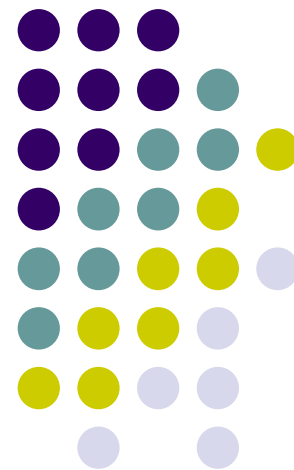


# 第2部分

## 多态性与虚函数

---

- 多态性的概念
- 静态联编和动态联编
  - 虚函数
- 纯虚函数与抽象类



## 1. 多态性的概念：

**多态性**是面向对象程序设计的重要特性。利用多态性可以设计和实现一个易于扩展的系统。

在C++中，多态性是指具有不同功能的函数用同一个函数名，即用同一函数名调用不同内容的函数。

**多态性的一般表述：**向不同的对象发送同一消息（调用函数），不同的对象会产生不同的行为（方法）。

例如，运算符+调用operator+函数，对不同类型数据的操作互不相同。

从系统实现的角度看，多态性分为两类：

①**静态多态性：**系统在编译的时候就能知道要调用的是哪个函数。也叫做编译时的多态性。（如函数重载）

②**动态多态性：**程序在运行过程中才动态地确定操作所针对的对象。又叫做运行时的多态性。

## 2. 静态联编和动态联编

静态联编:

实现的绑定  
的绑定  
程, 在编译时  
代码的绑定  
固定下来

```
class Rectangle: public Point
{ public:
    Rectangle(double i, double j, double k, double l);
    double Area() const {return w*h;}
```

实现  
动态联编!

静态联编

一个静态联编的例子:

```
#include <iostream>
using namespace std;
class Point
{ private:
    double x, y;
 public:
    Point(double i, double j);
    double Area() const;
```

```
int main()
{
    Rectangle rec(3.0, 5.2, 1.0, 1.0);
    fun(rec);
    return 0;
}
```

```
void fun(Point &s) //一般函数
{ cout << "Area=" << s.Area << endl;
```

请思考: 导致这种运行结果的原因是什么?

运行结果:  
Area=0



## 什么是动态联编：

根据目标对象的动态类型（而不是静态类型），在程序运行时（而不是在编译时）将函数名绑定到具体的函数实现上。

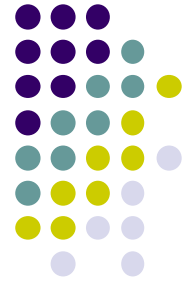
动态联编又称为动态绑定和晚期绑定。

问题提出：

程序员怎样告知编译程序  
哪些函数是静态联编？哪些函  
数是动态联编？

## 动态联编的实现——**虚函数**：

C++提供了**virtual**关键字用于指定函数是否为**虚函数**。如果在一个函数声明前面加上virtual，则这个函数为虚函数，系统将其进行**动态联编**。



## 动态联编的实现过程：

### 在程序编译时：

- ①为每一个有虚函数的类设置一个虚函数表v\_table, 一个指针数组, 存放每个虚函数的入口地址。
- ②在函数调用处插入一个隐藏的, 指向虚函数表的指针v\_pointer;

### 在程序运行中：

根据对象的v\_pointer, 在相应的虚函数表中获得函数入口, 来调用正确的函数。



### 3. 虚函数的定义与使用：

虚函数的定义方法：

① 在基类中以关键字**virtual**说明；

**virtual** <函数返回类型> <函数名>(<参数表>)

如: virtual double Area ( ) ;

② 在派生类中重新定义一个同名非静态成员函数。

3点说明：

- 派生类中同名函数必须与基类虚函数完全一致（即函数名、参数个数与类型、返回类型都相同）。
- 采用基类对象指针或引用来调用虚函数。
- 派生类必须以公用方式继承。

