第三章　类间数据与函数

本章导读

类与类之间的关系可以分为：组合、友元、继承以及嵌套关系。从某种意义上讲，类相当于一种自定义的数据类型，当一个新的类中申明该数据类型时，即为类的组合关系；和友元函数类似，在一个新的类中申明一个已经存在的类为友元，即为该类的友元类，友元类与友元函数有相似的特征；当一个类具有已存在的类某些特征时，同时赋予给类一些新的特征，此即为类的继承关系。继承是面向对象重要特征。在一个新的类内部可以包含一个新申明的类，该类依赖外部类而存在，这种类关系称之为嵌套。

学习目标：

1. 认识类的组合；
2. 认识友元类；
3. 认识类的继承；
4. 认识类的访问控制；
5. 认识类的嵌套；

本章目录

第一节　类的组合

在一个新类的定义中集成另外一个已经定义的类即为类的组合，如例程3-1所示，第32行即在Student新类中集成了Date类。如需在Student类中访问Date的数据，可以通过其public成员访问(通常为成员函数)，和其他访问方式相比，Student类没有特权。

例程3-1

```
第1行  #include<iostream>
第2行  #include<string>
第3行  using namespace std;
第4行
第5行  class Date{
第6行  public:
第7行      Date(int Year=1973,int Month=1,int Day=1){
第8行          this->Year=Year;
第9行          this->Month=Month;
第10行         this->Day=Day;
第11行     }
第12行     void print(){
```

```
第13行            std::cout<<this->Year<<"-"<<this->Month<<"-"<<this->Day<<std::endl;
第14行        }
第15行    private:
第16行        int Year,Month,Day;
第17行    };
第18行    class Student{
第19行    public:
第20行        Student(string Name,Date Birthday){
第21行            this->Name=Name;
第22行            this->BirthDay=Birthday;
第23行        }
第24行        const string getName()const{
第25行            return this->Name;
第26行        }
第27行        Date getBirth()const{
第28行            return this->BirthDay;
第29行        }
第30行    private:
第31行        string Name;
第32行        Date BirthDay;//Student组合进Date类
第33行    };
```

```
第34行  int main(){
第35行      Date Birth(1989,12,1);
第36行      Student stud0("John",Birth);
第37行      std::cout<<(stud0.getName())<<std::endl;
第38行      stud0.getBirth().print();
第39行
第40行      return 0;
第41行  }
```

例程3-2是类组合的另外一个示例，在Array类中集成Node类，从中可以看出，对Node类的操作都是通过Node类的public成员函数。

例程3-2

```
第1行   #include<iostream>
第2行   #include<string>
第3行   using namespace std;
第4行
第5行   class Node{
第6行   public:
第7行       Node(long size=100){
第8行           this->size=size;
第9行           this->pHead=new int[this->size];
第10行          this->pNext=NULL;
```

```
第11行        }
第12行        int *getAddrHead()const{//获得Node节点其实地址
第13行            return this->pHead;
第14行        }
第15行        void setAddrNext(Node *ptr){//设置pNext值
第16行            this->pNext=ptr;
第17行        }
第18行        long getSize()const{
第19行            return this->size;
第20行        }
第21行    private:
第22行        long size;
第23行        int *pHead;//Node节点起始地址
第24行        Node *pNext;//下一个节点地址
第25行    };
第26行    struct Array{
第27行    public:
第28行        Array(int size=100){
第29行            this->unitSize=size;
第30行            this->firstNode=new Node(this->unitSize);
第31行            this->lastNode=this->lastNode;
```

```
第32行        }
第33行        Node * getHead()const{
第34行            return this->firstNode;
第35行        }
第36行        void addNode(){
第37行            Node *newNodeAddr=new Node(this->unitSize);
第38行        }
第39行    private:
第40行        long unitSize;//每个节点内数据容量
第41行        Node * firstNode;//第一数据地址
第42行        Node * lastNode;//最后数据地址
第43行    };
第44行    int main(){
第45行        Array myArr(50);
第46行        int *p=myArr.getHead()->getAddrHead();
第47行        int size=myArr.getHead()->getSize();
第48行        for(int i=0;i<size;++i){
第49行            *(p+1)=i;
第50行        }
第51行
第52行        return 0;
```

```
第53行    }
```

