

# 继承、派生与多态性

北京邮电大学

网络与交换技术国家重点实验室

宽带网研究中心

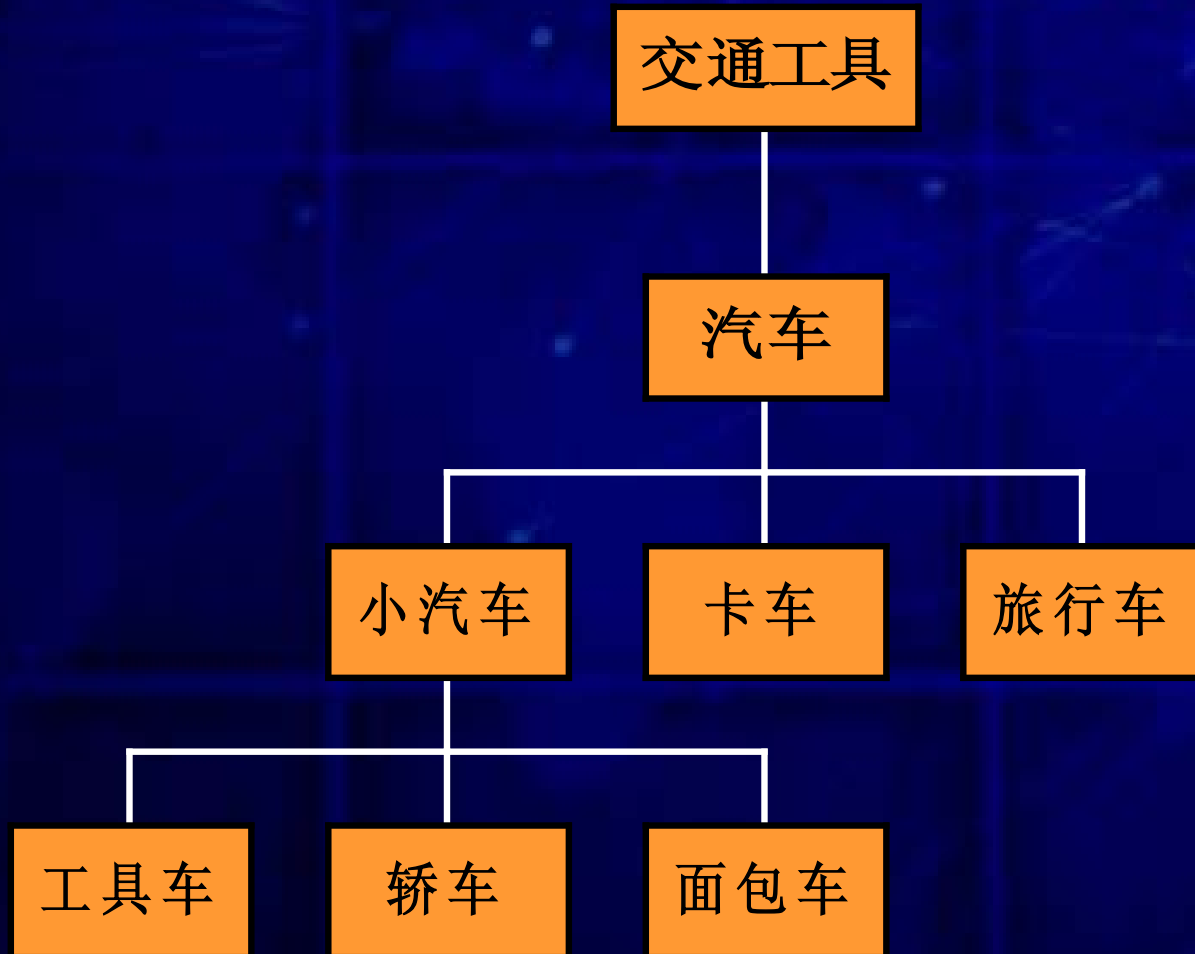
# 本章主要内容

- 类的继承与派生
- 类成员的访问控制
- 简单继承与多重继承
- 派生类的构造、析构函数
- 类成员的标识与访问

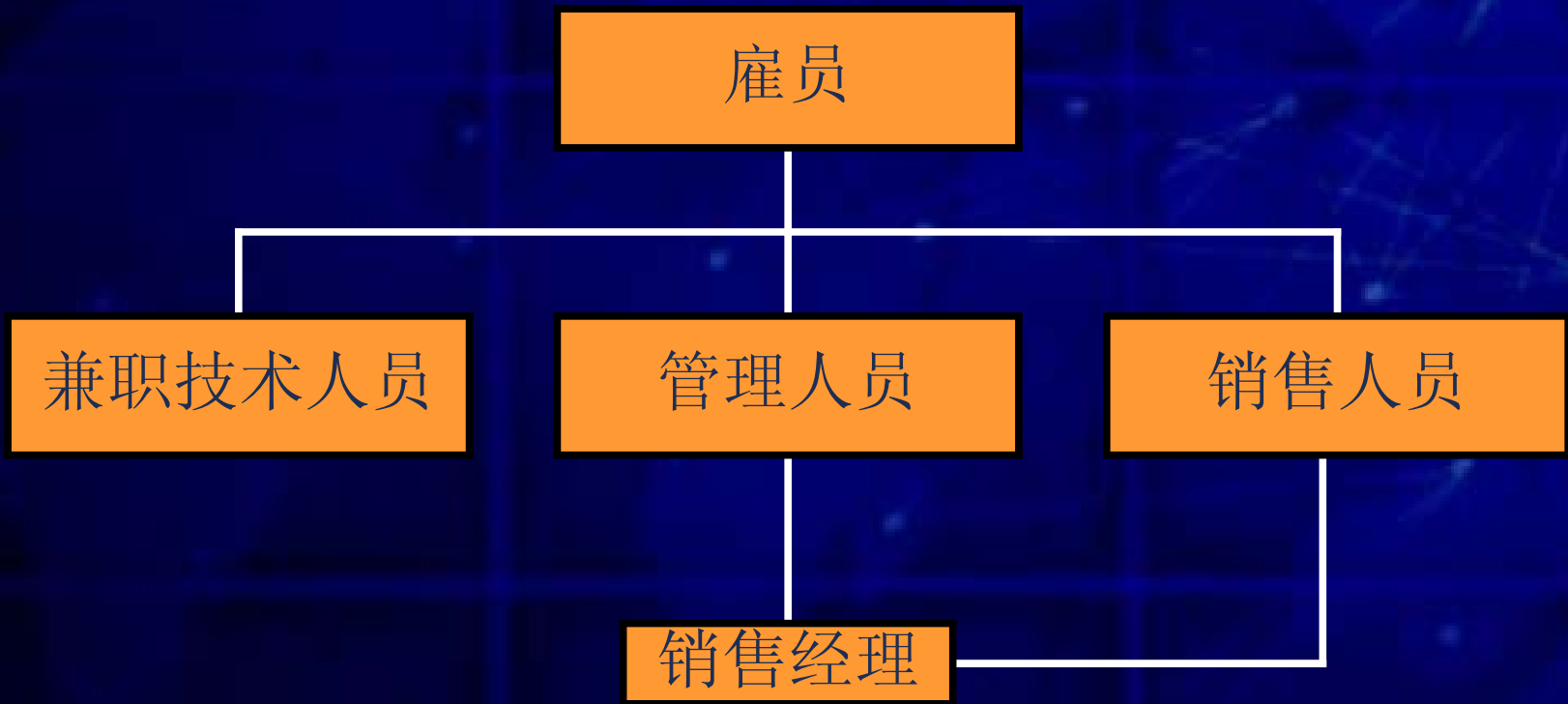
# 类的继承与派生

- 保持已有类的特性而构造新类的过程称为继承。
- 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。
- 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
  - ❑ 派生类将自动继承基类的所有特性（属性和方法）
  - ❑ 派生类可以定义新的特性（属性和方法）
  - ❑ 派生类可以对继承的方法定义新实现
- 派生是一个从抽象到具体的过程

# 现实世界对象的从属与继承关系



# 现实世界对象的从属与继承关系



# 继承与派生的目的

## ➤ 继承的目的

- ❑ 实现代码重用，派生问题可以继承原有问题中具有普遍性的解决方法
- ❑ 最大程度利用原有的成果

## ➤ 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，具体解决新问题

## ➤ 继承和派生通过 C++ 的类来实现

# 派生类的声明

```
class 派生类名: 继承方式 基类名  
{  
    成员声明;  
}
```

# 继承方式

## ➤ 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 public
- 私有继承 private (缺省)
- 保护继承 protected



# 公有继承(public)

- 基类的成员的访问属性 (public/protected/ private) 在派生类中访问属性保持不变
- 派生类中的成员函数可以直接访问基类中的 public 和 protected 成员，但不能访问基类的 private 成员。
- 对外部来说，通过派生类的对象只能访问基类的 public 成员

# 例：公有继承

```
class CBase
{
    public:
        int x;
    protected:
        int y;
    private:
        int z;
    ...
};

class CDerived:
    public CBase
{
    public:
        int a;
        void f(int i);
    ...
};
```

```
void CDerived::f(int i)
{
    x = i;        // ok
    y = i;        // ok
    z = i;        // error
}

int main()
{
    CDerived d;

    d.x = 1;      // ok
    d.y = 2;      // error
    d.z = 3;      // error
    d.a = 4;      // ok
    d.f(5);       // ok
    ...
};
```

# 私有继承(private)

- 基类的 public/protected/private 成员在派生类中的访问属性都变成 private
- 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
- 对外部来说，通过派生类的对象不能直接访问基类中的任何成员

# 例：私有继承

```
class CBase
{
    public:
        int x;
    protected:
        int y;
    private:
        int z;
    ...
};

class CDerived:
    private CBase
{
    public:
        int a;
        void f(int i);
    ...
};
```

```
void CDerived::f(int i)
{
    x = i;        // ok
    y = i;        // ok
    z = i;        // error
}

int main()
{
    CDerived d;

    d.x = 1;      // error
    d.y = 2;      // error
    d.z = 3;      // error
    d.a = 4;      // ok
    d.f(5);       // ok
    ...
};
```

