第10章　多继承类和虚基类

10.1 多继承

单继承是多继承的一种特例，多继承具有更强的类型表达能力。

多继承派生类有多个基类或虚基类。同一个类不能多次作为某个派生类的直接基类，但可多次作为一个派生类的间接基类(同编译程序相关)。

class QUEUE: STACK, STACK{}; //错误，出现两次

多继承派生类继承所有基类的数据成员和函数成员。

多继承派生类在继承基类时，各基类可采用不同的派生控制符。

基类之间的成员可能同名，基类与派生类的成员也可能同名。在出现同名时，如面向对象的作用域不能解析，可使用基类类名加作用域运算符::来指明要访问基类的成员。

- Java、C#、SmallTalk等单继承语言在描述多继承的对象时，常常通过对象成员委托（代理）实现多继承。

```
class A{ public： void f( ) {}};
class B{ public： void g( ) {}};
```

如果需要实现一个类，同时具有类A和类B的行为，但又不能多继承怎么办（如JAVA）？

采用代理模式，继承一个类，将另外一个类的对象作为数据成员

```
class C: public A{
    B b; //B类行为的代理
public:
    void g( ) { b.g( ): } //定义一个同名的g函数，但其功能
                          //委托对象b完成(通过调用b.g())，因此
                  //C::g的行为与B::g完全一致
    //A::f被C继承
};
//这样C就具有A的行为f和B的行为g,达到了多重继承的效果
```

10.1 多继承

委托代理在多数情况下能够满足需要，但当对象成员和基类物理上是同一个基类时（存在一个共同的基类），就可能对同一个物理对象重复初始化(可能是危险的和不必要的)。

两栖机车AmphibiousVehicle继承基类陆用机车LandVehicle，委托对象成员水上机车WaterVehicle完成水上功能。两栖机车可能对同一个物理对象Engine初始化(启动)两次。

```
class Engine{ /*...*/};
class LandVehicle: Engine{/*...*/};
class WaterVehicle: Engine{/*...*/};
class AmphibiousVehicle: LandVehicle{WaterVehicle  wv; };
```

10.1 多继承

用多继承方式定义派生类

1. 多继承派生类的定义：

class 派生类名:<派生方式> 基类1,<派生方式> 基类2,…{

　　<类体>

};

2. 同样存在派生类对象多次初始化同一(物理)基类对象问题。

public
protected
private

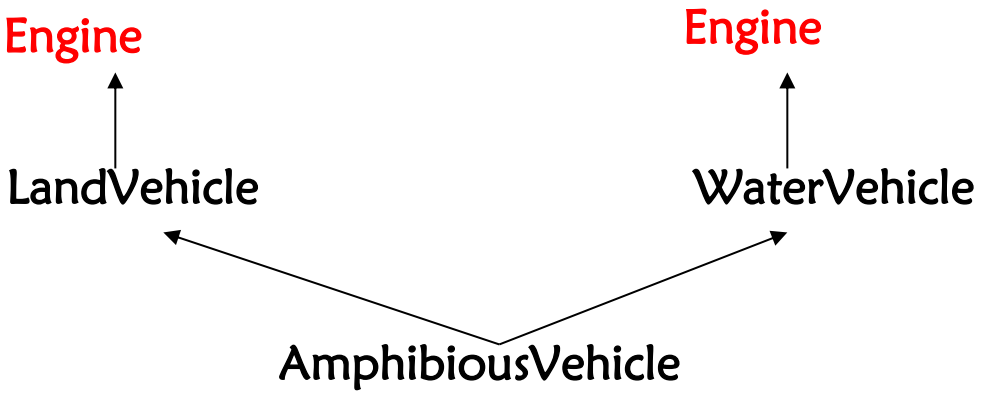
10.1 多继承

C++提供多继承机制描述两栖机车AmphibiousVehicle：

class AmphibiousVehicle：LandVehicle，WaterVehicle{};

