

1继承

一，继承的基本概念

1.1类与类之间的关系

- has-A, 包含关系，用以描述一个类由多个“部件类”构成，实现has-A关系用类的成员属性表示，即一个类的成员属性是另一个已经定义好的类。
- use-A, 一个类使用另一个类，通过类之间的成员函数相互联系，定义友元或者通过传递参数的方式来实现。
- is-A, 即继承关系，关系具有传递性。

1.2继承的相关概念

万事万物皆有继承这个现象，所谓的继承就是一个类继承了另一个类的属性和方法，这个新的类包含了上一个类的属性和方法，被称为子类或者派生类，被继承的类称为父类或者基类。

1.3.继承的特点

- 子类拥有父类的所有属性和方法（除了构造函数和析构函数）。
- 子类可以拥有父类没有的属性和方法。
- 子类是一种特殊的父类，可以用子类来代替父类。
- 子类对象可以当做父类对象使用。

1.4.继承的语法

类继承关系的语法形式

```
class 派生类名 : 基类名表
{
    数据成员和成员函数声明
};
```

基类名表 构成

访问控制 基类名₁ , **访问控制** 基类名₂ , ... , **访问控制** 基类名_n

访问控制表示派生类对基类的继承方式，使用关键字：

public	公有继承
private	私有继承
protected	保护继承

二，继承中的访问控制

1.继承的访问控制方式

我们在一个类中可以对成员变量和成员函数进行访问控制，通过C++提供的三种权限修饰符实现。在子类继承父类时，C++提供的三种权限修饰符也可以在继承的时候使用，分别为公有继承，保护继承和私有继承。这三种不同的继承方式会改变子类对父类属性和方法的访问。

2.三种继承方式对子类访问的影响

- public继承：父类成员在子类中保持原有的访问级别（子类可以访问public和protected）。
- private继承：父类成员在子类中变为private成员（虽然此时父类的成员在子类中体现为private修饰，但是父类的public和protected是允许访问的，因为是继承后改为private）。
- protected继承
 - 父类中的public成员会变为protected级别。
 - 父类中的protected成员依然为protected级别。
 - 父类中的private成员依然为private级别。
- 注意：父类中的private成员依然存在于子类中，但是却无法访问到。不论何种方式继承父类，子类都无法直接使用父类中的private成员。

3.父类如何设置子类的访问

- 需要被外界访问的成员设置为public。
- 只能在当前类中访问设置为private。
- 只能在当前类和子类中访问，设置为protected。

三，继承中的构造和析构函数

1.父类的构造和析构

当创建一个对象和销毁一个对象时，对象的构造函数和析构函数会相应的被C++编译器调用。当在继承中，父类的构造和析构函数是如何在子类中进行调用的呢，C++规定我们在子类对象构造时，需要调用父类的构造函数完成对继承而来的成员进行初始化，同理，在析构子类对象时，需要调用父类的析构函数对其继承而来的成员进行析构。

2.父类中的构造和析构执行顺序

- 子类对象在创建时，会先调用父类的构造函数，如果父类还存在父类，则先调用父类的父类的构造函数，依次往上推理即可。
- 父类构造函数执行结束后，执行子类的构造函数。
- 当父类的构造函数不是C++默认提供的，则需要在子类的每一个构造函数上使用初始化列表的方式调用父类的构造函数。
- 析构函数的调用顺序和构造函数的顺序相反。

3.继承中的构造函数示例



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

class Parent
{
protected:
    char * str;
public:
    Parent(char * str)
    {
        if (str != NULL)
        {
            this->str = new char[strlen(str) + 1];
            strcpy(this->str, str);
        }
        else {
            this->str = NULL;
        }
        cout << "父类构造函数..." << endl;
    }
}
```

```

    }
    ~Parent()
    {
        if (this->str != NULL)
        {
            delete[] this->str;
            this->str = NULL;
        }
        cout << "父类析构函数..." << endl;
    }
};

class Child:public Parent
{
private:
    char * name;
    int age;
public:
    /* 在构造子类对象时，调用父类的构造函数 */
    Child():Parent(NULL)
    {
        this->name = NULL;
        this->age = 0;
        cout << "子类无参构造函数..." << endl;
    }
    /* 在构造子类对象时，调用父类的构造函数 */
    Child(char * name,int age):Parent(name)
    {
        this->name = new char[strlen(name) + 1];
        strcpy(this->name, name);
        cout << "子类有参构造函数..." << endl;
    }
    ~Child()
    {
        if (this->name != NULL)
        {
            delete[] this->name;
            this->name = NULL;
        }
        cout << "子类析构函数..." << endl;
    }
};