只有满足上面3点，程序才会按动态联编的方式调用函数，否则虚函数将按静态联编的方式调用。



```
class Rectangle: public Point
{ public:
    Rectangle(double i, double j, double k, double l);
    virtual double Area() const {return w*h;}
private:
    double w,h;
};
```

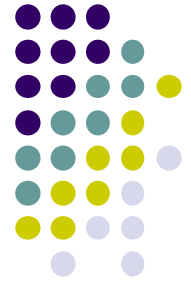
```
Rectangle::Rectangle(double i, double j, double k,
double l) :Point(i, j) { w=k; h=l; }
```

```
void fun(Point &s) //一般函数,参数是基类引用
{ cout<<"Area="<<s.Area()<<endl; }
```

```
#include<iostream>
using namespace std;
class Point
{ private:
    double x, y;
public:
    Point(double i, double j) {x=i; y=j;}
    virtual double Area() const
    { return 0.0;}
};
```

```
int main()
{
    Rectangle rec(3.0, 5.2, 15.0, 25.0);
    fun(rec);
    return 0;
}
```

**运行结果:**  
**Area=375**



## 总结虚函数的使用方法：

① 在基类用**virtual**声明成员函数为虚函数；

② 在派生类中重新定义此函数；

**C++规定，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。**

③ 定义一个指向基类的指针（或引用）；

④ 通过重新对该指针（或引用）做同类族对象赋值，调用该类对象同名虚函数。

虚函数与指向基类的指针（引用）变量配合使用，就能方便地调用同一类族中不同类对象的同名函数。从而实现运行时的多态性。





# 复习. 虚函数 ( Virtual Function )

## 类中采用虚函数:

```
class Student{
public:
    virtual void display(){
        cout<<"UnderGraduate\n";
    }
};
class GraduateStudent
    :public Student{
public:
    virtual void display(){
        cout<<"Graduate\n";
    }
};
注: 子类同名函数上的virtual可省
```

## fn函数实现非常简捷:

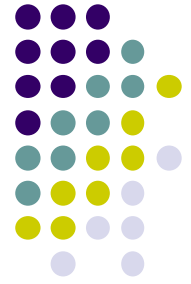
```
void fn(Student& x){
    x.display()
}
int main(){
    Student s;
    GraduateStudent gs;
    fn(s); // 显示大学生信息
    fn(gs); // 显示研究生信息
}
结果:
UnderGraduate
Graduate
```



多态性使得应用程序使用类体系中的祖孙对象共存的复杂局面达到了一种编程自在境界.

程序员从使用孤立的类(抽象数据类型), 到使用分层的类, 并且让各种对象“同场竞技”, 充分展现其个性. 尝到了对象化编程的真正乐趣.

C++类机制的虚函数就是冲着让类编程实质性地支持应用编程中对家族化对象操作依赖的目的, 从而面向对象来分析、设计和解决问题.



- 课堂练习:

写一个程序，定义基类**Point**，由它派生出**2**个派生类：

**Circle**(圆形)、

**Cylinder**(圆柱形)，

为上述三个类添加计算并显示面积的成员函数**PrintArea**（），要求**PrintArea**（）采用虚函数的形式，并通过基类指针调用虚函数**PrintArea**（）。**3**个图形的数据在定义对象时给定。

## 4. 避免误用虚函数 ( Avoiding Misusing Virtual Function )

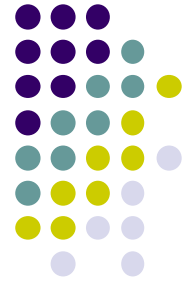


- 子类重载父类成员函数不能传播“虚”性
- 程序运行结果？

```
class Base{
public:
    virtual void fn(int x)
    {
        cout<<"Base\n";
    }
};

class Sub : public Base{
public:
    virtual void fn(double x)
    {
        cout<<"Sub\n";
    }
};

void test(Base& b)
{
    b.fn(3.5);
}
```



