



中国矿业大学(北京)  
China University of Mining and Technology (Beijing)

# 面向对象技术与C++程序设计

## 第六章 多态

授课教师：张潇

机电与信息工程学院


计算机系






## 第六章 多态

**多态性**是面向对象程序设计语言的又一重要特征，它是指不同对象接收到同一消息时会产生不同的行为。继承所处理的是**类与类之间的层次关系**问题，而多态则是处理类的层次结构之间，以及同一个类内部**同名函数**的关系问题。简单地说，多态就是在同一个类或继承体系结构的基类与派生类中，用同名函数来实现各种不同的功能。



多态与联编



运算符重载



虚函数



纯虚函数、抽象类



## 6.1 多态与联编

### 1、多态polymorphism

- 对象根据所接收的消息而做出动作，同样的消息为不同的对象接收时可导致完全不同的行动，该现象称为多态性。
- 简单的说：**单接口，多实现**

### 2、联编

一个程序常常会调用到来自于不同文件或C++库中的资源（如函数、对话框）等，需要经过编译、连接才能形成为可执行文件，在这个过程中要把调用函数名与对应函数（这些函数可能来源于不同的文件或库）关联在一起，这个过程就是**绑定**（binding），又称**联编**。



## 6.1 静态绑定和动态绑定

### 3、静态绑定与静态绑定

- **静态绑定**又称静态联编，是指在编译程序时就根据调用函数提供的信息，把它所对应的具体函数确定下来，即在编译时就把调用函数名与具体函数绑定在一起。
- **动态绑定**又称动态联编，是指在编译程序时还不能确定函数调用所对应的具体函数，只有在程序运行过程中才能够确定函数调用所对应的具体函数，即在程序运行时才把调用函数名与具体函数绑定在一起。



## 4、多态性的实现

- 编译时多态性： ---静态联编(连接)----系统在编译时就决定如何实现某一动作, 即对某一消息如何处理. 静态联编具有执行速度快的优点. 在C++中的编译时多态性是通过**函数重载和运算符重载**实现的。
- 运行时多态性： ---动态联编(连接)----系统在运行时动态实现某一动作, 即对某一消息在运行过程实现其如何响应. 动态联编为系统提供了灵活和高度问题抽象的优点, 在C++中的运行时多态性是通过**继承和虚函数**实现的。



## 6.2 函数重载

### 1、函数重载的意义

函数重载意义在于用户可以通过同一个函数名访问某一类或一组相关操作的函数，由编译器决定具体的函数调用，有助于复杂问题的简化。

### 2、函数重载的类型

- 函数重载可分为普通函数重载、类成员函数重载以及继承结构中基类和派生类成员函数的重载几种情况。

### 3、函数重载的要求

- 要求重载函数具有不同的函数原型，即重载函数要在**参数类型、参数个数或参数顺序**上有所区别，不能有同一作用域内的两个同名函数具有完全相同的参数表。



**【例6-1】重载复数的加法，实现复数实部与普通数据相加，两个复数相加的运算。**

```
#include<iostream>
using std::cout;
using std::endl;
class Complex{
    double real,imag;
public:
    Complex(double r=0,double i=0):real(r),imag(i){}
    Complex Add(double x){
        return Complex(x+real, imag);}
    Complex Add(Complex c){
        return Complex(real+c.real, imag+c.imag);}
    void display(){cout<<real<<" + "<<imag<<"i"<<endl;}
    double getReal(){return real;}
    double getImag(){return imag;}
};
```

. . . . .



```
Complex Add(double r,Complex c){  
    return(r+c.getReal(),c.getImag ());  
}  
Complex Add(Complex c1,Complex c2)  
{  
    return Complex(c1.getReal()+c2.getReal(),  
        c1.getImag()+c2.getImag());  
}  
int main()  
{  
    Complex c1(2,3),c2(5,6),c3;  
    c3=c1.Add(10); c3.display(); //L1  
    c3=c1.Add(c2); c3.display(); //L2  
    c3=Add(c1,c2); c3.display(); //L3  
    Add(Complex,Complex);  
}
```





## 6.3 运算符重载

- **运算符重载是C++的一项强大功能。通过重载，可以扩展C++运算符的功能，使它们能够操作用户自定义的数据类型，增加程序代码的直观性和可读性。**
- 本章主要介绍 类成员运算符重载与友元运算符重载，二元运算符与一元运算符重载，运算符++、--、[] 重载，及 流运算符<<和>>的重载



## 6.3.1 运算符重载的概念

### 1、运算符重载的概念

- C++的运算符对语言预定义类型是重载的
  - `int i=2+3;`
  - `double j=2+4.8;`
  - `float f=float(3.1)+float(2.0);`
- 对于上面的3个加法表达式，C++系统提供了类似于下面形式的运算符重载函数：
  - `int operator+(int,int);`
  - `double operator+(int,double);`
  - `float operator+(float,float);`



## 6.3.1 运算符重载的概念

- C++允许程序员通过重载扩展运算符的功能，使重载后的运算符能够对用户自定义的数据类型进行运算。

- 比如，设有复数类Complex，其形式如下：

```
class Complex{  
    double real,image;  
public:  
    .....  
};
```

- 假设定义了下面的复数对象，并且要实现两个复数相加的运算。

```
Complex c1,c2,c3;  
.....  
c1=c2+c3;
```



## 6.3.1 运算符重载的概念

- **what?**
  - 运算符重载，就是对已有的运算符重新进行定义，赋予其另一种功能，以适应不同的数据类型。
- **why?**
  - 使程序便于编写和阅读
  - 使程序定义类型与语言内建类型更一致
- **how?**
  - 使用特殊的成员函数
  - 使用自由函数，一般为友元



## 6.3.1 运算符重载的限制

- 可以重新定义大多数运算符，

+ - / \* % ^ & | ~ ! = < > +=

-= \*= /= %= ^= &= |= >>

>>= <<= == != <= >= []

() new new[] delete delete[]

- 不能重载某些特殊运算符，包括：. . \* :: ?: # sizeof typeid
- 只能被重载为类成员函数的运算符：= [] () ->
- 不能改变运算符的目、优先级、结合性
- 无隐含重载，即：定义了+，并不隐含定义+=
- 操作数中至少一个是自定义类型
- 程序定义的含义与运算符固有含义吻合
- 只能重载系统已有的运算符，不能创造新运算符



## 6.4 类运算符的重载

**C++为类提供了默认的重载运算符**

- ① 赋值运算符 (=) ;
- ② 取类对象地址的运算符 (&) ;
- ③ 成员访问运算符 (->) 。

**这些运算符不需要重载就可以使用，但要在类中使用其他运算符，就必须明确地重载它们。**



## 6.4.1 类成员运算符重载

### (1) 以非静态成员实现运算符重载

以类成员形式重载的运算符参数比实际参数少一个，第1个参数是以this指针隐式传递的。

```
class Complex{  
    double real, imag;  
public:  
    Complex operator+(Complex b){.....}  
    .....  
};
```



## 6.4.1 类成员运算符重载

### (2) 以友元函数实现运算符重载

如果将运算符函数作为类的友元重载，它需要的参数个数就与运算符实际需要的参数个数相同。

比如，若用友元函数重载Complex类的加法运算符，则形式如下：

```
class Complex{  
    .....  
    friend Complex operator+(Complex a,Complex b);  
    //声明  
    .....  
};
```

```
Complex operator+(Complex a, Complex b){.....}  
//定义
```





## 6.4.1.1 重载二元运算符

### 二元运算符的调用形式与解析

aa@bb 可解释成 aa.operator@(bb)  
或解释成 operator@(aa,bb)

第1种形式是@被重载为类的非静态成员函数的解释方式，这种方式要求运算符@左边的参数必须是一个对象，operator@是该对象的成员函数。

第2种形式是@作为类的友元或普通重载函数时的解释方式。



- 作为类的非静态成员函数的二元运算符，只能够有一个参数，这个参数是运算符右边的参数，它的第一个参数是通过this指针传递的，其重载形式如下：

```
class X{  
.....  
    T1 operator@(T2 b){ .....};  
}
```

其中，T1是运算符函数的返回类型，T2是参数的类型，原则上T1、T2可以是任何数据类型，但事实上它们常与X相同。



【例6-3】 有复数类Complex，利用运算符重载实现复数的加、减、乘、除等复数运算。

```
//CH6-3.cpp
#include<iostream>
using namespace std;
class Complex {
private:
    double r, i;
public:
    Complex (double R=0, double I=0):r(R), i(I){ };
    Complex operator+(Complex b);
    Complex operator-(Complex b);
    Complex operator*(Complex b);
    Complex operator/(Complex b);
    void display();
};
```



```
Complex Complex::operator +(Complex b){return Complex(r+b.r,i+b.i);}
```

```
Complex Complex::operator -(Complex b){return Complex(r-b.r,i-b.i);}
```

```
Complex Complex::operator *(Complex b){
```

```
    Complex t;
```

```
    t.r=r*b.r-i*b.i;
```

```
    t.i=r*b.i+i*b.r;
```

```
    return t;
```

```
}  
Complex Complex::operator /(Complex b) {
```

```
    Complex t;
```

```
    double x;
```

```
    x=1/(b.r*b.r+b.i*b.i);
```

```
    t.r=x*(r*b.r+i*b.i);
```

```
    t.i=x*(i*b.r-r*b.i);
```

```
    return t;
```

```
}
```

```
void Complex::display(){
```

```
    cout<<r;
```

```
    if (i>0) cout<<"+";
```

```
    if (i!=0) cout<<i<<"i"<<endl;
```

```
    }display();
```

```
};
```

```
int main(void) {
```

```
    Complex c1(1,2),c2(3,4),c3,c4,c5,c6;
```

```
    c3=c1+c2;
```

```
    c4=c1-c2;
```

```
    c5=c1*c2;
```

```
    c6=c1/c2;
```

```
    c1.display();
```

```
    c2.display();
```

```
    c3.display();
```

```
    c4.display();
```

```
    c5.display();
```

```
    c6.display();
```

```
}
```

• • • • • • • •



- **对于程序中的运算符调用：**

**c3=c1+c2;**

**c4=c1-c2;**

**c5=c1\*c2;**

**c6=c1/c2;**

- C++会将它们转换成下面形式的调用语句：

c3=c1.operator+(c2);

c4=c1.operator -(c2);

c5=c1.operator \*(c2);

c6=c1.operator /(c2);

**实际上，在程序中也可以直接写出这样的表达式，显式调用重载的运算符函数**



## 6.4.1.2 重载一元运算符

### 1、一元运算符

一元运算符只需要一个运算参数，如取地址运算符(&)、负数(-)、自增(++)、自减(--)等。

### 2、一元运算符常见调用形式为：

@a 或 a@

//隐式调用形式

a.operator@()

// 显式调用一元运算符@

- 其中的@代表一元运算符，a代表操作数。

@a代表前缀一元运算，如“++a”；

a@表示后缀运算，如“a++”。

### 3、@a将被C++解释为下面的形式之一

a.operator@()

operator@(a)



