Byte Alignment

许多计算机系统对基本数据类型的可允许地址做出了一些限制，要求某种类型的对象的地址必须是某个值k(通常是2、4、或8)的倍数。这种对齐限制简化了处理器和存储系统之间接口的硬件设计。例如，假设一个处理器总是从存储器中取8个字节出来，则地址必须为8的倍数。如果我们能保证所有的double类型数据的地址都是8的倍数，那么就可以用一个存储器操作来读或写值了。否则，我们可能需要执行两次存储器访问，因为对象可能分放在两个8字节存储器块中。没有对齐的数据会造成性能的降低，在一些系统（多数RISC芯片系统）中还会造成程序错误。即使在那些非对齐数据不会造成程序错误的系统中，由于性能的下降造成的影响也值得在任何可能的地方保持数据对齐。很多处理器对程序指令都有对齐的要求，多数RISC芯片要求指令必须对齐到4字节边界。

Misaligned

首先看一下未对齐的数据在不同的处理器上不同的访问原则，然后，我们会在 Example 一节中对此分别举例说明。

Sparc

从20世纪70年代末开始在UNIX上编程时，一定会对以下两个错误非常熟悉：

- bus error(core dumped)
- segmentation fault(core dumped)

这就是很长时间以来，当我们使用指针时总能轻易地遇到两种错误：总线错误和段错误。在SPARC上的SUNOS中出现这两个错误基本上都是当硬件告诉操作系统出现一个有问题的内存引用时，进程收到这两个错误的缺省操作就是进行信息转储并终止。

通常，段错误是因为不同的编程错误引起解除引用一个未初始化或非法值的指针而引起的。而神秘的总结错误则几乎都是由于对未对齐的内存进行访问造成地址总线的阻塞而引起的。因此，在SPARC结构的机器中访问未对齐数据是致命的。

如下面代码往往会导致一个总线错误：

```
char dog[10];
char *p = &dog[1];
unsigned long l = *(unsigned long *)p;
```

IA32

IA 体系结构中，并不要求数据在内存中以自然边界对齐(自然边界对齐是指，对于 Words 来说，自然边界为偶地址；而对于 Double Words 来说，自然边界为能够被 4 整除的地址；对于四字来说，自然边界则为能被 8 整除的地址)。但是，为了提高程序性能，数据结构(特别是栈)，要尽可能的在自然边界上对齐。原因就是，处理器访问一个自然边界对的数据仅

需一个内存接入周期，但访问一个未对齐数据则会需要两个内存接入周期。

《Intel Architecture Software Developer's Manual》:
Words, double words, and quad words do not need to be aligned in memory on natural boundaries.(The natural boundaries for words, double words, and quad words are even-numbered addresses, addresses evenly divisible by four, and addresses evenly divisible by eight, respectively.)However, to improve the performance of programs, data structures (especially stacks)should be aligned on natural boundaries whenever possible. The reason for this is that the processor requires two memory accesses to make an unaligned memory access; whereas, aligned accesses require only one memory access. A word or double word operand that crosses a 4-byte boundary or a quad word operand that crosses an 8-byte boundary is considered unaligned and requires two separate memory bus cycles to access it; a word that starts on an odd address but does not cross a word boundary is considered aligned and can still be accessed in one bus cycle.

不过并不是程序中的所有部份都需要4字节对齐，比如程序的代码就无需对齐，因为代码会经过预取及在cpu中排队（其实预取的时候一般都是一次取一批指令而且取的时候cpu在进行其它的流水操作），因此不对齐也不会对性能造成太大影响，下面这段解释了原因：

《INTEL 80386 PROGRAMMER'S REFERENCE MANUAL 1986》：
Due to instruction prefetching and queuing within the cpu, there is no requirement for instructions to be aligned on word or doubleword boundaries.(However, a slight increase in speed results if the target addresses of control transfers are evenly divisible by four.)

由此可见，无论对齐与否，IA32硬件总是能正确工作。不过，就如手册中写道，Intel还是建议要对齐数据以提高存储系统的性能。Linux沿用的对齐策略是2字节数据类型(如short)的地址必须是2的倍数，而较大的数据类型(如int, int *, float, double, etc.)的地址必须是4的倍数。

Microsoft Windows要求的对齐更严格：任何k字节(基本)对象的地址都必须是k的倍数。特别地，它要求一个double的地址应该是8的倍数。这种要求提高了存储，代价是浪费了一些空间。

在Linux中选项-malign-double/-mno-align-double，会使GCC为double类型的数据使用或不使用8字节对齐的地址。若使用了此选项，则需注意与4字节对齐方式下的库的兼容。

Arm

《ARM Architecture Referenece Manual》中提到：

在 ARM 中，通常希望字单元的地址是字对齐的(地址的低两位是 0b00)，半字单元的地址是对齐的(地址 0b0)。在存储访问操作中，如果存储单元的地址没有遵守上述的对齐规则，则称为非对齐(unaligned)的存储访问操作。

