一、 继承

1. 什么是继承

> 继承是类与类之间的关系，是一个很简单很直观的概念，与现实世界中的继承类似，例如儿子继承父亲的财产。
>
> 继承（Inheritance）可以理解为一个类从另一个类获取成员变量和成员函数的过程。例如B类 继承于A类，那么 B 就拥有 A 的成员变量和成员函数。被继承的类称为父类或基类，继承的类称为子类或派生类。 子类除了拥有父类的功能之外，还可以定义自己的新成员，以达到扩展的目的。
>
> **注意：并不是所有的类都能存在继承关系。必须满足：is a 判断**

1. is A 和 has A

> 大千世界，不是什么东西都能产生继承关系。只有存在某种联系，才能表示有继承关系。如：哺乳动物是动物，狗是哺乳动物，因此，狗是动物，等等。所以在学习继承后面的内容之前，先说说两个术语 `is A` 和 `has A` 。

- is A

> 是一种继承关系，指的是类的父子继承关系。表达的是一种方式：这个东西是那个东西的一种。例如：长方体与正方体之间--正方体是长方体的一种。正方体继承了长方体的属性，长方体是父类，正方体是子类。

- has A

> has-a 是一种组合关系，是关联关系的一种（一个类中有另一个类型的实例），是整体和部分之间的关系（比如汽车和轮胎之间），并且代表的整体对象负责构建和销毁部分对象，代表部分的对象不能共享。

2. 继承入门

> 通常在继承体系下，会把共性的成员，放到父类来定义，子类只需要定义自己特有的东西即可。或者父类提供的行为如不能满足，那么子类可以选择自定重新定义。

```cpp
#include <iostream>
#include <string>

using namespace std;

//父类
class Person{
    public:
        string name;
        int age ;
};


//子类
class Student:public Person{

};

int main() {

    //子类虽然没有声明name 和 age ，但是继承了person类，等同于自己定义的效果一样
    Student s;
    s.name = "张三";
    s.age = 18;

    cout << s.name << " = " << s.age << endl;


    return 0 ;
}
```

3. 访问权限回顾

> 当一个类派生自基类，该基类可以被继承为 **public、protected** 或 **private** 几种类型。继承类型是在继承父类时指定的。 如： `class Student : public Person` 。 我们几乎不使用 **protected** 或 **private** 继承，通常使用 **public**继承。

- public

> 表示公有成员，该成员不仅可以在类内可以被访问，在类外也是可以被访问的，是类对外提供的可访问接口；

- private

> 表示私有成员，该成员仅在类内可以被访问，在类的外面无法访问；

- protected

> 表示保护成员，保护成员在类的外面同样是隐藏状态，无法访问。但是可以在子类中访问。

1. 公有继承（public）

> 基类成员在派生类中的访问权限保持不变，也就是说，基类中的成员访问权限，在派生类中仍然保持原来的访问权限；

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

class person{
    public:
            string name;

    private:
            int age;
};

class student:public person{
    //name 和 age保持原有访问权限。
};

int main(){

    student s;
    s.name = "张三" ;
    s.age = 18 ; //编译错误！ 无法访问 age

    return 0 ;
}
```

2. 私有继承（private）

> 基类所有成员在派生类中的访问权限都会变为私有(private)权限；

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

class person{
    public:
            string name;

    private:
            int age;
};

class student:private person{
    //name 和 age保持全部变成private权限
};

int main(){

    student s;
    s.name = "张三" ;//编译错误！ 无法访问 name
    s.age = 18 ; //编译错误！ 无法访问 age

    return 0 ;
}
```

3. 保护继承（protected)

> 基类的共有成员和保护成员在派生类中的访问权限都会变为保护(protected)权限，在子类中具有访问权限，但是在类的外面则无法访问。

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

class person{
public:
    string name;

private:
    int age;
};

class student: protected person{
    //name 和 age保持原有访问权限。

public:
    void read(){
        s.name = "李四";
        s.age = 19 ;
        cout << s.age << " 的 " << s.name << " 在看书";
    }
};

int main(){

    student s;
    s.read();

    //类的外面无法访问，编译错误。
    s.name = "张三" ;
    s.age = 18 ;

    return 0 ;
}
```

4. 构造和析构

1. 继承状态

> 构造函数是对象在创建是调用，析构函数是对象在销毁时调用。但是在继承关系下，无论在对象的创建还是销毁，都会执行父类和子类的构造和析构函数。 它们一般会有以下规则：

> a. 子类对象在创建时会首先调用父类的构造函数;
>
> b. 父类构造函数执行完毕后,执行子类的构造函数;
>
> c. 当父类的构造函数中有参数时,必须在子类的初始化列表中显示调用;
>
> d. 析构函数执行的顺序是先调用子类的析构函数，再调用父类的析构函数

