# Struct 结构体

## 博远信息技术社 项目部 靳晰月

**概念：**

结构体是一些值的集合，这些值称为成员变量，且每个成员可以是不同类型的变量。

\*与数组进行区分：数组是一些相同类型元素的集合。

**功能：**

结构体可以用来描述复杂对象。

Ep.在描述一本书的名字、作者、售价等的时候，我们需要使用不同类型的变量，而单单用int、char等类型是不够的。这时我们就可以运用结构体设计出属于自己的变量类型用于描述对象。

**使用：**

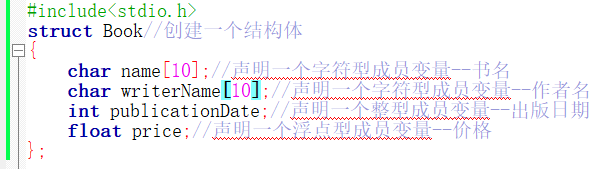
1. 结构体设计

我们以创建一本书的封面内容为例。

我们需要体现：书名、作者名、出版日期及定价。

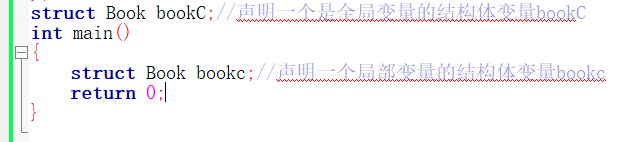
分析各成员变量的类型：书名、作者名为字符串；出版日期为整型；定价为浮点型。

于是我们可以得到这样的代码：

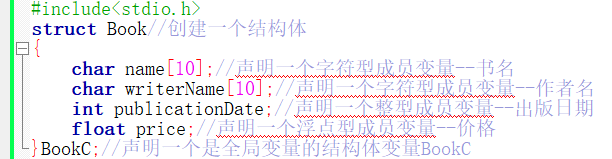


**\*注意：**

1. *在使用”struct”时，别忘了声明成员变量结束后的大括号外标”;”。*
2. *例中的”Book”只是结构体的类型名，我们尚未声明结构体变量。*
3. 结构体声明



