**C++ 对象的内存布局(上)**

**陈皓**

<http://blog.csdn.net/haoel>

[**点击这里查看下篇＞＞＞**](http://blog.csdn.net/haoel/archive/2008/10/15/3081385.aspx)

**前言**

07年12月，我写了一篇《[C++虚函数表解析](http://blog.csdn.net/haoel/archive/2007/12/18/1948051.aspx)》的文章，引起了大家的兴趣。有很多朋友对我的文章留了言，有鼓励我的，有批评我的，还有很多问问题的。我在这里一并对大家的留言表示感谢。这也是我为什么再写一篇续言的原因。因为，在上一篇文章中，我用了的示例都是非常简单的，主要是为了说明一些机理上的问题，也是为了图一些表达上方便和简单。不想，这篇文章成为了打开C++对象模型内存布局的一个引子，引发了大家对C++对象的更深层次的讨论。当然，我之前的文章还有很多方面没有涉及，从我个人感觉下来，在谈论虚函数表里，至少有以下这些内容没有涉及：

1）有成员变量的情况。

2）有重复继承的情况。

3）有虚拟继承的情况。

4）有钻石型虚拟继承的情况。

这些都是我本篇文章需要向大家说明的东西。所以，这篇文章将会是《[C++虚函数表解析](http://blog.csdn.net/haoel/archive/2007/12/18/1948051.aspx)》的一个续篇，也是一篇高级进阶的文章。我希望大家在读这篇文章之前对C++有一定的基础和了解，并能先读我的上一篇文章。因为这篇文章的深度可能会比较深，而且会比较杂乱，我希望你在读本篇文章时不会有大脑思维紊乱导致大脑死机的情况。;-)

**对象的影响因素**

简而言之，我们一个类可能会有如下的影响因素：

1）成员变量

2）虚函数（产生虚函数表）

3）单一继承（只继承于一个类）

4）多重继承（继承多个类）

5）重复继承（继承的多个父类中其父类有相同的超类）

6）虚拟继承（使用virtual方式继承，为了保证继承后父类的内存布局只会存在一份）

上述的东西通常是C++这门语言在语义方面对对象内部的影响因素，当然，还会有编译器的影响（比如优化），还有字节对齐的影响。在这里我们都不讨论，我们只讨论C++语言上的影响。

本篇文章着重讨论下述几个情况下的C++对象的内存布局情况。

1）**单一的一般继承**（带成员变量、虚函数、虚函数覆盖）

2）**单一的虚拟继承**（带成员变量、虚函数、虚函数覆盖）

3）**多重继承**（带成员变量、虚函数、虚函数覆盖）

4）**重复多重继承**（带成员变量、虚函数、虚函数覆盖）

5）**钻石型的虚拟多重继承**（带成员变量、虚函数、虚函数覆盖）

我们的目标就是，让事情越来越复杂。

**知识复习**

我们简单地复习一下，我们可以通过对象的地址来取得虚函数表的地址，如：

          typedef void(\*Fun)(void);

            Base b;

            Fun pFun = NULL;

            cout << "虚函数表地址：" << (int\*)(&b) << endl;

            cout << "虚函数表 — 第一个函数地址：" << (int\*)\*(int\*)(&b) << endl;

            // Invoke the first virtual function

            pFun = (Fun)\*((int\*)\*(int\*)(&b));

            pFun();

我们同样可以用这种方式来取得整个对象实例的内存布局。因为这些东西在内存中都是连续分布的，我们只需要使用适当的地址偏移量，我们就可以获得整个内存对象的布局。

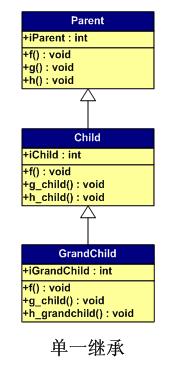
本篇文章中的例程或内存布局主要使用如下编译器和系统：

1）Windows XP 和 VC++ 2003

2）Cygwin 和 G++ 3.4.4

**单一的一般继承**

下面，我们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，父类，子类，孙子类都有自己的一个成员变量。而了类覆盖了父类的f()方法，孙子类覆盖了子类的g\_child()及其超类的f()。

我们的源程序如下所示：

class Parent {

public:

    int iparent;

    Parent ():iparent (10) {}

    virtual void f() { cout << " Parent::f()" << endl; }

    virtual void g() { cout << " Parent::g()" << endl; }

    virtual void h() { cout << " Parent::h()" << endl; }

};

class Child : public Parent {

public:

    int ichild;

