计算机科学基础II

第四章 类与对象

曹鹏
Email: caopeng@seu.edu.cn
Tel: 13851945861

课程班级群

群名称:计算机科学基础II(2022-202...

群　号:467263928

老师	学生
课件及其他学习资料发布	
作业发布	作业提交
批改结果发布	
其他通知发布	
答疑	

考核方式

	占比	说明
期中机试	20%	
期末笔试	40%	
期末机试	30%	
平时成绩	10%	考勤5%，作业5%

- 期中机试(20%)+平时成绩(10%)不涉及补考，补考仅针对期末笔试(40%)+期末机试(30%)

教学日程

	授课 (教一402)	实验 (计算中心机房)
第1周	-	-
第2周	周一8-9节 周四3-5节	-
第3~16周	周一8-9节	周四3-5节

课时分配

	教学	实验
第4章：类与对象	6	4
第6章：模板与数据结构	9	6
第7章：动态内存分配	10	8
第8章：派生与继承	9	6
第9章：流类库与输入/输出	6	4
第10章：异常处理	4	
总计	44	28

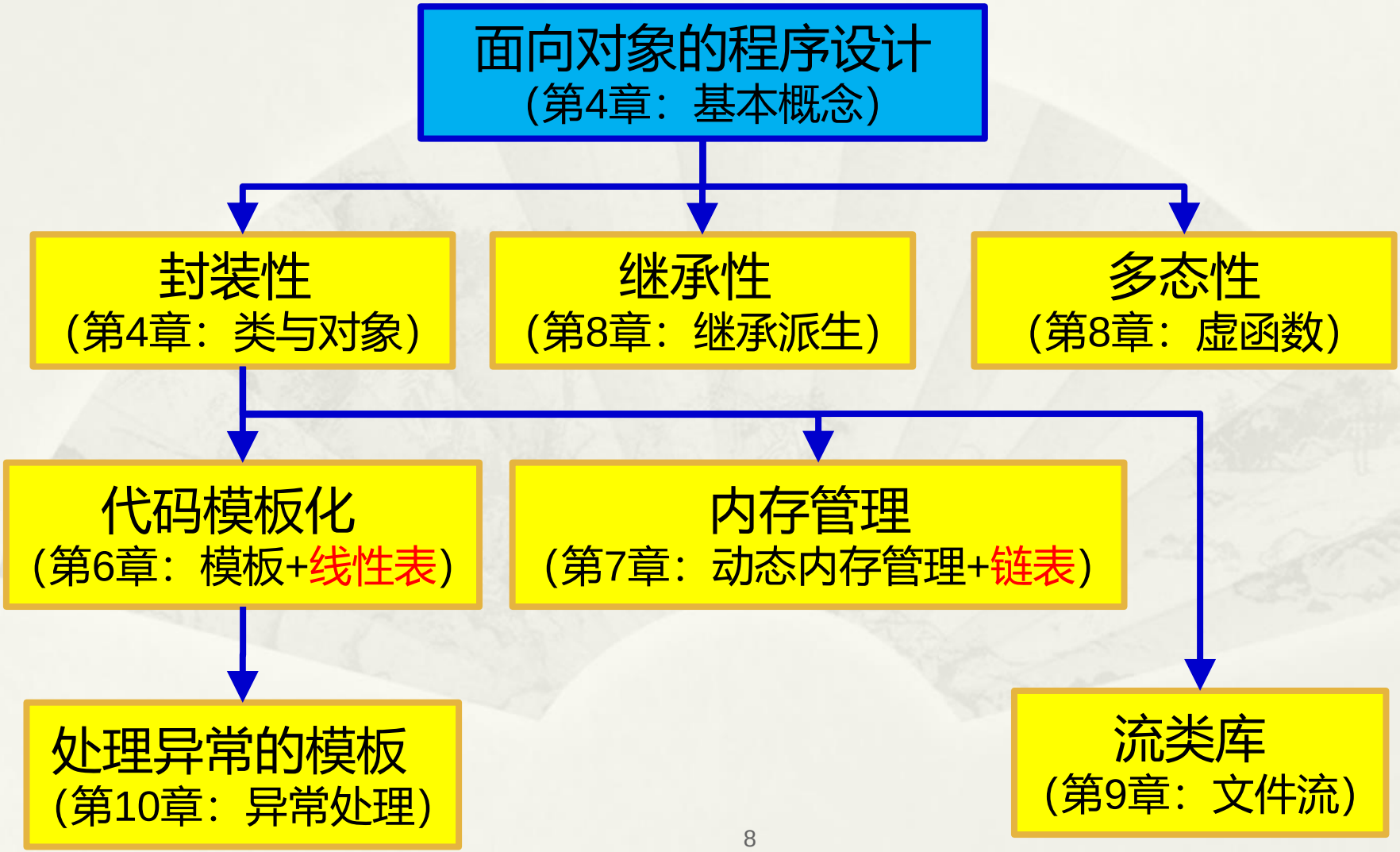
教材与参考书

- 教材
  - 《C++程序设计》第2版，吴乃陵、况迎辉，高等教育出版社，2006
  - 《C++程序设计实践教程》第2版，吴乃陵、李海文，高等教育出版社，2006
- 参考书
  - 《C++ Primer》，第五版，Stanley B.Lippman, Josee Lajoie, Barbara E .Moo 著
  - 《C++程序设计》(第2版)，谭浩强，清华大学出版社，2011.8
  - 《21天学通C++》，第7版，Siddhartha Rao著，袁国忠译，人民邮电出版社，2012)
  - 《C++ Primer Plus》，第六版，Stehpen Prata著,张海龙、袁国忠译，人民邮电出版社，2012
  - 《C++程序设计思想与方法》 翁惠玉编著 人民邮电出版社,2012

其他课前要说的.....

- 课内对例题代码上机验证，多运行，多调试
  - 提供教材/讲义电子版
- 作业
  - 每章1次作业，2~3题编程题，提交*.cpp代码
  - 不要所有代码放在一个.cpp文件
  - 不要代码放在.txt，.doc文件
- 课后答疑
  - 邮件(caopeng@seu.edu.cn)或qq
  - 主动告知姓名学号
  - 代码问题必须提供.cpp，切勿仅提供截图

教学内容


面向对象的程序设计
(第4章：基本概念)
封装性
(第4章：类与对象)
继承性
(第8章：继承派生)
多态性
(第8章：虚函数)
代码模板化
(第6章：模板+线性表)
内存管理
(第7章：动态内存管理+链表)
处理异常的模板
(第10章：异常处理)
流类库
(第9章：文件流)

本章提纲

- 面向对象的程序设计（4.2, 4.10节）
- 结构的基本概念*（4.8节）
- 类与对象的基本概念（4.1 节）
- 构造函数、复制构造函数与析构函数（4.3,4.4节）
- this指针*（5.4节）
- 运算符重载函数（4.5节）
- 友元（4.6节）
- 静态成员（4.7节）

