数组与指针分析

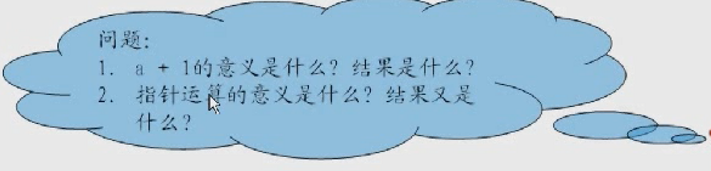


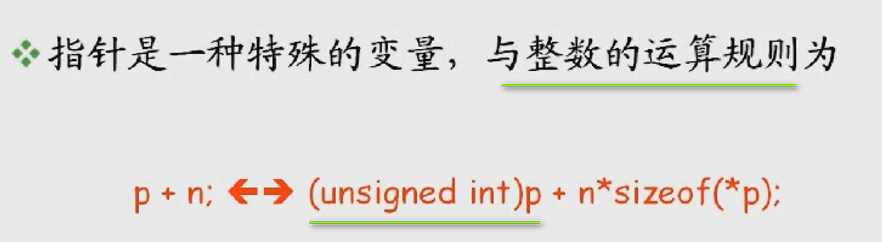




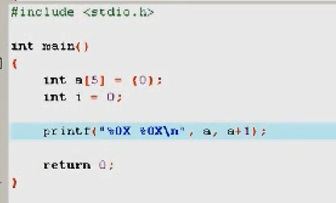


array\_type这里就是4个字节 4\*5 =20字节





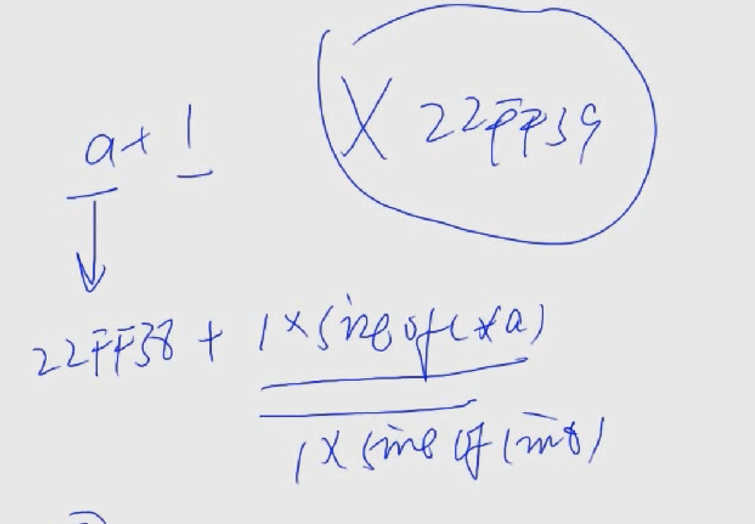
也就是指针p指向了同类型的数组元素 那么p+1就指向了当前元素的下一个元素 p-1就是当前元素的前一个元素



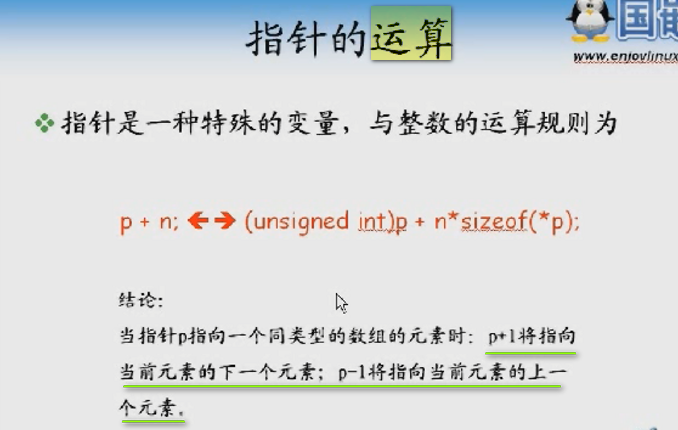


a是22ff38 a+1就是22ff3c ---- 相差4个字节

\*a指的是常量指针指向的元素 ---- a是一个int类型的数组 那么\*a表示第一个元素 所以 sizeof(\*a) 就是sizeof(a[0]) 实际上就是sizeof(int) 大小就是4B

