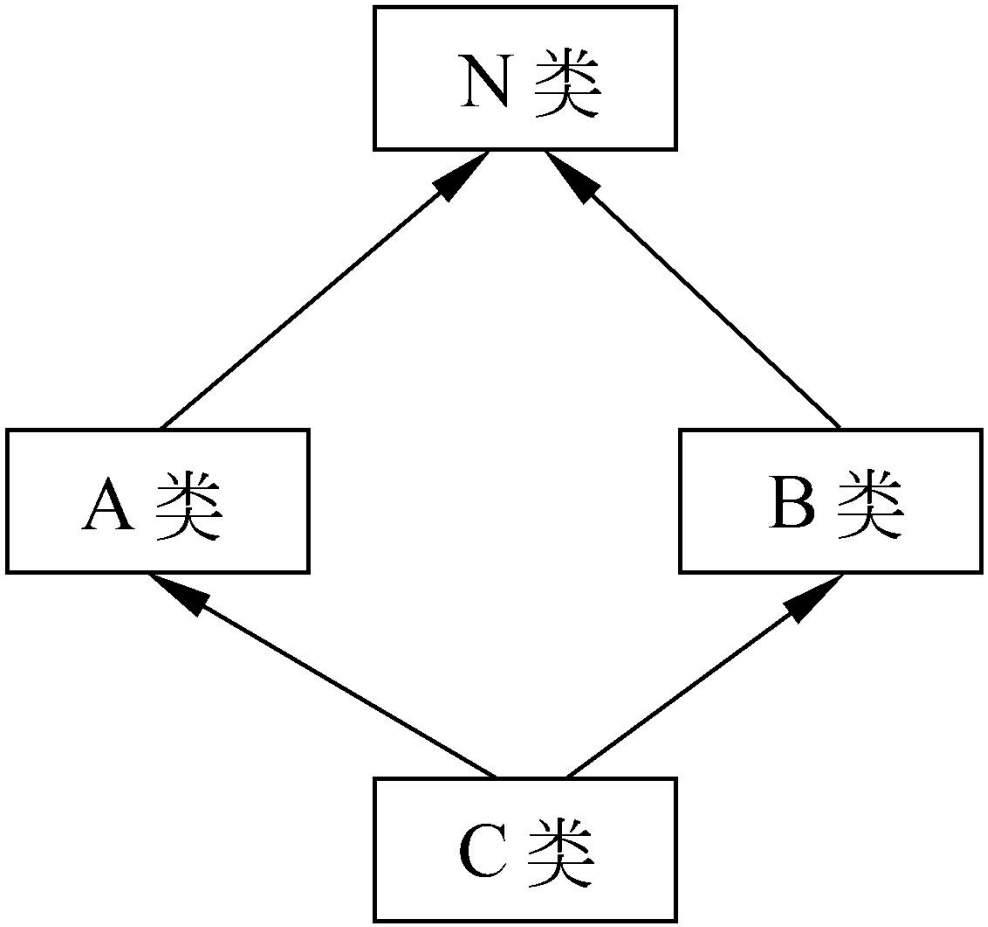
同名成员的二义性(ambiguous)问题

```
class N
{
public:
    int a;
    void display()
    {
       cout<<“N::a=“
       <<a<<endl;
    }
};
```



同名成员的二义性(ambiguous)问题

- (1) 两个基类有同名成员
  - 用基类名来限定，如c1.A::display();
- (2) 两个基类和派生类都有同名成员
  - 同名覆盖，派生类新增的同名成员覆盖了基类中的同名成员
  - 用基类名来限定，如c1.A::display();
- (3)两个基类从同一个基类派生
  - 通过直接基类名来限定，如c1.A::display();
  - c1.N::display();

基类与派生类的转换

- 1. 派生类对象向基类对象赋值
- 2. 派生类对象向基类对象的引用赋值或初始化
- 3. 函数的参数是基类对象引用，相应的实参可以用子类对象。
- 4. 派生类对象的地址可以赋给指向基类对象的指针变量。
- 不论哪种方式都只能访问派生类中的基类成员，而不能访问派生类增加的成员。

