华中科技大学

C 语言程序设计 期末笔试 考前解题笔记

Author：廖兴易

Time：2024.01.01

1、（2019 级电信 判断题）

int (*aPtr)[8], array[6][8];

aPtr=array+1; （√）

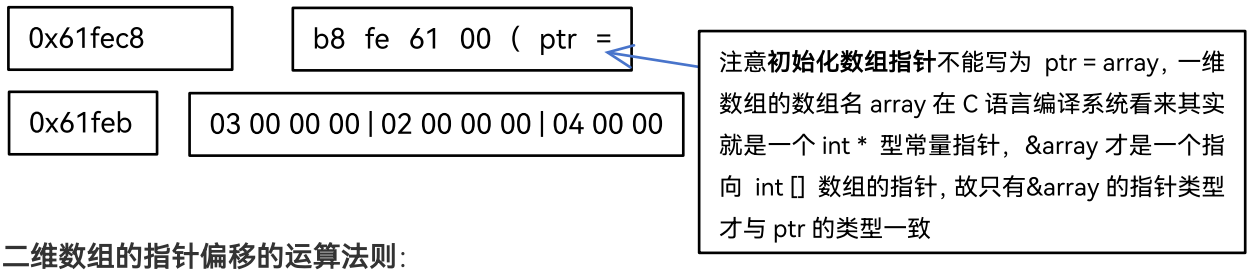
知识要点：C 语言 二维数组的下标运算与指针偏移

数组指针（指向数组的指针）：

定义/声明数组指针 ptr 的语法：datatype (*ptr)[size]，于是 **datatype [size]**就是 **ptr** 所指向的数组的类型和大小。

数组指针 ptr 指针偏移量的单位：所指向的数组的长度。

***ptr 间接访问的对象**是所指向的数组，**通过(*ptr)[i]访问相关数组的元素**，如下图所示。



二维数组的指针偏移的运算法则：

定义一个二维数组 int array[m][n];

首先，我们必须知道一个根本上的东西：在 C 语言编译系统看来，**二维数组 int array[XSIZE][YSIZE] 的地址类型是 int (*)[YSIZE]，二维数组名就是一个数组指针，一个指向第一个 int [YSIZE] 数组的常量指针；一维数组 int array[XSIZE]的地址类型就是 int *，一维数组名就是一个整型变量指针，一个指向数组第一个 int 整型变量的常量指针。**

同一维数组一样，二维数组的数组名也是指向第一个数组元素的（即 array 等于&array[0][0]），且二维数组的数组元素在计算机内的内存存放也是连续的。

但不一样的是：对于二维数组 array[m][n]，指针 array+i 中的 i 的偏移量的单位是 n*sizeof (int) 而不是 sizeof (int) ； 另外，对于二维数组名，虽然 array+ i 的值等于第二行第一个元素的地址，但是使用指针运算符引用 *(array+ i) 得到值依然只是一个指向 int [n] 数组的指针。这都是因为二维数组名具有数组指针的性质。

若要表示 array[i][j]的地址，应该是 *(array+ i)+ j，要复引用 array[i][j]，则应该是*(*(array+ i)+ j)。

2、(2019 级电信 填空题)

设 unsigned char x = 3, y = 5, a = 7, b = 14, c = 6, d = 8; float f =1.0; char result;

(1) f = b/x + d%y ; //求 f 的值 (7.0)

(2) !(x > y) && (++a - b) //求表达式值 (1)

(3) result = (a&b) ^ (c|d) ; //求 result 的值 (-1)

(4) a += b %= a+b ; //求 a 的值 (21)

(5) a++, a - d ; //求运算结果 (0)

(4)解析:

之前已经解决的疑难点——执行顺序到底是怎么在搞？看 C 语言规定的运算优先级、结合性;

现在仍存在的疑惑点——没有小括号,我怎么知道 %= 运算的除数是只有右边的 a 还是右边整个表达式 a+b？看 C 语言规定的各运算的运算对象是“变量名”还是“表达式”;

事实上这两个问题可以同时解决:**根据 C 语言规定的运算优先级、结合性、运算对象来给表达式在正确的位置加上小括号**;

这里我将统统讲清楚。

从左往右看表达式，依次有三个运算:

加后赋值运算 变量+=表达式，优先级 14，右结合;取余后赋值运算 变量%=表达式，优先级 14，右结合;加法运算 表达式+表达式，优先级 4，左结合;

运算对象 b 两边的运算优先级相同，按照右结合规则，b 与 %= 结合，即先执行 b %= a+b，再执行 a += b，就好像无形之中加了一个小括号，变成了 a += (b %= a+b) 一样，**不用担心我这里小括号的比喻可能不恰当，我这里小括号加在哪个位置也是根据优先级、结合性规则判断而来的，加了小括号后反而更有符合人类对优先级的直观理解（符合人类阅读习惯而已）**;

判断完执行顺序，那我们就先执行表达式 b %= a+b 吧，加法运算 a+b 的优先级更高，编译器优先执行 a+b，就像又在无形之中加了一个小括号，变成了 b %= (a+b) 一样，a+b 算出来的结果 21 作为表达式 a+b 的值，然后执行取余后赋值运算 （变量）b %= (a+b)（表达式），b 被赋值为 14;

然后我们执行加后赋值运算 a += (b %= a+b) ，a 的值由原先的 7 变为 21，Q. E. D ...

补充题 1、有以下程序:

```
int func(char *s){
    char *t=s;
    while(*t++);
    t--;
    return(t-s);
}
```

以下关于函数 func 功能叙述正确的是( )

A. 将字符串 s 赋值到 t

B. 比较两串的大小

C. 求字符串 s 的长度

D. 求字符串 s 所占字节数

解析:

之前已经解决的疑难点——**C 语言规定的自增运算与自减运算的前置/后置时表达式的运算法则，和 C 语言规定的运算优先级到底有什么关系，前者的概念到底是什么，两个概念冲突吗?**

while(*t++) 语句的执行逻辑:

解引用运算 *变量名 的优先级和自增运算 ++变量名（或者变量名++）相同,且都为右结合,因此 t 先与右边的自增运算符 ++ 结合，即优先执行 t++操作;

但是这里为自增运算符后置，因此 t++的效果只会在下一语句出现，**在当前语句，仍是在通过寄存器 eax 之前存储的指针 t 原来的值（原来所存储的地址 0x61feac）来参加当前语句的运算。所以++运算的优先级（CPU 是否优先执行++运算）与++运算前置/后置（效果是执行后立即在当前语句/表达式后面部分出现，还是在下一语句才出现），这两个概念的区分算是非常清晰明了了**；

在执行了 t++操作后，执行解引用运算 *t，由于这里自增运算符后置，因此虽然 t++操作已经优先执行，但是效果在下一个语句中才会出现， *t 所解引用的仍然是 t 原来的地址；

将解引用运算 *t 的结果作为当前语句的结果，然后判断：

如果结果不是‘\0’，则循环继续；

如果结果是‘\0’，则跳出循环；

当循环结束时，字符指针 t 指向字符串结尾字符‘\0’的下一位，因此再执行一次 t-- 后，t 才指向字符串的结尾字符‘\0’，从而 t - s 即为字符串的长度（长度计算不含结尾字符‘\0’）。

