

面向对象程序设计与实践2

交换与智能控制研究中心
北京邮电大学

面向对象的程序设计

□ 程序设计

- 程序=算法+数据结构
- 程序=对象+对象间的相互作用

□ 面向对象程序设计的优势

- 提高软件质量：实现数据与方法的封装，通过方法来操作改变数据，提高了数据访问的安全性
- 易于软件维护
- 支持软件重用，大大提高软件生产的效率
- 实现可重用的软件组件，实现软件设计的产业化
- ...

C++是一种混合语言

- C++是一种混合语言，不是纯粹的面向对象语言
- 面向对象的语言
 - Smalltalk是第一个真正面向对象的语言
 - Java, C#

面向对象程序设计的关键概念

□ 抽象

- 对具体问题（对象）进行分类概括，提取出这一类对象的共同性质并且加以描述的过程。

□ 封装

- 将抽象得到的数据和行为组合起来，并且对外隐藏实现细节。

□ 继承

- 在已有类的设计基础上，进行更具体、更细化的设计。

□ 多态

- 不同的派生类对同一个操作具有不同的行为实现。

继承的概念

□ 一个类是另一个类的一种特化 (specialization)

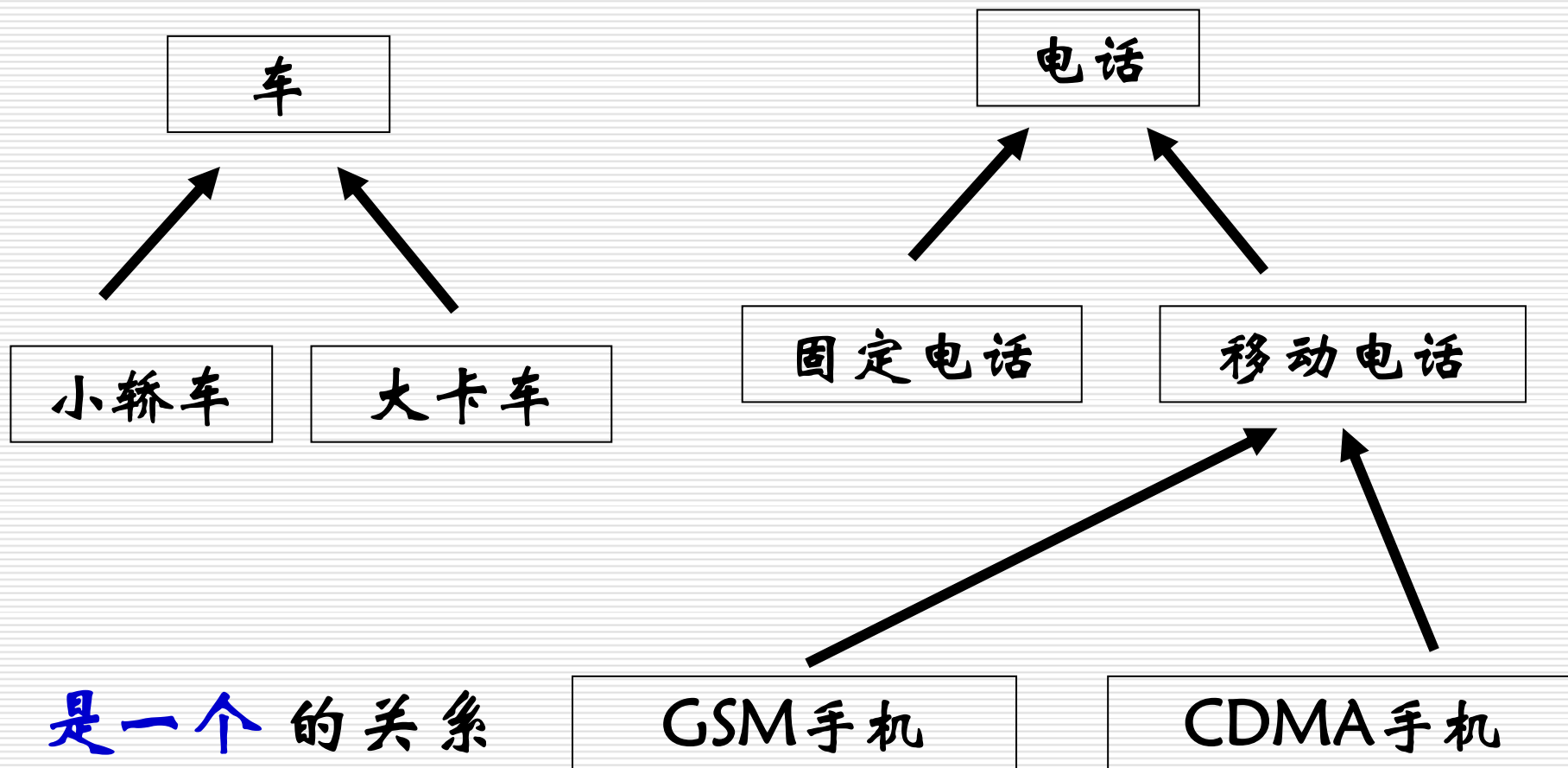
□ 目的

■ 定义能为两个或者更多的派生类提供共有元素的基类，从而写出更加精简的代码。

□ 共有元素

■ 子程序接口，内部实现，数据成员或数据类型等。避免多处出现重复的代码与数据。

继承实例



继承的方式

继承方式	基类成员的访问权限	派生类对基类成员的访问权限
public 公有继承	public private protected	public 不可见 protected
