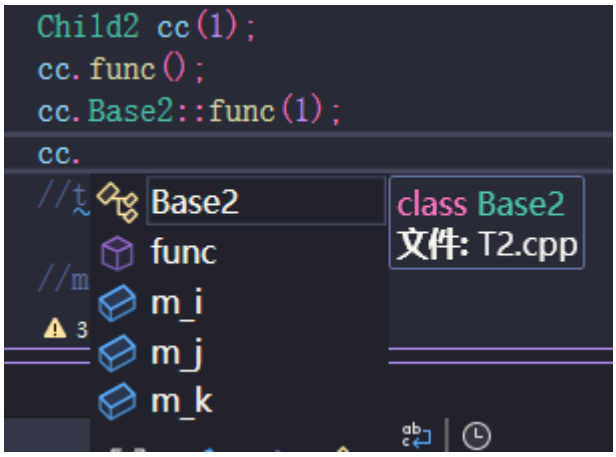
virtual

被 virtual 关键字修饰的函数，其函数地址会存在 **vtable**中，并且 当父类指针指向子类对象时，这个表里存的是，**子类**中重载了父类virtual函数的子类函数（这也是：多态中 为什么 加上 virtual修饰后，就可以确保调用 子类中的成员函数！！）

子类继承

1、子类无法继承父类的构造函数；而且，子类的构造函数必须能够 “委托调用父类的任一个构造函数”
$\color{red}{子类对象创造前，必须要要构造父类！！}$ （没有先创建父类对象，哪来的父类成员给子类继承呢!!）（如果想使得 子类的构造函数够简单，最直接的方法：保留父类的**无参构造函数**！！（编译器会隐式帮我们直接调用此无参构造，然后创建父类对象））

看下面这张图片： 

“cc.” 查看成员时，可以看见Base2（其父类）;而且，即使 Child2中，重载了func函数，使得父类的func函数被隐藏(你直接 cc. 看到的是子类中的 被重载后的成员函数)，但是 cc.Base2::func(1)任然可以调用父类中该成员函数

tips：在子类重载父类函数，使其被隐藏时，可以利用 using Base:"函数名"; 的方式，使得其在子类中可以直接被显示调用！！！（拓展：构造函数也是函数一种，而子类有自己的构造函数，也相当于重载父类构造函数了，所以 本质上用 using Base:Base，就可以不用在 子类的构造函数中，委托父类构造函数了）

```
class Base
{
public:
    Base(int i) :m_i(i) {}
    Base(int i, double j) :m_i(i), m_j(j) {}
    Base(int i, double j, string k) :m_i(i), m_j(j), m_k(k) {}

    void func(int i)
    {
        cout << "base class: i = " << i << endl;
    }

    void func(int i, string str)
    {
        cout << "base class: i = " << i << ", str = " << str << endl;
    }
```

```
    int m_i;
    double m_j;
    string m_k;
};

class Child : public Base
{
public:
    using Base::Base;
    using Base::func;
    void func()
    {
        cout << "child class: i'am luffy!!!" << endl;
    }
};

int main()
{
    Child c(250);
    c.func();
    c.func(19);
    c.func(19, "luffy");
    return 0;
}
```

2、委托构造 (有了委托构造，成员的初始化就别在初始化列表中了)**除非委托的是父类的构造函数，然后初始化子类自己的成员变量！！** ``` Test() {}; Test(int max) { this->m_max = max > 0 ? max : 100; }

```
Test(int max, int min):Test(max)
{
    this->m_min = min > 0 && min < max ? min : 1;
}

Test(int max, int min, int mid):Test(max, min)
{
    this->m_middle = mid < max && mid > min ? mid : 50;
}

// Test(int max, int min, int mid):Test(max, min),m_middle(max);    // 就别这样写了
```

