



中南大學
CENTRAL SOUTH UNIVERSITY



面向对象程序设计 Object Oriented Programing

第三章 类和对象


张宝一

地理信息系

zhangbaoyi@csu.edu.cn



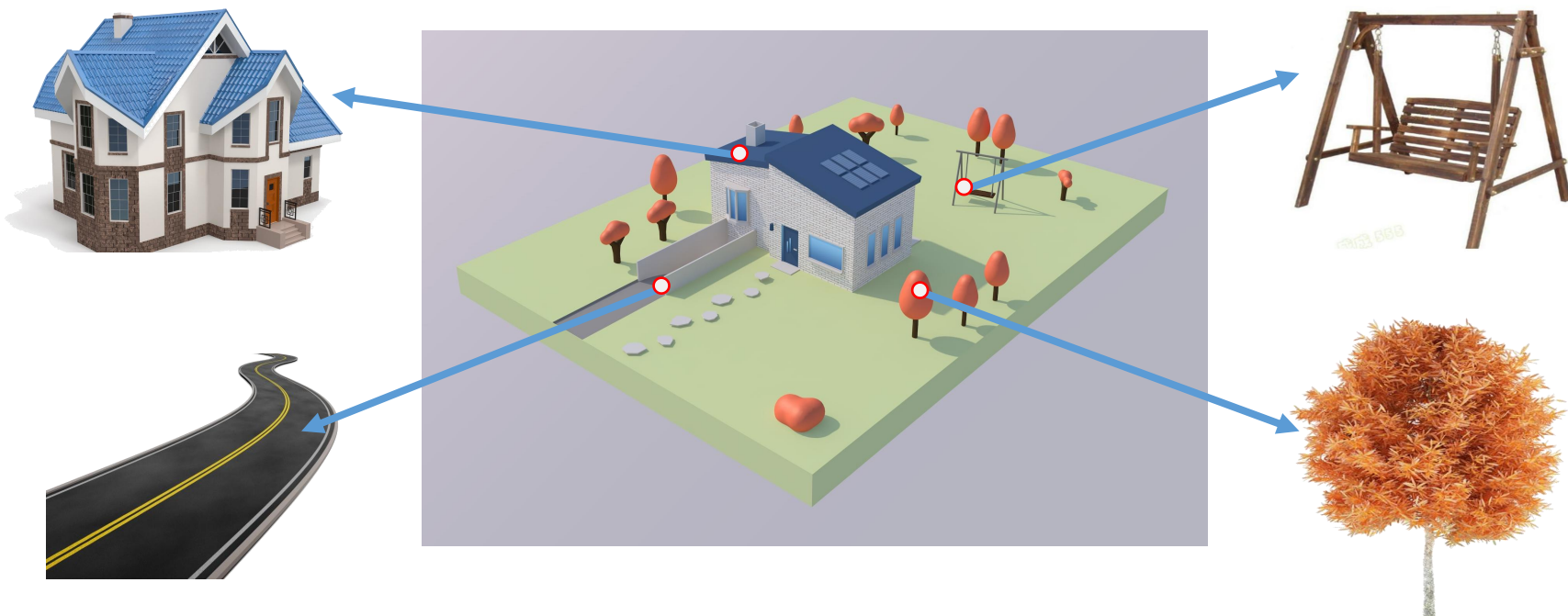
第三章：类和对象

- 类与对象定义 
- 构造函数与析构函数
- this指针
- 类的静态成员
- 类的友元
- 本章小结

3.1 类和对象

面向对象编程：

- 并不是一个技术或编程语言，而是一种编程指导思想。
- 把现实世界的具体事务全部看成一个一个的对象来实际问题。



3.1 类和对象

类与对象：

- 对象：是真实存在的具体实例，也可以是无形的事物（如：复数、计划等）
- 类：是对象共同特征的描述和抽象，如概念图。

对象
(具体实例)



✓ 实物/产品

类
(共性抽象)

✓ 设计/概念图

3.1 类和对象

类与对象：

- 对象：是真实存在的具体实例，也可以是无形的事物（如：复数、计划等）
- 类：是对象共同特征的描述和抽象，如概念图。

类

```
class car {  
    // 属性  
    // 行为  
}
```



海洋豹款

全新纯电轿跑

建议零售价：**21.28-28.98**万元起

海豹上市权益

拥“豹”钜惠礼

- 下订即可尊享88元抵扣8888元购车款

拥“豹”焕新礼

- 2年0息
- 6000元置换补贴

拥“豹”充电礼

- 首任非营运车主2年内享赠送充电桩及免费安装

拥“豹”无忧礼

- 首任非营运车主享6年或6次免费保养
- 首任非营运车主享3年3次（每年1次）免费取车或送车服务
- 非营运车主三电系统终身保修（首任车主）
- 整车包修期6年或15万公里
- 非营运车主享1年或3万公里内无责道路救援

拥“豹”智联礼

- 全系2年免费车机流量（5GB/月）
- 全系2年免费云服务
- 全系2年免费E-Call紧急救援、I-Call智慧客服
- 全系免费智能语音交互系统

拥“豹”畅享礼

- 全系赠送手动遮阳帘（精品）
- 650km四驱性能版及以上配置车型赠送iTAC智能扭矩控制系统

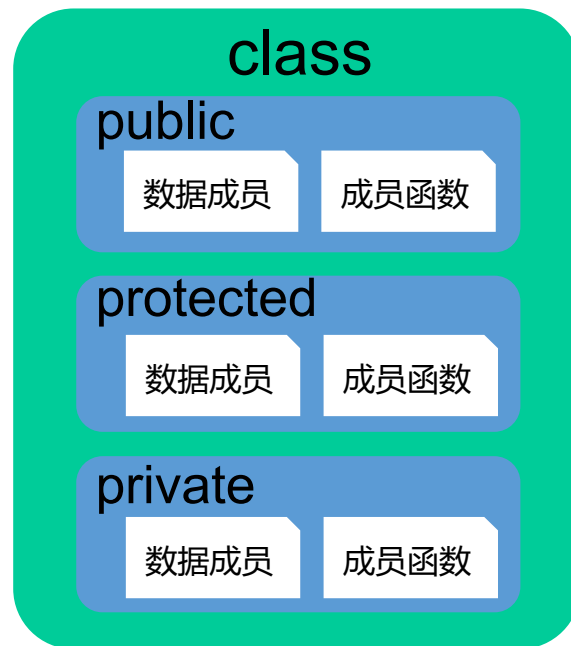
3.1 类和对象

3.1.1 类的定义

类也是一种用户自己定义的**数据类型**，和其他数据类型不同之处在于，类不仅包含了**数据**，而且还包含了对这些数据进行操作的**函数**。

□ 类定义的一般形式为：

```
class 类名{  
    public:  
        <公有数据成员和成员函数>;  
    protected:  
        <保护数据成员和成员函数>;  
    private:  
        <私有数据成员和成员函数>;  
};
```



3.1 类和对象

3.1.1 类的定义

```
// 采用结构化编程的文件访问
// 结构化程序
// 存储文件编号的变量
int fileNO;
// 打开文件的子程序
// （通过参数接收路径）
void openFile (string pathName)
{ /*省略*/ }
```

```
// 关闭文件的子程序
void closeFile () { /*省略*/ }
```

```
// 从文件读取一个字符的子程序
char readFile () { /*省略*/ }
```

