第七章 类的继承

7.1 类的继承与派生

7.2 访问控制

7.3 类型兼容规则

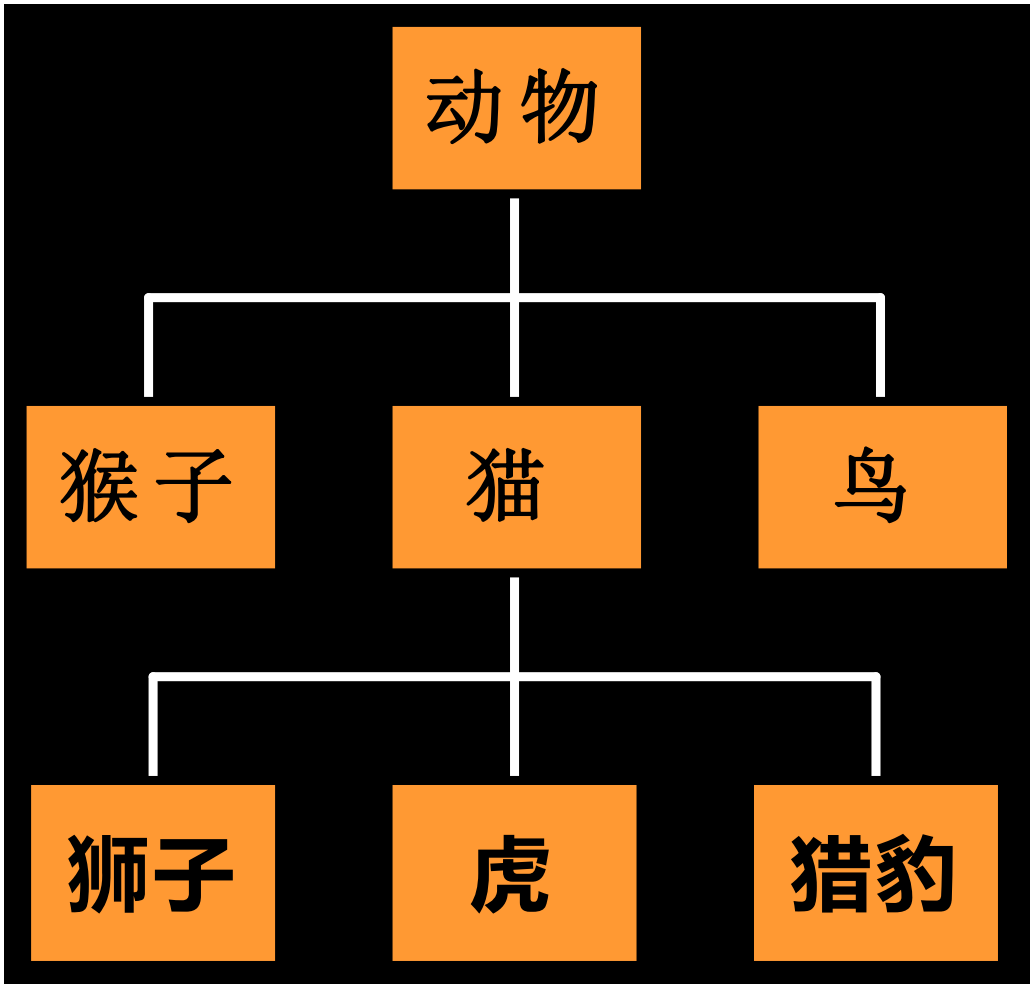
7.4 派生类的构造、析构函数

7.5 派生类成员的标识与访问

7.1 类的继承与派生

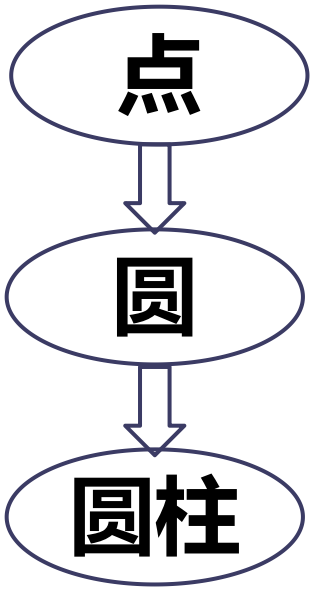
- **继承与派生**是同一过程从不同的角度来看
  - 保持已有类的特性而构造新类的过程称为**继承**。
  - 在已有类的基础上新增自己的特性而产生新类的过程称为**派生**。
- 被继承的已有类称为**基类（父类）**。
- 派生出的新类称为**派生类（子类）**。