- 一元运算符作为类成员函数重载时不需要参数，其形式如下：

```
class X{  
.....  
    T operator@(){.....};  
}
```

- T是运算符@的返回类型。从形式上看，作为类成员函数重载的一元运算符没有参数，但实际上它包含了一个隐含参数，即调用对象的 **this** 指针。



**【例6-4】** 设计一个时间类Time，它能够完成秒钟的自增运算。

**//CH6-4.cpp**

**#include<iostream>**

**using namespace std;**

**class Time{**

**private:**

**int hour, minute, second;**

**public:**

**Time(int h, int m, int s);**

**Time operator++();**

**void display();**

**};**

**Time::Time(int h, int m, int s) {**

**hour=h;**

**minute=m;**

**second=s;**

**if(hour>=24) hour=0;**

**//若初始小时超过24, 重置为0**

**if(minute>=60) minute=0;**

**//若初始分钟超过60, 重置为0**

**if(second>=60) second=0;**

**//若初始秒钟超过60, 重置为0**

**}**





```
Time Time::operator ++(){
++second;
if(second>=60) {
    second=0;
    ++minute;
    if(minute>=60){ minute=0;    ++hour;
        if(hour>=24) hour=0;
    }
}
return *this;
}

void Time::display(){
    cout<<hour<<":"<<minute<<":"<<second<<endl; }

int main(){
    Time t1(23,59,59);
    t1.display();
    ++t1;                //隐式调用方式
    t1.display();
    t1.operator ++();    //显式调用方式
    t1.display();
}
```



- 为了实现类对象的各种运算，除了将运算符重载为类的成员函数外，还可以将它重载为类的友元函数。
- 在有些情况下，只有将运算符重载为类的友元才能解决某些问题。比如，对于例6-3的复数类而言，假设有下面的加法运算：

- **Complex a,b(2,3);**
- **a=b+2;** //正确 **b.operator(2)**
- **a=2+b;** //错误



## 6.4.2.1 作为友元函数重载二元运算符

**重载二元运算符为类的友元函数时需要两个参数，其形式如下：**

```
class X{
```

```
.....
```

```
    friend T1 operator@(T2 a,T3 b);
```

```
}
```

```
T1 operator@ (T2 a,T3 b){ .....}
```

- **T1、T2、T3代表不同的数据类型，事实上它们常与类X相同。**



**【例6-5】** 对于例6-3中的复数类Complex，利用友元运算符重载实现复数的加法，用成员函数重载减法运算

```
//CH6-5.cpp  
#include <iostream>  
using namespace std;  
class Complex {  
private:  
    double r, i;  
public:  
    Complex (double R=0, double I=0) : r(R), i(I){ };  
    friend Complex operator+(Complex a,Complex b); //实现复数+复数  
    friend Complex operator+(Complex a,double b); //实现复数+双精度数  
    friend Complex operator+(double a,complex b); //实现双精度数+复数  
    Complex operator-(Complex a); //实现复数-复数  
    Complex operator-(double a); //实现复数-双精度数  
    void display(){  
        cout<<r;  
        if (i>0) cout<<"+";  
        if (i!=0) cout<<i<<"i"<<endl;}  
};
```



---

```
Complex operator+(Complex a,Complex b){  
    Complex t; t.r=a.r+b.r;  t.i=a.i+b.i;  
    return t; }
```

```
Complex operator+(Complex a, double b){  
    Complex t; t.r=a.r+b;  t.i=a.i;  
    return t; }
```

```
Complex operator+(double a, Complex b){  
    Complex t; t.r=a+b.r;  t.i=b.i;  
    return t; }
```

```
Complex Complex::operator-(Complex a){  
    return Complex(r-a.r,i-a.i);}
```

```
Complex Complex::operator-(double a) {    return  
Complex(r-a,i);}      . . . . .
```



```
void main(void){
```

```
Complex c1(1,2),c2(3,4),c3,c4,c5,c6;
```

```
c3=c1+c2; c3.display();
```

```
c3=2+c2; c3.display();
```

```
c3=c2+2; c3.display();
```

```
c4=c1-c2; c4.display();
```

```
// c5=4-c1; c5.display(); //L1 错误
```

```
c5=c1-4; c5.display ();
```

```
}
```



## 2、说明：

- ① 对于不要求左值且可以交换参数次序的运算符（如+、-、\*、/等运算符），最好用非成员形式（包括友元和普通函数）的重载运算符函数实现。因为在用运算符计算表达式的值时，如果参数的类型与运算符需要的类型不匹配，这种重载方式会对参数进行隐式类型转换。
- ② 对于前面分析过的“ $2+c2$ ”和“ $c2+2$ ”之类的对称运算表达式，也可以直接通过友元运算符重载实现。



## 6.4.2.2 作为友元函数重载一元运算符

- 用友元函数重载一元运算符时需要一个参数。如在例6-4中，将++运算符重载为Time类的友元函数的情况如下。

【例6-6】 用友元重载Time类的自增运算符++。

```
//CH6-6.cpp
```

```
class Time{
```

```
..... //省略的代码与例6-4相同
```

```
friend Time operator++(Time &t);
```

```
};
```





```
Time operator ++(Time &t) {
    ++t.second;
    if(t.second>=60){
        t.second=0;
        ++t.minute;
        if(t.minute>=60){
            t.minute=0;
            ++t.hour;
            if(t.hour>=24) t.hour=0;
        }
    }
    return t;
}

int main(){
    Time t1(23,59,59);
    t1.display();
    ++ t1;                //隐式调用方式
    t1.display();
    operator++(t1);       //显式调用方式
    t1.display();
}
```



- 在用友元和普通函数重载++、--这类一元运算符函数时，如果用**值传递**的方式设置函数的参数，就可能会发生错误，不能把运算结果返回给调用对象。



# 成员函数还是友元函数

---

一般来说，对于双目运算符，最好将其重载为友元函数，因为这样更方便些；而对于单目运算符，则最好重载为成员函数。

- 双目运算符不能重载为友元函数的：赋值运算符=；
- 单目运算符只能重载为成员函数的：函数调用运算符()、下标运算符[]和指针->等；
- 只能重载为友元函数的：输出运算符<<，第一个操作数一定是ostream类型



## 6.5 特殊运算符重载

### 6.5.1 运算符++和--的重载

- 特殊性：区分前缀、后缀

**++X;**                               **//前自增**

**X++;**                               **//后自增**

**--X;**                               **//前自减**

**X--;**                               **//后自减**

- 将它们重载为类的成员函数时就会都是下面的形式：

```
class X{
```

```
.....
```

```
    X operator++(){.....};                               //前自增
```

```
    X operator++(){.....};                               //后自增
```

```
}
```



## 6.5.1 运算符++和--的重载

- 若将它们重载为友元运算符，将是下面的形式：

```
class X{
```

```
    friend X operator++(X& o); //前自增的友元声明
```

```
    friend X operator++(X& o); //后自增的友元声明
```

```
}
```

- 问题：

无法区分到底是前自增还是后自增运算！同样的问题  
发生在自减运算符身上：--



- **C++编译器可以通过在运算符函数参数表中是否插入关键字int 来区分这两种方式**

- **前缀**

**operator -- ();**

**operator -- (X & x);**

- **后缀**

**operator -- (int);**

**operator -- (X & x, int);**



## ● 例题x.cpp

```
class X
```

```
{
```

```
public:
```

```
    void operator++ (int) { cout << "a" << endl; };
```

```
    void operator++() { cout << "b" << endl; };
```

```
};
```

```
int main ()
```

```
{
```

```
    X obj;
```

```
    ++obj;
```

```
    obj++;
```

```
}
```

b
a



**【例6-7】** 设计一个计数器counter，用类成员重载自增运算符实现计数器的自增，用友元重载实现计数器的自减。

```
//Eg6-7.cpp  
#include<iostream>  
using namespace std;  
class Counter{  
private:  
    int n;  
public:  
    Counter(int i=0){n=i;}  
    Counter operator++();  
    Counter operator++(int);  
    friend Counter operator--(Counter &c);  
    friend Counter operator--(Counter &c,int);  
    void display();  
};
```





```
Counter Counter::operator++(){
    ++n;
    return *this;
}
Counter Counter::operator++(int){
    n++;
    return *this;
}
Counter operator--(Counter &c){
    --c.n;
    return c;
}
Counter operator--(Counter &c,int){
    c.n--;
    return c;
}
void Counter::display(){
    cout<<"counter number = "<<n<<endl;
}
```

---



```
int main(){  
    Counter a;  
    ++a;           //调用Counter::operator++()  
    a.display();  
    a++;           //调用Counter::operator++(int)  
    a.display();  
    --a;           //调用operator--(Counter &c)  
    a.display();  
    a--;           //调用operator--(Counter &c,int)  
    a.display();  
}
```

程序运行结果如下：

```
counter number = 1  
counter number = 2  
counter number = 1  
counter number = 0
```



# 优先使用前缀操作符

---

在实现中，后缀操作会先构造一个临时对象，并将原对象保存，然后完成自增操作，最后将保存对象原值的临时对象返回。代码如下：

```
ClassName & ClassName::operator++()
```

```
{
```

```
    ClassAdd(1);
```

```
    return *this;
```

```
}
```

```
ClassName &
```

```
ClassName::operator++(int)
```

```
{
```

```
    ClassName temp(*this);
```

```
    ClassAdd(1);
```

```
    return temp;
```

```
}  
  
.  
.  
.  
.  
.  
.  
.  
.
```



## 6.5.2 重载赋值运算符=

### 1、赋值运算符“=”的重载特殊性

- 赋值运算进行时将调用此运算符
- 只能用成员函数重载
- 如果需要而没有定义时，编译器自动生成，该版本进行bit-by-bit复制
- 编译器生成的版本与copy constructor有类似问题

### ● 例题：默认的赋值运算符



## 【例6-8】 默认赋值运算符引起的指针悬挂问题。

//CH6-8.cpp

```
#include <iostream>
```

```
#include <string>
```

```
using namespace std;
```

```
class String{
```

```
    char *ptr;
```

```
    int n;
```

```
public:
```

```
    String(char * s,int a){
```

```
        ptr=new char[strlen(s)+1];
```

```
        strcpy(ptr,s);
```

```
        n=a;
```

```
    }
```

```
    ~String(){delete [ ]ptr;}
```

```
    void print(){cout<<ptr<<endl;}
```

```
};
```

. . . . .

