C 语言字节对齐问题详解

引言

考虑下面的结构体定义：

```
1 typedef struct{
2         char    c1;
3         short s;
4         char    c2;
5         int     i;
6 }T_FOO;
```

假设这个结构体的成员在内存中是紧凑排列的，且 c1 的起始地址是 0，则 s 的地址就是 1，c2 的地址是 3，i 的地址是 4。

现在，我们编写一个简单的程序：

复制代码

```
1 int main(void){
2         T_FOO a;
3         printf("c1 -> %d, s -> %d, c2 -> %d, i -> %d\n",
4                 (unsigned int)(void*)&a.c1 - (unsigned int)(void*)&a,
5                 (unsigned int)(void*)&a.s    - (unsigned int)(void*)&a,
6                 (unsigned int)(void*)&a.c2 - (unsigned int)(void*)&a,
7                 (unsigned int)(void*)&a.i    - (unsigned int)(void*)&a);
8         return 0;
9 }
```

复制代码

运行后输出：

```
1 c1 -> 0, s -> 2, c2 -> 4, i -> 8
```

为什么会这样？这就是字节对齐导致的问题。

本文在参考诸多资料的基础上，详细介绍常见的字节对齐问题。因成文较早，资料来源大多已不可考，敬请谅解。

一　什么是字节对齐

现代计算机中，内存空间按照字节划分，理论上可以从任何起始地址访问任意类型的

变量。但实际中在访问特定类型变量时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，而不是顺序一个接一个地存放，这就是对齐。

二　对齐的原因和作用

不同硬件平台对存储空间的处理上存在很大的不同。某些平台对特定类型的数据只能从特定地址开始存取，而不允许其在内存中任意存放。例如 Motorola 68000 处理器不允许 16 位的字存放在奇地址，否则会触发异常，因此在这种架构下编程必须保证字节对齐。

但最常见的情况是，如果不按照平台要求对数据存放进行对齐，会带来存取效率上的损失。比如 32 位的 Intel 处理器通过总线访问(包括读和写)内存数据。每个总线周期从偶地址开始访问 32 位内存数据，内存数据以字节为单位存放。如果一个 32 位的数据没有存放在 4 字节整除的内存地址处，那么处理器就需要 2 个总线周期对其进行访问，显然访问效率下降很多。

因此，通过合理的内存对齐可以提高访问效率。为使 CPU 能够对数据进行快速访问，数据的起始地址应具有“对齐”特性。比如 4 字节数据的起始地址应位于 4 字节边界上，即起始地址能够被 4 整除。

此外，合理利用字节对齐还可以有效地节省存储空间。但要注意，在 32 位机中使用 1 字节或 2 字节对齐，反而会降低变量访问速度。因此需要考虑处理器类型。还应考虑编译器的类型。在 VC/C++和 GNU GCC 中都是默认是 4 字节对齐。

三　对齐的分类和准则

主要基于 Intel X86 架构介绍结构体对齐和栈内存对齐，位域本质上为结构体类型。

对于 Intel X86 平台，每次分配内存应该是从 4 的整数倍地址开始分配，无论是对结构体变量还是简单类型的变量。

3.1 结构体对齐

在 C 语言中，结构体是种复合数据类型，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、结构体、联合等)的数据单元。编译器为结构体的每个成员按照其自然边界(alignment)分配空间。各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。

字节对齐的问题主要就是针对结构体。

3.1.1 简单示例

先看个简单的例子(32 位，X86 处理器，GCC 编译器)：

【例 1】设结构体如下定义：

复制代码

```
1 struct A{
2       int       a;
3       char      b;
4       short     c;
5 };
6 struct B{
7       char      b;
8       int       a;
9       short     c;
10 };
```

复制代码

已知 32 位机器上各数据类型的长度为：char 为 1 字节、short 为 2 字节、int 为 4 字节、long 为 4 字节、float 为 4 字节、double 为 8 字节。那么上面两个结构体大小如何呢？

结果是：sizeof(strcut A)值为 8；sizeof(struct B)的值却是 12。

结构体 A 中包含一个 4 字节的 int 数据，一个 1 字节 char 数据和一个 2 字节 short 数据；B 也一样。按理说 A 和 B 大小应该都是 7 字节。之所以出现上述结果，就是因为编译器要对数据成员在空间上进行对齐。

3.1.2 对齐准则

先来看四个重要的基本概念：

1) 数据类型自身的对齐值：char 型数据自身对齐值为 1 字节，short 型数据为 2 字节，int/float 型为 4 字节，double 型为 8 字节。

2) 结构体或类的自身对齐值：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较小者，即有效对齐值=min{自身对齐值，当前指定的 pack 值}。

基于上面这些值，就可以方便地讨论具体数据结构的成员和其自身的对齐方式。

其中，有效对齐值 N 是最终用来决定数据存放地址方式的值。有效对齐 N 表示“对齐在 N 上”，即该数据的“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存放。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存放，结构体本身也要根据自身的有效对齐值圆整(即结构体成员变量占用总长度为结构体有效对齐值的整数倍)。

以此分析 3.1.1 节中的结构体 B：

假设 B 从地址空间 0x0000 开始存放，且指定对齐值默认为 4(4 字节对齐)。成员变量 b 的自身对齐值是 1，比默认指定对齐值 4 小，所以其有效对齐值为 1，其存放地址 0x0000 符合 0x0000%1=0。成员变量 a 自身对齐值为 4，所以有效对齐值也为 4，只能存放在起始地址为 0x0004~0x0007 四个连续的字节空间中，符合 0x0004%4=0 且紧靠第一个变量。变量 c 自身对齐值为 2，所以有效对齐值也是 2，可存放在 0x0008~0x0009 两个字节空间中，符合 0x0008%2=0。所以从 0x0000~0x0009 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， 0x0000~0x0009=10 字节，(10＋2)％4＝0。所以 0x0000A~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。

