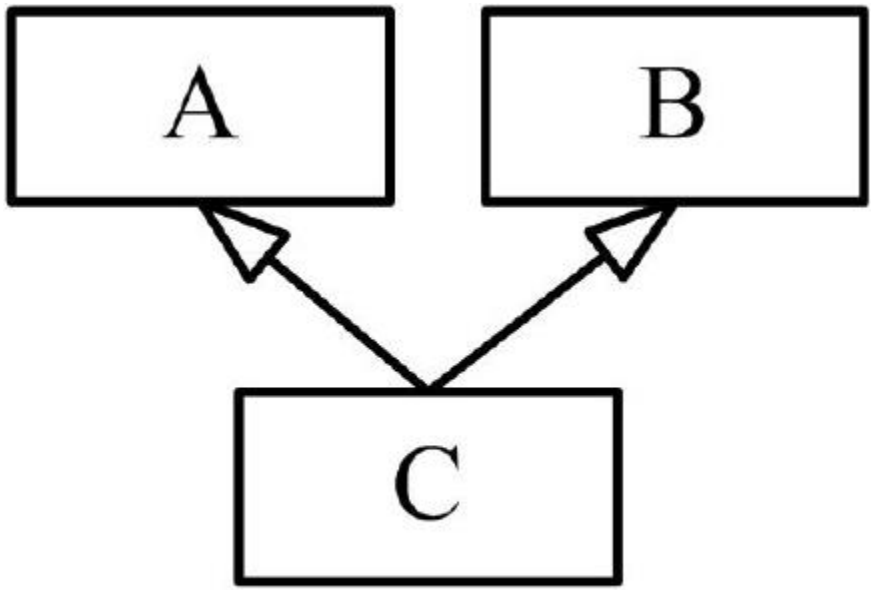
面向对象程序设计
420420

多重继承

- 1、多重继承派生类
- 2、二义性问题及名字支配规则

- 除去一个类从一个基类派生，C++还支持一个派生类同时继承多个基类。



- 1. 多重继承派生类的定义
- 如果已经定义了多个基类，那么定义多重继承的派生类的形式为：

```
class  派生类名:访问标号1  基类名1,访问标号2  基类名2,...  {  //派生类类体
       成员列表
};
```

```
class    A  {  };
class    B  :  public  A  {  };  //B继承A
class    C  :  public  A  {  };  //C继承A
class    D  :  public  B,public  C  {  };  //D多重继承B和C
```

- 2．多重继承派生类的构造函数
- 多重继承派生类的构造函数形式与单一继承时的构造函数形式基本相同，只是在派生类的构造函数初始化列表中调用多个基类构造函数。一般形式为：

```
派生类名(形式参数列表) : 基类名1(基类1构造函数实参列表),
                          基类名2(基类2构造函数实参列表),
                          …,
                          子对象名1(子对象1属类构造函数实参列表),
                          …,
                          派生类初始化列表
{
        派生类初始化函数体
}
```

- 其调用顺序是：
- ①调用基类构造函数，各个基类按定义时的次序先后调用；
- ②调用子对象构造函数，各个子对象按声明时的次序先后调用；
- ③执行派生类初始化列表；
- ④执行派生类初始化函数体；

【例36.1】多重继承举例

```
 1 #include <iostream>
 2 using namespace std;
 3 class Base1 //基类Base1
 4 {
 5         private:
 6                 int b1;
 7         public:
 8                 Base1(){b1=0;cout<<"默认构造Base1: "<<"b1="<<b1<<endl;}
 9                 Base1(int i){b1=i;cout<<"构造Base1: "<<"b1="<<b1<<endl;}
10 };
11 class Base2 //基类Base2
12 {
13         private:
14                 int b2;
15         public:
16                 Base2(){b2=0;cout<<"默认构造Base2: "<<"b2="<<b2<<endl;}
```

```
17              Base2(int j){b2=j;cout<<"构造Base2: "<<"b2="<<b2<<endl;}
18  };
19  class Base3 {    //基类Base3
20          public:
21              Base3(){cout<<"默认构造Base3: "<<endl;}
22  };
23  class Derive : public Base1, public Base2, public Base3 {  //派生类Derive
24          private:
25              Base1 memberBase1;  //子对象
26              Base2 memberBase2;  //子对象
27              Base3 memberBase3;  //子对象
28          public:
29              Derive(){cout<<"默认构造Derive."<<endl;}
30              Derive(int a,int b,int c,int d):
31                          Base1(a),Base2(b),memberBase1(c),memberBase2(d)
32                      {cout<<"构造Derive."<<endl;}
33  };
```

```
34
35  int  main()
36  {
37          cout<<"\n创建派生类对象obj1：  "<<endl;
38          Derive  obj1;
39          cout<<"\n创建派生类对象obj2(1,2,3,4)：  "<<endl;
40          Derive  obj2(1,2,3,4);
41          return  0;
42  }
```

