§18. 多态性与虚函数

1. 多态性的基本概念

多态的广义定义：一个事物有多种形态

面向对象方法学的定义：不同的对象能接受同一消息，并自主地对消息进行处理

★ 程序设计语言+类 = 基于对象的程序设计语言

★ 程序设计语言+类+多态 = 面向对象的程序设计语言

C++程序设计中多态的含义：同一作用域内有多个不同功能的函数可以具有相同的函数名

分类：

静态多态：在程序编译时已确定调用函数(函数重载)

动态多态：在程序运行时才确定操作的对象(虚函数)

§18. 多态性与虚函数

2. 多态性的引入

例：假设存在如下的类继承层次：

	数据成员	成员函数	
Point	坐标 x 坐标 y	构造 setPoint(x,y)设置x,y getX,getY, 取x,y的值 << 重载	
Circle	半径 radius	构造 setRadius(r) 设置半径 getRadius() 取半径 area() 求面积 << 重载	设圆心、取圆心 的函数继承
Cylinder	高度 height	构造 setHeight(h) 设置高度 getHeight() 取高度 area() 求表面积 volumn() 求体积 << 重载	设圆心、取圆心 设半径、取半径 的函数继承

§18. 多态性与虚函数

2. 多态性的引入

例：Point类的定义、实现及测试函数

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

const double pi=3.14159;

class Point {
    protected: //希望被子类继承但外部无法访问
       double x, y;
    public:
        Point(double a=0, double b=0);
        void setPoint(double a, double b);
        double getX() const;//常成员函数,不能修改值
        double getY() const;//常成员函数,不能修改值
        friend ostream &operator<< (ostream &out, const Point &p);
};

Point::Point(double a, double b) //初始化时用
{
    x=a;
    y=b;
}

void Point::setPoint(double a, double b)  //执行中用
{
    x=a;
    y=b;
}
```

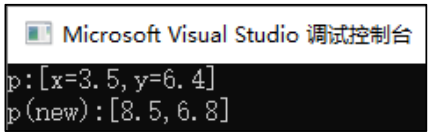
```
double Point::getX() const
{
    return x;
}

double Point::getY() const
{
    return y;
}

ostream &operator<< (ostream &out, const Point &p)
{
    out << "[" << p.x << "," << p.y << "]" << endl;
    return out;
}

int main()
{
    Point p(3.5, 6.4);
    cout << "p:[x=" << p.getX() << ",y=" << p.getY() << "]" << endl;

    p.setPoint(8.5, 6.8);
    cout << "p(new):" << p << endl;
    return 0;
}
```



§18. 多态性与虚函数

2. 多态性的引入

例：Circle类的定义、实现及测试函数

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

... class Point  的定义与实现 ...

class Circle : public Point {
    protected:  //准备被volumn继承
        double radius;
    public:
        Circle(double x=0, double y=0, double r=0);
        void  setRadius(double r);
        double getRadius() const;
        double area() const;
        friend ostream &operator<<(ostream &out, const Circle &c);
};

Circle::Circle(double x, double y, double r) : Point(x,y),radius(r)
{  //初始化表形式，函数体为即可
}

void Circle::setRadius(double r)
{
    radius = r;
}

double Circle::getRadius() const
{
    return radius;
}
```

```
Microsoft Visual Studio 调试控制台
c:圆心[x=3.5,y=6.4]
圆半径=5.2
圆面积=84.9486
point:[3.5,6.4]

c(new):圆心[x=5,y=5]
圆半径=7.5
圆面积=176.714
point(new):[5,5]
```

```
double Circle::area() const
{
    return pi * radius * radius;
}

ostream &operator<<(ostream &out, const Circle &c)
{
    out << "圆心[x=" << c.x << ",y=" <<c.y << "]" << endl
        << "圆半径=" << c.radius << endl
        << "圆面积=" << c.area() << endl;
    return out;
}

int main()
{
    Circle c(3.5, 6.4, 5.2);
    cout << "c:圆心[x=" << c.getX() << ",y=" << c.getY() << "]" << endl
         << "圆半径=" << c.getRadius() << endl
         << "圆面积=" << c.area() << endl;
    Point *p=&c;  //派生类出现在需要基类的位置
    cout << "point:" << *p << endl;  //输出Point的信息+多空一行

    c.setRadius(7.5);
    c.setPoint(5.0, 5.0);
    cout << "c(new):" << c; //<<重载最后已有endl
    Point &pRef=c;  //派生类出现在需要基类的位置
    cout << "point(new):" << pRef << endl;  //输出Point的信息
    return 0;
}
```

§18. 多态性与虚函数

2. 多态性的引入

例：Cylinder类的定义、实现及测试函数

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

... class Point  的定义与实现 ...
... class Circle 的定义与实现 ...

class Cylinder : public Circle {
    protected:
        double height;
    public:
        Cylinder(double x=0, double y=0, double r=0, double h=0);
        void setHeight(double h);
        double getHeight() const;
        double area() const;
        double volume() const;
        friend ostream &operator<<(ostream &out, const Cylinder &cy);
};
Cylinder::Cylinder(double x, double y, double r,double h) :
                                                Circle(x, y, r), height(h)
{   //初始化表形式，空函数体即可
}
void  Cylinder::setHeight(double h)
{   height = h;
}
double Cylinder::getHeight() const
{   return height;
}
double Cylinder::area() const
{   return 2 * Circle::area() + 2 * pi * radius * height;
}
double Cylinder::volume() const
{   return Circle::area() * height;
}
```