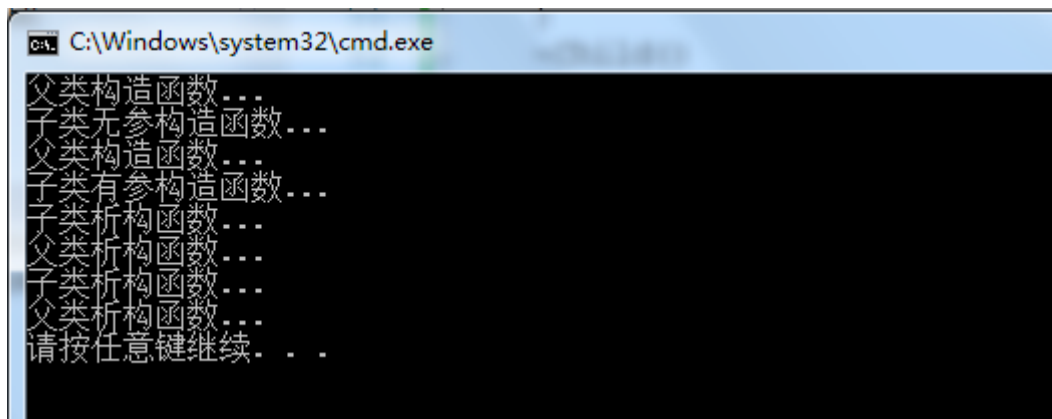
int main()
{
    Child c1;
    Child c2("王刚",22);

    return 0;
}

```



输出结果：



```
C:\Windows\system32\cmd.exe
父类构造函数...
子类无参构造函数...
父类构造函数...
子类有参构造函数...
子类析构函数...
父类析构函数...
子类析构函数...
父类析构函数...
请按任意键继续. . .
```

4.继承与组合情况混搭下的构造析构函数调用顺序

- 构造函数：先调用父类的构造函数，再调用成员变量的构造函数，最后调用自己的构造函数。
- 析构函数：先调用自己的析构函数，再调用成员变量的析构函数，最后调用父类的析构函数。

5.继承和组合情况混搭情况下的代码演示



```
# include<iostream>
using namespace std;
/* 定义父类 */
class Parent
{
protected:
    char * str;
public:
    Parent(char * str)
    {
        if (str != NULL)
        {
            this->str = new char[strlen(str) + 1];
            strcpy(this->str, str);
        }
        else {
            this->str = NULL;
        }
        cout << "父类构造函数..." << endl;
    }
    ~Parent()
    {
        if (this->str != NULL)
        {
            delete[] this->str;
            this->str = NULL;
        }
        cout << "父类析构函数..." << endl;
    }
};
/* 定义Object类 */
class Object
{
private:
    int i;
public:
    Object(int i)
```

```

    {
        this->i = i;
        cout << "Object的构造函数..." << endl;
    }
    ~Object()
    {
        cout << "Object的析构函数..." << endl;
    }
};
/* 定义子类 */
class Child:public Parent
{
private:
    /* 定义类成员变量 */
    Object obj;
    char * name;
    int age;
public:
    /* 在构造子类对象时，调用父类的构造函数和调用成员变量的构造函数 */
    Child():Parent(NULL),obj(10)
    {
        this->name = NULL;
        this->age = 0;
        cout << "子类无参构造函数..." << endl;
    }
    /* 在构造子类对象时，调用父类的构造函数和调用成员变量的构造函数 */
    Child(char * name,int age):Parent(name),obj(age)
    {
        this->name = new char[strlen(name) + 1];
        strcpy(this->name, name);
        cout << "子类有参构造函数..." << endl;
    }
    ~Child()
    {
        if (this->name != NULL)
        {
            delete[] this->name;
            this->name = NULL;
        }
        cout << "子类析构函数..." << endl;
    }
};