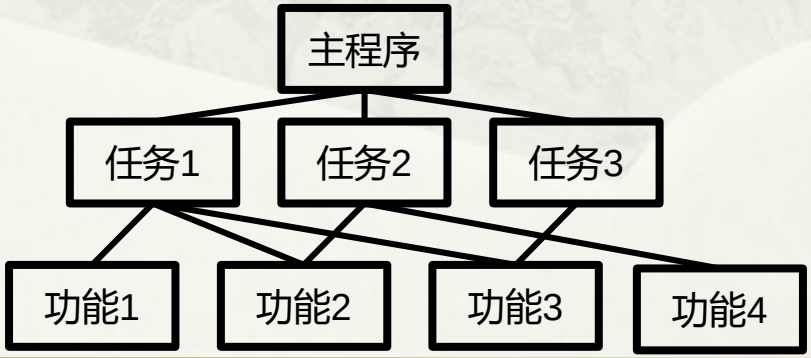
面向对象的程序设计

C++语言

- 既面向对象，也面向过程
- 封装：类；对象

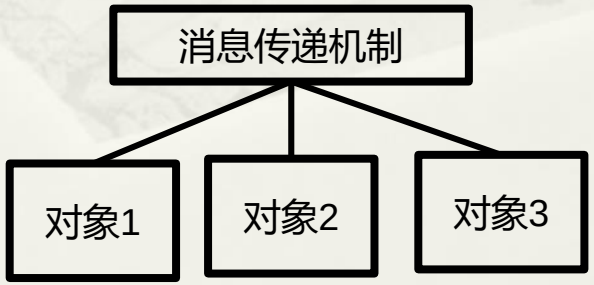
面向过程的程序设计

- 模块化方式组织
- 程序=算法+数据结构



面向对象的程序设计

- 消息传递方式组织
- 程序=（对象+对象+…）+消息
- 对象=算法+数据结构



面向对象的程序设计

- 封装性
- 继承性（派生性）
- 多态性

面向对象的程序设计

- 封装性
  - 将数据（属性）和函数（操作）封装为一个整体，称为类（非基本数据类型）
    - 类（个人）←属性（姓名，年龄）+操作（修改姓名，读取姓名）
  - 类实例化为对象（变量）
    - 对象（张三、李四）←类（个人）
- 继承性（派生性）
- 多态性

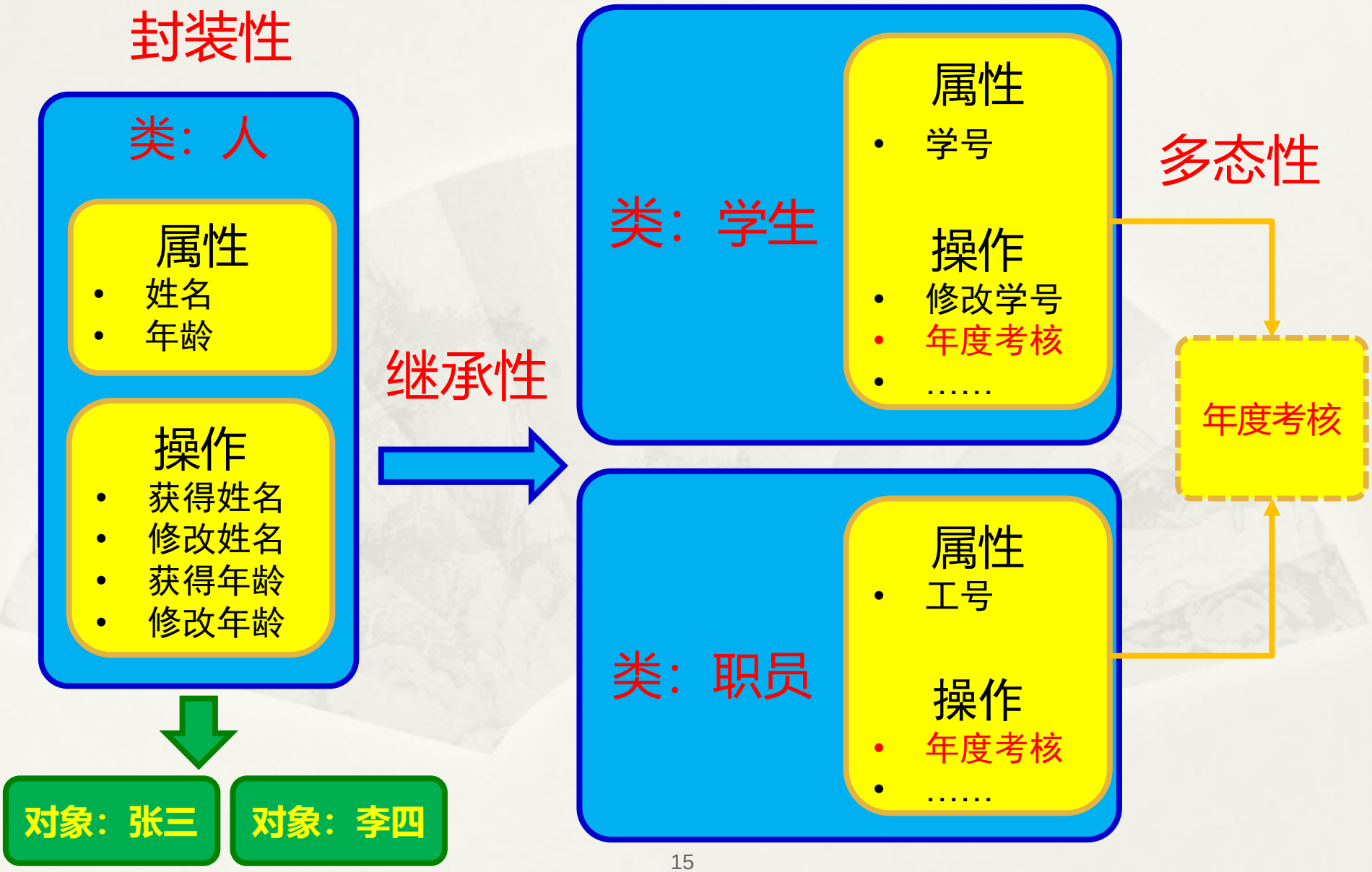
面向对象的程序设计

- 封装性
- 继承性（派生性）
  - 类（个人）←属性（姓名，年龄）+操作（修改姓名，读取姓名）
  - 类（学生）←属性（姓名，年龄，学号）+操作（修改姓名，读取姓名，修改学号）
  - 学生类由个人类继承（派生）而来，称为子类，继承了个人类（父类）的所有属性和操作
  - 在继承的基础上可增加新的属性和操作
- 多态性

