

---

# 第6课 类和对象 (继承与多态)

# 内容概述

---

1. 继承的概念
2. 成员的继承
3. 继承方式
4. 构造与析构
5. 拷贝构造与赋值
6. 移动构造与移动赋值
7. 赋值兼容
8. 静态类型与动态类型
9. 类作用域和shadow
10. 友元
11. 虚函数与多态
12. 动态绑定
13. 虚析构
14. 纯虚函数与抽象基类
15. 多态例子
16. 练习

# 继承: 引入

```
class Student { // 学生类
public:
    void eat(const string& food) { // 吃
        cout << "我吃" << food << endl;
    }
    void sleep(const string& addr) { // 睡
        cout << "我睡在" << addr << endl;
    }
    void study(const string& course) { // 学习
        cout << "我学" << course << endl;
    }
private:
    string name; // 姓名
    int age; // 年龄
};

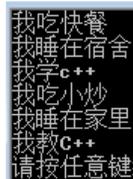
class Teacher { // 教师类
public:
    void eat(const string& food) { // 吃
        cout << "我吃" << food << endl;
    }
    void sleep(const string& addr) { // 睡
        cout << "我睡在" << addr << endl;
    }
    void teach(const string& course) { // 教学
        cout << "我教" << course << endl;
    }
private:
    string name; // 姓名
    int age; // 年龄
};
```

```
class Person { // 人类
public:
    void eat(const string& food) { // 吃
        cout << "我吃" << food << endl;
    }
    void sleep(const string& addr) { // 睡
        cout << "我睡在" << addr << endl;
    }
private:
    string name; // 姓名
    int age; // 年龄
};

class Student : public Person { // 学生类
public:
    void study(const string& course) { // 学习
        cout << "我学" << course << endl;
    }
};

class Teacher : public Person { // 教师类
public:
    void teach(const string& course) { // 教学
        cout << "我教" << course << endl;
    }
};
```

```
int main() {
    Student stu;
    stu.eat("快餐");
    stu.sleep("宿舍");
    stu.study("c++");
    Teacher tea;
    tea.eat("小炒");
    tea.sleep("家里");
    tea.teach("C++");
    return 0;
}
```



```
我吃快餐
我睡在宿舍
我学c++
我吃小炒
我睡在家里
我教C++
请按任意键
```

抽取共性, 代码复用

基类(父类) -- 派生类(子类)

# 继承：概念

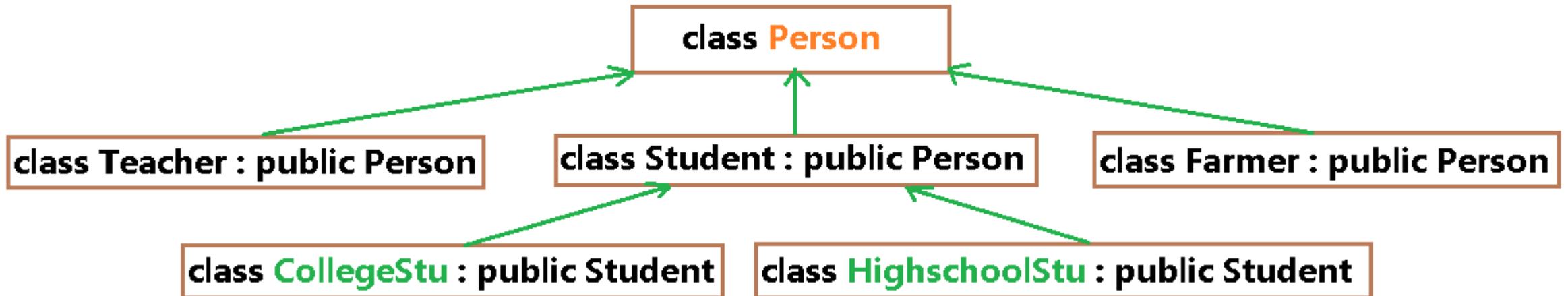
面向对象程序设计的核心思想：**封装，继承和多态**。

使用继承：可以定义相似的类型并对其相似关系建模。

通过继承联系在一起的类构成一种层次关系。

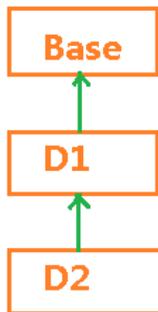
通常层次关系的根部有一个**基类**，其他类则直接或间接地从基类继承而来，这些继承得来的类称为**派生类**。

**基类**负责定义在层次关系中所有类的共同拥有的成员，而每个派生类定义各自特有的成员。



# 继承：成员的继承

```
#include <iostream>
#include <string>
using namespace std;
class Base {
public:
    void f1_base() {}
    void f2_base() {}
    int b_int1;
    int b_int2;
};
class D1 :public Base {
public:
    void f1_d1(){}
    int d1_int;
};
class D2 :public D1 {
public:
    void f2_d2() {}
    int d2_int;
};
int main() {
    Base b;
    D1 d1;
    D2 d2;
    cout << sizeof(b) << ", "
         << sizeof(d1) << ", "
         << sizeof(d2) << endl;
    return 0;
}
```



Base b  
8字节  
int b\_int1  
int b\_int2

D1 d1  
12字节  
int b\_int1  
int b\_int2  
int d1\_int

D2 d2  
16字节  
int b\_int1  
int b\_int2  
int d1\_int  
int d2\_int

代码区：

```
Base :: f1_base()
Base :: f2_base()
```

```
D1 :: f1_d1()
```

```
D2 :: f2_d2()
```

```
d2.b_int1 = 1;          d2.f1_base();
d2.b_int2 = 2;          d2.f2_base();
d2.d1_int = 3;          d2.f1_d1();
d2.d2_int = 4;          d2.f2_base();
```