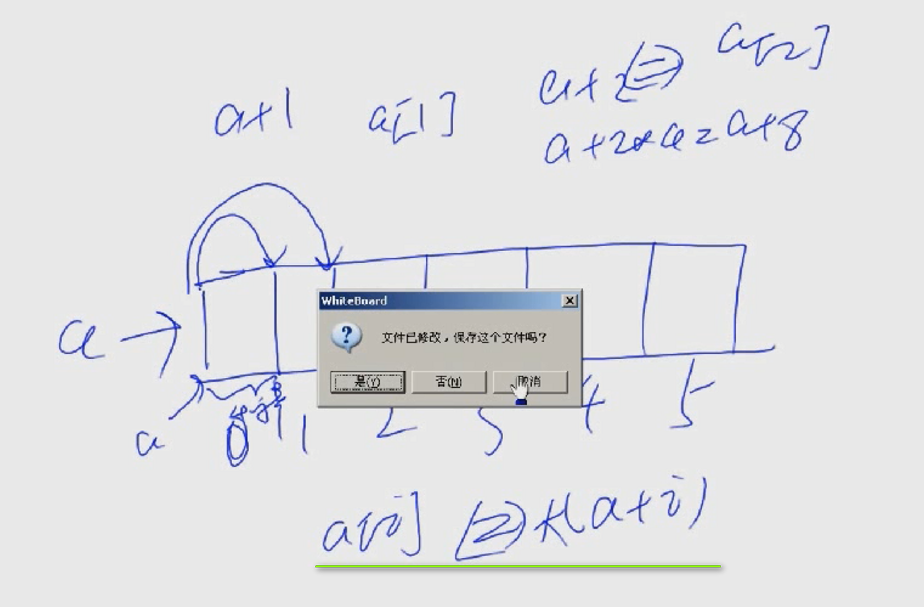


这就验证了

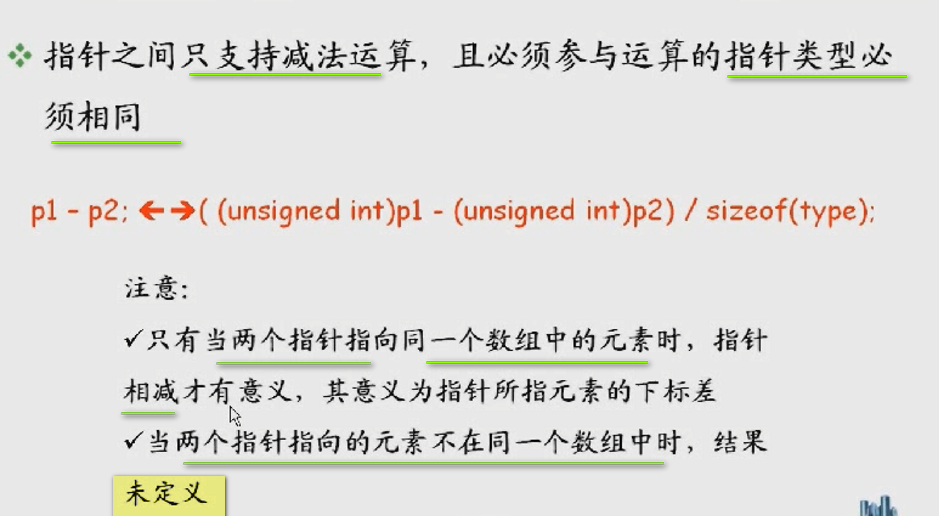


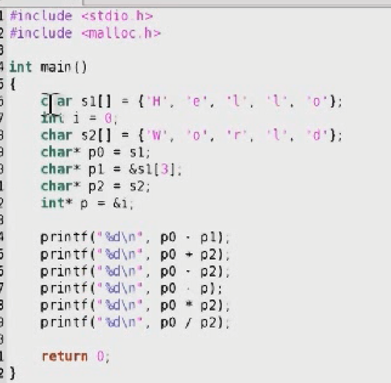
【**指针是可以运算的！！**】

所以 遍历一个数组就可以使用指针进行遍历



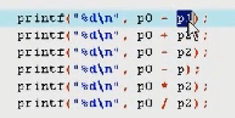
指针除了可以跟整数运算 是否可以做其他的运算？



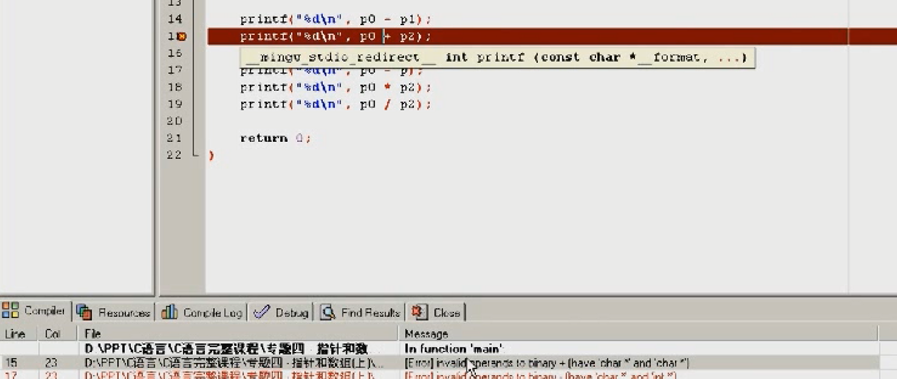


画一下内存分布