数据

函数

```
// 使用类进行汇总封装
class TextFileReader {
    // 存储文件编号的变量
    int fileNO;
    // 打开文件
    // （通过参数接收路径）
    void open (string pathName)
    { /*省略*/ }

    // 关闭文件
    void close () { /*省略*/ }

    // 从文件读取一个字符
    char read () { /*省略*/ }
};
```

3.1 类和对象

3.1.1 类的定义

示例：

// 学生类的定义

```
#include <iostream>
```

```
#include <cstdio> // printf()
```

```
using namespace std;
```

```
class CStudent
```

```
{
```

```
    public:
```

```
        string name;
```

```
        int age;
```

```
        void say() {
```

```
            printf("Hello, my name is %s, %d years old.", name, age);
```

```
        };
```

```
};
```

```
void main(){  
    CStudent stu;  
    stu.name = "Li lei";  
    stu.age = 17;  
  
    stu.say();  
}
```

输出：

Hello, my name is Li lei, 17 years old.

3.1 类和对象

3.1.2 类成员的访问控制



□ **public:** 公有数据成员和成员函数

公有成员定义了类的外部接口。可以在类定义的外部，通过类或类的对象来访问其数据成员、调用成员函数；



□ **private:** 私有数据成员和成员函数

私有成员是被隐藏的成员，只能在该类定义的内部，通过类的成员函数或外部的友元函数来访问其私有数据成员和私有成员函数；



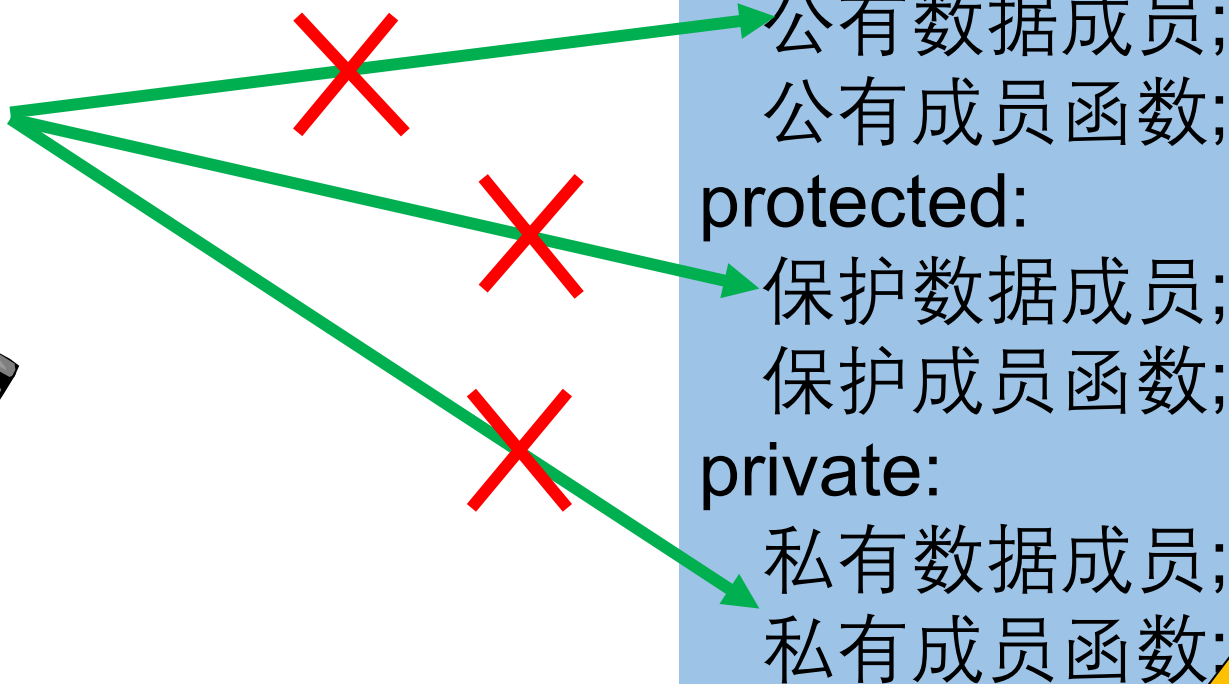
□ **protected:** 保护数据成员和成员函数

保护成员具有公有成员和私有成员的双重性质，可以被该类或派生类的成员函数或友元函数引用。

3.1 类和对象

3.1.2 类成员的访问控制

???



```
class A {  
public:
```

公有数据成员;
公有成员函数;

```
protected:
```

保护数据成员;
保护成员函数;

```
private:
```

私有数据成员;
私有成员函数;

```
};
```

成员函数

3.1 类和对象

3.1.2 类成员的访问控制

???



A a1;

```
class A {  
    public:  
        公有数据成员;  
        公有成员函数;  
    protected:  
        保护数据成员;  
        保护成员函数;  
    private:  
        私有数据成员;  
        私有成员函数;  
};
```



