



中南大學
CENTRAL SOUTH UNIVERSITY



面向对象程序设计

Object Oriented Programing

第五章 多态性和运算符重载

张宝一

地理信息系

zhangbaoyi@csu.edu.cn



本章主要内容

第2章

类的特性

第3章

封装

实现代码模块化

第4章

继承

实现代码扩展

第5章

多态性

实现函数(接口)重用

动态多态
(虚函数)

静态多态
(运算符重载、模板)

本章主要内容

重点:

- 多态性的定义
- 虚函数 (`virtual`)
- 运算符重载 (`operator`)
- 类模板 (`template`)

难点:

- 虚函数的作用机制
- 常用运算符重载 (如: `+` `-` `*` `/` `[]` `()` 等)
- 函数模板与类模板的实例化

第五章：多态性和运算符重载

5.1 多态性概述



5.2 虚函数

5.3 运算符重载

5.4 类模板

5.1 多态性概述

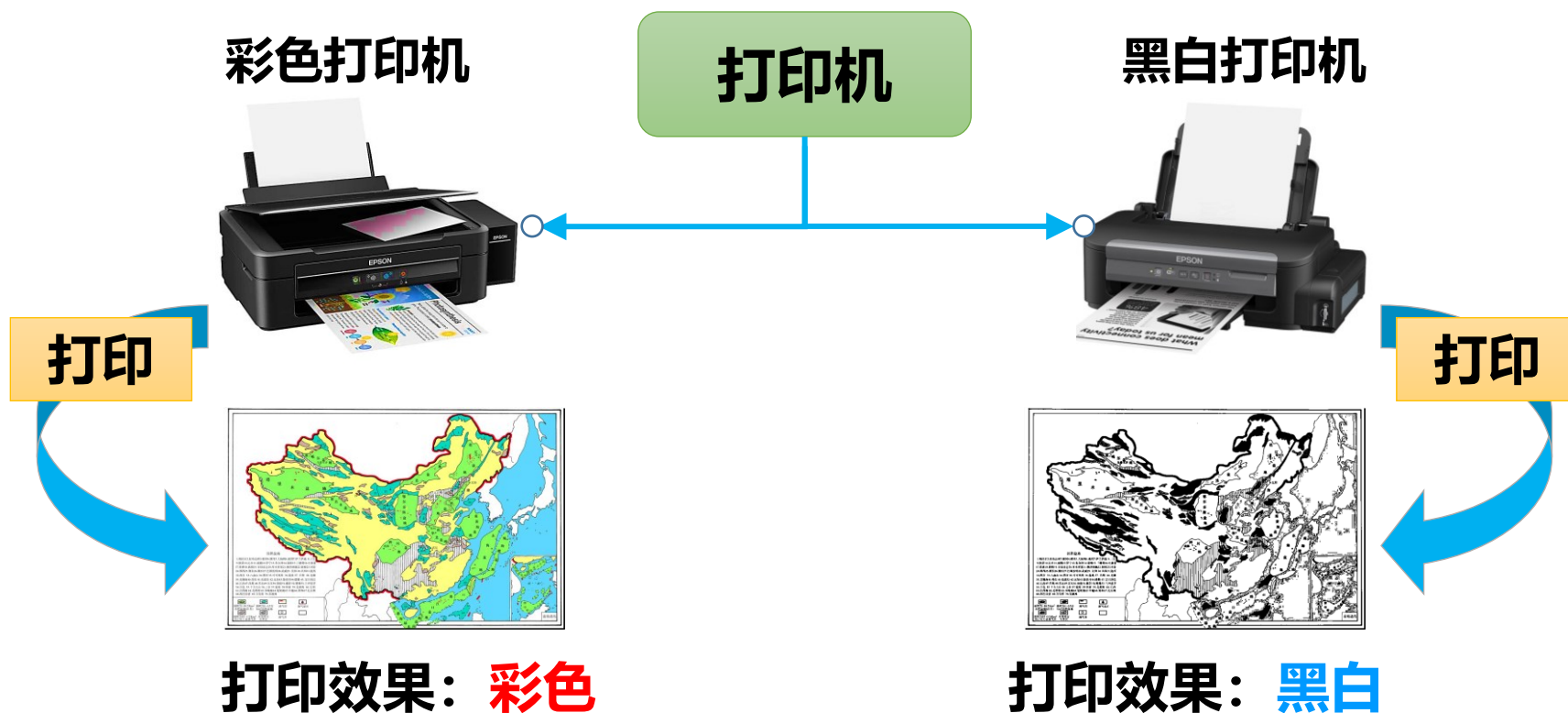
本节主要内容:

- 什么是多态性?
- 多态性的现实示例
- 为什么要用多态?
- 多态性的实现方式

5.1 多态性概述

5.1.1 什么是多态性?

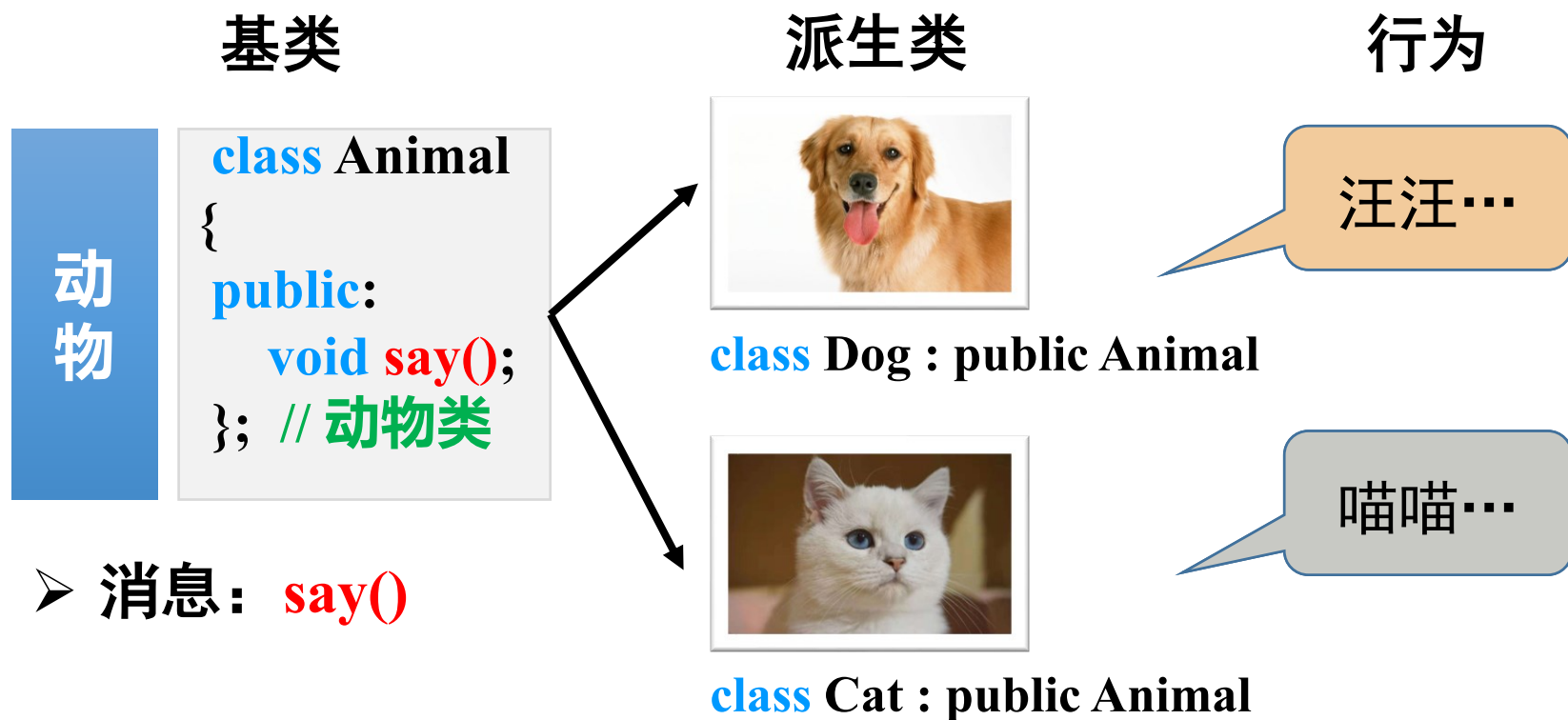
- **多态性**是指发出**同样的消息**被**不同类型的对象**接收时，可能导致**完全不同的行为**。



5.1 多态性概述

5.1.2 多态性的现实示例

- 多态性是指发出**同样的消息**被**不同类型的对象**接收时，可能导致**完全不同的行为**。



5.1 多态性概述

5.1.2 多态性的现实示例



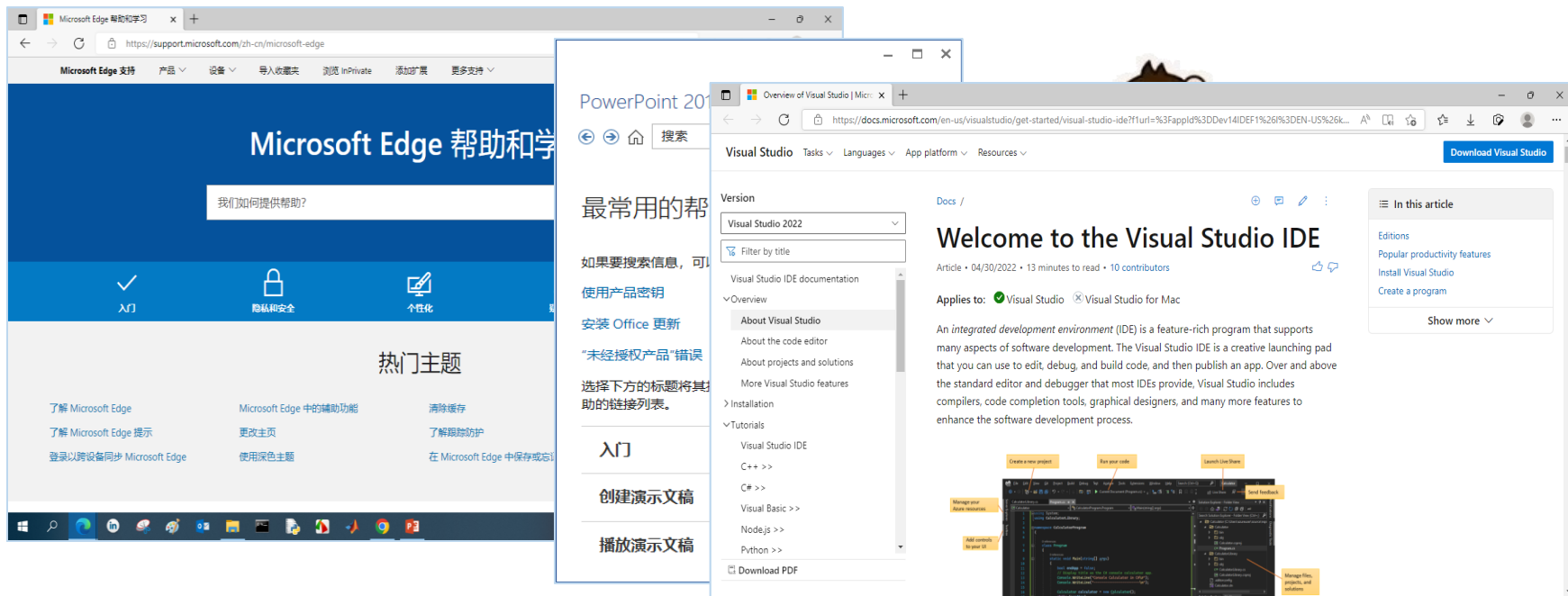
□ 表达“欢迎”：**欢迎** (中文)，**Welcome** (英语) ...

