

C++高级语言程序设计

王晨宇

北京邮电大学网络空间安全学院

第5章 类和对象

5.1 面向对象的思想

5.2 面向对象程序设计的基本特点

5.3 面向对象软件开发的基本过程

5.4 类和对象

5.5 构造函数与析构函数

5.6 友元

*5.7 静态成员

*5.8 const对象和成员函数

*5.9 应用实例

5.1 面向对象的思想

- 什么是面向过程？什么是面向对象？
 - 两种不同的思想、方法。
- 面向过程的程序设计
 - 用传统程序设计语言进行程序设计时，必须详细地描述解题的过程。程序设计工作主要围绕设计解题过程来进行，这种传统的程序设计方法称为面向过程的程序设计。

面向过程程序设计

- 特点

程序——处理数据的一系列过程;

数据与过程分离;

程序 = 数据结构 + 算法

- 缺点

- 重用性差

- 维护困难

面向对象程序设计

- 面向对象程序设计：将面向对象方法用于程序设计。
- 出发点与基本原则：
 - 模拟人类习惯思维方式，使开发软件的方法尽可能接近人类认识世界解决问题的方法。
- 对象作为模块,对象是对客观事物的自然的、直接的抽象和模拟，包含了数据及对数据的操作。

5.2 面向对象程序设计的基本特点

- 封装性
 - 将描述对象的数据及处理这些数据的数据集中起来放在对象内部，对象成为独立模块。
- 继承性
 - 从已有类(称为基类)派生出新类。
- 多态性
 - 同一个名字代表不同、但相似的功能。

5.3 面向对象软件开发的基本过程

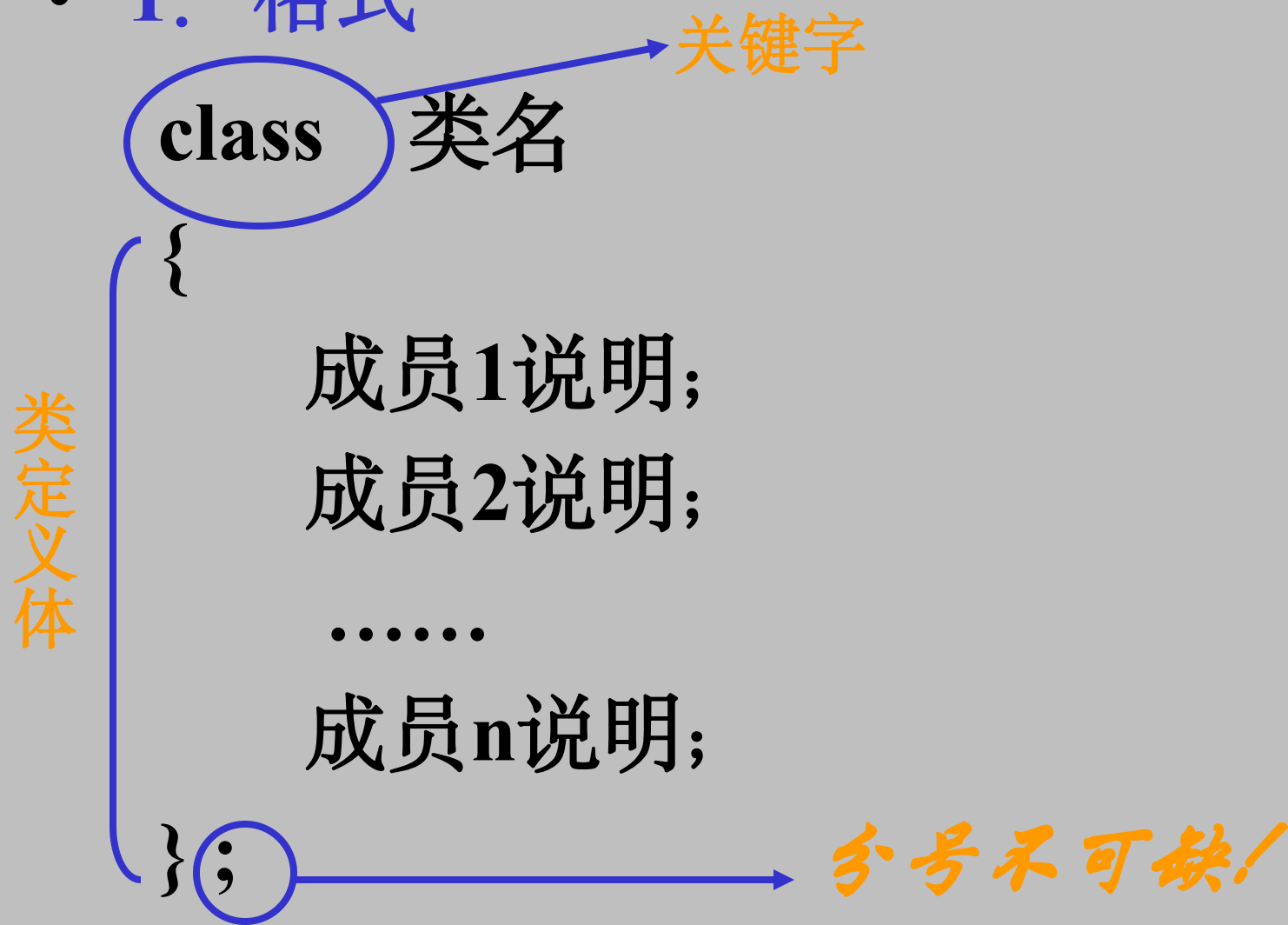
- 软件危机促进了软件工程的形成与发展。
- 软件工程：用系统工程学的原理和方法管理软件开发过程，开发过程分为分析、设计、编程、测试、维护等阶段。
- 面向对象的软件工程
 - 分析：明确系统必须做什么。
 - 设计：明确软件系统怎么做。
 - 实现：选用合适的面向对象编程语言，实现设计阶段描述的软件系统的各个类模块，并根据类的相互关系集成。
 - 测试：发现软件中的错误。
 - 维护：在软件交付用户使用期间，对软件所作的修改。