仅靠多继承仍然不能解决同一个物理对象初始化两次的问题。

可以采用全局变量、静态数据成员做标记，解决同一个物理对象初始化两次的问题；此外，还需要解决同一物理对象两次析构问题，这样会使程序的逻辑变得复杂化。



上述定义存在的问题：两栖机车要安装两个引擎Engine，可引入虚基类解决该问题。

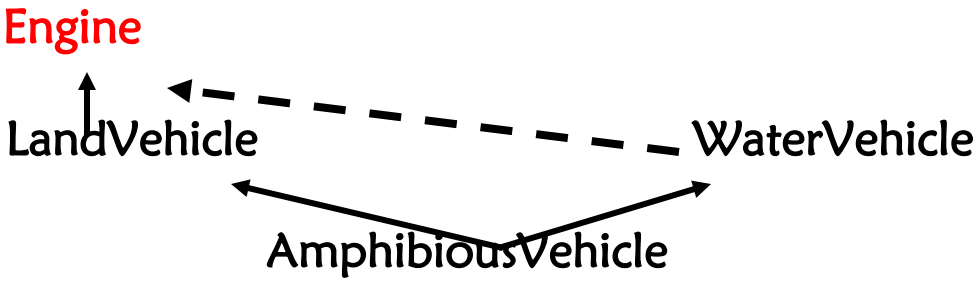
10.1 虚基类

虚基类用virtual声明，可把多个逻辑虚基类对象映射成同一个物理虚基类对象。

映射成的这个物理虚基类对象尽可能早的构造、尽可能晚的析构，且构造和析构都只进行一次。

若虚基类的构造函数都有参数，必须在派生类构造函数的初始化列表中列出虚基类的构造实参的值。

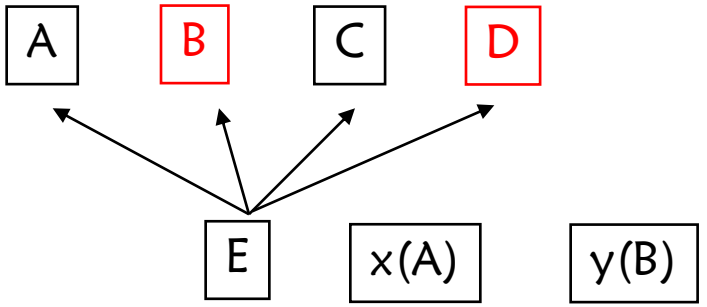
```
class Engine{ /*...*/ };
class LandVehicle: virtual public Engine{ /*...*/ };
class WaterVehicle: public virtual Engine{ /*...*/ };
class AmphibiousVehicle: LandVehicle, WaterVehicle{ };
```



10.2 虚基类

同一棵派生树中的同名虚基类，共享同一个存储空间；其构造和析构仅执行1次，且构造尽可能最早执行，而析构尽可能最晚执行。由派生类（根）、基类和虚基类构成一个派生树的节点，而对象成员将成为一棵新派生树的根。

```
struct A {};
struct B{};
struct C{};
struct D{};
struct E: A, virtual B, C, virtual D{
      A x;
      B y;
};
```



```
class Engine{ int power;  public：Engine (int  p)：  power (p)  {  } };
class LandVehicle：  virtual  public  Engine{
            int  speed;
public: //如从AmphibiousVehicle调LandVehicle, 则此处不会调用Engine (p)
            LandVehicle (int  s,  int  p)：Engine (p),  speed (s) { } //必须调用Engine (p)
};
class  WaterVehicle：public  virtual  Engine{
            int  speed;
public: //如从AmphibiousVehicle调WaterVehicle, 则此处不会调用Engine (p)
            WaterVehicle (int s,  int p)：speed (s), Engine (p) { } //必须调用Engine (p)
};
struct AmphibiousVehicle：LandVehicle,  WaterVehicle {
            AmphibiousVehicle (int  s1,  int  s2,  int  p) ：  //先构造虚基类再基类
            WaterVehicle (s2,  p),  LandVehicle (s1,  p),  Engine (p) {} //必须调用
                                                                  //Engine (p)
   //在整个AmphibiousVehicle派生树中，Engine (p) 只执行1次
}; //初始化顺序：  Engine (p) ,  LandVehicle (s1,  p) ,  WaterVehicle (s2,  p)
```

10.2 虚基类

- 虚拟基类特殊的初始化语义
- 在非虚拟派生中，派生类只能显式初始化其直接基类
- 而虚拟基类的初始化则成了每一级派生类的责任

```
class A{
        string msg;
public:
        A(string s):msg(s) { cout << "Constructor:" << msg << endl; }
};
class B:virtual public A{
        int b;
public:
        B(string s, int i):A(s),b(i){}
};
class C:virtual public A{
        int c;
public:
        C(string s, int j):A(s),c(j){}
};
class D:public B,public C{
        int d;
public:
        D(string s, int x, int y,int z):A(s),B(s,x),C(s,y),d(z){}
};
void main(int argc, char* argv[])
{
        B b("B", 1);          //B为最终派生类，由它调用虚基类A的构造函数
        C c("C", 2);          //C为最终派生类，由它调用虚基类A的构造函数
        D d("D",1,2,3);       //D为最终派生类，由它调用虚基类A的构造函数，
                              //B和C的构造函数不再调虚基类A的构造函数
}
```