3.1 类和对象

3.1.2 类成员的访问控制



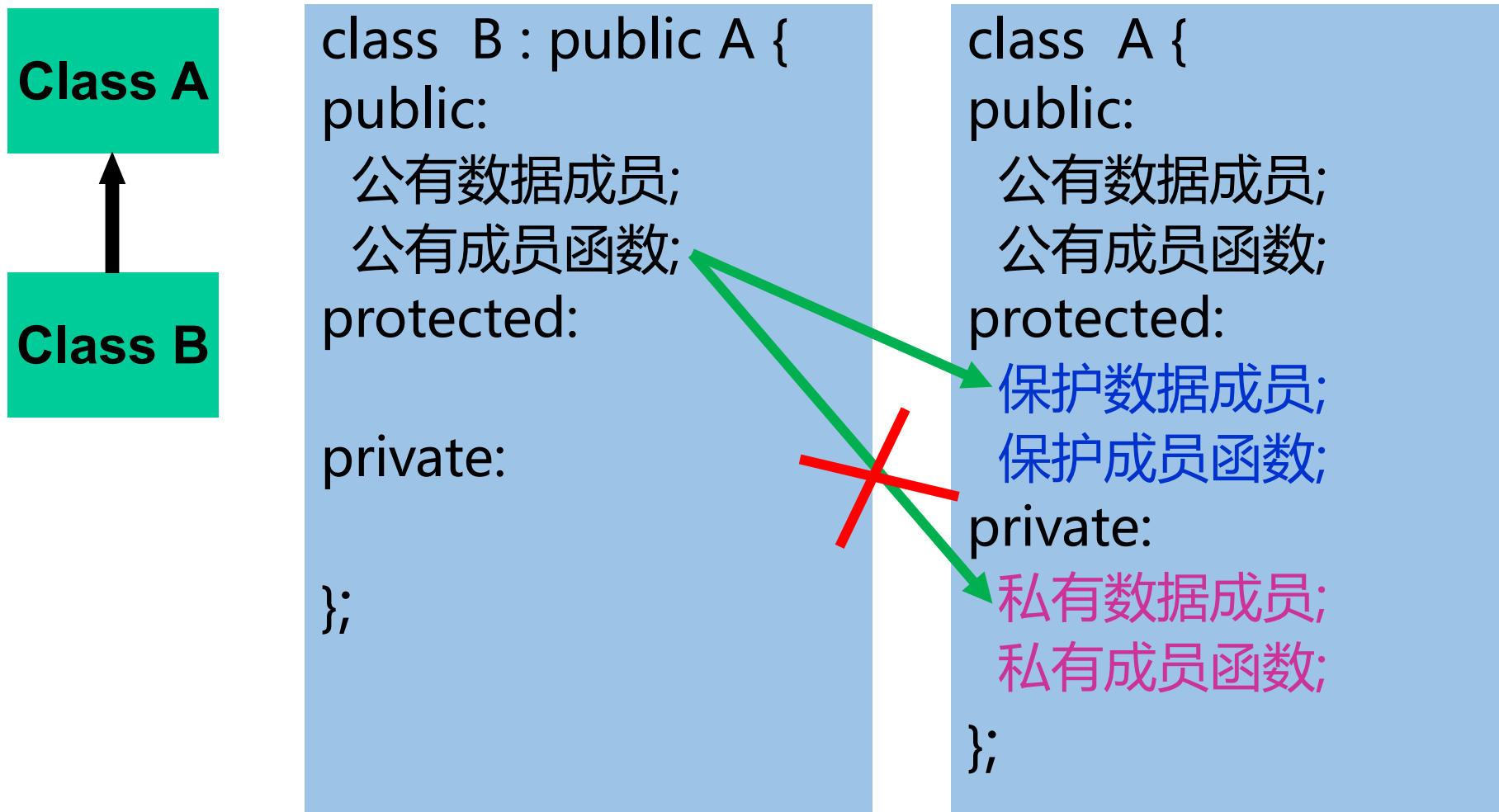
A a1;

```
class A {  
    public:  
        公有数据成员;  
        公有成员函数;  
    protected:  
        保护数据成员;  
        保护成员函数;  
    private:  
        私有数据成员;  
        私有成员函数;  
};
```

Protected 部分和
private部分在访问方式
上没有不同？

3.1 类和对象

3.1.2 类成员的访问控制



3.1 类和对象

3.1.3 成员函数的实现

成员函数实现的两种方式：

- 可以放在类体内；
- 可以放在类体外，但必须在体内给出原型说明。在类体外定义成员函数的一般形式为：

```
<返回类型> <类名>::<成员函数名> (参数说明)
{
    函数体;
}
```

其中，`::` 称为作用域运算符，`<类名>::` 表示其后的成员函数是属于该类的成员。

3.1 类和对象

3.1.3 成员函数的实现

// 定义一个点类(Point)

```
class CPoint {  
    private:  
        int x, y;  
    public:  
        void display() {  
            cout<<x<<" "<<y<<endl;  
        }  
};
```

- **成员变量**
- **数据成员**
- **属性、特性**

- **成员方法**
- **成员函数**
- **方法**

3.1 类和对象

3.1.3 成员函数的实现

// 定义一个点类(Point)

```
class CPoint {  
    private:  
        int x, y;  
    public:  
        void display();  
};  
void CPoint::display() {  
    cout << x << ", " << y << endl;  
}
```

- **成员变量**
- **数据成员**
- **属性、特性**

- **成员方法**
- **成员函数**
- **方法**

3.1 类和对象

3.1.4 对象的声明

类是一种用户自己定义的数据类型，称为类类型。可以使用这种新的类型在程序中声明新的变量，具有类类型的变量称为对象。对象的创建与声明与普通变量一致，一般格式为：

<类名> <对象名表>;

其中，<类名>是所定义的对象所属类的名字。<对象名表>中可以是一般的对象名，也可以是指向对象的指针名或对象数组。

例如：

```
class Point{  
    int x, y;  
};
```

```
Point p1, p2; // 类的对象  
Point* p = new Point(); // 对象指针  
Point pts[100]; // 对象数组
```

3.1 类和对象

3.1.5 对象的创建与销毁

思考问题：

- (1) 对象被创建时，对象的数据在内存空间存放何处？
- (2) 对象的生命周期如何？

对象的创建方式有两种：1. 直接声明对象（变量）；2. 对象指针

A.x	5
A.y	3
A.init	代码
A.GetX	代码
A.Gety	代码

B.x	6
B.y	2
B.init	代码
B.GetX	代码
B.Gety	代码

3.1 类和对象

3.1.5 对象的创建与销毁

思考问题：

- (1) 对象被创建时，对象的数据在内存空间存放何处？
- (2) 对象的生命周期如何？

对象的创建方式有两种：1. 直接声明对象（变量）；2. 对象指针
对象分配内存的方式：

- **静态分配内存**。在声明/定义对象时，系统自动分配存储空间，在对象什么周期结束时系统回收对象的存储空间，跟基本数据类型的变量类似。
如： `class Point{public: int x, y;}; Point p1, p2;`
- **动态分配内存**。通过对象指针与`new`操作符创建存储在堆中的对象（称为堆对象）。类似采用`new`动态内存申请创建的变量。堆对象的内存空间需要用户通过`delete`进行释放。
如： `Point* p1=new Point(); delete p;`

3.1 类和对象

3.1.6 对象成员的访问

对象的成员就是创建该对象的类的成员，包含数据成员和成员函数。通过对象，可以访问对象的公有成员。

<对象名> . <数据成员名>

<对象名> . <成员函数名>(<参数列表>)

<对象指针名> -> <数据成员名>

<对象指针名> -> <成员函数名>(<参数列表>)

例如：

```
class Point{  
    public:  
        int x, y;  
        void print():  
};
```

```
Point p1;  
p1.x = 10;  
p1.y = 20;  
p1.print();
```

点号

箭头

```
Point *p2=&p1;  
P2->x = 30;  
P2->y = 10;  
P2->print();
```

3.1 类和对象

3.1.6 对象成员的访问

// 学生类的定义

```
class CStudent
{
    public:
        string name;
        int age;
        void say() { /* 缺省*/ };
    private:
        int carId; //身份证
    protected:
        float score; //成绩
};
```

访问类的成员:

```
CStudent stu;
stu.name = “ 小明 ”; ( ✓ )
stu.age = 18; ( ✓ )
stu.say(); ( ✓ )

stu.carId = 420683; ( × )
int score = stu.score; ( × )
```

3.1 类和对象

- 关于类定义格式的描述中，()是错的。
 - A. 一般类的定义格式分为说明部分和实现部分
 - B. 一般类中包含有数据成员和成员函数
 - C. 类中成员有三种访问数据：公有、私有和保护
 - D. 成员函数都应是公有的、数据成员都应是私有的

3.1 类和对象

- 关于类成员函数的描述中，()是错的。
 - A. 类中可以说明一个或多个成员函数
 - B. 类中的成员函数只能定义在类体外
 - C. 定义在类体外的成员函数前加inline可以成为内联函数（内置函数）
 - D. 在类体外定义成员函数时，在函数名前除了加类型说明符外，还需作用域符来限定该成员函数所属的类