```
ostream &operator<< (ostream &out, const Cylinder &cy)
{   out << "柱圆心[x=" << cy.x << ",y=" << cy.y << "]" << endl
        << "柱半径=" << cy.radius << endl
        << "柱高=" << cy.height << endl
        << "柱表面积=" << cy.area() << endl
        << "柱体积=" << cy.volume() << endl;
    return out;
}
int main()
{  Cylinder cy1(3.5, 6.4, 5.2, 10.0);
   cout << "cy1:柱圆心[x=" << cy1.getX() <<",y=" << cy1.getY() << "]"<<endl
        << "柱半径=" << cy1.getRadius() << endl
        << "柱高度=" << cy1.getHeight() << endl
        << "柱表面积=" << cy1.area() << endl
        << "柱体积=" << cy1.volume() << endl;

   Point *p = &cy1;  //赋值兼容规则
   cout << "point:" << *p;
   Circle *pc = &cy1; //赋值兼容规则
   cout << "circle:" << *pc;
   cout << endl; //多空一行

   cy1.setHeight(15.0);
   cy1.setRadius(7.5);
   cy1.setPoint(5.0, 5.0);
   cout << "cy1(new):" << cy1;
   Point &pRef = cy1;  //赋值兼容规则
   cout << "point(new):" << pRef;
   Circle &cRef = cy1; //赋值兼容规则
   cout << "circle(new):" << cRef;

   return 0;
}
```

Microsoft Visual Studio 调试控制台

```
cy1:柱圆心[x=3.5,y=6.4]
柱半径=5.2
柱高度=10
柱表面积=496.623
柱体积=849.486
point:[3.5,6.4]
circle:圆心[x=3.5,y=6.4]
圆半径=5.2
圆面积=84.9486

cy1(new):柱圆心[x=5,y=5]
柱半径=7.5
柱高=15
柱表面积=1060.29
柱体积=2650.72
point(new):[5,5]
circle(new):圆心[x=5,y=5]
圆半径=7.5
圆面积=176.714
```

§18. 多态性与虚函数

2. 多态性的引入

在 Point -> Circle -> Cylinder 的继承层次中：

★ setPoint(),getX(),getY(),setRadius(),getRadius() 是继承

★ area()是支配规则

★ <<运算符是重载

★ 是静态多态，在编译时已确定应调用哪个函数

	数据成员	成员函数	
Point	坐标 x 坐标 y	构造 setPoint(x,y)设置x,y getX,getY，取x,y的值 << 重载	
Circle	半径 radius	构造 setRadius(r) 设置半径 getRadius() 取半径 area() 求面积 << 重载	设圆心、取圆心 的函数继承
Cylinder	高度 height	构造 setHeight(h) 设置高度 getHeight() 取高度 area() 求表面积 volumn() 求体积 << 重载	设圆心、取圆心 设半径、取半径 的函数继承

§18. 多态性与虚函数

3. 虚函数

3.1. 多个同名函数的使用

参数个数、参数类型完全相同：

同一类　　：不允许

不同独立类　：允许，被不同对象调用而区分

类的继承层次：允许，支配规则，用类作用域符区分

参数个数、参数类型不完全相同：

同一类　　：允许，重载

不同独立类　：允许，被不同对象调用而区分

类的继承层次：允许，支配规则，用类作用域符区分（再次强调，不是重载）

§18. 多态性与虚函数

3. 虚函数

3.2. 虚函数的引入

在类的继承层次中，对于参数个数、参数类型完全相同的同名函数，采用支配规则进行访问，要通过不同的对象来访问不同的同名函数(调用形式不同)

```
int main()
{
    Circle c(5.0, 5.0, 7.5);
    cout << c.area() << endl;    //圆面积

    Cylinder cyl(5.0, 5.0, 7.5, 10.0);
    cout << cyl.area() << endl;          //圆柱体表面积
    cout << cyl.Circle::area() << endl; //圆柱体底面积
}
```

```
Micros
176.714
824.667
176.714
```

调用形式不同
圆对象.area()
柱对象.area()
柱对象.圆::area()

通过赋值兼容规则，可使调用形式相同，但只能访问派生类中的基类部分

为了能采用同一调用形式来访问类继承层次中的同名函数，引入虚函数(调用形式相同)

右例，期望：
78.5397
471.238
但目前做不到

不满足期望的原因，采用了静态多态，在编译时已确定访问基类的area函数

```
int main()
{
   Circle   c1 (3.5, 6.3, 5.0);
   Cylinder cyl(3.5, 6.3, 5.0, 10.0);
   Circle  *p;
   p = &c1;
   cout << p->area() << endl; //调用形式相同

   p = &cyl;
   cout << p->area() << endl; //调用形式相同

   return 0;
}
```

```
Micros
78.5397
78.5397
```

调用形式相同
圆对象.area()
柱对象.无名圆.area()

§18. 多态性与虚函数

3. 虚函数

3.3. 虚函数的定义与使用

定义：在基类的函数定义前加virtual声明

```
改动Circle的定义
class Circle : public Point {
    protected:  //准备被volumn继承
        double radius;
    public:
        Circle(double x=0, double y=0, double r=0);
        void  setRadius(double r);
        double getRadius() const;
        virtual double area() const;
        friend ostream &operator<<(ostream &out, const Circle &c);
};
```

改动Circle的定义
但右侧main同上页，无任何变化

