1. **关于对象**

在C中，“数据”和“处理数据的操作(函数)”是分开声明的，也就是说，语言本身并没有支持“数据和函数之间”的关联性。这种程序写法称作“程序性的(procedural)”,即所谓面向过程。

抽象数据类型(Abstract data type,ADT): 抽象数据类型 = 逻辑结构 + 抽象运算，也就是说抽象数据类型是建立在逻辑结构上的一些抽象的运算.比如C++中的String class，其支持 = ，+等运算。

**数据抽象**：用ADT描述程序处理的实体时，强调的是其本质的特征、其所能完成的功能，以及它和外部用户的接口。我们需要注意的是：这里在描述数据时忽略了数据的形态，而是一种数据抽象，且不会涉及到高级程序语言中的具体实现和存储。

**数据封装**：将实体的外部特征和内部实现细节分离，并且对外部用户隐藏其内部实现细节（即抽象数据类型内部的各种定义），对于用户来说只需要使用即可，不需要知道内部的具体实现细节。

**加上封装后的布局成本，并没有增加！**C++在布局以及存取时间上的主要额外负担是由virtual引起的，包括virtual function机制和virtual base class。

* 1. **C++对象模式**

在C++中，有两种类数据成员，即static和nonstatic，以及三种类成员函数，static、nonstatic、virtual。

**简单对象模型**：最基本的模型，为了尽量减低C++编译器的设计复杂度而开发，赔上空间和运行期效率。此模型中，一个object是一些列的slots，每一个slots指向一个members，members按其声明顺序，各被指定一个slot。每个数据成员或者成员函数都有一个自己的slot。这个模型并未应用于实际产品，但关于索引及slot个数的观念被应用于C++“指向成员的指针”观念中。

**表格驱动对象模型**：为了对所有类的所有object都有一致的表达形式，这种模型把所有与成员相关的信息抽取出来，放在一个成员函数表和一个成员数据表内。Class object本身则内含指向这两个表格的指针。成员函数表是一系列的slots，每个slots对应一个成员函数，而数据成员表则直接持有数据本身。此模型虽然没有用于真正的C++编译器上，但成员函数表成为了虚函数的一个支持方案。

**C++对象模型**：最初设计的(目前仍有优势)C++对象模型是从简单对象模型派生来的。并对内存空间和存取时间做了优化。此模型中，nonstatic data members被配置与每一个class object之内，static data members则存放在个别的class object 之外。Static和nonstatic function members也被放在个别class object之外。虚函数则以两个步骤支持之------1、每一个类产生一堆指向虚函数的指针，放在表格中，被称作虚函数表(vtbl) 2、每个class object被安插一个指针，指向相关的虚函数表，这个指针被称作vptr。

不同的对象模型，会导致“现有的程序代码必须修改“以及”必须加入新的程序代码“两个结果。

* 1. **关键词所带来的差异**

什么时候一个人应该实用struct代替class？

**Class的真正特性是由声明本身所决定的。**

**策略性正确的struct**：C的设计技巧有时候可能成为C++的陷阱！比如把单一元素的数组放在一个struct的末尾，于是每个struct object可以拥有可变大小的数组:

Struct muble{

Char pc[1]

}；

处于同一个access section的数据，必定保证以其声明顺序出现在内存布局中。而放置在多个access section内的数据，排列顺序就不一定了。同理，基类和派生类的数据成员布局也没有谁先谁后的强制规定，因此也不保证前述C的技巧有效。虚函数的存在也会给C的技俩的有效性成为一个问号。所以最好的忠告是，不那么做！

C struct在C++中的一个合理用途，是当要传递“一个复杂的class object的全部或者部分“到某个C函数时，struct 声明可以将数据封装起来，并保证与C兼容得空间布局。然而这只在复合得情况下才存在。这种做法时，从C struct 中派生C++的部分:

Struct C\_point{……};

Class point : public C\_pont {…};

* 1. **对象的差异**

一些比较常见的范式：函数化程序设计、逻辑程序设计、语意数据模型、几何计算、数值计算、面向对象设计、原型设计、自然语言。

C++程序设计模式直接支持三种程序设计范式(programming paradigms)**:1.程序模型(procedural model) 2.抽象数据类型模型(abstract data type model) 3.面向对象模型(object-oriented model).**

纯粹以一种范式写程序，有助于整体行为的良好稳固，但如果混合了多种范式，可能造成意想不到的结果。比如，虽然可以直接或间接处理继承体系中的一个基类对象，但**只有通过指针或引用的间接处理，才能支持OO程序设计所需的多态性**。在面向对象设计范式中，程序员要处理一个未知实例，它的类型虽然有所界定，却有无穷可能，原则上，被指定的对象的真实类型在每一个特定的执行点前，是无法解析的！在C++中，只有通过指针和引用的操作才能够完成，相反，在ADT范式中，程序员处理的实例是一个拥有固定而单一类型的实例，其在编译期就完全定义好了！