3.1 类和对象

- 已知：类A中的一个成员函数说明为void Set(A&a);其中，A&a的含义是()。
 - A. 变量A与变量a按位与后用作函数Set()的形参
 - B. 类A的对象的引用a用作函数Set()的形参
 - C. 指向类A的指针a用作函数Set()的形参
 - D. 将类A的对象a的地址值赋给变量Set

3.1 类和对象

- (问题4)指出程序中的错误

```
class point{  
    int x,y;  
    public:  
        point(int ,int);  
}  
  
point::point(int a, int b): x(a),  
    y(b) {}  
  
void print(point *xy){  
    cout<<xy->x<<","  
        <<xy->y<<endl;  
}
```

- (问题5)指出程序中的错误

```
class student{  
    protected:  
        int semesterHours;  
        float gpa;  
};  
  
class teacher{  
    public:  
        void assignGrades(student &s) {  
            s. gpa = 100.0f;  
        };  
        void adjustHours(students &s) {  
            s. semesterHours = 64;  
        };  
};
```

第三章：类和对象

- 类与对象定义
- 构造函数与析构函数
- this指针
- 子对象和堆对象
- 类的静态成员
- 类的友元
- 本章小结



3.2 构造函数与析构函数

3.2.1 构造函数

在声明类的对象时进行的数据成员设置，称为对象的初始化。C++程序中类对象的初始化和清理工作，分别由两个特色成员函数来完成，即构造函数和析构函数。

(1) 构造函数的特点

构造函数是一种特殊的成员函数，对象的创建与初始化通过构造函数来完成：

```
class 类名{  
    public:  
        类名(<形参列表>){ /* 缺省 */ };    // 构造函数的声明与定义  
};  
<类名>::<类名>(<形参列表>) { /* 缺省 */ }; // 构造函数的实现
```

3.2 构造函数与析构函数

3.2.1 构造函数

(1) 构造函数的特点

- 构造函数名与类名相同；
- 不带返回值类型；
- 可以进行构造函数重载；
- 不能被显示调用，只能在创建对象时被自动调用。

```
class Point {  
    public:  
        int x, y;  
        Point (int px=0, int py=0) {x=px; y=py};  
        Point (Point &p) {x=p.x; y=p.y;}  
};  
Point p1(1, 2);    Point p2(p1);
```

3.2 构造函数与析构函数

3.2.1 构造函数

(2) 默认构造函数

默认构造函数是指不带参数的构造函数，即：

<类名>::<类名>();

如果用户没有显示地声明默认构造函数，则C++编译器会自动创建一个默认构造函数。

```
class Point{  
    public:  
        int x, y;  
        Point();    // 默认构造函数  
        Point(int x, int y); // 重构的构造函数  
};
```

```
Point p1;    Point p2(1, 3);    Point* p3=new Point();
```

3.2 构造函数与析构函数

3.2.1 构造函数

(3) 构造函数初始化列表

也可用带有成员初始化表的构造函数来对数据成员进行初始化工作，其一般形式如下：

```
类名 :: 构造函数名 (<参数表>) : <成员初始化表>
{
    // 构造函数体
}
```

初始化列表的一般形式为：

数据成员名1(初始值1), 数据成员名2(初始值2),

3.2 构造函数与析构函数

3.2.1 构造函数

(3) 构造函数初始化列表

```
class Date{  
    public:  
        Date(int y, int m, int d);  
    private:  
        int year, month, day;  
};  
  
Date::Date (int y, int m, int d)  
    : year(y), month(m), day(d) { }
```

3.2 构造函数与析构函数

3.2.2 拷贝构造函数

拷贝构造函数是一种特殊的构造函数，它的作用是用一个已经存在的对象去初始化另一个对象。为了保证不修改所引用的对象，通常将引用参数声明为const参数，格式如下：

```
<类名> :: <类名>(const <类名> &<对象名>)  
{  
    <函数体>  
}
```

拷贝构造函数的特点：

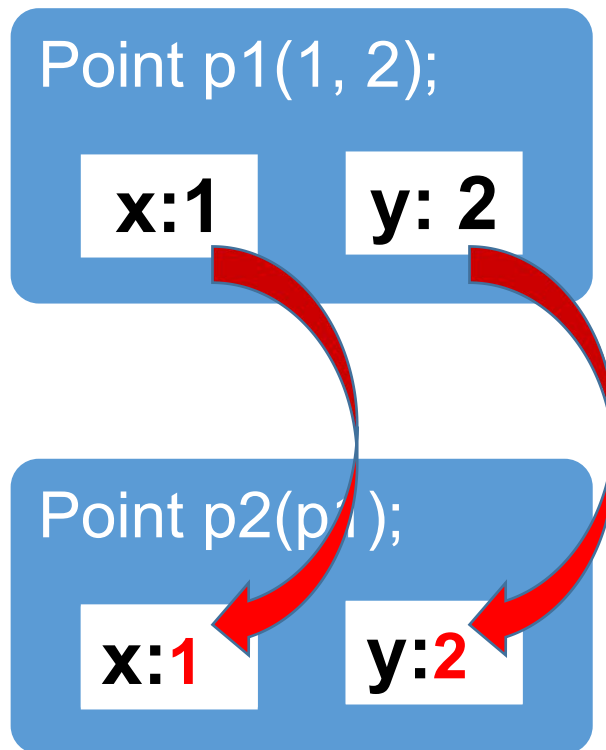
- (1) 函数名与类名相同，参数是该类对象的引用；
- (2) 不能显示调用，只有在①用类的一个对象去初始化该类的另一个对象时；②类的对象作为函数实参按值传递时；③函数的返回值为类的对象，函数执行完需要返回对象时。

