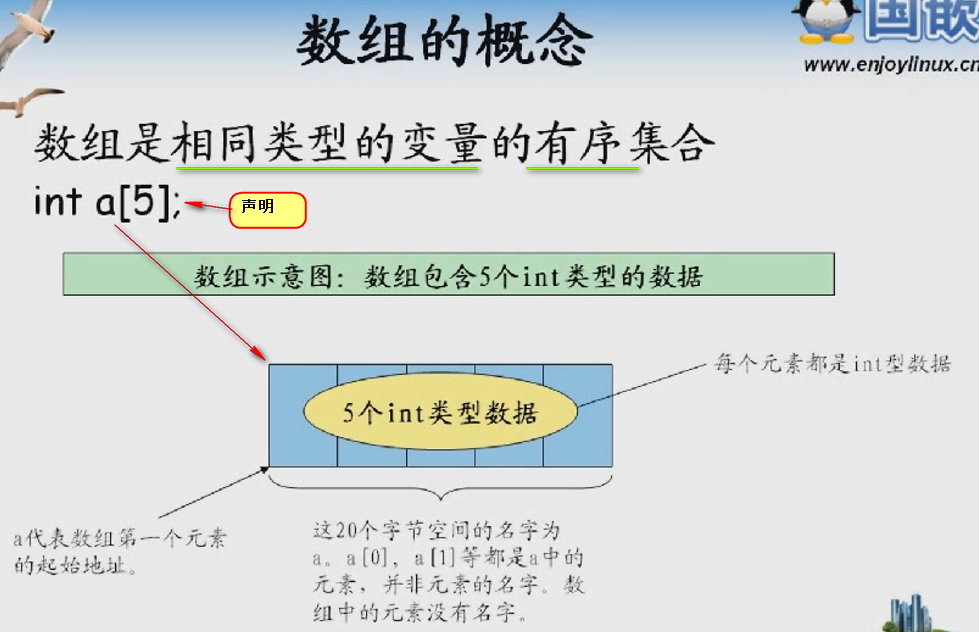
数组基础



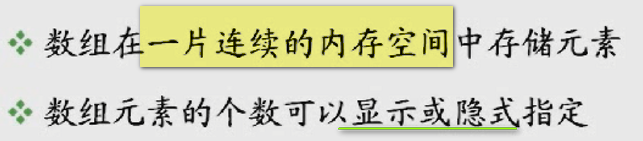


数组名代表分配数组空间的首地址

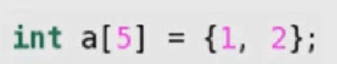
int a[5]; ---- 20个字节

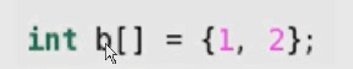
数组名本质是这20个字节的别名 --- 这个数组里面 a0 a1 a2代表元素 ---- 数组中的元素是没有名字的 ---- 只能通过数组名+下标来获取数组中的元素

