**深度探索C++对象模型**

# 第一章 关于对象

## 1. 使用class封装之后的布局成本

class并没有增加成本，data members直接内含在每一个class object之中，就像C struct一样。而member functions虽然被包含在class的声明之内，但是不出现在Object之中。每一个non-inline function只会产生一个函数实体。至于inline function则会在每一个调用使用的地方产生一个函数实体（在调用点展开函数体）。

## 2. C++静态成员函数、成员函数的存储位置

C++程序的内存格局通常分为四个区：全局数据区(data area)，代码区(code area)，栈区(stack area)，堆区(heap area)(即自由存储区)。全局数据区存放全局变量，静态数据和常量；所有类成员函数和非成员函数代码存放在代码区；为运行函数而分配的局部变量、函数参数、返回数据、返回地址等存放在栈区；余下的空间都被称为堆区。根据这个解释，我们可以得知在类的定义时，类成员函数是被放在代码区，而类的静态成员变量在类定义时就已经在全局数据区分配了内存，因而它是属于类的。对于非静态成员变量，我们是在类的实例化过程中(构造对象)才在栈区或者堆区为其分配内存，是为每个对象生成一个拷贝，所以它是属于对象的。

      类的静态成员函数和非静态成员函数的区别：静态成员函数和非静态成员函数都是在类的定义时放在内存的代码区的，因而可以说它们都是属于类的，但是类为什么只能直接调用静态类成员函数，而非静态类成员函数(即使函数没有参数)只有类对象才能调用呢？原因是类的非静态类成员函数其实都内含了一个指向类对象的指针型参数(即this指针)，因而只有类对象才能调用(此时this指针有实值)。不同的对象使用的是同一个函数代码段，它怎么能够分别对不同对象中的数据进行操作呢？原来C++为此专门设立了一个名为this的指针，用来指向不同的对象。

### 内联函数

 需要说明，不论成员函数在类内定义还是在类外定义，成员函数的代码段都用同一种方式存储。不要将成员函数的这种存储方式和inline(内联)函数的概念混淆。不要误以为用inline声明(或默认为inline)的成员函数，其代码段占用对象的存储空间，而不用inline声明的成员函数，其代码段不占用对象的存储空间。**不论是否用inline声明(或默认为inline)，成员函数的代码段都不占用对象的存储空间。**用inline声明的作用是在调用该函数时，将函数的代码段复制插入到函数调用点，而若不用inline声明，在调用该函数时，流程转去函数代码段的入口地址，在执行完该函数代码段后，流程返回函数调用点。inline与成员函数是否占用对象的存储空间无关，它们不属于同一个问題，不应搞混。

## 3. class在布局以及存取时间上主要的额外负担是由virtual引起

(1). virtual function 机制 用以支持一个有效率的“执行期绑定（runtime binding)”。

(2). virtual base class用以实现“多次出现在继承体系中的 base class，有一个单一的被共享的实体”。

(3). 当然还有一些多重继承下的额外负担，发生在“一个derived class和其第二或后继之 base class的转换”之间。

## 4. C++对象模型

在C++对象模型中，nonstatic data members 被放置在每一个class object之内，static data members则被存放在class object之外。static和nonstaitc function也被放在所有 class object 之外。virtual functions 则以两个步骤支持之：

1. 每一个class产生出一堆指向virtual functions的指针，放在表格之中，这个表格被称为virtual table(vtbl)；
2. 每一个class object被添加一个指针，指向相关的virtual table。通常这个指针被称为 vptr，vptr的设定和重置都由每一个class的constructor、destructor和copy assignment 运算符自动完成。

## 5. C++以下列方法支持多态

1. 经由一组隐含的转化操作。例如把一个derived class指针转换为一个指向其public base type的指针。
2. 经由virtual function机制。
3. 经由dynamic\_cast和typied运算符。

## 5. class object需要多少内存

(1). 其nonstatic data members的总和大小。

(2). 加上任何由于alignment的需求和padding（填补）上去的空间。

(3). 加上为了支持virtual而由内部产生的任何额外负担。

## 6. 重新认识指针类型

指针类型：告诉编译器如何解释某个特定地址中的内存内容及其大小（例如：一个string是传统的8 bytes（包括一个4byte的字符指针和一个用来表示字符串长度的整数）。转型（cast)其实是一个编译指令，大部分不会改变一个指针所含有的真正地址，它只影响“被指出值内存的大小和其内容”的解释方式。

# 第二章 构造函数语意学

## 2.1. Default Constructor

当编译器需要的时候，default constructor会被合成出来，只执行编译器所需要的任务（将members适当初始化）。

### 2.1.1. 带有Default Constructor的Member Class Object

如果一个class没有任何构造函数，但它内含一个member object，而后者有默认构造函数，那么这个class的隐式默认构造函数是nontrivial。编译器需要为该class合成默认构造函数。不过这个合成操作只有在constructor真正被调用的时候才会发生。

注意：编译器合成的默认构造函数只进行必要的操作，如调用member object的默认构造函数，但对于该class内的一些程序员自己定义的数据成员，如整数、整数指针、数组等，其初始化工作应该由程序员完成。

如果程序员已经为class定义了一个默认构造函数，编译器将不再合成默认构造函数。但该class的member object仍旧需要被初始化。编译器的处理是：如果一个class A内含一个或者一个以上member class objects，那么class A的每一个constructor必须调用每一个member classes的default constructor。编译器会扩张已存在的constructors，在其中安插一些代码，使得user code在被执行之前，先调用（调用顺序与member objects在class的声明次序一致）必要的default constructors。

### 2.1.2. 带有Default Constructor的Base class

如果一个没有任何构造函数的类派生自一个带有缺省构造函数的基类，那么这个派生类的缺省构造函数被视为nontrivial，并因此需要被合成出来。它将调用上一层base classes的default constructor(根据它们的声明次序)。对一个后继派生的class而言，这个合成的constructor和一个“被明确提供的default construnctor”没什么差异。

如果设计者提供多个构造函数，但是其中却没有缺省构造函数呢？编译器会扩张现有的每一个构造函数，将“用以调用所有必要之default constructors”的程序代码加进去。它不会合成一个新的缺省构造函数。如果同时亦存在着带有缺省构造函数的member class objects，那些缺省构造函数也会被调用—在所有基类构造函数都被调用之后。

### 2.1.3. 带有一个Virtual Function的Class

下面两种情况同样需要合成default constructor：

1. class声明（或继承）一个virtual function。
2. class派生自一个继承串链，其中一个或者更多的virtual base class。

下面两个扩张操作会在编译期间发生：

1. 一个virtual function table会被编译器产生出来，内放class的virtual functions的地址。
2. 在每一个class object中，一个额外的pointer member(vptr)会被编译器合成出来，内含相关的class vtbl的地址。

### 2.1.4. 带有一个Virtual Base Class的class

Virtual base class的实现法在不同编译器之间有很大差异，然而，每一个实现的共同点在于必须使virtual base class在其每一个derived class object中的位置，能够在执行期准备妥当。对于class所定义的每一个constructor编译器都会安插那些“允许每一个virtual base class 的执行期存取操作”的码。

### 总结

以上四种情况，会导致“编译器必须为未声明constructor的class合成一个default constructor”，这只是编译器（而非程序）的需要。它之所以能够完成任务，是借着“调用member object或base class的default constructor”或是“为每一个object初始化其virtual function机制或virtual base class机制”完成。至于没有存在这四种情况而又没有生命constructor的class实际上是不会被合成出来的。

在合成的default constructor中，只有base class subobjects(子对象）和member class objects会被初始化。所有其他的nonstatic data member ，如整数，整数指针，整数数组等是不会被初始化的，这些初始化操作对程序是必须的，但对编译器则并非需要的。

注意：C++新手一般有两个误解

1.任何class如果没有定义default constructor，就会被合成出来一个。

2.编译器合成出来的default constructor会明确设定class内每一个data member的默认值。

## 2.2. Copy Constructor

有三种情况，会以一个object的内容作为另一class object的初值：

* 1. 对一个object做明确的初始化操作。
  2. 当object被当做参数交给某个函数。
  3. 当函数返回一个class object。

这三种情况需要有copy constructor。

### 2.2.1. Default Memberwise Initialization

如果class没有提供一个explicit copy constructor时，当class object以“相同的另一个object作为初值时，其内部是以所谓的default memberwise initialization方式完成的。也就是把每一个内建的或派生的data member（例如一个数组或指针）的值，从某个object拷贝一份到另一个object上，但不拷贝其具体内容。例如只拷贝指针地址，不拷贝一份新的指针指向的对象，这也就是浅拷贝，不过它并不会拷贝其中member class object，而是以递归的方式实行memberwise initialization。