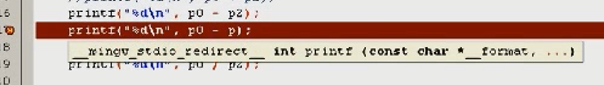




编译



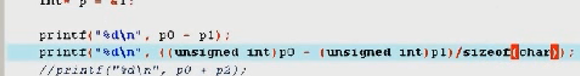
不合法



类型不同 一个是字符指针 一个是int指针 不同类型不能相减

 两个指针的加减乘除 只有减法是合法的

 第一个打印出-3 正确

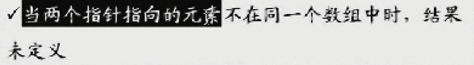


带入公式 得出的结果都是-3



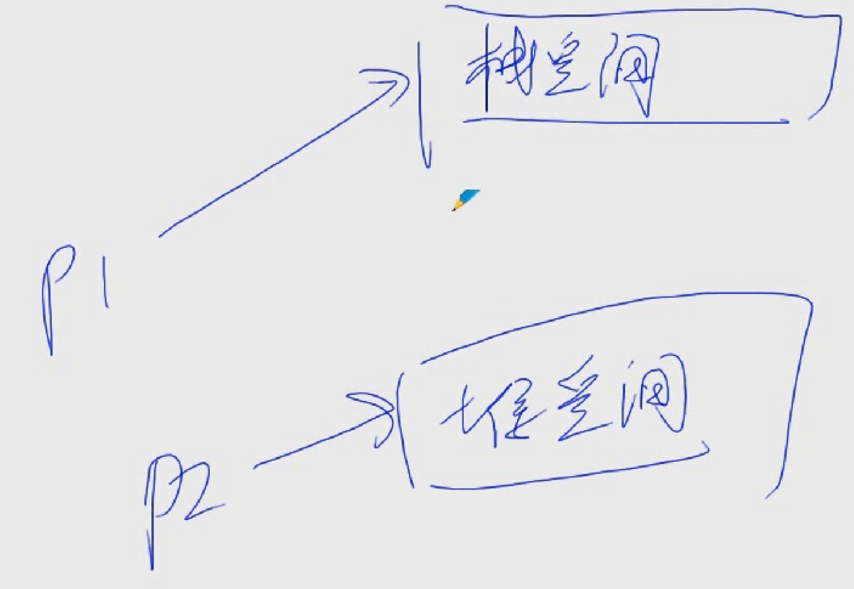
发现p0-p2 指向两个不同的数组的指针 进行相减 ----但是 结果是相差12

