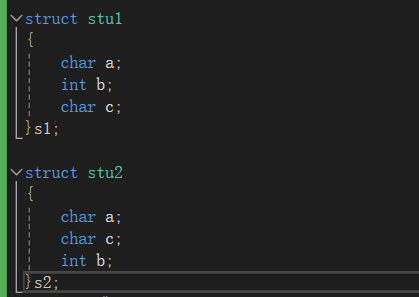
一.结构体的内存对齐

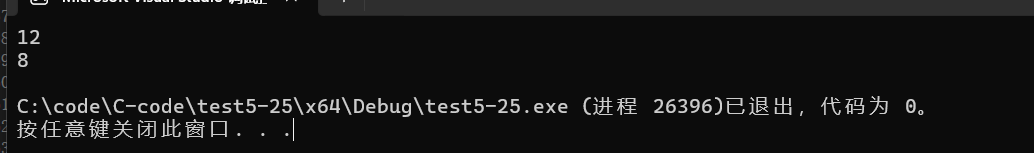
1.1引子

在进行结构体内存大小的计算介绍之前，我们先来看这样一个例子：



如果按照正常的逻辑，s1所占的内存字节数应该为1+4+1 = 6，与s2所占的内存大小相同，

但实际上当我们计算出他们各自的内存大小是，结果如下：



这里计算出s1所占内存的字节大小为12，而s2所占的内存字节大小为8。

为什么会是这样，那让我们从结构体的内存对齐的定义来讲起。

1.2结构体内存对齐的定义及理解

结构体的内存定义如下：

1. 结构体的第⼀个成员对⻬到和结构体变量起始位置偏移量为0的地址处

2. 其他成员变量要对⻬到某个数字（对⻬数）的整数倍的地址处。

对⻬数 = 编译器默认的⼀个对⻬数 与 该成员变量⼤⼩的较⼩值。

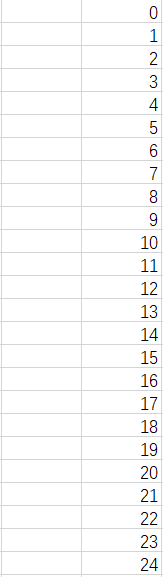
- VS 中默认的值为 8 - Linux中 gcc 没有默认对⻬数，对⻬数就是成员⾃⾝的⼤⼩

3. 结构体总⼤⼩为最⼤对⻬数（结构体中每个成员变量都有⼀个对⻬数，所有对⻬数中最⼤的）的 整数倍。

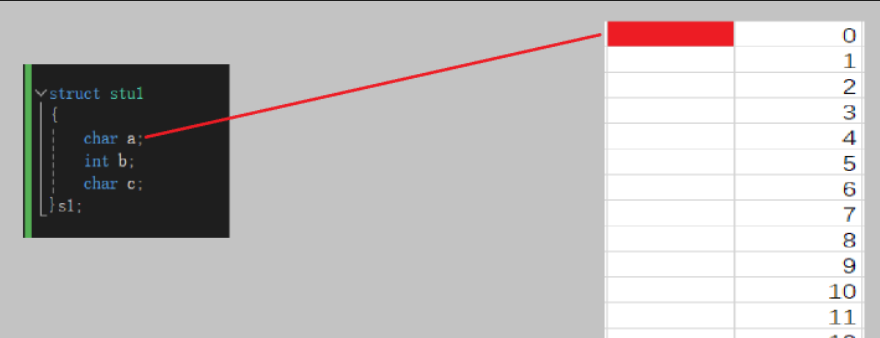
4. 如果嵌套了结构体的情况，嵌套的结构体成员对⻬到⾃⼰的成员中最⼤对⻬数的整数倍处，结构 体的整体⼤⼩就是所有最⼤对⻬数（含嵌套结构体中成员的对⻬数）的整数倍。