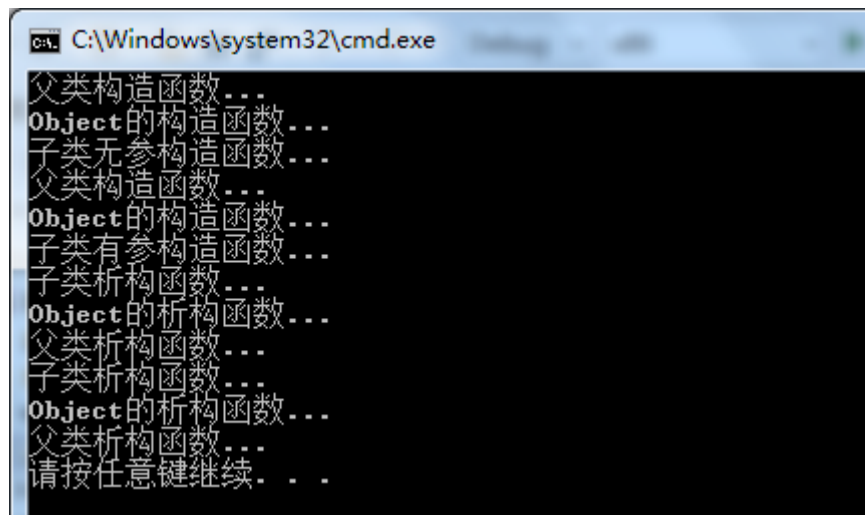
int main()
{
    Child c1;
    Child c2("王刚",22);

    return 0;
}

```



输出结果：



四，继承中的类型兼容性原则

1.继承中的同名成员

当在继承中，如果父类的成员和子类的成员属性名称相同，我们可以通过作用域操作符来显式的使用父类的成员，如果我们不使用作用域操作符，默认使用的是子类的成员属性。

2.继承中的同名成员演示



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

class PP
{
public:
    int i;
};
class CC:public PP
{
public:
    int i;
public:
    void test()
    {
        /* 使用父类的同名成员 */
        PP::i = 10;
        /* 使用子类的同名成员 */
        i = 100;
    }
    void print()
    {
        cout << "父类:" << PP::i << "," << "子类:" << i << endl;
    }
};

int main()
{
    CC cc;
    cc.test();
    cc.print();
    return 0;
}
```



3.类型兼容性原则

类型兼容性原则是指在需要父类对象的所有地方，都可以用公有继承的子类对象来替代。通过公有继承，子类获得了父类除构造和析构之外的所有属性和方法，这样子类就具有了父类的所有功能，凡是父类可以解决的问题，子类也一定可以解决。

4.类型兼容性原则可以替代的情况

- 子类对象可以当做父类对象来使用。
- 子类对象可以直接赋值给父类对象。
- 子类对象可以直接初始化父类对象。
- 父类指针可以直接指向子类对象。
- 父类引用可以直接引用子类对象。

5.类型兼容性原则示例



```
# include<iostream>
using namespace std;
/* 创建父类 */
class MyParent
{
protected:
    char * name;
public:
    MyParent()
    {
        name = "Helloworld";
    }
    void print()
    {
        cout << "name = " << name << endl;
    }
};
/* 创建子类 */
class MyChild:public MyParent
{
protected:
    int i;
public:
    MyChild()
    {
        i = 100;
        name = "I am Child";
    }
};

void main()
{
    /* 定义子类对象 */
    MyChild c;
    /* 用子类对象当做父类对象使用 */
    c.print();
}
```

```

    /* 用子类对象初始化父类对象 */
    MyParent p1 = c;
    p1.print();
    /* 父类指针直接指向子类对象 */
    MyParent * p2 = &c;
    p2->print();
    /* 父类对象直接引用子类对象 */
    MyParent& p3 = c;
    p3.print();
}

```



6. 继承中的static

- 继承中的static也遵循三种继承的基本访问控制原则。
- 继承中的static可以通过类名和域作用符的方式访问，也可以通过对象点的方式访问。

7. 继承中的static演示



```

#include<iostream>

using namespace std;
/* 父类 */
class MyP
{
public:
    static int i;
};
/* 初始化静态成员 */
int MyP::i = 10;
/* 子类 */
class MyC:public MyP
{
public:
    void test()
    {
        /* 直接访问 */
        cout << "i = " << i << endl;
        /* 通过父类访问 */
        cout << "Myp::i = " << MyP::i << endl;
        /* 通过子类访问 */
        cout << "MyC::i = " << MyC::i << endl;
    }
    void add()
    {
        i++;
    }
};
int main()
{
    MyC c;
    c.add();
    c.test();

    MyC c1;
}