面向对象的程序设计

- 封装性
- 继承性（派生性）
- 多态性
  - 个人→学生，操作（年度考核）：根据已修学分、考试成绩
  - 个人→职员，操作（年度考核）：根据工作量、业绩
  - 学生和工人是不同的人，都需要进行年度考核，但计算方法不同→同一接口，多种方法
  - 多态性以继承性为前提，表现于同一父类的不同子类间

面向对象的程序设计


封装性
类：人
属性
姓名
年龄
操作
获得姓名
修改姓名
获得年龄
修改年龄
对象：张三
对象：李四
继承性
类：学生
属性
学号
操作
修改学号
年度考核
……
类：职员
属性
工号
操作
年度考核
……
多态性
年度考核

结构

- 结构的概念
  - 非基本数据类型
  - 程序员自定义，用于组合数据的复合数据类型
  - 被组合的数据可以属于不同的数据类型，称为结构成员

结构

- 结构的使用
  - 结构类型的定义
  - 结构变量的定义
    - 结构变量的初始化
    - 结构变量的操作（读/写）
  - 结构指针的定义

结构类型的定义

◆定义格式

```
struct 结构类型名{
        类型名 变量1;
        类型名 变量2;
};
```

◆例子

```
struct complex{  //复数结构类型
        double real;  //实部
        double image;  //虚部
};
```

结构变量的定义

◆定义格式

①struct 结构类型名 结构变量名;
②结构类型名 结构变量名;
③定义结构类型的同时定义结构变量

◆例子

```
struct complex x;  //①
complex x;  //②
struct complex{
        double real;
        double image;
}x;  //③
```

结构指针的定义

◆定义格式

①struct 结构类型名 *结构指针名;
②结构类型名 *结构指针名;
③定义结构类型的同时定义结构指针

◆例子

```
struct complex *px;  //①
complex *px;  //②
struct complex{
        double real;
        double image;
}*px;  //③
```

结构变量的初始化

◆初始化格式

结构类型名 结构变量名={初始化列表};

◆例子

complex x={1.0, 2.0};

结构变量的操作

◆ 对整体操作

```
complex a={1.0, 2.0}, b;
b=a;
```

◆ 对不同成员操作

- ◆ 结构变量：操作符为.

```
complex x;
x.real=3.0;
```

- ◆ 结构指针：操作符为->

```
complex x, *py;
py=&x;
py->real=3.0;
```

为什么用->？
因为py还是real是指针变量?

```
struct s{
    int i;
    char *c;
} ss, *ps;
```

以下操作哪些错误

A) ss.i; 

B) ss->c; 

C) ps.i; 

D) ps->c; 

类

- 类的概念
  - 非基本数据类型
  - 程序员自定义，用于封装数据和操作的复合数据类型
  - 被封装的数据和操作称为类的成员
  - 可以作为函数参数类型，也可以作为函数返回值类型
- 类的成员
  - 数据：标识类的属性
  - 函数：标识类支持的操作

类

- 成员访问限定
  - 3种访问限定符：公有(public)、保护(protected)、私有(private)
  - 默认为私有(private)类型
  - 只有公有类型成员才可以在类外部访问；其他只能在类内部访问
    - 类外部：通过类对象
    - 类内部：在类的成员函数中

	公有	私有	保护
类外部（类对象）	√		
类内部（类成员函数）	√	√	√

类的定义

◆定义格式

class 类名{访问限定符：成员列表；};

◆例子

```
class complex{  //复数类                                  类头
private:
        double real;  //实部                    数据成员的定义
        double image;  //虚部                                  类体
public:
        double getreal(); //读取实部            函数成员的声明
        void     setreal(double); //修改实部
};
```

数据成员是私有的，函数成员是公有的，为什么这么定义呢？

函数成员的定义

◆在类定义外定义（类定义中只声明）

返回值类型 类名::函数名（参数表）{…};

◆域解析运算符::

```
class complex{public:double getreal();};
double complex::getreal(){return real;};
```

◆在类定义中定义

```
class complex{
private:
        double real;  //实部
public:
        double getreal(){return real;};
};
```

对象

- 对象的概念
  - 对象是类的实例，正如变量是数据类型的实例
  - 可以作为函数参数，也可以作为函数返回值
  - 编译器不为类分配存储空间，只为对象分配
  - 2种创建方式：静态和动态（第7章）

对象的使用

- 对象的创建
- 对象数组的创建
- 对象指针的创建
- 对象成员的操作

对象的创建

◆定义格式

类名 对象名;

◆例子

complex x;

对象数组的创建

◆定义格式

类名 对象名[数组个数];

◆例子

complex x[10];

对象指针的创建

- 定义格式

  类名 *对象指针名;

  complex *py;

- 注意创建对象指针时，并未对该指针指向的地址分配存储空间
- 对象指针的初始化
  - 指向已定义的其他同类对象

```
complex x,*py;
py=&x;
```

  - 动态分配存储空间（第7章）

对象成员的操作

- 只能对类的公有成员操作
- 对象变量：操作符为.
- 对象指针：操作符为->

对象成员的操作

```
class complex{
    Private: //私有成员变量
        double real;
    Public: //公有成员函数
        double getreal(){return real;}
        void    setreal(double r){real=r;}
};
complex x, *py=&x;
double r;
```

只能在类内部（成员函数中）访问私有成员

不能在类外部（通过类对象）访问私有成员

以下哪些能正确执行?

A) x.setreal(1.0);

B) r=py->getreal();

C) py->real=3.0;

D) r=x.real;

类与对象应用实例【例4.1】

◆ 类定义

```
class CGoods{  //商品类
private:  //即使无private作为访问限定符，也是默认私有
        char Name[21];     //商品名称
        int    Amount;     //商品数量
        float Price;       //商品单价
        float Total_value; //商品总价
public: //成员函数大多定义为公有，作为访问私有成员接口
        void   RegisterGoods(char[],int,float) ; //输入数据
        void   CountTotal(void) ;                //计算商品总值
        void   GetName(char[]) ;                 //读取商品名
        int    GetAmount(void) ;                 //读取商品数量
        float  GetPrice(void) ;                  //读取商品单价
        float  GetTotal_value(void) ;            //读取商品总价
};
```

类与对象应用实例【例4.1】

◆ 成员函数定义（类外）

```
void CGoods::RegisterGoods(char name[] , int amount , float price){
        strcpy(Name , name); //字符串复制函数
        Amount=amount;
        Price=price ;
}

void CGoods::CountTotal(void){
        Total_value = Price*Amount;
}

void CGoods::GetName(char name[]){
        strcpy(name , Name);
}
```

函数参数

成员变量

类与对象应用实例【例4.1】

◆ 成员函数定义（类外）

```
int  CGoods::GetAmount(void){
        return(Amount) ;
}

float CGoods::GetPrice(void){
        return(Price) ;
}

float CGoods::GetTotal_value(void){
        return(Total_value) ;
}
```

