VC 内存对齐准则（Memory alignment）

本文所有内容在建立在一个前提下：使用 VC 编译器。着重点在于：VC 的内存对齐准则；同样的数据，不同的排列有不同的大小，另外在有虚函数或虚拟继承情况下又有如何影响？

内存对齐？！What？Why?

对于一台 32 位的机器来说如何才能发挥它的最佳存取效率呢？当然是每次都读 4 字节（32bit），这样才可以让它的 bus 处于最高效率。实际上它也是这么做的，即使你只需要一个字节，它也是 读一个机器字长（这儿是 32bit）。更重要的是，有的机器在存取或存储数据的时候它要求数据 必须是对齐的，何谓对齐？它要求数据的地址从 4 的倍数开始，如若不然，它就报错。还有的机 器它虽然不报错，但对于一个类似 int 变量，假如它横跨一个边界的两端，那么它将要进行两次 读取才能获得这个 int 值。比方它存储在地址为 2~5 的四个字节中，那么要读取这个 int，将要 进行两次读取，第一次读取 0~3 四个字节，第二次读取 4~7 四个字节。但是如果我们把这个整 形的起始地址调整到 0,4,8…呢？一次存取就够了！这种调整就是内存对齐了。我们也可以依次 类推到 16 位或 64 位的机器上。

边界该如何调整

对于 32 位的机器来说，它当然最渴望它的数据的大小都是 4 Byte 或者 4 的倍数 Byte，这样它就能 最有效率的存取数据，当然如果数据小于 4Byte,那也是没问题的。那么编译器要做的便是尽量满 足这个要求。

这两天我断续对 VC 做了一些实验，并总结如下三条准则，你要明白的是这并非来自微软的官方文 档，但我自以为这些准则或许不全但应该都是正确的：

- 变量存放的起始位置 [2] 应为变量的大小与规定对齐量 [1] 中较小者的倍数。例如，假 设规定对齐量为 4，那么 char（1byte）变量应该存储在偏移量为 1 的倍数的地方，而整形变 量（4byte）则是从偏移量为 4 的倍数的地方，而 double（8 byte）也同样应存储在偏移量为 4 的倍数的地方，为什么不是 8？因为规定对齐量默认值为 4，而 4 < 8。在 VC 中**默认对齐量 为 8，而非 4**。
- 结构体整体的大小也应该对齐，对齐依照规定对齐量与最大数据成员两者中较小的进行。
- Vptr 影响对齐而 VbcPoint(Virtual base class pointer)不影响。

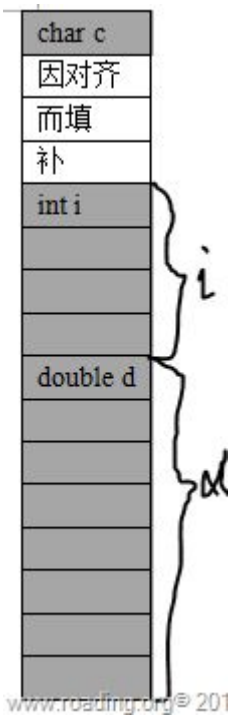
一个实例

对于类 T：

```
class T{
    char c;
    int i;
    double d;};
```

将其 sizeof 输出后的大小为 16，其内存布局如图 T.变量 c 从偏移量为 0 开始存储，而整形 i 第一个 符号条件的偏移量为 4，double 型 d 的第一个符号条件的为 8。整个对象的大小为 16，不需要再进 行额外的对齐。

图 T（类 T 的内存布局）：



同样的数据，不同的大小

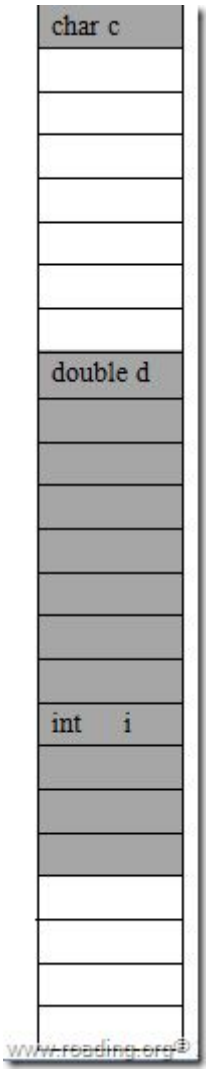
再看类 L,它与 T 存储同样类型的数据，仅仅是顺序不同罢了，那么它 sizeof 输出的大小是多少呢?