5.4 类和对象

- 在面向对象程序设计中，对象是构成程序的模块，即程序是由一组对象构成的，对象之间通过消息分工协作，共同完成程序的功能。
- 类是面向对象程序设计的核心，是对具有相同属性与行为的一组事物的抽象描述。利用类可以把数据和对数据所做的操作组合成一个整体，实现对数据的封装和隐藏。
- 类是用户自定义的数据类型，是创建对象的模型。

5.4.1 类的定义

- 1. 格式



- ## 2. 类的组成成员

(1)数据成员: 变量或对象。其类型为:

{ 基本类型: int、float、double、char、bool
复合类型: 数组、指针、引用、结构、枚举等

(2)成员函数

对数据成员进行操作。

■ 例:

```
class Circle{
```

```
    private:
```

```
        int radius;
```

```
    public:
```

```
        void setRadius(int r)
        { radius=r; }
```

```
        double area( )
        { return 3.14*radius*radius; }
```

```
};
```

数据成员

函数成员

1

函数成员

2

- 在定义一个类时，注意：
 - 类只是一种自定义数据类型，类中任何成员数据均不能使用关键字extern、auto或register指定其存储类型，也不能初始化。例如：

```
class Circle{  
    int radius=5;    //错误  
    extern float pi; //错误  
    .....          //省略其它成员  
};
```

- 成员函数可直接使用类中的任一成员。
- 类类型与结构体类型相似，结构体类型也可有函数成员，差别在于，类类型的缺省访问权限是private，结构体类型的缺省访问权限是public。

5.4.2 类的成员函数

- 成员函数必须在类体内给出原型说明，至于它的实现，可以放在类体内，也可以放在类体外。
 - 当成员函数所含代码较少时，一般直接在类中定义该成员函数；
 - 当成员函数中所含代码较多时，通常只在类中进行原型的说明，在类外对函数进行定义。

例:

```
class Person {  
    char name[12];  
    int age;  
    char sex[4];
```

```
public:
```

```
    void Print( ){  
        cout<<name<<','<<age<<','<<sex<<endl;  
    }
```

```
        .....//省略其它成员
```

```
};
```

在类中定义的
成员函数

例:

```
class Person {  
    char name[12];  
    int age;  
    char sex[4];
```

```
public:
```

```
    void Print ( );
```

```
    .....//省略其它成员
```

```
};
```

在类中声明成员函数的原型

在类外定义成员函数

```
void Person :: Print( )
```

```
{ cout<<name<<','<<age<<','<<sex<<endl; }
```

- 在类外定义成员函数时，函数名应该包含：

类名 + 作用域分辩符 (::) + 原函数名



指明该函数是哪个类的成员

5.4.3 对象

- 对象的本质

一个对象就是一个类的实例。一个对象就是一个具有某种类类型的变量。

- 对象的定义格式:

类名 对象名;

例:

Person p1,p2;

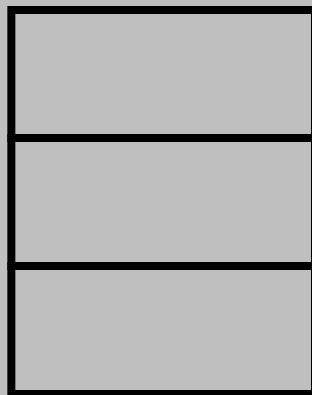
p1

p2

p1.name

p1.age

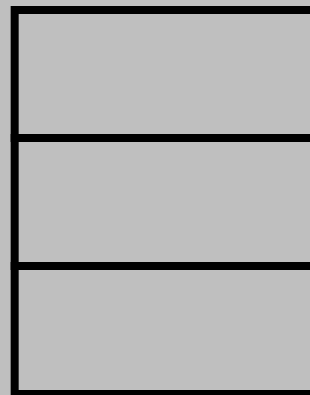
p1.sex



p2.name

p2.age

p2.sex



公共
代码
区

SetPerson()
GetName()
GetSex()
Print()

5.4.4 类成员的访问控制

访问权限控制	访问权限
private	只允许同类的成员函数访问
protected	允许同类及其派生类的成员函数访问
public	允许同一作用域的任何函数访问

- 在定义类时，指定其成员访问权限的原则：
 - 仅供该类的成员函数访问的成员应指定为私有的。
 - 若定义的成员在派生类中也需经常使用，则应指定其为保护的。
 - 若允许在类外使用成员时，应将其访问权限定义为公有的。

- 访问对象的成员

对象名.数据成员名

对象名.成员函数名 (参数表)

对象指针名->数据成员名

对象指针名->成员函数名 (参数表)

- 访问对象的成员时，要注意成员的访问权限。

例：

```
#include <iostream.h>
class Rectangle{
    private:
        float length,width;
    public:
        float Area( )
        {   return length*width;   }
        float Perimeter( )
        {   return 2*(length+width); }
};
```

以关键字class定义的类，成员的缺省访问属性为private。

```
void main( ) {
```

```
    Rectangle r;
```

```
    r.width=45;
```

```
    r.length=54.2;
```


```
    cout<<"the area of rectangle is"
```

```
        <<r.Area()<<endl;
```

```
    cout<<"the perimeter of rectangle is"
```

```
        <<r.Perimeter()<<endl;
```

```
}
```



非法访问
私有成员!

例:

```
#include<iostream.h>
```

```
class Rectangle{  
    float length,width;  
public:  
    void SetWidth(float newWidth)  
    { width=newWidth; }  
    void SetLength(float newLength)  
    { length=newLength; }  
    float Area( )  
    { return length*width; }  
    float Perimeter( )  
    { return 2*(length+width); }  
};
```

```
void main( ){  
    Rectangle r;           //定义对象  
  
    r.SetWidth(45);  
    r.SetLength(54.2);  
    cout<<"The area of rectangle is"  
        <<r.Area( )<<endl;  
    cout<<"The perimeter of rectangle is "  
        <<r. Perimeter( )<<endl;  
}
```


5.4.5 对象数组

- 若一个数组中每个元素都是同类型的对象，则称该数组为对象数组。
- 定义一维对象数组的格式为：
 <类名><数组名>[<整型常量表达式>];

