转载自：http://blog.csdn.net/21aspnet/article/details/6729724#comments ，方便大家参考学习

# C语言字节对齐

分类： C 基础2011-08-29 16:0624952人阅读评论(50)收藏举报

c 语言 struct 编译器数据结构 alignment

文章最后本人做了一幅图，一看就明白了，这个问题网上讲的不少，但是都没有把问题说透。

一、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。

二、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。假设上面整型变量的地址不是自然对齐，比如为0x00000002，则 CPU 如果取它的值的话需要访问两次内存，第一次取从0x00000002-0x00000003的一个 short，第二次取从0x00000004-0x00000005的一个 short 然后组合得到所要的数据，如果变量在0x00000003地址上的话则要访问三次内存，第一次为 char，第二次为 short，第三次为 char，然后组合得到整型数据。而如果变量在自然对齐位置上，则只要一次就可以取出数据。一些系统对对齐要求非常严格，比如 sparc 系统，如果取未对齐的数据会发生错误，举个例：

```
char ch[8];
char *p = &ch[1];
int i = *(int *)p;
```

运行时会报 segment error，而在 x86上就不会出现错误，只是效率下降。

三、正确处理字节对齐

对于标准数据类型，它的地址只要是它的长度的整数倍就行了，而非标准数据类型按下面的原则对齐：

数组 ：按照基本数据类型对齐，第一个对齐了后面的自然也就对齐了。
联合 ：按其包含的长度最大的数据类型对齐。
结构体： 结构体中每个数据类型都要对齐。
比如有如下一个结构体：

```
struct stu{
  char sex;
  int length;
```

```
  char name[10];
};
struct stu my_stu;
```

由于在 x86下，GCC 默认按4字节对齐，它会在 sex 后面跟 name 后面分别填充三个和两个字节使 length 和整个结构体对齐。于是我们 sizeof(my_stu)会得到长度为20，而不是15.

四、__attribute__选项

我们可以按照自己设定的对齐大小来编译程序，GNU 使用__attribute__选项来设置，比如我们想让刚才的结构按一字节对齐，我们可以这样定义结构体

```
struct stu{
  char sex;
  int length;
  char name[10];
}__attribute__ ((aligned (1)));

struct stu my_stu;
```

则 sizeof(my_stu)可以得到大小为15。

上面的定义等同于

```
struct stu{
  char sex;
  int length;
  char name[10];
}__attribute__ ((packed));
struct stu my_stu;
```

__attribute__((packed))得变量或者结构体成员使用最小的对齐方式，即对变量是一字节对齐，对域（field）是位对齐.

五、什么时候需要设置对齐

在设计不同 CPU 下的通信协议时，或者编写硬件驱动程序时寄存器的结构这两个地方都需要按一字节对齐。即使看起来本来就自然对齐的也要使其对齐，以免不同的编译器生成的代码不一样.

一、快速理解

1. 什么是字节对齐?

在 C 语言中，结构是一种复合数据类型，其构成元素既可以是基本数据类型（如 int、long、float 等）的变量，也可以是一些复合数据类型（如数组、结构、联合等）的数据单元。在结构中，编译器为结构的每个成员按其自然边界（alignment）分配空间。各个成员按照它们被声明的顺序在内存中顺序存储，第一个成员的地址和整个结构的地址相同。

为了使 CPU 能够对变量进行快速的访问,变量的起始地址应该具有某些特性,即所谓的”对齐”. 比如4字节的 int 型,其起始地址应该位于4字节的边界上,即起始地址能够被4整除.

2. 字节对齐有什么作用?

字节对齐的作用不仅是便于 cpu 快速访问,同时合理的利用字节对齐可以有效地节省存储空间。

对于32位机来说，4字节对齐能够使 cpu 访问速度提高，比如说一个 long 类型的变量，如果跨越了4字节边界存储，那么 cpu 要读取两次，这样效率就低了。但是在32位机中使用1字节或者2字节对齐，反而会使变量访问速度降低。所以这要考虑处理器类型，另外还得考虑编译器的类型。在 vc 中默认是4字节对齐的，GNU gcc 也是默认4字节对齐。

3. 更改 C 编译器的缺省字节对齐方式

在缺省情况下，C 编译器为每一个变量或是数据单元按其自然对界条件分配空间。一般地，可以通过下面的方法来改变缺省的对界条件：

- 使用伪指令#pragma pack (n)，C 编译器将按照 n 个字节对齐。
- 使用伪指令#pragma pack ()，取消自定义字节对齐方式。

另外，还有如下的一种方式：

- __attribute((aligned (n)))，让所作用的结构成员对齐在 n 字节自然边界上。如果结构中有成员的长度大于 n，则按照最大成员的长度来对齐。
- __attribute__ ((packed))，取消结构在编译过程中的优化对齐，按照实际占用字节数进行对齐。

