高级语言程序设计

UNIT 07　类的继承

主讲人：董性平　教授

邮箱：xingpingdong@whu.edu.cn

个人主页：
http://jszy.whu.edu.cn/dongxingping/zh_CN/index.htm

继承和派生的区别？

继承与派生其实是同一过程从不同的角度看：

① 我们将保持已有类的特性而**构造新类**的过程称为继承，说白了继承的目的就是实现原来设计与**代码的重用**，希望尽量利用原有的类。

② 然而当**新的问题**出现，原有程序无法解决或不能完全解决时，需要对原有程序进行改造，在已有类的基础上新增自己的特性而产生新类的过程称为派生

本讲提纲

基类和派生类

访问控制

类型兼容规则

派生类的构造、析构函数

派生类成员的标识与访问

程序实例——用高斯消去法解线性方程组

综合实例*——个人银行账户管理程序

深度探索

1. 基类和派生类

7.1 继承关系举例

继承与派生是同一过程从不同的角度来看!

被继承的已有类称为基类（或父类）。

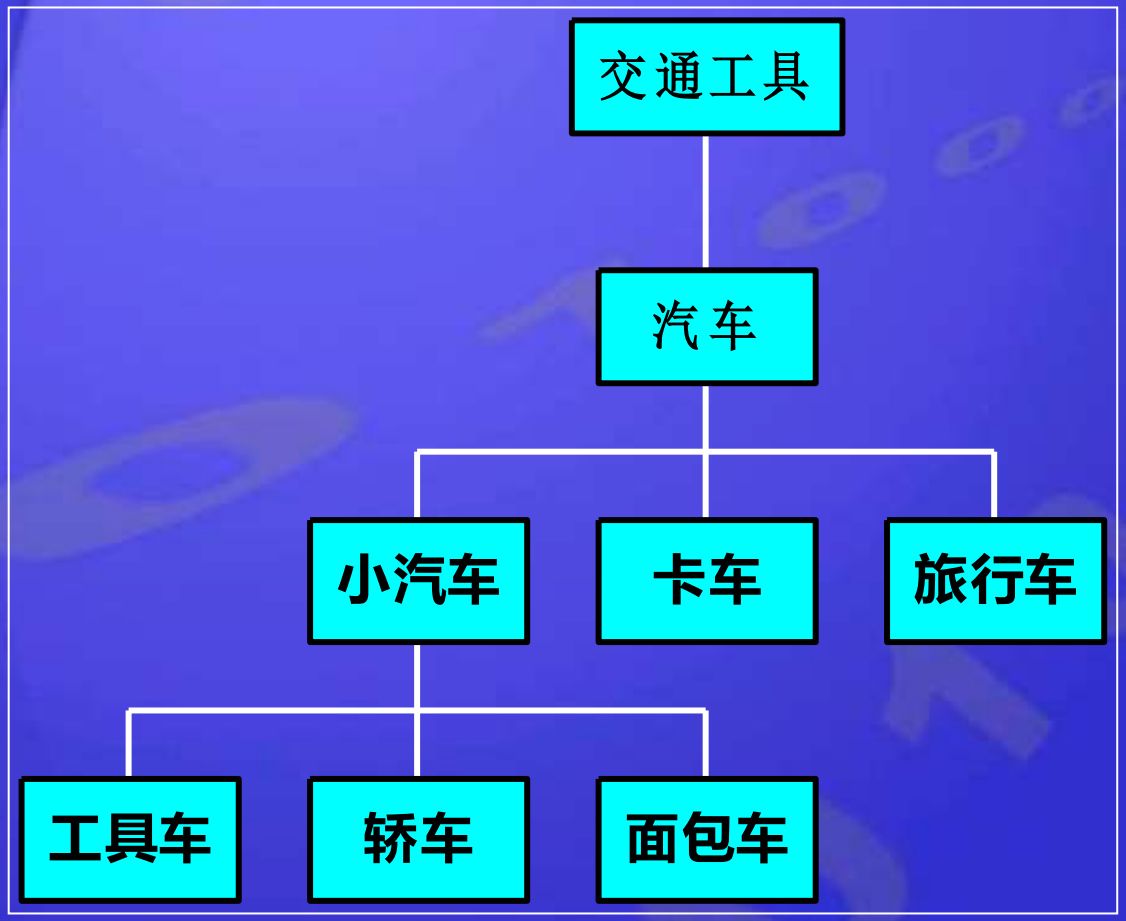
派生出的新类称为派生类。

直接参与派生出某类的基类称为直接基类，基类的基类甚至更高层的基类称为间接基类。

7.1 继承关系举例

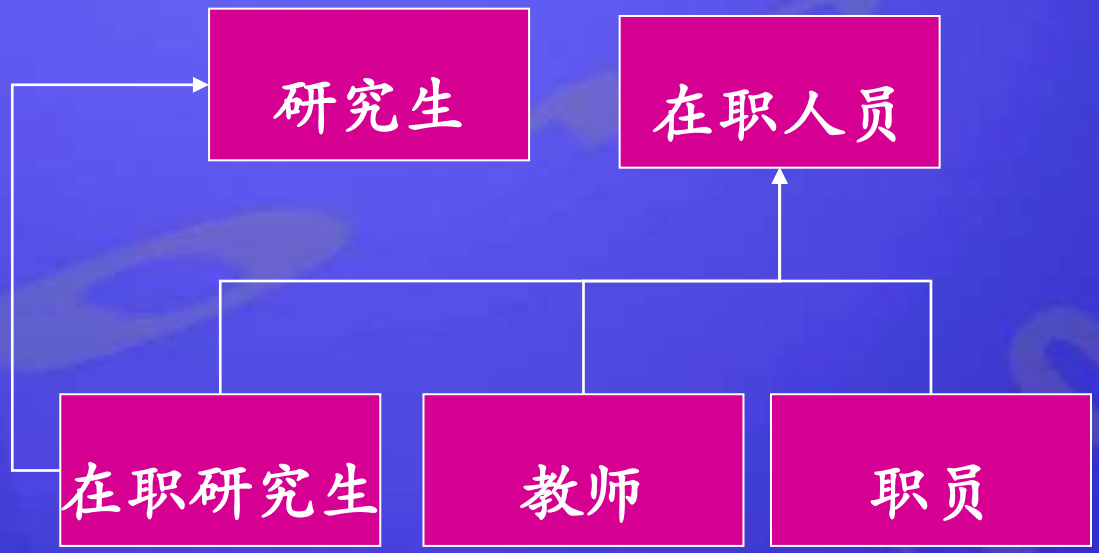
- 派生类可以具有基类的特性
- 共享基类的成员函数，使用基类的数据成员
- 还可以定义自己的新特性
- 定义自己的数据成员和成员函数

单继承



单继承：从一个基类派生的称为单继承

多继承



多继承：从两个或多个基类派生的称为多继承

类的层次关系

类继承层次结构：基类和派生类的实例集合

类继承：从基于对象程序设计扩展到面向对象程序设计

类继承层次结构好处：

- 对事物进行分类：基类表示了该种类型的一般概念，而子类表示了一个特殊概念
- 支持软件的增量开发：支持软件开发的逐步升级
- 对概念进行组合：继承关系中，基类成为派生类的一部分。

类的层次关系

类继承层次结构的好处：

继承层次结构的主要好处是可以针对基类的公有接口进行编程而不是针对组成继承层次的个别类型。通过这种方式，代码可以不受层次结构变化的影响。

类的层次关系

```
//运算基类
class Operation
{ public:
      ……
      virtual double GetResult( );
};
```

虚函数：用于支持动态绑定

继承和动态绑定：子类型多态性，为面向对象提供基础

```
//运算类
class OperationAdd: public Operation
{  public:
          double GetResult()
          {
                    double result = 0;
                    result = numbera + numberb;
                    return result;
          }
};
```

类的层次关系

如何确定类层次结构的成员，通常是以下步骤反复迭代的过程：

- 类层次结构的公有接口应该提供哪些操作
- 这些操作之中哪些应该被声明为虚拟的
- 单个的派生类需要哪些其他的操作
- 在抽象基类中应该声明哪些数据成员
- 单个的派生类要求哪些数据成员

