

# 第8讲多态性与虚函数

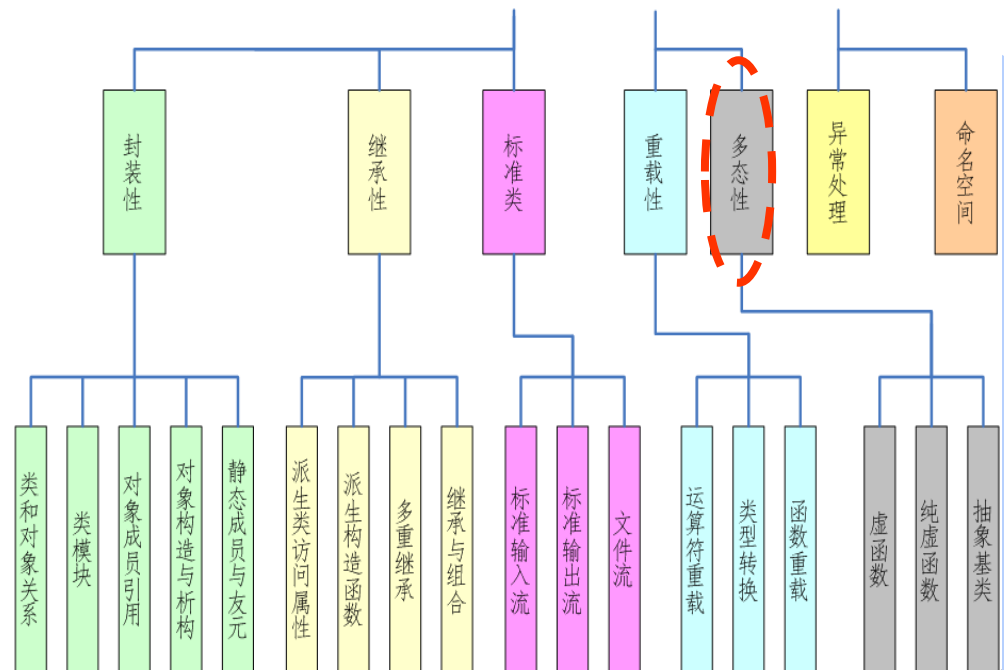
## 8.1 基类与派生类的兼容规则

## 8.2 多态性的概念

## 8.3 虚函数

## 8.4 纯虚函数与抽象类

C++的OOP程序 = 对象 + 消息 + 工具



## 8.1 基类与派生类的兼容规则

- 族类中，“类型兼容”规则指在需要基类对象的地方，都可使用公有派生类的对象来替代。替换后，派生类对象可作为基类对象使用，但只能使用派生类中的“基类成员”
- 派生类对象可以类型转换为基类对象  
基类对象 = 派生类对象
- 派生类对象可以类型转换为基类引用  
基类引用 = 派生类对象
- 派生类对象的指针可以类型转换为基类指针  
基类指针 = &派生类对象
- 类型转换后的对象或指针一律遵循基类接口的成员访问规则，但当派生类继承方式为`private`或`protected`时，禁止派生类向基类对象的类型转换



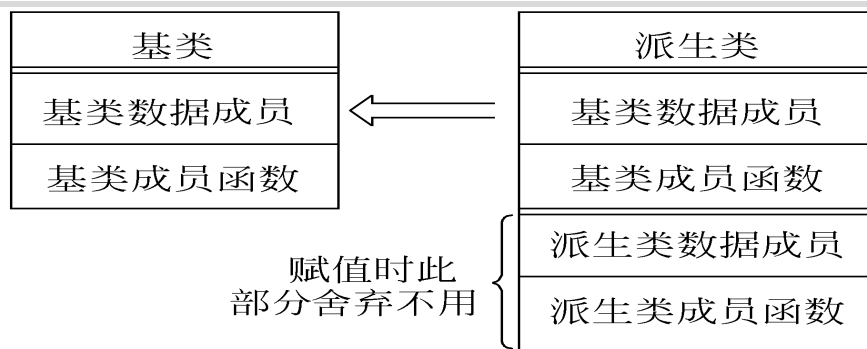
## (1) 派生类对象可以向基类对象赋值

A a1; //定义基类A对象a1

B b1; //定义类A公用派生类B对象b1

a1=b1; //用对象b1对基类对象a1赋值

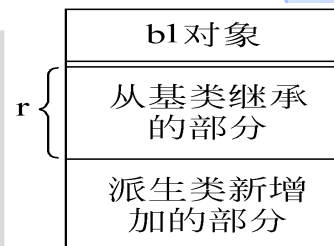
问题：是否可用基类对象对其派生类对象赋值？同一基类的不同派生类对象之间是否可以赋值？



## (2) 派生类对象可向基类对象的引用赋值

```

A a1; //定义基类A对象a1
B b1; //定义公用派生类B对象b1
A& r=a1; //定义基类A对象的引用变量r，并用a1对其
或 A &r=b1 //也可用派生类对象初始化引用变量r
    
```



□ 注意：此时r并不是b1的别名，只是b1中基类部分的别名，r与b1中基类部分共享同一段存储单元

### (3) 基类对象的指针变量可指向派生类对象

```

1  #include <iostream>
2  #include <string>
3  using namespace std;
4  class Student//声明Student类
5  {
6  public:
7      Student(int, string, float); //声明构造函数
8      void display(); //声明输出函数
9  private:
10     int num; string name; float score;};
11 Student::Student(int n, string nam, float s) //定义构造函数
12 {num=n; name=nam; score=s;}
13 void Student::display() //定义输出函数
14 {cout<<endl<<"num:"<<num<<endl;
15  cout<<"name:"<<name<<endl;
16  cout<<"score:"<<score<<endl;}
17 class Graduate:public Student //声明公用派生类Graduate
18 {
19 public:
20     Graduate(int, string, float, float); //声明构造函数
21     void display(); //声明输出函数
22 private:
23     float pay;};
24 Graduate::Graduate(int n, string nam, float s, float p):Student(n,nam,s),pay(p){ }
25 void Graduate::display() //定义输出函数
26 {Student::display(); //调用Student类的display函数
27  cout<<"pay="<<pay<<endl;}
28 int main()
29 {Student stud1(1001,"Li",87.5); //定义Student类对象stud1
30  Graduate grad1(2001,"Wang",98.5,563.5); //定义Graduate类对象grad1
31  Student *pt=&stud1; //定义指向Student类对象的指针并指向stud1
32  pt->display(); //调用stud1.display函数
33  pt=&grad1; //指针指向grad1
34  pt->display(); //调用grad1.display函数

```

C:\WINDOWS\system32\cmd.exe

```

num:1001
name:Li
score:87.5

```

```

num:2001
name:Wang
score:98.5

```

请按任意键继续. . .

