ struct和class的区别

□ struct的默认成员权限是public

□ class的默认成员权限是private



```

// 类的定义
struct Person {
    // 成员变量
    int m_age;
    // 成员函数
    void run() {
        cout << m_age << " run()" << endl;
    }
};
  
```

```

// 类的定义
class Person {
public:
    // 成员变量
    int m_age;
    // 成员函数
    void run() {
        cout << m_age << " run()" << endl;
    }
};
  
```

```

Person person;
// 访问person对象的成员变量
person.m_age = 20;
// 用person对象调用成员函数
person.run();

// 通过指针间接访问person对象
Person *p = &person;
p->m_age = 30;
p->run();
  
```

■ 上面代码中person对象、p指针的内存都是在函数的栈空间，自动分配和回收的

■ 可以尝试反汇编struct和class，看看是否有其他区别

■ 实际开发中，用class表示类比较多

■ 每个人都可以有自己的编程规范，没有统一的标准，没有标准答案，没有最好的编程规范

■ 变量名规范参考

□ 全局变量：g\_

□ 成员变量：m\_

□ 静态变量：s\_

□ 常量：c\_

□ 使用驼峰标识

# 对象的内存布局

■ 思考：如果类中有多个成员变量，对象的内存又是如何布局的？

```
struct Person {  
    int m_id;  
    int m_age;  
    int m_height;  
  
    void display() {  
        cout << "m_id is " << m_id << endl;  
        cout << "m_age is " << m_age << endl;  
        cout << "m_height is " << m_height << endl;  
    }  
};
```

# 对象的内存布局

```
Person person;
person.m_id = 1;
person.m_age = 2;
person.m_height = 3;
```

		内存地址	内存数据
&person	&person.m_id	0x00E69B60	1
		0x00E69B61	
		0x00E69B62	
		0x00E69B63	
	&person.m_age	0x00E69B64	2
		0x00E69B65	
		0x00E69B66	
		0x00E69B67	
	&person.m_height	0x00E69B68	3
		0x00E69B69	
		0x00E69B6A	
		0x00E69B6B	

■ `this`是指向当前对象的指针

■ 对象在调用成员函数的时候，会自动传入当前对象的内存地址

```
struct Person {  
    int m_id;  
    int m_age;  
    int m_height;  
  
    void display() {  
        cout << "m_id is " << this->m_id << endl;  
        cout << "m_age is " << this->m_age << endl;  
        cout << "m_height is " << this->m_height << endl;  
    }  
};
```

- 可以利用 `this.m_age` 来访问成员变量么?
- 不可以, 因为 `this` 是指针, 必须用 `this->m_age`



# 指针访问对象成员的本质

■ 思考：最后打印出来的每个成员变量值是多少？

```
Person person;  
person.m_id = 10;  
person.m_age = 20;  
person.m_height = 30;  
  
Person *p = (Person *) &person.m_age;  
p->m_id = 40;  
p->m_age = 50;  
  
person.display();
```

■ 答案：10 40 50

■ 思考

□ 如果将 `person.display()` 换成 `p->display()` 呢？

- 成员变量私有化，提供公共的getter和setter给外界去访问成员变量

```
struct Person {  
    private:  
        int m_age;  
    public:  
        void setAge(int age) {  
            this->m_age = age;  
        }  
        int getAge() {  
            return this->m_age;  
        }  
};
```

```
Person person;  
person.setAge(20);  
cout << person.getAge() << endl;
```

# 内存空间的布局

■ 每个应用都有自己独立的内存空间，其内存空间一般都有以下几大区域

□ 代码段（代码区）

✓ 用于存放代码

□ 数据段（全局区）

✓ 用于存放全局变量等

□ 栈空间

✓ 每调用一个函数就会给它分配一段连续的栈空间，等函数调用完毕后会自动回收这段栈空间

✓ 自动分配和回收

□ 堆空间

✓ 需要主动去申请和释放



- 在程序运行过程，为了能够自由控制内存的生命周期、大小，会经常使用堆空间的内存

- 堆空间的申请\释放

- malloc \ free

- new \ delete

- new [] \ delete []

- 注意

- 申请堆空间成功后，会返回那一段内存空间的地址

- 申请和释放必须是1对1的关系，不然可能会存在内存泄露

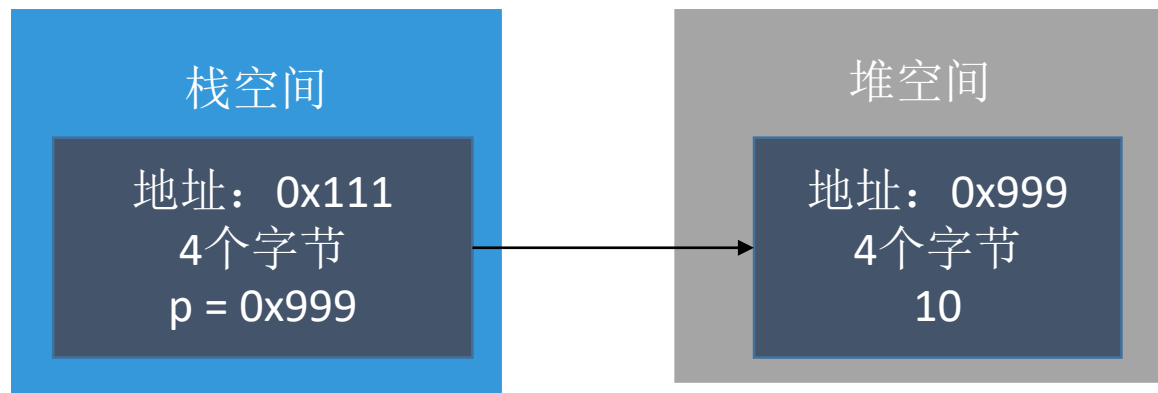
- 现在的很多高级编程语言不需要开发人员去管理内存（比如Java），屏蔽了很多内存细节，利弊同时存在

- 利：提高开发效率，避免内存使用不当或泄露

- 弊：不利于开发人员了解本质，永远停留在API调用和表层语法糖，对性能优化无从下手

## 下图是X86环境 (32bit)

```
int *p = (int *) malloc(4);  
*p = 10;
```



# 堆空间的初始化

```
int *p1 = (int *)malloc(sizeof(int)); // *p1未初始化
int *p2 = (int *)malloc(sizeof(int));
memset(p2, 0, sizeof(int)); // 将*p2的每一个字节都初始化为0
```

```
int *p1 = new int;           // 未被初始化
int *p2 = new int();         // 被初始化为0
int *p3 = new int(5);        // 被初始化为5
int *p4 = new int[3];        // 数组元素未被初始化
int *p5 = new int[3]();      // 3个数组元素都被初始化为0
int *p6 = new int[3]{};      // 3个数组元素都被初始化为0
int *p7 = new int[3]{ 5 };   // 数组首元素被初始化为5, 其他元素被初始化为0
```

