# 第五讲 继承

## 七、抽象基类与纯虚函数

## 八、多继承

# 7、抽象基类与纯虚函数

【例7-1】抽象基类、纯虚函数示例代码

```
/** Point.h **/

#ifndef Point_hpp
#define Point_hpp

#include <iostream>
using namespace std;

class Point{
public:
    Point(double i, double j);

    void print() const;

private:
    double x;
    double y;
};

#endif
```

```
/** Point.cpp **/

#include "Point.hpp"

Point::Point(double i, double j):x(i),y(j){

}

void Point::print() const{
    cout<<"x = "<<x<<endl;
    cout<<"y = "<<y<<endl;
}
```

```
/** Figure.h **/

#ifndef Figure_hpp
#define Figure_hpp

#include <iostream>
using namespace std;
#include "Point.hpp"

class Figure{//因为有纯虚函数，所以为抽象类
public:
    Figure (double i = 0, double j = 0);
    Point& location();
    void move(Point p);
    virtual void draw( ) = 0;//纯虚函数
    virtual void rotate(double) = 0;//纯虚函数

private:
    Point center;
};

#endif
```

```
/** Figure.cpp **/

#include “Figure.hpp"

Figure::Figure (double i, double j):center(i,j){

}

Point& Figure::location(){
    return center;
}

void Figure::move(Point p){
    center = p;
    draw( );
}
```

```
/** Circle.h **/

#ifndef Circle_hpp
#define Circle_hpp

#include <iostream>
using namespace std;
#include "Figure.hpp"

class Circle : public Figure{
public:
    Circle(double i, double j, double r);
    //子类中实现纯虚函数
    void draw();
    void rotate(double);

private:
    double radius;
};

#endif
```

```
/** Circle.cpp **/

#include "Circle.hpp"

Circle::Circle(double i, double j, double r) : Figure(i, j),
radius(r){
}

//子类中实现纯虚函数
void Circle::draw(){
    cout << "A circle with center ";
    location().print();
    cout << " and radius " << radius << endl;
}

void Circle::rotate(double){
    cout << "no effect.\n";
}
```

```
/** Square.h **/

#ifndef Square_hpp
#define Square_hpp

#include <iostream>
using namespace std;
#include "Figure.hpp"

class Square : public Figure{
public:
    Square(double i, double j, double d, double a);
    void draw();
    void rotate(double a);
    void vertices();

private:
    double side;
    double angle;
};

#endif
```

```
/** Square.cpp **/

#include "Square.hpp"

Square::Square(double i, double j, double d, double
a):Figure(i,j),side(d), angle(a){
}

void Square::draw( ){
    location().print( );
    cout<<"length:"<<side<<".\n"<<"side:"<<angle<<endl;
}

void Square::rotate(double a){
    angle += a;
    cout << "side:” << angle << endl;
}
void Square::vertices(){
    cout << "The square\n";
}
```

```
/** main.cpp **/

#include "Circle.hpp"
#include "Square.hpp"

int main(int argc, const char * argv[]) {

    Circle c(1, 2, 3);
    Square s(4, 5, 6, 7);
    Figure* f = &c;
    Figure& g = s;

    //Figure ff;
    //Figure aFigure(10,10); 不能构造对象
    //f = new Figure(10,10); 不能构造对象

    f -> draw();
    f -> move(Point(2, 2));

    g.draw( );
    g.rotate(1);
    g.move(Point(1,1));

    s.vertices( );
    return 0;
}
```

程序运行结果如下：

```
with:x = 1
y = 2
radius:3
with:x = 2
y = 2
radius:3
x = 4
y = 5
side:6
side:7
side:8
x = 1
y = 1
side:6
side:8
The square
```

示例分析:

(1) 纯虚函数: 函数 = 0 就是一个纯虚函数，纯虚函数没有函数体，不需要实现。在子类里实现纯虚函数的具体功能。

(2) 抽象基类: 拥有纯虚函数的类叫做抽象类，抽象类只能作为基类，不能构建对象。因为抽象内的纯虚函数没有函数体。

(3) 纯虚函数被定义在派生类中。如果派生类不重写基类的纯虚函数，则将纯虚函数继承为派生类的纯虚函数。因此，派生类也是一个抽象类。

# 八、 多继承

多继承指一个子类有两个或者两个以上的父类。

多继承通常是导致程序出现问题的原因。

多继承主要的三个常见问题:

(1) 不同的父类中有同名函数，如何区分?