```



```

        c1.add();
        c1.test();
        /* 通过子类对象访问 */
        c1.i = 100;
        c1.test();

        return 0;
    }

```



五，多继承

1.C++中的多继承

所谓的多继承就是指一个子类可以继承多个父类，子类可以获取多个父类的属性和方法。这种继承方式是不被推荐的，但是C++还是添加了，事实证明，多继承增加了代码的复杂度，而且任何可以通过多继承解决的问题都可以通过单继承的方式解决。多继承和单继承的基本知识是相同的。不需要再阐述，主要讲解下面的不同的地方。

2，多继承中的构造和析构

和单继承类似，还是首先执行父类的构造函数，此时有多个构造函数，则按照继承时的父类顺序来执行相应父类的构造函数，析构函数与此相反。

3.多继承中的二义性

一个类A，它有两个子类B1和B2，然后类C多继承自B1和B2，此时如果我们使用类A里面的属性，则根据上面的多继承的构造和析构，发现此时的类A会被创造两个对象，一个是B1一个是B2，此时使用A里面的属性则会出现编译器无法知道是使用B1的还是B2的。因此C++为我们提供了虚继承这个概念，即B1和B2虚继承自A，则在构造A对象的时候，只创建一个A的对象。

4.多继承的二义性代码示例

此时如果去除virtual关键字，尝试一下会报错。



```

# include<iostream>
using namespace std;
/* 类B */
class B
{
public:
    int a;
};
/* 虚继承自类B */
class B1 :virtual public B
{
};
/* 虚继承自类B */
class B2 :virtual public B
{
};
/* 继承自B1,B2 */
class C :public B1, public B2
{
}

```

```
};

void main()
{
    C c;
    c.B::a = 100;

    cout << c.B::a << endl;
    cout << c.B1::a << endl;
    cout << c.B2::a << endl;
}
```



C++的多态性用一句话概括就是：在基类的函数前加上virtual关键字，在派生类中重写该函数，运行时将会根据对象的实际类型来调用相应的函数。如果对象类型是派生类，就调用派生类的函数；如果对象类型是基类，就调用基类的函数

- 1: 用virtual关键字申明的函数叫做虚函数，虚函数肯定是类的成员函数。
- 2: 存在虚函数的类都有一个一维的虚函数表叫做虚表，类的对象有一个指向虚表开始的虚指针。虚表是和类对应的，虚表指针是和对象对应的。
- 3: 多态性是一个接口多种实现，是面向对象的核心，分为类的多态性和函数的多态性。
- 4: 多态用虚函数来实现，结合动态绑定。
- 5: 纯虚函数是虚函数再加上 = 0;
- 6: 抽象类是指包括至少一个纯虚函数的类。

纯虚函数: `virtual void fun()=0;`即抽象类！必须在子类实现这个函数，即先有名称，没有内容，在派生类实现内容。

下面看下c++语言虚函数实现多态的原理

自上一个帖子之间跳过了一篇总结性的帖子，之后再发，今天主要研究了c++语言当中虚函数对多态的实现，感叹于c++设计者的精妙绝伦

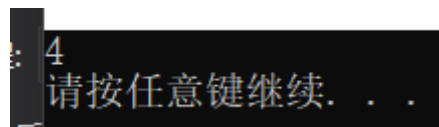
c++中虚函数表的作用主要是实现了多态的机制。首先先解释一下多态的概念，多态是c++的特点之一，关于多态，简而言之就是用父类的指针指向其子类的实例，然后通过父类的指针调用实际子类的成员函数，这种方法呢，可以让父类的指针具有多种形态，也就是说不需要改动很多的代码就可以让父类这一种指针，干一些很多子类指针的事情，这里是从虚函数的实现机制层面进行研究

在写这篇帖子之前对于相关的文章进行了查阅，基本上是大段的文字，所以我的这一篇可能会用大量的图形进行赘述（如果理解有误的地方，烦请大佬能够指出），接下来就言归正传：