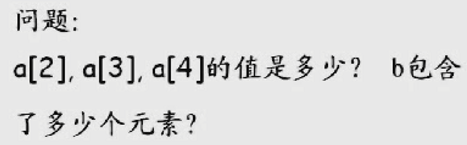
这张图就说明了数组的意义 数组名 等等



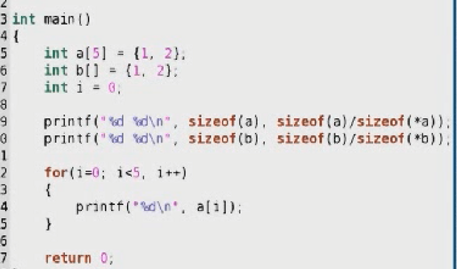
编译器怎么知道分配多大空间？

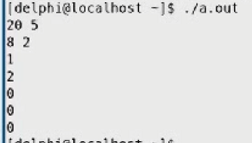
 这样是显示向编译器表明数组的方法

 这个没有大小 这时候 编译器发现初始化列表里面有两个元素 此时 编译器做法就是初始化列表中元素的个数 作为b的个数 ---- 这个就是隐式指定数组的大小



没有被声明的3 4 5应该是随机值 ---- 是这样的么？

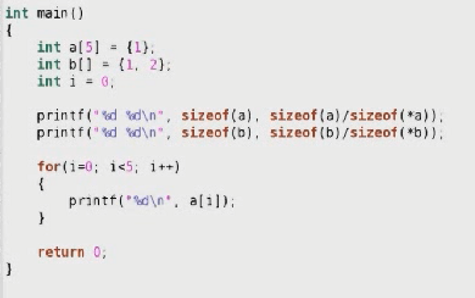
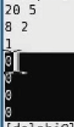




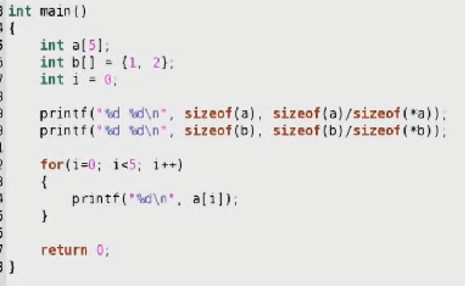
Sizeof(a) --- 20B sizeof(a)/sizeof(\*a)就是数组元素的个数 就是5 --- 打印出了20 5

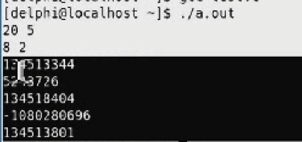
 b的分析是一致的

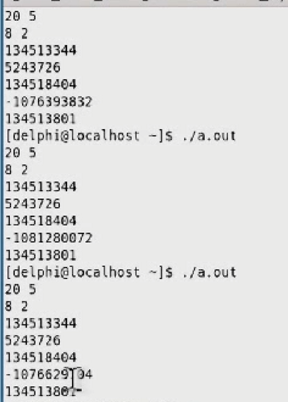
 打印没有初始化的元素 发现是0 ---- 这个是巧合 还是编译器规定？

调整不要初始化列表

 这时候看到 打印的是随机值



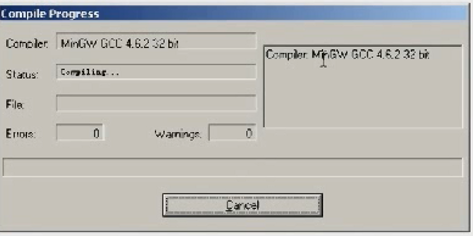


**编译器：也就是给一个指定大小的数组 初始化列表的时候 如果初始化列表的个数<指定数组大小 剩下的都是0初始化**

**如果不指定初始化列表 就全部都是随机值**

这个所有的编译器的行为

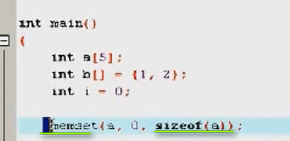
可以到windows检验一下



用的还是GCC

 和linux下面一模一样

C语言中把一个程序中的数组 初始化为0

 这个借助memset函数来做这件事 ---- 但是不是初始化 因为没在声明的时候 给出值 而是后面重新赋值 这个是赋值语句的操作 效率比初始化操作 --- memset里面是for循环 一个一个搞 效率更低

