第九章 多态性与虚函数

主讲教师：同济大学计算机科学与技术学院 陈宇飞
同济大学计算机科学与技术学院 龚晓亮

目录

- 多态的基本概念
- 虚函数的定义和使用
- 虚析构函数
- 纯虚函数与抽象类

目录

- 多态的基本概念
  - 多态的定义
  - 静态联编和动态联编
  - 多态的应用

1.1 多态的基本概念

赋值兼容规则+指针或引用+虚函数==多态

- 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
- 多态公有继承：
  - 同一个方法在派生类和基类中的行为不同，具体取决于调用该方法的对象

1.2 静态联编和动态联编

- 静态联编和动态联编
  - 将源代码中的函数调用解释为执行特定函数代码块被称为函数名联编
  - 编译器在编译过程中进行的联编称为静态联编
  - 编译器在程序运行时选择正确的虚方法代码，称为动态联编
- 多态的分类：
  - 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)

1.3 多态的应用

- 引例：

由于之前在俱乐部的工作经历，您成为了某银行的首席程序员。银行要求您完成的第一项工作是开发两个类：

① Brass类：基本账户

② BrassPlus类：添加透支保护特性的账户：如果用户签出超出其存款余额的支票且超出额不大，银行将支付支票且对超出部分收取额外费用并追加罚款

➢ 问题1：应从Brass公有派生出BrassPlus吗？

✓ 正确：BrassPlus类满足is-a条件。都将保存客户姓名、账号以及结余，都可以存款、取款和显示信息

1.3 多态的应用

- 引例：

由于之前在俱乐部的工作经历，您成为了某银行的首席程序员。银行要求您完成的第一项工作是开发两个类：

① Brass类：基本账户

② BrassPlus类：添加透支保护特性的账户：如果用户签出超出其存款余额的支票且超出额不大，银行将支付支票且对超出部分收取额外费用并追加罚款

➢ 问题2：BrassPlus是否需要添加新数据成员？

✓ 正确：BrassPlus类包含Brass所有信息及如下信息：透支上限、透支贷款利率，当前的透支总额

1.3 多态的应用

- 引例：

由于之前在俱乐部的工作经历，您成为了某银行的首席程序员。银行要求您完成的第一项工作是开发两个类：

① Brass类：基本账户

② BrassPlus类：添加透支保护特性的账户：如果用户签出超出其存款余额的支票且超出额不大，银行将支付支票且对超出部分收取额外费用并追加罚款

➢ 问题3：BrassPlus是否需要新增操作？

✓ 不需要，但有两种操作的实现不同！！！ ➡ 虚函数

a）对于取款操作，必须考虑透支保护

b）显示操作必须显示BrassPlus账户的其他信息

```
// brass.h -- bank account classes
class Brass{
private: …
public: …
    virtual void Withdraw(double amt);
    virtual void ViewAcct() const;
    virtual ~Brass() {}   //虚析构函数（模块9.3）
};
class BrassPlus : public Brass{
private:  //新增数据成员
    double maxloan;
    double rate;
    double owesBank;
public: …
    virtual void ViewAcct() const;
    virtual void Withdraw(double amt);
};
```

头文件
类的声明

完整程序见primer书

```
//brass.cpp -- bank account methods
#include"brass.h"
#include <iostream>
using namespace std;
…
void Brass::Withdraw(double amt){…}
void Brass::ViewAcct()const{…}

//redefine how Withdraw() and ViewAcct() work
void BrassPlus::Withdraw(double amt){
    …//取款操作，考虑透支保护
}
void BrassPlus::ViewAcct()const{
    …//显示操作，显示BrassPlus账户的其他信息
}
```

