《传智播客 C语言就业班》 第二讲

指针强化

char* p=100; //分配4个字节的内存！（32位系统）

***p在等号左边，是写内存；*p在等号右边，是读内存。**

```
int a=10;
int *p3=&a;
*p3=20;
int c= *p3;
```

【铁律1】指针是一种数据类型；即它指向的内存空间的数据类型；

- **指针也是一种变量，占有内存空间，用来保存内存地址；**
- **指针做函数参数，形参有多级指针的时候，要站在编译器的角度，只需要分配4个总结的内存（32位平台）。当我们使用内存的时候，我们才关心指针所指向的内存，是一维的还是二维的。**
- **指针步长（p++）是根据其所指的内存空间的数据类型来决定的**；p++等价于(unsigned char)p + sizeof(a)
- **注：不断地修改指针的值，相当于不断地改变指针的指向**

【铁律2】间接赋值是指针存在最大的意义。在被调函数里面，通过形参间接修改实参的值；

```
int  getFileLen(int *p)
{
    *p = 40;  //  p的值是a的地址 *a的地址间接修改a的值
}
```

用n级指针形参，去间接修改n-1级指针（实参）的值。

***就像一把钥匙，通过地址找到内存空间，间接修改了如变量a的值。**

☆☆☆☆☆**间接赋值成立的三个条件**：

- 1、定义1个变量（实参） 定义1个变量（形参）；
- 2、建立关联：把实参&取地址传给形参；
- 3、*形参，间接地修改实参的值。

【铁律3】理解指针必须和内存四区概念相结合，应用指针必须和函数调用相结合（指针做函数参数）

主调函数可把堆区、栈区、全局数据内存地址传给被调函数；

被调函数只能返回堆区、全局数据（临时内存会被析构）；

指针做函数参数，是有输入和输出特性的（在这里自己在下面总结一下：）

输入/*in*/：

情况1（一级指针做输入）内存空间是在主调函数提前分配好的（main函数中可以在栈、堆或全局区分配内存，见第一天课程） —— 比如最开始讲的选择排序的程序，在main函数分配好内存空间，如在栈区int a[]={3,1,2,4,5,7,6}; 再调用排序函数void sortArray(int a[], int num);进行选择排序，简单来说就是在两个for循环内执行int temp = a[i];a[i] = a[j];a[j] = temp; 之后再按顺序打印一维数组，相当于把main函数创建的数组又输入到void printArray(int a[], int num)到函数里面进行处理；

情况2（二级指针做输入），一般是把提前在main函数定义好的二维数组传到被调函数里面，基本同上。

输出/*out*/：

情况1（最常见的是二级指针做输出，用二级指针修改一级指针的值）：被调用函数malloc分配内存，再输出供主调函数（如main函数）使用，即在被调函数中修改实参一级指针的值 —— 比如main函数中定义一个指针char *p1 =NULL并调用函数getMem(&p1, ...) 然后被调函数getMem(char **myp1, ...)中通过tmp1=(char *)malloc(100)然后strcpy(tmp1, "112233")再*myp1 = tmp1；

情况2（三级指针做输出，用三级指针修改二级指针的值，基本同上，只不过malloc分配的内存空间里面存的是int*或char*等等而不是int或char了！具体例子：比如按行读取文件，把内容按照第三种内存模型打包数据传出，函数原型为int readFile2(const char *pfilename/*in*/, char ***p/*out*/, int *lineNum/*int out*/)，内部打造内存空间的代码类似于：

```
char **tmp = (char**)malloc(1024*sizeof(char*));
然后for(i=0;i<1024;i++)
    {
        tmp[i] = (char*)malloc(100*sizeof(char));
    }
```

.....最后*p = tmp;

情况3（一级指针做输出，用一级指针修改零级指针的值）：比如下图中int getLen(char *pFileName, int *pFileLen)，在主调函数分配一个用于存储文件长度的int类型的空间，然后在被调函数中获取文件长度，写入这个变量pFileLen对应的内存空间。