# 保护继承(protected)

- 基类的 public/protected成员在派生类中的访问属性都变成 protected
- 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
- 对外部来说，通过派生类的对象不能直接访问基类中的任何成员
- protected 成员的作用
  - ❑ 对类的外部，protected成员与 private 成员的性质相同
  - ❑ 对于其派生类来说，protected成员与 public 成员的性质相同
  - ❑ 既实现了对外数据隐藏，又方便继承，实现代码重用

# 类型兼容规则

➤ 从某个基类派生出的public 派生类的对象，可以作为基类对象来使用

□ 例如： 人类 ~~== 派生 ==~~ ➡ 学生类

学生对象 完全可以当作 人类对象 来使用

□ 但是，相反的方向则禁止

➤ 具体表现在：

□ 派生类的对象可以被赋值给基类对象

□ 派生类的对象可以用来初始化基类的引用

□ 指向基类的指针也可以用来指向派生类对象

□ 注：通过基类对象名、指针只能访问基类的成员

# 例：类型兼容

```
class CB0    //基类
{
public:
    void display() {...}
    void func_b0() {...}
    ...
};

class CB1 : public CB0
{
public:
    void display() {...}
    void func_b1() {...}
    ...
};

class CD : public CB1
{
public:
    void display() {...}
    void func_d() {...}
    ...
};
```

```
int main()
{
    CD  d, *pD  = &d;
    CB0 *pB0 = &d;
    CB1 *pB1 = &d;

    pD->display();
    pD->func_b0();
    pD->func_b1();
    pD->func_d();

    pB1->func_b0();
    pB1->display();
    pB1->func_b1();
    pB1->func_d();    // error

    pB0->display();
    pB0->func_b0();
    pB0->func_b1();    // error
    pB0->func_d();    // error
}
```

# 单继承与多重继承

## ➤ 单继承

- ❑ 派生类只从一个基类派生。

## ➤ 多重继承

- ❑ 派生类从多个基类派生
- ❑ 派生类继承多个基类的所有的属性和行为成员



# 多重继承时派生类的声明

```
class 派生类名: 继承方式1 基类名1,  
               继承方式2 基类名2,  
               ...  
{  
    成员声明;  
};
```

注意：每一个“继承方式”，只用于限制对紧随其后之基类的继承

# 例：多重继承

```
class A
{
    public:
        void setA(int);
        void showA();
        ...;
};
class B
{
    public:
        void setB(int);
        void showB();
        ...
};
class C : public A, private B
{
    public:
        void setC(int a, int b);
        void showC();
        ...
};
```

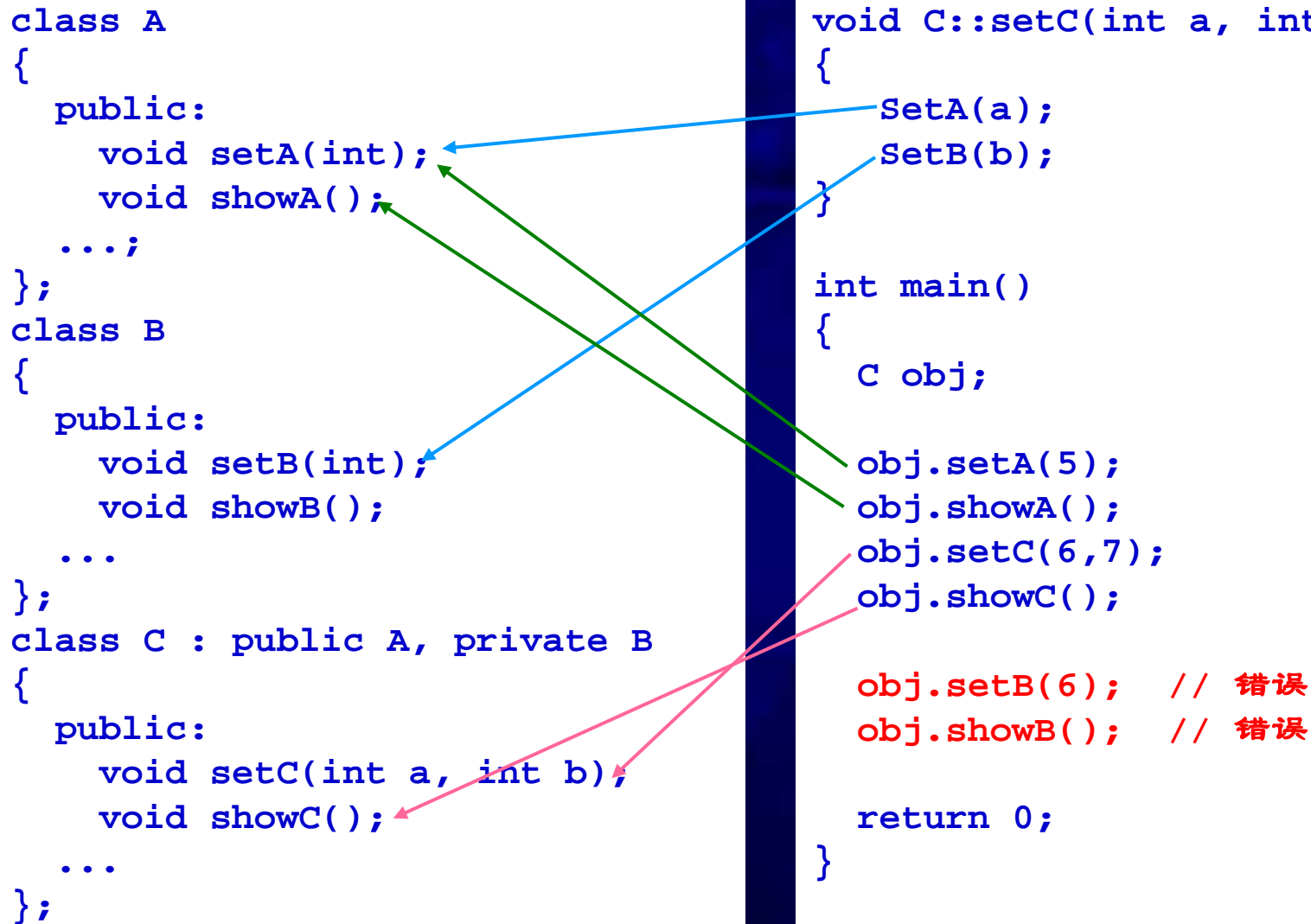
```
void C::setC(int a, int b)
{
    SetA(a);
    SetB(b);
}

int main()
{
    C obj;

    obj.setA(5);
    obj.showA();
    obj.setC(6,7);
    obj.showC();

    obj.setB(6); // 错误
    obj.showB(); // 错误

    return 0;
}
```