**Bitwise Copy Semantics(位逐次拷贝)**

这种递归的memberwise initialization是如何实现的呢？

答案就是Bitwise Copy Semantics和default copy constructor。如果class展现了Bitwise Copy Semantics，则使用bitwise copy（bitwise copy semantics编译器生成的伪代码是memcpy函数），否则编译器会生成default copy constructor。

**不要Bitwise Copy Semantics！**

那什么情况下class不展现Bitwise Copy Semantics呢？有四种情况：

1. 当class内含有一个member class object，而这个member class内有一个默认的copy构造函数[不论是class设计者明确声明，或者被编译器合成]；
2. 当class继承自一个base class，而base class有copy构造函数[不论是class设计者明确声明，或者被编译器合成]；
3. 当一个类声明了一个或多个virtual函数；
4. 当class派生自一个继承串链，其中一个或者多个virtual base class。

在前2种情况下，编译器必须将member或者base class的“copy constructor的调用操作”安插到被合成的copy constructor中。

### 2.2.2. 重新设定Virtual Table 的指针

第3种情况下，因为class包含virtual function，编译时需要做扩张操作：

1. 增加virtual function table，存放有作用的virtual function的地址；
2. 创建一个指向virtual function table的指针，安插在class object内。

所以，编译器对于每一个新产生的class object的vptr都必须被正确地赋值，否则将跑去执行其他对象的function。因此，编译器导入一个vptr到class之中时，该class 就不在展现bitwise semantics，必须合成Copy Constructor并将vptr适当地初始化。

比如下面的例子：首先，定义两个类，ZooAnimal和Bear：



ZooAnimal class object以另一个ZooAnimal class object作为初值，或Bear class object以另一个Bear class object作为初值，都可以直接靠“bitwise copy semantics”完成(除了可能会有的pointer member之外，为了简化，这里不考虑这种情况)。

当一个基类对象以其派生类的对象做初始化操作时，其vptr复制操作也必须保证安全，例如：



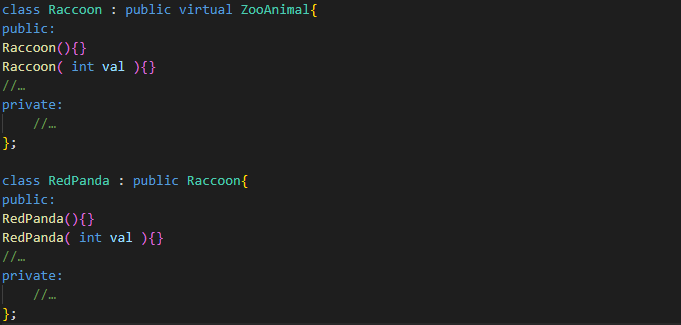
franny的vptr不可以被设定为指向Bear class的virtual table。合成出来的ZooAnimal copy constructor会明确设定object的vptr指向ZooAnimal class的virtual table，而不是从右手边的class object中将其vptr现值拷贝过来。

### 2.2.3. 处理Virtual Base Class Subobject

virtual base class的存在需要特别处理。一个class object 如果以另一个virtual base class subobject作为初值，而后者有一个virtual base class subobject，那么也会使“bitwise copy semantics”失效。

每一个编译器对于虚拟继承的支持承诺，都是表示必须让“derived class object中的virtual base class subobject位置”在执行期就准备妥当，维护“位置的完整性”是编译器的责任。Bitwise copy semantics可能会破坏这个位置，所以编译器必须自己合成出copy constructor。这也就是说，拷贝构造函数和默认构造器一样，需要的时候会进行构建，而并非程序员不写编译器就帮着构建。

例如：ZooAnimal成为Raccoon的一个虚拟基类，同时RedPanda public继承自Raccoon：



如果以一个Raccoon object作为另一个Raccoon object的初值，那么“bitwise copy”就已经足够：



然而如果企图以一个RedPanda object作为little\_critter的初值，编译器必须判断“后续当程序员企图存取其ZooAnimal subobject时是否能够正确地执行”：



在这种情况下，为了完成正确的little\_critter初值设定，编译器必须合成一个拷贝构造函数，安插一些代码以设定virtual base class pointer/offset的初值，对每一个members执行必要的memberwise初始化操作，以及执行其它的内存相关操作。

## 2.4 初始化列表

成员初始化列表跟在构造函数的原型后，以冒号开头。成员名是被指定的，后面是括在括号中的初始值，类似于函数调用的语法。如果成员是类对象，则初始值变成被传递给适当的构造函数的实参，该构造函数然后被应用在成员类对象上。

使用初始化列表和在构造函数内使用数据成员的赋值之间有什么区别？

两种实现的最终结果是一样的。在两个构造函数调用的结束处，三个成员都含有相同的值，区别是成员初始化表只提供该类数据成员的初始化。在构造函数体内对数据成员设置值是一个**赋值**操作。区别的重要性取决于数据成员的类型。

下面四种情况必须使用初始化列表来初始化class的成员：

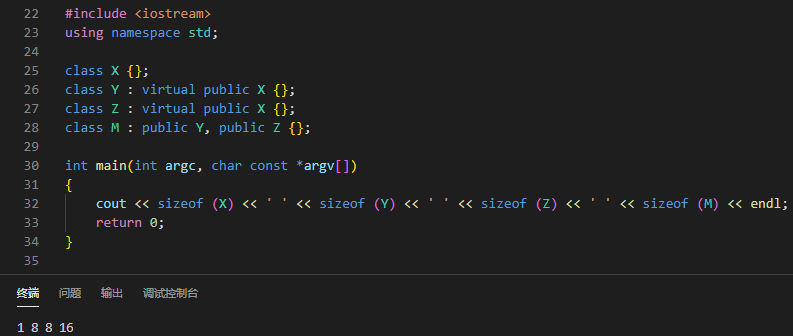
1. 当初始化一个reference member时；
2. 当初始化一个const member时；
3. 当调用一个base class的constructor，而它拥有一组参数（其实就是自定义的构造函数）时；
4. 当调用一个member class的constructor，而它拥有一组参数时。

不过，初始化的顺序是class members声明次序决定的，不是由初始化列表决定的。

# 第三章 Data 语意学

## 3.0. 前言

在如下代码中，在 64 位机器上的结果是：



X大小为1的原因是，编译器在class里面安插进一个char，这使得这一class的两个objects得以在内存中配置独一无二的地址。（另外一个可能原因是，对于一个struct，模数最小是1，不可能是0）

Y，Z大小受到三个因素的影响：

1. 语言本身所造成的额外负担，比如支持virtual。
2. 编译器对于特殊情况所提供的优化处理Virtual base class X subobject的1bytes大小也出现在class Y和Z身上。传统上它被放在devided class的固定部分的尾端。
3. Alignment的限制。

Empty virtual base class提供一个virtual interface，没有定义任何数据。某些新近的编译器(比如我现在用的 GCC4.8.1 MAC)提供特殊的处理。在这个策略下，一个empty virtual base class被视为devided class object最开头部分，也就是说它没有花费任何额外的空间。也就节省了上面的2中的1 byte。

一个virtual base class只会在devided class中存在一份实例，不管它在class继承体系中共出现多少次.