王保明老师对于指针做函数参数的总结：

编号	指针函数参数 内存分配方式（级别+堆栈）		主调函数 实参	被调函数 形参	备注
01	1级指针 （做输入）	堆	分配	使用	一般应用禁用
		栈	分配	使用	常用
		Int showbuf(char *p); int showArray(int *array, int iNum)			
02	1级指针 （做输出）	栈	使用	结果传出	常用
		int getLen(char *pFileName, int *pFileLen);			
03	2级指针 （做输入）	堆	分配	使用	一般应用禁用
		栈	分配	使用	常用
		int main(int argc ,char *argv[]); 指针数组 int shouMatrix(int [3][4], int iLine);二维字符串数组			
04	2级指针 （做输出）	堆	使用	分配	常用，但不建议用，转化成02

		int getData(char **data, int *dataLen);
		Int getData_Free(void *data);
		Int getData_Free(void **data); //避免野指针			

05	3级指针 （做输出）	堆	使用	分配	不常用
		int getFileAllLine(char ***content, int *pLine); int getFileAllLine_Free(char ***content, int *pLine);			

指针做函数参数，问题的实质不是指针，而是看内存块，内存块是1维、2维。如果基础类int变量，不需要用指针；

企业里面的接口经常见到的标识：

```
//客户端发报文
int cltSocketSend(void *handle /*in*/, unsigned char *buf /*in*/,  int buflen /*in*/);

//客户端收报文
int cltSocketRev(void *handle /*in*/  unsigned char *buf /*in*/, int *buflen /*in out*/);
```

多级指针做函数参数的理解：

```
//数据类型分为两种，一个是简单的数据类型，一个是复杂的数据类型。碰见复杂的数据类
型不能用简单的数据类型的思维去思考它。抛砖
/*
int getbuf01(char     *p); int getbuf01(char*       p);
int getbuf02(char **p); int getbuf02(char *    *p); getbuf02(char **          p);
int getbuf03(char (*p)[]); int getbuf03(char (*p)        []);   int getbuf03(char (       *p)[       ]);
int getbuf03(char p[10][30]);
int getbuf04(char *****p);
*/
```

- **角度1 站在C++编译器的角度理解，指针就是一个变量，除此之外啥也不是！**

  不管是1个*还是8个*，对C++编译器来讲，只会分配4个字节内存；

- **角度2 当要使用指针所指向内存空间的时候，我们关心这个内存块是一维的还是二维的；一般情况：1级指针代表1维，2级指针代表2维；3级指针一般做输出**；

多维数组做函数参数一般情况只表达到2维。

附录：【王保明老师经典语录】
1）指针也是一种数据类型，指针的数据类型是指它所指向内存空间的数据类型
2）间接赋值*p是指针存在的最大意义
3）理解指针必须和内存四区概念相结合
4）应用指针必须和函数调用相结合（指针做函数参数）
指针是子弹，函数是枪管；子弹只有沿着枪管发射才能显示它的威力；指针的学习重点不言而喻了吧。接口的封装和设计、模块的划分、解决实际应用问题；它是你的工具。
5）指针指向谁就把谁的地址赋给指针
6）指针指向谁就把谁的地址赋给指针，用它对付链表轻松加愉快
7）链表入门的关键是分清楚链表操作和辅助指针变量之间的逻辑关系
8）C/C++语言有它自己的学习特点；若java语言的学习特点是学习、应用、上项目；那么C/C++语言的学习特点是：学习、理解、应用、上项目。多了一个步骤吧。
9）学好指针才学会了C语言的半壁江山，另外半壁江山在哪里呢？你猜，精彩剖析在课堂。
10) 理解指针关键在内存，没有内存哪来的内存首地址，没有内存首地址，哪来的指针啊。