■ memset函数是将较大的数据结构（比如对象、数组等）内存清零的比較快的方法

```
Person person;  
person.m_id = 1;  
person.m_age = 20;  
person.m_height = 180;  
memset(&person, 0, sizeof(person));
```

```
Person persons[] = { { 1, 20, 180 }, { 2, 25, 165, }, { 3, 27, 170 } };  
memset(persons, 0, sizeof(persons));
```

# 对象的内存

- 对象的内存可以存在于3种地方
  - 全局区（数据段）：全局变量
  - 栈空间：函数里面的局部变量
  - 堆空间：动态申请内存（malloc、new等）

```
// 全局区
Person g_person;

int main() {
    // 栈空间
    Person person;

    // 堆空间
    Person *p = new Person;
    return 0;
}
```



# 构造函数 (Constructor)

- 构造函数（也叫构造器），在对象创建的时候自动调用，一般用于完成对象的初始化工作
- 特点
  - 函数名与类同名，无返回值（`void`都不能写），可以有参数，可以重载，可以有多个构造函数
  - 一旦自定义了构造函数，必须用其中一个自定义的构造函数来初始化对象
- 注意
  - 通过`malloc`分配的对象不会调用构造函数
- 一个广为流传的、很多教程\书籍都推崇的错误结论：
  - ~~默认情况下，编译器会为每一个类生成空的无参的构造函数~~
  - 正确理解：在某些特定的情况下，编译器才会为类生成空的无参的构造函数
  - ✓ （哪些特定的情况？以后再提）

# 构造函数的调用

```
struct Person {  
    int m_age;  
    Person() {  
        cout << "Person()" << endl;  
    }  
    Person(int age) {  
        cout << "Person(int age)" << endl;  
    }  
};
```

```
// 全局区  
Person g_p1;           // 调用Person()  
Person g_p2();          // 这是一个函数声明, 函数名叫g_p2, 返回值类型是Person, 无参  
Person g_p3(20);        // 调用Person(int)  
  
int main() {  
    // 栈空间  
    Person p1;           // 调用Person()  
    Person p2();          // 这是一个函数声明, 函数名叫p2, 返回值类型是Person, 无参  
    Person p3(20);        // 调用Person(int)  
  
    // 堆空间  
    Person *p4 = new Person;           // 调用Person()  
    Person *p5 = new Person();          // 调用Person()  
    Person *p6 = new Person(20);        // 调用Person(int)  
    return 0;  
}
```

# 默认情况下，成员变量的初始化

```
struct Person {  
    int m_age;  
};  
  
// 全局区 (成员变量初始化为0)  
Person g_p1;  
  
int main() {  
    // 栈空间 (成员变量不会被初始化)  
    Person p1;  
  
    // 堆空间  
    Person *p2 = new Person;           // 成员变量不会被初始化  
    Person *p3 = new Person();         // 成员变量初始化为0  
    Person *p4 = new Person[3];        // 成员变量不会被初始化  
    Person *p5 = new Person[3]();      // 3个Person对象的成员变量都初始化为0  
    Person *p6 = new Person[3]{};      // 3个Person对象的成员变量都初始化为0  
  
    return 0;  
}
```

■ 如果自定义了构造函数，除了全局区，其他内存空间的成员变量默认都不会被初始化，需要开发人员手动初始化

# 成员变量的初始化

## ■ 对象初始化

```
Person() {  
    memset(this, 0, sizeof(Person));  
}
```

# 析构函数 (Destructor)

- 析构函数（也叫析构器），在对象销毁的时候自动调用，一般用于完成对象的清理工作
- 特点
  - 函数名以~开头，与类同名，无返回值（`void`都不能写），无参，不可以重载，有且只有一个析构函数
- 注意
  - 通过`malloc`分配的对象`free`的时候不会调用析构函数
- 构造函数、析构函数要声明为`public`，才能被外界正常使用

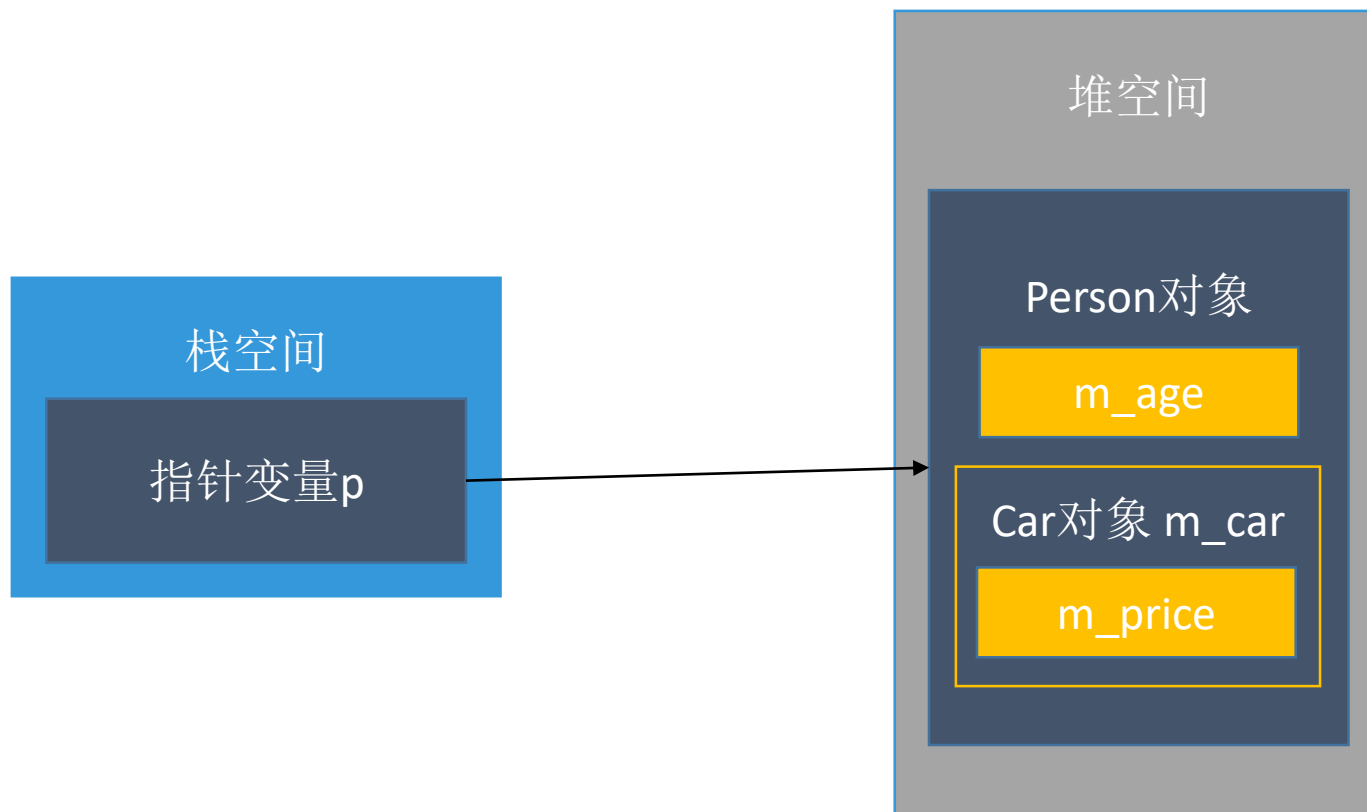
```
struct Car {  
    int m_price;  
};  
  
struct Person {  
    int m_age;  
    Car m_car;  
};  
  
Person p;
```