那它定义讲的这么多，是什么意思呢，让我们结合引子中的例子来解释。

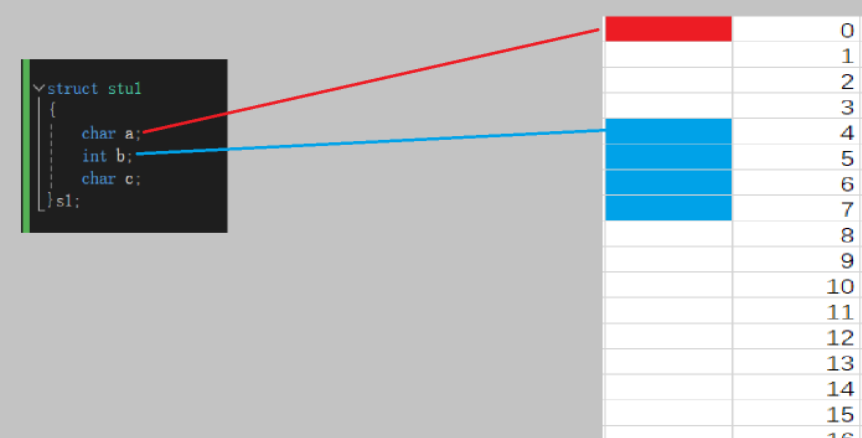
假设有这样一块内存空间如下：



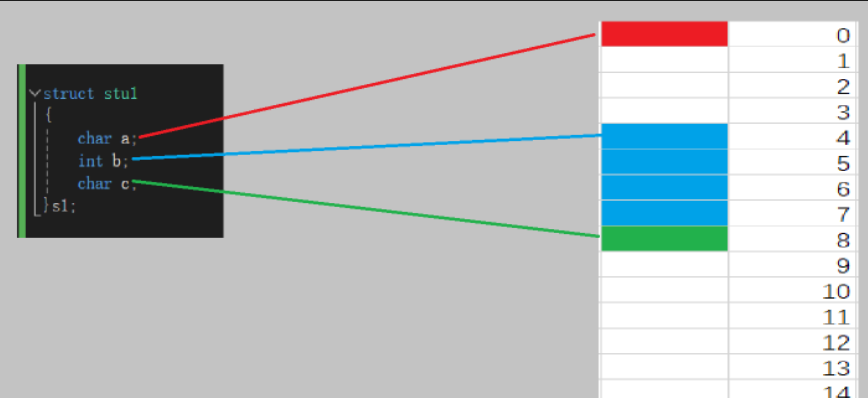
对于s1，首先我们来放它的a,放在起始位置0的位置：



接下来我们来放它的元素b，由于vs默认的对齐数为8，但int类型的变量大小为4所以我们要找到四的倍数位置开始放b，容易找到4便是b的起始位置：



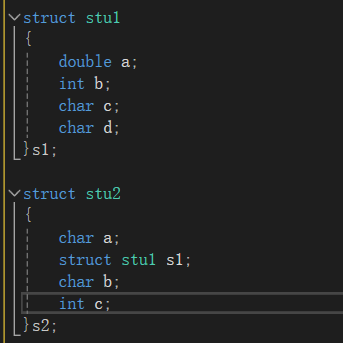
那么它的c元素由于它的大小为1小于8，所以这里我们要找1的倍数作为c元素存放的起始位置：



这时我们来计算它的内存大小，由于此时为9个字节由图而知，但结构体的内存大小为它所有变量中的最大对齐数的倍数来决定，而在本例中为4，所以计算出的s1的字节数为12。