**问题：**为什么程序中第31和29语句指向不同，却没有显示“pay”值？

# 分析下列程序结果

```
int main ()
{
    student
    stud1(1001, " li" , 87.5);
    graduate
    grad1(201, " wan" , 98, 532);
    student &stud-alias=grad1;
    student stud=grad1;
    stud1.display();
    grad1.display();
    stud.display();
    stud-alias.display();
    return( 0);
}
```

运行结果:

num:1001

name: li

scores:87.5

num:201

name: wan

Scores:98

Pay:532

num:201

name: wan

Scores:98

num:201

name: wan

Scores:98

## 8.2 多态性的概念

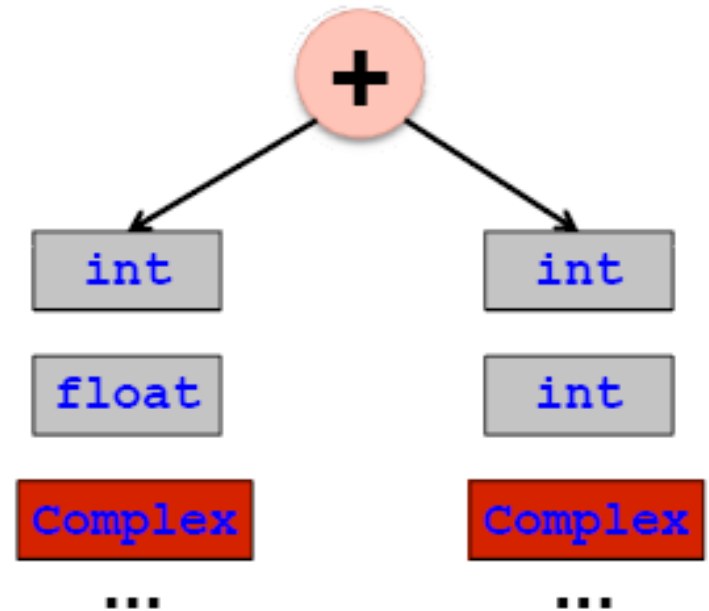
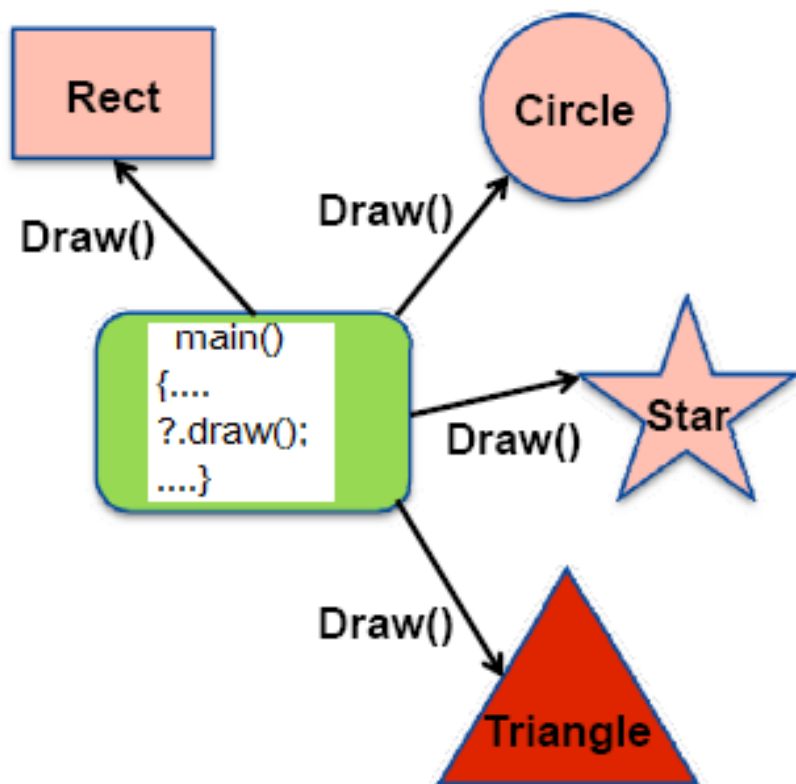
□ 多态性：程序向不同对象发送同一个消息（调用同名函数），不同对象在接收时会产生不同行为（实现不同功能）

□ 生活中的多态性例子





- 所谓**多态**，就是向不同的对象发送**同一消息**时，不同对象在接收时会产生**不同的行为**（即调用不同方法）



多态能对不同对象使用同样的接口，既提高程序维护的可靠性，又使得不同对象的内部设计与外部使用接口分离，实现软件分工协作。

## □ 多态性分类

- ✓ 重载多态：函数或运算符重载；
- ✓ 强制多态：数据或对象的类型强制转换；
- ✓ 包含多态：研究类族中定义同名成员函数的多态行为，主要通过虚函数来实现；
- ✓ 参数多态：类模板实例化时的多态性，即实例化后的各个类都具有相同的操作。

## □ 多态性实现方式分类

- ✓ 静态多态性：编译时多态性，例如：函数重载和运算符重载，强制类型转换等；
- ✓ 动态多态性：运行时多态性，程序运行过程中动态确定操作所针对的对象，主要通过虚函数(virtual function)实现。



# (1) 声明基类Point类

为什么不能改为Private

```

1 #include <iostream> //声明类Point
2 //using namespace std;
3 class Point
4 {public:
5     Point(float x=0,float y=0); //有默认参数的构造函数
6     void setPoint(float,float); //设置坐标值
7     float getX() const {return x;} //读x坐标
8     float getY() const {return y;} //读y坐标
9     friend ostream& operator << (ostream&, const Point&); //重载运算符"<<"
10 protected: //受保护成员
11     float x,y;};
12
13 Point::Point(float a,float b) //对x,y初始化
14 {x=a;y=b;} //设置x和y的坐标值
15 void Point::setPoint(float a,float b) //为x,y赋新值
16 {x=a;y=b;}
17 //重载运算符"<<", 使之能输出点的坐标
18 ostream & operator << (ostream &output,const Point &p)
19 {output<< "["<<p.x<<","<<p.y<< "]"<<endl;
20 return output;}
21 int main()
22 {Point p(3.5,6.4); //建立Point类对象p
23 cout<<"x="<<p.getX()<<","<<p.getY()<<endl; //输出p的坐标值
24 p.setPoint(8.5,6.8); //重新设置p的坐标值
25 cout<<"p(new):"<<p<<endl; } //用重载运算符"<<"输出p点坐标
26
27

```

```

C:\WINDOWS\system32\cmd.exe
x=3.5,y=6.4
p(new):[8.5,6.8]
请按任意键继续. . .