在这里class M的大小是16，它的大小由以下几点决定：

1. 被大家共享的唯一一个class X实例，大小为1 byte
2. Base class Y的大小，减去“因virtual base class X而配置”的大小，结果是8 bytes，class Y一样
3. class M自己的大小：0 byte
4. class M的alignment数量

这里加起来应该是17，对齐后应该是24, 但实际结果是16，原因是前面说的编译器的原因，没有在class M 的object中加上X的1 byte。

## 3.2 Data Member 的布局

nonstatic data members在class object中的排列顺序将和其声明的顺序一样的。但C++ standard允许编译器将多个access sections之中的data members自由排列，不必在乎他们的出现在class中的声明顺序。C++ standard要求，在同一个access section中，member的排列只需要符合“较晚出现的members排列在class object中较高的地址”即可，并不一定要连续排列。介于members之间的东西可能是边界填补(alignment)。

## 3.3 Data Member 的存取

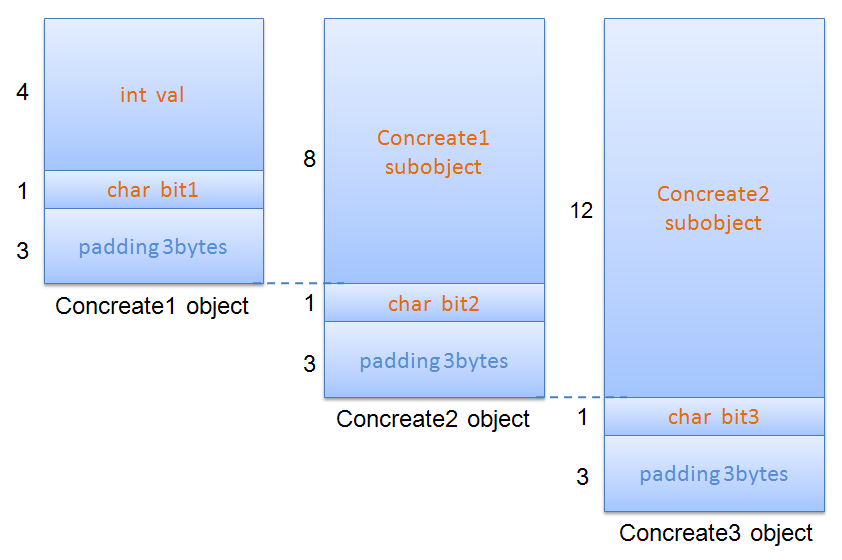
每一个member的存取许可（private public protected），以及与class的关联，并不会导致任何空间上或执行时间上的额外负担——不论是在个别的class objects或是在static data member本身。

static data members被视为global变量，只有一个实体，存放在程序的data segment之中，每次取static member就会被内部转化为对该唯一的extern实体的直接参考操作。若取一个static data member的地址，会得到一个数据类型的指针，而不是指向其class member的指针。

nonstatic data members：欲对一个nonstatic data member进行存取操作，编译器需要把class object的起始地址加上data member的偏移量（在编译时期就可以获知）。

## 3.4 继承与Data Member

### 3.4.1. 只要继承不要多态



这种情况并不会增加空间或存储时间上的额外负担。base class和derived class的objects都是从相同的地址开始，其差异只在于derived object比较大，用以容纳自建的nonstatic data members，把一个derived class object指定给base class的指针或引用，并不需要编译器去调停或修改地址，它很滋润的可以发生，而且提供了最佳执行效率。

### 3.4.2. 加上多态

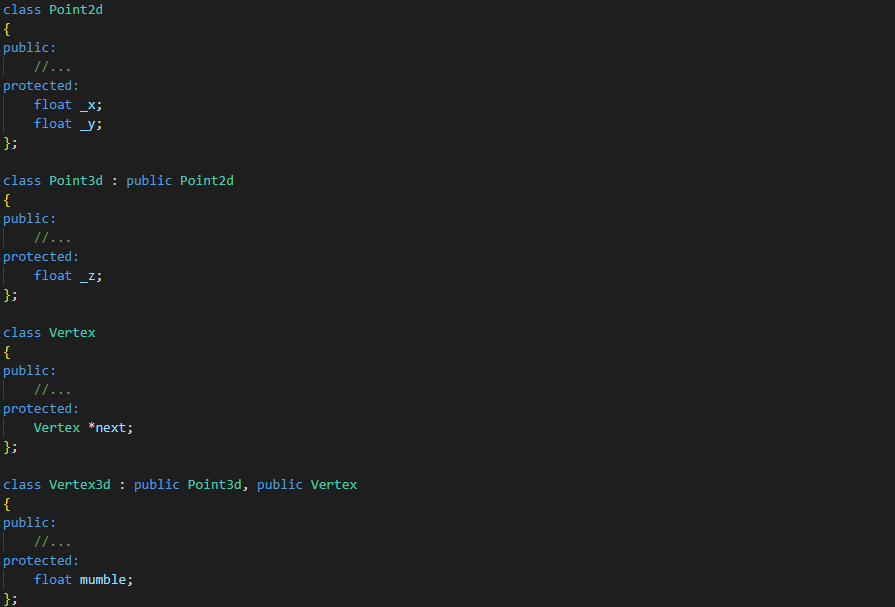
这种情况会带来空间和存取时间的额外负担：

1. 导入一个virtual table，用来存储它所声明的每一个virtual functions的地址。
2. 在每一个class object中导入一个vptr，提供执行期的链接，使每一个object能够找到相应的virtual table。
3. 加强constructor，使它能够为vptr设定初始值，让它指向class所对应的virtual table。
4. 加强destructor，使它能够消抹“指向class相关virtual table”的vptr。

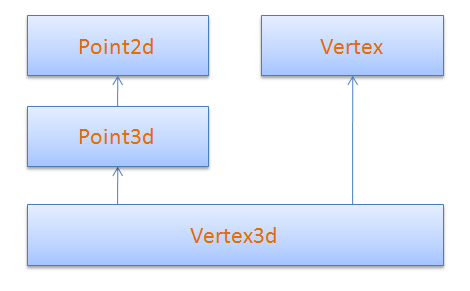
### 3.4.3. 多重继承

多重继承的主要问题发生于derived class objects和其第二或后继的base class objects之间的转换。对于一个多重派生对象，将其地址指定给“最左端（第一个）base class的指针”，情况和单一继承时相同，因为二者都指向了相同的起始地址。至于第二个或后面的base class的地址指定操作，则需要将地址修改过：加上（或减去，如果是downcast）介于中间的base class subobject(s)的大小。

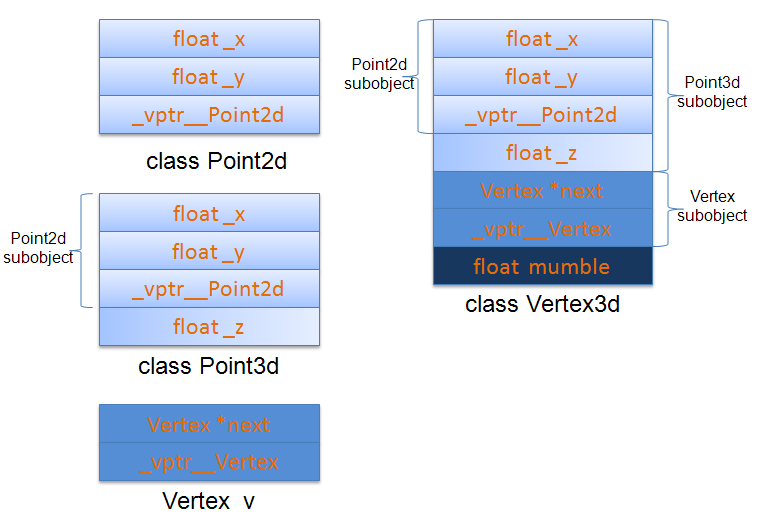
如果要存取第二个（或后面）的base class中的一个data member，不需要付出额外的成本，因为members的位置在编译时就固定了，因此存取member只是一个简单的offset的运算。比如下面的例子：