第二节　友元类

同友元函数一样，一个类可以将另外一个类申明为友元类。当类B成为类A的友元类时，则B类的所有的所有函数成员都是A类的友元函数，可以访问A类的私有和保护成员。如例程3-3所示，因为Student是Teacher的友元类，因此Student能访问Teacher的成员函数，如第26、29行所示。

例程3-3

```
第1行   #include<iostream>
第2行   #include<string>
第3行   using namespace std;
第4行   class Teacher{
第5行   public:
第6行       Teacher(){
第7行           std::cout<<"Now In Constructor===Teacher();"<<std::endl;
第8行       }
第9行       Teacher(string Name,string Description){
第10行          this->Name=Name;
第11行          this->Description=Description;
第12行      };
第13行      friend class Student;
第14行  private:
```

```
第15行        string Name;
第16行        string Description;
第17行    };
第18行    class Student{
第19行    public:
第20行        Student(string Name,string Description,int Score){
第21行            this->Name=Name;
第22行            this->Description=Description;
第23行            this->totalScore=Score;
第24行        }
第25行        void setTeacher(const Teacher &Mr){
第26行            this->Master=Mr;
第27行        }
第28行        void displayTeacher()const{
第29行            std::cout<<Master.Name<<std::endl<<Master.Description<<std::endl;
第30行        }
第31行    private:
第32行        Teacher Master;
第33行        string Name;
第34行        string Description;
第35行        int totalScore;
```

```
第36行  };
第37行  int main(){
第38行      Student Me("John","He is good Student",100);
第39行      Teacher Mr("Tom","He is good Teacher!!!");
第40行      Me.setTeacher(Mr);
第41行      Me.displayTeacher();
第42行
第43行      return 0;
第44行  }
```

关于友元，要注意几点：第一，友元关系不能传递，即B是A的友元类，C是B的友元类，C与B没有友元关系，不能进行数据共享；第二，友元关系是单向的，即B是A的友元类，但A不是B的友元类，A不能共享B的数据；第三，友元关系不能被继承，B是A的友元，B的派生类并不自动成为A的友元，即B的子孙并不自动成为A的友元。

第三节　类的继承

植物被分类为门、纲、目、科、属、种，每个子类都具有父类的特征，但同时具有一些新的特征，子类还可以继续分为子类。在人类社会，人具有一些共同特征，同时不同人群还具有特有的特征。这些关系，就是继承关系。父类还可以称为基类，子类还可以称为派生类。类的继承关系，是面向对象体系中最为重要的关系。如例程3-4所示。