**函数重载对于编译识别来说，返回类型是不起作用的。**

```
void fn(int);  
int fn(int);
```

**对于 `fn(3.5)` ; 无法判断应该调用哪个函数。而使调用遭遇编译失败。**

**注意：对于虚函数来说，声明：**

```
virtual Base* fn();  
virtual Sub* fn();
```

**应看作不能分辨其多态调用的虚函数，但却引编译认为是同一个虚函数而获得通过。**



在什么情况下应该声明虚函数：

- ① 函数所在的类会作为基类，并且有更改功能的需要；
- ② 对于继承后不需要更改功能的函数不要声明为虚函数；
- ③ 考虑对成员函数的调用是否通过指针或引用；

虚函数的意义：

如果说：数据封装使得代码模块化；

继承实现了代码重用；

那么，虚函数采用动态联编技术造就了程序设计的多态性——接口重用。

```
class CCircularShape
{ public:
    static const double PI;
    CCircularShape() { ..... }
    CCircularShape(double r) { radius
```

```
class CCircle : public CCircularShape
{ public:
    char *
    double
    double
};
```

这就是虚函数的妙用！  
基类指针p可调用同类族中  
不同类的虚函数——多态性。

```
class CCylinder : public CCircle
{ public:
    char *Type() const { return "圆柱"; }
    double SurfaceArea() const
    double Volume() const
```

### 一个虚函数的例子

```
class CSphere : public CCircularShape
{ public:
    char *Type() const { return "球"; }
    double SurfaceArea() const
    double Volume() const
};
```

```
class CCone : public CCircle
{ public:
    char *Type() const { return "圆锥"; }
    double SurfaceArea() const
    double Volume() const
};
```

```
void main()
{ CCircularShape *p = 0;
    //定义基类指针p
    p = new CCircle(10);
```

```
cout << p->Type() << "的表面积是:" <<
    p->SurfaceArea();
p = new CSphere(10);
cout << p->Type() << "的表面积是:" <<
    p->SurfaceArea();.....}
```

问, 在下面的程序中, main函数执行后会输出什么? 为什么?



```
class Shape {
public:
    virtual void draw(){cout << "Draw a shape" << endl;};
};

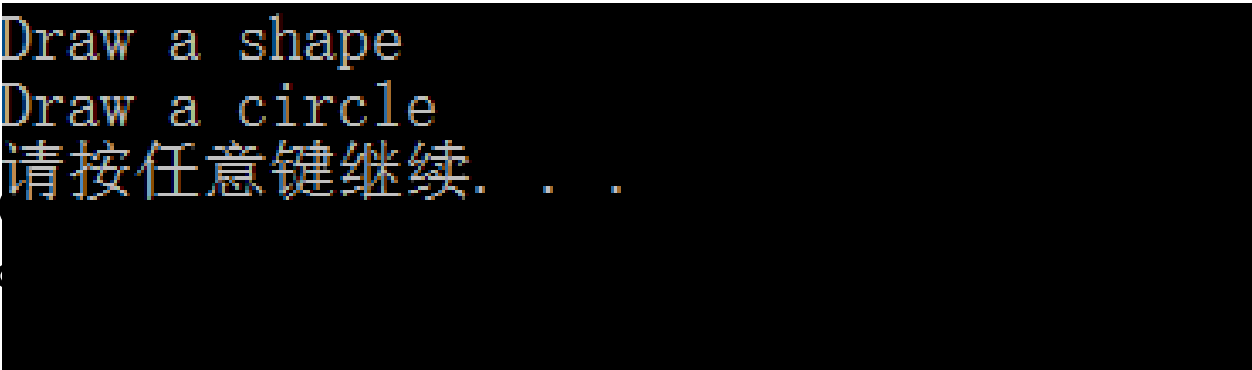
class Circle : public Shape{
public:
    void draw(){cout << "Draw a circle" << endl;};
};

int main()
{
    Shape shape;
    Circle circle;

    Shape* a[2]={&shape, &circle};

    for( int i=0; i<2; i++)
        a[i]->draw();
}
```

c:\users\czy\documents\visual studio 2010\Prj





# 析构函数是否需要声明为虚函数?

先看一个例题:

```
#include<iostream>
using namespace std;
class Point
{ public:
    Point(){}
    virtual ~Point(){cout<<"析构Point\n";}
};
```

```
class Circle:public Point
{ public:
    Circle(){}
    ~Circle(){cout<<"析构Circle\n";}
private:
    int radius;
};
```