比如:

Librar\_materials \*px = retrieve\_some\_material();

Librar\_materials &rx = \*px;

Librar\_materials dx = \*px;

在这里，无法确定px或者rx到底指向何种类型的对象，只能说要么是一个Librar\_maerials类的对象或者其子类对象，但是，可以确定dx只能是一个Librar\_materials对象。

C++通过以下方式支持多态：

1. 经由一组隐式转化操作，比如把一个派生类指针转化为一个指向其public base type的指针。

Shape \*ps = new circle();

1. 经由虚函数机制

Ps->rorate();

1. 经由dynamic\_cast和type\_id运算符

If(circle \*pc = dynamic\_cast<circle \*>(ps))…

多态的主要用途式经由一个共同接口来影响指针的封装。这个接口通常被定义于一个抽象的基类中。

需要多少内存才能表现一个class object？一般而言要有：

1. 其nonstatic data members 的总和大小
2. 加上任何由于alignment的需求而填补上的空间(alignment就是将数值调整到某数的倍数。在32位机上，通常为4bytes，以使得bus的“运输量”达到最高效率)
3. 加上为了支持virtual而由内部产生的任何额外负担(一个指针，不管指向任何类型，大小都是固定的)

**指针的类型**:指向不同类型之各指针间的差异，既不在其指针表示法不同，也不再其内容(仅存储地址)不同，而是在其所寻址出来的object类型不同，也就是说，指针类型会教导编译器如何解释某个特定地址中的内存内容及其大小。

**比如，**一个指向地址1000的整数指针，其在32位机上，将涵盖地址空间1000~1003(int是4位)。如果String是传统的8-bytes(包括一个4-bytes的字符指针和一个用来表示字符串长度的整数)，那么一个ZooAnimal指针将横跨地址空间1000~1015(4位int,8位string,4位虚表指针)。那么，一个类型是void的指向地址1000的指针，将涵盖怎样的地址空间呢？这是未知的，所以这就是为什么一个类型为void的指针只能拥有一个地址，而不同通过这个地址操作所指之object！

**加上多态后:**现在，定义一个Bear,作为一种ZoomAnimal。当然，经由public继承可以完成这项任务。

Class Bear : public ZooAnimal {

Public:

Bear();

~Bear();

//…

Void rorate();

Virtual void dance();

//…

Protected:

Enum Dances{…};

Dances dances\_kown;

Int cell\_block;

};

Bear b(“Yogi”);

Bear \*pb = &b;

Bear &rb = \*pb;

不管是指针或者引用都只需要一个word的空间(32位机上4bytes)。Bear object需要24bytes，也就是ZoomAnimal的16bytes加上Bear所带来的8bytes。

假设Bear object放在地址1000处，那么一个ZoomAnimal的指针与Bear的指针区别，它们每个都指向Bear Object的第一个byte，差别是，Bear的指针对应的地址涵盖整个Bear object，而ZoomAnimal的指针涵盖范围是Bear Object中的ZoomAnimal subobject(即属于基类的成分，注意，**虚表指针也属于基类的成分**！)。

但是注意，如果不是以指针或者引用的方式来获取Bear object，那么会引起所谓的切割(sliced)

Bear b;

ZoomAnimal za = b;

Za.rotate; //这里调用的不是bear的rorate而是zoomAnimal的

造成切割的原因是，编译器必须保证如果某个object含有一个或者一个以上的vptrs，那些vptrs的内容不会被基类对象初始化或者改变。

此外，za也不是一个Bear，二十一个ZoomAnimal,因为多态并不支持“直接存取object”这事之上(需要使用指针或者引用)。但如果以这样的方式：ZoomAnimal \*za =Bear b就可以使用多态了！

一个指针或者一个引用之所以支持多态，是因为它们并不引发内存中任何“与类型有关的内存委托操作”，会受到改变的，只有它们所指向的内存的“大小和内容解释方式”而已。**C++通过指针或者引用来支持多态，这种程序设计风格就叫做面向对象**！

C++也支持具体的ADT程序风格，如今被称为object-based(OB),例如string class就是一种非多态的数据类型。一个OB设计可能比一个对等的OO设计速度更快且空间紧凑。速度快事因为所有的函数调用操作都在编译器完成(一般编译期所完成的工作份额比例越大，速度越快)，对象构建起来不需要设置virtual机制；空间紧凑则是因为每一个class object不需要负担传统上为了支持virtual机制而需要的额外负荷，不过OB设计比较缺少弹性。程序设计往往在弹性(OO)和效率(OB)间做抉择。