5.1 多态性概述

5.1.2 多态性的现实示例

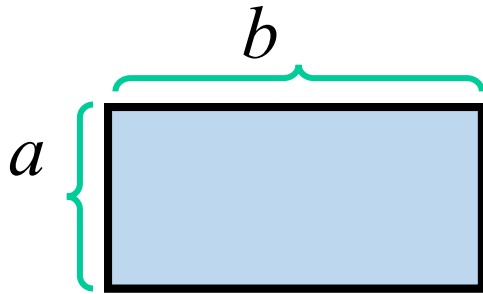
● 课堂小试验

分别在【浏览器】、【PowerPoint】、【Visual Studio】界面下**按下F1键**，你发现了什么？



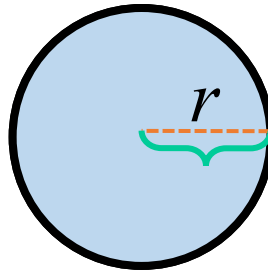
5.1 多态性概述

5.1.3 为什么要用多态?



矩形

```
class Rectangle {  
public:  
    double a, b;  
    double area(){  
        return a*b;  
    }  
};
```



圆形

```
class Circle {  
public:  
    double r;  
    double area(){  
        return PI*r*r;  
    }  
};
```

```
void print(Rectangle &s);  
void print(Circle &s);
```

```
void main(){  
    Rectangle s1;  
    Circle s2;
```

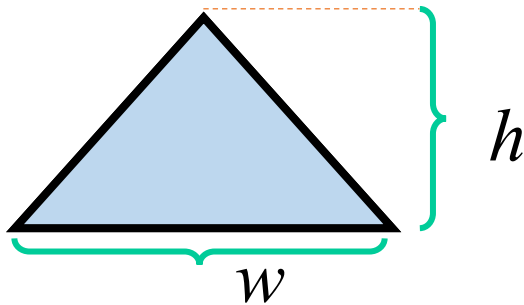
```
    cout<<s1.area();  
    cout<<s2.area();
```

```
    print(s1);  
    print(s2);
```

```
}
```

5.1 多态性概述

5.1.3 为什么要用多态?



三角形

```
class Triangle {  
public:  
    double w, h;  
    double area(){  
        return w*h/2.0;  
    }  
};
```

```
void print(Rectangle &s);
```

```
void print(Circle &s);
```

```
void print(Triangle &s); // 增加函数
```

```
void main() {
```

```
    Rectangle s1;
```

```
    Circle s2;
```

```
    Triangle s3;
```

```
    cout<<s1.area();
```

```
    cout<<s2.area();
```

```
    cout<<s3.area(); // 增加代码
```

```
    print(s1);
```

```
    print(s2);
```

```
    print(s3); // 增加代码
```

```
}
```

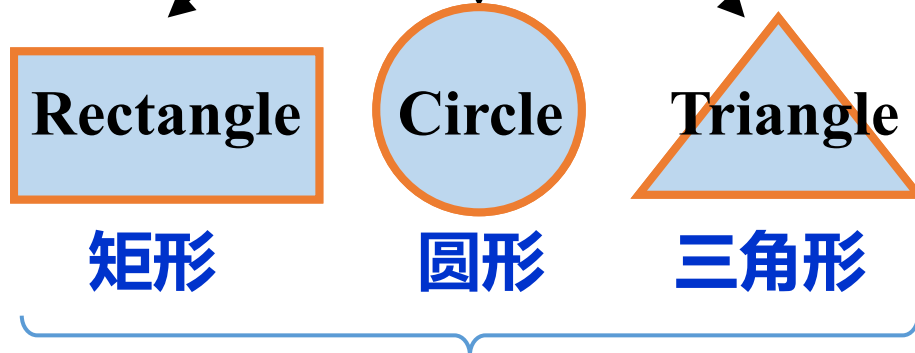
5.1 多态性概述

5.1.3 为什么要用多态?

基类



派生类

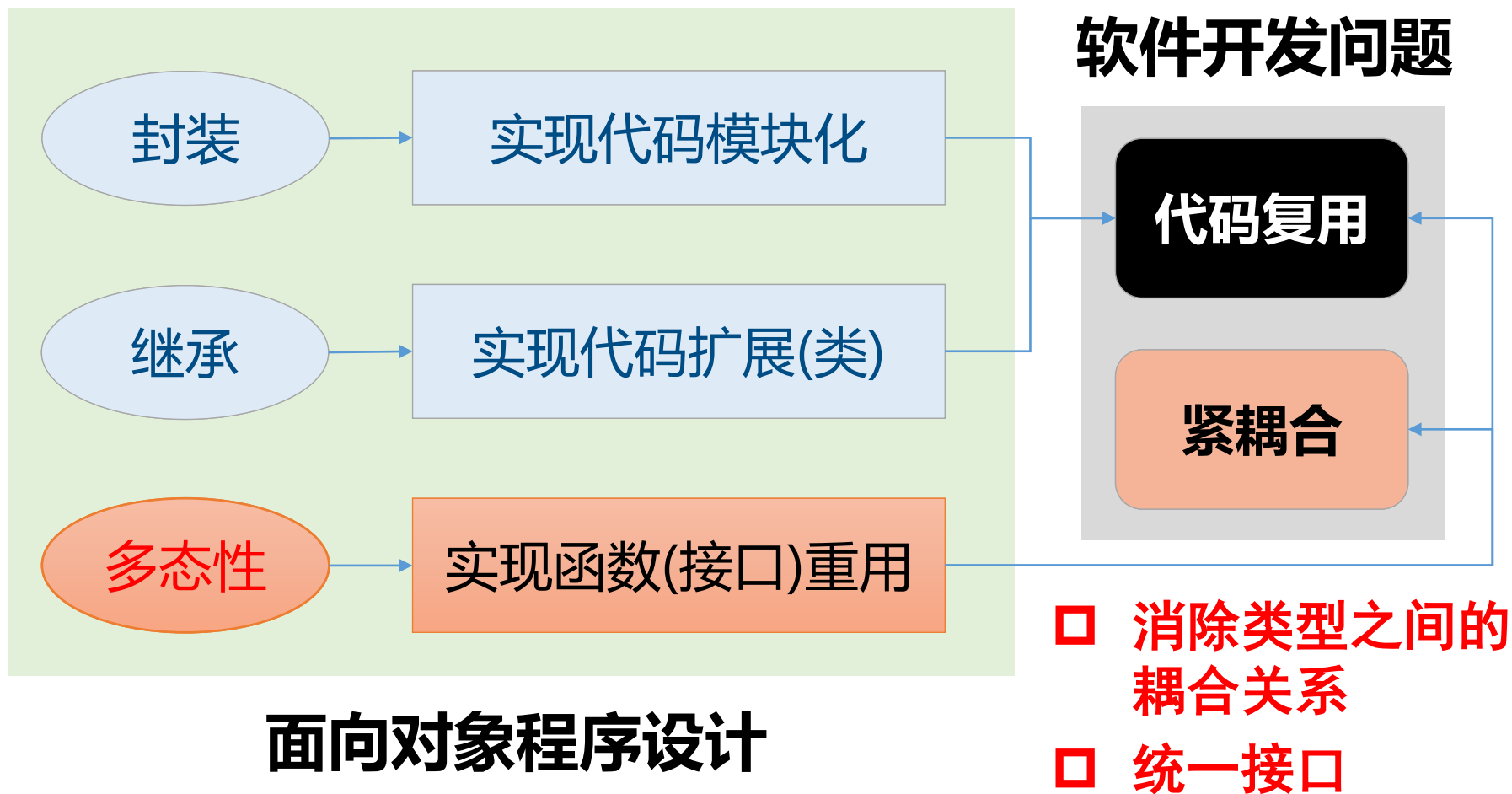


□ 消息: **print()**

```
void main() {  
    Shape *p;  
    p = new Rectangle;  
    p->print();  
  
    p = new Circle;  
    p->print();  
  
    p = new Triangle;  
    p->print();  
}
```

5.1 多态性概述

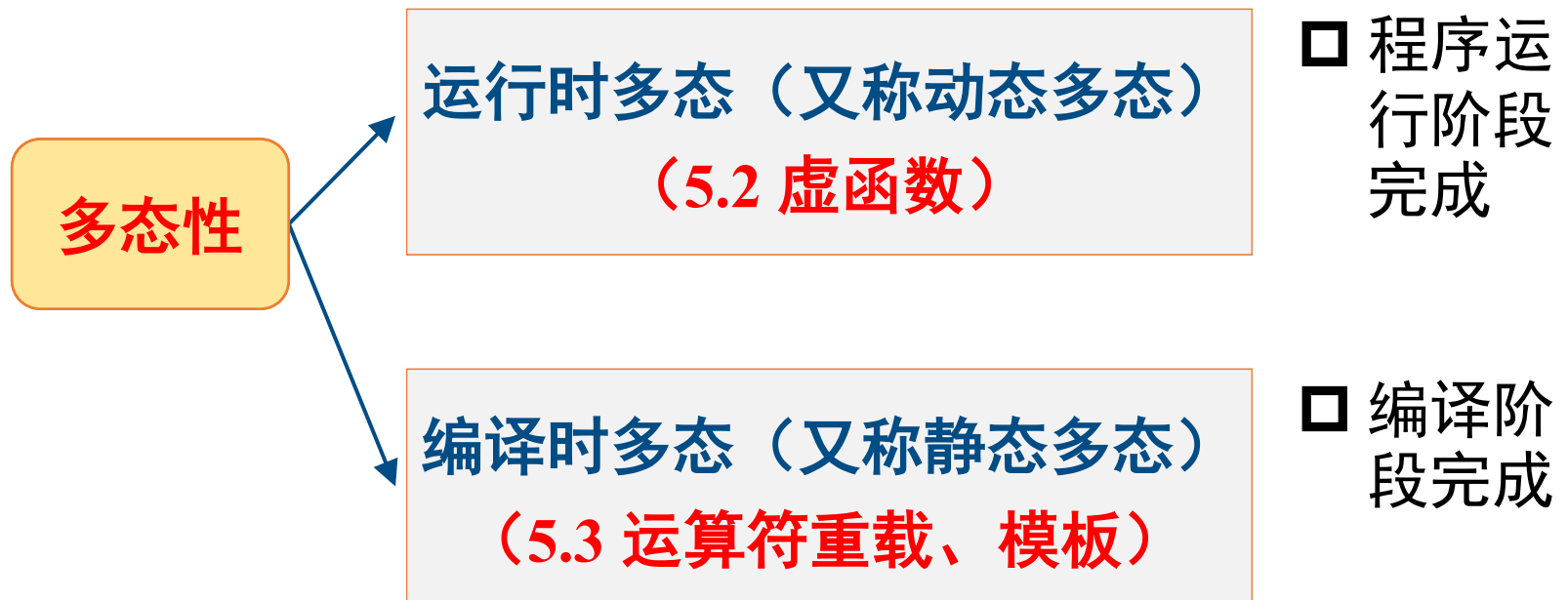
5.1.3 为什么要用多态?