例11.10

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

class Student                                                    //声明Student类
{
public:
    Student(int, string,float);  //声明构造函数
    void display( );
private:
    int num;
    string name;
    float score;
};
Student::Student(int n, string nam,float s)
{ num=n; name=nam; score=s;}
void Student::display( )
{
    cout<<endl<<"num:"<<num<<endl;
    cout<<"name:"<<name<<endl;
    cout<<"score:"<<score<<endl;
}
class Graduate:public Student
{
public:
    Graduate(int, string ,float,float);              //声明构造函数
    void display( );                                                    //声明输出函数
private:
    float pay;                                                          //工资
};

Graduate::Graduate(int n, string nam,float s,float p):Student(n,nam,s),pay(p){ }
void Graduate::display()
{
    Student::display();
    cout<<"pay="<<pay<<endl;
}
int main()
{
    Student stud1(1001,"Li",87.5);
    Graduate grad1(2001,"Wang",98.5,563.5);
    Student *pt=&stud1;
    pt->display( );          //调用stud1.display函数
    pt=&grad1;               //指针指向grad1
    pt->display( );          //调用grad1.display函数
}
```

第12章 多态性与虚函数

Polymorphism & Virtual Functions

12.1 多态性的概念

12.2 一个典型的例子

12.3 虚函数

12.4 纯虚函数与抽象类

12.1 多态性的概念

多态性(polymorphism)是面向对象程序设计的一个重要特征。

当向不同的对象发送同一个消息时，不同的对象在接收时会产生不同的行为。

通过继承产生的不同层次的派生类，与基类成员同名的成员在不同的派生类中有不同的含义。也可以说，多态性是“一个接口，多种方法”。

动态多态性，又称运行时多态性，在程序运行过程中才动态地确定操作所针对的对象，通过虚函数(virtual function)实现。

通过继承而产生了相关的不同的派生类，与基类成员同名的成员在不同的派生类中有不同的含义。也可以说，多态性是“一个接口，多种方法”。

从系统实现的角度看，多态性分为两类:

静态多态性，又称编译时多态性，在程序编译时系统就能决定调用的是哪个函数，如函数重载和运算符重载。

动态多态性，又称运行时多态性，在程序运行过程中才动态地确定操作所针对的对象，通过虚函数(virtual function)实现。

The concept of polymorphism

Polymorphism (implemented in C++ with virtual functions) is the third essential feature of an object oriented programming language, after data abstraction and inheritance.

Polymorphism allows improved code organization and readability as well as the creation of extensible programs that can be "grown" not only during the original creation of the project, but also when new features are desired.

The virtual function allows one type to express its distinction from another, similar type, as long as they're both derived from the same base type. This distinction is expressed through differences in behavior of the functions that you can call through the base class.

12.2 一个典型的例子-编译时多态性

例12.1 建立一个Point(点)类，包含数据成员x,y(坐标点)。

以Point类为基类，派生出一个Circle(圆)类，增加数据成员r(半径)；

再以Circle类为直接基类，派生出一个Cylinder(圆柱体)类，再增加数据成员h(高)。

编写程序，重载运算符“<<”和“>>”，使之能用于输出以上类对象。

先声明基类，再声明派生类，逐级进行，分步调试。

(1) 声明基类Point类

```
#include <iostream>
using namespace std;
class Point
{
public:
   Point(double a=0,double b=0) { x=a;y=b;}
   void setPoint(double a,double b) { x=a;y=b;}
   double getX( ) const {return x;}
   double getY( ) const {return y;}
   friend ostream & operator<<(ostream &,const Point &);//重载运算符“<<”，输出点坐标
protected:
   double x,y;
};
```

```
ostream & operator<<(ostream &output,const Point &p)
{
   output<<"["<<p.x<<","<<p.y<<"]"<<endl;
   return output;
}
int main( )
{
   Point p(3.5,6.4);
   cout<<"x="<<p.getX( )<<",y="<<p.getY( )<<endl;
   p.setPoint(8.5,6.8);        //重新设置p的坐标值
   cout<<"p(new):"<<p<<endl;
   return 0;
}
```

(2) 声明派生类Circle