# 派生类的构造函数

- 基类的构造函数不被直接继承成为派生类的构造函数，派生类中需要声明自己的构造函数

- 在执行派生类构造函数时，基类构造函数将被自动执行

- 派生类的构造函数

- 需要对本类中新增成员进行初始化

- 对继承的基类成员的初始化，可以在自动调用基类构造函数的时候，由基类构造函数完成

- 派生类构造函数也可以对继承的基类成员重新初始化

- 派生类的构造函数需要向基类构造函数传递参数

- 编写派生类的拷贝构造函数时，也可能需要向基类拷贝构造函数传递参数

# 派生类的构造函数

派生类名::派生类名(参数表):

基类名1(参数表1), 基类名2(参数表2), ...

成员1(参数表), 成员2(参数表), ...

{

成员的初始化;

}

# 例：派生类的构造函数

```
class A
{
    public:
        A()      {...};
        A(int x) {...};
        ...;
};

class B
{
    public:
        B()      {...};
        B(int y) {...};
        ...
};

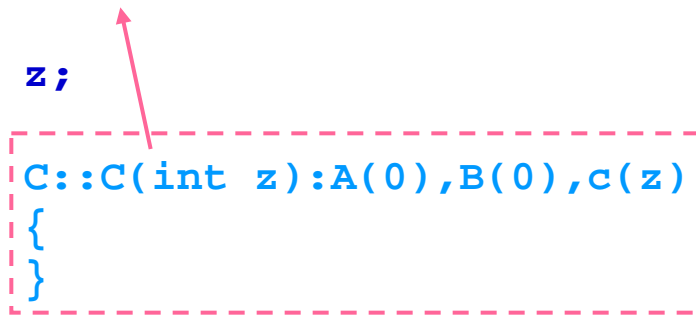
class C : public A, public B
{
    public:
        C();
        C(int z);
        C(int x, int y, int z);
    private:
        int c;
};
```

```
C::C()      // 调用A、B的默认构造函数
{
    c = 0;
}

C::C(int z):A(0),B(0)
{
    c = z;
}

C::C(int z):A(0),B(0),c(z)
{
}

C::C(int x, int y, int z):
    A(x),B(y),c(z)
{
}
```



# 构造函数的调用次序

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. 调用成员对象的构造函数，调用顺序按照它们在类中声明的顺序。
  - 对简单对象（如 int 等）的初始化也相当于调用其构造函数
3. 派生类的构造函数体中的内容。

# 派生类的析构函数

- 基类的析构函数也不被派生类继承，派生类自行声明，但派生类的析构函数会自动调用基类的析构函数
- 声明方法与一般类的析构函数相同。
- 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
  - 派生类、成员对象、基类



# 同名隐藏规则

当派生类与基类中有相同成员时：

- 若未加限定，则通过派生类对象访问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覆盖的同名成员，应使用基类名限定。



# 例：同名隐藏

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

```
class B1 //基类B1
{
```

```
    public:
        int nV;
        void func() {...};
        ...
};
```

```
class D1: public B1 // 派生类
```

```
{
    public:
        int nV; //同名数据成员
        void func(); //同名函数成员
};
```

```
void D1::func()
{
```

```
    B1::nV = 1;
```

```
    B1::func();
```

```
    nV = 0;
```

```
    func();
}
```

```
int main()
{
```

```
    D1 d;
```

```
    d.nV = -1;
```

```
    d.func();
```

```
    d.B1::nV = 1;
```

```
    d.B1::func();
```

```
    ...
```

```
};
```

# 类继承对象的内存映像

## ➤ 派生类继承了父类的所有方法和属性

父  
类  
对  
象

父类的 private 属性
父类的 protected 属性
父类的 public 属性
父类的 private 方法
父类的 protected 方法
父类的 public 方法

