C 语言字节对齐详解_czw-CSDN 博客_字节对齐

C语言字节对齐[12345]

不同系统下的C语言类型长度

Data Type	ILP32	ILP64	LP64	LLP64
char	8	8	8	8
short	16	16	16	16
int	32	64	32	32
long	32	64	64	32
long long	64	64	64	64
pointer	32	64	64	64

绝大部分64位的Unix，linux都是使用的LP64模型；32位Linux系统是ILP32模型；64位的Windows使用的是LLP64(long long and point 64)模型。

基本概念

许多计算机系统对基本数据类型合法地址做出了一些限制，要求某种类型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形成处理器和存储器系统之间的接口的硬件设计。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。

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。例如，假设一个处理器总是从存储器中取出8个字节，则地址必须为8的倍数。如果我们能保证将所有的double类型数据的地址对齐成8的倍数，那么就可以用一个存储器操作来读或者写值了。否则，我们可能需要执行两次存储器访问，因为对象可能被分放在两个8字节存储块中。

另外，假设一个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。

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。比如sparc系统，如果取未对齐的数据会发生错误，举个例：

```
1 char ch[8];
2 char *p = &ch[1];
3 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。

如何处理字节对齐

先让我们看编译器是按照什么样的原则进行对齐的：

1. 数据类型自身的对齐值：为指定平台上基本类型的长度。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。

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。

当数据类型为结构体时，编译器可能需要在结构体字段的分配中插入间隙，以保证每个结构元素都满足它的对齐要求。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（对于非对齐成员需要在其前面填充一些字节，保证其在对齐位置上），结构体本身也要根据自身的有效对齐值圆整(就是结构体总长度需要是结构体有效对齐值的整数倍)，此时可能需要在结构末尾填充一些空间，以满足结构体整体的对齐—向结构体元素中最大的元素对齐。

通过上面的分析，对结构体进行字节对齐，我们需要知道四个值：

- 指定对齐值：代码中指定的对齐值，记为packLen；
- 默认对齐值：结构体中每个数据成员及结构体本身都有默认对齐值，记为defaultLen；
- 成员偏移量：即相对于结构体起始位置的长度，记为offset；
- 成员长度：结构体中每个数据成员的长度（注结构体成员为补齐之后的长度），记为memberLen。

及两个规则：

1. 对齐规则：`offset % vaildLen = 0`，其中vaildLen为有效对齐值`vaildLen = min(packLen, defaultLen)`；
2. 填充规则：如成员变量不遵守对齐规则，则需要对其补齐；在其前面填充一些字节保证该成员对齐。需填充的字节数记为pad

二.Microsoft Windows的对齐策略:

在Windows中对齐要求更严–任何K字节基本对象的地址都必须是K的倍数，K=2，4，或者8.特别地，double或者long long类型数据的地址应该是8的倍数。可以看出Windows的对齐策略和Linux还是不同的。

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。

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。

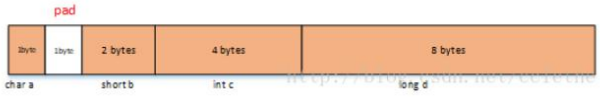
对于32位机来说，4字节对齐能够使cpu访问速度提高，比如说一个long类型的变量，如果跨越了4字节边界存储，那么cpu要读取两次，这样效率就低了。但是在32位机中使用1字节或者2字节对齐，反而会使变量访问速度降低。所以这要考虑处理器类型，另外还得考虑编译器的类型。在vc中默认是4字节对齐的，GNU gcc 也是默认4字节对齐。

结构体举例

例子1

```c
/************************
    > File Name: struct_test.c
    > Author:Marvin
    > Created Time: Thu 22 Mar 2018 07:19:46 PM CST
 ***********************/

#include<stdio.h>


int main()
{
    struct test {
        char a;
        short b;
        int c;
        long d;
    };
    struct test t = {'a',11,11,11};

    printf("size of struct t = %u\n", sizeof(t));

    return 0;
}
```

在64位centos上编译编译后结构struct test的布局如下:



由于要保证结构体每个元素都要数据对齐，因此必须在a和b之间插入1字节的间隙使得后面的short元素2字节对齐int元素4字节对齐long元素8字节对齐，这样最终test结构大小为16字节。

运行程序结果为:

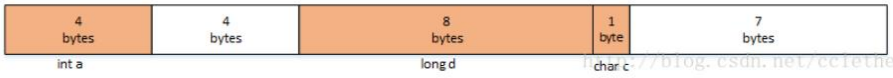
```
size of struct t = 16
```

例子2

现在考虑这样一个结构体:

```c
struct test2 {
    int a;
    long b;
    char c;
};
struct test2 t2 = {11,11,'c'};
```

在64位centos上编译编译后结构struct test2的布局如下:



结构体struct test2的自然对界条件为8字节，所以需要在最后的char型数据后面再填充7个字节使得结构体整体对齐。

运行程序结构为

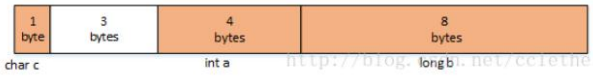
```
size of struct test2 = 24
```

例子3

不妨将结构体struct test2里面成员的顺序重新排列一下：

```
struct test3 {
    char c;
    int a;
    long b;
};
struct test3 t3 = {'c',11,11};
```

在64位centos上编译编译后结构struct test2的布局如下：



运行结果为：

```
size of struct test3 = 16
```

可见适当地编排结构体成员地顺序，可以在保存相同信息地情况下尽可能节约内存空间。

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.