(I 这种链式的构造函数调用不能形成一个闭环（死循环），否则会在运行期抛异常。

II 如果要进行多层构造函数的链式调用，建议将构造函数的调用的写在初始列表中而不是函数体内部，否则编译器会提示形参的重复定义。）

3、在修改之后的子类中，没有添加任何构造函数，而是添加了using Base::Base;这样就可以在子类中直接继承父类的所有的构造函数，通过他们去构造子类对象了。

```
class Base
{
public:
    Base(int i) :m_i(i) {}
    Base(int i, double j) :m_i(i), m_j(j) {}
    Base(int i, double j, string k) :m_i(i), m_j(j), m_k(k) {}

    int m_i;
    double m_j;
    string m_k;
};

class Child : public Base
{
public:
    using Base::Base;
};

Child c1(50);
Child c1(50,1.1);
Child c1(50,1.1,"11212");
```

谓词 仿函数 普通函数

```
/// 返回 bool 类型的仿函数叫谓词
/// 结构体谓词  接收两个参数，是二元谓词
struct Mycompare {
    bool operator()(const int p1, const int p2)const {
        return p1>p2;
    }
};

/// 类谓词
class Mycompare1 {
public:
    bool operator()(const int p1, const int p2)const {
        return p1 > p2;
    }
};

/// 普通函数
bool Mycompare2(const int p1, const int p2){
    return p1 > p2;
}

    /// 谓词 可以用在数据结构创建，改变插入后顺序 （如 map默认 key从小到大排，这里使用
Mycompare1，让key从大到小
    map<int, int, Mycompare1>m1;
```

```
    /// 普通函数  返回值是bool的，可以当作 自定义排序规则（如果是像力扣那种，算法和Mycompare都在一个class里，那么编译器有时会要求，Mycompare是static的）
    sort(v1.begin(), v1.end(), Mycompare2);

    /// 仿函数也可用在排序
    sort(v1.begin(), v1.end(),greater<int>());


    /// 仿函数
    重载函数调用操作符的类，其对象常称为函数对象
    函数对象使用重载的()时，行为类似函数调用，也叫仿函数
本质：
    函数对象(仿函数)是一个类，不是一个函数

    class MyCompare
    {
    public:
        bool operator()(int v1,int v2)
        {
        return v1 > v2;
        }
    };

    //2、函数对象可以有自己的状态
    class MyPrint
    {
    public:
        MyPrint()
        {
            count = 0;
        }
        void operator()(string test)
        {
            cout << test << endl;
            count++; //统计使用次数
        }

        int count; //内部自己的状态
    };

    //3、函数对象可以作为参数传递
    void doPrint(MyPrint &mp , string test)
    {
        mp(test);
    }

    void test03()
    {
        MyPrint myPrint;
        doPrint(myPrint, "Hello C++");
    }

    /// 仿函数还有运算/关系仿函数
```

```
negative<int>p; n=20;  p(n); // n变成-20
plus<int> p;cout << p(10, 20) << endl; // 输出30
```

有关greater/less 的排序构造问题

/// 仿函数中 greater代表的是，先前排序中，留下来的是谁 /// 如果是 vector：线性从前往后排的，那么greater会先把大的留下，最后结果就是 从大到小排序 /// 如果是堆：从底部往上排序（或插入），所以greater，使得大的留在底部，所以也就是 小的在堆顶，所以是小顶堆 priority_queue<int, vector, greater> minHeap;

拷贝构造的两种形式

1、显示拷贝 Person p3(p1); // p1 是已经存在的对象
2、隐式拷贝 Person p3=p1;

但是注意：对于2，编译器有两种解释：
存在自定义拷贝构造

```
class Peo {
    public:
        Peo() {
            cout << "Moren" << endl;
        }

        Peo(Peo& p1) {
            cout << "Copy" << endl;
        }
        int a;
};
```

Peo p3 = p1; 编译器调用自定义拷贝构造p3: Peo pe(p1); 且会输出 Copy

默认拷贝构造

```
 class Peo {
    public:
        Peo() {
            cout << "Moren" << endl;
        }
};
```