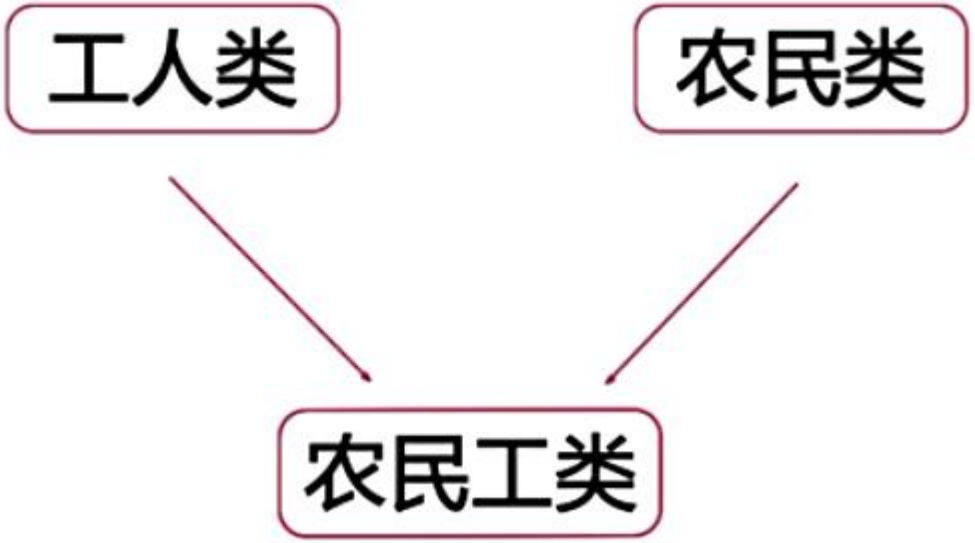
继承与派生问题举例



单继承和多继承

- 一个派生类只有一个基类，叫**单继承**
- 一个派生类有多个基类，这叫**多继承**



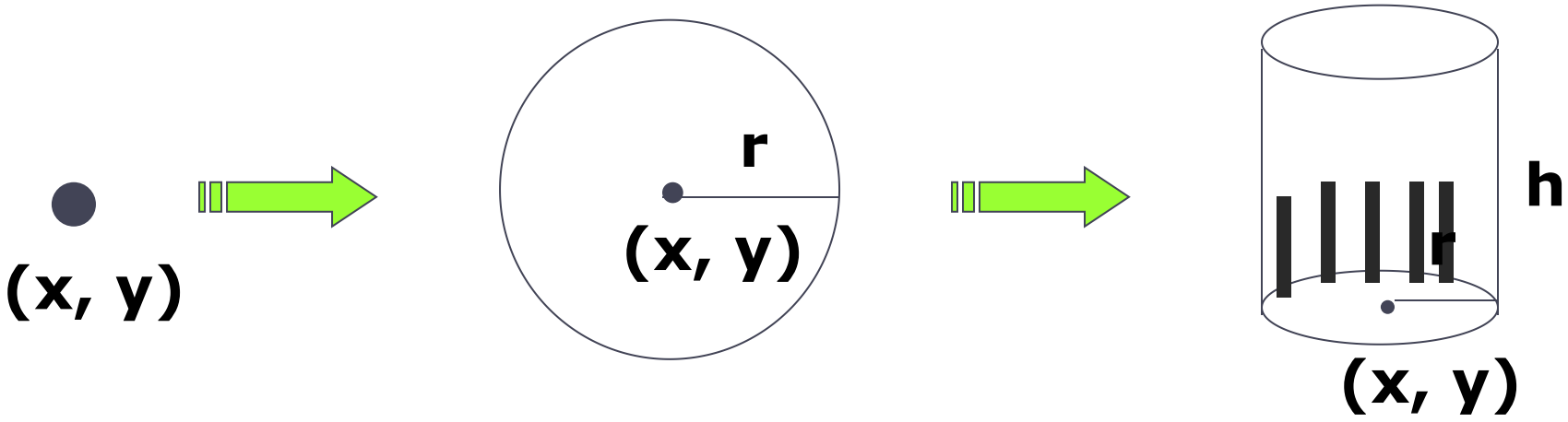


7.1.2派生类的声明

- 单继承时

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

继承方式：public protected private



Point Circle Cylinder

```
class Point{
    private:int x,y;
};
class Circle:public Point{...};
class Cylinder:public Circle{...};
```