```
int main()
{
   Circle   c1 (3.5, 6.3, 5.0);
   Cylinder cyl(3.5, 6.3, 5.0, 10.0);
   Circle  *p;
   p = &c1;
   cout << p->area() << endl; //调用形式相同

   p = &cyl;
   cout << p->area() << endl; //调用形式相同

   return 0;
}
```

```
Micros
78.5397
471.238
```

调用形式相同
圆对象.area()
柱对象.area()

满足期望的原因，采用了动态多态，在运行时
才确定访问哪个类的area函数

§18. 多态性与虚函数

3. 虚函数

3.3. 虚函数的定义与使用

定义：在基类的函数定义前加virtual声明

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

class A {
  public:
    virtual void display() {
       cout << "A::display()" << endl;
       }
};

class B:public A {
  public:
    void display() {
       cout << "B::display()" << endl;
       }
};

class C:public B {
  public:
    void display() {
       cout << "C::display()" << endl;
       }
};
```

```
//A类不加 virtual 的执行结果
int main()
{   A a, &ra=a, *pa=&a;
    B b, &rb=b, *pb=&b;
    C c, &rc=c, *pc=&c;
    A *p;
    a.display();    A::
    b.display();    B::
    c.display();    C::
    ra.display();   A::
    rb.display();   B::
    rc.display();   C::
    pa->display();  A::
    pb->display();  B::
    pc->display();  C::
    p=&a;
    p->display();   A::
    p=&b;
    p->display();   A::
    p=&c;
    p->display();   A::
}
```

```
//A类加 virtual 的执行结果
int main()
{   A a, &ra=a, *pa=&a;
    B b, &rb=b, *pb=&b;
    C c, &rc=c, *pc=&c;
    A *p;
    a.display();    A::
    b.display();    B::
    c.display();    C::
    ra.display();   A::
    rb.display();   B::
    rc.display();   C::
    pa->display();  A::
    pb->display();  B::
    pc->display();  C::
    p=&a;
    p->display();   A::
    p=&b;
    p->display();   B::
    p=&c;
    p->display();   C::
}
```

§18. 多态性与虚函数

3. 虚函数

3.3. 虚函数的定义与使用

定义：在基类的函数定义前加virtual声明

★ 未加virtual前，“基类指针=&派生类对象”/“基类引用=派生类对象”，适用赋值兼容规则，访问的是派生类中的基类部分

★ 加virtual后，突破此限制，访问派生类的同名函数

使用：

★ virtual在类定义时出现，函数体外实现部分不能加

```
class A {
  public:
    virtual void display();
};
virtual void A::display()
{
    cout << "A::display()" << endl;
}
```

error C2723: “A::display”：“virtual”说明符在函数定义上非法

```
class A {
  public:
    virtual void display();
};

class B:public A {
  public:
    virtual void display();
};

class C:public B {
  public:
    virtual void display();
};
```

必须

建议

建议

★ 在类的继承序列中，只需要在最开始的基类中加virtual声明，后续派生类可以不加(建议加)

§18. 多态性与虚函数

3. 虚函数

3.3. 虚函数的定义与使用

使用：

★ 类的继承序列中该同名函数的参数个数、参数类型必须完全相同

★ 若类的继承层次中同名虚函数仅返回类型不同，则象重载一样，认为是错误

```
class A {
  public:
    virtual void display();
};
class B:public A {
  public:
    virtual int display();
};
class C:public B {
  public:
    virtual double display();
};
```

编译错误

```
(10,17): error C2555: “B::display”: 重写虚函数返回类型有差异，且不是来自“A::display”的协变
(6): message : 参见“A::display”的声明
(14,20): error C2555: “C::display”: 重写虚函数返回类型有差异，且不是来自“B::display”的协变
(10): message : 参见“B::display”的声明
```

★ 若派生类中无同名函数，则自动继承基类

```
class A {
  public:
    virtual void display();
};
class B:public A {
    无 display 函数
};
class C:public B {
  public:
    virtual void display();
};
int main()
{ A a;
  B b;
  C c;
  A *p;
  p=&a;
  p->display(); A::
  p=&b;
  p->display(); A::
  p=&c;
  p->display(); C::
}
```

§18. 多态性与虚函数

3. 虚函数

3.3. 虚函数的定义与使用

使用：

★ 若派生类中有同名函数，其参数个数、参数类型与基类的虚函数不同，则失去多态性，按支配规则及赋值兼容规则处理

```
class A {
  public:
    virtual void display() {
       cout << "A::" << endl;
       }
};
class B:public A {
  public:
    void display(int x) {
       cout << "B::x" << endl;
       }
};
class C:public B {
  public:
    void display() {
       cout << "C::" << endl;
       }
};
```

问：三句错的语句如何改正确？
答：第一句可改，后两句无法改
因为基类无法访问派生类成员

```
int main()
{ A a, *p;
  B b;
  C c;
  b.display();     错
  b.display(1);    B::x
  p=&a;
  p->display();    A::
  p=&b;
  p->display();    A::
  p->display(1);   错
  p=&c;
  p->display();    C::
  p->display(1);   错
}
```

§18. 多态性与虚函数

3. 虚函数

3.3. 虚函数的定义与使用

使用：

★ 对于派生类中的其它非virtual仍适用赋值兼容规则

★ 只有通过基类指针/引用方式访问时才适用虚函数规则，
其它形式(对象/自身指针/引用)仍用原来的规则

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

class A {
  public:
    virtual void display() {
       cout << "A::display()" << endl;
       }
   };
class B:public A {
  public:
    void display() {
       cout << "B::display()" << endl;
       }
   };
class C:public B {
  public:
    void display() {
       cout << "C::display()" << endl;
       }
   };
```

```
int main()
{   A a, &ra=a, *pa=&a;
    B b, &rb=b, *pb=&b;
    C c, &rc=c, *pc=&c;
    A *p;
    a.display();
    b.display();
    c.display();
    ra.display();
    rb.display();
    rc.display();
    pa->display();
    pb->display();
    pc->display();
    p=&a;
    p->display();
    p=&b;
    p->display();
    p=&c;
    p->display();
}
```

```
A::
B::
C::
A::
B::
C::
A::
B::
C::
A::

B::

