

## § 12. 多态性与虚函数

### 12.1. 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)

## § 12. 多态性与虚函数

### 12. 2. 多态性的引入

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

## § 12. 多态性与虚函数

### 12.2. 多态性的引入

例：Point类的定义、实现及测试函数

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

class Point {
    protected: //希望被子类继承
        float x, y;
    public:
        Point(float a=0, float b=0);
        void setPoint(float a, float b);
        float getX() const; //常成员函数, 不能修改值
        float getY() const; //常成员函数, 不能修改值
        friend ostream &operator<<
            (ostream &output, const Point &p);
};

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

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

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

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

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

int main()
{
    Point p(3.5f, 6.4f);
    cout << "x=" << p.getX() << ", y=" << p.getY() << endl;
    p.setPoint(8.5f, 6.8f);
    cout << "p(new):" << p << endl; //endl没必要

    return 0;
}
```

x=3.5, y=6.4  
p(new): [8.5, 6.8]

## § 12. 多态性与虚函数

### 12.2. 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: //书上P.395 private 错, 要被volumn继承
    float radius;
public:
    Circle(float x=0, float y=0, float r=0);
    void setRadius(float r);
    float getRadius() const;
    float area() const;
    friend ostream &operator<<
        (ostream &output, const Circle &c);
};

Circle::Circle(float x, float y, float r):
    Point(x,y), radius(r)
{
} //初始化表形式, 空函数体即可
void Circle::setRadius(float r)
{
    radius = r;
}
float Circle::getRadius() const
{
    return radius;
}
```

```
float Circle::area() const
{
    return 3.14159f*radius*radius;
}

ostream &operator<<(ostream &output, const Circle &c)
{
    output << "Center=[" << c.x << ", " <<c.y
        << "], r=" << c.radius << ", area="
        <<c.area() << endl;

    return output;
}

int main()
{
    Circle c(3.5f, 6.4f, 5.2f);
    cout << "original circle:\nx=" << c.getX()
        << ", y=" << c.getY()
        << ", r=" << c.getRadius()
        << ", area=" << c.area() << endl;
    c.setRadius(7.5f);
    c.setPoint(5.0f, 5.0f);
    cout << "new circle:\n" << c;
    Point &pRef=c; //派生类出现在需要基类的位置
    cout << "pRef:" << pRef; //输出Point的信息
    return 0;
}
```

```
original circle:
x=3.5, y=6.4, r=5.2, area=84.9486
new circle:
Center=[5, 5], r=7.5, area=176.714
pRef: [5, 5]
```

## § 12. 多态性与虚函数

### 12.2. 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:
    float height;
public:
    Cylinder(float x=0, float y=0, float r=0, float h=0);
    void setHeight(float h);
    float getHeight() const;
    float area() const;
    float volume() const;
    friend ostream &operator<<
        (ostream &output, const Cylinder &cy);
};

Cylinder::Cylinder(float x, float y, float r, float h):
    Circle(x, y, r), height(h)
{
    //初始化表形式，空函数体即可
}

void Cylinder::setHeight(float h)
{
    height = h;
}

float Cylinder::getHeight() const
{
    return height;
}

float Cylinder::area() const
{
    return 2*Circle::area()+2*3.14159f*radius*height;
}
```

```
float Cylinder::volume() const
{
    return Circle::area()*height;
}

ostream &operator<<(ostream &output, const Cylinder &cy)
{
    output<< "Center=[" << cy.x << ", " << cy.y << "], r="
        << cy.radius << ", h=" << cy.height << "\narea="
        << cy.area() << ", volume=" << cy.volume() << endl;
    return output;
}

int main()
{
    Cylinder cyl(3.5f, 6.4f, 5.2f, 10.0f);
    cout << "\noriginal cylinder:\nx=" << cyl.getX()
        << ", y=" << cyl.getY() << ", r=" << cyl.getRadius()
        << ", h=" << cyl.getHeight() << "\narea="
        << cyl.area() << ", volume=" << cyl.volume() << endl;
    cyl.setHeight(15.0f);
    cyl.setRadius(7.5f);
    cyl.setPoint(5.0f, 5.0f);
    cout << "\nnew cylinder:\n" << cyl;
    Point &pRef = cyl; //赋值兼容规则
    cout << "\npRef as a point:" << pRef;
    Circle &cRef = cyl; //赋值兼容规则
    cout << "\ncRef as a circle:" << cRef;
    return 0;
}
```

