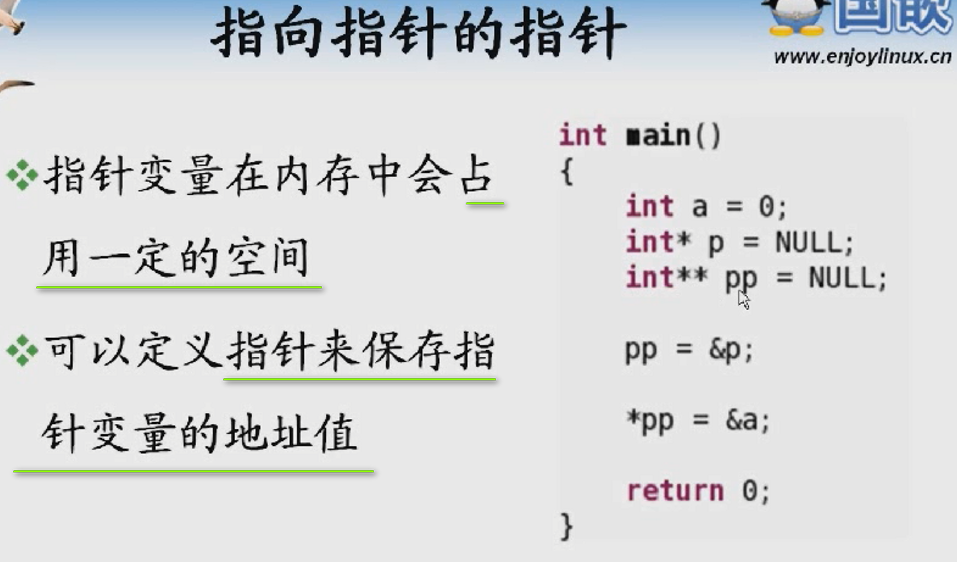


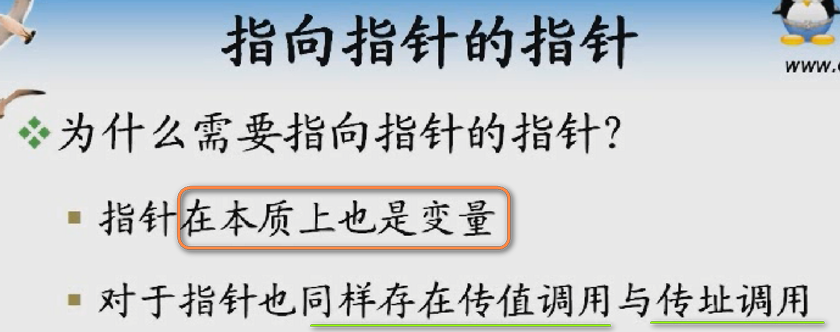
多维数组 和 多维指针

指向指针的指针 ----- 指针也是变量 内存中也占用一定的空间 那么 可以说明我们的指针也有自己的地址 所以 可以再定义一个指针来指向指针



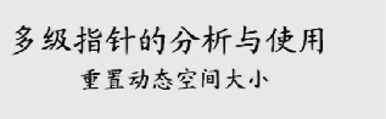
 还是指针

**指向指针的指针 的存在意义**



想要修改函数外部的指针的值





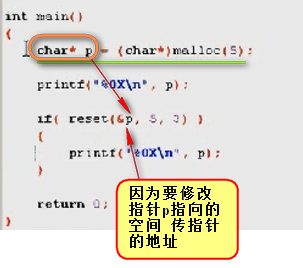
看下main

理论上需要5个字节 但是 一直在使用的就是前3个字节 后面两个浪费 把后面两个释放掉 只留前面3个字节

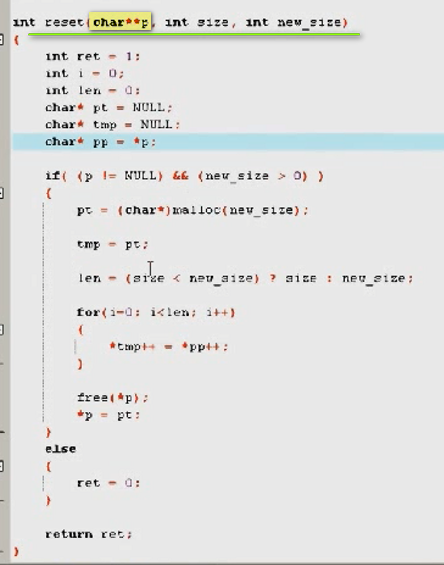
---- 或者扩展p指针指向的空间呢？

**无论是收缩p指针指向的空间 还是扩展p指针指向的空间 ------ 都是要在函数内部修改外部的变量 只是这里面在函数内部修改的是外部的指针变量 ----- 所以用到了指针的指针**

这样 p指向

 调用的时候 取的是指针调用的地址 --- 参数就是指针的指针【因为很多时候 你不能保证所有函数的返回不是一个地址】

看reset函数的具体实现 ----- 根据上面说的 传入的是指针的指针 那么 第一个参数就是char\*\* p ---- 因为是数组 但是 编译器把数组的维度拿掉 ---- 这样 需要传入老的数组的大小 所以 第二个参数是int size --- 最后把数组的新大小传递过来 int new\_size