10.2 虚基类

虚基类和基类同名必然会导致二义性访问，编译程序会对这种二义性访问报错。当出现这种情况时，可用作用域运算符限定要访问的成员。

有虚基类的派生类构造函数不能使用constexpr定义

10.3 派生类成员

当派生类有多个基类或虚基类时，不同基类或虚基类的成员之间可能出现同名；派生类和基类或虚基类的成员之间也可能出现同名。

出现上述同名问题时，必须通过面向对象的作用域解析，或者用类名加作用域运算符::指定要访问的成员，否则就会引起二义性问题。

当派生类成员和基类成员同名时，优先访问作用域小的成员，即优先访问派生类的成员。当派生类数据成员和派生类函数成员的参数同名时，在函数成员内优先访问函数参数。

```
struct A{
   int a,  b,  c,  d;
};
struct B{
   int b,  c;
protected:
   int e;
};
class C: public A,  public B{
   int a;
public:
   int  b;    int f (int c)   ;
} x;
```

```
int C::f (int c)   {
   int i=a;          //访问C::a
   i=A::a;
   i=b+c+d;        //访问C::b和参数c
   i=A::b+B::b;  //访问基类成员
   return A::c;
}
void main (void)   {
   in
   i=
   i=
   i=
   i=
}
```

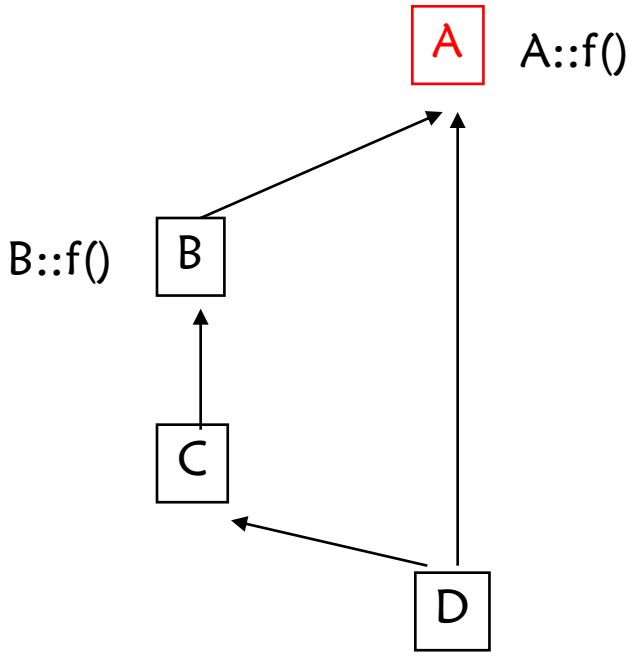
C的成员有：

```
private:
int C::a;
protected:
int B::e;
public:
int A::a, A::b,A::c,A::d;
int B::b,B::c;
int C::b, C::f()
```

```
struct A{
        void f() { cout << "A\n"; }
};
struct B:virtual A{
        void f() { cout << "B\n"; }
};
struct C:B{};
struct D:C,virtual A {};

void main(int argc, char* argv[])
{
        D d;
        B *pb = &d;                    //B和D之间为父子关系
        D *pd = &d;

        pb->f();                       //调用B::f(). B::f()被优先访问
        pd->f();              //调用B::f().pb为D类型指针,D分别从A和B继承了函
                //数成员f，但优先访问基类B的f（基类比虚基类作用域小）
}
```

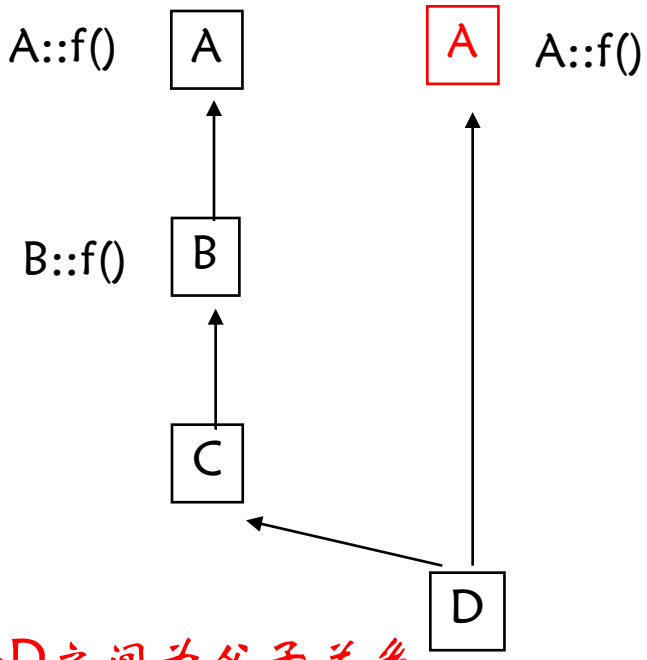


```
struct A{
        void f() { cout << "A\n"; }
};
struct B:A{
        void f() { cout << "B\n"; }
};
struct C:B{};
struct D:C,virtual A {};

void main(int argc, char* argv[])
{
        D d;
        B *pb = &d;
        D *pd = &d;

        pb->f();
        pd->f();

}
```