函数参数

成员变量

类与对象应用实例【例4.1】

◆ 主函数

```
#include<iostream>
#include<iomanip>
#include<cstring>
using namespace std;
int main( ){
        CGoods car ;
        char      str[21] ;
        int       number ;
        float     pr ;
        cout<<"请输入汽车型号：" ;
        cin.getline(str , 20) ;              //输入串长必须小于20
        cout<<"请依次输入汽车数量与单价：" ;
        cin>>number>>pr ;
```

类与对象应用实例【例4.1】

◆ 主函数

```
    car.RegisterGoods(str , number , pr) ;
    car.CountTotal() ;
    str[0]='\0' ;                    //字符串str清0
    car.GetName(str) ;               //给str赋值car.Name
    cout<<setw(20)<<str<<setw(5)<<car.GetAmount() ;
    cout<<setw(10)<<car.GetPrice()<<setw(20)
    <<car.GetTotal_value()<<endl ;
    return 0;
}
```

能不能直接访问

car.Total_value

car.Price

car.Amount

特殊的成员函数

- 构造函数(4.3)
  - 复制构造函数(4.4)
- 析构函数(4.3)

由对象的初始化说起

- 回顾结构变量的初始化
  - 结构类型名 结构变量名={初始化列表};
  - complex x={1.0, 2.0};

- 对象
  - 初始化：构造函数
  - 通过已有的对象初始化：复制构造函数
  - 注销：析构函数

  - 都是（都必须被定义为）公有函数

构造函数的概念

- 特殊的公有函数成员，用于初始化对象
  - 函数名：与类名相同
  - 返回类型：无（不是void类型，是没有返回类型）
  - 参数：可以没有，也可以定义多个参数
- 对象创建时自动调用，且仅在此时调用一次
- 支持重载多个构造函数，参数的个数和类型不同
- 默认构造函数
  - 如果程序员未定义，则系统自动生成一个默认构造函数，该默认构造函数无参数
  - 一旦程序员定义了（不管有参数无参数），则系统不再生成默认构造函数

构造函数的定义

- 在类定义中定义
- 在类定义外定义

构造函数的定义

◆ 在类定义中定义（且声明）

```
class complex{
private:
    double real, image;
public:
    complex(){real=0; image=0;};  //无参数

    complex(doube r)                    //1个参数
    {real=r; image=0;}; //也可改为参数作为虚部初值
                        //但只能实现一种
    complex(double r, double i)    //2个参数
    {real=r; image=i;};
};
```

构造函数的定义

◆ 在类定义外定义（在类定义中仅声明）

```
class complex{
private:
    double real, image;
public:                                          //函数声明
    complex();                                   //无参数
    complex(double r);                           //1个参数
    complex(double r, double i);                 //2个参数
};
complex::complex(){real=0; image=0;};

complex::complex(double r)
{real=r; image=0;};

complex::complex(double r, double i)
{real=r; image=i;};
```

构造函数的使用

◆ 对象创建时自动调用

```
class complex{
private:
    double real, image;
public:                                          //函数声明
    complex();                                   //无参数
    complex(double r);                           //1个参数
    complex(double r, double i);                 //2个参数
};
```

这时有没有默认构造函数? 没有

如果没有complex()呢? 还是没有

此时A)是否仍然正确? 不正确

以下调用方式能否创建类对象?

A) complex c1; 

B) complex c2(1.0); 

C) complex c3(1.0, 2.0); 

D) complex c4(); 

//定义了函数c4，无参数，返回值类型complex

构造函数应用实例【例4.1_1】

```
class CGoods{  //商品类
private:  //即使无private作为访问限定符，也是默认私有
        char Name[21];    //商品名称
        int    Amount;       //商品数量
        float Price;          //商品单价
        float Total_value;  //商品总价
public: //构造函数必须定义为公有
        CGoods();
        CGoods(char [], int, float);
        CGoods(char [], float);
};
```

构造函数应用实例【例4.1_1】

```
CGoods::CGoods(){
        Name[0]='\0' ; Price=0.0 ;
        Amount=0 ; Total_value=0.0 ;
}

CGoods::CGoods(char name[] , int amount , float price){
        strcpy(Name,name) ;       Price=price ;
        Amount=amount ;Total_value=price*amount ;
}

CGoods::CGoods(char name[] , float price){
        strcpy(Name,name) ;       Price=price ;
        Amount=0 ; Total_value=0.0 ;
}
```

析构函数的概念

- 特殊的公有函数成员，用于对象注销，释放资源
  - 函数名：类名前加字符~
  - 参数：无
  - 返回值：无
- 有且仅有一个析构函数（与构造函数不同！）
- 对象注销时，系统自动调用析构函数
- 默认析构函数
  - 如果程序员未定义析构函数，则系统自动生成一个

析构函数的定义

◆ 在类定义中定义

```
class complex{
public:
    ~complex(){};
};
```

◆ 在类的定义外定义（在类定义中声明）

```
class complex{
public:
    ~complex(); //声明
};
complex::~complex(){};
```

析构函数应用实例【例4.2】

◆ 类定义(rec.h)

```
class Rectangle {
        int left, top ;
        int right, bottom;
public:
        Rectangle(int l=0, int t=0, int r=0, int b=0);   //构造函数，带默认参数，默认值为全0
        ~Rectangle(){};  //析构函数，在此函数体为空
        void Assign(int l, int t, int r, int b);
        void SetLeft(int t){ left = t;}  // 以下4个函数皆为内联成员函数
        void SetRight( int t ){ right = t;}
        void SetTop( int t ){ top = t;}
        void SetBottom( int t ){ bottom = t;}
        void Show();
};
```

析构函数应用实例【例4.2】

◆ 类成员函数定义(rec.cpp)

```
#include <iostream>
#include "rect.h"
using namespace std;
Rectangle::Rectangle(int l , int t, int r, int b) {
        left = l; top = t;
        right = r; bottom = b;
} // 构造函数，带默认参数，默认值为全0，在声明中指定
void Rectangle::Assign(int l, int t, int r, int b){
        left = l; top = t;
        right = r; bottom = b;
}
void Rectangle::Show(){
        cout<<"left-top point is ("<<left<<","<<top<<")"<<'\n';
        cout<<"right-bottom point is ("<<right<<","<<bottom<<
")" <<'\n';
}
```

析构函数应用实例【例4.2】

◆ 主函数(Exp10_1.cpp)

```
#include <iostream>
#include "rect.h"
using namespace std;
int main(){
        Rectangle rect;
        cout<<"由默认的构造函数生成的rect： "<<endl;
        rect.Show();
        rect.Assign(100,200,300,400);
        cout<<"由赋值函数处理后的rect： "<<endl;
        rect.Show();
        Rectangle rect1(0,0,200,200);
        cout<<"由构造函数生成的rect1： "<<endl;
        rect1.Show();
        return 0;
}
```

