2020/05/22_C++_第13课_概念区分和类内存结构

笔记本: C++
创建时间: 2020/5/22 星期五 15:36
作者: ileemi
标签: 函数覆盖, 函数隐藏, 函数重载, 类的内存结构

概念区分

1、函数重载
2、函数覆盖（多态）
3、函数隐藏

函数重载

函数重载发生在同一个类的内部。这组函数具有相同的函数名，但是参数列表不相同，在函数调用过程中根据传入的实参类型，匹配最佳的函数并调用。

构成函数重载的条件：

- 相同作用域，子类与父类不构成重载
- 函数名一样，参数列表不一样（参数个数，参数顺序，参数类型）
- virtual 关键字，调用约定和返回值不做参考（不够成函数重载）

示例：

```
class A
{
public:
    virtual void Test0(int nVal)
    {
        cout << "A::Test0(int nVal)" << endl;
    }
    void Test0(float fVal)
```

```
    {
        cout << "A::Test0(float fVal)" << endl;
    }
};
class B : public A
{
public:
    //与父类不构成重载
    void Test0(char* sz)
    {
        cout << "B::Test0(char* sz)" << endl;
    }
};

int main()
{
    A a;
    a.Test0(5);
    a.Test0(5.2f);

    B b;
    //不构成重载，参数类型不匹配，报错
    //b.Test0(5.2f);
    return 0;
}
```

函数覆盖（多态）

> 子类重写的虚函数覆盖了虚表中父类的对应项

函数覆盖发生在子类与父类之间。父类中定义了一个虚函数，在子类中重新实现了这个函数，并且函数在子类和父类中具有相同的函数原型（函数名、参数列表），在调用函数过程中，根据对象的类型，调用相应类中的虚函数。

函数**覆盖**：

覆盖：编译期，编译器为子类构造虚表的时候，首先拷贝父类的虚表，然后将子类重写的虚函数覆盖父类中对应的虚表项。

```
以CWarrior派生类为例：

编译期：
  1、CHero 编译器构建虚表 vtable[] = {&CHero::UseSkills, &CHero::Speak}
  2、CWarrior 编译器拷贝父类的虚表到子类
          vtable[] = {&CHero::UseSkills, &CHero::Speak}
  3、CWarrior 编译器把子类中重写的父类的函数覆盖虚表中的对应项
          vtable[] = {&CWarrior::UseSkills, &CWarrior::Speak}]
  4、子类没有重写，使用父类的虚表

运行期：
  实例化一个战士的对象
  1、CHero     构造 __vfptr = CHero::vtable
  2、CWarrior 构造 __vfptr = CWarrior::vtable

使用：
  pHero->UseSkills(nSkill);
  先拿到虚表指针，然后取虚表的第一项(数组下标寻址)，之后拿到虚表中UseSkills的地址，接着调用
  (pHero->(__vfptr[0]))(nSkill);
```

构成函数覆盖的条件

- 子类重写的父类的虚函数（子类函数声明实现和父类函数的声明实现一致（返回值，调用约定，函数名，参数列表））
- 父类函数有 virtual 关键字，子类重写父类的虚函数时可以不写 `virtual` 关键字，为了区分子类中的重写父类虚函数和一般函数，当子类中的重写函数不写 `virtual` 关键字时，可以在重写函数的 `( )` 后添加 `override` 关键字，关键字起到一个说明性的作用。

override（重写）

override，该关键字写在子类重写函数后，表明该子类中的重写函数是重写父类的虚函数（一般函数不可以添加该关键字），父类的虚函数后面，也不可以使用该关键字。

用于子类重写父类的虚函数，起到一个说明性的作用

函数重载与函数覆盖的区别

- 函数重载是同一个类中的不同方法，函数覆盖是不同类中的同一个方法
- 函数重载的参数列表不同，函数覆盖的参数列表相同
- 函数重载调用时根据参数的类型选择对应的方法，而函数覆盖调用时根据对象类型选择对用类中的对应方法

函数隐藏（子类的名字隐藏父类的名字）

够成函数隐藏的条件：

- 子类和父类作用域不同
- 子类与父类函数同名，子类就会隐藏父类的方法，函数声明没有要求（参数，返回值，调用约定不做参考）

注意：子类函数与父类函数声明一致，父类有 virtual 关键字，构成函数覆盖

示例：

```
class A
{
public:
  void Test0(int nVal)
  {
    cout << "A::Test0(int nVal)" << endl;
  }
  virtual void Test0(char* sz)
  {
    cout << "A::Test0(char* sz)" << endl;
  }

  //虚函数不影响函数重载
};
class B :public A
{
public:
    void Test0(int nVal)
    {
      cout << "B::Test0(int nVal)" << endl;
    }

    //此时不构成函数隐藏，构成函数覆盖
    //void Test0(char* sz)
    //{
      //cout << "A::Test0(char* sz)" << endl;
    //}
};
int main()
{
    B b;
    b.Test0(5);
    //b.Test0("AAAA"); //隐藏了父类的函数

    //函数隐藏时可以通过下面的方式访问父类中被子类隐藏的方法，这里需要进行强转
    //没有强转，"AAAA"，是 const char*,const char* 不能转char*
    b.A::Test0((char*)"AAAA");
    return 0;
}
```