继承与派生的目的

继承的目的：实现代码重用。

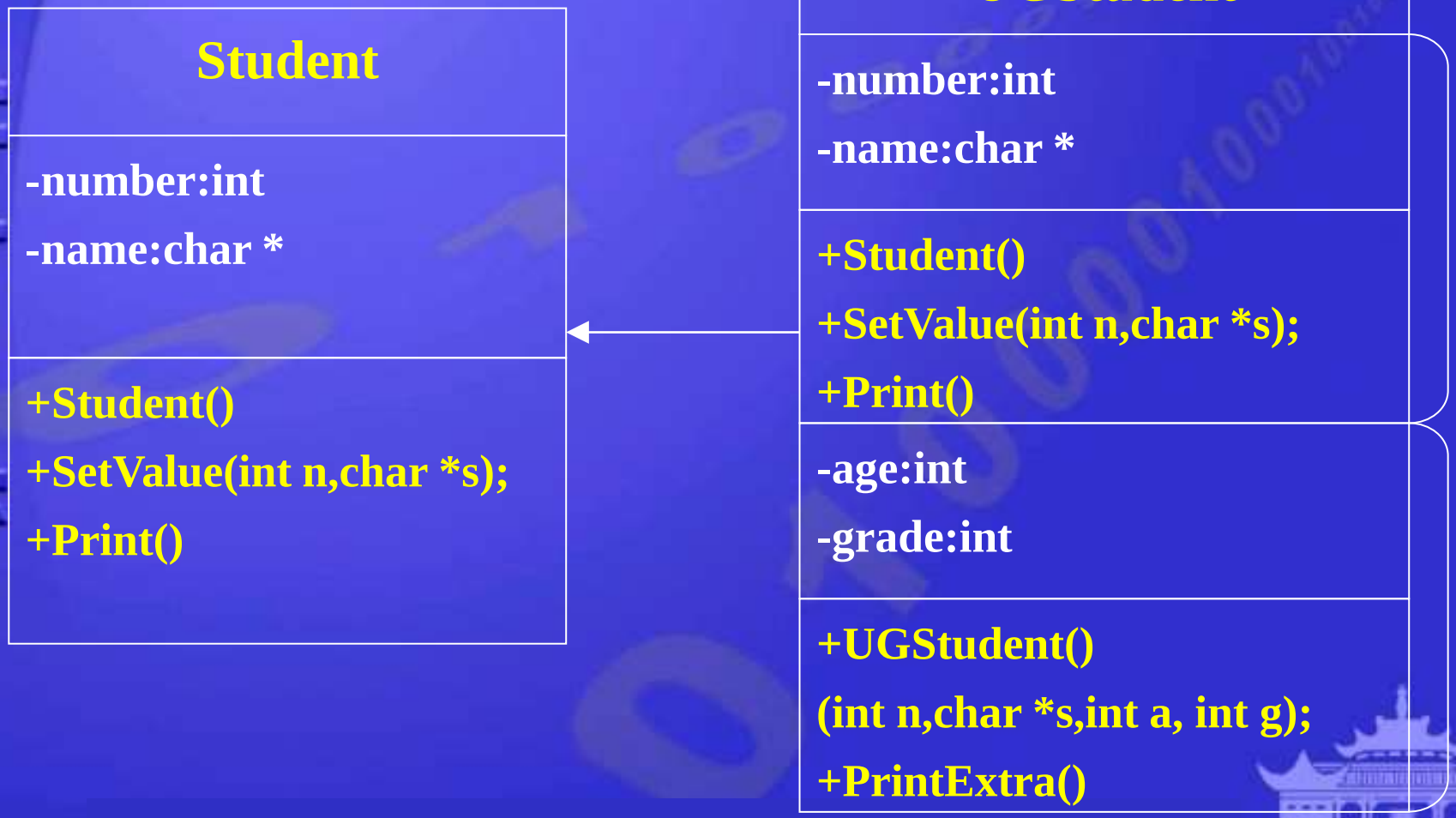
派生的目的：当新的问题出现，原有程序无法解决（或不能完全解决）时，需要对原有程序进行改造。

7.1.2 派生类的定义(单继承)

```
class 派生类名:[public/private/protected] 基类名
{
        派生类数据成员和函数成员定义
};
```

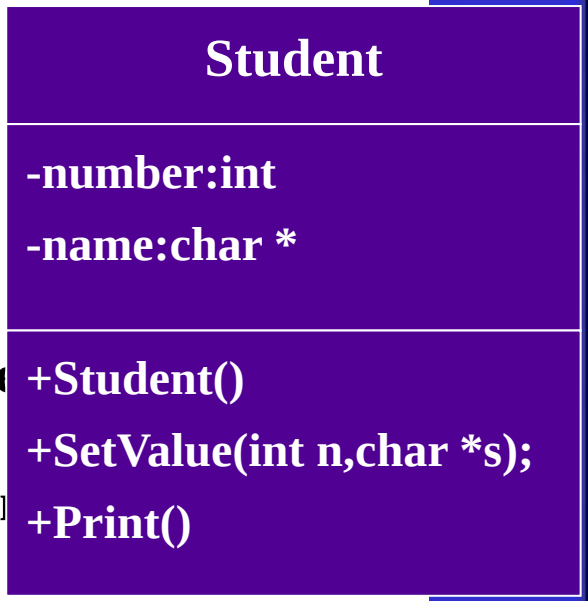
- 基类名：已有的类的名称；
- 派生类名：基础原有类的特性而生成的新类的名称；
- 派生方式：默认为私有继承
  public　公有继承
  private　私有继承
  protected 保护继承
- 派生方式的不同：
  派生类中新增成员对基类成员的访问控制；
  派生类外部，通过派生类对象对基类成员的访问控制。
- 派生类不能继承基类的构造函数和析构函数

单继承范例


Student
-number:int
-name:char *
+Student()
+SetValue(int n,char *s);
+Print()
UGStudent
-number:int
-name:char *
+Student()
+SetValue(int n,char *s);
+Print()
-age:int
-grade:int
+UGStudent()
(int n,char *s,int a, int g);
+PrintExtra()
继承自基类
新增成员

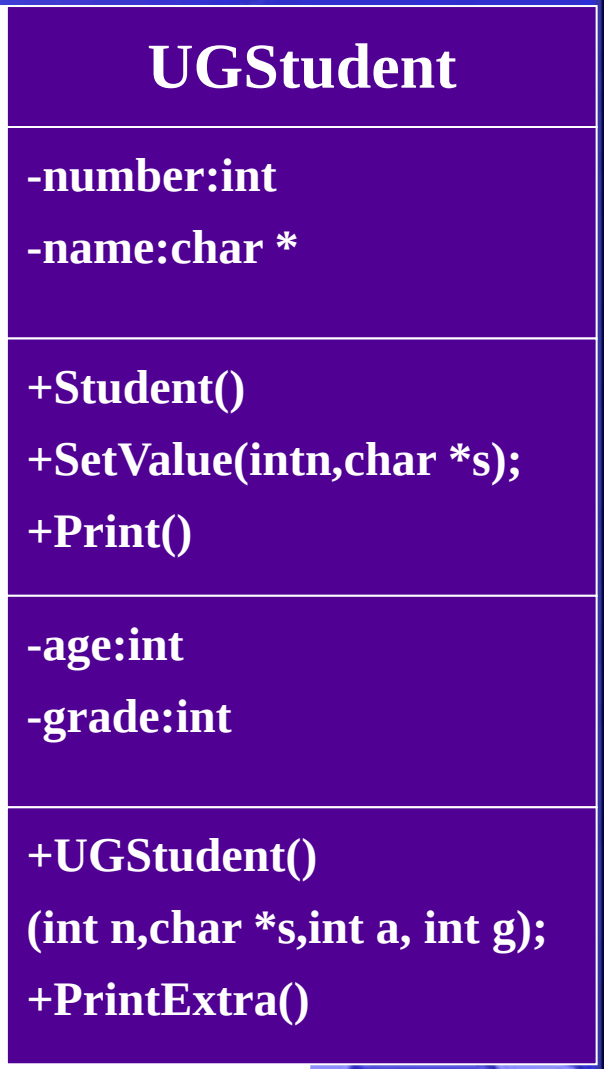
单继承范例

```
#include <iostream>
#include <string.h>
using namespace std;
 class Student
{   private:
       int number;
       char *name;
  public:
       Student(){ number=0;  name=new char[1];name
       void SetValue( int n,const char *s1)
       {  number=n; delete [] name; name = new char[
          strcpy(name, s1);
       }
       void Print()
       {
          cout<<"Number:"<<number<<endl;
          cout<<"Name:"<<name<<endl;
       }
      ~Student( ){ delete[] name;}
};
```



单继承范例

```
class UGStudent:public Student
{
  private:
      int age;  int grade;
  public:
      UGStudent(){ SetValue(0,""); age=0; grade=0; }
      UGStudent(int n,const char *s1,int a, int g)
      {
         SetValue(n,s1);
         age=a;            grade=g;
      }
      void PrintExtra()
      {
        cout<<"Age:"<<age<<endl;
        cout<<"Grade:"<<grade<<endl;
      }
};
```



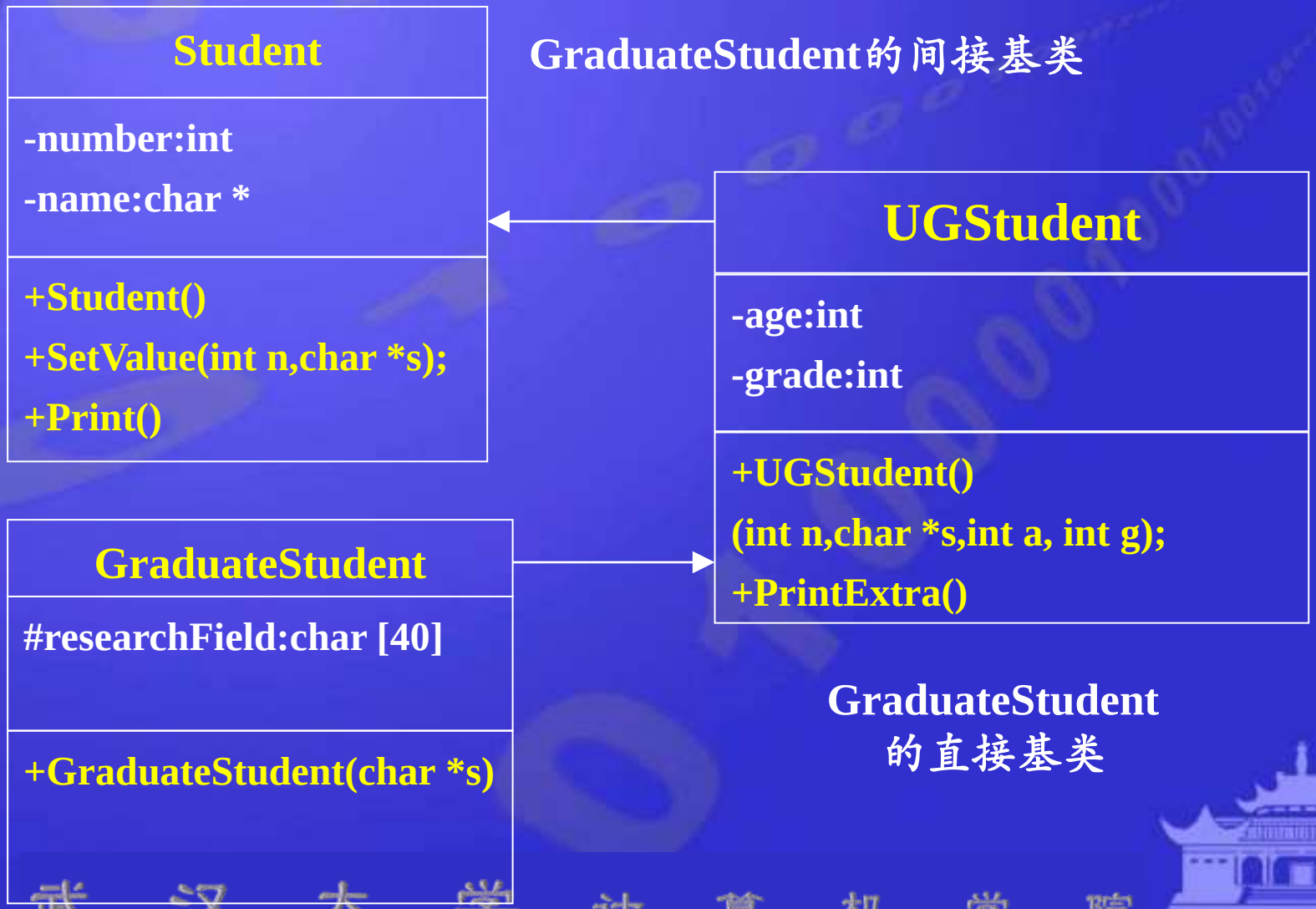
单继承范例

```
int main( )
{
   UGStudent st1(100,"wang",18,1);
   st1.Print();                  //调用基类的函数
   st1.PrintExtra();             //调用派生类的新定义的函数
   return 0;
};
```