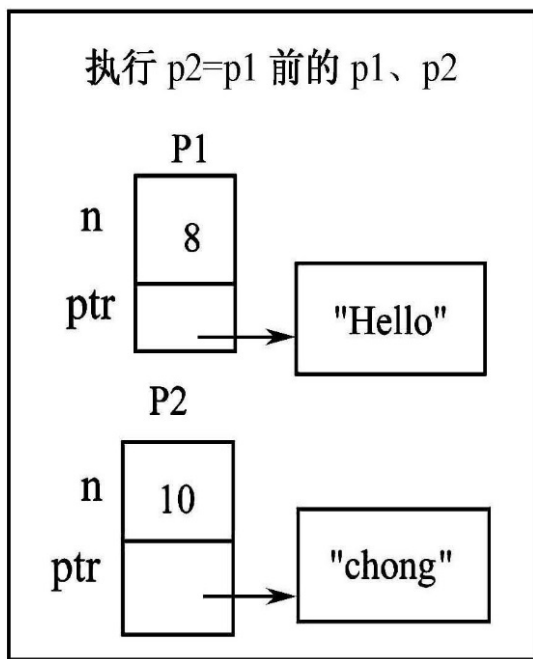


```
int main(){
    string p1("Hello",8);

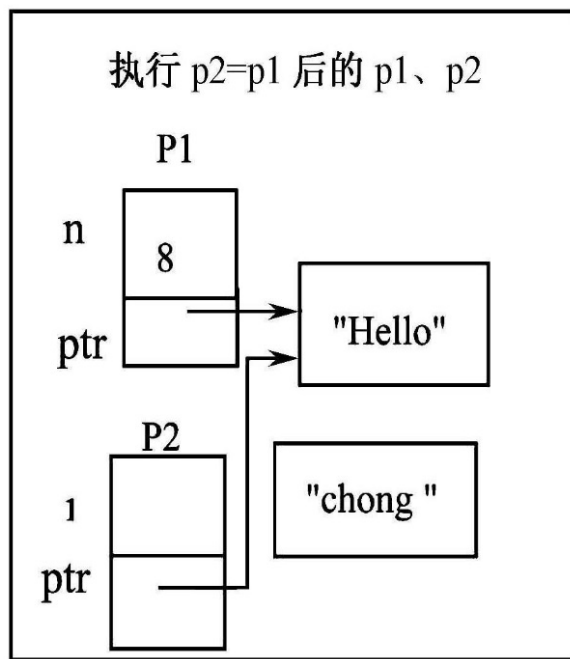
    { string p2("chong qing",10);
        p2=p1;
        cout<<"p2:";
        p2.print();
    }
    cout<<"p1:";
    p1.print();
}
```

本程序在将  
引发指针悬  
挂问题！

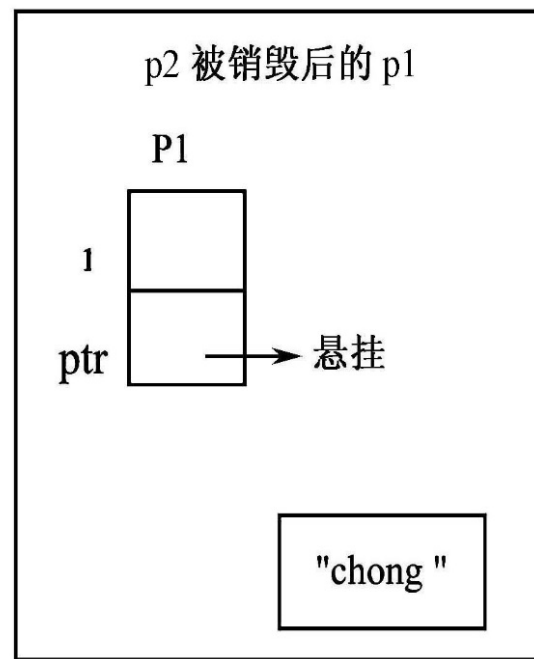
## ● 产生指针悬挂的原因分析



(a)



(b)



(c)



- **【例6-9】 重载类string的赋值运算符，解决赋值操作引起的指针悬挂问题**

```
class string{
    char *ptr;
public:
    string(char * s)
    { ptr=new char[strlen(s)+1]; strcpy(ptr,s); }
    ~string() { delete ptr; }
    string & operator=(const string & );
    void print() { cout<<ptr<<endl; }
};
string & string::operator=(const string &s )
{
    if(this==&s) return *this;
    delete ptr; ptr=new char[strlen(s.ptr)+1];
    strcpy(ptr,s.ptr); return *this;
}
```





# ！ 不要让编译器帮你重载赋值运算符！

---

在设计类似String这样的含有**指针**数据成员的类时，如果需要进行类对象之间的赋值操作，就应该重载该类的赋值运算符。

**赋值运算符是一个二元运算符，常返回本类对象的引用。形式如下：**

```
class X
{
    X& operator= (const X &source);
};
```



## ！ 赋值运算符重载需返回\*this的引用！

---

在C++中，内置数据类型是支持链式赋值的，即a=b=c。所以，对于用户自定义数据类型也要支持链式赋值。

```
class CStrting { };
```

```
CString str1("Hello C++");
```



```
CString str2, str3;
```

```
str3 = str2 = str1;
```

```
str3.operator=(str2.operator=(str1));
```

```
Cstring  Cstring: : operator=(const CString& str);
```

```
Cstring& Cstring: : operator=(const CString& str);
```



# 是对象还是引用？

- 效率
- 编译器不允许使用临时对象调用成员函数

```
class CSrting { };  
CString str1("Hello C++");  
CString str2( "Hello World" );  
CString str3;  
(str3 = str1) = str2;
```

为了能够灵活地使用赋值运算符，  
选择返回引用绝对是明智之举



# ！ 赋值运算符运算是不能被继承的！

---

- 编译器会自动生成派生类的赋值运算符重载函数，将基类中的operator=函数隐藏了。
- 不要忘记对基类成员变量进行赋值

```
class CString {private : pChar};
```

```
Class ColorString : public CString {private : m_dColor};
```

```
ColorString & ColorString::operator=(const ColorString& rhs)
```

```
{
```

```
if(this == &rhs) return *this;
```

```
m_dColor = rhs.m_dColor;
```

```
return *this;
```

```
}
```



```
CString::operator=(rhs);
```

---

• • • • •



# ！一定要检查自赋值！

---

**自赋值 (self assignment) 在C++语法中是允许的**

- 特别是加入引用后，自赋值看似更加合理，更加隐蔽
- `strA=strB`，当`strA`是`strB`的引用时

## 为什么检查自赋值？

- **效率**，特别是在具有内存分配的操作。如果可以在赋值运算符函数体的首部检测到是给自己赋值，就可以立即返回，从而可以节省大量的工作，否则必须去实现整个赋值操作。
- **保证正确性**，一个赋值运算符必须首先释放掉一个对象的资源（去掉旧值），然后根据新值分配新的资源。在自己给自己赋值的情况下，释放旧的资源将是灾难性的，因为在分配新的资源时会需要旧的资源。



```
CString& CString : : operator=(const CString& str)
{
    if(*this == str)
        return *this;
    if(pChar!=NULL)
        delete [] pChar;
    pChar = new char[strlen(str.pChar)+1];
    strcpy(pChar, str.pChar);
    return *this;
}
```

---

• • • • •



- **黄金定律：**如果需要给类的数据成员动态分配空间则必须实现赋值运算符。此时，赋值运算符的重载形式可以表示为以下形式：

```
ClassName& ClassName: : operator=(const  
ClassName& rhs)
```

```
{
```

自赋值检查

释放原有空间 & 申请新空间 & 数据复制

返回 \*this

```
}
```

- 如果是派生类，请不要忘记对基类成员变量的赋值进行处理。



# Copy constructor 和 operator= 的区别?

- 当有新对象生成时， copy constructor发生作用

已有对象间赋值时， operator=发生作用

X a;

X b(a);或 X b=a; //调用copy constructor

b = a; //调用operator=

PS: 如果赋值运算符重载的**参数不是引用**， 则  
先调用拷贝构造函数， 再调用赋值运算符重载





# 成对实现的运算符

---

- ==与!=    >与<    >=与<=
- +与+=    -与-=    \*与\*=    /与\*/

Complex c1, c2;

if(c1==c2) ...

if(!(c1==c2)) ... 没有找到接受 “Complex” 类型的左操作数的运算符

A op= B (op可能是+、-、\*、/等)

A op= B和A = A op B具有相同的含义  
符合习惯，减少代码重复、提高效率



## 6.5.3 重载[ ]

Vector.cpp

1、[ ]是一个二元运算符，其重载形式如下：

```
class X{  
.....  
    X& operator[](int n);  
};
```

2、重载[ ]需要注意的问题

- ① [ ]是一个二元运算符，其第1个参数是通过对象的this指针传递的，第2个参数代表数组的下标
- ② 由于[ ]既可以出现在赋值符“=”的左边，也可以出现在赋值符“=”的右边，所以重载运算符[ ]时常返回引用。
- ③ [ ]只能被重载为类的非静态成员函数，不能被重载为友元和普通函数。



**【例6-10】** 设计一个工资管理类，它可以根据职工的姓名录入和查询职工的工资，每个职工的基本数据有职工姓名和工资。

//CH6-10.cpp

#include <iostream>

#include <string>

using namespace std;

struct Person{

//职工基本信息的结构

double salary;

char \*name;

};

class SalaryManage{

Person \*employ;

//存放职工信息的数组

int max;

//数组下标上界

int n;

//数组中的实际职工人数

public:

SalaryManage(int Max=0){

max=Max;

n=0;

employ=new Person[max];

}





```
double &operator[](char *Name) {           //重载[], 返回引用
    Person *p;
    for(p=employ;p<employ+n;p++)
        if(strcmp(p->name,Name)==0)
            return p->salary;
    p=employ + n++;
    p->name=new char[strlen(Name)+1];
    strcpy(p->name,Name);
    p->salary=0;
    return p->salary;
}

void display(){
    for(int i=0;i<n;i++)
        cout<<employ[i].name<<"
"<<employ[i].salary<<endl;
}

};
```



```
int main(){
    SalaryManaege s(3);
    s["杜一为"]=2188.88;
    s["李海山"]=1230.07;
    s["张军民"]=3200.97;
    cout<<"杜一为\t"<<s["杜一为"]<<endl;
        cout<<"李海山\t"<<s["李海山"]<<endl;
    cout<<"张军民\t"<<s["张军民"]<<endl;

    cout<<"-----下为display的输出-----\n\n";
    s.display();
}
```



# \*增补 重载( )

1、运算符( )是函数调用运算符，也能被重载。  
且只能被重载为类的成员函数。

2、运算符( )的重载形式如下：

```
class X{
```

```
.....
```

```
    X& operator( )(参数表);
```

```
};
```

其中的参数表可以包括任意多个参数。

3、运算符( )的调用形式如下：

```
X Obj;
```

```
//对象定义
```

```
Obj.operator()(参数表);
```

```
//调用形式1
```

```
Obj(参数表);
```

```
//调用形式2
```