首先介绍一下为什么会引进多态呢，基于c++的复用性和拓展性而言，同类的程序模块进行大量重复，是一件无法容忍的事情，比如我设置了苹果，香蕉，西瓜类，现在想把这些东西都装到碗这个函数里，那么在主函数当中，声明对象是必须的，但是每一次装进碗里对于水果来说，都要用自己的指针调用一次装的功能，那为什么不把这些类抽象成一个水果类呢，直接定义一个水果类的指针一次性调用所有水果装的功能呢，这个就是利用父类指针去调用子类成员，但是这个思想受到了指针指向类型的限制，也就是说表面指针指向了子类成员，但实际上还是只能调用子类成员里的父类成员，这样的思想就变的毫无意义了，如果想要解决这个问题，只要在父类前加上virtual就可以解决了，这里就是利用虚函数实现多态的实例。

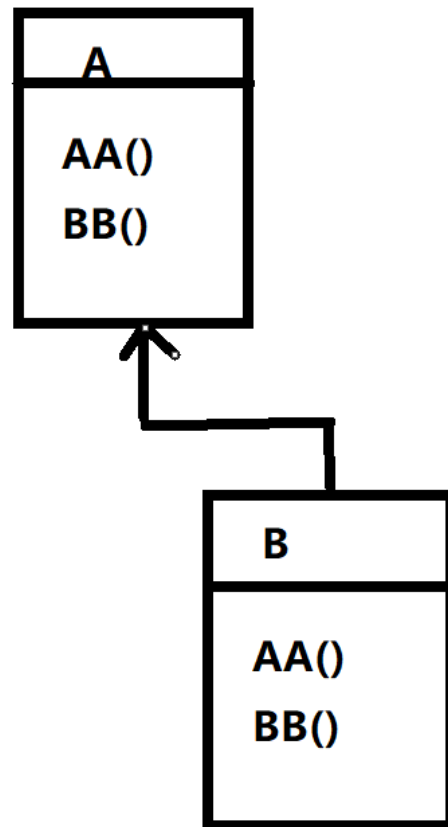
首先还是作为举例来两个类，在之前基础知识的帖子中提到过，空类的大小是一个字节（占位符），函数，静态变量都在编译期就形成了，不用类去分配空间，但是做一个小实验，看一看在定义了虚函数之后，类的大小是多少呢

```
#include<iostream>
using namespace std;
class CFather
{
public:
    virtual void AA()    //虚函数标识符
    {
        cout << "CFather :: AA()" << endl;
    }
    void BB()
    {
        cout << "CFather :: BB()" << endl;
    }
};
class CSon : public CFather
{
public:
    void AA()
    {
        cout << "CSon :: AA()" << endl;
    }
    void BB()
    {
        cout << "CSon :: BB()" << endl;
    }
};
int main()
{
    cout << sizeof(CFather) << endl;           //测试加了虚函数的类
    system("pause");
    return 0;
}
```