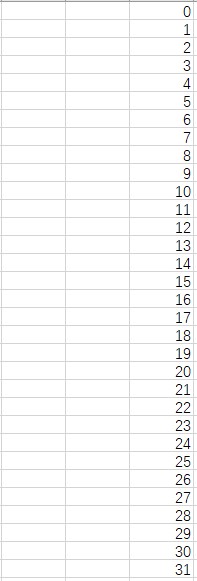
上面s2的计算也是同理，得出最终结果为8。

那么，如果在结构体中又存在结构体呢，也就是结构体嵌套，让我们来看下面一个例子：

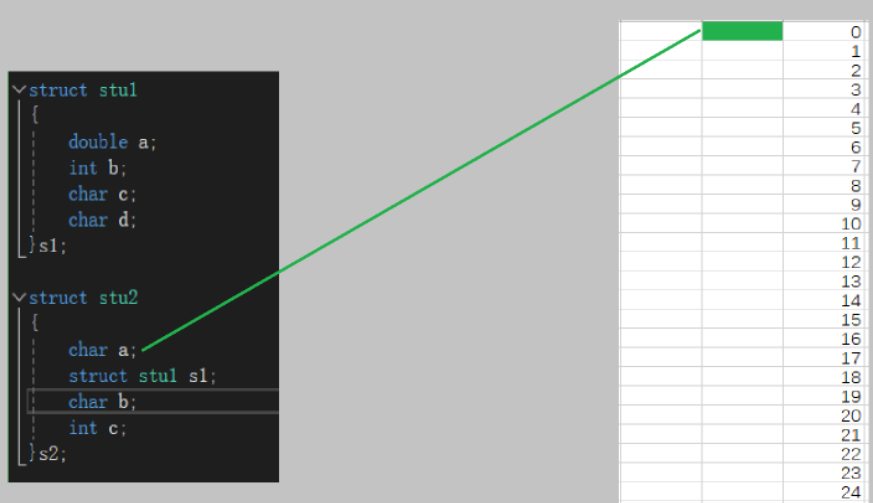


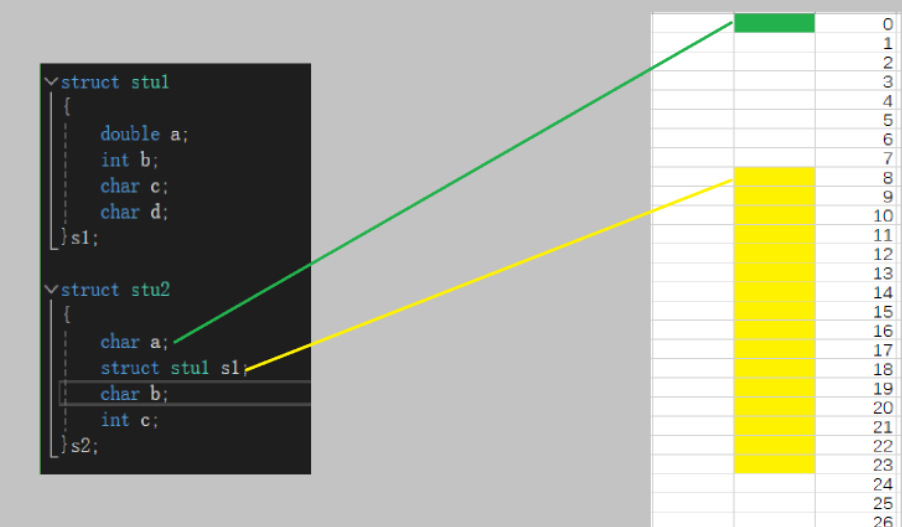
从上面的例子我们不难得知，s1计算出的字节大小为16。那我们如何来算s2呢？

让我们用以下的内存空间来模拟它的储存：

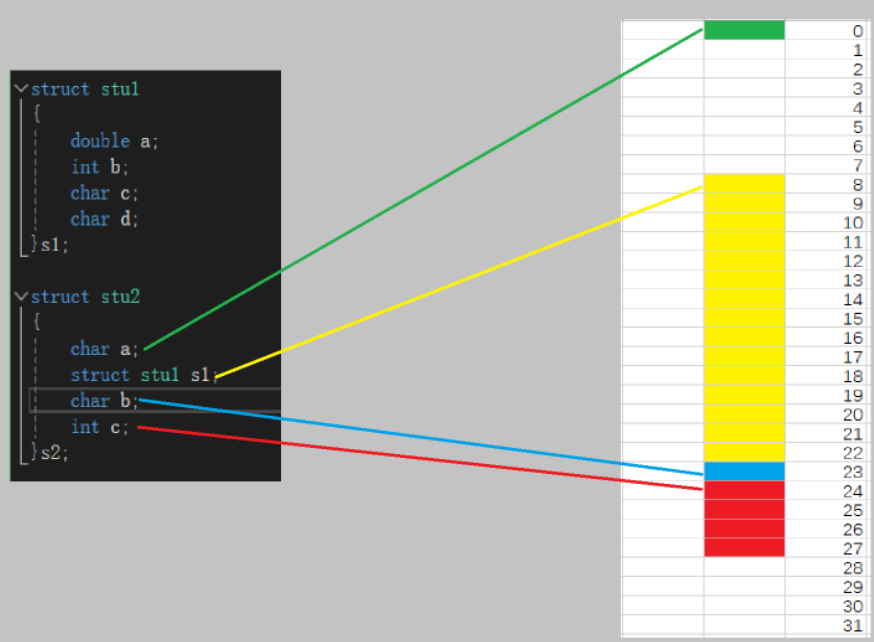


让我们先把a元素放好：



接下来我们来放s1，因为s1中的元素最大对齐数为8（double变量的内存大小为8，而vs的默认对齐数为8，所以s1中最大的为8），所以这里我们要找8的倍数作为起始位置来存放s1：  


下面的内容放置完后如下：



但s2的最大对齐数为8，所以它所占的内存大小为8的倍数，由上图可知应该为32个字节。

不过到这里或许又有人会有疑问，我们可以看到，上面的图解中，浪费了许多空间，那为什么不直接让他们连续储存？这样不能减少内存的占用吗？实际上，原因可总结为一句话：用内存来换取时间，以提高程序运行的效率。

1.3结构体为什么存在内存对⻬?

⼤部分的参考资料都是这样说的：

1. 平台原因 (移植原因)：