    Child():ichild(100) {}

    virtual void f() { cout << "Child::f()" << endl; }

    virtual void g\_child() { cout << "Child::g\_child()" << endl; }

    virtual void h\_child() { cout << "Child::h\_child()" << endl; }

};

class GrandChild : public Child{

public:

    int igrandchild;

    GrandChild():igrandchild(1000) {}

    virtual void f() { cout << "GrandChild::f()" << endl; }

    virtual void g\_child() { cout << "GrandChild::g\_child()" << endl; }

    virtual void h\_grandchild() { cout << "GrandChild::h\_grandchild()" << endl; }

};

我们使用以下程序作为测试程序：（下面程序中，我使用了一个int\*\* pVtab 来作为遍历对象内存布局的指针，这样，我就可以方便地像使用数组一样来遍历所有的成员包括其虚函数表了，在后面的程序中，我也是用这样的方法的，请不必感到奇怪，）

    typedef void(\*Fun)(void);

    GrandChild gc;

    int\*\* pVtab = (int\*\*)&gc;

    cout << "[0] GrandChild::\_vptr->" << endl;

    for (int i=0; (Fun)pVtab[0][i]!=NULL; i++){

                pFun = (Fun)pVtab[0][i];

                cout << "    ["<<i<<"] ";

                pFun();

    }

    cout << "[1] Parent.iparent = " << (int)pVtab[1] << endl;

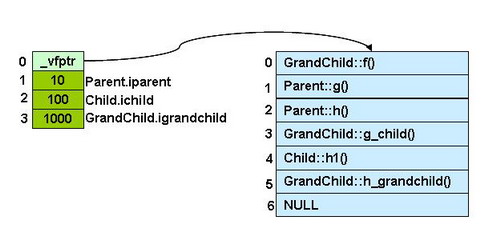
    cout << "[2] Child.ichild = " << (int)pVtab[2] << endl;

    cout << "[3] GrandChild.igrandchild = " << (int)pVtab[3] << endl;

其运行结果如下所示：（在VC++ 2003和G++ 3.4.4下）

|  |
| --- |
| **[0] GrandChild::\_vptr->**  **[0] GrandChild::f()**  **[1] Parent::g()**  **[2] Parent::h()**  **[3] GrandChild::g\_child()**  **[4] Child::h1()**  **[5] GrandChild::h\_grandchild()**  **[1] Parent.iparent = 10**  **[2] Child.ichild = 100**  **[3] GrandChild.igrandchild = 1000** |

使用图片表示如下：



可见以下几个方面：

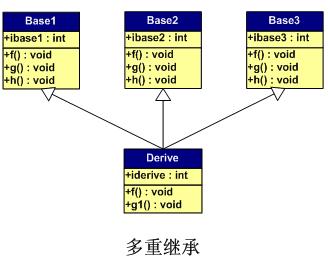
1）虚函数表在最前面的位置。

2）成员变量根据其继承和声明顺序依次放在后面。

3）在单一的继承中，被overwrite的虚函数在虚函数表中得到了更新。

**多重继承**

下面，再让我们来看看多重继承中的情况，假设有下面这样一个类的继承关系。注意：子类只overwrite了父类的f()函数，而还有一个是自己的函数（我们这样做的目的是为了用g1()作为一个标记来标明子类的虚函数表）。而且每个类中都有一个自己的成员变量：



我们的类继承的源代码如下所示：父类的成员初始为10，20，30，子类的为100

class Base1 {

public:

    int ibase1;

    Base1():ibase1(10) {}

    virtual void f() { cout << "Base1::f()" << endl; }

    virtual void g() { cout << "Base1::g()" << endl; }

    virtual void h() { cout << "Base1::h()" << endl; }

};

class Base2 {

public:

    int ibase2;

    Base2():ibase2(20) {}

    virtual void f() { cout << "Base2::f()" << endl; }

    virtual void g() { cout << "Base2::g()" << endl; }

    virtual void h() { cout << "Base2::h()" << endl; }

};

class Base3 {

public:

    int ibase3;

    Base3():ibase3(30) {}

    virtual void f() { cout << "Base3::f()" << endl; }

    virtual void g() { cout << "Base3::g()" << endl; }

    virtual void h() { cout << "Base3::h()" << endl; }

};

class Derive : public Base1, public Base2, public Base3 {

public:

    int iderive;

