面向对象方法与C++程序设计



第5章　　多态

大连理工大学
主讲人-赵小薇

理解多态



多态性（Polymorphism）是指同样的消息被不同类型的对象接收时产生不同行为的现象。

```
Complex c1(1,2), c2(3,4), c3, c4 ;
int i  = 6 ;
c3 = c1 + c2;
c4 =  i + c1;
```

消息都是“+”，执行加法运算

不同类型的消息接收者产生不同行为

程序是客观世界的体现，在现实世界中多态现象比比皆是。

消息都是 “铃声”

学校的第一节上课铃声响起……

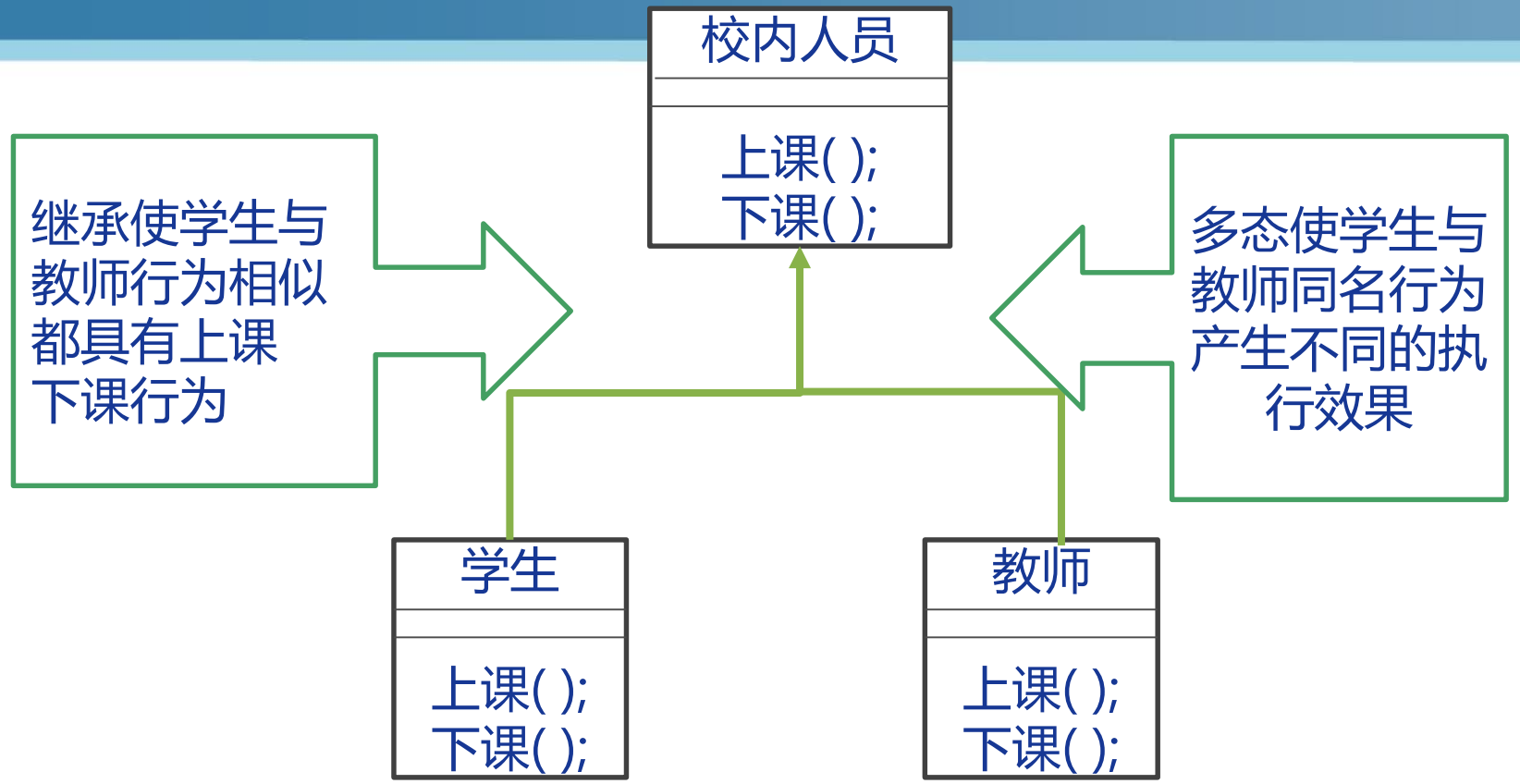
教师要走上讲台准备上课

学生在座位上做好上课准备

教务人

实验室

不同消息接收者产生不同行为



本章介绍的多态性与上一章介绍的继承性相结合，可以生成一系列虽彼此相似却又独一无二的类和对象。

多态的实现



- 多态划分为两类：**静态多态**与**动态多态**。
  - 静态多态是指在程序编译时系统就能够确定要调用的是哪个函数，因此这种多态也被称为**编译时多态**。通过函数的重载来实现。

```
class   A
int data
void get (char)
void get(int)
……
```

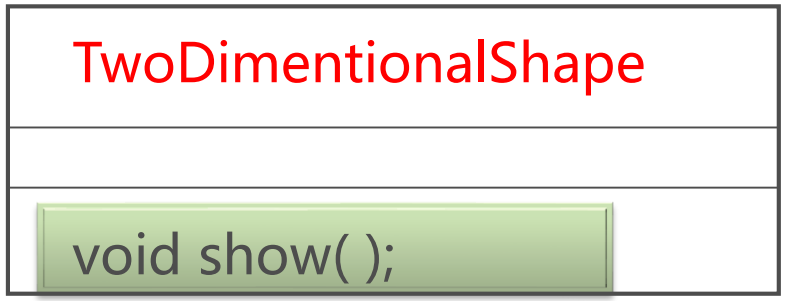
```
A  objA;
objA.get( ‘a’ );
objA.get (5);
```