例程3-4

```
第1行  #include<iostream>
第2行  #include<string>
第3行  using namespace std;
```

```
第4行
第5行  class Person{
第6行  public:
第7行      Person(string ID,string Name,bool Sex){
第8行          this->ID=ID;
第9行          this->Name=Name;
第10行         this->Sex;
第11行     };
第12行     void printBase()const{
第13行         cout<<"姓名:"<<this->Name<<endl
第14行                 <<"性别:"<<(this->Sex?"男":"女")<<endl
第15行                 <<"身份证号码:"<<this->ID<<endl;
第16行     }
第17行 private:
第18行     string ID;//身份证号码
第19行     string Name;//姓名
第20行     bool Sex;//性别，男为1，女为0
第21行 };
第22行 //学生类
第23行 class Student:public Person{
第24行 public:
```

```
第25行        //注意：基类构造函数调用
第26行        Student(int Grade,string Name,string ID,bool sex):Person(ID,Name,sex){
第27行            this->Grade=Grade;
第28行        }
第29行        void print()const{
第30行            printBase();
第31行            cout<<"年级:"<<this->Grade<<endl;
第32行        }
第33行    private:
第34行        int Grade;//年级
第35行    };
第36行    //教师类
第37行    class Teacher:public Person{
第38行    public:
第39行        //注意：基类构造函数调用
第40行        Teacher(string officeNo,string Title,string Name,string ID,bool sex):Person(ID,Name,sex){
第41行            this->officeNo=officeNo;
第42行            this->Title=Title;
第43行        }
第44行        void print()const{
第45行            printBase();
```

```
第46行            cout<<"职称:"<<this->Title<<endl
第47行                <<"办公室:"<<this->officeNo<<endl;
第48行        }
第49行    private:
第50行        string officeNo;//办公室编号
第51行        string Title;//职称
第52行    };
第53行    int main(){
第54行        Teacher Master("1123","1","Tom","2345",1);
第55行        Master.print();//调用派生类print();
第56行        Master.printBase();//调用基类printBase();
第57行        Student Stud(2,"Rose","6543",0);
第58行        Stud.print();//调用派生类print();
第59行
第60行        return 0;
第61行    }
```

当类与类之间具有继承关系时，派生类内可以访问基类的public和protected成员。

第四节　多重继承与虚基类

虚拟继承在一般的应用中很少用到，所以也往往被忽视，这也主要是因为在C++中，多重继承是不推荐的，也并不常用，而一旦离开

了多重继承，虚拟继承就完全失去了存在的必要。例程3-5是多重继承的示例，在第19行，类AB继承自类A和类B。

例程3-5

```
第1行  #include<iostream>
第2行  using namespace std;
第3行  class A{
第4行  public:
第5行      A(int a=1):valA(a){cout<<"In A's Constructor!!!\n";};
第6行      ~A(){cout<<"In AB's Deconstructor!!!\n";}
第7行      void print(){cout<<"Printing In A's Obj!!!\n";}
第8行  private:
第9行      int valA;
第10行 };
第11行 class B{
第12行 public:
第13行     B(int b=1):valB(b){cout<<"In B's Constructor!!!\n";};
第14行     ~B(){cout<<"In B's Deconstructor!!!\n";}
第15行     void print(){cout<<"Printing In B's Obj!!!\n";}
第16行 private:
第17行     int valB;
第18行 };
第19行 class AB:public B,public A{
```

```
第20行   public:
第21行       AB(int ab=1):valAB(ab){cout<<"In AB's Constructor!!!\n";};
第22行       ~AB(){cout<<"In AB's Deconstructor!!!\n";}
第23行       void print(){
第24行           cout<<"Printing In AB's Obj!!!\n";
第25行           B::print();
第26行           A::print();
第27行       }
第28行   private:
第29行       int valAB;
第30行   };
第31行   int main(){
第32行       AB ab;
第33行       ab.print();
第34行       return 0;
第35行   }
```

例程3-6是A和B都继承自Z，而AB继承自和A和B。按照常规继承方式，则A和B都将有Z的成分，第25行的代码将出现问题，即不知道Z是A的Z还是B的Z，出现指代不明确。采用虚拟继承，则将仅有一个副本，其关系如示意图3-1所示。