类内存结构

没有虚继承

1、单个类，没有虚函数的类对象
2、单个类，有虚函数的类对象
3、单重继承，没有虚函数的类对象
4、单重继承，有虚函数的类对象
5、多重继承，没有虚函数的类对象
6、多重继承，有虚函数的类对象

```
1、单个类，没有虚函数的类对象
class CTest
{
    //数据成员
    data_member1;
    data_member2;
    ...
}
内存排布（安装成员数据的顺序在内存中依次排列）：
    data_member1
    data_member2
    data_member3
    ...

2、单个类，有虚函数的类对象
class CTest
{
    //虚函数
    virtual void FunA();
    ...

    //数据成员
    data_member1;
    data_member2;
    ...
}

内存排布（有虚函数就有虚表指针）：

    __fvptr（虚表指针） ----------------------> 虚表
    data_member1
    data_member2
    data_member3
    ...

3、单重继承，没有虚函数的类对象（子类继承父类）
class CTestA
{
    dataA_member1;
    dataA_member2;
```

```
};

class CTestB : public CTestA
{
    dataB_member1;
    dataB_member2;
};
```

内存排布（父类中的数据成员排列在子类数据成员的前面）：

```
    CTestA:: dataA_member1;
    CTestA:: dataA_member2;
    ...

    CTestB:: dataB_member1;
    CTestB:: dataB_member2;
    ...
```

4、单重继承，有虚函数的类对象（子类继承父类）

```
class CTestA
{
    virtual void FunA();
    dataA_member1;
    dataA_member2;
};

class CTestB : public CTestA
{
    void FunA();
    dataB_member1;
    dataB_member2;
};
```

内存排布（有虚函数就有虚表指针）：

```
    __fvptrCTestA（虚表指针） ----------------------> 虚表
    A:: dataA_member1;
    A:: dataA_member2;
    ...

    B:: dataB_member1;
    B:: dataB_member2;
    ...
```

5、多重继承，没有虚函数的类对象

```
class CTestA
{
    dataA_member1;
    dataA_member2;
};

class CTestB
{
    dataB_member1;
    dataB_member2;
};

class CTestD : public CTestA, public CTestB
{
    dataD_member1;
    dataD_member2;
};
```

内存排布（没有虚函数就没有虚表指针）按照继承顺序排列：

```
    A:: dataA_member1;
    A:: dataA_member2;
    ...

    B:: dataB_member1;
    B:: dataB_member2;
    ...

    D:: dataD_member1;
    D:: dataD_member2;
    ...
```

6、多重继承，有虚函数的类对象

注意情况:

- 如果 CTestA 没有虚函数, 则 CTestA 没有虚表, CTestB 在内存中的位置会提前
- CTestD 重写的父类的虚函数, 会放到父类各自的虚表中
- CTestD 增加的虚函数, 会挂靠到 CTestA 的虚表中