C::
```

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

class A {
  public:
    virtual void display();
    void show();
};
class B:public A {
    void display();
    void show();
};
class C:public B {
  public:
    void display();
    void show();
};

int main()
{ A a, *p;
  B b;
  C c;
  p=&a;
  p->display(); A::
  p->show();    A::
  p=&b;
  p->display(); B::
  p->show();    A::
  p=&c;
  p->display(); C::
  p->show();    A::
}
```

§18. 多态性与虚函数

3. 虚函数

3.3. 虚函数的定义与使用

思考题：

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

class A {
  public:
    virtual void f1() { cout << "A::f1()" << endl; }
    virtual void f2() { cout << "A::f2()" << endl; }
    virtual void f3() { cout << "A::f3()" << endl; }
    virtual void f4() { cout << "A::f4()" << endl; }
            void f5() { cout << "A::f5()" << endl; }
};

class B:public A {
  public:
    void f1() { cout << "B::f1()" << endl; }
    void f3(double f) { cout << "B::f3()" << endl;}
    int  f4() { cout << "B::f4()" << endl; }
    void f5(double f) { cout << "B::f5()" << endl; }
};
```

```
int main()
{  B b;
   A *pa=&b;
   pa->f1();
   pa->f2();
   pa->f3();
   pa->f3(10);
   pa->f4();
   pa->f5();
   pa->f5(10);
   b.f1();
   b.f2();
   b.f3();
   b.f3(10);
   b.f4();
   b.f5();
   b.f5(10);
}
```

1、A、B类的定义语句中哪些会编译错？若A/B冲突，删除B中定义

2、保证A、B类定义不冲突的情况下，main中哪些语句会编译出错？

3、去除所有错误的语句，其余正确语句的执行结果？

§18. 多态性与虚函数

3. 虚函数

3.3. 虚函数的定义与使用

使用：

★ 若把函数重载理解为横向重载（同一类中），则虚函数可理解为纵向重载（类的继承层次中）

★ 非类的成员函数不能声明为多态

★ 类的静态成员函数不能声明为多态

支配规则、赋值兼容规则、虚函数的区别：

支配规则　　：通过自身对象、指针、引用访问(自身的)虚函数、普通函数

赋值兼容规则：通过基类指针、对象、引用访问(派生类中基类部分的)普通函数

虚函数　　：通过基类指针、引用访问(基类和派生类的同名)虚函数

§18. 多态性与虚函数

3. 虚函数

3.3. 虚函数的定义与使用

支配规则、赋值兼容规则、虚函数的区别：

```
class A {
  public:
    virtual f1(int x) {...}
    f2(int x) {...}
};
class B:public A {
  public:
    virtual f1(int x) {...}
    f2(int x) {...}
};
```

```
void fun1(A *pa)
{  pa->f1(10); //虚函数
   pa->f2(15); //赋值兼容
}
void fun2(A &ra)
{  ra.f1(10);  //虚函数
   ra.f2(15);  //赋值兼容
}
int main()
{
   A a;
   B b;
   fun1(&a);
   fun1(&b);
   fun2(a);
   fun2(b);
   a.f1(10);  //支配
   a.f1(10);  //支配
   b.f1(10);  //支配
   b.f2(15);  //支配
}
```

```
int main()
{  A a, *pa;
   B b;
   pa = &a;
   pa->f1(10); //支配
   pa->f2(15); //支配

   pa = &b;
   pa->f1(10); //虚函数
   pa->f2(10); //赋值兼容

   a.f1(10);   //支配
   a.f2(15);   //支配
   b.f1(10);   //支配
   b.f2(15);   //支配
}
```

```
int main()
{  B  b;
   A &ra = b;

   ra.f1(10);  //虚函数
   ra.f2(10);  //赋值兼容

   b.f1(10);   //支配
   b.f2(15);   //支配
}
```

引用一般不用在此处，因为只能始终指向b

```
int main()
{  A a, *pa = &a;
   B b, *pb = &b;

   pa->f1(10); //支配
   pa->f2(15); //支配

   pb->f1(10); //支配
   pb->f2(10); //支配

   a.f1(10);   //支配
   a.f2(15);   //支配
   b.f1(10);   //支配
   b.f2(15);   //支配
}
```

```
int main()
{  A a, &ra = a;
   B b, &rb = b;

   ra.f1(10);  //支配
   ra.f2(15);  //支配

   rb.f1(10);  //支配
   rb.f2(10);  //支配

   a.f1(10);   //支配
   a.f2(15);   //支配
   b.f1(10);   //支配
   b.f2(15);   //支配
}
```

§18. 多态性与虚函数

3. 虚函数

3.4. 静态关联与动态关联

关联：在编译系统中，确定标识符和存储地址的对应关系的过程称为关联

★ 包含了确定对象及所调用的函数的关系

★ 又称为联编、编联、束定、绑定(binding)

分类：

静态关联：在编译时确定对应关系(早期关联)

动态关联：在运行时确定对应关系(滞后关联)

§18. 多态性与虚函数

3. 虚函数

3.5. 虚析构函数

引入：在通过基类指针动态申请派生对象时，会出现对象撤销时无法调用派生类析构函数的问题

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

class A {
   public:
     ~A() { cout << "A析构" << endl; }
};

class B : public A {
  private:
    char *s;
  public:
    B() {  s = new char[10];  }
    ~B()
     {  delete s;
        cout << "B析构"  << endl; }
};
```

```
int main()
{   B b;
}  //运行结束后，系统自动调用
   //B的析构函数，再激活A析构
```

✓

```
int main()
{   B *pb = new B;
    delete pb; //调用B的析构函数
}              //再激活A的析构
```

✓

```
int main()
{   B *pb = new B;
    A *pa = pb;
    delete pb; //调用B的析构函数
}              //再激活A的析构
```