```

为什么出现这么多错?

```

输出
显示以下输出 (S): 生成
1>正在编译...
1>l.cpp
1>. \1.cpp (9) : error C2143: syntax error : missing ';' before '@'
1>. \1.cpp (9) : error C2433: 'ostream' : 'friend' not permitted on data declarations
1>. \1.cpp (9) : error C4430: missing type specifier - int assumed. Note: C++ does not support default-int
1>. \1.cpp (9) : error C2061: syntax error : identifier 'ostream'
1>. \1.cpp (9) : error C4430: missing type specifier - int assumed. Note: C++ does not support default-int
1>. \1.cpp (9) : error C2805: binary 'operator <<' has too few parameters
1>. \1.cpp (18) : error C2143: syntax error : missing ';' before '@'
1>. \1.cpp (18) : error C4430: missing type specifier - int assumed. Note: C++ does not support default-int
1>. \1.cpp (18) : error C2066: 'int ostream' : redefinition
1>. \1.cpp (9) : see declaration of 'ostream'
1>. \1.cpp (18) : error C2065: 'output' : undeclared identifier
1>. \1.cpp (18) : error C2059: syntax error : 'const'
1>. \1.cpp (19) : error C2143: syntax error : missing ';' before '{'
1>. \1.cpp (19) : error C2447: '{' : missing function header (old-style formal list?)

```

## (2) 声明派生类Circle

```

1  #include <iostream> //声明类Point
2  using namespace std;
3  class Point
4  {public: Point(float x=0,float y=0); //有默认参数的构造函数
5      void setPoint(float,float); //设置坐标值
6      float getX() const {return x;} //读x坐标
7      float getY() const {return y;} //读y坐标
8      friend ostream& operator << (ostream&, const Point&); //重载运算符"<<"
9  protected: float x,y;};
10 Point::Point(float a,float b) //对x,y初始化
11     {x=a;y=b;} //设置x和y的坐标值
12 void Point::setPoint(float a,float b) {x=a;y=b;} //为x,y赋新值
13 ostream & operator<<(ostream &output,const Point &p)
14     {output<<"["<<p.x<<","<<p.y<<""]<<endl; return output;}
15 class Circle:public Point//circle是Point类的公用派生类
16     {public: Circle(float x=0,float y=0,float r=0); //构造函数
17         void setRadius(float); //设置半径值
18         float getRadius() const; //读取半径值
19         float area () const; //计算圆面积
20         friend ostream &operator<<(ostream &,const Circle &);
21     private: float radius;};
22 //定义构造函数, 对圆心坐标和半径初始化
23 Circle::Circle(float a,float b,float r):Point(a,b),radius(r){ }
24 void Circle::setRadius(float r) {radius=r;} //设置半径值
25 float Circle::getRadius() const {return radius;} //读取半径值
26 float Circle::area() const{return 3.14159*radius*radius;}
27 ostream &operator<<(ostream &output,const Circle &c)
28     {output<<"Center=["<<c.x<<","<<c.y<<""],r="<<c.radius<<"",area="<<c.area()<<endl;
29     return output;}
30 int main()
31 {Circle c(3.5,6.4,5.2); //建立Circle类对象c, 并给定圆心坐标和半径
32  cout<<"original circle:\nx="<<c.getX()<<"",y="<<c.getY()<<"",r="<<c.getRadius()
33  <<"",area="<<c.area()<<endl; //输出圆心坐标、半径和面积
34  c.setRadius(7.5); c.setPoint(5,5); //设置圆半径和圆心坐标值x,y
35  cout<<"new circle:\n"<<c; //用重载运算符"<<"输出圆对象的信息
36  Point &pRef=c; //pRef是Point类的引用变量, 被c初始化
37  cout<<"pRef:"<<pRef; return 0;} //输出pRef的信息

```

重载多态

```

C:\WINDOWS\system32\cmd.exe
original circle:\nx=3.5, y=6.4, r=5.2, area=84.9486
new circle:\nCenter=[5,5],r=7.5,area=176.714
pRef: [5,5]
请按任意键继续. . .

```

Ref为什么不是C引用

调用哪个重载 (<<) ?

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4 class Student//声明基类Student
5 {public:
6     Student(int, string, float);//声明构造函数
7     void display();           //声明输出函数
8     protected:               //受保护成员，派生类可以访问
9         int num;
10        string name;
11        float score; };
12 //Student类成员函数的实现
13 Student::Student(int n, string nam, float s) //定义构造函数
14 { num=n; name=nam; score=s; }
15
16 void Student::display() //定义输出函数
17 { cout<<"num:"<<num<<"\nname:"<<name<<"\nscore:"<<score<<"\n\n"; }
18 //声明公用派生类Graduate
19 class Graduate:public Student
20 {public:
21     Graduate(int, string, float, float); //声明构造函数
22     void display(); //声明输出函数
23 private:
24     float pay; };
25 // Graduate类成员函数的实现
26 void Graduate::display() //定义输出函数
27 { cout<<"num:"<<num<<"\nname:"<<name<<"\nscore:"<<score<<"\npay="<<pay<<endl; }
28 Graduate::Graduate(int n, string nam, float s, float p):Student(n,nam,s),pay(p){ }
29 int main()
30 { Student stud1(1001,"Li",87.5); //定义Student类对象stud1
31   Graduate grad1(2001,"Wang",98.5,563.5); //定义Graduate类对象grad1
32   Student *pt=&stud1; //定义指向基类对象的指针变量pt
33   pt->display();
34   pt=&grad1;
35   pt->display(); return 0; }

```

```

c:\ C:\WINDOWS\system32\cmd.exe
num:1001
name:Li
score:87.5

num:2001
name:Wang
score:98.5

请按任意键继续. . .

```

```

1 #include <iostream>
2 #include <string>
3 using namespace std;
4 class Student//声明基类Student
5 {public:
6     Student(int, string, float);//声明构造函数
7     virtual void display(); //void display();声明虚函数
8 protected: //受保护成员, 派生类可以访问
9     int num;
10    string name;
11    float score; };
12 //Student类成员函数的实现
13 Student::Student(int n, string nam, float s) //定义构造函数
14 {num=n; name=nam; score=s; }
15
16 void Student::display() //定义输出函数
17 {cout<<"num:"<<num<<"\nname:"<<name<<"\nscore:"<<score<<"\n\n";}
18 //声明公用派生类Graduate
19 class Graduate:public Student
20 {public:
21     Graduate(int, string, float, float); //声明构造函数
22     void display(); //声明输出函数
23 private:
24     float pay;};
25 // Graduate类成员函数的实现
26 void Graduate::display() //定义输出函数
27 {cout<<"num:"<<num<<"\nname:"<<name<<"\nscore:"<<score<<"\npay="<<pay<<endl;}
28 Graduate::Graduate(int n, string nam, float s, float p):Student(n, nam, s), pay(p){ }
29 int main()
30 {Student stud1(1001, "Li", 87.5); //定义Student类对象stud1
31 Graduate grad1(2001, "Wang", 98.5, 563.5); //定义Graduate类对象grad1
32 Student *pt=&stud1; //定义指向基类对象的指针变量pt
33 pt->display();
34 pt=&grad1;
35 pt->display(); return 0; }

```