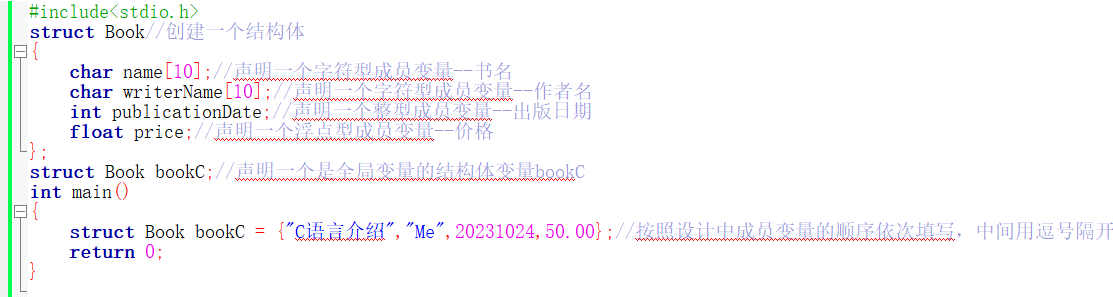
除去这两种常规的变量声明外，我们还可以这样：



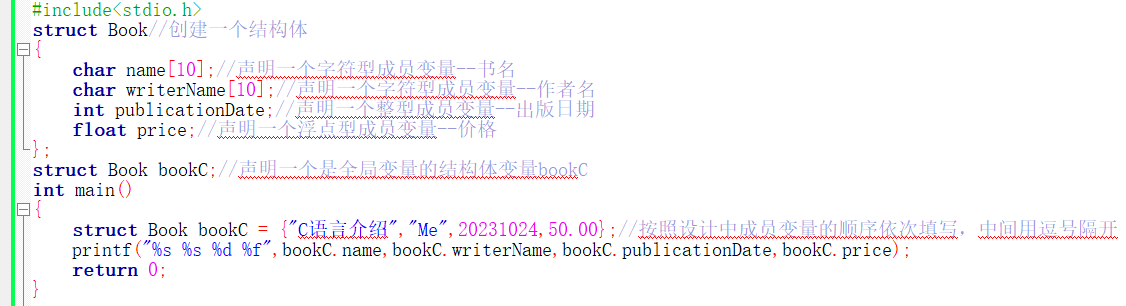
在”{}”外，”;”之前标注变量名，就可以得到一个是全局变量的结构体变量。

1. 结构体变量初始化

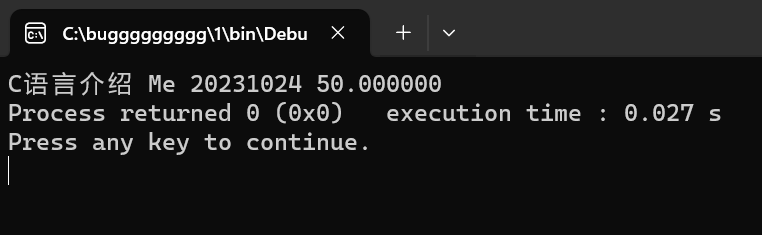
我们声明了一个结构体变量，接下来要填充其内部的成员变量，即结构体变量初始化。



如果我们想要将封面内容打印出来，就需要用到”.”操作符。”.”操作符在这里是用来访问结构体的操作符，例如：



我们来看一下输出结果：

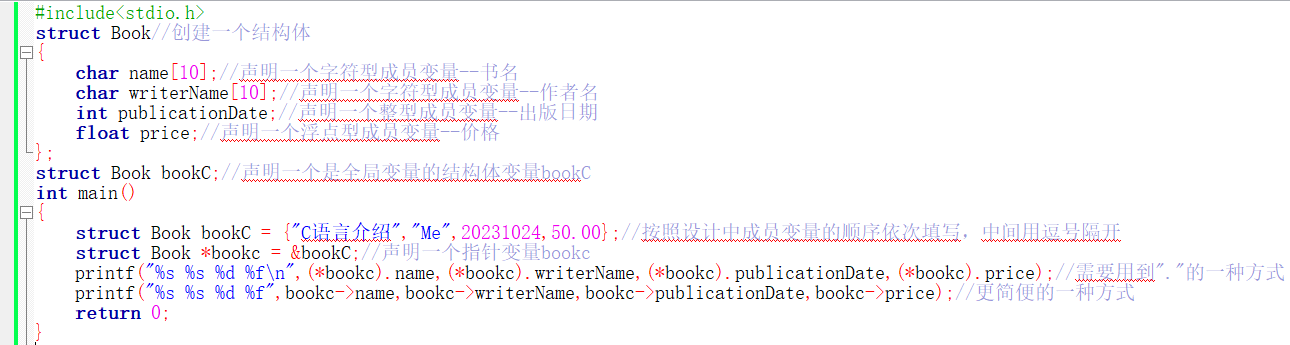


很简单吧XD？