之所以编译器在后面补充 2 个字节，是为了实现结构数组的存取效率。试想如果定义一个结构 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，数组中所有元素都紧挨着。如果我们不把结构体大小补充为 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。因此要把结构体补充成有效对齐大小的整数倍。其实对于 char/short/int/float/double 等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，所以他们的自身对齐值也就已知。

上面的概念非常便于理解，不过个人还是更喜欢下面的对齐准则。

结构体字节对齐的细节和具体编译器实现相关，但一般而言满足三个准则：

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}。

对于以上规则的说明如下：

第一条：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，然后寻找内存地址能被该基本数据类型所整除的位置，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。

第二条：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大小的整数倍，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。

第三条：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。

【例 2】假设 4 字节对齐，以下程序的输出结果是多少？

复制代码

```
/* OFFSET 宏定义可取得指定结构体某成员在结构体内部的偏移 */
#define OFFSET(st, field)         (size_t)&(((st*)0)->field)
typedef struct{
        char    a;
        short b;
        char    c;
        int     d;
        char    e[3];
}T_Test;

int main(void){
        printf("Size = %d\n    a-%d, b-%d, c-%d, d-%d\n    e[0]-%d, e[1]-%d, e[2]-%d\n",
                sizeof(T_Test), OFFSET(T_Test, a), OFFSET(T_Test, b),
                OFFSET(T_Test, c), OFFSET(T_Test, d), OFFSET(T_Test, e[0]),
                OFFSET(T_Test, e[1]),OFFSET(T_Test, e[2]));
        return 0;
}
```

复制代码

执行后输出如下：

```
Size = 16
    a-0, b-2, c-4, d-8
    e[0]-12, e[1]-13, e[2]-14
```

下面来具体分析：

首先 char a 占用 1 个字节，没问题。

short b 本身占用 2 个字节，根据上面准则 2，需要在 b 和 a 之间填充 1 个字节。

char c 占用 1 个字节，没问题。

int d 本身占用 4 个字节，根据准则 2，需要在 d 和 c 之间填充 3 个字节。

char e[3]；本身占用 3 个字节，根据原则 3，需要在其后补充 1 个字节。

因此，sizeof(T_Test) = 1 + 1 + 2 + 1 + 3 + 4 + 3 + 1 = 16 字节。

3.1.3 对齐的隐患

3.1.3.1 数据类型转换

代码中关于对齐的隐患，很多是隐式的。例如，在强制类型转换的时候：

复制代码

```
1 int main(void){
2       unsigned int i = 0x12345678;
3
4       unsigned char *p = (unsigned char *)&i;
5       *p = 0x00;
6       unsigned short *p1 = (unsigned short *)(p+1);
7       *p1 = 0x0000;
8
9       return 0;
10 }
```

复制代码

最后两句代码，从奇数边界去访问 unsigned short 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；但在 MIPS 或者 SPARC 上可能导致 error，因为它们要求必须字节对齐。

又如对于 3.1.1 节的结构体 struct B，定义如下函数：

```
1 void Func(struct B *p){
2       //Code
3 }
```

在函数体内如果直接访问 p->a，则很可能会异常。因为 MIPS 认为 a 是 int，其地址应该是 4 的倍数，但 p->a 的地址很可能不是 4 的倍数。

如果 p 的地址不在对齐边界上就可能出问题，比如 p 来自一个跨 CPU 的数据包(多种数据类型的数据被按顺序放置在一个数据包中传输)，或 p 是经过指针移位算出来的。因此要特别注意跨 CPU 数据的接口函数对接口输入数据的处理，以及指针移位再强制转换为结构指针进行访问时的安全性。

解决方式如下：

1) 定义一个此结构的局部变量，用 memmove 方式将数据拷贝进来。

```
1 void Func(struct B *p){
2       struct B tData;
3       memmove(&tData, p, sizeof(struct B));
4       //此后可安全访问 tData.a，因为编译器已将 tData 分配在正确的起始地址上
5 }
```

注意：如果能确定 p 的起始地址没问题，则不需要这么处理；如果不能确定(比如跨 CPU 输入数据、或指针移位运算出来的数据要特别小心)，则需要这样处理。

2) 用#pragma pack (1)将 STRUCT_T 定义为 1 字节对齐方式。

3.1.3.2 处理器间数据通信

处理器间通过消息(对于 C/C++而言就是结构体)进行通信时，需要注意字节对齐以及字节序的问题。

大多数编译器提供内存对其的选项供用户使用。这样用户可以根据处理器的情况选择不同的字节对齐方式。例如 C/C++编译器提供的#pragma pack(n) n=1，2，4 等，让编译器在生成目标文件时，使内存数据按照指定的方式排布在 1，2，4 等字节整除的内存地址处。

然而在不同编译平台或处理器上，字节对齐会造成消息结构长度的变化。编译器为了使字节对齐可能会对消息结构体进行填充，不同编译平台可能填充为不同的形式，大大增加处理器间数据通信的风险。

下面以 32 位处理器为例，提出一种内存对齐方法以解决上述问题。

对于本地使用的数据结构，为提高内存访问效率，采用四字节对齐方式；同时为了减少内存的开销，合理安排结构体成员的位置，减少四字节对齐导致的成员之间的空隙，降低内存开销。

对于处理器之间的数据结构，需要保证消息长度不会因不同编译平台或处理器而导致消息结构体长度发生变化，使用一字节对齐方式对消息结构进行紧缩；为保证处理器之间的消息数据结构的内存访问效率，采用字节填充的方式自己对消息中成员进行四字节对齐。

数据结构的成员位置要兼顾成员之间的关系、数据访问效率和空间利用率。顺序安排原则是：四字节的放在最前面，两字节的紧接最后一个四字节成员，一字节紧接最后一个两字节成员，填充字节放在最后。

举例如下：

复制代码

```
1 typedef struct tag_T_MSG{
2        long    ParaA;
3        long    ParaB;
4        short ParaC；
5        char    ParaD;
6        char    Pad;     //填充字节
7 }T_MSG;
```