(2) 两个父类有共同的祖先。

【例8-1】两个父类中的同名函数如何区分示例

```
/** BaseA.h **/

#ifndef BaseA_hpp
#define BaseA_hpp

#include <iostream>
using namespace std;

class BaseA{
public:
    BaseA(int i);
    virtual void print();
    int get_a();

private:
    int a;
};

#endif
```

```
/** BaseA.cpp **/

#include "BaseA.hpp"

BaseA::BaseA(int i):a(i){

}

void BaseA::print(){
    cout << a << endl;
}

int BaseA::get_a(){
    return a;
}
```

```
/** BaseB.h **/

#ifndef BaseB_hpp
#define BaseB_hpp

#include <iostream>
using namespace std;

class BaseB{
public:
    BaseB(int j);
    void print();
    int get_b();

private:
    int b;
};

#endif
```

```
/** BaseB.cpp **/

#include "BaseB.hpp"

BaseB::BaseB(int j):b(j){

}

void BaseB::print(){
    cout << b << endl;
}

int BaseB::get_b(){
    return b;
}
```

```
/** Derived.h **/

#ifndef Derived_hpp
#define Derived_hpp

#include "BaseA.hpp"
#include "BaseB.hpp"

//构造函数的调用顺序为继承的顺序
class Derived : public BaseA, public BaseB{
public:
    Derived(int i,int j,int k);
    void print();
    void get_ab();

private:
    int c;
};

#endif
```

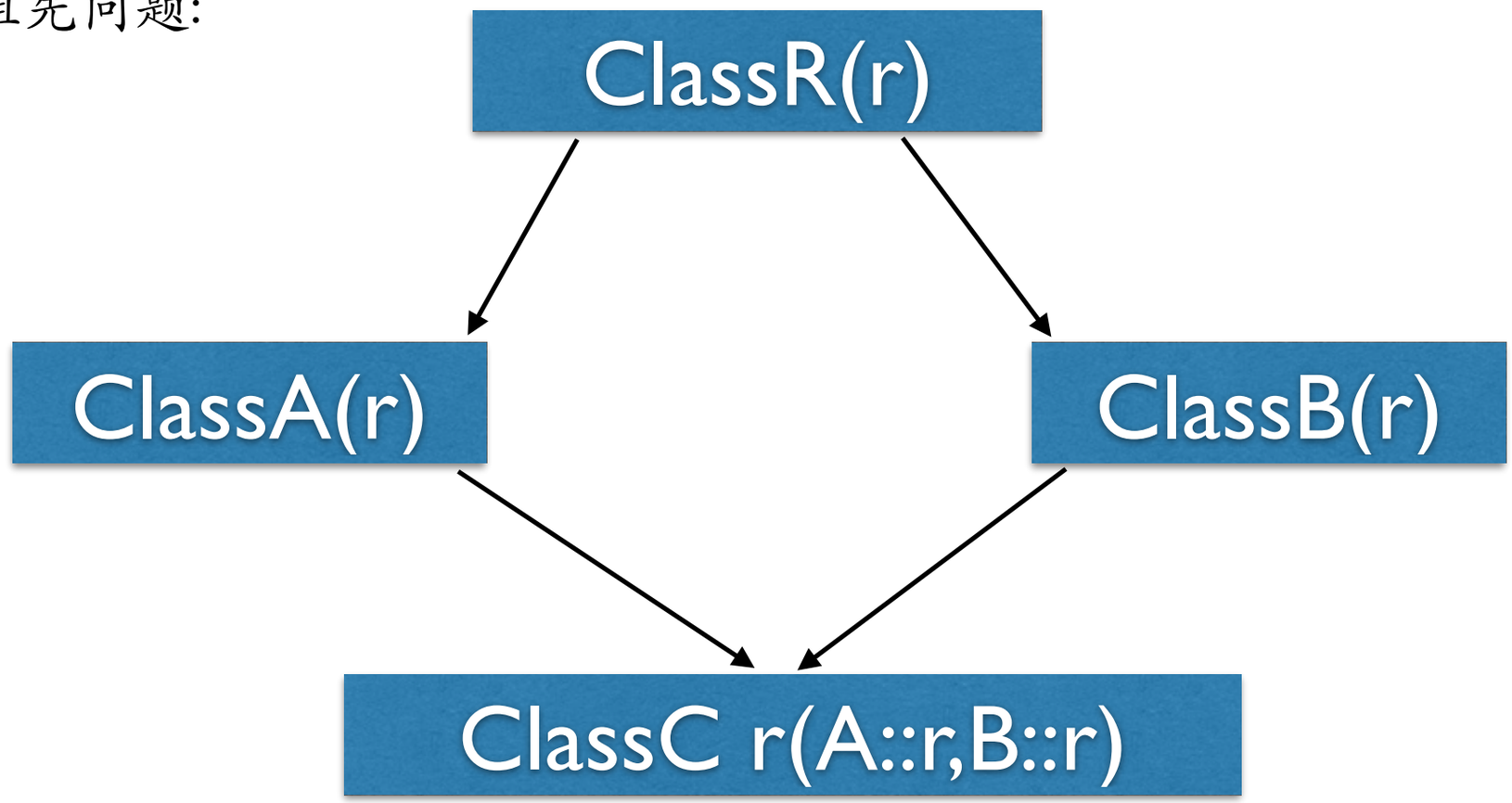
```cpp
/** Derived.cpp **/

#include "Derived.hpp"

Derived::Derived(int i,int j,int k):BaseA(i),BaseB(j),c(k){

}

//父类名::函数名，来区分不同版本的同名父类函数。
void Derived::print(){
    BaseA::print();
    BaseB::print();
}

void Derived::get_ab(){
    cout<<get_a()<<endl;
    cout<<get_b()<<endl;
    cout<<c<<endl;
}
```

```
/** main.cpp **/
#include "BaseA.hpp"
#include "BaseB.hpp"
#include "Derived.hpp"

int main(int argc, const char * argv[]) {

    Derived x(5, 8, 10);
    BaseA* ap = &x;
    BaseB* bp = &x;

    ap->print();     //调用子类C的print函数，A中的print是虚函数。
    bp->print();     //调用父类B的print函数，B中的print非虚函数。

    //bp->A::print( );  B和A是没有关系的这样调用时不被允许的

    x.BaseA::print( );  //用子类对象，调用被子类覆盖了的父类版本的函数。

    x.get_ab();

    return 0;
}
```