源程序文件
函数的实现

完整程序见primer书

```
//usebrass1.cpp -- testing bank account classes
#include <iostream>
#include "brass.h"
using namespace std;
int main()
{
    Brass Piggy("Porcelot Pigg", 381299, 4000.00);
    BrassPlus Hoggy("Horatio Hogg", 382288, 3000.00);

    Piggy.ViewAcct();
    Hoggy.ViewAcct();
    …
    Piggy.Withdraw(4200.00);
    Hoggy.Withdraw(4200.00);
    Hoggy.ViewAcct();
    …
    return 0;
}
```

源程序文件
调用函数

//方法是通过对象调用的，
并没有使用虚方法特性！

完整程序见primer书

```
//usebrass2.cpp -- polymorphic example
#include <iostream>
#include "brass.h"
const int CLIENT = 4;
using namespace std;
int main(){
    Brass * p_client[CLIENT]; //定义指向基类Brass的指针数组
    …
    for (i = 0; i < CLIENTS; i++){
        …
        if (kind == '1')
            p_clients[i] = new Brass(temp, tempnum, tempbal);
        else{  …
            p_clients[i] = new BrassPlus(temp, tempnum, tempbal, tmax, trate);
        }
    }
    for (i = 0; i < CLIENTS; i++){
        p_clients[i]->ViewAcct();  //多态
        cout << endl;
    }
    …
}
```

源程序文件
调用函数

完整程序见primer书

//usebrass2.cpp　程序分析

回顾：
赋值兼容规则+指针或引用+虚函数==多态

- Brass * p_client[CLIENT];

同时管理Brass和BrassPlus账户：定义指向基类Brass的指针数组，Brass指针既可以指向Brass对象，也可以指向BrassPlus对象。因此，可以使用一个数组来表示多种类型的对象（多态性）

- p_clients[i]->ViewAcct();

如果数组成员指向的是Brass对象，则调用Brass::ViewAcct();如果指向的是BrassPlus对象，则调用BrassPlus::ViewAcct()

如果Brass::ViewAcct()未被声明为虚的，
则在任何情况下都将调用Brass::ViewAcct()

目录

- 虚函数的定义和使用
  - 虚函数的定义
  - 虚函数的使用
  - 支配规则、赋值兼容规则、虚函数

2.1 虚函数的定义

- 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

```cpp
#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;
     }
};
```

```cpp
//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::
}
```

```cpp
//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::
}
```

2.2 虚函数的使用

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
- 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();
};
void A::display()
{
   cout << "A::display()" << endl;
}
```

2.2 虚函数的使用

- 只需要在最开始的基类中加virtual声明，后续派生类可以不加

```
class A {
public:
    virtual void display(); //必须加
};
class B:public A {
public:
    virtual void display(); //建议加，但可以不加
}
```

2.2 虚函数的使用

- 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。若同名虚函数仅返回类型不同，则编译报错

```
class A {     //编译报错
public:
     virtual void display();
};
class B:public A {
public:
     virtual int display();
};
```

- 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::
}
```

- 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;
   }
};
```

```
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
}
```

- 对于派生类中的其它非virtual仍适用赋值兼容规则

```
#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::
}
```

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

2.3 支配规则、赋值兼容规则、虚函数

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

```
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
}
```

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

```
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
}
```

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

```
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
}
```

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

```
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
}
```

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

```
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
}
```

目录

- 虚析构函数

虚析构函数引例：

```
// brass.h -- bank account classes
class Brass{
    …
    virtual ~Brass() {}    //虚析构函数
};
class BrassPlus : public Brass{
    …
};
```

- 如果析构函数不是虚的，则将只调用对应于指针类型的析构函数。即便指针指向BrassPlus对象，也只有Brass的析构函数被调用
- 如果析构函数是虚的，将调用相应对象类型的析构函数。若指针指向BrassPlus对象，将调用BrassPlus的析构函数（上例系统默认），再自动调用基类Brass的析构函数

3.1 虚析构函数

- 虚析构函数的作用：
  - 确保正确的析构函数调用顺序。解决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
- 虚析构函数的使用：
  - 适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
  - 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，无论是否包含动态申请空间的操作，都建议将析构函数声明为虚析构函数
  - 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，因此构造函数不能声明为虚函数

