**C++ 对象的内存布局(下)**

**陈皓**

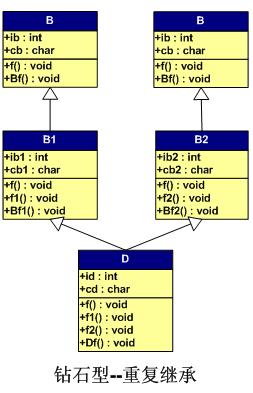
<http://blog.csdn.net/haoel>

[**＜＜＜点击这里查看上篇**](http://blog.csdn.net/haoel/archive/2008/10/15/3081328.aspx)

**重复继承**

下面我们再来看看，发生重复继承的情况。所谓重复继承，也就是某个基类被间接地重复继承了多次。

下图是一个继承图，我们重载了父类的f()函数。



其类继承的源代码如下所示。其中，每个类都有两个变量，一个是整形（4字节），一个是字符（1字节），而且还有自己的虚函数，自己overwrite父类的虚函数。如子类D中，f()覆盖了超类的函数， f1() 和f2() 覆盖了其父类的虚函数，Df()为自己的虚函数。

class B

{

    public:

        int ib;

        char cb;

    public:

        B():ib(0),cb('B') {}

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

        virtual void Bf() { cout << "B::Bf()" << endl;}

};

class B1 :  public B

{

    public:

        int ib1;

        char cb1;

    public:

        B1():ib1(11),cb1('1') {}

        virtual void f() { cout << "B1::f()" << endl;}

        virtual void f1() { cout << "B1::f1()" << endl;}

        virtual void Bf1() { cout << "B1::Bf1()" << endl;}

};

class B2:  public B

{

    public:

        int ib2;

        char cb2;

    public:

        B2():ib2(12),cb2('2') {}

        virtual void f() { cout << "B2::f()" << endl;}

        virtual void f2() { cout << "B2::f2()" << endl;}

        virtual void Bf2() { cout << "B2::Bf2()" << endl;}

};

class D : public B1, public B2

{

    public:

        int id;

        char cd;

    public:

        D():id(100),cd('D') {}

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

        virtual void f1() { cout << "D::f1()" << endl;}

        virtual void f2() { cout << "D::f2()" << endl;}

        virtual void Df() { cout << "D::Df()" << endl;}

};

我们用来存取子类内存布局的代码如下所示：（在VC++ 2003和G++ 3.4.4下）

    typedef void(\*Fun)(void);

    int\*\* pVtab = NULL;

    Fun pFun = NULL;

    D d;

    pVtab = (int\*\*)&d;

    cout << "[0] D::B1::\_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] ";    pFun();

    pFun = (Fun)pVtab[0][5];

    cout << "     [5] 0x" << pFun << endl;

    cout << "[1] B::ib = " << (int)pVtab[1] << endl;

    cout << "[2] B::cb = " << (char)pVtab[2] << endl;

    cout << "[3] B1::ib1 = " << (int)pVtab[3] << endl;

    cout << "[4] B1::cb1 = " << (char)pVtab[4] << endl;

    cout << "[5] D::B2::\_vptr->" << endl;

    pFun = (Fun)pVtab[5][0];

    cout << "     [0] ";    pFun();

    pFun = (Fun)pVtab[5][1];

    cout << "     [1] ";    pFun();

    pFun = (Fun)pVtab[5][2];

    cout << "     [2] ";    pFun();

    pFun = (Fun)pVtab[5][3];

    cout << "     [3] ";    pFun();

    pFun = (Fun)pVtab[5][4];

    cout << "     [4] 0x" << pFun << endl;

    cout << "[6] B::ib = " << (int)pVtab[6] << endl;

    cout << "[7] B::cb = " << (char)pVtab[7] << endl;

    cout << "[8] B2::ib2 = " << (int)pVtab[8] << endl;

    cout << "[9] B2::cb2 = " << (char)pVtab[9] << endl;

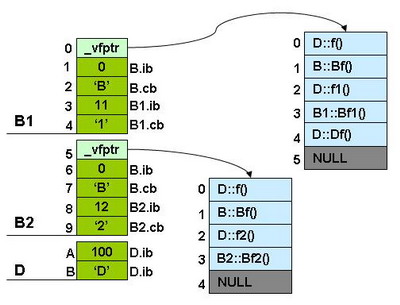
    cout << "[10] D::id = " << (int)pVtab[10] << endl;

    cout << "[11] D::cd = " << (char)pVtab[11] << endl;