复制代码

3.1.3.3 排查对齐问题

如果出现对齐或者赋值问题可查看：

1) 编译器的字节序大小端设置；

2) 处理器架构本身是否支持非对齐访问；

3) 如果支持看设置对齐与否，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。

3.1.4 更改对齐方式

主要是更改 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(GCC 特有语法)：

__attribute((aligned (n)))： 让所作用的结构成员对齐在 n 字节自然边界上。如果结构体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。

【注】__attribute__机制是 GCC 的一大特色，可以设置函数属性(Function Attribute)、变量属性(Variable Attribute)和类型属性(Type Attribute)。详细介绍请参考：

http://www.unixwiz.net/techtips/gnu-c-attributes.html

下面具体针对 MS VC/C++ 6.0 编译器介绍下如何修改编译器默认对齐值。

1) VC/C++ IDE 环境中，可在[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。

VC/C++中的编译选项有/Zp[1|2|4|8|16]，/Zpn 表示以 n 字节边界对齐。n 字节边界对齐是指一个成员的地址必须安排在成员的尺寸的整数倍地址上或者是 n 的整数倍地址上，取它们中的最小值。亦即：min(sizeof(member), n)。

实际上，1 字节边界对齐也就表示结构成员之间没有空洞。

/Zpn 选项应用于整个工程，影响所有参与编译的结构体。在 Struct member alignment 中可选择不同的对齐值来改变编译选项。

2) 在编码时，可用#pragma pack 动态修改对齐值。具体语法说明见附录 5.3 节。

自定义对齐值后要用#pragma pack()来还原，否则会对后面的结构造成影响。

【例 3】分析如下结构体 C：

复制代码

```
1 #pragma pack(2)   //指定按 2 字节对齐
2 struct C{
3       char    b;
4       int     a;
5       short c;
6 };
7 #pragma pack()    //取消指定对齐，恢复缺省对齐
```

复制代码

变量 b 自身对齐值为 1，指定对齐值为 2，所以有效对齐值为 1，假设 C 从 0x0000 开始，则 b 存放在 0x0000，符合 0x0000%1= 0；变量 a 自身对齐值为 4，指定对齐值为 2，所以有效对齐值为 2，顺序存放在 0x0002~0x0005 四个连续字节中，符合 0x0002%2=0。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。C 的自身对齐值为 4，所以其有效对齐值为 2。又 8%2=0，C 只占用 0x0000~0x0007 的八个字节。所以 sizeof(struct C) = 8。

注意，结构体对齐到的字节数并非完全取决于当前指定的 pack 值，如下：

复制代码

```
1 #pragma pack(8)
2 struct D{
3       char    b;
4       short a;
5       char    c;
6 };
7 #pragma pack()
```

复制代码

虽然#pragma pack(8)，但依然按照两字节对齐，所以 sizeof(struct D)的值为 6。因为：对齐到的字节数 = min｛当前指定的 pack 值，最大成员大小｝。

另外，GNU GCC 编译器中按 1 字节对齐可写为以下形式：

```
1 #define GNUC_PACKED __attribute__((packed))
2 struct C{
3       char    b;
4       int     a;
5       short c;
6 }GNUC_PACKED;
```

此时 sizeof(struct C)的值为 7。

3.2 栈内存对齐

在 VC/C++中，栈的对齐方式不受结构体成员对齐选项的影响。总是保持对齐且对齐在 4 字节边界上。

【例 4】

复制代码

```
#pragma pack(push, 1)    //后面可改为 1, 2, 4, 8
struct StrtE{
        char m1;
        long m2;
};
#pragma pack(pop)

int main(void){
        char a;
        short b;
        int c;
        double d[2];
        struct StrtE s;

        printf("a       address:     %p\n", &a);
        printf("b       address:     %p\n", &b);
        printf("c       address:     %p\n", &c);
        printf("d[0] address:     %p\n", &(d[0]));
        printf("d[1] address:     %p\n", &(d[1]));
        printf("s       address:     %p\n", &s);
        printf("s.m2 address:     %p\n", &(s.m2));
        return 0;
}
```

复制代码

结果如下：

复制代码

```
a       address:     0xbfc4cfff
b       address:     0xbfc4cffc
c       address:     0xbfc4cff8
d[0] address:     0xbfc4cfe8
d[1] address:     0xbfc4cff0
s       address:     0xbfc4cfe3
s.m2 address:     0xbfc4cfe4
```

复制代码

可以看出都是对齐到 4 字节。并且前面的 char 和 short 并没有被凑在一起(成 4 字节)，这和结构体内的处理是不同的。

至于为什么输出的地址值是变小的，这是因为该平台下的栈是倒着“生长”的。

3.3 位域对齐

3.3.1 位域定义

有些信息在存储时，并不需要占用一个完整的字节，而只需占几个或一个二进制位。例如在存放一个开关量时，只有 0 和 1 两种状态，用一位二进位即可。为了节省存储空间和处理简便，C 语言提供了一种数据结构，称为“位域”或“位段”。

位域是一种特殊的结构成员或联合成员(即只能用在结构或联合中)，用于指定该成员在内存存储时所占用的位数，从而在机器内更紧凑地表示数据。每个位域有一个域名，允许在程序中按域名操作对应的位。这样就可用一个字节的二进制位域来表示几个不同的对象。

位域定义与结构定义类似，其形式为：

struct 位域结构名

{ 位域列表 };

其中位域列表的形式为：

类型说明符位域名：位域长度

位域的使用和结构成员的使用相同，其一般形式为：

位域变量名.位域名

位域允许用各种格式输出。

位域在本质上就是一种结构类型，不过其成员是按二进位分配的。位域变量的说明与结构变量说明的方式相同，可先定义后说明、同时定义说明或直接说明。

位域的使用主要为下面两种情况：

1) 当机器可用内存空间较少而使用位域可大量节省内存时。如把结构作为大数组的元素时。

2) 当需要把一结构体或联合映射成某预定的组织结构时。如需要访问字节内的特定位时。

3.3.2 对齐准则

位域成员不能单独被取 sizeof 值。下面主要讨论含有位域的结构体的 sizeof。

C99 规定 int、unsigned int 和 bool 可以作为位域类型，但编译器几乎都对此作了扩展，允许其它类型的存在。位域作为嵌入式系统中非常常见的一种编程工具，优点在于压缩程序的存储空间。