程序运行结果如下：

```
5
8
8
5
5
8
10
```

共同祖先问题:




菱形继承: A、B各有一份来自于它们父类R的成员r，C会继承两份来自于R的成员r。

这种重复是合法的，但在许多情况下，不必要的，甚至是毫无意义的。

【例8-2】菱形继承问题示例代码

```
/** R.h **/

#ifndef R_hpp
#define R_hpp

#include <iostream>
using namespace std;

class R{
public:
    R(int anInt){
        r = anInt;
    }

    void printOn()const{
        cout<<"r="<<r<<endl;
    }

private:
    int r;
};

#endif
```

```
/** A.h **/

#ifndef A_hpp
#define A_hpp

#include "R.hpp"

class A:public R{
public:
    A(int int1,int int2):R(int2){
        a = int1;
    }

private:
    int a;
};

#endif
```

```
/** B.h **/

#ifndef B_hpp
#define B_hpp

#include "R.hpp"

class B:public R{
public:
    B(int int1,int int2):R(int2){
        b = int1;
    }

private:
    int b;
};

#endif
```

```
/** C.h **/

#ifndef C_hpp
#define C_hpp

#include "A.hpp"
#include "B.hpp"

class C : public A, public B{
public:
    C(int int1,int int2, int int3):A(int2,int3), B(int2,int3){
        c = int1;
    }

private:
    //类C中有两份r和两份printOn成员分别继承自父类A和父类B
    int c;
};

#endif
```

```cpp
/** main.cpp **/

#include "R.hpp"
#include "A.hpp"
#include "B.hpp"
#include "C.cpp"

int main(int argc, const char * argv[]) {
    R rr(10);          //r = 10
    A aa(20,30);       //r = 30; a = 20;
    B bb (40,50);      //r = 50; b = 40;
    C cc(5, 7, 9);

    rr.printOn();
    aa.printOn();
    bb.printOn();

    //有歧义，C中不知道输出是继承自A中的R 还是输出继承自B中的R
    //cc.printOn();

    return 0;
}
```

程序运行结果如下：

```
r=10
r=30
r=50
```

为了解决上面二意性的问题，我们使用虚继承。

虚继承:

（1）直接继承祖先的两个基类，在继承时加virtual。

（2）通过多重继承而来的那个子类（孙子辈），在构造函数时，要调用祖先类的构造函数。孙子辈的派生类，直接继承祖先类的成员，再继承两个父类各自拓展的成员。

【例8-2】解决菱形继承问题示例代码

```
/** R.h **/

#ifndef R_hpp
#define R_hpp

#include <iostream>
using namespace std;

class R{
public:
    R(int x=0):r(x){
    }

    void printOn(){
        cout<<"printOn R="<<r<<endl;
    }

private:
    int r;
};

#endif
```

```
/** A.h **/

#ifndef A_hpp
#define A_hpp

#include "R.hpp"

class A:public virtual R { //虚继承
public:
    A(int x,int y):R(x),a(y){
    }

    void printOn(){
        cout<<"a="<<a<<endl;
        R::printOn();
    }

private:
    int a;
};

#endif
```

```
/** B.h **/

#ifndef B_hpp
#define B_hpp

#include "R.hpp"

class B:public virtual R { //虚继承
public:
    B(int x,int y):R(x),b(y){
    }

    void printOn(){
        cout<<"a="<<b<<endl;
        R::printOn();
    }

private:
    int b;
};

#endif
```

```
/** C.h **/

#ifndef C_hpp
#define C_hpp

#include "A.hpp"
#include "B.hpp"

class C : public A, public B{
public:
    C(int x,int y,int z,int w):R(x),A(x, y),B(x, z),c(w){
    }

    void printOn(){
        cout<<"c="<<c<<endl;
        A::printOn();
        B::printOn();
    }

private:
    int c;
};

#endif
```

```
/** main.cpp **/
#include "R.hpp"
#include "A.hpp"
#include "B.hpp"
#include "C.cpp"

int main(int argc, const char * argv[]) {
    R rr(1000);
    A aa(2222,444);
    B bb(3333,111);
    C cc(1212,345,123,45);

    //因为虚继承C中的r和printOn函数直接从R类中继承,所有孙子类中只有一份
    cc.printOn();

    return 0;
}
```

程序运行结果如下:

```
c=45
a=345
r=1212
b=123
r=1212
```

在C中调用f函数，因为父类A和父类B中都有调用R类的f函数，所以当C类调用f函数出现两次祖先f问题。

解决办法：每个类把属于自己的工作，单独封装。

【例8-2】解决菱形继承重复调用问题示例代码

```
/** R.h **/

#ifndef R_hpp
#define R_hpp

#include <iostream>
using namespace std;

class R {
public:
    R (int x=0);
    virtual void f();

private:
    int r;
};

#endif
```

```
/** R.cpp **/

#include "R.hpp"

R::R (int x):r(x){

}

void R::f(){
    cout<<"r="<<r<<endl;
}
```

```
/** A.h **/

#ifndef A_hpp
#define A_hpp

#include "R.hpp"

class A : virtual public R{
protected:
    void fA();

public:
    A(int x,int y);
    void f();

private:
    int a;
};

#endif
```

```
/** A.cpp **/

#include "A.hpp"

void A::fA(){ //把A自己的工作封装成一个独立的函数
    cout<<"a="<<a<<endl;
}

A::A(int x,int y):R(x),a(y){
}

void A::f(){ //覆盖了父类R的虚函数。
    fA(); //调用fA( );
    R::f();
}
```

```
/** B.h **/

#ifndef B_hpp
#define B_hpp

#include "R.hpp"

class B : virtual public R{
protected:
    void fB();

public:
    B (int x,int y);
    void f();

private:
    int b;
};

#endif
```

```
/** B.cpp **/

#include "B.hpp"

void B::fB(){ //把B自己的工作封装成一个独立的函数
    cout<<"b="<<b<<endl;
}

B::B(int x,int y):R(x),b(y){
}

void B::f(){
    fB();
    R::f();
}
```

```
/** C.h **/

#ifndef C_hpp
#define C_hpp

#include "A.hpp"
#include "B.hpp"

class C : public A, public B{
protected:
    void fC();

public:
    C(int x,int y,int z,int w);
    void f();

private:
    int c;
};

#endif
```

```
/** C.cpp **/

#include "C.hpp"

void C::fC(){
    cout<<"c="<<c<<endl;
}

C::C(int x,int y,int z,int w):R(x),A(x, y),B(x, z),c(w){

}

void C::f(){
    R::f();     //单独调用祖先的函数
    A::fA();
    B::fB();
    fC();
}
```

```cpp
/** main.cpp **/

#include "C.hpp"

int main(int argc, const char * argv[]) {
    R rr(1000);
    A aa(2222,444);
    B bb(3333,111);
    C cc(1212,345,123,45);
    cout<<"==========="<<endl;
    cc.f();

    cout<<"==========="<<endl;
    R* p = &cc;
    p->f();
    p = &aa;
    p->f();

    cout<<"==========="<<endl;
    R& re1 = cc;//祖先的引用指向孙子对象
    re1.f();
    R& re2 = bb;
    re2.f();
    return 0;
}
```

程序运行结果如下：

```
===========
r=1212
a=345
b=123
c=45
===========
r=1212
a=345
b=123
c=45
a=444
r=2222
===========
r=1212
a=345
b=123
c=45
b=111
r=3333
```