private 私有继承	public private protected	private 不可见 private
protected 保护继承	public private protected	protected 不可见 protected

继承的内容

□ 确保只继承需要继承的部分

■ 派生类可以继承成员函数的接口和/或实现

□ 三种基本

虚函数

非虚函数

	可覆盖的	不可覆盖的
提供默认实现	可覆盖的子程序	不可覆盖的子程序
未提供默认实现	抽象且可覆盖的子程序	不会用到，无意义

纯虚函数

函数类型举例

```
class Shape  
{
```

```
    public:
```

```
        virtual void draw() const = 0;
```

```
        virtual void error(const string& msg);
```

```
        int objectID() const;
```

```
        ...
```

```
};
```

Shape a;



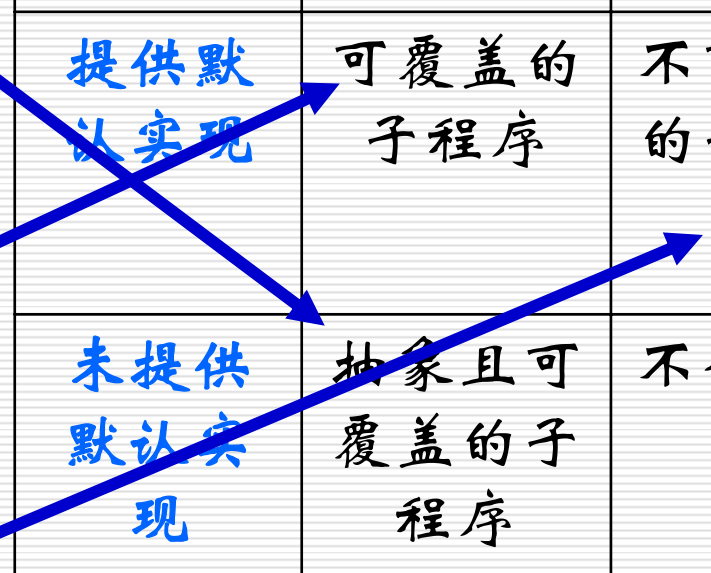
```
class Rectangle: public Shape { ... };
```

```
class Ellipse: public Shape { ... };
```

继承方式总结

- 声明纯虚函数的目的在于：使派生类仅仅只是继承函数的接口
- 声明虚函数的目的在于：使派生类继承函数的接口和缺省实现
- 声明非虚函数的目的在于：使派生类继承函数的接口和强制性实现