其对齐规则大致为：

1) 如果相邻位域字段的类型相同，且其位宽之和小于类型的 sizeof 大小，则后面的字段将紧邻前一个字段存储，直到不能容纳为止；

2) 如果相邻位域字段的类型相同，但其位宽之和大于类型的 sizeof 大小，则后面的字段将从新的存储单元开始，其偏移量为其类型大小的整数倍；

3) 如果相邻的位域字段的类型不同，则各编译器的具体实现有差异，VC6 采取不压缩方式，Dev-C++和 GCC 采取压缩方式；

4) 如果位域字段之间穿插着非位域字段，则不进行压缩；

5) 整个结构体的总大小为最宽基本类型成员大小的整数倍，而位域则按照其最宽类型字节数对齐。

【例 5】

```
1 struct BitField{
2         char element1    : 1;
3         char element2    : 4;
4         char element3    : 5;
5 };
```

位域类型为 char，第 1 个字节仅能容纳下 element1 和 element2，所以 element1 和 element2 被压缩到第 1 个字节中，而 element3 只能从下一个字节开始。因此 sizeof(BitField) 的结果为 2。

【例 6】

```
1 struct BitField1{
2         char element1     : 1;
3         short element2    : 5;
4         char element3     : 7;
5 };
```

由于相邻位域类型不同，在 VC6 中其 sizeof 为 6，在 Dev-C++中为 2。

【例 7】

```
1 struct BitField2{
2         char element1    : 3;
3         char element2    ;
4         char element3    : 5;
5 };
```

非位域字段穿插在其中，不会产生压缩，在 VC6 和 Dev-C++中得到的大小均为 3。

【例 8】

复制代码

```
1 struct StructBitField{
2        int element1     : 1;
3        int element2     : 5;
4        int element3     : 29;
5        int element4     : 6;
6        char element5    :2;
7        char stelement;    //在含位域的结构或联合中也可同时说明普通成员
8 };
```

复制代码

位域中最宽类型 int 的字节数为 4，因此结构体按 4 字节对齐，在 VC6 中其 sizeof 为 16。

3.3.3 注意事项

关于位域操作有几点需要注意：

1) 位域的地址不能访问，因此不允许将&运算符用于位域。不能使用指向位域的指针也不能使用位域的数组(数组是种特殊指针)。

例如，scanf 函数无法直接向位域中存储数据：

```
1 int main(void){
2        struct BitField1 tBit;
3        scanf("%d", &tBit.element2); //error: cannot take address of bit-field 'element2'
4        return 0;
5 }
```

可用 scanf 函数将输入读入到一个普通的整型变量中，然后再赋值给 tBit.element2。

2) 位域不能作为函数返回的结果。

3) 位域以定义的类型为单位，且位域的长度不能够超过所定义类型的长度。例如定义 int a:33 是不允许的。

4) 位域可以不指定位域名，但不能访问无名的位域。

位域可以无位域名，只用作填充或调整位置，占位大小取决于该类型。例如，char :0 表示整个位域向后推一个字节，即该无名位域后的下一个位域从下一个字节开始存放，同理 short :0 和 int :0 分别表示整个位域向后推两个和四个字节。

当空位域的长度为具体数值 N 时(如 int :2)，该变量仅用来占位 N 位。

【例 9】

```
1 struct BitField3{
2        char element1    : 3;
3        char    :6;
4        char element3    : 5;
5 };
```

结构体大小为 3。因为 element1 占 3 位，后面要保留 6 位而 char 为 8 位，所以保留的 6 位只能放到第 2 个字节。同样 element3 只能放到第 3 字节。

```
1 struct BitField4{
2        char element1    : 3;
3        char    :0;
4        char element3    : 5;
5 };
```

长度为 0 的位域告诉编译器将下一个位域放在一个存储单元的起始位置。如上，编译器会给成员 element1 分配 3 位，接着跳过余下的 4 位到下一个存储单元，然后给成员 element3 分配 5 位。故上面的结构体大小为 2。

5) 位域的表示范围。

位域的赋值不能超过其可以表示的范围；
位域的类型决定该编码能表示的值的结果。

对于第二点，若位域为 unsigned 类型，则直接转化为正数；若非 unsigned 类型，则先判断最高位是否为 1，若为 1 表示补码，则对其除符号位外的所有位取反再加一得到最后的结果数据(原码)。如：

```
1 unsigned int p:3 = 111;     //p 表示 7
2 int p:3 = 111;                    //p  表示-1，对除符号位之外的所有位取反再加一
```

6) 带位域的结构在内存中各个位域的存储方式取决于编译器，既可从左到右也可从右到左存储。

【例 10】在 VC6 下执行下面的代码：

复制代码

```
int main(void){
    union{
        int i;
        struct{
            char a : 1;
            char b : 1;
            char c : 2;
        }bits;
    }num;
```

```
    printf("Input an integer for i(0~15): ");
    scanf("%d", &num.i);
    printf("i = %d, cba = %d %d %d\n", num.i, num.bits.c, num.bits.b, num.bits.a);
    return 0;
}
```

复制代码

输入 i 值为 11，则输出为 i = 11, cba = -2 -1 -1。

Intel x86 处理器按小字节序存储数据，所以 bits 中的位域在内存中放置顺序为 ccba。当 num.i 置为 11 时，bits 的最低有效位(即位域 a)的值为 1，a、b、c 按低地址到高地址分别存储为 10、1、1(二进制)。

但为什么最后的打印结果是 a=-1 而不是 1？

因为位域 a 定义的类型 signed char 是有符号数，所以尽管 a 只有 1 位，仍要进行符号扩展。1 做为补码存在，对应原码-1。

如果将 a、b、c 的类型定义为 unsigned char，即可得到 cba = 2 1 1。1011 即为 11 的二进制数。

注：C 语言中，不同的成员使用共同的存储区域的数据构造类型称为联合(或共用体)。联合占用空间的大小取决于类型长度最大的成员。联合在定义、说明和使用形式上与结构体相似。