父类的 private 属性
父类的 protected 属性
父类的 public 属性
父类的 private 方法
父类的 protected 方法
父类的 public 方法

- ❑ 派生类中包含父类的所有成员

父类的内存映像

- ❑ 父类的成员成为了派生类的成员

- ❑ 在派生类中，对父类成员的访问权限与其自身成员一样，受到访问控制权限的限制

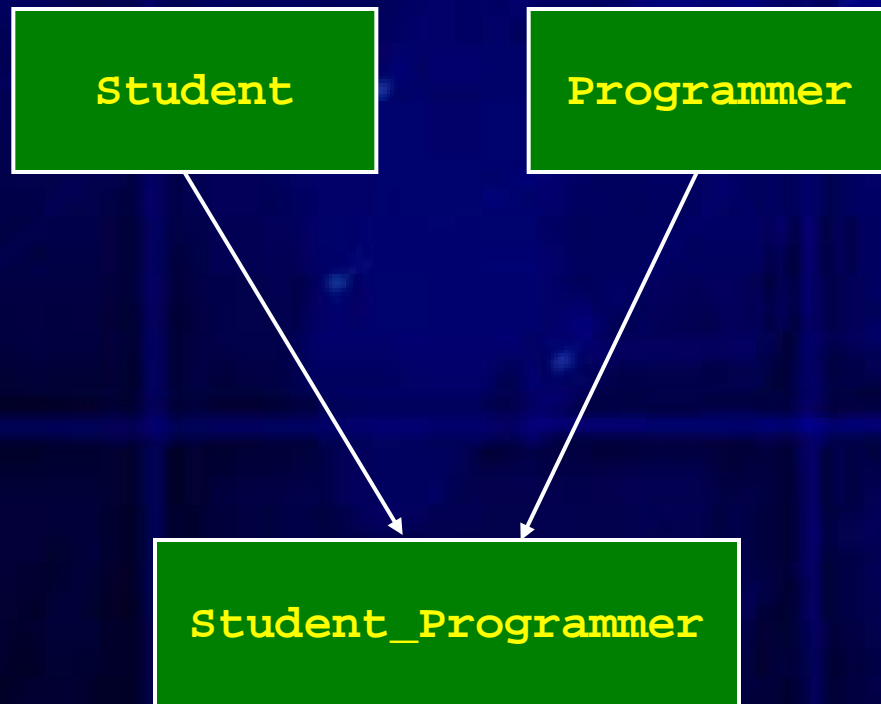
派生类的 private 属性
派生类的 protected 属性
派生类的 public 属性
派生类的 private 方法
派生类的 protected 方法
派生类的 public 方法

派生类对象

# 多重继承

## ➤ 派生类从两个以上的父类继承

□ 派生类将继承和拥有所有父类的属性和方法



# 多重继承

## ➤ 派生类从两个以上的父类继承

❑ 派生类将继承和拥有所有父类的属性和方法

```
class Student
{
protected:
    char * pName;
    ...
};

class Programmer
{
protected:
    char * pName;
    ...
};

class Student_Programmer :
public Student,
public Programmer
{
    ...
};
```

student 的属性和方法

pName

programmer 的属性和方法

pName

Student\_Programmer  
自身的属性和方法

# 多重继承

## ➤ 可能存在的二义性

```
class Student
{
protected:
    char * pName;
    ...
};
class Programmer
{
protected:
    char * pName;
    ...
};
class Student_Programmer :
public Student,
public Programmer
{
public:
    char * GetName();
    ...
};
```

```
char * Student_Programmer::GetName()
{
    return pName;
}
```

二义性：  
编译器不知道到底访问  
哪一个 pName 成员

# 多重继承

## ➤ 可能存在的二义性

```
class Student
{
protected:
    char * pName;
    ...
};
class Programmer
{
protected:
    char * pName;
    ...
};
class Student_Programmer :
public Student,
public Programmer
{
public:
    char * GetName();
    ...
};
```

```
char * Student_Programmer::GetName()
{
    return Student::pName;
// 或者:
// return Programmer::pName;
}
```

必须加类限定符“::”，以消除二义性。

当不存在二义性时，则类限定符不是必需的。

在访问存在二义性的方法时也必须使用限定符来进行修饰。

# 二义性问题

- 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），二义性将导致编译器报错
- 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。

# 多态性



# 多态性的概念

- 多态性是面向对象程序设计的重要特征之一。
- 多态性是指发出同样的消息被不同类型的对象接收时有可能导致完全不同的行为。
- 多态的实现：
  - ❑ 函数重载/运算符重载
  - ❑ 虚函数

# 运算符重载 (overloading)

```
class complex //复数类声明
{
public:
    complex(double r=0.0,double i=0.0) //构造函数
    {
        real=r;
        imag=i;
    }
    void display()
    {
        cout << "(" << real << "," << imag << ")"; //显示复数的值
    }
private:
    double real;
    double imag;
};
```