```

C:\WINDOWS\system32\cmd.exe
num:1001
name:Li
score:87.5

num:2001
name:Wang
score:98.5
pay=563.5
请按任意键继续. . .

```

```

C:\WINDOWS\system32
num:1001
name:Li
score:87.5

num:2001
name:Wang
score:98.5

请按任意键继续. . .

```

## 8.3 虚函数

- ❑ 函数重载：函数名相同但参数不同
  - ❑ 同名覆盖：在类族中，不同层次可能出现名字、参数个数和类型都相同而功能不同的函数。
- 
- ❑ 同名函数的调用方法
  - ❑ 通过类名限定符来唯一标识：`cy1.Circle::area( )`。通过不同的对象名去调用不同派生层次中的同名函数；
  - ❑ 虚函数：解决多层继承的同名函数统一调用问题。用同一调用形式，既能调用派生类、也能调用基类的同名函数。例如，用同一语句“`pt->display( );`”，在调用前给指针变量`pt`赋以不同类对象，即可调用不同派生层次中的`display`函数。

- 静态多态性的本质在于：你必须先告诉编译器对象的类型，然后编译器将根据该类型在编译时决定对象的行为
- 如果有多个Cylinder、Circle对象，它们只能被分开管理，不能共用一段代码：

```
Cylinder arrCyl[10];
... //输入每个数组元素的值
for (int i=0; i<10; i++)
{
    cout << arrCyl[i].area()
        << endl;
}
```

```
Circle arrCr[5];
... //输入每个数组元素的值
for (int i=0; i<5; i++)
{
    cout << arrCr[i].area()
        << endl;
}
```

能否共用  
一段代码？



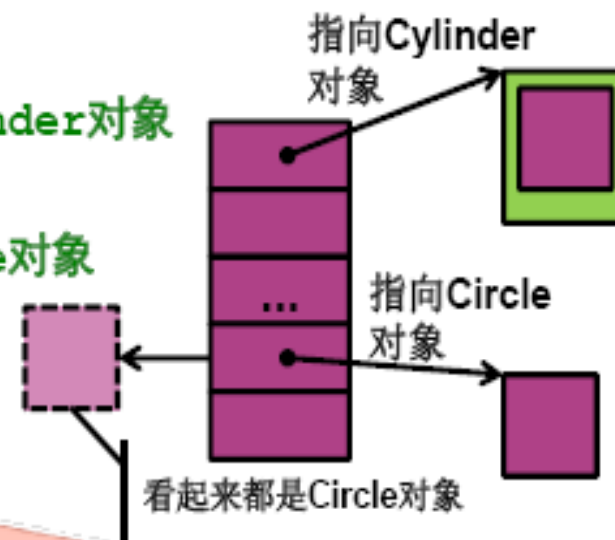
```
Circle * arrMix[15];
... //输入每个数组元素的值,既有Cylinder,也有Circle
for (int i=0; i<15; i++)
{
    cout << arrMix[i]->area() << endl;
}
```



- 如果希望由运行时对象在内存中的类型决定其行为，而不是在编译时按代码指定的类型决定其行为，需要引入动态多态性技术，C++使用虚函数解决这个问题

```
Circle * arrMix[15];
for (int i=0; i<10; i++) //前10个为Cylinder对象
    arrMix[i] = new Cylinder(0,0,i,i);
for (int i=10; i<15; i++) //后5个为Circle对象
    arrMix[i] = new Circle(0,0,i);
for (int i=0; i<15; i++)
{   cout << arrMix[i]->area() << endl;
}
```

```
class Circle
{   ...
    virtual float area() const;
    ...
};
```



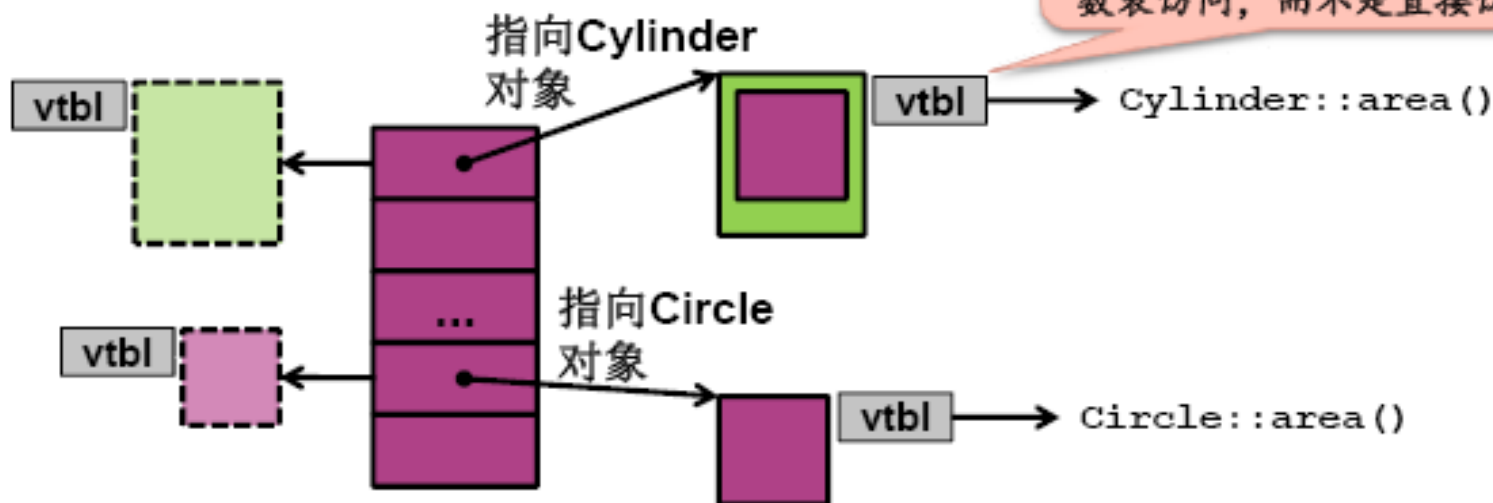
在当前类声明情况下，`arrMix[i]->area()` 只会调用 `Circle::area()`，因为每一个新建对象均被指派为 `Circle*`（基类指针）存放在 `arrMix` 数组中。



- 使用**Virtual**将基类成员函数声明为虚函数并且在派生类实现同名覆盖之后，同样使用基类指针访问基类和派生类对象时，将根据运行时指针**指向的内存对象的实际类型**调用相应的同名覆盖函数。

**virtual** 函数类型 函数名(参数表);

秘诀在于C++编译器让声明为virtual的函数通过虚函数表访问，而不是直接访问



```
Circle * arrMix[15];
...
cout << arrMix[i]->area();
```

也就是说，这些语句在编译时还不能确定调用哪一个area()函数，必须等候运行时arrMix[i]所存放的对象生成后才能决定

