C++之虚函数(Virtual Member Functions)

《深度探索 C++对象模型》是这样来说多态的：

> *在 C++ 中,多态表示"以一个 public base class 的指针（或引用），寻址出一个 derived class object"的意思。*

消极多态与积极多态

用基类指针来寻址继承类的对象，我们可以这样：

```
Point ptr=new Point3d;                    //Point3d 继承自 Point
```

在这种情况下，多态可以在编译期完成（虚基类情况除外），因此被称作消极多态（没有进行虚函数的调用）。相对于消极多态，则有积极多态——指向的对象类型需要在执行期在能决定[1]。积极多态的例子如虚函数和 RTTI：

```
//例 1，虚函数的调用 ptr->z();//例 2，RTTI 的应用 if(Point3d
*p=dynamic_cast<Point3d*>(ptr) )
    return p->z();
```

关于 RTTI 的笔记可见笔记 EH & RTTI。本文主要精力将集中于虚函数上。对于一个如上例关于虚函数的调用，要如何来保证在执行期调用的是正确的 `z()`实体——Point3d::z() 而不是调用了 Point::z()。来看看虚函数的实现机制吧，它将保证这一点。

单继承下的虚函数

虚函数的实现：

- 为每个有虚函数的类配一张虚函数表，它存储该类类型信息和所有虚函数执行期的地址。
- 为每个有虚函数的类插入一个指针（vptr）,这个指针指向该类的虚函数表。
- 给每一个虚函数指派一个在表中的索引。

用这种模型来实现虚函数得益于在 C++中,虚函数的地址在编译期是可知的，而且这一地址是固定不变的。而且表的大小不会在执行期增大或减小。

一个类的虚函数表中存储有类型信息（存储在索引为 0 的位置）和所有虚函数地址，这些虚函数地址包括三种：

- 这个类定义的虚函数，会改写（overriding）一个可能存在的基类的虚函数实体——假如基类也定义有这个虚函数。
- 继承自基类的虚函数实体，——基类定义有，而这个类却没有定义。直接继承之。
- 一个纯虚函数实体。用来在虚函数表中占座，有时候也可以当做执行期异常处理函数。

每一个虚函数都被指派一个固定的索引值，这个索引值在整个继承体系中保持前后关联，例如,假如 `z()`在 Point 虚函数表中的索引值为 2，那么在 Point3d 虚函数表中的索引值也为 2。

当一个类单继承自有虚函数的基类的时候，将按如下步骤构建虚函数表：

1. 继承基类中声明的虚函数——这些虚函数的实体地址被拷贝到继承类中的虚函数表中对于的 slot 中。
2. 如果有改写（override）基类的虚函数，那么在 1 中应将改写（override）的函数实体的地址放入对应的 slot 中而不是拷贝基类的。
3. 如果有定义新的虚函数，那么将虚函数表扩大一个 slot 以存放新的函数实体地址。

我们假设 `z()`函数在 Point 虚函数表中的索引为 4，回到最初的问题——要如何来保证在执行期调用的是正确的 z()实体？其中微妙在于，编译将做一个小小的转换：

```
ptr->z();//被编译器转化为：(*ptr->vptr[4])(ptr);
```

这个转换保证了调用到正确的实体，因为：

- 虽然我们不知道 ptr 所指的真正类型,但它可以通过 vptr 找到正确类型的虚函数表。
- 在整个继承体系中 z()的地址总是被放在 slot 4。

多重继承下的虚函数

在多重继承下，继承类需要为每一条继承线路维护一个虚函数表（也有可能这些表被合成为一个，但本质意义并没有变化）。当然这一切都发生在需要的情况下。

当使用第一继承的基类指针来调用继承类的虚函数的时候，与单继承的情况没有什么异样，问题出生在当以第二或后继的基类指针（或引用）的使用上。例如：

```
//假设有这样的继承关系：class Derived:public base1,public base2;//base1,base2 都定义有虚析构函数。base2 *ptr = new derived;//需要被转换为，这个转换在编译期完成 base2 *ptr = temp ? temp + sizeof(base1) : 0;
```