怎么算出来？



怎么理解未定义？ 就是没有意义 ---- 我们的一个指针 p1指向了栈空间的内存

P2指向了堆空间的内存

 相减有什么意义？ 没有意义 没有任何定义

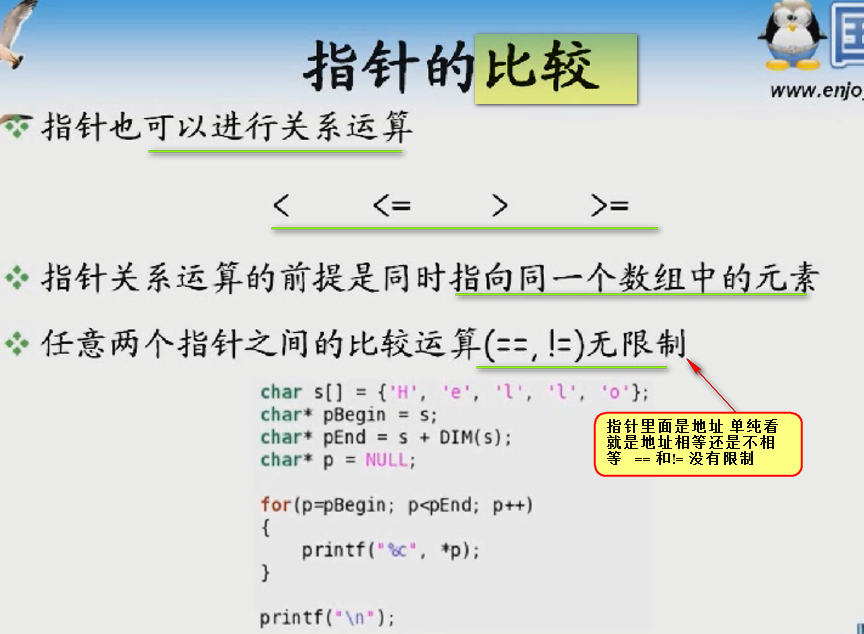
第二种未定义 就是我们这里面的结果 12 为什么是12 --- 我们这个例子比较特殊 p1 p2都是在栈上定义 和内存对齐有关系 默认四个字节对齐