程序运行结果如下：

|  |  |
| --- | --- |
| **GCC 3.4.4** | **VC++ 2003** |
| **[0] D::B1::\_vptr->**  **[0] D::f()**  **[1] B::Bf()**  **[2] D::f1()**  **[3] B1::Bf1()**  **[4] D::f2()**  **[5] 0x1**  **[1] B::ib = 0**  **[2] B::cb = B**  **[3] B1::ib1 = 11**  **[4] B1::cb1 = 1**  **[5] D::B2::\_vptr->**  **[0] D::f()**  **[1] B::Bf()**  **[2] D::f2()**  **[3] B2::Bf2()**  **[4] 0x0**  **[6] B::ib = 0**  **[7] B::cb = B**  **[8] B2::ib2 = 12**  **[9] B2::cb2 = 2**  **[10] D::id = 100**  **[11] D::cd = D** | **[0] D::B1::\_vptr->**  **[0] D::f()**  **[1] B::Bf()**  **[2] D::f1()**  **[3] B1::Bf1()**  **[4] D::Df()**  **[5] 0x00000000**  **[1] B::ib = 0**  **[2] B::cb = B**  **[3] B1::ib1 = 11**  **[4] B1::cb1 = 1**  **[5] D::B2::\_vptr->**  **[0] D::f()**  **[1] B::Bf()**  **[2] D::f2()**  **[3] B2::Bf2()**  **[4] 0x00000000**  **[6] B::ib = 0**  **[7] B::cb = B**  **[8] B2::ib2 = 12**  **[9] B2::cb2 = 2**  **[10] D::id = 100**  **[11] D::cd = D** |

下面是对于子类实例中的虚函数表的图：

我们可以看见，最顶端的父类B其成员变量存在于B1和B2中，并被D给继承下去了。而在D中，其有B1和B2的实例，于是B的成员在D的实例中存在两份，一份是B1继承而来的，另一份是B2继承而来的。所以，如果我们使用以下语句，则会产生二义性编译错误：

D d;

d.ib = 0;               //二义性错误

d.B1::ib = 1;           //正确

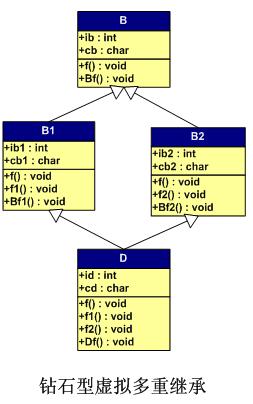
d.B2::ib = 2;           //正确

注意，上面例程中的最后两条语句存取的是两个变量。虽然我们消除了二义性的编译错误，但B类在D中还是有两个实例，这种继承造成了数据的重复，我们叫这种继承为重复继承。重复的基类数据成员可能并不是我们想要的。所以，C++引入了虚基类的概念。

**钻石型多重虚拟继承**

虚拟继承的出现就是为了解决重复继承中多个间接父类的问题的。钻石型的结构是其最经典的结构。也是我们在这里要讨论的结构：

上述的“重复继承”只需要把B1和B2继承B的语法中加上virtual 关键，就成了虚拟继承，其继承图如下所示：



上图和前面的“重复继承”中的类的内部数据和接口都是完全一样的，只是我们采用了虚拟继承：其省略后的源码如下所示：

class B {……};

class B1 : **virtual** public B{……};

class B2: **virtual** public B{……};

class D : public B1, public B2{ …… };

在查看D之前，我们先看一看**单一虚拟继承**的情况。下面是一段在VC++2003下的测试程序：（因为VC++和GCC的内存而局上有一些细节上的不同，所以这里只给出VC++的程序，GCC下的程序大家可以根据我给出的程序自己仿照着写一个去试一试）：

    int\*\* pVtab = NULL;

    Fun pFun = NULL;

    B1 bb1;

    pVtab = (int\*\*)&bb1;

    cout << "[0] B1::\_vptr->" << endl;

    pFun = (Fun)pVtab[0][0];

    cout << "     [0] ";

    pFun(); //B1::f1();

    cout << "     [1] ";

    pFun = (Fun)pVtab[0][1];

    pFun(); //B1::bf1();

    cout << "     [2] ";

    cout << pVtab[0][2] << endl;

    cout << "[1] = 0x";

    cout << (int\*)\*((int\*)(&bb1)+1) <<endl; //B1::ib1

    cout << "[2] B1::ib1 = ";

    cout << (int)\*((int\*)(&bb1)+2) <<endl; //B1::ib1

    cout << "[3] B1::cb1 = ";