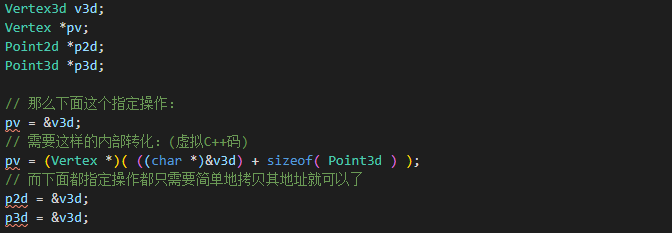


至此，Point2d、Point3d、Vertex、Vertex3d的继承关系如下图所示：



下面是该多重继承的数据布局示意图：

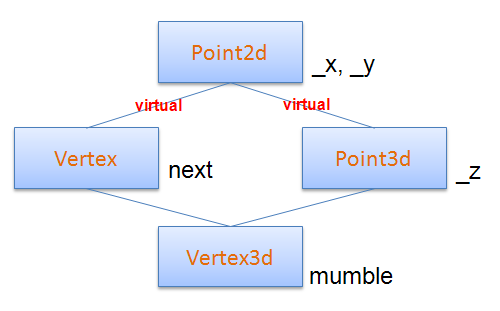


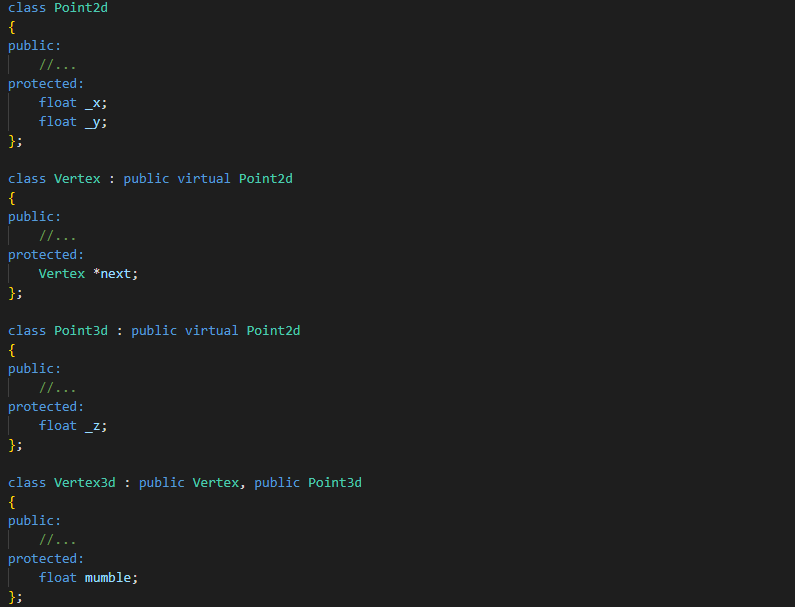


### 3.4.4. 虚拟继承

class如果含有一个或多个virtual base class subobjects将被分割为两部分：一个不变局部和一个共享局部。不变局部中的数据，总是能有固定的offset。这部分可以被直接存取，至于共享部分，所表现的就是virtual base class subobject，这个部分数据，其位置因为每次派生操作而有变化，所以只能间接存取。如果没有virtual functions的情况下，它们和C struct完全一样。

下图是Point2d、Point3d、Vertex、Vertex3d的继承体系：





# 第四章 Function语义学

## 4.1. Member的各种调用方式

### 非静态成员函数

实际上编译器是将member function被内化为nonmember的形式，经过下面转化步骤：

1. 改写函数的signature，给函数添加额外参数——this。

Point3d Point3d::magnitude( Point3d \*const this )

如果member function是const，则变成：

Point3d Point3d::magnitude( const Point3d \*const this )

1. 将对每一个nonstaitc data member的存取操作改为this指针来存取。
2. 将member function重写成一个外部函数。对函数名经过mangling处理，使之成为独一无二的语汇。

### 虚拟成员函数

将

ptr->f(); //f()为virtual member function

内部转化为

(\*ptr->vptr[1])(ptr);

其中：

* vptr表示编译器产生的指针，指向virtual table。它被安插在每一个声明有（或继承自）一个或多个virtual functions的class object中。
* 1是virtual table slot的索引值，关联到normalize()函数。
* 第二个ptr表示this指针。

### 静态成员函数

静态成员函数不能被声明为const volatile或virtual。

一个static member function会提出于class声明之外，并给予一个经过mangling的适当名称。如果取一个static member function的地址，获得的是其在内存的位置也就是地址，而不是一个指向“class member function”的指针，如下：

&Point::count();

会得到一个数值，类型是：unsigned int(\*)()，而不是：unsigned int(Point::\*)();

## 4.2. 虚拟成员函数

虚函数的一般实现模型：每一个class有一个virtual table，内含该class之中有作用的virtual function地址，然后每个object有一个vptr，指向virtual table所在。

识别一个class是否支持多态，唯一适当的方法使看看它是否有任何虚函数。只要class拥有一个虚函数，它就需要额外的执行信息：比如执行ptr->z()

1. ptr所指对象的真实类型，这可以使我们选择正确的z()实例；
2. z()实例的位置，以便我能够调用它。

在实现上，在每一个多态的class object身上增加两个members：

1. 一个字符串或数字，表示class的类型；
2. 一个指针，指向某表格，表格中持有程序的虚函数的执行期地址。

表格中的虚函数地址可以在编译时期获知（虚函数可经由其class object调用），这一组地址是固定不变的，执行期不会改变它。程序执行时，表格的大小和内容都不会改变，所以其建构和存取可以由编译器完全掌控，不需要执行期的任何介入。

执行期备妥那些地址后，另一个问题是：找到那些地址：

1. 为了找到表格，每个class object被安插了一个由编译器内部产生的指针，指向该表格；
2. 为了找到函数地址，每个虚函数被指派一个表格索引值。

这些工作都是编译器完成，执行期要做的是在特定的virtual table slot（记录虚函数地址）中激活虚函数。

### 单一继承

一个class只有一个virtual table，每个table内含其对应的class object中所有激活的虚函数地址。这些虚函数包括：

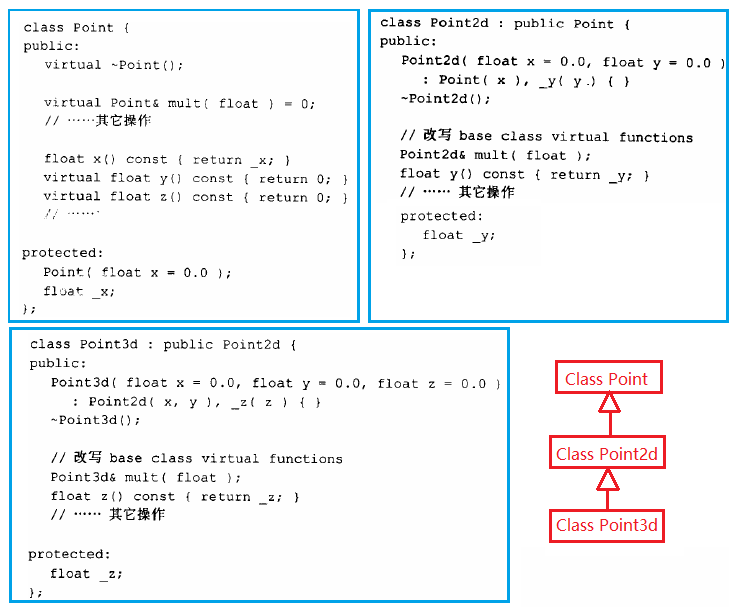
1. 该class定义的函数实例，它会overriding一个可能存在的base class virtual function；
2. 继承自base class的函数实例（derived class不overriding）
3. 一个纯虚函数实例。

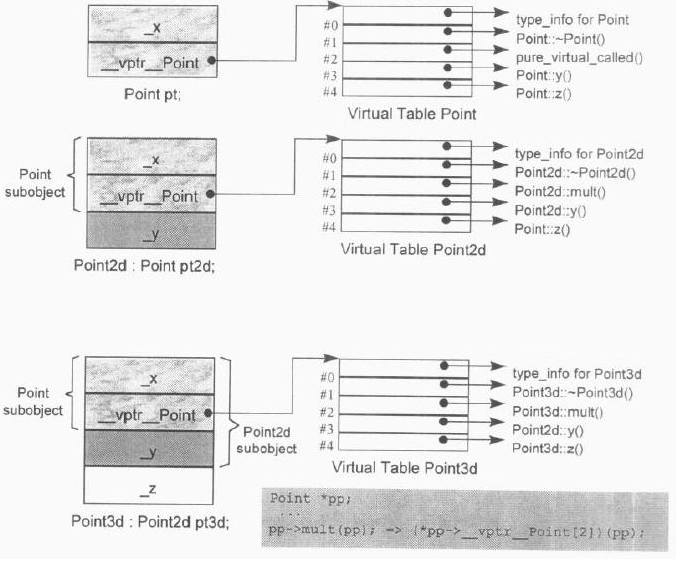
现在，如果我有这样的式子：ptr->z()，我如何有足够的知识能够在编译时期设定虚函数的调用呢？