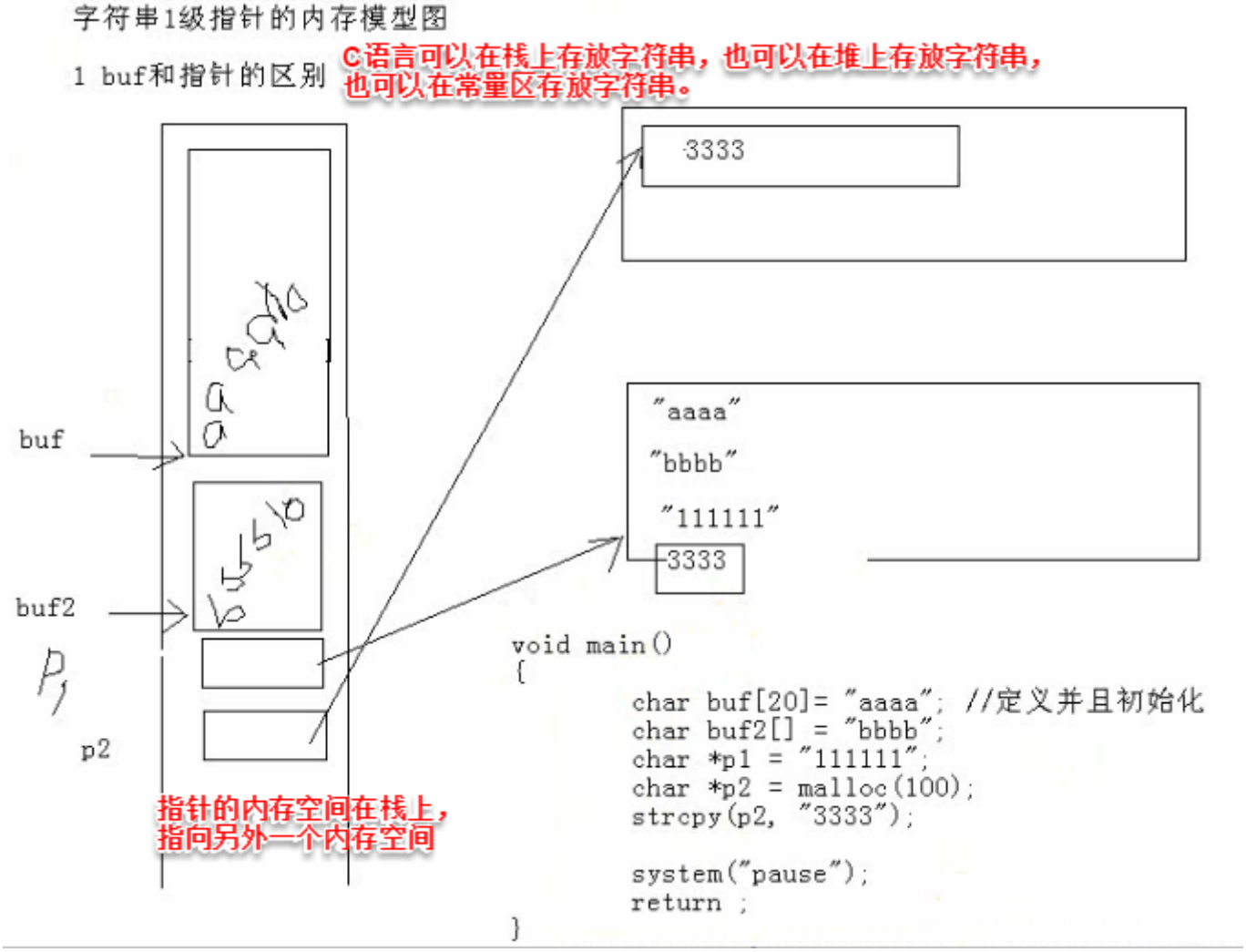
关于字符串：

在C语言中没有字符串类型，通过字符数组来模拟字符串。C风格字符串是以'\0'结尾的字符串；

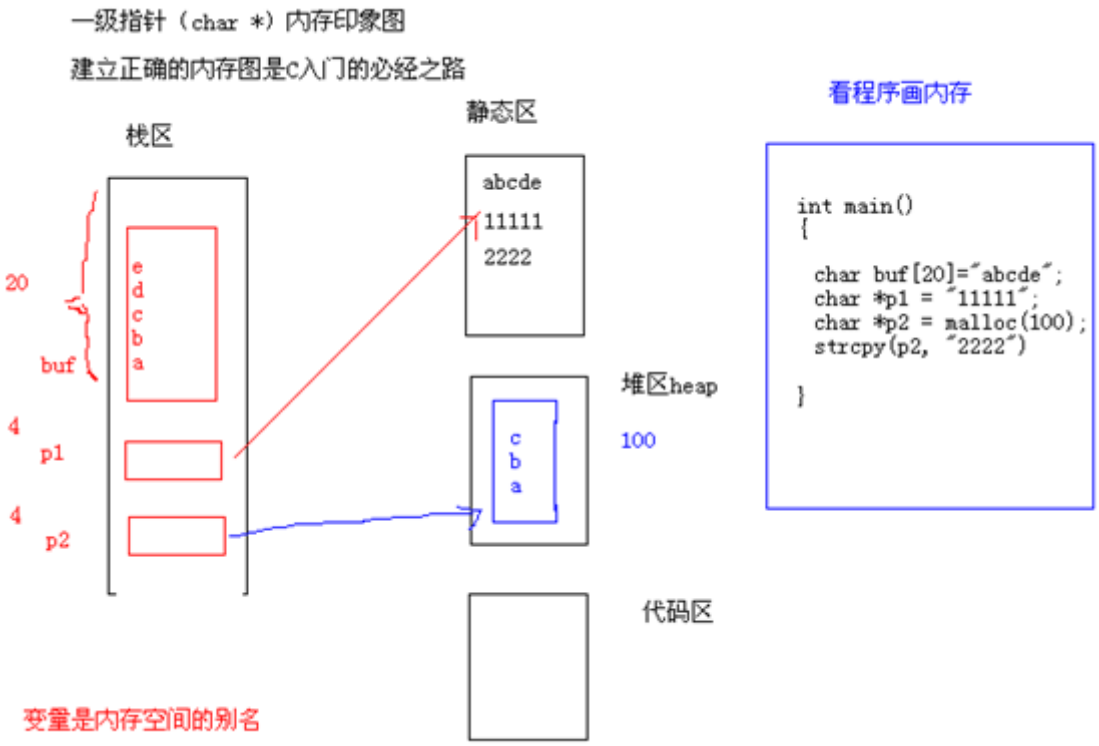
```
#define _CRT_SECURE_NO_WARNINGS
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
void main31()
{
	char buf1[] = { 'a','b','c','d' }; //buf1是一个数组，不是一个以结尾的C风格字符串
	char buf2[100] = { 'a','b','c','d' };
	char buf3[] = "abcd";  //用字符串来初始化字符数组；字符数组结尾包括一个'\0'，所以buf3字符数组的长度应该是4+1=5个字节；
                           //但作为字符串应该是4个字节
	printf("%s\n", buf2);
	printf("%d\n", buf2[0]); //仔细体会
	printf("%c\n", buf2[0]);  //仔细体会
	printf("%d\n", buf2[88]); //仔细体会
	printf("%d\n", sizeof(buf3)); //sizeof是C的关键字，
