**变量与地址**

变量其实就是程序员对某一块空间的命名（叫什么都无所谓），这块空间的绝对名字就是它的地址

**指针与指针变量**

指针就是地址（指针等价于地址），是一个具有指向作用的值，地址是一个常量，不允许改变

指针变量就是用来存放指针的变量，可以改变，你可以指向这，也可以指向那

不管你的指针是几级的指针，不管你的指针是什么样的类型，指针在某一个平台下所占的空间大小都是确定的，如int \*p; char \*q; float \*r;等，包括后面的结构体指针，在64位平台下都占8个字节，32位是4个字节

虽然指针类型的变量占用的字节个数都是统一的，但是不同类型的指针在做运算时，运算的意义不相同，比如int i = 1; int \*p; p = &i;，然后\*p，它会去到i所占地址取4个字节的内容，如果是int i = 1; char \*p; p = &i;，然后\*p，它会去到i所占地址取1个字节的内容，这就错了，所以指针变量和它所指向的值，它们两的类型一定要相符

int i = 1;

int \*p = &i;

int \*p这里，\*实际上是跟着int走的，为了方便理解，可抽象为TYPE NAME = VALUE;，在这里TYPE就是int \*

int i = 1;

int \*p = &i;

int \*\*q = &p;

2级指针变量q存储的是2级指针，指针就是地址，也就是1级指针变量的地址&p，1级指针变量p存储的是1级指针，指针就是地址，也就是变量的地址&i

同理n级指针变量存储的是n级指针，也就是n-1级指针变量的地址

**直接访问与间接访问**

直接访问就是直接用变量名去访问，间接访问就是用指针去访问

**空指针和野指针**

空指针，int \*p = NULL;，这个NULL是一个宏，一般情况下值为0，也就是我让指针指向了起始地址号为0的那块空间，有什么样的作用呢，我会得到一个提示，我如果当前指针非法操作，我会得到一个段错误的提示。什么时候要用空指针呢，那就是，指针定义出来，我还不知道它要指向谁或者不知道它要怎么来用，那就可以把指针定义为空，因为我们在当前系统上来规定，0号这块空间不分配给任何进程，同样也不会分配给你，所以当你企图去写这块空间的时候，当然会给你报段错误。把指针置空就是为了防止野指针

野指针，int \*p;，当前指针所指向的空间是不确定的，或者说压根就没有指向，你却使用了它，如果你直接去访问这个指针所指向的值（\*p），看你的编译器版本了，有的审得严的会报段错误，审得松的不会报，可以成功访问，但是如果你试图去修改这个指针所指向的值（\*p=1），那一定会报段错误。要避免野指针那就给它置空，因为0号这块空间不分配给任何进程，所以当你企图去操作这块空间的时候，一定会给你报错。

**空类型的指针**

空类型的指针，void \*p = NULL;，这是一个百搭的数据类型，任何类型的指针值都能够把自己的值赋给void \*，void \*也能够把自己的值赋给任何类型的指针。什么时候会用到呢，那就是，在我不确定你到底需要一个什么样的数据类型的时候

比如memcpy原型，void \*memcpy(void \*dest, const void \*src, size\_t n);，我们需要从src指向的位置拷贝n个字节到dest所指向的空间，我既不知道你原始数据存放的是什么类型的数值，我也不知道你的目标位置用来存放什么类型的数值，那怎么办，那就void \*

**定义与初始化的书写规则**

int i = 1;

int \*p = &i;，也可以int\* p = &i;

int \*\*q = &p;，也可以int\*\* q = &p;

**指针运算**

指针能做的运算并不多，无非是&，\*，以及一些关系运算，比如，两个指针来比较大小，比的是这两个所指向的地址值的高低，什么情况下有必要这么做，那就是，当两个指针所指向的空间是一块连续的内存空间，比如说，两个指针都指向同一个数组，在这种情况下，两个指针可以来进行一些关系或者是判断的一些比较，比如++，--这些运算，实际上是改变当前指针目前所在的一个位置

**指针与数组的关系**

指针与一维数组

int a[3] = {1, 2, 3};

int \*p = a;

int i;

那么a[i] = \*(a + i) = \*(p + i) = p[i]

并且&a[i] = a + i = p + i = &p[i]

看起来好像a和p是一样的，程序里能用a的地方就能用p，但实际上它们还是有区别的，本质上的区别就在于，a是数组名，是一个表示地址的常量，而p是一个变量，是一个用来存放地址的变量，总结一句话，a是常量，b是变量。a不能a++，因为a++等价于a = a+1，常量怎么可能在左边，但p++是可以的