派生类：全盘接收基类的成员（数据成员和成员函数）  
（除了构造函数和析构函数）

这里的构造函数指：默认构造，拷贝构造，移动构造

# 继承: 继承方式

```
class Base {  
public:  
    void f_pub(){}  
    int i_pub;  
protected:  
    void f_pro(){}  
    int i_pro;  
private:  
    void f_pri(){}  
    int i_pri;  
};
```

```
class D1 :public Base { //public继承  
public:  
    void test() {  
        //测试类内访问  
        f_pub(); //访问基类public成员函数,ok  
        i_pub = 10; //访问基类public数据成员,ok  
        f_pro(); //访问基类protected成员函数,ok  
        i_pro = 20; //访问基类protected数据成员,ok  
        //f_pri(); //访问基类private成员函数,error  
        //i_pri = 30; //访问记录private数据成员,error  
    }  
};
```

```
class D2 :protected Base { //protected继承  
public:  
    void test() {  
        //测试类内访问  
        f_pub(); //访问基类public成员函数,ok  
        i_pub = 10; //访问基类public数据成员,ok  
        f_pro(); //访问基类protected成员函数,ok  
        i_pro = 20; //访问基类protected数据成员,ok  
        //f_pri(); //访问基类private成员函数,error  
        //i_pri = 30; //访问记录private数据成员,error  
    }  
};
```

```
class D3 :private Base { //private继承  
public:  
    void test() {  
        //测试类内访问  
        f_pub(); //访问基类public成员函数,ok  
        i_pub = 10; //访问基类public数据成员,ok  
        f_pro(); //访问基类protected成员函数,ok  
        i_pro = 20; //访问基类protected数据成员,ok  
        //f_pri(); //访问基类private成员函数,error  
        //i_pri = 30; //访问记录private数据成员,error  
    }  
};
```

//测试类外访问

```
D1 d1; //D1 public 继承  
d1.f_pub(); //访问基类public成员函数,ok  
d1.i_pub = 10; //访问基类public数据成员,ok  
//d1.f_pro(); //访问基类protected成员函数,error  
//d1.i_pro = 20; //访问基类protected数据成员,error  
//d1.f_pri(); //访问基类private成员函数,error  
//d1.i_pri = 30; //访问记录private数据成员,error
```

```
D2 d2; //D2 protected 继承  
//d2.f_pub(); //访问基类public成员函数,error  
//d2.i_pub = 10; //访问基类public数据成员,error  
//d2.f_pro(); //访问基类protected成员函数,error  
//d2.i_pro = 20; //访问基类protected数据成员,error  
//d2.f_pri(); //访问基类private成员函数,error  
//d2.i_pri = 30; //访问记录private数据成员,error
```

```
D3 d3; //D3 private 继承  
//d3.f_pub(); //访问基类public成员函数,error  
//d3.i_pub = 10; //访问基类public数据成员,error  
//d3.f_pro(); //访问基类protected成员函数,error  
//d3.i_pro = 20; //访问基类protected数据成员,error  
//d3.f_pri(); //访问基类private成员函数,error  
//d3.i_pri = 30; //访问记录private数据成员,error
```

继承方式 \ 基类成员	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	不可见	不可见	不可见

# 继承：继承方式

继承方式规定了如何访问基类继承的成员。继承方式有 `public`, `protected`, `private`。

继承方式影响从基类继承来的成员的访问权限。

公有继承：基类的公有成员和保护成员在派生类中保持原有访问属性，基类私有成员在派生类中不可见。

保护继承：基类的公有成员和保护成员在派生类中成了保护成员，基类私有成员在派生类中不可见。

私有继承：基类的公有成员和保护成员在派生类中成了私有成员，基类私有成员在派生类中不可见。

继承方式 \ 基类成员	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	不可见	不可见	不可见

## struct与class的区别：

class是默认访问权限为private的struct

struct是默认访问权限为public的class

```
class{  
    int m; ← 默认是 private;  
};
```

```
struct{  
    int m; ← 默认是 public;  
};
```

`class D : Base {};` 默认是 `private` 继承

`struct D : Base{ };` 默认是 `public` 继承

```
class Base {  
public:  
    void f_pub() {}  
    int i_pub;  
protected:  
    void f_pro() {}  
    int i_pro;  
private:  
    void f_pri() {}  
    int i_pri;  
};
```

```
class D1 :private Base {  
public:  
    using Base::f_pro;  
    using Base::i_pro;  
private:  
    //using Base::f_pri; ←  
}; 错误,Base::f_pri不可见  
  
int main() {  
    D1 d1;  
    d1.f_pro();  
    d1.i_pro = 10;  
    return 0;  
}
```

在类的内部使用 `using` 声明，可以改变继承来的名字访问级别。

当然，首先必须是可访问的成员（不能是不可见的！）

# 继承：构造与析构

```
#include <iostream>
using namespace std;
class Base {
public:
    Base() { cout << "Base:默认构造" << endl; }
    ~Base() { cout << "Base:析构" << endl; }
};
class D1 :public Base {
public:
    D1() { cout << "D1:默认构造" << endl; }
    ~D1() { cout << "D1:析构" << endl; }
};
class D2 :public D1 {
public:
    D2() { cout << "D2:默认构造" << endl; }
    ~D2() { cout << "D2:析构" << endl; }
};
int main() {
    D2 d2;
    cout << "-----" << endl;
    return 0;
}
```

实际上：  
`D1():Base(){ }`  
`D2():D1(){ }`

非显式调用时：

派生类的构造函数会调用基类的默认构造函数。

```
Base:默认构造
D1:默认构造
D2:默认构造
-----
D2:析构
D1:析构
Base:析构
请按任意键继续
```

```
#include <iostream>
using namespace std;
class Base {
public:
    Base(int i) { cout << "Base:默认构造" << endl; }
    ~Base() { cout << "Base:析构" << endl; }
};
class D1 :public Base {
public:
    D1(int i) :Base(i) { cout << "D1:默认构造" << endl; }
    ~D1() { cout << "D1:析构" << endl; }
};
class D2 :public D1 {
public:
    D2(int i) :D1(i) { cout << "D2:默认构造" << endl; }
    ~D2() { cout << "D2:析构" << endl; }
};
int main() {
    D2 d2(1);
    cout << "-----" << endl;
    return 0;
}
```

