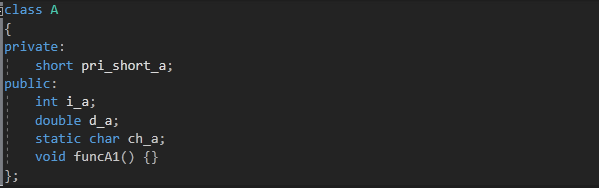
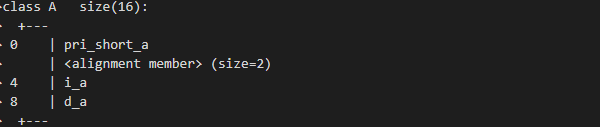
**C++类的布局（继承、多态）**

## A．无继承

### A.1. 无虚函数



A的大小及布局如下：

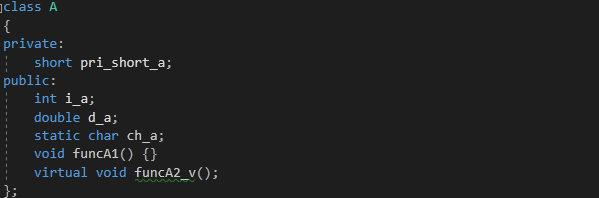


如上可以说明：

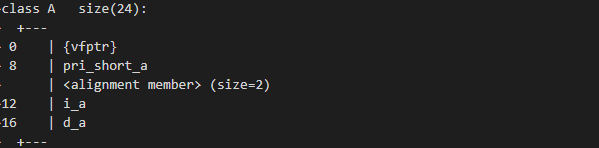
1. 静态数据成员虽然属于类，但不占用具体类对象的内存。
2. 成员函数不占用具体类对象内存空间，成员函数存在代码区。
3. 数据成员的访问级别并不影响其在内存的排布和大小，均是按照声明的顺序在内存中有序排布，并适当对齐。