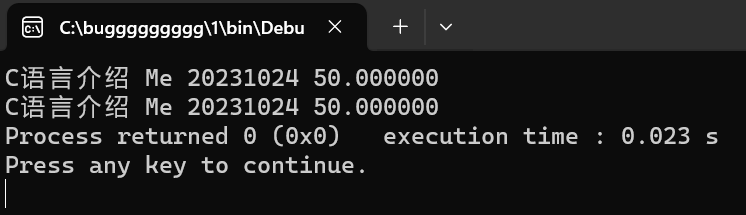
**\*注意：**

1. *括号内的元素要与结构体设计中一一对应。*
2. *在C语言中不存在结构体类型的强制转换。*
3. *可以设置更简短且易懂的变量名。*

接下来我们可以尝试指针打印。



我们来看一下输出结果：



可见，这两种输出方式得到的结果是一样的。我们可以使用”->”操作符替代”(\*变量).”的冗杂形式。

那么，既然直接打印和指针打印可以得到同样的效果，我们更倾向于使用哪一种打印方式呢？

当然是指针打印咯！

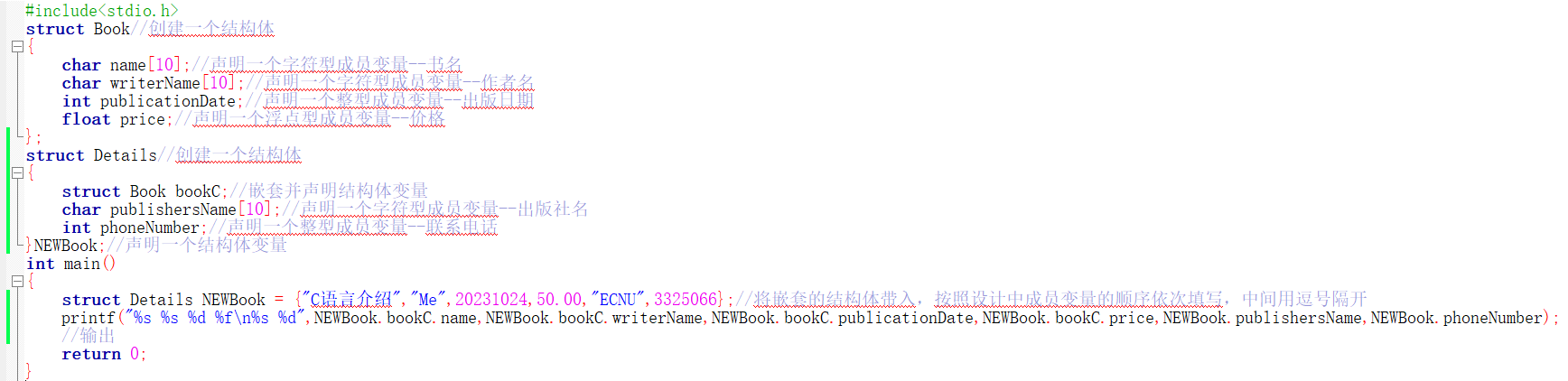
因为函数传参的时候，参数是需要压栈的。如果传递一个结构体变量的时候，结构体过大，参数压栈的系统开销比较大，会导致性能下降。所以更倾向于使用指针打印。