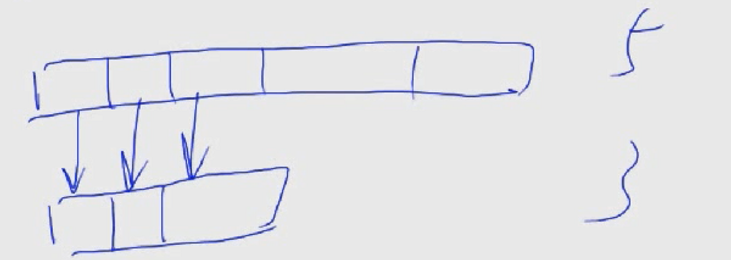
 这个是安全性判断

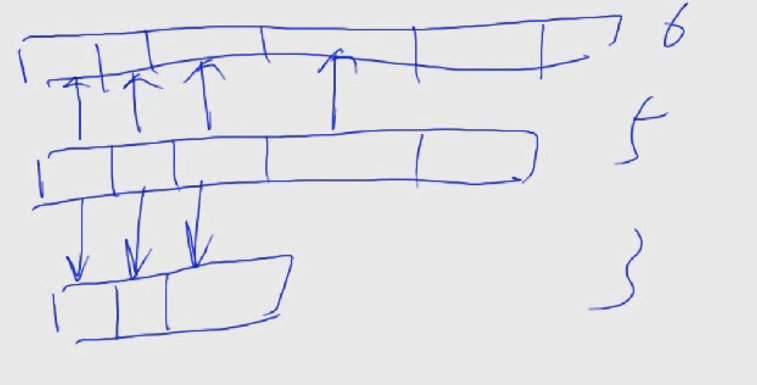
 获取有效长度

定义tmp的原因是：采用的指针的方式遍历 效率高 ----- 最后 这个tmp遍历完成之后 不是新空间的起始位置 ---如果没有tmp 那就要调整遍历指针的位置 麻烦 ----- 所以 使用tmp