例程3-6

```
第1行   #include<iostream>
第2行   using namespace std;
```

```
第3行  class Z{
第4行  public:
第5行      Z(int a=1):valZ(a){cout<<"In Z's Constructor!!!\n";};
第6行      ~Z(){cout<<"In Z's Deconstructor!!!\n";}
第7行      void print(){cout<<"Printing In Z's Obj!!!\n";}
第8行  private:
第9行      int valZ;
第10行 };
第11行 class A:virtual public Z{
第12行 public:
第13行     A(int a=1):valA(a){cout<<"In A's Constructor!!!\n";};
第14行     ~A(){cout<<"In AB's Deconstructor!!!\n";}
第15行     void print(){cout<<"Printing In A's Obj!!!\n";}
第16行 private:
第17行     int valA;
第18行 };
第19行 class B:virtual public Z{
第20行 public:
第21行     B(int b=1):valB(b){cout<<"In B's Constructor!!!\n";};
第22行     ~B(){cout<<"In B's Deconstructor!!!\n";}
第23行     void print(){cout<<"Printing In B's Obj!!!\n";}
```

```
第24行  private:
第25行      int valB;
第26行  };
第27行  class AB:public B,public A{
第28行  public:
第29行      AB(int ab=1):valAB(ab){cout<<"In AB's Constructor!!!\n";};
第30行      ~AB(){cout<<"In AB's Deconstructor!!!\n";}
第31行      void print(){
第32行          cout<<"Printing In AB's Obj!!!\n";
第33行          B::print();
第34行          A::print();
第35行          Z::print();//如果不采用虚拟集成，则基类Z将不明确
第36行      }
第37行  private:
第38行      int valAB;
第39行  };
第40行  int main(){
第41行      AB ab;
第42行      ab.print();
第43行      return 0;
第44行  }
```