1. 每次调用z()时，我并不知道ptr所指对象的真正类型，但我知道经由ptr可以存取到该对象的virtual table；
2. 虽然我不知道哪一个z()函数的实例会被调用，但我知道每一个z()函数地址都被放在virtual table的某个固定索引的slot中。

**多态调用的原理：**

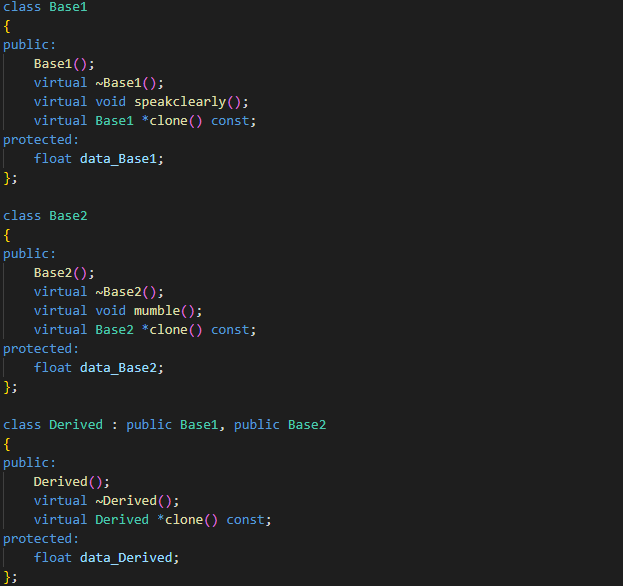
基类指针ptr指向派生类的对象，派生类的对象在构建时，会生成一个指向派生类虚表的虚指针，并设定初值。经由ptr可以存取到virtual table。基类的虚函数在派生类虚表中的位置保持不变，最终调用的是，派生类重写的虚函数。执行期所知道的是，所调用的函数的真实实体。



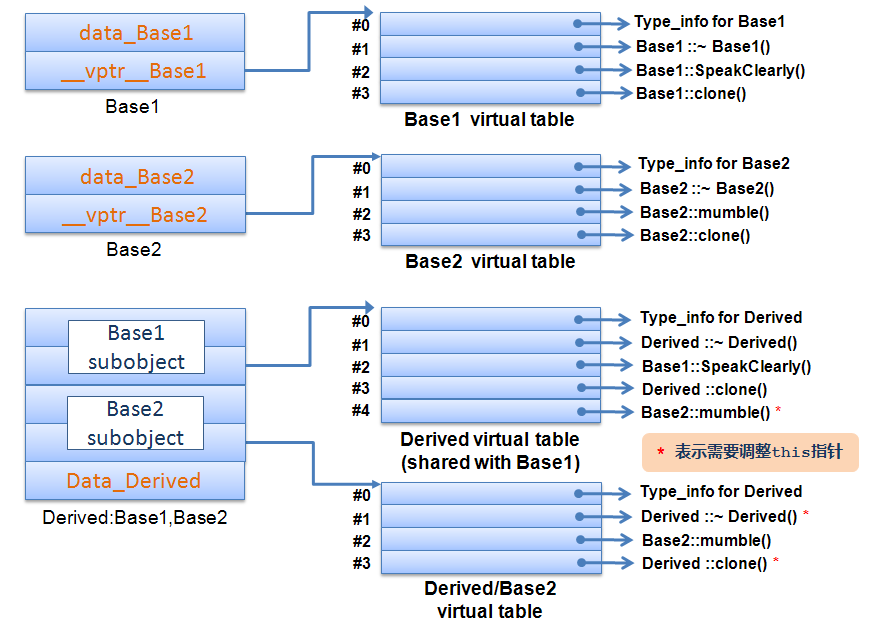


### 多重继承下的virtual function

有多少个基类，派生类就会有多少个虚表以及指向虚表的虚指针。在多重继承中支持virtual functions，其复杂度围绕在第二个以及后继的基类身上，以及“必须在执行期调整this指针”这一点上。以下面的class体系为例：



该多重继承体系的虚表布局情况如下所示：



基类的不同，被处理的虚表指针不同。以下三种情况需要调整指针：

1. 派生类指针指向派生类对象，使用虚表1，访问虚表2的函数时。
2. base2基类的指针指向派生类对象，调用析构函数。
3. 派生类对象虚函数执行的返回值是派生类指针，赋给base2基类指针。

首先，把一个从堆中配置而得的Derived对象的地址，指定给一个Base2指针：

Base2 \*pbase2 = new Derived;

新的Derived对象的地址必须调整，以指向其Base2 subobject。编译时期会产生以下的代码：

Derived \*temp = new Derived;

Base2 \*pbase2 = temp ? temp + sizeof(Base1) : 0;

当程序员要删除pbase2所指的对象时：

delete pbase2;

指针必须再次被调整，以便再一次指向Derived对象的起始处。一般规则是，经由指向“第二或后继之base class”的指针（或引用）来调用derived class virtual function，那么该调用操作所需“必要的this指针调整”操作，必须在执行期完成。也就是说，offset的大小，以及把offset加到this指针上头的那一小段程序代码，必须由编译器在某个地方插入。

调整this指针的另外一个负担是，由于两种不同的可能：(1)经由derived class(或第一个base class)调用，(2)经由第二个(或其后继)base class调用，同一函数在虚表中可能需要多笔对应的slots。例如：

Base1 \*pbase1 = new Derived;

Base2 \*pbase2 = new Derived;

//…

delete pbase1;

delete pbase2;

虽然两个delete导致相同的Derived destructor，但它们需要两个不同的virtual table slots：

* pbase1不需要调整this指针（因为Base1已经指向Derived对象都起始处）。其virtual table slot需放置真正的destructor地址。
* pbase2需要调整this指针，其virtual table slot需要相关的thunk地址。

**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 );

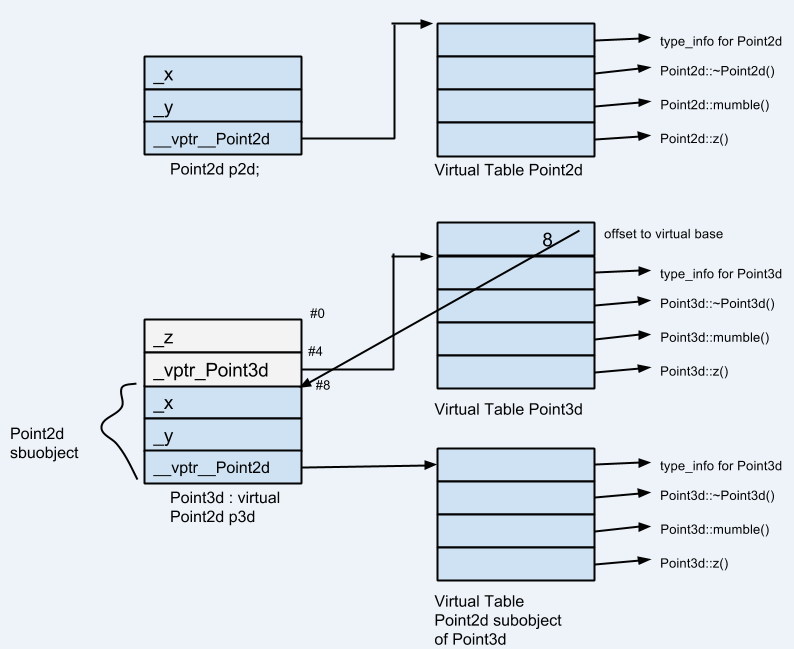
Derived::~Derived( this );

在多重继承下，一个derived class内含n-1个额外的virtual tables，n表示其上一层base classes的数目。对于本例而言，会有两个virtual table被编译器产生出来：