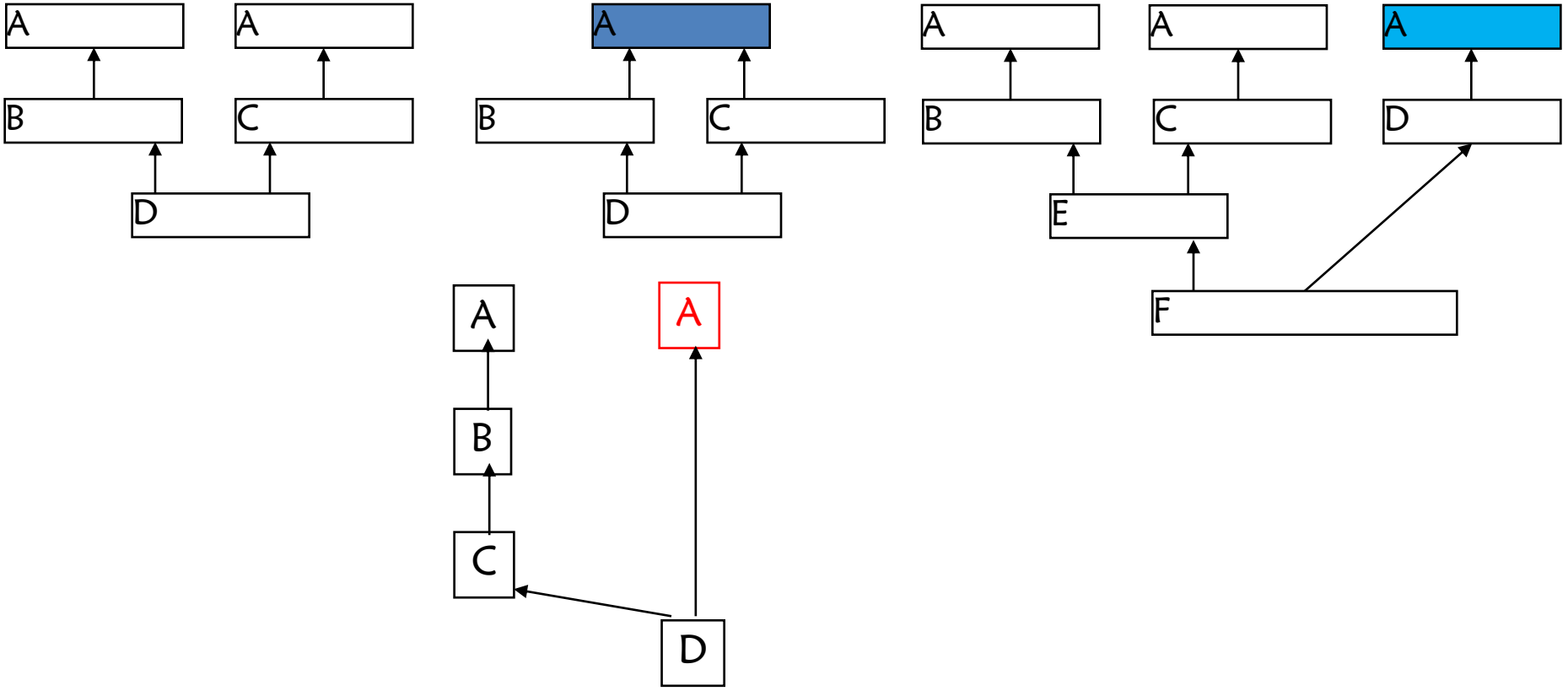
//B和D之间为父子关系

//OK,优先访问B::f()

//有二义性，f可能是基类A的，可能是基类B的，从
//这里可以看出，对于D而言，A和B作用域大小一样，
//都是基类作用域(左边分支)
//改成pd->A::f();或pd->B::f();

多重继承的内存布局（Memory Layout）

- 一个类不能多次成为一个派生类的直接基类（即使是虚基类）
  - 否则会报编译错误
- 一个类可以多次成为一个派生类的间接基类（即使不是虚基类）
- 一个类（A）可以同时成为一个派生类的直接基类和间接基类，但该类（A）作为直接基类时必须是虚基类。
  - 否则会报警告错误