获取到新的空间 并且把数据填充好之后 就把原有的空间释放掉  因为也不用了



 老的是5 新的是3

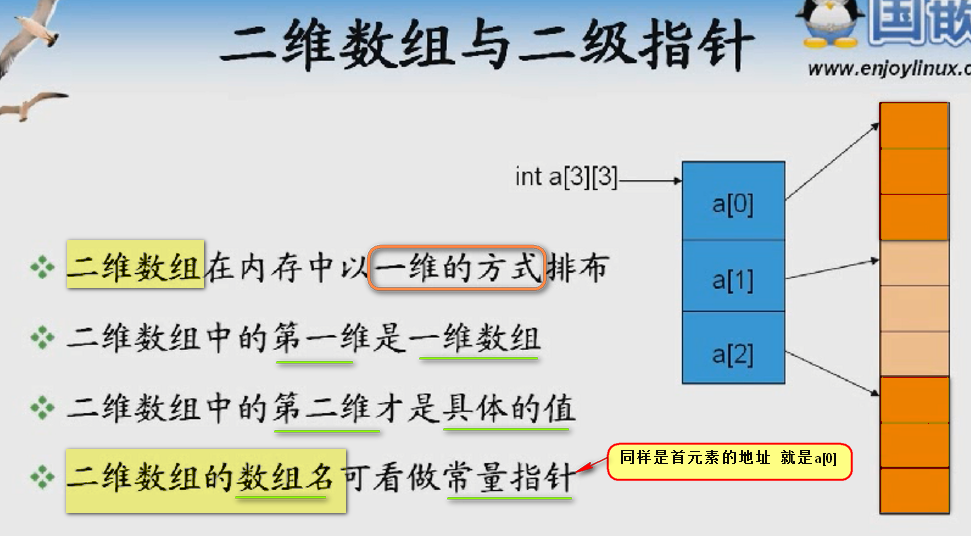
 老的是6 新的是3

有可能是扩充 有可能是剪裁

这样的reset函数是传址调用 传递的是指针的地址 ---- 所以定义了指针的指针

============================

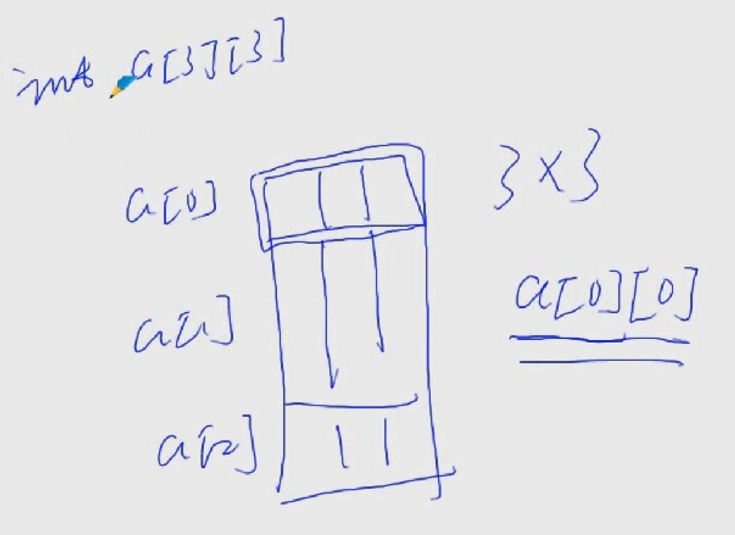
二维数组的数组名代表什么？ 要搞清楚二维数组在内存中的排布！！！C语言中 二维数组以一维的方式排布 ---- 因为我们的内存是一维的 是线性排布 ---- 那么 二维数组就要以一维方式排布



--- 二维数组的第一维 是一维数组 第二维是指定一维数组中的某一个元素

**二维数组名代表的同样是数组首元素的地址**

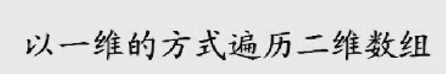
**只是二维数组的首元素是一个数组a[0]**



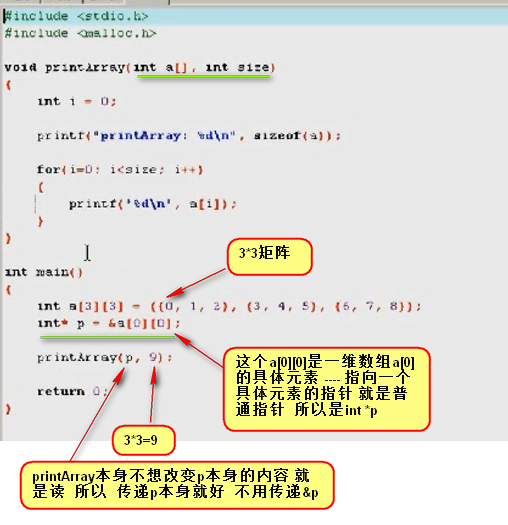
二维数组首元素 那么二维数组名就是数组指针 ----- 因为二维数组名就是首元素的地址 a[0]本身就是一个数组 ---- 那么指向这个数组的指针 --- **类型就是数组指针**

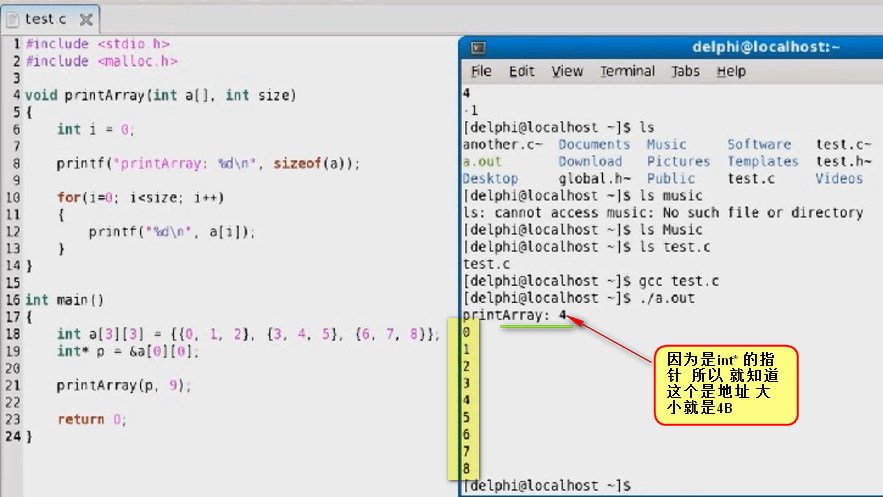
【一维数组的首元素的地址就是a ---- 那么首元素是a[0] ---- 那么指向首元素的指针就是 指向单纯数组值的指针 ----- 普通指针 比如 int\*】