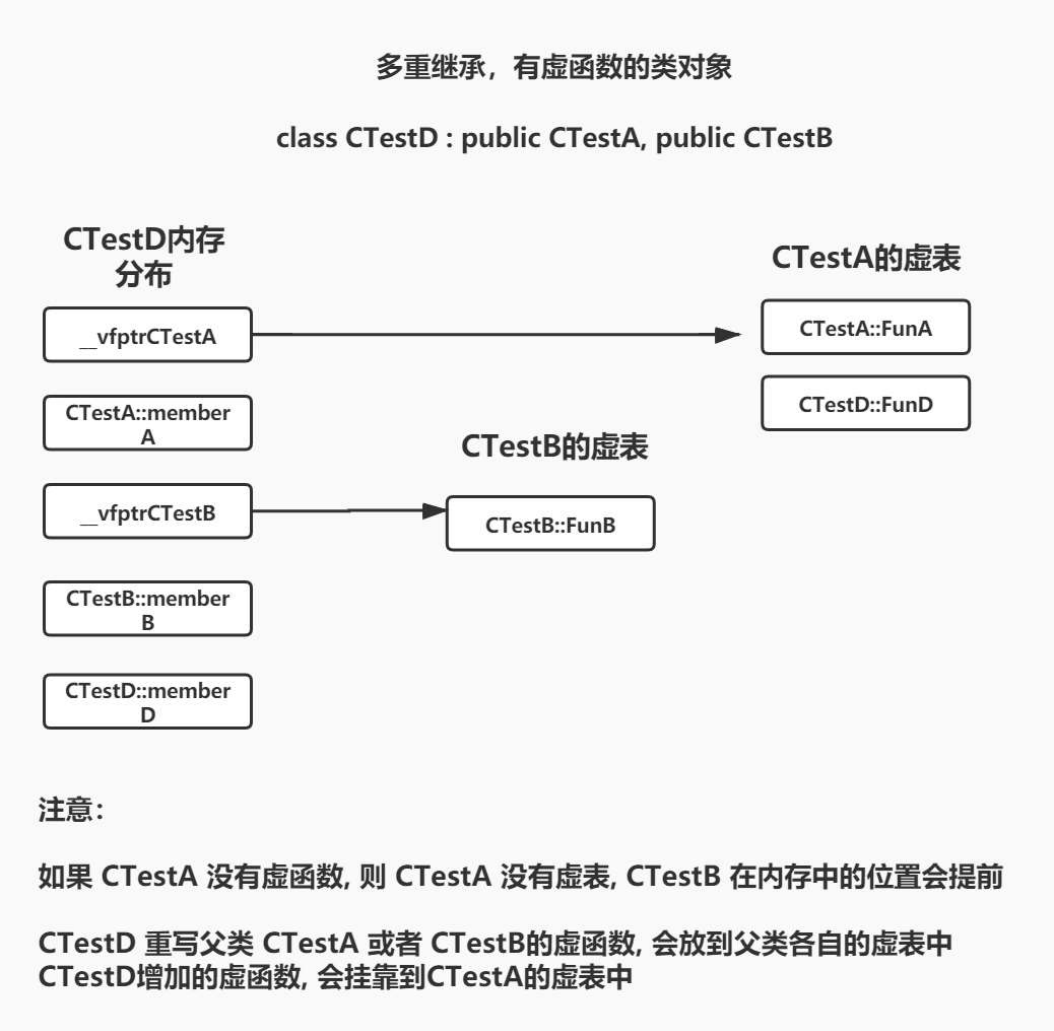
```
class CTestA
{
public:
    virtual void FunA()
    {
        cout << "CTestA::virtual void FunA()" << endl;
    }
private:
```

```
    int m_nA = 0xAAAAAAAA;
};

class CTestB
{
public:
    virtual void FunB()
    {
        cout << "CTestB::virtual void FunB()" << endl;
    }
private:
    int m_nB = 0xBBBBBBBB;
};


class CTestD : public CTestA, public CTestB
{
public:
    //形成函数覆盖，其放置在 CTestA 的虚表内
    virtual void FunA()
    {
        cout << "CTestD::virtual void FunA()" << endl;
    }
    virtual void FunB()
    {
        cout << "CTestD::virtual void FunB()" << endl;
    }

    virtual void FunD()
    {
        cout << "CTestD::virtual void FunD()" << endl;
    }
private:
    int m_nD = 0xDDDDDDDD;
};
int main()
{
    CTestD d;
    //sizeof(d) == 0x14
    /*
     (_vfptrCTestA)
    0x0019FEC8  78 9b 41 00-> (CTestA 的虚表) 0x00419B78  44 12 41 00 --
> 00411244     jmp    CTestA::FunA (0412490h)
    0x0019FECC  aa aa aa aa
    _vfptrCTestB
    0x0019FED0  80 9b 41 00 --> (CTestB 的虚表) 0x00419B80  47 14 41 00
--> 00411447  jmp    CTestB::FunB (04121C0h)
```

```
    0x0019FED4  bb bb bb bb
    0x0019FED8  dd dd dd dd
    */


    //如果CTestA没有虚函数，CTestA就没有虚表
    //sizeof(d) = 0x10

    //内存结构:
    //有虚函数的类 在内存中排列在最前
    /*
    _vfptrCTestB         CTestB 的虚表
    0x0019FECC  3c 9c 41 00 ------>0x00419C3C  47 14 41 00 -->
CTestA::FunA: 00411447  jmp    CTestB::FunB(0412490h)
    0x0019FED0  bb bb bb bb
    0x0019FED4  aa aa aa aa
    0x0019FED8  dd dd dd dd
    */

    // 子类CTestD 重写了父类CTestA的 虚函数
    // 子类CTestD 重写了父类CTestD的 虚函数
    //sizeof(d) == 0x14
    //内存分布
    /*
    _vfptrA         CTestA 的虚表          重写 CTestA 的虚函数
    0x0019FEC8  94 9b 41 00    ----> 0x00419B94  77 11 41 00 -->
CTestD::FunA:00411177  jmp     CTestD::FunA (0412520h)
    0x0019FECC  aa aa aa aa

    _vfptrB        CTestB 的虚表          重写 CTestB 的虚函数
    0x0019FED0  a0 9b 41 00  -----> 0x00419BA0  99 12 41 00 -->
CTestD::FunB: 0041129E  jmp    CTestD::FunB(0412650h)
    0x0019FED4  bb bb bb bb
    0x0019FED8  dd dd dd dd
    */

    /*
    当子类 CTestD 新增一个自己的虚函数
    子类新增加的虚函数会挂靠到父类 CTestA 的虚表里
    因为 CTestA 的虚表指针在类开始偏移为 0 的地方，这里地方是 CTestD 类
的首地址
    方便与 取虚表指针和虚函数

    */
    // sizeof(d) == 0x14
    // 内存分布
    /*
```

```
    _vfptrA          CTestA 的虚表       重写 CTestA 的虚函数
    0x0019FEC8  94 9b 41 00   ----> 0x00419B94  77 11 41 00 --->
CTestD::FunA:00411177  jmp    CTestD::FunA (0412520h)

    子类新增加的虚函数会挂靠到父类 CTestA 的虚表里
    0x0019FECC  aa aa aa aa   Add->0x00419B98  ea 11 41 00 --->
CTestD::FunD: 004111EA  jmp  CTestD::FunD(04126E0h)

    _vfptrB          CTestB 的虚表     重写 CTestB 的虚函数
    0x0019FED0  a0 9b 41 00  -----> 0x00419BA0  99 12 41 00 --->
CTestD::FunB: 0041129E  jmp   CTestD::FunB(0412650h)
    0x0019FED4  bb bb bb bb
    0x0019FED8  dd dd dd dd
    */



    return 0;
}
```