7) 位域的实现会因编译器的不同而不同，使用位域会影响程序可移植性。因此除非必要否则最好不要使用位域。

8) 尽管使用位域可以节省内存空间，但却增加了处理时间。当访问各个位域成员时，需要把位域从它所在的字中分解出来或反过来把一值压缩存到位域所在的字位中。

四 总结

让我们回到引言部分的问题。

缺省情况下，C/C++编译器默认将结构、栈中的成员数据进行内存对齐。因此，引言程序输出就变成"c1 -> 0, s -> 2, c2 -> 4, i -> 8"。

编译器将未对齐的成员向后移，将每一个都成员对齐到自然边界上，从而也导致整个结构的尺寸变大。尽管会牺牲一点空间(成员之间有空洞)，但提高了性能。

也正是这个原因，引言例子中 sizeof(T_ FOO)为 12，而不是 8。

总结说来，就是

在结构体中，综合考虑变量本身和指定的对齐值；

在栈上，不考虑变量本身的大小，统一对齐到 4 字节。

五　附录

5.1 字节序与网络序

5.1.1 字节序

字节序，顾名思义就是字节的高低位存放顺序。

对于单字节，大部分处理器以相同的顺序处理比特位，因此单字节的存放和传输方式一般相同。

对于多字节数据，如整型(32 位机中一般占 4 字节)，在不同的处理器的存放方式主要有两种(以内存中 0x0A0B0C0D 的存放方式为例)。

1) 大字节序(Big-Endian，又称大端序或大尾序)

在计算机中，存储介质以下面方式存储整数 0x0A0B0C0D 则称为大字节序：

数据以 8bit 为单位

低地址方向

0x0A

0x0B

0x0C

0x0D

高地址方向

数据以 16bit 为单位

低地址方向

0x0A0B

0x0C0D

高地址方向

其中，最高有效位(MSB，Most Significant Byte)0x0A 存储在最低的内存地址处。下个字节 0x0B 存在后面的地址处。同时，最高的 16bit 单元 0x0A0B 存储在低位。

简而言之，大字节序就是“高字节存入低地址，低字节存入高地址”。

这里讲个词源典故：“endian”一词来源于乔纳森·斯威夫特的小说《格列佛游记》。小说中，小人国为水煮蛋该从大的一端(Big-End)剥开还是小的一端(Little-End)剥开而争论，争论的双方分别被称为 Big-endians 和 Little-endians。

1980 年，Danny Cohen 在其著名的论文"On Holy Wars and a Plea for Peace"中为平息一场关于字节该以什么样的顺序传送的争论而引用了该词。

借用上面的典故，想象一下要把熟鸡蛋旋转着稳立起来，大头(高字节)肯定在下面(低地址)^_^

2) 小字节序(Little-Endian，又称小端序或小尾序)

在计算机中，存储介质以下面方式存储整数 0x0A0B0C0D 则称为小字节序：

数据以 8bit 为单位

高地址方向

0x0A

0x0B

0x0C

0x0D

低地址方向

数据以 16bit 为单位

高地址方向

0x0A0B

0x0C0D

低地址方向

其中，最低有效位(LSB，Least Significant Byte)0x0D 存储在最低的内存地址处。后面字节依次存在后面的地址处。同时，最低的 16bit 单元 0x0A0B 存储在低位。

可见，小字节序就是“高字节存入高地址，低字节存入低地址”。

C 语言中的位域结构也要遵循比特序(类似字节序)。例如：

```
1 struct bitfield{
2        unsigned char a: 2;
3        unsigned char b: 6;
4 }
```

该位域结构占 1 个字节，假设赋值 a = 0x01 和 b=0x02，则大字节机器上该字节为(01)(000010)，小字节机器上该字节为(000010)(01)。因此在编写可移植代码时，需要加条件编译。

注意，在包含位域的 C 结构中，若位域 A 在位域 B 之前定义，则位域 A 所占用的内存空间地址低于位域 B 所占用的内存空间。

对上述问题，详细的讲解可参考 http://www.linuxjournal.com/article/6788。

另见以下联合体，在小字节机器上若 low=0x01，high=0x02，则 hex=0x21：

复制代码

```
 1 int main(void){
 2        union{
 3               unsigned char hex;
 4               struct{
 5                      unsigned char low    : 4;
 6                      unsigned char high : 4;
 7               };
 8        }convert;
 9        convert.low = 0x01;
10        convert.high = 0x02;
11        printf("hex = 0x%0x\n", convert.hex);
12        return 0;
13 }
```

复制代码

5.1.2 网络序

网络传输一般采用大字节序，也称为网络字节序或网络序。IP 协议中定义大字节序为网络字节序。

对于可移植的代码来说，将接收的网络数据转换成主机的字节序是必须的，一般会有

成对的函数用于把网络数据转换成相应的主机字节序或反之(若主机字节序与网络字节序相同，通常将函数定义为空宏)。

伯克利 socket API 定义了一组转换函数，用于 16 和 32 位整数在网络序和主机字节序之间的转换。Htonl、htons 用于主机序转换到网络序；ntohl、ntohs 用于网络序转换到本机序。

注意：在大小字节序转换时，必须考虑待转换数据的长度(如 5.1.1 节的数据单元)。另外对于单字符或小于单字符的几个 bit 数据，是不必转换的，因为在机器存储和网络发送的一个字符内的 bit 位存储顺序是一致的。

5.1.3 位序

用于描述串行设备的传输顺序。一般硬件传输采用小字节序(先传低位)，但 I2C 协议采用大字节序。网络协议中只有数据链路层的底端会涉及到。

5.1.4 处理器字节序

不同处理器体系的字节序如下所示：

X86、MOS Technology 6502、Z80、VAX、PDP-11 等处理器为 Little endian；
Motorola 6800、Motorola 68000、PowerPC 970、System/370、SPARC(除 V9 外)等处理器为 Big endian；
ARM、PowerPC (除 PowerPC 970 外)、DEC Alpha，SPARC V9，MIPS，PA-RISC and IA64 等的字节序是可配置的。