用new开辟  
Circle类对象的  
临时动态存储空间

```
int main()
{ Point *p=new Circle;
  delete p;
  return 0;
}
```

用delete释  
放该空间

运行结果:  
析构Circle  
析构Point



## 虚析构函数:

- \* 如果基类的析构函数为虚析构函数，派生类的析构函数也均为虚析构函数。
- \* 如果基类的析构函数为虚析构函数，无论基类指针指的是哪一个派生类对象，当该对象撤销时，系统都会采用动态联编，调用相应的析构函数。

**程序中最好把析构函数声明为虚函数！即使基类不需要析构函数，也显式地定义一个函数体为空的虚析构函数。**

**构造函数不能声明为虚函数！！！！**

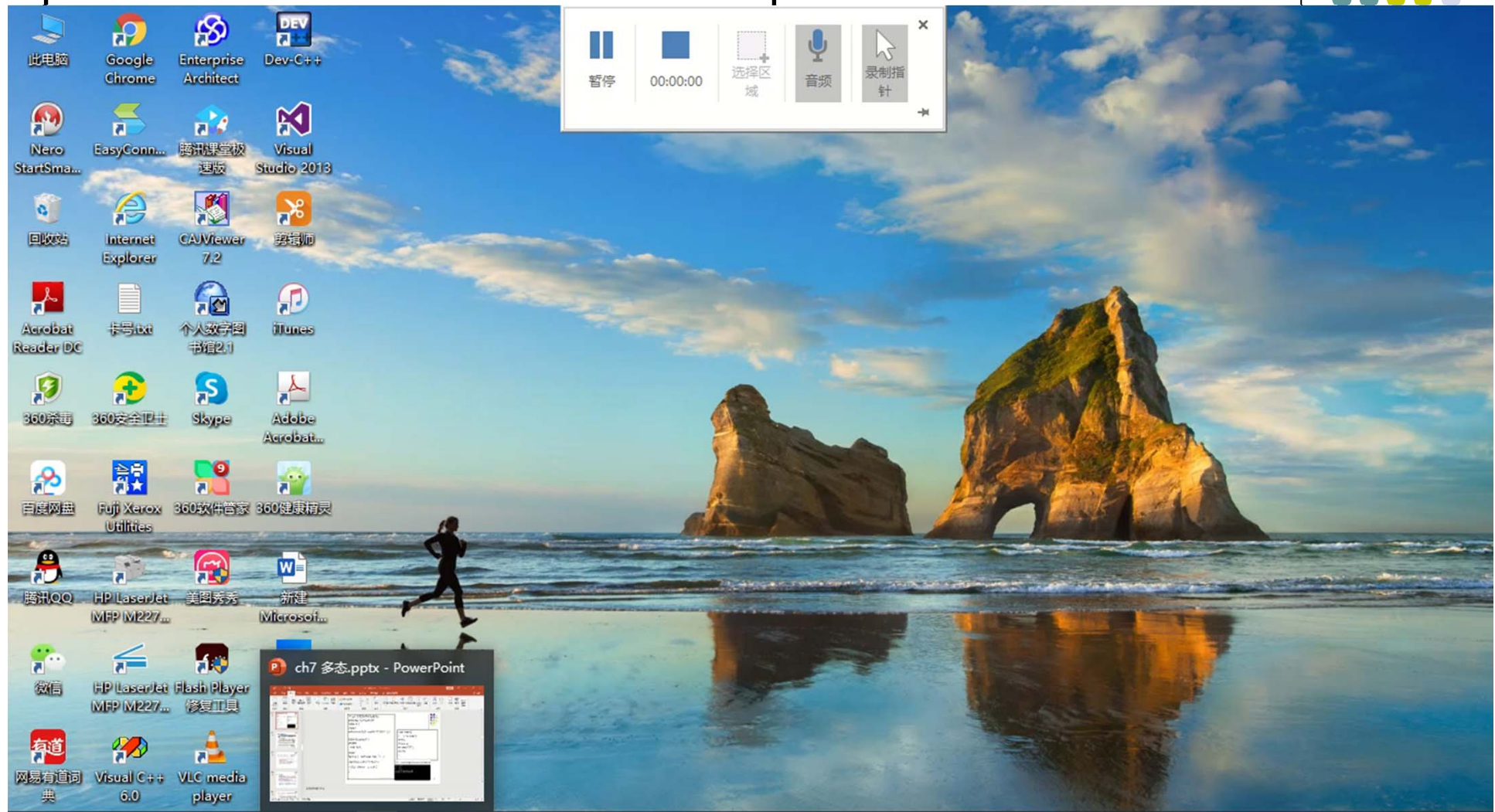


## 再次提醒:

- 1、多态性是通过成员函数捆绑不同类型的对象来体现的，所以，虚函数**一定是成员函数**，而且，**静态成员函数都不行**，因为它不捆绑对象，同样，**构造函数也不行**，因为它只产生对象，也不捆绑对象。
- 2、**析构函数可以是虚函数**，鼓励类继承体系中的每个类最好其析构函数都是虚函数。
- 3、一旦设置了虚函数，就与编译器达成了滞后联编的协议，函数必定分离于当前运行的模块，因而就**不可能是内联函数**了。

写出下面程序的输出结果。

```
#include <iostream.h>
```





#### 4. 纯虚函数和抽象类:

对于物理上无法实现而逻辑上又不得不存在的抽象的虚函数，可以将其在基类中用不包括任何代码的纯虚函数来定义。

纯虚函数的定义方法:

**virtual** <函数返回类型> <函数名>(<参数表>) = 0

```
virtual char *Type() const { return NULL; } //虚函数  
virtual double Volume() const { return 0; } //虚函数  
virtual double SurfaceArea() const { return 0; } //虚函数
```



```