**【例CH6-11】** 设计一个时间类Time，重载函数调用运算符( )，使它能够通过调整时 (hh)、分 (mm)、秒 (ss) 的数据。

```
#include <iostream>
using namespace std;
class Time{
private:
    int hh,mm,ss;
public:
    Time(int h=0,int m=0,int s=0):hh(h),mm(m),ss(s){}
    void operator()(int h,int m,int s) {
        hh=h;
        mm=m;
        ss=s;
    }
}
```



```
void ShowTime(){
    cout<<hh<<":"<<mm<<":"<<ss<<endl;
}

};

int main(){
    Time t1(12,10,11);
    t1.ShowTime();
    t1.operator()(23,20,34);
    t1.ShowTime();
    t1(10,10,10);
    t1.ShowTime();
}
```





## 6.6 输入/输出运算符重载

### 6.6.1 重载输出运算符<<

输出运算符<<也称为插入运算符，通过输出运算符<<的重载可以实现用户自定义数据类型的输出。

- 重载运算符<<的常见格式如下：

```
ostream &operator<<(ostream &os,classType  
    object) {  
    .....  
    os<< ...  
    //输出对象的实际成员数据  
    return os;  
    //返回ostream对象  
}
```



## 6.6.2 重载输入运算符>>

输入运算符>>也称为提取运算符，用于输入数据。通过输入运算符>>的重载，就能够用它输入用户自定义的数据类型

- 其重载形式如下：

```
istream &operator>>(istream &is,class_name  
    &object) {  
    .....  
    is>> ...  
    //输入对象object的实际成员数据  
    return is;    //返回istream对象  
}
```



## 6.6.3 重载运算符<<和>>举例

【例6-13】 有一销售人员类Sales，其数据成员有姓名name，身份证号id，年龄age。重载输入/输出运算符实现对Sales类数据成员的输入和输出。

```
//CH6-13.cpp
#include<iostream>
#include<string>
class Sales{
private:
    char name[10];
    char id[18];
    int age;
public:
    Sales(char *Name,char *ID,int Age);
    friend    Sales &operator<<(ostream &os,Sales &s);    //重载输出运算符
    friend    Sales &operator>>(istream &is,Sales &s);    //重载输入运算符
};
Sales::Sales(char *Name,char *ID,int Age) {
    strcpy(name,Name);
    strcpy(id,ID);
    age=Age;
}
```



```
Sales& operator<<(ostream &os,Sales &s) {  
    os<<s.name<<"\t";  
    os<<s.id<<"\t";  
    os<<s.age<<endl;  
    return s;  
}  
Sales &operator>>(istream &is,Sales &s) {  
    cout<<"输入雇员的姓名, 身份证号, 年龄"<<endl;  
    is>>s.name>>s.id>>s.age;  
    return s;  
}  
int main(){  
    Sales s1("黄梅","214198012111711",40);  
    cout<<s1;  
    cout<<endl;  
    cin>>s1;  
    cout<<s1;  
}
```

//输出姓名

//输出身份证号

//输出年龄

//L1

//L2

//L3

//L4

//L5



## 6.7 虚函数

### 6.7.1 虚函数的意义

#### 1、回顾：基类与派生类的**赋值相容**

- 派生类对象可以赋值给基类对象。
- 派生类对象的地址可以赋值给指向基类对象的指针。
- 派生类对象可以作为基类对象的引用。

#### ● 赋值相容的问题：

- 不论哪种赋值方式，都只能通过基类对象（或基类对象的指针或引用）访问到派生类对象从基类中继承到的成员，不能借此访问派生类定义的成员。

#### 2、虚函数使得通过基类对象的指针或引用访问派生类定义的成员可以施行。



## 6.7.1 虚函数的意义

**【例6-14】** 某公司有经理、销售员、小时工等多类人员。经理按周计算薪金；销售员每月底薪800元，然后加销售提成，每销售1件产品提取销售利润的5%；小时工按小时计算薪金。每类人员都有姓名和身份证号等数据。为简化问题，把各类人员的共有信息抽象成基类Employee，其他人员则继承该类的功能。



```
//CH6-14.cpp
#include <iostream>
#include <string>
using namespace std;
class Employee{
public:
    Employee(string Name ,string id){ name=Name; Id=id; }
    string getName(){ return name; }                //返回姓名
    string getID(){ return Id; }                    //返回身份证号
    float getSalary(){ return 0.0; }                //返回薪水
    void print() {                                   //输出姓名和身份证号
        cout<<"姓名: "<<name<<"\t\t 编号: "<<Id<<endl;
    }
private:
    string name;
    string Id;
};
```



```
class Manager:public Employee{
public:
    Manager(string Name,string id,float s=0.0):Employee(Name,id){
        WeeklySalary=s;
    }
    void setSalary(float s) { WeeklySalary=s; }    //设置经理的周薪
    float getSalary(){ return WeeklySalary; }      //获取经理的周薪
    void print(){                                  //打印经理姓名、身份证、周薪
        cout<<"经理: "<<getName()<<"\t\t 编号: "<<getID()
            <<"\t\t 周工资: "<<getSalary()<<endl;
    }
private:
    float WeeklySalary;                            //周薪
};

int main(){
    Employee e("黄春秀","NO0009"),*pM;
    Manager m("刘大海","NO0001",128);
    m.print();
    pM=&m;
    pM->print();
    Employee &rM=m;
    rM.print();
}
```





## 6.7.1 虚函数的意义

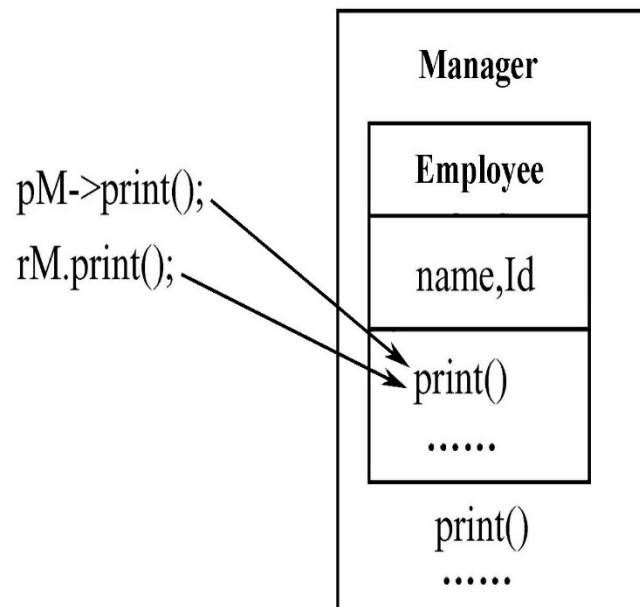
- 例6-14程序的运行结果如下：

经理：刘大海          编号：NO0001          周工资：128

姓名：刘大海          编号：NO0001

姓名：刘大海          编号：NO0001

- 输出的第2、3行表明，通过基类对象的指针和引用只访问到了在基类中定义的print函数。





- 将基类Employee的print指定为虚函数，如下形式：

```
class Employee{
```

```
.....
```

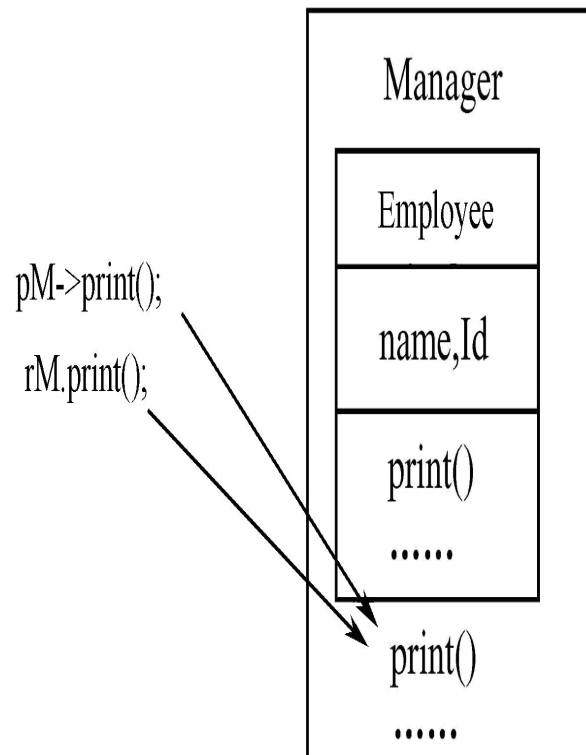
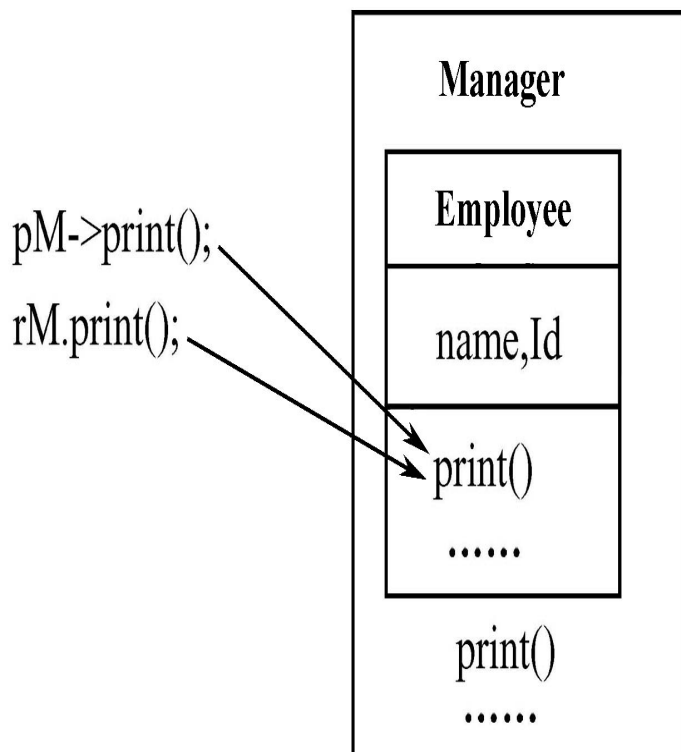
```
virtual void print(){ cout<<"姓名: "<<name<<"\t\t 编号: "<<Id<<endl; }  
};
```

- 将得到下面的程序运行结果：

经理：刘大海	编号：NO0001	周工资：128
经理：刘大海	编号：NO0001	周工资：128
经理：刘大海	编号：NO0001	周工资：128



- **基类指针或引用指向派生类对象时，虚函数与非虚函数的调用方式，图左为非虚函数，图右为虚函数**





## 6.7.1 虚函数的意义

### 1、什么是虚函数

- 用virtual关键字修饰的成员函数
- virtual关键字其实质是告知编译系统，被指定为virtual的函数采用动态联编的形式编译。

### 2、虚函数的定义形式

```
class x{  
.....  
Virtual rtype f(参数表);  
.....  
}
```



- **虚函数的虚特征：基类指针指向派生类的对象时，通过该指针访问其虚函数时将调用派生类的版本；**

— 例题没有虚函数的情况

```
class B
{public: void f ( ) {cout << "B::f";}; };
class D : public B
{public: void f ( ) { cout << "D::f"; };;};
```

```
int main()
{
    D d;
    B * pb = & d;
    pb->f( );
}
```