基类没有默认构造函数时：

必须在派生类构造函数的初始化列表处显式调用基类的构造函数。

# 继承：构造与析构

```
class Base {
public:
    Base(int i) :b1(set_b1(i)), b2(set_b2(i))
                { cout << "Base:默认构造" << endl; }
    ~Base() { cout << "Base:析构" << endl; }
    int set_b1(int i) { cout << "b1初始化" << endl; return i; }
    int set_b2(int i) { cout << "b2初始化" << endl; return i; }
private:
    int b1;
    int b2;
};

class D1 :public Base {
public:
    D1(int i) :d1_2(set_d1_2(i)), d1_1(set_d1_1(i)), Base(i)
              { cout << "D1:默认构造" << endl; }
    ~D1() { cout << "D1:析构" << endl; }
    int set_d1_1(int i) { cout << "d1_1初始化" << endl; return i; }
    int set_d1_2(int i) { cout << "d1_2初始化" << endl; return i; }
private:
    int d1_1;
    int d1_2;
};
```

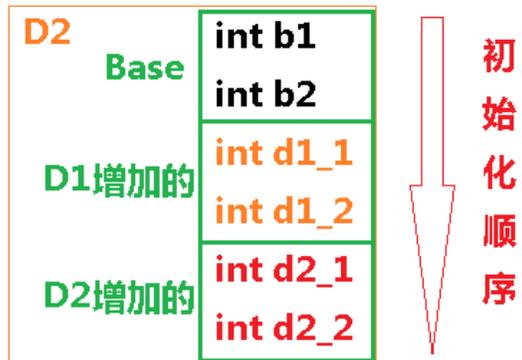
```
b1初始化
b2初始化
Base:默认构造
d1_1初始化
d1_2初始化
D1:默认构造
d2_1初始化
d2_2初始化
D2:默认构造
D2:析构
D1:析构
Base:析构
请按任意键继续
```

```
class D2 :public D1 {
public:
    D2(int i) :D1(i), d2_1(set_d2_1(i)), d2_2(set_d2_2(i))
              { cout << "D2:默认构造" << endl; }
    ~D2() { cout << "D2:析构" << endl; }
    int set_d2_1(int i) { cout << "d2_1初始化" << endl; return i; }
    int set_d2_2(int i) { cout << "d2_2初始化" << endl; return i; }
private:
    int d2_1;
    int d2_2;
};

int main() {
    D2 d2(1);
    cout << "-----" << endl;
    return 0;
}
```

D2 d2(1);

1. 准备调用d2的构造函数，找到D2(int i)与之匹配；
2. 在初始化列表找出显式表示的基类D1的构造函数调用，若没有，则尝试调用D1的默认构造函数，若也没有，则报错（编译错误）；
3. 同理调用Base的对应构造函数；
4. 执行Base构造函数的初始化列表，按数据成员的先后次序初始化，然后运行Base构造函数函数体；
5. 执行D1构造函数的初始化列表，次序同上，然后运行D1构造函数函数体；
6. 执行D2构造函数的初始化列表，次序同上，然后运行D2构造函数函数体。



# 继承：构造与析构

有子对象的类：

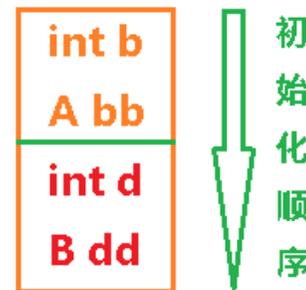
```
class A { //有默认构造
public:
    A() { cout << "A构造" << endl; }
    ~A() { cout << "A析构" << endl; }
};
```

```
class B { //没有默认构造
public:
    B(int i) { cout << "B构造" << endl; }
    ~B() { cout << "B析构" << endl; }
};
```

```
class Base {
public:
    Base(int i) :b(set_b(i)), bb(A()) //这里bb(A())可以省略
                { cout << "Base:构造" << endl; }
    ~Base() { cout << "Base:析构" << endl; }
    int set_b(int i) { cout << "b1初始化" << endl; return i;}
private:
    int b;
    A bb;
};
```

```
b1:初始化
A:构造
Base:构造
d:初始化
B:构造
D1:构造
-----
D1:析构
B:析构
Base:析构
A:析构
请按任意键继续
```

```
class D1 :public Base {
public:
    D1(int i) :d(set_d(i)), Base(i), dd(B(i))
                { cout << "D1:构造" << endl; }
    ~D1() { cout << "D1:析构" << endl; }
    int set_d(int i) {cout << "d初始化" << endl; return i;}
private:
    int d;
    B dd;
};
int main() {
    D1 d1(1);
    cout << "-----" << endl;
    return 0;
}
```



# 继承：拷贝构造和赋值运算符重载

```
class Base {
public:
    Base(int _b) :b(_b), pb(&b) { }
    int b;
    int *pb;
};
class D1 :public Base {
public:
    D1(int _b, int _d) :Base(_b), d(_d), pd(&d) { }
    int d;
    int *pd;
};
int main() {
    D1 d1(1,2);
    D1 d2(d1); //拷贝构造 d1 --> d2
    cout << d2.b << " " << d2.d << endl; // 1 2
    cout << d1.pb << " " << d1.pd << endl;
    cout << d2.pb << " " << d2.pd << endl; //浅拷贝
    D1 d3(0, 0);
    d3 = d2; //赋值运算符 d2 --> d3
    cout << d3.b << " " << d3.d << endl; // 1 2
    cout << d2.pb << " " << d2.pd << endl;
    cout << d3.pb << " " << d3.pd << endl; //浅拷贝
    return 0;
}
```

```
1 2
002BF9E0 002BF9E8
002BF9E0 002BF9E8
1 2
002BF9E0 002BF9E8
002BF9E0 002BF9E8
请按任意键继续. .
```

系统默认的拷贝构造和赋值运算符重载：**等位拷贝**；**浅拷贝**会调用基类的拷贝构造和赋值运算符重载。

```
class Base {
public:
    Base(int _b) :b(_b), pb(&b) { }
    Base(const Base& other)
        :b(other.b),pb(other.pb) {
        cout << "Base拷贝构造" << endl;
    }
    Base& operator=(const Base &other) {
        if (this != &other) {
            b = other.b;
            pb = other.pb;
        }
        cout << "Base赋值运算符重载" << endl;
        return *this;
    }
    int b;
    int *pb;
};
class D1 :public Base {
public:
    D1(int _b,int _d):Base(_b), d(_d), pd(&d){}
    int d;
    int *pd;
};
```

```
Base拷贝构造
1 2
0042F878 0042F880
0042F878 0042F880
Base赋值运算符重载
1 2
0042F878 0042F880
0042F878 0042F880
请按任意键继续. .
```