	可覆盖的	不可覆盖的
提供默认实现	可覆盖的子程序	不可覆盖的子程序
未提供默认实现	抽象且可覆盖的子程序	不会用到



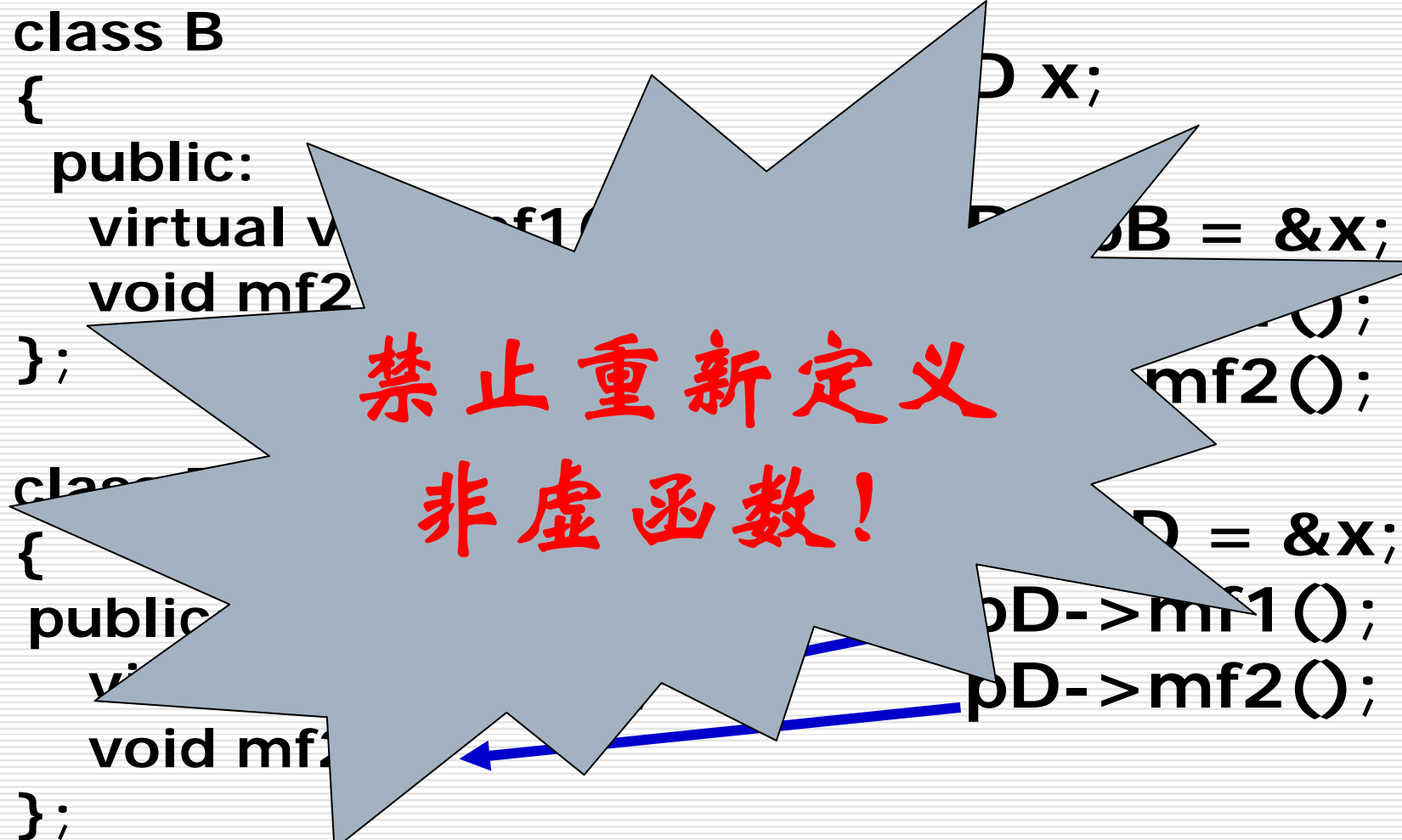
非虚函数不允许被覆盖

```
class B
{
public:
    virtual void mf1();
    void mf2();
};

class D
{
public:
    void mf1();
    void mf2();
};

D x;
B B = &x;
B b;
b.mf1();
b.mf2();

D* pD = &x;
pD->mf1();
pD->mf2();
```



禁止重新定义
非虚函数!

深入理解类之间的关系 (1)

□ 类之间的三个主要关系

- is-a, has-a, is-implemented-in-terms-of
- 是一个, 有一个, 根据某物实现出

□ 确定public继承塑模出is-a关系

- `class D : public B { ... };`
- 每一个类型为D的对象都是一个类型为B的对象

□ 例子

- `class 学生: public 人`

深入理解类之间的关系 (2)

□ 例子：企鹅是一种鸟。鸟会飞。

```
class Bird{  
public:  
    virtual void fly();    //鸟会飞  
    ...  
};  
  
class Penguin: public Bird{  
    ...    // 企鹅也会飞?  
};
```

深入理解类之间的关系 (3)

```
class Bird{  
    ...  
};  
class FlyingBird: Bird{  
public:  
    virtual void fly();        //会飞的鸟  
    ...  
};  
  
class Penguin: public Bird{  
    ...  
};
```

并不存在一个“适用于所有软件”的完美设计！

最佳设计，取决于你希望系统做什么事情！

深入理解类之间的关系 (4)