复制构造函数的概念

- 一种特殊的构造函数，通过一个类对象初始化另一个类对象
- 有且只有一个参数，参数类型为类对象的引用
- 例子

```
class complex{
private:
    double real, image;
public:
    complex(complex &c)
    {real=c.real; image=c.image};
};
```

- 默认复制构造函数
  - 如果程序员未定义，系统会自动生成一个默认复制构造函数
  - 功能：将一个类对象的所有数据成员的值复制到另一个类对象

复制构造函数的使用

以下情况下，复制构造函数被自动调用

- 一个对象通过另一个对象初始化时

```
complex x;
complex y(x);
complex z=x;   //也是调用复制构造函数
```

复制构造函数的使用

以下2种情况下，复制构造函数也被自动调用

- 一个对象以值传递的方式传入函数时

```
void f1(complex c);     //值传递，调用
void f2(complex &c);  //引用，不调用
```

- 一个对象以值传递的方式从函数返回时

```
complex f3();     //值传递，调用
complex &f4();  //引用，不调用
```

复制给谁呢?
复制给一个无名的临时对象
是否希望出现自动调用的复制?
不希望，不必要的复制会导致额外的运行开销。

构造函数与析构函数调用

类对象	调用构造函数	调用析构函数
全局定义	进入主函数之前	退出主函数时
局部定义	类对象定义时（每次）	离开局部域（定义该类对象的函数）时
静态局部定义	类对象定义时（首次）	退出主函数时

对照【例4.7】，理解构造函数和析构函数调用过程

演示对象创建和撤消的对应关系【例4.7】

```
#include <iostream>
using namespace std;
class complex{
private:
   double real, image;
public:
      complex(){real=0.0; image=0.0; cout<<"Initializing  0   0"<<endl;}
      complex(double r,double i=0.0){real=r; image=i;
            cout<<"Initializing "<<r<<'\t'<<i<<endl;}
      complex(complex &com); //复制构造函数
      ~complex(){cout <<"Destructor"<<endl;}  //析构函数
      void assign(complex com5){real=com5.real; image=com5.image;}
      void print(){cout<<real<<'+'<<image<<'i'<<endl;}
};
inline complex::complex(complex &com){real=com.real; image=com.image;
      cout<<"Copy"<<com.real<<'\t'<<com.image<<endl;}
complex global;
complex fun(complex com4){
      cout<<"Entering function"<<endl; global.assign(com4);
      cout<<"Exiting function"<<endl;
      return global;
}
int main(){
      cout <<"Entering main"<< endl;
      complex com1, com2(5.6, 7.5);
      complex com3=com1;
      com3.print();   global.print();
      com1=fun(com2);
      com1.print();
      cout<<"Exiting main"<<endl;
      return 0;
}
```

Initializing 0 0	默认构造函数建立global
Entering main	cout
Initializing 0 0	默认构造函数建立com1
Initializing 5.6 7.5	带参数构造函数建立com2
Copy 0 0	复制构造函数建立com3
0+0i	打印com3
0+0i	打印global
Copy 5.6 7.5	复制构造函数建立临时对象com4
Entering function	cout
Copy 5.6 7.5	复制构造函数建立临时对象com5
Destructor	退出assign()前析构函数撤销com5
Exiting function	退出全局函数fun()
Copy 5.6 7.5	复制构造函数建立无名临时对象
Destructor	退出fun()前析构函数撤销com4
Destructor	无名临时对象赋值给com1后撤销
5.6+7.5i	打印com1
Exit main	cout
Destructor	退出主函数前析构函数撤销com3
Destructor	退出主函数前析构函数撤销com2
Destructor	退出主函数前析构函数撤销com1
Destructor	退出主函数前析构函数撤销global

成员对象

- 以其他类的对象作为类成员，称为成员对象
- 使用成员对象的技术称为类聚合

```
class Date;  //声明Date类
class Person{
Private:
    Date  Birthday;
//Date类对象Birthday是Person类的成员
//称Birthday是成员对象
};
```

成员对象与构造函数

- 成员对象的初始化：通过A(Person)类构造函数完成成员对象b(Birthday)的初始化

A　　A　　b

类名::构造函数名(参数总表):对象成员1(参数名表1),对象成员2(参数名表2),……对象成员n(参数名表n){……}

- 构造函数执行过程
  - 首先，依次调用各对象成员的构造函数（按哪个次序？）
    - 对象成员在类定义中的顺序
    - ~~而不是，对象成员在构造函数初始化列表中的顺序~~
  - 然后，调用该类自己的构造函数

成员对象与析构函数

- 析构函数：与不含成员对象的类一样
- 析构函数执行过程（与构造函数相反）
  - 首先，调用该类自己的析构函数
  - 然后，按与构造函数相反的次序，调用各对象成员的构造函数
    - 按对象成员在类定义中相反的次序

对象成员应用实例【例4.6】

```
#include<iostream>
#include<cstring>
using namespace std;
class studentID{
        long value;
public:
        studentID(long id=0){
                value=id;
                cout<<"赋给学生的学号："<<value<<endl;
        }
        ~studentID(){
                cout<<"删除学号："<<value<<endl;
        }
};
```

对象成员应用实例【例4.6】

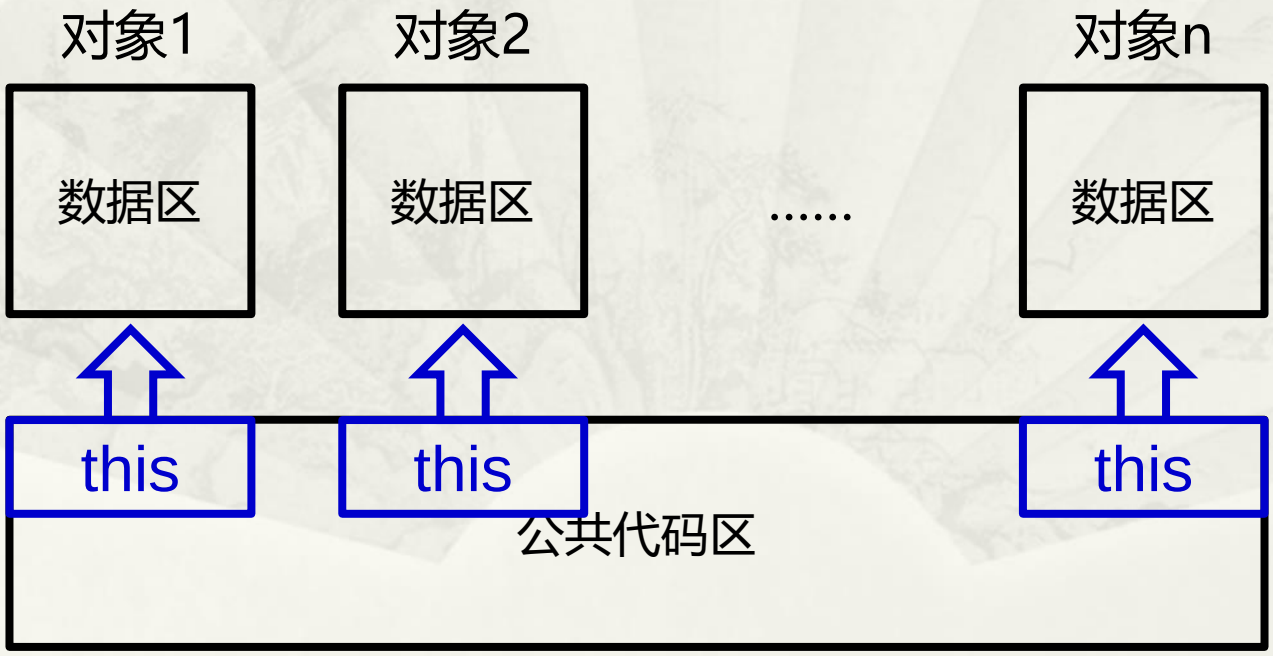
```
class student{
private:
        studentID id;
        char name[20];
public:
        student (char sname[]="no name",long sid=0):id(sid){
                cout<<"学生名："<<sname<<endl;
                strcpy(name,sname);
        }
        ~student(){cout<<"删除学生名："<<name<<endl;}
};

int main(){
        student ss("朱明",82020132);
        return 0;
}
```