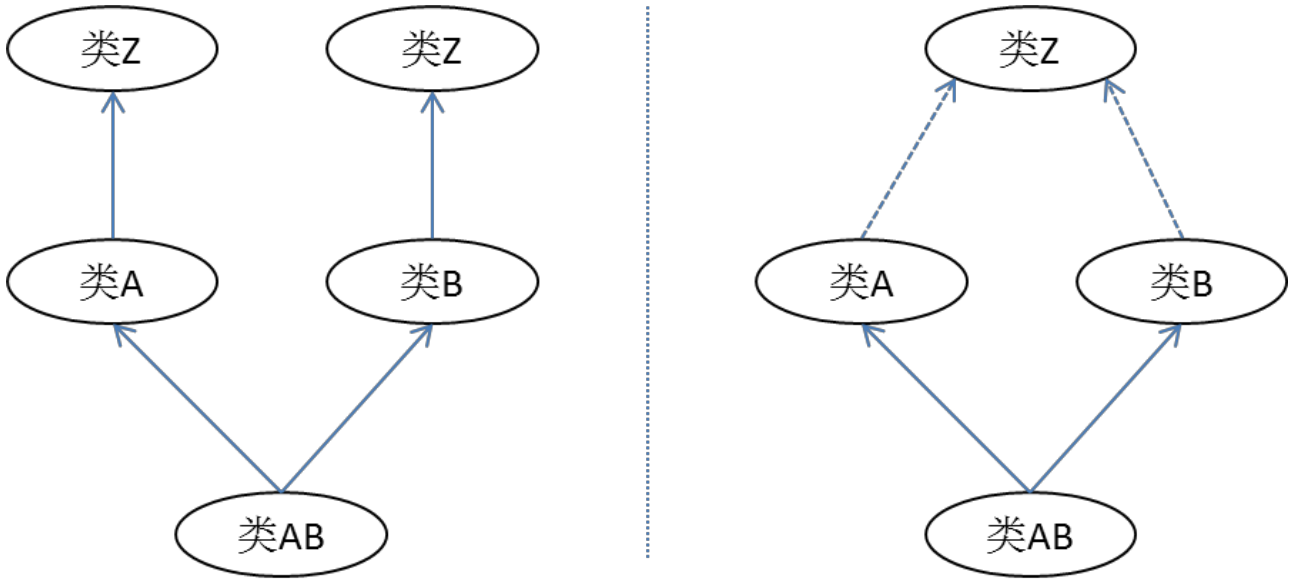

图3-1　虚拟(右)与非虚拟(左)继承关系

第五节　虚函数

例程3-7的Circle和Rectangle类继承自Shape类，第28行定义display()函数，其目的是分别Circle和Rectangle类的display()，其定义形式为void display(Shape *pShape)。在运行过程中，会自动根据运行场景，或选择Circle的display()函数，或选择Rectangle的display()函数，实现一个函数形态，自动选择需要的对象，其实现前提是参数Shape类中的display()必须申明虚函数(函数名称前增加关键virtual，如第6行所示)，同时，函数要通过指针或引用来访问虚函数。如将函数定义形式修改为void display(Shape &pShape)，则需将第34行的display(&shapeRect)修改为display(shapeRect)，第35行的display(&shapeRect)修改为display(shapeRect)形式，这种形式更加直观友好。

例程3-7

```
第1行  #include<iostream>
第2行  using namespace std;
第3行  const double PI=3.1415926;
```

```
第4行  class Shape{
第5行  public:
第6行      virtual void display()const{
第7行          cout<<"This is a Shape!!!"<<endl;
第8行      }
第9行  };
第10行 class Circle:public Shape{//圆
第11行 public:
第12行     Circle(int R=1):Radius(R){};
第13行     void display()const{
第14行         cout<<"I am a circle!!!Area="<<this->Radius*PI<<endl;
第15行     }
第16行 private:
第17行     int Radius;//半径
第18行 };
第19行 class Rectangle:public Shape{//矩形
第20行 public:
第21行     Rectangle(int W=1,int H=1):width(W),height(H){};
第22行     void display()const{
第23行         cout<<"I am a Rectangle!!!Area="<<this->width*this->height<<endl;
第24行     }
```

```
第25行   private:
第26行       int width,height;//宽和高
第27行   };
第28行   void display(Shape *pShape){
第29行       pShape->display();
第30行   }
第31行   int main(){
第32行       Rectangle shapeRect(2,3);
第33行       Circle shapeCircle(5);
第34行       display(&shapeRect);
第35行       display(&shapeCircle);
第36行
第37行       return 0;
第38行   }
```

用一个函数应对多个应用，是多态性体现。void display(Shape *pShape)或void display(Shape &pShape)的参数有确定类型，但由于Shape类是Circle和Rectangle的基类，且申明为虚函数，由此实现对象的动态绑定。虚函数经过派生之后，在其派生类中就可以实现运行过程中的动态。另外，虚函数必须申明为非静态的成员函数。

第六节　纯虚函数与抽象类

比较例程3-8和例程3-7可以发现，仅shape类中的display()函数发生变化，其他代码维持不变。例程3-8中，display()没有代码实现，其函数名称后增加了“=0”，表明其为纯虚函数，含有纯虚函数的类称之为抽象类。抽象类不能单独存在，依赖于派生类而实例化，即不能单独申明抽象类的对象。

```
第1行  #include<iostream>
第2行  using namespace std;
第3行  const double PI=3.1415926;
第4行  class Shape{
第5行  public:
第6行      virtual void display()const=0;
第7行  };
第8行  class Circle:public Shape{//圆
第9行  public:
第10行     Circle(int R=1):Radius(R){};
第11行     void display()const{
第12行         cout<<"I am a circle!!!Area="<<this->Radius*PI<<endl;
第13行     }
第14行 private:
第15行     int Radius;//半径
第16行 };
第17行 class Rectangle:public Shape{//矩形
第18行 public:
第19行     Rectangle(int W=1,int H=1):width(W),height(H){};
第20行     void display()const{
```

```
第21行            cout<<"I am a Rectangle!!!Area="<<this->width*this->height<<endl;
第22行        }
第23行    private:
第24行        int width,height;//宽和高
第25行    };
第26行    void display(Shape *pShape){
第27行        pShape->display();
第28行    }
第29行    int main(){
第30行        Rectangle shapeRect(2,3);
第31行        Circle shapeCircle(5);
第32行        display(&shapeRect);
第33行        display(&shapeCircle);
第34行
第35行        return 0;
第36行    }
```

