**内存对齐**

<http://baike.baidu.com/item/%E5%86%85%E5%AD%98%E5%AF%B9%E9%BD%90>

内存对齐”应该是[编译器](http://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E5%99%A8" \t "_blank)的“管辖范围”。编译器为程序中的每个“[数据单元](http://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%8D%95%E5%85%83)”安排在适当的位置上。但是C语言的一个特点就是太灵活，太强大，它允许你干预“内存对齐”。如果你想了解更加底层的秘密，“内存对齐”对你就不应该再透明了。

### 对齐原因

大部分的参考资料都是如是说的：

1、平台原因(移植原因)：不是所有的硬件平台都能访问任意地址上的任意数据的；某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异常。

2、性能原因：数据结构(尤其是栈)应该尽可能地在自然边界上对齐。原因在于，为了访问未对齐的内存，处理器需要作两次内存访问；而对齐的内存访问仅需要一次访问。

### 内存对齐对齐规则

每个特定平台上的[编译器](http://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E5%99%A8" \t "_blank)都有自己的默认“对齐系数”(也叫对齐模数)。程序员可以通过[预编译](http://baike.baidu.com/item/%E9%A2%84%E7%BC%96%E8%AF%91" \t "_blank)命令#pragma pack(n)，n=1,2,4,8,16来改变这一系数，其中的n就是你要指定的“对齐系数”。

规则：

1、[数据成员](http://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E6%88%90%E5%91%98" \t "_blank)对齐规则：结构(struct)(或联合(union))的数据成员，第一个数据成员放在offset为0的地方，以后每个数据成员的对齐**按照#pragma pack指定的数值和这个数据成员自身长度中，比较小的那个进行**。

2、结构(或联合)的整体对齐规则：在数据成员完成各自对齐之后，结构(或联合)本身也要进行对齐，对齐将**按照#pragma pack指定的数值和结构(或联合)最大数据成员长度中，比较小的那个进行**。

3、结合1、2可推断：当#pragma pack的n值等于或超过所有[数据成员](http://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E6%88%90%E5%91%98)长度的时候，这个n值的大小将不产生任何效果。

Win32平台下的微软C[编译器](http://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E5%99%A8)([cl.exe](http://baike.baidu.com/item/cl.exe" \t "_blank)for 80×86)的对齐策略：

1)[结构体](http://baike.baidu.com/item/%E7%BB%93%E6%9E%84%E4%BD%93)变量的首地址是其最长基本类型成员的整数倍；

备注：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](http://baike.baidu.com/item/%E5%86%85%E5%AD%98%E5%9C%B0%E5%9D%80" \t "_blank)能是该基本数据类型的整倍的位置，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。

2)结构体每个成员相对于结构体首地址的[偏移量](http://baike.baidu.com/item/%E5%81%8F%E7%A7%BB%E9%87%8F" \t "_blank)（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）；

备注: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。

3)[结构体](http://baike.baidu.com/item/%E7%BB%93%E6%9E%84%E4%BD%93)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](http://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E5%99%A8)会在最末一个成员之后加上填充字节（trailing padding）。

备注：

a、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。

b、如果结构体内存在长度大于处理器位数的元素，那么就以处理器的倍数为对齐单位；否则，如果结构体内的元素的长度都小于处理器的倍数的时候，便以结构体里面最长的[数据元素](http://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%85%83%E7%B4%A0)为对齐单位。

4) 结构体内类型相同的连续元素将在连续的空间内，和[数组](http://baike.baidu.com/item/%E6%95%B0%E7%BB%84" \t "_blank)一样。

### 内存对齐验证试验

我们通过一系列例子的详细说明来证明这个规则吧!

我试验用的[编译器](http://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E5%99%A8" \t "_blank)包括GCC 3.4.2和VC6.0的C编译器，平台为Windows XP + Sp2。

我们将用典型的struct对齐来说明。首先我们定义一个struct：

#pragma pack(n) /\* n = 1, 2, 4, 8, 16 \*/

struct test\_t {

int a;

char b;

short c;

char d[6];

};

#pragma pack(n)