运行结果：

```
创建派生类对象obj1：
默认构造Base1: b1=0
默认构造Base2: b2=0
默认构造Base3:
默认构造Base1: b1=0
默认构造Base2: b2=0
默认构造Base3:
默认构造Derive.

创建派生类对象obj2(1,2,3,4)：
构造Base1: b1=1
构造Base2: b2=2
默认构造Base3:
构造Base1: b1=3
构造Base2: b2=4
默认构造Base3:
构造Derive.
```

1. 二义性问题

- 多重继承时，多个基类可能出现同名的成员。在派生类中如果使用一个表达式的含义能解释为可以访问多个基类的成员，则这种对基类成员的访问就是不确定的，称这种访问具有二义性。C++要求派生类对基类成员的访问必须是无二义性的。

例如：

```
class  A  {public:
        void  fun()  {  cout<<"a.fun"<<endl;    }
};
class  B  {public:
        void  fun()  {  cout<<"b.fun"<<endl;    }
        void  gun()  {  cout<<"b.gun"<<endl;  }
};
class  C:public  A,public  B  {public:
        void  gun()  {  cout<<"c.gun"<<endl;  }  //重写gun()，无二义性
        void  hun()  {  fun();  }  //出现二义性，fun()来自A？fun()来自B？
};
int  main()
{
      C  c,*p=&c;
      return  0;
}
```

- 使用成员名限定可以消除二义性，例如：

```
c.A::fun();  //成员名限定消除二义性,限定A类的fun()
c.B::fun();  //成员名限定消除二义性，限定B类的fun()
p->A::fun();  //成员名限定消除二义性，限定A类的fun()
p->B::fun();  //成员名限定消除二义性，限定B类的fun()
```

- 基本形式为：

```
对象名.基类名::成员名
对象指针名->基类名::成员名
```

2. 名字支配规则

C++对于在不同的作用域声明的名字，可见性原则是：

- 如果存在两个或多个具有包含关系的作用域，外层声明了一个名字，而内层没有再次声明相同的名字，那么外层名字在内层可见；

- 如果在内层声明了相同的名字，则外层名字在内层不可见，这时称内层名字隐藏（或覆盖）了外层名字，这种现象称为隐藏规则。

- 在类的派生层次结构中，基类的数据成员和派生类新增的成员都具有类作用域，二者的作用域是不同的：基类在外层，派生类在内层。

- 如果派生类声明了一个和基类数据成员同名的新成员，派生的新成员就覆盖了基类同名成员，直接使用成员名只能访问到派生类的成员。

- 如果派生类中声明了与基类成员函数同名的新函数，即使函数的参数不同，从基类继承的同名函数的所有重载形式也都会被覆盖。

- 如果要访问被覆盖的成员，就需要使用基类名和作用域限定运算符来限定。（即基类名::成员名）

- 派生类D中的名字N覆盖基类B中同名的名字N，称为名字支配规则。如果一个名字支配另一个名字，则二者之间不存在二义性，当选择该名字时，使用支配者的名字，如：

```
c.gun();//使用C::gun
```

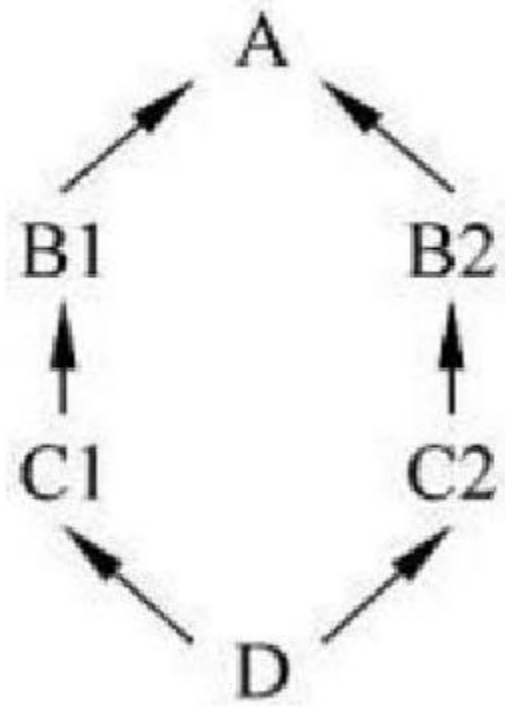
- 如果要使用被支配者的名字，则应使用成员名限定，例如：

```
c.B::gun();  //使用B::gun
```

多重继承

- 3、虚基类
- 4、多重继承应用举例

- C++提供虚基类（virtual base class）的机制，使得在继承间接共同基类时只保留一份成员。



A为虚基类

- 1. 虚基类的定义
- 虚基类是在派生类定义中，指定继承方式时声明的。声明虚基类的一般形式为：

```
class  派生类名: virtual  访问标号  虚基类名,... { //类体
       成员列表
};
```