基类定义了拷贝构造、赋值运算符重载：

派生类的默认拷贝构造和赋值运算符重载，也会自动地调用基类的拷贝构造和赋值运算符重载。

# 继承：拷贝构造和赋值运算符重载

```
class Base {
public:
    Base() { cout << "Base默认构造" << endl; }
    Base(int _b) : b(_b), pb(&b) { }
    Base(const Base& other)
        : b(other.b), pb(other.pb) {
        cout << "Base拷贝构造" << endl;
    }
    Base& operator=(const Base &other) {
        if (this != &other) {
            b = other.b;
            pb = other.pb;
        }
        cout << "Base赋值运算符重载" << endl;
        return *this;
    }
    int b;
    int *pb;
};
```

```
int main() {
    D1 d1(1,2);
    D1 d2(d1); //拷贝构造 d1 --> d2
    cout << d2.b << " " << d2.d << endl; //??? 2
    cout << d1.pb << " " << d1.pd << endl;
    cout << d2.pb << " " << d2.pd << endl;
    D1 d3(0, 0);
    d3 = d2; //赋值运算符 d2 --> d3
    cout << d3.b << " " << d3.d << endl; // 0 2
    cout << d2.pb << " " << d2.pd << endl;
    cout << d3.pb << " " << d3.pd << endl;
    return 0;
}
```

```
class D1 :public Base {
public:
    D1(int _b, int _d) :Base(_b), d(_d), pd(&d) {}
    D1(const D1& other)
        :d(other.d), pd(other.pd) { // 不正确写法
        cout << "D1拷贝构造" << endl;
    }
    D1 &operator=(const D1 &other) {
        if (this != &other) {
            d = other.d;
            pd = other.pd;
        }
        cout << "D1赋值运算符重载" << endl;
        return *this;
    }
    int d;
    int *pd;
};
```

```
Base默认构造
D1拷贝构造
-858993460 2
0014FB84 0014FBAC
CCCCCCCC 0014FBAC
D1赋值运算符重载
0 2
CCCCCCCC 0014FBAC
0014FB74 0014FBAC
请按任意键继续.
```

```
class D1 :public Base {
public:
    D1(int _b, int _d) :Base(_b), d(_d), pd(&d) {}
    D1(const D1& other)
        :Base(other), d(other.d), pd(other.pd) {
        cout << "D1拷贝构造" << endl;
    }
    D1 &operator=(const D1 &other) {
        Base::operator=(other); //必须加 Base::
        if (this != &other) {
            d = other.d;
            pd = other.pd;
        }
        cout << "D1赋值运算符重载" << endl;
        return *this;
    }
    int d;
    int *pd;
};
```

```
Base拷贝构造
D1拷贝构造
1 2
003CF804 003CF80C
003CF804 003CF80C
Base赋值运算符重载
D1赋值运算符重载
1 2
003CF804 003CF80C
003CF804 003CF80C
请按任意键继续.
```

派生类的拷贝构造，必须要在列表初始化中显式地指明调用基类中的哪个构造函数来初始化基类数据成员。  
若不指定：则调用默认构造。

派生类的赋值运算符重载，若自己实现，则必须自己调用基类的函数，以实现基类数据成员的赋值运算。  
若不调用：则基类数据成员将不会被赋值。

尽管在派生类中有时也可以对基类数据成员进行赋值操作，但不推荐。  
深拷贝还是浅拷贝或者引用计数等根据需求实现。

# 继承：移动构造和移动运算符重载

```
class Base {
public:
    Base(int _b) :b(_b), pb(new int(_b)) { }
    ~Base() {cout << pb<< " Base析构"<< endl; delete pb;}
    Base(Base&& other)
        :b(other.b), pb(other.pb) {
        other.pb = NULL; //保证移后源对象可析构
    }
    cout << "Base移动构造" << endl;
    Base& operator=(Base &&other) {
        if (this != &other) {
            b = other.b;
            pb = other.pb;
        }
        other.pb = NULL; //保证移后源对象可析构
        cout << "Base移动赋值运算符重载" << endl;
        return *this;
    }
    int b;
    int *pb;
};

int main() {
    D1 d1(1, 2);
    //D1 d2(d1); //错, 定义了移动构造, 必须自己定义拷贝构造
    D1 d2(std::move(d1)); //移动构造
    cout << d2.b << " " << d2.d << endl;
    cout << d1.pb << " " << d1.pd << endl; // NULL
    cout << d2.pb << " " << d2.pd << endl;
    D1 d3(0, 0);
    //d3 = d2; //错, 定义了移动构造, 必须自己定义拷贝赋值运算符重载
    d3 = std::move(d2);
    cout << d3.b << " " << d3.d << endl;
    cout << d2.pb << " " << d2.pd << endl;
    cout << d3.pb << " " << d3.pd << endl;
    return 0;
}
```

```
Base 移动构造
D1 移动构造
1 2
00000000 00000000
004B5690 004B56C0
Base 移动赋值运算符重载
D1 移动赋值运算符重载
1 2
00000000 00000000
004B5690 004B56C0
004B56C0 D1 析构
004B5690 Base 析构
00000000 D1 析构
00000000 Base 析构
00000000 D1 析构
00000000 Base 析构
请按任意键继续. . .
```

```
class D1 :public Base {
public:
    D1(int _b, int _d) :Base(_b), d(_d), pd(new int(_d)) { }
    ~D1() { cout << pd << " D1析构" << endl; delete pd; }
    D1(D1&& other) //必须加 std::move()
        :Base(std::move(other)), d(other.d), pd(other.pd) {
        other.pb = NULL; //保证移后源对象可析构
        cout << "D1移动构造" << endl;
    }
    D1 &operator=(D1 &&other) {
        //必须加 Base:: 和 std::move()
        Base::operator=(std::move(other));
        if (this != &other) {
            d = other.d;
            pd = other.pd;
        }
        other.pb = NULL; //保证移后源对象可析构
        cout << "D1移动赋值运算符重载" << endl;
        return *this;
    }
    int d;
    int *pd;
};
```