```
1. class Circle : public Point {
2.     float r;
3. public:
4.     Circle(float xx=0, float yy=0, float rr=0);
5.     void setRadius(float rr) { r=rr; }
6.     float getRadius() const { return r; }
7.     virtual float area() const;
8.     friend ostream& operator <<(ostream& out, const Circle& cr);
9. };

10. class Cylinder : public Circle {
11.     float h;
12. public:
13.     Cylinder(float xx=0, float yy=0, float rr=0, float hh=0);
14.     void setHeight(float hh) { h=hh; }
15.     float getHeight() const { return h; }
16.     float area() const;
17.     float volume() const;
18.     friend ostream& operator <<(ostream& out, const Cylinder& l);
19. };
```

```

1.  Circle::Circle(float xx, float yy, float rr)
2.  : Point(xx,yy), r(rr) {}
3.  float Circle::area() const //虚函数实现
4.  {  cout << "Circle area: ";
5.      return 3.14159*r*r; }
6.  ostream& operator <<(ostream& out, const Circle& cr)
7.  {  out << "(" << cr.getX() << "," << cr.getY() << ","
8.      << cr.r << "," << cr.area() << ")" << endl;
9.      return out;
10. }

11. Cylinder::Cylinder(float xx, float yy, float rr, float hh)
12. : Circle(xx,yy,rr), h(hh) {}
13. float Cylinder::area() const //虚函数实现
14. {  cout << "Cylinder area: ";
15.      return Circle::area()*2 + 3.14159*2*getRadius()*h; }
16. float Cylinder::volume() const
17. {  return Circle::area()*h; }
18. ostream& operator <<(ostream& out, const Cylinder& l)
19. {  out << "(" << l.getX() << "," << l.getY() << "," <<
20.      l.getRadius() << "," << l.h << "," << l.area() << "," <<
21.      l.volume() << ")" << endl;
22.      return out;
23. }

```

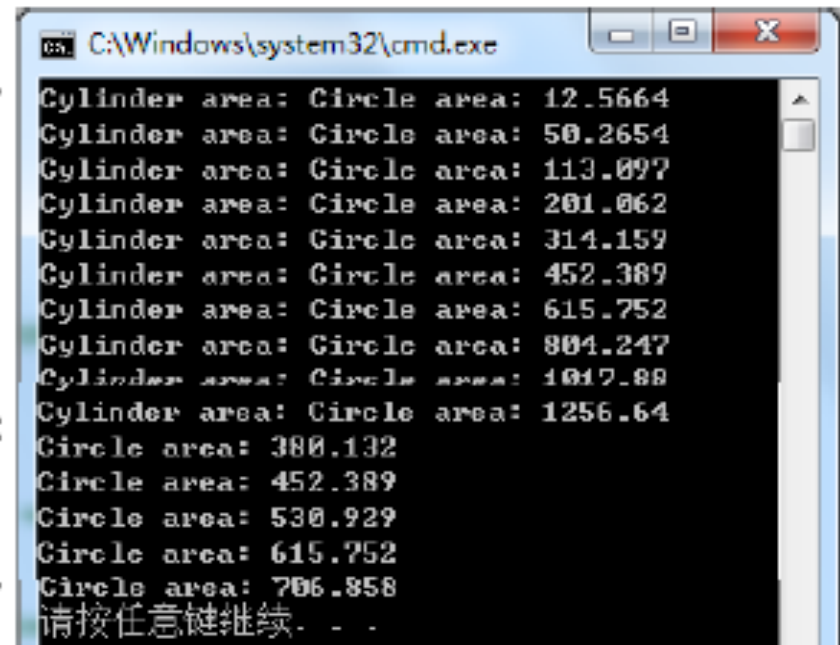
```

1. void main()
2. {
3.     Circle * arrMix[15];
4.     for (int i=0; i<10; i++) //前10个为Cylinder对象
5.         arrMix[i] = new Cylinder(0,0,i+1,i+1);
6.     for (int i=10; i<15; i++) //后5个为Circle对象
7.         arrMix[i] = new Circle(0,0,i+1);
8.     for (int i=0; i<15; i++)
9.     { cout << arrMix[i]->area() << endl;
10.    }
11.    ...
12. }

```

前10个调用  
Cylinder::area()

后5个调用  
Circle::area()



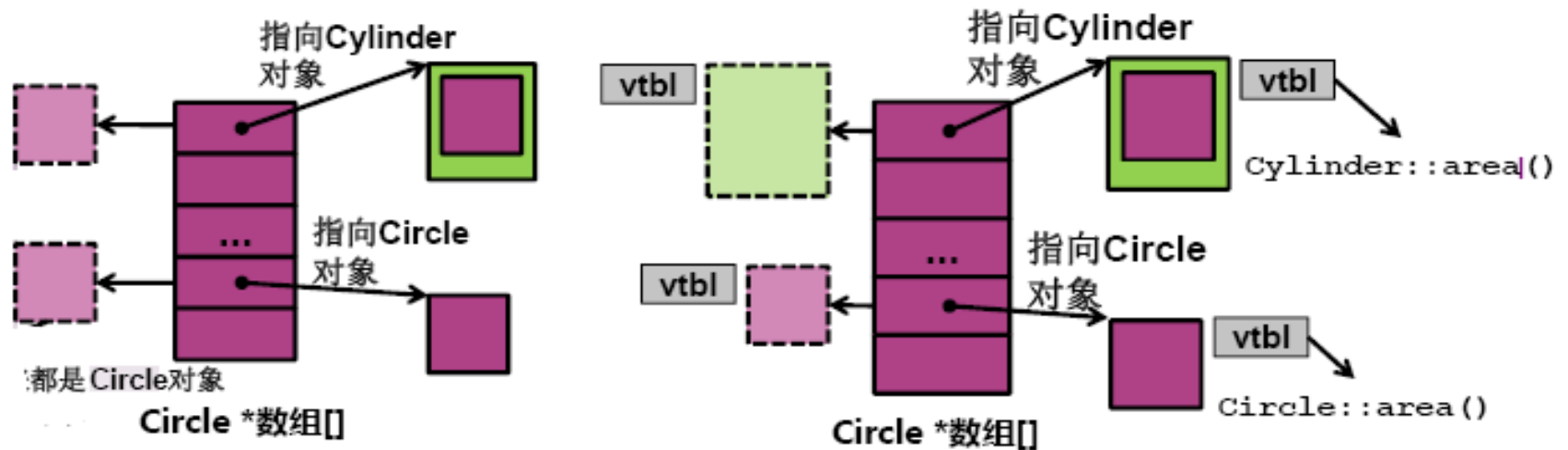
```

