虚继承

维基百科，自由的百科全书

对于继承概念中的虚函数，请参阅虚函数。

虚继承 是面向对象编程中的一种技术，是指一个指定的基类，在继承体系结构中，将其成员数据实例共享给也从这个基类型直接或间接派生的其它类。

举例来说：假如类*A*和类*B*各自从类*X*派生（非虚继承且假设类X包含一些数据成员），且类*C*同时多继承自类*A*和*B*，那么*C*的对象就会拥有两套*X*的实例数据（可分别独立访问，一般要用适当的消歧义限定符）。但是如果类*A*与*B*各自虚继承了类*X*，那么*C*的对象就只包含一套类*X*的实例数据。对于这一概念典型实现的编程语言是C++。

这一特性在多重继承应用中非常有用，可以使得虚基类对于由它直接或间接派生的类来说，拥有一个共同的基类对象实例。避免由于带有歧义的组合而产生的问题（如“菱形继承问题”）。其原理是，间接派生类（*C*）穿透了其父类（上面例子中的*A*与*B*），实质上直接继承了虚基类*X*。[1][2]

这一概念一般用于“继承”在表现为一个整体，而非几个部分的组合时。在C++中，基类可以通过使用关键字`virtual`来声明虚继承关系。

目录

问题的产生

考虑下面的类的层次和关系。

```
class Animal {
 public:
  virtual void eat();
};

class Mammal : public Animal {
 public:
  virtual void breathe();
};

class WingedAnimal : public Animal {
 public:
  virtual void flap();
};

// A bat is a winged mammal
class Bat : public Mammal, public WingedAnimal {
};

Bat bat;
```

按照上面的定义，调用bat.eat()是有歧义的，因为在Bat中有两个Animal基类（间接的），所以所有的Bat对象都有两个不同的Animal基类的子对象。因此，尝试直接引用Bat对象的Animal子对象会导致错误，因为该继承是有歧义的：

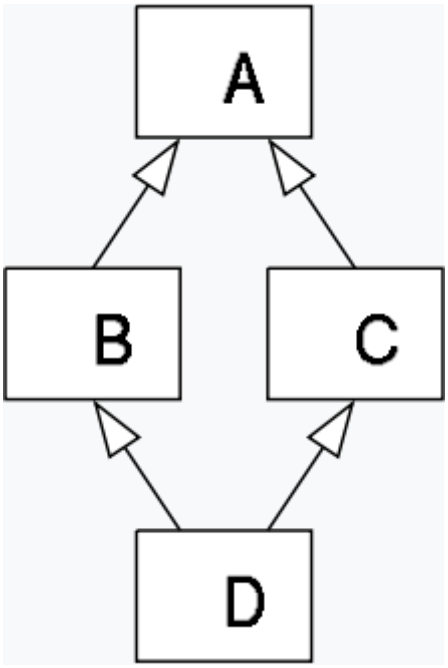
```
Bat b;
Animal &a = b; // error: which Animal subobject should a Bat cast into,
               // a Mammal::Animal or a WingedAnimal::Animal?
```

要消除歧义，需要显式的将bat转换为每一个基类子对象：

```
Bat b;
Animal &mammal = static_cast<Mammal&> (b);
Animal &winged = static_cast<WingedAnimal&> (b);
```

为了正确的调用eat()，还需要相同的可以消歧义的语句：`static_cast<Mammal&>(bat).eat()` 或 `static_cast<WingedAnimal&>(bat).eat()`.