3.2 构造函数与析构函数

3.2.2 拷贝构造函数

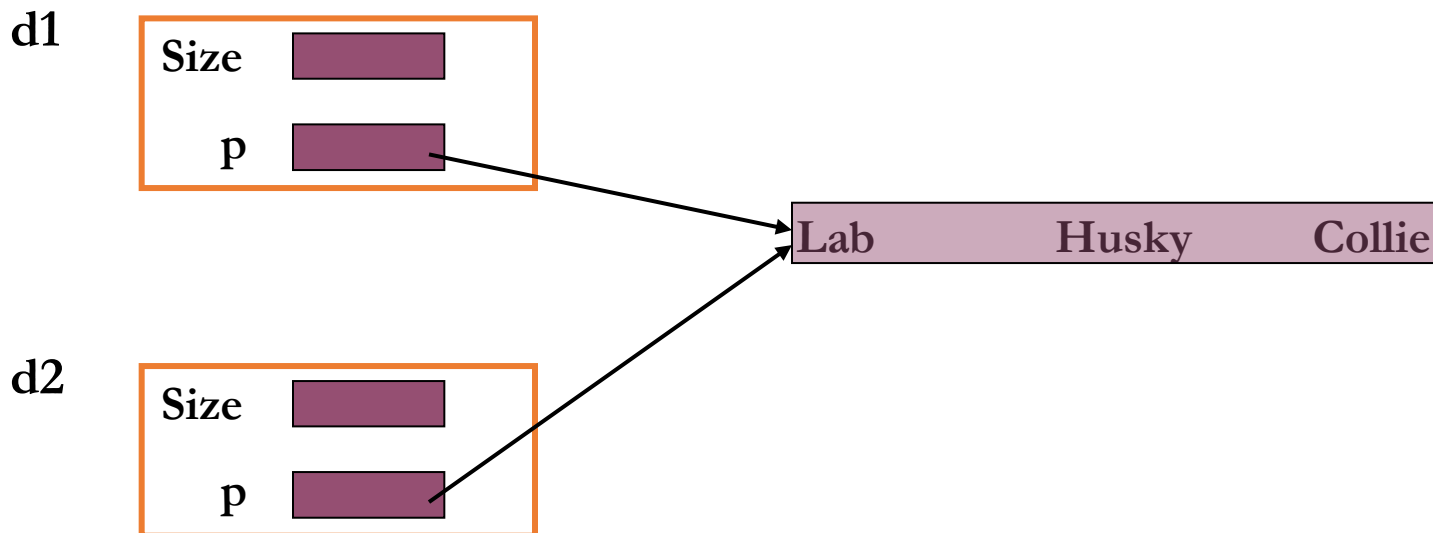
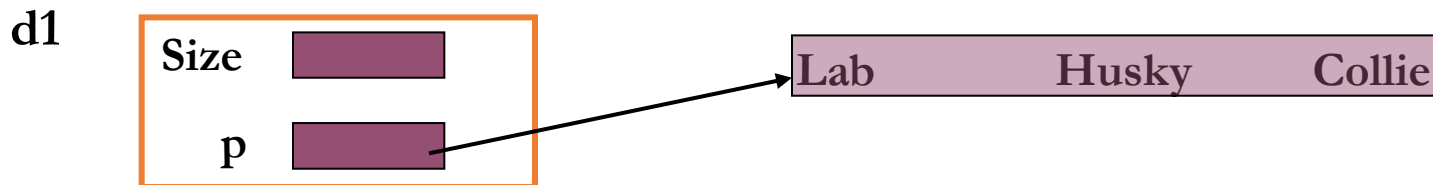
如果一个类中没有定义拷贝构造函数，则编译器会自动生成一个默认的拷贝构造函数。该函数的功能是将已知对象的所有数据成员的值拷贝给对应的对象的所有数据成员。

```
Class Point{  
public:  
    int x, y;  
    Point(int a, int b);  
    Point(const Point& p);  
};  
  
Point::Point(int a, int b):  
x(a), y(b) { /*略*/ }
```



3.2 构造函数与析构函数

3.2.2 拷贝构造函数



3.2 构造函数与析构函数

3.2.3 浅拷贝 & 深拷贝

浅拷贝就是由缺省的拷贝构造函数所实现的数据成员逐一赋值。如果类中含有指针类型的数据，浅拷贝就会产生错误。

```
class Point {  
public:  
    int *coord;    // 坐标(x, y, z)  
    int id; // 点ID  
    Point(int* data, int s);  
    ~Point( );  
};
```

```
Point::~~Point()  
{    cout<<"destructing..."<<id<<endl;  
    delete [ ]coord; }
```

3.2 构造函数与析构函数

3.2.3 浅拷贝 & 深拷贝

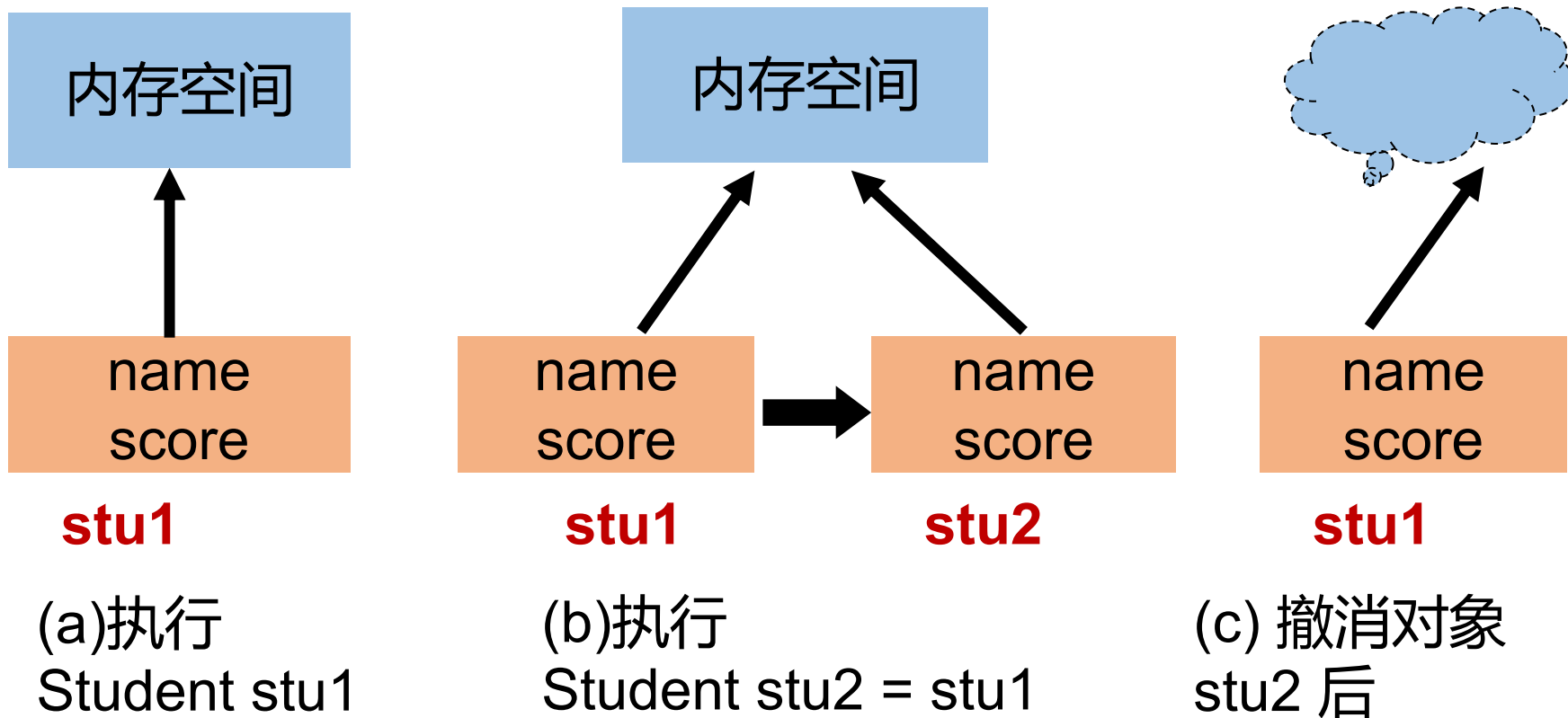
浅拷贝就是由缺省的拷贝构造函数所实现的数据成员逐一赋值。如果类中含有指针类型的数据，浅拷贝就会产生错误。

```
Point::Point(int *data, int s) {  
    cout<<"constructing..."<<s<<endl;  
    coord = data; id = s;  
}  
  
int main() {  
    int* data = new int[3]();  
    Point p1(data, 1);  
    Point p2=p1;    p2.id = 2;    p2.coord[0] = 2;  
    cout<<p1.coord[0]<<" "<<p1.coord[1]<<" "<<p1.coord[2]<<endl;  
    return 0;  
}
```