C:\Windows\system32\cmd.exe
Cylinder area: Circle area: 12.5664
Cylinder area: Circle area: 50.2654
Cylinder area: Circle area: 113.097
Cylinder area: Circle area: 201.062
Cylinder area: Circle area: 314.159
Cylinder area: Circle area: 452.389
Cylinder area: Circle area: 615.752
Cylinder area: Circle area: 804.247
Cylinder area: Circle area: 1017.88
Cylinder area: Circle area: 1256.64
Circle area: 380.132
Circle area: 452.389
Circle area: 530.929
Circle area: 615.752
Circle area: 706.858
请按任意键继续. . .

```

## 虚函数的作用

- ◆ 允许在派生类中重新定义与基类同名的函数，且可以通过**基类指针或引用**来访问基类和派生类中的同名函数
- ◆ 实现动态多态性：**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



不使用虚函数

```
Circle cir(0,0,2);
Cylinder cyl(-1,2,2,10);
Circle * pt = &cir;
cout << pt->area;
pt = &cyl;
cout << pt->area;
```

使用虚函数

## □ 虚函数使用方法

- ① 在基类用virtual声明成员函数为虚函数。这样就可以在派生类中重新定义此函数，为它赋予新的功能；
- ② 在类外定义虚函数时，不必再加virtual；
- ③ 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，并根据派生类的需要重新定义函数体；
- ④ 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。因此在派生类重新声明该虚函数时，virtual可加可不加，建议加；
- ⑤ 注意虚基类和虚函数的区别，虚基类解决多重继承的同名成员的重复继承问题。



## 8.3 虚函数

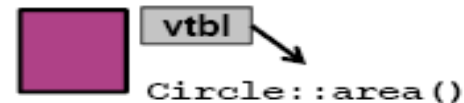
- ❑ 关联(binding)：确定调用具体对象的过程，即把一个函数名与一个类对象捆绑在一起；
- ❑ 静态关联：函数重载和通过对象名调用的虚函数，在编译时即可确定其调用的函数属于哪一个类。

- 如前所述，虚函数实现动态多态的原理在于C++编译器为包含虚函数的基类及其派生类族提供了**虚函数表**

```
Circle * pt;
... //pt指向基类或派生类对象
pt->area(); // (*pt->vtbl[0])()
```



- 由于编译时无法确定pt指针所指向对象的虚函数表中对应于area函数的一项具体填的是哪一个类的area函数地址，必须要到运行时才能确定具体调用的area函数版本，所以这种函数行为的绑定称为**动态绑定**，又叫**运行时绑定**



相比之下，函数重载、运算符重载、自动类型转换等均在编译时确定最终调用的函数地址，所以称为**静态关联**



## 8.3 虚函数

### □问题分析

- (1) 类外定义的非成员函数是否可以声明为虚函数？
- (2) 类的静态成员函数是否可以申明虚函数？
- (3) 内联函数是否可以申明虚函数？
- (4) 构造函数是否可以申明虚函数？
- (5) 析构函数是否可以申明虚函数？
- (6) 如果某基类成员函数被声明为虚函数后，在同一类族中的派生类是否可以再定义一个非virtual. 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？
- (7) 如果对多层继承的同名成员函数的调用是通过对象名，而不是通过基类指针或引用去访问，则声明虚函数有作用吗？

## □ 小结：虚函数与重载函数的关系

(1) 重载函数函数名称相同，参数不同；重载函数是在**作用域相同**的区域里定义的不同名字的不同函数；

(2) 虚函数函数原型完全一致，体现在基类和派生类的类层次结构中；

(3) 重载函数可以是成员函数或友员函数或一般函数，而虚函数**只能是成员函数**；

(4) 调用重载函数以所传递参数序列的差别作为调用不同函数的依据；虚函数则根据对象的不同来调用不同类的虚函数；

(5) 重载函数在编译时表现出多态性，是静态联编（早期绑定）；而虚函数在运行时表现出多态性，是动态联编（晚期绑定），动态联编是C++的精髓。

## 8.3 虚函数

### ◆ 虚析构造函数

□ 用new运算符建立了临时对象。用delete运算符撤销对象时，会发生什么情况？

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

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

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

注意指针变量P的类型

运行结果: Executing point destructor

□如将基类析构函数声明为虚析构函数，结果会怎样？此时调用析构函数是动态关联还是静态关联？

□**建议：**一般情况下，最好将基类析构函数声明为虚函数。

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

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

Excuting circle destructor

Excuting point destructor

## 8.4 纯虚函数与抽象类

- ❑ 基类中定义虚函数可能不是基类本身需求，而是派生类需要。例：基类Point中没有面积，但在其派生类Circle和Cylinder中都需要area函数
- ❑ 纯虚函数：基类申明时只申明一个函数名，没有函数体，即在申明时被“初始化”为0的函数。例：在Point类中  

```
virtual float area( ) const =0;
```
- ❑ 一般格式：virtual 函数类型 函数名 (参数表列) =0;
- ❑ 注意：
  - ✓ 纯虚函数没有函数体；最后的“=0”不表示函数返回值为0，只告诉编译系统“这是纯虚函数”；
  - ✓ 纯虚函数只有函数名而不具备函数功能，不能被调用。其作用是在基类中为其派生类保留一个函数名，以便派生类根据需要对它进行定义。其目地是实现多态性；
  - ✓ 问题：函数体为空的虚函数是纯虚函数，对吗？

## 8.4 纯虚函数与抽象类

- **抽象类 (abstract class)** : 凡是包含纯虚函数的类。因为纯虚函数是不能被调用，包含纯虚函数的类是无法建立对象；
- **目的**：不用来定义对象，只作为基类去建立派生类。用户在此基础上根据需求定义出功能各异的派生类，再用这些派生类去建立对象。
- **建议**
  - ✓ 面向对象编程，其层次结构的顶部是抽象类，甚至有几层都是抽象类。如果在抽象类所派生出的新类中对所有纯虚函数进行了定义，被赋予了功能。则该派生类就成为**可定义对象的具体类** (concrete class)。否则，仍为抽象类；
  - ✓ 抽象类不能定义对象，但可定义指向抽象类数据的指针变量。当派生类成为具体类后，就可以用该指针指向派生类对象，然后通过指针调用虚函数，实现多态性；
  - ✓ 抽象类不能作为参数、函数或显式转换等类型。

```

1  #include <iostream>
2  using namespace std;
3  class Shape//声明抽象基类Shape
4  {public:
5      virtual float area() const {return 0.0;}//虚函数
6      virtual float volume() const {return 0.0;} //虚函数
7      virtual void shapeName() const =0; //纯虚函数
8  }
9  class Point:public Shape//Point是Shape的公用派生类
10 {public:
11     Point(float=0,float=0);
12     void setPoint(float,float);
13     float getX() const {return x;}
14     float getY() const {return y;}
15     virtual void shapeName() const {cout<<"Point:";} //对虚函数进行再定义
16     friend ostream & operator<<(ostream &,const Point &);
17 protected:
18     float x,y;};
19 //定义Point类成员函数
20 Point::Point(float a,float b)
21 {x=a;y=b;}
22 void Point::setPoint(float a,float b)
23 {x=a;y=b;}
24 ostream & operator<<(ostream &output,const Point &p)
25 {output<<"["<<p.x<<","<<p.y<<"]"; return output;}
26

```