B::f



## ● 例题：虚函数版

```
class B
{public: virtual void f ( ) {cout << "B::f";}; };
class D : public B
{public: void f ( ) { cout << "D::f"; };;};
```

```
int main()
{
    D d;      B * pb = & d;
    pb->f( );
}
```

D::f

- 总结：通过指向派生类对象的基类指针访问函数成员时，
  - 非虚函数由指针类型决定调用的版本
  - 虚函数由指针指向的实际对象决定调用的版本



## 6.7.2 虚函数的特性

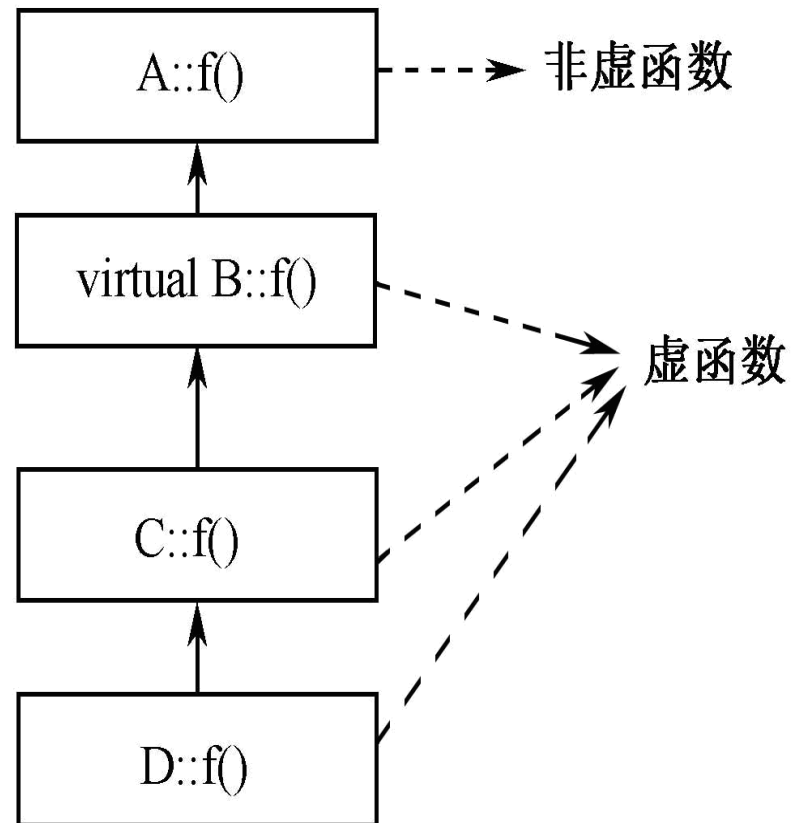
① 一旦将某个成员函数声明为虚函数后，它在继承体系中就永远为虚函数了

【例6-15】 虚函数与派生类的关系。

```
#include <iostream>
#include<string>
using namespace std;
class A {
public:
    void f(int i){cout<<"...A"<<endl;};
};
class B: public A {
public:
    virtual void f(int i){cout<<"...B"<<endl;}
};
class C: public B {
public:
    void f(int i){cout<<"...C"<<endl;}
};
```



```
class D: public C{
public:
    void f (int){cout<<"...D"<<endl;
};
int main(){
    A *pA,a;
    B *pB, b;   C c;   D d;
    pA=&a;   pA->f(1);
    pA=&b;   pA->f(1);
    pA=&c;   pA->f(1);
    pA=&d;   pA->f(1);
}
```







② 如果基类定义了虚函数，当通过基类指针或引用调用派生类对象时，将访问到它们实际所指对象中的虚函数版本。

- 例如，若把例6-15中的main的pA指针修改为pB，将会体现虚函数的特征。

```
int main(){
    A *pA,a;
    B *pB, b; C c; D d;
    // pB=&a; pB->f(1);           //错误，派生类不能访问基类对象
    pB=&b; pB->f(1);               //调用B::f
    pB=&c; pB->f(1);               //调用C::f
    pB=&d; pB->f(1);               //调用D::f
}
```



**③ 只有通过基类对象的指针和引用访问派生类对象的虚函数时，才能体现虚函数的特性。**

**【例6-16】 只能通过基类对象的指针和引用才能实现虚函数的特性。**

**//CH6-16.cpp**

**#include <iostream>**

**using namespace std;**

**class B{**

**public:**

**virtual void f(){ cout << "B::f"<<endl; };**

**};**

**class D : public B{**

**public:**

**void f(){ cout << "D::f"<<endl; };**

**};**



```
int main(){  
    D d;  
    B *pB = &d, &rB=d, b;  
    b=d;  
    b.f();  
    pB->f();  
    rB.f();  
}
```

**本程序的运行结果如下：** 第1行输出没有体现虚函数特征

**B::f**

**D::f**

**D::f**



#### ④ 派生类通过从基类继承的成员函数调用虚函数时，将访问到派生类中的版本。

**【例6-18】** 派生类D的对象通过基类B的普通函数f调用派生类D中的虚函数g

//CH6-18.cpp

```
#include <iostream>
using namespace std;
class B{
public:
    void f(){ g(); }
    virtual void g(){ cout << "B::g"; }
};
class D : public B{
public:
    void g(){ cout << "D::g"; }
};
int main(){
    D d;
    d.f();
}
```



## ● 例题6-19.cpp

```
class B
```

```
{public:
```

```
    void f ( ) { cout << "bf "; };
```

```
    virtual void vf ( ) { cout << "bvf "; };
```

```
    void ff ( ) { vf(); f(); };
```

```
    virtual void vff ( ) { vf(); f(); }; };
```

```
class D: public B
```

```
{public:
```

```
    void f ( ) { cout << "df "; };
```

```
    void ff ( ) { f(); vf(); };
```

```
    void vf ( ) { cout << "dvf "; }; }
```

```
int main()
```

```
{    D d;    B * pB = &d;
```

```
    pB->f();        pB->ff();
```

```
    pB->vf();        pB->vff();
```

```
}
```

```
bf dvf bf dvf dvf bf
```

• • • • • • • •



⑤ 派生类中的虚函数要保持其虚特征，必须与基类虚函数的函数原型完全相同，否则就是普通的重载函数，与基类的虚函数无关。

**【例6-17】** 基类B和派生类D都具有成员函数f，但它们的参数类型不同，因此不能体现函数f在派生类D中的虚函数特性。

**//CH6-17.cpp**

**#include <iostream>**

**using namespace std;**

**class B{**

**public:**

**virtual void f(int i){ cout << "B::f"<<endl; };**

**};**



```
class D : public B{
public:
    int f(char c){ cout << "D::f..."<<c<<endl; }
};
int main(){
    D d;
    B *pB = &d, &rB=d, b;
    pB->f('1');
    rB.f('1');
}
```

本程序的运行结果如下：

B::f

B::f



- ⑥只有类的非静态成员函数才能被定义为虚函数，类的构造函数和静态成员函数不能定义为虚函数。原因是虚函数在继承层次结构中才能够发生作用，而构造函数、静态成员是不能够被继承的。
- ⑦内联函数也不能是虚函数。因为内联函数采用的是静态联编的方式，而虚函数是在程序运行时才与具体函数动态绑定的，采用的是动态联编的方式，即使虚函数在类体内被定义，C++编译器也将它视为非内联函数。



- 例：对于下面的类，解释每个函数

```
class base {  
    string name() {return basename;}  
    virtual void print(ostream &os) {os<<basename;}  
private:  
    string basename;  
};  
base::base(string  
name):basename(name){}  
class derived { public base {  
    void print(ostream &os){base::print(os); os<<" "<<men;}  
private:  
    int men;  
};  
derived::derived(string name,int  
val):base(name),men(val){}
```

如果该代码有问题，如何修正？

给定上题中的类和如下对象，确定在运行时调用哪个函数：

```
base bobj;      base *bp1=&bobj;  base &br1=bobj;  
derived dobj;  base *bp2=&dobj;  base &br2=dobj;  
(1)bobj.print();    (2)dobj.print();  (3)bp1->name();  
(4)bp2->name();  (5)br1.print();    (6)br2.print();
```

(1)调用**base**类中定义的**print**函数：因为**bobj**是**base**类对象。

(2)调用**derived**类中定义的**print**函数：因为**dobj**是**derived**类对象。

(3)调用**base**类中定义的**name**函数：因为**name**是非虚函数，且**bp1**的类型为**base\***。

(4)调用**base**类中定义的**name**函数：因为**name**是非虚函数，且**bp2**的类型为**base\***。

(5)调用**base**类中定义的**print**函数：因为通过引用调用虚函数实行动态绑定，且**br1**引用的实际对象为**base**类对象。

(6)调用**derived**类中定义的**print**函数：因为通过引用调用虚函数实行动态绑定，且**br2**引用的实际对象为**derived**类对象。



## 6.7.3 虚析构造函数

- **基类析构造函数几乎总是为虚析构造函数。假定使用delete和一个指向派生类的基类指针来销毁派生类对象，如果基类析构造函数不为虚，就如一个普通成员函数，delete函数调用的就是基类析构造函数。在通过基类对象的引用或指针调用派生类对象时，将致使对象析构不彻底！**



【例6-20】 在非虚析构函数的情况下，通过基类指针对派生对象的析构是不彻底的。

```
#include <iostream>
using namespace std;
class A{
public:
    ~A(){ cout<<"call A::~~A()"<<endl; }
};
class B:public A{
    char *buf;
public:
    B(int i){buf=new char[i];}
    ~B(){
        delete [] buf;
        cout<<"call B::~~()"<<endl;
    }
};
int main(){
    A* a=new B(10);
    delete a;
}
```

程序运行结果：

**call A::~~A()**

此结果表明没有析  
构buf



```
class A{
public:
    virtual ~A(){
        cout<<"call A::~~A()"<<endl;    };
class B:public A{
    char *buf;
public:
    B(int i){buf=new char[i];}
    virtual ~B(){
        delete [] buf;
        cout<<"call B::~~()"<<endl;    };

int main(){
    A* a=new B(10);
    delete a;
}
```

程序运行结果:

**call B::~~()**

**call A::~~A()**

此结果表明回收了  
buf空间!



