

# 继承

## 类之间的关系

### 8.1 类之间的关系

一个大的应用程序，通常由多个类构成，类与类之间互相协同工作。在面向对象技术中，类是数据和操作的集合，它们之间有三种主要关系：`has-a`、`uses-a` 和 `is-a`。

`has-a` 表示类的包含关系，用于描述一个类由多个“部件类”构成。例如，一辆汽车包含发动机、轮子、电池和喇叭等部件，一台计算机由主机、显示器、键盘等部件组成。C++实现 `has-a` 的关系用类成员表示，即一个类中的数据成员是另一个已经定义的类。

`uses-a` 表示一个类部分地使用另一个类。例如，装配计算机时，可以从显示器生产厂家提取不同型号的显示器。又如，操作系统有一个时钟对象，用于保存当前的日期和时间。时钟对象有返回当前日期和时间的成员函数。其他对象可以通过调用时钟对象的函数获取系统日期或时间。在面向对象的技术中，这种关系通过类之间成员函数的相互联系，定义友元或对象参数传递来实现。

`is-a` 表示一种分类方式，描述类的抽象和层次关系。例如，植物分类系统如图 8-1 所示。

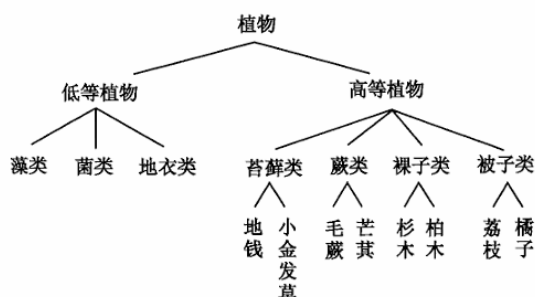


图 8-1 植物分类系统

`is-a` 关系具有传递性。例如，高等植物、蕨类、芒蕨都是植物，具有植物的共同特征；而每种植物都有与其他植物不同的特征。`is-a` 关系不具有对称性。例如，不是所有植物都属于蕨类。

`is-a` 机制称为“继承”。继承是我们常用的一种思维或工作模式。例如，一个服装设计师通常保留许多服装式样（类），每款式样都有自身的特点（属性）。当设计师要设计一种新的款式时，他不需要重新进行设计，只需要找出一种接近的式样进行修改就可以。例如，改变布料，为领子、袖子添加装饰等。这种在原有类的基础上生成（派生）新类的方式就是继承。设计师还可以根据不同款式，如结合旗袍和西式裙的不同特点，设计一种中西结合的裙装。这种由多个类派生一个新的类的方法，称为“多继承”。

## 派生类中访问静态成员的规则

### 1. 基类的静态成员

- 静态成员是类级别的，派生类和基类共享同一个静态成员。
- 派生类可以直接访问基类的静态成员，但需要注意访问权限（`public`、`protected` 或 `private`）。

### 2. 派生类是否可以定义同名的静态成员？

- 如果派生类定义了一个与基类同名的静态成员（变量或函数），派生类的静态成员会**隐藏**基类的静态成员。
- 如果需要同步访问基类和派生类的同名静态成员，可以使用 **作用域解析运算符** `::`

```
class Student {
protected:
    string 学号;                // 学号
    static int 总学生人数;      // 静态成员变量：总学生人数
    static double 总学费收入;  // 静态成员变量：总学费收入

public:
    // 构造函数
    Student(string 学号, double 学费) : 学号(学号) {
        总学生人数++;
        总学费收入 += 学费;
    }

    // 析构函数
    ~Student() {
        总学生人数--;
        总学费收入 -= 学费;
    }

    // 静态成员函数：获取总学生人数
    static int 获取总学生人数() {
        return 总学生人数;
    }

    // 静态成员函数：获取总学费收入
    static double 获取总学费收入() {
        return 总学费收入;
    }
};

// 初始化静态成员变量
int Student::总学生人数 = 0;
double Student::总学费收入 = 0.0;
```