对应的内存结构如下：

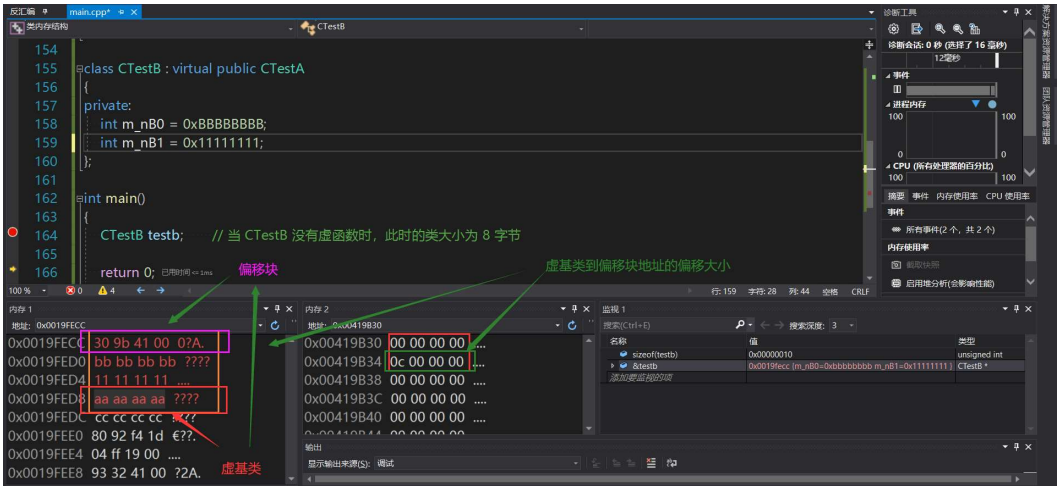


有虚继承

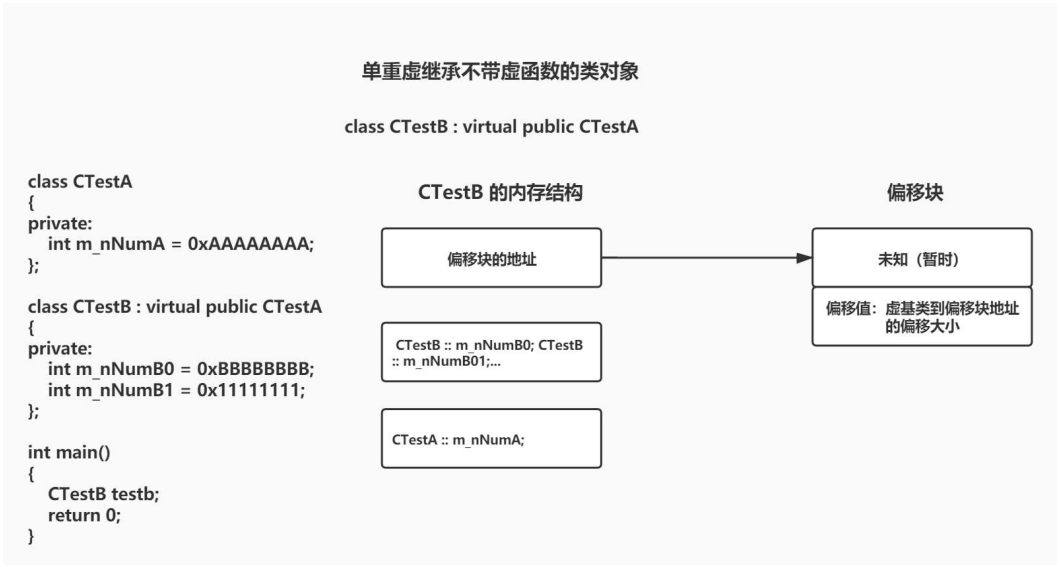
1、单重虚继承，没有虚函数的类对象
2、单重虚继承，有虚函数的类对象
3、菱形继承，没有虚函数的类对象
4、菱形继承，有虚函数的类对象

1、单重虚继承不带虚函数的类对象
代码示例：

```
class CTestA
{
private:
    int m_nNumA = 0xAAAAAAAA;
};

class CTestB : virtual public CTestA
{
private:
    int m_nNumB0 = 0xBBBBBBBB;
    int m_nNumB1 = 0x11111111;
};

int main()
{
    // 当 CTestB 没有虚函数时，此时的类大小为 8 字节
    CTestB testb;

    /*
    0x0019FECC  30 9b 41 00 ----------> 0x00419B30  00 00 00 00   //未知
    0x0019FED0  bb bb bb bb      0x00419B34  0c 00 00 00    //虚基类到偏移块地址的偏移大小
    0x0019FED4  11 11 11 11
    0x0019FED8  aa aa aa aa
    */
    return 0;
}
```