```
#include <iostream>

using namespace std;

class person{
public :
    person(){
        cout << "调用了父类构造函数" << endl;
    }
    ~person(){
        cout << "调用了父类析构函数" << endl;
    }
}

class student: public person{
public :
    student(){
        cout << "调用了子类类构造函数" << endl;
    }
    ~student(){
        cout << "调用了子类析构函数" << endl;
    }
};


int main() {
    Student s1
    return 0;
}
```

2. 继承和组合

> 如果在继承状态下，子类中的成员又含有其他类的对象属性，那么他们之间的构造很析构调用顺序，遵循以下原则：
>
> a. 先调用父类的构造函数,再调用组合对象的构造函数,最后调用自己的构造函数;
>
> b. 先调用自己的析构函数,再调用组合对象的析构函数,最后调用父类的析构函数。

```cpp
#include <iostream>

using namespace std;

//父类
class Person{

public :

    Person(){
        cout << "调用了父类构造函数" << endl;
    }

    ~Person(){
        cout << "调用了父类析构函数" << endl;
    }

};

//其他类
class  A{
public :
    A(){
       cout << "调用A的构造函数" << endl;
    }

    ~A(){
        cout << "调用A的析构函数" << endl;
    }
};

//子类
class Student: public Person{

public :
    Student(){
        cout << "调用了子类类构造函数" << endl;
    }
```

```
    ~Student(){
        cout << "调用了子类析构函数" << endl;
    }
public:
    A a;
};


int main() {
    Student s1(18 , "zhangsan");
    return 0;
}
```

3. 调用父类有参构造

> 继承关系下，子类的默认构造函数会隐式调用父类的默认构造函数，假设父类没有默认的无参构造函数，那么子类需要使用参数初始化列表方式手动调用父类有参构造函数。
>
> 一般来说在创建子类对象前，就必须完成父类对象的创建工作，也就是在执行子类构造函数之前，必须先执行父类的构造函数。c++ 使用初始化列表来完成这个工作

- 父类

```
#include <iostream>

using namespace std;

class Person{

private :
    int age ;
    string name ;

public :
    Person(int age , string name){
        cout << "调用了父类构造函数" << endl;
        this->age = age ;
        this->name = name;
    }
};
```

- 子类

> 子类只能使用初始化列表的方式来访问父类构造

```
class Student: public Person{

public :
    Student(int age , string name):Person(age ,name){
        cout << "调用了子类类构造函数" << endl;
    }
};

int main(){
    Student s1(18 , "zs");
    return 0 ;
}
```

4. 再说初始化列表

> 初始化列表在三种情况下必须使用:

> - 情况一、需要初始化的数据成员是对象，并且对应的类没有无参构造函数
> - 情况二、需要初始化const修饰的类成员或初始化引用成员数据；
> - 情况三、继承关系下，父类没有无参构造函数情况
>
> 初始化列表的赋值顺序是按照类中定义成员的顺序来决定

1. 常量和引用的情况

> 类中成员为 引用 或者 const修饰 的成员

```
#include <iostream>
#include <string>

using namespace std;

class stu{

public:
    const string name; //常量不允许修改值，所以不允许在构造里面使用  = 赋值
    int &age; //

    stu(string name , int age):name(name),age(age){
        cout << "执行构造函数" <<endl;
    }
};

int main(){

    stu s1("张三" , 88);
    cout << s1.name << " = " << s1.age << endl;

    return 0 ;
}
```

2. 初始化对象成员

> 类中含有其他类的对象成员，如果要初始化，只能使用初始化类列表方式。

```
#include <iostream>

using namespace std;

class A{

public:
    int number;
    A(int number):number(number){
        cout << "执行了A的构造函数" <<endl;
    }
};

class stu{
public:
    A a;

    stu():a(9){

        cout << "执行了stu的构造函数" <<endl;
    }
};

int main(){
    stu s;
    return 0;
}
```

5. 重写父类同名函数

> 在继承中，有时候父类的函数功能并不够强大，子类在继承之后，可以对其进行增强扩展。 如果还想调用你父类的函数，可以使用 `父类::函数名()` 访问