不是所有的硬件平台都能访问任意地址上的任意数据的；某些硬件平台只能在某些地址处取某些特定

类型的数据，否则抛出硬件异常。

2. 性能原因：

数据结构(尤其是栈)应该尽可能地在⾃然边界上对⻬。原因在于，为了访问未对⻬的内存，处理器需要

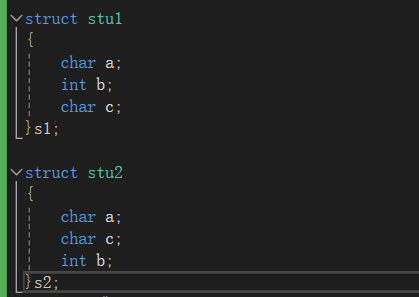
作两次内存访问；⽽对⻬的内存访问仅需要⼀次访问。假设⼀个处理器总是从内存中取8个字节，则地

址必须是8的倍数。如果我们能保证将所有的double类型的数据的地址都对⻬成8的倍数，那么就可以

⽤⼀个内存操作来读或者写值了。否则，我们可能需要执⾏两次内存访问，因为对象可能被分放在两

个8字节内存块中。

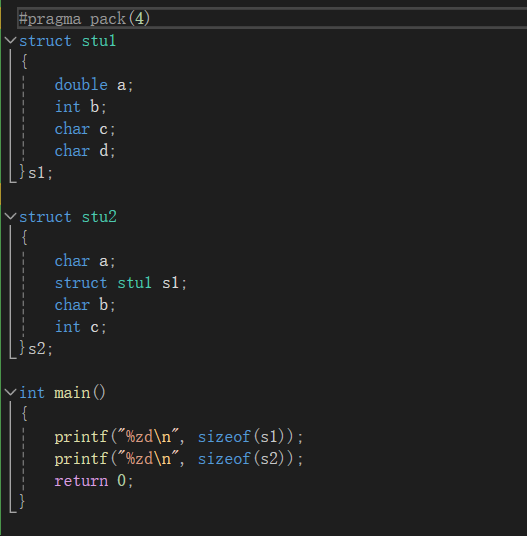
总体来说：结构体的内存对⻬是拿空间来换取时间的做法。也就有了我们上面的说法

那在设计结构体的时候，我们既要满⾜对⻬，⼜要节省空间，如何做到呢？答案很简单，那就是把占用空间小的成员尽量的集中放在一起，比如我们引子中的两个例子，我们可以看到他们中的元素是一样的：  


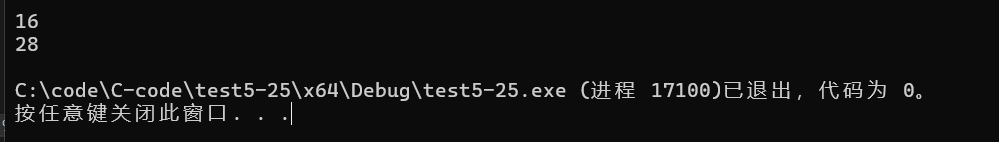
但s1却占了12个字节的内存，而s2却只占了8个字节的内存，其实从结构体的内存对齐的定义上我们就不难看出，当我们把占用内存空间小的成员集中在一起时，他们之间就算是有空隙也比混着放要少。