多！么！简！单！XP！

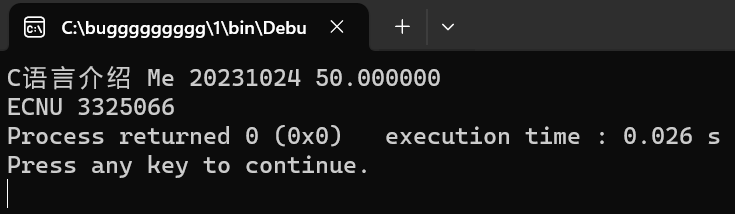
1. 结构体嵌套

为了表达直接易懂，在结构体中我们同样可以使用嵌套结构。

如我们仍然需要形容一本书的封面，但我们希望更细节的部分得到描述，那么我们可以写出以下的代码：



我们来看一下输出结果：

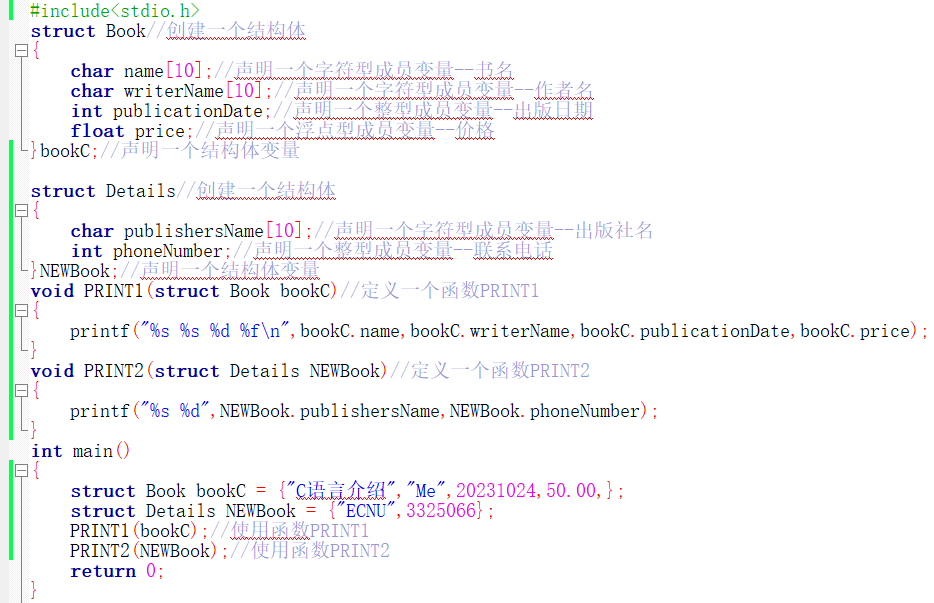


**\*注意：**

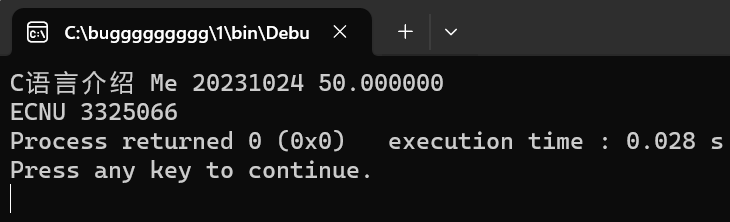
1. *创建几个结构体就要声明几个结构体变量。*
2. *在对指定成员变量进行访问时，要从最外层结构体一步步指出该成员变量所在的结构体，最后指出该成员变量。*
3. 结构体与函数

结构体当然可以和函数结合。当我们需要重复进行某个步骤时，我们可以定义一个函数，使表达简化。

如我们需要打印一本书的封面内容，而不想在main函数中出现的表述太过冗长，我们就可以定义以下的函数：



我们来看一下输出结果：

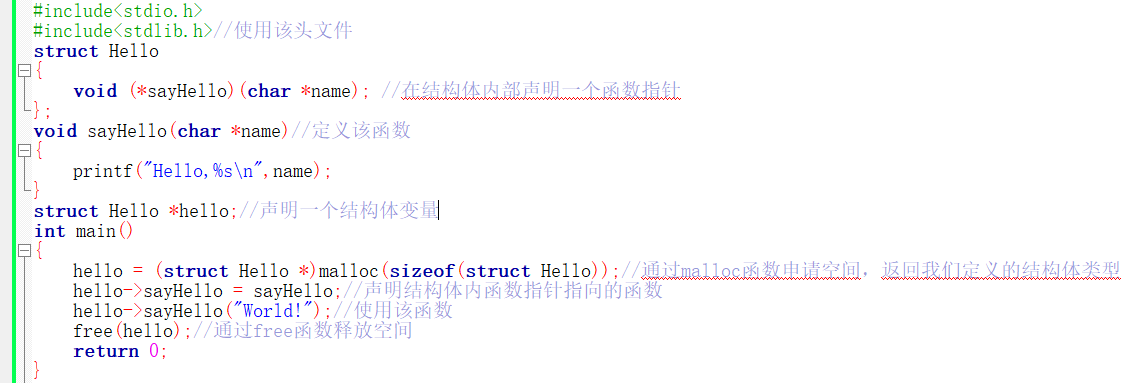


还可以拓展其他函数，结果还是一样的喔！

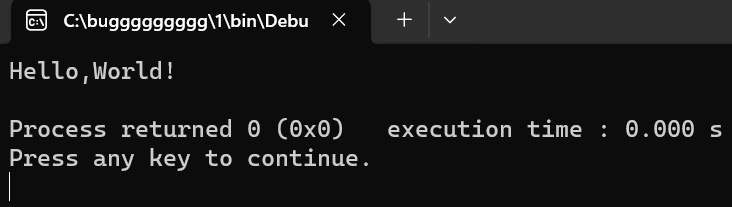
以上是简单的在结构体外部结合函数。那么如果我们想要在结构体内部定义函数，可不可以实现呢？当然也可以，只不过会比较麻烦。