```cpp
#include <iostream>

using namespace std;

class WashMachine{
public:
    void wash(){
        cout << "洗衣机在洗衣服" << endl;
    }
};


class SmartWashMachine : public WashMachine{
public:
    void wash(){
        cout << "智能洗衣机在洗衣服" << endl;
        cout << "开始添加洗衣液~~" << endl;
        //调用父类的函数
        WashMachine::wash();
    }
};

int main(){

    SmartWashMachine s;
    s.wash();

    return 0 ;
}
```

6. 多重继承

> C++ 允许存在多继承，也就是一个子类可以同时拥有多个父类。只需要在继承时，使用逗号进行分割即可。

```
using namespace std;

class Father{
public:
    void makeMoeny(){
        cout << "赚钱" << endl;
    }
};
class Mother{
public:
    void makeHomeWork(){
        cout << "做家务活" << endl;
    }
};

class Son:public Father , public Mother{

};

int main(){

    Son s ;
    s.makeMoeny();
    s.makeHomeWork();

    return 0 ;
}
```

1. 多重继承的构造函数

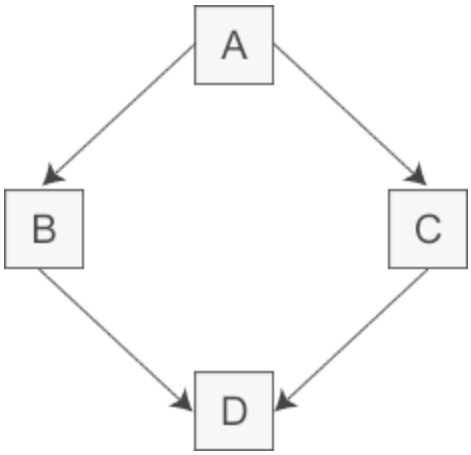
> 多继承形式下的构造函数和单继承形式基本相同，只是要在子类的构造函数中调用多个父类的构造函数 。 他们调用的顺序由定义子类时，继承的顺序决定。

```cpp
#include <iostream>

using namespace std;

class Father{

    string name;
public:
    Father(string name):name(name){
        cout << "执行父亲构造函数" <<endl;
    }
};



class Mother{
    int age;

public:
    Mother(int age):age(age){
        cout << "执行母亲构造函数" <<endl;
    }

};
class Son:public Father , public Mother{

public:
    Son(string name ,int age):Father(name),Mother(age){
        cout << "执行孩子构造函数" <<endl;
    }
};

int main(){

    Son s("无名氏" ,38);

    return 0 ;
}
```

菱形继承与虚继承

菱形继承

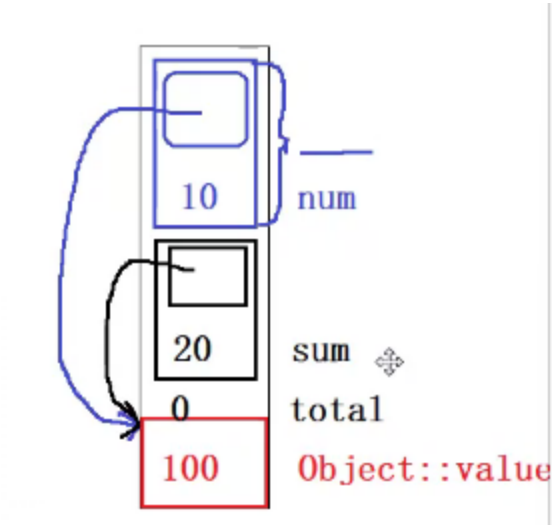
——该继承会导致造成公共基类在派生类对象中存在多个实例



使用虚继承来解决菱形继承问题

```
class Object
{
int value;
public:
Object(int x = 0) : value(x){}
};

class derive: virtual public Object//虚继承
{
int num;
public:
Base(int x = 0) : num(x), Object(x + 10){}
};

class Test :virtual public Object
{
int sum;
public:
Test(int x = 0): sum(x), Object(x + 10){}
};

class Det : public base,public Test
{
private:
int total;
public:
Det(int x=0):total(x),base(x+10),Test(x+20),Object(x+100){}
};

int main()
{
  det d(0);
  return 0;
}
12345678910111213141516171819202122232425262728293031323334
```

d1内存分配图如下



object首先被创建

被虚继承的类称为虚基类；

虚基类在派生类对象中存放于vbtable中；

虚基类在派生类中被构造时，在原本存储该基类对象的位置上创建一个指针，来指向虚基类实例的位置；

从而保证虚基类在派生类中只会有一个实例存在

虚基类在派生类构造时会被直接当作父类继承

7. 类的前置声明

> 一般来说，类和 变量是一样的，必须先声明然后再使用，如果在某个类里面定义类另一个类的对象变量，那么必须在前面做前置声明，才能编译通过。

