结构体字节对齐

3.1.2 对齐准则

先来看四个重要的基本概念：

1) 数据类型自身的对齐值：char型数据自身对齐值为1字节，short型数据为2字节，int/float型为4字节，double型为8字节。

2) 结构体或类的自身对齐值：其成员中自身对齐值最大的那个值。

3) 指定对齐值：#pragma pack (value)时的指定对齐值value。

4) 数据成员、结构体和类的有效对齐值：自身对齐值和指定对齐值中较小者，即有效对齐值=min{自身对齐值，当前指定的pack值}。

基于上面这些值，就可以方便地讨论具体数据结构的成员和其自身的对齐方式。

其中，有效对齐值N是最终用来决定数据存放地址方式的值。有效对齐N表示"对齐在N上"，即该数据的"存放起始地址%N=0"。而数据结构中的数据变量都是按定义的先后顺序存放。第一个数据变量的起始地址就是数据结构的起始地址。结构体的成员变量要对齐存放，结构体本身也要根据自身的有效对齐值圆整(即结构体成员变量占用总长度为结构体有效对齐值的整数倍)。

以此分析3.1.1节中的结构体B：

假设B从地址空间0x0000开始存放，且指定对齐值默认为4(4字节对齐)。成员变量b的自身对齐值是1，比默认指定对齐值4小，所以其有效对齐值为1，其存放地址0x0000符合0x0000%1=0。成员变量a自身对齐值为4，所以有效对齐值也为4，只能存放在起始地址为0x0004~0x0007四个连续的字节空间中，符合0x0004%4=0且紧靠第一个变量。变量c自身对齐值为 2，所以有效对齐值也是2，可存放在0x0008~0x0009两个字节空间中，符合0x0008%2=0。所以从0x0000~0x0009存放的都是B内容。

再看数据结构B的自身对齐值为其变量中最大对齐值(这里是b)所以就是4，所以结构体的有效对齐值也是4。根据结构体圆整的要求， 0x0000~0x0009=10字节，(10+2)%4=0。所以0x0000A~0x000B也为结构体B所占用。故B从0x0000到0x000B 共有12个字节，sizeof(struct B)=12。

之所以编译器在后面补充2个字节，是为了实现结构数组的存取效率。试想如果定义一个结构B的数组，那么第一个结构起始地址是0没有问题，但是第二个结构呢？按照数组的定义，数组中所有元素都紧挨着。如果我们不把结构体大小补充为4的整数倍，那么下一个结构的起始地址将是0x0000A，这显然不能满足结构的地址对齐。因此要把结构体补充成有效对齐大小的整数倍。其实对于char/short/int/float/double等已有类型的自身对齐值也是基于数组考虑的，只是因为这些类型的长度已知，所以他们的自身对齐值也就已知。

上面的概念非常便于理解，不过个人还是更喜欢下面的对齐准则。

结构体字节对齐的细节和具体编译器实现相关，但一般而言满足三个准则：

1) 结构体变量的首地址能够被其最宽基本类型成员的大小所整除；

2) 结构体每个成员相对结构体首地址的偏移量(offset)都是成员大小的整数倍，如有需要编译器会在成员之间加上填充字节(internal adding)；

3) 结构体的总大小为结构体最宽基本类型成员大小的整数倍，如有需要编译器会在最末一个成员之后加上填充字节(trailing padding)。

对于以上规则的说明如下：

第一条：编译器在给结构体开辟空间时，首先找到结构体中最宽的基本数据类型，然后寻找内存地址能被该基本数据类型所整除的位置，作为结构体的首地址。将这个最宽的基本数据类型的大小作为上面介绍的对齐模数。

第二条：为结构体的一个成员开辟空间之前，编译器首先检查预开辟空间的首地址相对于结构体首地址的偏移是否是本成员大小的整数倍，若是，则存放本成员，反之，则在本成员和上一个成员之间填充一定的字节，以达到整数倍的要求，也就是将预开辟空间的首地址后移几个字节。

第三条：结构体总大小是包括填充字节，最后一个成员满足上面两条以外，还必须满足第三条，否则就必须在最后填充几个字节以达到本条要求。

参考链接：[C语言字节对齐问题详解]

[字节对齐（强制对齐以及自然对齐）]

注意：GCC的最高只能被4整除，也就是说，long long 也是4个字节对齐