**二维数组的首元素是一个一维数组 ---- 指向这个一维数组的指针就是数组指针**



有人不信 那么 就以一维的方式遍历二维数组

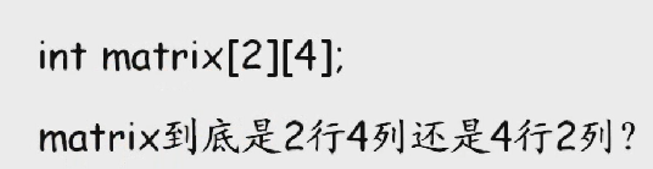


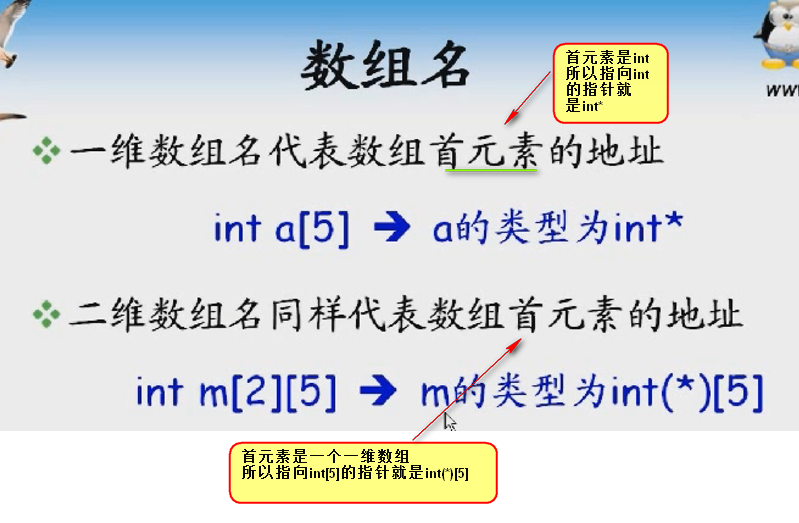


***我们的二维数组中的每一个元素 都是以一维数组的方式打印出来 --- C语言中二维数组 物理存储 是以一维的方式进行存储的 ---- 所以 只要你愿意 可以以一维的方式定义二维数组***

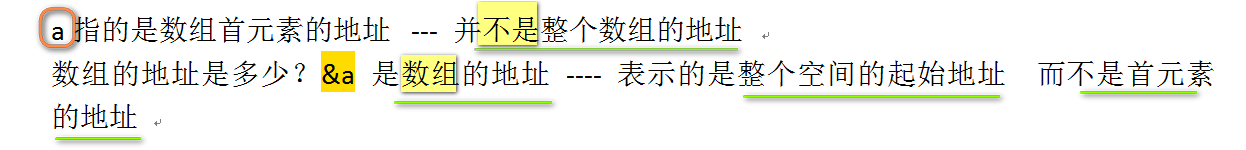
【注意 给的是元素的指针 就是int\*p 但是 长度却是整个二维数组的长度9 而不是一维数组的长度3 但是 结果都遍历出来了 说明是一维排列】

刚才说了 二维数组内存是一维的方式线性排列

【思考题】

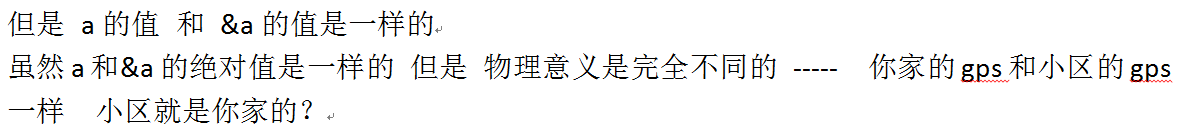


【a是一维数组的情况下



所以 记住一个 就是 数组名永远是数组首元素的地址 ---- 无论几维

但是 数组首元素的地址物理意义上却不是整个数组的地址 ---- 尽管a和&a值是一样的 ----- 这是因为a是常量指针 不是变量 所以 两个值一样

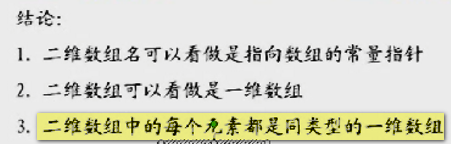


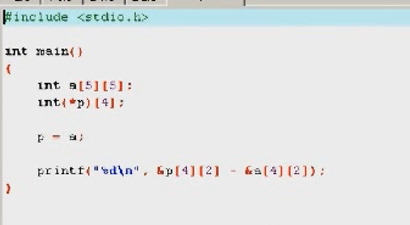
那么 这些同样适合二维数组

***但是注意的是 如果是指针变量p 就和&p的值是不一样的！ 上面一样的原因是因为是指针常量***】



【注意 上节课都讲了 定义指向数组的指针 是type(\*name)[size] 】





这个是一个面试题

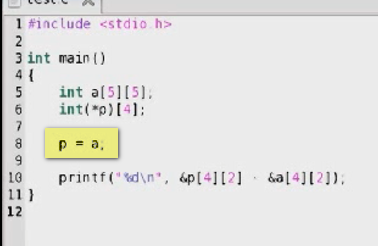
刚才说了a就是二维数组名 ---- 就是二维数组首元素的地址 ---- 二维数组的首元素是一个一维数组 也就是a[0] ---- 那么指向这个a[0]的指针类型就是指向数组的指针 ---- 所以 a的类型就是指向数组的指针 ---- 所以 赋值给p