1. 非对齐的指令预取操作

当处理器处于 ARM 状态期间，如果写入到寄存器 PC 中的值是非字对齐的，要么指令执行的结果不可预知，要么地址值中最低两位被忽略；当处理器处于 Thumb 状态期间，如果写入到寄存器 PC 中的值是非半字对齐的，要么指令执行的结果不可预知，要么地址值中最低位被忽略。

如果系统中指定，当发生非对齐的指令预取操作时，忽略地址值中相应的位，则由存储系统实现这种“忽略”。也就是说，这时该地址值原封不动地送到存储系统。

2. 非对齐的数据访问操作

对于 Load/Store 操作，如果是非对齐的数据访问操作，系统定义了下面 3 种可能的结果

- 执行的结果不可预知
- 忽略字单元地址的低两位的值，即访问地址为(address AND 0XFFFFFFFC)的字单元；忽略半字单元地址的最低位的值，即访问地址为(address AND 0XFFFFFFFE)的半字单元。
- 忽略字单元地址值中的低两位的值；忽略半字单元地址的最低位的值。由存储系统实现这种“忽略”。

当发生非对齐的数据访问时，到底采用上述 3 种处理方法中的哪一种，是由各指令指定的。

Avoiding Alignment Issues

编译器通常会通过让所有的数据自然对齐来避免引发对齐问题。实际上，开发者在字节对齐上也许和对待字节序、位序上的态度差不多，那就是不必花费太大心思，也许真正对此犯愁的只有那些搞 GCC 的家伙们。可是，当程序员使用指针太多，对数据的访问超出编译器的预期时，就会引发问题了。

一个数据类型长度较小，它本来是对齐的，如果你用一个指针进行类型转换，并且转换后类型长度较大，那么通过改指针进行数据访问时就会引发对齐问题。如文章开始提到的例子所示，这个例子将一个指向char型的指针当作指向unsigned long型的指针来用，这会试图从一个并不能被 4 整除的内存地址上载入 32 位unsigned long型数据，这时，任何不可能的事情也许都可能发生。当然，也可能什么事都不会发生。

在实际编程中，只要你能意识到这种情况，你基本上就不会犯错了。

Alignment of Nonstandard Types

非标准(复合的)C 数据类型对齐原则：

- 对于数组，按照基本数据类型进行对齐
- 对于联合，按照它包含的长度最大的数据类型进行对齐
- 对于结构，保证它包含的长度最大的基本数据类型能够对齐

通常结构对齐需满足下面 3 个原则：

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于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）。

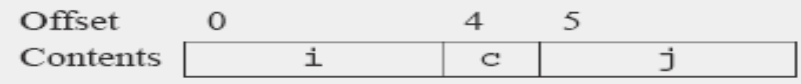
Structure Padding

ANSI C明确规定，不允许编译器改变结构体内成员对象的次序。在C语言中，函数往往通过在结构体地址上加上偏移量来计算变量的位置。因此对于有结构的代码，为了保证结构体中每一个成员都能自然对齐，编译器可能需要在域的分配中插入间隙，以保证每个结构元素都满足它的对齐要求，而结构本身对它的起始地址也有一些对齐要求。

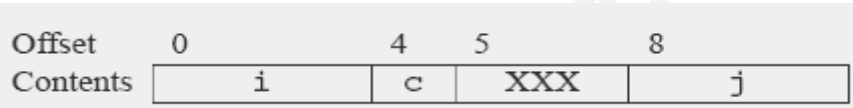
比如下面结构：

```
typedef struct
{
    int     i;
    char    c;
    int     j;
}S;
```

常理推断，该结构在内存中排列为：



那么sizeof(S)应该等于9。可是这样排列是不能满足域i,j的4字节对齐要求的。所以，编译器做了相应的填充以满足这种要求：



结果，j的偏移量为8，而整个结构的大小为12字节。此外，编译器必须保证任何S *类型指针都满足4字节对齐的要求。

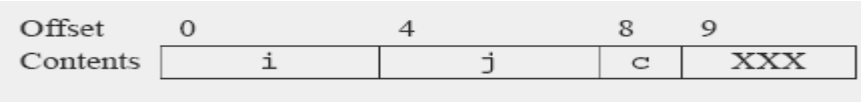
另外，对于结构：

```
typedef struct
{
    int     i;
    int     j;
    char    c;
}S1;
```

编译器仍会进行末尾填充，因为如果不这样，将无法保证下面数据的对齐要求：

```
S1 data[4];
```

稍加考虑可知，只有如下图填充后才可以保证数据的对齐要求。



通常，你可以通过重新排列结构中的对象来避免填充。这样既可以得到一个较小的结构体，又能保证无需填补它也是自然对齐的。如：

```
struct foo_struct                struct foo_struct
{                                {
    char dog;                        unsigned long cat;
    unsigned long cat;               unsigned short pig;
    unsigned short pig;              char fox;
```

```
    char fox;                       char dog;
}                                 }
```