运行结果:

```
Number:100
Name:wang
Age:18
Grade:1
```

单继承范例



7.1.2派生类的定义（多继承）

多继承可以看作是单继承的扩展，派生类与每个基类之间的关系可以看作是一个单继承。在C++中，多继承的定义格式如下：

```
class 派生类名:继承方式1 基类名1,...,继承方式n 基类名n
{
   派生类新定义成员
};
```

多继承范例

例如：

```
class Derived: public Base1, private Base2
{
public:
        Derived ();
        ~Derived ();
};
```

7.1.3 派生类生成过程

吸收基类成员

- ✓ 吸收基类成员之后，派生类实际上就包含了它的全部基类中除构造函数和析构函数之外的所有成员。

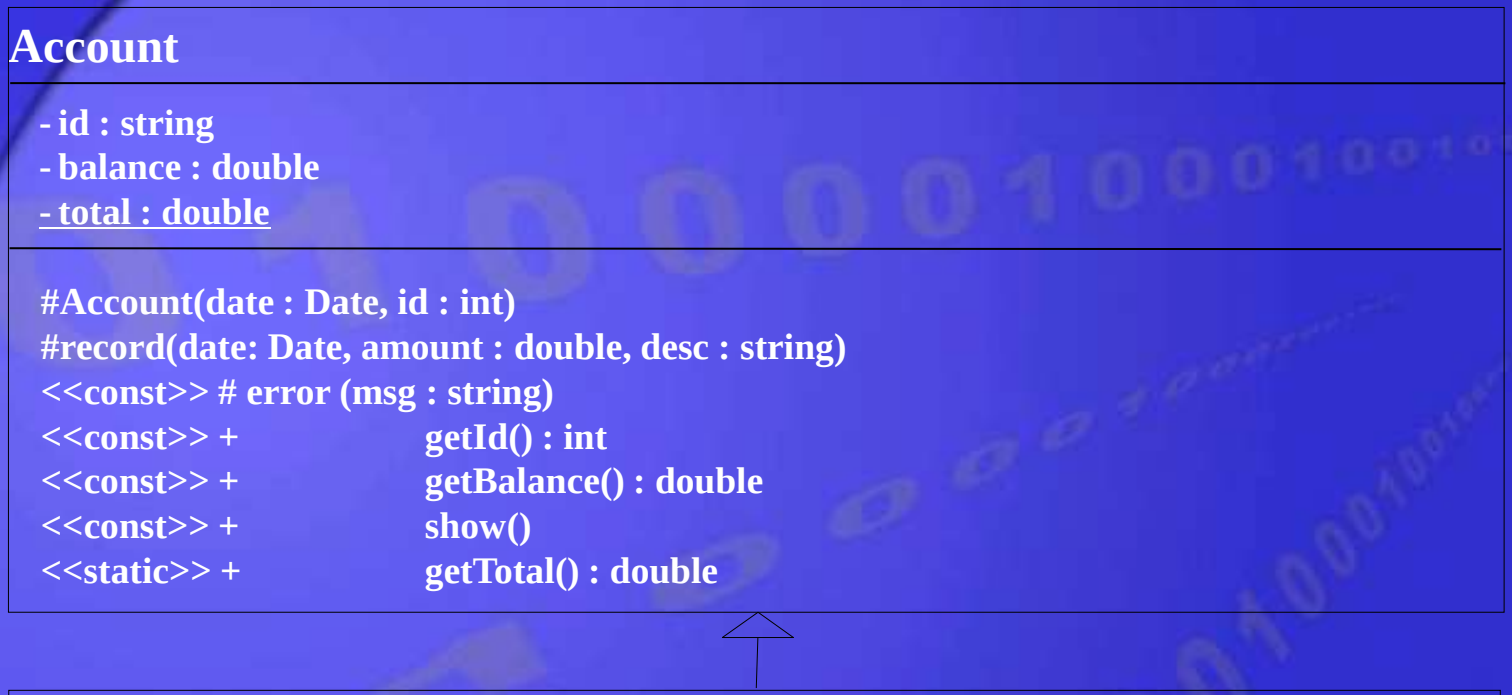
改造基类成员

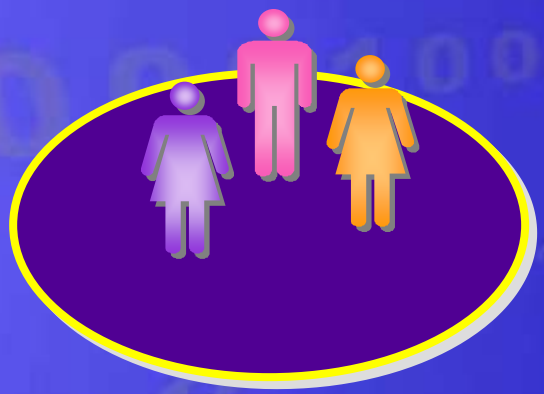
- ✓ 如果派生类声明了一个和某基类成员同名的新成员（如果是成员函数，则参数表也要相同，参数不同的情况属于重载），派生的新成员就覆盖了外层同名成员

添加新的成员

- ✓ 派生类新成员的加入是继承与派生机制的核心，是保证派生类在功能上有所发展

账户类VS信用账户类





2. 访问控制

7.2 访问控制

不同继承方式的影响主要体现在：

- ✓ 派生类成员对基类成员的访问权限
- ✓ 通过派生类对象对基类成员的访问权限

三种继承方式

- ✓ 公有继承
- ✓ 私有继承
- ✓ 保护继承

7.2.1 公有继承(public)

基类的public和protected成员的访问属性在派生类中保持不变，但基类的private成员不可直接访问。

派生类中的成员函数可以直接访问基类中的public和protected成员，但不能直接访问基类的private成员。

通过派生类的对象只能访问基类的public成员。

例7-1 公有继承举例类图

例7-1 公有继承举例

```
//Point.h
#ifndef _POINT_H
#define _POINT_H

class Point {       //基类Point类的定义
public:             //公有函数成员
    void initPoint(float x = 0, float y = 0)
    { this->x = x; this->y = y;}
    void move(float offX, float offY)
    { x += offX; y += offY; }
    float getX() const { return x; }
    float getY() const { return y; }
private:            //私有数据成员
    float x, y;
};

#endif //_POINT_H
```

例7-1（续）

```
//Rectangle.h
#ifndef _RECTANGLE_H
#define _RECTANGLE_H
#include "Point.h"
class Rectangle: public Point {      //派生类定义部分
public:                              //新增公有函数成员
    void initRectangle(float x, float y, float w, float h) {
        initPoint(x, y);             //调用基类公有成员函数
        this->w = w;
        this->h = h;
    }
    float getH() const { return h; }
    float getW() const { return w; }
private:                             //新增私有数据成员
    float w, h;
};
#endif //_RECTANGLE_H
```

例7-1 (续)

```
#include <iostream>
#include <cmath>
using namespace std;

int main() {
    Rectangle rect;          //定义Rectangle类的对象

    //设置矩形的数据
    rect.initRectangle(2, 3, 20, 10);
    rect.move(3,2);          //移动矩形位置
    cout << "The data of rect(x,y,w,h): " << endl;
    //输出矩形的特征参数
    cout << rect.getX() <<", "
         << rect.getY() << ", "
         << rect.getW() << ", "
         << rect.getH() << endl;
    return 0;
}
```

7.2.2 私有继承(private)

基类的public和protected成员都以private身份出现在派生类中，但基类的private成员不可直接访问。

派生类中的成员函数可以直接访问基类中的public和protected成员，但不能直接访问基类的private成员。

通过派生类的对象不能直接访问基类中的任何成员。

例7-2 私有继承举例

```
//Point.h
#ifndef _POINT_H
#define _POINT_H

class Point {       //基类Point类的定义
public:             //公有函数成员
    void initPoint(float x = 0, float y = 0)
    { this->x = x; this->y = y;}
    void move(float offX, float offY)
    { x += offX; y += offY; }
    float getX() const { return x; }
    float getY() const { return y; }
private:            //私有数据成员
    float x, y;
};

#endif //_POINT_H
```

例7-2 （续）

```
//Rectangle.h
#ifndef _RECTANGLE_H
#define _RECTANGLE_H
#include "Point.h"
class Rectangle: private Point {    //派生类定义部分
public:                             //新增公有函数成员
    void initRectangle(float x, float y, float w, float h) {
        initPoint(x, y);            //调用基类公有成员函数
        this->w = w;
        this->h = h;        }
    void move(float offX, float offY) {
        Point::move(offX, offY);    } //重新定义函数?
    float getX() const { return Point::getX(); }
    float getY() const { return Point::getY(); }
    float getH() const { return h; }
    float getW() const { return w; }
private:                            //新增私有数据成员
    float w, h;
};
#endif //_RECTANGLE_H
```

