《C++深度解析教程》 – 第 51 课补充说明

面向对象程序最关键的地方在于必须能够表现三大特性：封装，继承，多态！封装指的是类中的敏感数据在外界是不能访问的；继承指的是可以对已经存在的类进行代码复用，并使得类之间存在父子关系；多态指的是相同的调用语句可以产生不同的调用结果。因此，如果希望用 C 语言完成面向对象的程序，那么肯定的，必须实现这三个特性；否则，最多只算得上基于对象的程序（程序中能够看到对象的影子，但是不完全具备面向对象的 3 大特性）。

课程中通过 void* 指针保证具体的结构体成员是不能在外界被访问的，以此模拟 C++ 中 private 和 protected。因此，在头文件中定义了如下的语句：

```
typedef void Demo;

typedef void Derived;
```

Demo 和 Derived 的本质依旧是 void， 所以，用 Demo* 指针和 Derived* 指针指向具体的对象时，无法访问对象中的成员变量，这样就达到了“外界无法访问类中私有成员”的封装效果！

继承的本质是父类成员与子类成员的叠加，所以在用 C 语言写面向对象程序的时候，可以直接考虑结构体成员的叠加即可。课程中的实现直接将 struct ClassDemo d 作为 struct ClassDerived 的第一个成员，以此表现两个自定义数据类型间的继承关系。因为 struct ClassDerived 变量的实际内存分布就是由 struct ClassDemo 的成员以及 struct ClassDerived 中新定义的成员组成的，这

样就直接实现了继承的本质，所以说 struct ClassDerived 继承自 struct ClassDemo。

下一步要实现的就是多态了！多态在 C++ 中的实现本质是通过虚函数表完成的，而虚函数表是编译器自主产生和维护的数据结构。因此，接下来要解决的问题就是如何在 C 语言中自定义虚函数表？课程中认为通过结构体变量模拟 C++ 中的虚函数表是比较理想的一种选择，所以有了下面的代码：

```
struct VTable
{
    int (*pAdd)(void*, int);
};
```

必须要注意的是，课程中由于复制粘贴的缘故，误将 pAdd 指针的类型定义成了 int (*)(Derived*, int) ， 这从 C 语言的角度算不上错误，因为 Derived* 的本质就是 void* ， 所以编译运行都没有问题。但是，从面向对象的角度，这里可以说是一种语义上的错误！因为 pAdd 必须可以指向父类中定义的 Add 函数版本，也可以指向子类中定义的 Add 函数版本，所以说用 Derived* 作为第一个参数表示实际对象并不合适，应该直接使用 void* 。

有了类型后就可以定义实际的虚函数表了，在 C 语言中用具有文件作用域的全局变量表示实际的虚函数表是最合适的，因此有了下面的代码：

```
// 父类对象使用的虚函数表
```

```
static struct VTable g_Demo_vtbl =
{
    Demo_Virtual_Add
};
// 子类对象使用的虚函数表
static struct VTable g_Derived_vtbl =
{
    Derived_Virtual_Add
};
```

每个对象中都拥有一个指向虚函数表的指针，而所有父类对象都指向 g_Demo_vtbl，所以所有子类对象都指向 g_Derived_vtbl。当一切就绪后，实际调用虚函数的过程就是通过虚函数表中的对应指针来完成的。