```
class father; //所有前置声明的类，在某个类中定义的时候，只能定义成引用或者指针。

class son{
public:
    //father f0; //因为这行代码，单独拿出来说，会执行B类的无参构造，
    //但是编译器到此处的时候，还不知道B这个类的构造长什么样。
    father &f1;
    father *f2;

    son(father &f1 , father *f2):f1(f1),f2(f2){

    }
};


class father{

};


int main(){

    //  father b; //---> 执行B的构造函数。
    father f1;
    father f2;

    son s(f1 ,&f2);

    return 0 ;
}
```

二、多态

1. 什么是多态

> 多态按字面的意思就是多种形态。当类之间存在层次结构，并且类之间是通过继承关联

> 时，就会用到多态。
>
> C++ 多态意味着调用成员函数时，会根据调用函数的对象的类型来执行不同的函数。为了更为详细的说明多态，此处我们划分为**静态多态**和 **动态多态** 两种状态来讲解
>
> 平常我们说的多态：**动态多态**

1. 静态多态

> 静态多态是编译器在**编译期间**完成的，编译器会根据实参类型来选择调用合适的函数，如果有合适的函数可以调用就调，没有的话就会发出警告或者报错 。 该种方式的出现有两处地方： **函数重载** 和 **泛型编程 | 函数模板** .

```
int Add(int a, int b){
    return a + b;
}

double Add(double a, double b)
{
    return a + b;
}

int main()
{
    Add(10, 20);

    Add(10.0,20.0);

    return 0;
}
```

2. 动态多态

> 动态多态：指只有在运行的时候才能决定到底调用哪个类的函数，到底是父类的函数还是子类的函数
>
> 它是在**程序运行时**根据父类的引用（指针）指向的对象来确定自己具体该调用哪一个类的虚函数。

```
A *a = new A();

Father * f = new Fahter();

Father * f2 = new Child();
```

> 动态多态的必须满足两个条件：
>
> 1. 父类中必须包含虚函数，并且子类中一定要对父类中的虚函数进行**重写**。
> 2. 通过父类对象的指针或者引用调用虚函数。

2. 联编机制

1. 父类指针指向子类对象

> 通常情况下，如果要把一个引用或者指针绑定到一个对象身上，那么要求引用或者指针必须和对象的类型一致。 不过在继承关系下，父类的引用或指针可以绑定到子类的对象，这种现象具有欺骗性，因为在使用这个引用或者指针的时候，并不清楚它所绑定的具体类型，无法明确是父类的对象还是子类的对象。

```
#include <string>

using namespace std;

class person{

    string name;
    int age;
};

class student:public person{


};

int main(){

    //父类指针指向父类对象
    person *p = new person();

    //子类指针指向子类对象
    student *s = new student();

    //父类指针指向子类对象
    person *ps = new student();

    return 0 ;
}
```

2. 静态类型和动态类型

> 只有在继承关系下，才需要考虑静态和动态类型，这里仅仅是强调类型而已。所谓的 静态类型 指的是，在编译时就已经知道它的变量声明时对应的类型是什么。
> 而 动态类型 则是运行的时候，数据的类型才得以确定。
>
> 只有在 引用 或者 指针 场景下，才需要考虑 静态或者动态类型。因为非引用或者非指针状态下，实际上发生了一次拷贝动作。

- 静态类型

> 静态类型：不需要运行，编译状态下即可知晓具体的类型

```
#include<string>

using namespace std;

int main(){

    //编译状态下，即可知晓 a,b的类型。
    int a = 3;
    string b = "abc";

    return 0 ;
}
```

- 动态类型

> 只有真正运行代码了，才能知晓具体的类型

```
class Father{

};

class Child:public Father{

};

 //动态类型:
 //f在编译时，类型是Father ，但在运行时，真正的类型由getObj来决定。目前不能明确getObj返回的是Father的对象
Child getObj(){
    Child c ;
    return c;
}

int main(){

        Father &f = getObj();

    return 0 ;
}
```

3. 访问同名函数

> 父类的引用或指针可以绑定到子类的对象， 那么在访问同名函数时，常常出现意想不到的效果。

```cpp
#include<iostream>

using namespace std;

class father{
public:
    void show(){
        cout << "father show" << endl;
    }
};

class children : public father{
public:
    void show(){
        cout << "children  show" << endl;
    }
};

int main(){
    father f = children();
    f.show(); // 打印father show

    return 0 ;
}
```

4. 静态联编和动态联编

> 程序调用函数时，到底执行哪一个代码块,走哪一个函数呢？由编译器来负责回答这个问题。将源码中的函数调用解释为执行特定的函数代码，称之为 函数名联编 。

> 在C语言里面，每个函数名都对应一个不同的函数。但是由于C++里面存在重载的缘故，编译器必须查看函数参数以及函数名才能确定使用哪个函数，编译器可以在编译阶段完成这种联编，在编译阶段即可完成联编也被称为： 静态联编 | 早期联编 。 程序在运行期间才决定执行哪个函数，这种称之为 动态联编 | 晚期联编