Peo p3 = p1; 对于赋值操作，相当于调用默认的拷贝构造函数，赋值p1对象属性: p3=p1;

默认情况下，c++编译器至少给一个类添加3个函数

1．默认构造函数(无参，函数体为空)

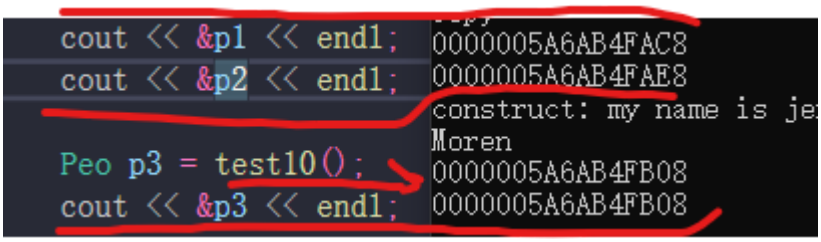
2．默认析构函数(无参，函数体为空)

3．默认拷贝构造函数，对属性进行值拷贝

构造函数调用规则如下：

如果用户定义有参构造函数，c++不在提供默认无参构造，但是会提供默认拷贝构造

如果用户定义拷贝构造函数，c++不会再提供其他构造函数



p2=p1; 是创建一个新的类对象，地址不一样 （但是注意，如果p1和p2中有指针，则指针本身p1和p2各自有一份不同的指针值，但是 指针指向的地方是一样的，所以会出现 p1释放了一块空间，那么p2的该指针就变成野指针了！！！） p3是接收函数的返回值（一个class类型）；是直接拷贝过来（相当于引用，p3是test10中class对象的别名）

类的大小

1.类的大小与什么有关系？ 与类大小有关的因素：普通成员变量，虚函数，继承（单一继承，多重继承，重复继承，虚拟继承） 与类大小无关的因素：静态成员变量，静态成员函数及普通成员函数 总结一下就是：对象自己独有的，则消耗自己内存；大家可共享的，就不占自己的内存！

成员变量的声明顺序不同，对于类自身的内存构造排版会有影响；将来自己声明和定义类的时候，要注意**内存地址问题**

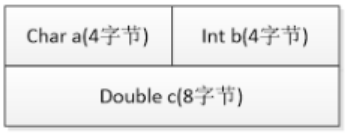
3.一般类的大小（注意内存对齐）

首先上两个类的示例：

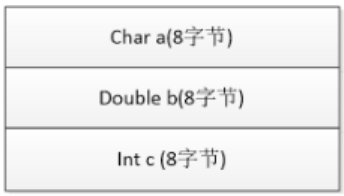
```
class base1
{
private:
    char a;
    int b;
    double c;
};
class base2
{
private:
    char a;
    double b;
    int c;
};
```

虽然上述两个类成员变量都是一个char，一个int，一个double，但是不同的声明顺序，会导致不同的内存构造模型，对于base1，base2，其成员排列是酱紫的：

base1：



base2：



base 1类对象的大小为16字节，而base 2类对象的大小为24字节，因为不同的声明顺序，居然造成了8字节的空间差距，因此，我们将来在自己声明类时，一定要注意到内存对齐问题，优化类的对象空间分布。

虚函数：需要 vfptr进行动态编译；随着对象不同，指向的虚函数本体不同；内存布局的最一开始是vfptr（virtual function ptr）即虚函数表指针（只要含虚函数，一定有虚函数表指针，而且该指针一定位于类内存模型最前端），接下来是Base类的成员变量，按照在类里的声明顺序排列，当然啦，还是要像上面一样注意内存对齐原则！ （单继承下的每个类对象**vfptr**只有一个 ---- 指向虚函数表；多继承下的类对象，有多个