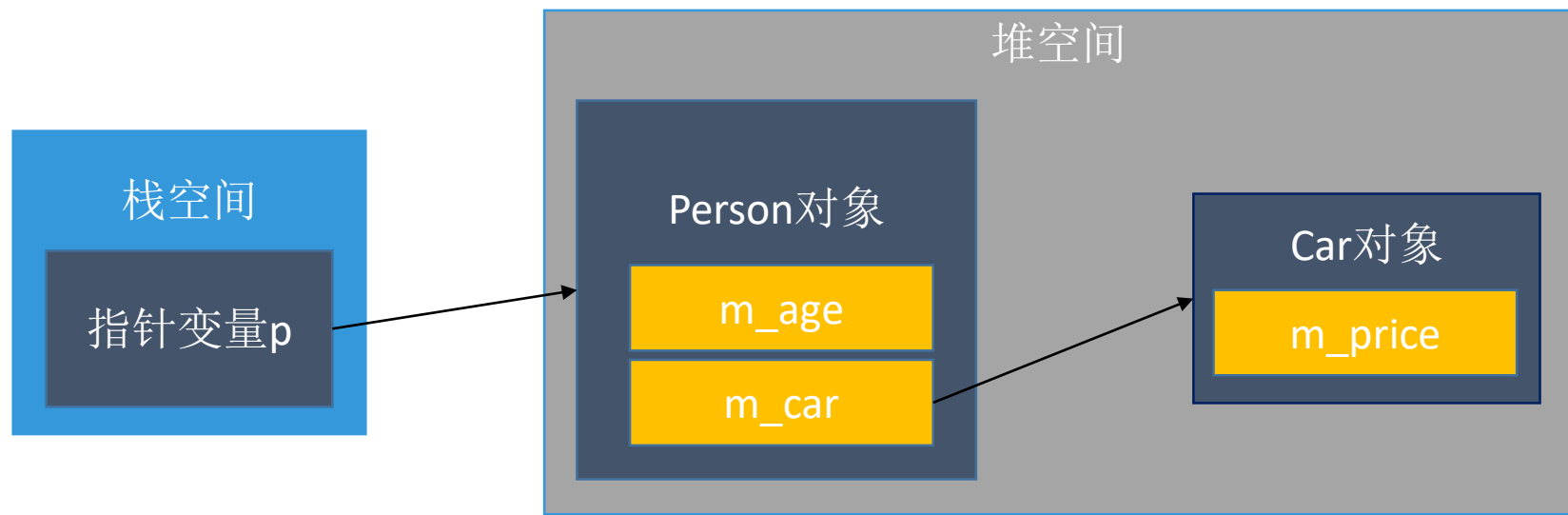
```
struct Car {  
    int m_price;  
};
```

```
struct Person {  
    int m_age;  
    Car m_car;  
};
```

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



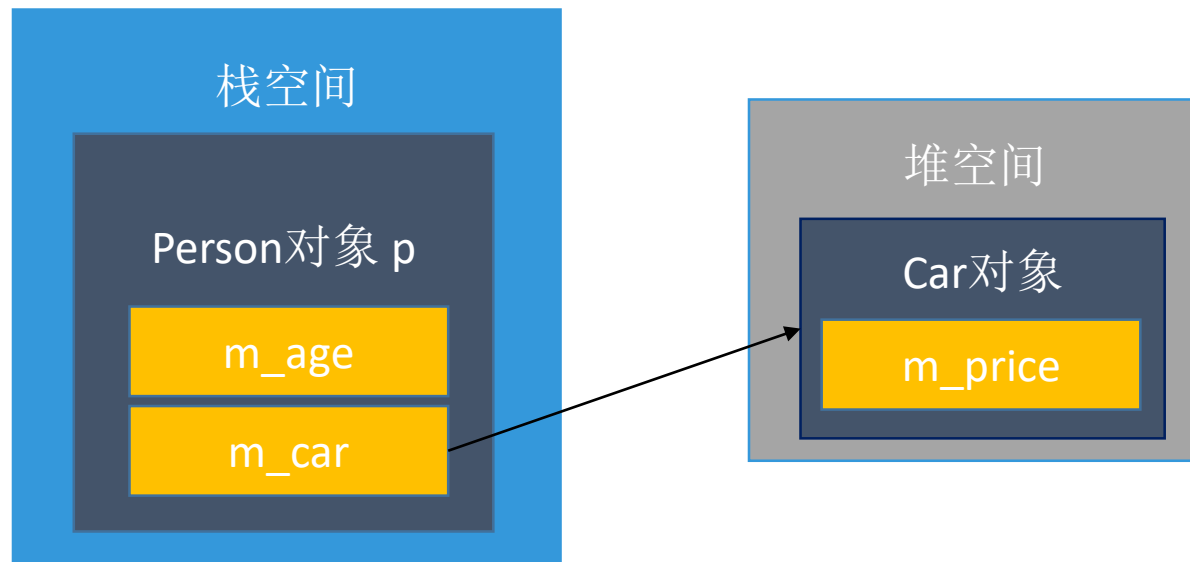
```
struct Car {  
    int m_price;  
};  
  
struct Person {  
    int m_age;  
    Car *m_car;  
    Person() {  
        m_car = new Car();  
    }  
    ~Person() {  
        delete m_car;  
    }  
};  
  
Person *p = new Person();
```



- 对象内部申请的堆空间，由对象内部回收



```
struct Car {  
    int m_price;  
};  
  
struct Person {  
    int m_age;  
    Car *m_car;  
    Person() {  
        m_car = new Car();  
    }  
    ~Person() {  
        delete m_car;  
    }  
};  
  
Person p;
```



# 声明和实现分离

Person.h

测试

```
1  #pragma once
2  class Person {
3  private:
4      int m_age;
5  public:
6      void setAge(int age);
7      int getAge();
8      Person();
9      ~Person();
10 };
```

Person.cpp

测试

(全

```
1  #include "Person.h"
2
3  Person::Person() {
4  }
5
6  void Person::setAge(int age) {
7      this->m_age = age;
8  }
9
10 int Person::getAge() {
11     return this->m_age;
12 }
13
14 Person::~~Person() {
15 }
```

## ■ 命名空间可以用来避免命名冲突

```
namespace MJ {  
    int g_age;  
  
    class Person {  
    };  
  
    void test() {  
    }  
}  
  
int main() {  
    MJ::g_age = 20;  
    MJ::Person *p = new MJ::Person();  
    MJ::test();  
    return 0;  
}
```

```
using namespace MJ;
```

```
g_age = 20;  
Person *p = new Person();  
test();
```

```
using MJ::g_age;  
g_age = 20;
```