对应的内存结构如下：



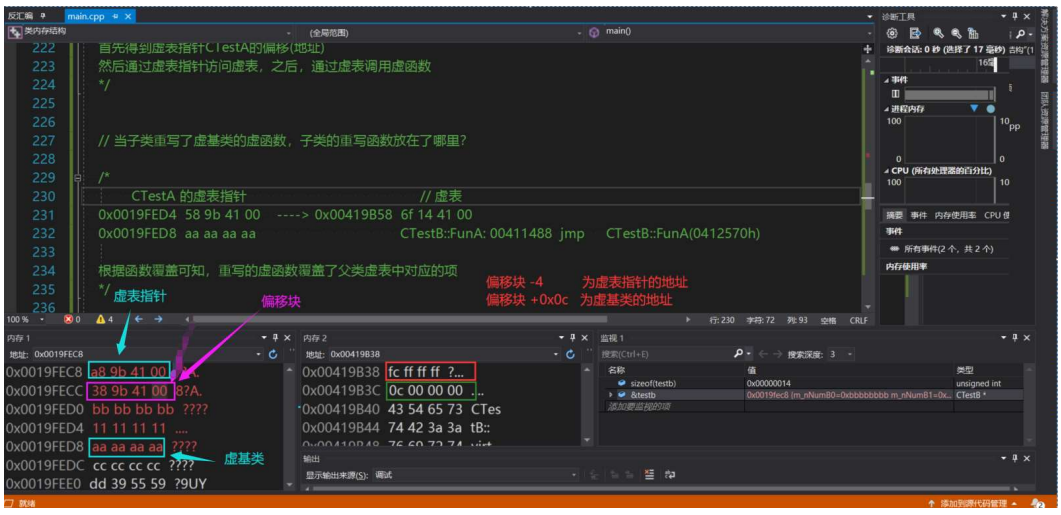
2、单重虚继承，有虚函数的类对象

代码示例：

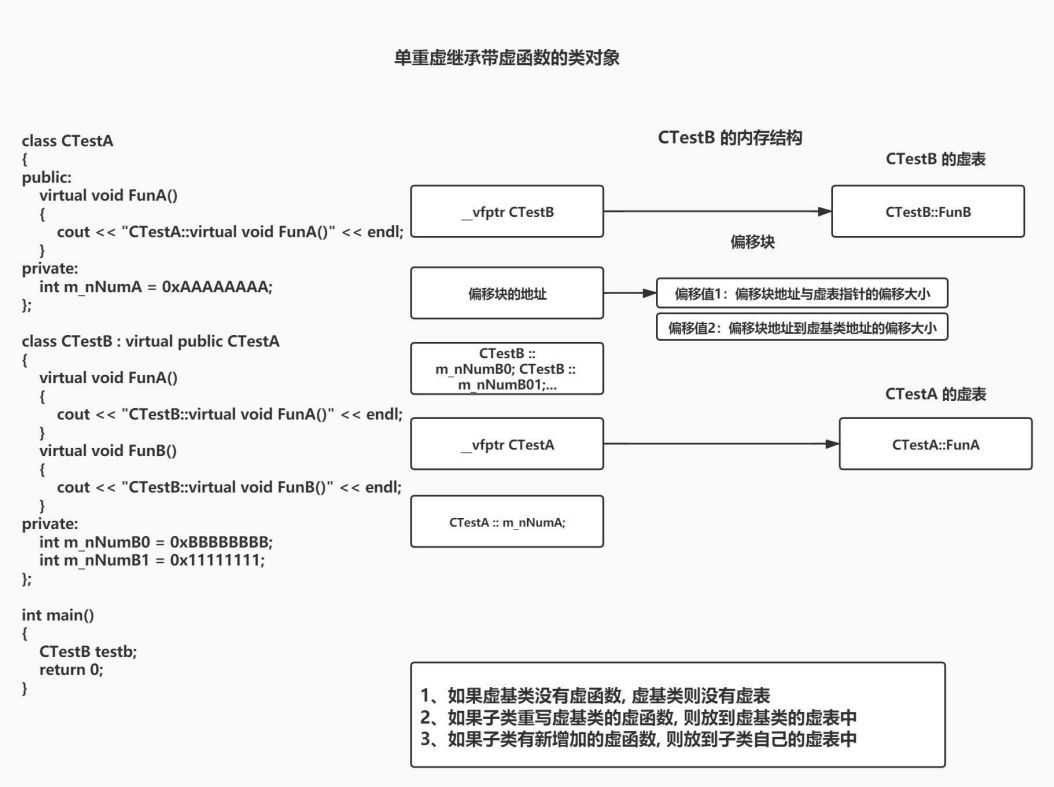
```
class CTestA
{
public:
    virtual void FunA()
    {
        cout << "CTestA::virtual void FunA()" << endl;
    }
private:
    int m_nNumA = 0xAAAAAAAA;
};

class CTestB : virtual public CTestA
{
public:
    virtual void FunA()
    {
        cout << "CTestB::virtual void FunA()" << endl;
    }
```

```
    virtual void FunB()
    {
        cout << "CTestB::virtual void FunB()" << endl;
    }
private:
    int m_nNumB0 = 0xBBBBBBBB;
    int m_nNumB1 = 0x11111111;
};

int main()
{
    CTestB testb;
    /*
                        偏移块的地址
    0x0019FEC8  5c 9b 41 00      ----> 0x00419B5C  00 00 00 0    // 未知
    0x0019FECC  bb bb bb bb   ----> 0x00419B60  0c 00 00 00  // 虚基类到
偏移块地址的偏移大小
    0x0019FED0  11 11 11 11

          CTestB的虚表指针                                        //
虚表
    0x0019FED4  58 9b 41 00     ----> 0x00419B58  6f 14 41 00
    0x0019FED8  aa aa aa aa     CTestA::FunA: 0041146F  jmp
CTestA::FunA(0412270h)
    */

    /*
    取虚函数的过程
    首先得到虚表指针CTestA的偏移(地址)
    然后通过虚表指针访问虚表，之后，通过虚表调用虚函数
    */


    // 当子类重写了虚基类的虚函数，子类的重写函数放在了哪里？

    /*
    CTestA 的虚表指针
    0x0019FED4  58 9b 41 00     ----> (虚表) 0x00419B58  6f 14 41 00
    0x0019FED8  aa aa aa aa        TestB::FunA: 00411488  jmp
CTestB::FunA(0412570h)

    根据函数覆盖可知，重写的虚函数覆盖了父类虚表中对应的项
    */


    // 当子类在多一个自己的虚函数，其存放位置
```

```
    /*
    __vfptr CTestB                          CTestB的虚表
    0x0019FEC4  58 9b 41 00 ------> 0x00419B58  8d 14 41 00
CTestB::CTestB : 0041148D  jmp     CTestB::FunB(0411D50h)
    偏移块的地址
    0x0019FEC8  64 9b 41 00 -----> 0x00419B64  fc ff ff ff (-4的补码)
0x00419B68  0c 00 00 00

    0x0019FECC  bb bb bb bb
    0x0019FED0  11 11 11 11
    CTestA 的虚表指针
    0x0019FED4  60 9b 41 00 -----> (虚表) 0x00419B60  88 14 41 00
CTestB::FunA : 00411488  jmp      CTestB::FunA(0412570h)
    0x0019FED8  aa aa aa aa
    */

    // 当父类没有虚函数的时候，其在内存中也就没有虚表
    /*
     __vfptr CTestB
    0x0019FEC8  a8 9b 41 00

    偏移块的地址
    0x0019FECC  38 9b 41 00

    CTestB的数据成员
    0x0019FED0  bb bb bb bb
    0x0019FED4  11 11 11 11

    CTestA的数据成员
    0x0019FED8  aa aa aa aa
    */
    return 0;
}
```