10.4 构造与析构

- 析构和构造的顺序相反，派生类对象的构造顺序：

  (1)按定义顺序构造倒派生树中所有虚基类；

  (2)按定义顺序构造派生类的所有直接基类；

  (3)按定义顺序构造(初始化)派生类的所有数据成员，包括对象成员、const成员和引用成员；

  (4)执行派生类自身的构造函数体。

- 如果构造中虚基类、基类、对象成员、const及引用成员又是派生类，则派生类对象重复上述构造过程，但同名虚基类对象在同一棵派生树中仅构造一次。

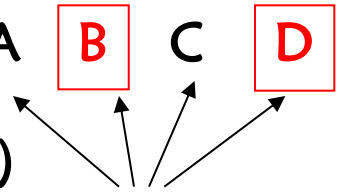
- 由派生类(根)、基类和虚基类构成一个派生树的节点，而对象成员将成为一棵新派生树的根。

多继承派生类的构造过程

```
#include <iostream.h>
struct A {
   A( ) { cout<<'A';}
};
struct B {
   B( ) { cout<<'B';}
};
struct C {
   int a;   int &b;
   const int c;
   C(char d): c(d), a(d) ,b(a)
   { cout<<d; }
};
struct D{
   D( ) { cout<<'D';}
};
```

```
struct E: A, virtual B, C, virtual D{
   A x, y;
   B z;
   E( ):z( ), y( ), C('C')
   {
      cout<<'E';
   }
};

void main(void)
{
   E  e;
}
```