用 "+"、"-"、"<<" 能够实现复数的加减运算、输出吗？

答案：可以，重载 "+"、"-"、"<<" 运算符

# 运算符重载的实质

## ➤ 运算符重载与函数重载本质相同

- 只不过重载的是 C++ 中保留的运算符  $+ - * /$

## ➤ 运算符重载是对已有的运算符赋予多重含义

## ➤ 必要性

- C++ 中预定义的运算符其运算对象只能是基本数据类型，而不适用于用户自定义类型（如类）

## ➤ 实现机制

- 将指定的运算表达式转化为对运算符函数的调用，运算对象转化为运算符函数的输入参数

- 编译系统对重载运算符的选择，遵循与函数重载选择相同的原则

# 规则 and 限制

➤ 可以重载C++中除下列运算符外的所有运算符：

. \* :: ?:

➤ 只能重载C++语言中已有的运算符，不可臆造新的。

➤ 不改变原运算符的优先级和结合性。

➤ 不能改变操作数个数。

➤ 经重载的运算符，其操作数中至少应该有一个是自定义类型，class 或 struct

# 操作符重载的两种形式

## ➤ 将操作符重载为类成员函数

- ❑ 操作符函数是类成员函数

## ➤ 重载为友元函数

- ❑ 操作符函数是类的友元函数

- ❑ 若操作符函数只访问类中的 public 成员，则无须是友元

# 运算符重载函数

## ➤ 声明形式

函数类型 operator 运算符 (形参)

{

.....

}

## ➤ 重载为类成员函数时

参数个数 = 原操作数个数 - 1 (后置++、--除外)

## ➤ 重载为友元函数时

参数个数 = 原操作数个数

且至少应该有一个是自定义类型的形参

# 运算符成员函数的设计

## ➤ 二元运算符（以+为例）

$a + b$

其中  $a$  为  $A$  类对象， $b$  可以使各种类型

## ➤ + 被重载为 $A$ 类的一个成员函数 $A::operator+(Type\ b)$

□ 运算符成员函数与普通成员函数没有本质差别

## ➤ 表达式 $a + b$ 相当于

$a.A::operator+(b)$

# 例：复数类的 +- 重载

```
class complex //复数类声明
{
private:
    double real, imag;
public:
    complex(double r=0.0,double i=0.0) {...}; //构造函数
    void display() {...}; //显示复数的值
    complex operator+(complex c2)
    {
        complex c;
        c.real = real + c2.real;
        c.imag = imag + c2.imag;
        return c;
    };
    complex operator-(complex c2)
    {
        return complex(real-c2.real, imag-c2.imag);
    };
};
```

$c1+c2$      $c1-c2$



# 运算符成员函数的设计

## ➤ 前置一元运算符 (+a -a ++a --a 等)

❑ 以 -a 为例, a 是 A 类的对象

operator- 被重载为 A 类的成员函数, 且没有参数

❑ 表达式 -a 相当于 a.A::operator-()

❑ 同样: --a 相当于 a.A::operator--()

++a 相当于 a.A::operator++()

```
class complex //复数类声明
{
public:
    ...
    complex operator-()
    {
        return complex(-real, -imag);
    };
};
```

# 运算符成员函数

## ➤ 后置一元运算符 ++和--, 需要特殊处理

- ❑ 以 `a--` 为例, `a` 是 `A` 类的对象  
`operator--` 被重载为 `A` 类的成员函数, 有一个整型参数 (但未用)
- ❑ 表达式 `a--` 相当于 `a.A::operator--(0)`
- ❑ 同样: `a++` 相当于 `a.A::operator++(0)`

```
class complex //复数类声明
{
public:
    ...
    complex & operator--()      // --c
    {
        real--;    imag--;
        return *this;
    };
    complex operator--(int)     // c--
    {
        complex temp = *this;
        --*this;
        return temp;
    };
};
```

# 重载赋值类运算符=(+=, -=, ...)

```
class Name{
    const char* s;
    //...
};
Class Table{
    Name* p;
    int sz;
    Table(int s=20) {
        p= new Name[sz=s];
    }
    Table& operator=(const Table&);
    //...
};
```

Table x, y, z;  
若要实现 x=y=z;

```
Table& Table::operator=(const Table&t)
{
    if(this !=&t) //当心自赋值: t=t
    {
        delete[]p; //删除老元素
        p=new Name[sz=t.sz];
        for(int i=0; i<sz;i++)
            p[i]=t.p[i];
    }
    return *this;
}
```

# 运算符友元函数

- 如果需要重载一个运算符，使之能够用于操作某类对象的私有成员，可以将此运算符重载为该类的友元函数。
- 函数的形参代表依自左至右次序排列的各操作数。
- 后置单目运算符 ++ 和 -- 的重载函数，需要特殊处理，形参列表中要增加一个 int，但不必写形参名。

# 运算符友元函数

## ➤ 二元运算符 (+/\*-等) 重载

$a + b$  等同于 `operator+(a, b)`

## ➤ 前置一元运算符 (+ - ++ -- 等) 重载

$++a$  等同于 `operator++(a)`

## ➤ 后置单目运算符 (++和--)重载

$a++$  等同于 `operator++(a, 0)`

# 例：复数类的重载

```
class complex //复数类声明
{
    friend complex operator+(complex c1,complex c2);
    friend complex operator-(complex c1,complex c2);
    friend complex operator++(complex& c1);          // ++c
    friend complex operator++(complex& c1,int);      // c++
private:
    double real, imag;
public:
    ...
};

complex operator+(complex c1,complex c2)
{
    c1.real += c2.real;
    c1.imag += c2.imag;
    return c1;
}
```