### A.2. 有虚函数

在1.1中的类A里增加一个虚函数：



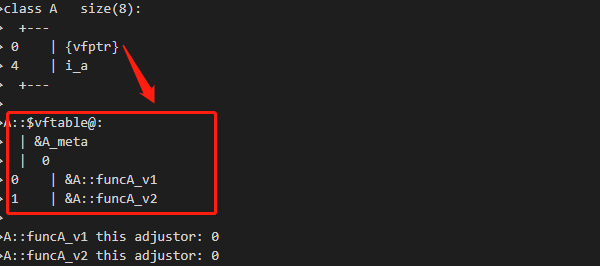
其内存大小及布局如下：



可以看到，A的起始处存储的是虚指针vptr，指针大小是4字节，这里是为了对齐8字节。为方便观察，之后的讨论中，我们统一把数据成员都改为int类型，占4字节。

现在我们再加一个虚函数funcA\_v2()：



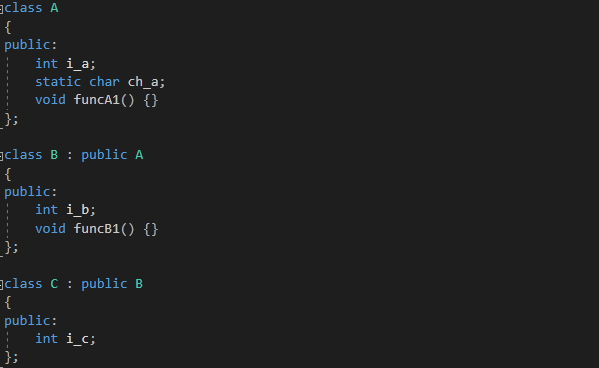


所以，不论再多虚函数，都只会有一个虚指针vptr，不会改变类的大小。不同之处在于，虚指针所指向的虚表中会多一个项目，即指向另一个虚函数的地址。

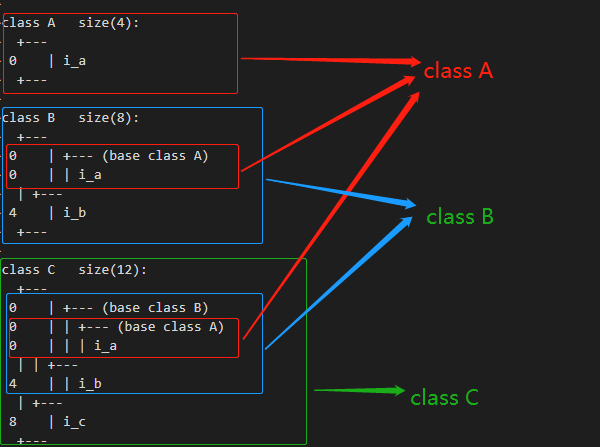
## B. 单一继承

### B.1. 单一继承且无虚函数

如下，我们设计了类A、B和C，其中，B继承自A，C继承自B：



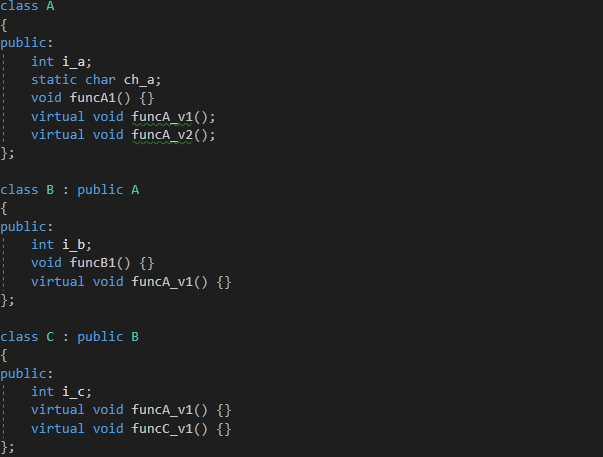
内存布局如下：



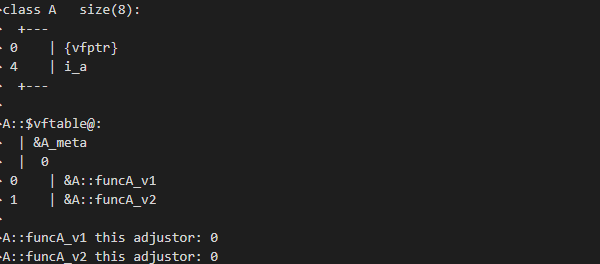
单一继承的内存布局很清晰，每个派生类中起始位置都是Base class subobject。现在我们在类中增加虚函数，观察在单一继承+有虚函数的情况下，类的内存布局。

### B.2. 单一继承且有虚函数

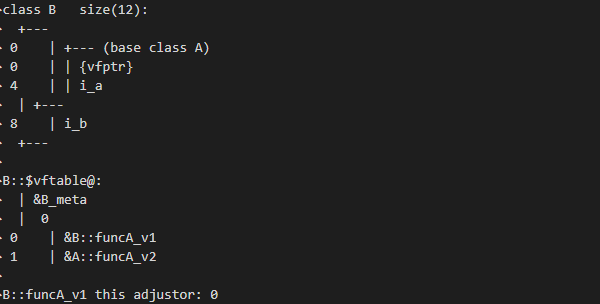
如下，类A增加了两个虚函数funcA\_v1()和funcA\_v2()；类B继承自A，覆写funcA\_v1()；类C继承自B，重写funcA\_v1()，且有自己定义的一个虚函数funcC\_v1()：