```
class Base {
public:
    Base(){}
    ~Base(){cout << "Base destructor is called!" << endl;} //普通析构函数
};
class Subclass:public Base{
private: int *ptr;
public:
    Subclass(){ptr = new int;}
    ~Subclass(){
        cout << "Subclass destructor is called!" <<endl; delete ptr;
    }
};
int main(){
    Base * a = new Subclass;  //基类指针方式
    delete a;          //Base析构
    return 0;          //ptr分配的内存没有得到释放
}
```

```
Base destructor is called!
```

```
class Base {
public:
    Base(){}
    ~Base(){cout << "Base destructor is called!" << endl;} //普通析构函数
};
class Subclass:public Base{
private: int *ptr;
public:
    Subclass(){ptr = new int;}
    ~Subclass(){
        cout << "Subclass destructor is called!" <<endl; delete ptr;
    }
};
int main(){
    Subclass * a = new Subclass;  //派生类对象/指针方式
    delete a;       //Subclass析构
    return 0;       //Base析构
}
```

```
Subclass destructor is called!
Base destructor is called!
```

```
class Base {
public:
    Base(){}
    virtual ~Base(){cout << "Base destructor is called!" << endl;} //虚析
};                                                                  构函数
class Subclass:public Base{
private: int *ptr;
public:
    Subclass(){ptr = new int;}
    ~Subclass(){
        cout << "Subclass destructor is called!" <<endl; delete ptr;
    }
};
int main(){
    Base * a = new Subclass;  //基类指针方式
    delete a;          //Subclass析构
    return 0;          //Base析构
}
```

```
Subclass destructor is called!
Base destructor is called!
```

```
class Base {
public:
    Base(){}
    virtual ~Base(){cout << "Base destructor is called!" << endl;} //虚析
};                                                                  构函数
class Subclass:public Base{
private: int *ptr;
public:
    Subclass(){ptr = new int;}
    ~Subclass(){
        cout << "Subclass destructor is called!" <<endl; delete ptr;
    }
};
int main(){
    Subclass * a = new Subclass;  //派生类对象/指针方式
    delete a;          //Subclass析构
    return 0;          //Base析构
}
```