□ 例子：正方形是一种特殊的矩形。

□ **class Square : public Rectangle**

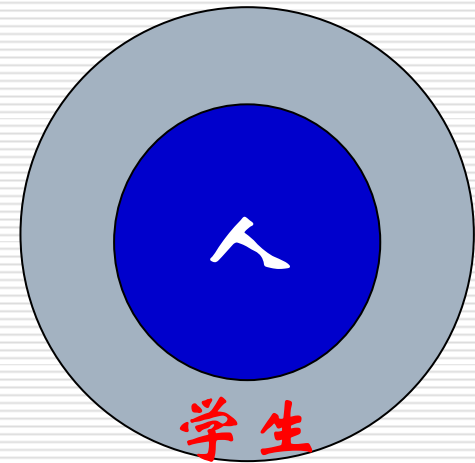
```
class Rectangle{
public:
    virtual void setHeight(int newHeight);
    virtual void setWidth(int newWidth);
    virtual int height( ) const;        // 返回当前值
    virtual int width( ) const;
    ...
};

void makeBigger(Rectangle& r)
{
    int oldHeight = r.height();
    r.setWidth(r.width() + 10);
    assert(r.height() == oldHeight ); //判断r的高度是否未曾改变
}
```

深入理解类之间的关系 (5)

□ 是一个的关系 等价于 public 继承?

```
class Square : public Rectangle{ ... };  
Square s;  
  
assert(s.width() == s.height() );  
makeBigger(s);  
assert(s.width() == s.height() );  
...
```



- public 继承是一种“能力扩大、增加”的概念
- 自然语言中：“是一个”有些时候是“能力缩小、减少”的概念

深入理解类之间的关系 (6)

□ 复合关系 (composition)

■ has-a, is-implemented-in-terms-of

```
class Person {  
    ...  
private:  
    string name;  
    string address;  
    PhoneNumber fixedNumber;  
    PhoneNumber mobileNumber;  
    ...  
};
```

深入理解类之间的关系 (7)

- 软件处理两个不同的领域，因此复合关系有两重意义，也就是两种关系
 - 有一个：has-a
 - 真实世界中的某些事物，如：人，汽车---属于应用领域部分
 - 根据某物实现：is-implemented-in-terms-of
 - 抽象世界的人工制品，如：缓冲区，查询树等----软件的实现领域

深入理解类之间的关系 (8)

```
class Set : public list { ... };
```

?

```
class Set {  
public:  
    bool member() const;  
    void insert(...);  
    void remove(...);  
    ...  
private:  
    list items;  
};
```

深入理解类之间的关系 (9)

```
class Person { ... };  
class Student : private person { ... };  
void eat (const Person& p);  
void study(const Student & s);
```

```
Person p;  
Student s;
```

```
eat(p);  
eat(s);
```

深入理解类之间的关系 (10)

□ 私有继承

- 编译器不会将派生类自动转换为基类;
- 通过私有继承, 基类中的所有属性在派生类中都变成了私有属性。

□ 私有继承

- 相当于: 根据某物实现的关系
- 纯粹的实现技术, 在软件的设计层面没有意义, 只是在软件的实现层面有意义。

```
class A : Private B
{
    ...
};
```

```
class A
{
    private:
        B b;
};
```

深入理解类之间的关系 (11)

- 使用“根据某物实现”代替私有继承
 - 即轻易不要使用私有继承

- 保护继承
 - 忘了它吧!

深入理解类之间的关系——小结

□ 继承关系

- 公有继承：是一个的关系
- 私有继承：根据某物实现的关系---不建议使用
- 保护继承：忘了它吧

□ 复合关系

- 应用领域层面：有一个的关系
- 软件实现领域：根据某物实现的关系---推荐使用

多态的概念

- 多态性是指发出同样的消息被不同类型的对象接收时有可能导致完全不同的行为。
 - 通俗来说就是：调用同一个函数/方法，不同的对象会产生不同的行为。

- 例如：draw()函数
 - 长方形对象画出来就是长方形
 - 正方形对象画出来就是正方形
 - 椭圆形对象画出来就是椭圆形

多态的实现方式

□ 四种实现方式:

■ 表面的多态性

□ 强制多态

□ 重载

■ 真正的多态性

□ 模板---类型参数化多态

□ 虚函数----包含多态

强制多态

□ 通过将数据进行类型转换实现

■ 类似于

```
switch()  
{  
    case 1:  
    case 2:  
    case 3:  
    ...  
}
```

重载

□ 给同一个名字赋予不同的含义

■ 通过参数类型、数目、返回值不同实现

- 仅仅返回值不同不是重载
- 函数const属性不同属于重载

□ 函数重载/操作符重载

模板

- 顾名思义：同一样一段程序，针对不同的参数类型，执行不同的行为，完成相同的功能。
- 函数模板与类模板
- 模板与重载
 - 模板可以处理任意类型的参数，参数类型不确定；重载处理的参数类型在重载函数声明是确定。
 - 模板处理不同类型的参数时，参数数目是相同的；重载函数的参数数目可以完全不同。