# 例：复数类的重载

```
complex operator-(complex c1,complex c2)
{
    c1.real -= c2.real;
    c1.imag -= c2.imag;
    return c1;
}

complex operator++(complex& c1)           // ++c1
{
    c1.real ++;
    c1.imag ++;
    return c1;
}

complex operator++(complex& c1,int)
{
    complex temp = c1;
    ++c1;
    return temp;
}
```

# 例：复数类 << 的重载

```
// 将复数 a 输出到流式IO, 表示成 (re,im) 形式输出
ostream& operator <<(ostream& Ostr, complex c)
{
    return Ostr << "(" << c.re << "," << c.im << ")";
}

int main()
{
    complex a,b;
    a.re = 10.;
    a.im = 15.;
    b.re = 25.5;
    b.im = -3.5;

    cout << "a=" << a << endl;
    cout << "b=" << b << endl;
    cout << "a+b" << a+b << endl;

    return 0;
}
```

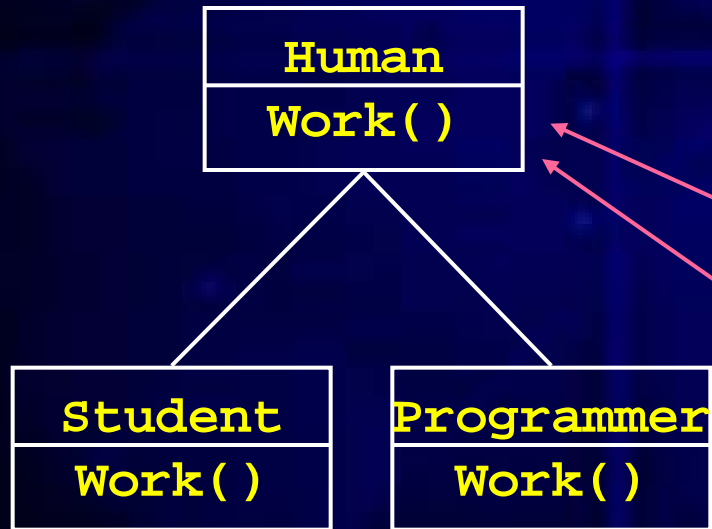


# C++ 类的多态性

## ➤ 类的多态性是面向对象的程序设计的重要支柱

- ❑ 多态性体现在，当我们向某种类型的对象发送相同消息时（即调用对象的方法），允许不同对象进行不同的操作
- ❑ 支持动态联编
  - ⊗ 在程序运行的过程中，确定具体调用对象的哪一个方法
- ❑ 不提供多态性，难以提高代码重用的效率

# 多态性的必要性



```
Student a;  
Programmer b;  
Human * p;
```

```
p = &a;  
p->Work();
```

```
p = &b;  
p->Work();
```

与自然语  
义不符

# 多态性的必要性

```
class Phone
{
    ...
public:
    int call(int iNumber);

    int dial(int Number);
    int Conversation();
    int Hangup();
    ...
};

int Phone::call(int iNumber)
{
    ...

    dial(int Number);
    Conversation();
    Hangup();

    ...
}
```

```
int Phone::dial(int Number)
{
    ...
}
int Phone::Conversation()
{
    ...
}
int Phone::Hangup()
{
    ...
}

int main()
{
    Phone my_phone;
    my_phone.call(62281342);
    ...
}
```

原有  
问题

# 多态性的必要性

```
class MobilePhone : public Phone
{
    ...
public:
    int call(int iNumber);

    int dial(int Number);
    // int Conversation(); 不用重写
    int Hangup();
    ...
};

int MobilePhone::call(int iNumber)
{
    ...

    dial(int Number);
    Conversation();
    Hangup();

    ...
}
```

这个流程虽然与父类的流程相同，  
但 `MobilePhone::call()` 也必须  
重写，否则将会使用父类实现，  
调用父类方法 `Phone::dial()`、  
`Phone::Conversation()` ...

```
int MobilePhone::dial(int Number)
{
    ... // 与父类不同
}

int MobilePhone::Hangup()
{
    ... // 与父类不同
}

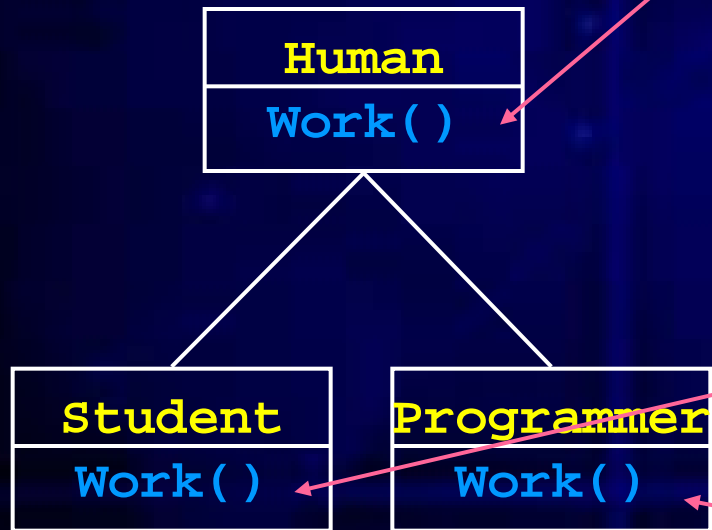
int main()
{
    MobilePhone my_phone;
    my_phone.call(13900012345);
    ...
}
```