## ■ 命名空间不影响内存布局

```
namespace MJ {  
    int g_age;  
}
```

```
namespace FX {  
    int g_age;  
}
```

```
using namespace MJ;  
using namespace FX;
```

```
// 这句代码能编译通过么?  
g_age = 20;
```

# 命名空间的嵌套

- 有个默认的全局命名空间，我们创建的命名空间默认都嵌套在它里面

```
namespace MJ {  
    namespace SS {  
        int g_age;  
    }  
}  
  
int main() {  
    MJ::SS::g_age = 10;  
  
    using namespace MJ::SS;  
    g_age = 20;  
  
    using MJ::SS::g_age;  
    g_age = 30;  
  
    return 0;  
}
```

```
int g_no;  
  
namespace MJ {  
    namespace SS {  
        int g_age;  
    }  
}  
  
int main() {  
    ::g_no = 20;  
  
    ::MJ::SS::g_age = 30;  
  
    return 0;  
}
```

# 命名空间的合并

- 以下2种写法是等价的

```
namespace MJ {  
    int g_age;  
}
```

```
namespace MJ {  
    int g_no;  
}
```

```
namespace MJ {  
    int g_age;  
    int g_no;  
}
```

# 命名空间的合并

```
Person.h  测试
1  #pragma once
2
3  namespace MJ {
4      class Person {
5      public:
6          Person();
7          ~Person();
8      };
9  }
```

```
Person.cpp  测试
1  #include "Person.h"
2
3  namespace MJ {
4      Person::Person() {
5      }
6
7      Person::~~Person() {
8      }
9  }
```

# 其他编程语言的命名空间

- Java
  - Package
- Objective-C
  - 类前缀



- 继承，可以让子类拥有父类的所有成员（变量\函数）

```
struct Person {  
    int m_age;  
    void run() {  
        cout << "Person::run()" << endl;  
    }  
};  
  
struct Student : Person {  
    int m_no;  
    void study() {  
        cout << "Student::study()" << endl;  
    }  
};
```

```
Student student;  
student.m_age = 20;  
student.m_no = 1;  
student.run();  
student.study();
```

- 关系描述

- Student是子类（subclass，派生类）
- Person是父类（superclass，超类）

- C++中没有像Java、Objective-C的基类

- Java: java.lang.Object

- Objective-C: NSObject

# 对象的内存布局

```
struct Person {  
    int m_age;  
};  
  
struct Student : Person {  
    int m_no;  
};  
  
struct GoodStudent : Student {  
    int m_money;  
};
```

```
GoodStudent gs;  
gs.m_age = 20;  
gs.m_no = 1;  
gs.m_money = 666;
```

		内存地址	内存数据
&gs	&gs.m_age	B8BCEFF708	20
		B8BCEFF709	
		B8BCEFF70A	
		B8BCEFF70B	
	&gs.m_no	B8BCEFF70C	1
		B8BCEFF70D	
		B8BCEFF70E	
		B8BCEFF70F	
	&gs.m_money	B8BCEFF710	666
		B8BCEFF711	
		B8BCEFF712	
		B8BCEFF713	

■ 父类的成员变量在前，子类的成员变量在后

# 成员访问权限

- 成员访问权限、继承方式有3种

- `public`: 公共的, 任何地方都可以访问 (struct默认)

- `protected`: 子类内部、当前类内部可以访问

- `private`: 私有的, 只有当前类内部可以访问 (class默认)

- 子类内部访问父类成员的权限, 是以下2项中权限最小的那个

- 成员本身的访问权限

- 上一级父类的继承方式

- 开发中用的最多的继承方式是`public`, 这样能保留父类原来的成员访问权限

- 访问权限不影响对象的内存布局

# 初始化列表

- 特点
  - 一种便捷的初始化成员变量的方式
  - 只能用在构造函数中
  - 初始化顺序只跟成员变量的声明顺序有关

- 图片中的2种写法是等价的

```
struct Person {  
    int m_age;  
    int m_height;  
    Person(int age, int height) {  
        this->m_age = age;  
        this->m_height = height;  
    }  
};
```

```
struct Person {  
    int m_age;  
    int m_height;  
    Person(int age, int height) :m_age(age), m_height(height) {  
    }  
};
```

■ m\_age、m\_height的值是多少

```
int myAge() { return 10; }
int myHeight() { return 170; }

struct Person {
    int m_age;
    int m_height;
    Person(int age, int height) :m_height(myHeight()), m_age(myAge()) {
        // ...
    }
};

Person p(20, 180);
```

■ 10、170

■ m\_age、m\_height的值是多少

```
struct Person {  
    int m_age;  
    int m_height;  
    Person(int age, int height) :m_height(height), m_age(m_height) {  
        }  
};  
  
Person p(20, 180);
```

■ 未知、180

# 初始化列表与默认参数配合使用

```
struct Person {  
    int m_age;  
    int m_height;  
    Person(int age = 0, int height = 0) :m_age(age), m_height(height) { }  
};  
  
int main() {  
    Person person1;  
    Person person2(18);  
    Person person3(20, 180);  
  
    getchar();  
    return 0;  
}
```

- 如果函数声明和实现是分离的
- 初始化列表只能写在函数的实现中
- 默认参数只能写在函数的声明中

# 构造函数的互相调用

```
struct Person {  
    int m_age;  
    int m_height;  
    Person() :Person(0, 0) { }  
    Person(int age, int height) :m_age(age), m_height(height) { }  
};
```

■ 注意：下面的写法是错误的，初始化的是一个临时对象

```
struct Person {  
    int m_age;  
    int m_height;  
    Person() {  
        Person(0, 0);  
    }  
    Person(int age, int height) :m_age(age), m_height(height) { }  
};
```



# 父类的构造函数

- 子类的构造函数默认会调用父类的无参构造函数
- 如果子类的构造函数显式地调用了父类的有参构造函数，就不会再去默认调用父类的无参构造函数
- 如果父类缺少无参构造函数，子类的构造函数必须显式调用父类的有参构造函数

# 继承体系下的构造函数示例

```
struct Person {  
    int m_age;  
    Person() :Person(0) {  
    }  
    Person(int age) :m_age(age) {  
    }  
};  
  
struct Student : Person {  
    int m_no;  
    Student() :Student(0, 0) {  
    }  
    Student(int age, int no) :Person(age), m_no(no) {  
    }  
};
```

# 构造、析构顺序

```
struct Person {
    Person() {
        cout << "Person::Person()" << endl;
    }
    ~Person() {
        cout << "Person::~~Person()" << endl;
    }
};

struct Student : Person {
    Student() {
        // call Person::Person()
        cout << "Student::Student()" << endl;
    }
    ~Student() {
        cout << "Student::~~Student()" << endl;
        // call Person::~~Person()
    }
};
```

```
int main() {
    {
        Student student;
    }

    getchar();
    return 0;
}
```

```
Person()
Student()
~Student()
~Person()
```