```
class Circle:public Point
{
public:
   Circle(double a=0,double b=0,double
 r=0) :Point(a,b),radius(r){ }//构造函数
   void setRadius(double r){radius=r;} //设置半径值
   double getRadius( ) const {return radius;} //读取半径值
   double area ( ) const {return 3.14159*radius*radius;} //计算
圆面积
   friend ostream &operator<<(ostream &,const Circle &);//重
载运算符“<<”，输出圆的信息
protected:
   double radius;
};
```

```
//重载运算符“<<”，输出圆的信息
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(3.5,6.4,5.2);
   cout<<"original circle:\nx="<<c.getX()<<", y="
      <<c.getY()<<", r="<<c.getRadius( )
      <<", area="<<c.area( )<<endl;
   c.setRadius(7.5);
   c.setPoint(5,5);
   cout<<"new circle:\n"<<c; //用重载运算符“<<”输出圆对象的信息
   Point &pRef=c;
   cout<<"pRef:"<<pRef;
   return 0;
}
```

(3) 声明Circle的派生类Cylinder

```
class Cylinder:public Circle
{
public:
   Cylinder (double a=0,double b=0,double r=0,double
 h=0) :Circle(a,b,r),height(h){}
   void setHeight(double h){height=h;}
   double getHeight( ) const {return height;}
   double area( ) const;
   double volume( ) const;
   friend ostream& operator<<(ostream&,const Cylinder&);//
 重载运算符“<<”
protected:
   double height;
};
```

```
double Cylinder::area( ) const                    //计算圆柱表面积
{ return 2*Circle::area( )+2*3.14159*radius*height;}
double Cylinder::volume() const                   //计算圆柱体积
{return Circle::area()*height;}
//重载运算符“<<”
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 cy1(3.5,6.4,5.2,10);
    cout<<"\noriginal cylinder:\n"<<cy1 <<endl;
    cy1.setHeight(15); //设置圆柱高
    cy1.setRadius(7.5);          //设置圆半径
    cy1.setPoint(5,5);  //设置圆心坐标值x,y
    cout<<"\nnew cylinder:\n"<<cy1;
    Point &pRef=cy1;
    cout<<"\npRef as a Point:"<<pRef;
    Circle &cRef=cy1;
    cout<<"\ncRef as a Circle:"<<cRef;
    return 0;
}
```

12.3 虚函数

12.3.1 虚函数的作用

在多重继承时，从不同的基类中会继承一些重复的数据。这就是继承的成员同名而产生的二义性(ambiguous)问题。

(1) 两个基类有同名成员

(2) 两个基类和派生类三者都有同名成员

(3) 两个基类是从同一个基类派生的

c1.A::a=3; c1.A::display();//同名覆盖与虚基类

在同一个类族的继承层次结构中，在不同的层次中可以出现名字相同、参数个数和类型都相同而功能不同的函数。

(1)在程序中通过不同的对象名去调用不同派生层次中的同名函数(同名覆盖)。

(2)定义派生类指针调用该成员函数，则系统会调用派生类对象中的同名函数；

(3) 指向基类对象的指针只能访问派生类中的基类成员，而不能访问派生类增加的成员。

能否用同一个调用形式，既能调用派生类又能调用基类的同名函数？

例如，用同一个语句“pt->display( );”可以调用不同派生层次中的display函数，只需在调用前使指针pt指向不同的类对象即可。

在例11.10(及12.2)中进行了这种尝试，但指向基类对象的指针不能访问派生类增加的成员。

例12.2 基类与派生类中有同名函数-运行时多态性

```
#include <iostream>          //同例11.10
#include <string>
using namespace std;
class Student                //声明基类Student
{
public:
   Student(int, string,double);     //声明构造函数
   virtual void display( );         //声明输出函数
protected:
   int num;  string name;  double score;
};
```