经过调整，前者12字节，后者只有8字节。不过，不是任何时候都可以这样对结构体进行调整的。如：该结构体早已投入使用或是为某个标准定义的或是为了某些原因必须使用的某种固定的次序，则该结构体明显不能再进行这样的调整。

Align & Pack in GCC

确保每种数据类型都是按照指定方式来组织和分配的，即每种类型的对象都满足它的对齐限制，就可保证实施对齐。编译器在汇编代码中放入命令，指明全局数据所需的对齐。

GCC里提供了保留字__attribute__，用来属性描述。而aligned/packed属性描述符用来改变编译器对变量或结构的自然对齐方式。如果担心文件某处定义相同名字的宏，你也可以在这些属性描述符前加上“__”，如，你可以用__aligned__来代替aligned。

Aligned

此属性为变量或结构成员变量指定一个最小的对齐字节数。如：

```
int x __attribute__ ((aligned (16))) = 0;
```

编译器不再使用默认的 4 字节对齐而是在 16 字节的边界上分配全局变量 x。

也可以指定结构成员的对齐属性。如下例，可以使 2 个 int 对齐至 double 边界上：

```
struct foo { int x[2] __attribute__ ((aligned (8))); };
```

可见，上述表示与使用一个 union 的作用是一致的。

```
struct foo { union { int x[2]; double d;};};
```

上面例子中，我们显式地给编译器指出对变量或结构域所期望的对齐字节数。其实，我们也可以把这个工作留给编译器。编译器会在目标机上对变量使用最大最有效的对齐。如：

```
struct S { short f[3]; } __attribute__ ((aligned));
```

在上例中，如果 short 的大小是 2 字节，那么整个结构 S 的大小就是 6 字节。大于等于 6 的最小的 2 的幂是 8，所以编译器会设置结构 S 对齐于 8 字节的地址处。

指定函数的对齐是不可能的，函数的对齐由机器需求决定且不能改变。你也不能对一个 typedef 定义的名称指定对齐，因为这样的名称仅仅是一个别名而不是一个显式的类型。

此外，有些 linker 支持的最大对齐可能很小。假如你的 linker 只支持最大为 8 字节的对齐，那么，即使你指定 aligned(16)属性，linker 也仅会给你 8 字节的对齐。

aligned 属性只能增加变量的对齐值，但你可以使用 pack 属性来减少它。

Packed

packed 属性指定变量或结构域进行最小可能的对齐，对于变量 1 字节对齐，对于位域 1 位对齐。如：

```
struct foo
{
```

```
  char a;
  int x[2] __attribute__ ((packed));
};
```

此时，这个结构的大小就为 9bytes，而不是自然对齐时的 12bytes 了。

对结构，联合，枚举使用 packed 属性相当于对其每个成员使用 packed 属性。使用时__attribute__要紧挨着结构结尾的花括号。

Compile option

GCC/G++ : -fpack-struct
Sun Workshop cc/CC : -misalign

GCC 中使用这个选项，会使所有的结构中不再有填充。但你最好不要这样做，因为这样不仅会大大降低程序的效率，同时，也会存在与系统库不兼容的可能。

还有一些其它 GCC 选项如-falign-functions，-falign-labels，-falign-loops 等就不讲了，感兴趣可翻阅 GCC 手册。

Usage

下面以packed属性为例，讲一下在实际中的使用。

1) 结构内部成员的pack

```
struct foo
{
    char a;
    int  b __attribute__ ((packed));
};
```

2) 整个结构的pack

```
struct foo
{
    char a;
    int  b;
}__attribute__ ((packed));
```

3) 文件范围的pack

```
#pragma pack(1)
struct foo
{
    char a;
    int  b;
};
#pragma pack()
```