# 类指针的关系

## 用基类指针引用派生类对象

在一般情况下，基类指针指向派生类对象时，只能引用基类成员。

如果试图引用派生类中特有的成员，则必须通过强制类型转换把基类指针转换成派生类指针，否则编译器会报告语法错误。

```
class Animal {
public:
    // 基类的虚函数
    virtual void speak() {
        cout << "Animal speaks!" << endl;
    }
};

// 派生类 Dog
class Dog : public Animal {
public:
    // 重写基类的虚函数
    void speak() override {
        cout << "Dog barks!" << endl;
    }
    void fetch() { // 派生类特有的成员函数
        cout << "Dog fetches the ball!" << endl;
    }
};

// 派生类 Cat
class Cat : public Animal {
public:
    // 重写基类的虚函数
    void speak() override {
        cout << "Cat meows!" << endl;
    }
};

int main() {
    // 创建派生类对象
    Dog myDog;
```

```

Cat myCat;

// 使用基类指针引用派生类对象
Animal* animalPtr;

// 指向 Dog 对象
animalPtr = &myDog;
animalPtr->Speak(); // 输出: Dog barks!

// 指向 Cat 对象
animalPtr = &myCat;
animalPtr->Speak(); // 输出: Cat meows!

// 尝试调用派生类特有的成员函数（会报错！）
// animalPtr->fetch(); // 编译错误: 基类指针无法访问派生类成员

// 必须强制转换为派生类指针才能访问特有成员
Dog* dogPtr = static_cast<Dog*>(animalPtr);
dogPtr->fetch(); // 输出: Dog fetches the ball!

return 0;
}

```

## 用派生类指针引用基类对象

派生类指针只有经过强制类型转换 之后，才能引用基类对象。

//不打算写，因为没必要，你不是专家

## 总结

基类指针无法访问派生类成员，但是可以引用派生类对象

派生类指针只有经过强制类型转换 之后，才能引用基类对象。

井水不犯河水，就是两个都要通过强制转换才能引用对方

## 动态/静态联编

## 2. 代码里的“点餐”

在C++中：

- **静态联编（编译时决定）：**
  - 直接通过 **对象** 调用函数（比如 `obj.func()`）。
  - 通过 **基类指针/引用** 调用 **非虚函数**（比如 `basePtr->func()`，但 `func()` 不是虚函数）。
  - **编译器一看代码，立刻知道该调用哪个函数**，就像提前订好套餐。
- **动态联编（运行时决定）：**
  - 通过 **基类指针/引用** 调用 **虚函数**（比如 `basePtr->func()`，且 `func()` 是 `virtual` 的）。
  - **程序运行时才会看指针实际指向的对象类型**，就像到店现点，服务员根据你的选择上菜。

## 3. 关键区别

特性	静态联编	动态联编
决定时机	编译时（写代码时就定好了）	运行时（程序跑起来才知道）
如何调用	直接调用对象函数 或 基类指针调非虚函数	基类指针调虚函数
多态支持	不支持（只能调基类或固定版本的函数）	支持（可以调派生类的重写函数）
性能	更快（编译时直接绑定，无额外开销）	稍慢（运行时查表，有轻微开销）

## (1) 静态联编 (非虚函数)

cpp

复制

```
class Animal {
public:
    void speak() { cout << "Animal sound" << endl; } // 非虚函数
};

class Dog : public Animal {
public:
    void speak() { cout << "Woof!" << endl; } // 重写, 但不是虚函数重写
};

int main() {
    Animal* ptr = new Dog();
    ptr->speak(); // 输出 "Animal sound" (静态联编, 调用基类版本)
    return 0;
}
```

### 为什么?

`Animal::speak()` 不是虚函数, 编译器一看 `ptr` 是 `Animal*` 类型, 直接调 `Animal` 的版本, 不管 `ptr` 实际指向 `Dog`。

## (2) 动态联编 (虚函数)

cpp

复制

```
class Animal {
public:
    virtual void speak() { cout << "Animal sound" << endl; } // 虚函数
};

class Dog : public Animal {
public:
    void speak() override { cout << "Woof!" << endl; } // 重写虚函数
};

int main() {
    Animal* ptr = new Dog();
    ptr->speak(); // 输出 "Woof!" (动态联编, 调用派生类版本)
    return 0;
}
```

### 为什么?

`Animal::speak()` 是虚函数, 程序运行时发现 `ptr` 实际指向 `Dog`, 所以调用 `Dog` 的版本, 实现多态。

在C++中,要实现动态联编,必须使用( )调用虚函数 A 基类指针、

# 与虚函数

## 关键对比

特性	静态联编	动态联编
绑定时机	编译时	运行时
依据	指针 / 引用的声明类型	对象的实际类型
函数类型	非虚函数	虚函数 (virtual)
性能	快 (编译时确定)	稍慢 (需通过虚函数表查找)
典型应用	函数重载、普通成员函数调用	多态 (基类指针操作派生类对象)