    Derive():iderive(100) {}

    virtual void f() { cout << "Derive::f()" << endl; }

    virtual void g1() { cout << "Derive::g1()" << endl; }

};

我们通过下面的程序来查看子类实例的内存布局：下面程序中，注意我使用了一个s变量，其中用到了sizof(Base)来找下一个类的偏移量。（因为我声明的是int成员，所以是4个字节，所以没有对齐问题。关于内存的对齐问题，大家可以自行试验，我在这里就不多说了）

             typedef void(\*Fun)(void);

               Derive d;

                int\*\* pVtab = (int\*\*)&d;

                cout << "[0] Base1::\_vptr->" << endl;

                pFun = (Fun)pVtab[0][0];

                cout << "     [0] ";

                pFun();

                pFun = (Fun)pVtab[0][1];

                cout << "     [1] ";pFun();

                pFun = (Fun)pVtab[0][2];

                cout << "     [2] ";pFun();

                pFun = (Fun)pVtab[0][3];

                cout << "     [3] "; pFun();

                pFun = (Fun)pVtab[0][4];

                cout << "     [4] "; cout<<pFun<<endl;

                cout << "[1] Base1.ibase1 = " << (int)pVtab[1] << endl;

**int s = sizeof(Base1)/4;**

                cout << "[" << s << "] Base2::\_vptr->"<<endl;

                pFun = (Fun)pVtab[s][0];

                cout << "     [0] "; pFun();

                Fun = (Fun)pVtab[s][1];

                cout << "     [1] "; pFun();

                pFun = (Fun)pVtab[s][2];

                cout << "     [2] "; pFun();

                pFun = (Fun)pVtab[s][3];

                out << "     [3] ";

                cout<<pFun<<endl;

                cout << "["<< s+1 <<"] Base2.ibase2 = " << (int)pVtab[s+1] << endl;

**s = s + sizeof(Base2)/4;**

                cout << "[" << s << "] Base3::\_vptr->"<<endl;

                pFun = (Fun)pVtab[s][0];

                cout << "     [0] "; pFun();

                pFun = (Fun)pVtab[s][1];

                cout << "     [1] "; pFun();

                pFun = (Fun)pVtab[s][2];

                cout << "     [2] "; pFun();

                pFun = (Fun)pVtab[s][3];

                 cout << "     [3] ";

                cout<<pFun<<endl;

                s++;

                cout << "["<< s <<"] Base3.ibase3 = " << (int)pVtab[s] << endl;

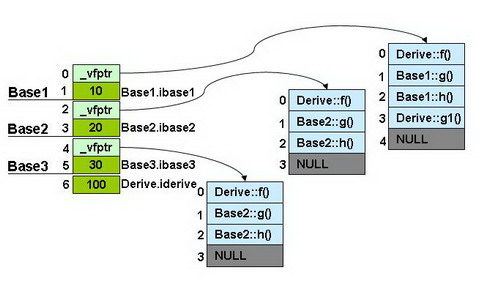
                s++;

                cout << "["<< s <<"] Derive.iderive = " << (int)pVtab[s] << endl;

其运行结果如下所示：（在VC++ 2003和G++ 3.4.4下）

|  |
| --- |
| **[0] Base1::\_vptr->**  **[0] Derive::f()**  **[1] Base1::g()**  **[2] Base1::h()**  **[3] Driver::g1()**  **[4] 00000000       注意：在GCC下，这里是1**  **[1] Base1.ibase1 = 10**  **[2] Base2::\_vptr->**  **[0] Derive::f()**  **[1] Base2::g()**  **[2] Base2::h()**  **[3] 00000000       注意：在GCC下，这里是1**  **[3] Base2.ibase2 = 20**  **[4] Base3::\_vptr->**  **[0] Derive::f()**  **[1] Base3::g()**  **[2] Base3::h()**  **[3] 00000000**  **[5] Base3.ibase3 = 30**  **[6] Derive.iderive = 100** |

使用图片表示是下面这个样子：



我们可以看到：

1）  每个父类都有自己的虚表。

2）  子类的成员函数被放到了第一个父类的表中。

3）  内存布局中，其父类布局依次按声明顺序排列。

4）  每个父类的虚表中的f()函数都被overwrite成了子类的f()。这样做就是为了解决不同的父类类型的指针指向同一个子类实例，而能够调用到实际的函数。