7.1.2派生类的声明

- 多继承时

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；};

例如：

class Worker{...};

class Farmer{...};

class FarmerWorker:**public Farmer,public Worker**{...};

7.1.3 继承与派生的目的

- 继承的目的：实现代码重用。
- 派生的目的：对原有程序进行改进。

- 吸收基类成员
- 改造基类成员
- 添加新的成员

7.1.3 继承与派生的目的

- 吸收基类成员
  - 派生类包含了基类中除构造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

7.1.3 继承与派生的目的

- 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

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。
- 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

```
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 show( )
    { cout<<x<<","<<y<<endl; }
private:        //私有数据成员
    float x, y;
};
```

例7-1 (续)

```
class Circle: public Point {//派生类定义部分
public:    //新增公有函数成员
  void initCircle(float x, float y, float r1) {
        initPoint(x, y); //调用基类公有成员函数
        this->r = r1;
  }
private:   //新增私有数据成员
  float r;
};
```

例7-1 (续)

```
int main() {
  Circle c1;      //定义Circle类的对象
  c1.initCircle(2, 3, 20);//设置圆的数据
  c1.show( );   //派生类对象访问基类公有成员
  return 0;
}
```

7.2.2 私有继承(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。
- 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

```
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 show( )
    { cout<<x<<","<<y<<endl; }
private:        //私有数据成员
    float x, y;
};
```

例7-2（续）

```
class Circle: private Point {//派生类定义
public:    //新增公有函数成员
  void initCircle(float x, float y, float r1) {
        initPoint(x, y); //调用基类公有成员函数
        this->r = r1;
  }
private:  //新增私有数据成员
  float r;
};
```

例7-2 (续)

```
int main() {
  Circle c1;      //定义Circle类的对象
  c1.initCircle(2, 3, 20);//设置圆的数据
  //c1.show( );//不能访问基类公有成员
  return 0;
}
```

7.2.3 保护继承(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。
- 通过派生类的对象不能直接访问基类中的任何成员

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。
- 基类对象名、指针只能访问从基类继承的成员

例7-3 类型转换规则举例

```
class Base { //基类Base定义
public:
  void display() const {
        cout << "Base::display()" << endl;
  }
};
//公有派生类Derived定义
class Derived: public Base {
public:
  void display() const {
        cout << "Derived::display()"<<endl;
  }
};
```

例7-3 (续)

```
void fun(Base *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
  Base base;            //声明Base类对象
  Derived derived;  //声明Derived类对象
  fun(&base);         //用Base对象的指针调用函数
  fun(&derived);    //用Derived对象的指针调用函数
  return 0;
}
```

运行结果:

```
Base::display()
Base::display()
```

继承时的构造函数

- 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

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

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

单一继承时的构造函数举例 (续)

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

同学们：
想想运行结果？

运行结果：

```
B's constructor called.
C's constructor called.
5
6
C's destructor called.
B's destructor called.
```

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), ..., 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), ..., 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

```
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构造函数无参数
public:
  Base2(      ) { cout<<"Constructing Base2  *"<<endl;}
};
```

例7-4 (续)

```
//派生新类Derived，注意基类名的顺序
class Derived:public Base2,public Base1{
public:
  Derived(int a,int b):Base1(a){ c=b;}
private:
   int c;
};

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

运行结果：

```
constructing Base2 *
constructing Base1 1
```

例7-4 (续)

```
class Derived{
public:
   Derived(int a,int b):member1(a){  c=b;}
   private:
   //注意成员对象名的个数与顺序
   Base1 member1;
   Base2 member2;
    int c;
};
int main() {
   Derived obj(1, 2);
   return 0;
}
```

运行结果：

```
constructing Base1 1
constructing Base2 *
```

例7-4 (续)

```
//注意基类名的个数与顺序，
//注意成员对象名的个数与顺序
class Derived: public Base2, public Base1{
public:
  Derived(int a,int b):Base1(a),member1(b){ }
  private:
  Base1 member1;
  Base2 member2;
};
int main() {
  Derived obj(1, 2);
  return 0;
}
```

运行结果：