运行结果:
constructing...1
2 0 0
destructing...2
destructing...1

3.2 构造函数与析构函数

3.2.3 浅拷贝 & 深拷贝



3.2 构造函数与析构函数

3.2.3 浅拷贝 & 深拷贝

深拷贝：为了解决浅拷贝出现的问题，就**必须显式地定义一个自己的拷贝构造函数**，使得不仅拷贝数据成员，还为对象分配各自的内存空间，这就是所谓的**深拷贝**。

```
Point::Point(const Point& p) {  
    cout<<"Copy constructing..."<<p.id<<endl;  
    coord=new int[3]();  
    if (coord !=nullptr) {  
        memcpy(coord, p.coord);  
        id=p.id;  
    }  
}
```

运行结果:

constructing...1

Copy constructing 李宁

Destructing...李宁

Destructing...李宁

3.2 构造函数与析构函数

3.2.4 析构函数


析构函数也是一种特殊的类成员函数，它的作用是在对象消失时执行一些清理任务（如：释放由构造函数分配的内存等）。

```
<类名> :: ~<类名>()  
{  
    <函数体>  
}
```

析构函数也只能声明为公有函数，因为它在释放对象时被自动调用。析构函数的特点：

- （1）函数名与类名相同，与构造函数不同之处在于函数名前增加了“~”符号，表示与构造函数功能相反。
- （2）析构函数没有参数，因此不能重载，唯一性；
- （3）不带返回值类型，不能被显示调用。

第三章：类和对象

- 类与对象定义
- 构造函数与析构函数
- this指针 
- 子对象和堆对象
- 类的静态成员
- 类的友元
- 本章小结

3.3 this指针

3.3.1 this指针的背景

当定义了一个类的若干个对象后，每个对象都有属于自己的数据成员，但所有对象的成员函数代码却合用一份。

成员函数是怎样辨别当前调用自己的是那个对象，从而对当前调用自己的对象的数据成员进行操作的？

```
class A {  
    public:  
        A(int x) { this->x = x; }  
        void disp(){ cout<<"x= "<< this->x <<endl;}  
    private:  
        int x; };  
        A a(10); a.disp();
```

3.3 this指针

3.3.2 理解this指针本质

this指针是**每个对象中隐藏的指针**。this指针是默认的当一个对象生成后，这个对象的**this指针就指向内存中保存该对象数据的存储空间的首地址**。在类的成员函数中使用这个this指针，就好像this指针是类的一个自动隐藏的私有成员一样。

当通过一个对象调用成员函数时，系统**先将该对象的地址赋给this指针，然后调用成员函数，成员函数对对象的数据成员进行操作时，隐含使用了this指针**。

3.3 this指针

3.3.2 理解this指针本质

this指针是一个const类指针，其指向类的实例化对象，存储对象在内存空间的地址，不能在程序中修改或赋值。this指针是一个局部变量，其作用域仅在一个对象内部。

```
#include <iostream>
using namespace std;
class Student
{
    string name; float score;
public:
    void getname() {cout <<name<<endl;}
};
int main() { Student p; p.getname(); }
```


Student p1, p2;

p1 Name: Li ning
Score: 98

p2 Name: Zhang yi
Score: 78

p1.getname(this);
p2.getname(this);

第三章：类和对象

- 类与对象定义
- 构造函数与析构函数
- this指针
- 子对象和堆对象 
- 类的静态成员
- 类的友元
- 本章小结

3.4 子对象和堆对象

3.4.1 子对象

一个对象作为另一个类的成员时，该对象称为类的子对象。子对象实际上是某个类的数据成员。子对象的一般形式：

```
class <类名X>{  
    ...  
    <类名1> <子对象1>;  
    <类名2> <子对象2>;  
    ...  
};
```

例如：

```
class A { /*略*/};  
class B {  
    ...  
    private:  
        A a; //子对象  
    ...  
};
```

3.4 子对象和堆对象

3.4.2 子对象的初始化

子对象的初始化一般采用类的构造函数初始化列表。形式如下：

```
class Part {  
    int x;  
    public:  
        Part(int a) : x(a) {}  
};  
class Whole {  
    Part p1, p2; //子对象  
    public:  
        Whole(int a, int b);  
};  
  
Whole::Whole(int a, int b)  
: p1(a), p2(b)  
{ }
```

3.4 子对象和堆对象

3.4.3 堆对象

通过动态内存管理分配存储空间的对象称为堆对象，创建和删除堆对象的两个运算符：new 和 delete

(1) 创建堆对象

<类名>* <对象指针名> = new <类名>(<初始列表>);
<类名>* <对象指针名> = new <类名>[对象数组大小];

例如：

```
class Point{  
public:  
    int x, y;  
    Point(int a, int b){x=a; y=b;}  
};
```

```
Point *p = new Point(2, 3);  
Point *pA = new Point[100];
```

3.4 子对象和堆对象

3.4.3 堆对象

通过动态内存管理分配存储空间的对象称为堆对象，创建和删除堆对象的两个运算符：new 和 delete

(2) 删除/销毁堆对象

delete <对象指针名>;
delete[] <对象指针名>;

例如：

```
class Point{  
public:  
    int x, y;  
    Point(int a, int b){x=a; y=b;}  
};
```


```
Point *p = new Point(2, 3);  
Point *pA = new Point[100];
```

```
delete p;  
delete[] pA;
```


3.4 子对象和堆对象

- 下列关于对象的描述中，()是错误的。
 - A. 子对象是类的一种数据成员，它是另一个类的对象
 - B. 子对象不可以是自身类的对象
 - C. 对子对象的初始化要包含在该类的构造函数中
 - D. 一个类中只能含有一个子对象作其成员