- 例：如果这个类定义有错，可能是什么错？

```
class AbstractObject {
```

```
Public:
```

```
    virtual void doit();
```

```
    //other members not including any of the  
    copy-control functions
```

```
};
```

- 假定有两个基类Base1和Base2，其中每一个定义了一个名为print的虚成员和一个虚析构函数。从这些基类派生下面的类，其中每一个类都重定义了print函数：

```
class D1:public Base1 {...}
```

```
class D2:public Base2 {...}
```

```
class Ml:public D1, public D2 {...}
```

使用下面的指针确定在每个调用中使用哪个函数：

```
Base1 *pb1=new Ml;   Base2 *pb2=new Ml;
```

```
D1 *pd1=new Ml;      D2 *pd2=new Ml;
```

```
(1)pb1->print(); (2)pd1->print; (3)pd2->print();
```

```
(4)delete pb2; (5)delete pd1; (6)delete pd2;
```



## 6.8 虚函数的实现技术

### 1、虚函数的实现

#### — 早期绑定与后期绑定

- early-binding: 编译时间完成的绑定
- late-binding: 运行时间完成的绑定
- 虚函数是通过后期绑定实现的

#### — 编译时间行为和运行时间行为

- compiling-time vs. running-time
- 编译时间行为: 通过静态分析就可以确定的行为
- 运行时间行为: 通过动态分析 (运行过程的分析)
- 虚函数是通过动态行为分析实现的运行时间行为





## 2、C++实现虚函数的技术

- **C++通过虚函数表来实现虚函数的动态绑定。**  
在编译带有虚函数的类时，C++将为该类建立一个虚函数表（vtable），在虚函数表中存放指向本类虚函数的指针，这些指针指向本类的虚函数地址。
- 在有虚函数的类对象中，C++除了为它保存每个数据成员外，还保存了一个指向本类虚函数表的指针（vptr）。



**class X**

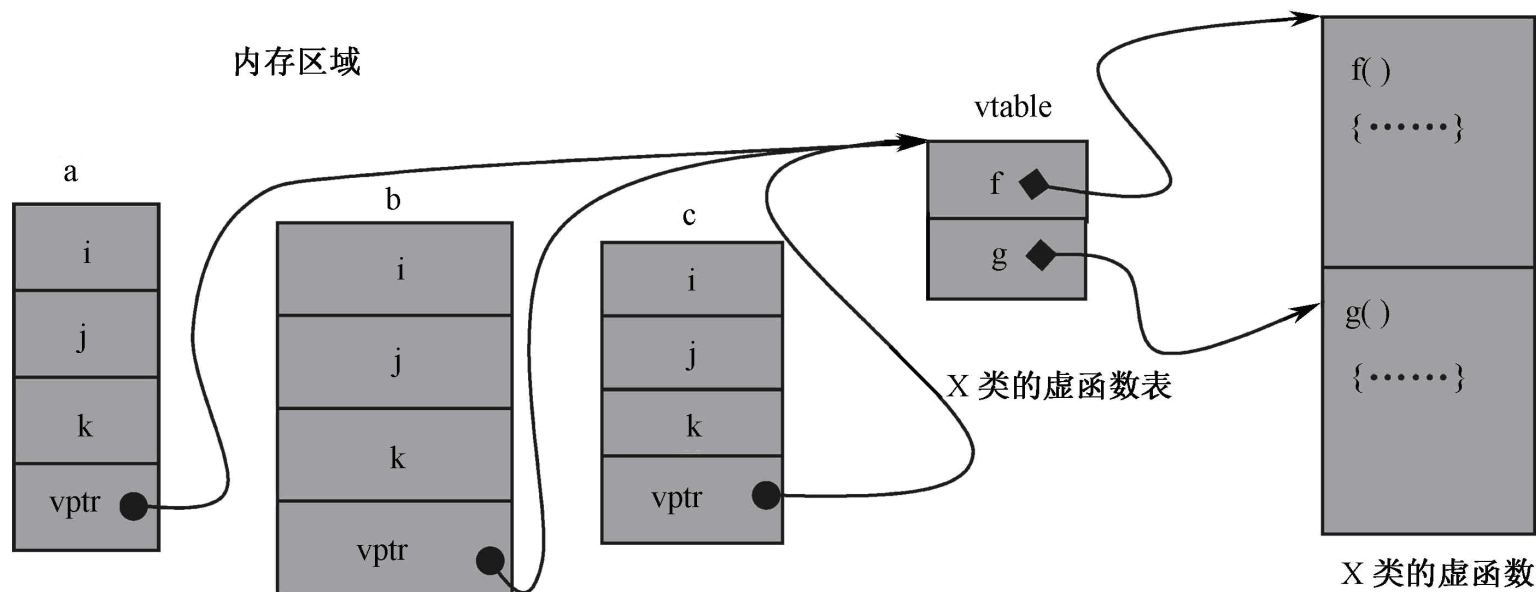
**{ int i, j, k;**

**public:**

**virtual f( ) {...};**

**virtual g( ) {...}; };**

**X a, b, c;**





**class X**

**{ int i, j, k;**

**public:**

**virtual f ( ) {...}**

**virtual g ( ) {...} };**

**class D: public X**

**{ int j, m;**

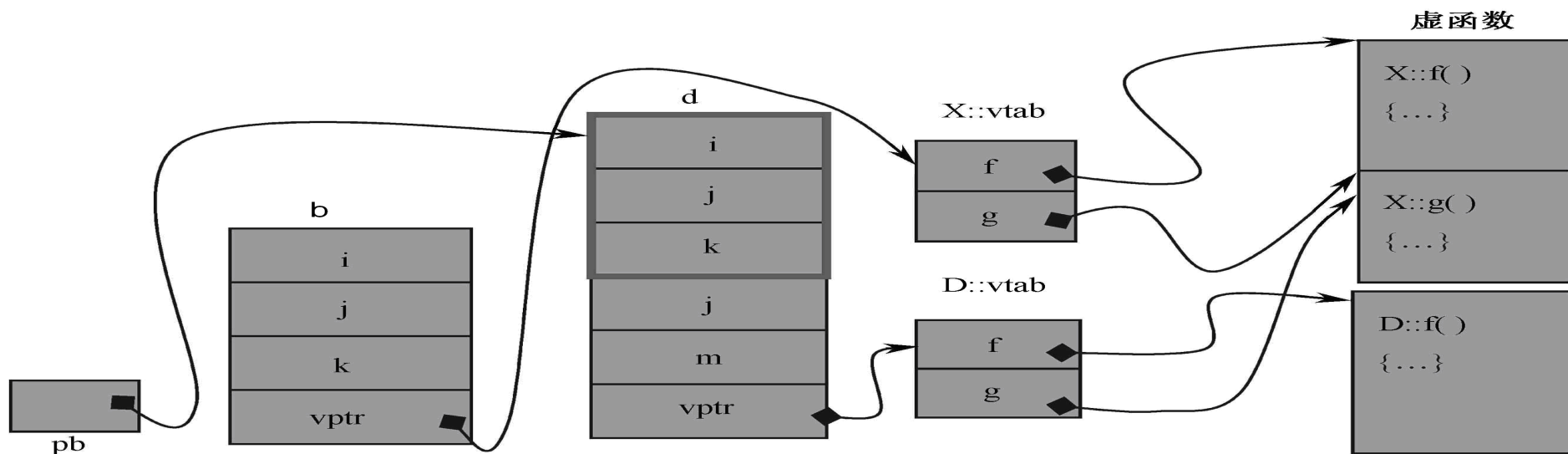
**public:**

**void f( ){ }**

**void main()**

**{ D d; X b, \*pb = &d;**

**pb->f( ); pb->g( ); }**





## 6.9 纯虚函数和抽象类

### 6.9.1、纯虚函数的概念

- 仅定义函数原型而不定义其实现的虚函数
- 在声明时被初始化为0的虚类成员函数。

— How?

```
class X
```

```
{
```

```
    virtual ret_type func_name (param) = 0;
```

```
}
```



## 6.9.2、抽象类的概念

- What is an abstract class?
  - **包含一个或多个纯虚函数的类**
- Using abstract class
  - **不能实例化抽象类**
  - **但是可以定义抽象类的指针和引用**
- Converting abstract class to concrete class
  - **定义一个抽象类的派生类**
  - **定义所有纯虚函数**



- **C++对抽象类具有以下限定：**
  - **抽象类中含有纯虚函数，由于纯虚函数没有实现代码，所以不能建立抽象类的对象。**
  - **抽象类只能作为其他类的基类，可以通过抽象类对象的指针或引用访问到它的派生类对象，实现运行时的多态性。**
  - **如果派生类只是简单地继承了抽象类的纯虚函数，而没有重新定义基类的纯虚函数，则派生类也是一个抽象类。**



## 【例6-21】 抽象图形类及其应用。

//CH6-21.cpp

#include <iostream>

using namespace std;

class Figure{

protected:

double x,y;

public:

void set(double i,double j){ x=i; y=j; }

virtual void area()=0;

//纯虚函数

};

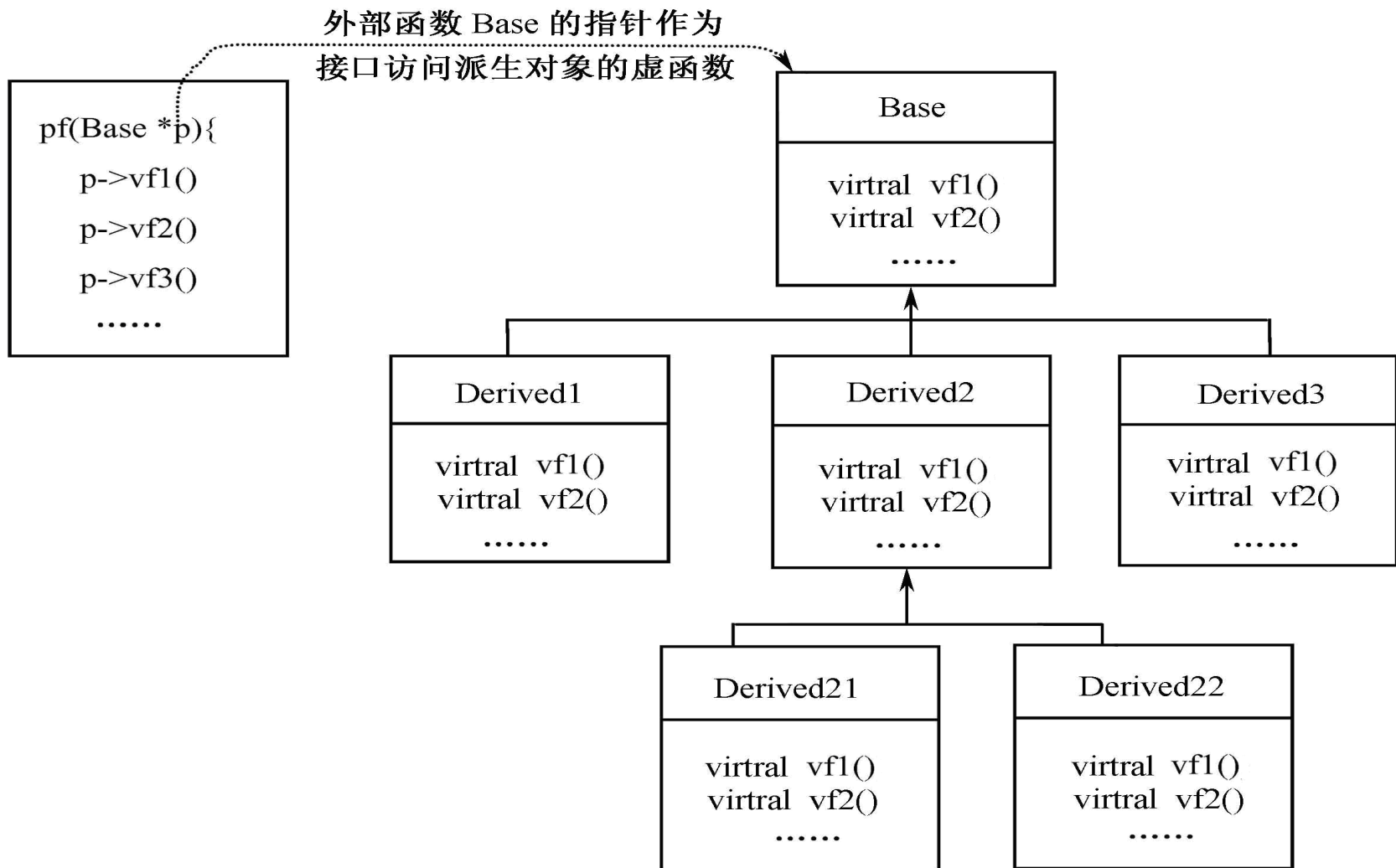


```
class Triangle:public Figure{
public:
    void area(){cout<<"三角形面积: "<<x*y*0.5<<endl;}//重写基类纯虚函数
};
class Rectangle:public Figure{
public:
    void area(int i){cout<<"这是矩形, 它的面积是: "<<x*y<<endl;}
    //重载了基类的area, 并没有重写基类的纯虚函数, 所以该类仍为抽象类
};
int main(){
    Figure *pF;
    // Figure f1;           //L1, 错误
    // Rectangle r;         //L2, 错误
    Triangle t;            //L3
    t.set(10,20);
    pF=&t;
    pF->area();             //L4
    Figure &rF=t;
    rF.set(20,20);
    rF.area();             //L5
}
```