Class A的内存布局如下，如同A.2，这里不再解释：

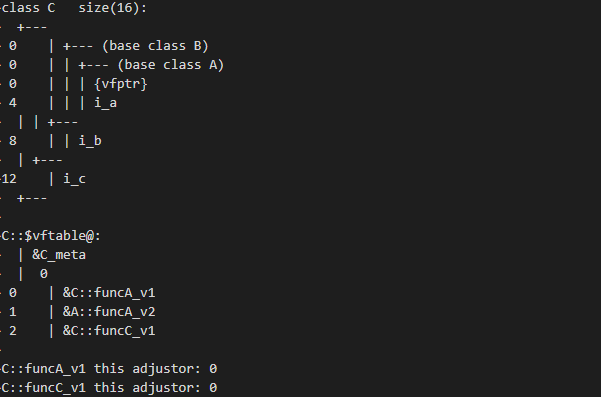


Class B的内存布局如下：

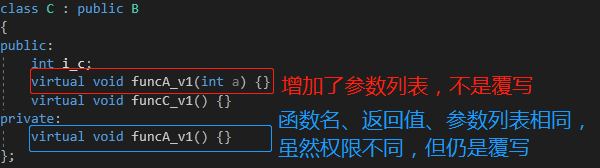


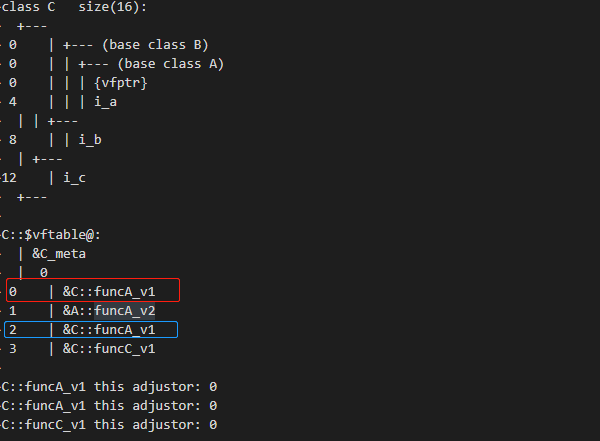
B中首先也是基类A subobject，同样含有一个虚指针vptr。由于B覆写了funcA\_v1()，故虚表中第一个索引处的函数地址是&B::funcA\_v1()。

理解了B的内存布局，接下来C的内存布局也就不必赘述：



必须要提及两点：**虚析构函数**和**覆写**。虚析构函数在B.3.中详述。**怎么才算是覆写？——类的继承里，子类里含有与父类里同名的虚函数，函数名、函数返回值类型和参数列表必须相同，权限可以不同**。如上面示例中，B和C都覆写了A的funcA\_v1()。下面的例子说明了这一点：





### B.3. 虚析构函数

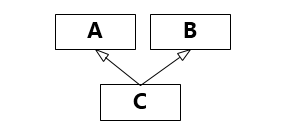
《Effective C++》第三版，Item 07：为多态基类声明virtual析构函数。

当一个派生类对象经由一个base class指针被删除，而该base class带着一个non-virtual析构函数，其结果未有定义——实际执行时通常发生的是对象的derived成分没被销毁。所以上述的类设计其实有错误，带多态性质的基类应该声明一个virtual析构函数。如果class带有任何virtual函数，它就应该拥有一个virtual析构函数。