```
//Student类成员函数的实现
//定义构造函数
Student::Student(int n, string nam,double s)
{num=n;name=nam;score=s;}

void Student::display( )        //定义输出函数
{
   cout<<"num:"<<num<<"\nname:"
       <<name<<"\nscore:"<<score
       <<"\n\n";
}
```

```
//声明公用派生类Graduate
class Graduate:public Student
{
public:
   Graduate(int, string, double, double);   //声明构造函数
   void display( );                          //声明输出函数
private:
   double pay;
};
```

```
// Graduate类成员函数的实现
void Graduate::display( )                    //定义输出函数
{
   cout<<"num:"<<num<<"\nname:"
       <<name<<"\nscore:"<<score
       <<"\npay="<<pay<<endl;
}
Graduate::Graduate(int n, string nam,double s,double
   p):Student(n,nam,s),pay(p){ }
```

```
int main()
{
    Student stud1(1001,"Li",87.5);
    Graduate grad1(2001,"Wang",98.5,563.5);
    Student *pt=&stud1;        //定义指向基类对象的指针变量pt
    pt->display( );
    pt=&grad1;
    pt->display( );
    return 0;
}
```

虚函数的作用是可以通过基类指针或引用来访问基类和派生类中(增加)的同名函数。

虚函数实现的动态多态性: 同一类族中不同类的对象，对同一函数调用作出不同的响应。

虚函数的使用方法是:

(1) 在基类用virtual声明成员函数为虚函数。

在类外定义虚函数时，不必再加virtual。

(2) 在派生类中重新定义此函数，要求函数名、函数类型、函数参数个数和类型全部与基类的虚函数相同。

当一个成员函数被声明为虚函数后，其派生类中的同名函数都自动成为虚函数。

如果派生类中没有对基类的虚函数重新定义，则派生类简单地继承其直接基类的虚函数。

(3) 定义一个指向基类对象的指针变量，并使它指向同一类族中需要调用该函数的对象。

(4) 通过该指针变量调用此虚函数，此时调用的就是指针变量指向的对象的同名函数。

函数重载处理的是同一层次上的同名函数问题，而虚函数处理的是不同派生层次上的同名函数问题，前者是横向重载，后者可以理解为纵向重载。

虚函数与重载不同的是: 同一类族的虚函数的首部是相同的，而函数重载时函数的首部是不同的(参数个数或类型不同)。

12.3.2 静态关联与动态关联

对于调用同一类族中的虚函数，应当在调用时用一定的方式告诉编译系统，要调用的是哪个类对象中的函数。

确定调用的具体对象的过程称为关联(binding)。在这里是指把一个函数名与一个类对象捆绑在一起，建立关联。一般地说，关联指把一个标识符和一个存储地址联系起来。

函数重载和通过对象名调用的虚函数，在编译时即可确定其调用的虚函数属于哪一个类，其过程称为静态关联(static binding)。

通过基类指针调用虚函数时，编译系统通过指针与虚函数的结合来实现多态性。在运行阶段，基类指针可以先后指向不同的类对象，从而调用同一类族中不同类的虚函数。此过程称为动态关联(dynamic binding)。

12.3.3 在什么情况下应当声明虚函数

(1) 若成员函数所在的类会作为基类，成员函数在派生类中有可能更改功能，一般应该将它声明为虚函数。

(2)若如果是通过基类指针或引用去(而不是对象名)调用成员函数，则应当声明为虚函数。

(3) 在基类中为派生类预留函数名(纯虚函数)。

说明

(1)只能用virtual将类的成员函数声明为虚函数。因为虚函数的作用是允许在派生类中对基类的虚函数重新定义，只能用于类的继承层次结构中。

(2) 一个成员函数被声明为虚函数后，在同一类族中的类就不能再定义一个非virtual的但与该虚函数具有相同的参数(包括个数和类型)和函数返回值类型的同名函数。

12.3.4 虚析构函数

如果用new运算符建立了临时对象，若基类中有析构函数，并且定义了一个指向该基类的指针变量。在程序用带指针参数的delete运算符撤销对象时，会发生一个情况:

系统会只执行基类的析构函数，而不执行派生类的析构函数。

例12.3 基类中有非虚析构函数时的执行情况