```

27 class Circle:public Point //声明Circle类
28 {public:
29     Circle(float x=0,float y=0,float r=0);
30     void setRadius(float);
31     float getRadius( ) const;
32     virtual float area( ) const;
33     virtual void shapeName( ) const {cout<<"Circle:";} //对虚函数进行再定义
34     friend ostream &operator<<(ostream &,const Circle &);
35 protected:
36     float radius;};
37 //声明Circle类成员函数
38 Circle::Circle(float a,float b,float r):Point(a,b),radius(r){ }
39 void Circle::setRadius(float r){radius=r;}
40 float Circle::getRadius( ) const {return radius;}
41 float Circle::area( ) const {return 3.14159*radius*radius;}
42 ostream &operator<<(ostream &output,const Circle &c)
43 { output<<"["<<c.x<<","<<c.y<<"], r="<<c.radius;
44     return output;}
45
46 class Cylinder:public Circle//声明Cylinder类
47 {public:
48     Cylinder (float x=0,float y=0,float r=0,float h=0);
49     void setHeight(float);
50     virtual float area( ) const;
51     virtual float volume( ) const;
52     virtual void shapeName( ) const {cout<<"Cylinder:";} //对虚函数进行再定义
53     friend ostream& operator<<(ostream&,const Cylinder&);
54 protected:
55     float height;
56 };

```

```

57 //定义Cylinder类成员函数
58 Cylinder::Cylinder(float a,float b,float r,float h)
59 { :Circle(a,b,r),height(h){ }
60 void Cylinder::setHeight(float h){height=h;}
61 float Cylinder::area() const
62 { return 2*Circle::area()+2*3.14159*radius*height;}
63 float Cylinder::volume() const
64 {return Circle::area()*height;}
65 ostream &operator<<(ostream &output,const Cylinder& cy)
66 { output<<"["<<cy.x<<","<<cy.y<<"],r="
67 <<cy.radius<<","<<h="<<cy.height; return output;}
68
69 int main( )
70 {Point point(3.2,4.5); //建立Point类对象point
71 Circle circle(2.4,1.2,5.6); //建立Circle类对象circle
72 Cylinder cylinder(3.5,6.4,5.2,10.5); //建立Cylinder类对象cylinder
73 point.shapeName(); //静态关联
74 cout<<point<<endl;
75 circle.shapeName(); //静态关联
76 cout<<circle<<endl;
77 cylinder.shapeName(); //静态关联
78 cout<<cylinder<<endl<<endl;
79 Shape *pt; //定义基类指针
80 pt=&point; //指针指向Point类对象
81 pt->shapeName(); //动态关联
82 cout<<"x="<<point.getX()<<","<<y="<<point.getY()<<"\narea="<<pt->area()
83 <<"\nvolume="<<pt->volume()<<"\n\n";
84 pt=&circle; //指针指向Circle类对象
85 pt->shapeName(); //动态关联
86 cout<<"x="<<circle.getX()<<","<<y="<<circle.getY()<<"\narea="<<pt->area()
87 <<"\nvolume="<<pt->volume()<<"\n\n";
88 pt=&cylinder; //指针指向Cylinder类对象
89 pt->shapeName(); //动态关联
90 cout<<"x="<<cylinder.getX()<<","<<y="<<cylinder.getY()<<"\narea="<<pt->area()
91 <<"\nvolume="<<pt->volume()<<"\n\n";
92 return 0;}

```

```

C:\WINDOWS\system32\cmd.exe
Point:[3.2,4.5]
Circle:[2.4,1.2], r=5.6
Cylinder:[3.5,6.4], r=5.2, h=10.5

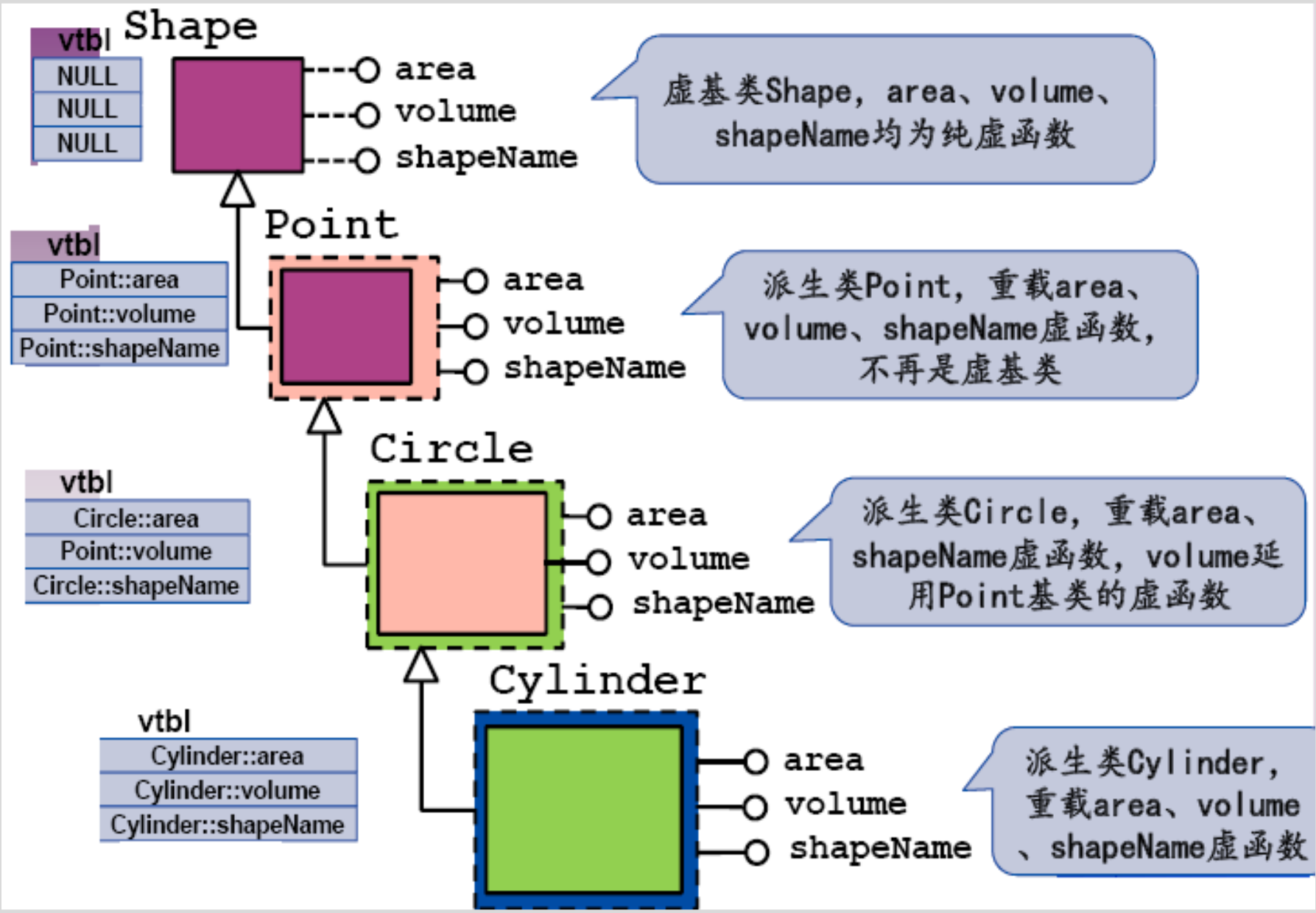
Point:x=3.2,y=4.5
area=0
volume=0

Circle:x=2.4,y=1.2
area=98.5203
volume=0

Cylinder:x=3.5,y=6.4
area=512.959
volume=891.96

请按任意键继续. . .