    cout << (char)\*((int\*)(&bb1)+3) << endl; //B1::cb1

    cout << "[4] = 0x";

    cout << (int\*)\*((int\*)(&bb1)+4) << endl; //NULL

    cout << "[5] B::\_vptr->" << endl;

    pFun = (Fun)pVtab[5][0];

    cout << "     [0] ";

    pFun(); //B1::f();

    pFun = (Fun)pVtab[5][1];

    cout << "     [1] ";

    pFun(); //B::Bf();

    cout << "     [2] ";

    cout << "0x" << (Fun)pVtab[5][2] << endl;

    cout << "[6] B::ib = ";

    cout << (int)\*((int\*)(&bb1)+6) <<endl; //B::ib

    cout << "[7] B::cb = ";

其运行结果如下（我结出了GCC的和VC++2003的对比）：

|  |  |
| --- | --- |
| **GCC 3.4.4** | **VC++ 2003** |
| **[0] B1::\_vptr ->**  **[0] : B1::f()**  **[1] : B1::f1()**  **[2] : B1::Bf1()**  **[3] : 0**  **[1] B1::ib1 : 11**  **[2] B1::cb1 : 1**  **[3] B::\_vptr ->**  **[0] : B1::f()**  **[1] : B::Bf()**  **[2] : 0**  **[4] B::ib : 0**  **[5] B::cb : B**  **[6] NULL : 0** | **[0] B1::\_vptr->**  **[0] B1::f1()**  **[1] B1::Bf1()**  **[2] 0**  **[1] = 0x00454310 该地址取值后是-4**  **[2] B1::ib1 = 11**  **[3] B1::cb1 = 1**  **[4] = 0x00000000**  **[5] B::\_vptr->**  **[0] B1::f()**  **[1] B::Bf()**  **[2] 0x00000000**  **[6] B::ib = 0**  **[7] B::cb = B** |

这里，大家可以自己对比一下。关于细节上，我会在后面一并再说。

下面的测试程序是看子类D的内存布局，同样是VC++ 2003的（因为VC++和GCC的内存布局上有一些细节上的不同，而VC++的相对要清楚很多，所以这里只给出VC++的程序，GCC下的程序大家可以根据我给出的程序自己仿照着写一个去试一试）：

    D d;

    pVtab = (int\*\*)&d;

    cout << "[0] D::B1::\_vptr->" << endl;

    pFun = (Fun)pVtab[0][0];

    cout << "     [0] ";    pFun(); //D::f1();

    pFun = (Fun)pVtab[0][1];

    cout << "     [1] ";    pFun(); //B1::Bf1();

    pFun = (Fun)pVtab[0][2];

    cout << "     [2] ";    pFun(); //D::Df();

    pFun = (Fun)pVtab[0][3];

    cout << "     [3] ";

    cout << pFun << endl;

    //cout << pVtab[4][2] << endl;

    cout << "[1] = 0x";

    cout <<  (int\*)((&dd)+1) <<endl; //????

    cout << "[2] B1::ib1 = ";

    cout << \*((int\*)(&dd)+2) <<endl; //B1::ib1

    cout << "[3] B1::cb1 = ";

    cout << (char)\*((int\*)(&dd)+3) << endl; //B1::cb1

    //---------------------

    cout << "[4] D::B2::\_vptr->" << endl;

    pFun = (Fun)pVtab[4][0];

    cout << "     [0] ";    pFun(); //D::f2();

    pFun = (Fun)pVtab[4][1];

    cout << "     [1] ";    pFun(); //B2::Bf2();

    pFun = (Fun)pVtab[4][2];

    cout << "     [2] ";

    cout << pFun << endl;

    cout << "[5] = 0x";

    cout << \*((int\*)(&dd)+5) << endl; // ???

    cout << "[6] B2::ib2 = ";

    cout << (int)\*((int\*)(&dd)+6) <<endl; //B2::ib2

    cout << "[7] B2::cb2 = ";

    cout << (char)\*((int\*)(&dd)+7) << endl; //B2::cb2

    cout << "[8] D::id = ";

    cout << \*((int\*)(&dd)+8) << endl; //D::id

    cout << "[9] D::cd = ";

    cout << (char)\*((int\*)(&dd)+9) << endl;//D::cd

    cout << "[10]  = 0x";

    cout << (int\*)\*((int\*)(&dd)+10) << endl;