虚函数

- 虚函数是动态绑定的基础，实现多态的核心概念。
- 是非静态的成员函数。
- 在类的声明中，在函数原型之前写virtual。
- virtual 只用来说明类声明中的原型，不能用在函数实现时。
- 具有继承性，基类中声明了虚函数，派生类中无论是否说明，同原型函数都自动为虚函数。
- 在派生类中，可以对基类的虚函数进行重写（覆盖），而不是重载。
- 在调用对象虚函数时，根据对象所属的派生类，决定调用哪个函数实现（动态联编）
 - 相对于静态联编：编译时即确定执行哪一段代码

多态的例子 (1)

```
class Shape {  
    public:  
        // 所有的形状都要提供一个函数绘制它们本身  
        virtual void draw() const = 0;  
};  
class Rectangle: public Shape {  
    public:  
        virtual void draw() const;           // 画矩形 Draw A  
};  
class Circle: public Shape {  
    public:  
        virtual void draw() const;           // 画圆形 Draw B  
};
```

多态的例子 (2)

// ps 静态类型 = Shape*

Shape *ps;

ps 动态类型: ?

// pc 静态类型 = Shape*

Shape *pc = new Circle;

pc 动态类型:
Circle

// pr 静态类型 = Shape*

Shape *pr = new Rectangle;

pr 动态类型:
Rectangle

多态的例子 (3)



调用 `Circle::draw()`

`ps = pc;`



调用 `Rectangle::draw()`

`ps->draw();`

被调用，被调用的函数就由那个对象的动态类型决定。

`ps = pr;`

`ps->draw();`

ps 的动态类型: `Rectangle`

多态小结

☐ 对象指针

■ 静态类型：声明时决定

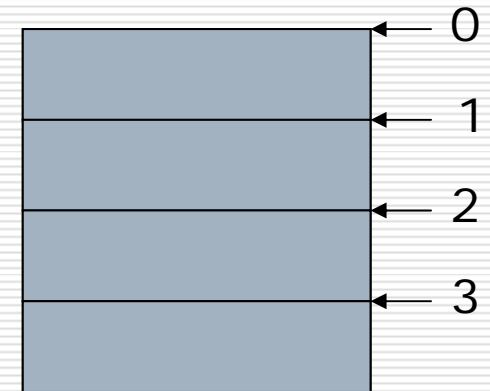
☐ 决定所调用的非虚函数的代码

■ 动态类型：程序运行时决定，指向哪个对象，那么该对象的静态类型就是本指针当前的动态类型

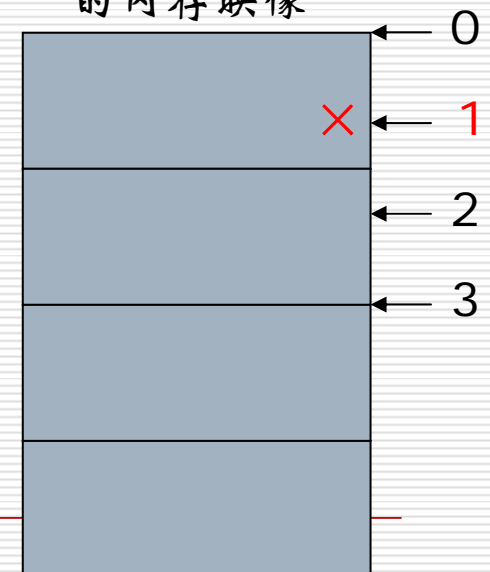
☐ 决定所调用的虚函数的代码

决不能把多态应用于数组 (1)

Tree的内存映像



BalancedTree
的内存映像



`i*sizeof(an object in the array,`
`就是Tree对象的大小)`

```
//输出树
void printTreeArray(const Tree array[], int number)
{
    for ( int i = 0; i < number; i++)
        cout << array[i];    // *(array+i)
}
```

```
Tree treeArray[10];
printTreeArray(treeArray , 10);
```

```
//指向父类的指针可以指向子类
BalancedTree bTreeArray[10];
printTreeArray(bTreeArray , 10);
```

决不能把多态应用于数组 (2)

□ 必须牢记：

- 多态和指针运算是不能在一起使用的，这样的代码不可能有好结果。
- 推广：而数组操作总是要涉及到指针运算的，因此，决不能把多态应用于数组。

动态绑定机制 (1)

□ 函数调用捆绑/绑定

- 把函数体与函数调用相联系称为绑定 (binding)
- 早绑定/静态绑定
 - 程序运行之前 (由编译器和连接器) 完成

□ 晚绑定 (相对于非虚函数的早绑定/静态绑定)

