OOP 之继承

--面向对象的第二个特征是：继承

类的派生

在编程的时候，经常遇到具有类似属性，但是细节或者行为存在差异的组件。在这种情形下，一种解决方案是将每个组件声明为一个类，并在每个类中实现所有的属性，这将导致大量重复的代码。另一种解决方案是使用继承，从同一个基类派生出类似的类，在基类中实现所有通用的功能，并在派生类中覆盖基本的功能，以实现让每个类都独一无二的行为。

C++派生语法如下：

```
//基类
class Base {
    // Base class members
};

//派生类
class Derived: public Base {
    // derived class members
};
```

前面我们写了一个 Person 类，如果我们需要编写 Student 和 Worker 类，必然也需要 Person 的三个属性：age/name/id，这时候我们采用继承来解决这个问题。

我们编写代码如下：

```
class Student: public Person {
```

```
private:
    string _school;

};

class Worker: public Person {
private:
    string _factory;

};
```

这里我们让 Student 和 Worker 继承了 Person，而且添加了各自的成员。继承之后的类具有两部分成员：一是从基类继承而来的成员，二是自己本身添加的成员。

我们在 main 中测试一下：

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

class Person {
private:
    int _id;
    string _name;
    int _age;

public:

    Person() :
            _id(-1), _name("none"), _age(-1) {
    }

    Person(int id, const string &name, int age) :
            _id(id), _name(name), _age(age) {
    }

    void print(std::ostream &os) const {
        os << "id: " << _id << " name: " << _name << " age: " << _age << endl;
    }
};

class Student: public Person {
```

```
private:
    string _school;

};

class Worker: public Person {
private:
    string _factory;

};

int main() {

    Student s;
    s.print(cout);

}
```

可以看到 Student 的对象也可以正常调用 print 函数，说明 Student 类本身也具有这个函数。

protected 关键字

上面的代码中，我们做一些改动，在 Student 中添加一个 test 方法，里面去改动 name 属性，如下：

```
class Student: public Person {
private:
    string _school;

public:
    void test(){
        _name = "test";
    }
};
```

编译报错：‘std::string Person::_name’ is private，这是因为 name 为 private 属性，只能在 Person 类内部访问，即使是派生类也无法访

问。

如果需要修改基类的某个属性，可以把基类的属性设为 protected。protected 关键字的含义是：该属性在本类和派生类的内部可以访问。

这里我们总结下三种访问权限：

假设有有 base 类和 derived 类（public 继承）：

private 成员作用域仅限于 base 内部

public 成员作用域在所有位置

protected 成员作用在 base 和 derived 内部

我们这里采用的继承方式为 public 继承，如果一个类里面有 public、private、protected 成员，他们经过 public 继承，在派生类中的访问标号为：public、不可访问、protected。

总结起来就是：

Private 为个人日记，在派生类中无法被访问

Protected 为家族秘籍，在派生类家族体系中可以访问

Public 为完全公开的东西，在派生类中可以随意的访问

Private 在派生类的内部不可见，但是通过基类本身的函数可以间接访问

继承体系下的函数调用

1.派生类调用从基类继承而来的函数

我们刚才见过，Student 可以正常调用继承来的 print 函数。

2.派生类自己额外添加的函数

我们为 Student 添加一个 test 函数。如下：

```
class Student: public Person {
private:
    string _school;

public:
    void test(){
        cout << "test" << endl;
    }
};
```

在 main 中调用 s.test() 仍然是正常的，实际上，对 test 的调用和继承体系无关。

3.派生类重写了从基类继承而来的函数

我们在 Student 中重写 print 函数。

不改变参数列表：

```
class Student: public Person {
private:
    string _school;

public:
    void test() {
        cout << "test" << endl;
    }
    void print(std::ostream &os) const {
        os << "print" << endl;
    }
};
```

打印结果为 print，说明调用的是我们自己定义的版本。

如果改变参数列表：

```
class Student: public Person {
private:
    string _school;
```

```
public:
    void test() {
        cout << "test" << endl;
    }
    void print() const {
        cout << "print" << endl;
    }

};
```

运行程序可知，此时仍然调用我们重写的版本。

如果两个函数都存在，如下：

```
class Student: public Person {
private:
    string _school;

public:
    void print() const {
        cout << "print1" << endl;
    }

    void print(std::ostream &os) const {
        os << "print2" << endl;
    }

};
```

在 main 中这样写：

```
Student s;
s.print();
s.print(cout);
```

打印结果为：

```
print1
print2
```

同样说明调用了我们重写的版本。这里无论哪种情况，只要我们在派生类中写了一个函数，和基类的函数重名（无论参数表是否相同），那么通过派生类对象调用的总是派生类重写的函数，我们称派生类的 print 隐藏了基类的 print。

那么基类的 print 就无法调用了吗？显然不是。我们对两个 print 做改动如下：

```
class Student: public Person {
private:
    string _school;

public:
    void print() const {
        cout << "print1" << endl;
        Person::print(cout);
    }

    void print(std::ostream &os) const {
        os << "print2" << endl;
        Person::print(os);
    }