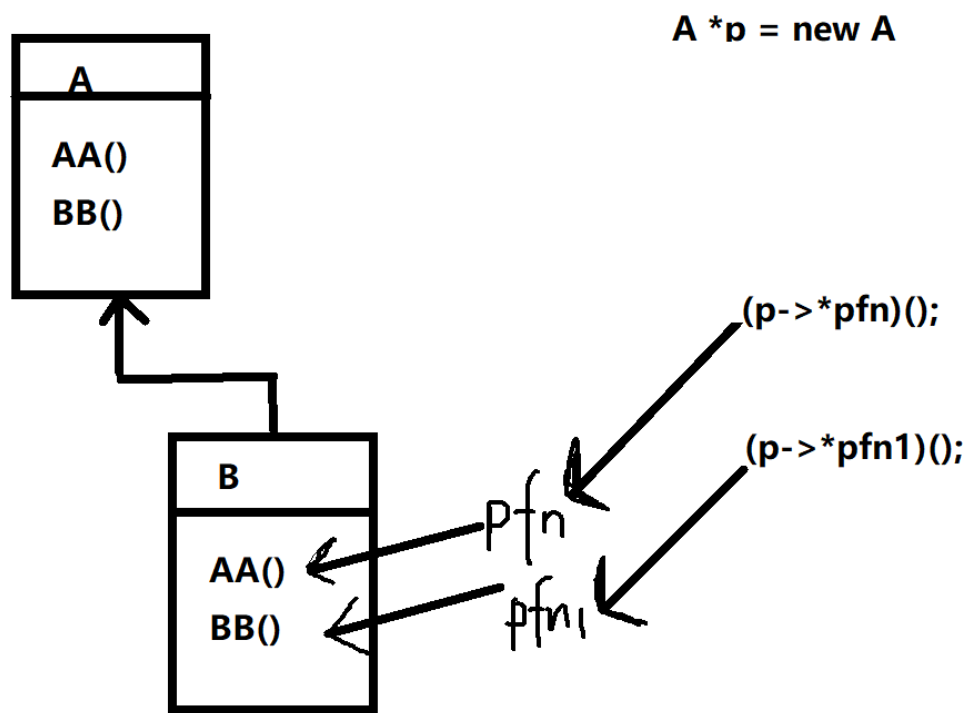


很明显类里装了一个 4 个字节的东西，除了整形int，就是指针了，没错这里装的就是函数指针

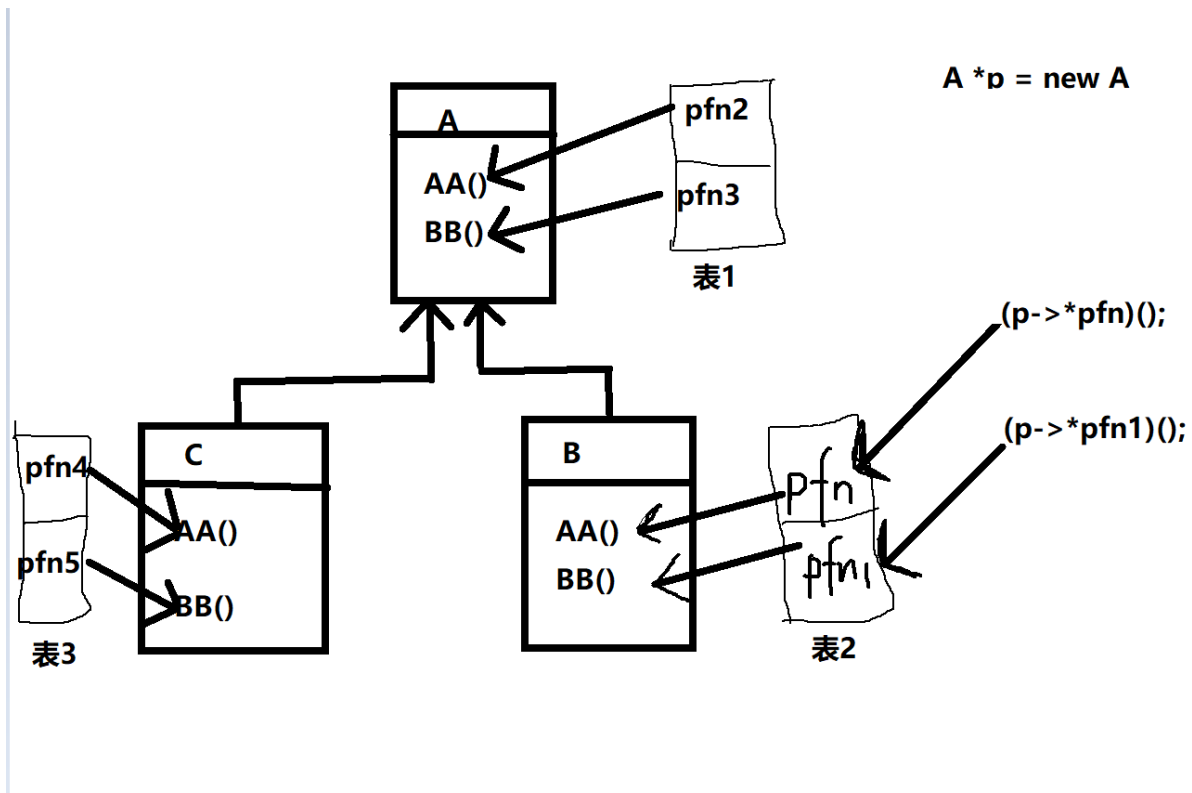
先把这个代码，给抽象成图形进行理解，在这CFather为A，CSon为B



此时就是一个单纯的继承的情况，不存在虚函数，然后我new一个对象，`A *p = new A`；那么 `p -> AA()`，必然是指向A类中的AA()函数，那么函数的调用有两种方式 一种函数名加 () 直接调用，一种是利用函数指针进行调用，在这里我想要调用子类的，就可以利用函数指针进行调用，假设出来两个函数指针，来指向B类中的两个成员函数，如果我父类想要调用子类成员，就可以通过 p指针去调用函数指针，再通过函数指针去调用成员函数

