# 问题

虚函数可以说是在涉及C++的面试问题中经久不衰的话题了，这里就介绍一下虚函数的概念以及相关的常见问题。

# 初识虚函数

- 虚函数是指在**基类内部**声明的成员函数前添加**关键字 virtual 指明的函数**
- 虚函数存在的意义是为了**实现多态，让派生类能够重写(override)**其基类的成员函数
- 派生类重写基类的虚函数时，可以添加 virtual 关键字，但不是必须这么做
- 虚函数**是动态绑定的，在运行时才确定**，而非虚函数的调用在编译时确定
- 虚函数**必须是非静态成员函数**，因为静态成员函数需要在编译时确定
- **构造函数不能是**虚函数，因为虚函数是动态绑定的，而构造函数创建时需要确定对象类型
- **析构函数一般是**虚函数
- 虚函数一旦声明，就一直是虚函数，派生类也无法改变这一事实

下面举个例子来帮助理解：

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

class  Base {
public:
    void f() { cout << "Base::f" << endl; };     //一般成员函数
    virtual void f1() { cout << "Base::f1" << endl; };  //虚函数
private:
    char aa[3];
};
class Derived : public Base {
public:
    void f1() { cout << "Derived::f1" << endl; };    //重写基类的虚函数f1()
private:
    char bb[3];
};

int main() {
    Base a;
    Derived b;
    Base *p = &a;
    Base *p0 = &b;
    Base *p1 = &a;
    Base *p2 = &b;
    Derived *p3 = &b;

    p->f();     //基类的指针调用自己的基类部分 Base::f(), 打印结果为 Base::f
    p0->f();    //基类的指针调用派生类的基类部分 Base::f(), 打印结果为 Base::f

    p1->f1();  //基类的指针调用基类自身的函数 Base::f1(), 打印结果为 Base::f1
```

```
31      p2->f1();  //基类的指针调用派生类重写的函数 Derived::f1(), 打印结果为
    Derived::f1
32      p3->f1();  //派生类调用自己重写的函数 Derived::f1(), 打印结果为 Derived::f1
33      return 0;
34  }
```

# 虚函数的工作机制

**主要思路**

虚函数表 + 虚表指针

**具体实现**

● 编译器在含有虚函数的类中创建一个虚函数表，称为vtable，这个vtable用来存放虚函数的地址。另外还隐式地设置了一个虚表指针，称为vptr，这个vptr指向了该类对象的虚函数表。

● 派生类在继承基类的同时，也会继承基类的虚函数表（暂且可以认为派生类此时包含了两个虚函数表所有的内容，具体接着看下面两种情况）。

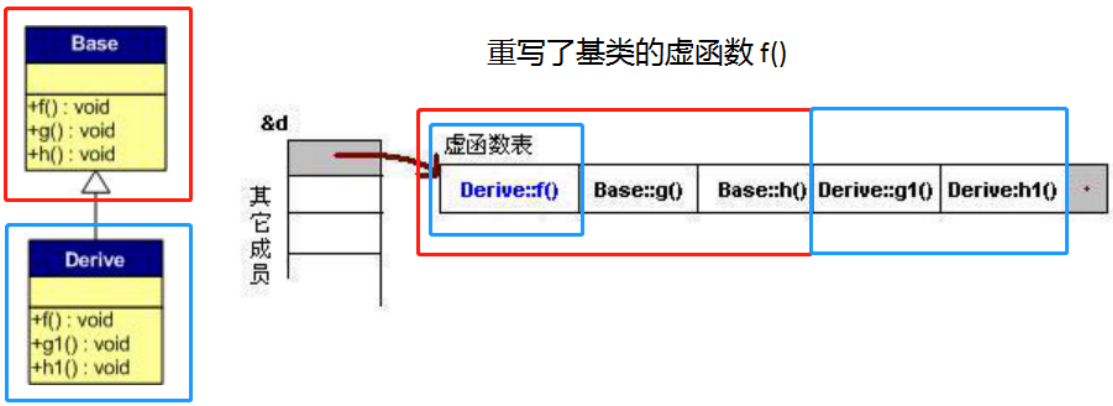
● 当派生类重写（override）了基类的虚函数时，则会将重写后的虚函数的地址 **替换掉** 由基类继承而来的虚函数表中对应虚函数的地址（有点绕，多读一遍这一句就清楚了，这也是重写的含义所在吧，也就是覆盖掉基类部分虚函数的地址）。

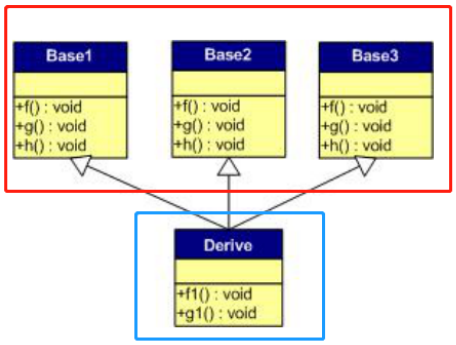
● 若派生类没有重写，则由基类继承而来的虚函数的地址将直接保存在派生类的虚函数表中。

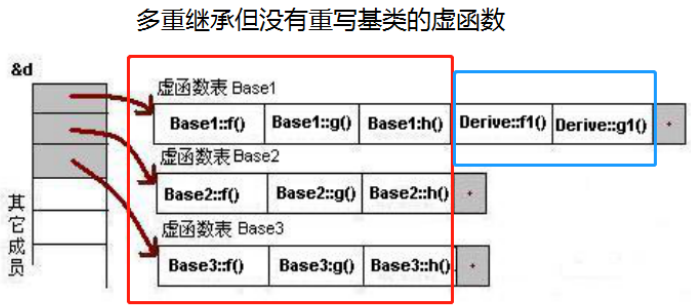
**补充**

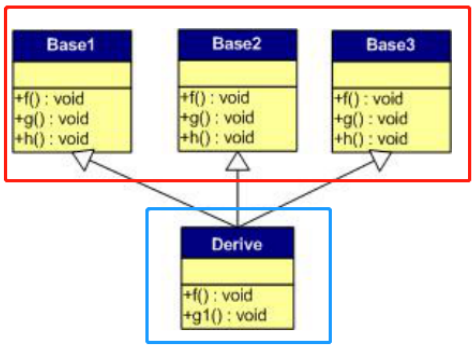
每个类都只有一个虚函数表，**该类的所有对象共享这个虚函数表**，而不是每个实例化对象都分别有一个虚函数表。

**图片理解**



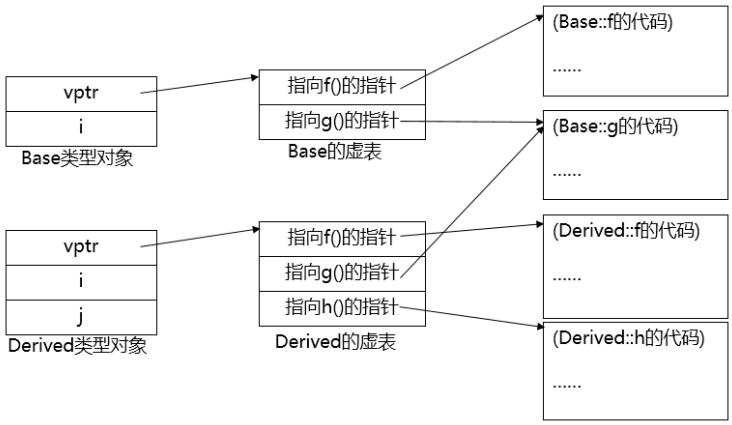





如果上面4张图还不足以帮助你理解的话，还可以更直观地看下面这张图：