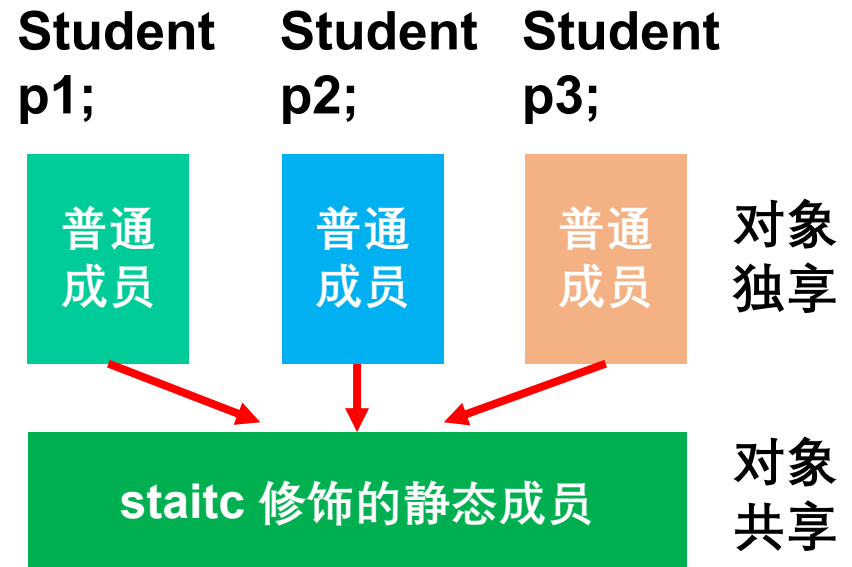
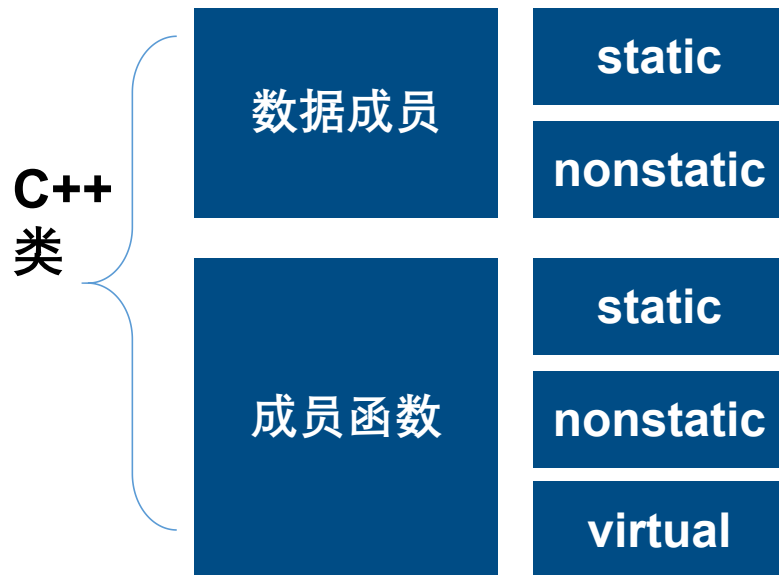
第三章：类和对象

- 类与对象定义
- 构造函数与析构函数
- this指针
- 子对象和堆对象
- 类的静态成员 
- 类的友元
- 本章小结

3.5 类的静态成员

3.5.1 静态成员 (static)

在一个类中，若将一个数据成员或成员函数说明为static，这种成员称为类的**静态成员**。静态成员是类的所有对象共享的数据成员或成员函数，不与具体的某个对象关联。主要用于实现不同对象之间的数据共享。静态数据成员类似全局变量，而静态成员函数类似全局函数。



3.5 类的静态成员

3.5.1 静态数据成员

在一个类中，若将一个数据成员说明为static，这种成员称为**静态数据成员**。

与一般的数据成员不同，无论建立多少个类的对象，都只有一个静态数据的拷贝。从而实现了同一个类的不同对象之间的数据共享。

定义静态数据成员的格式如下：

```
static 数据类型 静态数据成员名;
```

```
<数据类型> <类名>::静态数据成员名=<初始值>
```

3.5 类的静态成员

3.5.2 静态数据成员的使用示例

```
class Student {  
    public:  
        Student(char *name1,char *stu_no1,float score1);  
        ~Student();  
        void show();                // 输出姓名、学号和成绩  
        void show_count_sum_ave(); // 输出学生人数、总成绩和平均成绩  
    private:  
        char *name;                // 学生姓名  
        char *stu_no;              // 学生学号  
        float score;               // 学生成绩  
        static int count;           // 静态数据成员，统计学生人数  
        static float sum;           // 静态数据成员，统计总成绩  
        static float ave;           // 静态数据成员，统计平均成绩  
};
```

3.5 类的静态成员

3.5.2 静态数据成员的使用示例

```
Student::Student(char *name1,char *stu_no1,float score1 )
{
    name=new char[strlen(name1)+1];  strcpy(name,name1);
    stu_no=new char[strlen(stu_no1)+1]; strcpy(stu_no,stu_no1);
    score=score1;
    ++count;                        // 累加学生人数
    sum=sum+score;                  // 累加总成绩
    ave=sum/count;                  // 计算平均成绩
}
Student::~~Student()
{
    delete []name;
    delete []stu_no;
    --count;
    sum=sum-score;
}
```

3.5 类的静态成员

3.5.2 静态数据成员的使用示例

```
void Student::show()
{
    cout<<"\n name: "<<name;
    cout<<"\n stu_no: "<<stu_no;
    cout<<"\n score: "<<score; }
void Student::show_count_sum_ave()
{
    cout<<"\n count: "<<count; // 输出静态数据成员count
    cout<<"\n sum: "<<sum; // 输出静态数据成员sum
    cout<<"\n ave: "<<ave; // 输出静态数据成员ave
}

int Student::count=0; // 静态数员count初始化
float Student::sum=0.0; // 静态数员sum初始化
float Student::ave=0.0; // 静态数员ave初始化
```

3.5 类的静态成员

3.5.2 静态数据成员的使用示例

```
void main()
```

```
{
```

```
    Student stu1("Lining","1901",90);
```

```
        stu1.show();
```

```
        stu1.show_count_sum_ave();
```

```
    Student stu2("Zhangyi","1902",85);
```

```
        stu2.show();
```

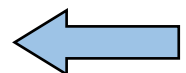
```
        stu2.show_count_sum_ave();
```

```
}
```

静态成员
count=2

静态成员
sum=175

静态成员
ave=87.5



3.5 类的静态成员

3.5.3 关于静态数据成员的几点说明

1. 静态数据成员的定义要加上关键字static
2. 静态数据成员的初始化应在类申明之后，定义对象之前，在类外单独进行初始化格式为：

数据类型 类名::静态数据成员名 = 值

如： `int Student::count = 0;`

3. 静态数据成员属于类，而不像普通数据成员那样属于某一对象。因此可以使用以下方式访问静态数据成员：

类名::静态数据成员名

如： `Student::count`

3.5 类的静态成员

3.5.3 关于静态数据成员的几点说明

4. 静态数据成员与静态变量一样，在编译时创建并初始化。
它在该类的任何对象被建立之前就存在。因此，公有的静态数据成员可以在对象定义之前被访问。对象定义后，也可以通过对象访问公有的静态数据成员，访问格式为：

对象名.静态数据成员名

对象指针->静态数据成员名

5. 私有静态数据成员不能被类外部函数访问，也不能用对象进行访问。
6. C++支持静态数据成员的一个主要原因是可以不必使用全局变量。依赖于全局变量的类几乎都是违反面向对象程序设计的封装原理。

3.5 类的静态成员

3.5.4 静态成员函数

静态成员函数属于整个类，是该类所有成员共享的成员函数。定义静态成员函数的格式如下：

static 返回类型 静态成员函数名（参数表）；

与静态数据成员类似，调用公有静态成员函数的一般格式有如下几种：

类名::静态成员函数名(实参表)

对象. 静态成员函数名(实参表)

对象指针->静态成员函数名(实参表)

3.5 类的静态成员

3.5.5 静态成员函数的使用示例

```
#include <cmath>
class MyMath {
public:
    static double PI;    // PI
    static double sin(double deg);
    static double cos(double deg);
};
```

```
MyMath::PI = 3.1415926;
double MyMath::sin(double deg) {return ::sin(deg/180*PI);}
double MyMath::cos(double deg) {return ::cos(deg/180*PI);}
```