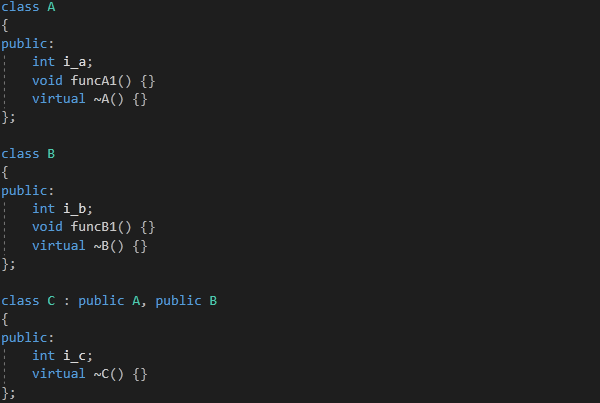
在接下来的示例中，我们将加上虚析构函数。

## C. 多重继承

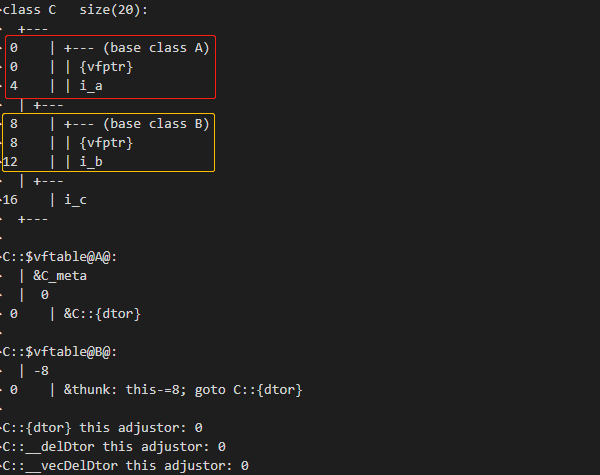
### C.1. 多重继承



如下是一个简单的继承关系，class C同时继承自A和B：



类A和B的内存布局如同A.2。而类C的内存布局如下：



可见，派生类C中依其继承的基类的顺序，存放了各个基类subobject及各自的vptr，然后才是Class C自己的数据成员。需要解释上图中的thunk：

**Thunk解释**：所谓thunk是一小段assembly代码，用来(1)以适当的offset值调整this指针，(2)跳到virtual function去。例如，经由一个Base2指针调用Derived destructor，其相关的thunk可能看起来是下面这个样子：

//虚拟C++代码

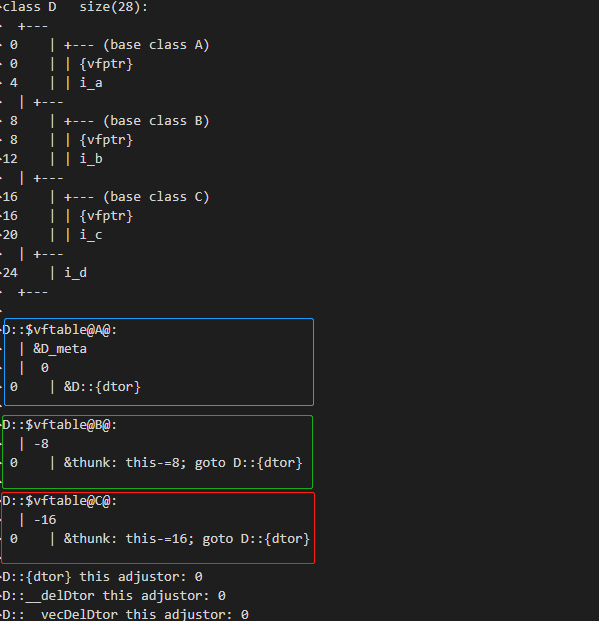
pbase2\_dtor\_thunk:

this += sizeof( base1 );

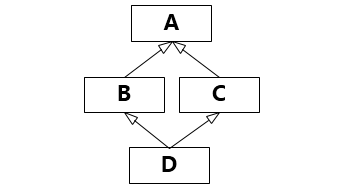
Derived::~Derived( this );

根据上面的解释，经由class A的指针调用C的析构函数，其offset等于0；而经由class B调用C的析构函数，其offset等于8，如同上图所示：this-=8。

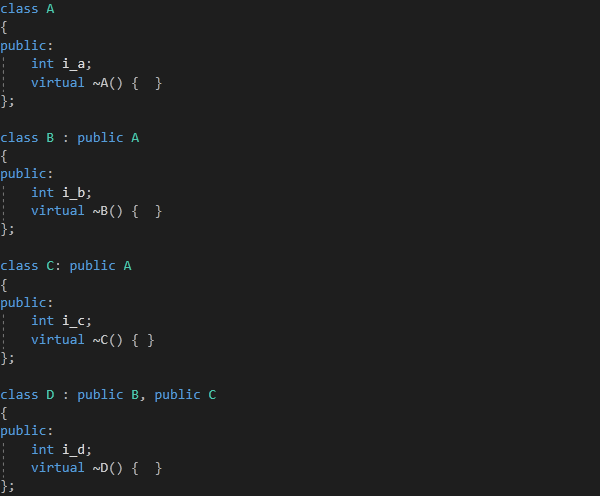
同时也可以想到，随着base class的数量增多，派生类里也会首先顺序存放各个基类subobject。而派生类中也会记录其到各个base subobject的offset。如下图是类D同时继承类A、B、C：