```
Subclass destructor is called!
Base destructor is called!
```

目录

- 纯虚函数与抽象类
  - 基本概念
  - 抽象类的定义
  - 抽象类的使用
  - 空虚函数
  - 应用实例

引例思考：

假设您正在开发一个图形程序，该程序会显示圆和椭圆等

问题：应从Ellipse派生出Circle吗？

✓满足is-a条件：圆是椭圆的特殊情况

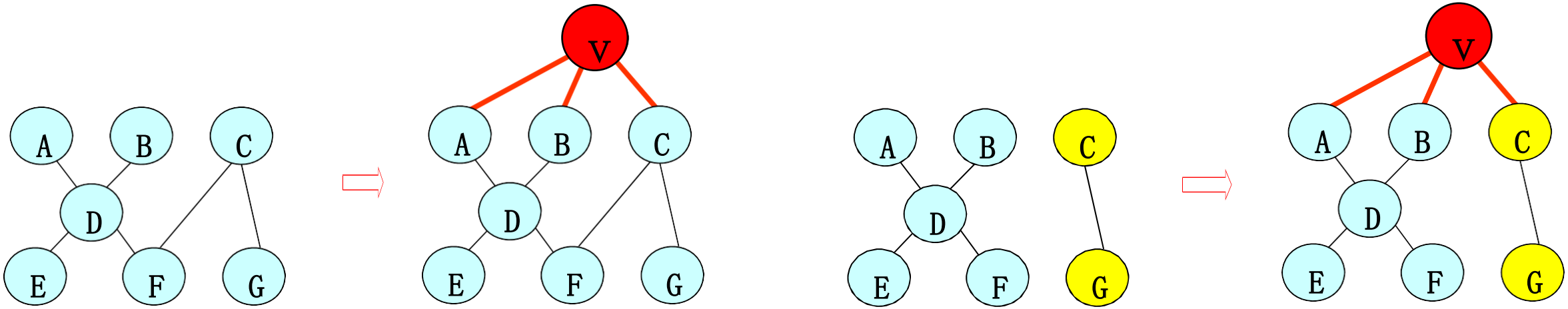
✓派生是笨拙的：圆只需要一个半径就可以描述大小和形状，不需要长轴和短轴，因此有信息冗余

✓更好的解决办法：

- 从Ellipse和Circle类抽象出共性，放到抽象类（abstract base class, ABC）中，然后从ABC派生出Ellipse和Circle类
- 方便使用基类指针同时管理Ellipse和Circle对象

4.1 基本概念

- 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

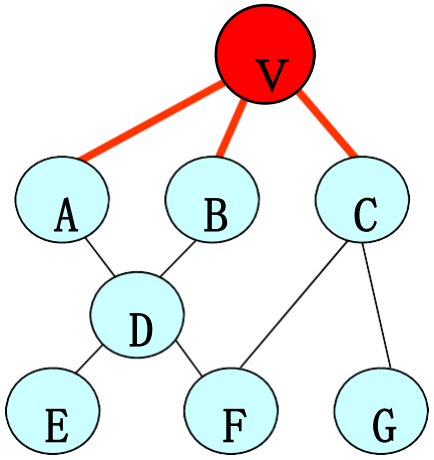


4.1 基本概念

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();
```



- 思考：

1）V是否还能定义对象？

可以： V v1;

2）是否有意义？

V的对象实际无意义

3）如何解决？

将V定义为抽象类

4.2 抽象类的定义

- 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，表示该函数没有实际意义，也不被调用
- 纯虚函数的声明：
  - virtual 返回类型 函数名(参数表) = 0;

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

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

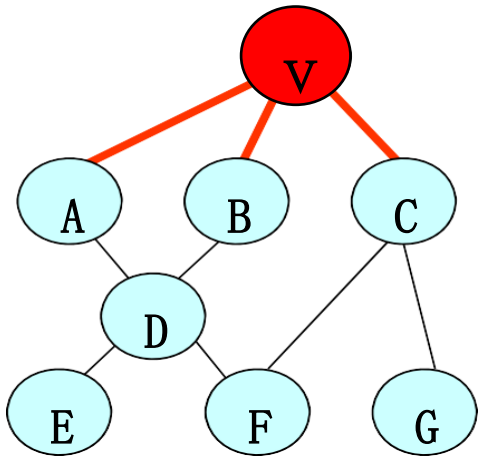
4.3 抽象类的使用

➢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)
```

4.3 抽象类的使用

➢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)

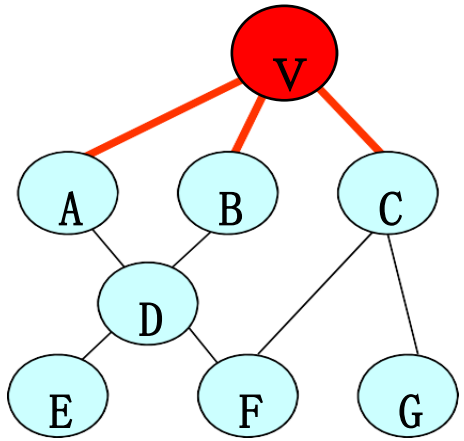



