OOP 之动态绑定

--面向对象的第三个特征是：动态绑定

基类的指针或者引用指向派生类对象

看下面的程序：

```
Student s1(12, "zhangsna", 234, "test1");
Person *p = &s1;
```

完整代码：

```
#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) :
            Person(s), _school(s._school) {

    }

    Student &operator=(const Student &s) {
        if (this != &s) {
            Person::operator=(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");
    Person *p = &s1;
}
```

编译通过，说明基类的指针可以指向派生类的对象。我们用这个

指针去调用程序，分下列几种情况：

1.调用基类中存在派生类没有改写的函数

我们把 Student 中的 print 注释掉。

```
Student s1(12, "zhangsna", 234, "test1");
Person *p = &s1;
p->print(cout);
```

打印结果，发现调用的是基类的版本

2.调用基类中不存在派生类单独添加的函数

我们在 Student 中单独添加一个函数 test：

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

然后在 main 中用指针调用：

```
Student s1(12, "zhangsna", 234, "test1");
Person *p = &s1;
p->test();
```

发现编译失败，错误信息为：

error: ‘class Person’ has no member named ‘test’

3.调用派生类改写的函数（分参数相同和不同）

取消掉 print 的注释，然后添加另外一个无参数的 print。

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

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

运行程序，发现调用的依然是基类的版本

原因在于：我们可以让基类 Person 的指针指向 Student 对象，但

是这种情况下我们用指针 p 进行函数调用时，编译器以为指针 p 指向的其实是一个 Person 对象。所以上面的各种结果都有了合理的解释。

.基类指针和派生类指针之间的相互转化

1.既然基类的指针可以指向派生类，那么派生类的指针能否转化为基类指针？

```
Student s1(12, "zhangsna", 234, "test1");
Student *ps =&s1;
Person *p = ps;
```

编译完全无问题，说明派生类指针可以自动转化为基类的指针，甚至不需要强制类型转换。

2.基类的指针能否转化为派生类指针？

```
Student s1(12, "zhangsna", 234, "test1");
Person *p = &s1;
Student *ps = p;
```

编译报错：error: invalid conversion from ‘Person*’ to ‘Student*’

加上强制类型转化：

```
Student *ps = (Student*)p;
```

此时编译通过。这种把基类指针转化为派生类指针的行为，需要进行强制类型转化，我们称其为“向下塑形”，向下塑形本质上是不安

全的，需要程序员人工保证安全性。

虚函数

前面我们用基类的指针指向派生类的对象，派生类改写了基类的 print 函数，结果指针调用 print，仍然是 Person 的版本，这是因为指针 p 把它指向的对象看作是一个 Person 对象。

我们现在做一些改动，在基类的 print 声明前加一个关键字 virtual，代码如下：

```
#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) {
    }

    virtual 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) :
                Person(s), _school(s._school) {

        }

        Student &operator=(const Student &s) {
            if (this != &s) {
                Person::operator=(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");
        Person *p = &s1;
        p->print(cout);
    }
```

此时我们发现，print 调用的居然是 Student 的版本！

这种现象就叫做动态绑定，也就是所谓的多态。

用 virtual 修饰的函数就叫做虚函数。

静态绑定

回想函数重载部分，我们提过，构成一个函数唯一标示的要素有：函数名、形参列表，在类中还包括类名、const 属性，但是从不包括返回值。这些要素可以构成函数的唯一标示：函数签名。

事实上，对于我们以往的各种函数调用，编译器在编译期间就能根据调用的特征跟函数签名做对比，从而在编译器就能确定调用哪一个函数。

这种编译器间的绑定行为称为静态绑定，又称为静态联编。

动态绑定的定义和条件

我们前面提到了多态的情况，面向对象的多态就是动态绑定。这里需要两个条件：

1.调用的函数为虚函数

2.用基类的指针指向派生类的对象

满足了上面两个条件，编译器在碰到 p->print(cout)这类语句时，如果是以前的情况，那么编译器会根据 p 的类型是 Person*，print 是 Person

的成员函数而唯一确定其函数调用的版本。

但是在这里，因为满足了动态绑定的条件，编译器察觉到 print 是一个虚函数，所以编译器在编译时并不确定究竟调用哪个 print 版本，而是把这件事推迟到运行期，根据 p 实际指向的对象来决定调用哪个版本。

这种把函数调用的绑定行为推迟到运行期间的现象就叫做动态绑定。

总结如下：

当用 base 指针或引用操纵派生类对象时，如果调用的是虚函数，那么在运行期间根据指针指向的对象的实际类型来确定调用哪一个函数。如果不是虚函数，在编译期间，根据指针或引用的类型就可以确定调用哪一个函数。

注意：虚函数具有继承性，如果基类中具有某个 virtual 函数，派生类具有同名函数而且参数相同，返回值兼容基类版本，那么派生类的也是虚函数。

函数的覆盖和隐藏

有下列三种情况：

1.基类中有 virtual 函数 print，派生类中也有 print，参数相同。这

是动态绑定的情况。这叫做函数的覆盖。

2.基类中有 virtual 函数，派生类中也有同名函数，但是不满足动态绑定的情况。

3.基类中有非 vitual 函数，派生类具有同名函数，无论参数是否相同。

后两者称为函数的隐藏。

虚函数表

动态绑定的原理

虚指针

虚函数表

抽象类和纯虚函数

不能实例化的类称为抽象类。

虚析构函数

对于含有虚函数的继承体系，通常需要把基类的析构函数改为 virtual 函数。

如果没有声明为 virtual 会怎样？

例如 Person *p = &s;当执行 delete p 的时候，这里是静态绑定，所以调用的是 Person 的析构函数，Student 并没有被完整销毁。

如果我们声明为 virtual，那么根据动态绑定，p 指向的是 Student 对象，这么 delete 调用的是 Student 的析构函数。

练习：写程序验证：

```
#include <iostream>
using namespace std;
class Base {
public:
    ~Base() {
        cout << "Base destroy" << endl;
    }
};

class Derived: public Base {
public:
    ~Derived() {
        cout << "Derived destroy" << endl;
    }
};

int main() {

    Base *pb = new Derived;

    delete pb;
}
```

把 Base 函数设为 virtual，再次观察结果。

接口继承/实现继承

类的设计清单：

作者： 郭春阳 OOP 之动态绑定

1.类是否需要自己实现构造函数？

2.类的数据成员是否为 private？

3.每个构造函数是否初始化了所有的数据？

4.类是否需要析构函数？

5.类的析构函数是否需要为 virtual？

6.类是否需要拷贝构造函数？

7.类是否需要赋值运算符？

8.赋值运算符有没有正确处理自身赋值？

9.类是否需要一些相关联的操作符？

10.删除数组时是否调用了 delete[]？

11.在拷贝构造函数和赋值运算符的形参是否使用了 const？

12.当函数参数中有 reference 时，是否应该为 const？

13.是否适当的将类的成员函数声明为 const？

一个课堂练习：

当前有基类 Computer，它有一个纯虚函数 price 和 display，下面派生出许多品牌，那么某职员去采购不同品牌的电脑，如果把这不同品牌的电脑放入到一个容器中，从而方便的计算总价格？