✓

```
int main()
{   A *pa = new B;
    delete pa; //仅调用A析构，s无法释放
}              //运行不错，丢内存
```

✗

§18.多态性与虚函数

3.虚函数

3.5.虚析构函数

引入：在通过基类指针动态申请派生对象时，会出现对象撤销时无法调用派生类析构函数的问题

解决：将基类的析构函数声明为虚函数

```
virtual ~ 类名()
{
   函数体
}
```

§18. 多态性与虚函数

3. 虚函数

3.5. 虚析构函数

引入：在通过基类指针动态申请派生对象时，会出现对象撤销时无法调用派生类析构函数的问题

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

class A {
   virtual public:
      ~A() { cout << "A析构" << endl; }
};

class B : public A {
  private:
    char *s;
  public:
    B() {  s = new char[10];  }
    ~B()
     {  delete s;
        cout << "B析构"  << endl; }
};
```

```
int main()
{   B b;
}  //运行结束后，系统自动调用
   //B的析构函数，再激活A析构
```

✓

```
int main()
{   B *pb = new B;
    delete pb; //调用B的析构函数
}              //再激活A的析构
```

✓

```
int main()
{   B *pb = new B;
    A *pa = pb;
    delete pb; //调用B的析构函数
}              //再激活A的析构
```

✓

```
int main()
{   A *pa = new B;
    delete pa; //调用B析构
}              //再激活A的析构
```

✓

§18. 多态性与虚函数

3. 虚函数

3.5. 虚析构函数

引入：在通过基类指针动态申请派生对象时，会出现对象撤销时无法调用派生类析构函数的问题

解决：将基类的析构函数声明为虚函数

```
virtual ~ 类名()
{
    函数体
}
```

★ 使用于派生类中有动态申请空间的情况，虽然类的继承序列中析构函数名不同，系统会自动当作虚函数处理

★ 虚析构函数调用时，先派生类，再基类

（普通虚函数　：只调派生类，不调基类

普通析构函数：先调派生类，再调基类）

★ 析构函数声明为虚函数后，通过基类、派生类自己生成的对象在释放时也不会出错，因此一般在类的继承序列中，建议将析构函数声明为虚析构函数

★ 构造函数不能声明为虚函数（虚函数只有和对象结合才能呈现多态，构造函数时对象正在生成）

§18. 多态性与虚函数

3. 虚函数

3.5. 虚析构函数

引入：在通过基类指针动态申请派生对象时，会出现对象撤销时无法调用派生类析构函数的问题

虚析构函数

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

class A {
  public:
    virtual ~A() { cout << "A析构" << endl; }
};
class B:public A {
  private:
    char *s;
  public:
    B() { s=new char[10]; }
    ~B() { delete s; cout << "B析构" << endl; }
};
```

基类指针方式

```
int main()
{
   A *pa = new B;
   delete pa;        B析构
   return 0;         A析构
}
```

派生类对象/指针方式

```
int main()
{
   B *pb = new B;
   delete pb;        B析构
   return 0;         A析构
}
```

普通析构函数

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

class A {
  public:
    ~A() { cout << "A析构" << endl; }
};
class B:public A {
  private:
    char *s;
  public:
    B() { s=new char[10]; }
    ~B() { delete s; cout << "B析构" << endl; }
};
```

基类指针方式

```
int main()
{
   A *pa = new B;
   delete pa;        A析构
   return 0;         丢内存!
}
```

派生类对象/指针方式

```
int main()
{
   B *pb = new B;
   delete pb;        B析构
   return 0;         A析构
}
```

§18. 多态性与虚函数

4. 纯虚函数与抽象类

4.1. 空虚函数

引入：在类的继承层次中，派生类都有同名函数，而基类没有，为使用虚函数机制，需要建立一条从基类到派生类的虚函数路径，因此在基类中定义一个同名空虚函数

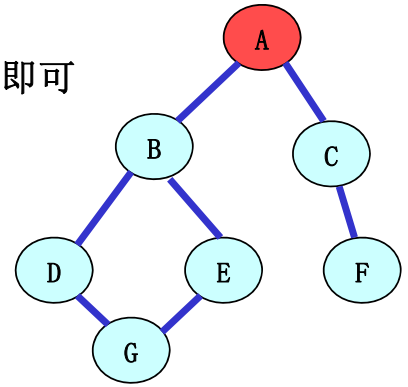
形式：函数名、参数个数、参数类型、返回值与派生类相同，函数体为空即可

```
class A {
  有实际意义的数据成员及函数
  virtual void display() { }
  virtual int show(){ return 0; }
};
class B:public A {
  有实际意义的数据成员及函数
  void display(){ 具体实现 }
  int show() { 具体实现 }
};

//CDEFG等相同
  1.有各自有意义的数据成员及函数
  2.各自独立的display及show函数
    参数个数、参数类型、返回值同
    但实现过程各不相同
```

A中定义display及show后可用统一方法调用

```
例：F f;  B b;
    A *pa = &f;
    pa->display();
    pa->show(10);
    pa = &b;
    pa->display();
    pa->show(15);
```



★ 基类的该函数虽然无意义，但基类的其它部分仍有意义，可定义对象、引用、指针等并进行正常操作