 这样才是初始化

一个数组如何初始化为全0？

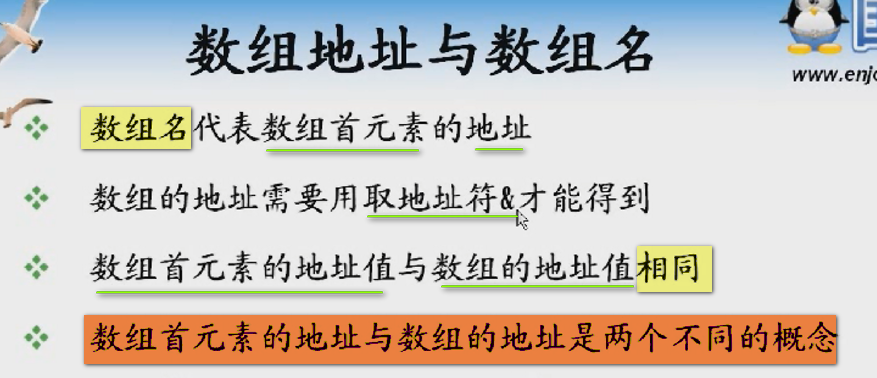
 ------ 这样 按照编译器的规则 就知道 后面的都是0

 这就巧妙利用编译器规则 指定一个数组之后 第一个值为0 后面是编译器给出的默认值就是0

这样整个数组就被初始化为0

memset不叫初始化 效率远低于初始化

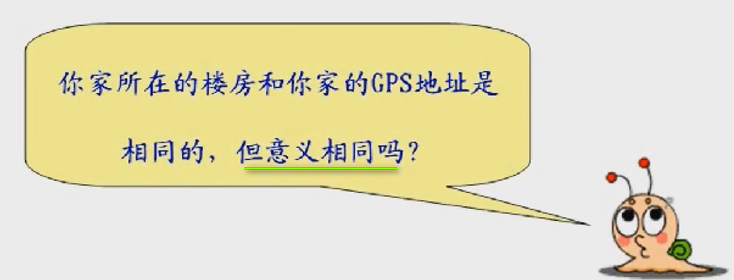
----------------



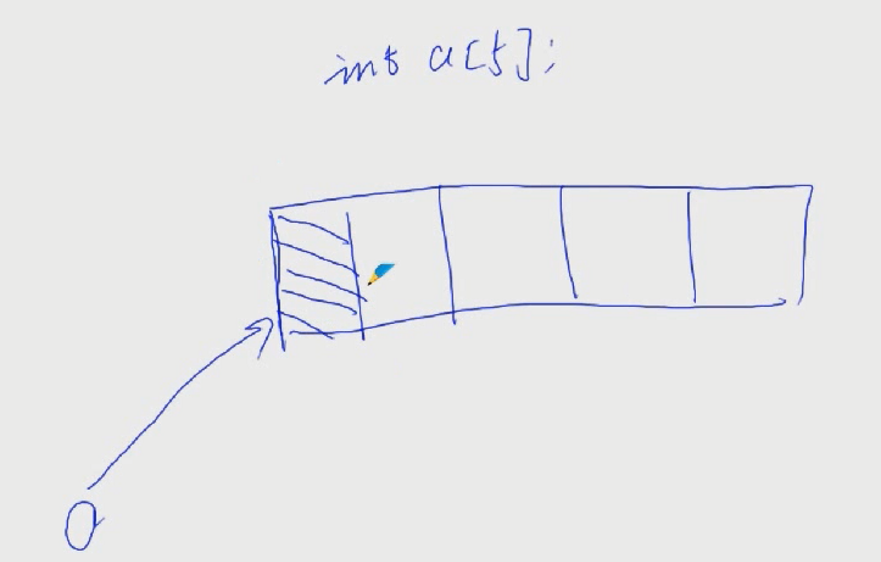
数组名仅仅代表数组第一个元素的地址 而不是整个数组的地址 ===== 有人问 数组第一个元素的地址 和 数组的地址有什么差别么？

*你家住的楼房的gps的地址是多少 还有你屋子里面的gps是多少 ---- 一测 是一样的*

*按照之前的理解 ---- 楼房的gps地址和我家的gps地址是一样的 ----- 错误的结论就是 楼房就是我家 我家就是这栋楼房 ----- 这样理解就是错误的*