例5.5 对象数组的使用。

```
#include <iostream.h>
```

```
class Circle{  
    int radius;  
public:  
    void setRadius(int r){ radius=r; }  
    int getRadius( ){ return radius; }  
    double area( ){ return 3.14*radius*radius;}  
};
```

类Circle的
定义

```
void main( )
```

```
{ Circle c[3];
```

定义对象数组

```
    c[0].setRadius(6); c[1].setRadius(2);
```

```
    c[2].setRadius(15);
```

```
    for(int i=0;i<3;i++)
```

```
        cout<<"radius:"<<c[i].getRadius( )
```

```
        <<" ,area:"<<c[i].area( )<<endl;
```

```
}
```

5.4.6 this指针

- 一个成员函数被调用时，系统自动向它传递一个隐含的指针，该指针是调用该成员函数的对象的指针，在成员函数的函数体中可直接用this使用该指针。
- 在成员函数的实现中，当访问该对象的某成员时，系统自动使用这个隐含的this指针。在定义成员函数的函数体时，通常省略this指针。
- 例：Circle类的成员函数

```
void setRadius(int r)
```

```
this->radius=r; }
```

this指针具有如下的缺省说明：

```
Circle *const this;
```

- 有时不得不显式使用this指针。
 - 数据成员名与函数成员的形参同名。 例:

```
class Name{
    char* name;
public:
    void SetName(char* name)
    { if(name)
        { this->name=new char[strlen(name)+1];
          strcpy(this->name,name);
        }else this->name=NULL;
        }
    .....
};
```

– 引用对象本身。 例:

```
class Name{
    char* name;
public:
    void CopyName(Name& n)
    { if(this==&n) return;//避免对象自我复制
      delete name;
      if(n.name)
      { name=new char[strlen(n.name)+1];
        strcpy(name,n.name);
      }else name=NULL;
    }
    .....
};
```

5.5 构造函数和析构函数

- 构造函数和析构函数：类的特殊成员。
- 构造函数：
 - 创建对象，并初始化对象的数据成员。
- 析构函数：
 - 撤销对象，收回它所占内存，及完成其它清理工作。

5.5.1 构造函数

- 构造函数的定义格式

```
ClassName(<形参表>)
```

```
{ ... } //函数体
```

①它与所在类同名。

②它不得有返回值，甚至连关键字void也不许有。

例:

```
class Person{
    char name[12];
    int age;
    char sex[4];
public:
    Person(const char*n,int a,const char*s){
        strcpy(name,n);
        age=a;
        strcpy(sex,s);
    }
    ..... //省略其它成员函数
};
```


例:

```
class Person{  
    char name[12];  
    int age;  
    char sex[4];
```

```
public:
```

```
    Person(const char* n,int a,const char* s);
```

```
    ... //其它成员函数说明
```

```
};
```

```
Person::Person(const char* n,int a,const char* s)
```

```
{    strcpy(name,n);
```

```
    age=a;
```

```
    strcpy(sex,s);
```

```
}
```




也可在类中声明原型，在类外定义。

- 构造函数的特殊性:

①构造函数在定义类对象时由系统自动调用。例:

```
void main( ){  
    Person p( “张三” ,20, “男” );  
    .....  
}
```



当遇此定义语句时，系统就调用构造函数:

```
Person(const char* n,int a,const char* s)
```

创建对象p，并用实参初始化它的数据成员。

②构造函数可以重载。 例:

```
class Point{  
    float x,y;  
public:  
    Point( ){x=0;y=0};           //(1)  
    Point(float a,float b){x=a;y=b;} //(2)  
    //...  
};
```

```
void main( ){  
    Point p1;           //匹配(1)  
    Point p2(2.0,4.2); //匹配(2)  
}
```

- 构造函数中初始化数据成员的方法

- 在函数体中用赋值等语句进行
 - 使用成员初始化列表

初始化列表

①格式:

在形参表和函数体之间，以：号开头，由多个以逗号分隔的初始化项构成。即

:数据成员名1(初值1),⋯,数据成员名n(初值n)

②举例:

```
Person::Person( ):name("zhang"),age(22)
{ strcpy(sex,"男"); }
```

③执行顺序: 带有初始化表的构造函数执行时，首先执行初始化表，然后再执行函数体。

数据成员的初始化

- 普通数据成员的初始化既可在函数体进行，也可在初始化表进行。例：

(方法1)

```
Point::Point(int x1,int y1)
{ x=x1; y=y1; }
```

或(方法2)

```
Point::Point(int x1,int y1):x(x1),y(y1){ }
```

- `const`数据成员、`const`对象成员和从基类继承的数据成员的初始化必须用初始化表。

缺省的构造函数

①C++规定，每个类至少有一个构造函数，否则就不能创建该类的对象。例：

```
class Rectangle{  
    float length,width;  
public:  
    float Area( );  
    float Perimeter( );  
};
```

```
..... //类的实现  
void main( ){  
    Rectangle rect;  
    .....  
}
```

能否创建Rectangle类的对象？

②类中若未定义构造函数，则C++编译系统自动提供一个缺省的构造函数。缺省的构造函数无参，函数体为空，仅创建对象，不作任何初始化工作。

```
class Rectangle{  
    float length,width;  
public:  
    float Area( );  
    float Perimeter( );  
};
```

因系统提供了默认的
构造函数:

```
Rectangle( ){ }
```

```
..... //类的实现  
void main( )  
{ Rectangle rect;  
    .....  
}
```


③只要类中定义了一个构造函数，C++编译器就不再提供缺省的构造函数，如还想要无参构造函数，则必须自己定义。例：

```
class Circle{  
    int radius;  
public:  
    Circle(int r)  
    { radius=r; } //(1)  
};
```

因该类有带一个参数的构造函数，故系统不再提供无参的缺省构造函数！

```
Circle c1;  
Circle c2(5);
```

✗ 没有无参构造函数
✓ 匹配(1)

对上例的修改方法1:

```
class Circle{
```

```
    int radius;
```

```
    public:
```

```
        Circle(int r){ radius=r; }//(1)
```

```
        Circle( ){ }          //(2)
```

```
};
```



添加无参构造函数!

```
Circle c1(3);
```

✓ 匹配(1)

```
Circle c2;
```

✓ 匹配(2)