`MobilePhone::Call()`

派生的  
新问题

# 多态性支持——虚函数

将 Work() 定义  
为虚成员函数



```
Student a;  
Programmer b;  
Human * p;
```

```
p = &a;  
a.Work();  
p->Work();
```

```
p = &b;  
b.Work();  
p->Work();
```

# 虚成员函数

- 虚函数是动态绑定的基础。
- 在调用对象虚函数时，根据对象所属的派生类，决定调用哪个函数实现（动态联编）
- 是非静态的成员函数。
- 在类的声明中，在函数原型之前写 `virtual`。
- `virtual` 只用来说明类声明中的原型，不能用在函数实现时。
- 具有继承性，基类中声明了虚函数，派生类中无论是否说明，同原型函数都自动为虚函数。
- 在派生类中，可以对基类的虚函数进行重写（覆盖），而不是重载

# 多态性支持——虚函数

```
class Phone
{
public:
    virtual int call(int iNumber);
    virtual int dial(int Number);
    virtual int Conversation();
    virtual int Hangup();
    ...
};

int Phone::call(int iNumber)
{
    ...
    dial(int Number);
    Conversation();
    Hangup();
    ...
}

int Phone::dial(int Number)
{
    ...
}

int Phone::Conversation()
{
    ...
}

int Phone::Hangup()
{
    ...
}
```

```
class MobilePhone : public Phone
{
public:
    virtual int dial(int Number);
    virtual int Hangup();
    ...
};

int MobilePhone::dial(int Number)
{
    ...
}

int MobilePhone::Hangup()
{
    ...
}
```

**只需要重写 MobilePhone 的特殊部分的代码其他都从基类中继承（最大限度的重用）**

# 多态性支持——虚函数

```
int main()
{
    MobilePhone my_phone;
    my_phone.call(13900012345);
    ...
}
```

将调用基类的 Phone::Call()

```
int Phone::call(int iNumber)
{
    ...
    dial(int Number);
    Conversation();
    Hangup();
    ...
}
```

在基类的 Phone::Call() 中将自动调用  
派生类的方法实现 MobilePhone::dialCall()  
和 MobilePhone:: Hangup()



# 静态绑定与动态绑定

## ➤ 绑定（联编）

- ❑ 程序自身彼此关联的过程，确定程序中的操作调用与执行该操作的代码间的关系。

## ➤ 静态联编

- ❑ 联编工作出现在编译阶段，用对象名或者类名来限定要调用的函数。

## ➤ 动态联编

- ❑ 联编工作在程序运行时执行，在程序运行时才确定将要调用的函数。

# 虚函数与动态联编

## ➤ 先前联编/静态联编 (Early Binding/Static Binding)

- ❑ 由编译程序在编译时确定调用特定类的特定方法
- ❑ 效率高

## ➤ 迟后联编/动态联编 (Late Binding/Dynamic Binding)

- ❑ 编译程序在编译时无法确定调用哪一个类的方法，这种情况下，需要在运行时动态地确定具体调用哪一个类的方法
- ❑ 灵活性好、高度的抽象性，但需要额外的运行开销

```
main()
{
    Phone * my_phone;
    my_phone = new MobilePhone;

    my_phone->fee();
    ...
}
```

若 `fee()` 不是虚函数，则被静态联编为调用 `Phone::fee()`

若 `fee()` 是虚函数，则需要动态联编，在运行时根据目标对象的类型，调用其相应的 `fee()`

# 抽象类

带有纯虚函数的类称为抽象类:

```
class 类名
```

```
{
```

```
    virtual 类型 函数名(参数表)=0;
```

```
    //纯虚函数, 只定义方法调用的格式
```

```
    ...
```

```
}
```

# 抽象类

## ➤ 作用

- ❑ 抽象类为抽象和设计的目的而声明，将有关的数据和行为组织在一个继承层次结构中，保证派生类具有要求的行为。
- ❑ 对于暂时无法实现的函数，可以声明为纯虚函数，留给派生类去实现。

## ➤ 注意

- ❑ 抽象类只能作为基类来使用。
- ❑ 不能声明抽象类的对象。
- ❑ 构造函数不能是虚函数，析构函数可以是虚函数。

# 例：抽象类

```
#include <iostream>
using namespace std;
class B0                                //抽象基类B0声明
{ public:                                //外部接口
    virtual void display( )=0;           //纯虚函数成员
};
class B1: public B0
{
    public:
        void display(){cout<<"B1::display()"<<endl;} //虚成员函数
};

class D1: public B1
{
    public:
        void display(){cout<<"D1::display()"<<endl;} //虚成员函数
};
```