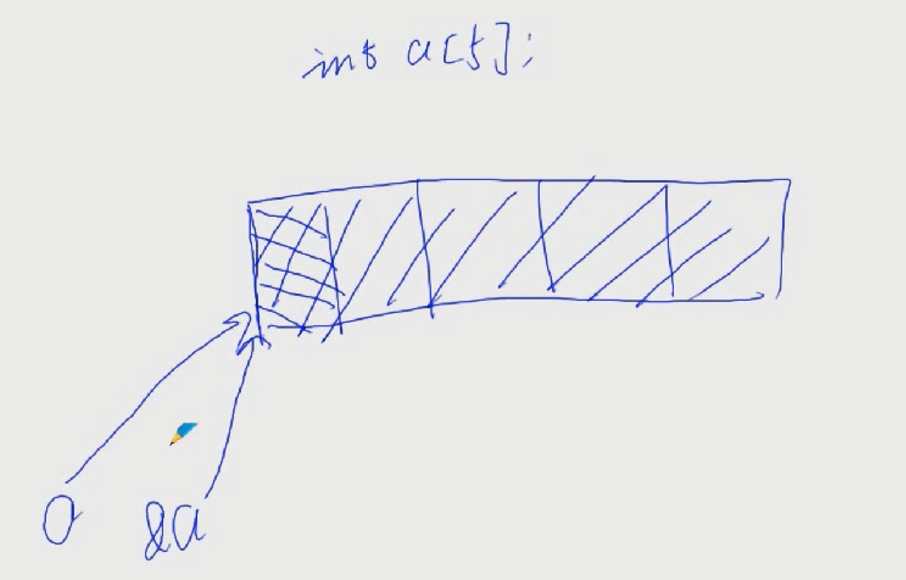


**数组首元素的地址 和 数组的地址 是完全不同的两个概念**



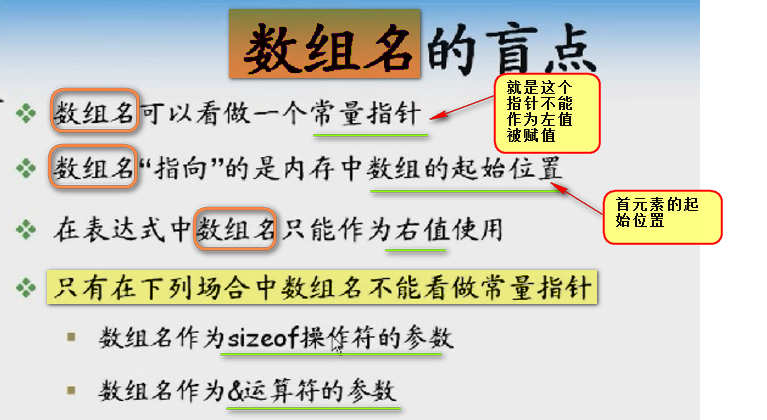
a指的是数组首元素的地址 --- 并不是整个数组的地址

数组的地址是多少？&a 是数组的地址 ---- 表示的是整个空间的起始地址 而不是首元素的地址



但是 a的值 和 &a的值是一样的

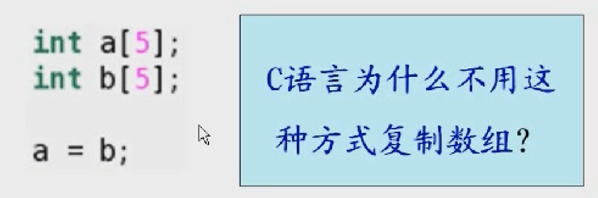
虽然a和&a的绝对值是一样的 但是 物理意义是完全不同的 ----- 你家的gps和小区的gps一样 小区就是你家的？



【注意 程序中 是 “看做” 常量指针 但是 实际上 不是常量指针】

【记住数组名不再看成常量指针的情况】

如果C语言支持这样的复制数组



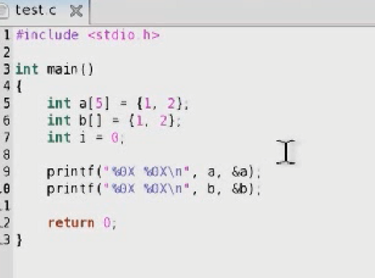
C语言为什么不用这种方式 原因就是C语言中的规则 一般情况下 数组名是一个常量指针 Type\* const constPtr; ----- 这个constPtr只能指向唯一的一个内存单元 不能指向别的了