	//用来求字符数组的长度（求数组所占内存空间的大小）！
	//数组是一种数据类型，本质是固定大小内存块的别名。

	printf("%d\n", strlen(buf3)); //strlen是一个函数，求字符串的长度。
	return;
}
//用指针来操作字符串
void main()
{
	int i = 0;
	char *p = NULL;
	char buf5[] = "abcdefg";
	for (i = 0; i < strlen(buf5); i++)
	{
		printf("%c ", buf5[i]);
	}
	p = buf5;
	for (i = 0; i < strlen(buf5); i++)
	{
		printf("%c ", *(p + i));
	}
}
//注意[ ] 和 *的推导过程： buf5[i]====>buf5[0+i]====>*(buf5+i)
//[]的本质： 和*p一样，只不过负符合程序员的阅读习惯！
//buf5是一个指针，只读的常量；buf5是一个常量指针，析构的时候，要保证buf5所指向的内存空间安全释放。不能赋值buf5=0x11;（错误提示：左操作数必须为左值）
```

char*一级指针内存模型建立

（字符串内存模型）



注意char *p="abcd";内存空间不能被修改（指向的是全局区的"abcd"，即常量字符串），而char buf[1024]="abcd"; 相当于显式地分配了临时区的内存空间， 内存空间可以被修改。