如果不做出上面的转换，那么 ptr 指向的并不是 derived 的 base2 subobject 。后果是，ptr 将一个 derived 类型当做 base2 类型来用。

当要 `delete ptr` 时又面临了一次转换，因为在 `delete ptr` 的时候，需要对整个对象而不是其子对象施行 delete 运算符，这期间需要调整 ptr 指向完整的对象起点，因为不论是调用正确的析构函数还是 delete 运算符都需要一个指向对象起点的指针，想一想给予一个 derived 类的成员函数指向 base2 subobjuect 的 this 指针会发生什么吧。因为 ptr 的具体类型并不知道，所以必须要等到执行期来完成。

Bjame 的解决方法是将每一个虚函数表的 slot 扩展，以使之存放一个额外的偏移量。于是虚函数的调用：

```
(*ptr->vptr[1])(ptr);//将变成：(*ptr->vptr[1].addr)(ptr+*ptr->vptr[1].offset);
```

其中使用 `ptr->vptr[1].addr` 用以获取正确的虚函数地址，而 `ptr+*ptr->vptr[1].offset` 来获得指向对象完整的起点。这种方法的缺点显而易见，代价过大了一点，所有的情况都被这一种占比较小的情况拖累。

还有一种叫做 thunk 的方法，thunk 的作用在于：

1. 以适当的 offset 值来 this 调整指针.
2. 跳到虚函数中去。

Thunk 技术即是：虚函数表中的 slot 仍然继续放一个虚函数实体地址，但是如果调用这个虚函数需要进行 this 调整的话,该 slot 中的地址就指向一个 Thunk 而不是一个虚函数实体的地址。

书中纷杂的讲到不少中种情况，但我以我的理解，做如下小结：

多继承下的虚函数，影响到虚函数的调用的实际质上为 this 的调整。而 this 调整一般为两种：

1. 调整指针指向对应的 subobject，一般发生在继承类类型指针向基类类型指针赋值的情况下。
2. 将指向 subobject 的指针调整回继承类对象的起始点，一般发生在基类指针对继承类虚函数进行调用的时候。

第一点，使得该基类指针指向一个与其指针类型匹配的子对象，唯有如此才能保证使得该指针在执行与其指针类型相匹配的特定行为的正确性。比方调用基类的成员，获得正确的虚函数地址。可以想象如果不调整，用 ptr 存取 base2 subobject 的数据成员时，会发生什么？调用 base2 的成员函数的时候，其成员函数接受的 this 指针指向 derived 类型对象，这

又会发生什么？结果是整个对象的内存结构有可能都被破坏。还有别忘了，vptr 也可以看做一个数据成员，要找到虚函数，前提是获取正确的 vptr 偏移量。

而第二点，显然是让一个继承类的虚函数获取一个正确的 this 指针，因为一个继承类虚函数要的是一个指向继承类对象的 this 指针，而不是指向其子对象。

第一顺序继承类之所以不需要进行调整的关键在于，其 subobject 的起点与继承类对象的起点一致。

虚拟继承下的虚函数

Lippman 说，如果一个虚基类派生自另一虚基类，而且它们都支持虚函数和非静态数据成员的时候，编译器对虚基类的支持就像迷宫一样复杂。其实我原想告诉他，我是怀着一颗勇

士之心而来的　　。

虽然书中没有介绍太多，但不难猜测的是在虚继承情况下，复杂点在仍旧在于 this 指针的调整，然而其复杂度显然又在多继承之上，因为又多了一个 vbptr 了。

消极多态与积极多态之于我来说是一个新的概念，在《深度探索 C++对象模型》中也并没有给出明确的定义，在网上也没有看到关于它们对它们的明确定义。但我根据书中前后文及例子，推测 Lippman 所说的是这样的意思——根据是否为执行期多态来判断是积极多态还是消极多态。