对上例的修改方法2：

```
class Circle{
```

```
    int radius;
```

```
    public:
```

```
        Circle(int r=0){ radius=r; } //(1)
```

```
};
```

- 参数带默认值的构造函数!
- 所有参数都带默认值的构造函数也是缺省的构造函数。

```
Circle c1(3);
```

✓ 匹配(1)

```
Circle c2;
```

✓ 匹配(1)

使用参数带缺省值的构造函数时，应防二义性。如：

```
class Circle{  
    int radius;  
public:  
    Circle(){ }                //L1  
    Circle(int r=0){ radius=r; } //L2  
};
```

Circle c1; //L3 错误! 产生二义性

Circle c2(5); //匹配L2行的构造函数

当编译L3行的语句时，出错。因Circle类定义了两个缺省的构造函数，编译器不知道是调用L1行定义的Circle()还是调用L2行定义的Circle(int r=0)。

5.5.2 析构函数

- 作用：
 - 与构造函数相反，在对象被撤消时由系统自动调用，做清理工作。
- 特点：
 - 析构函数名必须与类名相同，并在其前面加上字符“~”，以便和构造函数名相区别。
 - 析构函数没有返回值，函数名前也不能用关键字void。
 - 析构函数没有参数，换言之，析构函数是唯一的，析构函数无法重载。

- 例5.9 调用析构函数。

```
#include<iostream.h>
class Point{
    int x,y;
public:
    Point(){
        x=y=0;cout<<"Default constructor called.\n";
    }
    Point(int a,int b){
        x=a;y=b;
        cout<<"Constructor with parameters called.\n";
    }
    ~Point( ){
        cout<<"Destructor called."<<x<<','<<y<<'\n';
    }
    void print( ){ cout<<x<<','<<y<<'\n'; }
};
```

为了观察调用构造函数和析构函数的过程，分别在构造函数和析构函数内添加了输出调用信息的语句。

```
void main( )  
{ Point p1(5,8),p2;  
  p1.print( );  
  p2.print( );  
  { Point p2(3,2);  
    p2.print();  
  }  
  cout<<"Exit main!\n";  
}
```

程序运行结果:

Constructor with parameters called.

Default constructor called.

5,8

0,0

Constructor with parameters called.

3,2

Destructor called.3,2

Exit main!

Destructor called.0,0

Destructor called.5,8

从该程序的输出结果看:

- 当创建对象时, 系统自动调用相匹配的构造函数;
- 当撤销对象时, 系统自动调用析构函数。
 - 对象撤销的顺序与创建的顺序正好相反。
 - 在相同的生存期的情况下, 先创建的对象后撤销, 后创建的对象先撤销。

- 若类的定义中没有显式定义析构函数，则编译器自动为该类产品产生一个缺省的析构函数。其格式为：

```
ClassName::~~ClassName( ){ }
```

它不执行任何操作。

- 无需显式定义析构函数的典型情况：
 - 撤消对象时不做任何清理工作。
- 需要显式定义析构函数的典型情况：
 - 类中包含指向动态内存的指针数据成员。由于它们所指向的动态内存空间是对象自己管理的，因此在对象即将撤消时，需通过析构函数将所管理的动态内存空间及时释放。

- 例5.10 使用析构函数，收回动态分配的存储空间。

```
#include<iostream.h>
```

```
#include<string.h>
```

```
class Person{
```

```
    char* name;
```

```
    int age;
```

```
public:
```

```
    Person(char* n,int a)
```

```
    { if(n)
```

```
        { name=new char[strlen(n)+1]; strcpy(name,n); }
```

```
        else name=0;
```

```
        age=a;
```

```
    }
```

```
    ~Person( ){ delete []name; }
```

```
    void Print(){ cout<<name<<','<<age<<endl; }
```

```
};
```

```
void main( ){ Person p( "John" ,20); p.Print( ); }
```

申请动态内存

释放动态内存

*5.5.3 构造函数的类型转换功能

- 如果类定义中提供了只带一个参数（没有其它参数，或其它参数都有缺省值）的构造函数，则当一个其它类型的数值或变量x赋值给该类的对象A时，由于类型不一致，系统自动以x为实参调用该类的单参数构造函数，创建一个该类的临时对象，并将该临时对象赋值给A，从而实现了隐式类型转换。

- 例5.11 构造函数完成类型转换功能的例子。

```
#include<iostream.h>
```

```
class Money{
```

```
    int yuan,jiao,fen;
```

```
public:
```

```
    Money (double d)
```

```
    { int t=d*100;
```

```
      yuan=t/100;jiao=(t-yuan*100)/10;fen=t%10;
```

```
    }
```

```
    Money (int y=0,int j=0,int f=0)
```

```
    { yuan=y;jiao=j;fen=f; }
```

```
    void print( )
```

```
    { cout<<yuan<< “元” <<jiao<< “角” <<fen<< “分\n”; }
```

```
};
```



单参数
构造函数

```
void main( )  
{ Money m1(25,5,7),m2(2.65);  
  m1.print( );  
  m2.print( );  
  Money m3=4.57;  
  m3.print( );  
  m3= 3.29 ;  
  m3.print( );  
}
```

并不做隐式类型转换，等价于：
Money m3(4.57);

隐式类型转换：调用单参数构造函数，创建Money类型的临时对象。

*5.5.4 拷贝构造函数

- 拷贝构造函数：特殊的构造函数，其形参是本类的对象的引用，其作用是使用一个已经存在的对象(由拷贝构造函数的参数指定的对象)去初始化一个新的同类的对象。
- 定义一个拷贝构造函数的一般形式为：
 ClassName::ClassName(ClassName &c)
 { ... } //函数体完成对应数据成员的赋值