1.4修改默认对齐数

补充一个知识点，在vs2022里，它的默认对齐数为8，如果我们想要修改它的默认对齐数，则需要用到#pragma pack()来修改（括号里面放的是要修改的值），示例如下：



在未修改之前，输出的结果应该为16，32。但修改后得到的结果如下：



结果读者可自行计算验证，这里不再次给出计算过程。

（注意在我们日常修改默认对齐数时，最好设置为2的倍数，不要设置为1，3，5这样的奇数）

二.结构体实现位段

2.1什么是位段？

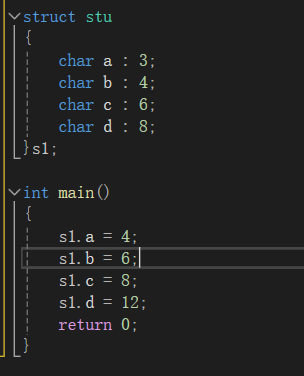
位段的声明和结构是类似的，有两个不同：

1. 位段的成员必须是 int、unsigned int 或signed int ，在C99中位段成员的类型也可以

选择其他类型。

2. 位段的成员名后边有⼀个冒号和⼀个数字。

⽐如：



2.2位段的内存分配

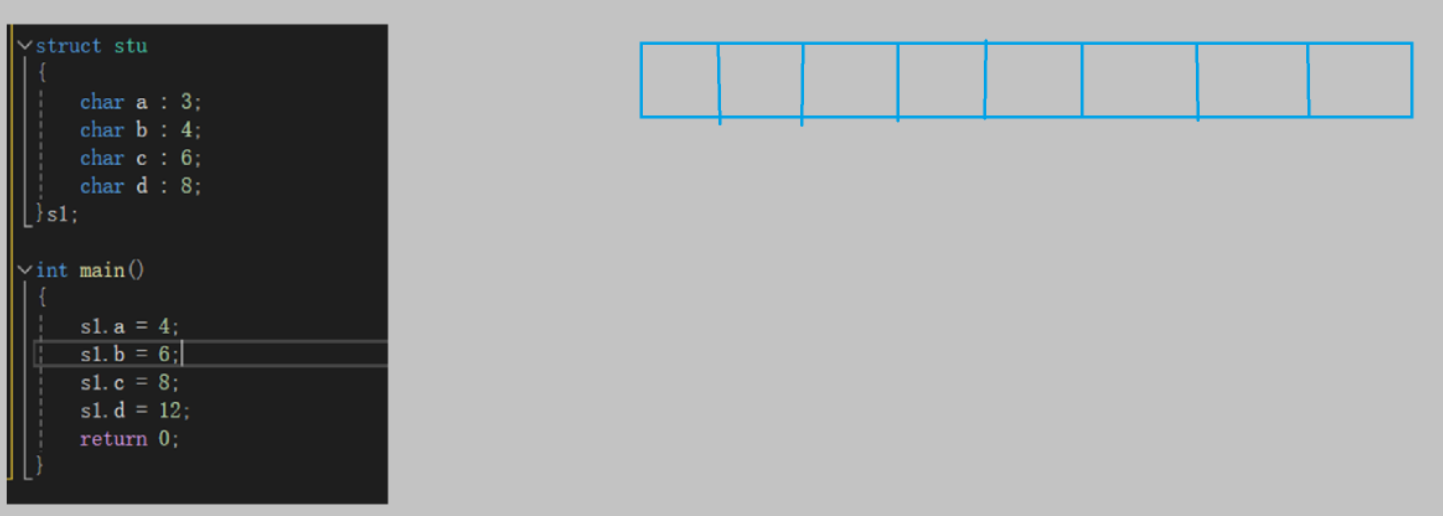
1. 位段的成员可以是 int unsigned int signed int 或者是 char 等类型