1. **Data语意学**

一个表面看上去是空的类，其实并不是空的！因为编译器为它安插了一个隐藏的char，这样使得这个类被实例化后，每个对象都有独一无二的地址(即这个char的地址)。

一个class的data members，一般而言，可以表现出class在程序执行时的某种状态。Nonstatic data members放置的是“个别的class object“感兴趣的数据。Static data members 则放置的是”整个class“感兴趣的数据。在程序中，不管class被产生出多少份实例，static data members永远只存在一份实例，位于global data segment中(甚至该class没有任何实例，其static data members也已经存在)。

每一个class object必须拥有足够的大小以容纳它的所有nonstatic data members。有时候可能比想象的还大，因为：1、由编译器自动加上的额外data members，用以支持某些语言特性。2、因为alignment(边界调整)的需要。

**3.1Data Member的绑定**

早期的程序设计遵循“防御性设计风格“。实际上这些风格到今天还在，虽然必要性自C++2.0后就消失了。这类语言规则被称作“member rewriting rule”，意思是，一个inline函数实体，在整个class声明未被完全看见之前，是不会被评估求值的。也就是说，如果一个inline函数在class声明之后被立刻定义的化，对其本体的分析，会直到整个class声明都出现了才开始，因此在inline函数躯体内的一个data member绑定操作，会在整个class 都声明后才开始执行。

但是，这对member function的argument list并不为真。Argument list中的名称还是会在它们第一次遭遇时被适当地决议完成。因此。这种状况仍然需要某种防御性的程序风格:总是把”nested type声明”(比如typename xx xx)放在class的起始处！

**3.2Data Member的布局**

C++ Standard要求，在同一个access section(也就是private、public、protected等区段)中，members的排列只需符合“较晚出现的members在class object中有较高的地址”这一条件即可。也就是说，各个members不一定要连续排列，members之间可能为了alignment填补一些bytes。

编译器还可能会合成一些内部使用的data members，以支持整个对象模型。Vptr就是这样的东西。目前所有的编译器都会将其安插在一个“内涵virtual function之class”的object 内。传统上把vptr放在所有显式声明的最后，但如今也有编译器将之放在一个class object的最前端。

目前各家编译器都是把一个以上的access section连锁在一起，依照声明顺序，成为一个连续区块。Access section的多寡并不会招致额外负担。比如在一个access section内声明8个members和在8个access section内总共声明8个members，得到的object大小相同。

**3.3Data Member的存取**

**Static Data Members:**按其字面含义，被编译器提取于class之外，并被视作一个global变量(但只在class的生命范围内可见)。每个static data members只有一个实例，存放在程序的data segment中，每次程序参阅(取用)static member时，就会被内部转化为对该唯一extern实例的直接参考操作。从指令执行的观点来看，这是C++中“通过一个指针和通过一个对象来存取member，结论完全相同”的唯一情况。，因为members其实并不在class object内，即存取static members并不需要通过class object(经由member selection operators，也就是.运算符对一个static data member进行操作不过是文法上的一种方便而已)。

如果有两个class，每一个都声明了一个static member FreeList，那么当它们都被放入程序的data segment时，就会导致名称冲突。编译器的解决方法是暗中对每一个static data member编码，以获得一个独一无二的程序识别码。

**Nonstatic Data Members:**Nonstatic data members直接存放在每一个class object中。除非经过显示的(explicit)或者隐式的(implicit)class object，否则没有方法直接存取它们(static data members则可以直接访问，因为如上文所言，实际并不存放于对象实例中)。只要程序员在一个member function中直接处理一个nonstatic data member，所谓“implicit class object”就会发生。比如

Point 3d

Point3d::translate(const Point3d &pt){

x += pt.x;

y += pt.y;

z += pt.z;

}

表面上看到对于x,y,z的直接存取，实际上是经由一个”implicit class object”(**由this指针表达**)完成的，即实际上这个函数的参数是

Point 3d

Point3d::translate(Point3d \*const this , const Point3d &pt){

This->x += pt.x;

This->y += pt.y;

This->z += pt.z;

}

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 members的偏移位置(offset).比如 origin.\_y = 0.0，那么起始地址&origin.\_y等于&origin + (&Point3d::y-1).之所以有个-1操作，是因为指向data member的指针，其offset总是被加上1，这样可以使得编译器区分出“一个指向data member的指针，用以指出class的第一个member”和”一个指向data member的指针，没有指出任何member”的两种情况。

每一个nonstatic data member的偏移位置在编译器即可获知。存取一个nonstatic data members的效率和存取一个C struct member或者一个nonderived class的member是相同的。

但是，如果引入虚拟继承，虚拟继承将为“经由base class subobject存取class members引入一层间接性”，比如

Point3d \*pt3d;

Pt3d->\_x = 0.0;