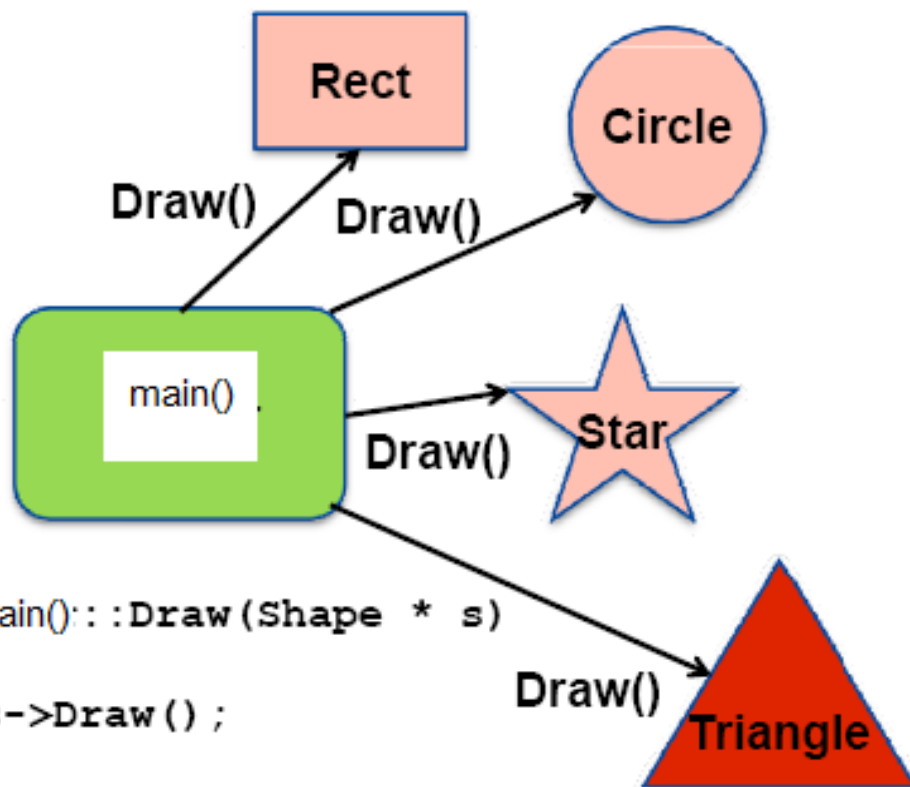
```



## 本讲重点分析

- 所谓**多态**，就是指向不同的对象发送**同一消息**时，不同的对象在接收时会产生**不同的行为**
- 函数重载、运算符重载、类型转换属于**静态多态技术**
- 为了把对象的行为留到程序运行时动态确定，需要使用**动态多态技术**，通过**虚函数**实现
- 包含虚函数的基类，其**析构函数**最好也声明为虚函数
- 引入**多态性的好处**：类的设计者与类的使用者分工协作
- 引入**抽象基类**的意义：抽象基类体现了本类族中各类的共性，把本类族中共有的成员函数集中在抽象基类中声明，作为本类族的公共接口。从抽象基类派生出的多个类具有共同的接口

# 本讲重点分析



多态的真正威力在于：你可以事先设计一些使用其它模块的代码，对模块的实现细节却可以一无所知。这样，一个模块的修改乃至功能的添加都不会再会影响到其它模块的代码了。

类的使用者与类的设计者分离。程序员的注意力集中在处理普遍性，而到执行环境中再处理特殊性。多态性把操作的细节留给类的设计者(多为专业人员)去完成，而让程序人员(类的使用者)只需要做一些宏观性的工作，告诉系统做什么，而不必考虑怎么做，极大地简化了应用程序的编码工作。

## 第8次作业必做题目2道（10周末交）

第1题要求：在第7次作业第2题基础上，对程序进行如下改进从teacher类派生Prof（教授）类，从Prof类派生Female-Prof类（女教授）；对这3类分别定义一个同名函数（PAY（））来计算3类人员的工资；并要求在main()函数中，采用指向Teacher类指针变量来调用这3个类中的PAY（）函数。这3类人员的工资如何发放，权利交给同学；但要求不一样。建议用多态性来实现程序功能。

第2题：声明一个哺乳动物Mammal类，再由此派生出狗Dog类，二者都定义Speak（）成员函数（函数内容自己确定），基类中定义为虚函数。声明一个Dog类的对象，调用Speak（）函数，观察运行结果。



选做题：声明一个Shape抽象类，在此基础上派生出Rectangle和Circle类，二者都有GetArea( )函数计算对象的面积，GetPerim( )函数计算对象的周长。给定部分程序代码，根据题意要求和基类代码，完成整个程序？

```
#include<iostream.h>

class Sharp
{
public:
    Sharp() {}
    ~Sharp() {}
    virtual float GetArea()=0;
    virtual float GetPerim()=0;
};
```



# 面向对象程序设计特点总结

- 抽象性：类是对象的抽象，对象是类的具体实例；类模板是类的抽象，类是类模板的具体实例；基类是派生类的抽象，派生类是基类的具体实例等；
- 封装性：类成员的3种访问属性；公有接口与私有实现的分离；类申明与成员函数定义的分离；构造函数；
- 继承性：3种继承方式，派生类成员的4种访问属性；派生类的构造函数；多重继承中函数同名问题与虚基类；
- 多态性：函数重载、运算符重载、多层派生的函数同名问题与虚函数；纯虚函数与抽象类。

# 面向对象程序设计关键技术总结

- 对象成员的3种访问方式；6种构造函数；对象指针（this）；数据保护的6种常类型；对象动态建立与释放；对象的复制（浅拷贝与深拷贝问题）；静态成员，友元函数；
- 运算重载规则；成员函数与友元函数重载；类型转换函数与转换构造函数；
- 派生类的构造函数的实现方法；基类与派生类的兼容性问题；
- 虚函数的引用方法；虚析构函数；
- Cin和cout；文件操作与文件流；
- 异常处理。