在这个例子中，我们可能并不需要Animal被继承两次，我们只想建立一个模型来说明这层关系（Bat 属于 Animal）；Bat 是 Mammal 也是 WingedAnimal 并不意味着它是两个 Animal：Animal定义的功能由Bat来实现（上面“*是*”的属性实际上是“*实现需求*”的含义），且一个Bat只实现一次。“只*是*一个”的真正含义是Bat只有一种实现eat()的方法，无论是从Mammal的角度还是从WingedAnimal的角度来看。（在上面的第一段代码示例中我们看到eat()并没有在Mammal或WingedAnimal中被重载，所以这两个Animal子对象实际上是以相同的方式运作，但这只是一个不完善的例子，从C++的角度来看二者之间正好没有实际的区别。）



菱形类继承图示。

若将上面的关系以图形方式表示看起来类似菱形，所以这一情况也被称为**菱形继承**。虚继承可以解决这一问题。

解决方法

我们可以按如下方式重新声明上面的类：

```
class Animal {
 public:
  virtual void eat();
};

// Two classes virtually inheriting Animal:
class Mammal : public virtual Animal {
 public:
  virtual void breathe();
};

class WingedAnimal : public virtual Animal {
 public:
  virtual void flap();
};

// A bat is still a winged mammal
class Bat : public Mammal, public WingedAnimal {
};
```

Bat::WingedAnimal中的Animal部分现在和Bat::Mammal中的Animal部分是相同的了，这也就是说Bat现在有且只有一个共享的Animal部分，所以对于Bat::eat()的调用就不再有歧义了。另外，直接将Bat实例分派给Animal实例的过程也不会产生歧义了，因为现在只存在一種可以转换为Animal的Bat实体了。

因为Mammal实例的起始地址和其Animal部分的内存偏移量直到程序运行分配内存时才会明确，所以虚继承应用给Mammal和WingedAnimal建立了虚表（vtable）指针（“vpointer”）。因此“Bat”包含vpointer, Mammal, vpointer, WingedAnimal, Bat, Animal。这里共有两个虚表指针，其中最派生类的对象地址所指向的虚表指针，指向了最派生类的虚表；另一个虚表指针指向了WingedAnimal的类的虚表。Animal虚继承而来。在上面的例子里，一个分配给Mammal，另一个分配给WingedAnimal。因此每个对象占用的内存增加了两个指针的大小，但却解决了Animal的歧义问题。所有Bat类的对象都包含这两个虚指针，但是每一个对象都包含唯一的Animal对象。假设一个类Squirrel声明继承了Mammal，那么Squirrel中的Mammal对象的虚指针和Bat中的Mammal对象的虚指针是不同的，尽管他们占用的内存空间大小是相同的。这是因为在内存中Mammal到Animal的距离是相同的。虚表不同而实际上占用的空间相同。

虚基类的初始化

由于虚基类是多个派生类共享的基类，因此由谁来初始化虚基类必须明确。C++标准规定，由最派生类直接初始化虚基类。因此，对间接继承了虚基类的类，也必须能直接访问其虚继承来的祖先类，也即应知道其虚继承来的祖先类的地址偏移值。

例如，常见的“菱形”虚继承例子中，两个派生类、一个最派生类的构造函数的初始化列表中都可以给出虚基类的初始化；但只由最派生类的构造函数实际执行虚基类的初始化。

g++与虚继承

g++编译器生成的C++类实例，虚函数与*虚基类地址偏移值*共用一个虚表（vtable）。类实例的开始处即为指向所属类的**虚指针**（vptr）。实际上，一个类与它的若干祖先类（父类、祖父类、...）组成部分共用一个虚表，但各自使用的虚表部分依次相接、不相重叠。

g++编译下，一个类实例的虚指针指向该类虚表中的第一个虚函数的地址。如果该类没有虚函数（或者虚函数都写入了祖先类的虚表，覆盖了祖先类的对应虚函数），因而该类自身虚表中没有虚函数需要填入，但该类有虚继承的祖先类，则仍然必须要访问虚表中的虚基类地址偏移值。这种情况下，该类仍然需要有虚表，该类实例的虚指针指向类虚表中一个值为0的条目。

该类其它的虚函数的地址依次填在虚表中第一个虚函数条目之后（内存地址自低向高方向）。虚表中第一个虚函数条目之前（内存地址自高向低方向），依次填入了typeinfo（用于RTTI）、虚指针到整个对象开始处的偏移值、虚基类地址偏移值。因此，如果一个类虚继承了两个类，那么对于32位程序，虚继承的左父类地址偏移值位于vptr-0x0c，虚继承的右父类地址偏移值位于vptr-0x10.