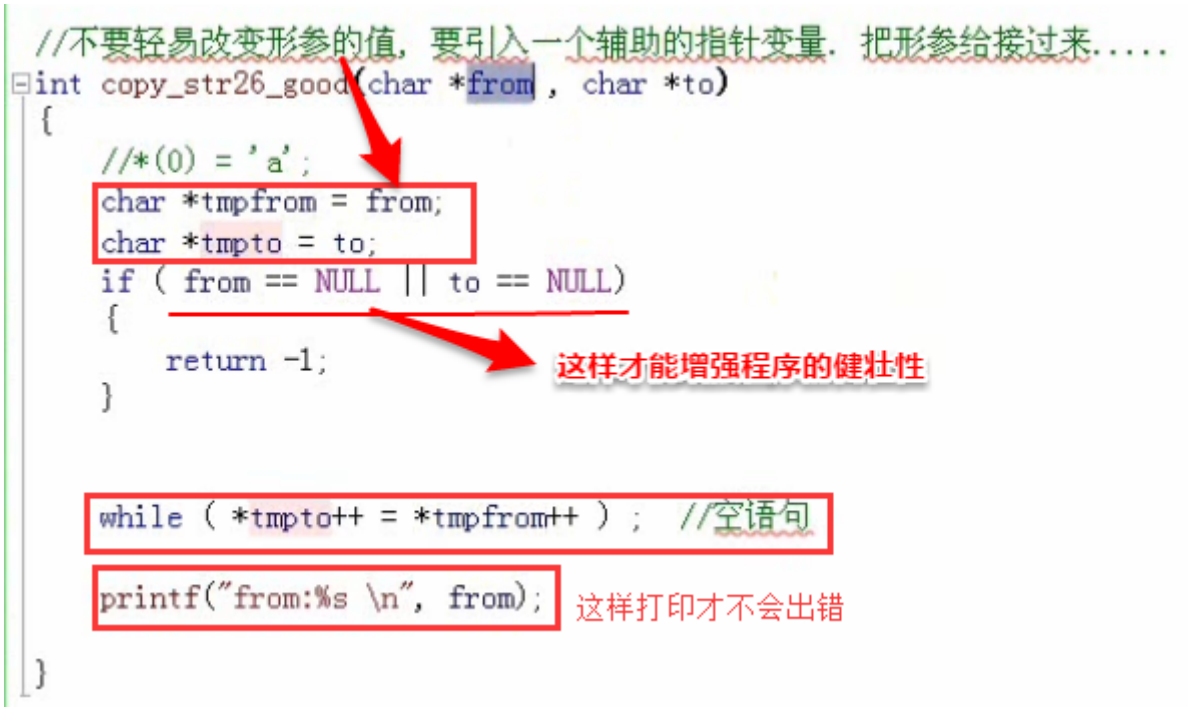


字符串做函数参数

字符串操作常见工程开发模型

业务模型&业务测试模型分离=====>接口封装和设计第一步

```
void copy_str(char *from, char *to)
{
    while (*from != '\0')
    {
        *to++ = *from++;
    }
    *to = '\0';
}

void copy_str2(char *from, char *to)
{
    while ((*to++ = *from++)) //无需写!='\0'，因为*to='\0'之后相当于while(0)跳出循环
    {
        ;
    }
}

void main()
{
    char *from = "abcdefg";
    char buf[100];
    copy_str(from, buf);
    printf("buf:%s \n", buf);
    return;
}
```