```
class A {
  有实际意义的数据成员及函数
  virtual void display() { }
  virtual int show(){ return 0; }
};
//BCDEFG的定义

void main()
{   C c1;
    A a1, *pa=&c1;
    a1.****;          //正常操作
    pa->show(20);     //虚函数形式
}
```

§18. 多态性与虚函数

4. 纯虚函数与抽象类

4.1. 空虚函数

4.2. 纯虚函数与抽象类

面向对象方法学的含义：为了对各类进行归纳，在更高的层次、更抽象的级别上考虑问题，简化复杂性，引入抽象类

下中国象棋
下国际象棋
下围棋
下跳棋
} 下棋

C++的具体应用：在类的多继承层次中，可能会出现多个基类并存的现象，若各基类有同名函数并希望使用虚函数机制，则需要引入一个更高层次的类，该类无实际意义，不进行具体操作，称为抽象类

§18. 多态性与虚函数

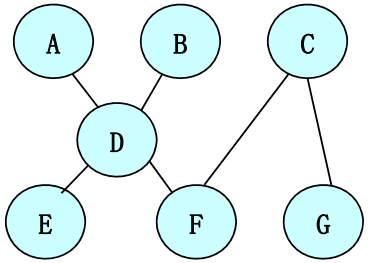
4. 纯虚函数与抽象类

4.1. 空虚函数

4.2. 纯虚函数与抽象类

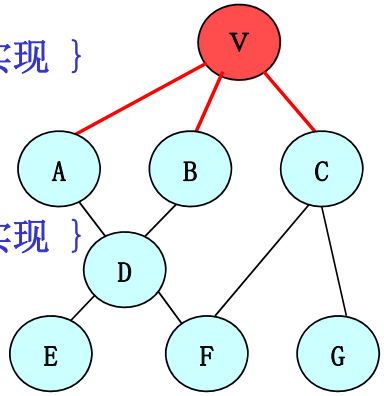
C++的具体应用：在类的多继承层次中，可能会出现多个基类并存的现象，若各基类有同名函数并希望使用虚函数机制，则需要引入一个更高层次的类，该类无实际意义，不进行具体操作，称为抽象类

```
假设A-G每个类都有display
且参数个数、参数类型、返回值都相同
class A {
  其它成员及函数
  void display(){ 具体实现 }
};
...
class E:public D {
  其它成员及函数
  void display(){ 具体实现 }
};
```




```
假设A-G每个类都有display
class V {
  public:
    不需要其它成员及函数
    virtual void display()
    { }
};
class A:public V { //继承V
  其它成员及函数
  void display(){ 具体实现 }
};
class E:public D {
  其它成员及函数
  void display(){ 具体实现 }
};
```

```
A a1;
G g1;
V *p;
p = &a1;
p->display()
p=&g1;
p->display();
```



§18. 多态性与虚函数

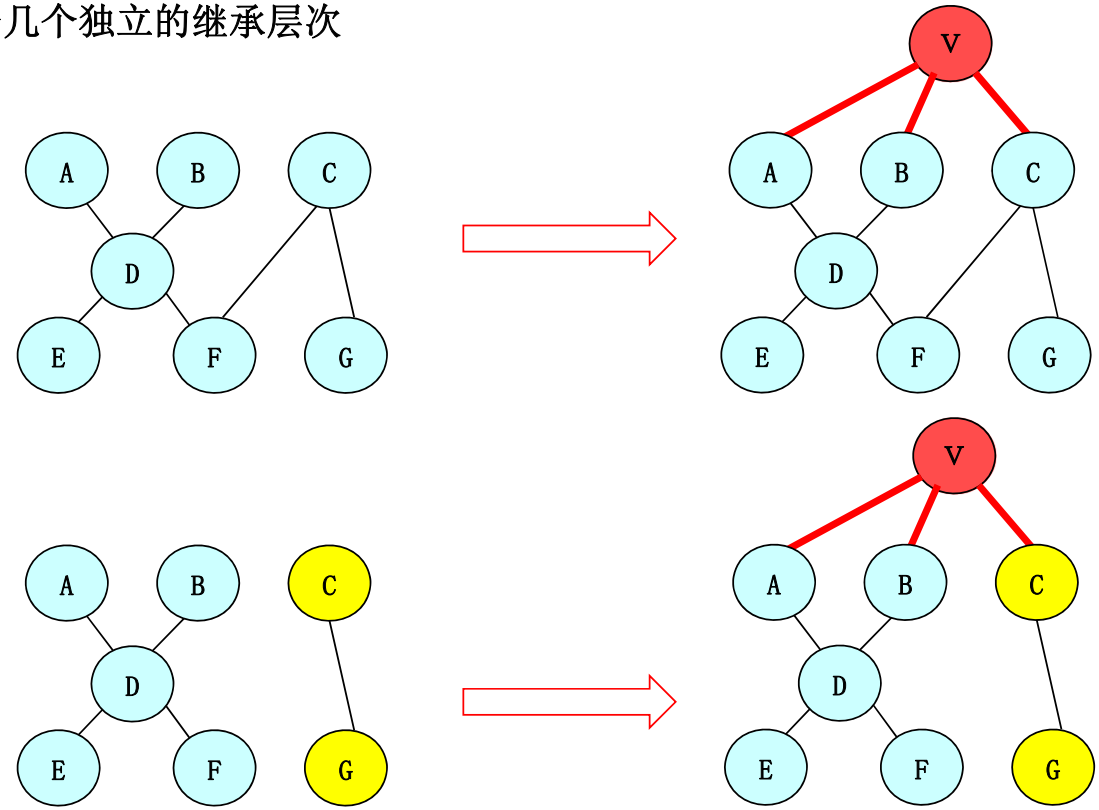
4. 纯虚函数与抽象类

4.1. 空虚函数

4.2. 纯虚函数与抽象类

C++的具体应用：在类的多继承层次中，可能会出现多个基类并存的现象，若各基类有同名函数并希望使用虚函数机制，则需要引入一个更高层次的类，该类无实际意义，不进行具体操作，称为抽象类

★ 也可以用于统一几个独立的继承层次



§18. 多态性与虚函数

4. 纯虚函数与抽象类

4.1. 空虚函数

4.2. 纯虚函数与抽象类

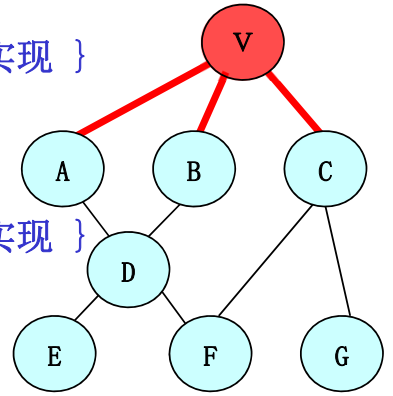
C++的具体应用：在类的多继承层次中，可能会出现多个基类并存的现象，若各基类有同名函数并希望使用虚函数机制，则需要引入一个更高层次的类，该类无实际意义，不进行具体操作，称为抽象类

存在的问题：
V还能定义对象 V v1;
但V的对象实际无意义

解决方法：
将V定义为抽象类