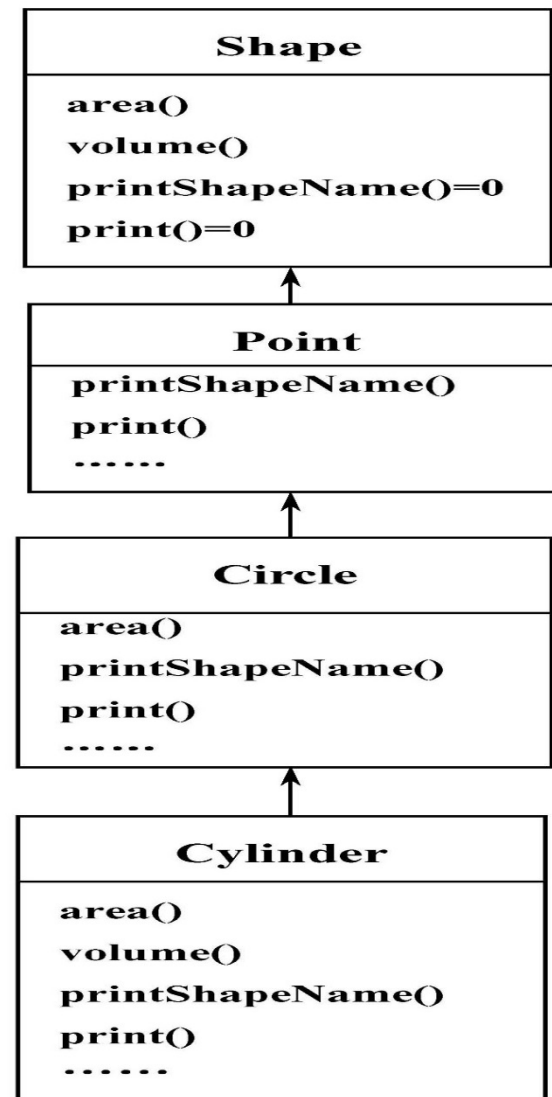
## ● 抽象类的主要用途——作接口



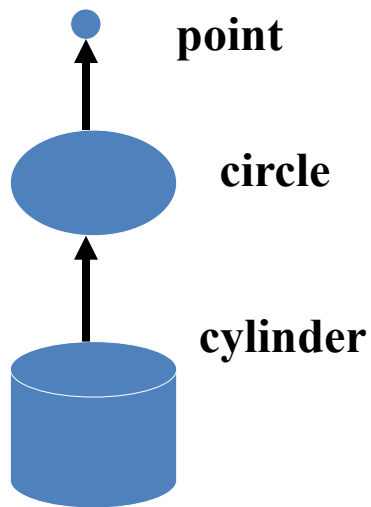


## 6.11 抽象类的应用

**【例】** 建立一个如右图所示图形类的继承层次结构。基类Shape是抽象类，通过它能够访问派生类Point、Circle、Cylinder的类名、面积、体积等内容。

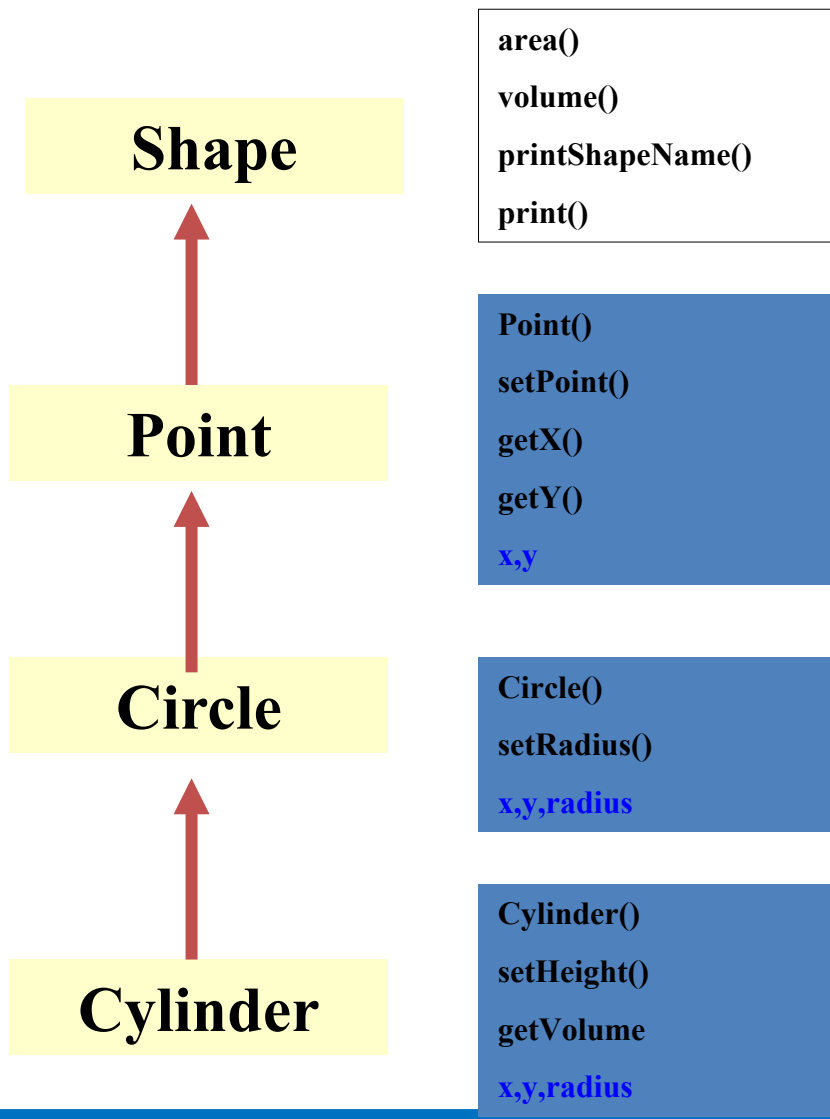


**【例6-25】** 建立一个如图6-8所示图形类的继承层次结构。基类Shape是抽象类，通过它能够访问派生类Point、Circle、Cylinder的类名、面积、体积等内容。





## 6.11抽象出虚基类：Shape



Shape只是概念上的几何图形，永远不会有称为shape的对象存在，它的存在只是为了提供Point, Circle, Cylinder的公有接口。所以shape的成员函数定义为：

```
area(){return 0;}  
volume(){return 0;}  
printShapeName()=0;  
print()=0;
```



# Shape. h

```
#ifndef SHAPE_H
#define SHAPE_H
#include <iostream.h>

class Shape {
public:
    virtual double area() const { return 0.0; }
    virtual double volume() const { return 0.0; }

    virtual void printShapeName() const = 0;
    virtual void print() const = 0;
};

#endif
```



# Shape. cpp

---

- 不需要此源文件,因为没有函数要定义



# Point. h

```
#ifndef POINT1_H
#define POINT1_H
#include "shape.h"

class Point : public Shape {
public:
    Point( int = 0, int = 0 );
    void setPoint( int, int );
    int getX() const { return x; }
    int getY() const { return y; }
    virtual void printShapeName() const
        { cout << "Point: "; }
    virtual void print() const;
private:
    int x, y;
};

#endif
```



# Point.cpp

```
#include "point1.h"
```

```
Point::Point( int a, int b ) { setPoint( a, b ); }
```

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

```
void Point::print() const  
{ cout << '[' << x << ", " << y << ']'< }<
```





# Circle.h

---

```
#ifndef CIRCLE1_H
#define CIRCLE1_H
#include "point1.h"

class Circle : public Point {
public:
    Circle( double r = 0.0, int x = 0, int y = 0 );
    void setRadius( double );
    double getRadius() const;
    virtual double area() const;
    virtual void printShapeName() const { cout << "Circle: "; }
    virtual void print() const;
private:
    double radius; // radius of Circle
};

#endif
```



# Circle.cpp

---

```
#include "circle1.h"
```

```
Circle::Circle( double r, int a, int b ) : Point( a, b )  
{ setRadius( r ); }
```

```
void Circle::setRadius( double r ) { radius = r > 0 ? r : 0; }
```

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

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

```
void Circle::print() const  
{ Point::print(); cout << "; Radius = " << radius;  
}
```



# Cylinder.h

```
#ifndef CYLINDR1_H
#define CYLINDR1_H
#include "circle1.h"

class Cylinder : public Circle {
public:
    Cylinder( double h = 0.0, double r = 0.0,
             int x = 0, int y = 0 );

    void setHeight( double );
    double getHeight();
    virtual double area() const;
    virtual double volume() const;
    virtual void printShapeName() const {cout << "Cylinder: ";}
    virtual void print() const;
private:
    double height;
};

#endif
```



# Cylinder.cpp

```
#include "cylindr1.h"
```

```
Cylinder::Cylinder( double h, double r, int x, int y ) : Circle( r, x, y )  
{ setHeight( h ); }
```

```
void Cylinder::setHeight( double h )  
{ height = h > 0 ? h : 0; }  
double Cylinder::getHeight() { return height; }
```