其执行效率在\_x是一个struct member、一个class member、单一继承、多重继承的情况下完全相同。但是如果\_x是一个virtual base class 的member，存取效率会下降一点。因为这时候不能说pt必定指向哪一种class type(不知道编译期这个member真正的offset位置)，所以这个存取操作必须延期执行。

**3.4继承与Data Member**

在C++继承模型中，一个derived class object所表现出来的东西，是自己的members加上其base classes members的总和。至于derived class members和base classes members的排列顺序，则并未在C++ Standard中强制指定。理论上编译器可以自由安排之。

**只要继承不要多态:**一般而言，具体继承相对于虚拟继承，并不会增加空间或存取时间上的额外负担。把两个原本独立不相干的classes凑成一对”type/subtype”，并带有继承关系，容易犯重复设计相同函数的关系。同时，把一个class分解为两层或更多层，可能会为了“表现class体系之抽象化”而膨胀所需的空间。C++语言保证，出现在derived class中的base class subobject有其完整原样性(在基类中alignment的部分也会继承下来)。之所以保留alignment的部分，是因为如果C++语言把derived class members和base subobject捆绑在一起，去除填补空间，在执行赋值操作时，就会将“被捆绑在一起、继承而得的”members内容覆盖掉。

**加上多态:**为支持多态，需要带来空间和存取时间上的额外负担。

1. 导入一个虚函数表，用来存放其所声明的每一个虚函数地址。此表元素个数一般试被声明的虚函数数目，再加上一两个slots用以支持runtime type identification
2. 再每一个class object中导入一个vptr，提供执行期的链接，使得每一个对象能够找到相应的虚函数表。
3. 加强constructor，使它能够为vptr设定初值，让其指向虚函数表。
4. 加强destructor，使它能够消抹“指向class相关的虚函数表”的vptr。

那么，把vptr放在哪里最好呢？在cfront里，是放在class object的末尾，其可以保持base class C struct的对象布局，因而允许在C程序代码中使用。这种做法在C++最初问世时广为使用。而到了C++2.0后，开始支持虚拟继承及抽象类，并且由于OOC的崛起，编译器开始把vptr放在class object的起头处。这种做法的好处时，对于多重继承下，通过指向class member的指针调用virtual function会带来帮助。代价是失去了与C的兼容性。

**TIPS:**值得注意的是，即使派生类中拥有基类所不具有的虚函数，这个虚函数也会存在于同一个虚函数表中。即使用基类指针指向派生类，其中的虚函数表也会存在这个“基类所没有”的虚函数。当然，这个基类指针是无法调用它的，因为是在编译期间的语法检查，然后在运行时会通过虚函数表动态绑定实际上才调用的派生类的对应重写方法。

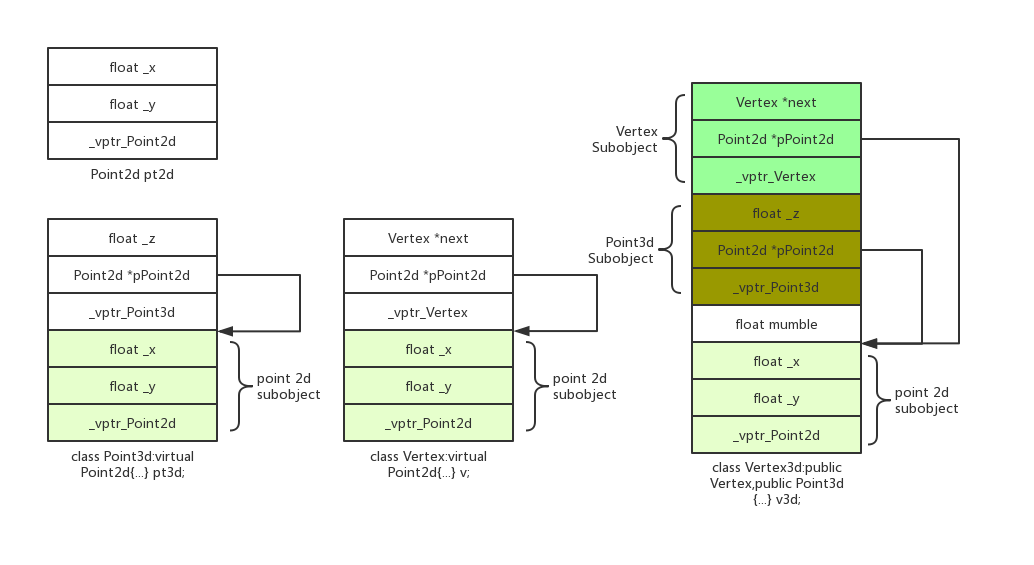
**多重继承:**单一继承提供了一种“自然多态”，base class和derived class的object都是从相同的地址开始，其间差异只在于derived class比较大，用以多容纳它的nonstatic data members。即使把一个derived class object指定给base class的指针或者引用，并不需要通过编译器去调停或修改地址，而且提供了最佳执行效率。