比如我们想表达一个最简单的”Hello,World!”，要怎么在结构体内部使用函数实现呢？

我们可以这样做：



我们来看一下输出结果：



毫不意外啦~我们得到了一个复杂操作版”Hello,World!”。

\*注意：

1.使用结构体内部函数时，

1. 结构体与数组

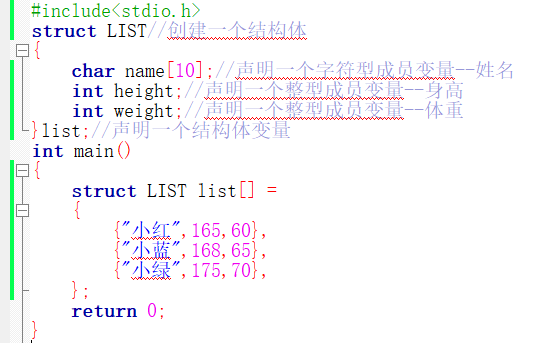
数组要求内部的元素为同类型，那么结构体数组呢？自然是内部的元素全为结构体类型。

我们以列出一个身高体重登记表为例。

我们需要体现：姓名、身高和体重。

分析各成员变量的类型：姓名为字符串；身高、体重为整型。

于是我们可以得到这样的代码：



这样，我们就可以将多个数据依次存储在结构体数组中了。

**\*注意：**

*1.不同的元素间用”,”隔开。*

七、结构体大小

结构体的使用方法我们大致已经清楚了，那么结构体到底占用多大的空间呢？

这个问题应该算是结构体中比较复杂的一个板块了，我们一步一步解释。

**内存对齐：***数据存储地址按一定的规则存储，使数据地址能够被访问*(不是所有的硬件平台都能访问任意地址上的任意数据；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)*，或节省被访问的时间*(为了访问未对齐的内存，处理器需要做两次内存访问；而对其的内存访问只需要一次)*。*

**指定对齐方式：***平台自身设定的内存对齐方式，不由成员变量自身所占字节大小决定，是对于该平台来说的一个默认方式。*

该方式不同平台可能不一样。

**首地址**：*结构体中第一个成员变量存储的位置。*

该位置由该结构体中，自身所占字节数最大的成员变量所占字节的整数倍(如果存在嵌套结构体，包含嵌套结构体在内的所有成员变量中自身所占字节数最大的成员变量)，或指定对齐方式的整数倍决定。在这两者中，最后由它们中较小的那个数字作为最终的首地址位置。

上述内容可以这样表示--

MIN{自身所占字节数最大的成员变量所占字节的整数倍，指定对齐方式的整数倍}

**每个成员变量的对齐数：***结构体中成员变量自身所占字节的个数。*

**最大对齐数：***结构体内所有成员变量自身所占字节个数的最大值。*

**结构体中每个成员相对于首地址的偏移量：***按照内存对齐规则的每个成员变量相对于首地址之前空出的地址个数。*

该数字由每个成员变量自身所占字节数的整数倍，或指定对齐方式的整数倍决定。在这两者中，最后由它们中较小的那个数字作为最终的偏移量。

上述内容可以这样表示--

MIN{每个成员变量自身所占字节数所占字节的整数倍，指定对齐方式的整数倍}

**结构体的总大小：***结构体所占内存的总大小。*

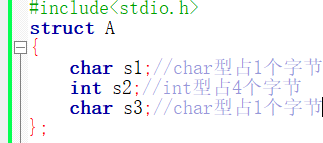
该数字由该结构体中，自身所占字节数最大的成员变量所占字节的整数倍(如果存在嵌套结构体，包含嵌套结构体在内的所有成员变量中自身所占字节数最大的成员变量)，或指定对齐方式的整数倍决定。在这两者中，最后由它们中较小的那个数字作为最终的总大小。