对应的内存结构如下：

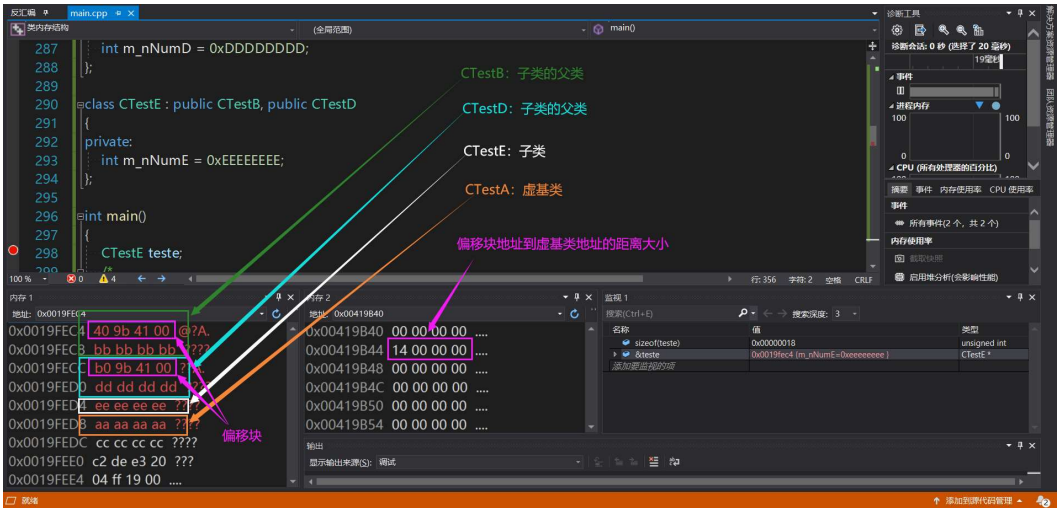


3、菱形继承，没有虚函数的类对象

代码示例：

```
/*
class CTestA;
class CTestB :virtual public CTestA;
class CTestD :virtual public CTestA;
class CTestE :public CTestB, public CTestD;
*/
class CTestA
{
private:
    int m_nNumA = 0xAAAAAAAA;
};

class CTestB : virtual public CTestA
{
private:
    int m_nNumB = 0xBBBBBBBB;
};

class CTestD : virtual public CTestA
{
private:
    int m_nNumD = 0xDDDDDDDD;
};
```

```
class CTestE : public CTestB, public CTestD
{
private:
    int m_nNumE = 0xEEEEEEEE;
};

int main()
{
    CTestE teste;

    return 0;
}
```

