指针知识点

一、指针，

在计算机科学中，**指针**（Pointer）是编程语言中的一个对象，利用地址，它的值直接指向（points to）存在电脑存储器中另一个地方的值。

由于通过地址能找到所需的变量单元，可以说指针是对数据对象或函数的一种引用。指针有多重用途，例如定义“传址调用”函数，它还可以实现动态数据结构，例如链表和树。

通常，管理大量数据的有效方法不是直接处理数据本身，而是使用指向数据的指针。

0.指针的声明

非数组对象的指针： 类型 *[类型限定符列表] 名称 [= 初始化器]

举个例子：

```
int *p; // 声明p作为一个指向int的指针
int var = 77;
p = &var; //使得p指向变量var
int *p1 = &var; // 类型限定符例如const,volatile等
printf("var value = %d location %p\np value = %d location %p p1 value = %d, p1 location = %p", var, &var, p , &p, p1, &p1);
```

1.指针的类型

从语法的角度看，你只要把指针声明语句里的指针名字去掉，剩下的部分就是这个指针的类型。这是指针本身所具有的类型。让我们看看例一中各个指针的类型：

(1)int*ptr;//指针的类型是int*

(2)char*ptr;//指针的类型是char*

(3)int**ptr;//指针的类型是int**

怎么样？找出指针的类型的方法是不是很简单？

2.指针所指向的类型

当你通过指针来访问指针所指向的内存区时，指针所指向的类型决定了编译器将把那片内存区里的内容当做什么来看待。

从语法上看，你只须把指针声明语句中的指针名字和名字左边的指针声明符*去掉，剩下的就是指针所指向的类型。例如：

(1)int*ptr; //指针所指向的类型是int

(2)char*ptr; //指针所指向的的类型是char

(3)int**ptr; //指针所指向的的类型是int*

在指针的算术运算中，指针所指向的类型有很大的作用。

指针的类型(即指针本身的类型)和指针所指向的类型是两个概念。当你对C 越来越熟悉时，你会发现，把与指针搅和在一起的"类型"这个概念分成"指针的类型"和"指针所指向的类型"两个概念，是精通指针的关键点之一。

3.指针的值----或者叫指针所指向的内存区或地址

指针的值是指针本身存储的数值，这个值将被编译器当作一个地址，而不是一个一般的数值。在32 位程序里，所有类型的指针的值都是一个32 位整数，因为32 位程序里内存地址全都是32 位长。指针所指向的内存区就是从指针的值所代表的那个内存地址开始，长度为sizeof(指针所指向的类型)的一片内存区。以后，我们说一个指针的值是XX，就相当于说该指针指向了以XX 为首地址的一片内存区域；我们说一个指针指向了某块内存区域，就相当于说该指针的值是这块内存区域的首地址。指针所指向的内存区和指针所指向的类型是两个完全不同的概念。在例一中，指针所指向的类型已经有了，但由于指针还未初始化，所以它所指向的内存区是不存在的，或者说是无意义的。

以后，每遇到一个指针，都应该问问：这个指针的类型是什么？指针指的类型是什么？该指针指向了哪里？（重点注意）

4 指针本身所占据的内存区

指针本身占了多大的内存？你只要用函数sizeof(指针的类型)测一下就知道了。在32 位平台里，指针本身占据了4 个字节的长度。指针本身占据的内存这个概念在判断一个指针表达式（后面会解释）是否是左值时很有用。

二、指针的算术运算

指针可以加上或减去一个整数。指针的这种运算的意义和通常的数值的加减运算的意义是不一样的，以单元为单位。例如：

例二：

```
char a[20];
int *ptr=(int *)a; //强制类型转换并不会改变a 的类型
ptr++;
```

在上例中，指针ptr 的类型是int*,它指向的类型是int，它被初始化为指向整型变量a。接下来的第3句中，指针ptr被加了1，编译器是这样处理的：它把指针ptr 的值加上了sizeof(int)，在32 位程序中，是被加上了4，因为在32 位程序中，int 占4 个字节。由于地址是用字节做单位的，故ptr 所指向的地址由原来的变量a 的地址向高地址方向增加了4 个字节。由于char 类型的长度是一个字节，所以，原来ptr 是指向数组a 的第0 号单元开始的四个字节，此时指向了数组a 中从第4 号单元开始的四个字节。我们可以用一个指针和一个循环来遍历一个数组，看例子：

例三：

```
int array[20]={0};
int *ptr=array;
for(i=0;i<20;i++)
{
    (*ptr)++;
    ptr++;
}
```