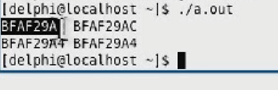
所以 数组名不能出现在左值的位置

所以 a =b 这种不支持

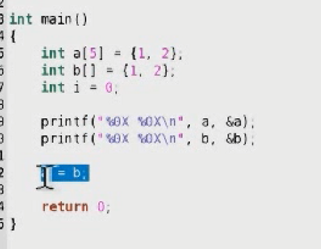
下面的程序加强对数组名的理解

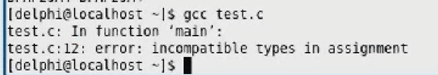
\*\*数组名和数组的地址 的绝对值是否一致



 两个的地址是一样的 但是 含义不一样

\*\*C为什么不这样赋值？

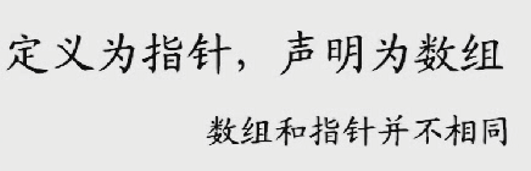
 ----- 一个2个元素 另一个5个 怎么赋值？



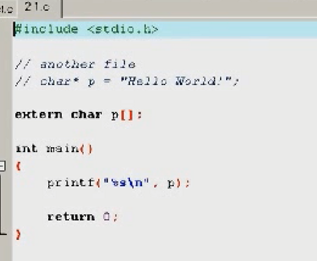
编译器直接给出了无法兼容的类型赋值

a b都是常量指针 所以 不能出现在左值的位置

同时 都不知道你的数组的大小 ---- 所以这种方式不支持



数组和指针真的一样么？编译器处理的行为真的一样么？

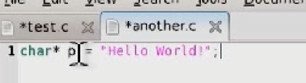


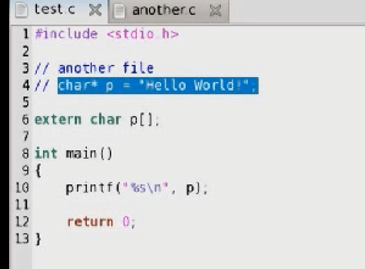
程序里面声明了一个外部的数组 ---- p[]是在另一个文件里面声明的 ---- 另一个文件实际上是一个指针 

按理说 应该是打印Hello World

运行一下这段代码

Another.c里面定义一个全局变量





编译通过

 结果打印出来是乱码

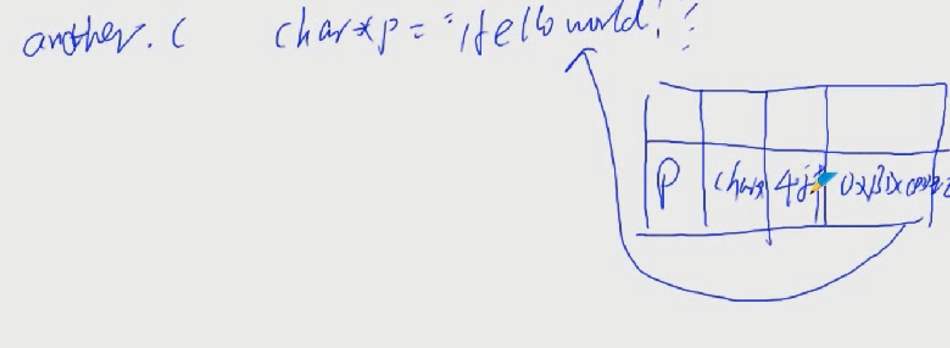
**------ 代码的角度可以把数组名看成常量指针 只是看成！！！ 我们看看数组是如何被编译器处理的 指针又是如何被编译器处理的！**