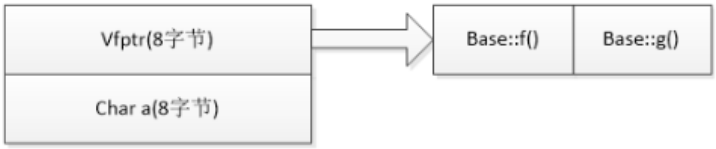
vfptr，指向每个基类下的虚函数表）

4.含虚函数的单一继承

首先呈上示意类：（64位，指针大小8字节）

```
class Base
{
private:
    char a;
public:
    virtual void f();
    virtual void g();
};
class Derived:public Base
{
private:
    int b;
public:
    void f();
};
class Derived1:public Base
{
private:
    double b;
public:
    void g();
    virtual void h();
};
```

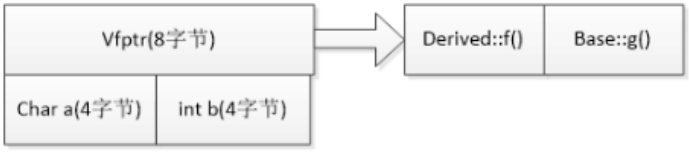
基类Base中含有一个char型成员变量，以及两个虚函数，此时Base类的内存布局如下：



内存布局的最一开始是vfptr（virtual function ptr）即虚函数表指针（只要含虚函数，一定有虚函数表指针，而且该指针一定位于类内存模型最前端），接下来是Base类的成员变量，按照在类里的声明顺序排列，当然啦，还是要像上面一样注意内存对齐原则！

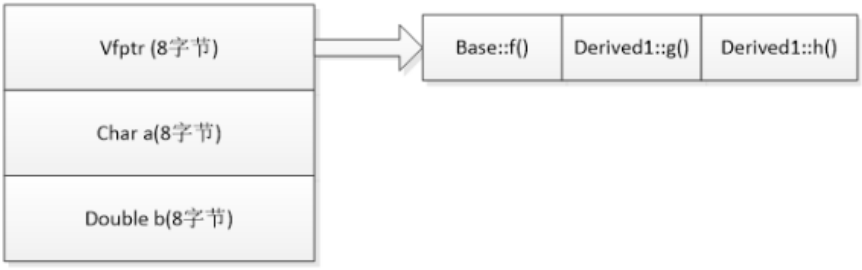
继承： 1、单继承

继承类Derived继承了基类，重写了Base中的虚函数f()，还添加了自己的成员变量，即int型的b，这时，Derived的类内存模型如下：



此种情况下，最一开始的还是虚函数表指针，只不过，在Derived类中被重写的虚函数f()在对应的虚函数表项的Base::f()已经被替换为Derived::f()，接下来是基类的成员变量char a，紧接着是继承类的成员变量int b，按照其基类变量声明顺序与继承类变量声明顺序进行排列，并注意内存对齐问题。

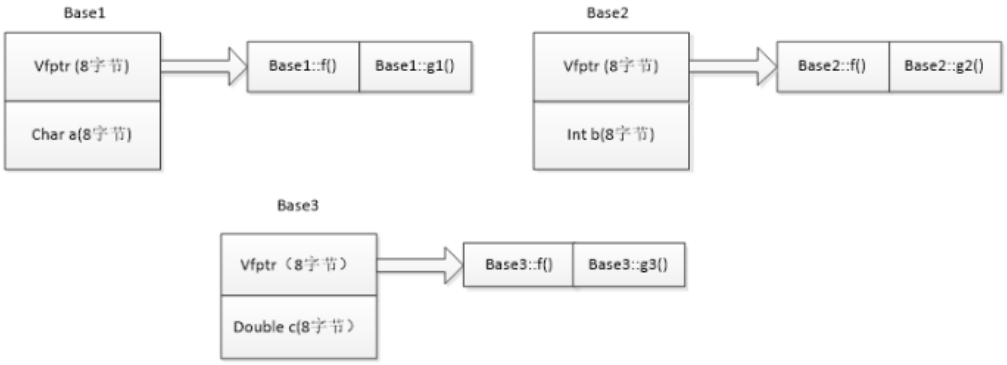
继承类Derived1继承了基类，重写了Base中的虚函数g()，还添加了自己的成员变量（即double型的b）与自己的虚函数（virtual h()），这时，Derived1的类内存模型如下：