## 进阶示例：虚函数与非虚函数并存

cpp ^

运行 复制 粘贴 设置 全屏

```
#include <iostream>
using namespace std;

class Base {
public:
    void nonVirtual() { cout << "Base::nonVirtual()" << endl; } // 非虚
    virtual void virtualFunc() { cout << "Base::virtualFunc()" << endl; } // 虚
};

class Derived : public Base {
public:
    void nonVirtual() { cout << "Derived::nonVirtual()" << endl; } // 隐藏基类函数
    void virtualFunc() override { cout << "Derived::virtualFunc()" << endl; } // 重写虚函数
};

int main() {
    Base* ptr = new Derived();

    ptr->nonVirtual(); // 静态联编: Base::nonVirtual()
    ptr->virtualFunc(); // 动态联编: Derived::virtualFunc()

    delete ptr;
    return 0;
}
```

静态：基类 动态：派生类

cpp ^

运行

```
#include <iostream>
using namespace std;

class B {
public:
    virtual void vf1() { cout << "B::vf1()" << endl; }
    virtual void vf2() { cout << "B::vf2()" << endl; }
    void f() { cout << "B::f()" << endl; }
};

class D : public B {
public:
    void vf1() override { cout << "D::vf1()" << endl; }
    void vf2(int i) { cout << "D::vf2(int)" << endl; } // 注意：这是一个重载，而非重写
    void f() { cout << "D::f()" << endl; }
};

int main() {
    B* pB = new D;
    // pB->vf2(10); // 错误：B类中没有vf2(int)版本
    pB->vf1();      // 动态联编，调用D::vf1()
    pB->vf2();      // 动态联编，调用B::vf2()
    pB->f();        // 静态联编，调用B::f()
    delete pB;
    return 0;
}
```

## 显式/隐式转换



## 1. 隐式转换（偷偷摸摸的转换）

**定义：**编译器自动帮你把一种类型转成另一种类型，不需要你手动写转换代码。

**特点：**悄悄发生，你可能都没注意到！

### 例子1：基本数据类型的隐式转换

cpp

复制

```
int a = 10;
double b = a; // 隐式转换: int -> double
```

- a 是 int 类型，b 是 double 类型。
- 编译器自动把 int 转成 double，你不用写 (double)a。

### 例子2：自定义类的隐式转换（构造函数实现）

cpp

复制

```
class MyString {
public:
    MyString(int size) { ... } // 非 explicit 构造函数
};

void printString(MyString s) { ... }

int main() {
    printString(10); // 隐式转换: int -> MyString
    return 0;
}
```

- printString 需要 MyString 类型的参数，但你传了一个 int (10) 。
- 因为 MyString 有一个 MyString(int) 构造函数，编译器自动调用它，把 10 转成 MyString 对象。
- 你没写任何转换代码，编译器偷偷帮你做了！



## 2. 显式转换（明明白白的转换）

**定义：**你必须手动写代码告诉编译器：“我要把这个类型转成另一个类型”。

**特点：**强制、明确，一眼就能看出来！

### 例子1：基本数据类型的显式转换（C风格）

```
cpp 复制

int a = 10;
double b = (double)a; // 显式转换: int -> double
```

- 你明确写了 `(double)a`，告诉编译器：“把 `a` 转成 `double`”。

### 例子2：C++ 的 `static_cast`（更安全的显式转换）

```
cpp 复制

int a = 10;
double b = static_cast<double>(a); // 显式转换: int -> double
```

- `static_cast` 是 C++ 推荐的显式转换方式，比 C 风格的 `(double)a` 更安全。

### 例子3：自定义类的显式转换（`explicit` 构造函数）

```
cpp 复制

class MyString {
public:
    explicit MyString(int size) { ... } // 加了 explicit 关键字
};

void printString(MyString s) { ... }

int main() {
    // printString(10); // 错误！不能隐式转换
    printString(MyString(10)); // 正确！必须显式调用构造函数
    return 0;
}
```

# 多态

## 类层次中重名成员

在派生类中访问重名成员时，屏蔽基类的同名成员

直接访问派生类的

如果想访问基类的，需要用作用域解析符 `::`：

## 4. 关键规则总结

场景	行为
派生类定义同名成员变量	派生类的成员变量会隐藏基类的同名成员变量（直接访问时优先派生类）。
派生类定义同名成员函数	派生类的成员函数会覆盖基类的同名成员函数（直接调用时优先派生类）。
如何访问基类的重名成员	用 基类名::成员名 显式指定（如 hero.Character::power）。

```
int main() {
    Hero hero;

    // 1. 调用基类的attack()
    hero.Character::attack(); // 输出: Character attacks with power 10

    // 2. 访问基类的power
    cout << "Base power: " << hero.Character::power << endl; // 输出: Base power: 10

    // 3. 仍然可以调用派生类的版本（默认行为）
    hero.attack(); // 输出: Hero attacks with power 100
    cout << "Hero's power: " << hero.power << endl; // 输出: Hero's power: 100

    return 0;
}
```