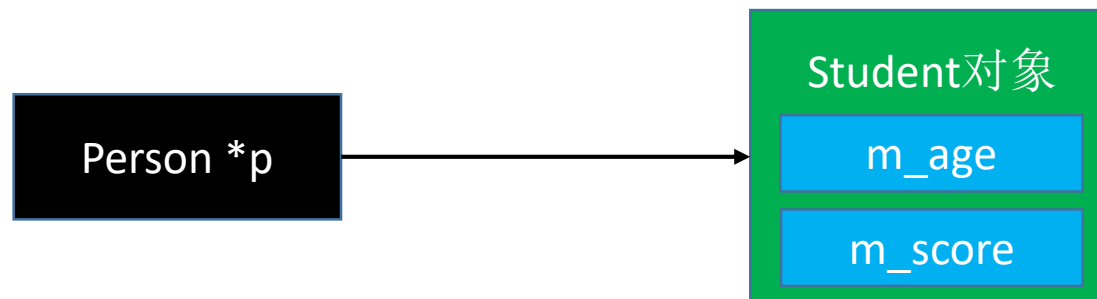
# 父类指针、子类指针

■ 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）

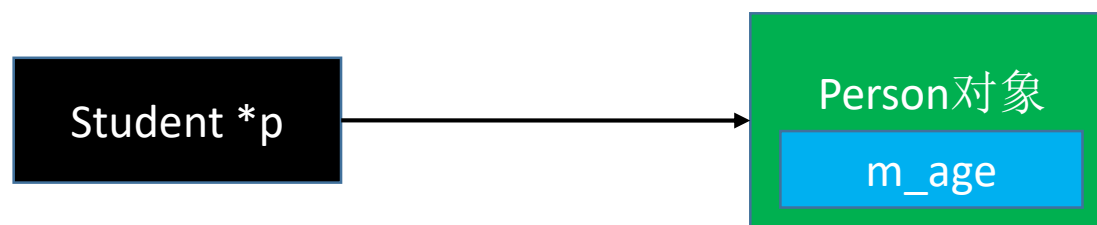
■ 子类指针指向父类对象是不安全的

```
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;
```



- 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

- 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



```
// 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		内存地址	内存数据
cat	0x00E69B60	0x00B89B64	虚表	0x00B89B64	0x00DC14F1
	0x00E69B61			0x00B89B65	
	0x00E69B62			0x00B89B66	
	0x00E69B63			0x00B89B67	
&m_age	0x00E69B64	20		0x00B89B68	0x00DC14CE
	0x00E69B65			0x00B89B69	
	0x00E69B66			0x00B89B6A	
	0x00E69B67			0x00B89B6B	
&m_life	0x00E69B68	0			
	0x00E69B69			Animal::speak的调用地址: 0x00DC14F1	
	0x00E69B6A				
	0x00E69B6B			Cat::run的调用地址: 0x00DC14CE	

# VS的内存窗口

调试(D) 团队(M) 工具(T) 测试(S) 分析(N) 窗口(W) 帮助(H)

**窗口(W)**

- 图形(C)
- 继续(C) F5
- 全部中断(K) Ctrl+Alt+Break
- 停止调试(E) Shift+F5
- 全部拆离(L)
- 全部终止(M)
- 重新启动(R) Ctrl+Shift+F5
- 应用代码更改(A) Alt+F10
- 性能探查器(F)... Alt+F2
- 附加到进程(P)... Ctrl+Alt+P
- 其他调试目标(H)
- 探查器(F)
  - 逐语句(S) F11
  - 逐过程(O) F10
  - 跳出(T) Shift+F11
- 快速监视(Q)... Shift+F9
- 切换断点(G) F9
- 新建断点(B)
- 删除所有断点(D) Ctrl+Shift+F9
- 禁用所有断点(N)

**断点(B)** Ctrl+Alt+B

**异常设置(X)** Ctrl+Alt+E

**输出(O)**

**显示诊断工具(T)** Ctrl+Alt+F2

**GPU 线程(U)**

**任务(S)** Ctrl+Shift+D, K

**并行堆栈(K)** Ctrl+Shift+D, S

**并行监视(R)**

**监视(W)**

**自动窗口(A)** Ctrl+Alt+V, A

**局部变量(L)** Ctrl+Alt+V, L

**即时(I)** Ctrl+Alt+I

**调用堆栈(C)** Ctrl+Alt+C

**线程(H)** Ctrl+Alt+H

**模块(O)** Ctrl+Alt+U

**进程(P)** Ctrl+Alt+Z

**内存(M)**

- 反汇编(D) Ctrl+Alt+D
- 寄存器(G) Ctrl+Alt+G

**内存 1(1)** Ctrl+Alt+M, 1

**内存 2(2)** Ctrl+Alt+M, 2

**内存 3(3)** Ctrl+Alt+M, 3

**内存 4(4)** Ctrl+Alt+M, 4

# 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;  
    }  
};
```