例7-2 (续)

```
#include <iostream>
#include <cmath>
using namespace std;

int main() {
    Rectangle rect;          //定义Rectangle类的对象

    rect.initRectangle(2, 3, 20, 10);//设置矩形的数据
    rect.move(3,2);          //移动矩形位置
    cout << "The data of rect(x,y,w,h): " << endl;
    cout << rect.getX() <<", "       //输出矩形的特征参数
         << rect.getY() << ", "
         << rect.getW() << ", "
         << rect.getH() << endl;
    return 0;
}
```

7.2.3 保护继承(protected)

基类的public和protected成员都以protected身份出现在派生类中，但基类的private成员不可直接访问。

派生类中的成员函数可以直接访问基类中的public和protected成员，但不能直接访问基类的private成员。

通过派生类的对象不能直接访问基类中的任何成员

protected 成员的特点与作用

对建立其所在类对象的模块来说，它与 private 成员的性质相同。

对于其派生类来说，它与 public 成员的性质相同。

既实现了数据隐藏，又方便继承，实现代码重用。

例： protected 成员举例

```
class A {
protected:
    int x;
};

int main() {
    A a;
    a.x = 5;    //错误
}
```


例(续)

```
class A {
protected:
    int x;
};

class B: public A{
public:
    void function();
};

void B:function() {
    x = 5;   //正确
}
```

多继承举例

```
class A {
public:
    void setA(int);
    void showA() const;
private:
    int a;
};
class B {
public:
    void setB(int);
    void showB() const;
private:
    int b;
};
```

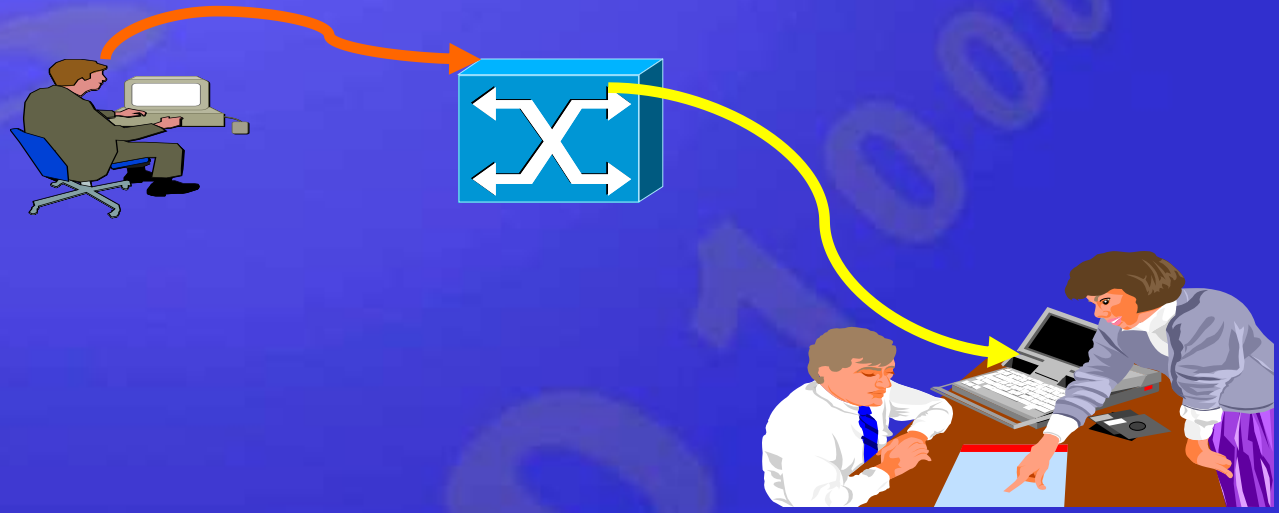
```
class C : public A, private B {
public:
    void setC(int, int, int);
    void showC() const;
private const:
    int c;
};
```

多继承举例(续)

```
void  A::setA(int x) {
    a=x;
}
void B::setB(int x) {
    b=x;
}
void C::setC(int x, int y, int z) {
    //派生类成员直接访问基类的
    //公有成员
    setA(x);
    setB(y);
    c = z;
}
//其他函数实现略
```

```
int main() {
    C obj;
    obj.setA(5);
    obj.showA();
    obj.setC(6,7,9);
    obj.showC();
    // obj.setB(6);  错误
    // obj.showB(); 错误
    return 0;
}
```

3. 类型兼容规则



7.3 类型转换规则

一个公有派生类的对象在使用上可以被当作基类的对象，反之则禁止。具体表现在：

- ✓ 派生类的对象可以隐含转换为基类对象。
- ✓ 派生类的对象可以初始化基类的引用。
- ✓ 派生类的指针可以隐含转换为基类的指针。

通过基类对象名、指针只能使用从基类继承的成员

例7-3 类型转换规则举例

```
#include <iostream>
using namespace std;
class Base1 {                          //基类Base1定义
public:
    void display() const {
        cout << "Base1::display()" << endl;
    }
};
class Base2: public Base1 {            //公有派生类Base2定义
public:
    void display() const {
        cout << "Base2::display()" << endl;
    }
};
class Derived: public Base2 {          //公有派生类Derived定义
public:
    void display() const {
        cout << "Derived::display()" << endl;
    }
};
```

例7-3 (续)

绝对不要重新定义继承而来的非虚函数!

```
void fun( Base1 *ptr ) {                    //参数为指向基类对象的指针
    ptr->display();                         //"对象指针->成员名"
}

int main() {                                //主函数
    Base1 base1;                            //声明Base1类对象
    Base2 base2;                            //声明Base2类对象
    Derived derived;                        //声明Derived类对象

    fun(&base1);                            //用Base1对象的指针调用fun函数
    fun(&base2);                            //用Base2对象的指针调用fun函数
    fun(&derived);                          //用Derived对象的指针调用fun函数

    return 0;
}
```

运行结果:

```
Base1::display()
Base1::display()
Base1::display()
```

4. 派生类的构造和析构函数

7.4.1 构造函数

基类的构造函数不被继承，派生类中需要声明自己的构造函数。

定义构造函数时，只需要对本类中新增成员进行初始化，对继承来的基类成员的初始化，自动调用基类构造函数完成。

派生类的构造函数需要给基类的构造函数传递参数

单一继承时的构造函数

派生类名::派生类名(基类所需的形参，本类成员所需的形参):基类名(参数表), 本类成员初始化列表

```
{
        //其他初始化;
};
```


单一继承时的构造函数举例

```
#include<iostream>
using namecpace std;

class B {
public:
    B();
    B(int i);
    ~B();
    void print() const;
private:
    int b;
};
```

单一继承时的构造函数举例(续)

```
B::B() {
    b=0;
    cout << "B's default constructor called." << endl;
}
B::B(int i) {
    b=i;
    cout << "B's constructor called." << endl;
}
B::~B() {
    cout << "B's destructor called." << endl;
}
void B::print() const {
    cout << b << endl;
}
```

单一继承时的构造函数举例(续)

```
class C: public B {
public:
    C();
    C(int i, int j);
    ~C();
    void print() const;
private:
    int c;
};
C::C() {
    c = 0;
    cout << "C's default constructor called." << endl;
}
C::C(int i,int j): B(i), c(j){
    cout << "C's constructor called." << endl;
}
```

单一继承时的构造函数举例(续)

```
C::~C() {
    cout << "C's destructor called." << endl;
}
void C::print() const {
    B::print();
    cout << c << endl;
}

int main() {
    C obj(5, 6);
    obj.print();
    return 0;
}
```


多继承时的构造函数

派生类名::派生类名(参数表):基类名1(基类1初始化参数表), 基类名2(基类2初始化参数表), ...基类名n(基类n初始化参数表), 本类成员初始化列表

```
{
    //其他初始化;
};
```

派生类与基类的构造函数

当基类中声明有**缺省构造函数**或未声明构造函数时，派生类构造函数可以不向基类构造函数传递参数，也可以不声明，构造派生类的对象时，基类的缺省构造函数将被调用。

当需要执行基类中**带形参的构造函数**来初始化基类数据时，派生类构造函数应在初始化列表中为基类构造函数提供参数。

多继承且有内嵌对象时的构造函数

派生类名::派生类名(形参表):基类名1(参数), 基类名2(参数), ...基类名n(参数), 本类对象成员和基本类型成员初始化列表

{

//其他初始化

};

构造函数的执行顺序

① 调用基类构造函数，调用顺序按照它们被继承时声明的顺序（从左向右）。

② 对本类成员初始化列表中的基本类型成员和对象成员进行初始化，初始化顺序按照它们在类中声明的顺序。对象成员初始化是自动调用对象所属类的构造函数完成的。

③ 执行派生类的构造函数体中的内容。

例7-4 派生类构造函数举例

```
#include <iostream>
using namespace std;
class Base1 {      //基类Base1，构造函数有参数
public:
    Base1(int i) { cout << "Constructing Base1 " << i << endl; }
};

class Base2 {      //基类Base2，构造函数有参数
public:
    Base2(int j) { cout << "Constructing Base2 " << j << endl; }
};

class Base3 {      //基类Base3，构造函数无参数
public:
    Base3() { cout << "Constructing Base3 *" << endl; }
};
```

例7-4 (续)

```
class Derived: public Base2, public Base1, public Base3 {
//派生新类Derived，注意基类名的顺序
public:                                    //派生类的公有成员
    Derived(int a, int b, int c, int d): Base1(a), member2(d), member1(c),
     Base2(b)
     { }
     //注意基类名的个数与顺序，//注意成员对象名的个数与顺序
private:                                   //派生类的私有成员对象
     Base1 member1;
     Base2 member2;
     Base3 member3;
};

int main() {
     Derived obj(1, 2, 3, 4);
     return 0;
}
```

运行结果：

```
constructing Base2 2
constructing Base1 1
constructing Base3 *
constructing Base1 3
constructing Base2 4
constructing Base3 *
```