对象成员应用实例【例4.6】

this指针(5.4)

- 同一个类的不同类对象的成员函数共用同一份代码
- 不同类对象的成员函数如何区分各自的数据?



this指针(5.4)

- this指针是由系统自动产生的指针变量
  - 变量名： this
  - 变量类型：类指针
  - 指向的地址：类对象自身
  - 作用范围：只在类的成员函数中有效

```
*this            //指向类对象自身
this->xxx    //访问类成员，合法但不常用
(*this).xxx  //访问类成员，合法但不常用

complex *this;   //错误，不能定义
&this ;                 //错误，不能取地址
this=xxx;               //错误，不能赋值
```

this指针(5.4)

- this指针是由系统自动产生的
  - 本质上是一个隐藏的函数参数
  - 示例1：调用成员函数

```
void complex::setreal(double r){real=r;}
//相当于以下代码
void complex::setreal(complex *this, double r)
{this->real=r;}

double re;
complex c1;

c1.setreal(re);
//相当于以下代码
setreal(&c1, re);
```

this指针(5.4)

- this指针是由系统自动产生的
  - 本质上是一个隐藏的函数参数
  - 示例2：访问成员数据

```
void CGoods::RegisterGoods(char*name,int
amount,float price){
        strcpy(this->Name,name);
        this->Amount=amount;
        this->Price=price;
}
```

运算符重载(4.5)

- 内置数据类型，系统支持的运算

```
int a, b, c;
a=b;     //赋值运算符=
a=b+c; //加法运算符+
++a;     //递增运算符++
```

- 程序员自定义类，系统未支持运算，需要程序员通过运算符重载实现

```
complex a, b, c;
a=b;
a=b+c;
++a;
```

- 运算符重载函数和构造函数（复制构造函数）、析构函数一样，是一种特殊的类的函数成员

运算符重载函数的声明与定义

- 声明格式

返回值类型 operator 重载的运算符（参数表）；

- 定义格式（在类外定义）

返回值类型 类名::operator 重载的运算符(参数表){};

- 关键字operator标识
- 重载的运算符为合法的运算符
  - 单目运算符：++,--,!
  - 双目运算符：+-*/%=><
  - 不支持重载的运算符
    - 三目运算符?:
    - 类成员操作符. ->
    - 域解析运算符::
    - 字长运算符sizeof

运算符重载函数的定义

◆ 单目运算符，无参数

```
//!a，相当于a.operator!()
complex complex::operator!();

//前置，++a，相当于a.operator++()
complex  complex::operator++();

//后置，a++，相当于a.operator++(0)
complex complex::operator++(int);
```

运算符重载函数的定义

◆ 单目运算符，无参数

```
class zoo{
    int n_tiger, n_lion;
public:
    zoo  operator++() //前置，++a
            ++n_tiger;
            ++n_lion;
            return *this;
    }
    zoo  operator++(int) //后置，a++
            zoo tmp(*this);
            ++n_tiger;
            ++n_lion;
            return tmp;
    }

}
```

运算符重载函数的调用

- 双目运算符，1个参数。左操作数调用函数，右操作数为参数
  - a+b，a调用函数，b是实参，相当于a.operator+(b)，返回值不是a，而是a+b的和
  - a=b，a调用函数，b是实参，相当于a.operator=(b)，返回值不是a，可以是与a相等的类对象（也可以是a的引用）

运算符重载函数应用实例——复数运算【例4.8】

```
class complex{
private:
    double real, image;
public:
    complex(double r=0.0, double i=0.0) {real=r; image=i;};
    complex(complex &c)       {real=c.real; image=c.image;};
    void Print(){
        cout<<"Real="<<Real<<'\t'<<"Image="<<Image<<'\n';
    }
    complex operator+(complex); //复数相加
    complex operator+(double);   //复数+实数
    complex operator=(complex); //复数赋值
    complex operator+=(complex);
    double  abs(void);
    complex operator*(complex);  //复数相乘
    complex operator/(complex);  //复数相除
};
```

运算符重载函数应用实例——复数运算【例4.8】

```
//实现1：变量sum存储复数的和
complex complex::operator+(complex c){
        complex sum(real+c.real, image+c.image);
        return sum;
};

//实现2：不定义变量sum，实际上仍创建了无名对象
complex complex::operator+(complex c)
{return complex(real+c.real, image+c.image);};

//实现3：避免参数值传递时调用复制构造函数➔改为引用
complex complex::operator+(const complex &c)
{return complex(real+c.real, image+c.image);}
```

返回值是否调用了复制构造函数? 有

返回值传递方式能否改为引用? 不能

加法运算符重载函数

	值传递 （不带&）	引用（带&）
	◆会调用复制构造函数	◆不会调用复制构造函数
参数	◆不会修改实参，形参的修改与实参无关	◆会修改实参，形参的修改即为对实参本身的修改
返回值	◆常将函数中的局部变量作为返回值	◆绝不能将函数中的局部变量作为返回值

运算符重载函数应用实例——复数运算【例4.8】

```
//复数+实数
complex complex::operator+(double d){
        return complex(real+d, image);
};
```

运算符重载函数应用实例——复数运算【例4.8】

```
complex complex::operator+=(complex c){
        complex temp;
        temp.Real=Real+c.Real;
        temp.Image=Image+c.Image
        Real=temp.Real;
        Image=temp.Image;
        return temp;
}

complex complex::operator=(complex c){
        complex temp;
        temp.Real=c.Real;
        temp.Image=c.Image;
        Real=temp.Real;
        Image=temp.Image;
        return temp;
}
```

更新temp对象的Real和Image
目的：更新返回值

更新自身的Real和Image
目的：更新自身

为什么要多出个temp?
为什么不干脆返回自身?

