C++类对象的大小

一个实例引出的思考

```
class X{};class Y:virtual public X{};class Z:virtual public X{};class A:public Y, public Z{};
```

猜猜 sizeof 上面各个类都为多少？

Lippman 的一个法国读者的结果是：

```
sizeof X yielded 1                    sizeof Y yielded 8
sizeof Z yielded 8                    sizeof A yielded 12
```

我在 vs2010 上的结果是：

```
sizeof X yielded 1   sizeof Y yielded 4   sizeof Z yielded 4   sizeof Z yielded 8
```

当我们对于 C++对象的内存布局知之甚少的情况下，想搞清这些奇怪现象的缘由将是一件非常困难的事情。不过下文会为你一一解惑。

事实上，对于像 X 这样的一个的空类，编译器会对其动点手脚——隐晦的插入一个字节。为 什么要这样做呢？插入了这一个字节，那么 X 的每一个对象都将有一个独一无二的地址。如果不插入这一个字节呢？哼哼，那对 X 的对象取地址的结果是什么？两个不同的 X 对象间 地址的比较怎么办？

我们再来看 Y 和 Z。首先我们要明白的是实现虚继承，将要带来一些额外的负担——额外需要 一个某种形式的指针。到目前为止，对于一个 32 位的机器来说 Y、Z 的大小应该为 5，而不 是 8 或者 4。我们需要再考虑两点因素：内存对齐（alignment—）和编译器的优化。

alignment[1] 会将数值调整到某数的整数倍，32 位计算机上位 4bytes。内存对齐可以 使得总线的运输量达到最高效率。所以 Y、Z 的大小被补齐到 8 就不足为奇了。

那么在 vs2010 中为什么 Y、Z 的大小是 4 而不是 8 呢？我们先思考一个问题，X 之所以被插入 1 字节是因为本身为空，需要这一个字节为其在内存中给它占领一个独一无二的地址。但 是当这一字节被继承到 Y、Z 后呢？它已经完全失去了它存在的意义，为什么？因为 Y、Z 各 自拥有一个虚基类指针，它们的大小不是 0。既然这一字节在 Y、Z 中毫无意义，

那么就没 必要留着。也就是说 vs2010 对它们进行了优化，优化的结果是去掉了那一个字节,而 Lippman 的法国读者的编译器显然没有做到这一点。

当我们现在再来看 A 的时候，一切就不是问题了。对于那位 Lippman 的法国读者来说，A 的 大小是共享的 X 实体 1 字节,X 和 Y 的大小分别减去虚基类带来的内存空间，都是 4。A 的总计 大小为 9，alignment 以后就是 12 了。而对于 vs2010 来说，那个一字节被优化后，A 的大小 为 8，也不需再进行 alignment 操作。

总结

影响 C++类的大小的三个因素：

- 支持特殊功能所带来的额外负担（对各种 virtual 的支持）。
- 编译器对特殊情况的优化处理。
- alignment 操作，即内存对齐。

关于更多的 memory alignment（内存对齐）的知识见 VC 内存对齐准则（Memory alignment） ↩