上述内容可以这样表示--

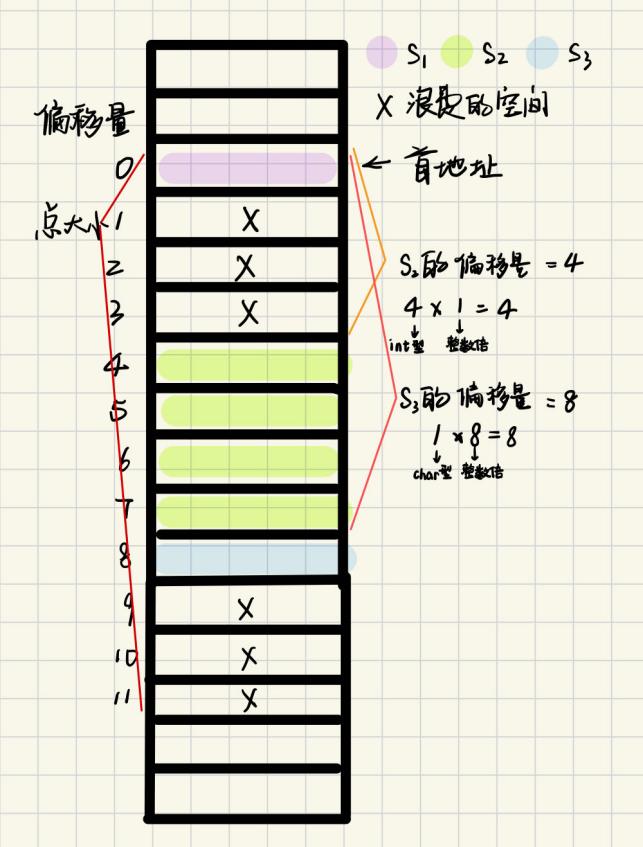
MIN{自身所占字节数最大的成员变量所占字节的整数倍，指定对齐方式的整数倍}

基本的概念阐释清楚后，我们通过例子逐步了解对齐规则。

我们创建这样一个结构体：

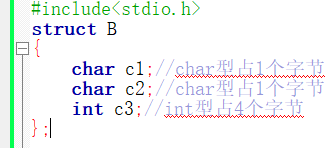


我们设定这个结构体是这样存储的：

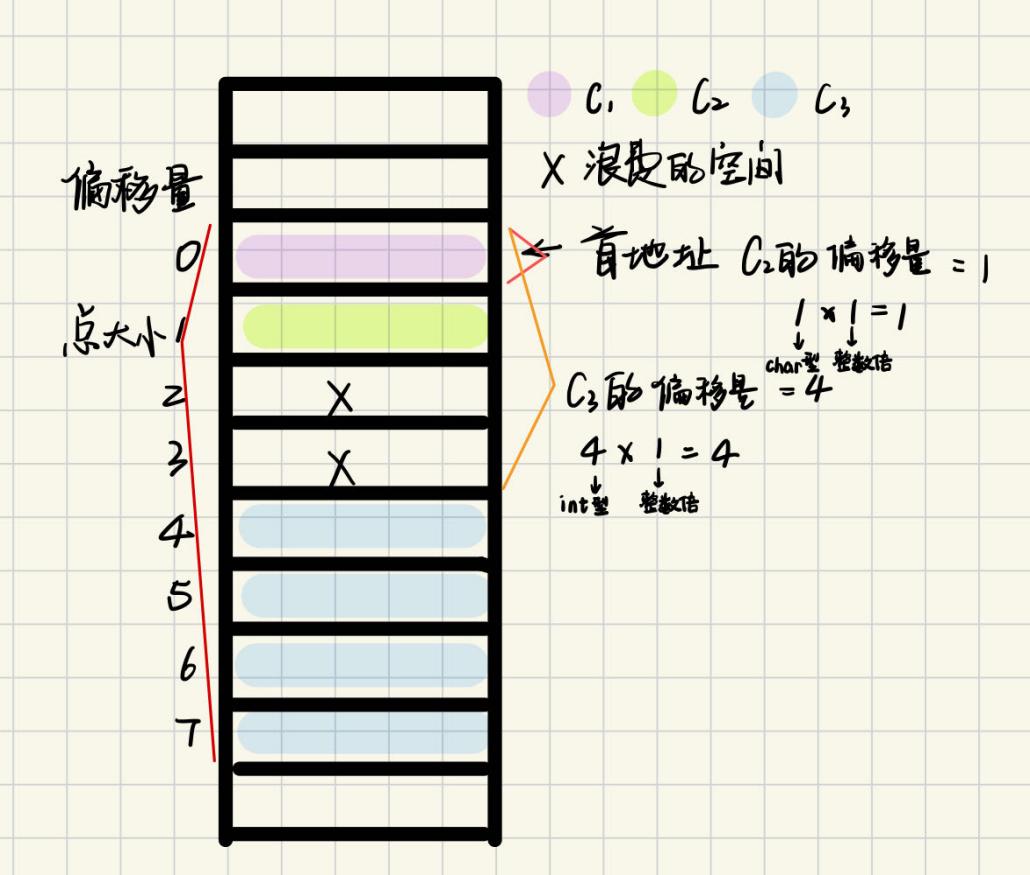


我们不难看出，这个结构体的总大小是”12”。有的同学可能奇怪，为什么不是”9”呢？别忘了前面阐释过的总大小的概念。本例中，自身所占字节最大的成员变量是”s2”，int类型，占4个字节。9不是4的整数倍，4\*2=8<9<4\*3=12，所以这个结构体的总大小为”12”。