# 继承

在C++中，不加说明，则默认的继承方式是（ ）。

private

# 规则

### 一个类不能被多次说明为一个派生类的直接基类

## 1. "一个类不能被多次说明为一个派生类的直接基类"

含义：

在**直接继承**时，一个基类**只能被显式继承一次**（不能重复写在派生类的继承列表中）。

错误示例：

cpp

复制

```
class A {};  
class B : public A, public A {}; // 错误! A被重复直接继承
```

为什么不允许：

- 直接重复继承会导致**同一基类的成员在派生类中存在多份拷贝**，引发二义性（比如访问基类成员时不知道用哪一份）。
- C++明确规定：**直接基类列表中每个基类只能出现一次**。

## ### 可以不止一次地成为间接基类

### 2. "可以不止一次地成为间接基类"

含义：

虽然不能直接重复继承，但一个基类**可以通过不同的继承路径被间接继承多次**（形成"菱形继承"结构）。

正确示例（菱形继承）：

cpp

复制

```
class A {};  
class B : public A {}; // B间接继承A  
class C : public A {}; // C间接继承A  
class D : public B, public C {}; // D通过B和C两条路径间接继承A两次
```

## 虚继承

“当不同的类具有相同的间接基类时，（）。”

选项：

- A) 各派生类无法按继承路线产生自己的基类版本
- B) 为了建立唯一的间接基类版本，应该声明间接基类为虚基类
- C) 为了建立唯一的间接基类版本，应该声明派生类虚继承基类
- D) 一旦声明虚继承，基类的性质就改变了，不能再定义新的派生类

## 1. 理解题目背景

题目讨论的是“不同的类具有相同的间接基类”的情况，也就是 **多重继承中的菱形继承问题**。

### 菱形继承示例

```
graph TD
    Base
    Base --> Derived1
    Base --> Derived2
    Derived1 --> MostDerived
    Derived2 --> MostDerived
```

- Derived1 和 Derived2 都继承自 Base。
- MostDerived 同时继承 Derived1 和 Derived2，导致 Base 被间接继承两次（MostDerived 会有两份 Base 的成员）。

## 如何让 MostDerived 只保留一份 Base

**选项C：为了建立唯一的间接基类版本，应该声明派生类虚继承基类**

- 正确。
- 要让 MostDerived 只保留一份 Base，需要在 Derived1 和 Derived2 的继承声明中使用 virtual：

```
cpp
class Derived1 : virtual public Base { ... };
class Derived2 : virtual public Base { ... };
class MostDerived : public Derived1, public Derived2 { ... };
```

- 这样 MostDerived 就只有一份 Base 成员，避免了菱形继承问题。
- “声明派生类虚继承基类”是正确的表述。

**选项D：一旦声明虚继承，基类的性质就改变了，不能再定义新的派生类**

- 错误。
- 虚继承只是改变了继承方式，不会影响基类 Base 的性质，仍然可以正常定义新的派生类。
- 例如，仍然可以定义 `class NewDerived : public Base { ... };`。

# 虚函数

# 虚函数与基类指针

类指针可以指向派生类对象，但只能访问派生类从基类继承的成员（以及基类自身的成员，前提是这些成员在派生类中没有被隐藏或覆盖等情况）”

```
class Base {
public:
    // 基类的成员变量
    int baseVar;
    // 基类的成员函数
    void baseFunc() {
        cout << "Base class function: baseVar = " << baseVar << endl;
    }
};

// 派生类，公有继承基类 Base
class Derived : public Base {
public:
    // 派生类新增的成员变量
    int derivedVar;
    // 派生类新增的成员函数
    void derivedFunc() {
        cout << "Derived class function: derivedVar = " << derivedVar << endl;
    }
};

int main() {
    // 创建派生类对象
    Derived derivedObj;
    // 基类指针指向派生类对象
    Base* basePtr = &derivedObj;

    // 基类指针可以访问派生类从基类继承的成员变量和成员函数
    basePtr->baseVar = 10;
    basePtr->baseFunc();

    // 错误示例：基类指针不能直接访问派生类新增的成员
    // basePtr->derivedVar = 20; // 编译报错，基类指针无法访问派生类特有的成员
    // basePtr->derivedFunc();   // 编译报错，基类指针无法访问派生类特有的成员

    // 如果要访问派生类新增成员，需要进行向下转型（需确保安全性，这里只是演示语法）
```