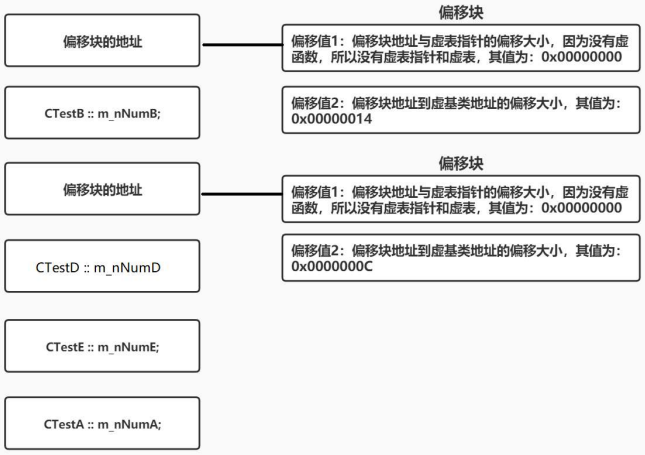


对应的内存结构如下：

菱形继承，没有虚函数的类对象

```
class CTestA
{
private:
    int m_nNumA = 0xAAAAAAAA;
};

class CTestB : virtual public CTestA
{
private:
    int m_nNumB = 0xBBBBBBBB;
};

class CTestD : virtual public CTestA
{
private:
    int m_nNumD = 0xDDDDDDDD;
};

class CTestE : public CTestB, public CTestD
{
private:
    int m_nNumE = 0xEEEEEEEE;
};

int main()
{
    CTestE teste;

    return 0;
}
```