pack 改变了自然对齐方式，降低了程序的执行效率，并容易使程序出错，尤其是 pack(1)的结构中包含着结构数组，对结构数组成员的赋值一定要小心。

GCC 手册中说明不能指定函数的对齐(aligned)，但在实际应用中，#pragma pack()却可以应用在函数上，当然效率也是非常的低了。

Example

下面的例子说明在IA32、Xscale(Big-endian)及Arm9上对未对齐地址访问结果,应该可以看出，在IA32系统中，即使访问未对齐的地址空间，你也总能得到你想到的结果，只不过效率上会受到影响。但在Xscale及Arm上，这种行为却是一种不可预知的行为，不能得到你相要的结果。

```
int main(void)
{
    char    a[8] = {0x1,0x2,0x3,0x4,0x5,0x6,0x7,0x8};
    int     *p,*p1,*p2,*p3;

    p = (int *)(a);
    printf("p[%#x]:*p [%#x]\n",p,*p);
    p1 = (int *)(a+1);
    printf("p1[%#x]:*p1 [%#x]\n",p1,*p1);
    p2 = (int *)(a+2);
    printf("p2[%#x]:*p2 [%#x]\n",p2,*p2);
    p3 = (int *)(a+3);
    printf("p3[%#x]:*p3 [%#x]\n",p3,*p3);

    return 0;
}

[Xscale:Big-endian]
p [0x1940b74]:*p  [0x1020304] /* OK */
p1[0x1940b75]:*p1 [0x4010203] /* not 0x02030405 */
p2[0x1940b76]:*p2 [0x3040102] /* not 0x03040506 */
p3[0x1940b77]:*p3 [0x2030401] /* not 0x04050607 */

[Arm9:Little-endian]
p [0x560b50]:*p  [0x4030201] /* OK */
p1[0x560b51]:*p1 [0x1040302] /* not 0x02030405 */
p2[0x560b52]:*p2 [0x2010403] /* not 0x03040506 */
p3[0x560b53]:*p3 [0x3020104] /* not 0x04050607 */

[Linux On X86]
p [0xbffff360]:*p  [0x4030201] /* OK */
p1[0xbffff361]:*p1 [0x5040302] /* OK */
p2[0xbffff362]:*p2 [0x6050403] /* OK */
p3[0xbffff363]:*p3 [0x7060504] /* OK */
```

而上述代码若在SPARC架构的Solaris中运行，应该会出现致命错误(没有验证)。

HOWTO

如前所述，只要你能意识到这种情况，你基本上就不会犯错了。但有的时候，确实需要访问未对齐的内存地址，那么该如何处理呢？下面以网上一网友的问题来说明这种情况：

Q：使用了#pragma pack()，因为需要读取来自 Windows 客户端的报文，对端使用#pragma pack(1)压缩了所使用的数据结构，数据结构如下：

```
#pragma pack(1)
typedef struct pkt_hdr_struct
{
	uint8_t pkt_ver;
	uint32_t pkt_type;
	uint32_t pkt_len;
} pkt_hdr_t;
#pragma pack()
```

为了采用这个结构读取网络数据，Solaris 端的服务程序需要强制转换匹配该结构，但是一旦企图读取紧接在 pkt_ver 成员之后的 pkt_type 成员，崩溃了。尝试过其他办法，首先用一个字符指针读取第一个字节，然后指针增一，把该指针强制类型转换成(uint32_t *)，然后读取数据，依然崩溃。

从上述可知，强转导致指针访问未对齐内存地址是程序崩溃的原因。那么，我们如何处理才能得到正确的结果呢？

1. 就使用单字节读取，然后拼装成相应的数据类型

```
int read_misaligned_int ( int * iptr )
{
	int i;
	int value;
	char * ptr = ( char * )iptr;
	char * vptr = ( char * )&value;

	for ( i = 0; i < sizeof( int ); i++ )
	{
		*vptr++ = *ptr++;
	}

	return( value );
}
```

2. 使用 memcpy、memset

```
int		value;
pkt_hdr_t	pkt;
memcpy(&value, (char *)&pkt + 1, sizeof(int));
```

上述两种方法均可正确得到所需结果。

除此之处，为了避免这种因为未对齐引发的错误，在编程时，尤其是网络编程时我们需注意的有：

1. union & align

还是以一个小例子来说明。

下面是 Richard Stevens 写的一个关于判别机器是大端还小端的一个小程序：

```
UNPv1 : UNIX Network Programming, Volume 1 [Stevens 1998] (W. Richard Stevens)

#include  "unp.h"
int main(int argc, char **argv)
{
	union {
	   short    s;
	   char     c[sizeof(short)];
	} un;
```

```
    un.s = 0x0102;
    printf("%s: ", CPU_VENDOR_OS);
    if (sizeof(short) == 2) {
        if (un.c[0] == 1 && un.c[1] == 2)
            printf("big-endian\n");
        else if (un.c[0] == 2 && un.c[1] == 1)
            printf("little-endian\n");
        else
            printf("unknown\n");
    } else
        printf("sizeof(short) = %d\n", sizeof(short));

    exit(0);
}
```

不知大家有没有对程序中 union 的使用感到有点不解，如前所述，联合在此处便用来对齐了。尤其网络编程，使用大量的缓冲收发数据，此时应该注意缓冲区与数据结构的对齐问题，不妨也使用 union 来避免对齐问题。

2. malloc & align

分配存储器的库函数(如 malloc)的设计必须使得它返回的指针能满足最糟糕情况的对齐限制，通常是 4 或 8。所以 malloc 分配的空间一定是会满足对齐要求的。

Reference