Levphy的博客 (/)

(/) 程序猿人生路上的精彩片段

内存对齐规则之我见

2017年03月23日 星期四, 发表于 武汉

如果你对本文有任何的建议或者疑问, 可以在 这里给我提 Issues (https://github.com/levphy/levphy.github.io/issues), 谢谢! :)

内存对齐的原因和理由就不多说了，主要是为了性能和平台移植等因素，编译器对数据结构进行了内存对齐

考虑下面的实例:

```
#include<iostream>
using namespace std;
struct A{
    char a;
    int b;
    short c;
};

struct B{
    short c;
    char a;
    int b;
};
int main(){
    cout<<sizeof(A)<<endl;
    cout<<sizeof(B)<<endl;
    return 0;
}
```

上面两个结构体A和B成员变量类型相同，但是占用的内存空间大小(单位:字节)却不一样

sizeof(A) = 12

sizeof(B) = 8

为了分析造成这种现象的原因，我们不得不提及内存对齐的3大规则:

1. **对于结构体的各个成员，第一个成员的偏移量是0，排列在后面的成员其当前偏移量必须是当前成员类型的整数倍**

2. **结构体内所有数据成员各自内存对齐后，结构体本身还要进行一次内存对齐，保证整个结构体占用内存大小是结构体内最大数据成员的最小整数倍**
3. **如程序中有#pragma pack(n)预编译指令，则所有成员对齐以n字节为准(即偏移量是n的整数倍)，不再考虑当前类型以及最大结构体内类型**

以上面结构体A为例，第一个成员a是char类型，占用1个字节空间，偏移量为0，第二个成员b是int类型，占用4个字节空间，按照规则1，b的偏移量必须是int类型的整数倍，所以编译器会在a变量后面插入3字节缓冲区，保证此时b的偏移量(4字节)是b类型的整数倍(当前恰好是1倍)，第3个成员c为short类型，此时c的偏移量正好是4+4=8个字节，已经是short类型的整数倍，故b与c之间不用填充缓冲字节。但这时，结构体A的大小为8+2=10个字节，按照规则2，结构体A大小必须是其最大成员类型int的整数倍，所以在10个字节的基础上再填充2个字节，保证最后结构体大小为12，以符合规则2.

数据成员--前面偏移量-- 成员自身占用

(char) a	0	1
缓冲补齐	1	3(规则1)
(int) b	4	4
(short) c	8	2
缓冲补齐	10	2(规则2)

类似的，结构体B成员的分析如下：

数据成员--前面偏移量--成员自身占用

short c	0	2
char a	2	1
缓冲补齐	3	1(规则1)
int b	4	4

另一个更复杂的例子:

```
struct BU
{
    int number;             //4字节
    union UBffer
    {
        char buffer[13];    //填充3字节，该成员占16字节空间
        int number;
    }ubuf;
    int aa;                 //占4字节空间,当前偏移量已补齐为20
    double dou;             //占8字节空间
}bu;
```

sizeof(BU) = 4 + 13 + 3(补齐) + 4 + 8 = 32，分析方法类似，在计算aa的偏移量时，我们可以肯定的是一定是int类型的整数倍，由于不作任何缓冲补齐的情况下，number + buffer = 17字节，为了符合规则1，需要填充3个字节。

结构体BU稍微变换下aa和dou成员顺序，则结果就大不相同:

```
struct BC
{
    int number;             //4字节
    union UBffer
    {
        char buffer[13];    //填充7字节，该成员占20字节空间
        int number;
    }ubuf;
    double dou;             //占8字节空间，当前偏移量已补齐为24
    int aa;                 //占4字节空间，当前占用空间36字节，最大double类型，还需要根据规则2补
}bu;
```

此时sizeof(BC) = 4 + 13 + 7(规则1补齐) + 8 + 4 + 4(规则2补齐) = 40 (8的整数倍)

我们可能对于结构体类包含union类型成员抱有疑虑，再考虑下面实例：

```
struct BD
{
    short number;
    union UBffer
    {
        char buffer[13];
        int number;
    }ubuf;
}bc;
```

运行结果是sizeof(BD) = 2 + 2 + 13 +3 = 20，可能你会问，为什么不是2+13+1 = 16，这是因为union类型比较特殊，计算union成员的偏移量时，需要根据union内部最大成员类型来进行缓冲补齐，所以为了保证偏移量为union最大成员int类型的整数倍，需要在number(short类型)后面填充2个字节，前面例子中number是int类型，就没有这个必要了。

再比如:

```
struct BE
{
    short number;
    union UBffer
    {
        char buffer[13];
        double number;
    }ubuf;
}bc;
```

它的运行结果是sizeof(BE) = 2 + 6 + 13 + 3 = 24，number后面为了与double类型进行对齐而补齐了6个字节，最后再按照规则2补齐了3个字节

考虑规则3:

举个例子，在#pragma pack(1)时，以1个字节对齐时，属于最简单的情况，结构体大小是所有成员的类型大小的和。所以sizeof(BU) = sizeof(BC) = 29,这时与成员变量顺序不再相关。其他指定的字节对齐也很好分析。一般而言，奇数个字节对齐没有意义，正常情况下，编码人员不关心编译器对内存对齐所作的工作。

上面的例子都想明白之后，内存对齐的规则应该了然于胸了。 :)

PS: C语言中offsetof()函数可用于查看特定的结构体成员在结构体中的偏移量，编程时可以用于验证上面的说法。其实现类似如下:

```
#define offsetof(type, member) (size_t)&(((type *)0)->member)
```

原理是，强制将结构体(类型为type)的起始地址置为0，然后输出其成员的地址，该地址的大小就是成员在结构体中的偏移量。

规则之外的例子

C99中定义了柔性数组机制，因此对于一个结构体，如果最后一个成员是数组的话，结构体大小与该成员是否是柔性数组有密切关系。

```
struct sds{
    unsigned int len;
    unsigned int free;
    char buf[0];//或char buf[]
};
```

当结构体定义中，最后一个成员是数组且数组大小为0或没标记时，该成员数组是柔性数组，不计入结构体大小，因此sizeof(sds) = 8

而下面的结构体sd的sizeof(sd) = 12，因为最后一个数组成员是普通数组，适用于上述补齐规则。

```
struct sds{
    unsigned int len;
    unsigned int free;
    char buf[1];
};
```

我们知道，C++为了兼容C，保留了struct关键字，但是实际上C++中的struct是一个默认访问控制权限为public的class。C++标准规定:一个空类的大小为1个字节，因此在C++中，sizeof(空类或空结构体) = 1，在C语言中，sizeof(空结构体) = 0。

Levphy (/about.html)

levphy@qq.com (mailto:levphy@qq.com)