这个例子将整型数组中各个单元的值加1。由于每次循环都将指针ptr加1 个单元，所以每次循环都能访问数组的下一个单元。

再看例子：

例四:

```
char a[20]="You_are_a_coder";
int *ptr=(int *)a;
ptr+=5;
```

在这个例子中，ptr 被加上了5，编译器是这样处理的：将指针ptr 的值加上5 乘sizeof(int)，在32 位程序中就是加上了5 乘4=20。由于地址的单位是字节，故现在的ptr 所指向的地址比起加5 后的ptr 所指向的地址来说，向高地址方向移动了20 个字节。 在这个例子中，没加5 前的ptr 指向数组a 的第0 号单元开始的四个字节，加5 后，ptr 已经指向了数组a 的合法范围之外了。虽然这种情况在应用上会出问题，但在语法上却是可以的。这也体现出了指针的灵活性。如果上例中，ptr 是被减去5，那么处理过程大同小异，只不过ptr 的值是被减去5 乘sizeof(int)，新的ptr 指向的地址将比原来的ptr 所指向的地址向低地址方向移动了20 个字节。 下面请允许我再举一个例子:(一个误区)

例五:

```
#include<stdio.h>
int main()
{
    char a[20]=" You_are_a_girl";
    char *p=a;
    char **ptr=&p;
    //printf("p=%d\n",p);
    //printf("ptr=%d\n",ptr);
    //printf("*ptr=%d\n",*ptr);
    printf("**ptr=%c\n",**ptr);
    ptr++;
    //printf("ptr=%d\n",ptr);
    //printf("*ptr=%d\n",*ptr);
    printf("**ptr=%c\n",**ptr);
}
```

误区一、

输出答案为Y 和o

误解:ptr 是一个char 的二级指针,当执行ptr++;时,会使指针加一个sizeof(char),所以输出如上结果,这个可能只是少部分人的结果.

误区二、

输出答案为Y 和a误解:ptr 指向的是一个char *类型,当执行ptr++;时,会使指针加一个sizeof(char *)(有可能会有人认为这个值为1,那就会得到误区一的答案,这个值应该是4,参考前面内容), 即&p+4; 那进行一次取值运算不就指向数组中的第五个元素了吗?那输出的结果不就是数组中第五个元素了吗?答案是否定的.

正解:

ptr 的类型是char **,指向的类型是一个char *类型,该指向的地址就是p的地址(&p),当执行ptr++;时,会使指针加一个sizeof(char*),即&p+4;那*(&p+4)指向哪呢,这个你去问上帝吧,或者他会告诉你在哪?所以最后的输出会是一个随机的值,或许是一个非法操作.

总结一下:

一个指针ptrold 加(减)一个整数n 后，结果是一个新的指针ptrnew，ptrnew 的类型和ptrold 的类型相同，ptrnew 所指向的类型和ptrold所指向的类型也相同。ptrnew 的值将比ptrold 的值增加(减少)了n 乘sizeof(ptrold 所指向的类型)个字节。就是说，ptrnew 所指向的内存区将比ptrold 所指向的内存区向高(低)地址方向移动了n 乘sizeof(ptrold 所指向的类型)个字节。指针和指针进行加减：两个指针不能进行加法运算，这是非法操作，因为进行加法后，得到的结果指

向一个不知所向的地方，而且毫无意义。两个指针可以进行减法操作，但必须类型相同，一般用在数组方面，不多说了。

三、运算符&和*

这里&是取地址运算符，*是间接运算符。

&a 的运算结果是一个指针，指针的类型是a 的类型加个*，指针所指向的类型是a 的类型，指针所指向的地址嘛，那就是a 的地址.

*p 的运算结果就五花八门了。总之*p 的结果是p 所指向的东西，这个东西有这些特点：它的类型是p 指向的类型，它所占用的地址是p所指向的地址。

例六：

```
int a=12; int b; int *p; int **ptr;
p=&a; //&a 的结果是一个指针，类型是int*，指向的类型是
//int，指向的地址是a 的地址。
*p=24; //*p 的结果，在这里它的类型是int，它所占用的地址是
//p 所指向的地址，显然，*p 就是变量a。
ptr=&p; //&p 的结果是个指针，该指针的类型是p 的类型加个*，
//在这里是int **。该指针所指向的类型是p 的类型，这
//里是int*。该指针所指向的地址就是指针p 自己的地址。
*ptr=&b; //*ptr 是个指针，&b 的结果也是个指针，且这两个指针
//的类型和所指向的类型是一样的，所以用&b 来给*ptr 赋
//值就是毫无问题的了。
**ptr=34; //*ptr 的结果是ptr 所指向的东西，在这里是一个指针，
//对这个指针再做一次*运算，结果是一个int 类型的变量。
```

四、指针表达式

一个表达式的结果如果是一个指针，那么这个表达式就叫指针表式。

下面是一些指针表达式的例子：

例七：

```
int a,b;
int array[10];
int *pa;
pa=&a; //&a 是一个指针表达式。
Int **ptr=&pa; //&pa 也是一个指针表达式。
*ptr=&b; //*ptr 和&b 都是指针表达式。
pa=array;
pa++; //这也是指针表达式。
```

```
char a[20]=" You_are_a_girl";
char *p=a;
char **ptr=&p;
//printf("p=%d\n",p);
//printf("ptr=%d\n",ptr);
//printf("*ptr=%d\n",*ptr);
printf("**ptr=%c\n",**ptr);
ptr++;
//printf("ptr=%d\n",ptr);
//printf("*ptr=%d\n",*ptr);
printf("**ptr=%c\n",**ptr);
```