使用方法: **MyMath::sin(30);**

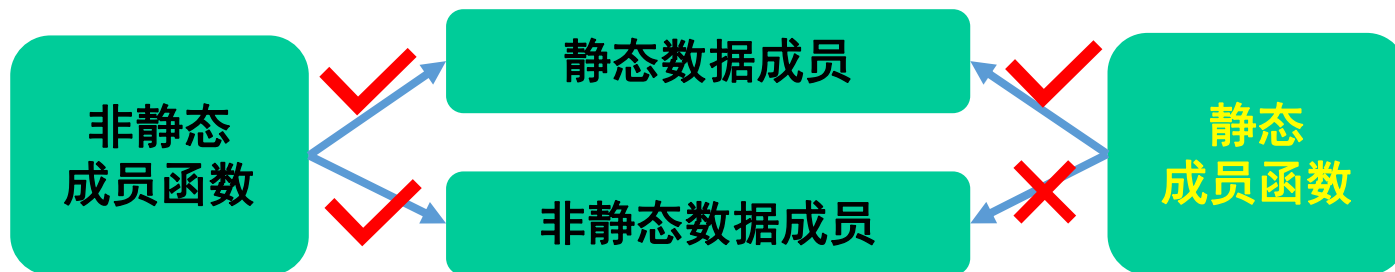
输出: **0.5**

3.5 类的静态成员

3.5.6 关于静态成员函数的几点说明

1. 静态成员函数可以定义为内嵌，也可以在类外定义，**在类外定义时不能加关键字static**
2. 因为静态成员函数不具体作用于某个对象，一般情况下，**静态成员函数内部不能访问非静态成员变量，也不能调用非静态成员函数。**

思考题：为什么在静态成员函数内不能调用非静态成员函数？



3.5 类的静态成员

➤ 静态成员函数访问非静态成员

```
class A {  
public:  
    static void test() {  
        m_staticA += 1;  
    }  
private:  
    static int m_staticA;  
    int m_a;  
};
```



```
class A {  
public:  
    static void test(A *a)  
    {  
        a->m_a += 1;  
    }  
    void hello() { }  
private:  
    static int m_staticA;  
    int m_a;  
};
```

3.5 类的静态成员

➤ 静态成员函数访问非静态成员

```
class A {  
public:  
    static void test() {  
        m_staticA += 1;  
    }  
private:  
    static int m_staticA;  
    int m_a;  
};
```



```
class A {  
public:  
    A() { m_gA = this;}  
    static void test() {  
        m_gA.m_a += 1;  
    }  
    void hello() { }  
private:  
    static int m_staticA;  
    static A *m_gA;  
    int m_a;  
};
```

3.5 类的静态成员

3.5.6 关于静态成员函数的几点说明

1. 静态成员函数可以定义为内嵌，也可以在类外定义，**在类外定义时不能加关键字static**
2. 因为静态成员函数不具体作用于某个对象，一般情况下，**静态成员函数内部不能访问非静态成员变量，也不能调用非静态成员函数。**
3. 私有静态成员函数不能被类外部函数和对象访问。
4. 静态成员函数可在建立任何对象之前处理静态数据成员，普通成员函数则不能。

3.5 类的静态成员

3.5.6 关于静态成员函数的几点说明


5. 编译系统将静态成员函数限定为内部连接，也就是说，与现行文件相连接的其它文件中的同名函数不会与该函数发生冲突。
6. 在一般的成员函数中都隐含一个this指针，用来指向对象本身，而**静态成员函数则没有this指针**。
7. 静态成员函数一般不访问类中的非静态成员。若确实需要，静态成员函数只能通过对象名（或指向对象的指针）访问非静态成员。

3.5 类的静态成员

3.5.6 关于静态成员函数的几点说明

- 在静态成员的描述中，()是错的。
 - A. 静态成员分为静态数据成员和静态成员函数
 - B. 静态数据成员定义后必须在类体内进行初始化
 - C. 静态数据成员初始化不使用其构造函数
 - D. 静态成员函数中不能直接引用非静态成员

第三章：类和对象

- 类与对象定义
- 构造函数与析构函数
- this指针
- 子对象和堆对象
- 类的静态成员
- 类的友元 
- 本章小结

3.6 类的友元

3.6.1 友元 (friend)

友元是一扇通向私有成员(保护成员)的后门。友元既可以是不属于任何类的一般函数，也可以是另一个类的成员函数，还可以是整个的类。

在类里声明一个普通函数或类的成员函数或类，加上关键字 friend，就成为了该类的友元函数或友元类，它可以访问该类的切成员。其原型为：

```
class 类名 {  
    ....  
    friend 数据类型 友元函数名(<参数列表>);  
    friend class 友元类名;  
};
```

3.6 类的友元

3.6.2 友元的使用示例

友元函数（或友元类）不是当前类的成员函数（或数据成员），而是独立于当前类的外部函数（或类），但它可以访问该类的所有对象的成员，包括私有成员、保护成员和公有成员。

```
class Point{  
public:  
    Point(double xi, double yi) {X = xi; Y=yi; }  
    friend double dist (Point &a, Point &b);  
private:  
    int X, Y;  };
```

```
double dist(Point &a, Point &b){  
    double dx = a.X-b.X; double dy=a.Y-b.Y;  
    return sqrt(dx*dx+dy*dy); };
```

```
void main()  
{  
    Point p1(3,5),  
          p2(4,6);  
    double d = dist(p1, p2);  
    cout<<"distance="<<d;  
}
```

输出：
distance=1.41421

3.6 类的友元

3.6.2 友元的使用示例

```
class Student{  
public:  
    friend class Teacher;  
    Student(){}  
private:  
    int id, score;  };
```

```
class Teacher{  
public:  
    Teacher(int I, int J){a.id=I; a.score=J;}  
    void disp(){cout<<"No="<<a.id<<" , Score="<<a.score;}  
private:  
    Student a; };
```

```
void main()  
{  
    Teacher t1(1001,89), t2(1002, 78);  
    t1.disp();  
    t2.disp();  
}
```

输出:

No=1001, Score=89

3.6 类的友元

3.6.3 关于友元的几点说明


1. **友元函数不是成员函数**，在类外部定义友元函数时不必在函数名前加上“类名::”
2. 友元函数一般带有一个该类的入口参数。
3. 当一个函数需要访问多个类时，友元函数就显得非常重要。
4. 友元的作用是提高编程效率，但友元破坏了类的整体性与封装性，与面向对象的程序设计思想相背，应当谨慎使用。

3.6 类的友元

3.6.3 关于友元的几点说明

1. 友元关系是单向的，不具备交换性
即若 $A \rightarrow B$ ，并不等于 $B \rightarrow A$
2. 友元关系不具有传递性
即若 $A \rightarrow B$ ， $B \rightarrow C$ ，并不能推出 $A \rightarrow C$

第三章：类和对象

- 类与对象定义
- 构造函数与析构函数
- this指针
- 子对象和堆对象
- 类的静态成员
- 类的友元
- 本章小结 

本章小结

重点:

- 类的定义 (class)
- 对象成员的访问
- 构造函数、析构函数
- 拷贝构造函数
- 类的静态成员

难点:

- 浅拷贝、深拷贝
- 静态成员函数
- 类的友元