2. 位段的空间上是按照需要以4个字节（ int ）或者1个字节（ char ）的⽅式来开辟的。

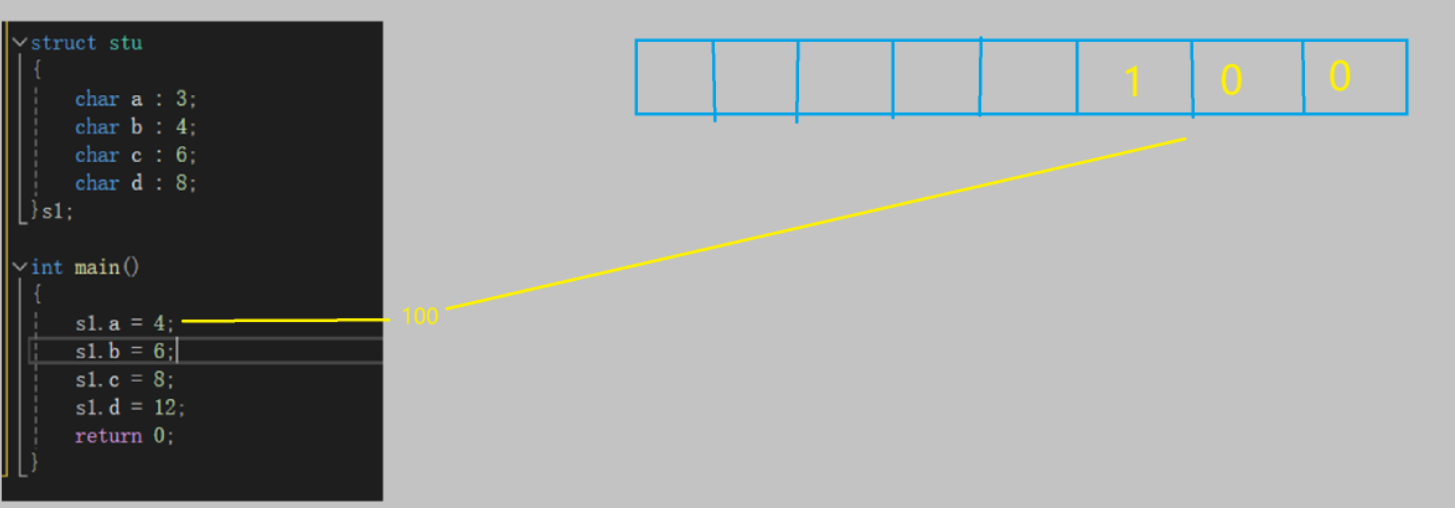
3. 位段涉及很多不确定因素，位段是不跨平台的，注重可移植的程序应该避免使⽤位段。