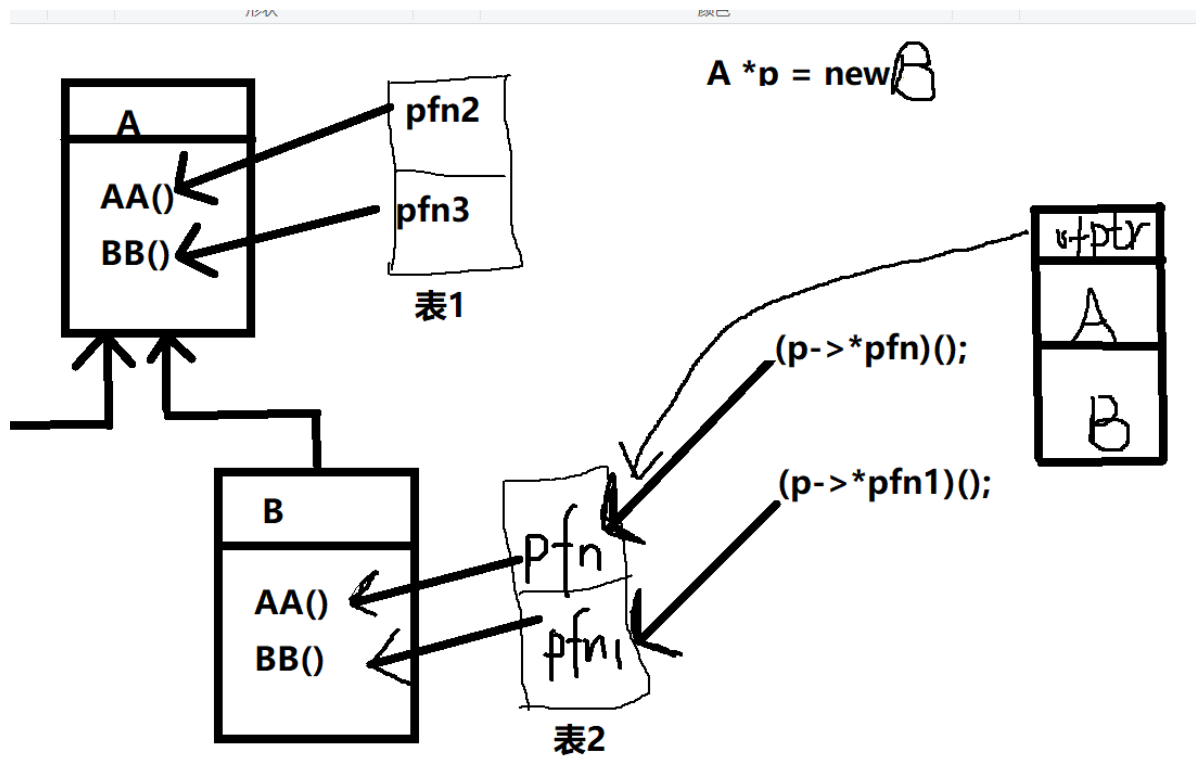


每一个函数都可以用一个函数指针去指着，那么每一类中的函数指针都可以形成自己的一个表，这个就叫做虚函数表



那么在创建对象后，为什么类中会有四个字节的内存空间呢？

在C++的标准规格说明书中说到，编译器必需要保证虚函数表的指针存在于对象中最前面的位置（这是为了保证正确取到虚函数的偏移量）。这意味着我们通过对象实例的地址得到这张虚函数表，然后就可以遍历其中函数指针，并调用相应的函数。也就是说这四个字节的指针，代替了上图中 $(p \rightarrow pfn)()$ 的作用，指向了函数指针，也就是说，在使用了虚函数的父类成员函数，虽然写的还是 $p \rightarrow AA()$ ，实际上却是 $(p \rightarrow (vfptr[0]))$ ，而指向哪个虚函数表就由，创建的对象来决定