virtual char *Type() const = 0; //纯虚函数  
virtual double Volume() const = 0; //纯虚函数  
virtual double SurfaceArea() const = 0; //纯虚函数
```



```
virtual char *Type() const = 0;           //纯虚函数  
virtual double Volume() const = 0;       //纯虚函数  
virtual double SurfaceArea() const = 0;   //纯虚函数
```

- 纯虚函数是声明时被初始化成“0”的成员函数。
- ①纯虚函数没有函数体；
- ②最后面的“=0”并不表示函数返回值为0，它只起形式上的作用，告诉编译系统“这是纯虚函数”；
- ③这是一个声明语句，最后应有分号。



## ● 四、纯虚函数

### ● 纯虚函数注意：

(1)只有函数的名字而**不具备函数的功能**，不能被调用。它只是通知编译系统：“在这里声明一个虚函数，留待派生类中定义”。在派生类中对此函数提供定义后，它才能具备函数的功能，可被调用。

(2)**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。

如果在基类中没有保留函数名字，则无法实现多态性。

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。



# 第11讲 再论多态：虚函数和抽象类



## ● 四、抽象类

- **概念：**有一些类，它们不以创建对象为目的，而用来作为基类去建立派生类。它们作为一种基本类型提供给用户，用户在这个基础上根据自己的需要定义出功能各异的派生类，然后用这些派生类去建立对象。这种只作为一种基本类型用作继承的类，称为抽象类(**abstract class**)。
- **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。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。