{x(A) y(A) z(B)}

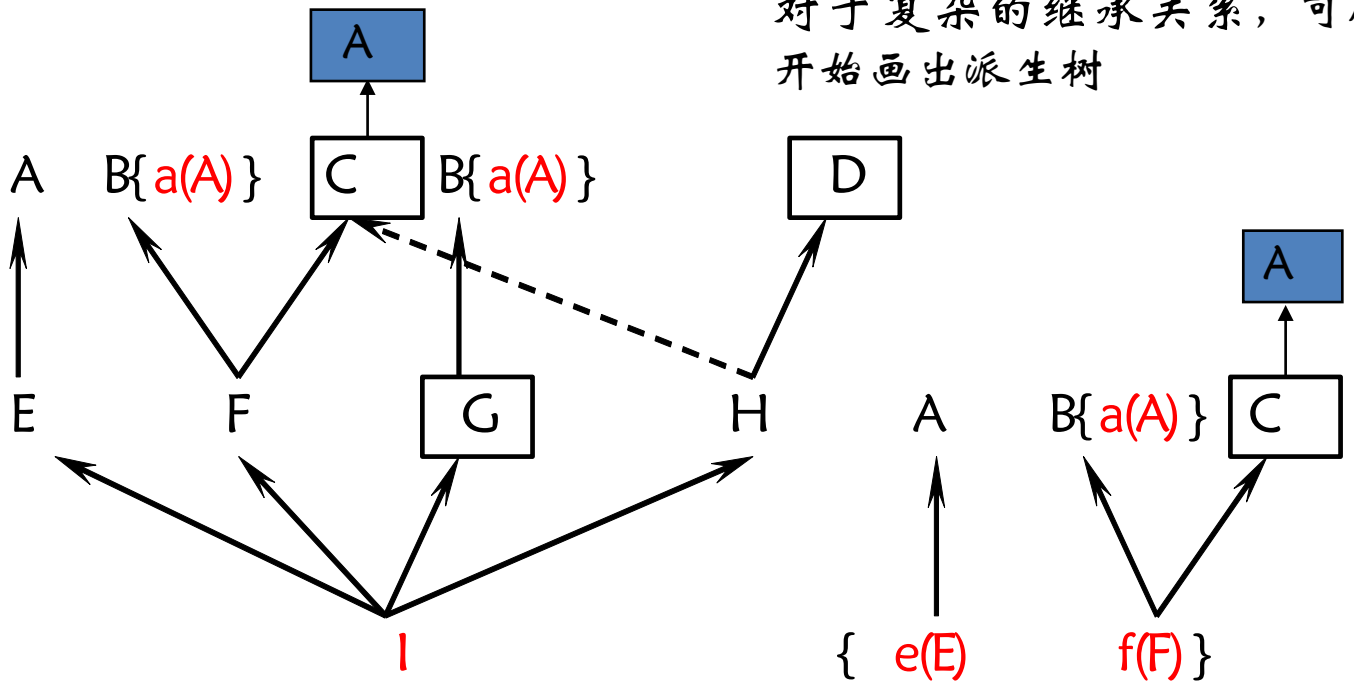
输出：

BDACAABE

B→D→A→C→x(A)→y(A)→z(B)→E

```
#include <iostream.h>
struct A{ A ( )  { cout<<'A'; } };
struct B { const A a; B ( ) { cout<< 'B' ; } }; //a作为新根
   B( ):a( ){}
struct C: virtual A{ C ( )  { cout<<'C'; } };
struct D{ D ( )  { cout<<'D'; } };
struct E: A{ E ( ) { cout<< 'E' ; } }; //等价E( ):A( ){…}
struct F: B, virtual C{ F ( ) { cout<<'F'; } };
struct G: B{ G ( ):B( ) { cout<<'G'; } };
struct H: virtual C, virtual D{ H ( ) { cout<<'H'; } };
struct I: E, F, virtual G, H{
   E e;  F f; //对象成员e、f作为新根
   I ( ) { cout<<'I'; } };
void main (void)  { I i; }
```

对于复杂的继承关系，可从最终派生类开始画出派生树



首先确定四个虚基类的构造顺序。从派生类I倒推：

要构造I，必须按定义顺序依次构造E、F、G、H。

1：E没有虚基类不予考虑；

2：构造F：按派生顺序依次构造虚基类A、C，对应输出为AC；

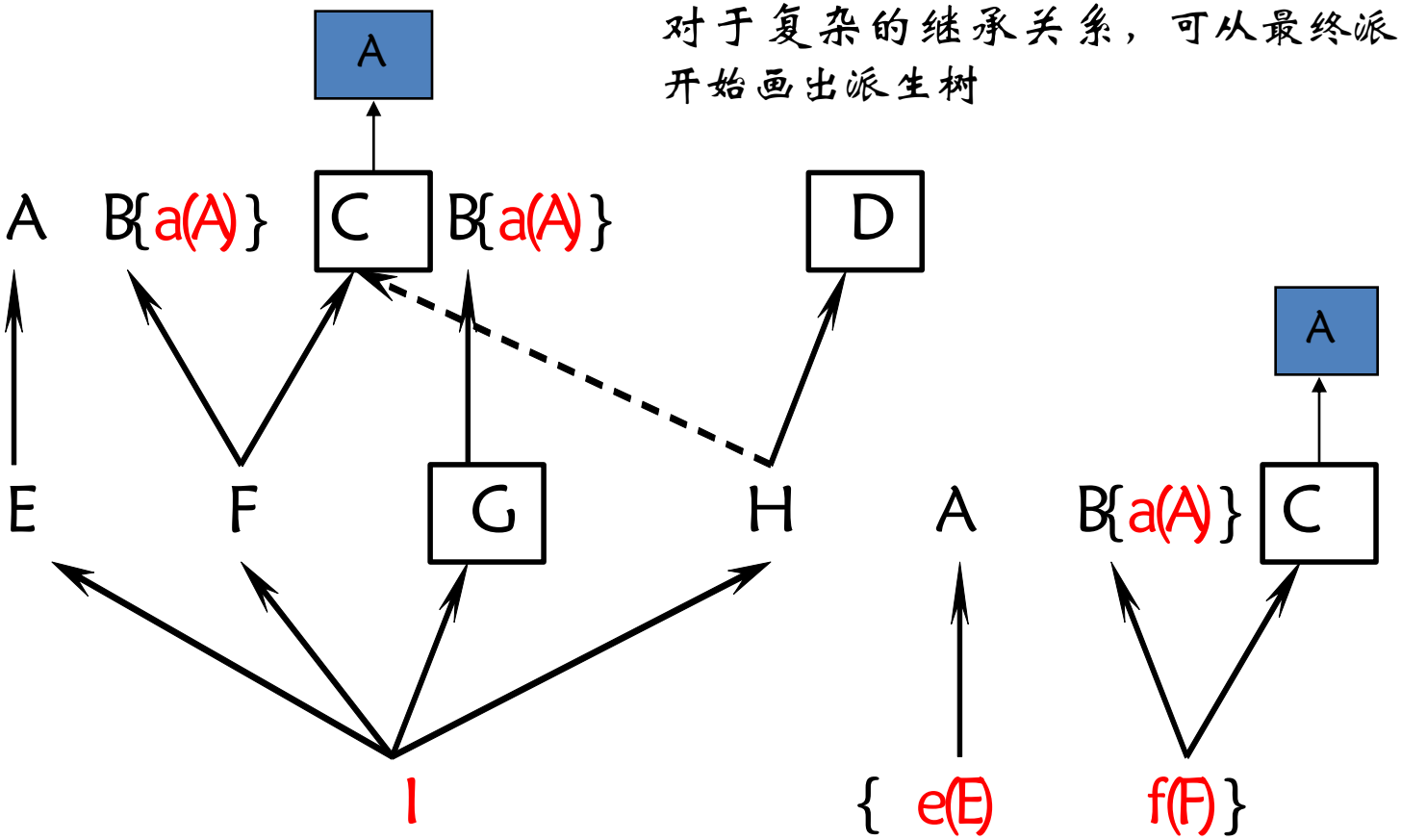
3：构造虚基类G：先构造G的基类B。构造类B必须先构造A类成员a，再调用B的构造函数体。因此对应输出为ABG；