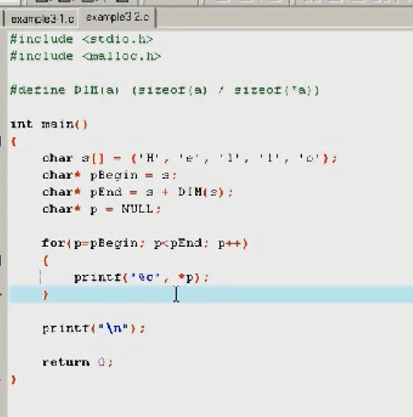


【课后练习 12怎么得出来的】

这个和编译器的内存对齐有关系 ---- 所以 结果也是未定义的

如果我们让编译器按照8字节对齐 结果又不一样的了

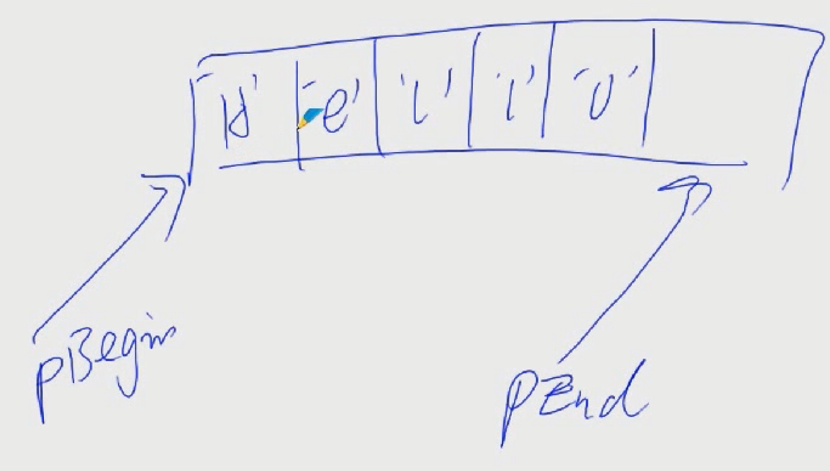




使用指针遍历数组 DIM是宏定义

 ---- 这个DIM(s)算出来之后 就是6

那么s+6就跳到了最后一个元素的后面



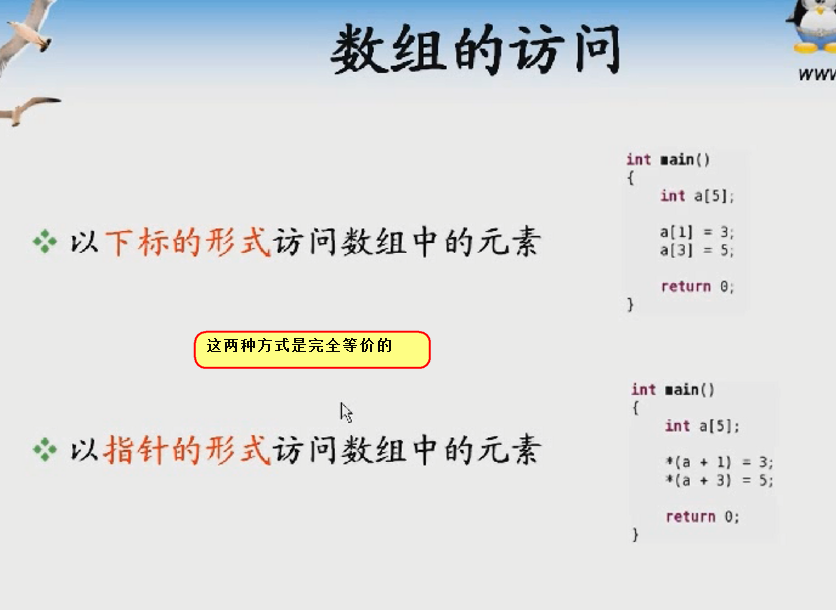
问这里面pEnd算不算越界呢？不算 因为 我们的pEnd也没访问或者修改里面的元素 --- 只是指向