### C.2. 菱形继承



如上图是一个菱形继承的示意图，类B和C均继承自类A，类D同时继承类B和C，代码如下：



类A的内存布局很简单，如A.2。类B和C的内存布局如B.2。接下来看类D的内存布局：

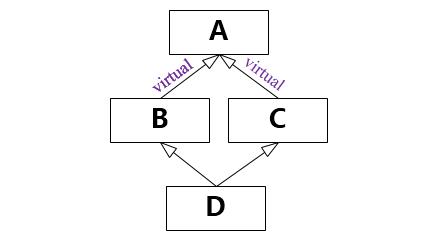


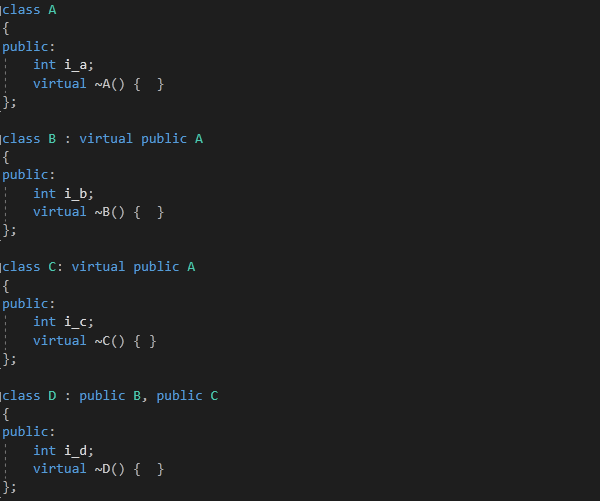
如上图，D中依次存放基类B subobject和基类C subobject。其中B和C中均存放一份class A subobject。

### C.3. 虚拟继承

从菱形继承的most-derived class（即C.2.中的class D）的内存布局可以看出，subobject A有两份，所以A的data member也存了两份，但实际上对于D而言，只需要有一份subobject A即够了。菱形继承不仅浪费存储空间，而且造成了数据访问的二义性。虚拟继承可以很好地解决这个问题。

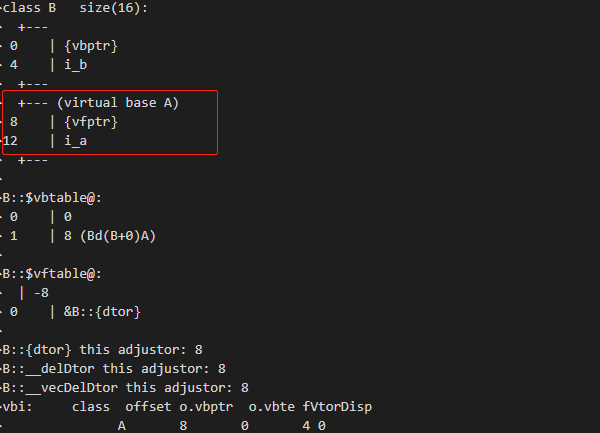
同样以C.2.中的继承关系为例，不过这次我们B和C对A的继承都加上了关键字**virtual**。





接下来看看各个类的内存布局。

A的内存布局同A.2。类B和C的内存布局如B.2？是吗？不是！如下图：



可以看到，class B中有两个虚指针：第一个指向B自己的虚表，第二个指向虚基类A的虚表。而且，从布局上看，class B的部分要放在前面，虚基类A的部分放在后面。在class B中虚基类A的成分相对内存起始处的偏移offset等于class B的大小（8字节）。C的内存布局和B类似。

这个布局与之前的不一样：为什么基类subobject反而放到后面了？——Class如果内含一个或多个virtual base subobjects，将被分割成两部分：一个不变区域和一个共享区域。不变区域中的数据，不管后继如何衍化，总有固定的offset（从object的开头算起），所以这一部分可以直接存取。而共享区域所表现的就是virtual base class subobject。这部分数据的位置会因为每次的派生操作而发生变化，所以它们只可以被间接存取。

接下来看class D的内存布局：直接的基类B和C按照声明的继承顺序，在D的内存中顺序安放。紧接着是D的data member。然后是共享区域virtual base class A。