【注意这个分析过程 记住二维数组的首元素就是一个一维数组】

【看到 这个指向数组的指针p定义的是指向一个int[4]类型的一维数组 ----- 这个一维数组有4个元素】

是不是0？

运行一下





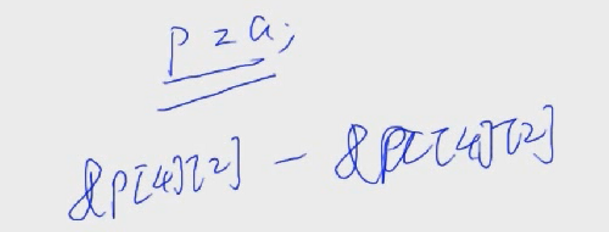
给出了一个警告 说第八行不匹配 ----- 但是编译通过



打印出来-4 ？

int a[5][5] --- a代表了int(\*)[5]

我们的p的类型是 int(\*p)[4] ---- 编译器给出警告



这个是指针运算 ----- 指针运算要求是同一个数组元素 ---- 所以相减的前提没有问题 ---- 但是 为什么是-4？

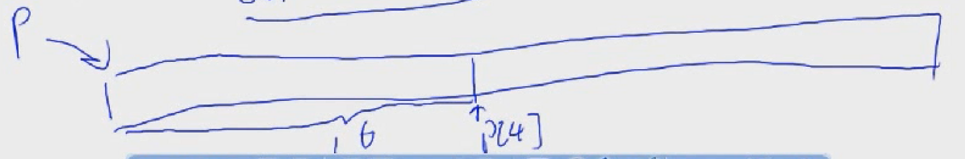
p[4] 就是p+4 ---- 为什么？ 因为p的类型是 int(\*p)[4] ---- 指向一个四个元素的一维数组 ---- 那么p[4]就是p+4 ---- **一次移动4个int 也就是一次移动一个数组长度 ---- 是自身定义决定的**

开始的时候

二维数组一维排列

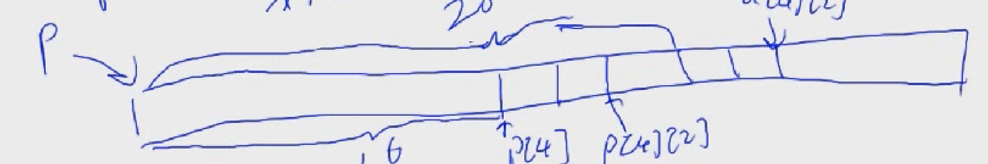


P最开始指向开始 p+4一次跳4个int ---- 这样是16个元素



这个a[4] ---- 就是a+4 ----- a定义是int a[5][5] ----- 那么 a的类型就是int(\*)[5] ---- 一次移动5个int ----- a[4]就是20个字节

这样p[4][2]就是在p[4]的基础上向后偏移两个int元素的位置



a[4][2]是第22个 p[4][2]是第16个元素 --------

前面讲过指针运算等于下标差

这样18-22就是-4

所以 编译器给出的警告非常有用

由于a代表的数组指针的类型和p代表的数组指针的类型不一致



指针类型不匹配

【**所以 无论是几维的数组 这种运算不要考虑维度 因为算的是地址 所以 就变成一维线性排列 ---- 数组形式变成指针 ---- 一次跳几个 完全是定义安排的** ----- 所以 这样看到 数组类型中的数组大小是非常重要的！！！！！】

通过这个题目 注意两点：

\*\*二维数组名代表二维数组首元素的地址 类型是数组指针类型

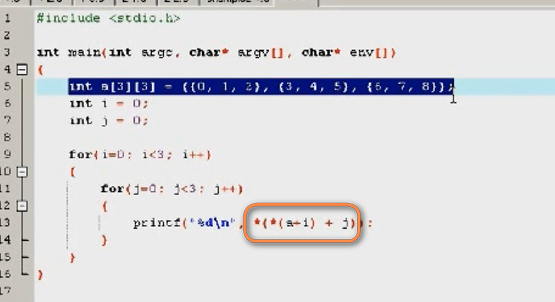
\*\*数组指针类型 ：大小+指向数组中数组元素的类型 两个缺一不可 否则就会认为是不同的指针（和前面讲数组类型是一样的）