4：构造H，必须先构造虚基类D，对应输出为D。

因此，四个虚基类构造完毕后，输出为ACABGD

按定义顺序自左至右、自下而上地构造倒派生树中所有虚基类；

对于复杂的继承关系，可从最终派生类开始画出派生树



接着构造I的基类

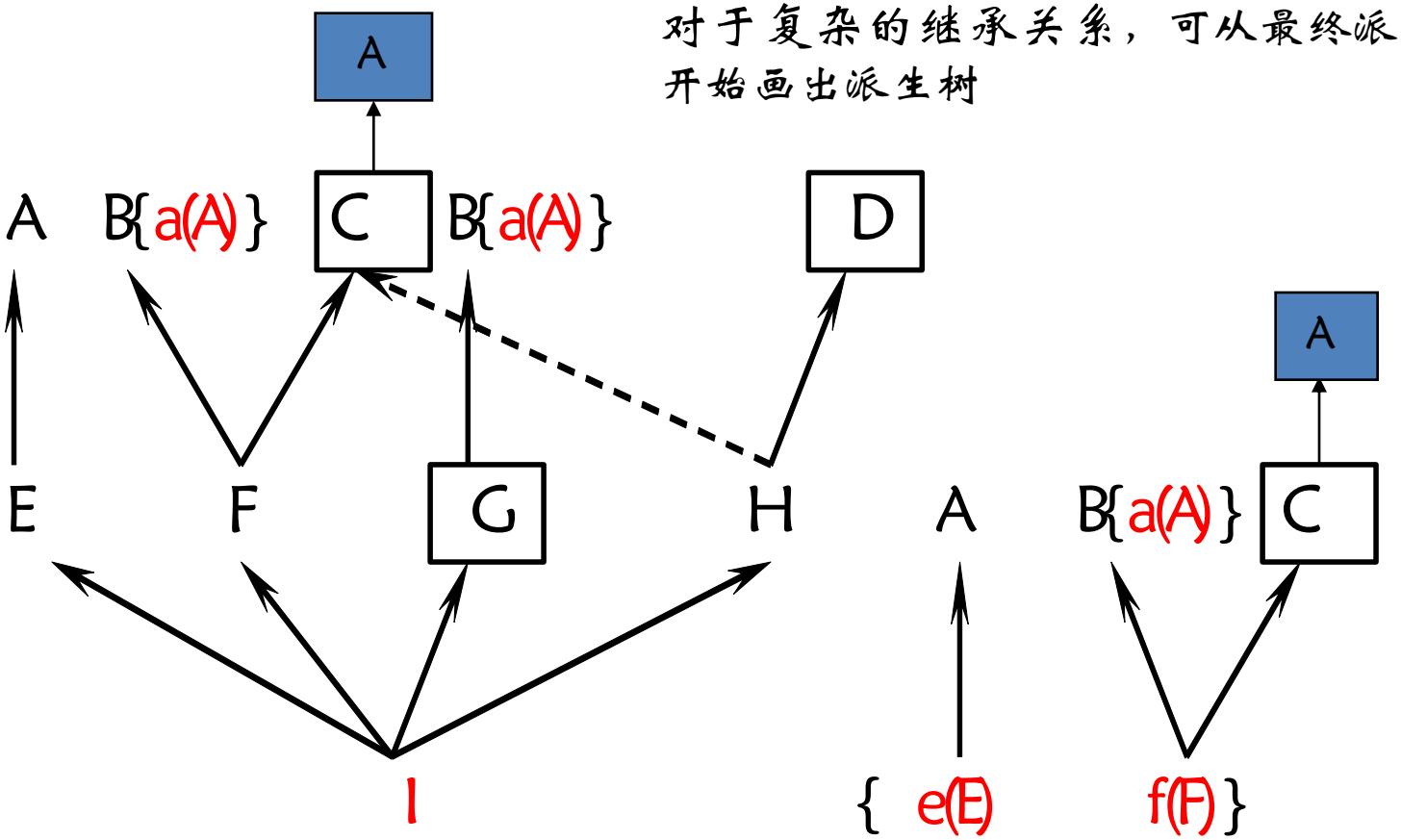
1：构造E: 必须先构造A，因此对应输出为AE

2：构造F：由于虚基类A、C已构造好，只需要先构造B。前面已分析构造B输出AB，因此基类F构造好时的输出为ABF

3：构造H：输出H

因此当I的基类构造完毕，输出为AE ABF H。