5.1 多态性概述

5.1.4 多态性的实现方式

- 多态从实现的角度来讲可以划分为两类：**运行时的多态** 和 **编译时的多态**。



5.2 虚函数

本节主要内容:

- 虚函数的定义
- 虚析构函数
- 纯虚函数
- 抽象类

5.2 虚函数

5.2.1 动态多态性

多态性是面向对象程序设计的重要特征之一。多态性是指发出同样的消息被不同类型的对象接收时导致完全不同的行为。

动态多态是一种运行期绑定机制，通过这种机制，实现将函数名绑定到函数具体实现代码的目的。

一个函数在内存当中起始的地址就称为这个函数的入口地址。

动态多态就是将函数名称动态地绑定到函数入口地址的运行期绑定机制。

5.2 虚函数

5.2.1 动态多态性

编译期绑定是函数需要执行的代码由编译器在编译阶段确定了的。也就是说将某个函数名和其入口地址进行绑定在一起。

运行期绑定是直到程序运行之时（不是在编译时刻），才将函数名称绑定到其入口地址。

如果对一个函数的绑定发生在运行时刻而非编译时刻，我们就称该函数是**动态多态**的。

5.2 虚函数

5.2.1 动态多态性的前提条件

- 必须存在一个继承体系结构。
- 继承体系结构中的一些类必须具有同名的 `virtual` 成员函数（`virtual` 是关键字）
- 至少有一个基类类型的指针或基类类型的引用。这个指针或引用可用来对 `virtual` 成员函数进行调用。

5.2 虚函数

5.2.1 虚函数的定义

虚函数是实现动态多态性的重要机制。

□ 例5.1:

```
class Animal {  
public:  
    virtual void say(){ cout<<"[*]"; }  
};  
  
class Dog : public Animal {  
public: void say() { cout<<"汪"; }  
};  
  
class Cat : public Animal {  
public: void say() { cout<<"喵"; }  
};
```

```
void main() {  
    Animal a, *p;  
    Dog dog;  
    Cat cat;  
    → p=&a;    p->say();  
    → p=&dog;  p->say();  
    → p=&cat;  p->say();  
}
```

运行结果:

[*]汪喵

5.2 虚函数

5.2.1 虚函数的定义

□ 虚函数的定义

虚函数的定义是在基类中需要定义为虚函数的成员函数的声明中冠以关键字**virtual**。虚函数定义如下：

```
virtual 返回值类型 函数名 (形参表) { // 函数体 }
```

当基类中的某个成员函数被声明为虚函数后，此虚函数就可以在一个或多个派生类中被重新定义。

□ 虚函数的作用

虚函数实现了在基类定义通用接口，而在派生类定义接口的具体实现方法，即实现“**同一接口，多种方法**”。

5.2 虚函数

5.2.1 虚函数的定义

基类中定义了虚函数后，遵循以下的规则来判断派生类中成员函数是否为虚函数：

- (1) 该函数与基类的虚函数有相同的名称。
- (2) 该函数与基类的虚函数有相同的参数个数和类型。
- (3) 该函数与基类的虚函数有相同的返回类型。

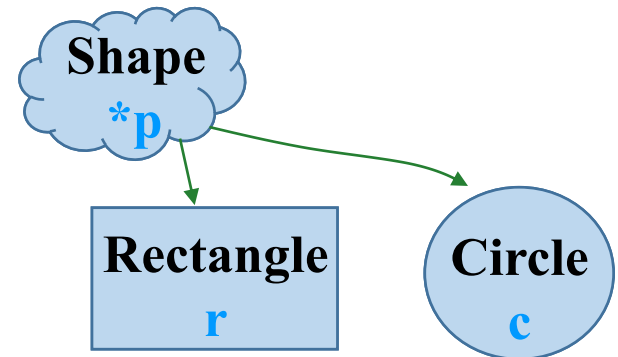
```
class Shape {  
public:  
    virtual float area();  
    //虚函数  
};
```

```
class Circle : public Shape {  
public:  
    float area(); // 是否为虚函数?  
    float area(double r);  
};
```

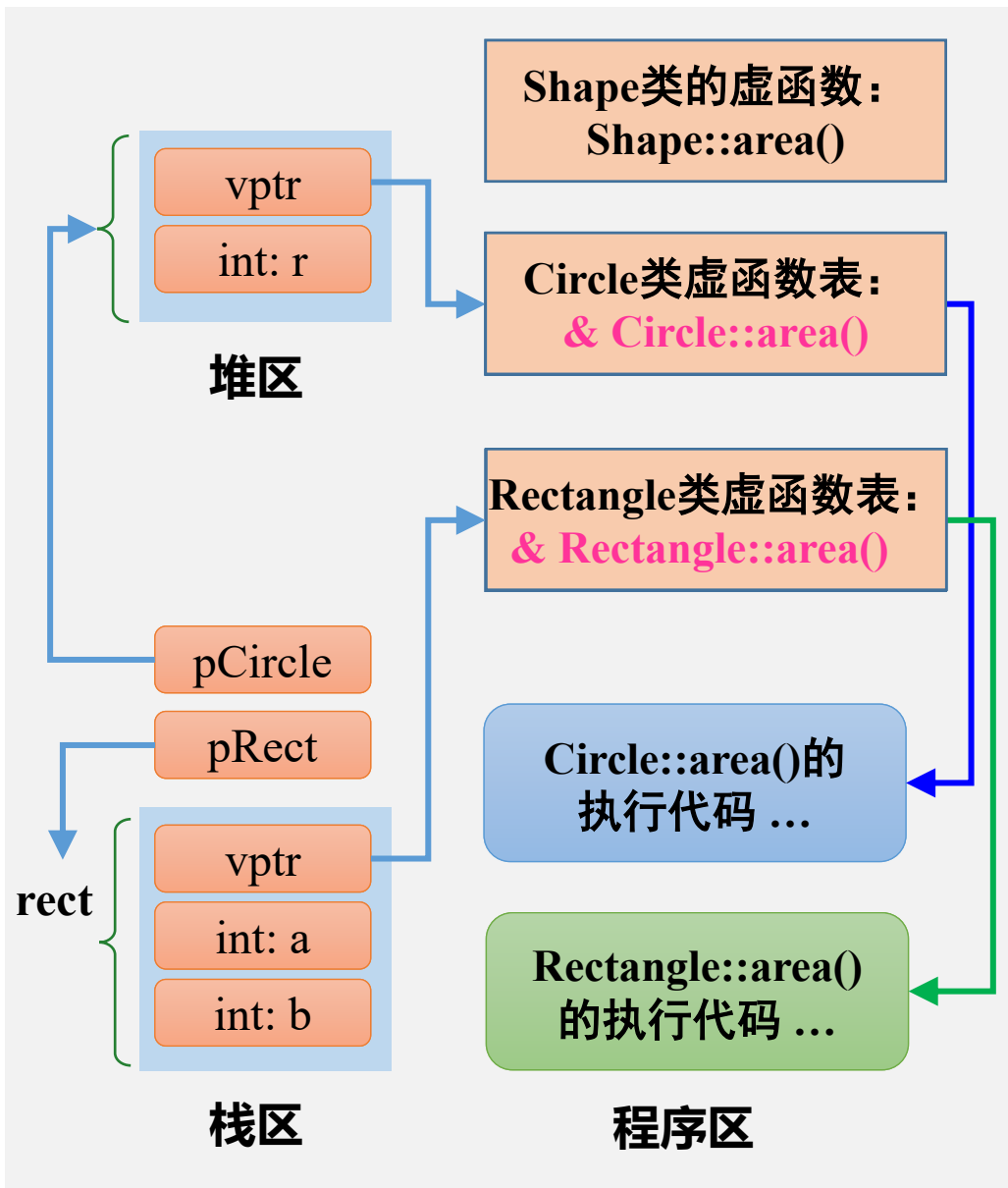
例5.2 虚函数的定义举例

```
class Shape {  
    public:  
        virtual void area() //定义虚函数  
        { cout<<"Shape::area"<<endl; }  
};  
  
class Circle : public Shape {  
    public:  
        void area() //派生类中重定义  
        { cout<<"Circle::area"<<endl; }  
};  
  
class Rectangle : public Shape {  
    public:  
        void area() //派生类中重定义  
        { cout<<"Rect::area"<<endl; }  
};
```

```
void main() {  
    Shape *p, s;  
    Circle c;  
    Rectangle r;  
    p = &s;  
    p->area(); // Shape::area  
    p = &c;  
    p->area(); // Circle::area  
    p = &r;  
    p->area(); // Rect::area  
}
```



例5.2 虚函数的作用机制



```
// GIS图形面积计算
#define PI 3.14159
class Shape {
public:
    virtual float area{return 0;};
};
class Circle : public Shape {
public:
    float r;
    float area(){return PI*r*r;};
};
class Rectangle : public Shape {
public:
    float a, b;
    float area(){return a*b;};
};

void main() {
    Rectangle rect;
    Shape *pRect, *pCircle;
    pRect = &rect;
    pRect->area();
    pCircle = new Circle;
    pCircle->area();
}
```

例5.2 虚函数的作用机制

编译过程 (以 pRect->area() 为例)

- 确定 pRect 的类型，如：Shape*
- 在Shape类中，寻找名字为area，且参数可以匹配的函数。
- 若找不到，编译错误。
- 若找到，该函数为Virtual函数吗？
- 若不是，编译成pRect->Shape::area()
- 若是虚函数，采用动态绑定，根据pRect所指对象的vptr（虚函数表）获得对应匹配的虚函数地址，执行函数。
如：
根据pRect->vptr找到
Rectangle::area()，执行矩形面积计算函数。

```
// GIS图形面积计算
#define PI 3.14159
class Shape {
public:
    virtual float area{return 0;};
};
class Circle : public Shape {
public:
    float r;
    float area(){return PI*r*r;};
};
class Rectangle : public Shape {
public:
    float a, b;
    float area(){return a*b;};
};
void main() {
    Rectangle rect;
    Shape *pRect, *pCircle;
    pRect = &rect;
    pRect->area();
    pCircle = new Circle;
    pCircle->area();
}
```


5.2 虚函数

5.2.1 虚函数的定义

使用虚函数需要注意：

(1) 通过定义虚函数使用多态性机制时，派生类应该从它的基类公有派生。

```
class Shape {  
public:  
    virtual float area();  
    //虚函数  
};
```

```
Shape *p=new Circle;  
P->area(); // 错误
```

```
class Circle : public Shape {  
    public:  
        float area(); // 虚函数  
};
```

```
class Circle : private Shape {  
    public:  
        float area(); // 虚函数  
};
```

对吗?