- 捆绑在运行时发生
- 编译期间建立虚函数表 (vtable)
- 保存指向本类虚函数的指针 (vptr)

动态绑定机制 (2)

□ 无虚函数的对象内存映像

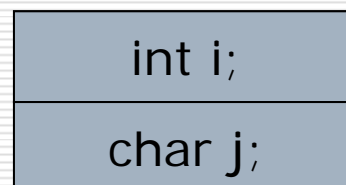
■ $\text{sizeof}(a) = \text{sizeof}(\text{int}) + \text{sizeof}(\text{char});$

```
class A
{
    int i;
    char j;
    func() {};
}
```

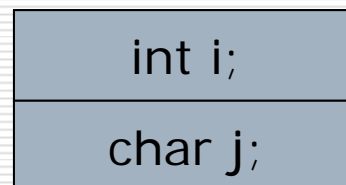
```
a.func();
//      CALL ABCD;
```

```
b.func();
//      CALL ABCD;
```

对象a内存映像



对象b内存映像



ABCD



动态绑定机制 (3)

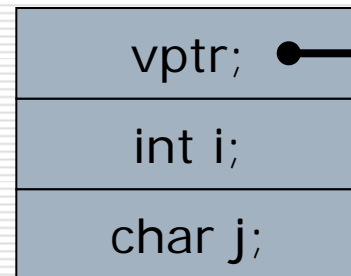
func(): ABC1
&A::func1: ABC2
&A::func2: ABC3
&B::func2: ABC4

□ 无虚函数的对象内存映像

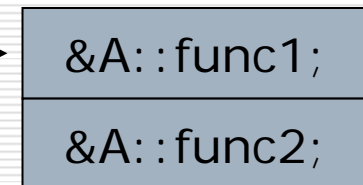
■ $\text{sizeof}(a) = \text{sizeof}(\text{int}) + \text{sizeof}(\text{char}) + 4;$

```
class A
{
    int i;
    char j;
    func() {};
    virtual func1() {};
    virtual func2() {};
};
class B : public class A
{
    virtual func2() {};
};
```

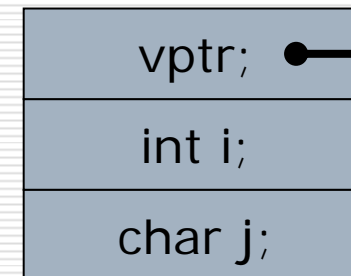
对象a内存映像



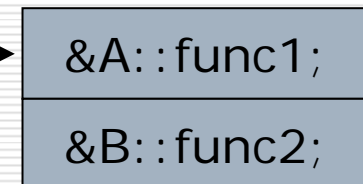
VTABLE



对象b内存映像



VTABLE



动态绑定机制 (4)

□ 非虚/虚函数地址

- 整个继承类系只有一个

□ 虚函数表 (vtable)

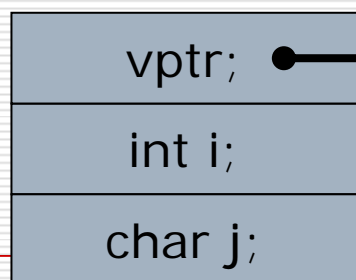
- 每个类只有一个

□ 指向本类虚函数的指针 (vptr)

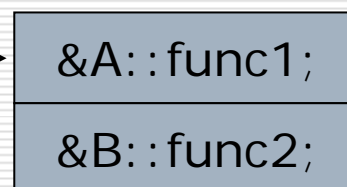
- 每个对象有一个

```
a.func();  
// call ABCD  
  
a.func2();  
// call word [vptr + 2]
```

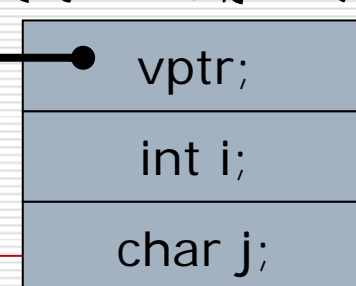
对象a1内存映像



VTABLE



对象a2内存映像



类的实现

□ 属性

□ 构造函数：初始化资源

□ 行为/方法

□ 析构函数：释放资源

构造函数 (1)

```
class Empty { };
```

```
class Empty {  
public:  
    Empty() {};  
    Empty(const Empty& rhs) { };  
    ~Empty() {};  
  
    Empty& operator=(const Empty & rhs){ };  
};
```

构造函数 (2)

□ Empty() {...};

□

```
Empty a;           //Empty(), default构造函数调用
                    //什么都不做, 与赋0值完全不同!
```

□

```
Empty b(a);        // Empty(const Empty& rhs), copy构造函数调用
```

```
Empty c=a;         //Empty& operator=(const Empty & rhs)
                    //copy 赋值构造函数调用
```