我们再比如创建这样一个结构体：



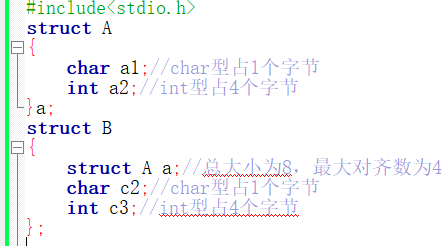
我们设定这个结构体是这样存储的：

****

我们不难看出，这个结构体的总大小是”8”。本例中，自身所占字节最大的成员变量是”c3”，int类型，占4个字节。8是4的整数倍。总大小仍为”8”。

那么，如果我们在结构体中嵌套一个结构体，它的内存又要怎么对齐呢？

我们比如创建这样一个结构体：



我们设定这个结构体是这样存储的：



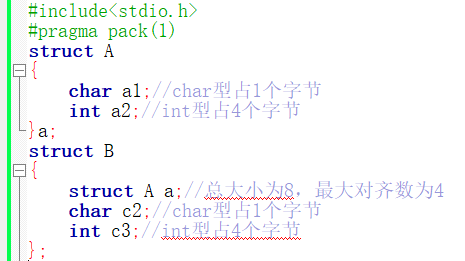
嵌套的结构体也是一样的对齐规则。

我们不难发现，按照这样的规则，会有部分空间被浪费，如果们能改变对齐数，就可以改变每个成员变量的偏移量大小，就可以仅可能的占用空间了。

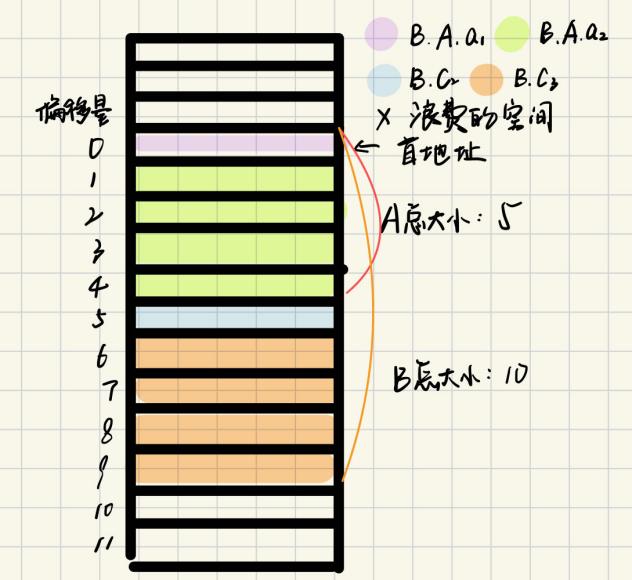
于是我们可以用这样的代码改变默认的对齐数：

(其中n的取值可以为1,2,4,8,16。)

如果我们这样做，令n的取值为1：



那么，这个结构体的存储形式就会变成这样：

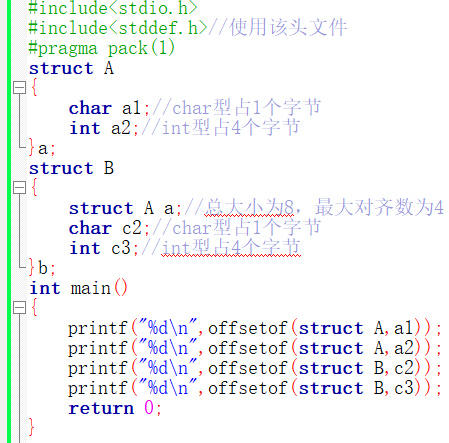


这样修改对齐数为1后，结构体不存在内存对齐了，就没有空间会被浪费了。

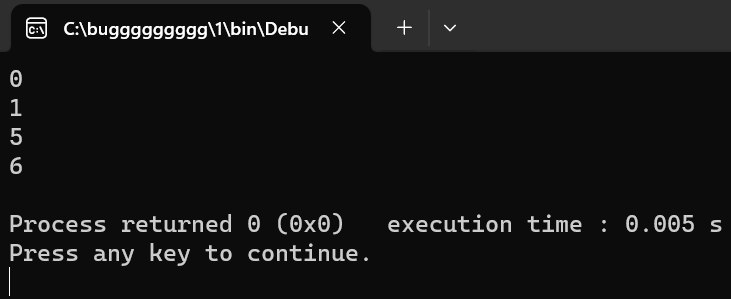
\*补充：

如果需要计算结构体成员变量相对于首地址的偏移量，我们可以使用这样一个代码：

“offsetof(数据类型,成员变量)”



我们来看一下输出结果：



于是我们就打印出了每个成员变量相对于首地址的偏移量。

**以上就是关于结构体的入门详解知识了。**

如果在使用时遇到了问题，可以多去问问老师同学喔！