(X)

□ 若采取私有继承，则虚函数为派生类的私有成员无法在类外调用

5.2 虚函数

5.2.1 虚函数的定义

使用虚函数需要注意：

- (2) 一个虚函数无论被公有继承多少次，它仍然保持其虚函数的特性。
- (3) 虚函数必须是其所在类的成员函数，不能是友元函数，也不能是静态成员函数，但虚函数可以在另一个类中被声明为友元函数。
- (4) 内联函数不能是虚函数，即使虚函数在类的内部定义，编译时仍将其看成是非内联的。
- (5) 构造函数不能是虚函数。
- (6) 析构函数可以是虚函数，而且通常声明为虚函数。

5.2 虚函数

5.2.1 虚函数与重载

- 虚函数与重载函数的关系
 - 函数重载仅是要求函数名相同
 - 重载虚函数要求与基类中原型完全相同
- 在虚函数中，若
 - a. 仅仅返回类型不同，其余均相同，系统按出错处理
 - b. 仅函数名相同，函数原型不同，则认为是一般的函数重载

5.2 虚函数

5.2.1 重载、覆盖与遮蔽

- 重载

重载函数和虚函数的区别在于前者是编译期绑定的，后者是运行期绑定

```
class C
{
public:
    C() { /*...*/ }
    C(int x) { /*...*/ }
};
```

```
void f(double d) { /*...*/ }
void f(char c) { /*...*/ }
int main( )
{
    C c1;
    C c2( 26 );
    f( 3.14 );
    f( 'Z' );
    //...
}
```

5.2 虚函数

5.2.1 重载、覆盖与遮蔽

- 覆盖

```
#include <iostream>
using namespace std;
class B{
public:
    void m() { cout << "B::m"
<<endl;}
};
class D : public B {
public:
    void m() { cout << "D::m"
<<endl;}
};
```

```
int main( )
{
    B* p;    //pointer to base
    class
    p = new D; //create a D
    object
    p->m();    //invoke m = p-
    >B::m();
    return 0;
}
```

输出结果： B::m

5.2 虚函数

5.2.1 重载、覆盖与遮蔽

- 遮蔽 <注意>

```
class B{
public:
    void m(int x) { cout << x <<endl;}
};
class D : public B{
public:
    void m() { cout << "Hi" <<endl;}
};
int main( ){
    D d1;
    d1.m();
    d1.m( 26 ); //d1.B::m( 26 );
    return 0;
}
```

```
class B {
public:
    virtual void m(int x)
        { cout << x <<endl;}
};
class D : public B {
public:
    virtual void m() { cout
        << "Hi" <<endl;}
};
int main( ) {
    D d1;
    d1.m();
    d1.m( 26 ); //d1.B::m( 26 );
    return 0;
}
```

5.2 虚函数

5.2.1 名字共享

- ◆ 重载函数名的顶层函数。
- ◆ 重载构造函数。
- ◆ 非构造函数是同一个类中名字相同的成员函数。
- ◆ 继承层次中的同名函数。

5.2 虚函数

5.2.2 虚析构函数

□ 虚析构函数的声明：

```
class 基类名 {  
    public:  
        virtual ~<基类名> () { // 函数体 }  
        // ...  
};
```

例如：

```
class Base {  
    public:    virtual ~Base() { }; //虚析构函数  
};
```


5.2 虚函数

5.2.3 虚析构函数

□ 虚析构函数的作用

```
class Shape { // 基类
public:
    virtual ~Shape () { cout<<"Shape::~destructor \n"; } //虚析构函数
};
class Rectangle : public Shape { // 派生类
public:
    int w, h; // 宽度和高度
    ~Rectangle() { cout<<"Rectangle::~destructor \n";}
};
void main () {
    Shape *p = new Rectangle;
    delete p;
}
```

运行结果:

```
Rectangle::~destructor
Shape::~destructor
```

5.2 虚函数

5.2.3 虚析构函数

□ 何时需要虚析构函数？

虚析构函数是为了解决基类指针指向派生类对象，并用基类指针delete派生类对象，导致的内存泄漏问题。

例如：

```
class Base {  
    public: virtual ~Base () {}  
};  
  
class Subclass : public Base {  
    public:  
        int* p;  
        Subclass(int n){p = new int[n];}  
        ~Subclass() { delete p; }  
};
```

```
void main() {  
    Base *pb = new Subclass(5);  
    delete pb;  
    return;  
}
```

运行结果： ???

- Subclass:: ~Subclass()
- Base:: ~Base()

5.2 虚函数

5.2.3 纯虚函数

纯虚函数是一个在基类中说明的虚函数，它在该基类中没有定义，但要求在它的派生类中必须定义自己的版本，或重新说明为纯虚函数。

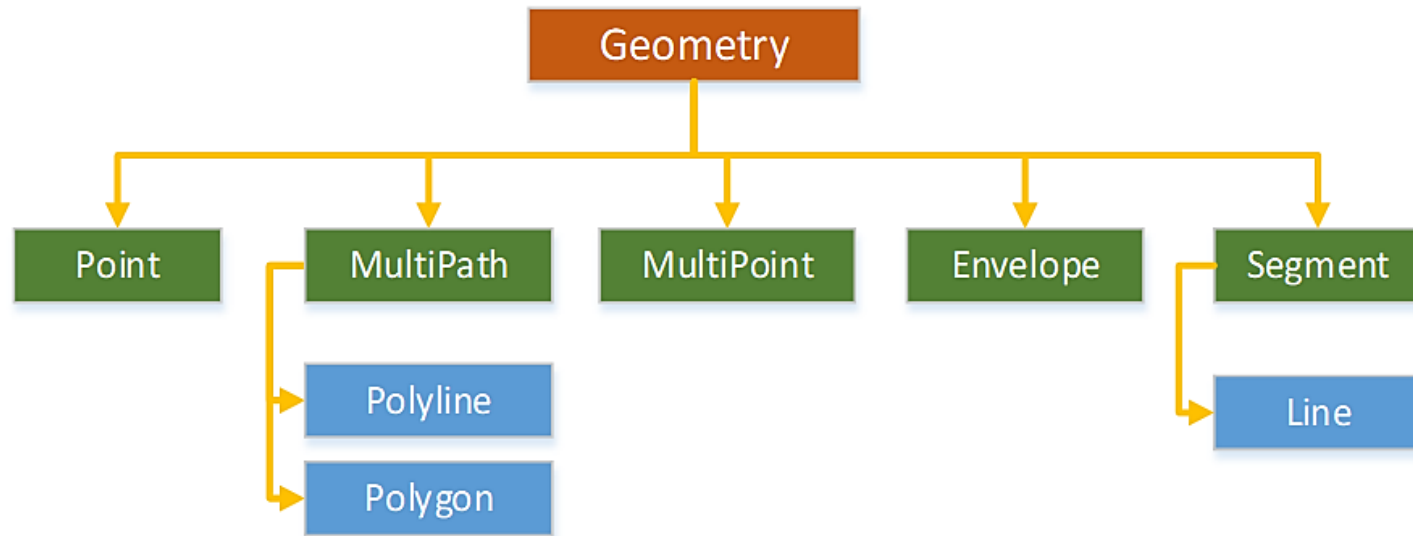
□ 纯虚函数的定义形式如下：

```
class <类名> {  
    public:  
        virtual 返回值类型 函数名(参数表) = 0 ;  
        // ...纯虚函数  
};
```

5.2 虚函数

5.2.3 纯虚函数

□ GIS中空间对象的基类



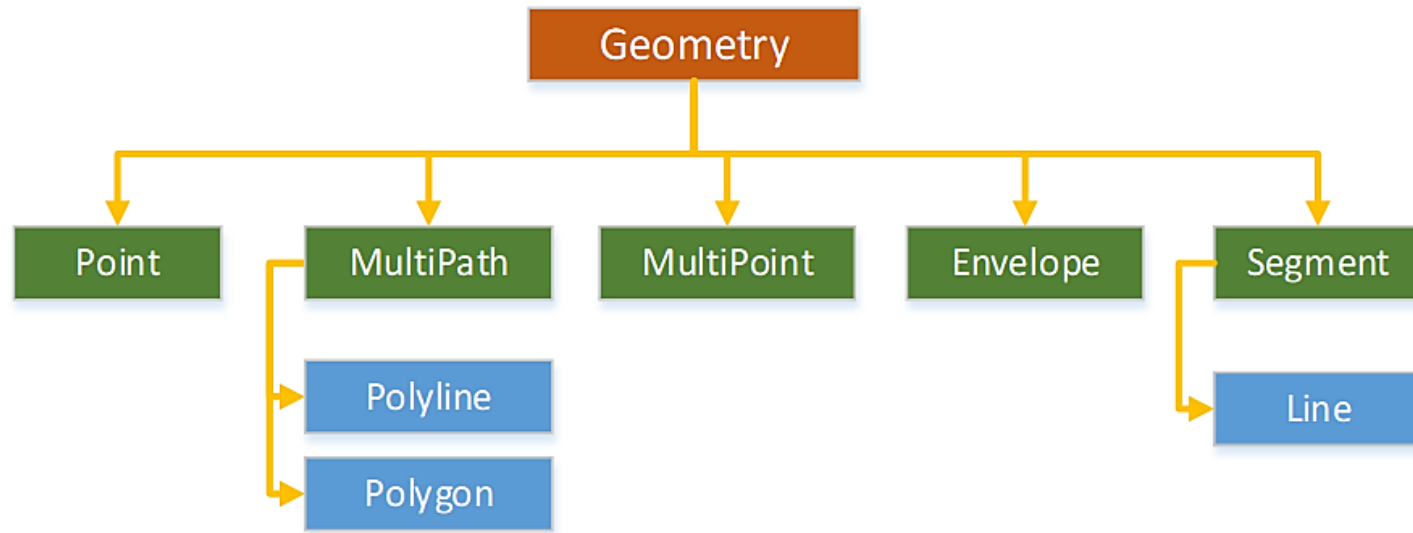
Geometry类:

- 几何类型 (geometry): 点、面、折线或多点。
- 空间参考 (spatial_reference): Beijing54、CGCS2000、WGS-84
- 中心点 (centroid): 返回标注点

5.2 虚函数

5.2.3 纯虚函数

□ GIS中空间对象的基类



```
class Shape {  
public:  
    virtual float area()=0;  
    // 纯虚函数  
};
```



```
class Circle : public Shape {  
public:  
    float r;  
    float area() { return PI*r*r; }  
};
```

例5.3：纯虚函数的使用

```
class Circle {  
public:  
    void setR(int x){ r=x; }  
    virtual void show() = 0; // 纯虚函数  
protected:  
    int r; };  
  
class Area : public Circle {  
public:  
    void show() {  
        cout<<"Area is "<<3.14*r*r<<endl;}  
}; // 派生类中实现show()  
  
class Perimeter : public Circle{  
public:  
    void show() {cout<<"Perimeter is "<<2*3.14*r<<endl;}  
}; // 派生类中实现show()
```

```
void main()  
{  
    Circle *ptr;  
    Area ob1;  
    Perimeter ob2;  
    ob1.setr(10);  
    ob2.setr(10);  
    ptr=&ob1; ptr->show();  
    ptr=&ob2; ptr->show();  
}
```

运行结果:

```
Area is 314  
Perimeter is 62.8
```

5.2 虚函数

5.2.4 抽象类

如果一个类至少有一个纯虚函数，那么就称该类为抽象类。

□ 抽象类有以下几点说明：

```
class <抽象类的类名> {  
    public:  
        virtual 返回值类型 函数名(参数表) = 0 ;  
        // ...纯虚函数  
};
```

5.2 虚函数

5.2.4 抽象类

□ 抽象类有以下几点说明

(1) 抽象类只能作为其他类的基类来使用，**不能建立抽象类对象**，其纯虚函数的实现由派生类给出。

(2) 不能从具体类(不包含纯虚函数的普通类)派生出抽象类。

(3) 抽象类**不能用作参数类型**，**函数返回类型或显式转换的类型**。

(4) 可以声明**指向抽象类的指针或引用**。指针可以指向它的派生类，从而可以实现多态性。

5.2 虚函数

5.2.4 抽象类

□ 抽象类的定义及使用示例

```
class Shape {
```

```
public:
```

```
    virtual void roateShape (int) = 0;
```

```
    // ... ... 抽象类
```

```
};
```

// 以下定义对吗?

Shape s1; ()

Shape f(); ()

Shape g(Shape s); ()

Shape *ptr; ()

Shape& h(Shape &); ()

5.3 运算符重载

本节主要内容:

- 运算符重载的定义
- 成员运算符重载函数
- 友元运算符重载函数
- 常见运算符的重载（如：**=**、**[]**、**()**、**<<**、**>>**）

5.3 运算符重载

5.3.1 运算符重载的作用

C++中对于基本数据类型的常用运算符：

(1) **双目算术运算符**： +、-、*、/、%

```
int i=2, j=5;
```

```
int z = i + j;
```

(2) **关系运算符**： ==、>、<、>=、<=、!=

```
double x=2.3, y=5.4;
```

```
x > y
```

(3) **赋值运算符**： =、+=、-=、*=、/=、%=

```
char a='c', b='g'; b=a;
```

(4) **其他运算符**： [] (下标)、 () (括号)、 , (逗号)、 -> (成员访问)

```
int a[100];
```

```
a[3] = 5;
```

5.3 运算符重载

5.3.1 运算符重载的作用

定义一个简单的复数类Complex:

```
class Complex {  
public:  
    double real, imag;  
    Complex(double r=0, double i=0)  
    { real=r; imag=i; }  
};
```

若要把类Complex的两个对象com1和com2加在一起, 下面的语句是不能实现的:

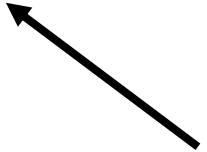
```
int main() {  
    complex com1(1.1,2.2), com2(3.3,4.4), total;  
    total=com1+com2; // 错误!!! (基本运算符无法使用)  
    return 0; }
```

5.3 运算符重载

5.3.2 运算符重载的定义

运算符重载的定义：

```
类型名 operator@ (参数表)
{
    // 函数体
}
```



(要重载的运算符)

在运算符重载的实现过程中，首先把指定的运算表达式转化为对运算符函数的调用，运算对象转化为运算符函数的实参，然后，根据实参的类型来确定需要调用的函数。这个过程是在编译过程中完成的。

5.3 运算符重载

5.3.3 运算符重载的规则

运算符是在C++系统内部定义的，它们具有特定的语法规则，如参数说明、运算顺序、优先级别等。因此，运算符重载时必须遵守一定的规则：

1. 重载的运算符应当与原有的功能类似；
2. 只能重载原来已经有定义的运算符，不能臆造新的运算符；
3. C++中的以下5个运算符不能重载：类属关系运算符“.”、作用域分辨符“::”、成员指针运算符“*”、sizeof运算符和三目运算符“?:”；
4. 不能改变运算符的操作数个数，即单目运算符只能重载为单目运算符，双目运算符只能重载为双目运算符；
5. 重载之后运算符的优先级不能改变，要改变只能通过（ ）进行；

5.3 运算符重载

5.3.3 运算符重载的规则

运算符是在C++系统内部定义的，它们具有特定的语法规则，如参数说明、运算顺序、优先级别等。因此，运算符重载时必须遵守一定的规则：

6. 重载之后运算符的结合性不能改变；

如： $a / b * c = (a / b) * c$ // 左结合

7. 不能改变运算符对预定义类型数据的操作方式。经重载的运算符，其操作数中至少应该有一个是自定义类型；

8. 重载运算符含义必须清楚，不能有二义性。

5.3 运算符重载

5.3.4 运算符重载函数的两种形式

□ **重载为类成员函数**。（称为成员运算符函数）

在C++中，可以把运算符函数定义成某个类的成员函数，称为**成员运算符函数**。

□ **重载为非成员函数**（称为友元运算符函数）

在C++中，可以把运算符重载函数定义成某个类的友元函数，称为**友元运算符函数**。

• 扩展阅读：<https://www.runoob.com/cplusplus/cpp-overloading.html>

5.3 运算符重载

5.3.4 运算符重载函数的两种形式

□ 成员运算符重载函数的定义

成员运算符函数的原型在类的内部声明格式如下:

```
class X {  
    //...  
    返回类型 operator 运算符 (形参表);  
};
```

在类外定义成员运算符函数的格式如下:

```
返回类型 X::operator 运算符 (形参表) {  
    // 函数体  
}
```

5.3 运算符重载

5.3.4 运算符重载函数的两种形式

□ 成员运算符重载函数——双目运算符重载

对双目运算符而言，成员运算符函数的形参表中仅有一个参数，它作为运算符的右操作数，此时当前对象作为运算符的左操作数，它是通过this指针隐含地传递给函数的。如：

```
class X {  
    // ... ..  
    int operator + (X a);  
};
```

例子： 重载函数进行复数运算

```
class Complex {  
    private:  
        double real, imag;  
    public:  
        Complex(double r=0.0, double i=0.0);  
        Complex operator+(Complex& b);  
        Complex operator*(Complex& b);  
        void display();  
};
```



```
Complex::Complex (double r=0, double i=0)  
{ real=r; imag=i;};
```

```
Complex Complex::operator+(Complex& b){  
    complex temp;  
    temp.real=real+b.real;  
    temp.imag=imag+b.imag;  
    return temp;  
}
```

```
Complex complex::operator*(Complex& b)  
{ //add your code here }
```

```
void Complex::display() {  
    cout<<real;  
    if(imag>0) cout<<"+";  
    if(imag !=0) cout<<imag<<"i"<<endl;  
}
```

□ **使用运算符重载:**

```
Complex A1(2.3, 4.6), A2(3.6, 2.8), A3, A4;  
A3=A1+A2;      A4=A1*A2;
```


5.3 运算符重载

5.3.4 运算符重载函数的两种形式

□ 成员运算符重载函数——单目运算符重载

对单目运算符而言，成员运算符函数的参数表中没有参数，此时当前对象作为运算符的一个操作数。

```
class coord {  
    public:  
        coord (int i=0,int j=0) {x=i; y=j;}  
        void print(){  
            cout<<" x: "<<x<<" y: "<<y;}  
        coord operator ++(); //前置  
    private:  
        int x, y;  
};
```

```
coord coord::operator++()  
{  
    ++x;  
    ++y;  
    return *this; }  
void main () {  
    coord ob(11,22);  
    ob.operator++();  
    ++ob; }
```

5.3 运算符重载

5.3.4 运算符重载函数的两种形式

□ 成员运算符重载函数——单目运算符重载

一般而言，采用成员函数重载单目运算符时，以下两种方法是等价的：

@aa; // 隐式调用：++ob

aa.operator@(); // 显式调用：ob.operator++()

成员运算符函数operator @所需的一个操作数由对象aa通过this指针隐含地传递。因此，在它的参数表中没有参数。

5.3 运算符重载

5.3.4 运算符重载函数的两种形式

□ 友元运算符重载函数的定义

友元运算符重载函数在类的内部声明格式如下：

```
class X {  
    friend 返回类型 operator 运算符 (形参表);  
    //...  
};
```

在类外定义友元运算符函数的格式如下：

```
返回类型 operator 运算符 (形参表) {  
    // 函数体  
}
```


5.3 运算符重载

5.3.4 运算符重载函数的两种形式

□ 友元运算符重载函数——双目运算符重载

当用友元函数重载双目运算符时，两个操作数都要传递给运算符函数，且至少应有一个自定义类型的形参。

一般而言，如果在类X中采用友元函数重载双目运算符@，而aa和bb是类X的两个对象，则以下两种函数调用方法是等价的：

```
aa @ bb;           // 隐式调用  
operator @(aa,bb); // 显式调用
```

例子：复数运算（友元函数）

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

```
class Complex {
```

```
private:
```

```
    double  real, imag;
```

```
public:
```

```
    Complex(double r=0.0, double i=0.0);
```

```
    friend Complex operator+(Complex& a, Complex& b);
```

```
    friend Complex operator*(Complex& a, Complex& b);
```

```
    void display();
```

```
};
```

```
Complex::Complex (double r, double i)
```

```
{ real=r; imag=i;};
```

```
Complex operator+(Complex& a, Complex& b) {  
    Complex temp;  
    temp.real=a.real+b.real;  
    temp.imag=a.imag+b.imag;  
    return temp;  
}
```

```
void Complex::display() {  
    cout<<real;  
    if(imag>0) cout<<“+”;  
    if(imag !=0) cout<<imag<<“i”<<endl;  
}
```