动态多态

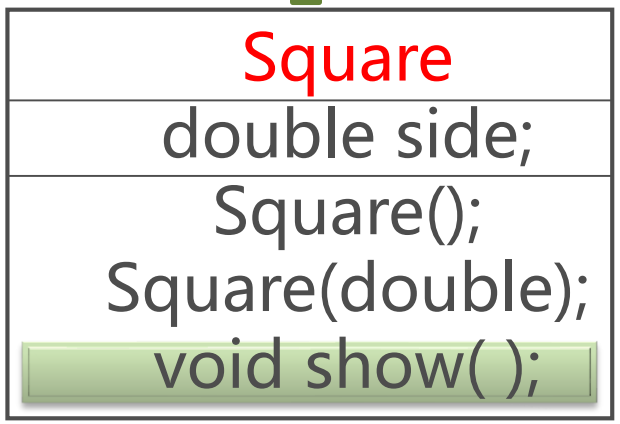


- 动态多态性是指程序在编译时并不能确定要调用的函数，直到运行时系统才能动态地确定操作所针对的具体对象，它又被称为**运行时多态**。
- 动态多态是通过虚函数（virtual function）来实现的。





继承




```
// 二维图形输出函数
void TwoDimensionalShape::show(){
  cout<<"这是二维图形"<<endl;
}
// 正方形默认构造函数
Square::Square():side(1){ }
// 正方形重载构造函数
Square::Square(double side):side(side){ }
// 正方形输出函数
void Square::show(){
 cout<<"这是边长为"<<side
        <<"的正方形" <<endl;
}
```



```
// 主函数
int main(){
TwoDimensionalShape t;                    // 创建二维图形对象t
 t.show();                                // 基类对象调用基类函数
 Square s(3);                             // 创建正方形对象s
 s.show();                                // 派生类对象调用同名函数
 TwoDimensionalShape * members[2];  //二维图形指针数组
  members[0]=&t;
  members[1]=&s;
  for(int i=0;i<2;i++)
      members[i]->show();                 //调用的是哪一个函数呢?
return 0;
}
```

程序运行结果：

这是一个二维图形
这是边长为3的正方形
这是一个二维图形
这是一个二维图形

无论指向的是哪种类型的对象，通过基类指针调用的都是基类中定义的show函数

虚函数



- C++中的虚函数的作用是允许在派生类中重新定义与基类同名的函数，并且可以通过基类指针或者基类引用来访问这个同名函数。虚函数成员声明的语法为：

```
virtual 函数类型 函数名（参数列表）；
```

注意以下两点：

1．virtual只能使用在类定义的函数原型声明中，不能在成员函数实现的时候使用，也不能用来限定类外的普通函数

2．virtual具有继承性，在派生类覆盖基类虚成员函数时，既可以使用virtual，也可以不用virtual来限定，二者没有差别，默认派生类中的重写函数是具有virtual的。



将基类TwoDimensionalShape中的函数show声明为虚函数，如下：

```
class TwoDimensionalShape {                    //二维图形类
public:
    virtual void show();                       //输出二维图形信息
};          ……
```

主函数不变，程序运行结果产生了变化：

这是一个二维图形
这是边长为3的正方形
这是一个二维图形
这是边长为3的正方形



- 虚函数来说，基类与派生类同名函数的参数列表是完全相同的，唯一不同的就是函数所属的类不同，也就是调用函数的对象不同。

```
for(int i=0;i<3;i++)
    members[i]->show();
```

- 编译器是无法知道members[i] 在运行时指向的对象是何种类型的，因此并不执行关联。
- 到了运行阶段，基类指针变量members[i]被赋值，指向了具体的对象，此时members[i]的类型是确定无疑的，再关联。
- 被称为**动态关联**（dynamic binding），由于是在编译后执行该过程，因此也被称为**滞后关联**（late binding）。

虚析构函数



- 当基类的析构函数为虚函数时，无论指针指的是同一类族的哪一个类对象，对象撤销时，系统会采用动态关联，调用相应的析构函数，完成该对象的清理工作。
- 习惯把析构函数声明为虚函数，即使基类并不需要析构函数，以确保撤销动态存储空间时能够得到正确的处理。
- 构造函数是不能声明为虚函数的。

举例



```
class TwoDimensionalShape {                          //二维图形类
public:
        TwoDimensionalShape ();                        //构造函数
        ~ TwoDimensionalShape ();                      //析构函数
};
class Square : public TwoDimensionalShape {     //正方形类
private:
        double side;                                   //正方形边长
public:
        Square();                                     //默认构造函数
        Square(double);                                //构造函数
        ~Square();                                     //析构函数
};
```



```
// 主函数
int main(){
        TwoDimensionalShape * t = new Square(9);
        delete t;
    return 0;
}
```

程序运行结果

二维图形构造函数

正方形构造函数

二维图形析构函数



- 程序用带指针参数的delete运算符来撤销对象时，因为该指针的类型是基类TwoDimensionalShape类型的，因此系统只会执行基类的析构函数，而派生类的析构函数并没有被执行。最终将导致因内存不足而引起的程序终止。
- 避免上述错误最有效的方法就是将基类的析构函数声明为虚函数。

程序改进



```
class TwoDimensionalShape {                          //二维图形类
public:
        TwoDimensionalShape ();                        //构造函数
        virtual ~ TwoDimensionalShape ();              //析构函数
};
class Square : public TwoDimensionalShape {     //正方形类
private:
        double side;                                   //正方形边长
public:
        Square();                                     //默认构造函数
        Square(double);                                //构造函数
         ~Square();                                    //析构函数
};
```