```
假设A-G每个类都有display
class V {
  public:
    不需要其它成员及函数
    virtual void display()
    { }
};
class A:public V { //继承V
  其它成员及函数
  void display(){ 具体实现 }
};
class E:public D {
  其它成员及函数
  void display(){ 具体实现 }
};
```

```
A a1;
G g1;
V *p;
p = &a1;
p->display()
p=&g1;
p->display();
```



§18. 多态性与虚函数

4. 纯虚函数与抽象类

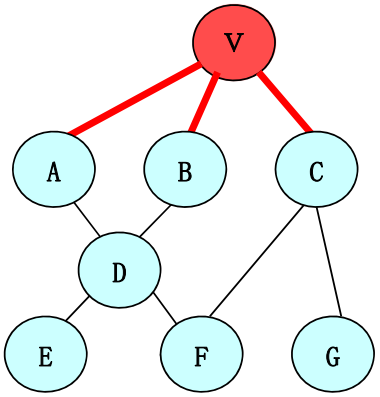
4.2. 纯虚函数与抽象类

抽象类的定义：C++中无明确的关键字定义，只要声明某一成员函数为纯虚函数即可

纯虚函数的声明：

virtual 返回类型 函数名(参数表) = 0;

★ 表示该函数没有实际意义，也不被调用



例：

```
class V {
  public:
     virtual void display()=0;
};
```

假设A-G每个类都有display

```
class V {
  public:
    不需要其它成员及函数
    virtual void display()
    { }
};
```

```
class V {
  public:
    virtual void display()=0;
};
```

§18. 多态性与虚函数

4. 纯虚函数与抽象类

4.2. 纯虚函数与抽象类

抽象类的使用：

★ 抽象类不能有实例对象，但可用于声明指针或引用

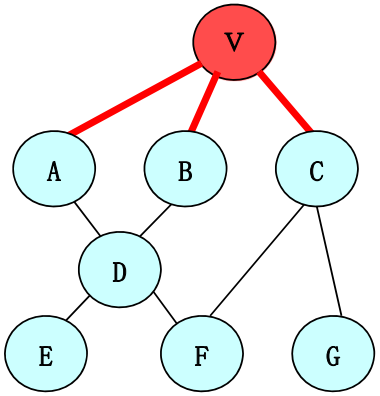
```
class V {
    public:
        virtual void display()=0;
    };
V v1;                  //错误
V *p;                  //正确
V &p=派生类对象;       //正确(一般用于函数形参)
```

★ 抽象类中定义数据成员及有实际意义的成员函数都是无意义的，但为了简化继承序列，可以进行定义，供派生类使用

（会导致理解混乱，初学者不推荐）

例：假设 A,B,C中都有int a,b成员

```
class A {
   protected:
      int a,b;
};
class B {
   protected:
      int a,b;
};
class C {
   protected:
      int a,b;
};
```




```
class V {
   protected:
      int a, b;
};
class A:public V {
   protected:
      ...
};
class B:public V {
   protected:
      ...
};
class C:public V {
   protected:
     ...
};
```

§18. 多态性与虚函数

4. 纯虚函数与抽象类

4.2. 纯虚函数与抽象类

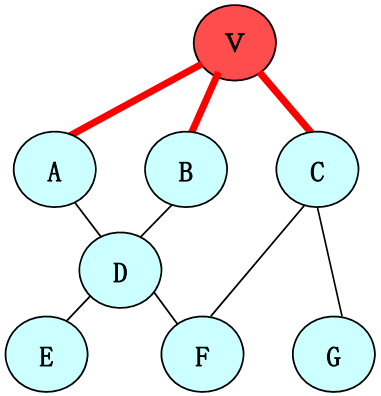
抽象类的使用：

★ 抽象类的直接派生类的同名虚函数必须定义，否则继承抽象类的纯虚函数，也成为抽象类(若不需要，可定义为空虚函数)

```
若：class V {
      public:
        virtual void display()=0;
      };