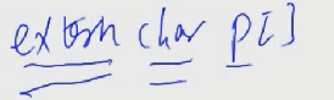
anther.c里面 char\* p = “Hello World”;

test.c里面 extern char p[];

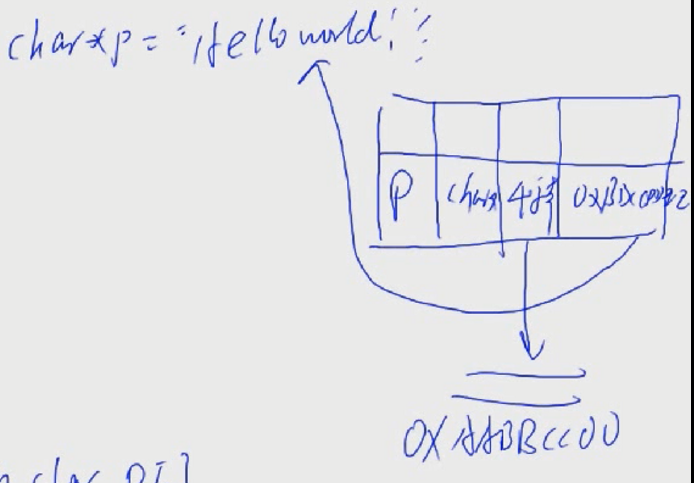
编译器编译的时候 放一个符号 – 放一个p到符号表

然后为p分配空间 发现p是指针 就分配4B空间 并且值指向Hello World 一个地址



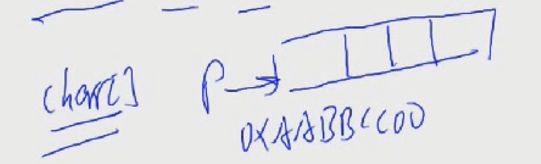
编译test.c 发现extern  知道p在符号表中已经存在了 --- 到符号表中找

现在假设 为p分配的4B的空间的首地址是 



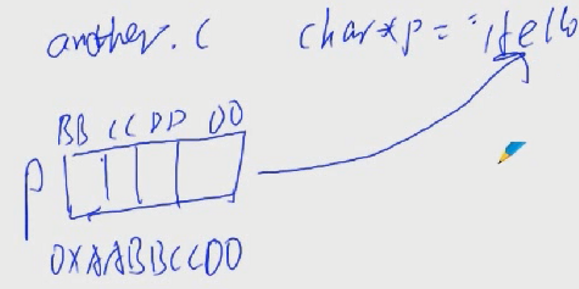
编译器发现 数组p 代表了一片空间 这篇空间是从开始的四个字节

并且 是char的数组



这就是编译器做的事情

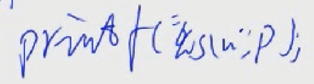
但是 这四个字节实际上是Hello World的地址 ----- 所以 打印出来是乱码