\*p++计算优先级是\*p之后再p++，相当于先取值，指针再移动1

如果是(\*p)++，那就先取值，然后p所指向的那个值再+1

其实数组名并不重要，在定义数组时甚至可以没有

int \*p = (int [3]) {1, 2, 3};

int i;

for(i = 0; i < 3; i ++)

printf(&p[i], p[i]);

这没有数组，这相当于创建了一个匿名的对象或者说是临时对象，其实严格来说是有数组的，但数组没有名字，我只是通过p来引用数组当中所有的元素，包括打印出数组元素的地址，这里就可以看出，指针和数组的关系其实就是一种引用方式而已，你的一维数组和一维指针除了一个是常量，一个是变量外，完全可以等量代换

指针与二维数组

a[i][j] = \*(\*(a + i) + j)，相当于a是一个行指针，移动i行，然后取\*变成列指针，再移动j列，最后取\*得到元素值

int a[2][3] = {1, 2, 3, 4, 5, 6};

int i, j;

int \*p;

这个时候p = a;这句话就有问题了，理论上来讲，一般的编译器会在编译的时候给你报一个指针类型赋值时不兼容的警告，它会认为你的等号右边的a是一个地址常量，它所表示的是一个可以在行间进行跳跃的一个指针，而左边的p是一个非常典型的在列上移动的，从指针的角度来讲，它俩是不一样的，所以你程序这样写肯定会有一个警告，还是建议调到没有警告为止，怎么解决呢，可以用p = \*a;，因为\*a相当于\*(a + 0);，把a这个行指针降级了，降成了列指针，这不就匹配上了，还可以用p = &a[0][0];，道理一样，这么做，就没有警告了

指针与字符数组

使用起来跟前面讲的没什么区别

char str[] = “I love china!”;

char \*p = str+7;

puts(str);，输出的是I love china!

puts(p);，从p所指向的位置开始，依次向后输出，直到尾0位置，输出的是china!

char str[] = “hello”;

str = “world”;，显然是错的，str可是常量，要想str指向world，用下面的

strcpy(str, “world”);

char str[] = “hello”;

sizeof(str)的结果是6，strlen(str)的结果是5

如果是

char \*str = “hello”;

sizeof(str)的结果是8（因为64位机器指针的大小都是固定为8个字节的），strlen(str)的结果依然是5

char \*str = “hello”;

strcpy(str, “world”);，是错误的，因为你试图用”world”去修改指针str指向的串常量“hello”，要想str指向world，用下面的

str = “world”;，指针嘛，放弃指向hello这个串，转而指向world这个串就可以了

**指针常量与常量指针的关系（const与指针）**

const int a;

int const a;

const非常重要，它的作用是把某些内容进行常量化

如果希望某个数一直不变，但一改能全改，以前我们一直用的宏，比如

#define PI 3.14

但宏有个缺点就是不检查语法你即使

#define PI abcd，它在预处理阶段或者说编译阶段是不报错的，它只做替换，一点问题都没有，其实更保险的做法是

const float pi = 3.14;，加const的意思就是，我需要当前的变量一直保存当前的常量值不变，就相当于pi这个float类型的变量被const给常量化了，这种方法通常用来保存一些不希望发生变化的常量值，优点是检查语法，你给个abcd显然是不成立的

const float pi = 3.14;

pi = 3.14159;，显然是错误的，可我就是想改怎么办呢，用指针

float \*p = &pi;

\*p = 3.14159;，这就能改，但是编译器会给你报一个警告，这块空间我不想改，你却通过一个间接引用给我改了，这哪是警告啊，在很多时候都构成error了，所以还是那句话，要调到没有警告为止，怎么办呢，那就是，当前指针也应该修饰一个const

const float \*p = &pi;

\*p = 3.14159;，这时这句话就错误了，你既然把pi用const修饰了，我还是不希望你改变pi

const int \*p;

int const \*p;

都是常量指针，只是写法不同，类型放在前面和后面在这种写法下没有区别

常量指针，指针的指向可以发生变化，但是指针指向的当前空间里的值不能发生变化，也就是说当前指针所指向的目标变量不能变（其实应该说不能通过指针去改变指针所指向的值）

int i = 1;

int j = 100;

const int \*p = &i;

i = 10;，正确

p = &j;，正确

\*p = 10;，错误

这就与上面的

const float pi = 3.14;

pi = 3.14159;，错误

float \*p = &pi;

\*p = 3.14159;

呼应上了，pi被const修饰了，我不能通过pi改，但我可以用指针来改，因为指针变量没有加const修饰，所以const来限制某块空间的值不能变化，只是限制不能通过这个名字去变化，而不是说那块空间就被锁定了

int \*const p;