# 第11讲 再论多态：虚函数和抽象类



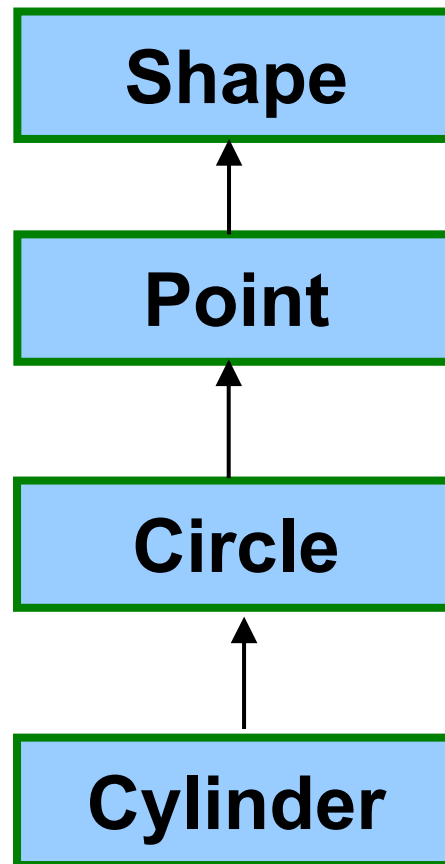
## ● 抽象类

- 一个类层次结构中当然也可不包含任何抽象类，每一层次的类都是实际可用的，可以用来建立对象的。但是，许多好的面向对象的系统，其层次结构的顶部是一个抽象类，甚至顶部有好几层都是抽象类。
- 如果在抽象类所派生出的新类中对基类的所有纯虚函数进行了定义，那么这些函数就被赋予了功能，可以被调用。这个派生类就不是抽象类，而是可以用来定义对象的具体类（**concrete class**）。
- 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。
- 虽然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。

# 第11讲 再论多态：虚函数和抽象类



- 四、抽象类
  - 抽象类例



# 第11讲 再论多态：虚函数和抽象类



- 四、抽象类
  - 抽象类例-(1)Shape类

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

# 第11讲 再论多态：虚函数和抽象类



- 抽象类
  - 抽象类例-(2)Point类

```
#ifndef POINT_H
#define POINT_H //声明Point类
#include "Shape.h"
class Point:public Shape { //Point是Shape的公用派生类
public:
    Point(float=0,float=0);
    void setPoint(float,float);
    float getX( ) const {return x;}
    float getY( ) const {return y;}
    virtual void shapeName( ) const { cout<<"Point:";}
    //对虚函数进行再定义
    friend ostream& operator<<(ostream&,const Point &);
protected:
    float x,y; };
#endif //point.h
```

# 第11讲 再论多态：虚函数和抽象类

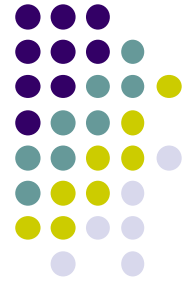


## ● 四、抽象类

### ● 抽象类例-(2)Point类

```
//定义Point类成员函数
#include "Point.h"
Point::Point(float a,float b)
{
    x=a;
    y=b;
}
void Point::setPoint(float a,float b)
{
    x=a;
    y=b;
}
ostream& operator<<(ostream&output
                    ,const Point& p){
    output<<"["<<p.x<<","<<p.y<<"]";
    return output;
} //point.cpp
```

# 第11讲 再论多态：虚函数和抽象类



## ● 四、抽象类

### ● 抽象类例-(3)Circle类

```
#ifndef CIRCLE_H
#define CIRCLE_H
//声明Circle类
#include "Point"
class Circle:public Point {
public:
    Circle(float x=0,float y=0,float r=0);
    void setRadius(float);
    float getRadius( ) const;
    virtual float area( ) const;
    virtual void shapeName( ) const {cout<<"Circle:";}
    //对虚函数进行再定义
    friend ostream& operator<<(ostream&,const Circle &);
protected:
    float radius;
};
#endif //circle.h
```