对于复杂的继承关系，可从最终派生类开始画出派生树



接着构造I的对象成员

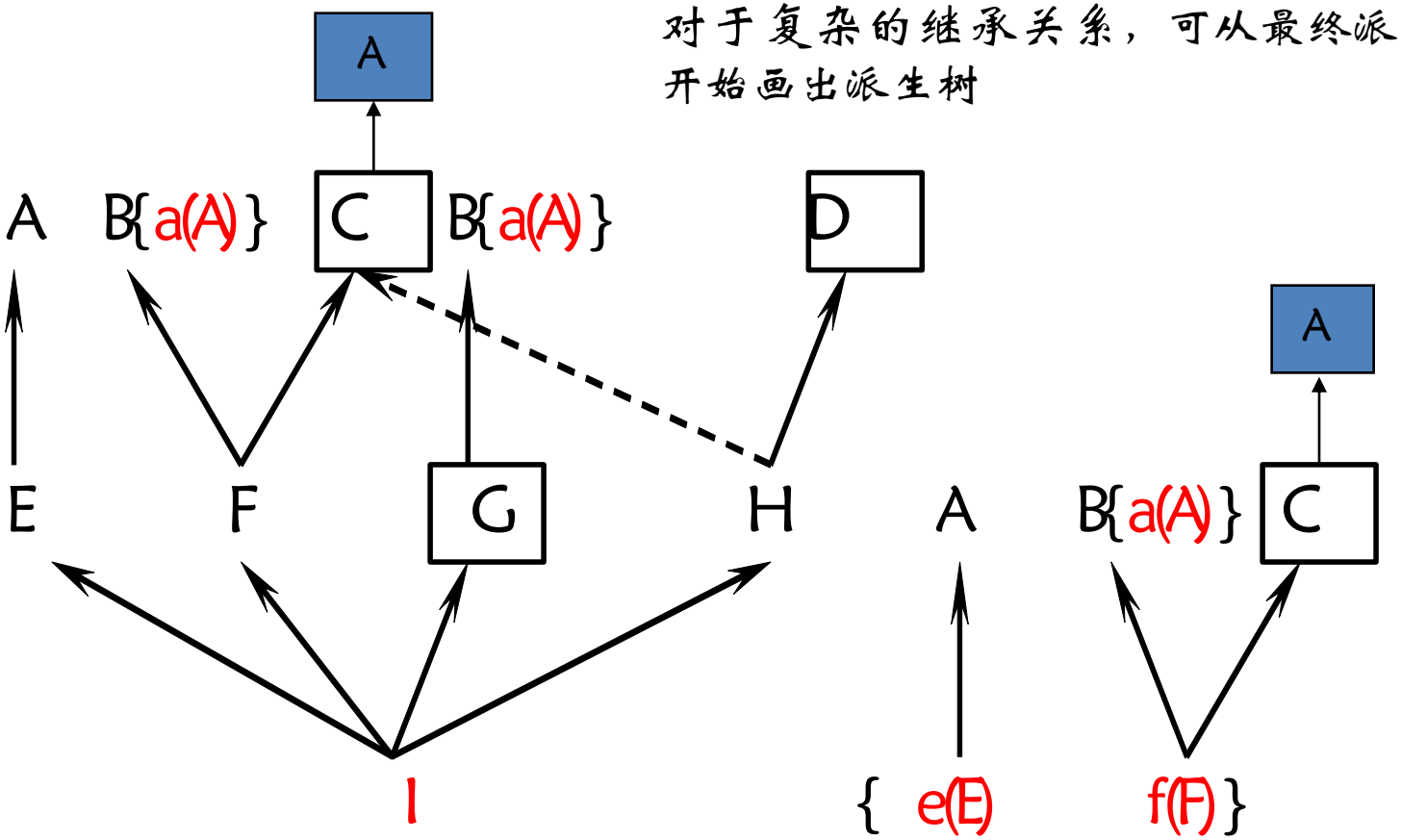
1：构造e (E): 必须先构造A，因此对应输出为AE

2：构造f (F)：首先按派生顺序依次构造虚基类A、C；再构造B（先构造B的对象成员a，再调用B的构造函数体）；最后构造F。因此对应输出为ACABF

因此当I的对象成员构造完毕，输出为AE ACABF。

最后调用I自己的构造函数，输出I

对于复杂的继承关系，可从最终派生类开始画出派生树



六棵派生树（根红色），输出：

ACABGD AE ABF H AE ACABF I