□ **使用友元运算符重载函数：**

```
Complex A1(2.3, 4.6), A2(3.6, 2.8);  
Complex A3, A4;  
A3=A1+A2; A4=A1*A2;
```

5.3 运算符重载

5.3.4 运算符重载函数的两种形式

□ 友元运算符重载函数——双目运算符重载

重载运算符函数的两种返回方式:

1) 返回正式类对象

```
Complex operator+ (Complex a, Complex b) {  
    Complex temp;  
    temp.real=a.real+b.real; temp.imag=a.imag+b.imag;  
    return temp;  
}
```

2) 返回无名临时对象

```
Complex operator+ (Complex a, Complex b) {  
    return Complex(a.real+b.real, a.imag+b.imag);  
}
```

重载运算符函数的两种返回方式比较

1) 返回正式的对象

创建一个局部对象temp(这时会调用构造函数), 执行return语句时, 会调用拷贝构造函数, 将temp的值拷贝到主调函数中的一个无名临时对象中。当函数operator+结束时, 会调用析构函数析构对象temp, 然后temp消亡。

2) 返回无名临时对象

表面上看起来像是对构造函数的调用, 但其实并非如此。这是临时对象语法, 它的含义是创建一个临时对象并返回它。是直接将一个无名临时对象创建到主调函数中。因而执行效率高。

5.3 运算符重载

5.3.4 运算符重载函数的两种形式

□ 友元运算符重载函数——单目运算符重载

友元函数重载单目运算符时，需要一个显式的操作数。
如：

```
class X{ friend X operator ++(X& obj); };
```

一般而言，如果在类X中采用友元函数重载单目运算符@，而aa是类X的对象，则以下两种函数调用方法是等价的：

@aa;	// 隐式调用
operator@(aa);	// 显式调用

5.3 运算符重载

说 明

1. 运算符重载函数operator@()可以返回任何类型，包括void类型。但一般返回与操作数相同的类型。
2. 一般在重载运算符时保持C++中该运算符原有的含义。
3. 在C++中，用户不能定义新的运算符。
4. 重载运算符实际上就是重载函数。因此，可以为同一个运算符定义多个重载函数来进行不同的操作。
5. 不能使用友元函数重载的运算符有：
= () [] ->

5.3 运算符重载

5.3.4 运算符重载函数的两种形式

□ 成员运算符函数与友元运算符重载函数的比较

(1) 对双目运算符而言, 成员运算符函数带有一个参数, 而友元运算符函数带有两个参数; 对单目运算符而言, 成员运算符函数不带参数, 而友元运算符函数带一个参数。

(2) 双目运算符一般可被重载为友元运算符函数或成员运算符函数, 但有一种情况必须使用友元函数。

```
AB AB::operator+ (int x)
{
    AB temp;
    temp.a=this.a+x;
    temp.b=this.b+x;
    return temp;
}
```

```
AB ob
ob=ob+200;
ob=200+ob;//对吗?
```


5.3 运算符重载


5.3.4 运算符重载函数的两种形式

□ 成员运算符函数与友元运算符重载函数的比较

(3) C++大部分运算符既可说明为成员运算符函数，又可说明为友元运算符函数。究竟选择哪一种运算符好一些，没有定论，这主要取决于实际情况和程序员的习惯。以下经验可供参考：

- 单目运算选择成员函数
- 对于 `=` `()` `[]` `->` 只能使用成员函数
- 对于 `+=` `-=` `/=` `*=` `&=` `!=` `~=` `%=` `>>=` `<<=` 建议重载为成员函数
- 对于其它运算符，建议重载为友元函数

第五章：多态性和运算符重载

- 多态性概述
- 虚函数
- 纯虚函数和抽象类
- 运算符重载
- 常见运算符的重载 
- 类型转换与模板
- 本章小结

5.3 常见运算符重载

5.5.1 赋值运算符 “=” 的重载

对任一类X, 如果没有用户自定义的赋值运算符函数, 那么系统自动地为其生成一个缺省的赋值运算符函数, 定义为类X中的成员到成员的赋值, 例如:

```
X &X::operator=(const X& source)  
{  
  
//...成员间赋值  
  
}
```

若obj1和obj2是类X的两个对象, 则编译程序遇到如下语句:

```
obj1=obj2;
```

就调用缺省的赋值运算符函数, 将对象obj2的数据成员的值逐个赋给对象obj1的对应数据成员中。

例：使用缺省的赋值运算符(浅拷贝)产生错误的例子。

```
#include <iostream.h>
class STRING {
public:
    STRING(char *s) {
        cout<< "Constructor called." <<endl;
        ptr=new char[strlen(s)+1];
        strcpy(ptr,s);
    }
    ~STRING () {
        cout<<"Destructor called---"<<ptr<<endl;
        delete[] ptr; }
private:
    char *ptr;
};
```

```
int main()
{
    STRING p1("book");
    STRING p2("jeep");
    p1=p2;
    return 0
}
```

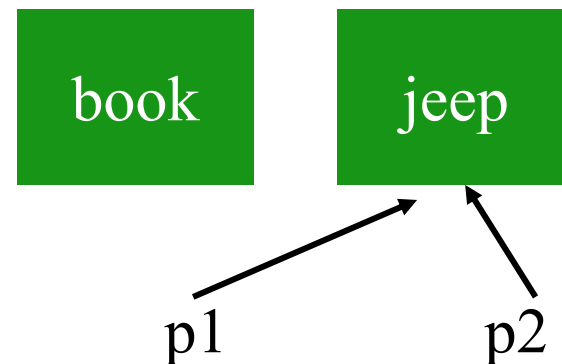
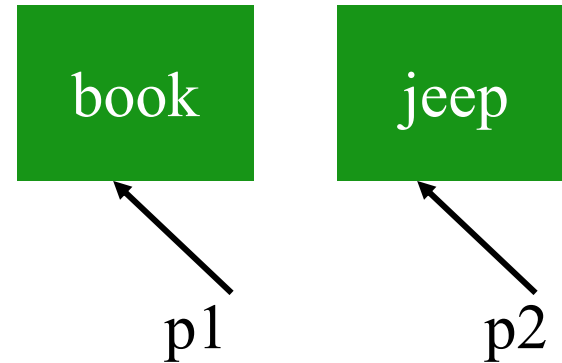
运行结果：

Constructor called.

Constructor called.

Destructor called---book

Destructor called---葺葺葺葺



例(修正): 使用缺省的赋值运算符(浅拷贝)产生错误的例子。

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

```
class STRING {
```

```
public:
```

```
    STRING(char *s) {
```

```
        cout<< "Constructor called." <<endl;
```

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

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

```
    }
```

```
    ~STRING () {
```

```
        cout<<"Destructor called---"<<ptr<<endl;
```

```
        delete[] ptr; }
```

```
    STRING& operator=(const STRING &); //声明重载
```

```
private: char *ptr;
```

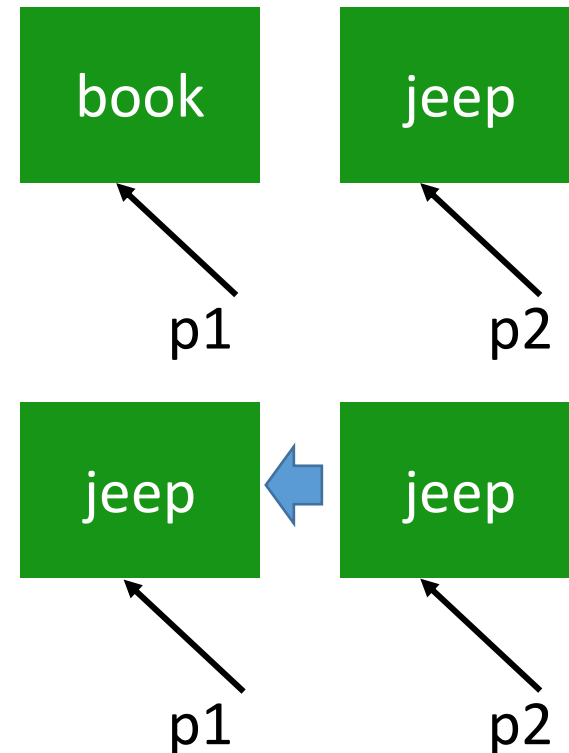
```
};
```

STRING & STRING::operator=(const STRING& s)

```
{ if (this==&s) return *this;           // 防止s=s的赋值
  delete[] ptr;                          // 释放掉原区域
  ptr=new char[strlen(s.ptr)+1];         // 分配新区域
  strcpy(ptr, s.ptr);                    // 字符串拷贝
  return *this;
}
```

int main()

```
{
  STRING p1("book");
  STRING p2("jeep");
  p1=p2;
  return 0
}
```



5.5 常见运算符重载

5.5.1 赋值运算符 “=” 的重载

说明：

1. 类的赋值运算符 = 只能重载为成员函数，而不能重载为友元函数。
2. 类的赋值运算符 = 可以被重载，但重载后的运算符函数 `operator=()` 不能被继承。

5.5 常见运算符重载

5.5.2 下标运算符 “[]” 的重载

在C++中，在重载下标运算符[]时认为它是一个双目运算符。
即对于X[Y]，C++是这样理解的：

[] 双目运算符
X 左操作数
Y 右操作数

对下标运算符重载定义只能使用成员函数：

```
返回类型 类名::operator[](一个形参)
{
    // 函数体
}
```

例子：下标运算符重载例子

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

```
class Vector4 {
```

```
public:
```

```
    Vector4(int a1,int a2, int a3,int a4)
```

```
    { v[0]=a1;v[1]=a2;v[2]=a3;v[3]=a4;}
```

```
    int& operator[] (int bi);
```

```
private:
```

```
    int v[4];
```

```
};
```

```
int& Vector4::operator[](int bi) {
```

```
    if (bi<0||bi>=4) {
```

```
        cout<<"Bad subscript!\n";
```

```
        exit(1);
```

```
    }
```

```
    return v[bi]; }
```

```
int main ( )
```

```
{
```

```
    Vector4 v(1,2,3,4);
```

```
    cout<<v[2]<<endl;
```

```
    v[3]=v[2];
```

```
    cout<<v[3]<<endl;
```

```
    v[2]=22;
```

```
    cout<<v[2]<<endl;
```

```
    return 0;
```

```
}
```

5.5 常见运算符重载

5.5.2 下标运算符 “[]” 的重载

说明:

1. 重载下标运算符[]的一个优点是增加C++数组检索的安全性。
2. 重载下标运算符[]时, 返回一个int 的引用, 可以使重载的下标运算符[]用在赋值语句的左边。

5.5 常见运算符重载

5.5.3 函数调用运算符“()”的重载

在C++中，在重载函数调用运算符()时认为它是一个双目运算符。即对于X(Y)，C++是这样理解的：

() 双目运算符

X 左操作数

Y 右操作数

对函数调用运算符重载定义只能使用成员函数：

返回类型 类名::operator()(形参表)