☆☆重点学习代码结构☆☆

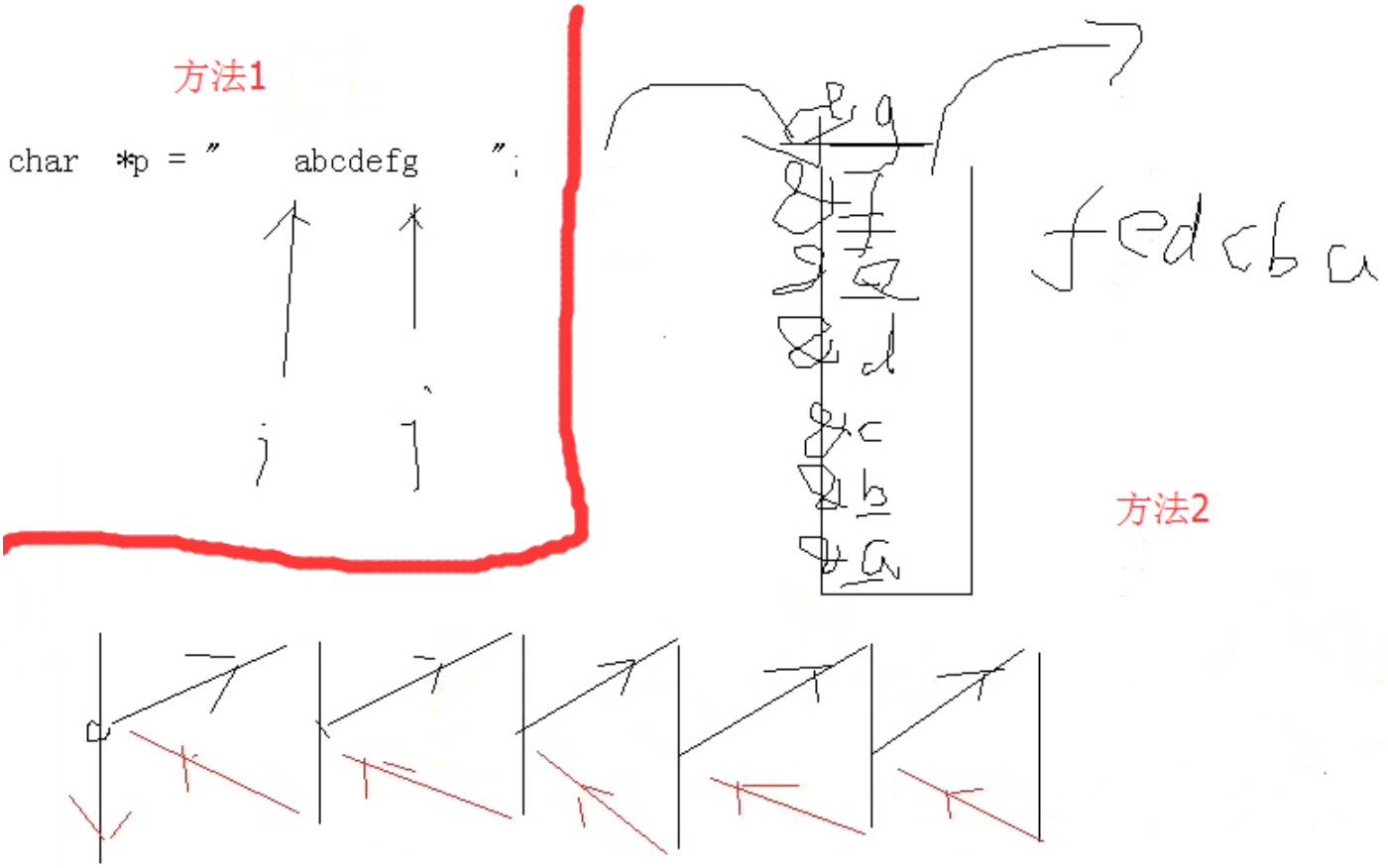
```
//char *p = "11abcd1123abcdkalakdfabcdalksdjfkabcdqqq";
//问题：求字符串p中abcd出现的次数；
//请自定义函数接口，完成上述需求；
//自定义的业务函数和main函数分开；
int getCount(char *mystr /*in*/, char *sub/*in*/, int *ncount)
{
	int ret = 0;
	int tmpCount = 0;
	char *p = mystr;//不要轻易改变形参的值；
	if (mystr == NULL || sub == NULL || ncount == NULL)
	{
		ret = -1;
		printf("func getCount() err:%d  (mystr==NULL || sub==NULL || nount==NULL)\n", ret);
		return ret;
	}
	do {
		p = strstr(p, sub);
		if (p != NULL)
		{
			tmpCount++;
			p = p + strlen(sub); //指针达到下次查找的条件
		}
		else
		{
			break;
		}
	} while (*p != '\0');
	*ncount = tmpCount;
	return ret;
}
int main()
{
	int ret = 0;
	int ncount = 0;
	char *p = "11abcd1123abcdkalakdfabcdalksdjfkabcdqqq";
	char sub[] = "abcd";
	ret = getCount(p, NULL, &ncount); //F9加断点，F11进去
	if (ret != 0)
	{
```

```
        printf("func getCount() err:%d \n", ret);
        return ret;
    }
    printf("ncount:%d\n", ncount);
    system("pause");
}
```