## 虚函数的继承性

一旦声明为虚函数，所有派生类同名函数自动保持虚特性

```

#include<iostream>
using namespace std;
class Base
{ public:
    Base(char xx)
    { x = xx; }
    virtual void who() //说明虚函数
    { cout << "Base class: " << x << "\n"; }
protected:
    char x;
};
class First_d : public Base
{ public :
    First_d( char xx, char yy ) : Base( xx ) {y = yy; }
    void who() //默认说明虚函数
    { cout<<"First derived class: "<<x<<,"<<y<<"\n"; }
protected:
    char y;
};
class Second_d : public First_d
{ public:
    Second_d( char xx, char yy, char zz ) : First_d( xx, yy )
    { z = zz; }
    void who() //默认说明虚函数
    { cout<<"Second derived class: "<<x<<,"<<y<<,"<<z <<"\n"; }
protected:
    char z;
};

```

```
int main()
{   Base  B_obj( 'A' );
    Base  * p;
    First_d F_obj( 'T', 'O' );
    Second_d S_obj( 'E', 'N', 'D' );
    p = & B_obj;
```

• 251 •

```
        p -> who();
        p = &F_obj;
        p -> who();
        p = &S_obj;
        p -> who();
    }
```

程序运行结果:

```
Base class: A
First derived class: T, O
Second derived class: E, N, D
```

Base 类中的 who 函数冠以关键字 virtual 被说明为虚函数。之后，派生类相同界面的成员函数 who 由于默认其具有虚特性，因而可以省略 virtual 说明符。在 main 函数中，语句：

```
p->who();
```

出现了三次，由于 who 函数的虚特性，随着 p 指向不同对象，每次可以执行不同实现的版本。

基类的虚函数 who 不但是 Base 类的实现版本，而且它的函数原型定义了一种接口，这种接口在派生类中重载了不同的实现版本。

## 定义虚函数的注意事项



定义虚函数时注意如下 4 点。

- ① 一旦一个成员函数被说明为虚函数，则不管经历多少派生类层次，所有界面相同的重载函数都保持虚特性。因为派生类也是基类。
- ② 虚函数必须是类的成员函数。不能将虚函数说明为全局（非成员）函数，也不能说明为静态成员函数。因为虚函数的动态联编必须在类层次中依靠 this 指针实现。
- ③ 不能将友元说明为虚函数，但虚函数可以是另一个类的友元。
- ④ 析构函数可以是虚函数，但构造函数不能是虚函数。

以上规则，都可以用“作用于类体系的动态联编依赖基类指针指向派生类对象，调用虚函数的不同版本”这一事实加以说明。

第一个，也就是同名函数

2. 虚函数必须是类的成员函数，不能是全局函数或静态成员函数

3. 友元函数不能是虚函数，但虚函数可以是另一个类的友元

友元函数不是类成员：友元函数是外部函数，没有类的成员属性，无法放入虚函数表（vtable）

4. 析构函数可以是虚函数，但构造函数不能是虚函数

构造函数不能是虚函数：

- 原因：
  - 对象未完全构造：构造函数执行时，派生类部分还未初始化，虚指针（vptr）未指向派生类的虚函数表，无法实现动态绑定

## 纯虚函数

### 末尾加0

在C++中，**纯虚函数**的声明方式是在虚函数声明的末尾加上 `= 0`，表示该函数没有默认实现，必须由派生类重写。

在下面函数原型中，`( )`声明了fun为纯虚函数 `C virtual void fun()=0;`

### 条件

1. **纯虚函数**必须满足两个条件：- 使用 `virtual` 关键字声明。- 在函数声明末尾加上 `= 0`。
2. **抽象类**是包含至少一个纯虚函数的类，不能直接创建对象。
3. 派生类必须重写基类的所有纯虚函数，否则派生类也会成为抽象类。
- 4.