- 需要注意，为了保证虚基类在派生类中只继承一次，应当在该基类的所有直接派生类中声明为虚基类。否则仍然会出现对基类的多次继承。

【例36.2】虚基类举例

```
1  #include <iostream>
2  using namespace std;
3  class A// 声明为基类A
4  {
5  public://外部接口
6          A(int n){ nv=n; cout<<"Member of A"<<endl; }//A类的构造函数
7          void fun(){ cout << "fun of A" << endl; }
8  private:
9          int nv;//私有成员
10 };
11 class B1 :virtual public A     //声明A为虚基类
12 {
13 public:
14         B1(int a):A(a){ cout<<"Member of B1"<<endl; }//B1类的构造函数
```

```
15 private:
16         int nv1;
17 };
18 class B2 :virtual public A     //声明A为虚基类
19 {
20 public:
21         B2(int a) :A(a){ cout << “Member of B2” << endl; }//B2类的构造函数
22 private:
23         int nv2;
24 };
25 class C :public B1, public B2
26 {
27 public:
28         // 派生类的构造函数的成员初始化列表中必须列出对虚基类构造函数的调用
29         C(int a):A(a),B1(a),B2(a){cout << "Member of C" << endl;}
```

```
30          void fund(){ cout << "fund of C" << endl; }
31  private:
32          int nvd;
33  };
34  int main()
35  {
36          C c1(1);
37          c1.fund();
38          c1.fun(); //不会产生二义性
39          return 0;
40  }
```

运行结果：

```
Member of A
Member of B1
Member of B2
Member of C
fund of C
fun of A
```

2.虚基类的初始化

- 如果在虚基类中定义了带参数的构造函数，而且没有定义默认构造函数，则在其**所有**派生类(包括直接派生和间接派生)中，都要通过构造函数的初始化表对虚基类进行初始化。例如：

```
class A { public: A(int) {} }; //定义基类A, A有带参数的构造函数
class B : virtual public A { public: B(int a):A(a) {} };
//对基类A初始化
class C : virtual public A { public: C(int a):A(a) {} };
//对基类A初始化
class D : public B,public C
{ public: D(int a):A(a),B(a),C(a) {} };
```

- 在最后的派生类中不仅要负责对其直接基类进行初始化，还要负责对虚基类初始化。

关于虚基类的说明：

（1） 一个类可以在一个类族中既被用作虚基类，也被用作非虚基类。

（2）派生类的构造函数的成员初始化列表中必须列出对虚基类构造函数的调用；如果未列出，则表示使用该虚基类的默认构造函数。

（3）在一个成员初始化列表中同时出现对虚基类和非虚基类构造函数的调用时，虚基类的构造函数先于非虚基类的构造函数执行。

【例36.3】

```
1  #include <iostream>
2  using namespace std;
3  enum Color {Red,Yellow,Green,White}; //颜色枚举类型
4  class Circle {          //圆类Circle的定义
5          float radius;  //私有数据成员
6  public:
7          Circle(float r) { radius=r;    //构造函数
8                  cout<<"Circle initialized!"<<endl;
9          }
10         ~Circle() {                    //析构函数
11                 cout<<"Circle    destroyed!"<<endl;
12         }
13         float Area() {
14                 return 3.1415926*radius*radius;
15         }
16 };
```

36.4 多重继承应用举例

```
17 class Table {    //桌子类Table的定义
18         float height;            //私有数据成员
19 public:
20         Table(float h) {          //构造函数
21                 height=h;
22                 cout<<"Table initialized!"<<endl; }
23         ~Table() { cout<<“Table destroyed!”<<endl; }    //析造函数
24         float Height() { return height; }
25 };
26 class RoundTable:public Table,public Circle {//圆桌类的定义
27         Color color;                  //私有数据成员
28 public:
29         RoundTable(float h,float r,Color c);    //构造函数
30         int GetColor() { return color; }
31         ~RoundTable() { cout<<"RoundTable destroyed!"<<endl; } //析造函数
32 };
```

```
33 RoundTable::RoundTable(float h,float r,Color c):
                                        Table(h),Circle(r)//圆桌构造函数的定义
34 {
35      color=c;
36      cout<<"RoundTable initialized!"<<endl;
37 }
38 int main()
39 {
40     RoundTable cir_table(15.0,2.0,Yellow);
41     cout<<"The table properties are:"<<endl;
42     cout<<“Height=”<<cir_table.Height()<<endl;//调用Table类的成员函数
43     cout<<"Area="<<cir_table.Area()<<endl; //调用circle类的成员函数
44     cout<<“Color=”<<cir_table.GetColor()<<endl; //调用RoundTable类的成员函数
45     return 0;
46 }
```

运行结果：

```
Table   initialized!
Circle   initialized!
RoundTable   initialized!
The   table   properties   are:
Height=15
Area=12.5664
Color=1
RoundTable   destroyed!
Circle     destroyed!
Table   destroyed!
```