4. 举例说明

例1

```
struct test
{
char x1;
short x2;
float x3;
char x4;
};
```

由于编译器默认情况下会对这个 struct 作自然边界（有人说“自然对界”我觉得边界更顺口）对齐，结构的第一个成员 x1，其偏移地址为0，占据了第1个字节。第二个成员 x2为 short 类型，其起始地址必须2字节对界，因此，编译器在 x2和 x1之间填充了一个空字节。结构的第三个成员 x3和第四个成员 x4恰好落在其自然边界地址上，在它们前面不需要额外的填充字节。在 test 结构中，成员 x3要求4字节对界，是该结构所有成员中要求的最大边界单元，因而 test 结构的自然对界条件为4字节，编译器在成员 x4后面填充了3个空字节。整个结构所占据空间为12字节。

例2

```
#pragma pack(1) //让编译器对这个结构作1字节对齐
```

```
struct test
{
char x1;
short x2;
float x3;
char x4;
};
#pragma pack() //取消1字节对齐，恢复为默认4字节对齐
```
这时候 sizeof(struct test)的值为8。

例3

```
#define GNUC_PACKED __attribute__((packed))
struct PACKED test
{
char x1;
short x2;
float x3;
char x4;
}GNUC_PACKED;
```
这时候 sizeof(struct test)的值仍为8。

二、深入理解

什么是字节对齐,为什么要对齐?

TragicJun 发表于 2006-9-18 9:41:00 现代计算机中内存空间都是按照 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类型变量的时候经常在特定的内存地址访问，这就需要各种类型数据按照一定的规则在空间上排列，而不是顺序的一个接一个的排放，这就是对齐。

对齐的作用和原因：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。比如有些架构的 CPU 在访问一个没有进行对齐的变量的时候会发生错误,那么在这种架构下编程必须保证字节对齐.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，如果一个 int 型（假设为32位系统）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这32bit，而如果存放在奇地址开始的地方，就需要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32bit数据。显然在读取效率上下降很多。

二.字节对齐对程序的影响:

先让我们看几个例子吧(32bit,x86环境,gcc 编译器):

设结构体如下定义：

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

```
        char b;
        int a;
        short c;
};
```
现在已知32位机器上各种数据类型的长度如下:
char:1(有符号无符号同)
short:2(有符号无符号同)
int:4(有符号无符号同)
long:4(有符号无符号同)
float:4　　double:8
那么上面两个结构大小如何呢?
结果是:
sizeof(strcut A)值为8
sizeof(struct B)的值却是12
结构体 A 中包含了4字节长度的 int 一个,1字节长度的 char 一个和2字节长度的 short 型数据一个,B 也一样;按理说 A,B 大小应该都是7字节。
之所以出现上面的结果是因为编译器要对数据成员在空间上进行对齐。上面是按照编译器的默认设置进行对齐的结果,那么我们是不是可以改变编译器的这种默认对齐设置呢,当然可以.例如:
```
#pragma pack (2) /*指定按2字节对齐*/
struct C
{
        char b;
        int a;
        short c;
};
#pragma pack () /*取消指定对齐，恢复缺省对齐*/
```
sizeof(struct C)值是8。
修改对齐值为1：
```
#pragma pack (1) /*指定按1字节对齐*/
struct D
{
        char b;
        int a;
        short c;
};
#pragma pack () /*取消指定对齐，恢复缺省对齐*/
```
sizeof(struct D)值为7。
后面我们再讲解#pragma pack()的作用.
三.编译器是按照什么样的原则进行对齐的?
　　先让我们看四个重要的基本概念：

1.数据类型自身的对齐值：
　　对于 char 型数据，其自身对齐值为1，对于 short 型为2，对于 int,float,double

类型，其自身对齐值为4，单位字节。
2.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。
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小的那个值。
有了这些值，我们就可以很方便的来讨论具体数据结构的成员和其自身的对齐方式。有效对齐值 N 是最终用来决定数据存放地址方式的值，最重要。有效对齐 N，就是表示“对齐在 N 上”，也就是说该数据的"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来排放的。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排放，结构体本身也要根据自身的有效对齐值圆整(就是结构体成员变量占用总长度需要是对结构体有效对齐值的整数倍，结合下面例子理解)。这样就不能理解上面的几个例子的值了。
例子分析：
分析例子 B;