运算符重载函数应用实例——复数运算【例4.8】

```
complex complex::operator+=(complex c){
        //complex temp;
        //temp.Real=Real+c.Real;
        //temp.Image=Image+c.Image;
        //Real=temp.Real;
        //Image=temp.Image;
        //return temp;
}
```

```
Real=Real+c.Real;
Image=Image+c.Image;
return *this;
```

```
complex complex::operator=(complex c){
        //complex temp;
        //temp.Real=c.Real;
        //temp.Image=c.Image;
        //Real=temp.Real;
        //Image=temp.Image;
        //return temp;
}
```

```
Real=c.Real;
Image=c.Image;
return *this;
```

运算符重载函数应用实例——复数运算【例4.8】

```
complex complex::operator=(complex c){
        Real=c.Real; Image=c.Image;
        return *this;
}
complex complex::operator=(complex& c){   //参数引用传递，避免调
        Real=c.Real; Image=c.Image;          用复制构造函数
        return *this;
}
complex& complex::operator=(complex c){   //返回值引用传递，避免
        Real=c.Real; Image=c.Image;          调用复制构造函数
        return *this;
}
complex& complex::operator=(complex& c){//参数和返回值都引用传递，
        Real=c.Real; Image=c.Image;          避免调用复制构造函数
        return *this;
}
```

赋值运算符重载函数vs.复制构造函数

```
complex a;
complex b=a;
//调用哪个函数?
```

复制构造函数

```
complex a, b;
b=a;
//这又调用哪个函数?
```

赋值操作符重载函数

运算符重载函数应用实例——复数运算【例4.8】

```
double complex::abs(void){
        return sqrt(Real*Real+Image*Image);
}

complex complex::operator*(complex c){
        return complex(Real*c.Real-Image*c.Image ,
        Real*c.Image+c.Real*Image);
}

complex complex::operator/(complex c){
        double d=c.Real*c.Real+c.Image*c.Image ;
        return complex((Real*c.Real+Image*c.Image)/d ,
        (Image*c.Real-Real*c.Image)/d) ;
}
```

运算符重载函数应用实例——复数运算【例4.8】

```
int main(void){
        complex c1(1.0,1.0) , c2(2.0,2.0) , c3(4.0,4.0) , c;
        double d=0.5 ;
        c1.Print();
        c=c2+c3; c.Print();
        c+=c1; c.Print();
        c=c+d; c.Print();                    //复数加实数
        c=c3*c2; c.Print();
        c=c3/c1; c.Print();
        cout<<"c3的模为： "<<c3.abs()<<endl;
        c=c3=c2=c1; c.Print();               //连续赋值
        c+=c3+=c2+=c1; c.Print();            //连续加赋值
        return 0;
}
```

操作1：对象c2执行c2+c3的操作
操作2：对象c执行c=(操作1返回值)的操作

运算符重载函数应用实例——复数运算【例4.8】

```
int main(void){
        complex c1(1.0,1.0) , c2(2.0,2.0) , c3(4.0,4.0) , c;
        double d=0.5 ;
        c1.Print();
        c=c2+c3; c.Print();
        c+=c1; c.Print();
        c=c+d; c.Print();                    //复数加实数
        c=c3*c2; c.Print();
        c=c3/c1; c.Print();
        cout<<"c3的模为： "<<c3.abs()<<endl;
        c=c3=c2=c1; c.Print();               //连续赋值
        c+=c3+=c2+=c1; c.Print();            //连续加赋值
        return 0;
}
```

操作1：对象c2执行c2=c1的操作
操作2：对象c3执行c3=(操作1返回值)的操作
操作3：对象c执行c=(操作2返回值)的操作

运算符重载函数应用实例——复数运算【例4.8】

```
int main(void){
        complex c1(1.0,1.0) , c2(2.0,2.0) , c3(4.0,4.0) , c;
        double d=0.5 ;
        c1.Print();
        c=c2+c3; c.Print();
        c+=c1; c.Print();
        c=c+d; c.Print();                    //复数加实数
        c=c3*c2; c.Print();
        c=c3/c1; c.Print();
        cout<<"c3的模为： "<<c3.abs()<<endl;
        c=c3=c2=c1; c.Print();               //连续赋值
        c+=c3+=c2+=c1; c.Print();            //连续加赋值
        return 0;
}
```

操作1：对象c2执行c2+=c1的操作
操作2：对象c3执行c3+=(操作1返回值)的操作
操作3：对象c执行c+=(操作2返回值)的操作

运算符重载函数应用实例——复数运算【例4.8】

```
int main(void){
        complex c1(1.0,1.0) , c2(2.0,2.0) , c3(4.0,4.0) , c;
        double d=0.5 ;
```

c1.Print();	Real=1.0 Image=1.0
c=c2+c3; c.Print();	Real=6.0 Image=6.0
c+=c1; c.Print();	Real=7.0 Image=7.0
c=c+d; c.Print(); //复数加实数	Real=7.5 Image=7.0
c=c3*c2; c.Print();	Real=0.0 Image=16.0
c=c3/c1; c.Print();	Real=4.0 Image=0.0
cout<<"c3的模为： "<<c3.abs()<<endl;	c3的模为：5.65685
c=c3=c2=c1; c.Print(); //连续赋值	Real=1.0 Image=1.0
c+=c3+=c2+=c1; c.Print(); //连续加赋值	Real=4.0 Image=4.0

```
        return 0;
}
```

操作符重载函数的不足

- 复数+复数

```
//c1+c2
complex operator+(const complex&);
//相当于c1.operator+(c2)
```

- 复数+实数

```
//c1+d
complex operator+(double);
//相当于c1.operator+(d)
```

- 实数+复数?

```
//d+c1
//相当于实现d.operator+(c1)
//double是内置数据类型，无法为其重载+
```

友元函数(4.6)

- 函数声明时，用关键字friend标识（函数定义时不用再标识）
- 友元函数在类内声明，但不是类的成员函数
- 友元函数不受访问权限关键字限制，可以在类外访问类的私有成员➔对封装性的削弱

友元函数的声明、定义和使用

◆ 实数+复数

```
class complex{
    …
    //声明
    friend complex operator+(double, const complex&);
};
//定义时不需再加friend，也不是成员函数
complex operator+(double d, const complex& c)
{
    return complex(d+c.real, c.image);
}
int main(){
    …
    //使用
    c=d+c1;  //相当于c=operator+(d, c1)
}
```

友元函数与操作符重载函数的差别

- 可用友元函数实现，但无法用操作符重载函数实现
  - 双目运算符，第一个操作数为内置数据类型。如：实数+复数
- 可用操作符重载函数实现，但无法用友元函数实现
  - = [] ()