 可以看到 p没有循环到pEnd本身

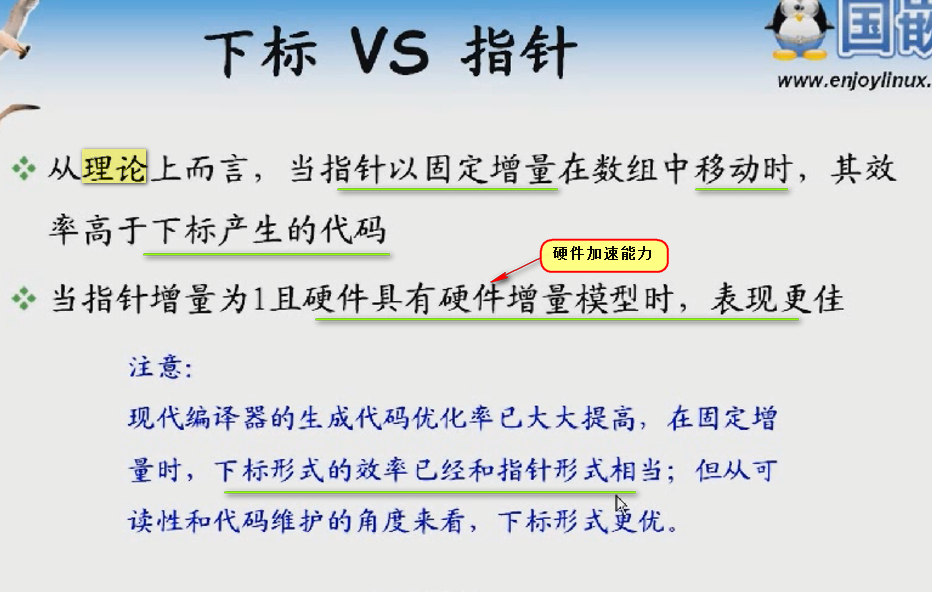
运行一下这个程序

编译通过

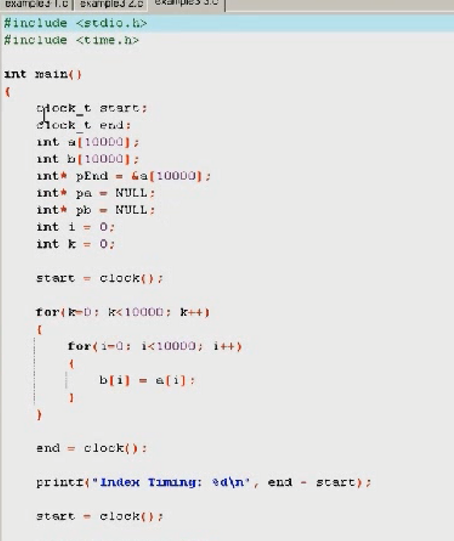
 整个字符串遍历了一遍 用了指针的比较运算



下标和指针到底用哪个？



对比下标形式跑的快 还是 指针快

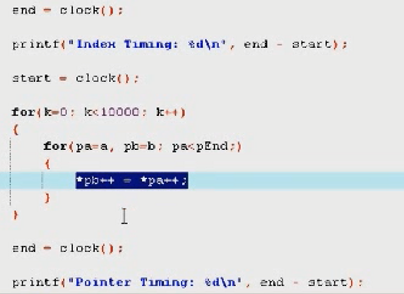


A和b都由1w个元素 for循环进行复制

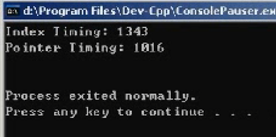
Clock();是time.h中的函数

这个是下标的形式

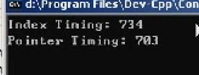
下面使用指针的形式



看谁的时间比较多 谁的时间比较少



相对而言 指针形式 节约大概300多ms

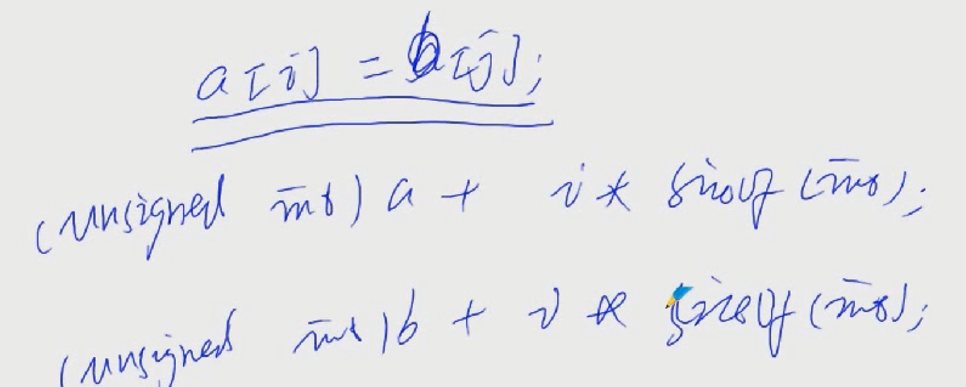


**指针的形式比较快**

指针的形式 以递增的形式访问数组 比下标快 --- 理论依据是什么？