7.4.2 复制构造函数

若建立派生类对象时没有编写复制构造函数，编译器会生成一个隐含的复制构造函数，该函数先调用基类的复制构造函数，再为派生类新增的成员对象执行拷贝。

若编写派生类的复制构造函数，则需要为基类相应的复制构造函数传递参数。

例如:

```
C::C(const C &c1): B(c1) {…}
```

//c1为什么不是B类引用？类型兼容规则

7.4.3 析构函数

析构函数也不被继承，派生类自行声明

声明方法与一般（无继承关系时）类的析构函数相同。

不需要显式地调用基类的析构函数，系统会自动隐式调用。

析构函数的调用次序与构造函数相反。

例7-5 派生类析构函数举例

```
#include <iostream>
using namespace std;
class Base1 {      //基类Base1，构造函数有参数
public:
    Base1(int i) { cout << "Constructing Base1 " << i << endl; }
    ~Base1() { cout << "Destructing Base1" << endl; }
};

class Base2 {      //基类Base2，构造函数有参数
public:
    Base2(int j) { cout << "Constructing Base2 " << j << endl; }
    ~Base2() { cout << "Destructing Base2" << endl; }
};

class Base3 {      //基类Base3，构造函数无参数
public:
    Base3() { cout << "Constructing Base3 *" << endl; }
    ~Base3() { cout << "Destructing Base3" << endl; }
};
```

例7-5 (续)

```
class Derived: public Base2, public Base1, public Base3 {
//派生新类Derived，注意基类名的顺序
public:                          //派生类的公有成员
    Derived(int a, int b, int c, int d): Base1(a), member2(d), member1(c),
     Base2(b) { }
     //注意基类名的个数与顺序，
     //注意成员对象名的个数与顺序
private:                         //派生类的私有成员对象
     Base1 member1;
     Base2 member2;
     Base3 member3;
};

int main() {
     Derived obj(1, 2, 3, 4);
     return 0;
}
```

运行结果：

```
Constructing Base2 2
Constructing Base1 1
Constructing Base3 *
Constructing Base1 3
Constructing Base2 4
Constructing Base3 *
Destructing Base3
Destructing Base2
Destructing Base1
Destructing Base3
Destructing Base1
Destructing Base2
```

7.4.4 delete构造函数

delete用来禁止默认构造函数或删除复制构造函数阻止拷贝（第4章）
类的继承中，基类中删除的构造函数，派生类中也是删除状态。

```
class Base {
public:
    Base() = default;

    Base(string _info) : info(std::move(_info)) {}

    Base(Base &) = delete; //删除复制构造函数
    Base(Base &&) = delete;//删除移动构造函数
private:
    string info;
};
```

```
class Derived : public Base {

};

Derived d1; //正确，合成了默认构造函数
Derived d2(d1); //错误，删除复制构造函数
Derived d3(std::move(d1)); //错误，删除移动构造函数
```

5. 派生类成员的标识和访问

在派生类中，成员按访问属性分为：

① 不可访问的成员：从基类私有成员继承而来

② 私有成员：新增或从基类继承而来

③ 保护成员：新增或从基类继承而来

④ 公有成员：对外接口

7.5.1 作用域分辨符

当派生类与基类中有相同成员时:

若未特别限定，则通过派生类对象使用的是派生类中的同名成员。

如要通过派生类对象访问基类中被隐藏的同名成员，应使用基类名和作用域分辨符（::）来限定。

例7-6 多继承同名隐藏举例

```
#include <iostream>
using namespace std;
class Base1 {                                          //定义基类Base1
public:
    int var;
    void fun() { cout << "Member of Base1" << endl; }
};
class Base2 {                                          //定义基类Base2
public:
    int var;
    void fun() { cout << "Member of Base2" << endl; }
};
class Derived: public Base1, public Base2 {            //定义派生类Derived
public:
    int var;                                           //同名数据成员
    void fun() { cout << "Member of Derived" << endl; }    //同名函数成员
};
```

例7-6（续）

```
int main() {
    Derived d;
    Derived *p = &d;

    d.var = 1;              //对象名.成员名标识
    d.fun();                //访问Derived类成员

    d.Base1::var = 2;       //作用域操作符标识
    d.Base1::fun();         //访问Base1基类成员

    p->Base2::var = 3;      //作用域操作符标识
    p->Base2::fun();        //访问Base2基类成员

    return 0;
}
```

二义性问题

当派生类从多个基类派生，而这些基类又从同一个基类派生，则在访问此共同基类中的成员时，将产生二义性——采用虚基类来解决。

在多继承时，基类与派生类之间，或基类之间出现同名成员时，将出现访问时的二义性（不确定性）——采用虚函数（参见第8章）或同名隐藏来解决。

二义性问题举例

```
class A {
public:
    void  f();
};
class B {
public:
    void f();
    void g()
};
```

```
class C: public A, public B {
public:
    void g();
    void h();
};
```

如果定义：C c1;

则 c1.f() 具有二义性

而 c1.g() 无二义性（同名隐藏）

解决方法一：用类名来限定

c1.A::f()　或　c1.B::f()

解决方法二：同名隐藏

在C 中声明一个同名成员函数f()，f()再根据需要调用

A::f()　或　B::f()

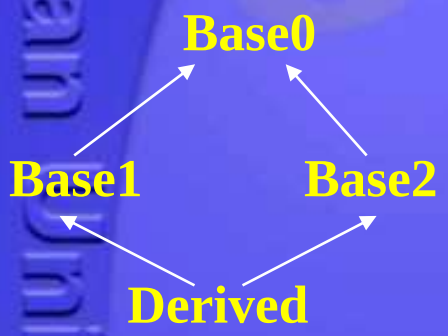
例7-7 多继承同名隐藏举例

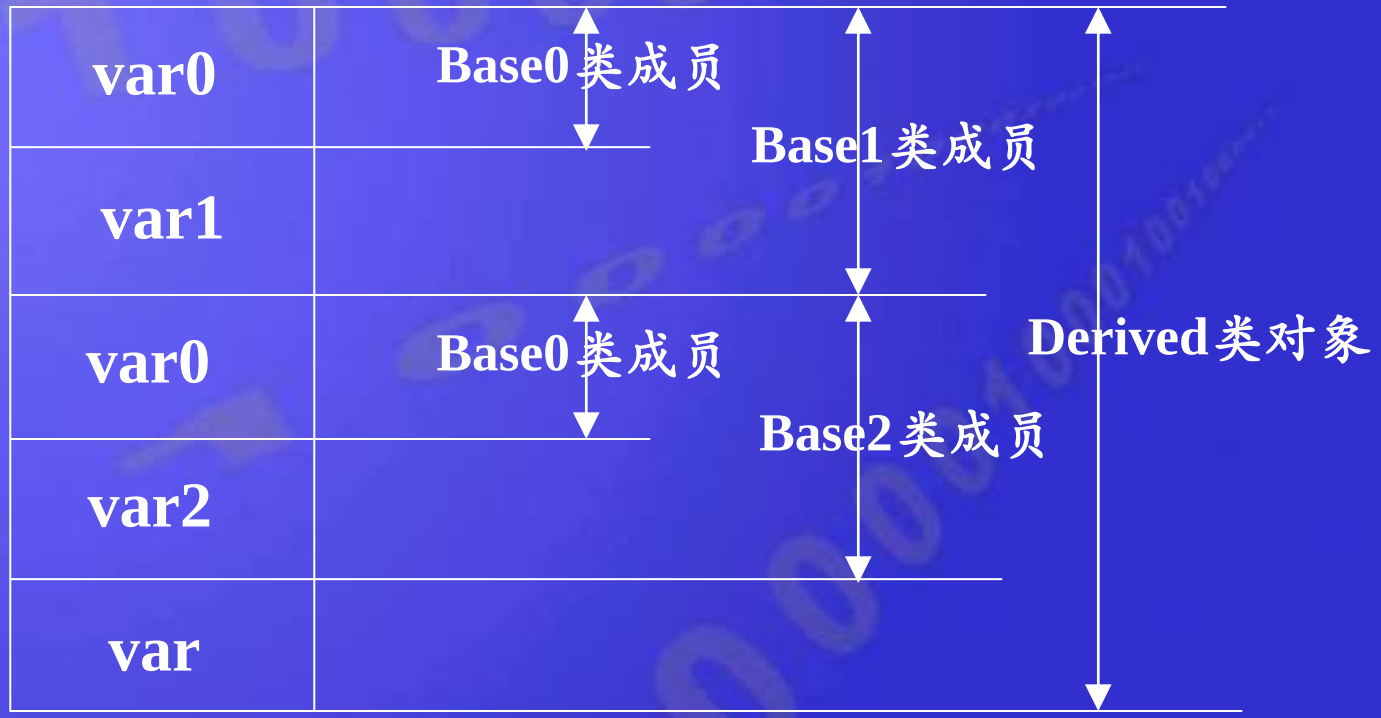
```
//7_7.cpp
#include <iostream>
using namespace std;
class Base0 {                          //定义基类Base0
public:
    int var0;
    void fun0() { cout << "Member of Base0" << endl; }
};
class Base1: public Base0 {            //定义派生类Base1
public:                                //新增外部接口
    int var1;
};
class Base2: public Base0 {            //定义派生类Base2
public:  //新增外部接口
    int var2;
};
```

例7-7（续）

```
class Derived: public Base1, public Base2 {//定义派生类Derived
public:                                     //新增外部接口
    int var;
    void fun() { cout << "Member of Derived" << endl; }
};

int main() {                                //程序主函数
    Derived d;                              //定义Derived类对象d

    d.Base1::var0 = 2;                      //使用直接基类
    d.Base1::fun0();
    d.Base2::var0 = 3;                      //使用直接基类
    d.Base2::fun0();

    return 0;
}
```