■ 如果存在父类指针指向子类对象的情况，应该将析构造函数声明为虚函数（虚析构造函数）

□ delete 父类指针时，才会调用子类的析构造函数，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：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;  
};
```

■ 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;

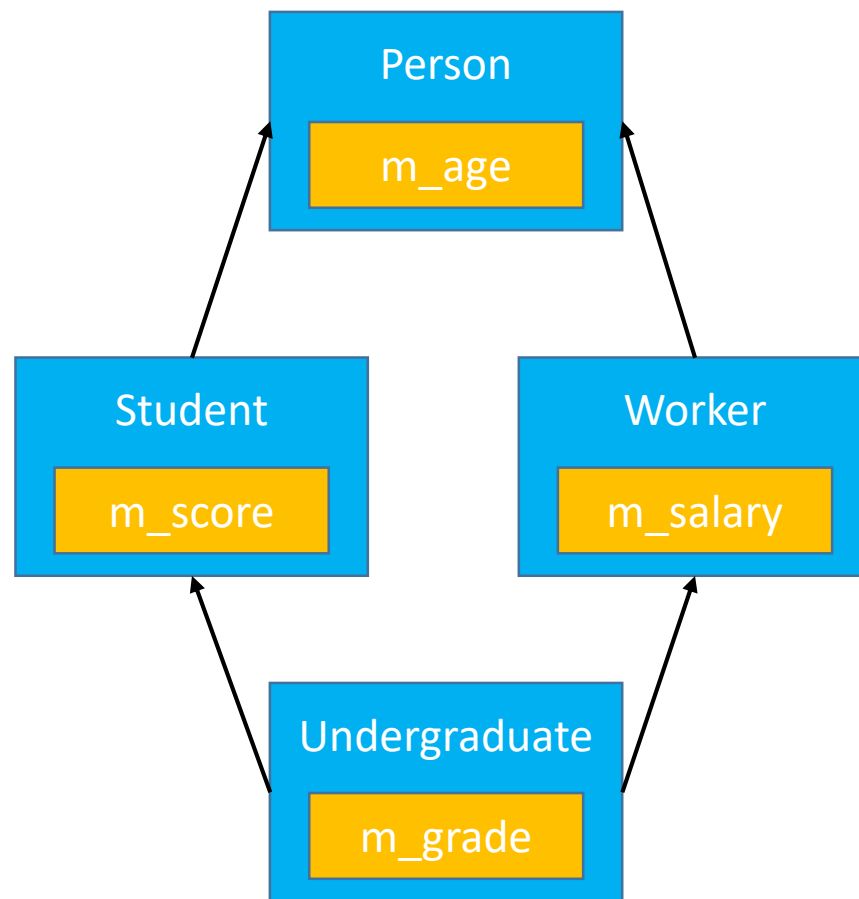
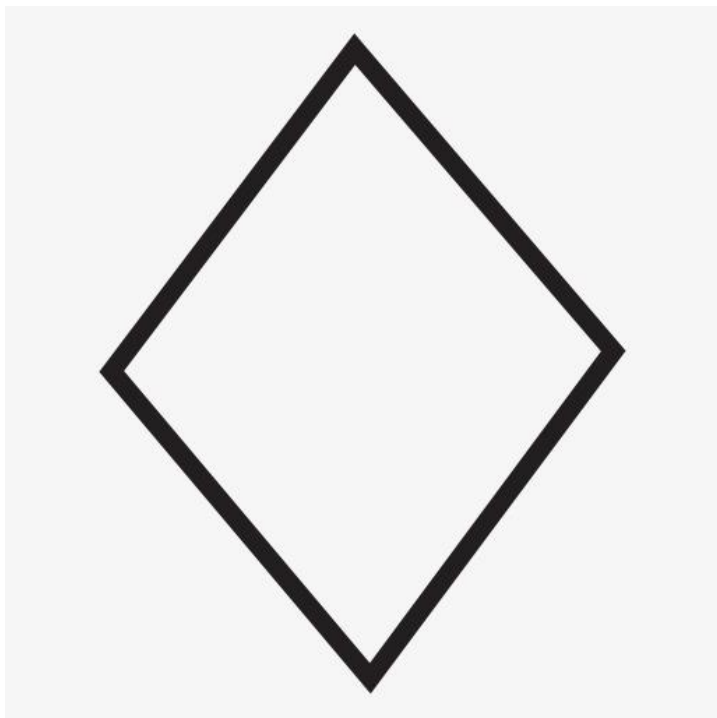
cout << ug.Undergraduate::m_age << endl; // 10
cout << ug.Student::m_age << endl; // 20
cout << ug.Worker::m_age << endl; // 30
```

	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

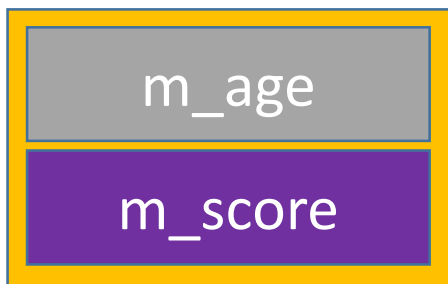
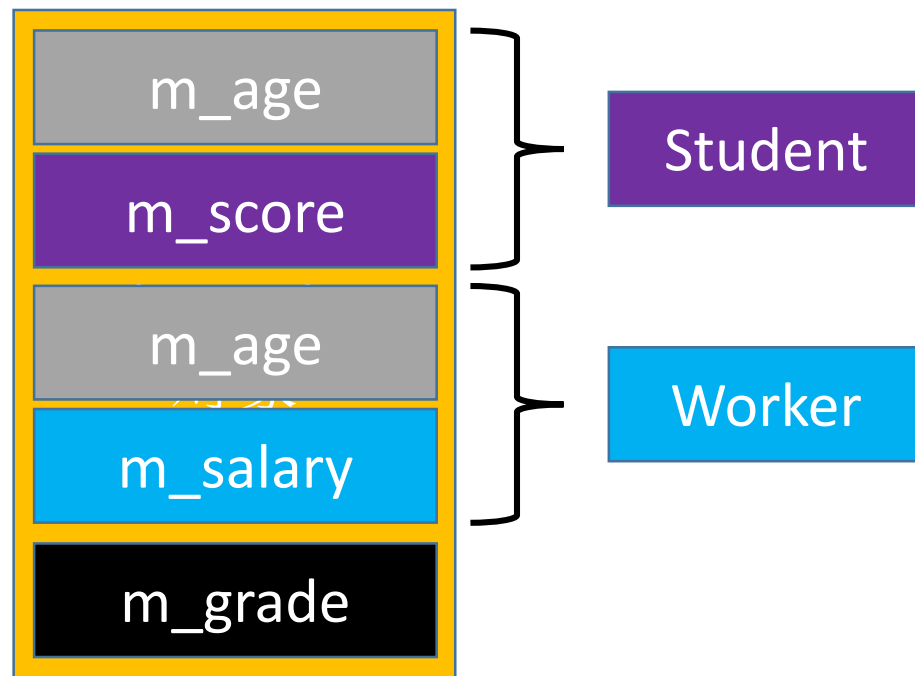
## ■ 菱形继承带来的问题