指针常量，指针的指向不能变化，但指针所指向目标变量的值可以发生变化

int i =1;

int j = 100;

int \* const p = &i;

\*p = 10;，正确

p = &j;，错误

const int \*const p;

既是指针常量又是常量指针，指针的指向不能变化，指针所指向目标变量的值也不可以发生变化

int i =1;

int j = 100;

const int \* const p = &i;

p = &j;，错误

\*p = 10;，错误

讲完了指针常量和常量指针就能够解释为什么很多函数或者说系统调用，它都会在某些值的前面修饰const

比如open的函数原型int open(const char \*pathname, int flags);，第一个参数是常量指针，对方传给你一个文件，我来打开进行操作，对于用户传过来的pathname，我是用常量指针来接管，那就说明我通过pathname这个指针不能够改变你传过来那个串的内容，因为它保护的是目标变量不能变，你放心传参，我不会通过这个指针对你传过来的内容做任何修改。我们在封装一些函数给别人用的时候，也应该借鉴这种封装方式，当你做一些接口函数来给别人用的时候，如果需要对方传过来一些指定的你不会去修改或者说不允许被擅自修改的一些内容的话，你最好要加上const来修饰，这是一种约定俗成的使用方法

**指针数组和数组指针**

指针数组

定义

[存储类型] 数据类型 \*数组名[长度]

本质上是一个数组，只不过里面存放的是一个个指针

如int \*arr[3];，根据TYPE NMAE = VALUE;，它的type就是int \*[3]，NAME就是arr

char \*name[5] = {“Follow me”, “Basic”, “Great”, “Fortran”, “Computer”};

这就是一个数组，数组中一共有5个元素，每个元素都是char \*，所以我用串常量来给每个char \*赋初值，输出时

for(int i = 0; i < 5; i++)

puts(name[i]);

数组指针

定义

[存储类型] 数据类型 (\*指针名)[下标] = 值

总之，它本质上还是一个指针，指针指向什么呢，指向数组

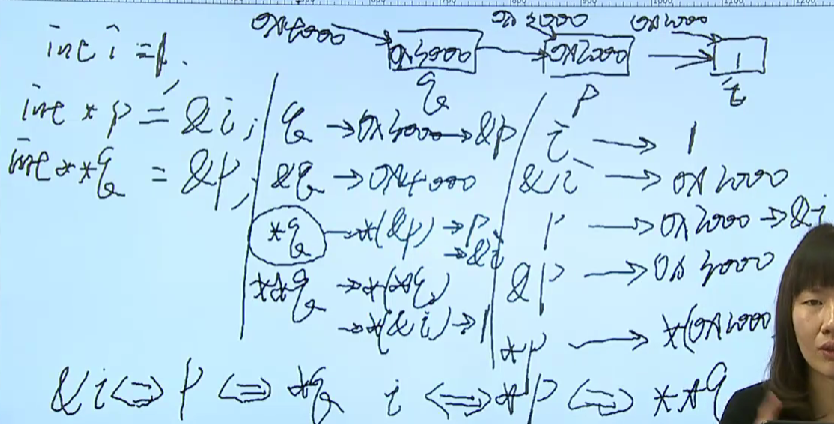
如int (\*p)[3];，根据TYPE NMAE = VALUE;，它的type就是int[3] \*，NAME就是p，原来你习惯写int \*p;，那就是定义了一个指针变量用来指向一个整型元素，p做一个加1的操作，指的是移动一个int类型的大小，那现在我是定义了一个指针变量用来指向int[3]这样一个元素的起始位置，由于我p指向的是int[3]这样的类型，那我p做一个加1的操作，是一下移动3个int类型的大小

int a[2][3] = {1, 2, 3, 4, 5, 6};

int (\*p)[3] = a;

前面讲过，如果a是一维数组，p是一维指针，那么程序后面除了违背对常量（a是常量，p是变量）的操作外，其它地方a和p均可互换，现在一维变二维，结论依然如此，只要不违背对常量的操作，a和p依然可等量替换

**多级指针**



如果你真的理解了1级指针，那n级指针都不怕，一般来说，掌握到2级指针就够了

pthread\_join函数原型int pthread\_join(pthread\_t thread, void \*\*retval);，功能是给一个线程收尸，第二个参数就是一个void \*\*retval，意思是把你所要收的线程的值拿回来，而线程给你的值是一个void \*，所以你需要做一个一级指针变量void \*的地址，来把对方给你的void \*给接收一下