第七节　访问控制

在C++中，关键字public和private被称为访问控制，除这二者之外，protected也用于访问控制。访问控制又被称为可见性。从public到protected再到private，可见性越来越低。在一个类中，public成员可以通过类的对象直接访问，如例程3-9第39行所示，而protected和private则不允许，如第40、41行所示。

派生类一般都以public方式继承基类，此时派生类的对象可以直接访问public成员，而不能访问protected和private成员，如第43-

45行所示；当以protected继承基类时，public成员被降格为protected，protected不发生变化，private已不能再降，此时派生类的对象不能访问基类所有成员；当以private继承基类时，public和protected都降格为private成员，private维持不变，此时派生类的对象不能访问基类所有成员。

当派生类以public、protected或private继承基类时，其成员函数可以访问基类的public和protected数据成员和函数成员，如例程3-9中X、Y、Z类的print()函数。

例程3-9

```
第1行  #include<iostream>
第2行  using namespace std;
第3行  class ABC{
第4行  public:
第5行      ABC(int a=1,int b=2,int c=3):a(a),b(b),c(c){}
第6行      int a;
第7行  protected:
第8行      int b;
第9行  private:
第10行     int c;
第11行 };
第12行 class X:public ABC{
第13行 public:
第14行     void print(){
第15行         cout<<this->a<<endl;//Yes，public成员仍然为public
```

```
第16行            cout<<this->b<<endl;//Yes，protected成员仍然为protected
第17行            //cout<<this->c<<endl;//No，被隔离，相当于hidden
第18行        }
第19行    };
第20行    class Y:protected ABC{
第21行    public:
第22行        void print(){
第23行            cout<<this->a<<endl;//Yes，public降格为protected
第24行            cout<<this->b<<endl;//Yes，protected维持为protected
第25行            //cout<<this->c<<endl;//No，被隔离，相当于hidden
第26行        }
第27行    };
第28行    class Z:private ABC{
第29行    public:
第30行        void print(){
第31行            cout<<this->a<<endl;//Yes，public被降格为private
第32行            cout<<this->b<<endl;//Yes，protected被降格为private
第33行            //cout<<this->c<<endl;//No，被隔离，相当于hidden
第34行        }
第35行    };
第36行
```

```
第37行  class fromZ:public Z{//Z private from ABC
第38行      void print(){
第39行          //cout<<this->a<<endl;//No，由于在Z中降格为private成员，因此不能被访问
第40行          //cout<<this->b<<endl;//No，由于在Z中降格为private成员，因此不能被访问
第41行          //cout<<this->c<<endl;//No，由于在Z中降格为private成员，因此不能被访问
第42行      }
第43行  };
第44行  int main(){
第45行      ABC abc;
第46行      abc.a=100;//Yes，a为public成员，能被外部访问
第47行      //abc.b=200;//No，b为protected成员，不能被外部访问
第48行      //abc.c=300;//No，c为private成员，不能被外部访问
第49行
第50行      X x;//X以public方式继承ABC
第51行      x.a=100;//Yes，a在X中仍然为public，能被外部访问
第52行      //x.b=200;//No，b在X中仍然为protected，不能被外部访问
第53行      //x.c=300;//No，c在X中已被隐藏，不能被外部访问
第54行      x.print();
第55行
第56行      //Y以protected方式继承ABC，public成员降格为protected成员，
第57行      //派生类对象y不可以访问原public成员，protected成员和私有成员
```

```
第58行        Y y;
第59行        //y.a=100;//No，public成员被降格为protected成员，仅能被类和派生类成员函数访问，外部不能访问
第60行        //y.b=200;//No，protected成员仍维持为protected成员，仅能被类和派生类成员函数访问，外部不能访问
第61行        //y.c=300;//No，c在Y中已被隐藏，派生类已不能访问，外部不能访问
第62行        y.print();
第63行
第64行        //z以private方式集成ABC，所有访问属性降低为private级别
第65行        //派生对象z不可以访问员public、protected成员和private成员
第66行        Z z;//Z以private方式继承ABC
第67行        //z.a=100;//No，a为private成员，不能被外部访问
第68行        //z.b=200;//No，b为private成员，不能被外部访问
第69行        //z.c=300;//No，c为private成员，不能被外部访问
第70行        z.print();
第71行
第72行        fromZ fromz;
第73行        //fromz.a=100;//No，a为private成员，不能被外部访问
第74行        //fromz.b=200;//No，b为private成员，不能被外部访问
第75行        //fromz.c=300;//No，c为private成员，不能被外部访问
第76行
第77行        return 0;
第78行    }
```