a[i] = a[j]; ----- 在程序里面 按照公式 ---- 就会生成代码是

两边都按照公式进行展开

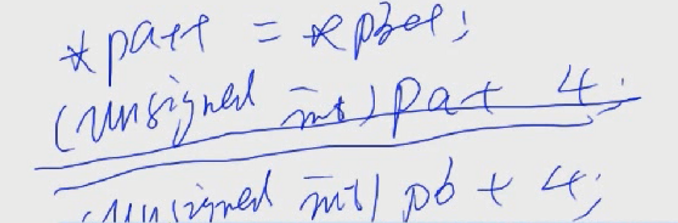


这样执行了两次乘法 乘法很耗时

对于指针

 这个是指针递增的量相同

编译器就做了这样的优化 每次直接加4 ------ 根据公式

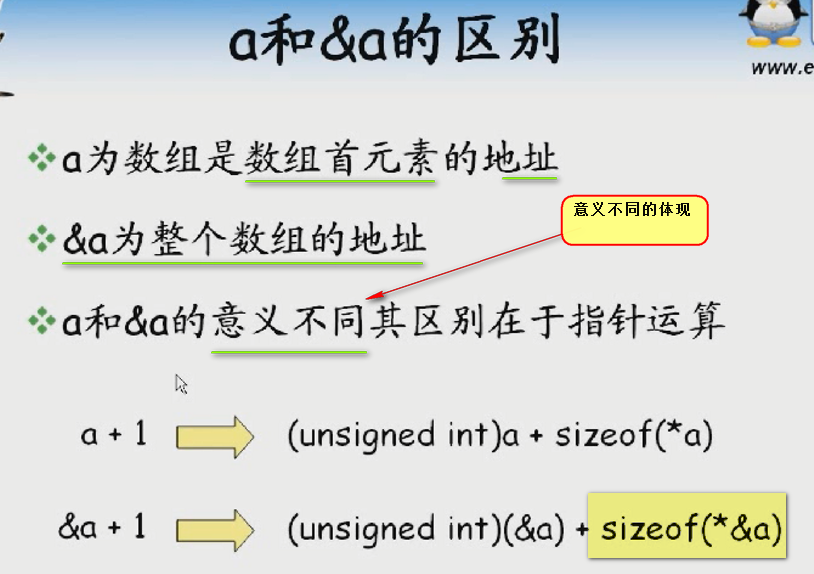


编译出的代码 下面的指针都是固定的加四 下标就是有两次乘法计算

所以 运行结果也反映了这个问题

有时候 看到下标的快一些 原因就是 我的这个进程的for循环可能被打断 执行了其他的任务 这样 clock()出来的结果 就可能有这种问题

所以 执行一百次 至少有90次是下标形式更慢

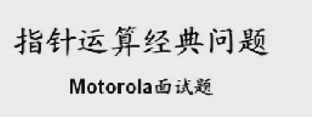


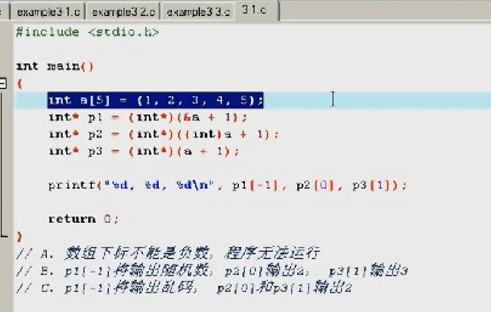
**这样&a+1 就是 (unsigned int)(&a) + sizeof(a)**