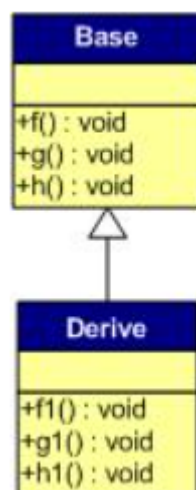


至此，就能理解如何用虚函数这个机制来实现多态的了

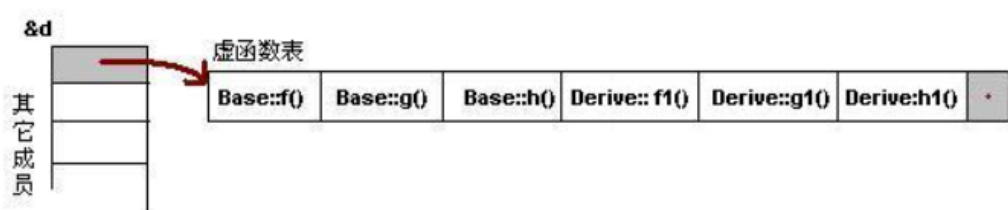
下面，我将分别说明“无覆盖”和“有覆盖”时的虚函数表的样子。没有覆盖父类的虚函数是毫无意义的。我之所以要讲述没有覆盖的情况，主要目的是为了给一个对比。在比较之下，我们可以更加清楚地知道其内部的具体实现。

无虚数覆盖

下面，再让我们来看看继承时的虚函数表是什么样的。假设有如下所示的一个继承关系：



请注意，在这个继承关系中，子类没有重载任何父类的函数。那么，在派生类的实例中，Derive d; 的虚函数表：

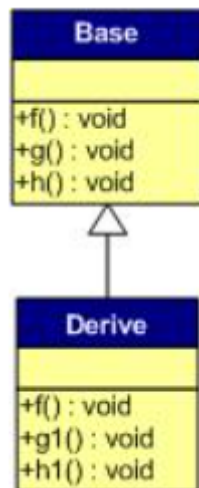


我们可以看到下面几点：

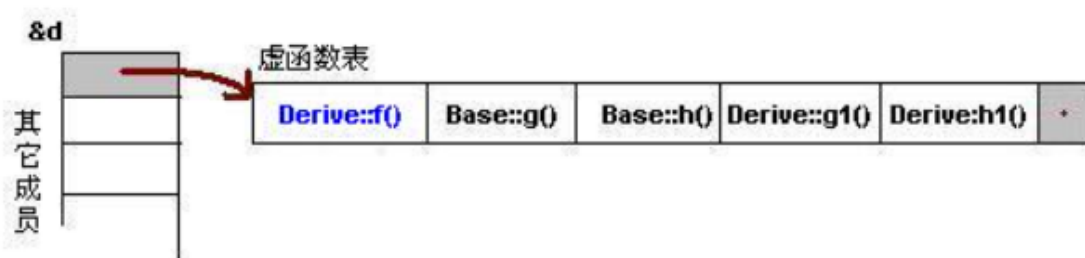
- 1) 虚函数按照其声明顺序放于表中。
- 2) 父类的虚函数在子类的虚函数前面。

有虚数覆盖

覆盖父类的虚函数是很显然的事情，不然，虚函数就变得毫无意义。下面，我们来看一下，如果子类中有虚函数重载了父类的虚函数，会是一个什么样子？假设，我们有下面这样的一个继承关系。



为了让大家看到被继承过后的效果，在这个类的设计中，我只覆盖了父类的一个函数：f()。那么，对于派生类的实例，其虚函数表会是下面的一个样子：



我们从表中可以看到下面几点，

- 1) 覆盖的f()函数被放到了虚表中原来父类虚函数的位置。
- 2) 没有被覆盖的函数依旧。

这样，我们就可以看到对于下面这样的程序，

```
Base *b = new Derive();
b->f();
```

由b所指的内存中的虚函数表的f()的位置已经被Derive::f()函数地址所取代，于是在实际调用发生时，是Derive::f()被调用了。这就实现了多态。