则：ABC中的display()必须定义
    即使B中不需要display，也要定义

    class B:public V {
       public:
         void display() { }
       };
```



空虚函数与纯虚函数的区别：

空虚函数：类的其它成员有实际含义，可生成对象

纯虚函数：无实例对象，无实际含义，仅为了在更高的层次上统一类

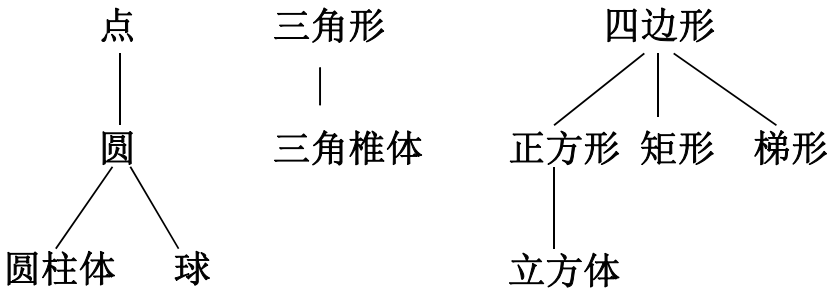
§18. 多态性与虚函数

4. 纯虚函数与抽象类

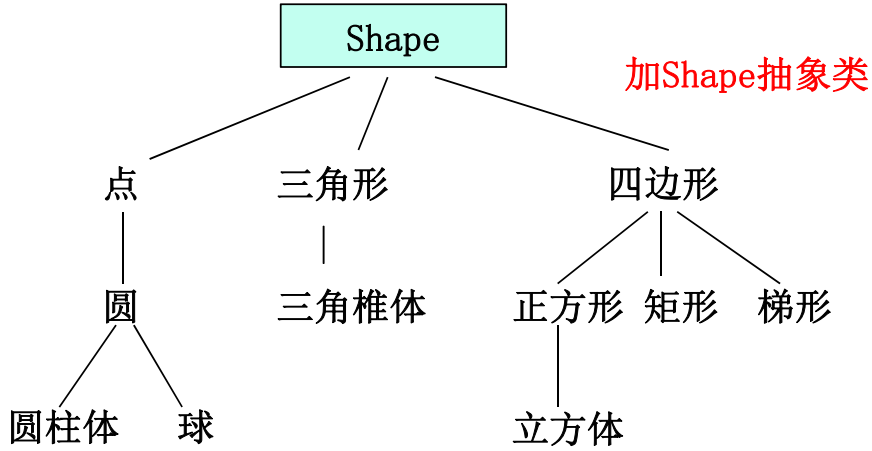
4.2. 应用实例

例：将各种几何形状的求面积、表面积、...等做成一个继承序列

三个独立继承序列



加Shape抽象类

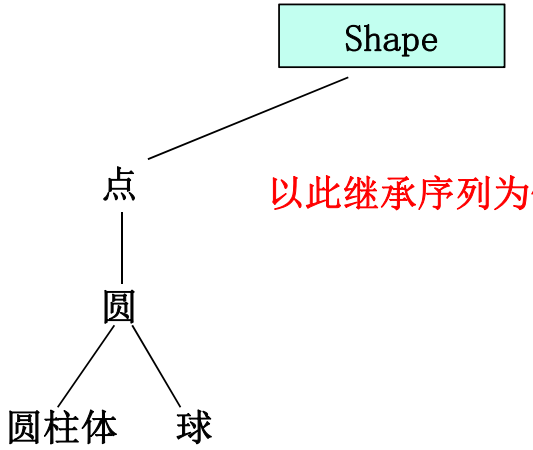


§18. 多态性与虚函数

4. 纯虚函数与抽象类

4.3. 应用实例

```
class Shape {
    public:
        virtual double area() const { return 0.0; }
        virtual double volumn() const { return 0.0; }
        virtual void shapeName() const = 0;
    };
```

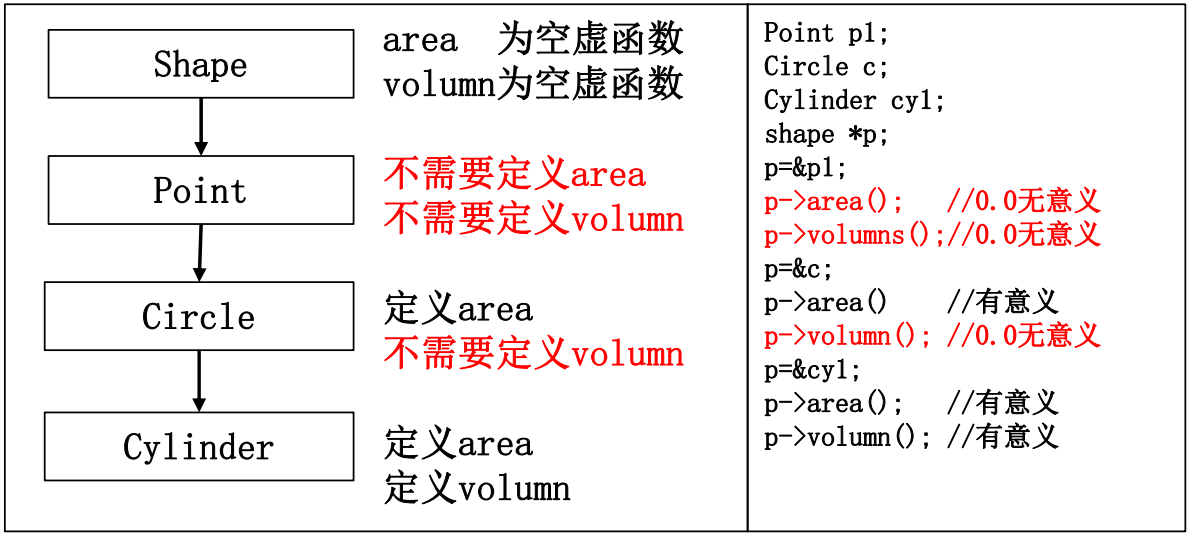


以此继承序列为例

★ area()为空虚函数，在Point中可不再定义

★ volumn()为空函数，在Point、Circle中可不再定义

★ 选择shapeName()为纯虚函数，为了声明抽象类，且shapeName()每个类中必须再次定义




```
Point p1;
Circle c;
Cylinder cy1;
shape *p;
p=&p1;
p->area();   //0.0无意义
p->volumns();//0.0无意义
p=&c;
p->area()    //有意义
p->volumn(); //0.0无意义
p=&cy1;
p->area();   //有意义
p->volumn(); //有意义
```