5.1.5 字节序编程

请看下面的语句：

```
1 printf("%c\n", *((short*)"AB") >> 8);
```

在大字节序下输出为'A'，小字节序下输出为'B'。

下面的代码可用来判断本地机器字节序：

复制代码

```
 1 //字节序枚举类型
 2 typedef enum{
 3         ENDIAN_LITTLE = (INT8U)0X00,
 4         ENDIAN_BIG        = (INT8U)0X01
 5 }E_ENDIAN_TYPE;
 6
 7 E_ENDIAN_TYPE GetEndianType(VOID)
 8 {
 9         INT32U dwData = 0x12345678;
10
11         if(0x78 == *((INT8U*)&dwData))
12                 return ENDIAN_LITTLE;
```

```
13          else
14                  return ENDIAN_BIG;
15 }
16
17 //Start of GetEndianTypeTest//
18 #include <endian.h>
19 VOID GetEndianTypeTest(VOID)
20 {
21 #if _BYTE_ORDER == _LITTLE_ENDIAN
22          printf("[%s]<Test Case> Result: %s, EndianType = %s!\n", __FUNCTION__,
23                  (ENDIAN_LITTLE != GetEndianType()) ? "ERROR" : "OK", "Little");
24 #elif _BYTE_ORDER == _BIG_ENDIAN
25          printf("[%s]<Test Case> Result: %s, EndianType = %s!\n", __FUNCTION__,
26                  (ENDIAN_BIG != GetEndianType()) ? "ERROR" : "OK", "Big");
27 #endif
28 }
29 //End of GetEndianTypeTest//
```

复制代码

在字节序不同的平台间的交换数据时，必须进行转换。比如对于 int 类型，大字节序写入文件：

```
1 int i = 100;
2 write(fd, &i, sizeof(int));
```

小字节序读出后：

复制代码

```
 1 int i;
 2 read(fd, &i, sizeof(int));
 3 char buf[sizeof(int)];
 4 memcpy(buf, &i, sizeof(int));
 5 for(i = 0; i < sizeof(int); i++)
 6 {
 7          int v = buf[sizeof(int) - i - 1];
 8          buf[sizeof(int) - 1] =    buf[i];
 9          buf[i] = v;
10 }
11 memcpy(&i, buf, sizeof(int));
```

复制代码

上面仅仅是个例子。在不同平台间即使不存在字节序的问题，也尽量不要直接传递二进制数据。作为可选的方式就是使用文本来交换数据，这样至少可以避免字节序的问题。

很多的加密算法为了追求速度，都会采取字符串和数字之间的转换，在计算完毕后，必须注意字节序的问题，在某些实现中可以见到使用预编译的方式来完成，这样很不方便，如果使用前面的语句来判断，就可以自动适应。

字节序问题不仅影响异种平台间传递数据，还影响诸如读写一些特殊格式文件之类程序的可移植性。此时使用预编译的方式来完成也是一个好办法。

5.2 对齐时的填充字节

代码如下：

复制代码

```
 1 struct A{
 2         char    c;
 3         int     i;
 4         short s;
 5 };
 6 int main(void){
 7         struct A a;
 8         a.c = 1; a.i = 2; a.s = 3;
 9         printf("sizeof(A)=%d\n", sizeof(struct A));
10         return 0;
11 }
```

复制代码

执行后输出为 sizeof(A)=12。

VC6.0 环境中，在 main 函数打印语句前设置断点，执行到断点处时根据结构体 a 的地址查看变量存储如下：

可见填充字节为 0xCC，即 int3 中断。

5.3 pragma pack 语法说明

#pragma pack(n)

#pragma pack(push, 1)

#pragma pack(pop)

1) #pragma pack(n)

该指令指定结构和联合成员的紧凑对齐。而一个完整的转换单元的结构和联合的紧凑对齐由/ Z p 选项设置。紧凑对齐用 pack 编译指示在数据说明层设置。该编译指示在其出现后的第一个结构或者联合说明处生效。该编译指示对定义无效。

当使用#pragma pack (n) 时，n 为 1、2、4、8 或 1 6 。第一个结构成员后的每个结构成员都被存储在更小的成员类型或 n 字节界限内。如果使用无参量的#pragma pack，结构成

员被紧凑为以/ Z p 指定的值。该缺省/ Z p 紧凑值为/ Z p 8。

2. 编译器也支持以下增强型语法：

#pragma pack([[{ push | pop } ,] [identifier,]] [n])

若不同的组件使用 pack 编译指示指定不同的紧凑对齐, 这个语法允许你把程序组件组合为一个单独的转换单元。

带 push 参量的 pack 编译指示的每次出现将当前的紧凑对齐存储到一个内部编译器堆栈中。编译指示的参量表从左到右读取。如果使用 push，则当前紧凑值被存储起来；如果给出一个 n 值，该值将成为新的紧凑值。若指定一个标识符，即选定一个名称，则该标识符将和这个新的的紧凑值联系起来。

带一个 pop 参量的 pack 编译指示的每次出现都会检索内部编译器堆栈顶的值，并使该值为新的紧凑对齐值。如果使用 pop 参量且内部编译器堆栈是空的,则紧凑值为命令行给定的值，并将产生一个警告信息。若使用 pop 且指定一个 n 值，该值将成为新的紧凑值。

若使用 pop 且指定一个标识符，所有存储在堆栈中的值将从栈中删除，直到找到一个匹配的标识符。这个与标识符相关的紧凑值也从栈中移出，并且这个仅在标识符入栈之前存在的紧凑值成为新的紧凑值。如果未找到匹配的标识符, 将使用命令行设置的紧凑值，并且将产生一个一级警告。缺省紧凑对齐为 8。

pack 编译指示的新的增强功能让你在编写头文件时，确保在遇到该头文件的前后的紧凑值是一样的。

5.4 Intel 关于内存对齐的说明

以下内容节选自《Intel Architecture 32 Manual》。

字、双字和四字在自然边界上不需要在内存中对齐。(对于字、双字和四字来说，自然边界分别是偶数地址，可以被 4 整除的地址，和可以被 8 整除的地址。)