移动构造和移动赋值运算符重载：  
和拷贝构造、拷贝赋值运算符重载类似。

注意加上 std::move()

# 继承：移动构造和移动运算符重载

假如数据成员中有子对象，那么这些对象都要支持移动才可以。  
并且在移动构造和移动赋值运算符重载时，要写上 `std::move(xxx)`

例如：D1 中有数据成员 `string s`;

那么 移动构造：`D1(D1&& other) : Base(std::move(other)), s(std::move(other.s)),...`

移动赋值：`s = std::move(other.s);`

基类中假如没有拷贝构造、移动构造、拷贝赋值运算符重载、移动赋值运算符重载，则派生类中也不会有这些函数（因为派生类中的这些函数会调用基类的这些函数）。

所以一般要显式地给出默认定义：

```
Base(const Base&) = default;
```

```
Base(Base&&) = default;
```

```
Base& operator=(const Base&) = default;
```

```
Base& operator=(Base &&) = default;
```

# 继承：赋值兼容

在拷贝构造中：

```
Base(const Base& other):b(other.b), pb(other.pb) {} //Base拷贝构造函数
```

```
D1(const D1& other) :Base(other),d(other.d),pd(other.pd) {} //D1拷贝构造函数
```

在D1的拷贝构造中，我们 Base(other), 其中 other 的类型是 const D1& , 但是 Base 拷贝构造函数的参数是 const Base&, 显然是不同的类型，为啥可以这样做？（**隐式转换**）

**赋值兼容**：派生类对象（引用或指针），可以赋给基类对象（引用或指针）【**隐式转换**】

- 1: 派生类的对象可以赋值给基类对象。
- 2: 派生类的对象可以初始化基类的引用。
- 3: 派生类对象的地址可以赋给指向基类的指针。

上面3条的前提：**public继承**。

public继承的派生类实际上具备基类的所有功能，凡是基类能解决的问题，public派生类都可以解决。

# 继承：赋值兼容

```
#include <iostream>
using namespace std;
class Base {
public:
    Base() = default;
    Base(const Base& other) :b(other.b)
        { cout << "Base copy" << endl; }
    void f() { cout << "Base f()" << endl; }
private:
    int b;
};
class D1:public Base {
public:
    D1() = default;
    D1(const D1& other) :Base(other), d(other.d)
        { cout << "D1 copy" << endl; }
    void f() { cout << "D1 f()" << endl; }
private:
    int d;
};
```



```
Base copy
Base copy
Base f()
D1 f()
002AF0E4002AF0E4
Base f()
Base f()
请按任意键继续.
```

```
int main() {
//1.基类对象 和 派生类对象 之间 的拷贝与赋值
//只能将派生类对象拷贝或赋值给基类对象,反之不行
D1 d1;
Base b1(d1); //ok,显式初始化,只执行 Base拷贝构造
Base b2 = d1; //ok,拷贝初始化,只执行 Base拷贝构造
b2 = d1; //ok,可以将派生类对象赋值给基类对象
//d1 = b1; //error,不能将基类对象赋值给派生类对象
//[上面这样的做法虽然可以,但是一般不这样用]
b1.f(); //Base f()
d1.f(); //D1 f()

//2.基类的引用：用派生类对象来初始化
Base &rb1 = d1; //ok,基类引用 用派生类对象初始化
cout << &rb1 << &d1 << endl; //地址一样,说明没有临时量
//D1 &rd1 = b1; //error,派生类引用不能用基类对象初始化
rb1.f(); //Base f(),rb1是Base的引用,所以调用Base的f

//3.基类的指针：指向派生类对象
Base *pb1 = &d1; //ok,基类指针 指向派生类对象
//D1 *pd1 = &b1; //error,派生类指针不能指向基类对象
pb1->f(); //Base f(),pb1是Base的指针,所以调用Base的f
return 0;
}
```

只能使用基类的接口

# 继承：静态类型和动态类型

**静态类型**：编译时已知，变量声明的类型；

**动态类型**：运行时才知道，变量在内存中对象的真实类型。

```
class D1 : public Base{ ... }
```

```
Base b1; // 对象b1的静态类型是Base，动态类型也是Base
```

```
D1 d1; // 对象d1的静态类型是D1，动态类型也是D1
```

若既不是引用也不是指针，那么它的动态类型和静态类型一致。

```
Base &rb = b1; // 引用，rb的静态类型和动态类型都是Base&
```

```
Base &rd = d1; // 引用，rd的静态类型是Base&，动态类型是D1
```

```
Base *pb = &b1; // 指针，pb的静态类型和动态类型都是Base*
```

```
Base *pb = &d1; // 指针，pb的静态类型是Base*，动态类型是D1*
```

**基类的指针或引用**的静态类型可能与其动态类型不一致。

# 继承：类作用域和shadow

```
struct Base {
    Base() :d(0) { }
    void base_show() {
        cout << "Base d: " << d << endl;
    }
    int d;
};
struct D1: Base {
    D1(int i) :d(i) { }
    void d1_show() { //这里的 d 是哪个d? D1中的!
        cout << "D1 d: " << d << endl;
    }
    void d1_show2() {
        cout<<"d1_show2: Base::d "<<Base::d<<endl;
    }
    int d;
};
int main() {
    Base b1;
    b1.base_show(); // 0
    cout << b1.d << endl; // 0
    D1 d1(10);
    d1.d1_show(); // 10
    cout << d1.d << endl; // 10
    d1.d1_show2(); // 0
    cout << d1.Base::d << endl; // 0
    return 0;
}
```



```
Base d: 0
0
D1 d: 10
10
d1_show2: Base::d 0
0
请按任意键继续. . .
```

```
struct Base {
    void f() { cout << "Base f()" << endl; }
};
struct D1: Base {
    void f(int i) { cout<<"D1 f(int)"<<endl; }
};
int main() {
    Base b1;
    b1.f(); // Base f()
    //b1.f(1); //错,Base类中没有 f(int)函数
    D1 d1;
    d1.f(10); // D1 f(int)
    //d1.f(); //错,Base继承下来的f()被遮掩(shadow)
    d1.Base::f(); //要调用Base的f(),要加Base::
    return 0;
}
```

```
Base f()
D1 f(int)
Base f()
请按任意键
```