- **例5.12** 使用完成拷贝功能的构造函数。

```
#include<iostream.h>
```

```
#include<math.h>
```

```
class Point{
```

```
    int x,y;
```

```
public:
```

```
    Point(int a=0,int b=0){ x=a;y=b; }
```

```
    Point(const Point& t) //拷贝构造函数
```

```
    { x=t.x; y=t.y;
```

```
      cout<<"调用了拷贝构造函数!\n";
```

```
    }
```

```
    Point& operator=(const Point& t)
```

```
{ x=t.x; y=t.y;
```

```
  cout<<"调用了拷贝赋值函数!\n";
```

```
  return * this;
```

```
}
```

```
int GetX( ){ return x; }
```

```
int GetY( ){ return y; }
```

```
void Show( ){ cout<<"x="<<x<<",y="<<y<<"\n'; }
```

```
};
```

```
double Distance(Point p1,Point p2)
```

```
{ int dx=p1.GetX()-p2.GetX();  
  int dy=p1.GetY()-p2.GetY();  
  return sqrt(dx*dx+dy*dy);  
}
```

```
Point Mid(Point& p1,Point& p2)
```

```
{ int x=(p1.GetX()+p2.GetX())/2;  
  int y=(p1.GetY()+p2.GetY())/2;  
  return Point(x,y);  
}
```

```
void main( )
```

```
{ Point p0,p1(8,15);
```

```
  Point p2(p1);//调用拷贝构造函数
```

```
  Point p3=p1; //等价: Point p3(p1);
```

```
  p1.Show(); p2.Show(); p3.Show();
```

```
  p3=Mid(p0,p3);
```

```
  cout<<Distance(p0,p3)<<endl;
```

```
}
```

- 函数Distance(Point,Point)的形参为类的对象，当执行函数调用Distance(p0,p3)时，实参初始化形参，两次调用拷贝构造函数，分别用实参对象p0和p3初始化形参对象p1和p2。

- 若函数形参是对象的引用，则调用该函数时，不调用拷贝构造函数，如本例的Mid函数所示。

调用了拷贝构造函数!

调用了拷贝构造函数!

x=8,y=15

x=8,y=15

x=8,y=15

调用了拷贝赋值函数!

调用了拷贝构造函数!

调用了拷贝构造函数!

8.06226

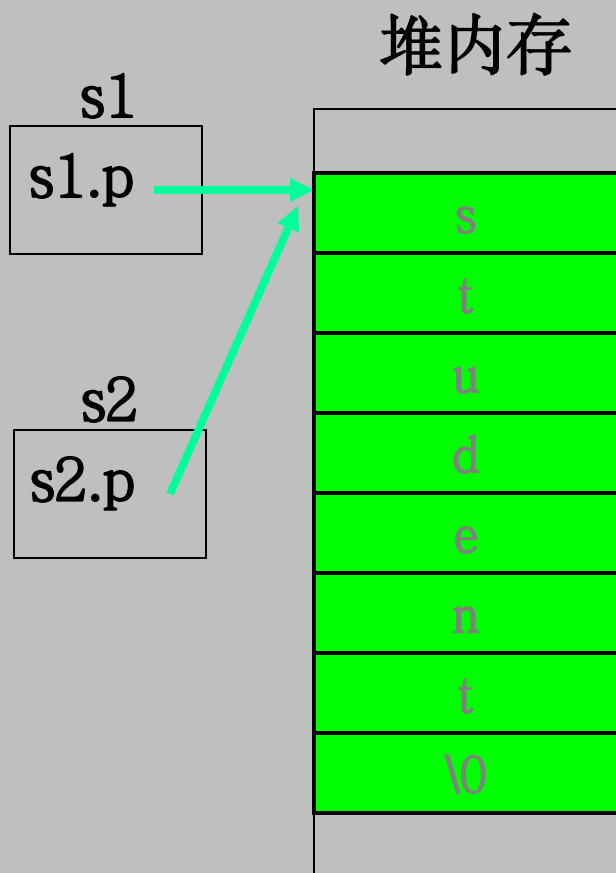
- 默认拷贝构造函数：
 - 若定义类时未显式定义该类的拷贝构造函数，则编译系统会在编译该类时自动生成一个默认拷贝构造函数。
 - 默认拷贝构造函数的功能是把一个已经存在的对象的每个数据成员的值逐个复制到新建立对象对应的数据成员中。
 - 例5.12中类Point的拷贝构造函数与默认的拷贝构造函数功能一样，不必显式定义。
 - **何时显式定义拷贝构造函数？** 若类中包含指向动态内存的指针时，用默认的拷贝构造函数初始化对象，将带来严重问题，如例5.13所示。

- 例5.13 默认的拷贝构造函数。

```
#include<iostream.h>
#include<string.h>
```

```
class String{
    char *p;
public:
    String(char*c=0)
    { if(c)
      { p=new char[strlen(c)+1]; strcpy(p,c);}
      else p=NULL;
    }
    ~String( ){ delete []p; }
    void Show( ){ cout<<"string="<<p<<"\n"; }
};
```

```
void main( )
{ String s1("student");
  String s2(s1);//A
  s1.Show( );
  s2.Show( );
}
```



- 执行A行语句：调用默认拷贝构造函数，用s1对象初始化s2对象，使s2对象的成员指针p与s1对象的成员指针p指向同一片内存。
- 主函数结束：先撤消s2，s2对象释放其所指动态内存；其后撤销s1对象，但因s1对象所指动态内存已被s2对象释放，造成重复释放同一块动态内存，出现运行错误。
- 定义拷贝构造函数，避免上述错误：

```
String(String&s)
{ if(s.p){
    p=new char[strlen(s.p)+1];
    strcpy(p,s.p);
}else p=0;
}
```

*5.5.5 对象成员与构造函数

- 对象成员：一个类的对象可做另一个类的数据成员。
- 例如：

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

class Person{
    char name[12];
    char sex[4];
    Date birthday;    //对象成员
public:
    Person(char*, char*,int,int,int);
    ...
};
```

- 对象成员的初始化

- 通过初始化表进行，如：

```
Person::Person(char*n,char*s,int y,int m,int d)
```

```
    :birthday(y,m,d)
```

```
{ strcpy(name,n); strcpy(sex,s); }
```

- 若未在初始化表中初始化对象成员，则系统自动调用该对象成员的缺省的构造函数来初始化。
 - 若类中包含对象成员，则在创建该类的对象时，先调用对象成员的构造函数，初始化相应的对象成员，后执行该类的构造函数，初始化该类的其他成员。
 - 如果一个类包含多个对象成员，对象成员的构造函数的调用顺序由它们在该类中的说明顺序决定，而与它们在初始化表中的顺序无关。

- 例5.14 初始化对象成员。