首先我们首先确认在试验平台上的各个类型的size，经验证两个[编译器](http://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E5%99%A8" \t "_blank)的输出均为：

sizeof(char) = 1

sizeof(short) = 2

sizeof(int) = 4

我们的试验过程如下：通过#pragma pack(n)改变“对齐系数”，然后察看sizeof(struct test\_t)的值。

1、1[字节对齐](http://baike.baidu.com/item/%E5%AD%97%E8%8A%82%E5%AF%B9%E9%BD%90)(#pragma pack(1))

输出结果：sizeof(struct test\_t) = 13[两个编译器输出一致]

分析过程：

1) 成员数据对齐

#pragma pack(1)

struct test\_t {

int a; /\* int型，长度4 > 1 按1对齐；起始offset=0 0%1=0；存放位置区间[0,3] \*/

char b; /\* char型，长度1 = 1 按1对齐；起始offset=4 4%1=0；存放位置区间[4] \*/

short c; /\* short型，长度2 > 1 按1对齐；起始offset=5 5%1=0；存放位置区间[5,6] \*/

char d[6]; /\* char型，长度1 = 1 按1对齐；起始offset=7 7%1=0；存放位置区间[7,C] \*/

};/\*char d[6]要看成6个char型变量\*/

#pragma pack()

成员总大小=13

2) 整体对齐

整体对齐系数 = min((max(int,short,char), 1) = 1

整体大小(size)=$(成员总大小) 按 $(整体对齐系数) 圆整 = 13 /\*13%1=0\*/ [注1]

2、2[字节对齐](http://baike.baidu.com/item/%E5%AD%97%E8%8A%82%E5%AF%B9%E9%BD%90)(#pragma pack(2))

输出结果：sizeof(struct test\_t) = 14 [两个[编译器](http://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E5%99%A8" \t "_blank)输出一致]

分析过程：

1) 成员数据对齐

#pragma pack(2)

struct test\_t {

int a; /\* int型，长度4 > 2 按2对齐；起始offset=0 0%2=0；存放位置区间[0,3] \*/

char b; /\* char型，长度1 < 2 按1对齐；起始offset=4 4%1=0；存放位置区间[4] \*/

short c; /\* short型，长度2 = 2 按2对齐；起始offset=6 6%2=0；存放位置区间[6,7] \*/

char d[6]; /\* char型，长度1 < 2 按1对齐；起始offset=8 8%1=0；存放位置区间[8,D] \*/

};

#pragma pack()

成员总大小=14

2) 整体对齐

整体对齐系数 = min((max(int,short,char), 2) = 2

整体大小(size)=$(成员总大小) 按 $(整体对齐系数) 圆整 = 14 /\* 14%2=0 \*/

3、4[字节对齐](http://baike.baidu.com/item/%E5%AD%97%E8%8A%82%E5%AF%B9%E9%BD%90)(#pragma pack(4))

输出结果：sizeof(struct test\_t) = 16 [两个[编译器](http://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E5%99%A8" \t "_blank)输出一致]

分析过程：

1) 成员数据对齐

#pragma pack(4)

struct test\_t {

int a; /\* int型，长度4 = 4 按4对齐；起始offset=0 0%4=0；存放位置区间[0,3] \*/

char b; /\* char型，长度1 < 4 按1对齐；起始offset=4 4%4=0；存放位置区间[4] \*/

short c; /\*short型， 长度2 < 4 按2对齐；起始offset=8 8%4=0；存放位置区间[8,9] \*/

char d[6]; /\* char型，长度1 < 4 按1对齐；起始offset=C C%4=0；存放位置区间[0C,10] \*/

};

#pragma pack()

成员总大小=14

2) 整体对齐

整体对齐系数 = min((max(int,short,char), 4) = 4

整体大小(size)=$(成员总大小) 按 $(整体对齐系数) 圆整 = 16 /\*16%4=0\*/

4、8[字节对齐](http://baike.baidu.com/item/%E5%AD%97%E8%8A%82%E5%AF%B9%E9%BD%90)(#pragma pack(8))

输出结果：sizeof(struct test\_t) = 16 [两个[编译器](http://baike.baidu.com/item/%E7%BC%96%E8%AF%91%E5%99%A8" \t "_blank)输出一致]

分析过程：

1) 成员数据对齐

#pragma pack(8)

struct test\_t {

int a; /\* int型，长度4 < 8 按4对齐；起始offset=0 0%4=0；存放位置区间[0,3] \*/

char b; /\* char型，长度1 < 8 按1对齐；起始offset=4 4%1=0；存放位置区间[4] \*/

short c; /\* short型，长度2 < 8 按2对齐；起始offset=6 6%2=0；存放位置区间[6,7] \*/

char d[6]; /\* char型，长度1 < 8 按1对齐；起始offset=8 8%1=0；存放位置区间[8,D] \*/

};