声明在内层作用域的函数不会重载声明在外层作用域的函数。

若在派生类中的成员与基类成员同名，则无论参数列表是否相同，基类成员都将被遮盖 (shadow)

# 继承：名字查找

```
class Base {
public:
    void f() { cout << "Base f()" << endl; }
    void f(double d) { cout << "Base f(double)" << endl; }
};
class D1:public Base {
public:
    void f(int i) { cout << "D1 f(int)" << endl; }
    void f(const string &s) { cout << "D1 f(string)" << endl; }
private:
    void f(short i) { cout << "D1 f(short)" << endl; }
};
```

代码区：函数

Base::f()
Base::f(double)
D1::f(int)
D1::f(const string &)
D1::f(short)

Base 作用域

f()
f(double)
D1 作用域
f(int)
f(const string &)
f(short)

**名字查找流程**：从**静态类型**中开始查找，若找到，再进行类型检查；若没有找到，则在它的直接基类中查找，直到继承链的顶端。若还是没有，则编译器报错。

```
int main() {
    Base b1;
    b1.f(); // Base f()
    b1.f(1.2); // Base f(double)
    D1 d1;
    d1.f(10); // D1 f(int)

    //d1.f(static_cast<short>(10)); //错,private类外无法访问
    //d1.f(); //错,Base继承下来的f()被遮掩(shadow)
    d1.Base::f(); //要调用Base的f(),要加Base::
    d1.Base::f(1); //Base f(int)

    //静态类型与动态类型不同时：
    Base &rb = d1;
    rb.f(); // Base f()
    rb.f(10); // Base f(double) 隐式转换10->10.0
    //rb.f(string("abc")); //错,Base中没有
    Base *pb = &d1;
    pb->f(); // Base f()
    pb->f(10.0); // Base f(double)
    //pb->f(string("abc")); //错,Base中没有
    return 0;
}
```

```
Base f<>
Base f(double)
D1 f(int)
Base f<>
Base f(double)
Base f<>
Base f(double)
Base f<>
Base f(double)
请按任意键继续
```

# 继承: using声明式

```
class Base {
public:
    void f() { cout << "Base f()" << endl; }
    void f(double d) { cout << "Base f(double)" << endl; }
    void f(int i, int j) { cout << "Base f(int,int)" << endl; }
};
class D1:public Base {
public:
    using Base::f;
    int f() { cout << "D1 f()" << endl; return 0; }
    void f(const string &s) { cout << "D1 f(string)" << endl; }
};
```

成员函数可以重载；

假如派生类中希望基类的所有重载版本都**可见**，那么必须覆盖所有的版本，或者一个也不覆盖。（麻烦！）

使用基类成员函数的**using声明**，可以解决这个问题。

```
Base f<>
Base f<double>
D1 f<>
Base f<double>
Base f<int,int>
-----
Base f<>
Base f<double>
Base f<>
Base f<double>
请按任意键继续.
```

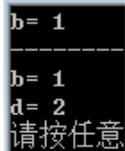
```
int main() {
    Base b1;
    b1.f(); // Base f()
    b1.f(1.2); // Base f(double)
    D1 d1;
    d1.f(); // D1 f() 是D1中的f()
    d1.f(1.2); // Base f(double)
    d1.f(1, 2); // Base f(int,int)
    cout << "-----" << endl;
    //静态类型与动态类型不同时：
    Base &rb = d1;
    rb.f(); // Base f()
    rb.f(10.1); // Base f(double)
    //rb.f(string("abc")); //错,Base中没有
    Base *pb = &d1;
    pb->f(); // Base f()
    pb->f(10.0); // Base f(double)
    //pb->f(string("abc")); //错,Base中没有
    return 0;
}
```

# 继承:友元

友元关系不能交换、不能传递，也不能继承。

```
class Base {
    friend ostream& operator<<(ostream &out, const Base &base);
    int b = 1;
};
class D1:public Base {
    friend ostream& operator<<(ostream &out, const D1 &d1);
    int d = 2;
};
ostream& operator<<(ostream &out, const Base &base) {
    out << "b= " << base.b;
    //out << base.d; //error, Base的友元只能访问Base的成员
    return out;
}
ostream& operator<<(ostream &out, const D1 &d1) {
    //out << d1.b; //error, D1的友元只能访问D1的成员
    out << static_cast<const Base&>(d1) << endl;
    out << "d= " << d1.d ;
    return out;
}
```

```
int main() {
    Base b1;
    cout << b1 << endl;
    cout << "-----" << endl;
    D1 d1;
    cout << d1 << endl;
    return 0;
}
```



```
b= 1
-----
b= 1
d= 2
请按任意
```

# 多态引入

```
class Person {
public:
    Person(int _age,const string& _name)
        :age(_age),name(_name){}

    virtual void show()const{
        cout << "我是" << name << ", "
            << age << "岁。" << endl;
    }
protected:
    int age;
    string name;
};

class Student:public Person {
public:
    Student(int _age,const string& _name,const string& _course)
        :Person(_age,_name),stu_course(_course){}
    void show()const {
        cout << "我是" << name << ", " << age <<
            "岁。我在学: " << stu_course << endl;
    }
private:
    string stu_course;
};
```

```
class Teacher :public Person {
public:
    Teacher(int _age,const string& _name,const string& _course)
        :Person(_age, _name), tea_course(_course) {}
    void show()const {
        cout << "我是" << name << ", " << age <<
            "岁。我在教: " << tea_course << endl;
    }
private:
    string tea_course;
};

void show(const Person& pp) {      注意pp的静态类型和动态类型
    pp.show();
}

int main() {
    Student stu1(22, "张三", "C++");
    show(stu1); //期望:我是张三,22岁。我在学C++。
    Teacher tea1(32, "李四", "C语言");
    show(tea1); //期望:我是李四,32岁。我在教C语言。
    return 0;
}
```