友元函数应用实例——复数运算【例4.8_1】

```
class complex{
private:
    double real, image;
public:
    complex(double r=0.0, double i=0.0) {real=r; image=i;};
    complex(complex &c)       {real=c.real; image=c.image;};
    void Print(){
        cout<<"Real="<<Real<<'\t'<<"Image="<<Image<<'\n';
    }
    friend complex operator+(const complex &,const complex &);
    friend complex &operator +=(complex &,const complex &);
    friend double abs(complex &);
    friend complex operator*(const complex &,const complex &);
    friend complex operator/(const complex &,const complex &);
};
```

友元函数应用实例——复数运算【例4.8_1】

```
complex operator+(const complex & c1,const complex & c2){
        return complex(c1.Real+c2.Real,c1.Image+c2.Image);
}

complex &operator +=(complex &c1,const complex &c2){
        c1.Real=c1.Real+c2.Real;
        c1.Image=c1.Image+c2.Image;
        return c1;   //参数引用传递，可作为返回值引用传递
}

double abs(complex &c){
        return sqrt(c.Real*c.Real+c.Image*c.Image);
}
```

友元函数应用实例——复数运算【例4.8_1】

```
complex operator*(const complex & c1,const complex & c2){
        return complex(c1.Real*c2.Real-c1.Image*c2.Image ,
c1.Real*c2.Image+c2.Real*c1.Image);
}

complex operator/(const complex & c1,const complex & c2){
        double d=c2.Real*c2.Real+c2.Image*c2.Image ;
        return complex((c1.Real*c2.Real+c1.Image*c2.Image)/d ,
(c1.Image*c2.Real-c1.Real*c2.Image)/d) ;
}
```

友元函数应用实例——复数运算【例4.8_1】

```
int main(void){
        complex c1(1.0,1.0) , c2(2.0,2.0) , c3(4.0,4.0) , c;
        double d=0.5 ;
        c1.Print();
        c=c2+c3; c.Print();
        c+=c1; c.Print();
        c=c+d; c.Print();                    //复数加实数
        c=d+c; c.Print();                    //实数加复数
        c=c3*c2; c.Print();
        c=c3/c1; c.Print();
        cout<<"c3的模为： "<<c3.abs()<<endl;
        c=c3=c2=c1; c.Print();               //连续赋值
        c+=c3+=c2+=c1; c.Print();            //连续加赋值
        return 0;
}
```

可以用运算符重载函数实现吗？不行

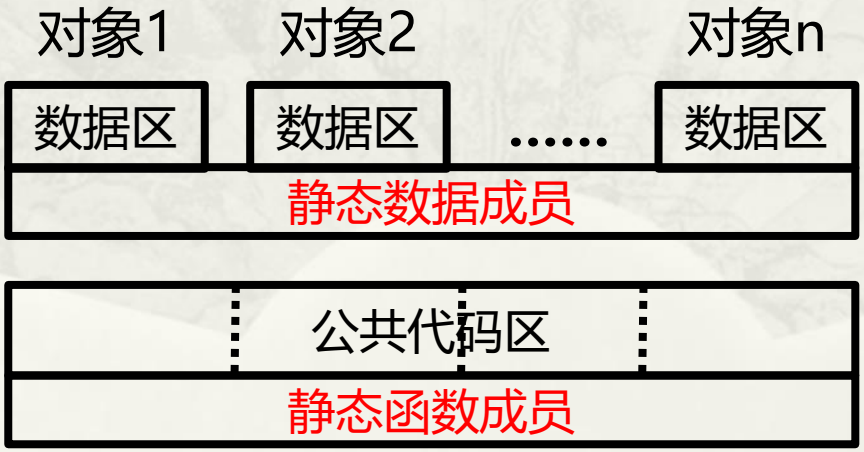
友元类

- 友元类用关键字friend标识
- 友元类中的任意成员函数都是另一个类的友元函数，可以在类外访问另一个类的私有成员
  ➔对封装性的削弱
- 例子

```
class Person{
        …
        friend class Pet;  //Pet类为Person类的友元类
};
```

静态成员(4.7)

- 静态成员的概念
  - 所有同类对象共享的类成员
- 静态成员包括
  - 静态数据成员：相当于同类对象间的全局变量
  - 静态函数成员
- 静态成员的定义
  - 在类定义中用关键字static修饰



静态数据成员的定义和初始化

- 定义
  - 在类定义中用关键字static修饰
- 初始化
  - 在类定义外对静态数据成员做定义性说明
  - 必须做一次且只能做一次

静态数据成员应用实例【例4.9】

```
#include <iostream>
using namespace std;
class  Ctest{
private:
    static int count;
public:
    Ctest()   {++count; cout<<"对象数量="<<count<<'\n';  }
    ~Ctest() {--count;   cout<<"对象数量="<<count<<'\n';  }
};

//对静态数据定义性说明
int Ctest::count=0;

int main(void){
    Ctest a[3];
    return 0;
}
```

```
对象数量=1    //a[0]构造函数产生
对象数量=2    //a[1]构造函数产生
对象数量=3    //a[2]构造函数产生
对象数量=2    //a[2]析构函数产生
对象数量=1    //a[1]析构函数产生
对象数量=0    //a[0]析构函数产生
```

本章小结(1)

- 面向对象的程序设计（4.2, 4.10节）
  - 封装性；继承性；多态性
- 结构的基本概念（4.8节）
  - 数据的组合
  - 定义：关键字struct
  - 访问成员：操作符（变量. 指针->）
- 类与对象的基本概念（4.1 节）
  - 数据（属性）和函数（操作）的组合→封装性
  - 定义：关键字class
  - 类成员：数据成员；函数成员
  - 成员访问限定符：公有(public)；私有(private)；保护(protected)
  - 对象是类的实例
- this指针（5.4节）
  - 系统自动产生的指向类对象自身的指针
  - 只在类的成员函数中有效

本章小结(2)

- 构造函数、复制构造函数与析构函数（4.3,4.4节）
  - 构造函数：初始化对象。函数名与类名相同；无返回值；可重载
    complex();
    complex(double);
    complex(double, double);
  - 复制构造函数：复制对象；对象参数值传递；对象返回值值传递。参数为类引用
    complex(complex&);
  - 析构函数：注销对象。参数名为~类名；无返回值；无参数
    ~complex();
- 运算符重载函数（4.5节）
  - 使得自定义类支持运算符操作
  - 定义：关键字operator
- 友元（4.6节）
  - 友元函数和友元类可以访问私有成员（封装性的削弱）
  - 定义：关键字friend
  - 优势：拓展了运算符重载的范围。实数+复数
- 静态成员（4.7节）
  - 同一个类的所有对象共享（封装性的削弱）
  - 定义关键字static

End