```
class Base {
public:
    virtual void f();
    virtual void g();
private:
    int i;
};

class Derived: public Base {
public:
    virtual void f(); //覆盖Base::f
    virtual void h(); //新增的虚函数
private:
    int j;
};
```




# 虚函数与运行时多态

首先要理解什么是多态？

多态的实现主要分为静态多态和动态多态，静态多态主要是重载（overload），在编译时就已经确定；而动态多态是通过虚函数机制类实现，在运行时动态绑定。

**动态多态是指基类的指针指向其派生类的对象，通过基类的指针来调用派生类的成员函数。**如何理解这句话？我们再来看看本文开头的一段代码：

```
int main() {
    Base a;
    Derived b;           //派生类的对象
    Base *p1 = &a;
    Base *p2 = &b;       //基类的指针，指向派生类的对象
    p1->f1();
    p2->f1();  //基类的指针调用派生类重写的虚函数 Derived::f1(), 打印结果为
Derived::f1
    return 0;
}
```

如果基类通过引用或者指针调用的是非虚函数，无论实际的对象是什么类型，都执行基类所定义的函数。即：

```
int main() {
    Base a;
    Derived b;
    Base *p = &a;        //基类的指针，指向基类的对象
    Base *p0 = &b;       //基类的指针，指向派生类的对象
    //f()是非虚函数，所以无论指向是基类的对象a还是派生类的对象b，执行的都是基类的函数f()
    p->f();
    p0->f();
    return 0;
}
```

现在可以理解什么是运行时多态了。

**C++类的多态性是通过虚函数来实现的。如果基类通过引用或指针调用的是虚函数时，我们并不知道执行该函数的对象是什么类型的，只有在运行时才能确定调用的是基类的虚函数还是派生类中的虚函数，这就是运行时多态。**

```
int main() {
    Base a;
    Derived b;
    Base *p1 = &a;       //基类的指针，指向基类的对象
    Base *p2 = &b;       //基类的指针，指向派生类的对象
    //f1()是虚函数,只有运行时才知道真正调用的是基类的f1(),还是派生类的f1()
    p1->f1();    //p1指向的是基类的对象，所以此时调用的是基类的f1()
    p2->f1();    //p2指向的是派生类的对象，所以调用的是派生类重写后的f1()
    return 0;
}
```

总结一下：

**多态性其实就是想让基类的指针具有多种形态，能够在尽量少写代码的情况下让基类可以实现更多的功能**。比如说，派生类重写了基类的虚函数f1()之后，基类的指针就不仅可以调用自身的虚函数f1()，还可以调用其派生类的虚函数f1()，这是不是就可以多实现一些操作了呀。

# 虚函数与静态函数的区别

**静态函数在编译时已经确定，而虚函数是在运行时动态绑定的。**

虚函数因为用了虚函数表的机制，所以在调用的时候会**增加一次内存开销**。

# 参考资料

C++虚函数表解析

虚函数