* 一个主要实体，与Base1(最左端base class)共享；
* 一个次要实体，与Base2(第二个base class)有关。

针对每一个virtual table，Derived对象中有对应的vptr。vptrs将在constructor(s)中被设立初值(经由编译器所产生出来的码)。

## 4.3. 虚继承下的 Virtual Functions



## 4.4. 指向成员函数的指针

取一个非静态数据成员的地址，得到的时该member在class布局的offset+1；取一个非静态成员函数的地址，如果是nonvirtual，则得到它在内存中的真正地址；对一个virtual member function取其地址，所获得的是一个索引值。

指向member function的指针声明语法，以及指向"member selection运算符"的指针，其作用就是作为this指针的空间保留者。这也就是为什么静态成员函数(没有this指针)的类型是“函数指针”，而不是指向member function指针的原因。

### 支持指向"Virtual member Functions"之指针

对一个非静态成员函数取地址，将获得该函数在内存中的地址。对一个虚函数而言，它的地址在编译时期是未知的，所知道的仅仅是虚函数在其相关的Virtual Table中的索引值。所以，对一个Virtual member function取地址，能获得的只是一个索引值。

### 多重继承下指向成员函数的指针

## 4.5. 内联函数

一般而言，处理一个inline函数，有两个阶段：

1. 分析函数定义，以决定函数的“本质内联能力”。如果函数因其复杂度或者其建构问题，被判断为不可称为内联，它会被转为一个static函数，并在被编译模块内产生对应的函数定义。
2. 真正的内联函数扩展操作时在调用的那一点上。这会带来参数的求值操作和临时对象的管理。

在内联扩展期间，每一个形式参数都被对应的实际参数取代。

# 第五章 构造、析构、拷贝语义学

## 5.1. 无继承情况下的对象构造

## 5.2. 继承体系下的对象构造

当定义一个object如下：

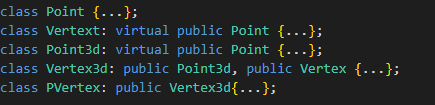
T object;

如果T有一个构造函数（不论是用户定义的还是编译器合成的），构造函数会被调用。对于一个class，编译器所做的扩充操作大致如下：

1. 记录在member initialization list中的data members初始化操作会被放进constructor的函数本体，并以members的声明顺序为顺序。
2. 如果有一个member并没有出现在member initialization list中，但它有一个default constructor，那么该default constructor必须被调用。
3. 在那之前，如果class有vptr，它们必须被设定初值，指向适当的virtual table(s)。
4. 在那之前，所有上一层的base class constructor必须被调用，以base class的声明顺序[继承顺序]为顺序(与member initialization list中的顺序没有关联)
   1. 如果base class被列于member initialization list中，那么任何显式指定的参数都应该传递过去
   2. 如果base class没有被列于member initialization list中，而它有default constructor(或者default memberwise constructor)，那么就调用之。
   3. 如果base class是多重继承下的第二或后继的base class，那么this指针必须有所调整。
5. 在那之前，所有virtual base class constructor 必须被调用，从左到右，从最深到最浅：
   1. 如果class被列于member initialization list中，那么如果有任何显式指定的参数，都应该传递过去。若没有列于list中，而class有一个default constructor，亦应该调用之。
   2. 此外，class中的每一个virtual base class subobject的偏移位置必须在执行期可被存取。
   3. 如果class object是最底层的class，其constructor 可被调用；某些用于支持这些行为的机制必须被放进来。

### 虚拟继承

假设有如下class的派生情况：

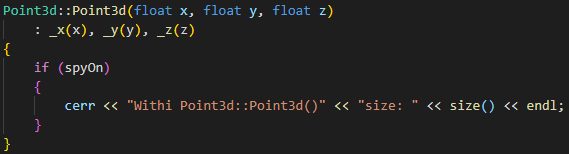


Point3d和Vertex同为Vertex3d的subobjects时，Point3d和Vertex不能调用Point的构造函数。Point的构造函数应该由最底层的class（most-derived），即Vertex3d来调用。

在类似的情况中，虚基类的构造函数的调用有明确的定义：只有当一个完整的class object被定义出来时，它才会被调用；如果object只是某个完整object的subobject，它就不会被调用。

### vptr初始化语义学

同样以上图的类继承关系为例，假设这个继承体系中每一个class都定义了一个virtual function size()，此函数负责传回class的大小。那么，如果我们有如下的调用：



那么在定义PVertex object时，前述的5个constructor应该调用哪个class的size呢？

在Point3d constructor中调用size()函数，必须被决议为Point3d::size()而不是PVertex::size()。更一般地说，在一个class(本例Point3d)的constructor(和destructor)中，经由构造函数的对象(本例为PVertex)来调用一个virtual function，其函数实例应该是在此class(本例为Point3d)中有作用的那个。我们知道，constructor的调用顺序是：由根源而末端(bottom up)、由内而外(inside out)。在 PVertex constructor执行完毕之前，PVertex 并不是一个完整对象；Point3d constructor执行之后，只有Point3d subobject构造完毕。

我们知道，虚拟函数的决议用的是 vptr，那么只要控制 vptr 的初始化和设定操作就能达到目的了。我们对vptr的初始化操作为：

在base class constructor调用操作之后，但是在程序员供应的代码或是"member initialization list中所列的members初始化操作"之前

令每一个base class constructor设定其对象的 vptr，使它指向相关的 vtbl 后，构造中的对象就可以严格而正确地变成“构造过程中所幻化出来的每一个 class”的对象。也就是说，一个PVertex对象会先形成一个Point对象、一个Point3d对象、一个Vertex对象、一个 Vertex3d对象，然后才成为一个PVertex对象。constructor的执行算法如下：

1. derived class constructor中，“所有virtual base classes”及“上一层base class”的constructor 会被调用；
2. 上述完成后，对象的vptr(s)被初始化，指向相关的vtbl(s)
3. 如果有member initialization list的话，将在constructor体内扩展开来。这必须在vptr被设定之后才做，以免有一个virtual member function被调用。
4. 最后，执行程序员所提供的代码

由上可以得出下面的结论：在class的constructor的member initialization list中调用该 class的一个虚拟函数，因为vptr保证能够在member initialization list被扩展之前设定好，所以是安全的。但是可能出现该虚拟函数依赖未设定好的其他members的情况，在这种情况下，是不安全的。

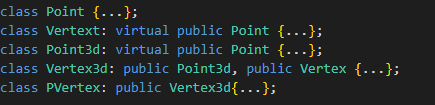
## 5.3. 对象复制语义学

如果一个class显示出bitwise copy语义，则拷贝构造赋值函数是trivial，不会被合成出来。在以下情况，一个class不会表现出bitwise copy语义：

* 1. 当class内含一个member object，而其class有一个拷贝赋值函数；
  2. 当一个class的base class有一个拷贝赋值函数时；
  3. 当一个class声明了任何虚函数；
  4. 当一个class继承自一个虚基类（不论此虚基类有没有copy operator）时。

## 5.5. 析构语义学

如果class没有定义析构函数，那么只有在class内含的member object（或class自己的base class）拥有析构函数的情况下，编译器才会自动合成出一个来。否则，析构函数被视为不需要，也就不需被合成（即使它拥有虚函数），当然更不需要被调用。



在虚拟继承的情况下，如上面图所示的继承结构，如果我们提供一个Vertex3d的析构函数，编译器会扩展它：

1. 如果object内含一个vptr，那么首先重设（reset）相关的virtual table；
2. 析构函数的本体现在被执行，也就是说vptr会在程序员的代码执行之前被reset；
3. 如果class拥有member class object时，而后者拥有析构函数，那么它们会以其声明顺序相反的顺序被调用；
4. 如果有任何直接的（上一层）nonvirtual base classes拥有析构函数，它们会以其声明顺序相反的顺序被调用；
5. 如果有任何虚基类有析构函数，而目前讨论的这个class是most-derived class，那么它们会以其原来的构造顺序的相反顺序被调用。

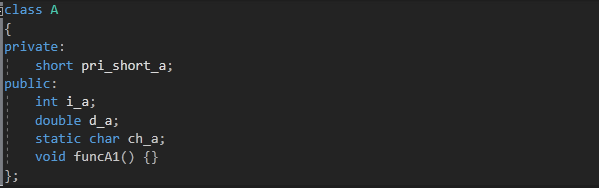
# 第六章 执行期语义学

* **全局对象**：C++程序中所有全局对象都放在数据段中，如果程序员没有给全局对象赋予初始值，全局对象的值将会被配置为0。
* **局部静态对象**：构造函数和析构函数都只能被调用一次，即使该函数可能会被调用多次。