对于一个多重派生对象，将其地址制定给“最左端(也就是第一个)”base class的指针，情况将和单一继承相同，因为二者都指向相同的起始地址。至于第二个或者后继的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downcat的话)介于中间的base class subobject大小。

如果要存取第二个(或后继)base class中的一个data member，将会是怎样的情况？ 要付出额外成本吗？不，members的位置在编译时就固定了，因此存取只是一个 简单的offset运算，就如单一继承一样简单。不管是经由一个指针，一个引用还是 一个object来存取。

虚拟继承:多重继承的一个语意上的副作用是，它必须支持某种形式的“shared subobject继承”。在菱形继承中，为保证基类的subobject只有一份，引入了虚拟继承。一般的方法如下:class内如果含有一个或者多个虚基类suboject，将分割为两部分，一个不变区和一个共享区。不变区中的数据，不论后继如何变化，总是拥有固定的offset，所以这一部分数据可以直接存取。至于共享区，所表现的就是virtual base class subobject。这一部分的数据，其位置会因为每次派生的操作而有变化。所以它们只可以被间接存取。以下说明实现虚拟继承的策略。

先安排好derived class的不变部分，再建立其共享部分。为存取class的共享部分，编译器会在每一个derived class object中安插一些指针，每个指针可以指向一个virtual base class。要存取继承的来的virtual base subobject members，可以通过相关指针间接完成。**如下图**：



这个模型有两个缺点:

1、首先是每个对象必须对其每一个virtual base class背负一个额外指针。

2、其次，由于虚拟继承串链的加长，导致间接存取层次增加。

至于第一个问题，一般而言有两种解决方法，Microsoft编译器引入所谓的Virtual

Base class table.每一个class object如果有一个或者多个virtual base classes，就会由编译器安排一个指针，指向virtual base classe table。至于真正的virtual base class

指针，当然会被放在表格中。第二个解决方法，是在virtual function table中放置virtual base class 的 offset(而不是地址)。

一般而言，virtual base class 最有效的一种运用形式是，一个抽象的virutal base class，没有任何data members。

**3.5对象成员的效率**

…跳过

**3.6指向Data Member的指针**

指向data member的指针，是一个神秘但是颇有用处的语言特性，特别是如果需要调查class members的底层布局的话。

但事实上，如果去取data members的地址，传回值却总是多1。理由是，为了区分一个“没有指向任何data member的指针”和一个指向“第一个data member的指针”。

比如下例

Float Point3d::\*p1 = 0;(没有指向任何data member，且地址为0)

Float Point3d::\*p2 = &Point3d::x;(指向了data member，原本的偏移量也是0)

(Point3d::\*的意思是，指向Point3d data member的指针类型)

要如何区分它们呢？---答案就如上述所言。

因此，在真正使用该值指出一个member之前，请先减去1。

认识指向data member的指针后，要解释:

&Point3d::z;和&origin.z间的差异就很明确了。鉴于“取得一个nonstatic data member“的地址，将会得到它在类中的偏移量。即前者；而”取一个绑定在真正的class object上的data member地址，将会得到该member在内存中的真正地址”，即后者。且前者返回的是float Point3d::\*,而后者返回的是float \*。

而把&origin.z的结果减去z的偏移量(即&Point3d::z)，然后加上1，就会得到origin的起始地址啦！

**指向members的指针的效率问题:**

跳过

1. **Function 语意学**

**4.1member的各种调用方式**

**Nonstatic Member Functions**:C++的设计准则之一是，nonstatic member function至少必须和一般的nonmember function有相同的效率。实际上，为此，member function 会被转化为nonmember的形式！

名称的特殊处理:一般而言，member的名称前会被加上class名称，形成独一无二的命名。比如class Bar{public:int ival;…}，其中的ival在name mangling后可能变为ival\_3bar.这样做的原因是，经过派生操作，可以将member绝对清楚地指出来，目前，编译器对name mangling还没有统一标准。

**Virtual Member Functions**:如果normalize()是一个virtual member function，那么以下调用

\*ptr->normalize()

将会被内部转换为

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

其中vptr是编译器产生的指针，指向虚函数表。事实上其名称也会被mangled，因为在一个复杂的派生系统中，可能存在多个vptrs。

而上述的1是virtual table slot的索引，关联到normalize()函数

第二个ptr表示this指针。

**Static Member Functions:**如果是静态成员函数，调用操作将会转换为一般的nonmember函数调用。事实上，只有当一个或者多个nonstatic data members在member function中被直接存取时，才需要class object。Class object提供了this指针用于这种形式的函数调用。这个this指针把“在member function中存取的nonstatic class members”绑定于“object内对应的members”之上。如果没有任何一个member被直接存取，事实上就不需要this指针，因此也就没有必要通过一个class object来调用一个member function。不过C++到目前为止并不能辨识这种情况。