\*\*如果强制把不同类型的指针相互传递 就会造成运算结果和你的想法不一样的错误

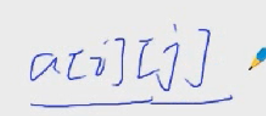


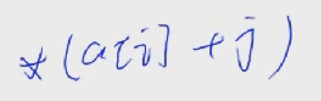
之前讲过下标形式和指针形式的对应关系

在二维数组中 可以以指针的方式遍历二维数组 而不是下标的形式



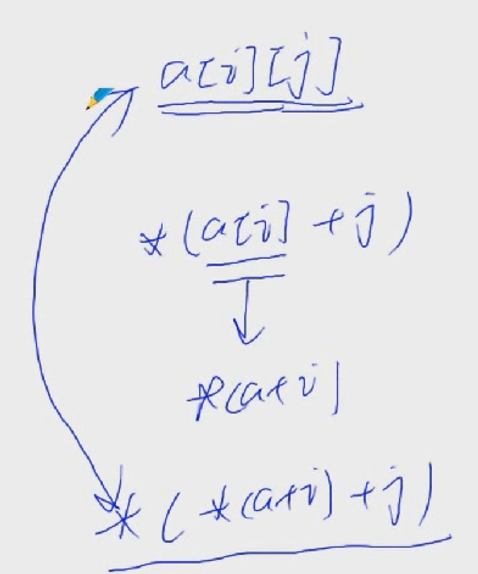
分析二维数组中指针形式和下标形式的对应关系

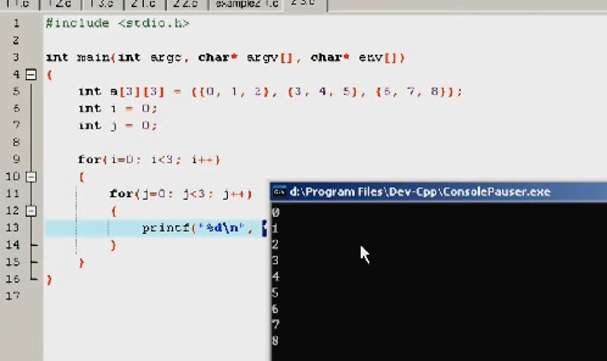
 两个下标改写

j --- a[i]是一个数组 那么a[i][j]表示a[i]这个一维数组的第j个元素-------就可以写成 

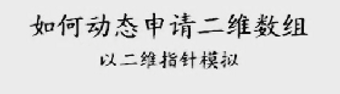
继续改写 a[i]又是一个一维的形式 --- 类比改写 \*(a+i)

**所以 a[i][j] -> \*(a[i]+j) -> \*(\*(a+i)+j)**





\*(a+i) ---换成下标形式 a[i] ----- \*(a[i]+j) ---继续换成下标 a[i][j] 等价看回来

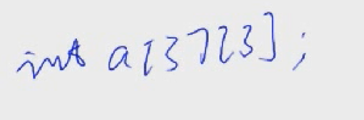


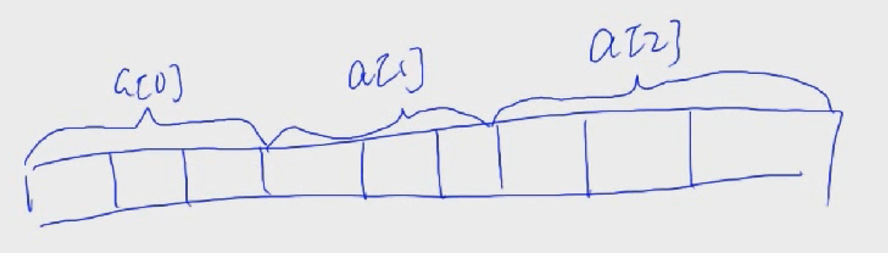
扩展思考 动态申请一个二维数组

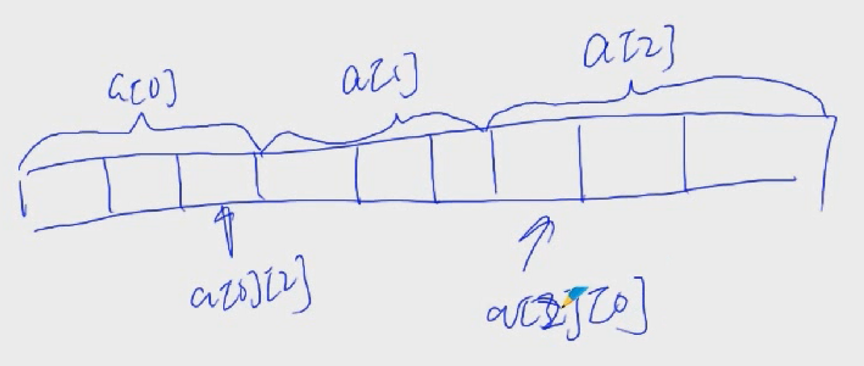
动态申请一个二维数组 ---- 考虑C语言如何实现二维数组

【typedef这个东西 可以把已知的写到一起 也可以分开 typedef INT int（分开） ---------- typedef int(\*p)[9]; ---- 这样 要定义的p和int[9]就合到了一起 typedef int f(int,int) ----- 也合到了一起 这f就是别名 ----- 后面的函数的指针和数组指针的这种都是这个技巧 ----- 这样做之后 就可以把函数指针 数组指针通过别名 统一写成 Type\* 写到后面】