# 第11讲 再论多态：虚函数和抽象类



- 四、抽象类
  - 抽象类例-(3)Circle类

```
//circle.cpp
#include "circle.cpp"

Circle::Circle(float a,float b,float r):Point(a,b),radius(r){ }

void Circle::setRadius(float r) { radius = r; }
float Circle::getRadius( ) const {return radius;}

float 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11讲 再论多态：虚函数和抽象类



## ● 四、抽象类

### ● 抽象类例-(4)Cylinder类

```
#ifndef CYLINDER_H
#define CYLINDER_H
#include "Circle.h"
class Cylinder:public Circle {
public:
    Cylinder (float x=0,float y=0,float r=0,float h=0);
    void setHeight(float);
    virtual float area( ) const;
    virtual float volume( ) const;
    virtual void shapeName( ) const { cout<<"Cylinder:";}
    //对虚函数进行再定义
    friend ostream& operator<<(ostream&,const Cylinder&);
protected:
    float height;    };
#endif
```



# 第11讲 再论多态：虚函数和抽象类

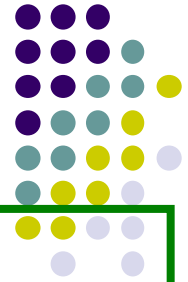


## ● 四、抽象类

### ● 抽象类例-(4)Cylinder类

```
#include "Circle.h" //Cylinder.cpp
Cylinder::Cylinder(float a,float b,float r,float h):Circle(a,b,r),height(h)
{ }
void Cylinder::setHeight(float h)
{ height=h;}
float Cylinder::area( ) const
{ return 2*Circle::area( )+2*3.14159*radius*height; }
float Cylinder::volume( ) const
{ return Circle::area( )*height; }
ostream& operator<<(ostream& output,const Cylinder& cy)
{ output<<"["<<cy.x<<","<<cy.y<<"], r="
                                     <<cy.radius<<," h="<<cy.height;
  return output;
}
```

# 第11讲 再论多态：虚函数和抽象类



- 四、抽象类
  - 抽象类例-(5)main

```
#include "Circle.h"
#include "Cylinder.cpp"
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; //基类指针
  pt=&point; //指针指向Point类对象
  pt->shapeName( ); //动态关联
```

```
cout<<"x="<<point.getX( )<<
      ",y="<<point.getY( )<<
      "\\narea="<<pt>area( )<<
      "\\nvolume="<<pt->volume()<<"\\n\\n";
```

```
pt=&circle; //指向Circle类对象
pt->shapeName( ); //动态关联
cout<<"x="<<circle.getX( )<<
      ",y="<<circle.getY( )<<
      "\\narea="<<pt>area( )<<
      "\\nvolume="<<pt>volume( )<<"\\n\\n";
```

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

# 第11讲 再论多态：虚函数和抽象类



## ● 四、抽象类

### ● 抽象类例-(6)总结

- (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，它一般并不是现实存在的对象的抽象(例如圆形(**Circle**)就是千千万万个实际的圆的抽象)，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。

# 第11讲 再论多态：虚函数和抽象类



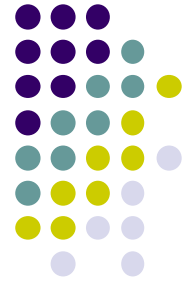
- 四、抽象类

- 抽象类例-(6)总结

- (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有相同的函数名、函数类型、参数个数和类型的函数，均为虚函数(不论在派生类中是否用**virtual**声明)。

- (7)使用虚函数提高了程序的可扩充性。把类的声明与类的使用分离。这对于设计类库的软件开发商来说尤为重要。开发商设计了各种各样的类，但不向用户提供源代码，用户可以不知道类是怎样声明的，但是可以使用这些类来派生出自己的类。



## 抽象类：

包含一个或多个纯虚函数的类叫做抽象类。

## 有关抽象类：

- ① 不能创建抽象类的对象；
- ② 抽象类不能作为参数类型、函数返回和显式转换；
- ③ 可以定义指向抽象类的指针；
- ④ 不能调用抽象类的纯虚函数，它仅仅是为派生类提供的一个公共接口；

**C++采用抽象类，将接口和实现分离开来。**