多态与虚函数

基类和派生类对象之间赋值兼容关系

凡是基类对象出现的场合，都可以用公有派生类对象来替换。

派生类对象可以赋值给基类对象

例子如下:

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

class Base {
public:
    void display() const {
        cout << "Base display" << endl;
    }
};

class Derived : public Base {
public:
    void display() const {
        cout << "Derived display" << endl;
    }
};

int main() {
    Derived d;
    Base b = d;
    b.display();
    return 0;
}
```

将派生类对象的地址赋值给基类指针

```
class A {
public:
    A(int a): m_a(a) {}
    void display() const { cout << "A: m_a = " << m_a << endl; }
    int m_a;
};

class B: public A {
public:
    B(int a, int b): A(a), m_b(b) {}
    void display() const { cout << "B: m_a = " << m_a << ", m_b = " << m_b <<
endl; }
    int m_b;
};

int main() {
    A *a = new A(1);
```

```
    B *b = new B(2, 3);
    // 基类对象的指针指向派生类对象
    // 通过a只能访问派生类对象的m_a，不能访问m_b
    // 通过a访问的是类A的display函数
    a = b;
    return 0;
}
```

派生类对象可以初始化基类对象的引用

```
class A {
public:
    A(int a): m_a(a) {}
    void display() const { cout << "A: m_a = " << m_a << endl; }
    int m_a;
};

class B: public A {
public:
    B(int a, int b): A(a), m_b(b) {}
    void display() const { cout << "B: m_a = " << m_a << ", m_b = " << m_b <<
endl; }
    int m_b;
};

int main() {
    B b(2, 3);
    // 通过a可以访问b的m_a，但是不能访问b的m_b
    // 通过a访问的是类A的display函数
    A &a = b;  // a是基类对象的引用，b是派生类对象，b可以用来初始化a
    return 0;
}
```

函数的形参是基类对象或基类对象的引用

```
class A {
public:
    A(int a): m_a(a) {}
    void display() const { cout << "A: m_a = " << m_a << endl; }
protected:
    int m_a;
};

class B: public A {
public:
    B(int a, int b): A(a), m_b(b) {}
    void display() const { cout << "B: m_a = " << m_a << ", m_b = " << m_b <<
endl; }
protected:
    int m_b;
};

void f(A &a) { a.display(); }  // 如果传入的是派生类对象，通过a访问的是类A的display函数
```

```
int main() {
    B b(2, 3);
    f(b);
    return 0;
}
```

下面这个例子就有问题了

```
class Pet {
public:
    void speak() { cout << "how does a pet speak?" << endl; }
};

class Dog: public Pet {
public:
    void speak() { cout << "wang!" << endl; }
};

class Cat: public Pet {
public:
    void speak() { cout << "miao!" << endl; }
};

void hello(Pet &pet) { pet.speak(); }

int main() {
    Pet *pet1;
    Dog dog; Cat cat;
    pet1 = &dog; pet1->speak();
    Pet &pet2 = cat; pet2.speak();
    hello(cat);
    return 0;
}
```

这个的结果会是

```
how does a pet speak?
how does a pet speak?
how does a pet speak?
```

那有什么方法可以解决这个问题呢？虚函数!

虚函数

virtual关键字修饰的成员函数称为虚函数。

```
virtual void display() const { cout << "Base display" << endl; }
```

解决形式如下:

```
class Pet {
public:
```

```
    virtual void speak() { cout << "how does a pet speak?" << endl; }  // 基类中定义虚函数
};

class Dog: public Pet {
public:
    void speak() { cout << "wang!" << endl; }  // 派生类重写的函数，自动变成虚函数，virtual可省略
};

class Cat: public Pet {
public:
    void speak() { cout << "miao!" << endl; }  // 派生类重写的函数，自动变成虚函数，virtual可省略
};