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

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

```
void Cylinder::print() const  
{  
    Circle::print();  
    cout << "; Height = " << height;  
}
```





# main.cpp

---

```
#include <iostream.h>
#include <iomanip.h>
#include "shape.h"
#include "point.h"
#include "circle.h"
#include "cylindr.h"
```

```
void virtualViaPointer( const Shape * );
void virtualViaReference( const Shape & );
```



# main.cpp

```
void virtualViaPointer( const Shape *baseClassPtr )
{
    baseClassPtr->printShapeName();
    baseClassPtr->print();
    cout << "\nArea = " << baseClassPtr->area()
    << "\nVolume = " << baseClassPtr->volume() << "\n\n";
}
```

```
void virtualViaReference( const Shape &baseClassRef )
{
    baseClassRef.printShapeName();
    baseClassRef.print();
    cout << "\nArea = " << baseClassRef.area()
    << "\nVolume = " << baseClassRef.volume() << "\n\n";
}
```



# main.cpp

```
int main()
{
    cout << setiosflags( ios::fixed | ios::showpoint )
    << setprecision( 2 );

    Point point( 7, 11 );
    Circle circle( 3.5, 22, 8 );
    Cylinder cylinder( 10, 3.3, 10, 10 );

    point.printShapeName();
    point.print();
    cout << '\n';

    circle.printShapeName();
    circle.print();
    cout << '\n';

    cylinder.printShapeName();
    cylinder.print();
    cout << "\n\n";

    Shape *arrayOfShapes[ 3 ];

    arrayOfShapes[ 0 ] = &point;

    arrayOfShapes[ 1 ] = &circle;

    arrayOfShapes[ 2 ] = &cylinder;

    cout << "Virtual function calls made off "
    << "base-class pointers\n";

    for ( int i = 0; i < 3; i++ )
        virtualViaPointer( arrayOfShapes[ i ] );

    cout << "Virtual function calls made off "
    << "base-class references\n";

    for ( int j = 0; j < 3; j++ )
        virtualViaReference( *arrayOfShapes[ j ] );

    return 0;
}
```

图中虚线为调用:

baseclassPtr->printShapeName();的过程

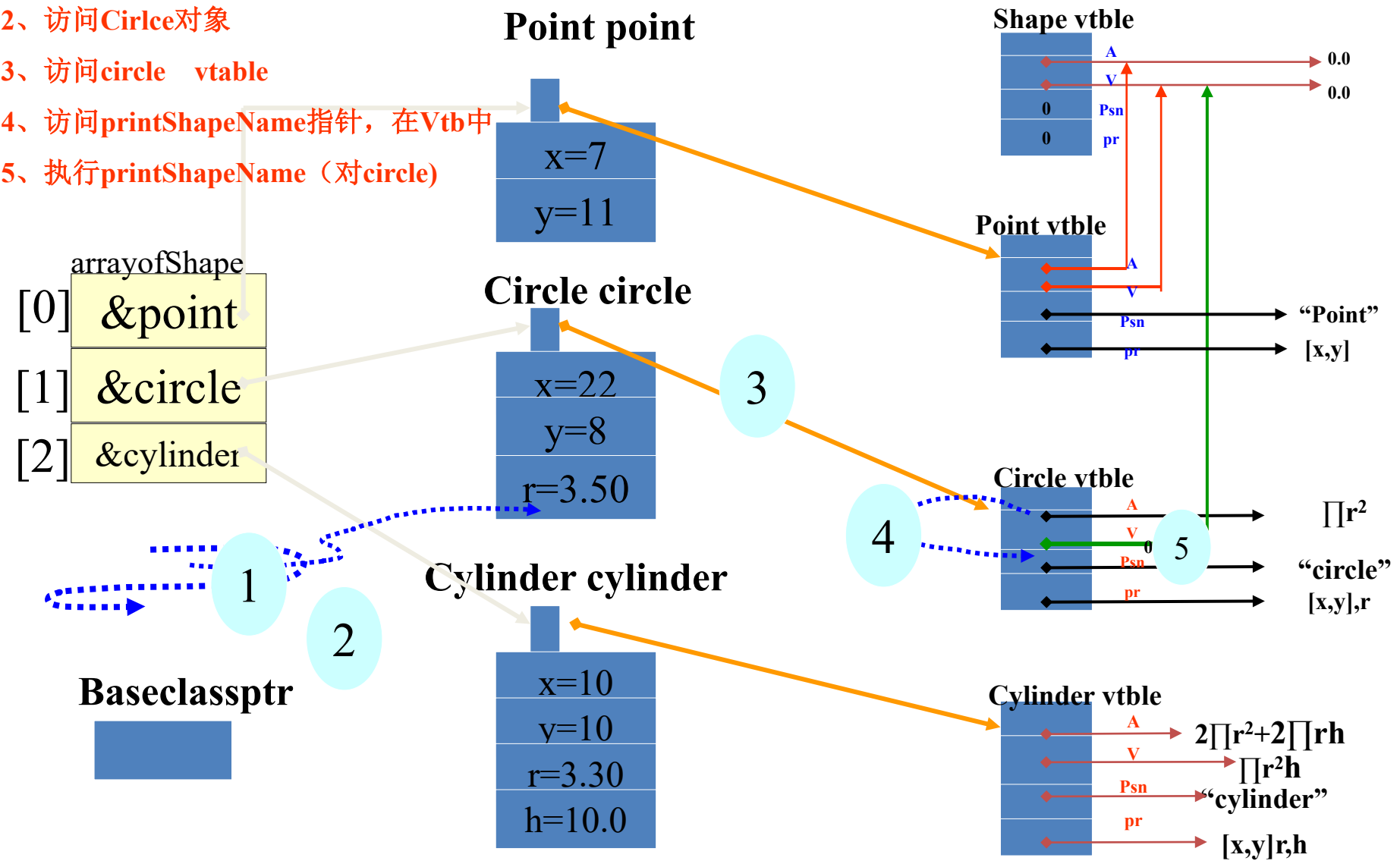
1、将&circle传入baseClassPtr

2、访问Circlce对象

3、访问circle vtable

4、访问printShapeName指针，在Vtb中

5、执行printShapeName（对circle）







## 6.10 运行时**类型信息**

### 1、RTTI

- **运行时类型信息 (Run-time Type Information, RTTI) 提供了在程序运行时刻确定对象类型的方法，是面向对象程序语言为解决多态问题而引入的一种语言特性，在最初的非多态程序设计语言中，并没有RTTI机制。**
- **在C++中，用于支持RTTI的运算符有：  
dynamic\_cast, typeid, type\_info**



## 6.10.1 `dynamic_cast`

### 1、`dynamic_cast`的用途

- `dynamic_cast`是一个强制类型转换操作符，主要用于多态基类的指针或引用与派生类指针或引用之间的转换，它是在程序运行时刻执行的。
- `const_cast`、`static_cast`和`reinterpret_cast`强制类型转换则是在编译时期完成的。

### 2、`dynamic_cast`的用法

`dynamic_cast`<目标类型>(表达式)



## 6.10.1 `dynamic_cast`

### 3、`dynamic_cast`强制类型

- **向上强制转换、向下强制转换**
- 向上转换是指在类的继承层次结构中，从派生类向基类方向的转换，即把派生类对象的指针或引用转换成基类对象的指针或引用，这种转换常用C++的默认方式完成。
- 与向上强制转换的方向相反，向下转换是指在类的继承层次结构中，从基类向派生类方向的转换，即把基类对象的指针或引用转换成派生类对象的指针或引用。



## 【例6-22】用dynamic\_cast实现基类与派生类对象之间的动态转换。

//CH6-22.cpp

```
#include<iostream>
```

```
using namespace std;
```

```
class Base{
```

```
public:
```

```
    virtual void f(){ cout<<"f in Base!"<<endl; }  
};
```



```
class Derived:public Base{
    void f(){ cout<<"f in Derived!"<<endl; }
};
int main(){
    Base *pb,b;
    Derived d,*pd;
    pb=&d;    //默认转换，编译时完成，是常用方式
    pb->f();
    pd=&d;
    pb=dynamic_cast<Base *>(&d);//向上转换，运行时完成
    pb->f();
    pb=dynamic_cast<Base *>(pd);//向上转换，运行时完成
    pb->f();
}
```

- 例：给定下面的类层次，其中每个类都定义了 public 默认构造函数和虚析构函数。

```
class A { /* ... */ };
```

```
class B : public A { /* ... */ };
```

```
class C : public B { /* ... */ };
```

```
class D : public B, public A { /* ... */ };
```

- 如果有，下面哪些 dynamic\_casts 失败？

- (a) A \*pa = new C;

```
    B *pb = dynamic_cast< B* >(pa);
```

- (b) B \*pb = new B;

```
    C *pc = dynamic_cast< C* >(pb);
```

- (c) A \*pa = new D;

```
    B *pb = dynamic_cast< B* >(pa);
```



## 6.10.2 typeid

- **typeid操作符在程序运行时判定一个对象的真实数据类型，typeid定义于头文件typeinfo中，它的用法如下：**

**typeid(exp)**



## 【例6-23】 用typeid判定数据的类型。

```
//CH6-23.cpp
#include <iostream>
using namespace std;
class A{};
int main(){
    A a,*p;
    A &rA=a;
    cout<<"1: "<<typeid(a).name()<<endl;
    cout<<"2: "<<typeid(p).name()<<endl;
    cout<<"3: "<<typeid(rA).name()<<endl;
    cout<<"4: "<<typeid(3).name()<<endl;
    cout<<"5: "<<typeid("this is
string").name()<<endl;
    cout<<"6: "<<typeid(4+9.8).name()<<endl;
}
```





**【例6-24】 在多态程序中，利用typeid获取基类指针所指的  
实际对象，并进行不同的成员函数调用。**

**//CH6-24.cpp**

**#include <iostream>**

**#include <typeinfo>**

**using namespace std;**

**class B{**

**int x;**

**public:**

**virtual void f(){ cout<<"1: B::f()"<<endl; }**

**};**

**class D1:public B{**

**public:**

**virtual void g(){ cout<<"2: D1::g()"<<endl; }**

**};**



```
class D2:public B{
    int x;
public:
    virtual void f(){ cout<<"3: D2::f() "<<endl; }
    void h(){ cout<<"4: D2::h()\n"; }
};
void AccessB(B *pb){
    if (typeid(*pb)==typeid(B))
        pb->f();
    else if (typeid(*pb)==typeid(D1)) {
        D1 *pd1=dynamic_cast<D1 *>(pb);
        pd1->g();
    }
    else if (typeid(*pb)==typeid(D2)) {
        D2 *pd2=dynamic_cast<D2 *>(pb);
        pd2->h();
    }
}
int main(){
    B b;
    D1 d1;
    D2 d2;
    AccessB(&b);           //输出:      1: B::f()
    AccessB(&d1);          //输出:      2: D1::g()
    AccessB(&d2);          //输出:      4: D2::h()
}
```