

# Align in system

对齐

## 什么是对齐？

现代计算机中内存空间都是按照字节（byte）划分的，从理论上讲似乎对任何类型的变量的访问可以从任何地址开始，但实际情况是在访问特定变量的时候经常在特定的内存地址访问，这就需要各类型数据按照一定的规则在空间上排列，而不是顺序地一个接一个地排放，这就是对齐。

---

## 计算机为什么要对齐？

各个硬件平台对存储空间的处理上有很大的不同。一些平台对某些特定类型的数据只能从某些特定地址开始存取，其他平台可能没有这种情况。但是最常见的是，如果不按照适合其平台的要求对数据存放进行对齐，会在存取效率上带来损失。比如有些平台每次读都是从偶地址开始，一个int型（假设为32位）如果存放在偶地址开始的地方，那么一个读周期就可以读出，而如果存放在奇地址开始的地方，就可能会需要2个读周期，并对两次读出的结果的高低字节进行拼凑才能得到该int数据，显然在读取效率上下降很多。这也是空间和时间的博弈。在网络程序中，掌握这个概念可是很重要的：如果在不同平台之间（比如在Windows和Linux之间）传递2进制流（比如结构体），那么在这两个平台间必须要定义相同的对齐方式，不然莫名其妙地出了一些错，可是很难排查的。

---

## 对齐的实现：

通常，我们写程序的时候，不需要考虑对齐问题，编译器会替我们选择适合目标平台的对齐策略。当然，我们也可以通知给编译器传递预编译指令而改变对指定数据的对齐方法，比如写入预编译指令#**pragma pack(2)**，即告诉编译器按两字节对齐。

但是，正因为我们一般不需要关心这个问题，所以，如果编辑器对数据存放做了对齐，而我们不了解的话，常常会对一些问题感到迷惑。最常见的就是**struct**数据结构的**sizeof**结果，比如以下程序：

```
#include<stdio.h>
int main(void)
{
    struct A
    {
        char a;
        short b;
        int c;
    };
    printf("结构体类型A在内存中所占内存为：%d字节。\\n", sizeof(struct A));
    return 0;
}
```

输出结果为：8字节。如果我们将结构体中的变量声明位置稍加改动（并不改变变量本身），请再看以下程序：

```
#include<stdio.h>
int main(void)
{
    struct A
    {
        short b;
        int c;
        char a;
    };
}
```

```
    printf("结构体类型A在内存中所占内存为：%d字节。\\n", sizeof(struct A));  
    return 0;  
}
```

输出结果为**12**字节。

问题来了，他们都是同一个结构体，为什么占用的内存大小不同呢？为此，我们需要对对齐算法有所了解。

## 对齐算法

由于各个平台和编译器的不同，现以**32位**，**vc++6.0**系统为例，来讨论编译器对**struct**数据结构中的各成员如何进行对齐的。

首先，我们给出四个概念：

- 1) 数据类型自身的对齐值：就是基本数据类型的自身对齐值，比如**char**类型的自身对齐值为**1**字节，**int**类型的自身对齐值为**4**字节。
- 2) 指定对齐值：预编译命令#**pragma pack (value)** 指定的对齐值**value**。
- 3) 结构体或者类的自身对齐值：其成员中自身对齐值最大的那个值，比如以上的**struct A**的对齐值为**4**。
- 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的那个值。

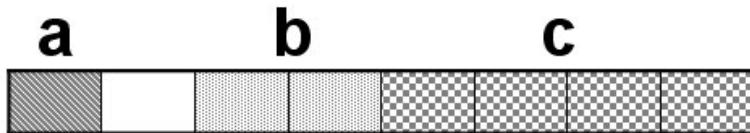
设结构体如下定义：

```
struct A  
{  
    char a;  
    short b;  
    int c;  
};
```

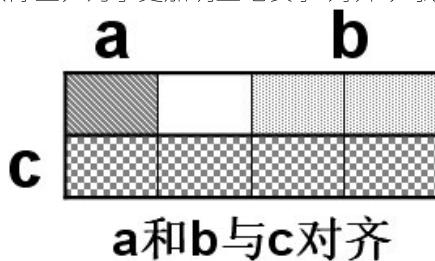
**a**是**char**型数据，占用**1**字节内存；**short**型数据，占用**2**字节内存；**int**型数据，占用**4**字节内存。因此，结构体**A**的自身对齐值为**4**。于是，**a**和**b**要组成**4**个字节，以便与**c**的**4**个字节对齐。而**a**只有**1**个字节，**a**与**b**之间便空了一个字节。我们知道，结构体类型数据是按顺序存储结构一个接一个向后排列的，于是其存储方式为：

**C**语言字节对齐（以**32位**系统为例）

其中空白方格无数据，是浪费的内存空间，共占用**8**字节内存。



实际上，为了更加明显地表示“对齐”，我们可以将以上结构想象为以下的行排列：



对于另一个结构体定义：

```
struct A
```

```

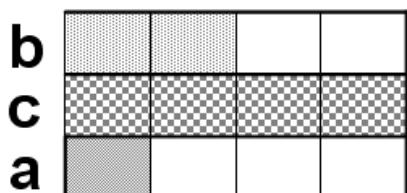
{
    short b;
    int c;
    char a;
};

```

其内存存储方式为：



同样把它想象成行排列：



可见，浪费的空间更多。

其实，除了结构体之外，整个程序在给每个变量进行内存分配时都会遵循对齐机制，也都会产生内存空间的浪费。但我们要知道，这种浪费是值得的，因为它换来的是效率的提高。

以上分析都是建立在程序默认的对齐值基础之上的，我们可以通过添加预定义命令`#pragma pack(value)`来对对齐值进行自定义，比如`#pragma pack(1)`，对齐值变为1，此时内存紧凑，不会出现内存浪费，但效率降低了。效率之所以降低，是因为：如果存在更大字节数的变量时（比1大），比如int类型，需要进行多次读周期才能将一个int数据拼凑起来。

字 word = 2 bytes = 16 bits

字节 byte = 8 bits

位 bit

## 16位编译器

Type	占用内存字节
char	1
char*	2
short int	2
int	2
unsigned int	2
float	4
double	8
long	4
long long	8
unsigned long	4

## 32位编译器

Type	占用内存字节

char	1
char*	4 (32位的寻址空间是 $2^{32}$ , 即32个bit, 也就是4个字节。同理64位编译器)
short int	2
int	4
unsigned int	4
float	4
double	8
long	4
long long	8
unsigned long	4

## 64位编译器

Type	占用内存字节
char	1
char*	8
short int	2
int	4
unsigned int	4
float	4
double	8
long	8
long long	8
unsigned long	8

### Difference between 32/64 bits system

64bit CPU拥有更大的寻址能力，最大支持到16GB内存（因为目前cpu地址总线为34条，寻址范围 $2^{10} \times 2^{10} \times 2^{10} \times 2^4 = 16G$ ），而32bit只支持4G内存(有32位总线)。64位CPU一次可提取64位数据，比32位提高了一倍，理论上性能会提升1倍。但这是建立在64bit操作系统，64bit软件的基础上的。

## 什么是64位处理器？

之所以叫做“64位处理器”，是因为电脑内部都是实行2进制运算，处理器（CPU）一次处理数据的能力也是2的倍数。8位处理器、16位处理器、32位处理器和64位处理器，其计数都是2的倍数。一次处理的数据越大，该电脑处理信息的能力越来越大；因此64位处理在先天就比32位处理器具有快速的能力。