```
constructing Base2 *
constructing Base1 1
constructing Base1 2
constructing Base2 *
```

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) {...}
  ```

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;
}
```

例7-5 (续)

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
```

7.5.1 作用域限定

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操作符（::）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, piblic 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

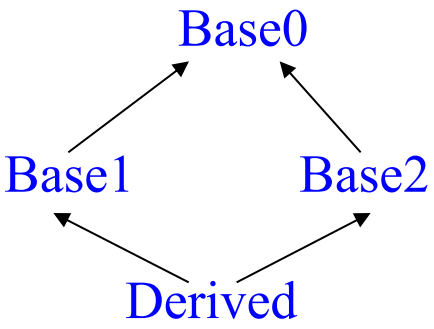
```
//7_7.cpp
#include <iostream>
using namespace std;
class Base0 { //定义基类Base0
public:
  int var0;
  void fun0() { cout << "Member of Base0" << endl; }
};
class Base1: public Base0 { //定义派生类Base1
public:       //新增外部接口
  int var1;
};
class Base2: public Base0 { //定义派生类Base2
public:       //新增外部接口
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

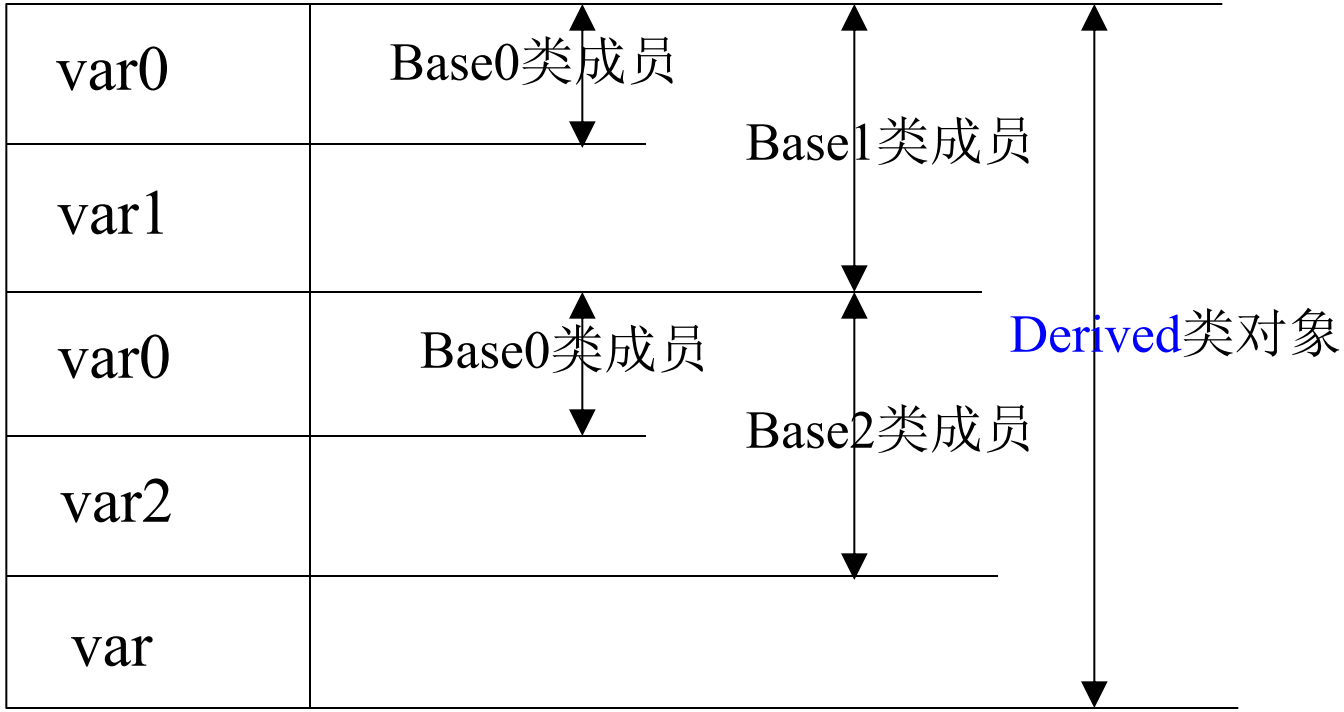
例7-7（续）