```
original cylinder:
x=3.5, y=6.4, r=5.2, h=10
area=496.623, volume=849.486
```

```
new cylinder:
Center=[5, 5], r=7.5, h=15
area=1060.29, volume=2650.72
```

```
pRef as a point:[5, 5]
//比书上多空一行!!!
cRef as a
circle:Center=[5, 5], r=7.5, area=176.714
```

## § 12. 多态性与虚函数

### 12.2. 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

## § 12. 多态性与虚函数

### 12. 3. 虚函数

#### 12. 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 (不是重载, 书P. 347 P. 374错)

## § 12. 多态性与虚函数

### 12.3. 虚函数

#### 12.3.2. 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)

```
Circle c(5.0, 5.0, 7.5);  
cout << c.area() << endl; //圆面积  
  
Cylinder cyl(3.5, 6.4, 5.2, 10.0);  
cout << cyl.area() << endl; //圆柱体表面积
```

```
Cylinder cyl(3.5, 6.4, 5.2, 10.0);  
cout << cyl.area() << endl; //圆柱体表面积  
  
Circle &cRef = cyl, *p=&cyl;  
cout << cRef.area() << endl; //圆面积  
cout << p->area() << endl; //圆面积
```

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)

```
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; //圆面积  
  
p = &cyl;  
cout << p->area() << endl; //圆柱体表面积
```

期望!!!  
目前不行



## § 12. 多态性与虚函数

### 12.3. 虚函数

#### 12.3.3. 虚函数的定义与使用

定义：在基类的函数定义前加virtual声明

```
如例12.1定义Point, Circle 和 Cylinder
int main()
{ Circle  c1 (3.5f, 6.3f, 5.0f);
  Cylinder cyl(3.5f, 6.3f, 5.0f, 10.0f);
  Circle  *p;
  p = &c1;
  cout << p->area() << endl; 78.5397
  p = &cyl;
  cout << p->area() << endl; 78.5397
  return 0;
}
```

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

不变

改动Circle的定义

```
class Circle:public point {
protected:
    float radius;
public:
    Circle(float x=0, float y=0, float r=0);
    void  setRadius(float r);
    float getRadius() const;
    virtual float area() const;
    friend ostream &operator<<
        (ostream &out, const Circle &c);
};
```

```
int main()
{ Circle  c1 (3.5f, 6.3f, 5.0f);
  Cylinder cyl(3.5f, 6.3f, 5.0f, 10.0f);
  Circle  *p;
  p = &c1;
  cout << p->area() << endl; 78.5397
  p = &cyl;
  cout << p->area() << endl; 471.238
  return 0;
}
```

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

## § 12. 多态性与虚函数

### 12.3. 虚函数

#### 12.3.3. 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>
using 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::
}
```

## § 12. 多态性与虚函数

### 12.3. 虚函数

#### 12.3.3. 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，基类指针变量 = &派生类对象形式。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;  
}
```

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

★ 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)

## § 12. 多态性与虚函数

### 12.3. 虚函数

#### 12.3.3. 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 float display();  
};
```

编译错误

```
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::  
}
```

- ★ 若派生类中无同名函数, 则自动继承基类



## § 12. 多态性与虚函数

### 12.3. 虚函数

#### 12.3.3. 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
}
```

## § 12. 多态性与虚函数

### 12.3. 虚函数

#### 12.3.3. 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>
using 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::