## 虚析构函数

### 为什么需要虚析构函数？

**当通过基类指针删除派生类对象时，如果基类析构函数不是虚函数，会导致派生类部分未被正确销毁\*\*，引发内存泄漏或资源泄漏。**

## **虚析构函数的作用**

**将基类析构函数声明为虚函数后，通过基类指针删除派生类对象时，会先调用派生类的析构函数，再调用基类的析构函数，确保资源被完全释放。**

**当通过基类指针删除派生类对象时，如果基类析构函数不是虚函数，会导致派生类部分未被正确销毁，引发内存泄漏或资源泄漏。**

## **先派生后基类**

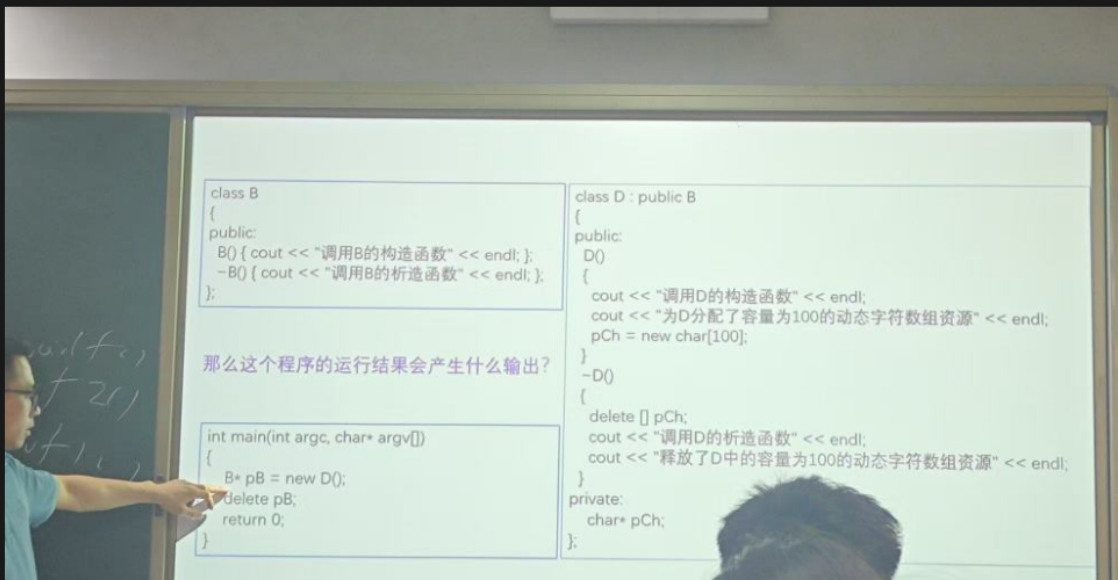
请输入代码块名称

C++

自动换行 开启

复制

```
1 class Animal {
2 public:
3     ~Animal() { cout << "Animal destroyed!" << endl; } // 非虚析构函数
4 };
5
6 class Dog : public Animal {
7 public:
8     ~Dog() { cout << "Dog destroyed!" << endl; }
9     // ~Dog() override { cout << "Dog destroyed!" << endl; }改成这样就可以了
10 };
11
12 int main() {
13     Animal* pet = new Dog();
14     delete pet; // 只调用 Animal 的析构函数，Dog 的析构函数被忽略！
15     return 0;
16 }
```



该程序的运行结果输出为：

请输入代码块名称

C++

自动换行 开启

复制

- 1 调用B的构造函数
- 2 调用D的构造函数
- 3 为D分配了容量为100的动态字符数组资源
- 4 调用B的析构函数

`delete pB` 时只会调用基类 `B` 的析构函数

然后释放 `pB` 所指向的内存空间，但由于没有正确调用 `D` 类的析构函数来释放 `D` 中动态分配的字符数组资源，会导致内存泄漏。

先构造派生类，再构造基类

先析构派生类.....

# L: 员工类

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

const double BASIC_SALARY = 2000.0;    // 一般员工基本工资
const double JOB_SALARY = 3000.0;     // 销售经理职务工资
const double COMMISSION_RATE = 0.005; // 提成率5/1000

// 基类：一般员工
class Employee {
protected:
    string number;    // 员工编号
    string name;      // 员工姓名
    double basicSalary; // 基本工资

public:
    // 构造函数
    Employee(string num, string n, double salary = BASIC_SALARY)
        : number(num), name(n), basicSalary(salary) {}

    // 输入员工信息
    virtual void input() {
        cout << "请输入员工编号: ";
        cin >> number;
        cout << "请输入员工姓名: ";
        cin >> name;
        basicSalary = BASIC_SALARY;
    }

    // 计算工资
    virtual double pay() const {
        return basicSalary;
    }

    // 输出工资条
    virtual void print() const {
        cout << "员工编号: " << number << endl;
        cout << "员工姓名: " << name << endl;
    }
};
```