```
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
```

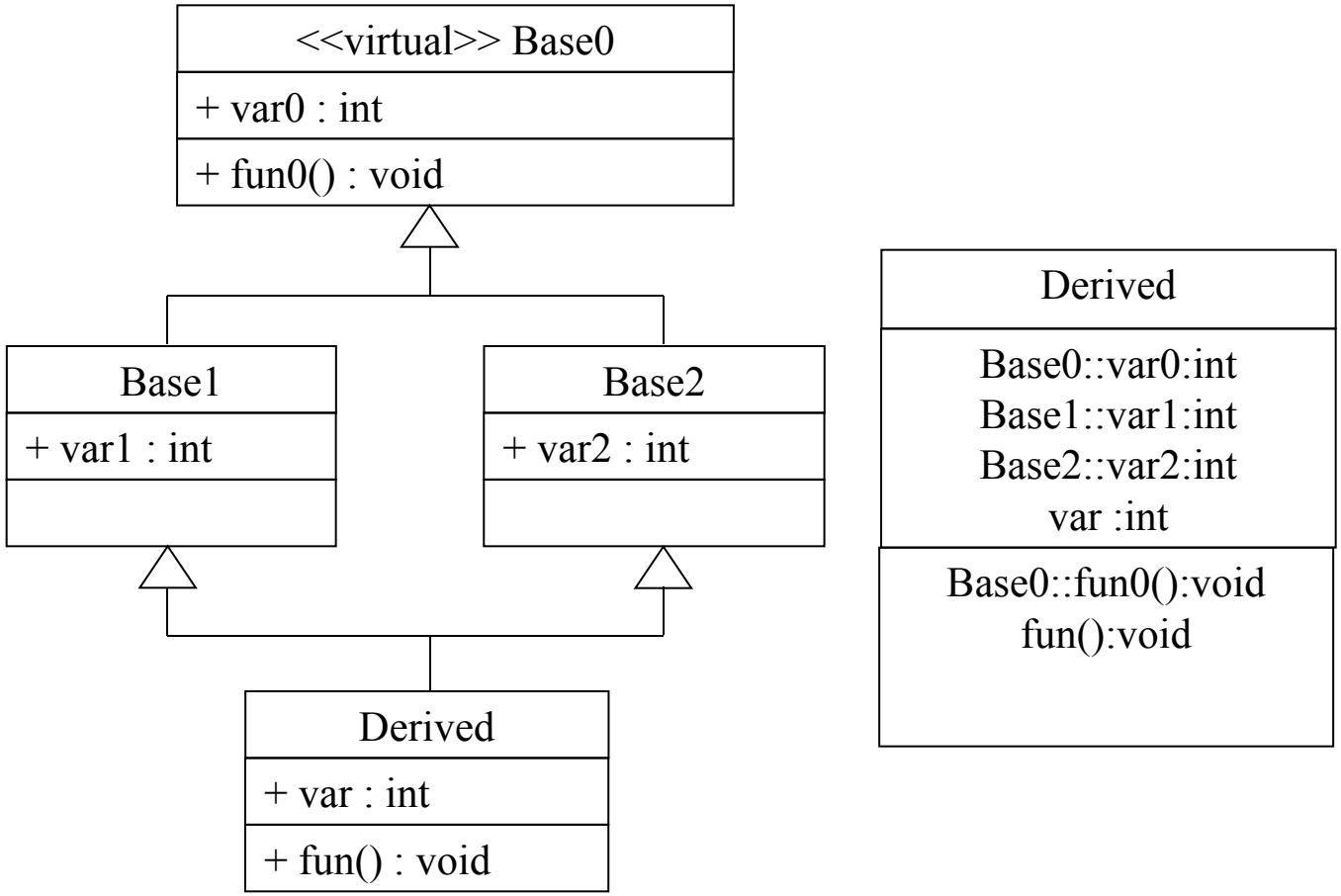
问题：冗余以及因冗余而导致的不一致性

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。

7.5.3 虚基类及其派生类构造函数

- 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;
}
```

7.9 小结

主要内容

- 类的继承、类成员的访问控制、单继承与多继承、派生类的构造和析构函数、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。