- 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



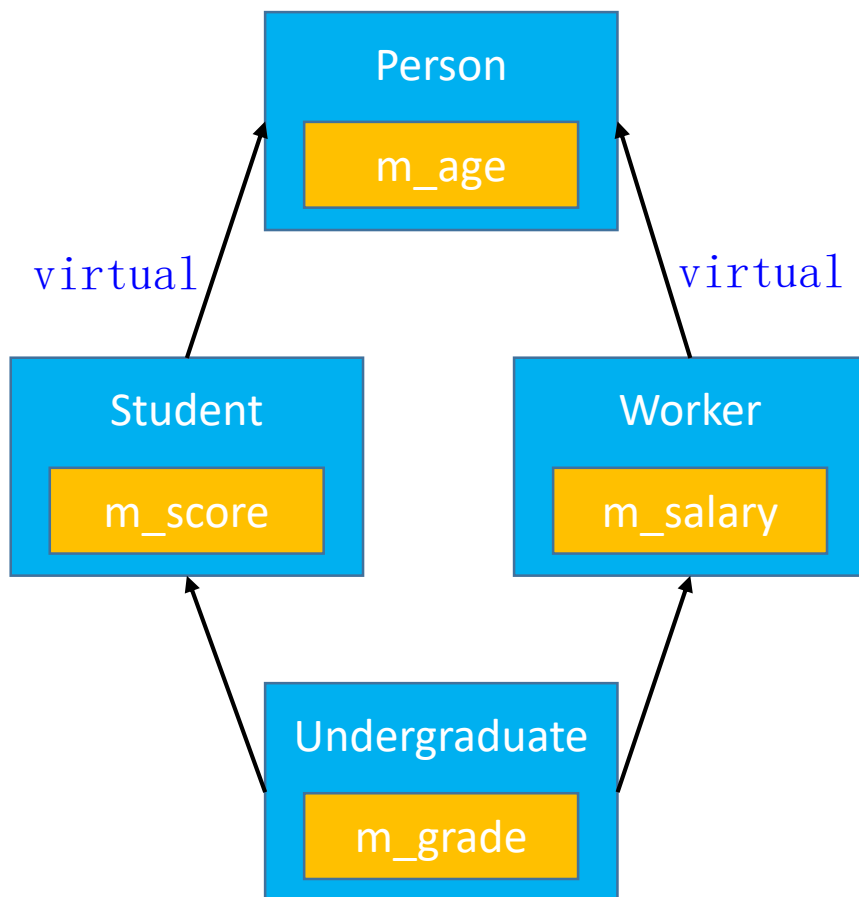
# 菱形继承

```
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）

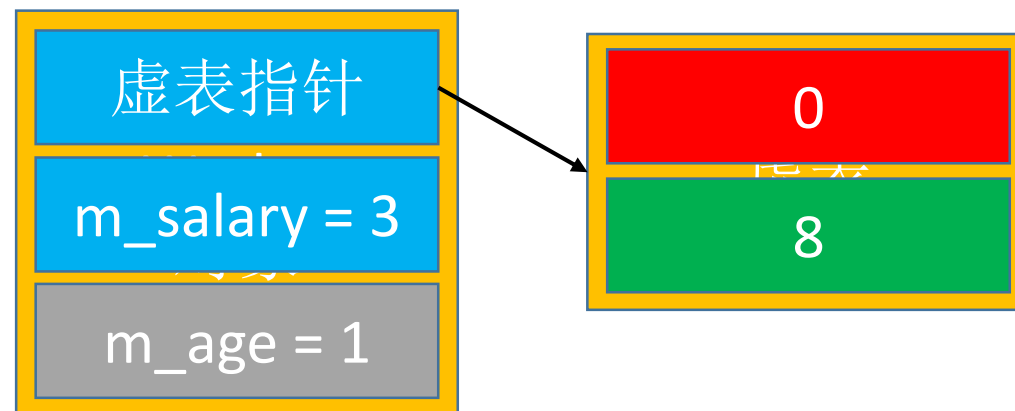
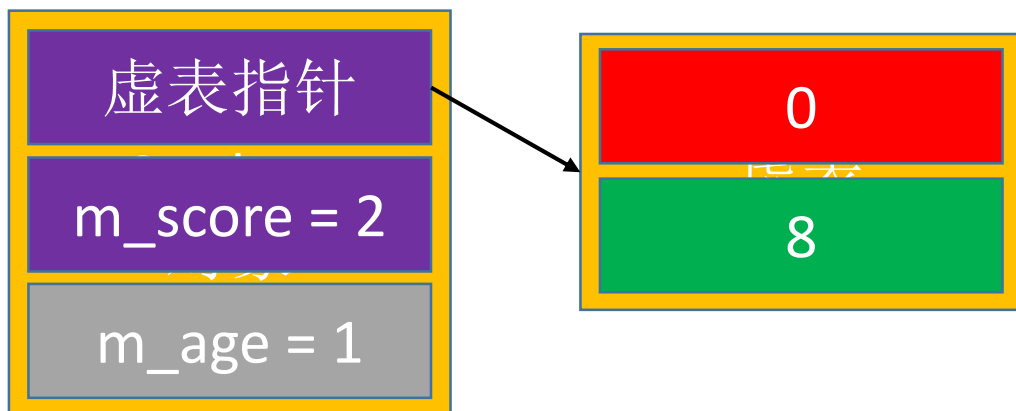
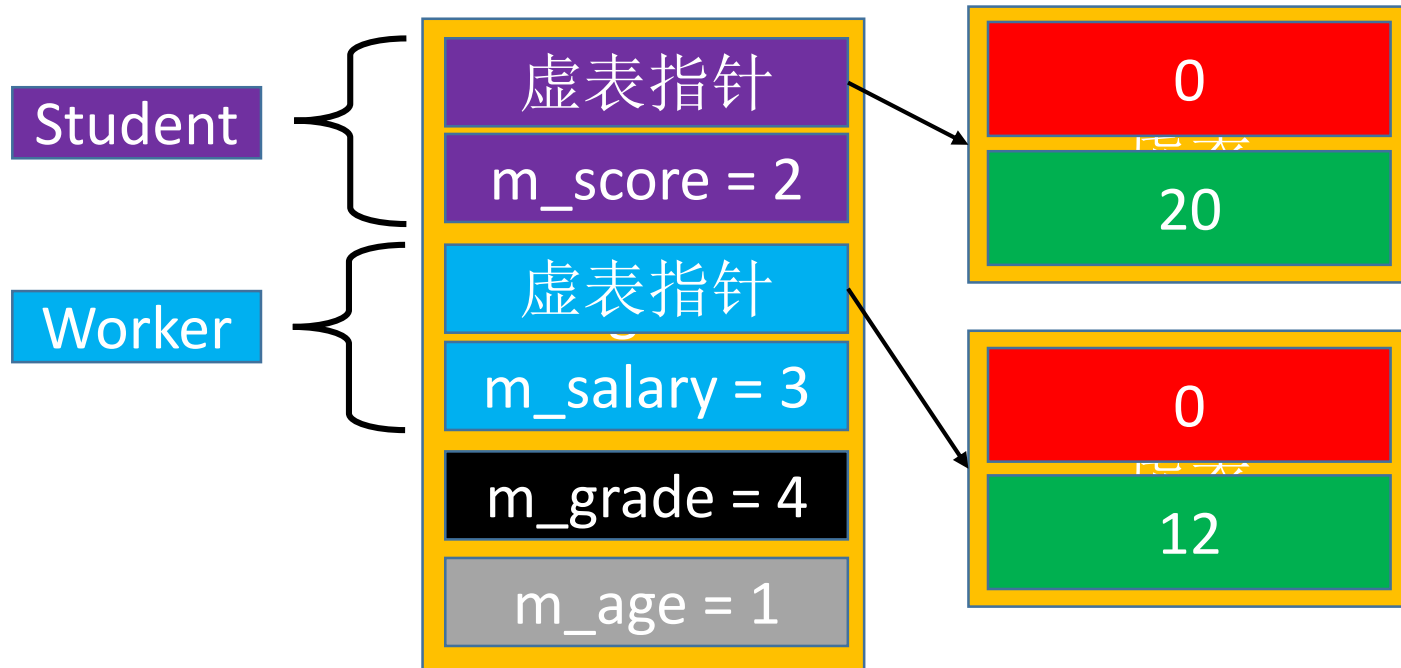
虚基类第一个成员变量与本类起始的偏移量

```
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：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