```
#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();
    p->show();
    p=&b;
    p->display();
    p->show();
    p=&c;
    p->display();
    p->show();
}
```

## § 12. 多态性与虚函数

### 12.3. 虚函数

#### 12.3.3. 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(float f) { cout << "B::f3()" << endl; }
    int f4() { cout << "B::f4()" << endl; }
    void f5(float 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？

## § 12. 多态性与虚函数

### 12. 3. 虚函数

#### 12. 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



## § 12. 多态性与虚函数

### 12.3. 虚函数

#### 12.3.3. 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()
{ 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()
{ 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, &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
}
```

## § 12. 多态性与虚函数

### 12. 3. 虚函数

#### 12. 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)

## § 12. 多态性与虚函数

### 12.3. 虚函数

#### 12.3.5. 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

```
class A {  
    ...  
};  
class B:public A {  
    private:  
        char *s;  
    public:  
        B() { s=new char[10]; }  
        ~B() { delete s; }  
};
```

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

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

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

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

## § 12. 多态性与虚函数

### 12.3. 虚函数

#### 12.3.5. 虚析构造函数

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)

## § 12. 多态性与虚函数

### 12.3. 虚函数

#### 12.3.5. 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

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

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

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

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

```
int main()  
{ A *pa = new B;  
    ...  
    delete pa; //调用B的析构函数,  
                释放s  
}
```

## § 12. 多态性与虚函数

### 12.3. 虚函数

#### 12.3.5. 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:
    virtual ~A() { cout << "A::" << endl; }
};

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

虚析构函数

基类指针方式

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

派生类对象/指针方式

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

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

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

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

普通析构函数

基类指针方式

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

派生类对象/指针方式

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

## § 12. 多态性与虚函数

### 12.4. 纯虚函数与抽象类

#### 12.4.1. 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

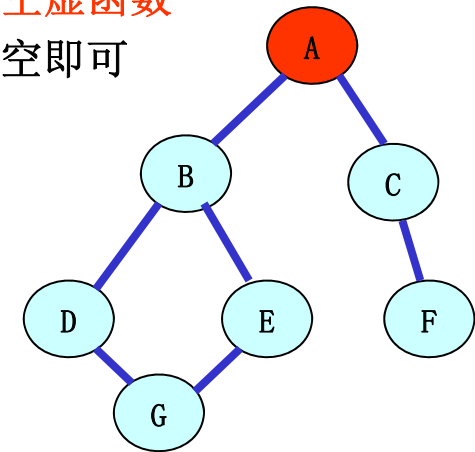
```
class A {  
    有实际意义的数据成员及函数  
    virtual void display() { }  
    virtual int show() { return 0; }  
};  
class B:public A {  
    有实际意义的数据成员及函数  
    void display() { 具体实现 }  
    int show() { 具体实现 }  
};
```

//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);
```

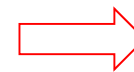


```
class A {  
    有实际意义的数据成员及函数  
    virtual void display() { }  
    virtual int show() { return 0; }  
};
```

//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  
}
```

★ 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



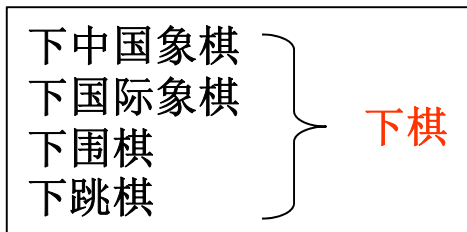
## § 12. 多态性与虚函数

### 12. 4. 纯虚函数与抽象类

#### 12. 4. 1. 空虚函数

#### 12. 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



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



## § 12. 多态性与虚函数

### 12.4. 纯虚函数与抽象类

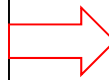
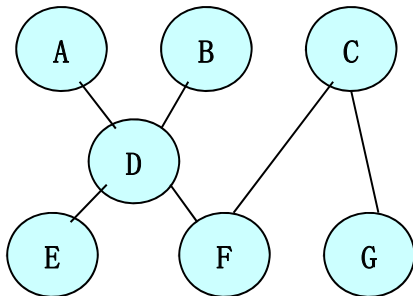
#### 12.4.1. 空虚函数