一个类的祖先类有复杂的虚继承关系，则该类的各个虚基类偏移值在虚表中的存储顺序尊重自该类到祖先的深度优先遍历次序。

Microsoft Visual C++与虚继承

Microsoft Visual C++与g++不同，把类的虚函数与虚基类地址偏移值分别放入了两个虚表中，前者称为虚函数表vftbl，后者称虚基类表vbtbl。因此一个类实例可能有两个虚指针分别指向类的虚函数表与虚基类表，这两个虚指针分别称为虚函数表指针vftbl与虚基类表指针vbtbl。当然，类实例也可以只有一个虚指针，或者没有虚指针。虚指针总是放在类实例的数据成员之前，且虚函数表指针总是在虚基类表指针之前。因而，对于某个类实例来说，如果它有虚基类指针，那么虚基类指针可能在类实例的0字节偏移处，也可能在类实例的4字节偏移处（对于32位程序来说），这给类成员函数指针的实现带来了很大麻烦。

一个类的虚基类指针指向的虚基类表的首个条目，该条目的值是虚基类表指针所在的地址到该类的实例的内存首地址的偏移值。即`&(obj.vbtbl) - &obj`。虚基类第2、第3、... 个条目依次为该类的最左虚继承父类、次左虚继承父类、...的内存地址相对于虚基类表指针自身地址，即 &(obj.vbtbl)的偏移值。

如果一个类同时有虚继承的父类与祖父类，则虚祖父类放在虚父类前面。

另外需要注意的是，类的虚函数表的第一项之前的项（即*(obj.vftbl-1)）为最派生类实例的内存首地址到当前虚函数表指针的偏移值，即mostDerivedObj-obj.vftbl。派生类的虚函数覆盖基类的虚函数时，在基类的虚函数表的对应条目写入的是一个“桩”(thunk)函数的入口地址，以调整this指针指向到派生类实例的地址，再调用派生类的对应的虚函数。例如：this -= offset; call DerivedClass:virtFunc;

虚继承的应用：不可派生的finally类

一个类如果不希望被继承，类似于Java中的具有finally性质的类，这在C++中可以用虚继承来实现：

```cpp
template<typename T> class MakeFinally{
   private:
       MakeFinally(){};//只有MakeFinally的友类才可以构造MakeFinally
       ~MakeFinally(){};
   friend T;
};

class MyClass:public virtual  MakeFinally<MyClass>{};//MyClass是不可派生类

//由于虚继承，所以D要直接负责构造MakeFinally类，从而导致编译报错，所以D作为派生类是不合法的。
class D: public MyClass{};
//另外，如果D类没有实例化对象，即没有被使用，实际上D类是被编译器忽略掉而不报错


int main()
{
MyClass var1;
// D var2;  //这一行编译将导致错误，因为D类的默认构造函数不合法
}
```

参见

多态 (计算机科学)

引用

1. Andrei Milea. Solving the Diamond Problem with Virtual Inheritance. http://www.cprogramming.com/: Cprogramming.com. [2010-03-08]. “One of the problems that arises due to multiple inheritance is the diamond problem. A classical illustration of this is given by Bjarne Stroustrup (the creator of C++) in the following example:”
2. Ralph McArdell. C++/What is virtual inheritance?. http://en.allexperts.com/: All Experts. 2004-02-14 [2010-03-08]. （原始内容存档于2010-01-10）. “This is something you find may be required if you are using multiple inheritance. In that case it is possible for a class to be derived from other classes which have the same base class. In such cases, without virtual inheritance, your objects will contain more than one subobject of the base type the base classes share. Whether this is what is the required effect depends on the circumstances. If it is not then you can use virtual inheritance by specifying virtual base classes for those base types for which a whole object should only contain one such base class subobject.”