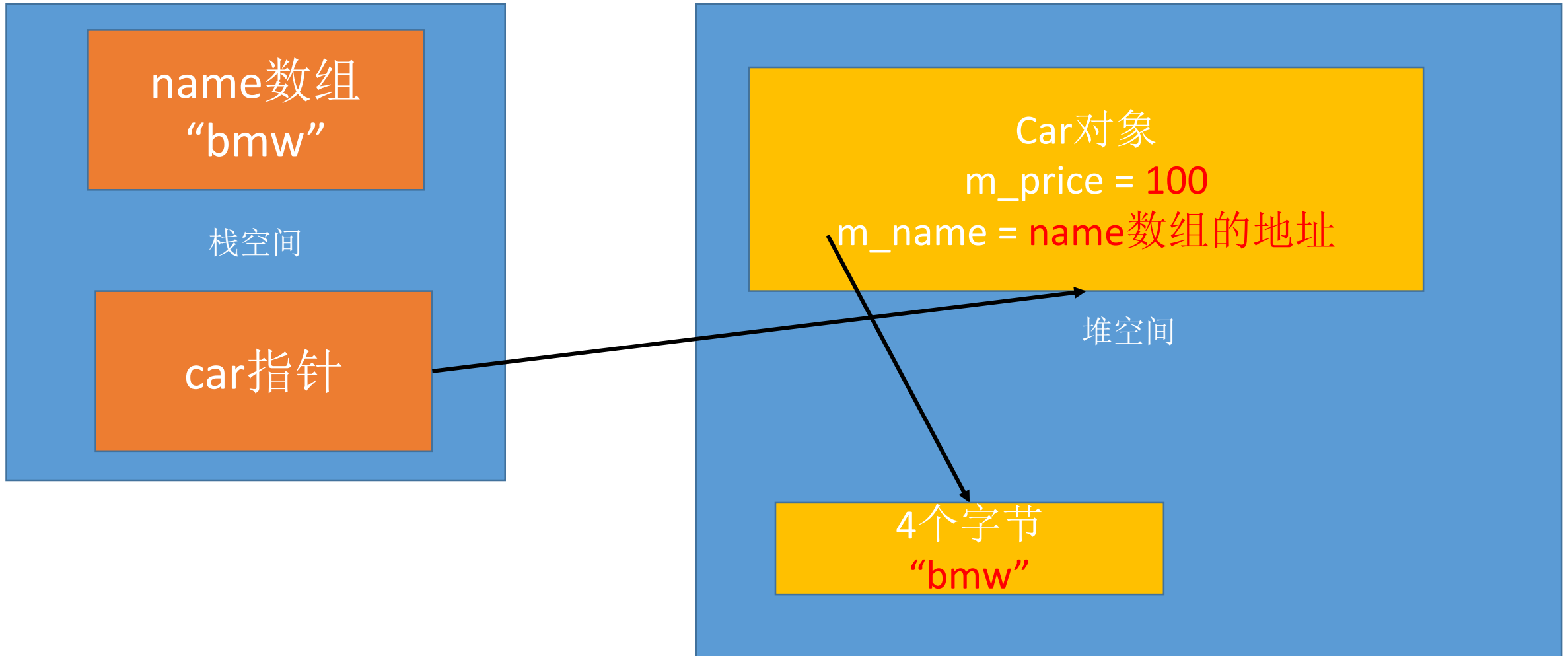
```
Car car(100, "BMW 730Li");  
Car car2 = car;  
Car car3(car2);  
Car car4;  
car4 = car3;
```

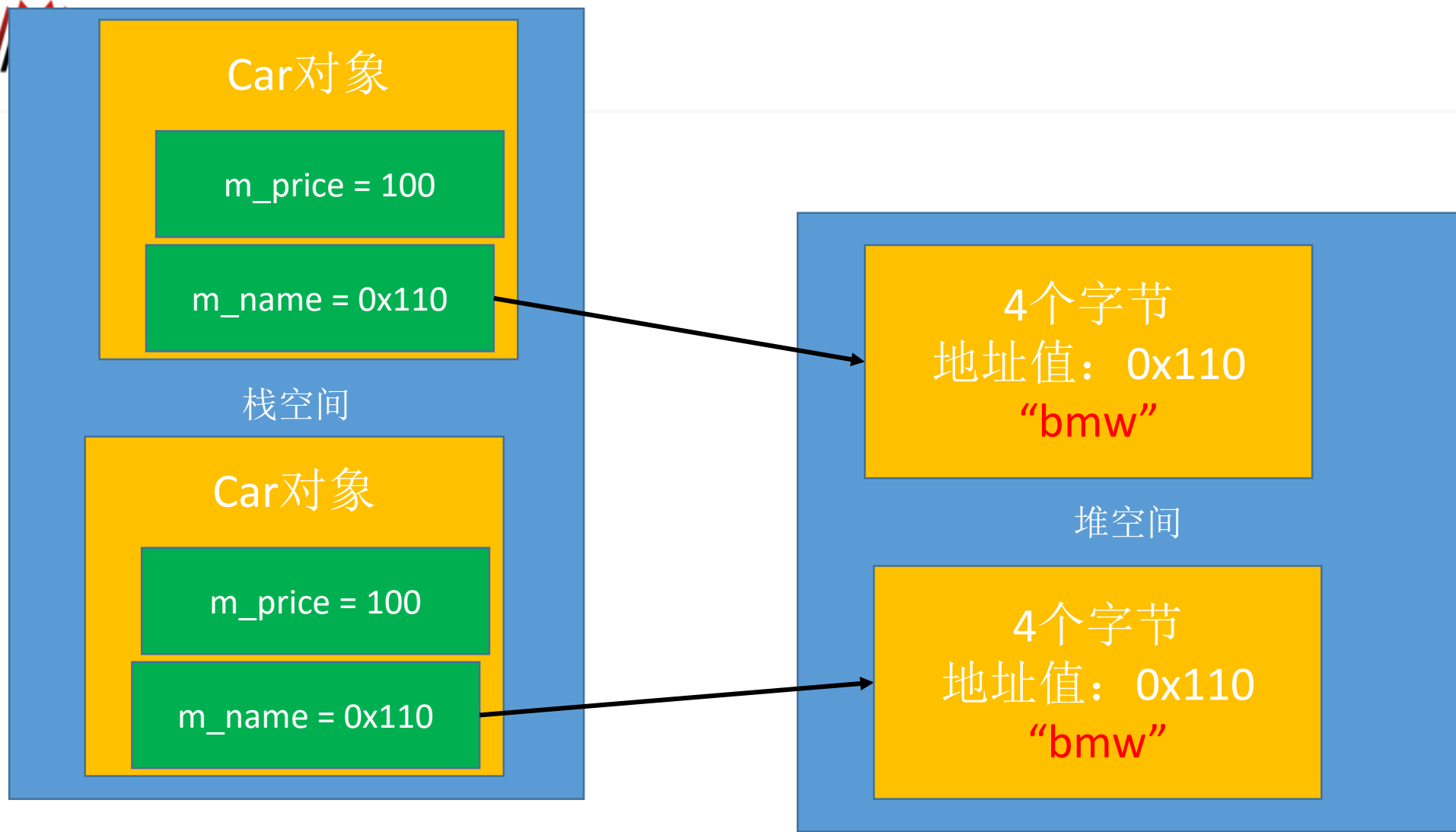
■ car4 = car3是一个赋值操作（默认是浅复制），并不会调用拷贝构造函数



# 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





```
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();
}
```