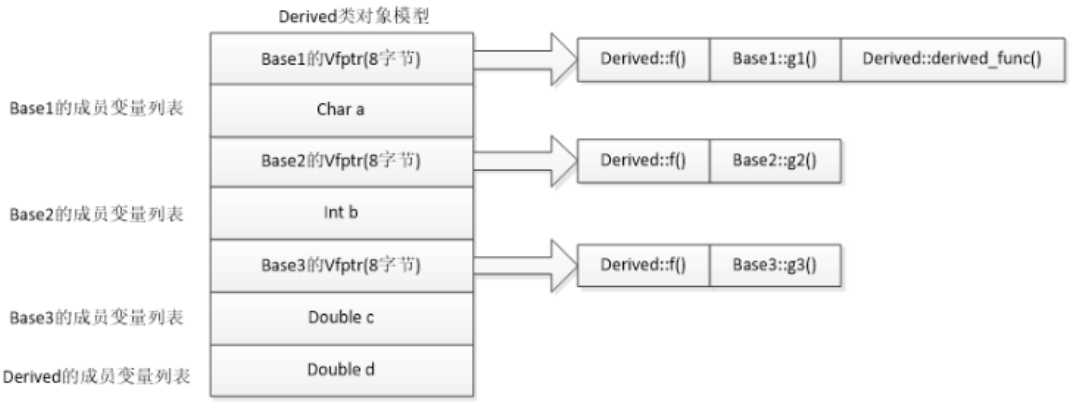
此种情况下，Derived1类一开始仍然是虚函数表指针，只是在Derived1类中被重写的虚函数g()在对应的虚函数表项的Base::g()已经被替换为Derived1::g()，新添加的虚函数virtual h()位于虚函数表项的后面，紧跟着基类中最后声明的虚函数表项后，接下来仍然是基类的成员变量，紧接着是继承类的成员变量。

2、多重继承（按继承顺序：先排基类1，再排基类2...）

首先继承类多重继承了三个基类，此外继承类重写了三个基类中都有的虚函数virtual f()，还添加了自己特有的虚函数derived_func()，那么，新的继承类内存布局究竟是什么样子的呢？请看下图！先来看3个基类的内存布局：



紧接着是继承类Derived的内存布局：



to虚函数表 Note：想想这里的底层操作会怎样的：？ --如果子类没有重写父类的虚函数，则是不是，直接读父类的虚函数表就可以了 --如果子类重写了父类的虚函数，则拷贝父类的虚函数表，并进行相应修改 (如果不重新拷贝的话：是否和解决hash冲突那样，“挂东西？”（一个格子里面的value是Listnode* head，然后Listnode节点上存储的 value=自己对应的虚函数地址））

但是，虚拟继承下继承类$\color{red}{自己定义了新的虚函数}$，则该虚函数表的指针情况还会和编译器有关

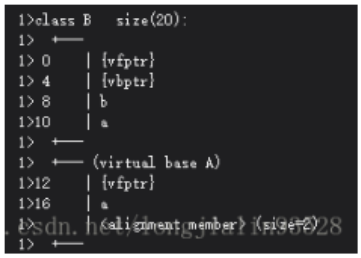
在vs环境下，采用虚拟继承的继承类会有自己的虚函数表指针（假如基类有虚函数，并且继承类添加了自己新的虚函数）