C语言运算符优先级

优先级	运算符	名称或含义	使用形式	结合方向	说明
1	[]	数组下标	数组名[常量表达式]	左到右	--
	()	圆括号	(表达式) /函数名(形参表)		--
	.	成员选择（对象）	对象.成员名		--
	->	成员选择（指针）	对象指针->成员名		--
2	-	负号运算符	-表达式	右到左	单目运算符
	~	按位取反运算符	~表达式		
	++	自增运算符	++变量名/变量名++		
	--	自减运算符	--变量名/变量名--		
	*	取值运算符	*指针变量		
	&	取地址运算符	&变量名		
	!	逻辑非运算符	!表达式		
	(类型)	强制类型转换	(数据类型)表达式		--
	sizeof	长度运算符	sizeof(表达式)		--
3	/	除	表达式/表达式	左到右	双目运算符
	*	乘	表达式*表达式		
	%	余数（取模）	整型表达式%整型表达式		
4	+	加	表达式+表达式	左到右	双目运算符
	-	减	表达式-表达式		
5	<<	左移	变量<<表达式	左到右	双目运算符
	>>	右移	变量>>表达式		
6	>	大于	表达式>表达式	左到右	双目运算符
	>=	大于等于	表达式>=表达式		
	<	小于	表达式<表达式		
	<=	小于等于	表达式<=表达式		
7	==	等于	表达式==表达式	左到右	双目运算符
	! =	不等于	表达式!= 表达式		
8	&	按位与	表达式&表达式	左到右	双目运算符
9	^	按位异或	表达式^表达式	左到右	双目运算符
10	|	按位或	表达式|表达式	左到右	双目运算符
11	&&	逻辑与	表达式&&表达式	左到右	双目运算符
12	||	逻辑或	表达式||表达式	左到右	双目运算符
13	?:	条件运算符	表达式1? 表达式2: 表达式3	右到左	三目运算符
14	=	赋值运算符	变量=表达式	右到左	--
	/=	除后赋值	变量/=表达式		--
	*=	乘后赋值	变量*=表达式		--
	%=	取模后赋值	变量%=表达式		--
	+=	加后赋值	变量+=表达式		--
	-=	减后赋值	变量-=表达式		--
	<<=	左移后赋值	变量<<=表达式		--
	>>=	右移后赋值	变量>>=表达式		--
	&=	按位与后赋值	变量&=表达式		--
	^=	按位异或后赋值	变量^=表达式		--
	|=	按位或后赋值	变量|=表达式		--
15	,	逗号运算符	表达式,表达式,...	左到右	--

这里把 C 语言规定的优先级、结合性表列出来，以供参考。

我以后绝对不会再在这个地方进行任何讲解了，**现在这个地方仅有的疑问已经全部解决**，唯一可能出现的问题就是忘了某个运算的运算法则、运算优先级、运算结合性，如果还不明白，我劝你尽早放弃，也许你的放弃，可以成就更多的人。

补充例题 2、（2020 级计科 填空题）

请根据下面的声明，计算（6）~（10）题表达式的值并填入各题后面的下划线中。

```
struct st {
    int n;
    struct st *next;
};
Struct st a[3]={5, &a[1], 7, &a[2], 9, 0 }, *p=&a[0];
int b[6] = {2,3,5,6,7,9}, *q = &b[1];
int x=2, y=4;
```

(6) ++p->n 的值为：_____　　(7) (*p++).n 的值为：_____

最后再来示范一次吧，**怕你考试时由疑惑这个 ++ 到底是作用于结构体指针 p 还是作用于 p->n**:

从左到右来看表达式，成员选择运算 对象指针->成员名，优先级 1，左结合；自增运算 ++变量名，优先级 2，右结合，前置；

成员选择运算的优先级更高，优先执行 p->n，**就好像加了小括号一样，像这样 ++ (p->n)**，因 p=&a[0]，故 p->n 的运算结果是 a[0]->n = 5；再执行自增运算 ++ (p->n)（运算对象），也就是执行 ++ 5，运算结果为 6，这就是表达式 ++p->n 的值。

3、（2019 级电信 改错题）

函数 void swap (char *, char *, int)实现两个长度相同字符串的内容交换。

```
void swap (char *pstr1, char *pstr2, int len ){
    char *tmp;
    tmp = pstr1;
    pstr1 = pstr2;
    pstr2 = tmp;
}
```

解析:

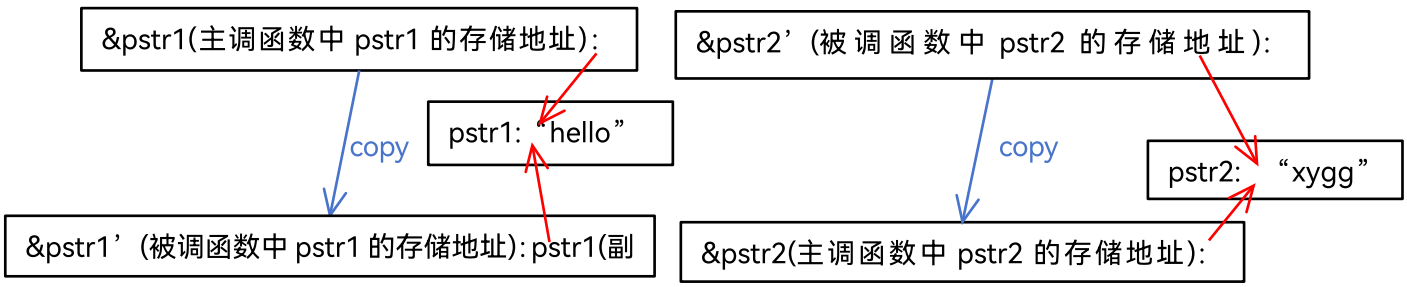
传入被调函数（swap 函数）的指针参数 pstr1，pstr2 仍属于“传值”的范畴，传入的只是拷贝了主调函数中 pstr1，pstr2 的值（两个字符串的地址），只不过是主调函数中 pstr1，pstr2 副本，在被调函数 swap 函数中对 pstr1，pstr2 的值的任何操作，都对主函数中的 pstr1，pstr2 没有任何效应。

一种典型的错误改法是:

```
void swap (char *pstr1,char *pstr2, int len ){
   char *tmp;
   *tmp = *pstr1;
   *pstr1 = *pstr2;
   *pstr2 = *tmp;
}
```

其错误点在下面的知识点中已经指出来了，我们对字符指针变量指向的字符串常量是不被编译器允许的，但是当两字符串长度相同时，由 <string.h> strcpy 函数做的修改被 codeBlocks 编译器允许。

```
void swap (char *pstr1,char *pstr2, int len)
{
   char tmp[len+1];
   strcpy(tmp,pstr1);
   strcpy(pstr1,pstr2);
   strcpy(pstr2,tmp);
   return;
}
```



知识：C 语言 字符数组赋值 字符指针变量赋值

举例如下：

初始化一个**字符数组**：

char a[10] ;

1、**定义时，直接用字符串赋值，大括号可以省略**

char a[10]={"hello"} ;

char a[10]= "hello" ;

注意：不能先定义再给它赋值，如 char a[10]; a[10]= "hello"；这样是错误的!

2、**定义时，对数组中字符逐个赋值**

char a[10]={'h','e','l','l','o'};

3、**如果不是在定义时对字符串赋值，那么只能使用 <string.h> strcpy 函数完成字符串赋值了**

char a[10] ;

strcpy(a,"hello") ;

易错情况：

1、char a[10]; a[10]="hello";

数组名 array 在 C 语言编译系统看来其实就是一个 char *型常量指针，指向的是仅 1 个字符空间，怎么能容纳一个字符串？况且 a[10]也是不存在的!

2、char a[10]; a="hello";

这种错误容易出现，a 虽然是指针，但是它已经指向在栈区（heap，存放局部变量）中分配的 10 个字符空间，现在又要让 a 指向代码段（text segment，存放常量和函数，通常只能读取不能写入）中的 hello 字符串常量，这里的指针 a 出现混乱，不被编译器允许。

还有：**不能使用关系运算符“==”来比较两个字符串，只能用 <string.h> strcmp 函数来处理。**

初始化一个**字符指针变量**：

char *str;

1、定义时，直接用字符串赋值，从而完成初始化，或者用 <string.h> strcpy 函数完成初始化

char *str = "hello";

或者

char *str;

strcpy(str，“hello”);

2、如果不是在定义时赋值，或者已经赋过值，也可以直接用字符串赋值，不过这里使用 <string.h> strcpy 函数完成字符串赋值一般会被编译器判定为非法（codeblocks 例外）

char *str ="hello" ;

str = "xygg";

这里第二行代码就是正确的，因为**可以更改指针变量本身的指向**，但是下面的代码就是**典型的错误**，因为字符串常量存放在代码段（text segment，通常只能读取不能写入）中，**没有权限去修改字符串常量中的数据**：

char *str ="hello" ;

*str = "xygg";

或者

char *str ="hello" ;

str[0] = ‘x’； str[1] = ‘y’ ； str[2] = ‘g’； str[3] = ‘g’；

或者

char *str ="hello" ;

strcpy (str, "hello") ;

给字符指针变量赋值的本质，是在内存的代码段申请一段内存存放一个字符串常量，然后让指针变量获取字符串的首元素地址。