```
//假设A,B,C中都有int a,b成员
class A {
  protected: int a,b;
};
class B {
  protected: int a,b;
};
class C {
  protected: int a,b;
};
```

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

4.3 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)



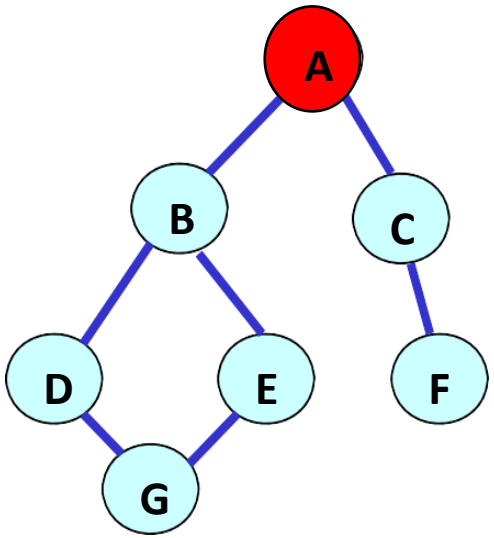
若：
```
class V {
public:
    virtual void display()=0;
};
```
则：ABC中的display()必须定义，即使B中不需要display
```
class B:public V {
public:
    void display() { }
};
```

4.4 空虚函数

- 空虚函数的概念：
  - 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
- 空虚函数的形式：
  - 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

4.4 空虚函数

- 示例：




```
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();
pa = &b;
pa->display();
pa->show();
```

4.4 空虚函数

➢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(); //虚函数形式
}
```

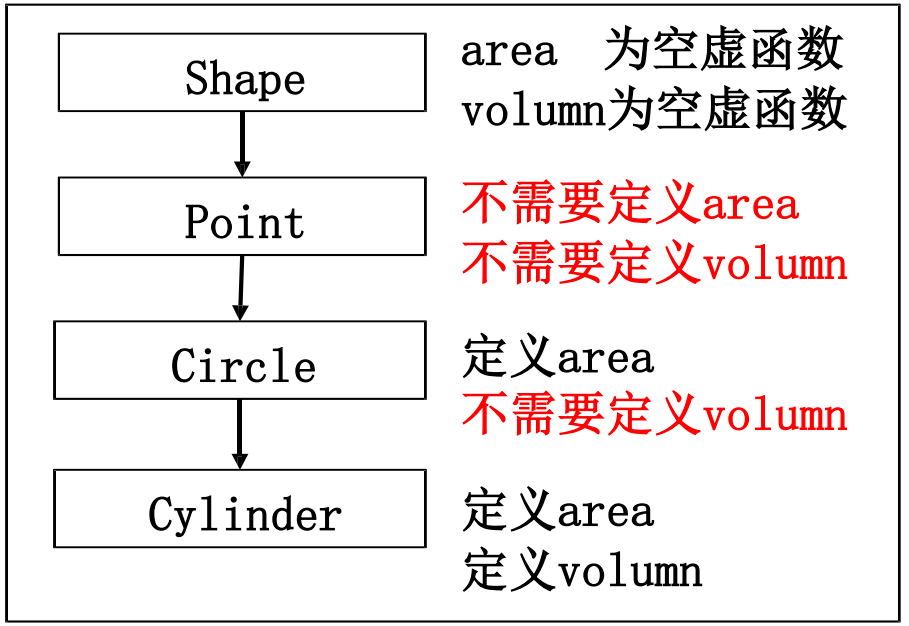
4.4 空虚函数

- 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

4.5 应用实例

```
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()为空虚函数，在Point、Circle中可不再定义
- 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



总结

- 多态的基本概念（掌握）
  - 多态的定义
  - 静态联编和动态联编
  - 多态的应用
- 虚函数的定义和使用（熟悉）
  - 虚函数的定义
  - 虚函数的使用
  - 支配规则、赋值兼容规则、虚函数
- 虚析构函数（熟悉）
- 纯虚函数与抽象类（了解）
  - 基本概念
  - 抽象类的定义
  - 抽象类的使用
  - 空虚函数
  - 应用实例