#### 12.4.2. 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

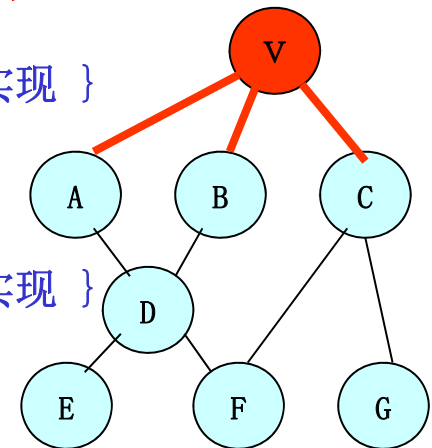
```
class A {  
    其它成员及函数  
    void display() { 具体实现 }  
};  
...  
class E:public D {  
    其它成员及函数  
    void display() { 具体实现 }  
};
```



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();
```



## § 12. 多态性与虚函数

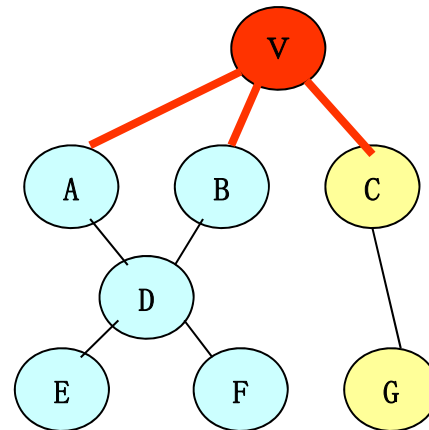
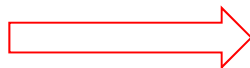
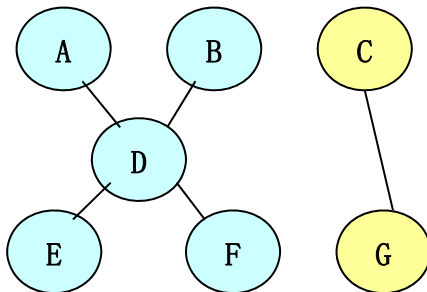
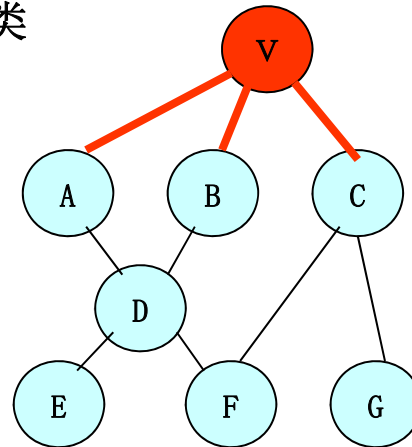
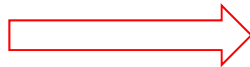
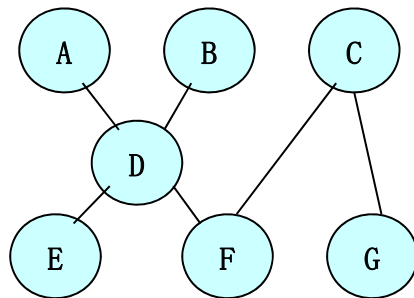
### 12.4. 纯虚函数与抽象类

#### 12.4.1. 空虚函数

#### 12.4.2. 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



## § 12. 多态性与虚函数

### 12.4. 纯虚函数与抽象类

#### 12.4.1. 空虚函数

#### 12.4.2. 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

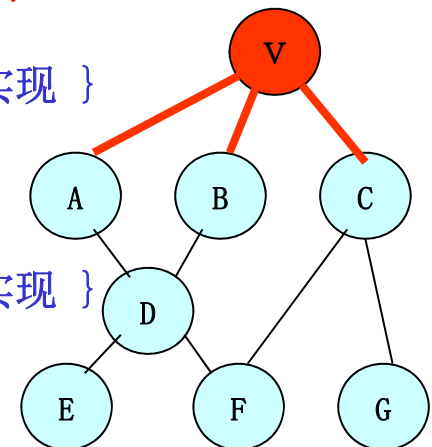
存在的问题：  
V还能定义对象 V v1;  
但V的对象实际无意义

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();
```