C语言中

 编译器是把这个以一维方式存放的

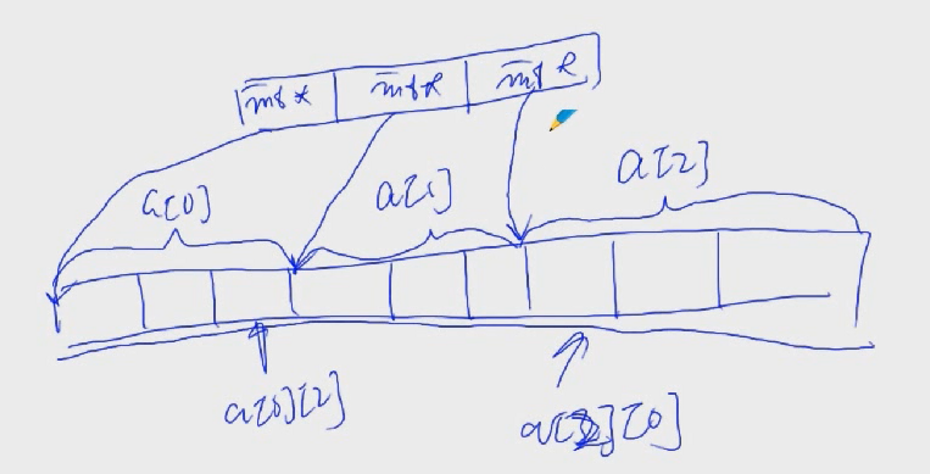


 如何动态申请二维数组 ----- 模拟编译器的行为 应该一下子把这块所有的内存都申请下来 ---- 这样3\*3申请出来9个int

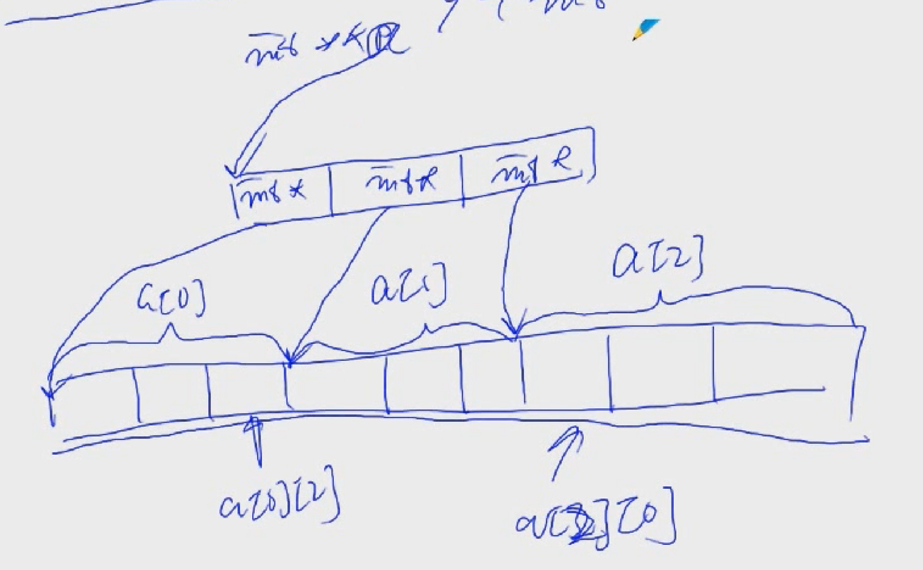
得到这些int之后 如何模拟成二维？

现在是1维的大数组 ----- 我们实际上应该是给这个一维数组一个边界 哪一块是a[0] 哪一块是a[1] 哪一块是a[3] ------ 就是需要存储这些边界位置 ----- 位置 弄到c语言中不就是指针么 ------ 那么 我们实际上就要存储的是多个指针

如下图：



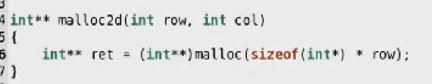
那么现在 我们有一个控制每一个a[i]边界的数组  我们是不是应该为这个数组取一个名字 ---- 数组名就是这个指针数组的首元素的地址 就是第一个int\*的地址 ----- 数组的名字本身是常量 我们没有办法手动取 ---- 那么我们可不可以搞一个指针变量指向这个指针数组呢？ 这样指针变量就可以被我们传递和使用了 ---- 那么这个指向指针数组的指针就是int\*(\*p)[3] ----- 按照图中就是这个 ---- 但是 使用上 我们直接找到这个首元素就可以了 那么 类型就是指向int\*的指针 ---- int\*\*p就够用了



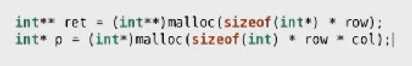
这样 我们的p实际上就是数组名

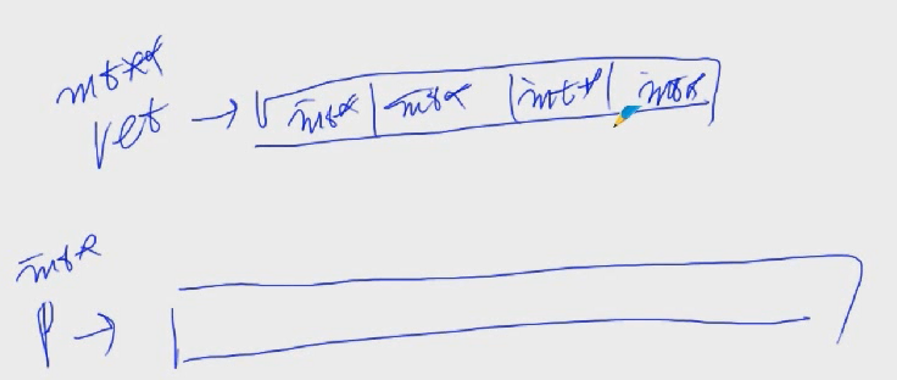
因为要返回这个二维数组的数组名 所以



 模拟的是 就是 我申请的是row个int\* 每个int\*的大小是sizeof(int\*) ---返回值类型强转成(int\*\*)