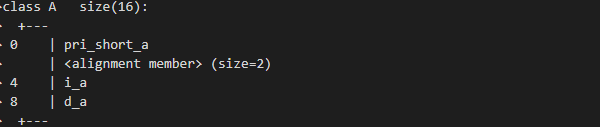
# C++类的布局（继承、多态）

## A．无继承

### A.1. 无虚函数



A的大小及布局如下：

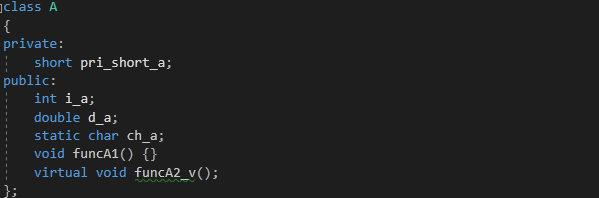


如上可以说明：

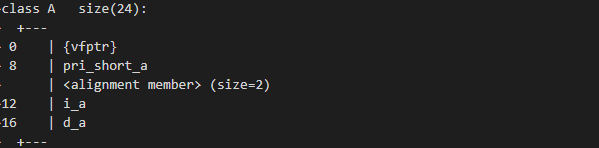
1. 静态数据成员虽然属于类，但不占用具体类对象的内存。
2. 成员函数不占用具体类对象内存空间，成员函数存在代码区。
3. 数据成员的访问级别并不影响其在内存的排布和大小，均是按照声明的顺序在内存中有序排布，并适当对齐。

### A.2. 有虚函数

在1.1中的类A里增加一个虚函数：



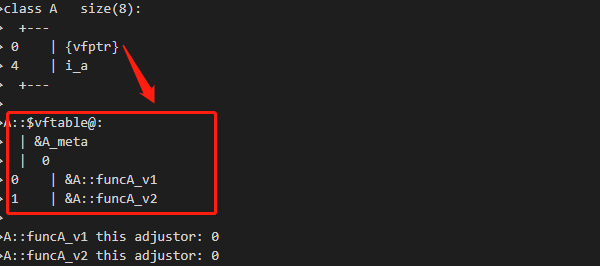
其内存大小及布局如下：



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()：



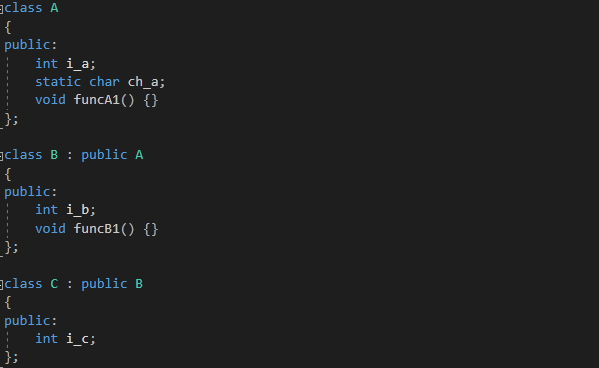


所以，不论再多虚函数，都只会有一个虚指针vptr，不会改变类的大小。不同之处在于，虚指针所指向的虚表中会多一个项目，即指向另一个虚函数的地址。

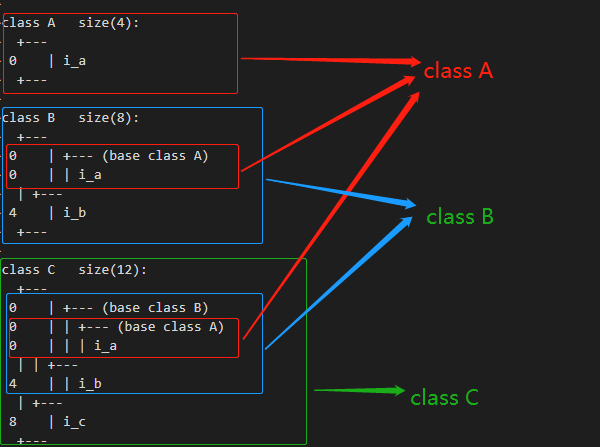
## B. 单一继承

### 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：



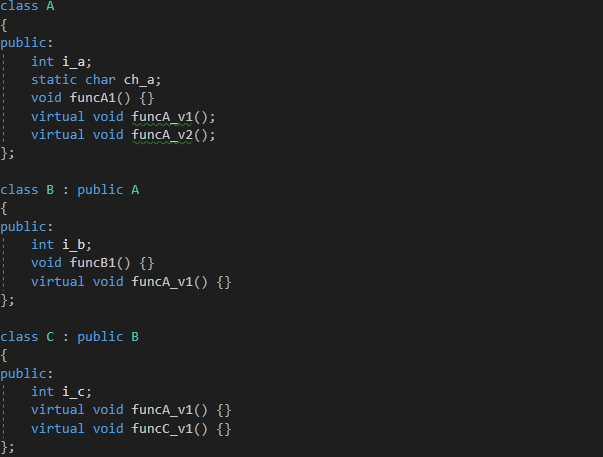
内存布局如下：



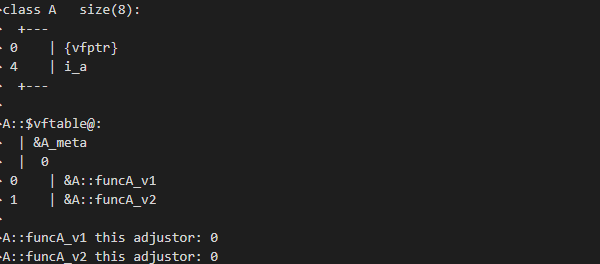
单一继承的内存布局很清晰，每个派生类中起始位置都是Base class subobject。现在我们在类中增加虚函数，观察在单一继承+有虚函数的情况下，类的内存布局。

### B.2. 单一继承且有虚函数

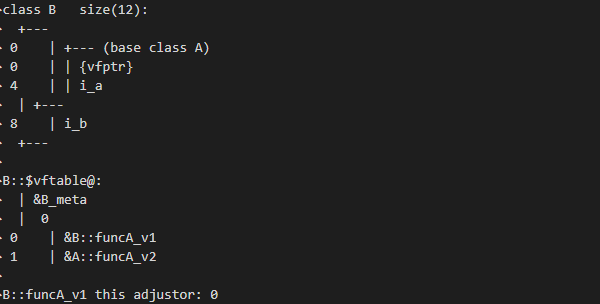
如下，类A增加了两个虚函数funcA\_v1()和funcA\_v2()；类B继承自A，覆写funcA\_v1()；类C继承自B，重写funcA\_v1()，且有自己定义的一个虚函数funcC\_v1()：



Class A的内存布局如下，如同A.2，这里不再解释：

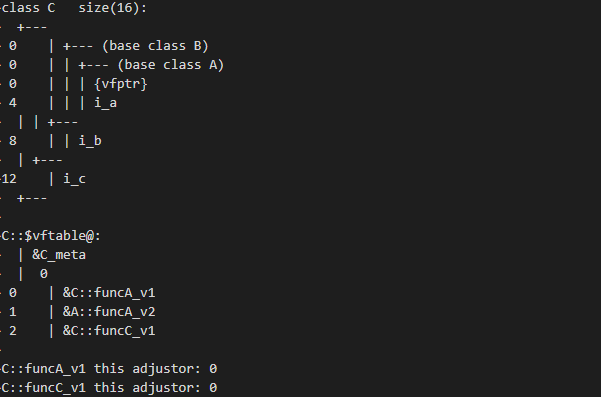


Class B的内存布局如下：

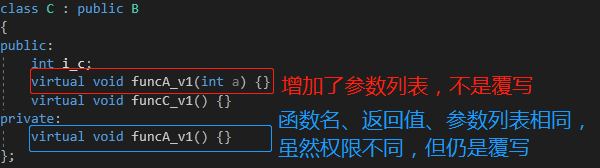


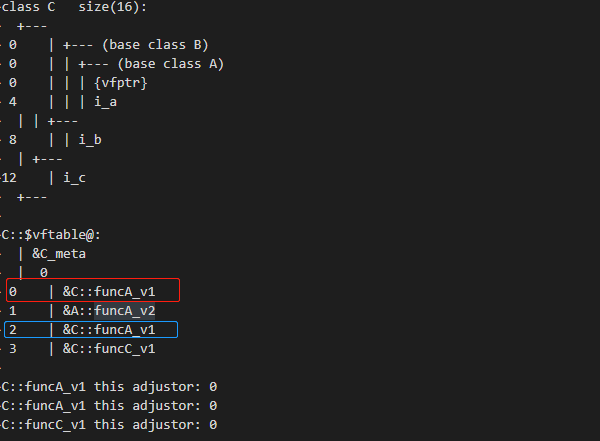
B中首先也是基类A subobject，同样含有一个虚指针vptr。由于B覆写了funcA\_v1()，故虚表中第一个索引处的函数地址是&B::funcA\_v1()。