派生类C的对象的存储结构示意图：





有二义性：

```
Derived d;
d.var0
d.Base0::var0
```

无二义性：

```
d.Base1::var0
d.Base2::var0
```

问题：冗余以及因冗余而导致的不一致性

7.5.2 虚基类

虚基类的引入

- ✓ 用于有共同基类的场合

声明

- ✓ 以virtual修饰说明基类
  例： **class B1:virtual public B**

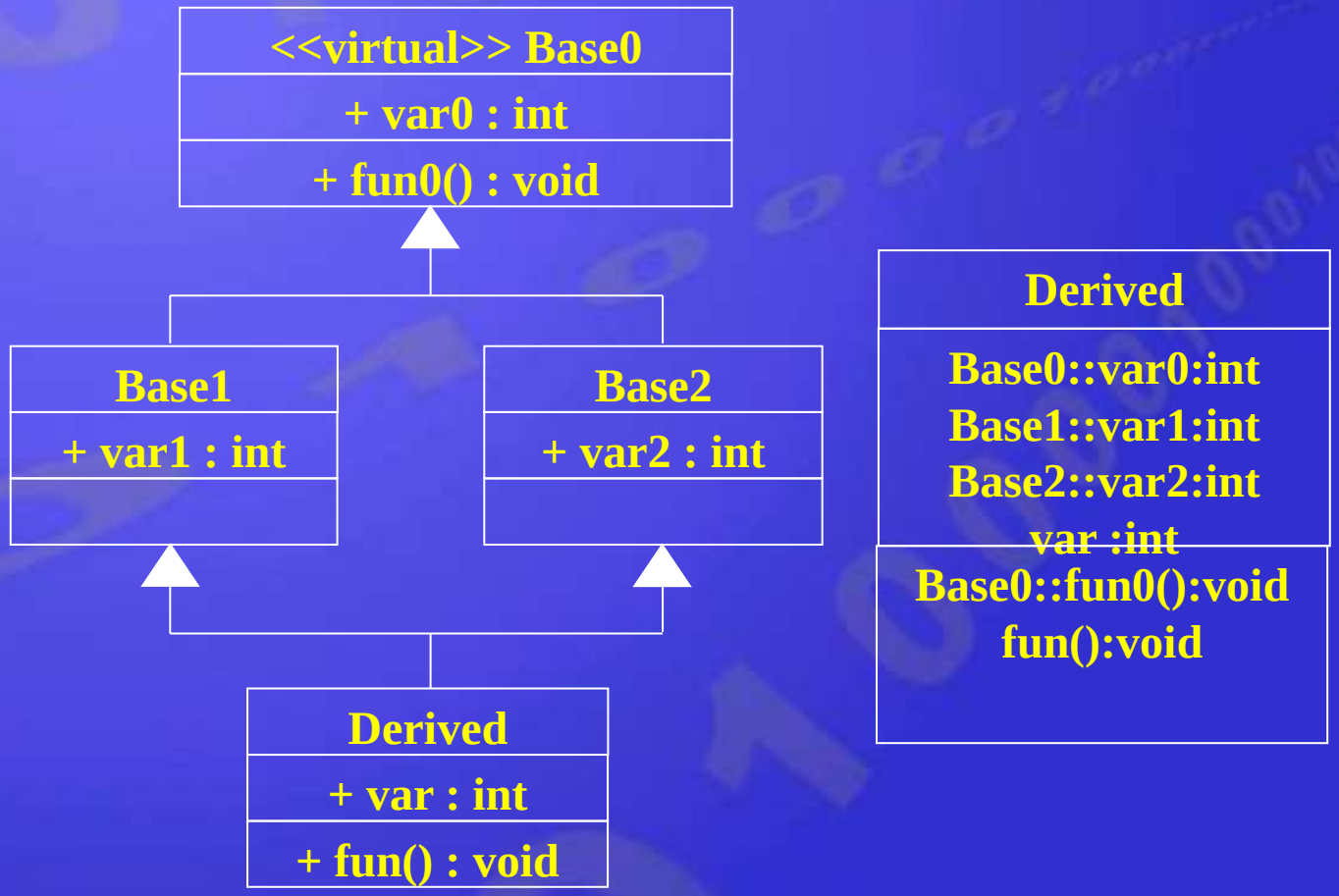
作用

- ✓ 主要用来解决多继承时可能发生的对同一基类继承多次而产生的二义性问题.
- ✓ 为最远的派生类提供唯一的基类成员，而不重复产生多次拷贝

注意:

- ✓ 在第一级继承时就要将共同基类设计为虚基类。

例7-8 虚基类举例


<<virtual>> Base0
+ var0 : int
+ fun0() : void
Base1
+ var1 : int
Base2
+ var2 : int
Derived
+ var : int
+ fun() : void
Derived
Base0::var0:int
Base1::var1:int
Base2::var2:int
var :int
Base0::fun0():void
fun():void

例7-8 (续)

```
#include <iostream>
using namespace std;
class Base0 {                                   //定义基类Base0
public:
    int var0;
    void fun0() { cout << "Member of Base0" << endl; }
};
class Base1: virtual public Base0 {             //定义派生类Base1
public:                                         //新增外部接口
    int var1;
};
class Base2: virtual public Base0 {             //定义派生类Base2
public:                                         //新增外部接口
    int var2;
};
```

例7-8 (续)

```
class Derived: public Base1, public Base2 {
//定义派生类Derived
public:                          //新增外部接口
    int var;
    void fun() {
        cout << "Member of Derived" << endl;
    }
};

int main() {                      //程序主函数
    Derived d;                    //定义Derived类对象d
    d.var0 = 2;                   //直接访问虚基类的数据成员
    d.fun0();                     //直接访问虚基类的函数成员
    return 0;
}
```

7.5.3 虚基类及其派生类构造函数

建立对象时所指定的类称为最（远）派生类。

虚基类的成员是由最派生类的构造函数通过调用虚基类的构造函数进行初始化的。

在整个继承结构中，直接或间接继承虚基类的所有派生类，都必须在构造函数的成员初始化表中给出对虚基类的构造函数的调用。如果未列出，则表示调用该虚基类的默认构造函数。

在建立对象时，只有最派生类的构造函数调用虚基类的构造函数，该派生类的其他基类对虚基类构造函数的调用被忽略。

有虚基类时的构造函数举例

```
#include <iostream>
using namespace std;

class Base0 {                                   //定义基类Base0
public:
    Base0(int var) : var0(var) { }
    int var0;
    void fun0() { cout << "Member of Base0" << endl; }
};
class Base1: virtual public Base0 {             //定义派生类Base1
public:                                         //新增外部接口
    Base1(int var) : Base0(var) { }
    int var1;
};
class Base2: virtual public Base0 {             //定义派生类Base2
public:                                         //新增外部接口
    Base2(int var) : Base0(var) { }
    int var2;
};
```

有虚基类时的构造函数举例(续)

```
class Derived: public Base1, public Base2 {
//定义派生类Derived
public:                          //新增外部接口
    Derived(int var) : Base0(var), Base1(var), Base2(var) { }
    int var;
    void fun() { cout << "Member of Derived" << endl; }
};

int main() {                     //程序主函数
    Derived d(1);                //定义Derived类对象d
    d.var0 = 2;                  //直接访问虚基类的数据成员
    d.fun0();                    //直接访问虚基类的函数成员
    return 0;
}
```

6. 程序实例

——用高斯消去法解线性方程组

7.6.1 算法基本原理

算法原理求解Ax = b

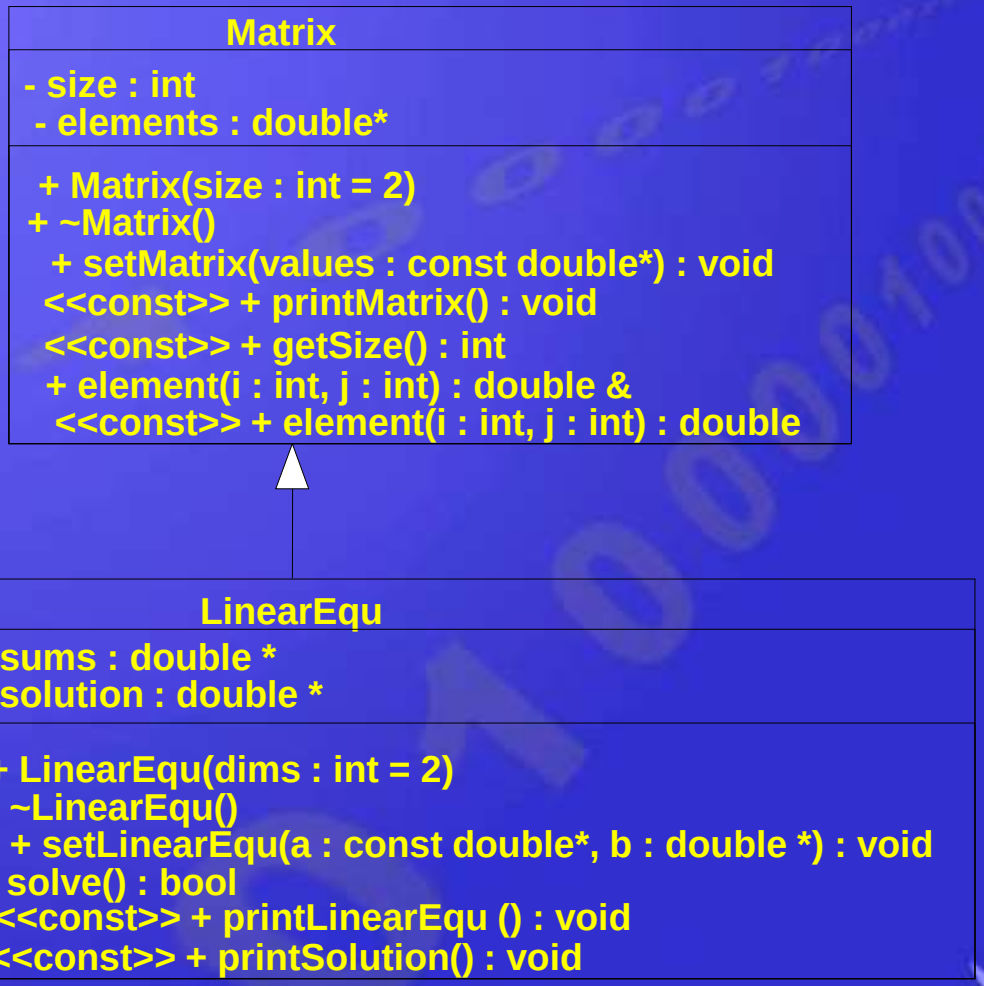
$$\mathbf{A}=\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n} \\ a_{10} & a_{11} & \cdots & a_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix} \quad \mathbf{x}=\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} \quad \mathbf{b}=\begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix}$$