};
```

打印结果为：

```
print1
id: -1 name: none age: -1
print2
id: -1 name: none age: -1
```

我们还可以在外面显式调用，在 main 中这样写：

```
s.Person::print(cout);
```

打印结果仍然是正常的。这就说明了被派生类隐藏的函数可以通过指定基类的类名来调用。

继承时的对象布局

派生类内部含有一个无名的基类对象，下面才是派生类自己的成

员，所以构造派生类时会先构造基类。

例如：

```
class Base {
public:
    Base() {
        cout << "Base create" << endl;
    }
};

class Derived: public Base {
public:
    Derived() {
        cout << "Derived create" << endl;
    }
};
```

在 main 中生成一个 Derived 对象，观察对象构造的顺序。

练习：自己写程序添加析构函数，观察析构的顺序。

派生类到基类的转化

根据上面的对象布局，派生类可以赋值给基类。例如：

```
Derived d;

Base b;

b = d;
```

这段代码是可行的，但是程序在运行的时候只复制了 derived 的基类部分，而对于派生类单独的部分被切除了，这种无意间裁剪数据导致 Derived 变成 Base 的行为成为切除。

要避免切除问题，最好的办法是传递 const 引用或者指针。

构造和析构顺序

前面提到派生类和基类的构造顺序，如果一个派生类还有其他类的对象做成员，那么基类构造函数、派生类构造函数和成员对象的构造函数，三者的执行顺序是怎样的？

```
class Base {
public:
    Base() {
        cout << "Base create" << endl;
    }

    ~Base() {
        cout << "Base destroy" << endl;
    }
};

class Other {
public:
    Other() {
        cout << "Other create" << endl;
    }
    ~Other() {
        cout << "Other destroy" << endl;
    }
};


class Derived: public Base {
public:
    Derived() {
        cout << "Derived create" << endl;
    }
    ~Derived() {
        cout << "Derived destroy" << endl;
    }
private:
    Other _other;
```

```
};
```

在 main 中生成 Derived 的对象，可以观测到他们的构造和析构顺序。

继承与构造函数

在前面的代码中，我们分别为 Person 提供了两个构造函数，分别是：

Person();

Person(int id, const string &name, int age);

现在我们想为 Student 提供两个构造函数，分别为：

Student(int id, const string name, int age, const string &school)和

Student();

第二个默认无参数的很容易，全部为空就可以了：

```
Student() :
        _school("none") {
}
```

但是第一个含有参数的构造函数，我们的目的是把前三个参数传给基类继承而来的三个属性。但是三个属性为 private，也就是说在 Student 中无法直接访问三个属性。

这里的解决方案是采用初始化列表，这样做：

```
Student(int id, const string name, int age, const string &school) :
        Person(id, name, age), _school(school) {

}
```

继承与复制控制

这里涉及到两个函数，拷贝构造函数和赋值运算符。手工实现这两者都必须显式调用基类的版本。

看下面的代码：

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

class Person {
private:
    int _id;
    string _name;
    int _age;

public:

    Person() :
        _id(-1), _name("none"), _age(-1) {
    }

    Person(int id, const string &name, int age) :
        _id(id), _name(name), _age(age) {
    }

    void print(std::ostream &os) const {
        os << "id: " << _id << " name: " << _name << " age: " << _age <<
endl;
    }
};

class Student: public Person {
private:
    string _school;
public:
    Student() :
        _school("none") {
    }

    Student(int id, const string name, int age, const string &school) :
        Person(id, name, age), _school(school) {
```

```
    }

    Student(const Student &s) :
            _school(s._school) {

    }

    Student &operator=(const Student &s) {
        if (this != &s) {
            _school = s._school;
        }
        return *this;
    }

    void print(std::ostream &os) const {
        Person::print(os);
        os << _school << endl;
    }

};

class Worker: public Person {
private:
    string _factory;

};

int main() {

    Student s1(12, "zhangsna", 234, "test1");
    s1.print(cout);
    Student s2(s1);
    s2.print(cout);

    Student s3;
    s3 = s1;
    s3.print(cout);
}
```

最后的打印结果表明程序没有正常复制和赋值，因为我们在 Student 中没有显式调用 Person 的复制构造函数或者赋值运算符。

正确的做法是这样：

```
Student(const Student &s) :
        Person(s), _school(s._school) {

}

Student &operator=(const Student &s) {
    if (this != &s) {
        Person::operator=(s);
        _school = s._school;
    }
    return *this;
}
```

在复制构造函数中，我们在初始化列表中用 Student 对象 s 初始化基类，在赋值运算符中，则是显式调用了基类的赋值运算符。

禁止复制

在复制控制一节，我们学到禁止一个类复制的做法是将其拷贝构造函数和赋值运算符设为私有，而且只有声明，没有实现。

如果我们这里有 10 个类都需要禁止复制，那么可以每个类都进行上面的操作，但这样导致大量的重复代码，好的解决方案是采用继承。

我们写一个类：

```
class noncopyable {
public:
    noncopyable() {
    }
    ~noncopyable() {
    }
private:
    noncopyable(const noncopyable &);
    noncopyable &operator=(const noncopyable &);
};
```

这样凡是继承该类的类均失去了复制和赋值的能力。