在gcc环境下及mac下使用clion，采用虚拟继承的继承类没有自己的虚函数表指针（假如基类有虚函数，无论添加自己新的虚函数与否），而是共用父类的虚函数表指针

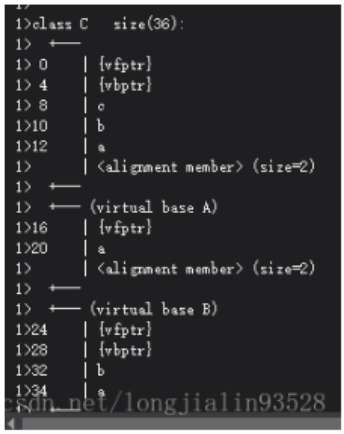
在*VS2010*中，在项目——属性——配置属性——*C/C++*——命令行——其他选项中添加选项“*/d1reportAllClassLayout*”。再次编译时候，编译器会输出所有定义类的对象模型。由于输出的信息过多，我们可以使用“*Ctrl+f*”查找命令，找到对象模型的输出。

接下来看到了class B 与class A的内存模型

class B内存模型：



class C内存模型：



首先认识两种指针，vfptr（虚函数指针），vbptr（虚基类指针），因为虚拟继承的子类通常只存一份父类，因此，需要一个偏移量去找父类，这时vbptr就显示出了其用途

vs下：

sizeof（B）=自己的vfptr（8）+自己的vbptr（8）+自己的两个char[2]（8，内存对齐)+父类的vfptr（8）+父亲的char[2](8,这是因为内存对齐的原因)=40

sizeof（C）=自己的vfptr（8）+自己的vbptr（8，用来寻找虚拟继承的父类B）+3*char[2](8,内存对齐)

+sizeof（B）=64

GCC共享虚函数表指针（无论虚拟继承的子类是否添加了自己的虚函数），也就是说父类如果已经有虚函数表指针，那么子类中共享父类的虚函数表指针空间，不再占用额外的空间，VC在虚继承情况下

（1）没有添加自己的新虚函数，则共享父类虚函数表指针

（2）添加自己的新虚函数，则不共享父类虚函数表指针，自己拥有自己新的虚函数表指针

那也就是说，我们在64位系统，gcc编译下，B的大小应该是

sizeof（B）=自己的vbptr（8）+自己的两个char[2]（8，内存对齐)+父类的vfptr（8）+父亲的char[2](8,这是因为内存对齐的原因)=32

有了虚基类指针就可以找到父类，自然能找到父类的虚函数表指针，使用虚函数

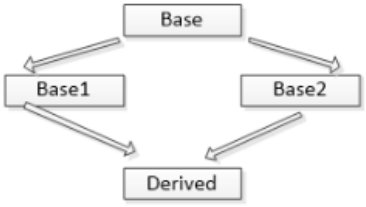
sizeof（C）=自己的vbptr（8，用来寻找虚拟继承的父类B）+3*char[2](8,内存对齐)+sizeof（B）=48

虚继承：vbptr

（2）虚拟继承会给继承类添加一个虚基类指针（virtual base ptr 简称vbptr），其位于类虚函数指针后面，成员变量前面，若基类没有虚函数，则vbptr其位于继承类的最前端

关于虚拟继承，首先我们看看为什么需要虚拟继承及虚极继承解决的问题。

虚极继承主要是为了解决**菱形继承下公共基类的多份拷贝问题：**



二义性问题：

```
Derived de;
de.a=10;//这里是错误的，因为不知道操作的是哪个a
```

重复继承下空间浪费：

Derived重复继承了两次Base中的int a，造成了无端的空间浪费

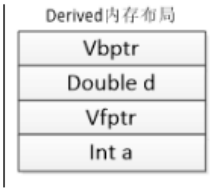
虚拟继承是怎么解决上述问题的?