{ // 函数体

}

例： 声明一个矩阵类，重载函数运算符()

```
class Matrix {
```

```
    public:
```

```
        Matrix (int, int);
```

```
        int& operator( ) (int, int);
```

```
    private:
```

```
        int *m;
```

```
        int row, col;
```

```
};
```

```
Matrix ::Matrix (int row, int col){...}
```

```
int& Matrix::operator()(int r, int c)
```

```
{
```

```
    return (*(m+r*col+c);
```

```
}
```

```
void main ( ) {
```

```
    Matrix a(10,10);
```

```
    cout << a(3, 4);
```

```
    a(3,4)=35;
```

```
    // ... .. }
```

哪个是左操作数？

哪个是右操作数？

5.5 常见运算符重载

5.5.4 输出运算符 “<<” 的重载

定义输出运算符 “<<” 重载函数的格式如下：

```
class 类名 {  
    friend ostream& operator<<(ostream& out, 类名& obj)  
    {  
        out<<obj.成员变量;  
        ...  
        return out;  
    }  
};
```

例：复数类重载运算符 “<<”

```
class Complex {  
    public:  
        Complex(double r=0.0, double i=0.0);  
        friend ostream& operator<<(ostream&out, Complex& c);  
    private:  
        double  real, imag  
};  
Complex::Complex (double r, double i) { real=r; imag=i;};  
ostream& operator<<(ostream& out, Complex& c)  
{  
    out<<com.real;  
    if(c.imag>0) out<<“+”;  
    if(c.imag !=0) out<<c.imag<<“i”<<endl;  
    return out;  
}
```

例：复数类重载运算符 “<<”

```
class complex {  
    public:  
        complex(double r=0.0, double i=0.0);  
        friend ostream& operator<<(ostream& out, complex& com);  
    private:  
        double real, imag;  
};  
complex::complex (double r, double i):  
{ real=r; imag=i;};  
ostream& operator<<(ostream& out, complex& com) {  
    out<<com.real;  
    if(com.imag>0) out<<“+”;  
    if(com.imag !=0) out<<com.imag<<“i”<<endl;  
    return out;  
}
```

```
int main() {  
    complex A1(2.3,4.6);  
    cout<<A1;  
    return 0;  
}
```

运行结果:
2.3+4.6i

5.5 常见运算符重载

5.5.5 输入运算符“>>”的重载

定义输出运算符“>>”重载函数的格式如下：

```
class 类名{  
    friend ostream& operator>>(ostream& in, 类名& obj)  
    {  
        in>>obj.成员变量;  
        ...  
        return in;  
    }  
};
```

第五章：多态性和运算符重载

- 多态性概述
- 虚函数
- 纯虚函数和抽象类
- 运算符重载
- 常见运算符的重载
- 类型转换与模板
- 本章小结



5.6 类型转换与模板

5.6.1 类型转换

□ 系统预定义类型间的转换

1. 隐式类型转换规则

- 在赋值表达式A=B的情况下, B的值需要转换为A的类型后进行赋值.
- 当char 或short型与int 型进行运算时, 将char 或short型转换为 int 型
- 当两个操作数类型不一致时, 在算术运算前低级别的自动转换为高级别类型

5.6 类型转换与模板

5.6.1 类型转换

□ 系统预定义类型间的转换

2. 显式类型转换方法

-- 强制转换法：（类型名）表达式

如： `double i=2.2, j=3.2;`

`cout<<(int) (i+j);`

-- 函数法：类型名（表达式）

如： `double i=2.2, j=3.2;`

`cout<<int (i+j);`

5.6 类型转换与模板

5.6.1 类型转换

□ 类类型与系统预定义类型间的转换

1. 通过构造函数进行类型转换

为了用构造函数完成类型转换，类内必须至少定义一个只带有一个参数（或者其它参数都带有缺省值）的构造函数。当需要类型转换时，系统自动调用该构造函数数据，创造该类的一个临时对象，该对象由被转换的值初始化，从而实现了类型转换。

例：通过构造函数进行类型转换

```
class A {  
    public:  
        A( int n);  
        void print( );  
    private:  
        int num;  
};  
A::A(int n)  
{  
    num=n;  
    cout<<"Initializing with:"  
    cout<<num<<endl;  
}
```

```
void A::print() {  
    cout<<"num="<<num;  
    cout<<endl;  
}  
  
void main ( ) {  
    A a=A(3);  
    a.print();  
    A b=6; // 调用构造函数  
    b.print();  
    b=8;  // 调用构造函数  
    b.print();  
}
```

5.6 类型转换与模板

5.6.1 类型转换

□ 类类型与系统预定义类型间的转换

2. 通过类类型转换函数进行类型转换

通过构造函数进行类型转换, 只能从基本类型向自定义类型转换, 而不能将自定义类型的数据转换为基本类型的数据, 但类型转换函数则可以进行这种转换。

类类型转换函数定义的一般格式:

```
class 源类类名 {  
    operator 目的类型() {  
        //函数体  
        return 目的类型的数据;  
    }  
    ... ..  
};
```

类类型转换函数的一般使用方法:

```
X obj;  
cout<<int (obj);
```


例：通过类类型转换函数进行显式类型转换

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

```
class complex {
```

```
    public:
```

```
        complex(double r=0,double i=0)
```

```
        { real=r; imag=i; }
```

```
        operator double( )
```

```
        { return real; }
```

```
        operator int( )
```

```
        { return int(real); }
```

```
        void print()
```

```
        { cout<<real<<"+"<<imag<<" i"<<endl;}
```

```
    private:
```

```
        double real, imag;
```

```
};
```

```
int main(int argc, char* argv[]) {  
    complex a(2.2, 4.4);  
    cout<<"a=";  
    a.print();  
    cout<<"double(a)=";  
    cout<<double(a)<<endl;  
    complex b(4,6);  
    cout<<"b=";  
    b.print();  
    cout<<"int(b)=";  
    cout<<int(b)<<endl;  
    return 0;  
}
```

```
a=2.2+4.4 i  
double(a)=2.2  
b=4+6 i  
int(b)=4  
Press any key to continue
```

5.6 类型转换与模板

5.6.1 类型转换

□ 类类型与系统预定义类型间的转换

1. 类类型转换函数只能定义为一个类的成员函数，不能定义为友元函数。
2. 类类型转换函数既没有参数，也不显式给出返回类型。
3. 类类型转换函数中必须有返回目的类型数据，即有：
return 目的类型数据。
4. 一个类可以定义多个类类型转换函数。C++将自动选择匹配最佳的类类型转换函数执行，在可能出现二义性的情况下，应显式使用类类型转换函数进行类型转换。

5.6 类型转换与模板

5.6.2 C++为什么引入模板?

□ 模板是实现多态性的一种重要方式，实现泛型编程

```
int max (int x, int y)
{ return (x>y)?x: y; }
```

```
float max (float x, float y)
{ return (x>y)?x: y; }
```

```
double max (double x, double y)
{ return (x>y)?x: y; }
```

宏定义:

```
# define max (x, y)
( (x>y)?x: y)
```

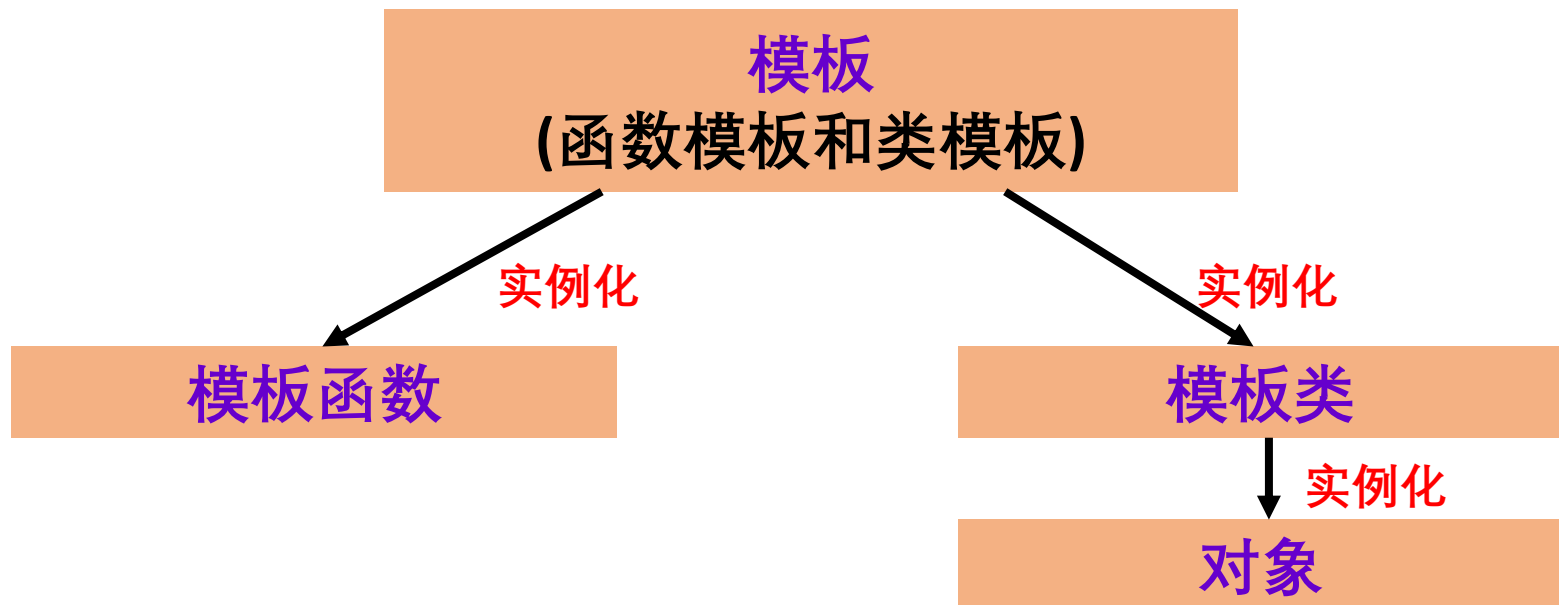
缺点：宏定义避开了C++的类型检查机制，易导致错误。

特点：函数体完全相同，参数类型不同。

5.6 类型转换与模板

5.6.3 模板的概念

模板是实现代码重用机制的一种工具，它可以实现类型参数化，即把类型定义为参数，从而实现了真正的代码重用。模板分为函数模板和类模板，它们分别允许用户构造模板函数和模板类。



5.6 类型转换与模板

5.6.4 函数模板的声明

函数模板的一般说明形式如下：

template < typename [或class] 类型参数 >

返回类型 函数名(模板形参表)

{

函数体

}

其中，template是一个声明模板的关键字，它表示声明一个模板。

例如：将求最大值函数max()定义成函数模板

```
template < typename T>
```

```
T max(T x,T y)
```

```
{
```

```
    return (x>y)?x:y;
```

```
}
```

也可以定义成如下形式：

```
template < class T>
```

```
T max(T x,T y)
```

```
{
```

```
    return (x>y)?x:y;
```

```
}
```

其中：

T为类型参数，它既可取系统预定义的数据类型,又可以取用户自定义的类型。

5.6 类型转换与模板

5.6.5 函数模板的使用

将T实例化的参数称为模板实参，用模板实参实例化的函数称为模板函数。

当编译系统发现有一个函数调用：

函数名(模板实参表);

将根据模板实参表中的类型生成一个函数即模板函数。该模板函数的函数体与函数模板的函数定义体相同。

5.6 类型转换与模板

5.6.5 函数模板的使用

1. 在template语句与函数模板定义语句之间不能有别的语句.

```
template <class T>  
int i;           // 错误  
T max(T x,T y)  
{ return (x>y)?x:y; }
```
2. 模板函数类似于重载函数, 只不过它更严格一些. 在函数重载时, 每个函数体可以执行不同的动作. 但同一函数模板实例化后的所有模板函数都必须执行相同的动作.
3. 在函数模板中允许使用多个类型参数, 每个类型参数前必须有关键字class/typename.