```cpp
#include<iostream>

using namespace std;

class WashMachine{
public:
    void wash(){
        cout << "洗衣机在洗衣服" << endl;
    }
};


class SmartWashMachine : public WashMachine{
public:
    void wash(){
        cout << "智能洗衣机在洗衣服" << endl;

    }

};

int main(){
    WashMachine *w1= new WashMachine(); //父类指针指向父类对象  打印：洗衣机在洗衣服
    w1->wash();

    SmartWashMachine *s  = new SmartWashMachine();  //子类指针指向子类对象 打印：  智能洗衣机...
    s->wash();

    WashMachine *w2 = new SmartWashMachine(); //父类指针指向子类对象 打印..洗衣机在洗衣服
    w2->wash();

    return 0 ;
}
```

5. 为什么要区分两种联编

> 动态联编在处理子类重新定义父类函数的场景下，确实比静态联编好，静态联编只会无脑的执行父类函数。但是不能因此就否定静态联编的作用。动态联编状态下，为了能够让指针顺利访问到子类函数，需要对指针进行跟踪、这需要额外的开销。但是并不是所有的函

> 数都处于继承状态下，那么此时静态联编更优秀些。

> 编写c++代码时，不能保证全部是继承体系的父类和子类，也不能保证没有继承关系的存在，所以为了囊括两种情况，c++ 才提供了两种方式。

> 正所谓两害相权取其轻，考虑到大部分的函数都不是处在继承结构中，所以效率更高的静态联编也就成了默认的的选择。

练习

> 一家海洋馆开业了，门口挂着牌子说：欢迎鲨鱼进来游泳。 过了几天，再贴出告示，欢迎鳄鱼也进来游泳，又过了几天，说欢迎罗非鱼也进来游泳，最后干脆直接说了，只要是鱼类的都可以进来游泳。请使用面向对象的思想，设计海洋馆馆接收鱼类的过程。 多态。
>
> //全局函数
>
> void swiming(鲨鱼， 鳄鱼， 罗非鱼， 金鱼， 鳗鱼，.... ....){
>
> }

3. 虚函数

1. 虚函数入门

> C++中的虚函数的作用主要是实现了多态的机制， 有了虚函数就可以在父类的指针或者引用指向子类的实例的前提下，然后通过父类的指针或者引用调用实际子类的成员函数。这种技术让父类的指针或引用具备了多种形态。定义虚函数非常简单，只需要在函数声明前，加上 `virtual` 关键字即可。 在父类的函数上添加 virtual 关键字，可使子类的同名函数也变成虚函数。
>
> 如果基类指针指向的是一个基类对象，则基类的虚函数被调用， 如果基类指针指向的是一个派生类对象，则派生类的虚函数被调用。

```
#include <iostream>

using namespace std;

class WashMachine{
public:
    virtual void wash(){
        cout << "洗衣机在洗衣服" << endl;
    }
};


class SmartWashMachine : public WashMachine{
public:
   virtual  void wash(){
        cout << "智能洗衣机在洗衣服" << endl;

    }
};


int main(){

    WashMachine *w2 = new SmartWashMachine(); //父类指针指向子类对象 打印..洗衣机在洗衣服
    w2->wash();

    return 0 ;
}
```

2. 虚函数的工作原理

> 了解虚函数的工作原理，有助于理解虚函数。
>
> 通常情况下，编译器处理虚函数的方法是：给每一个对象添加一个隐藏指针成员，它指向一个数组，数组里面存放着对象中所有函数的地址。这个数组称之为虚函数表（virtual function table v-table）。表中存储着类对象的虚函数地址。

vftable什么时候产生？于何处存储？
编译期；rodata段（只读数据段）

class中有虚函数就会创建虚表；
虚表存储各虚函数的函数指针；

编译时，编译器若发现派生类对象有基类的同名函数，则会发生同名覆盖，虚表中的函数指针被替换；