例八：

```
char *arr[20];
char **parr=arr; //如果把arr 看作指针的话，arr 也是指针表达式
char *str;
str=*parr; //*parr 是指针表达式
str=*(parr+1); //*(parr+1)是指针表达式
str=*(parr+2); //*(parr+2)是指针表达式
```

由于指针表达式的结果是一个指针，所以指针表达式也具有指针所具有的四个要素：指针的类型，指针所指向的类型，指针指向的内存区，指针自身占据的内存。

好了，当一个指针表达式的结果指针已经明确地具有了指针自身占据的内存的话，这个指针表达式就是一个左值，否则就不是一个左值。在例七中，&a 不是一个左值，因为它还没有占据明确的内存。*ptr 是一个左值，因为*ptr 这个指针已经占据了内存，其实*ptr 就是指针pa，既然*pa 已经在内存中有了自己的位置，那么*ptr 当然也有了自己的位置。

##

五、数组和指针的关系

数组的数组名其实可以看作一个指针。看下例：

例九：

```
int array[10]={0,1,2,3,4,5,6,7,8,9},value;
value=array[0]; //也可写成：value=*array;
value=array[3]; //也可写成：value=*(array+3);
value=array[4]; //也可写成：value=*(array+4);
```

上例中，一般而言数组名array 代表数组本身，类型是int[10]，但如果把array 看做指针的话，它指向数组的第0 个单元，类型是int* 所指向的类型是数组单元的类型即int。因此*array 等于0 就一点也不奇怪了。同理，array+3 是一个指向数组第3 个单元的指针，所以*(array+3)等于3。其它依此类推。

例十：

```
char *str[3]={
    "Hello,thisisasample!",
    "Hi,goodmorning.",
    "Helloworld"
};
char s[80];
strcpy(s,str[0]); //也可写成strcpy(s,*str);
strcpy(s,str[1]); //也可写成strcpy(s,*(str+1));
strcpy(s,str[2]); //也可写成strcpy(s,*(str+2));
```

上例中，str 是一个三单元的数组，该数组的每个单元都是一个指针，这些指针各指向一个字符串。把指针数组名str 当作一个指针的话，它指向数组的第0 号单元，它的类型是char **，它指向的类型是char *。

*str 也是一个指针，它的类型是char *，它所指向的类型是char，它指向的地址是字符串"Hello,thisisasample!"的第一个字符的地址，即'H'的地址。注意:字符串相当于是一个数组,在内存中以数组的形式储存,只不过字符串是一个数组常量,内容不可改变,且只能是右值.如果看成指针的话,他即是常量指针,也是指针常量.

str+1 也是一个指针，它指向数组的第1 号单元，它的类型是char **，它指向的类型是char*。

(str+1)也是一个指针，它的类型是char，它所指向的类型是char，它指向"Hi,goodmorning."的第一个字符'H'

下面总结一下数组的数组名(数组中储存的也是数组)的问题:

声明了一个数组TYPE array[n]，则数组名称array 就有了两重含义：

第一，它代表整个数组，它的类型是TYPE[n]；

第二，它是一个常量指针，该指针的类型是TYPE*，该指针指向的类型是TYPE，也就是数组单元的类型，该指针指向的内存区就是数组第0 号单元，该指针自己占有单独的内存区，注意它和数组第0 号单元占据的内存区是不同的。该指针的值是不能修改的，即类似array++的表达式是错误的。在不同的表达式中数组名array 可以扮演不同的角色。在表达式sizeof(array)中，数组名array 代表数组本身，故这时sizeof 函数测出的是整个数组的大小。

在表达式*array 中，array 扮演的是指针，因此这个表达式的结果就是数组第0 号单元的值。sizeof(*array)测出的是数组单元的大小。

表达式array+n（其中n=0，1，2，.....）中，array 扮演的是指针，故array+n 的结果是一个指针，它的类型是TYPE*，它指向的类型是TYPE，它指向数组第n号单元。故sizeof(array+n)测出的是指针类型的大小。在32 位程序中结果是4

例十一:

```
int array[10];
int (*ptr)[10];
ptr=&array;:
```

上例中ptr 是一个指针，它的类型是int(*)[10]，他指向的类型是int[10] ，我们用整个数组的首地址来初始化它。在语句ptr=&array中，array 代表数组本身。

本节中提到了函数sizeof()，那么我来问一问，sizeof(指针名称)测出的究竟是指针自身类型的大小呢还是指针所指向的类型的大小?

答案是前者。例如：

int(*ptr)[10];

则在32 位程序中，有：

sizeof(int(*)[10])==4

sizeof(int[10])==40

sizeof(ptr)==4

实际上，sizeof(对象)测出的都是对象自身的类型的大小，而不是别的什么类型的大小。