void hello(Pet &pet) { pet.speak(); }  // speak为虚函数，运行时发现pet引用的是Cat类的对象

int main() {
    Pet *pet1;
    Dog dog; Cat cat;
    // speak为虚函数，运行时发现pet1指向Dog类的对象，于是调用Dog类的speak函数
    pet1 = &dog; pet1->speak();
    Pet &pet2 = cat; pet2.speak();  // speak为虚函数，运行时发现pet2引用的是Dog类的对象
    hello(cat);  // 传入Cat类的对象
    return 0;
}
```

虚析构函数

C++中不能将构造函数定义为虚函数，但是可以讲析构函数定义为虚函数。

还是上面的例子，如果将析构函数定义为虚函数，那么delete pet1和pet2时，会调用Dog类的析构函数，而不是Pet类的析构函数。全代码如下:

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

class Pet {
public:
    Pet() { cout << 1 << endl; }
    ~Pet() { cout << 2 << endl; }
};

class Dog : public Pet {
public:
    Dog() : Pet() { cout << 3 << endl; }
    ~Dog() { cout << 4 << endl; }
};

int main() {
    Dog dog;
    return 0;
}
```

这时候的输出结果是

```
1
3
4
2
```

此时修改为

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

class Pet {
public:
    Pet() { cout << 1 << endl; }
    ~Pet() { cout << 2 << endl; }
};

class Dog : public Pet {
public:
    Dog() : Pet() { cout << 3 << endl; }
    ~Dog() { cout << 4 << endl; }
};

int main() {
    Pet *pet1 = new Dog();
    delete pet1;
    return 0;
}
```

这时候的输出结果是

```
1
3
2
```

那该怎么做呢?

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

class Pet {
public:
    Pet() { cout << 1 << endl; }
    virtual ~Pet() { cout << 2 << endl; } // 虚析构函数
};

class Dog : public Pet {
public:
    Dog() : Pet() { cout << 3 << endl; }
    ~Dog() { cout << 4 << endl; } // 派生类中的virtual可省略，这里也是虚析构函数
};
```

```
int main() {
    Pet* pet = new Dog;
    // 析构函数是虚函数，运行时发现pet指向的是Dog类的对象，调用Dog类的析构函数
    delete pet;
    return 0;
}
```

这样的输出结果是

```
1
3
4
2
```

这就是虚析构函数的作用。

多态性

联编：把函数名与函数体连接在一起的过程称为联编。通俗的讲,就是确定某个同名标识到底要调用哪个函数

静态联编：编译阶段进行的联编称为静态联编。即编译时,通过函数重载实现如

```
void display(int a) { cout << "display int" << endl; }
void display(double a) { cout << "display double" << endl; }
```

```
class Pet {
    virtual void speak() { cout << "how does a pet speak?" << endl; }
};

class Dog {
    void speak() { cout << "wang!wang!" << endl; }
};

Pet pet;
Dog dog;
Pet* pPet;

pPet = &pet; pPet->speak(); // speak为虚函数，在运行时才决定调用Pet类的speak函数
pPet = &dog; pPet->speak(); // speak为虚函数，在运行时才决定调用Dog类的speak函数
```

多态性：向不同对象发送同一消息，不同对象在接收时会产生不同的行为。直观理解就是，同样是`speak`，`Pet`类和`Dog`类调用时会产生不同的行为。

纯虚函数和抽象类

```
class Pet { // 包含纯虚函数的类叫抽象类
public:
    virtual void speak() = 0; // 纯虚函数，只有函数声明，没有实现
};

class Dog : public Pet {
public:
```

```
    void speak() { cout << "wang!" << endl; }
};

int main() {
    // Pet pet; // 报错！抽象类不能实例化(创建对象)，因为纯虚函数没有实现
    Dog dog;
    Pet* pet1 = &dog; // 正确，抽象类的指针，可以指向派生类的对象
    pet1->speak(); // speak是虚函数，运行时决定调用Dog类的speak函数

    Pet& pet2 = dog; // 正确，抽象类的引用，可以引用派生类的对象
    pet2.speak(); // speak是虚函数，运行时决定调用Dog类的speak函数
    return 0;
}
```

抽象类中除了可以有虚函数外，还可以有数据成员和成员函数。

```
class Pet {
public:
    virtual void speak() = 0; // 纯虚函数
    int m_a; // 数据成员
    void display() { cout << "Pet display" << endl; } // 成员函数
};
```

抽象类派生出子类后，子类必须实现纯虚函数，否则子类也是抽象类。

```
class Pet {
public:
    virtual void speak() = 0; // 纯虚函数
};

class Dog : public Pet { // 子类没有实现纯虚函数，所以Dog类也是抽象类
public:
    void display() { cout << "Dog display" << endl; }
};
```