```
#include<iostream.h>
```

```
class Point{  
    int x,y;  
public:  
    Point(int a,int b)  
    { x=a; y=b;  
      cout<<"Constructor of class Point is called.\n";  
    }  
    void Show( )  
    { cout<<"x="<<x<<"",y="<<y<<endl; }  
};
```

```
class Rectangle{
```

```
    Point p;
```

```
    int length,width;
```

```
public:
```

```
    Rectangle(int l,int w,int a,int b):p(a,b)
```

```
{ length=l; width=w;
```

```
    cout<<"Constructor of class Rectangle is called.\n";
```

```
}
```

```
void Show( )
```

```
{ p.Show( );
```

```
    cout<<"length="<<length<<",width="<<width<<endl;
```

```
}
```

```
};
```

```
void main( ){ Rectangle r(5,4,45,55); r.Show( ); }
```

程序运行结果:

Constructor of class Point is called.

Constructor of class Rectangle is called.

x=45,y=55

length=5,width=4

5.6 友元

- 有时需要在对象外部频繁访问对象私有的或保护的数据成员，若通过调用公有成员函数间接访问，由于参数传递、类型检查等都需占用时间，必然降低程序的运行效率。
- 使用友元可在对象外部直接访问对象私有的和保护的数据成员。
- 友元以类的封装性的有限破坏为代价来提高程序的运行效率，应谨慎使用。
- 友元分为：友元函数和友元类。

5.6.1 友元函数

- 友元函数的说明格式:

friend <type> FuncName(<args>);

- 例5.15 用友元函数的方法计算两点距离。

```
#include<iostream.h>
```

```
#include <math.h>
```

```
class Point{
```

```
    int x,y;
```

```
public:
```

```
    Point(int i=0,int j=0){ x=i;y=j; }
```

```
    int GetX( ){ return x; }
```

```
    int GetY( ){ return y; }
```

```
    friend double distance(Point&,Point&);
```

```
};
```

普通函数distance()声明为Point类的友元函数。


```
double distance(Point& p1,Point& p2)
{ int dx=p1.x-p2.x,dy=p1.y-p2.y;
  return sqrt(dx*dx+dy*dy);
}
```

```
void main( )
{ Point a(8,15),b(3,7);
  cout<<"The distance is:"<<distance(a,b)<<endl;
}
```

友元函数distance()可通过对象名直接访问Point类的对象p1和p2的私有数据成员x和y,提高了程序的效率。

程序运行结果:

The distance is:9.43398

- 友元函数说明:

- 友元函数应在类中说明，可放在类的私有、公有或保护部分，效果一样。友元函数体可在类内或类外定义。
- 友元函数不是类的成员，不带this指针，因此必须将对象名或对象的引用作为友元函数的参数，并在函数体中用运算符“.”来访问对象的成员。如上例中的p1.x。
- 友元函数可直接访问相关类中的所有成员。
- 一个类的成员函数也可作为另一个类的友元函数，例如:

```
class A{
```

```
...
```

```
int f(...);
```

```
};
```

```
class B{
```

```
...
```

//成员定义

```
friend int A::f(...); //声明类A的成员函数f( )
```

```
};
```

// 是类B的友元

在声明这个友元函数时需要在函数名前面加上它的类名和作用域运算符“::”。

5.6.2 友元类

- 若声明A类为B类的友元，则A类的所有成员函数都成为B类的友元函数。有时友元类的使用也是必要的选择。
- 例5.16 友元类。

```
#include <iostream.h>
```

```
#include <string.h>
```

```
class Date{
```

```
    int year,month,day;
```

```
public:
```

```
    Date(int y=0,int m=0,int d=0){ year=y;month=m;day=d;}
```

```
    void display(){ cout<<year<<month<<day; }
```

```
    friend class Person;
```

```
};
```

```

class Person{
    char name[12];
    char sex[4];
    Date birthday;
public:
    Person(char*n,int y,int m,int d,char*s)
    { strcpy(name,n); strcpy(sex,s);
      birthday.year=y; birthday.month=m; birthday.day=d;
    }
    void display( )
    { cout<<"name:"<<name<<",sex:"<<sex;
      cout<<" ,birthday:"<<birthday.year<<'.' '<<
        <<birthday.month<<'.'<<birthday.day<<endl;
    }
};

void main( )
{ Person p( "张三" ,1983,8,20," 男" ); p.display( ); }

```

程序运行结果:

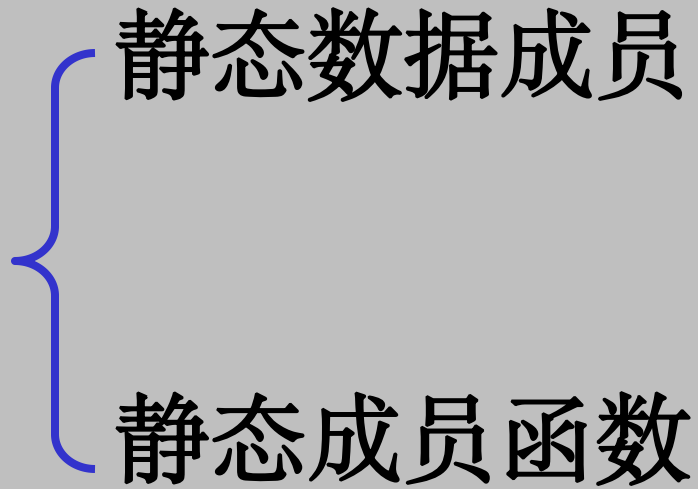
name:张三,sex:男,birthday:1983.8.20

因Person类为Date类的友元，在Person类的成员函数可直接操作Date类的对象birthday的数据成员。

- 友元的其它注意事项：
 - 友元关系没有传递性。例如，说明类A是类B的友元，类B是类C的友元时，类A并不一定是类C的友元。
 - 友元关系不具有交换性，即说明类A为类B的友元时，类B并不一定是类A的友元。
 - 友元关系没有继承性。因友元函数不是类的成员函数，当然不存在继承关系。
 - 谨慎使用友元。友元提高程序的运行效率是通过破坏类的封装性取得的，若过多过滥使用友元，则会严重降低程序的可维护性。

*5.7 类的静态成员

- 分类:



5.7.1 静态数据成员

- 静态数据成员的必要性

//学生类