经过推算得到计算公式:

$$x_{n-1}=b_{n-1}/a_{n-1,n-1}$$

$$x_i=b_i-\sum_{j=i+1}^{n-1}a_{ij}x_j, \qquad i=n-2,\cdots,1,0$$

7.6.2 程序设计分析


Matrix
- size : int
- elements : double*
+ Matrix(size : int = 2)
+ ~Matrix()
+ setMatrix(values : const double*) : void
<<const>> + printMatrix() : void
<<const>> + getSize() : int
+ element(i : int, j : int) : double &
<<const>> + element(i : int, j : int) : double
LinearEqu
- sums : double *
- solution : double *
+ LinearEqu(dims : int = 2)
+ ~LinearEqu()
+ setLinearEqu(a : const double*, b : double *) : void
+ solve() : bool
<<const>> + printLinearEqu () : void
<<const>> + printSolution() : void

7.6.3 源程序及说明

例7-9 全选主元高斯消去法解线性方程组，整个程序分为5个独立的文档：

- Matrix.h、LinearEqu.h文件中包括矩阵类Matrix和线性方程组类LinearEqu的定义
- Matrix.cpp、LinearEqu.cpp文件中包括这两个类的成员函数实现文件；
- lequmain.cpp文件包括程序的主函数，主函数中定义了一个类LinearEqu的对象，通过这个对象求解一个四元线性方程组。

例7-9（续）

```
//Matrix.h 文件一，Matrix类定义
#ifndef _MATRIX_H
#define _MATRIX_H
class Matrix {                                          //基类Matrix定义
public:                                                 //外部接口
    Matrix(int size = 2);                               //构造函数
    ~Matrix();                                          //析构函数
    void setMatrix(const double *values);//矩阵赋初值
    void printMatrix() const;                           //显示矩阵
    int getSize() const { return size; }                //得到矩阵大小
    double &element(int i, int j)
  { return elements[i * size + j]; }
    double element(int i, int j) const
  { return elements[i * size + j]; }
private:                                                //保护数据成员
    int size;                                           //矩阵的大小
    double *elements;                                   //矩阵存放数组首地址
};
#endif //_MATRIX_H
```

```
//LinearEqu.h  文件二，LinearEqu类定义
#ifndef _LINEAR_EQU_H
#define _LINEAR_EQU_H
#include "Matrix.h"
class LinearEqu: public Matrix { //公有派生类LinearEqu定义
public:                          //外部接口
    LinearEqu(int size = 2);     //构造函数
    ~LinearEqu();                //析构函数
    //方程赋值
    void setLinearEqu(const double *a, const double *b);
    bool solve();                //全选主元高斯消去法求解方程
    void printLinearEqu() const; //显示方程
    void printSolution() const;  //显示方程的解
private:                         //私有数据
    double *sums;                //方程右端项
    double *solution;            //方程的解
};
#endif //_LINEAREQU_H
```

例7-9（续）

例7-9（续）

```
//Matrix.cpp  文件三，Matrix类实现
#include "Matrix.h"//包含类的定义头文件
#include <iostream>
using namespace std;
void Matrix::setMatrix(const double *values) {      //设置矩阵
    for (int i = 0; i < size * size; i++)
            elements[i] = values[i];                //矩阵成员赋初值
}
//矩阵Matrix类的构造函数
Matrix::Matrix(int size/* = 2 */) : size(size) {
    elements = new double[size * size];             //动态内存分配
}
Matrix::~Matrix() {                  //矩阵Matrix类的析构函数
    delete[] elements;                              //内存释放
}
void Matrix::printMatrix() const {                  //显示矩阵的元素
    cout << "The Matrix is:" << endl;
    for(int i = 0; i < size; i++) {
            for(int j = 0; j < size; j++)
                    cout << element(i, j) << " ";
            cout << endl;      }
}
```

```
//LinearEqu.cpp  文件四，LinearEqu类实现
#include "LinearEqu.h"                    //包含类的定义头文件
#include <iostream>
#include <cmath>
using namespace std;
LinearEqu::LinearEqu(int size/* = 2 */) : Matrix(size) {
//用size调用基类构造函数
    sums = new double[size];              //动态内存分配
    solution = new double[size];
}
LinearEqu::~LinearEqu()        {          //派生类LinearEqu的析构函数
    delete[] sums;                        //释放内存
    delete[] solution;
    //会自动调用基类析构函数
}
void LinearEqu::setLinearEqu(const double *a, const double *b) {
//设置线性方程组
    setMatrix(a);                         //调用基类函数
    for(int i = 0; i < getSize(); i++)
        sums[i] = b[i];
}
```

例7-9（续）

```
void LinearEqu::printLinearEqu() const {//显示线性方程组
    cout << "The Line eqution is:" << endl;
    for (int i = 0; i < getSize(); i++) {
        for (int j = 0; j < getSize(); j++)
            cout << element(i, j) << " ";
            cout << "    " << sums[i] << endl;
        }
}

void LinearEqu::printSolution() const {//输出方程的解
    cout << "The Result is: " << endl;
    for (int i = 0; i < getSize(); i++)
        cout << "x[" << i << "] = " << solution[i] << endl;
}

inline void swap(double &v1, double &v2) {//交换两个实数
    double temp = v1;
    v1 = v2;
    v2 = temp;
}
```

例7-9（续）

```
bool LinearEqu::solve() {                    //全选主元素高斯消去法求解方程
    int *js=new int[getSize()];              //存储主元素所在列号的数组
    for (int k = 0; k < getSize() - 1; k++)  {//选主元素
        int is;                              //主元素所在行号
        double max = 0;                      //所有元素的最大值
        for (int i = k; i < getSize(); i++)
            for (int j = k; j < getSize(); j++) {
                double t = fabs(element(i, j));
                if (t > max) {
                    max = t;
                    js[k] = j;
                    is = i;
                }
            }
        if (max == 0) {
            delete[] js;
            return false;
        }
```

例7-9（续）

```
        else {//通过行、列交换，把主元素交换到第k行第k列
                if (js[k] != k)
                        for (int i = 0; i < getSize(); i++)
                          swap(element(i, k), element(i, js[k]));
                if (is != k) {
                        for (int j = k; j < getSize(); j++)
                           swap(element(k, j), element(is, j));
                        swap(sums[k], sums[is]);
                }
        }
        //消去过程
        double major = element(k, k);
        for (int j = k + 1; j < getSize(); j++)
                element(k, j) /= major;
        sums[k] /= major;
        for (int i = k + 1; i < getSize(); i++) {
                for (int j = k + 1; j < getSize(); j++)
                  element(i, j) -= element(i, k) * element(k, j);
                sums[i] -= element(i, k) * sums[k];
        }
}
```

例7-9（续）

```
    //判断剩下的一个元素是否等于0
    double d = element(getSize() - 1, getSize() - 1);
    if (fabs(d) < 1e-15) {
            delete[] js;
            return false;
    }
    //回代过程
    solution[getSize() - 1] = sums[getSize() - 1] / d;
    for (int i = getSize() - 2; i >= 0; i--) {
            double t = 0.0;
            for (int j = i + 1; j <= getSize() - 1; j++)
                    t += element(i, j) * solution[j];
            solution[i] = sums[i] - t;
    }
    js[getSize() - 1] = getSize() - 1;
    for (int k = getSize() - 1; k >= 0; k--)
            if (js[k] != k) swap(solution[k], solution[js[k]]);
    delete[] js;
    return true;
}
```

例7-9（续）

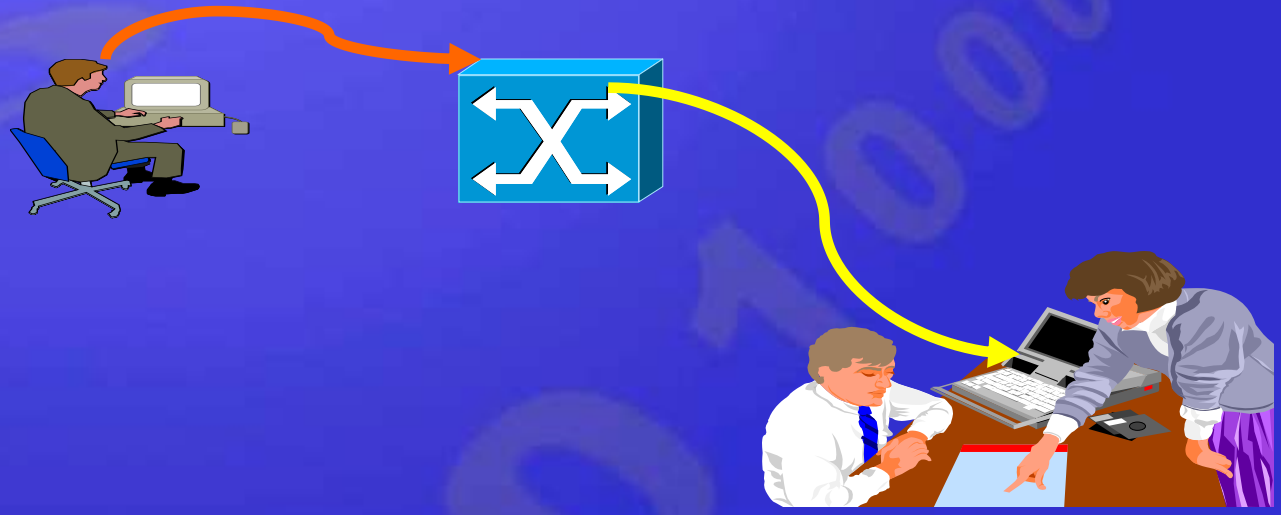
```
//7_9.cpp  文件五，主函数
#include "LinearEqu.h" //类定义头文件
#include <iostream>
using namespace std;

int main() {        //主函数
    double a[]=   { //方程系数矩阵
            0.2368, 0.2471, 0.2568, 1.2671,                 //第一行
            0.1968, 0.2071, 1.2168, 0.2271,                 //第二行
            0.1581, 1.1675, 0.1768, 0.1871,                 //第三行
            1.1161, 0.1254, 0.1397, 0.1490 };   //第四行
    double b[] = { 1.8471, 1.7471, 1.6471, 1.5471 };//方程右端项
    LinearEqu equ(4);     //定义一个四元方程组对象
    equ.setLinearEqu(a,b);          //设置方程组
    equ.printLinearEqu();           //输出方程组
    if (equ.solve())            //求解方程组
            equ.printSolution();        //输出方程组的解
    else
            cout<<"Fail"<<endl;
    return 0;
}
```