```
#include <iostream>
using namespace std;
class Point                    //定义基类Point类
{ public:
  Point( ){ }                  //Point类构造函数
  ~Point()                     //Point类析构函数
  { cout    <<"executing Point destructor"
            <<endl;
  }
};
```

```
class Circle:public Point      //定义派生类Circle类
{ public:
  Circle( ){ }                 //Circle类构造函数
  ~Circle( )                   //Circle类析构函数
  { cout     <<"executing Circle destructor"
             <<endl;
  }
  private:
  int radius;
};
```

```
int main( )
{
   //用new开辟动态存储空间
   Point *p=new Circle;
   //用delete释放动态存储空间
   delete p;
   return 0;
}
```

程序只执行了基类Point的析构函数，而没有执行派生类Circle的析构函数。

原因是通过指向基类对象的指针，只能访问派生类中的基类成员，而不能访问派生类增加的成员。

如果希望能执行派生类Circle的析构函数，可以将基类的析构函数声明为虚析构函数，如

```
virtual ~Point()
{cout<<"executing Point destructor"<<endl;}
```

当基类的析构函数为虚函数时，无论指针指的是同一类族中的哪一个类对象，系统会采用动态关联，调用相应的析构函数，对该对象进行清理工作。

如果将基类的析构函数声明为虚函数时，所有派生类的析构函数也都自动成为虚函数。

最好把基类的析构函数声明为虚函数。

即使基类并不需要析构函数，也可以显式地定义一个函数体为空的虚析构函数，以保证在撤销动态分配空间时能得到正确的处理。

构造函数不能声明为虚函数。

12.4 纯虚函数与抽象类

12.4.1 纯虚函数

纯虚函数(pure virtual function)是在声明虚函数时被“初始化”为0的函数。

virtual double area( ) const =0;

声明纯虚函数的一般形式是

virtual 函数类型 函数名 (参数表列) =0;

注意

①纯虚函数没有函数体；

②最后面的“=0”不表示函数返回值为0，它只通知编译系统“这是纯虚函数”；

③这是一个声明语句，最后应有分号。

纯虚函数的作用是在基类中为其派生类保留一个函数的名字，以便派生类根据需要对它进行定义。

如果在一个类中声明了纯虚函数，而在其派生类中没有对该函数定义，则该虚函数在派生类中仍然为纯虚函数。

12.4.2 抽象类

抽象类(abstract class)又称抽象基类(abstract base class)，是一种不用来定义对象而只是作为一种基本类型用作基类的类。

抽象类的作用是作为一个类族的共同基类，或者说，为一个类族提供一个公共接口。

凡是包含纯虚函数的类都是抽象类。因为纯虚函数是不能被调用的，包含纯虚函数的类是无法建立对象的。

如果在派生类中没有对所有纯虚函数进行定义，则此派生类仍然是抽象类，不能用来定义对象。

抽象类不能定义对象(或者说抽象类不能实例化)，但是可以定义指向抽象类数据的指针变量。当派生类成为具体类之后，就可以用这种指针指向派生类对象，然后通过该指针调用虚函数，实现多态性的操作。

12.4.3 应用实例

例12.4 虚函数和抽象基类的应用。

在本章例12.1介绍了以Point为基类的点—圆—圆柱体类的层次结构。现在要对它进行改写，在程序中使用虚函数和抽象基类。

类的层次结构的顶层是抽象基类Shape(形状)。

Point(点), Circle(圆), Cylinder(圆柱体)都是Shape类的直接派生类和间接派生类。

程序如下:

第(1)部分

```
#include <iostream>
using namespace std;
class Shape  //声明抽象基类Shape
{
public:
   virtual double area( ) const {return 0.0;}      //虚函数
   virtual double volume() const {return 0.0;}  //虚函数
   virtual void shapeName() const =0;      //纯虚函数
};
```

第(2)部分