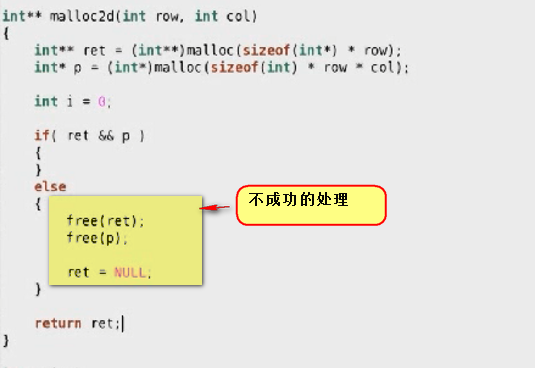
之后申请联系的内存空间

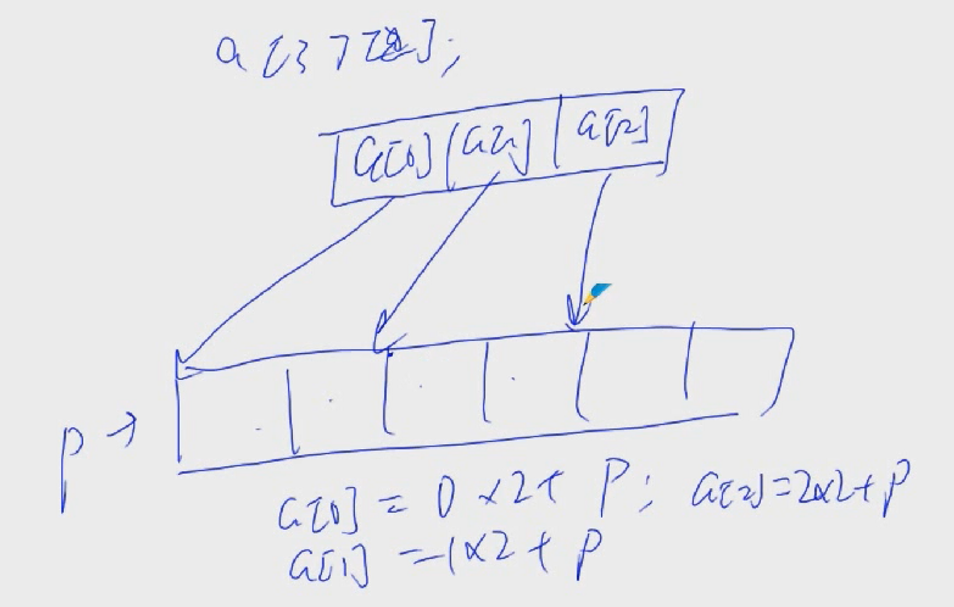


这样  两个内存空间申请出来了

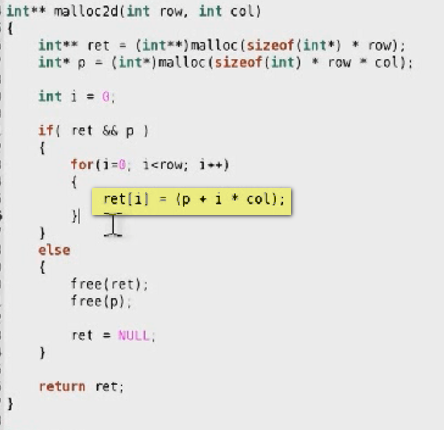
下面就是对应的地址存储关系还没有

为了安全起见 看看堆空间中的申请是否生效 进行安全判断

 不成功也要释放空间 避免泄露



【可以把每次循环的表达式写出来 然后找下规律】



现在释放内存

 注意 因为malloc2d返回的是一个int \*\*

所以 我们free的时候 应该释放两片内存 ------ 找到两片内存的首地址 ----- 一维数组元素的首元素是a[0][0](实际上就是对应的p[0] 但是p在malloc2d中没有返回来 这里面就是a[0][0]) ------ 那么a[0][0]的地址就a[0]

所以 是free(a[0]); 释放的是 这一片内存

再释放中间那一层的指针数组 ---- 首元素是a[0] ----首元素的地址就是a ----- 所以 再次释放的是free(a); 

这样 就给free完成了 【free一定是地址 而不是元素本身】

**这个就是如何在堆空间申请二维数组的程序**

***为什么没有使用一维的指针申请二维数组？【思考】***