编译器发现是指针 printf



发现p是指针 ------ 到p所代表的内存 把里面地址取出来 找对应的空间 才是hello world

如果 我们直接 

发现这个是数组 是一个Char类型的 就把数组对应的内存空间的内容打印出来 就直接打印出来

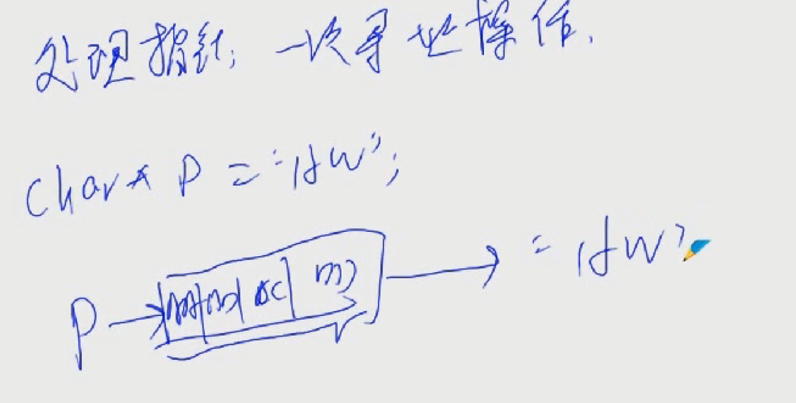
 没有真正打印

这个是本质原因 也就是 我们的编译器做的动作如下

处理指针和数组不同

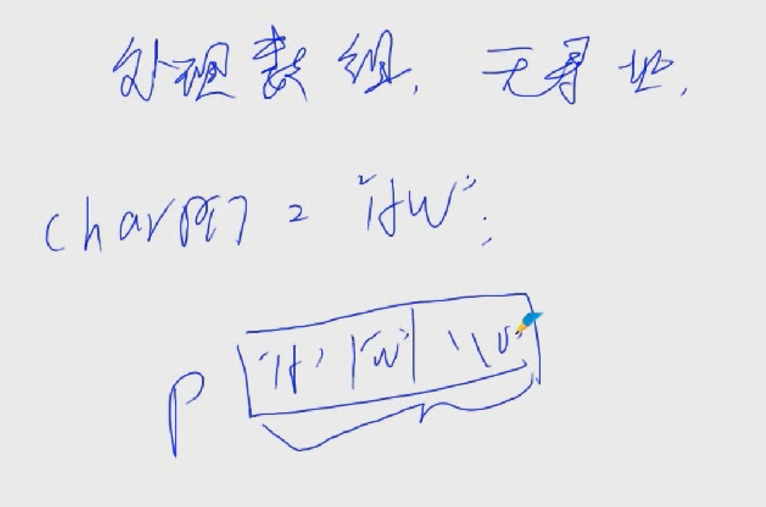
\*\*\*处理指针 要经过一次寻址 --- 寻址操作的过程 就是

Char\* p = “HW”;



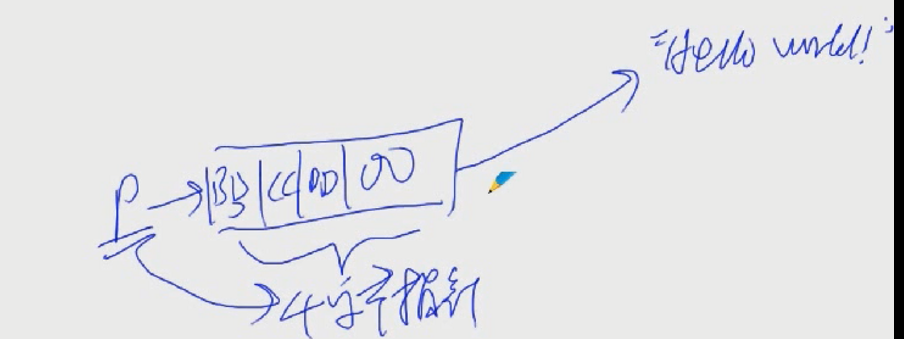
访问p的时候 先访问这里面的内存空间 然后得到的地址 获取最终的内存的内容

\*\*\*\*\*处理数组 无寻址

 没有那一次的寻址操作 ----- 所以 就认为这块的空间就是数组中的内容 但是 指针多一次寻址

 把指针本身的内容打印出来 就是乱码的

p本身代表4B的指针 里面放了一个地址



在test.c里面 就直接认为这四个字节就是内容 ----- 我们手工做这次寻址

因为这个p误认为是数组 ---- 所以 强制类型转换



这样是一个指针 ---- 强转之后是一个指针 unsigned int\* ----

如何把这四个字节的地址取出来？

 ----- 这样得到地址之后 \*一下 就可以打印结果了



模仿编译器的行为 在写错的情况下 手工寻址 原理就是 做一次寻址 ---- 将首地址强转为unsigned int\* 【地址没有负数 所以 无符号型】

虽然有办法解决这个问题 代码里面遵循一个原则 --- -声明为一个指针 外部文件也声明一个指针 否则数组 就是数组对应

因为编译器处理指针 和 处理数组不同

这个就是经典的例子

---------------------