第八节　类的嵌套

类的嵌套是在类的内部申明一个新类，该类仅在外部类的内部存在，如例程3-10所示。被嵌套类被封装在外部类中，不能单独存在。如果一个类仅能在某个类的内部存在，采用嵌套类是不错的选择。

例程3-10

```
第1行  #include<iostream>
第2行  using namespace std;
第3行  class List{
第4行  public:
第5行      List(int size=100){
第6行          this->size=size;
第7行          this->pDataHead=new Node(this->size);
第8行      }
第9行  private:
第10行     class Node{
第11行     public:
第12行         Node(int size=100){
第13行             cout<<"I am in Node Constructor!!!"<<endl;
第14行             this->size=size;
第15行             this->pNode=new int[this->size];
第16行         }
第17行         int *getNodeAddr()const{
```

```
第18行                return this->pNode;
第19行            }
第20行            void print()const;//原型申明
第21行            int size;
第22行            int *pNode;
第23行            int *pNext;
第24行        };
第25行
第26行        int size;
第27行        Node *pDataHead;
第28行    };
第29行    //注意此处函数申明格式
第30行    void List::Node.print()const{
第31行        //函数具体实现代码
第32行    }
第33行    int main(){
第34行        List myList;
第35行
第36行        return 0;
第37行    }
```

在第20行void print()const申明print()函数的原型，其实现在第30-32行实现，注意其print()前作用域运算符的使用，List::Node

表明Node在List域内。

第九节　类关系中的函数

类可以集成数据成员和函数成员。一个类只有被实例化后，其内部的函数和数据才能被应用。在不同的类关系中，不同类型的函数，有较为复杂的关系。

在例程3-11中，类与类之间有多种关系。在类A中，申明类B是类A的友元类。当申明A a时(第49行)，可以发现类A的a对象执行实例化时，类B并没有被实例化。在类A中申明B是其友元类，但A并不包含B的数据。相反，当在实例化类B的b对象时(第53行)，由于B类中有申明类A的实例a对象，因此首先实例化A类(A类的构造函数被执行)，然后执行B类的构造函数。类Y继承自类C，当实例化类Y为y对象时(第51行)，首先基类被实例化(执行C的构造函数)，然后才是派生类被实例化(执行Y的构造函数)。类B除了组合A类外，还嵌套类D，当39行被注释时，类D虽然在类X申明原型，但未使用，此时不会执行类D的构造函数，知道第30行doSome()函数被执行时，类D才会被实例化。如果第39行未被注释，则优先实例化类D，然后示例化类X。析构函数严格按照构造函数相反次序执行，因为要考虑潜在的相关性。另外，构造函数调用次序完全不受构造函数的初始化表达式的次序影响，该次序是以后成员在类中的申明次序所决定。