虚基继承可以使得上述菱形继承情况下最终的Derived类只含有一个Base类，Base类在虚拟继承后，位于继承类内存布局最后面的位置，继承类通过vbptr寻找基类中的成员及vfptr。

虚拟继承对继承类的内存布局影响可以先看以下示例代码，理解以后，我们在最后列出上述菱形虚拟继承情况下Base1，Base2与Derived代码及内存布局，看到虚拟继承起的作用。

```
class base
{
public:
    int a
    virtual void f();
}
class derived:virtual public base
{
public:
    double d;
    void f();
}
```

Derived类内存布局如下图，由于虚拟继承，Derived只会有一个最初基类的拷贝，该拷贝位于类对象模型的最下面，而想要访问到基类的元素，需要vbptr指明基类的位置（vbptr作用），假如Base中含有虚函数，而继承类中没有增添自己的新的虚函数，那么Derived类统一的布局如下：



Derived类内存布局如下图，由于虚拟继承，Derived只会有一个最初基类的拷贝，该拷贝位于类对象模型的最下面，而想要访问到基类的元素，需要vbptr指明基类的位置（vbptr作用），假如Base中含有虚函数，而继承类中没有增添自己的新的虚函数，那么Derived类统一的布局如下：

如果添加了自己的新的虚函数(代码如下)：

```
class base
{
public:
    int a
    virtual void f();
}
class derived:virtual public base
{
public:
    double d;
    void f();
    virtual void g();//这是Derived类自己新添加的虚函数
}
```

那么Derived在VC下继承类会有自己的虚函数指针，而在Gcc下是共用基类的虚函数指针，其分布如下

多重继承且含虚继承

现在有了上述代码的理解我们可以写出菱形虚拟继承代码及每个类的内存布局：

```
class Base
{
public:
    int a;}
class Base1:public virtual Base
{
}
class Base2:public virtual Base
{
}
class Derived:public Base1,public Base2
{
private:
    double b;
public:
}
```

带实线的框是类确确实实有的，带虚线是针对Base，及Base1，Base2做了扩展后的情况：

Base有虚函数，Base1还添加了自己新的虚函数，Base1也有自己成员变量，Base2添加了自己新的虚函数，Base2也有自己成员变量，则上图全部虚线中的部分都将存在于对象内存布局中。

几点解释： 1、Base1 和 Base2是虚拟继承Base的，所以Base成员（包括Base的vfptr）在Base1和Base2的内存中是在最下面的；（通过Vbptr调用到Base成员） 2、Derived继承Base1和Base2是普通的多重继承，所以按照多重继承的布局方式进行布局 ---对于Base1和Base2来说：其成员有 vfptr其自己的虚函数表指针；vbptr指向Base；Basei各自的成员变量；最下面的Base的成员：Vfptr和int a（而对于 最下面的Base的成员：Vfptr和int a，编译器往下继承时，复制一份：也就是Derived布局中的最下方（上图最右的【框4】）） ---按继承顺序放置**Base1和Base2**成员（所以可得上图最右的Derived类的内存布局：【框1】Base1；【框2】Base2；【框3】Derived成员变量；【框4】Base）

总结下：

在C++中，当子类和父类各自都有虚函数时，子类会有多个虚函数表（vtable），而每个虚函数表通常由一个虚函数表指针（vfptr）指向。因此，子类会有多个vfptr指针，每个指针指向不同的虚函数表。

具体来说：

1. **单继承的情况：**
   - 如果子类继承了父类，并且父类和子类都有虚函数，子类一般只有一个vfptr指针。这个指针指向的vtable中包含了父类的虚函数和子类的虚函数。
2. **多重继承的情况：**
   - 如果子类从多个基类继承，而每个基类都有虚函数，子类会为每个有虚函数的基类单独生成一个vfptr指针。所以子类中可能有多个vfptr指针，分别对应不同的基类的vtable。
3. **虚继承的情况：**
   - 如果子类涉及虚继承，情况会更加复杂，vfptr的数量可能会增加，以便正确处理虚拟基类中的虚函数。