□ Empty(int, char) {...};

■ 普通的构造函数, 编译器不会自动构造

构造函数 (3)

□ 没有初始化，就使用不确定的属性值

- 这会导致程序运行结果的不确定性/属性

```
class Empty {  
public:  
    Empty(int , char ) {...};  
    ...  
};
```

0初
(有

- 解: `Empty a;` // 编译通不过了!
// 缺少Empty()函数的声明与实现

- 编写一个带有实参的构造函数: `Empty(int, char) {...};`

构造函数 (4)

- 属性的类型包含const和引用类型变量的情况下，编译器拒绝自动生成copy/copy赋值构造

- ```
class Object {
public:
 string& name;
 ...
};
```

```
Object a("name1");
Object c("name2");
Object a=c; //编译通不过了!
```

# 构造函数 (5)

- 属性的类型包含指针类型变量的情况下，编程容易引发问题。

- ```
class Object {  
public:  
    char* name;  
    ...  
};
```

```
Object a("name1");  
Object c("name2");  
Object b(c); //编译通过,但有问题!  
Object a=c;  //编译通过,但有问题!
```

...ent构造函数

```
b.name = c.name;  
a.name = c.name;
```

构造函数 (6)

❑ 不能以任何种构造函数的方式实例化对象。

❑

```
class Product {  
private:  
    Product (const Human& rhs);  
    Product ();  
    ...  
};
```

```
Product a;           // X
```

```
Product b(a);        // X
```

// 任何人一旦初始化就一定有名字，因此

```
Product a;
```

```
Product b(a);
```

// 是系统不接受的，但是编译器说OK!

构造函数 (7)



class Singleton {

private static Singleton *_instance=null;

private Singleton(){ }

public static Singleton getInstance() {

if(_instance == null) {

_instance = new Singleton();

}

return *_instance;

}

}

Singleton s = getInstance();

例的

赋值和初始化 (1)

- 赋值: assignment (=)
- 初始化: initialization


```
class Phonenumner { ... };
class AddressEntry {
public:
    AddressEntry(const string& name, const string& address,
                 const list<Phonenumner>& phones);
private:
    string _name;
    string _address;
    list<Phonenumner> _phones;
    ...
};

AddressEntry(const string& name, const string& address,
             const list<Phonenumner>& phones)
{
    _name = name;
    _address = address;
    _phones = phones;    //这些都是赋值，而不是初始化!
    ...
};
```

赋值和初始化 (2)

□ 构造函数调用顺序:

- 先调用基类构造函数再调用派生类构造函数

构造好比分东西

先长辈，后朋友，最后自己

析构好比干活

先自己，后朋友，最后长辈

函数时间轴

函数、

`_name`的copy赋值构造函数、`_address`的copy赋值构造函数、`_phones`的copy赋值构造函数

```
AddressEntry(const string& name, const string& address,  
             const list<Phonenumber>& phones)  
{  
    _name = name;  
    _address = address;  
    _phones = phones;    //这些都是赋值，而不是初始化!  
    ...  
};
```

```
AddressEntry(const string& name, const string& address,  
             const list<Phonenumber>& phones)  
    :_name(name),  
    _address(address), _phones(phones)  
{  
};
```

函数开销：_name的构造函数、_address的构造函数、
_phones的构造函数、AddressEntry的构造函数

赋值和初始化 (3)

□ 对于大多数的类型而言

- 先调用default构造函数，再调用copy assignment构造函数，比起只调用copy构造函数代价要高，甚至要高得多。

□ 对于内置类型而言，效率是一致的

- 内置类型也需要手工初始化，编译器不会替你做。
- 而考虑到一致性，内置数据类型最好也这样采用成员初值列初始化。

□ 注意：

- 如果采用成员初值列初始化，尽量将所有的成员都采用这种方式，以免遗漏。
- 建议：初值列的顺序与声明顺序保持一致

赋值和初始化 (4)

□ 尽可能延后变量定义式的出现时间

```
string encryptPassword(const string & password)
{
    string encrypted;
    if(password.length() < 20) return "password too short";
    encrypted(password);
    return encrypted;
};
```

```
string encryptPassword(const string & password)
{
    if(password.length() < 20) return "password too short";
    string encrypted;
    encrypted(password);
    return encrypted;
};
```

赋值和初始化 (5)

- 尽可能延后变量定义式的出现时间
 - 延后变量的定义直到该变量必须被使用;
 - 延后变量的定义直到能够获得该变量的初值。

```
string encryptPassword(const string & password)
{
    if(password.length() < 20) return "password too short";
    string encrypted(password);
    return encrypted;
};
```