类 L:

```
class L{
    char c;
    double d;
    int i;};
```

它 sizeof 后的结果或许会令你大吃一惊，或许不会（如果你有认真读前面的两条准则）。L sizeof 后的结果是 24！同样是一个 int，一个 char，一个 double 却整整多出了 8 个字节。这期间 发生了什么？我们依据前面两条规则来看看。C 存储于 0 的位置，1~7 都不能整除 8，所以 d 存储 在 8~15，16 给 i 正好合适，i 存储在 16~19。总共花费了 20 个字节，抱歉不是 8 的倍数，还得补 齐 4 个。现在你可以看看图 L 的关于类 L 的内存布局，再比较一下类 L 和类 T 的内存布局。

图 L(类 L 的布局)



我得出了这样一条并不权威的结论，因为我还没听有人这样说过：在声明数据成员的时候，将 最大字节数的变量放在最前面 [3],切忌不要将大小差距很大的类型交替声明。

Vptr 影响对齐而 VbcPoint(Virtual base class pointer) 不影响

前面的实例只涉及前两条准则，现在我们来看看第三条的两个实例：

```
class X{char a;};class Y: virtual public X{};
```

Y 的大小为:a 占一个字节，VbcPoint（我称他为虚基类指针）占四个字节。我们不论 a 与 VbcPoint 的位置如何摆放，如果将 VbcPoint 等同于一个成员数据来看的话，sizeof(Y) 都应该为 8.实际上 它是 5！就我目前的水平，我只能先将其解释为 VbcPoint 不参与对齐。

对于 vptr 这个问题则不存在：

```
class X{
    char a;
    virtual int vfc(){};}
```

sizeof（X）的大小确实为 8.

关于#pragma pack(n)

用#pragma pack(n)改变规定对齐量试试  。

> 规定对齐量：实际上并没有这么一个名词，是我为了方便而造出来的。在 VC 中这个"规定对齐量"会有一个默认值，这个默认值一般为 8，我原来一直以为这个值以为是 4，至于它为什么为 8，我现在还不知道。。我们也可以通过#pargma pack(n)来规定这个值，目前 n 可以为 1,2,4,8,16。 ↩
>
> 这个起始位置指的是相对于结构体（类）来说的。 ↩

此处有一点问题，这个问题由独酌逸醉提出，他认为将最小的数据放在最前面可能会更好，我们有进行过讨论，但可惜的是由于在 2011/11/24 日数据库丢失，我只能用备份还原，所以丢失了一些数据，无疑，本文的评论也在其中。不过我对这个问题映像深刻，因为我在写这篇博客的时候便困惑于到底成员是应该放在之前还是之后，因为这两种情况我都找不到强有力的理由来支撑它们。后来使我确信从大到小排列好于从小到大排列的理由在于，从大到小排列一般无需成员之间的对齐，唯一的对齐工作是最后进行的整个结构体对齐的工作。毫无疑问的是，这应该是最节省内存的方式。再之后，独酌提出从小到大可能好些，虽然没有给出有说服力的理由，但却使我无比困惑，我当时虽然认为从大到小的排列更有优势，但却实在想不出一个实例能使得它优于从小到大排列的。不过最终我击垮了自己的理由，在继承状况下从大到小排列很容易被打破，比方，基类的成员为一个 char,继承类的成员为 double,int,char 虽然基类和继承类都是按从大到小的顺序排列的，但是继承类的内存布局最终会使 char,double,int,char，此时既不能避免成员对齐，又导致后面的结构体对齐。暂时获得的最终结果是从小到大排列是更好的一种排列方式。（2011/12/31 增补） ↩