```
class Student{  
    int No;  
    char *name;  
    ...  
}
```

若要统计学生
总数，这个数
据存放在什么
地方呢？

类外变量？

不能实现数据的隐藏！

类中增加数据成员？

冗余！不一致！

- **静态数据成员的定义：**先在类体内，做引用性说明，即在该成员名的类型说明符前加关键字static。后在文件作用域做定义性说明。

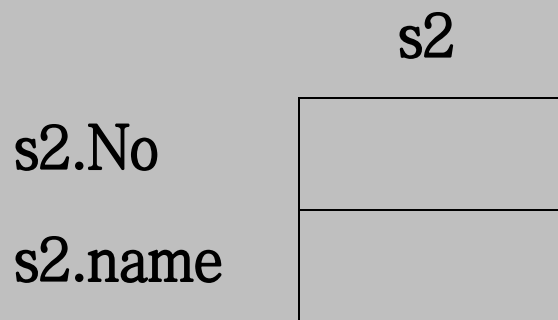
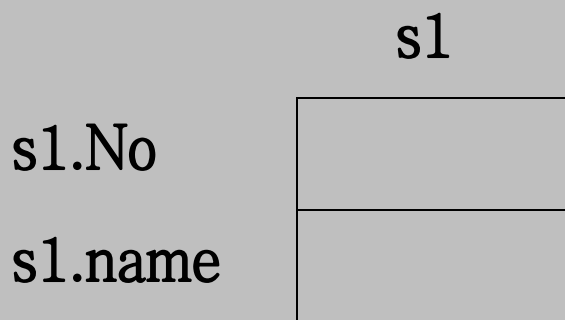
- **例：**

```
class Student{  
    int No;  
    char *name;  
    static int count;//引用性说明：说明静态  
                        //数据成员count的访问权限和作用域  
    ...  
};
```

```
int Student::count=0;//定义性说明：为静态  
                        //数据成员count分配内存和初始化
```


- **与普通数据成员的区别：** 类的普通数据成员在类的每个对象中都拥有一个拷贝；而静态数据成员，每个类只有一个拷贝，由该类的所有对象共同维护和使用，从而实现了同一类的不同对象之间的数据共享。

例： void main(){
 Student s1,s2;
}



- 普通成员函数必须具体作用于某个对象，而静态成员函数并不具体作用于某个对象。因此，静态成员不需要通过对象就能访问。
- 静态数据成员的访问方式：

类名::静态数据成员名

或

对象名.静态数据成员名

或

对象指针->静态数据成员名

或

引用.静态数据成员名

- 例5.17 具有静态数据成员的Apple类。

```
#include<iostream.h>
```

```
class Apple{  
    float weight;  
    static int count;//静态数据成员的引用性说明  
    static float total_weight;  
public:  
    Apple(float=0);  
    ~Apple( );  
    void Show( );  
};
```

```
Apple::Apple(float w)
{ weight=w; count++;
  total_weight+=weight;
  cout<<"Number of apples="<<count<<"\n";
}

Apple::~~Apple( )
{ count--; total_weight-=weight;
  cout<<"Number of apples="<<count<<"\n";
}

void Apple:: Show( )
{ cout<<"weight="<<weight
  <<",total_weight="<<total_weight<<"\n";
}
```

//静态数据成员的定义性说明： 默认值为0

```
int Apple::count;
```

```
float Apple::total_weight;
```

```
void main( )
```

```
{  Apple redone(100);
```

```
    Apple greenone(200);
```

```
    Apple yellowone(250);
```

```
    redone.Show( );
```

```
    greenone.Show( );
```

```
    yellowone.Show( );
```

```
}
```

5.7.2 静态成员函数

- 采用静态成员函数的好处是可以不依赖于任何对象，直接访问静态数据成员。
- 静态成员函数的引用格式：

static 返回值类型 成员函数名(参数表);

- 静态成员函数的定义格式：

<类名>::<静态成员函数>(<参数表>)

或

<对象名>.<静态成员函数>(<参数表>)

例5.18 具有静态数据成员和静态成员函数的Apple类。

```
#include<iostream.h>
class Apple{
    float weight;
    static int count;
    static float total_weight;
public:
    Apple(float=0);
    ~Apple();
    void ShowWeight();
    static void ShowCount();
    static void ShowTotalWeight();
};
```

```
Apple::Apple(float w)
{ weight=w;count++; total_weight+=weight;
  cout<<"Number of apples="<<count<<'\n';
}
Apple::~~Apple( )
{ count--; total_weight-=weight;
  cout<<"Number of apples="<<count<<'\n';
}
void Apple:: ShowWeight( )
{ cout<<"weight="<<weight<<'\n'; }
void Apple::ShowCount( )
{ cout<<"Number of apples="<<count<<'\n'; }
void Apple::ShowTotalWeight( )
{ cout<<"Total weight of apples="
  <<total_weight<<'\n';
}
```



```
int Apple::count;  
float Apple::total_weight;
```

```
void main( )  
{ Apple::ShowCount( );  
  Apple::ShowTotalWeight( );  
  Apple redone(100),greenone(200),yellowone(250);
```

```
  redone.ShowWeight( );  
  greenone.ShowWeight( );  
  yellowone.ShowWeight( );
```

```
  Apple::ShowCount( );  
  Apple::ShowTotalWeight( );  
}
```

程序运行结果:

Number of apples=0
Total weight of apples=0

weight=100

weight=200

weight=250

Number of apples=3

Total weight of apples=550

- 静态成员函数可直接访问静态数据成员，但不能直接访问非静态数据成员(因其没有this指针)。若要访问非静态数据成员，则可通过形参对象来访问。如：