5.6 类型转换与模板

5.6.5 函数模板的使用

4. 同一般函数一样，函数模板也可以重载.

```
template<typename type>    template<typename type>
type min(type x, type y)    type min(type x, type y , type z)
{   return (x<y)?x:y; }      {   type t; t= (x<y)?x:y ;
                               return (t<z)?t;z ;
                               }
```

5. 函数模板与同名的非模板函数可以重载（思考：调用顺序？）

```
template<typename T>
T min(T x, T y){}
int min(int x, int y)
{   cout<<“调用非模板函数:”; return (x<y)?x:y; }
```

5.6 类型转换与模板

5.6.6 类模板与模板类

类模板允许用户为类定义一种模式，使得类中的某些数据成员，某些成员函数的参数或返回值，能取任意数据类型。

声明一个类模板与声明函数模板的格式类似，必须以关键字template开始，然后是类名，其格式如下：

```
template <typename[或class] Type>
```

```
class 类名 {
```

```
    //...
```

```
};
```

模板参数类型参数



5.6 类型转换与模板

5.6.6 类模板与模板类

类模板中的成员函数和重载的运算符函数必须为函数模板。他们可以放在类模板体内定义，也可以放在类模板的外部定义，具体定义形式如下：

```
template <typename[或class] Type>
```

```
返回值类型 类模板<类型名表>::函数名(参数表)
```

```
{
```

```
    // 函数体
```

```
};
```

5.6 类型转换与模板

5.6.6 类模板与模板类

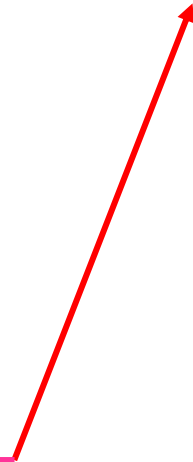
在类定义中，欲采用通用数据类型的数据成员，成员函数的参数或返回值前面需要加上Type.

```
const int size=10
template <class Type>
class stack {
public:
    void init(){ tos=0; }
    void push (Type ch);
    Type pop( );
private:
    Type stck[size];
    int tos;  };
```

5.6 类型转换与模板

5.6.6 类模板与模板类

在类体外定义有模板形参的成员函数时，需要在函数体外进行模板声明，并且在函数名前的类名后缀上 “<Type>”



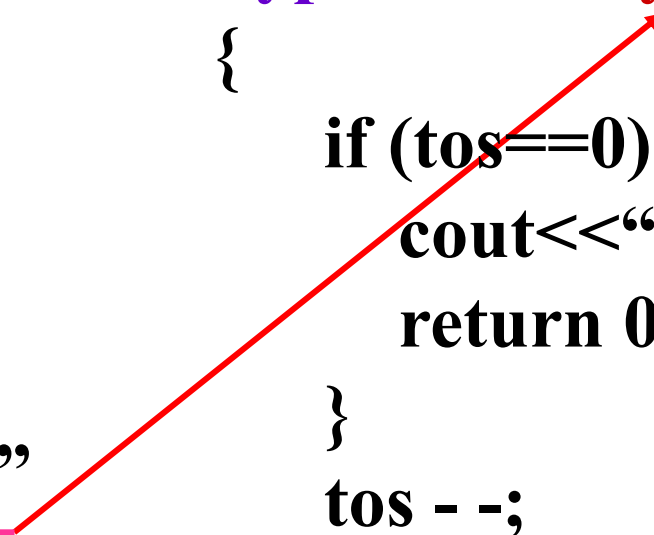
```
template <class Type>
void stack<Type>::push(Type
ob)
{
    if (tos==size) {
        cout<<“stack is full ”;
        return;
    }
    stck[tos]=ob;
    tos++;
}
```

5.6 类型转换与模板

5.6.6 类模板与模板类

在类体外定义有模板形参的成员函数时，需要在函数体外进行模板声明，并且在函数名前的类名后缀上 “<Type>”

```
template <class Type>
Type stack<Type>::pop()
{
    if (tos==0) {
        cout<<“stack is empty ”;
        return 0;
    }
    tos - -;
    return stck[tos];
}
```



5.6 类型转换与模板

5.6.6 类模板与模板类

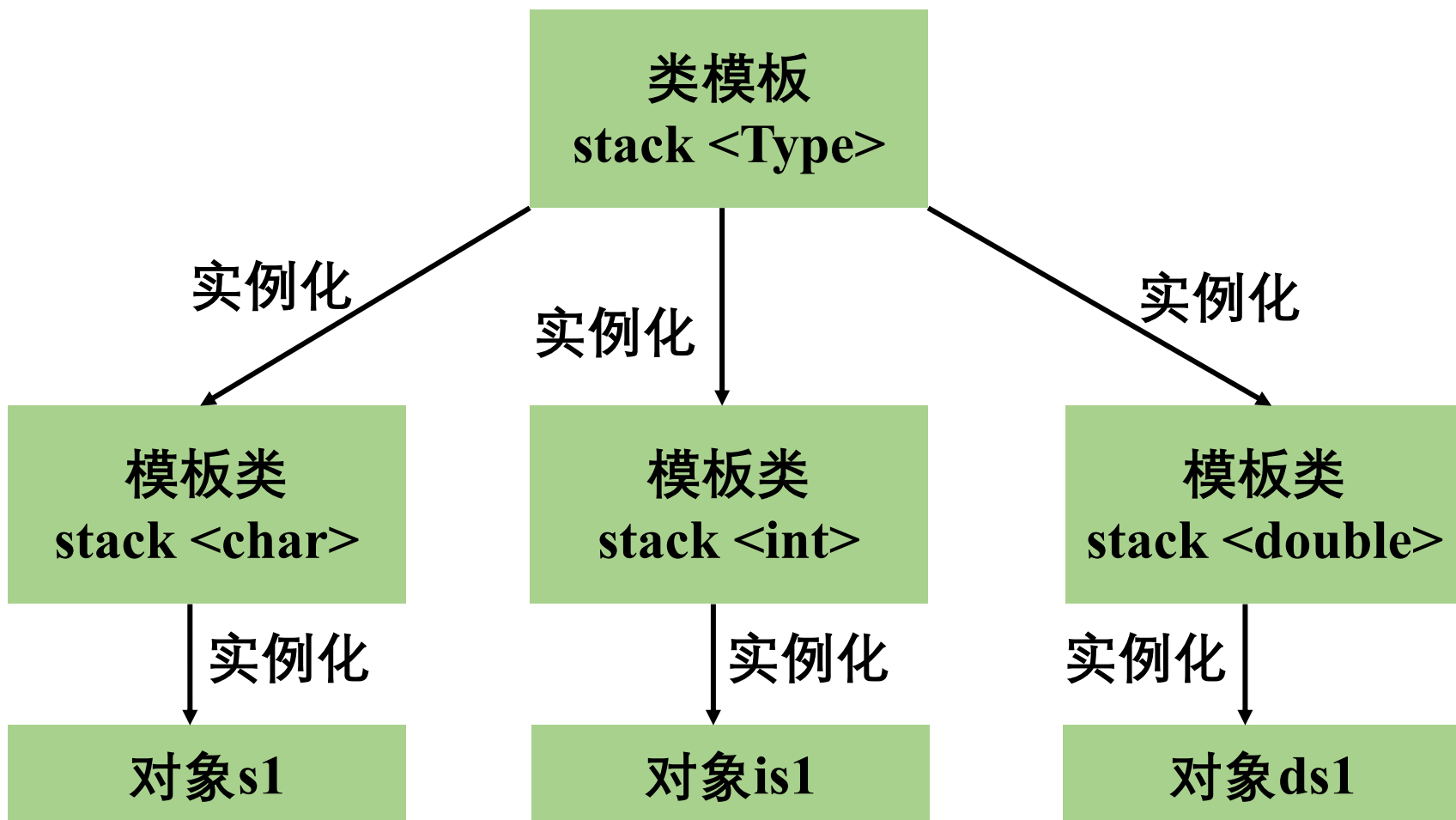
类模板不是代表一个具体的、实际的类，而是代表着一类类。实际上，类模板的使用就是将类模板实例化成一个具体的类，它的格式为：

类模板名 <实际的类型> 对象名;

例: **stack <char> s1, s2;**

5.6 类型转换与模板

5.6.6 类模板与模板类



5.6 类型转换与模板

5.6.6 类模板与模板类

//具有多个类参数的类模板

```
template< class T1, class T2, class T3 >
```

```
class Sample
```

```
{
```

```
public:
```

```
    T2 m( T3  p) { /*...*/ }
```

```
private:
```

```
    T1  x;
```

```
};
```

5.6 类型转换与模板

5.6.6 类模板与模板类

模板的函数式参数

类模板必须至少有一个类参数，可以有多个参数。类模板还可以有非类参数的参数。一般称之为函数类型参数，一个类模板可以有多个函数类型参数，这些参数的数据类型可以是内建类型或自定义数据类型。

```
template< class T1, int X, float Y >
```

```
class Sample
```

```
{
```

```
    //...
```

```
};
```

5.6 类型转换与模板

5.6.6 类模板与模板类

类模板的派生

```
template<class T>  
class base { // ... } ;
```

```
template<class T>  
class derive:public base<T> { // ... } ;
```

与一般的类派生定义相似，只是指出它的基类时要加上模板参数，如base<T>

本章小结

重点:

- 多态性的作用
- 虚函数
- 运算符重载和类模板
- 常见运算符的重载

难点:

- 纯虚函数
- 抽象类
- 类型转换