#pragma pack()

成员总大小=14

2) 整体对齐

整体对齐系数 = min((max(int,short,char), 8) = 4

整体大小(size)=$(成员总大小) 按 $(整体对齐系数) 圆整 = 16 /\*16%4=0\*/

5、16[字节对齐](http://baike.baidu.com/item/%E5%AD%97%E8%8A%82%E5%AF%B9%E9%BD%90)(#pragma pack(16))

输出结果：sizeof(struct test\_t) = 16 [两个编译器输出一致]

分析过程：

1) 成员数据对齐

#pragma pack(16)

struct test\_t {

int a; /\* int型，长度4 < 16 按4对齐；起始offset=0 0%4=0；存放位置区间[0,3] \*/

char b; /\* char型，长度1 < 16 按1对齐；起始offset=4 4%1=0；存放位置区间[4] \*/

short c; /\* short型，长度2 < 16 按2对齐；起始offset=6 6%2=0；存放位置区间[6,7] \*/

char d[6]; /\* char型，长度1 < 16 按1对齐；起始offset=8 8%1=0；存放位置区间[8,D] \*/

};

#pragma pack()

成员总大小=14

2) 整体对齐

整体对齐系数 = min((max(int,short,char), 16) = 4

整体大小(size)=$(成员总大小) 按 $(整体对齐系数) 圆整 = 16 /\*16%4=0\*/

### 内存对齐基本结论

8字节和16[字节对齐](http://baike.baidu.com/item/%E5%AD%97%E8%8A%82%E5%AF%B9%E9%BD%90)试验证明了“规则”的第3点：“当#pragma pack的n值等于或超过所有数据成员长度的时候，这个n值的大小将不产生任何效果”。另外内存对齐是个很复杂的东西，读者不妨把上述[结构体](http://baike.baidu.com/item/%E7%BB%93%E6%9E%84%E4%BD%93" \t "_blank)中加个double型成员进去练习一下，上面所说的在有些时候也可能不正确。呵呵^\_^

[注1]

什么是“圆整”？

举例说明：如上面的8字节对齐中的“整体对齐”，整体大小=9 按 4 圆整 = 12

圆整的过程：从9开始每次加一，看是否能被4整除，这里9，10，11均不能被4整除，到12时可以，则圆整结束。

上面文字表述太不直观了，鄙人给段代码直观的体现出来，代码如下：

#pragma pack(4) /\* n = 1, 2, 4, 8, 16 \*/  
　　struct test\_t{  
　　int a;  
　　char b;  
　　short c;  
　　char d[6];  
　　}ttt;  
　　void print\_hex\_data(char \*info, char \*data, int len)  
　　{  
　　int i;  
　　dbg\_printf("%s:\n\r", info);  
　　for(i = 0; i < len; i++){  
　　dbg\_printf("%02x ", (unsigned char)data[i]);  
　　if (0 == ((i+1) % 32))  
　　dbg\_printf("\n");  
　　}  
　　dbg\_printf("\n\r");  
　　}  
　　int main()  
　　{  
　　ttt.a = 0x1a2a3a4a;  
　　ttt.b = 0x1b;  
　　ttt.c = 0x1c2c;  
　　char \*s = "123456";  
　　memcpy(ttt.d, s, 6);  
　　print\_hex\_data("struct\_data", (char \*)&ttt, sizeof(struct test\_t));  
　　return 0;  
　　}

#pragma pack(1)的结果：

4a 3a 2a 1a 1b 2c 1c 31 32 33 34 35 36

#pragma pack(2)的结果：

4a 3a 2a 1a 1b 00 2c 1c 31 32 33 34 35 36

#pragma pack(4)的结果：

4a 3a 2a 1a 1b 00 2c 1c 31 32 33 34 35 36 00 00

#pragma pack(8)的结果：

4a 3a 2a 1a 1b 00 2c 1c 31 32 33 34 35 36 00 00

#pragma pack(16)的结果：

4a 3a 2a 1a 1b 00 2c 1c 31 32 33 34 35 36 00 00