析构函数 (1)

```
class Timerkeeper{  
public:  
    Timekeeper() {};  
    ~Timekeeper() {};  
    ...  
};
```

```
class AtomicClock: public Timerkeeper{...};  
class WaterClock: public Timerkeeper{...};  
class WristClock: public Timerkeeper{...};
```

```
Timerkeeper *timer;  
timer = new AtomicClock;  
timer = new WaterClock;  
...
```

```
delete timer;
```

析构函数 (2)

❑ delete timer; // Timerkeeper timer

```
class Timerkeeper{  
public:  
    Timekeeper() {};  
    virtual ~Timekeeper() {};  
    ...  
};
```

❑ 规则

- 基类都应该将析构函数声明为虚析构函数
 - ❑ 当可能通过基类指针删除派生类对象时
- 不处于任何继承体系的类没有这个必要

深入理解传值与传引用 (1)

```
class person{
private:
    string name;
    string address;
};
class student: public person{
private:
    string schoolname;
    string schooladdress;
};

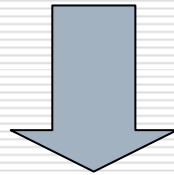
bool validateStudent(student s){};

student a;
bool sIsOK=validateStudent(a);
```

- ☐ student构造函数1次
- ☐ schoolname, schooladdress构造函数调用各1次
- ☐ person构造函数调用1次
- ☐ name, address构造函数调用各1次
- ☐ name, address析构函数调用各1次
- ☐ person析构函数调用1次
- ☐ schoolname, schooladdress析构函数调用各1次
- ☐ student析构函数1次

深入理解传值与传引用 (2)

❑ `bool validateStudent(student s){};`



❑ `bool validateStudent(const student &s){};`

深入理解传值与传引用 (3)

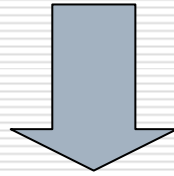
□ 对象切割问题

```
class window{
public:
    string name() const;
    virtual void display() const;
};
class windowWithBar: public window{
public:
    virtual void display() const;
};
void printNameAndDisplay(window w)
{
    cout << w.name();
    w.display();
};

windowWithBar win;
printNameAndDisplay(win);
```

深入理解传值与传引用 (4)

❑ `void printNameAndDisplay(window w)`



❑ `void printNameAndDisplay(const window& w)`

深入理解传值与传引用 (5)

- 是不是所有的传值都改为传引用呢?
 - 不对

- C++的内置类型：以传值方式传递参数
 - 内置类型没有构造函数与析构函数的开销
 - 深入C++的编译器，引用往往以指针的形式实现，因此，传递引用往往意味着传指针

- 注意：在返回对象时，绝对要慎重返回对象的引用

深入理解传值与传引用 (6)

□ 函数返回值传递引用

- 栈上内存 (stack)

- 堆上内存

```
class Rational{  
private  
    int n,d;  
public:  
    const Rational& operator * ( const Rational& lhs,  
                                const Rational& rhs);  
    ...  
};
```

深入理解传值与传引用 (7)

□ 函数返回值传递引用

■ 栈上内存 (stack)

■ 堆上内存

Rational w, a, b, c;
w = a * b * c; //申请的内存呢?

```
const Rational& operator * ( const Rational& lhs, const Rational& rhs);  
{  
    Rational result(lhs.n * rhs.n, lhs.d * rhs.d);  
    return result;  
}
```

```
const Rational& operator * ( const Rational& lhs, const Rational& rhs);  
{  
    Rational *result = new Rational(lhs.n * rhs.n, lhs.d * rhs.d);  
    return *result;  
}
```

自增运算符 (++) 和自减运算符 (--)

- 前缀形式: ++i, --i
- 后缀形式: i++, i--
- 自增运算符 (++)
 - ++i: 先加, 返回加后的值;
 - i++: 返回原来的值, 再值加1。

自增运算符 (++) 和自减运算符 (--)

```
++i;  
i++;
```

```
Object& operator++(Object& i)  
{  
    *  
    r  
}  
const Object& operator++(const Object& i)  
{  
    c  
    ++  
    return oldValue ;  
}
```

Object i;

```
//(i.operator++()). operator++()  
++++ i;
```

```
//(i.operator++(0)). operator++(0)  
i++++;
```

```
// i.operator++(0) 的返回值是一个const  
// 设为a, 而a.operator++(0)是非法的!
```

```
return oldValue ;
```

□ int 参数

用作标志符

不给出参数

返回值

返回引用

返回const对象

率：使用前缀

一致性

谢谢！