这么一来，在存取static data members时产生了一些不规则性。如果class的设计者把static data member声明为nonpublic(这是好习惯)，那么就必须提供一个或者多个member functions来存取该member。因此，尽管可以不依靠class object来存取一个static member，但其存取函数却得绑定于一个class object。独立于class object之外得操作，在某个时候特别重要----当class得设计者希望支持“没有class object存在”的情况。

解决之道，即是由cfront2.0引入的static member functions。其特征在于并没有this指针，以下的次要特性统统根源于主特性

1. 不能够直接存取其class中的nonstatic members
2. 不能够被声明为const、volatile或virtual（const修饰符用于表示函数不能修改成员变量的值，该函数必须是含有this指针的类成员函数，函数调用方式为thiscall，而类中的static函数本质上是全局函数，调用规约是\_\_cdecl或\_\_stdcall,不能用const来修饰它）
3. 不需经由class object才被调用----虽然大部分时候它是这样被调用的！

**4.2虚拟成员函数**

在C++中，多态表示“以一个public base class的指针或引用，寻址处一个derived class object”的意思。

多态机能主要扮演一个输送机制的角色，经由它，我们可以在程序的任何地方采取一组public derived类型。这种多态形式被称为是“消极的”，可以在编译时期完成----virtual base class的情况除外。如下:

Point \*ptr;

Ptr = new Point2d;

当被指出的对象真正被使用时，多态也就变成积极的了。如下:

Ptr->z();

那么，什么信息才能让我们在执行期调用正确的实例？需要知道：

1. ptr所指对象的真实类型，这可以选择正确的z()实例
2. z()实例的位置，以便可正确调用之

为了实现之，第一点引入了一个字符串或数字，用于表示class的类型。

第二点则引入了虚函数表的概念，为了寻找表格，每个class object都被安插了一个由编译器内部产生的指针，指向该表格；为了找到函数地址，每一个虚函数又被指派了一个表格索引值。这些工作都由编译器完成，执行期所要做的，只是在特定的virtual table slot(记录着虚函数的地址)中激活virtual functions。

一个class只会有一张虚函数表。每一个表内含其对应之class object中所有active virtual function 函数实例的地址。这些active virtual function包括:

1. 这一类所定义的函数实例。其会改写一个可能存在的基类虚函数函数实例。
2. 继承自基类的函数实例。这是在derived class决定不改写virtual function时才会出现的情况
3. 一个pure\_virtual\_call()函数实例，它既可以扮演pure virtual function的空间保卫角色，也可以当做执行期异常处理函数(有时候会用到)