    //---------------------

    cout << "[11] D::B::\_vptr->" << endl;

    pFun = (Fun)pVtab[11][0];

    cout << "     [0] ";    pFun(); //D::f();

    pFun = (Fun)pVtab[11][1];

    cout << "     [1] ";    pFun(); //B::Bf();

    pFun = (Fun)pVtab[11][2];

    cout << "     [2] ";

    cout << pFun << endl;

    cout << "[12] B::ib = ";

    cout << \*((int\*)(&dd)+12) << endl; //B::ib

    cout << "[13] B::cb = ";

    cout << (char)\*((int\*)(&dd)+13) <<endl;//B::cb

下面给出运行后的结果（分VC++和GCC两部份）

|  |  |
| --- | --- |
| **GCC 3.4.4** | **VC++ 2003** |
| **[0] B1::\_vptr ->**  **[0] : D::f()**  **[1] : D::f1()**  **[2] : B1::Bf1()**  **[3] : D::f2()**  **[4] : D::Df()**  **[5] : 1**  **[1] B1::ib1 : 11**  **[2] B1::cb1 : 1**  **[3] B2::\_vptr ->**  **[0] : D::f()**  **[1] : D::f2()**  **[2] : B2::Bf2()**  **[3] : 0**  **[4] B2::ib2 : 12**  **[5] B2::cb2 : 2**  **[6] D::id : 100**  **[7] D::cd : D**  **[8] B::\_vptr ->**  **[0] : D::f()**  **[1] : B::Bf()**  **[2] : 0**  **[9] B::ib : 0**  **[10] B::cb : B**  **[11] NULL : 0** | **[0] D::B1::\_vptr->**  **[0] D::f1()**  **[1] B1::Bf1()**  **[2] D::Df()**  **[3] 00000000**  **[1] = 0x0013FDC4   该地址取值后是-4**  **[2] B1::ib1 = 11**  **[3] B1::cb1 = 1**  **[4] D::B2::\_vptr->**  **[0] D::f2()**  **[1] B2::Bf2()**  **[2] 00000000**  **[5] = 0x4539260    该地址取值后是-4**  **[6] B2::ib2 = 12**  **[7] B2::cb2 = 2**  **[8] D::id = 100**  **[9] D::cd = D**  **[10]  = 0x00000000**  **[11] D::B::\_vptr->**  **[0] D::f()**  **[1] B::Bf()**  **[2] 00000000**  **[12] B::ib = 0**  **[13] B::cb = B** |

关于虚拟继承的运行结果我就不画图了（前面的作图已经让我产生了很严重的厌倦感，所以就偷个懒了，大家见谅了）

在上面的输出结果中，我用不同的颜色做了一些标明。我们可以看到如下的几点：

1）无论是GCC还是VC++，除了一些细节上的不同，其大体上的对象布局是一样的。也就是说，先是B1（黄色），然后是B2（绿色），接着是D（灰色），而B这个超类（青蓝色）的实例都放在最后的位置。

2）关于虚函数表，尤其是第一个虚表，GCC和VC++有很重大的不一样。但仔细看下来，还是VC++的虚表比较清晰和有逻辑性。

3）VC++和GCC都把B这个超类放到了最后，而VC++有一个NULL分隔符把B和B1和B2的布局分开。GCC则没有。

4）VC++中的内存布局有两个地址我有些不是很明白，在其中我用红色标出了。取其内容是-4。接道理来说，这个指针应该是指向B类实例的内存地址（这个做法就是为了保证重复的父类只有一个实例的技术）。但取值后却不是。这点我目前还并不太清楚，还向大家请教。

5）GCC的内存布局中在B1和B2中则没有指向B的指针。这点可以理解，编译器可以通过计算B1和B2的size而得出B的偏移量。

**结束语**

C++这门语言是一门比较复杂的语言，对于程序员来说，我们似乎永远摸不清楚这门语言背着我们在干了什么。需要熟悉这门语言，我们就必需要了解C++里面的那些东西，需要我们去了解他后面的内存对象。这样我们才能真正的了解C++，从而能够更好的使用C++这门最难的编程语言。

在文章束之前还是介绍一下自己吧。我从事软件研发有十个年头了，目前是软件开发技术主管，技术方面，主攻Unix/C/C++，比较喜欢网络上的技术，比如分布式计算，网格计算，P2P，Ajax等一切和互联网相关的东西。管理方面比较擅长于团队建设，技术趋势分析，项目管理。欢迎大家和我交流，我的MSN和Email是：[haoel@hotmail.com](mailto:haoel@hotmail.com)