没有 virtual:

加上 virtual:

# 虚函数与多态

- 1: 在基类中用 **virtual** 声明成员函数为虚函数。类外实现虚函数时，不能加**virtual**。
- 2: 当基类的一个成员函数被声明为虚函数后，其派生类中完全相同的函数也是虚函数。**virtual**可写可不写，一般都加上**virtual**以示清晰。
- 3: 派生类中重新定义此函数称为覆写（**override**），要求**函数名**，**返回值类型**，**函数参数列表**全部一样。（返回值类型有例外，返回本类的指针或引用）
- 4: 派生类中覆写基类虚函数，为了避免写错，可加上**override**关键字来显式说明派生类中的虚函数，让编译器来帮助发现错误。如：`virtual void f() override;`
- 5: 可以将一个虚函数指定为**final**，拒绝派生类覆写(**override**)该函数。如 `virtual void f() final;`
- 6: 派生类中的覆写的虚函数，访问权限(**public/protected/private**)可由派生类自己决定。
- 7: 静态成员函数不能声明为虚函数，因为静态成员属于类，不专属于某个对象。
- 8: 内联函数不能声明为虚函数，因为内联函数在编译是已被明确的执行代码替换。

多态性(**polymorphism**): 具有继承关系的多个类型称为多态类型，可以使用这些类型的“多种形式”而无须在意它们的差异。

多态的条件:

- 1: 存在继承关系，基类中有虚函数
- 2: 派生类中有同名的虚函数，并且**override**基类虚函数
- 3: 通过已被子类对象赋值的**基类指针或引用**，调用虚函数。

# 动态绑定

---

非虚函数的调用是在**编译时绑定**的，直接根据静态类型决定调用哪个函数。

通过对象进行的虚函数调用也是在编译时绑定的。（**通过对象调用：决定了静态类型和动态类型一致**）

指针或引用的静态类型和动态类型有可能不一致。

通过基类的指针或引用调用基类中的虚函数时，编译时并不能确定该函数真正作用的对象是什么类型，直到运行时才会决定执行哪个函数版本，判断依据是**指针或引用所绑定对象的真实类型**。

当且仅当通过指针或引用调用虚函数时，才会在运行时解析该调用，也只有在这种情况下对象的动态类型可能和静态类型不同。

编译时绑定：静态绑定

运行时绑定：动态绑定

# 虚析构

```
class Base {
public:
    ~Base() { cout << "Base析构\n"; }
};
class D1 :public Base {
public:
    D1():pi(new int(0)){ }
    ~D1() { delete pi; cout << "delete\n"; }
}; int * pi;

class Base2 {
public:
    virtual ~Base2() { cout << "Base析构\n"; }
};
class D2 :public Base2 {
public:
    D2() :pi(new int(0)) { }
    ~D2() { delete pi; cout << "delete\n"; }
}; int * pi;
```

```
int main() {
    Base *pb = new D1;
    delete pb;
    //pb的静态类型是 Base,所以会调用 Base的析构
    //D1的析构没有运行到,内存泄漏
    cout << "-----\n";
    Base2 *pb2 = new D2;
    delete pb2;
    //因为Base2的析构函数是虚函数
    //所以在调用pb2的析构时,会调用它实际类型的析构
    //pb2的实际类型是D2的指针,正确释放内存
    return 0;
}
```



一个基类的析构函数总是设置为虚函数。  
可写为: `virtual ~A() = default;`  
三/五原则的特例。  
虚析构将阻止合成移动操作。

# 纯虚函数和抽象基类

**纯虚函数**: `virtual` 返回类型 函数名(参数列表) = 0;

含有纯虚函数的类，称为**抽象基类**，不可实例化(不能创建对象)，目的是提供公共接口。

如果一个基类中声明了纯虚函数，而在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，派生类仍然为抽象基类。

**虚函数与纯虚函数比较**:

虚函数：如果一个类中声明了虚函数，这个函数是实现的，它的作用是为了能让这个函数在他的子类里面能被**override**，这样就可以实现动态多态。

纯虚函数：只是一个接口，是个函数的声明而已，它留在子类中实现。

虚函数在派生类中可以不**override**。

纯虚函数必须在派生类中实现（若不实现，则派生类还是抽象基类）。

虚函数的类用于“**实现继承**”，即**继承接口的同时也继承了基类的实现**。

纯虚函数用于“**接口继承**”，即纯虚函数关注的是接口的统一性，实现由派生类完成。

带纯虚函数的类叫做抽象基类，这种类不能直接生成对象。

# 多态例子: shape面积

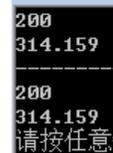
```
class Shape { //抽象基类
public:
    //纯虚函数(接口)
    virtual double area()const = 0;
}; virtual ~Shape() = default;//虚析构

class Rectangle :public Shape {
public:
    Rectangle(double w, double h)
        :width(w), height(h) {}
    virtual double area()const override{
        return width*height;
    }
private:
    double width;
}; double height;

class Circle :public Shape {
public:
    Circle(double r) :radius(r) {}
    virtual double area()const override {
        return 3.1415926*radius*radius;
    }
private:
    double radius;
};
```

```
double get_area(const Shape& sp) {
    return sp.area();
}

int main() {
    Rectangle r1(10.0, 20.0);
    Circle c1(10.0);
    Shape *p = nullptr;
    p = &r1;
    cout << p->area() << endl; //200
    p = &c1;
    cout << p->area() << endl; //314.159
    cout << "-----\n";
    cout << get_area(r1) << endl;//200
    cout << get_area(c1) << endl;//314.159
    return 0;
}
```



```
200
314.159
-----
200
314.159
请按任意
```

通过基类的指针或引用调用基类中的**虚函数**时，编译时并不能确定该函数真正作用的对象是什么类型，直到运行时才会决定执行哪个函数版本，判断依据是**指针或引用所绑定对象的真实类型**。