CTestE 的内存结构



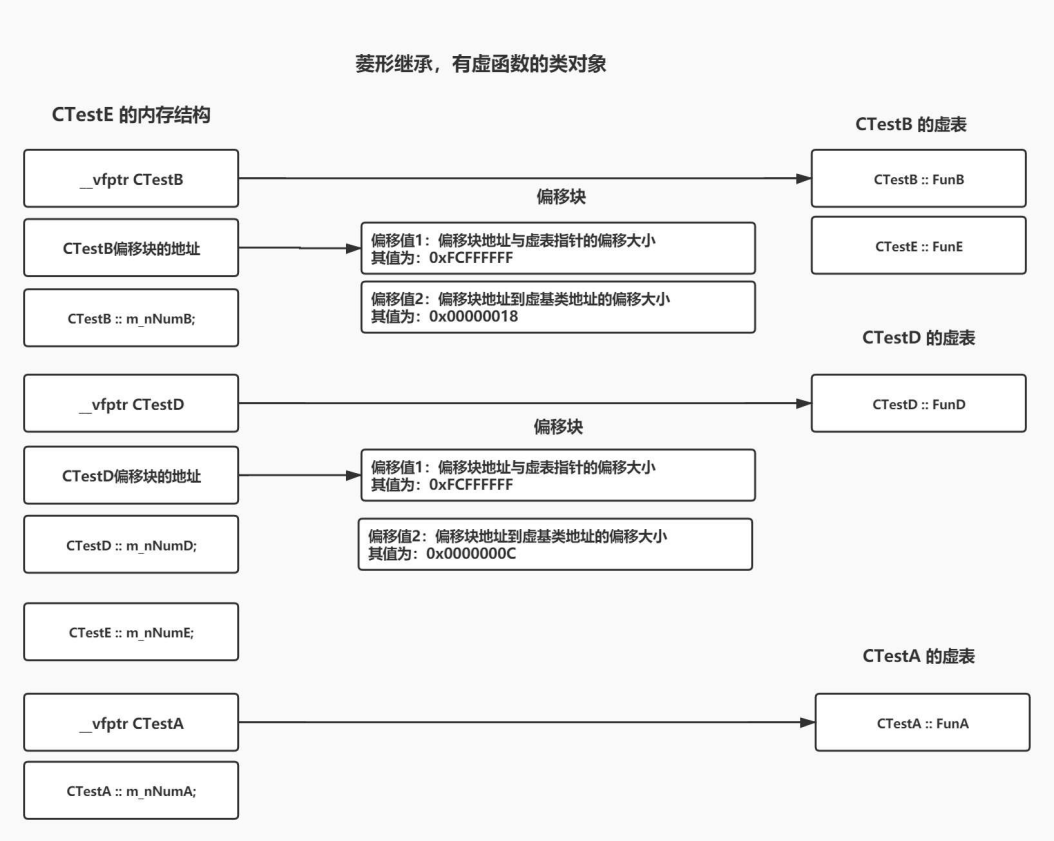
4、菱形继承，有虚函数的类对象

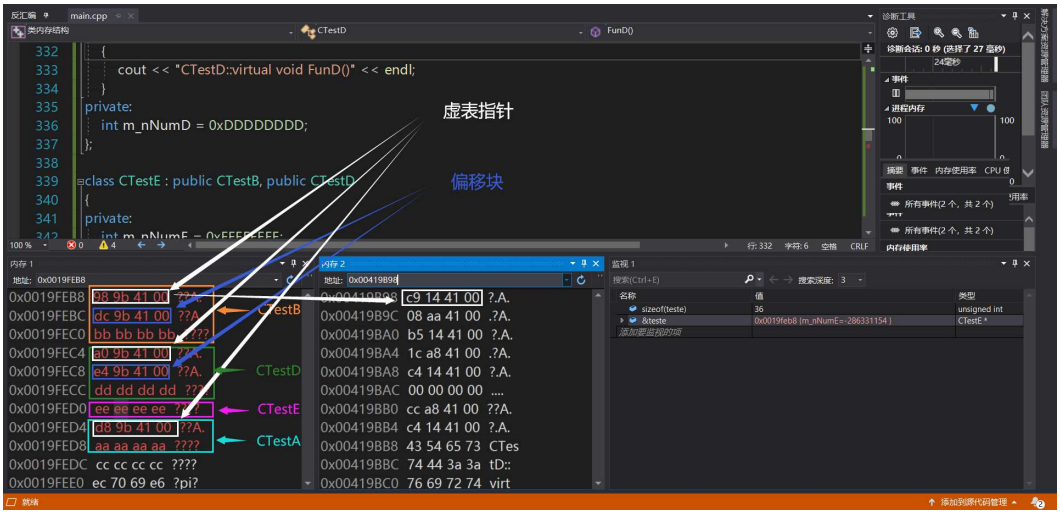
代码示例：

```
class CTestA
{
public:
    virtual void FunA()
    {
        cout << "CTestA::virtual void FunA()" << endl;
    }
private:
    int m_nNumA = 0xAAAAAAAA;
};

class CTestB : virtual public CTestA
{
public:
    virtual void FunB()
    {
        cout << "CTestB::virtual void FunB()" << endl;
    }
private:
    int m_nNumB = 0xBBBBBBBB;
};

class CTestD : virtual public CTestA
{
public:
    virtual void FunD()
    {
        cout << "CTestD::virtual void FunD()" << endl;
    }
private:
    int m_nNumD = 0xDDDDDDDD;
};

class CTestE : public CTestB, public CTestD
{
public:
    /*
    在子类中重写CTestA、CTestB、CTestD中的虚函数，
    根据函数覆盖的原则，重写的虚函数分别放入到对应的类虚表中，对原来的
进行覆盖
    */
    virtual void FunA()
    {
        cout << "CTestE::virtual void FunA()" << endl;
    }
    virtual void FunB()
```

```
    {
        cout << "CTestE::virtual void FunB()" << endl;
    }
    virtual void FunD()
    {
        cout << "CTestE::virtual void FunD()" << endl;
    }
    /*
    如果在子类中重写了自己的虚函数，其被挂靠到第一个父类的虚表中去
    */
    virtual void FunE()
    {
        cout << "CTestE::virtual void FunE()" << endl;
    }
private:
    int m_nNumE = 0xEEEEEEEE;
};

int main()
{
    CTestE teste;

    return 0;
}
```

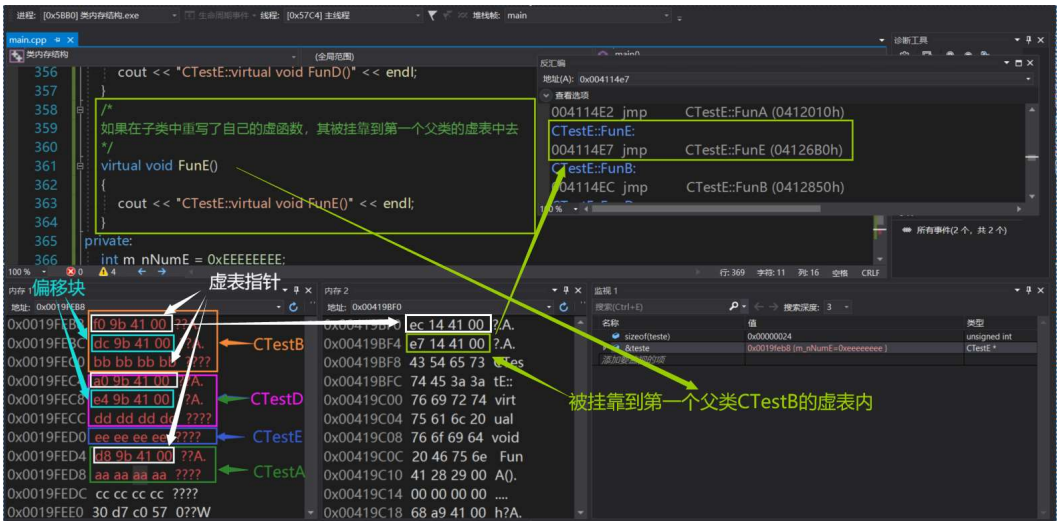
对应的内存结构如下：





- 在子类中重写CTestA、CTestB、CTestD中的虚函数，根据函数覆盖的原则，重写的虚函数分别放入到对应的类虚表中，对原来的进行覆盖。
- 如果在子类中重写了自己的虚函数，其被挂靠到第一个父类的虚表中去

对应的内存如下：



虚函数和菱形继承以后能不用就不用