每一个虚函数都被指派一个固定的索引值，这个索引在整个继承体系中保持与特定的虚函数的关系。

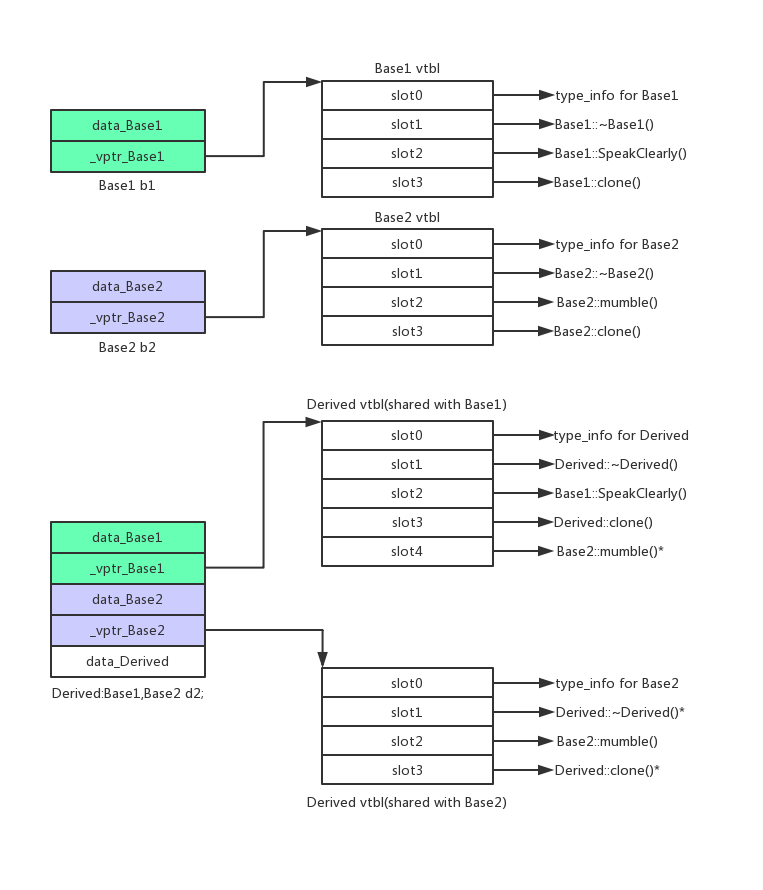
那么，回到ptr->z()的例子，到底其在编译时期是如何调用的呢？

1. 一般而言，每次调用z()时，并不知道ptr所指向对象的真正类型。但却直到ptr可以存取到对象的virtual table
2. 虽然不知道哪一个z()函数实例会被调用，但却直到每一个z()函数地址都会放在slot4中

这些信息可以使得编译器将之转化为:

( \*ptr->vptr[ 4 ] )( ptr );

在这一转化中，vptr表示编译器所安插的指针，指向virtual table;4表示z()被指派的slot编号。唯一一个在执行期才能知道的东西是，slot所指的到底是哪一个z()函数实例。

**多重继承下的virtual functions:**在多重继承之下，一个derived class内含n-1个额外的virtual tables，n表示其上一层的base classes的个数(因此，单一继承将不会由额外的virtual tables)。针对每一个virtual tables，derived对象中有对应的vptr。为了调节执行期链接器的效率，一些编译器把多个virtual tables连锁为一个。指向次要表格的指针，可以由主要表格名称加上offset而来。在这样的策略下，每一个class只有一个具名的virtual table。

**多重继承下的虚表布局如下**

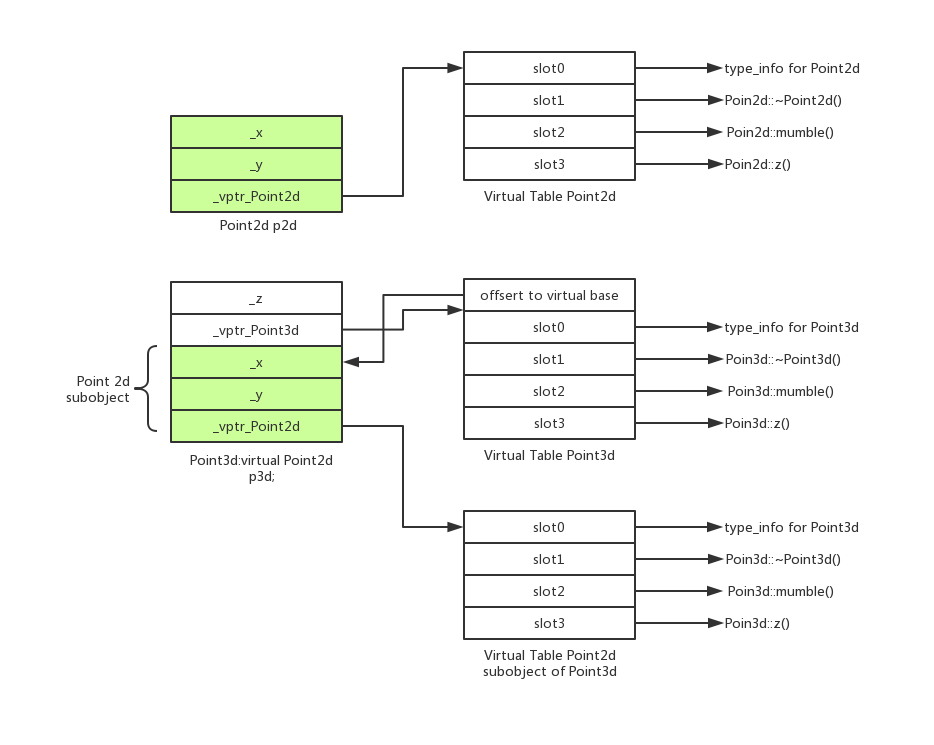
**注：**

1.子类虚函数会覆盖每一个父类的每一个同名虚函数。

2.父类中没有的虚函数而子类有(即子类新增的虚函数)，填入第一个虚函数表中，且用父类指针是不能调用。

3.父类中有的虚函数而子类没有，则不覆盖。仅子类和该父类指针能调用。

**虚拟继承下的Virtual functions**:注意，尽量不要在一个虚基类中声明nonstatic data members。如果这么做，会离复杂的深渊越来越近！

****

虚拟继承下的虚表布局如下:

**4.3函数效能**

略

**4.4指向member function的指针**

取一个nonstatic member function的地址，如果该函数是nonvirtual，得到的结果是它在内存中的真正地址。然而这个值也不是完全的。它需要被绑定于某个class object的地址上，才能够通过它调用函数。所有的nonstatic member functions都需要对象的地址。