# 多态例子:USB设备

```
class USB { //抽象基类
public:
    virtual void connect() = 0;
    virtual void dis_connect() = 0;
    virtual ~USB() {} //虚析构
};

class UDisk : public USB {
public:
    virtual void connect() override {
        cout << "U connect" << endl; }
    virtual void dis_connect() override {
        cout << "U disconnect" << endl; }
};

class Mouse : public USB {
public:
    virtual void connect() override {
        cout << "M connect" << endl; }
    virtual void dis_connect() override {
        cout << "M disconnect" << endl; }
};

class Keyboard : public USB {
public:
    virtual void connect() override {
        cout << "K connect" << endl; }
    virtual void dis_connect() override {
        cout << "K disconnect" << endl; }
};
```

```
//void testU(UDisk *u){
// u->connect();
// u->dis_connect();
//}
//void testM(Mouse *m){
// m->connect();
// m->dis_connect();
//}
//void testK(Keyboard *k){
// k->connect();
// k->dis_connect();
//}

//参数类型设置为抽象基类类型的指针 }
//可接收派生类指针
//利用多态根据实际类型调用相应的方法
//减少了冗余的代码
```

```
void test(USB *u) {
    u->connect();
    u->dis_connect();
}
```

```
int main() {
    UDisk *u = new UDisk;
    Mouse *m = new Mouse;
    Keyboard *k = new Keyboard;
    test(u); test(m); test(k);
    delete u; delete m; delete k;
    cout << "-----\n";
    USB *u1 = new UDisk;
    USB *m1 = new Mouse;
    USB *k1 = new Keyboard;
    test(u1); test(m1); test(k1);
    delete u1; delete m1; delete k1;
    return 0;
}
```

```
U connect
U disconnect
M connect
M disconnect
K connect
K disconnect
-----
U connect
U disconnect
M connect
M disconnect
K connect
K disconnect
请按任意键继续
```

# 练习1

```
class Base {
public:
    virtual void func() = 0;
    virtual void func(double) {
        cout << "Base(double)\n";
    }
    void func(int) {
        cout << "Base(int)\n";
    }
};
class D1 : public Base {
public:
    virtual void func() override {
        cout << "D1()\n";
    }
    void func(const char*) {
        cout << "D1(const char*)\n";
    }
};
class D2 :public D1 {
public:
    using Base::func;
    void func(bool) {
        cout << "D2(bool)\n";
    }
};
```

```
int main() {
    //Base b1;
    D1 d1;
    d1.func();
    //d1.func(1.2);
    //d1.func(1);
    d1.func("abc");
    D2 d2;
    d2.func();
    d2.func(1.2);
    d2.func(1);
    d2.func("123");
    d2.func(true);
    cout << "-----\n";
    Base *p1 = new D1;
    p1->func();
    p1->func(1.2);
    p1->func(1);
    //p1->func("123");
    p1->func(true);
    D1 *p2 = new D2;
    p2->func();
    //p2->func(1.2);
    //p2->func(1);
    p2->func("123");
    //p2->func(true);
    return 0;
}
```

//问题1：错误,有纯虚函数,抽象基类无法创建对象

//问题2：正确, D1 override了func(), 输出：D1()

//问题3：错误, D1重新定义了func,shadow了基类的所有func函数

//问题4：错误, 同问题3

//问题5：正确 D1(const char\*)

//问题6：ok,using Base::func,将Base的所有func函数的重载实例添加到D2的作用域, D1() [注意

//问题7：同问题6, Base(double)

//问题8：同问题6, Base(int)

//问题9：ok(注意),D1中的func(const char\*)被shadow,隐式转换执行func(bool) D2(bool)

//问题10：ok D2(bool)

//问题11：虚函数调用,真实类型的调用 D1()

//问题12：虚函数调用,但是D1没有override Base(double)

//问题13：非虚函数调用,静态类型的调用 Base(int)

//问题14：错误,静态类型中找不到 func(const char\*)

//问题15：隐式转换 Base(int)

//问题16：虚函数调用,真实类型的调用 D1() (D2没有override func(),所以是D1中的)

//问题17：错误,静态类型D1中没有 func(double),也没有可以隐式转换匹配的

//问题18：错误,同上

//问题19：D1(const char\*)

//问题20：错误,静态类型D1中没有 func(bool),也没有可以隐式转换匹配的

```
D1<>
D1(const char*)
D1<>
Base(double)
Base(int)
D2(bool)
D2(bool)
-----
D1<>
Base(double)
Base(int)
Base(int)
D1<>
D1(const char*)
请按任意键继续.
```

# 练习2

```
#include <iostream>
#include <memory>
using namespace std;
class Base {
public:
    Base(int i):p_base(new int(i)){}
    ~Base() { delete p_base; }
    virtual void show()const {
        cout << "Base: " << *p_base << endl;;
    }
private:
    int *p_base;
};
class D1 : public Base {
public:
    D1(int i,int j):Base(i),p_d1(new int(j)){}
    ~D1() { delete p_d1; }
    virtual void show()const override {
        cout << "D1: " << *p_d1 << endl;;
    }
private:
    int *p_d1;
};
```

```
void print(const Base &b) {
    b.Base::show();
    b.show();
}

int main() {
    auto pd1 = make_shared<D1>(1, 2); //pd1的类型是什么
    print(*pd1); //运行结果?
    //智能指针,一样有多态
    unique_ptr<Base> pd2 = make_unique<D1>(3, 4);
    print(*pd2); //运行结果?

    return 0; //这个代码整体上有什么问题?
}
```

```
Base: 1
D1: 2
Base: 3
D1: 4
请按任意键
```

# 练习3

```
class Base{
public:
    virtual void f1()const{
        cout << "Base::f1()\n";
    }
    void f2()const{
        f1();
        cout << "Base::f2()\n";
    }
    virtual ~Base() = default;
};

class D1 : public Base{
public:
    virtual void f1()const override{
        cout << "D1::f1()\n";
    }
    virtual void f2()const{
        cout << "D1::f2()\n";
    }
};
```

```
int main(){
    Base *p = new D1;
    p->f2(); //输出结果是什么?
    delete p;
    return 0;
}
```

```
D1::f1<>
Base::f2<>
请按任意键继续. . .
```