```
class Base
{
private:
    int value;
public:
Base(int x = 0) : value(x){}
virtual void add() {}
virtual void fun() {}
virtual void print() const {}
}

class derive  : public Base
{
private: int sum;
public:
derive  (int x = 0) : base(x+10) , sum(x){}
virtual void add() {}
virtual void fun() {}
virtual void print() const {};

int main()
{
  derive  derive  (10);//大小为12个字节，8（sum，value）+ 4 （虚表指针vfptr）
}
12345678910111213141516171819202122232 4
```

为了调用虚表中的函数指针，每个有虚函数的类都会额外开辟4个字节来存储一个虚指针_vfptr，用其来索引向自己类的虚表；

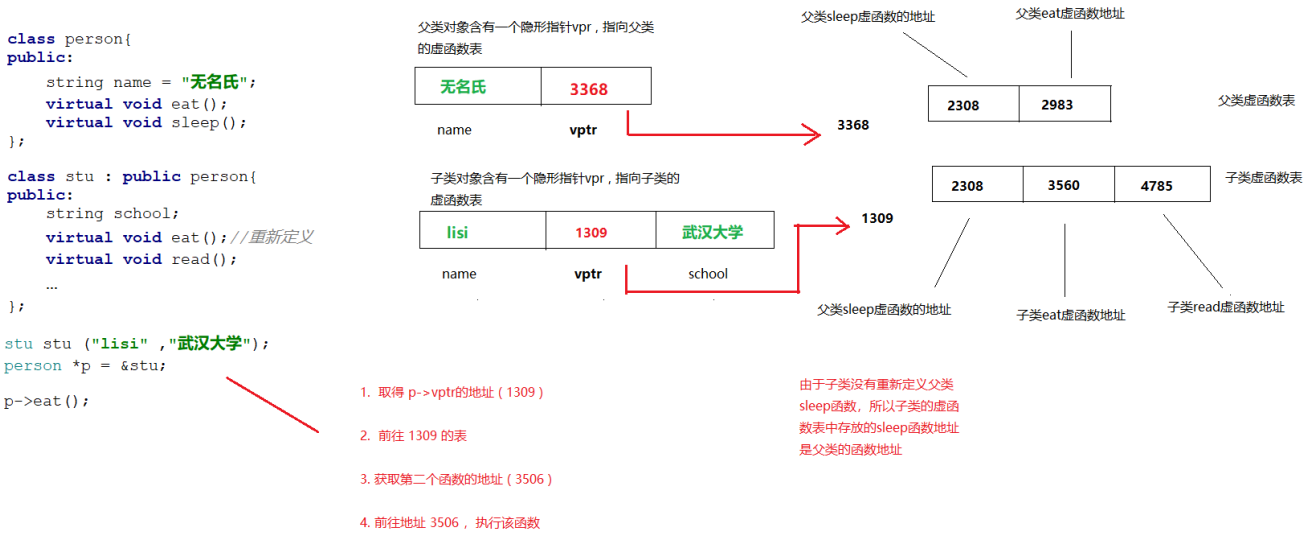
虚表指针_vfptr在构造时候写入对象的存储空间，其用来指向该类的vftable

一个类的虚表只有一份
虚表指针存在于派生类对象的隐藏基类中

如在上例中，创建derive对象时，首先调用base的构造函数创建隐藏base类；因其有虚函数，编译器同时创建虚表指针_vfptr，使其指向base的虚表；之后构建derive，对所有vftable中的同名函数进行`同名覆盖`，同时，该派生类中的虚表指针改为指向derive的虚表；

即-父类中的虚函数会被子类中相同的函数覆盖；该过程发生于子类在构建时的虚函数表中

> 父类对象包含的指针，指向父类的虚函数表地址，子类对象包含的指针，指向子类的虚函数表地址。
>
> 如果子类重新定义了父类的函数，那么函数表中存放的是新的地址，如果子类没有重新定义，那么表中存放的是父类的函数地址。
>
> 若子类有自己虚函数，则只需要添加到表中即可。



3. 构造函数可以是虚函数吗

> 构造函数不能为虚函数，因为虚函数的调用，需要虚函数表(指针)，而该指针存在于对象开辟的空间中，而对象的空间开辟依赖构造函数的执行，这就是鸡和蛋的矛盾问题了。