例7-9（续）

例7-9（续）

```
运行结果为：
The Line eqution is:        //方程组
0.2368 0.2471 0.2568 1.2671      1.8471
0.1968 0.2071 1.2168 0.2271      1.7471
0.1581 1.1675 0.1768 0.1871      1.6471
1.1161 0.1254 0.1397 0.149      1.5471
The Result is:                   //方程组的解
X[0]=1.04058
X[1]=0.987051
X[2]=0.93504
X[3]=0.881282
```

8. 深度探索



7.8.1 组合与继承

组合与继承：通过已有类来构造新类的两种基本方式

组合：B类中存在一个A类型的内嵌对象

- ✓ 有一个（has-a）关系：表明每个B类型对象“有一个”A类型对象
- ✓ A类型对象与B类型对象是部分与整体关系
- ✓ B类型的接口不会直接作为A类型的接口

“has-a”举例

```
class Engine {     //发动机类
public:
    void work();  //发动机运转
    ……
};
class Wheel {      //轮子类
public:
    void roll();     //轮子转动
    ……
};
class Automobile {          //汽车类
public:
    void move();  //汽车移动
private:
    Engine engine;          //汽车引
    擎
    Wheel wheels[4];//4个车轮
    ……
};
```

意义

- ✓ 一辆汽车有一个发动机
- ✓ 一辆汽车有四个轮子

接口

- ✓ 作为整体的汽车不再具备发动机的运转功能，和轮子的转动功能，但通过将这些功能的整合，具有了自己的功能——移动

公有继承的意义

公有继承：A类是B类的公有基类

- ✓ 是一个（is-a）关系：表明每个B类型对象“是一个”A类型对象
- ✓ A类型对象与B类型对象是一般与特殊关系
  - ✓ 回顾类的兼容性原则：在需要基类对象的任何地方，都可以使用公有派生类的对象来替代
- ✓ B类型对象包括A类型的全部接口

“is-a”举例

```
class Truck: public Automobile{
//卡车
public:
    void load(…);          //装货
    void dump(…);          //卸货
private:
    ……
};

class Pumper: public Automobile {
//消防车
public:
    void water(); //喷水
private:
    ……
};
```

意义

- ✓ 卡车是汽车
- ✓ 消防车是汽车

接口

- ✓ 卡车和消防车具有汽车的通用功能（移动）
- ✓ 它们还各自具有自己的功能（卡车：装货、卸货；消防车：喷水）

7.8.2 派生类对象的内存布局

派生类对象的内存布局

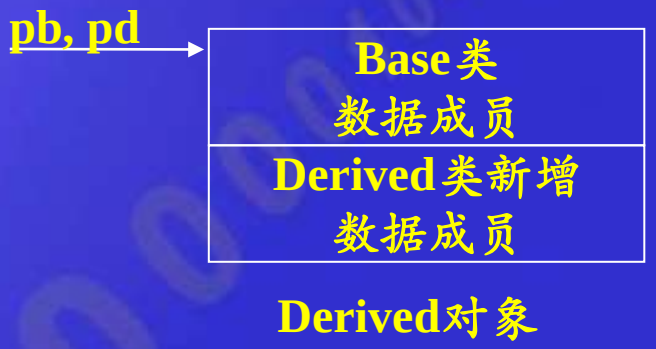
- ✓ 因编译器而异
- ✓ 内存布局应使类型兼容规则便于实现
  - ✓ 一个基类指针，无论其指向基类对象，还是派生类对象，通过它来访问一个基类中定义的数据成员，都可以用相同的步骤

不同情况下的内存布局

- ✓ 单继承：基类数据在前，派生类新增数据在后
- ✓ 多继承：各基类数据按顺序在前，派生类新增数据在后
- ✓ 虚继承：需要增加指针，间接访虚基类数据

单继承情形

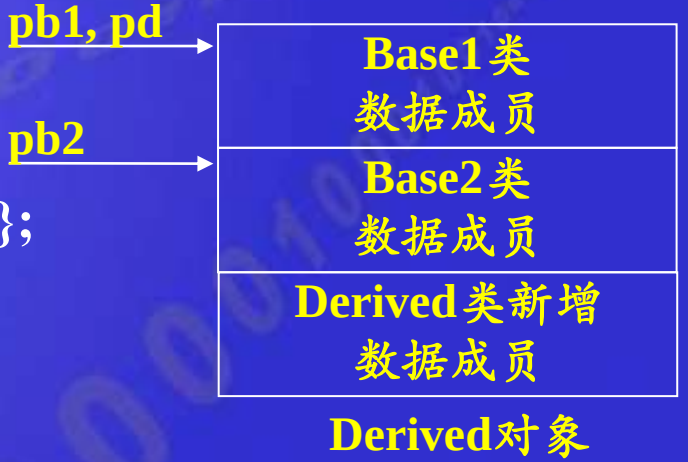
```
class Base { … };
class Derived: public Base { … };

Derived *pd = new Derived();
Base *pb = pd;
```



Derived类型指针pd转换为Base类型指针时，地址不需要改变

多继承情形

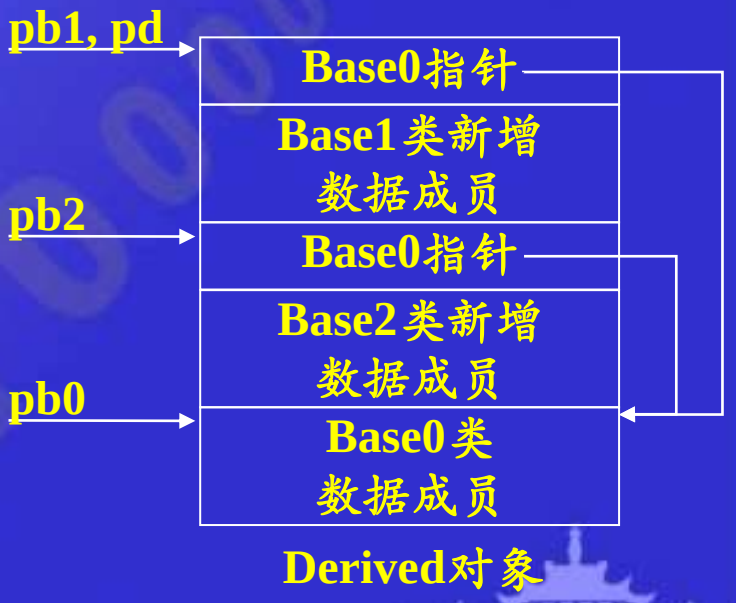
```
class Base1 { … };
class Base2 { … };
class Derived: public Base1, public Base2 { … };

Derived *pd = new Derived();
Base1 *pb1 = pd;
Base2 *pb2 = pd;
```



Derived类型指针pd转换为Base2类型指针时，原地址需要增加一个偏移量

虚拟继承情形

```
class Base0 { … };
class Base1: virtual public Base0 { … };
class Base2: virtual public Base0 { … };
class Derived: public Base1, public Base2 { … };

Derived *pd = new Derived();
Base1 *pb1 = pd;
Base2 *pb2 = pd;
Base0 *pb0 = pb1;
```



通过指针间接访问虚基类的数据成员

7.8.3 基类向派生的转换及其安全性问题

基类向派生类的转换

- ✓ 基类指针可以转换为派生类指针
- ✓ 基类引用可以转换为派生类引用
- ✓ 需要用static_cast显式转换

例：

```
Base *pb = new Derived();
Derived *pd = static_cast<Derived *>(pd);
Derived d;
Base &rb = d;
Derived &rd = static_cast<Derived &>(rb);
```

类型转换时的注意事项(1)

基类对象一般无法被显式转换为派生类对象

- ✓ 对象到对象的转换，需要调用构造函数创建新的对象
- ✓ 派生类的复制构造函数无法接受基类对象作为参数

执行基类向派生类的转换时，一定要确保被转换的指针和引用所指向或引用的对象符合转换的目的类型：

- ✓ 对于Derived *pd = static_cast<Derived *>(pb);
- ✓ 一定要保证pb所指向的对象具有Derived类型，或者是Derived类型的派生类。

类型转换时的注意事项(2)

如果A类型是B类型的虚拟基类，A类型指针无法通过static_cast隐含转换为B类型的指针

- ✓ 可以结合虚继承情况下的对象内存布局，思考为什么不允许这种转换

void指针参加的转换，可能导致不可预期的后果:

- ✓ 例：（Base2是Derived的第二个公共基类）

```
Derived *pd = new Derived();
void *pv = pd;   //将Derived指针转换为void指针
Base2 *pb = static_cast<Base2 *>(pv);
```

- ✓ 转换后pb与pd有相同的地址，而正常的转换下应有一个偏移量
- ✓ 结论：有void指针参与的转换，兼容性规则不适用

更安全更灵活的基类向派生类转换方式——dynamic_cast，将在下一讲介绍

本讲小结

本讲主要知识点

主要内容

1. 类的继承
2. 类成员的访问控制
3. 单继承与多继承
4. 派生类的构造和析构函数
5. 类成员的标识与访问

达到的目标：理解类的继承关系，学会使用继承关系实现代码的重用。

第7讲 课后练习

作业：

7-5、定义一个基类Shape，在此基础上派生出Rectangle和Circle，二者带有getArea()函数计算对象的面积。使用Rectangle类创建一个派生类Square()。

7-11、定义一个基类BaseClass，从它派生出DerivedClass，BaseClass有成员函数fn1()、fn2()，DerivedClass也有成员函数fn1()、fn2()，在主函数中声明一个DerivedClass的对象，分别 用DerivedClass的对象以及BaseClass和DerivedClass的指针来调用fn1()、fn2()，观察运行结果。

本 讲 结 束

有问题吗?