C 内存对齐

先想一个问题：

```
struct   stu{
char sex;
int length;
char name[10];
};
sizeof (struct   stu) = ???
```

如果你的答案是：15 。那你该仔细看看下面的分析了！！！！

1. 概念

对齐跟数据在内存中的位置有关。如果一个变量的内存地址正好位于它长度的整数倍，他就被称做自然对齐。比如在 32 位 cpu 下，假设一个整型变量的地址为 0x00000004，那它就是自然对齐的。

2. 为什么要字节对齐

需要字节对齐的根本原因在于 CPU 访问数据的效率问题。

计算机中数据的内存中的存储方式是自下往上的。

代码中定义：

```
int a =0x12345678 ;

char b=0x35;

int c=0xABCDEFFF;

char d=0x79 ;
```

理论上的内存分布图是这样的：

char 型指针	short 指针	int 型指针	内存	物理地址
19	9	4		19
18				18
17	8			17
16				16
15	7	3		15
14				14
13	6			13
12				12
11	5	2		11
10				10
9	4		0x79(char)	9
8			0XFF(int)	8
7	3	1	0XEF(int)	7
6			0XCD(int)	6
5	2		0xAB(int)	5
4			0x35(char)	4
3	1	0	0x78(int)	3
2			0x56(int)	2
1	0		0x34(int)	1
0			0x12(int)	0

CPU 在读取数据 a 时，可以直接使用 int 型指针，一次性读取完整的 0x12345678；然后用 char 型指针读取 b，也是一次性读取完整的 0x35 。

由于 0xABCDEFFF 跨越了两个 int 地址，CPU 在读取 c 时，无法直接用一个 int 型指针读取完整的 0xABCDEFFF。只能先使用一个 char 指针读取 0xAB，再用一个 short 指针读取一个 0xCDEF，再用一个 char 指针读取 0xFF。程序的执行效率可想而知了。

再想想，如果 c 后面定义的数据 d 也都是 int 型呢。计算机仅是读这些数据就要读疯了。。。

内存对齐就是要解决这个问题的。

3. 正确处理字节对齐

内存对齐就是把相应类型的数，放在一个相应指针能一次性读取玩的地方，比如上图中的 a 和 b 。

a 所占的4个字节正好在一个 int 型指针的范围内。b所占的1个字节正好在 一个char型指针的范围内。同理，c 就是典型的没对齐了。

在内存对齐的方式中，上面的 a、b、c、d 的内存分布应该是这样的

char 型指针	short 指针	int 型指针	内存	物理地址
19	9	4		19
18				18
17	8			17
16				16
15	7	3	填充字节	15
14			填充字节	14
13	6		填充字节	13
12			0x79(char)	12
11	5	2	0XFF(int)	11
10			0XEF(int)	10
9	4		0xCD(int)	9
8			0xAB(int)	8
7	3	1	填充字节	7
6			填充字节	6
5	2		填充字节	5
4			0x35(char)	4
3	1	0	0x78(int)	3
2			0x56(int)	2
1	0		0x34(int)	1
0			0x12(int)	0

虽然内存实际使用量远大于理论值，但是数据读取的效率却大大提高了。

这一点对于计算机和大内存手机来说无所谓，但对于 Cortex-M 单片机等小内存设备来说可能是致命的。他消耗了比预想要多的内存。

4. 如何避免因为内存对齐消耗过多内存

再来看看之前我们说过的一个例子：

```
struct {
    int a ;
    char b;
    int c;
    char d;
}test1;

struct {
    int a ;
    char b;
    char d;
    int c;

}test2;
```

test1 的内存分布：

char 型指针	short 指针	int 型指针	内存	物理地址
19	9	4		19
18				18
17	8			17
16				16
15	7	3	填充字节	15
14			填充字节	14
13	6		填充字节	13
12			0x78(char)	12
11	5	2	0XFF(int)	11
10			0XEF(int)	10
9	4		0xCD(int)	9
8			0xAB(int)	8
7	3	1	填充字节	7
6			填充字节	6
5	2		填充字节	5
4			0x35(char)	4
3	1	0	0x78(int)	3
2			0x56(int)	2
1	0		0x34(int)	1
0			0x12(int)	0

test2 的内存分布：

char 型指针	short 指针	int 型指针	内存	物理地址
19	9	4		19
18				18
17	8			17
16				16
15	7	3		15
14				14
13	6			13
12				12
11	5	2	0XFF(int)	11
10			0XEF(int)	10
9	4		0xCD(int)	9
8			0xAB(int)	8
7	3	1	填充字节	7
6			填充字节	6
5	2		0x78(char)	5
4			0x35(char)	4
3	1	0	0x78(int)	3
2			0x56(int)	2
1	0		0x34(int)	1
0			0x12(int)	0

test1 占用了 16 个字节，test2 只占用了 12 个字节。蚊子再小也是肉！

Keil 编译器自动 4 字节对齐的。在自己编写 malloc 函数时需要注意也要对齐。

5. 如何手动对齐

通常不是自己去手动实现 malloc 的话，不需要手动对齐。编译器都是自动对齐的。x86 平台下的编译器都是自动对齐的。x86 平台下也不会有人去手动实现 malloc 吧！

ARM Keil 默认的自动对齐的。由于 ARM 内存没有堆管理，在移植操作系统或文件系统时，可能需要手动实现 malloc。

ARM Keil 中使用修饰词 __align 。如指定定义的 char 数组 4 字节对齐：

```
__align(4)  char sample[] ;
```