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

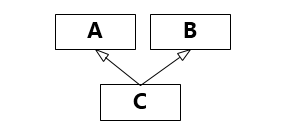
《Effective C++》第三版，Item 07：为多态基类声明virtual析构函数。

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。

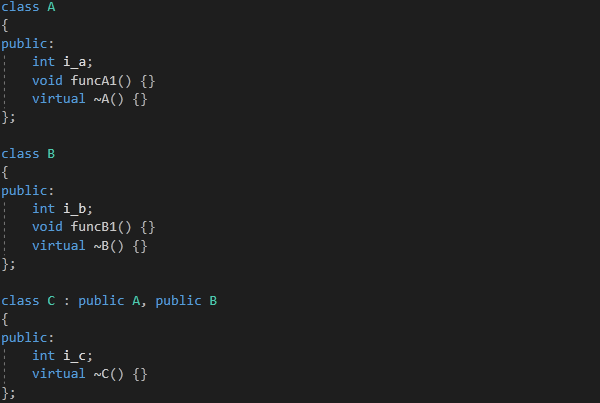
在接下来的示例中，我们将加上虚析构函数。

## C. 多重继承

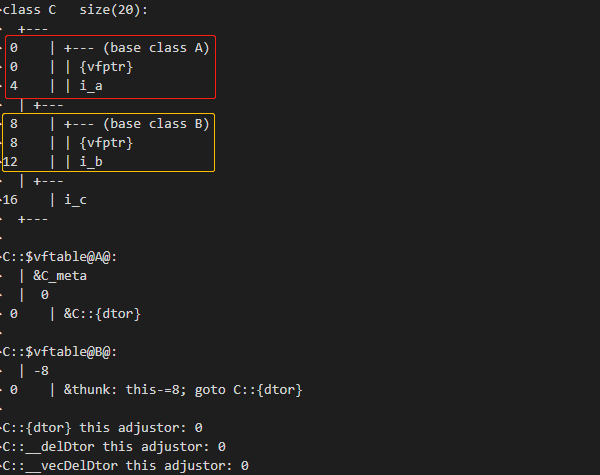
### C.1. 多重继承



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

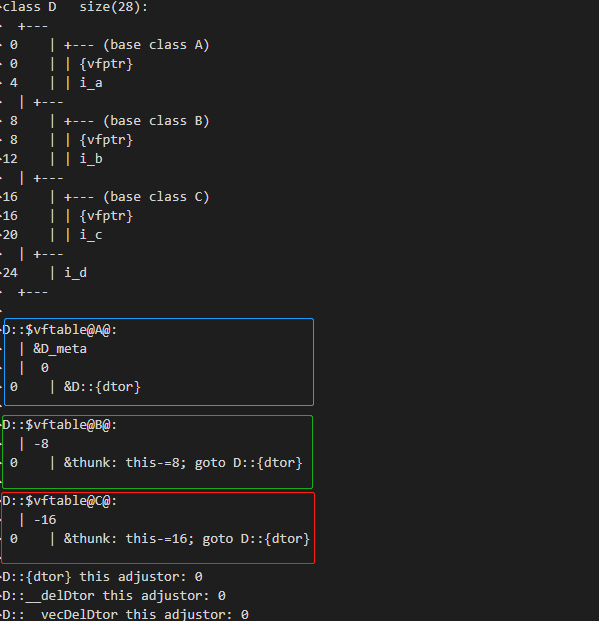
pbase2\_dtor\_thunk:

this += sizeof( base1 );

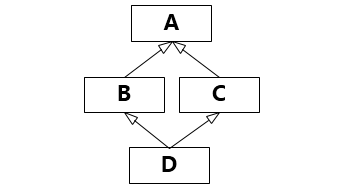
Derived::~Derived( this );

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。

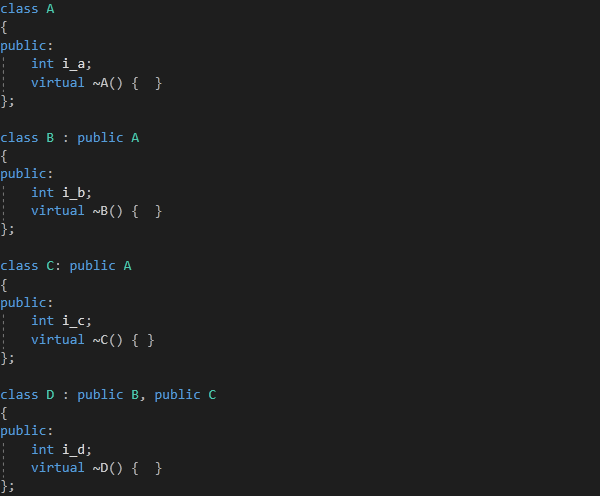
同时也可以想到，随着base class的数量增多，派生类里也会首先顺序存放各个基类subobject。而派生类中也会记录其到各个base subobject的offset。如下图是类D同时继承类A、B、C：



### 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：

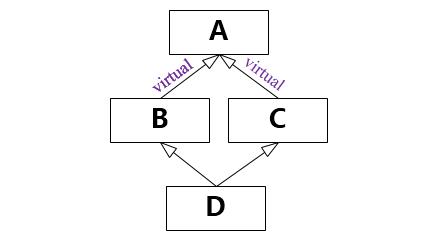


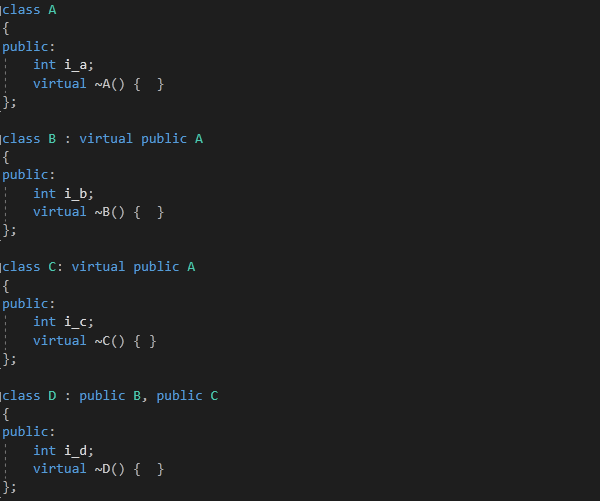
如上图，D中依次存放基类B subobject和基类C subobject。其中B和C中均存放一份class A subobject。

### 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。

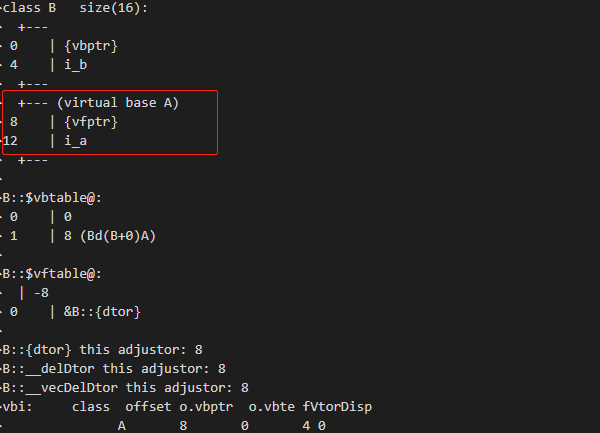
同样以C.2.中的继承关系为例，不过这次我们B和C对A的继承都加上了关键字**virtual**。





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。