```

        cout << "基本工资: " << basicSalary << endl;
        cout << "月薪: " << pay() << endl;
    }
};

// 派生类：销售员工
class Salesman : public Employee {
protected:
    double sales; // 销售额

public:
    // 构造函数
    Salesman(string num, string n, double s = 0.0)
        : Employee(num, n), sales(s) {}

    // 输入销售员工信息
    void input() override {
        Employee::input();
        cout << "请输入销售额: ";
        cin >> sales;
    }

    // 计算工资（基本工资 + 销售额 * 提成率）
    double pay() const override {
        return basicSalary + sales * COMMISSION_RATE;
    }

    // 输出工资条
    void print() const override {
        Employee::print();
        cout << "销售额: " << sales << endl;
        cout << "提成: " << sales * COMMISSION_RATE << endl;
    }
};

// 派生类：销售经理
class Salesmanager : public Salesman {
private:
    double jobSalary; // 职务工资

public:
    // 构造函数

```

```

Salesmanager(string num, string n, double s = 0.0)
    : Salesman(num, n, s), jobSalary(JOB_SALARY) {}

// 输入销售经理信息
void input() override {
    Employee::input();
    cout << "请输入销售额: ";
    cin >> sales;
    jobSalary = JOB_SALARY;
}

// 计算工资（基本工资 + 职务工资 + 销售额 * 提成率）
double pay() {
    return basicSalary + jobSalary + sales * COMMISSION_RATE;
}

// 输出工资条
void print() const override {
    Employee::print();
    cout << "职务工资: " << jobSalary << endl;
    cout << "销售额: " << sales << endl;
    cout << "提成: " << sales * COMMISSION_RATE << endl;
}
};

int main() {
    vector<Employee*> employees; // 存储员工指针的容器
    int choice;

    cout << "员工工资管理系统" << endl;
    cout << "1. 添加一般员工" << endl;
    cout << "2. 添加销售员工" << endl;
    cout << "3. 添加销售经理" << endl;
    cout << "4. 显示所有员工工资" << endl;
    cout << "0. 退出" << endl;

    while (true) {
        cout << "\n请选择操作: ";
        cin >> choice;

        if (choice == 0) break;
    }
}

```

点

```
string num, name;
double sales;

switch (choice) {
    case 1: {
        Employee* emp = new Employee("", ""); // 值得注意的一

        emp->input();
        employees.push_back(emp);
        break;
    }
    case 2: {
        Salesman* sm = new Salesman("", "");
        sm->input();
        employees.push_back(sm);
        break;
    }
    case 3: {
        Salesmanager* sm = new Salesmanager("", "");
        sm->input();
        employees.push_back(sm);
        break;
    }
    case 4: {
        cout << "\n所有员工工资信息:" << endl;
        for (const auto& emp : employees) {
            emp->print();
            cout << "-----" << endl;
        }
        break;
    }
    default:
        cout << "无效选择, 请重新输入!" << endl;
}

// 释放内存
for (auto emp : employees) {
    delete emp;
}
```

```
    return 0;
}
```

```
1 Employee(string n, string s, double b = 0) : number(n), name(s), basicsalary(b)
2 {
3     basicsalary = BASIC;
4     //这会导致传入的b被忽略
5     //正确:
6     //Employee(string n, string s) : number(n), name(s), basicsalary(BASIC) {}
7
```

## 怎么调用

记住这个格式：记得要释放内存！！

```
vector<Employee*> employees; // 存储员工指针的容器
Salesmanager* sm = new Salesmanager("", "");
    sm->input();
    employees.push_back(sm);
    break;

for (auto &a : employees)

{

    delete a;

}
```

## 不要重复输入cin



## 不要重复输入cin

```
void input()override{
    Employee::input();
    cout<<"请输入销售额:";
    cin>>sales;
    while (!(cin >> sales) || sales < 0) {
        cout << "输入无效，请输入一个非负数值: ";
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), '\n'); // 清除输入缓冲区
    }
}
```

这样子是会错误的

如果你输入正确的SALES,后面while又要判断一次，但是因为有效，所以循环体不会执行，导致错误，所以咧去掉cin>>sales就好了

## L:几何体类