```
#include <iostream>

using namespace std;

class father{
public:
    virtual father(){ //报错!
        cout <<"父亲构造函数~! ~" << endl;
    }
};

int  main(){

    father f ;
    return 0 ;
}
```

4. 析构函数可以是虚函数吗

> 在继承体系下， 如果父类的指针可以指向子类对象，这就导致在使用 `delete` 释放内存时，却是通过父类指针来释放，这会导致父类的析构函数会被执行，而子类的析构函数并不会执行，此举有可能导致程序结果并不是我们想要的。究其原因，是因为静态联编的缘故，在编译时，就知道要执行谁的析构函数。

> 为了解决这个问题，需要把父类的析构函数变成虚拟析构函数，也就是加上 `virtual` 的定义。一旦父类的析构函数是虚函数，那么子类的析构函数也将自动变成虚函数。
>
> 一句话概括： 继承关系下，所有人的构造都不能是虚函数，并且所有人的析构函数都必须是虚函数。
>
> 只要在父亲的析构函数加上 virtual ， 那么所有的析构函数都变成 虚函数

```cpp
#include <iostream>

using namespace std;

class father{

 public:
    virtual ~father(){
        cout << "执行父类析构函数" << endl;
    }
};


class son : public father{
    ~son(){
         cout << "执行子类析构函数" << endl;
    }
};

int main(){

    father *f = new son(); //父类指针指向子类对象

    //创建的是子类对象，理应执行子类的析构函数
    delete f;


    return 0 ;
}
```

练习

> 动物都有觅食的行为，但是每种动物吃的食物都不太一样。请使用面向对象的思想，配合虚函数来设计动物觅食的行为。

4. override 关键字

> 在继承关系下，子类可以重写父类的函数，但是有时候担心程序员在编写时，有可能因为粗心写错代码。所以在C++ 11中，推出了 `override` 关键字，用于表示子类的函数就是重写了父类的同名函数 。 不过值得注意的是， `override` 标记的函数，必须是虚函数。
>
> `override` 并不会影响程序的执行结果，仅仅是作用于编译阶段，用于检查子类是否真的重写父类函数

```
#include <iostream>

using namespace std;

class father{
public:
    virtual void run(){
        cout << "父亲在跑步" << endl;
    }
};


class son : public father{
public:
   virtual  void run() override{ //表示重写父类的函数
        cout << "孩子在跑步" << endl;
    }
};
```

5. final 关键字

> 在c++11 推出了final关键字，其作用有两个： (1)、禁止虚函数被重写；(2)、禁止类被继承。
>
> 注意： 只有虚函数才能被标记为final， 其他的普通函数无法标记final。

- 标记在类上

```
class person final{ //表示该类是最终类，无法被继承

};


//编译错误，无法被继承。
class student : public person{

};
```

- 标记在函数上

```
class person {
    virtual void run() final{ //表示该方法时最终方法，无法被重写

    }
};
class student : public person{

    //编译错误，方法无法被重写。
    void run(){

    }
};
```

6. =delete 和 =default

> 这两个关键字平常使用的不多，一般出现在类的特殊成员函数上。**=delete** 用于表示该函数禁止被调用，**=default** 以及使用编译器默认提供的函数功能。

1. =delete

```
#include <iostream>
#include <string>

using namespace std;

class stu{

    string name;
    int age;

public:
    stu(string name , int age):name(name) , age(age){
        cout << "执行stu的构造函数" << endl;
    }


    //表示禁止调用拷贝构造函数
    stu(stu & s) =delete;


    ~stu(){
        cout << "执行stu的析构函数" << endl;
    }
};

int main(){

    stu s("张三",18) ;

    //编译错误。
    stu s1= s;

    return 0 ;
}
```

2. =default

> 一旦定义了有参构造函数之后，编译器将不会替我们生成无参构造函数 。 此时可以自己编写无参构造函数（哪怕函数体是空的），但是此举增加了程序员的编程工作量。更值得一提的是，手动定义的无参构造函数的代码执行效率要低于编译器自动生成的无参构造函数。

```
#include <iostream>
#include <string>

using namespace std;

class stu{

    string name;
    int age;

public:

    stu() = default;

    stu(string name , int age):name(name) , age(age){
        cout << "执行stu的构造函数" << endl;
    }

    ~stu(){
        cout << "执行stu的析构函数" << endl;
    }
};

int main(){

    stu s("张三",18) ;