```
class Point:public Shape      //声明Point类
{
public:
   Point(double=0,double=0);
   void setPoint(double,double);
   double getX( ) const {return x;}
   double getY( ) const {return y;}
   virtual void shapeName( ) const {cout<<"Point:";}
   friend ostream & operator<<(ostream &,const Point &);
protected:
   double x,y;
};
```

```
//定义Point类成员函数
Point::Point(double a,double b)
{ x=a;y=b;}

void Point::setPoint(double a,double b)
{ x=a;y=b;}

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

第(3)部分

```
class Circle:public Point              //声明Circle类
{
public:
   Circle(double x=0,double y=0,double r=0);
   void setRadius(double);
   double getRadius( ) const;
   virtual double area( ) const;
   virtual void shapeName( ) const {cout<<"Circle:";}
   friend ostream &operator<<(ostream &,const Circle &);
protected:
   double radius;
};
```

```
//声明Circle类成员函数
Circle::Circle(double a,double b,double
 r):Point(a,b),radius(r){ }

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

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

double Circle::area( ) const {return 3.14159*radius*radius;}

ostream &operator<<(ostream &output,const Circle &c)
{
   output<<"["<<c.x<<","<<c.y<<"], r="<<c.radius;
   return output;
}
```

第(4)部分

```
class Cylinder:public Circle        //声明Cylinder类
{
public:
   Cylinder (double x=0,double y=0,double r=0,double h=0);
   void setHeight(double);
   virtual double area( ) const;
   virtual double volume( ) const;
   virtual void shapeName( ) const
   { cout<<"Cylinder:";}
   friend ostream& operator<<(ostream&,const Cylinder&);
protected:
   double height;
};
```

```
//定义Cylinder类成员函数
Cylinder::Cylinder(double a,double b,double
 r,double h):Circle(a,b,r),height(h){ }

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

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

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

```
//定义Cylinder类成员函数

ostream &operator<<(ostream &output,const
 Cylinder& cy)
{
   output  <<"["<<cy.x<<","<<cy.y
           <<"],r="<<cy.radius
           <<", h="<<cy.height;
   return output;
}
```

第(5)部分

```
int main( )
{
    Point point(3.2,4.5);
    Circle circle(2.4,1.2,5.6);
    Cylinder cylinder(3.5,6.4,5.2,10.5);
    point.shapeName();                    //静态关联
    cout<<point<<endl;
    circle.shapeName();                        //静态关联
    cout<<circle<<endl;
    cylinder.shapeName();                 //静态关联
    cout<<cylinder<<endl<<endl;
```

```
Shape *pt;                                  //定义基类指针
pt=&point;
pt->shapeName( );                           //动态关联
cout      <<"x="<<point.getX( )<<",y="
          <<point.getY( )<<"\narea="<<pt->area( )
          <<"\nvolume="<<pt->volume()<<"\n\n";
pt=&circle;
pt->shapeName( );                           //动态关联
cout      <<"x="<<circle.getX( )<<",y="
          <<circle.getY( )<<"\narea="
          <<pt->area( )  <<"\nvolume="
          <<pt->volume( )<<"\n\n";
```

```
    pt=&cylinder;
    pt->shapeName( );                    //动态关联
    cout     <<"x="<<cylinder.getX( )
             <<",y="<<cylinder.getY( )
             <<"\narea="<<pt->area( )
             <<"\nvolume="<<pt->volume( )<<"\n\n";
    return 0;
}
```

结论

(1) 一个基类如果包含一个或一个以上纯虚函数，就是抽象基类。抽象基类不能也不必要定义对象。

(2) 抽象基类与普通基类不同，它可以没有任何物理上的或其他实际意义方面的含义。

(3) 在类的层次结构中，顶层或最上面的几层可以是抽象基类。抽象基类体现了本类族中各类的共性，把各类中共有的成员函数集中在抽象基类中声明。

(4) 抽象基类是本类族的公共接口。或者说，从同一基类派生出的多个类有同一接口。

(5) 区别静态关联和动态关联。

(6) 如果在基类声明了虚函数，则在派生类中凡是与该函数完全相同的函数均为虚函数。

(7) 使用虚函数提高了程序的可扩充性。

作业

- **P416：4、5**