无论如何，为了提高程序的性能，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；然而，对齐的内存访问仅需要一次访问。

一个字或双字操作数跨越了 4 字节边界，或者一个四字操作数跨越了 8 字节边界，被认为是未对齐的，从而需要两次总线周期来访问内存。一个字起始地址是奇数但却没有跨越字边界被认为是对齐的，能够在一个总线周期中被访问。

某些操作双四字的指令需要内存操作数在自然边界上对齐。如果操作数没有对齐，这些指令将会产生一个通用保护异常(#GP)。双四字的自然边界是能够被 16 整除的地址。其他操作双四字的指令允许未对齐的访问(不会产生通用保护异常)，然而，需要额外的内存总线周期来访问内存中未对齐的数据。

5.5 不同架构处理器的对齐要求

RISC 指令集处理器(MIPS/ARM)：这种处理器的设计以效率为先，要求所访问的多字节数据(short/int/ long)的地址必须是为此数据大小的倍数，如 short 数据地址应为 2 的倍数，long 数据地址应为 4 的倍数，也就是说是对齐的。

CISC 指令集处理器(X86)：没有上述限制。

对齐处理策略

访问非对齐多字节数据时(pack 数据)，编译器会将指令拆成多条(因为非对齐多字节数据可能跨越地址对齐边界)，保证每条指令都从正确的起始地址上获取数据，但也因此效率比较低。

访问对齐数据时则只用一条指令获取数据，因此对齐数据必须确保其起始地址是在对齐边界上。如果不是在对齐的边界，对 X86 CPU 是安全的，但对 MIPS/ARM 这种 RISC CPU 会出现“总线访问异常”。

为什么 X86 是安全的呢？

X86 CPU 是如何进行数据对齐的。X86 CPU 的 EFLAGS 寄存器中包含一个特殊的位标志，称为 AC(对齐检查的英文缩写)标志。按照默认设置，当 CPU 首次加电时，该标志被设置为 0。当该标志是 0 时，CPU 能够自动执行它应该执行的操作，以便成功地访问未对齐的数据值。然而，如果该标志被设置为 1，每当系统试图访问未对齐的数据时，CPU 就会发出一个 INT 17H 中断。X86 的 Windows 2000 和 Windows 98 版本从来不改变这个 CPU 标志位。因此，当应用程序在 X86 处理器上运行时，你根本看不到应用程序中出现数据未对齐的异常条件。

为什么 MIPS/ARM 不安全呢？

因为 MIPS/ARM CPU 不能自动处理对未对齐数据的访问。当未对齐的数据访问发生时，CPU 就会将这一情况通知操作系统。这时，操作系统将会确定它是否应该引发一个数据未对齐异常条件，对 vxworks 是会触发这个异常的。

5.6 ARM 下的对齐处理

有部分摘自 ARM 编译器文档对齐部分。

对齐的使用：

1) __align(num)

用于修改最高级别对象的字节边界。在汇编中使用 LDRD 或 STRD 时就要用到此命令__align(8)进行修饰限制。来保证数据对象是相应对齐。

这个修饰对象的命令最大是 8 个字节限制，可以让 2 字节的对象进行 4 字节对齐，但不能让 4 字节的对象 2 字节对齐。

__align 是存储类修改，只修饰最高级类型对象，不能用于结构或者函数对象。

2) __packed

进行一字节对齐。需注意：

不能对 packed 的对象进行对齐；
所有对象的读写访问都进行非对齐访问；
float 及包含 float 的结构联合及未用__packed 的对象将不能字节对齐；
__packed 对局部整型变量无影响。
强制由 unpacked 对象向 packed 对象转化时未定义。整型指针可以合法定义为 packed，如__packed int* p(__packed int 则没有意义)

对齐或非对齐读写访问可能存在的问题：

复制代码

```
1 //定义如下结构，b 的起始地址不对齐。在栈中访问 b 可能有问题，因为栈上数据对齐访问
2 __packed struct STRUCT_TEST{
3       char a;
4       int    b;
5       char c;
6 };
7 //将下面的变量定义成全局静态(不在栈上)
8 static char *p;
9 static struct STRUCT_TEST a;
10 void Main(){
11      __packed int *q; //定义成__packed 来修饰当前 q 指向为非对齐的数据地址下面的访问则可以
12
13      p = (char*)&a;
14      q = (int*)(p + 1);
15      *q = 0x87654321;
16      /*  得到赋值的汇编指令很清楚
17      ldr         r5,0x20001590 ; = #0x12345678
18      [0xe1a00005]    mov         r0,r5
19      [0xeb0000b0]    bl          __rt_uwrite4    //在此处调用一个写 4 字节的操作函数
20
21      [0xe5c10000]    strb        r0,[r1,#0]      //函数进行 4 次 strb 操作然后返回，正确访问数据
22      [0xe1a02420]    mov         r2,r0,lsr #8
23      [0xe5c12001]    strb        r2,[r1,#1]
```

```
24      [0xe1a02820]    mov       r2,r0,lsr #16
25      [0xe5c12002]    strb      r2,[r1,#2]
26      [0xe1a02c20]    mov       r2,r0,lsr #24
27      [0xe5c12003]    strb      r2,[r1,#3]
28      [0xe1a0f00e]    mov       pc,r14
29
30      若 q 未加__packed 修饰则汇编出来指令如下(会导致奇地址处访问失败)：
31      [0xe59f2018]    ldr       r2,0x20001594 ; = #0x87654321
32      [0xe5812000]    str       r2,[r1,#0]
33      */
34      //这样很清楚地看到非对齐访问如何产生错误，以及如何消除非对齐访问带来的问
题
35      //也可看到非对齐访问和对齐访问的指令差异会导致效率问题
36 }
```

复制代码

5.7 《The C Book》之位域篇

While we're on the subject of structures, we might as well look at bitfields. They can only be declared inside a structure or a union, and allow you to specify some very small objects of a given number of bits in length. Their usefulness is limited and they aren't seen in many programs, but we'll deal with them anyway. This example should help to make things clear:

复制代码

```
1 struct{
2       unsigned field1 :4; //field 4 bits wide
3       unsigned        :3; //unnamed 3 bit field(allow for padding)
4       signed field2   :1; //one-bit field(can only be 0 or -1 in two's complement)
5       unsigned        :0; //align next field on a storage unit
6       unsigned field3 :6;
7 }full_of_fields;
```

复制代码

Each field is accessed and manipulated as if it were an ordinary member of a structure. The keywords signed and unsigned mean what you would expect, except that it is interesting to note that a 1-bit signed field on a two's complement machine can only take the values 0 or -1. The declarations are permitted to include the const and volatile qualifiers.