    //编译错误。
    stu s1 = s;

    return 0 ;
}
```

7. 纯虚函数

> 纯虚函数是一种特殊的虚函数，C++中包含纯虚函数的类，被称为是“抽象类”。抽象类不能创建对象，只有实现了这个纯虚函数的子类才能创建对象。C++中的纯虚函数更像是“只提供声明，没有实现”，是对子类的约束。

纯虚函数就是没有函数体，同时在定义的时候，其函数名后面要加上“= 0”。

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


class WashMachine{
public:
    //没有函数体，表示洗衣机能洗衣服，但是具体怎么洗，每个品牌不一样
    virtual void wash() = 0;
};

class HaierMachine:public WashMachine{
public :
    virtual void wash(){
        cout << "海尔牌洗衣机在洗衣服" << endl;
    }
};

class LittleSwanMachine:public WashMachine{
public :
    virtual void wash(){
        cout << "小天鹅洗衣机在洗衣服" << endl;
    }
};

int main(){

    //WashMachine w;  错误，抽象类无法创建对象
    WashMachine *w1 = new HaierMachine() ;
    WashMachine *w2 = new LittleSwanMachine() ;

    return 0 ;
}
```

抽象类的一些特征

1. 如果有一个类当中有纯虚函数，那么这个类就是抽象类
2. 抽象类是无法创建对象的，因为一旦能够创建对象，里面的纯虚函数没有函数体，也就不知道要执行什么逻辑了，所以禁止抽象类创建对象。
3. 抽象类当中也可以有普通的成员函数，虽然父类不能创建对象，但是子类可以创

建，所以这些函数可以由子类访问。

4. 如果一个子类继承了一个父类（父类是抽象类），那么子类就必须重写所有的纯虚函数，否则视子类为抽象类，因为继承体系下，等同于子类拥有了和父类一样的代码。

8. 抽象类和接口

> 所谓接口，其实就是用于描述行为和功能，并不会给出具体的实现。C++中没有提供类似 `interface` 这样的关键字来定义接口，纯虚函数往往承担起了这部分功能，可以看成是对子类的一种约束。
>
> 抽象类可以用来定义一种事物的行为特征，洗衣机：洗衣服。

```
class Person{
            Person() =default; // 可以用于初始化成员函数
      virtual ~Person()=default; //防止子类析构函数无法被调用问题


      //每个人吃什么，做什么都不一样，，即可声明为纯虚函数
      virtual void eat() = 0 ;
      virtual void work() = 0 ;
      //...

};
```

练习

> 黑马马戏团最近来了3种动物，猴子、熊猫、老虎，
>
> 每种动物都具有昵称，年纪，大家除了具备吃饭的技能之外，
>
> 每种动物都具有自己特有的技能:
>
> 猴子荡秋千，
>
> 老虎穿越火环，

熊猫踩独轮单车。

请使用面向对象的思想来设计该题。

1.要体现出来动物园

2.要体现出来动物---猴子|熊猫 |老虎之间的关系

3.要体现动物园和动物之间的关系。

4.动物园应该是一个类，需要用容器来存储动物。

5.要使用:继承、多态、纯虚函数、虚函数。

int main() {

1.创建动物园

2.创建猴子、熊猫、老虎，要收纳到动物园去

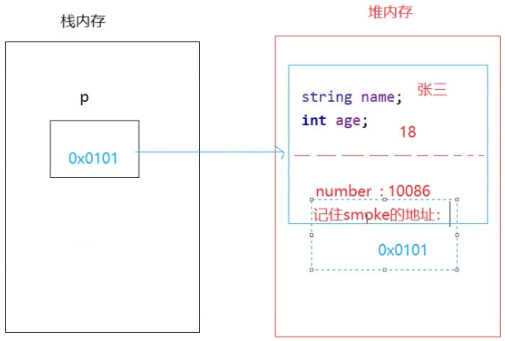
3.展示动物园都有什么动物，动物的信息具备的技能打印出来。

}

对象细节

1.创建子类对象的时候，会同时创建父类对象吗？不会

```
class person{                          C++的规则就是变量|成员在哪个类里面，就
public:                                由哪个类来初始化它。
    string name;
    int age;
    person(string name , int age ) : name(name),age(age){
        cout << "person有参 " << this <<endl;
    }                                  smoke();
};
class stu :public person{
public:
    string number;
    stu(string name , int age , string number ) : person(name , age ),number(number){
        cout << "stu有参 "  << this<<endl;
    }
};

int main() {
    person * p = new stu( name: "张三" , age: 18 , number: "10086");
    return 0;
}
```



2.创建对象的隐式转换

1.当我们的类里面存在有参构造函数，并且这个有参构造函数的参数只有一个的时候，默认情况下，编译器允许这种隐式转换的情况存在

2.但是这种情况容易让人产生误解。左边的类型和右边的类型不对等

3.程序员也可以手动禁止这种隐式转换，在只有一个参数的有参构造函数的签名加上 explicit