比如，一个指向member function的指针，其声明如下:

Double(

Point::\*

pmf

)();

然后可以这样定义并初始化该指针:

Double (Point::\*coord)() = &Point::x;

也可以这样指定其值:

Coord = &Point::y;

欲调用它，可以这么做:

(origin.\*coord)();

或

(ptr->\*coord)();

指向member function的指针声明语法，以及指向member selection运算符的指针，其作用是作为this指针的空间保留者。这也就是为什么static member function(没有this指针)的类型是”函数指针”，而不是”指向member function的指针”之缘故。

**支持“指向Virtual Member Function”的指针:**对一个nonstatic member function取地址，得到的是该函数在内存中的地址。而对一个virtual function取地址，其地址在编译期是未知的，所能知道的仅是virtual function在其相关的virtual table中的索引值，也就是说，对一个虚成员函数取地址，所能获得的仅是一个索引值。

**在多重继承下，指向member function的指针**:许多编译器在自身内部根据不同的classes提供多种指向member functions的指针形式。例如Microsoft就提供了三种风味:

1. 一个单一继承实例(其中持有vcall thunk地址或者是函数地址)
2. 一个多重继承实例(其中持有faddr和delta两个members)
3. 一个虚拟继承实例(其中持有4个members)

**4.5Inline Functions**

关键词inline只是一种请求。如果这个请求被接受，编译器就必须认为它可以用一个表达式合理地将这个函数扩展开来。所谓合理即指，其执行成本比一般的函数调用返回机制来的负荷低。

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”intrinsic inline ability”(本质的inline能力)。“intrinsic”在这里意指”与编译器有关”。
2. 真正的inline函数扩展操作是在调用的那一点上。这回带来参数的求值操作以及临时性对象的管理。

**形式参数:**在inline扩展期间，每一个形式参数都会被对应的实际参数取代。如果说有什么副作用，那就是不可以简单地一一封塞程序中出现的每一个形式参数，因为这会导致对于实际参数的多次求值操作。一般而言，面对“会带来副作用的实参”，通常需要引入临时性对象。换句话说，如果实参是一个常量表达式，我们可以在替换前完成其求值操作;后继的inline替换，就可以直接把常量”绑”上去，如果既不是个常量表达式，也不是个有副作用的表达式，那就直接替换之。

举例，假设有如下inline函数:

Inline int

Min(int i, int j)

{

Return i<j ? I : j;

}

下面有三个调用操作

Inline int

Bar(){

Int minval;

Int val1 = 1024;

Int val2 = 2048;

/\*(1)\*/Minval = min(val1,val2);

/\*(2)\*/Minval = min(1024,2048);

/\*(3)\*/Minval = min(foo,bar()+1);

Return minval;

}

标记为(1)的那一行会被扩展为:

Minval = val1 < val2 ? val1:val2;

标记为(2)的那一行直接拥抱常量:

Minval = 1024;

标记为(3)的那以后则引发参数的副作用，需要引入临时对象，以避免重复求值

Int t1;

Int t2;

Minval = (t1 = foo()),(t2 = bar() +1)，

t1 < t2? t1 : t2;

**局部变量**:一般而言，inline函数中的每一个局部变量都必须放在函数调用的一个封闭区段中，拥有一个独一无二的名称。如果inline函数以单一表达式扩展多次，则每次扩展都需要自己的一组局部变量。如果inline函数以分离的多个式子被扩展多次，那么只需要一组局部变量，就可以重复使用(因为它们被放在一个封闭区段中，有自己的scope).

如下:

Inline int

Min(int i, int j)

{

Int minval = I < j ? i : j;

Return minval;

}

{

Int local\_var;

Int minval;

//…

Minval = min(val1,val2);

}

其在编译器中可能是这个样子:

{

Int local\_var;

Int minval;

Int \_min\_lv\_minval;//将inline函数的局部变量处以mangling操作

Minval =

( \_min\_lv\_minval = val1 < val2? Val1 : val2),

\_min\_lv\_minval;

}

Inline函数中的局部变量，再加上有副作用的参数，可能会导致大量临时性对象的产生。特别是如果它以单一表达式被扩展多次的话，如下面操作：

Minval = min(val1,val2) + min(foo(),foo()+1);

Inline函数对于封装提供了一种必要的支持，可以有效存取封装于class中的nonpublic数据，它同时也是C程序中大量使用的#define的一个安全替代品---特别是如果宏中的参数有副作用的话。然而一个inline函数如果被调用太多次，会产生大量扩展码，使得程序大小暴涨。

一如前文所述，参数带有副作用，或是以一个单一表达式做多重调用，或是再inline函数中有多个局部变量，都会产生临时性对象！