C结构体大小计算总结

陈鸿峥

1 基本概念

以下几个单位需要懂得区分，字的概念可以暂时不理。

- 位(bit/b)：计算机处理、存储、传输信息的最小单位
- 字节(Byte/B) 1 Byte = 8 bit：现代计算机主存按**字节编址**，字节是最小可寻址单位
- 字(Word)：表示被处理信息的单位，用来度量数据类型的宽度*

一台32位的电脑，一个字等于4个字节，字长为32位。

C语言的`sizeof`返回的是数据类型的**字节**大小，如`sizeof(int)=4`，而不是32，不要搞混了。

2 结构体大小的计算

结构体的大小不是简单的内部数据大小之后，而是要依据一些基本原则进行存储：

1. 数据成员对齐规则：结构体的**第一个**数据成员放在偏移量(offset)为0的地方，之后每个数据成员存储的起始位置要从**该**成员大小的整数倍开始
2. 结构体作为成员：如果一个结构里有某些结构体成员，则结构体成员要从其内部最大元素大小的整数倍地址开始存储
3. 收尾工作：结构体的总大小，必须是其内部最大成员的整数倍，不足的要补齐

上次的例子解释有误，在此重新解释。

例 1.
```
struct point
{
    int index; // [0]...[3]
    double x; // [8]...[15]
    double y; // [16]...[24]
};
```

分析.
- *index*占*4*字节。
- *x*由原则*1*从*double*大小*(8)*的整数倍*(8)*开始存储，占*8*字节；*index*的后四字节补齐。
- *y*占*8*字节。
- 故一共*24*字节，且不因*32*位机和*64*位机而异。

*字长是指CPU中**数据通路的宽度**，也指计算机一次能处理的二进制的长度，等于CPU内部总线的宽度或运算器的位数或通用寄存器的宽度；字和字长的宽度可以一样，也可以不同，通常是字节的整数倍

例 2. ***struct*** *data*

```
{
    int id;          // [0]...[3]
    double weight;   // [8]...[15]
    float height;    // [16]...[19]
};
```

分析.
- *id*占*4*字节。
- *weight*由原则*1*从*double*大小*(8)*的整数倍*(8)*开始存储，占*8*字节。
- *height*占接下来*4*字节。
- 最后由原则*3*，总大小要为最大成员*double*的整数倍*(24)*，补齐最后*4*个字节，总共*24*字节。

例 3. ***struct*** *data2*

```
{
    char name[2];    // [0],[1]
    int id;          // [4]...[7]
    double score;    // [8]...[15]
    short grade;     // [16],[17]
    struct data d;   // [24].....[47]
    int last;        // [48]...[51]
};
```

分析.
- *char*占*1*字节，数组则×*2*。
- 由原则*1*，*id*从*int*大小*(4)*的整数倍*(4)*开始存储，占*4*字节。
- 由原则*1*，*score*从*double*大小*(8)*的整数倍*(8)*开始存储，占*8*字节。
- *short*占*2*字节。
- 由原则*2*，*struct*要从内部元素最大的一个*(double)*的整数倍*(24)*开始存放，由例*2*占*24*字节。
- *int*占接下来*4*字节。
- 由原则*3*，总大小要为最大成员*double*（单独考虑数组和结构体内元素）的整数倍，故总共*56*字节。

例 4. ***struct*** *s1*

```
{
    char c1; // [0]
    int i;   // [4]...[7]
    char c2; // [8]
};

struct s2
{
    char c1; // [0]
    char c2; // [1]
    int i;   // [4]...[7]
};
```

分析. 同理之前的分析，*s1*大小为*12*，*s2*大小为*8*，由本例中可以看出结构体内数据的顺序也是会影响结构体的大小的。

3 位域

位域用来**显性地**声明一个数据的存储**位**大小，声明方法如下：

```
<type><variable>:<bit size>
```

如`int a: 1`表示只用1位存储int类型。

涉及到位域(bit field)的问题则会更加复杂，但这很大程度上与编译器的实现有关。如按照百度百科上给出的VC的准则是：

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大小，则后面的字段将从新的存储单元开始，其偏移量为该类型大小的整数倍
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（不同位域字段存放在不同的位域类型字节中），Dev-C++和GCC都采取压缩方式

例 5.
```
// A structure without forced alignment
struct test1
{
   unsigned int x: 5;
   unsigned int y: 8;
};

// A structure with forced alignment
struct test2
{
   unsigned int x: 5; //[0]...[3]
   unsigned int: 0;
   unsigned int y: 8; //[4]...[8]
};
```

分析. 前者相邻字段类型相同，紧邻存储，共*13*位，向上补齐为*4*字节*(32*位*)*；后者多了一个*0*位域，用来强制对齐边界，即*x*存完*5*位后强制对齐为*32*位，而后从第*4*个字节开始存储*y*，故一共*8*个字节。

例 6.
```
struct test
{
    int a : 20;           // [0]...[2]
    char b : 3;           // [3]
    char c : 0;           // new [4]
    unsigned char c : 5; // [4],[5]
};
```

分析.
- *a*占*20*位，向上补齐即*3*字节*(24*位*)*
- *int*和*char*不同类型，不压缩，新开一个字节存储
- *0*位域相当于新开一个字节，用来强制对齐边界
- *c*占*5*位，向上补齐即*2*字节*(8*位*)*
- 由原则*3*，*int*的整数倍，故一共*8*字节

例 7. ***struct*** *test*

```
{
    int a : 1;
    int b : 2;
    inc c : 4;
    ind d : 4;
};
```

分析. *a,b,c,d*都可以被压缩存储，共占*11*位，向上对齐为*4*字节*(32*位*)*

关于位域的其他小知识

- 由第1部分提到的，现在的机器都采用字节编址，而位域的加入，使得数据的地址可能不是字节的整数倍，因而也就不能用指针进行访问，如`&test.a`是被禁止的。
- 对于超出位域大小的赋值，编译器往往会报error或warning，如a字段仅1位，赋值`a=2`会被警告
- 对于超出位域大小的声明，编译器同样会报error或warning，如`int a : 33`是不被允许的
- 位域不能被声明为静态变量，也不能是数组

4 总结

上面所展示的内容很多都是与编译器具体实施相关的，记住基本原则，然后不要太较真就好。遇到不确定的情况，最好自己写程序验证一下，在自己机器上跑跑就知道结果了。面向32位机器的编译可以在编译时添加`-m32`指令，面向64位机器则添加`-m64`，这样就可以在一台机子上同时验证两种情况。

5 参考资料

1. c/c++ struct的大小以及sizeof用法, https://www.cnblogs.com/dingxiaoqiang/p/8059329.html
2. Bit field, https://en.cppreference.com/w/cpp/language/bit_field
3. 位域, https://baike.baidu.com/item/%E4%BD%8D%E5%9F%9F/9215688?fr=aladdin
4. Bit Fields in C, https://www.geeksforgeeks.org/bit-fields-c/