让我们以2.1为例来看该位段的内存分配：

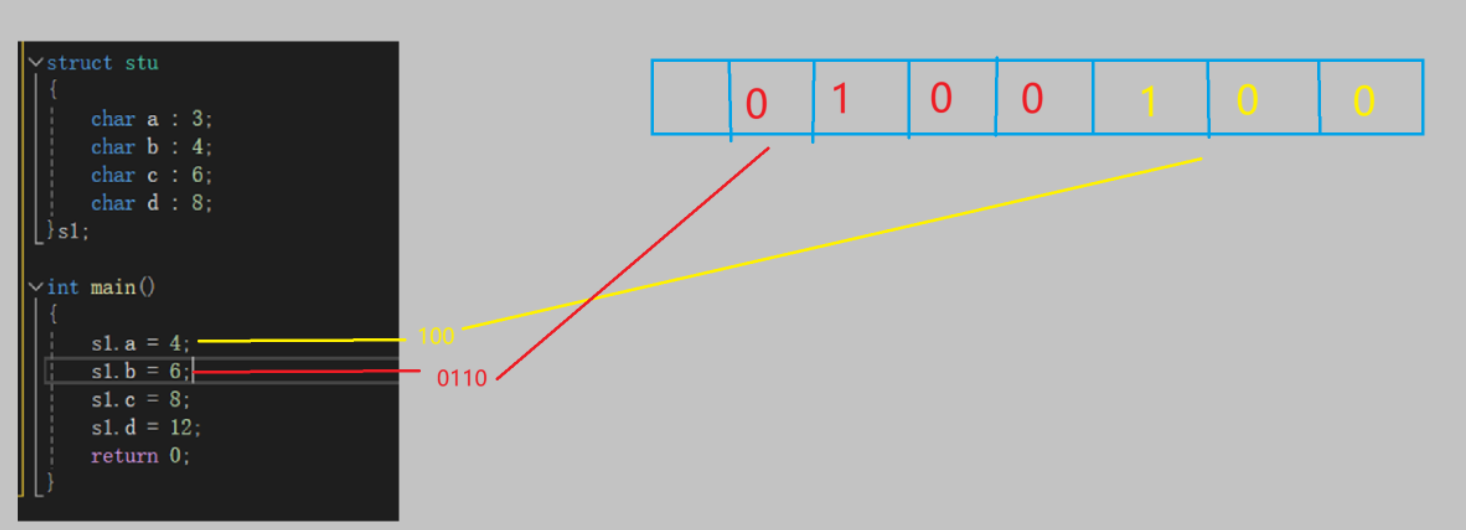
首先第一个元素为char我们这里需要开辟一个字节的空间：



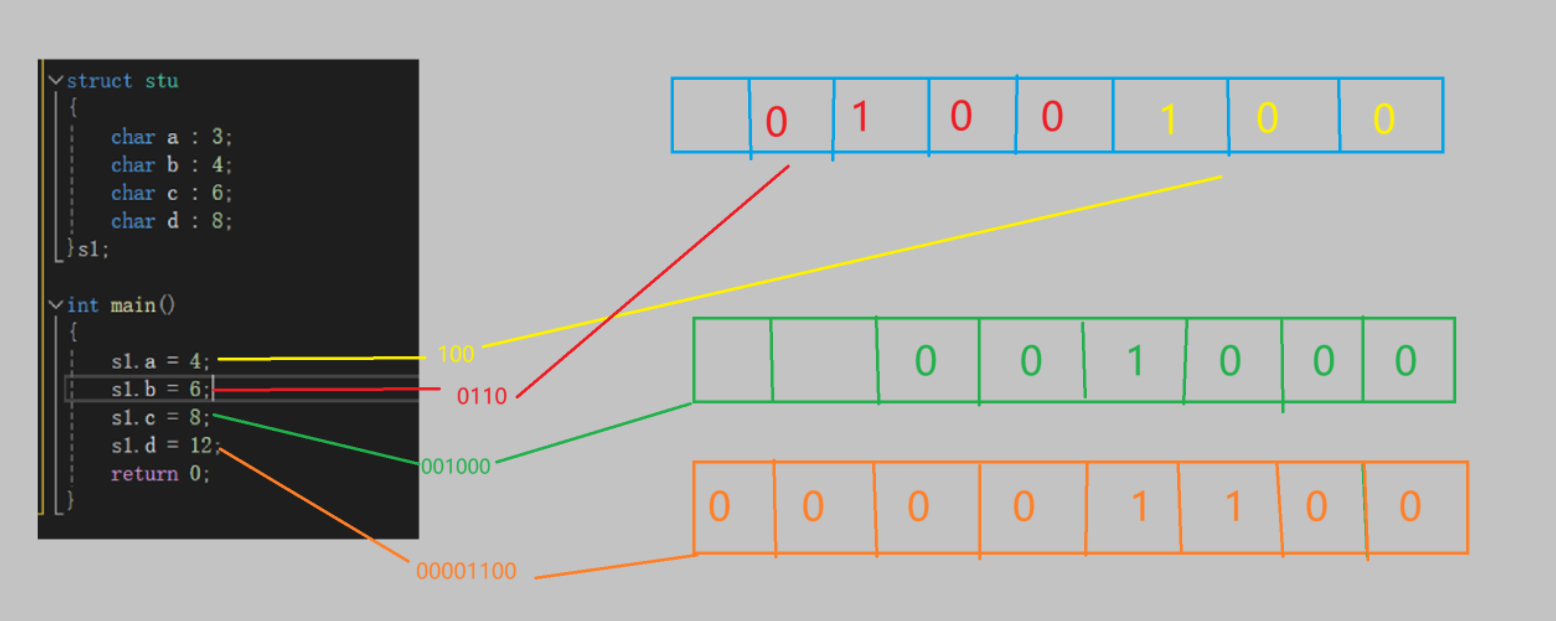
一共是8个比特位，接下来我们放元素a，因为它赋值的二进制形式为00000100，分配了三个字节空间，所以截取100放入：