总结来说，子类中的vfptr指针的数量取决于继承结构和每个基类中是否存在虚函数。在简单的单继承情况下，通常只有一个vfptr指针；在多继承或虚继承的情况下，vfptr指针的数量可能会增加。

多重继承且存在虚继承

内存布局解释

plaintext 复制代码

```
+-------------------------------------+
| Base1::vfptr                        | // 指向 Base1 的虚函数表
+-------------------------------------+
| Base1::vbptr                        | // 指向 Base 在 Derived 中的位置
+-------------------------------------+
| Base2::vfptr                        | // 指向 Base2 的虚函数表
+-------------------------------------+
| Base2::vbptr                        | // 指向 Base 在 Derived 中的位置
+-------------------------------------+
| Derived::vfptr                      | // 指向 Derived 的虚函数表
+-------------------------------------+
| Base::int a                         | // Base 的成员（由 vbptr 指向）
+-------------------------------------+
| double b                            | // Derived 的成员
+-------------------------------------+
```

在这个内存布局中，`vfptr`指针通常位于`vbptr`指针之前。这种安排可能是因为`vfptr`的使用频率更高，而且放在前面可以更快地访问虚函数表。总体上，这个布局有效地解决了多重继承和虚继承中的内存管理问题。

在C++的内存布局中，`vfptr`（虚函数表指针）和`vbptr`（虚基类指针）的位置是由编译器决定的，但通常情况下，编译器会遵循一些惯例。

常见的内存布局顺序

- `vfptr`通常在前：大多数情况下，编译器会将`vfptr`放在类对象布局的开头，因为`vfptr`通常是用来处理类的多态行为的，它是虚函数调用机制的核心。将`vfptr`放在对象的开头，可以使虚函数的调用更加高效。
- `vbptr`通常在`vfptr`之后：`vbptr`指向虚基类的偏移量表，用于在虚继承时找到虚基类的成员。由于`vbptr`的使用频率通常不如`vfptr`高，编译器可能将它放在`vfptr`之后。

抽象类

不可以创建对象，不可以用作参数，不可以做返回值

但是可以作为指针；可以当作基类被继承 可以进行多态使用！！

A* a；　(A为抽象类，但是可以有 A*，这时还未初始化呢)

类的成员对象 类的成员调用

```cpp
class Test2 : public Test1<int> {
public:
    void Printf() {
        for (int i = 0; i < 1000; i++) {
            cout << "The Message From P" << endl;
        }
    }

    void Printf1() {
        for (int i = 0; i < 1000; i++) {
            cout << "The Message From P1" << endl;
        }
    }
};

int main() {
    Test2 ttt;

    thread t1(&Test2::Printf, &ttt);
    thread t2(&Test2::Printf1, &ttt);
```

创建线程时，需要绑定的是 函数对象的地址，是把函数作为一个 对象去处理的！！

函数名+() 属于函数调用，这里就报错了

```cpp
thread t1(&Test2::Printf(), &ttt);
thread t2(&Test2::Printf1(), &ttt);
```

而且需要的是 函数对象，所以这也是为啥 绑定 类的成员函数时，需要传入一个类对象的原因：得先有一个类对象，才能有成员函数对象

如果像 Printfout函数，本身就是一个独立的函数对象，所以直接绑定即可 并且，函数名就是函数地址

```
void Printfout() {
    for (int i = 0; i < 1000; i++) {
        cout << "The Message From Pout" << endl;
    }
}


int main()
{
    shared_ptr<Test> sp1(new Test);
    cout << "use_count: " << sp1.use_count() << endl;
    shared_ptr<Test> sp2 = sp1->getSharedPtr();
    cout << "use_count: " << sp1.use_count() << endl;

    Test2 ttt;
    ttt.Printf1();

    thread t1(&Test2::Printf,&ttt);
    thread t2(&Test2::Printf1,&ttt);
    thread t3(Printfout);
```