sizeof(a)是整个数组的大小 ----- 整个数组的大小 20个字节

这样&a+1就指向到最后去了

这个就是意义不同的地方





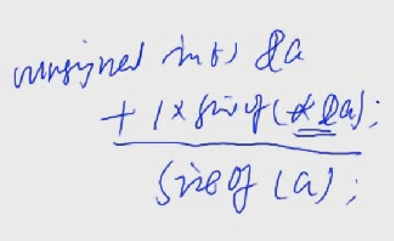
给出输出 给出三个选项 A B C

**现在 就是 数组的下标能不能是负数？**

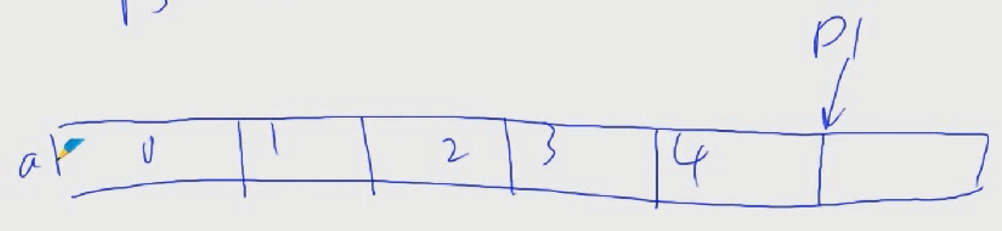
指针运算 究竟等价于什么？

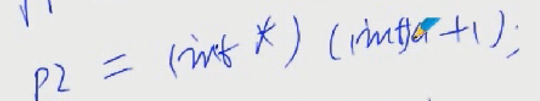


按照指针运算

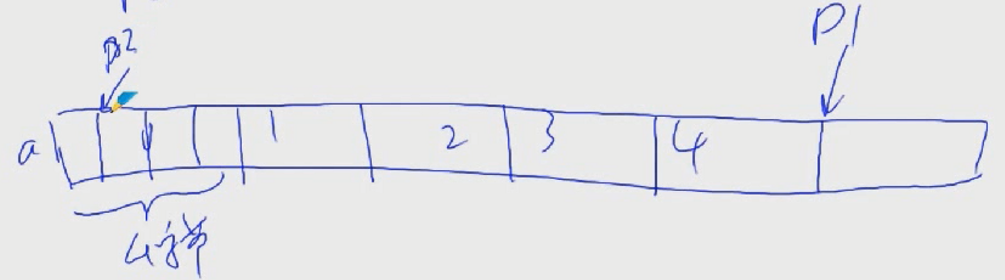


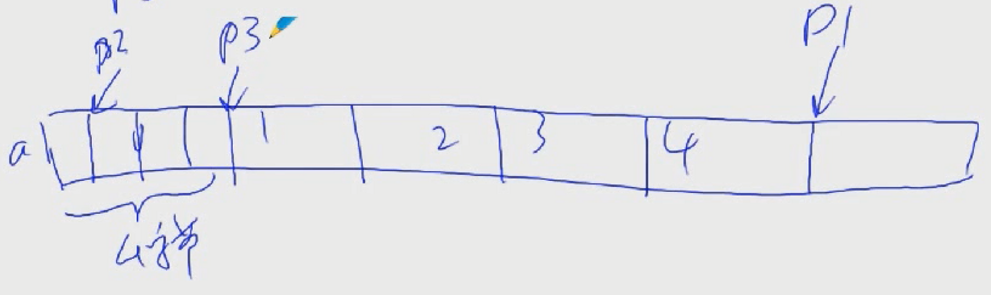
P1最终指向了末尾元素的后一个





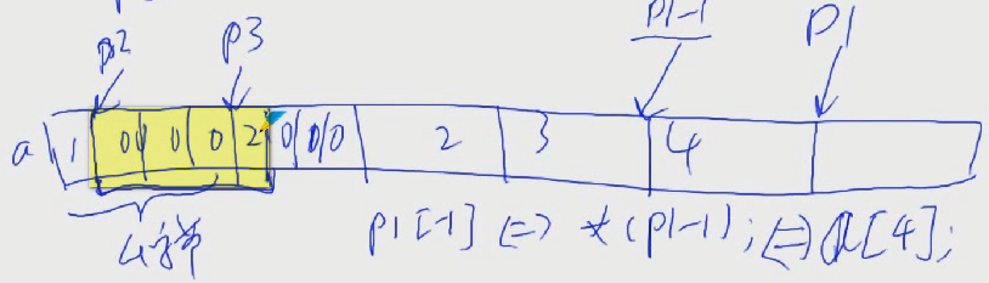
**这个是数学运算了 现在a是四个字节 那么 +1 ---- 地址上加一个字节**





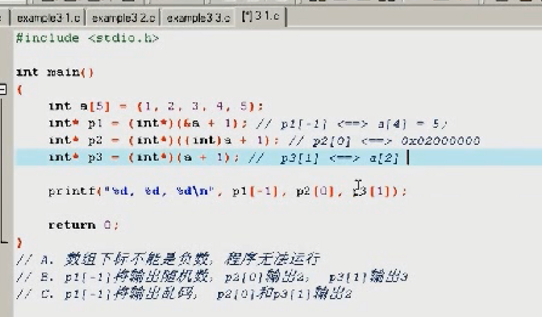
p1[-1] 就等价于\*(p1-1) ----- 打印的是a[4]

p2[0] 因为p2[0] 指向的a的四个字节的第二个 错位了 ---- 这样p2[0] ---



打印出来的高位 是 0x2000

P3[1] 很好办 打印a[2]



运行一下



把这个十六进制数转换成十进制33554432

证明我们的分析是完全正确的

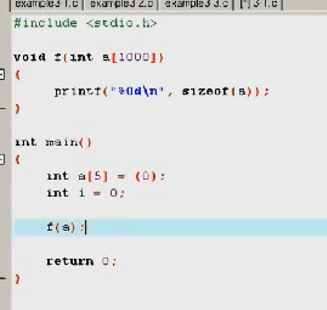
编译器确实是按照分析来运行程序的

这里面考的就是指针运算

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



当函数定义为数组参数的时候 ------ 但是 编译器就把数组参数退化为一个指针 这样长度信息就会被丢失 ----- 所以 要多写一个参数 表示数组的大小



运行结果发现为4  在函数中打印出来的

a的大小就是4 ---- 但是数组参数是int a[10000] ----- 调用的实参 是五个元素 但是 编译器没有给出警告 就是因为编译这个函数的时候 编译器把这个数组参数直接替换成指向数组的指针



所以 编译通过

**在C语言中 不会有数组参数的函数 ----- 即使写了 也会被替换成指针**



这仅仅是被看做常量指针 不是真正的指针 ---- 指针和数组对于编译器的处理不一样

这个地方是指针和数组最最重要的差别 本质终归是数组