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

假设 B 从地址空间0x0000开始排放。该例子中没有定义指定对齐值，在笔者环境下，该值默认为4。第一个成员变量 b 的自身对齐值是1，比指定或者默认指定对齐值4小，所以其有效对齐值为1，所以其存放地址0x0000符合0x0000%1=0.第二个成员变量 a，其自身对齐值为4，所以有效对齐值也为4，所以只能存放在起始地址为0x0004到0x0007这四个连续的字节空间中，复核0x0004%4=0,且紧靠第一个变量。第三个变量 c,自身对齐值为2，所以有效对齐值也是2，可以存放在0x0008到0x0009这两个字节空间中，符合0x0008%2=0。所以从0x0000到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，0x0009到0x0000=10字节,(10+2)%4=0。所以0x0000A 到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;其实如果就这一个就来说它已将满足字节对齐了,因为它的起始地址是0,因此肯定是对齐的,之所以在后面补充2个字节,是因为编译器为了实现结构数组的存取效率,试想如果我们定义了一个结构 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,数组中所有元素都是紧挨着的,如果我们不把结构的大小补充为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了,因此我们要把结构补充成有效对齐大小的整数倍.其实诸如:对于 char 型数据，其自身对齐值为1，对于 short 型为2，对于 int,float,double 类型，其自身对齐值为4，这些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了,所以他们的自身对齐值也就已知了.
同理,分析上面例子 C：

```
#pragma pack (2) /*指定按2字节对齐*/
struct C
{
        char b;
        int a;
        short c;
```

};
#pragma pack () /*取消指定对齐，恢复缺省对齐*/
第一个变量 b 的自身对齐值为1，指定对齐值为2，所以，其有效对齐值为1，假设 C 从0x0000开始，那么 b 存放在0x0000，符合0x0000%1=0;第二个变量，自身对齐值为4，指定对齐值为2，所以有效对齐值为2，所以顺序存放在0x0002、0x0003、0x0004、0x0005四个连续字节中,符合0x0002%2=0。第三个变量 c 的自身对齐值为2，所以有效对齐值为2，顺序存放在0x0006、0x0007中，符合0x0006%2=0。所以从0x0000到0x00007共八字节存放的是 C 的变量。又 C 的自身对齐值为4，所以 C 的有效对齐值为2。又8%2=0,C 只占用0x0000到0x0007的八个字节。所以 sizeof(struct C)=8.
四.如何修改编译器的默认对齐值?
1.在 VC IDE 中，可以这样修改：[Project]|[Settings],c/c++选项卡 Category 的 Code Generation 选项的 Struct Member Alignment 中修改，默认是8字节。
2.在编码时，可以这样动态修改：#pragma pack .注意:是 pragma 而不是 progma.
五.针对字节对齐,我们在编程中如何考虑?
如果在编程的时候要考虑节约空间的话,那么我们只需要假定结构的首地址是0,然后各个变量按照上面的原则进行排列即可,基本的原则就是把结构中的变量按照类型大小从小到大声明,尽量减少中间的填补空间.还有一种就是为了以空间换取时间的效率,我们显示的进行填补空间进行对齐,比如:有一种使用空间换时间做法是显式的插入 reserved 成员：

```
struct A{
   char a;
   char reserved[3];//使用空间换时间
   int b;
}
```

reserved 成员对我们的程序没有什么意义,它只是起到填补空间以达到字节对齐的目的,当然即使不加这个成员通常编译器也会给我们自动填补对齐,我们自己加上它只是起到显式的提醒作用.
六.字节对齐可能带来的隐患:
代码中关于对齐的隐患，很多是隐式的。比如在强制类型转换的时候。例如：

```
unsigned int i = 0x12345678;
unsigned char *p=NULL;
unsigned short *p1=NULL;
p=&i;
*p=0x00;
p1=(unsigned short *)(p+1);
*p1=0x0000;
```

最后两句代码，从奇数边界去访问 unsignedshort 型变量，显然不符合对齐的规定。
在 x86上，类似的操作只会影响效率，但是在 MIPS 或者 sparc 上，可能就是一个 error,因为它们要求必须字节对齐.
七.如何查找与字节对齐方面的问题:
如果出现对齐或者赋值问题首先查看
1. 编译器的 big little 端设置
2. 看这种体系本身是否支持非对齐访问
3. 如果支持看设置了对齐与否,如果没有则看访问时需要加某些特殊的修饰来标志其特殊访问操作

举例：

[cpp]view plaincopy

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

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

#pragma pack (2) /*指定按2字节对齐*/
struct C {
char b;
int a;
short c;
};
#pragma pack () /*取消指定对齐，恢复缺省对齐*/


#pragma pack (1) /*指定按1字节对齐*/
struct D {
char b;
```

```
29  int a;
30  short c;
31  };
32  #pragma pack ()/*取消指定对齐，恢复缺省对齐*/
33
34  int s1=sizeof(struct A);
35  int s2=sizeof(struct B);
36  int s3=sizeof(struct C);
37  int s4=sizeof(struct D);
38  printf("%d\n",s1);
39  printf("%d\n",s2);
40  printf("%d\n",s3);
41  printf("%d\n",s4);
42  }
```

输出：

8
12
8
7

修改代码：

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

输出：

4
4

输出都是4，说明之前的 int 影响对齐！

看图就明白了