```
class Myclass{
    int x;
    static int y;
public:
    static int f(Myclass m);
};
int Myclass::y=9;
int Myclass::f(Myclass m)
{ cout<<x+y;    //对x的访问是错误的
  return m.x+y; //正确
}
```

*5.8 const对象和成员函数

- 常成员函数：在参数表后用const修饰。
 - 定义格式：
<类型说明符> <函数名>(<参数表>) **const**;
 - 特点：常成员函数执行期间不应修改其所作用的对象。因此，在常量成员函数中，不能修改成员变量的值（静态成员变量除外），也不能调用该对象的非const成员函数间接改变本对象的数据成员(静态成员函数除外)，否则编译时会产生出错信息。
 - 用途：**主要用于定义不改变数据成员的成员函数。**
- const对象：用const说明的对象。
 - 定义格式：
const <类名> <对象名>(<初值>);
 - 特点：其数据成员在该对象的生存期间不能改变。
 - 例如：
const Date birthday(1983,7,13);
 - const对象只能调用常成员函数(构造函数和析构函数除外)，以保证其数据成员不被修改。

- 例5.19 常成员函数举例。

```
#include<iostream.h>
```

```
class Date{
```

```
    int year,month,day;
```

```
public:
```

```
    Date(int y,int m,int d)
```

```
    { year=y;month=m;day=d; }
```

```
    void SetDate(int y,int m,int d)
```

```
    { year=y;month=m;day=d; }
```

```
    void display( )
```

```
    { cout<<year<<','<<month<<','<<day<<endl; }
```

```
    void display( ) const
```

```
    { cout<<year<<','<<month<<','<<day<<endl; }
```

```
};
```

```
void main( )  
{ Date weekend(2005,7,8);  
  weekend.display( );  
  weekend.SetDate(2005,7,22);  
  weekend.display( );  
  
  const Date birthday(1983,7,13);  
  birthday.display( );  
}
```

程序运行结果:

2005,7,8

2005,7,22

1983,7,13

- 常对象和常成员函数的说明:

- 常成员函数在类外定义时也应带const关键字。例如:

```
void Date::display( ) const
```

```
{ cout<<year<<','<<month<<','<<day<<endl; }
```

- 常成员函数不能直接改变本对象的数据成员，也不能调用该对象的非const成员函数间接改变本对象的数据成员。

- const可用于区分函数的重载。如类Date中有声明:

```
void display( );
```

```
void display( ) const;
```

- 常对象只能调用其常成员函数，不能调用其它成员函数。 **若常对象调用非const成员函数，如:**

```
birthday.SetDate(2005,7,22);
```

则编译时将出现出错信息。

*5.9 应用实例

- 例5.20 用类和对象编程解决实际问题。设有若干学生数据，包括学号、姓名、数学成绩和C++成绩，求每位学生的总分及每门课程的平均分。
- 分析：
 - Student类包括学号、姓名等数据成员，以及对这些数据进行操作的功能成员。其中，静态数据成员count、msum和csum，分别存放学生人数、数学和C++课程的总分。
 - 链表类StuList用于存放学生数据，可完成链表创建、输出等任务。它声明为类Student的友元，更便于高效操作Student类的数据成员。

```
#include<iostream.h>
```

```
#include<string.h>
```

```
class Student{           //学生类
```

```
protected:
```

```
    int no;               //学号
```

```
    char name[10];        //姓名
```

```
    float math,cpp;       //数学成绩、C++成绩
```

```
    static int count;     //累计总人数
```

```
    static float msum,csum;//分别累计数学和C++总分
```

```
    Student* next;
```

```
public:
```

```
    Student( ){ }
```

```
    Student(int,char*,float,float,Student* =NULL);
```

```
    float sum( ) const    //计算每位学生的总分
```

```
    { return math+cpp; }
```



```

void show1( ) const //输出每位学生的基本信息
{ cout<<no<<"\t"<<name<<"\t"
  <<math<<"\t"<<cpp<<"\t"<<sum( )<<endl;
}
static void show2( ) //输出每门课程的平均分
{ if(count)
  cout<<(msum/count)<<"\t"<<(csum/count)<<endl;
}
friend class StuList; //友元类
};

Student::Student(int no,char*name,float math,
  float cpp,Student*next)
{ Student::no=no; strcpy(Student::name,name);
  this->math=math; this->cpp=cpp; this->next=next;
  count++; msum+=math; csum+=cpp;
}

```

```
class StuList{
    Student*head;
public:
    StuList( )
    { head=NULL; }
    ~StuList( );
    void create( );           //创建链表
    void print( ) const;     //输出链表
};
```

```
StuList::~~StuList( ){
    Student*p;
    while(head)
    { p=head; head=head->next; delete p; }
}
```

```
void StuList::create( ){ //尾结点插入法创建单向链表
    Student*p1,*p2;
    int no;
    char name[10];
    float math,cpp;
    cout<<"输入学号,-1表示结束:";
    cin>>no;
    while(no!=-1){
        cout<<"输入姓名及数学、C++语言成绩:",
        cin>>name>>math>>cpp;
        p1=new Student(no,name,math,cpp);
        if(!head) head=p1,p2=p1;else p2->next=p1,p2=p1;
        cout<<"输入学号,-1表示结束:";
        cin>>no;
    }
}
```

```
void StuList::print( ) const
{  Student* p=head;
   cout<<"学号\t姓名\t数学\tC++语言\t总分\n";
   while( p ){ p->show1( ); p=p->next; }
   cout<<"每门平均分:\t";
   Student::show2( );
}
```

```
int Student::count;    //定义静态数据成员并初始化
float Student::msum,Student::csum;
```

```
void main( )
{  StuList li;
   li.create( );
   li.print( );
}
```

程序的一次运行结果如下:

输入学号,-1表示结束:1✓

输入姓名及数学、C++语言成绩:Tom 90 95✓

输入学号,-1表示结束:2✓

输入姓名及数学、C++语言成绩:Sun 99 83✓

输入学号,-1表示结束:3✓

输入姓名及数学、C++语言成绩:Pod 63 53✓

输入学号,-1表示结束:-1✓

学号	姓名	数学	C++语言	总分
----	----	----	-------	----

1	Tom	90	95	185
---	-----	----	----	-----

2	Sun	99	83	182
---	-----	----	----	-----

3	Pod	63	53	116
---	-----	----	----	-----

每门平均分:	84	77		
--------	----	----	--	--