## § 12. 多态性与虚函数

### 12.4. 纯虚函数与抽象类

#### 12.4.2. 纯虚函数与抽象类

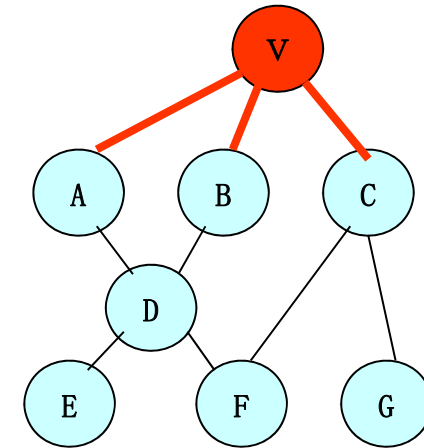
抽象类的定义：C++中无明确的关键字定义，只要声明某一成员函数为**纯虚函数**即可  
纯虚函数的声明：

```
virtual 返回类型 函数名(参数表) = 0;
```

例：

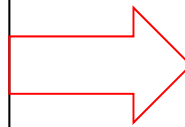
```
class V {  
    public:  
        virtual void display()=0;  
};
```

★ 表示该函数没有实际意义，也不被调用



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;  
};
```

## § 12. 多态性与虚函数

### 12.4. 纯虚函数与抽象类

#### 12.4.2. 纯虚函数与抽象类

抽象类的使用:

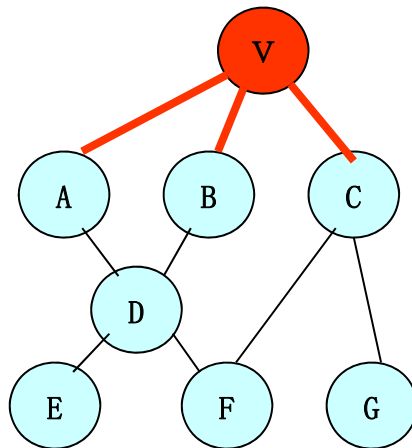
★ 抽象类不能有实例对象，但可用于声明指针或引用

```
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，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:  
        ...  
};
```

## § 12. 多态性与虚函数

### 12. 4. 纯虚函数与抽象类

#### 12. 4. 2. 纯虚函数与抽象类

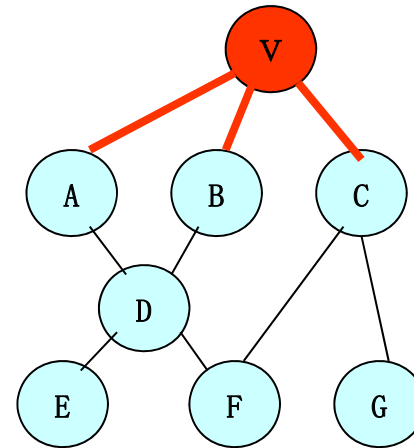
抽象类的使用：

★ 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

## § 12. 多态性与虚函数

### 12. 4. 纯虚函数与抽象类

#### 12. 4. 2. 应用实例(例12. 4 P. 407 - 413)

P. 411 - 413 总结性说明

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

★ 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