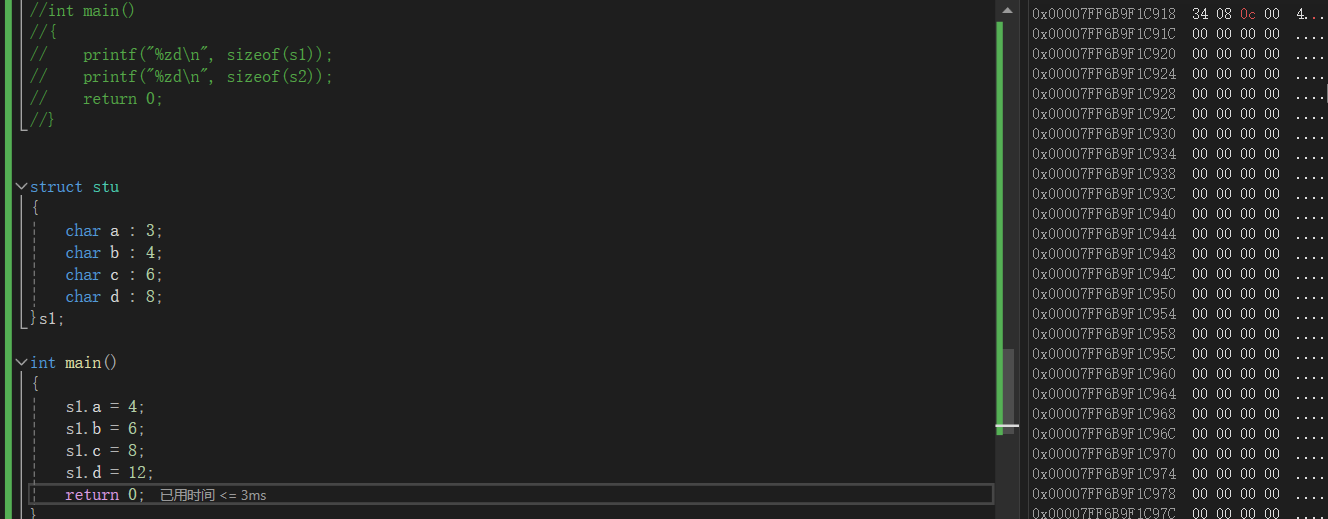
元素b的二进制形式为00000110，分配4个字节，截取部分为0110：



这里我们发现已经塞不下c和d了，又由于c为6个字节，d为8个字节，所以我们要开辟两个新的char空间：(红色部分输错了，应该为0110)



所以他们在内存中的存放应为：34 08 0c（红色部分数字应该为0110，这里作者粗心输错了还请见谅），让我们看看是否如此：



可以看到我们的答案是正确的。

2.3位段的跨平台问题

1. int 位段被当成有符号数还是⽆符号数是不确定的。

2. 位段中最⼤位的数⽬不能确定。（16位机器最⼤16，32位机器最⼤32，写成27，在16位机器会

出问题。

3. 位段中的成员在内存中从左向右分配，还是从右向左分配，标准尚未定义。

4. 当⼀个结构包含两个位段，第⼆个位段成员⽐较⼤，⽆法容纳于第⼀个位段剩余的位时，是舍弃

剩余的位还是利⽤，这是不确定的。

总结：

跟结构相⽐，位段可以达到同样的效果，并且可以很好的节省空间，但是有跨平台的问题存在。

2.4位段使⽤的注意事项

位段的⼏个成员共有同⼀个字节，这样有些成员的起始位置并不是某个字节的起始位置，那么这些位置处是没有地址的。内存中每个字节分配⼀个地址，⼀个字节内部的bit位是没有地址的。

所以不能对位段的成员使⽤&操作符，这样就不能使⽤scanf直接给位段的成员输⼊值，只能是先输⼊放在⼀个变量中，然后赋值给位段的成员。