```
1  class Shape {
2  public:
3      virtual double perimeter() const = 0;
4      virtual double area() const = 0;
5      virtual void show()=0;
6      virtual ~Shape() = default;
7  };
8
9  // 圆类
10 class Circle : public Shape {
11     const double PI=3.1415926;
12     double radius;
13 public:
14     Circle() : radius(rand() % 100) {}
15
16     double perimeter() const override {
17         return 2 * PI * radius;
18     }
19
20     double area() const override {
21         return PI * radius * radius;
22     }
23
24     void show() override {
```

怎么定义PI (常量

优先用 `constexpr`：若常量值在编译时已知（如数学常数、配置参数），优先使用 `constexpr` 以提升性能

使用 `constexpr` 关键字（编译时常量）

```
1 static constexpr double PI = 3.1415926; // 定义编译时常量
```

使用 `const` 关键字（运行时常量）

```
const double PI = 3.1415926; // 定义运行时常量
```

宏定义

```
#define PI 3.1415926
```

# L: three majors

设计一个学生成绩管理系统。某学院有三个专业 ComputerScienceStudent（计科专业学生）、NetworkEngineeringStudent（网络工程专业学生）和 SoftwareEngineeringStudent（软件工程专业学生），共计 400 名学生。计科专业开设有课程英语、高等数学和人工智能，网络工程专业有课程英语、高等数学和计算机网络，软件工程专业有课程英语、高等数学和软件工程等课程。每个学生有学号、姓名、各科成绩、平均成绩等信息。

(1) 设计一个基类学生类 Student；

(2) 从学生类 Student 分别派生三个类 ComputerScienceStudent（计科专业学生）、NetworkEngineeringStudent（网络工程专业学生）和 SoftwareEngineeringStudent（软件工程专业学生）；

(3) 每个派生类中有一个成员函数 Input() 用于产生学生的相关信息（学号、姓名和各科成绩等），可用随机数产生；

(4) 每个派生类中有一个成员函数 Show() 用于显示学生的相关信息（学号、姓名、各科成绩和平均成绩等）；

(5) 在 main 函数中：

- 随机产生 50 名三个专业的学生（可用随机数产生： $\text{rand}() \% 3$ ，0 - 计科 / 1 - 网络 / 2 - 软件）指针放入指针数组中；
- 对数组的学生调用 Input 函数，随机产生各学生的相关信息；
- 对数组的学生调用 Show 函数，显示学生的信息。

```

1
2  class Student
3  {
4  protected:
5      int id;
6      string name;
7      double EnglishScore;
8      double MathScore;
9
10 public:
11     Student() {}
12     virtual ~Student() {}
13     virtual void Input()
14     {
15         id = rand() % (324166999 - 324166000 + 1) + 324166000;
16         EnglishScore = static_cast<double>(rand()) / RAND_MAX * 100;
17         MathScore = static_cast<double>(rand()) / RAND_MAX * 100;
18     }
19     virtual void Show() = 0;
20 };

```

```

class ComputerScienceStudent : public Student
{

protected:

    double AIScore;


public:

    ComputerScienceStudent() {}

    ~ComputerScienceStudent() {}

    void Input() override
    {

```

```

        Student::Input();

        name = GenerateName();


        AIScore = static_cast<double>(rand()) / RAND_MAX * 100;

    }

    void Show() override

```

## 怎么随机生成名字



```

1  string GenerateName()
2  {
3      vector<char> words;
4      for (char c = 'a'; c <= 'z'; ++c)
5      {
6          words.push_back(c);
7      }
8
9      int nameLength = rand() % 3 + 6; //(8-6+1)+6也就是6到8
10     string name;
11     for (int i = 0; i < nameLength; ++i)
12     {
13         int k = words.size(); // 26
14         int index = rand() % k; // 0-26
15         char capital = words[index];
16         if (i == 0)
17         {
18             capital = toupper(capital); // 首字母大写
19         }
20         name += capital;
21     }
22     return name;
23 }

```

# 随机生成学生



```
1  Student *students[400];
2      for (int i = 0; i < 400; ++i)
3      {
4          int random = rand() % 3;
5          switch (random)
6          {
7              case 0:
8                  students[i] = new ComputerScienceStudent();
9                  break;
10             case 1:
11                 students[i] = new NetworkEngineeringStudent();
12                 break;
13             case 2:
14                 students[i] = new SoftwareEngineeringStudent();
15                 break;
16             }
17             students[i]->Input();
18         }
19
20     for (int i = 0; i < 400; ++i)
21     {
22         cout << setfill('0') << setw(3) << i + 1 << " ";
23         students[i]->Show();
24     }
25
26     for (int i = 0; i < 400; ++i)
27     {
28         delete students[i];
29     }
```