The main use of bitfields is either to allow tight packing of data or to be able to specify the fields within some externally produced data files. C gives no guarantee of the ordering of fields within machine words, so if you do use them for the latter reason, you program will not only be non-portable, it will be compiler-dependent too. The Standard says that fields are packed into ‘storage units’, which are typically machine words. The packing order, and whether or not a bitfield may cross a storage unit boundary, are implementation defined. To force alignment to a storage unit boundary, a zero width field is used before the one that you want to have aligned.

Be careful using them. It can require a surprising amount of run-time code to manipulate

these things and you can end up using more space than they save.

Bit fields do not have addresses—you can't have pointers to them or arrays of them.

5.8 C 语言字节相关面试题

5.8.1 Intel/微软 C 语言面试题

请看下面的问题：

复制代码

```
1 #pragma pack(8)
2 struct s1{
3       short a;
4       long    b;
5 };
6 struct s2{
7       char c;
8       s1      d;
9       long long e;    //VC6.0 下可能要用__int64 代替双 long
10 };
11 #pragma pack()
```

复制代码

问：1. sizeof(s2) = ？ 2. s2 的 s1 中的 a 后面空了几个字节接着是 b？

【分析】

成员对齐有一个重要的条件，即每个成员分别按自己的方式对齐。

也就是说上面虽然指定了按 8 字节对齐，但并不是所有的成员都是以 8 字节对齐。其对齐的规则是：每个成员按其类型的对齐参数(通常是这个类型的大小)和指定对齐参数(这里是 8 字节)中较小的一个对齐，并且结构的长度必须为所用过的所有对齐参数的整数倍，不够就补空字节。

s1 中成员 a 是 1 字节，默认按 1 字节对齐，而指定对齐参数为 8，两值中取 1，即 a 按 1 字节对齐；成员 b 是 4 个字节，默认按 4 字节对齐，这时就按 4 字节对齐，所以 sizeof(s1) 应该为 8；

s2 中 c 和 s1 中 a 一样，按 1 字节对齐。而 d 是个 8 字节结构体，其默认对齐方式就是所有成员使用的对齐参数中最大的一个，s1 的就是 4。所以，成员 d 按 4 字节对齐。成员 e 是 8 个字节，默认按 8 字节对齐，和指定的一样，所以它对到 8 字节的边界上。这时，已经使用了 12 个字节，所以又添加 4 个字节的空，从第 16 个字节开始放置成员 e。此时长度为 24，并可被 8(成员 e 按 8 字节对齐)整除。这样，一共使用了 24 个字节。

各个变量在内存中的布局为：

c***aa**

bbbb****

dddddddd　　——这种“矩阵写法”很方便看出结构体实际大小！

因此，sizeof(S2)结果为 24，a 后面空了 2 个字节接着是 b。

这里有三点很重要：

1) 每个成员分别按自己的方式对齐，并能最小化长度；

2) 复杂类型(如结构)的默认对齐方式是其最长的成员的对齐方式，这样在成员是复杂类型时可以最小化长度；

3) 对齐后的长度必须是成员中最大对齐参数的整数倍，这样在处理数组时可保证每一项都边界对齐。

还要注意，“空结构体”(不含数据成员)的大小为 1，而不是 0。试想如果不占空间的话，一个空结构体变量如何取地址、两个不同的空结构体变量又如何得以区分呢？

5.8.2 上海网宿科技面试题

假设硬件平台是 intel x86(little endian)，以下程序输出什么：

复制代码

```
1 //假设硬件平台是 intel x86(little endian)
2 typedef unsigned int uint32_t;
3 void inet_ntoa(uint32_t in){
4         char    b[18];
5         register    char    *p;
6         p = (char *)&in;
7 #define UC(b) (((int)b)&0xff) //byte 转换为无符号 int 型
8         sprintf(b, "%d.%d.%d.%d\n", UC(p[0]), UC(p[1]), UC(p[2]), UC(p[3]));
9         printf(b);
10 }
11 int main(void){
12        inet_ntoa(0x12345678);
13        inet_ntoa(0x87654321);
14        return 0;
15 }
```

复制代码

先看如下程序：

复制代码

```
1 int main(void){
2       int a = 0x12345678;
3       char *p = (char *)&a;
4       char str[20];
5       sprintf(str,"%d.%d.%d.%d\n", p[0], p[1], p[2], p[3]);
6       printf(str);
7       return 0;
8 }
```

复制代码

按照小字节序的规则，变量 a 在计算机中存储方式为：

高地址方向

0x12

0x34

0x56

0x78

低地址方向

p[3]

p[2]

p[1]

p[0]

注意，p 并不是指向 0x12345678 的开头 0x12，而是指向 0x78。p[0]到 p[1]的操作是 &p[0]+1，因此 p[1]地址比 p[0]地址大。输出结果为 120.86.52.18。

反过来的话，令 int a = 0x87654321，则输出结果为 33.67.101.-121。

为什么有负值呢？因为系统默认的 char 是有符号的，本来是 0x87 也就是 135，大于 127 因此就减去 256 得到-121。

想要得到正值的话只需将 char *p = (char *)&a 改为 unsigned char *p = (unsigned char *)&a 即可。

综上不难得出，网宿面试题的答案为 120.86.52.18 和 33.67.101.135。