例程3-11

```
第1行  #include<iostream>
第2行  using namespace std;
第3行  class A{
第4行  public:
第5行      A(){cout<<"I am in A's Constructor!!!"<<endl;}
第6行      ~A(){cout<<"I am in A's Deconstructor!!!"<<endl;}
第7行      void print(){cout<<"Printing In Class A!!!"<<endl;}
第8行      friend class B;
第9行  };
```

```
第10行  class B{
第11行  public:
第12行      B(){cout<<"I am in B's Constructor!!!"<<endl;}
第13行      ~B(){cout<<"I am in B's Deconstructor!!!"<<endl;}
第14行      void print(){cout<<"Printing In Class B!!!"<<endl;}
第15行      void execAprint(){this->a.print();};
第16行  private:
第17行      A a;
第18行  };
第19行  class C{
第20行  public:
第21行      C(){cout<<"I am in C's Constructor!!!"<<endl;}
第22行      ~C(){cout<<"I am in C's Deconstructor!!!"<<endl;}
第23行      void print(){cout<<"Printing In Class C!!!"<<endl;}
第24行  };
第25行  class X{
第26行  public:
第27行      X(){cout<<"I am in C's Constructor!!!"<<endl;}
第28行      ~X(){cout<<"I am in C's Deconstructor!!!"<<endl;}
第29行      void print(){cout<<"Printing In Class X!!!"<<endl;}
第30行      void doSome(){D d;}
```

```
第31行   private:
第32行       A a;//组合关系
第33行       class D{
第34行       public:
第35行           D(){cout<<"I am in D's Constructor!!!"<<endl;}
第36行           ~D(){cout<<"I am in D's Deconstructor!!!"<<endl;}
第37行           void print(){cout<<"Printing In Class D!!!"<<endl;}
第38行       };
第39行       //D d;//删除行前注释符，将随X实例化时，实例化类D的d对象
第40行   };
第41行   class Y:public C{
第42行   public:
第43行       Y(){cout<<"I am in Y's Constructor!!!"<<endl;}
第44行       ~Y(){cout<<"I am in Y's Deconstructor!!!"<<endl;}
第45行       void print(){cout<<"Printing In Class Y!!!"<<endl;}
第46行   };
第47行   int main(){
第48行       cout<<"\nNow Exec:A a"<<" --->";
第49行       A a;
第50行       cout<<"\nNow Exec:Y y"<<" --->";
第51行       Y y;
```

```
第52行        cout<<"\nNow Exec:B b"<<" --->";
第53行        B b;
第54行        cout<<"\nNow Exec:X x"<<" --->";
第55行        X x;
第56行        cout<<"\nNow Exec:x.doSome()"<<" --->";
第57行        x.doSome();
第58行
第59行        cout<<endl<<endl;
第60行        return 0;
第61行    }
```

```
Now Exec:A a   --->I am in A's Constructor!!!

Now Exec:Y y   --->I am in C's Constructor!!!
I am in Y's Constructor!!!

Now Exec:B b   --->I am in A's Constructor!!!
I am in B's Constructor!!!

Now Exec:X x   --->I am in A's Constructor!!!
I am in C's Constructor!!!

Now Exec:x.doSome()   --->I am in D's Constructor!!!
I am in D's Deconstructor!!!


I am in C's Deconstructor!!!
I am in A's Deconstructor!!!
I am in B's Deconstructor!!!
I am in A's Deconstructor!!!
I am in Y's Deconstructor!!!
I am in C's Deconstructor!!!
I am in A's Deconstructor!!!
```

图3-2

当基类含有纯虚函数和纯虚函数时，其构造函数与析构函数执行顺序与普通基类类似。当然，含有纯虚函数的抽象类不能单独实例化，需要与派生类一起才能被实例化。

不是所有的函数都能自动地从基类继承到派生类中。构造函数和析构函数用来处理对象的创建和析构操作，只知道当前类的对象。另外，赋值运算符也不能被继承，它的操作类似于构造函数。从常理上讲，基类重载赋值运算符，仅仅知道当前类的所有成员，并不知道派生类将会发生什么变化。

静态成员函数可以被继承到派生类中，如果重新定义一个静态成员函数，所有在基类中的其他重载函数都会被隐藏。如果改变基类中一个函数的特征，所有使用该函数名字的基类版本都会被隐藏。另外，静态成员函数不可以是虚函数。