项目开发字符串模型

- **字符串反转模型**

两种方法：1.左图（**指针两头堵**） 2.右图（递归）



理解递归 2个点比较重要： 1）参数的入栈模型 2）函数的嵌套调用返回流程

代码如下：

```
//字符串翻转_方法：
void main41()
{
    char buf[] = "abcdefg";
    int length = strlen(buf);
    char *p1 = buf;
    char *p2 = buf + length - 1;
    while (p1 < p2)
    {
        char c = *p1;
        *p1 = *p2;
        *p2 = c;
        ++p1;
        --p2;
    }
    printf("buf:%s \n", buf);
```

```
    return;
}

//字符串翻转_方法：递归方法，入栈再出栈；
//第一种方法：通过递归的方式，逆序打印，如下。
//第二种方法：递归和全局变量（把逆序的结果存入全局变量） 的方法
//先定义一个全局变量g_buf，然后在inverse函数strncat(g_buf, p, 1);
//但是多线程会存在问题，即全局变量的加锁解锁
//第三种方法：递归和非全局变量（递归指针做函数参数，此时定义一个临时内存mybuf,
//然后和buf同时作为函数参数传入inverse函数； ）
void inverse(char *p)
{
    if (p == NULL)
    {
        return;
    }
    if (*p == '\0')
    {
        return;
    }
    inverse(p + 1); //此时没有执行打印，而是执行了让字符串的每一个地址入栈
    printf("%c", *p);
}

void main()
{
    char buf[] = "abcdefg";
    inverse(buf);
}
```

示例代码：根据key获取value，并去除value两边的空格

```
/*****************************************************************
题目：
键值对（” key = value” ）字符串，在开发中经常使用；
要求：请自己定义一个接口，实现根据key获取value
要求：编写测试用例。
要求：键值对中间可能有n多空格，请去除空格
注意：键值对字符串格式可能如下：
"key1 = value1"
"key2 =      value2    "
"key3  = value3"
"key4       = value4"
"key5   =   value5"
"key6   =value6"
"key7   =   "
*****************************************************************/
#define _CRT_SECURE_NO_WARNINGS
#include "stdio.h"
#include "stdlib.h"
#include "string.h"

int trimSpace(char *str, char *newstr)
{
    char *p = str;
    int ncount = 0;
    int i = 0;
    int j = strlen(p) - 1;
    int m = 0;
    if (str == NULL || newstr == NULL)
    {
        printf("func trimSpace() err \n");
        return -1;
    }
```

```
    while (isspace(p[i]) && p[i] != '\0')
    {
        i++;
    }
    while (isspace(p[j]) && p[j] != '\0')
    {
        j--;
    }
    ncount = j - i + 1;
    strncpy(newstr, str + i, ncount);
    newstr[ncount] = '\0';
    return 0;
}

int getvalueByKey(char *keyvaluebuf, char *keybuf, char *valuebuf)
{
    //首先写思路，1  2  3...  !
    int ret = 0;
    char *p = NULL; //用辅助指针变量接过来
    if (keyvaluebuf == NULL || keybuf == NULL || valuebuf == NULL)
     // 不要写成 if(keyvaluebuf && keybuf && valuebuf){一坨}
     // 这样查错会非常困难，多层缩进很难看，且不利于加日志；
     // 学习本文代码风格，灵性！
    {
        return -1;
    }
    //1 查找key是不是在母串中
    p = keyvaluebuf;  //辅助指针变量初始化
    p = strstr(p, keybuf);
    if (p == NULL)
    {
        return -1;
    }
    //让辅助指针变量重新达到下一次检索的条件
    p = p + strlen(keybuf);
    //2 看有没有等号=
    p = strstr(p, "=");
    if (p == NULL)
    {
        return -1;
    }
    //让辅助指针变量重新达到下一次检索的条件
    p = p + strlen("=");
    //3 在等号后面去除空格，借用之前demo3写好的函数trimSpace(),
     //去掉字符串两边的空格。trim意为修剪、整理。
    ret = trimSpace5(p, valuebuf);
    if (ret != 0) //调用别人的函数之后，需要立即从返回值判断调用是否正确！
    {
        printf("func trimSpace5() err:%d \n", ret);
        return ret;
    }
    return ret;
}
```

理解指针常量和常量指针，const

```
void main()
{
    const int a = 10; //结论C语言中的const修饰的变量 是假的  C语言中的const 是一个冒牌货
    //a = 11;
    {
        int *p = &a;
        *p = 100;
        printf("a:%d \n", a);
    }
```

区分const char *p 和 char const *p的方法：记住“**指针变量和它所指向的内存空间变量，是两个不同的概念**” ；

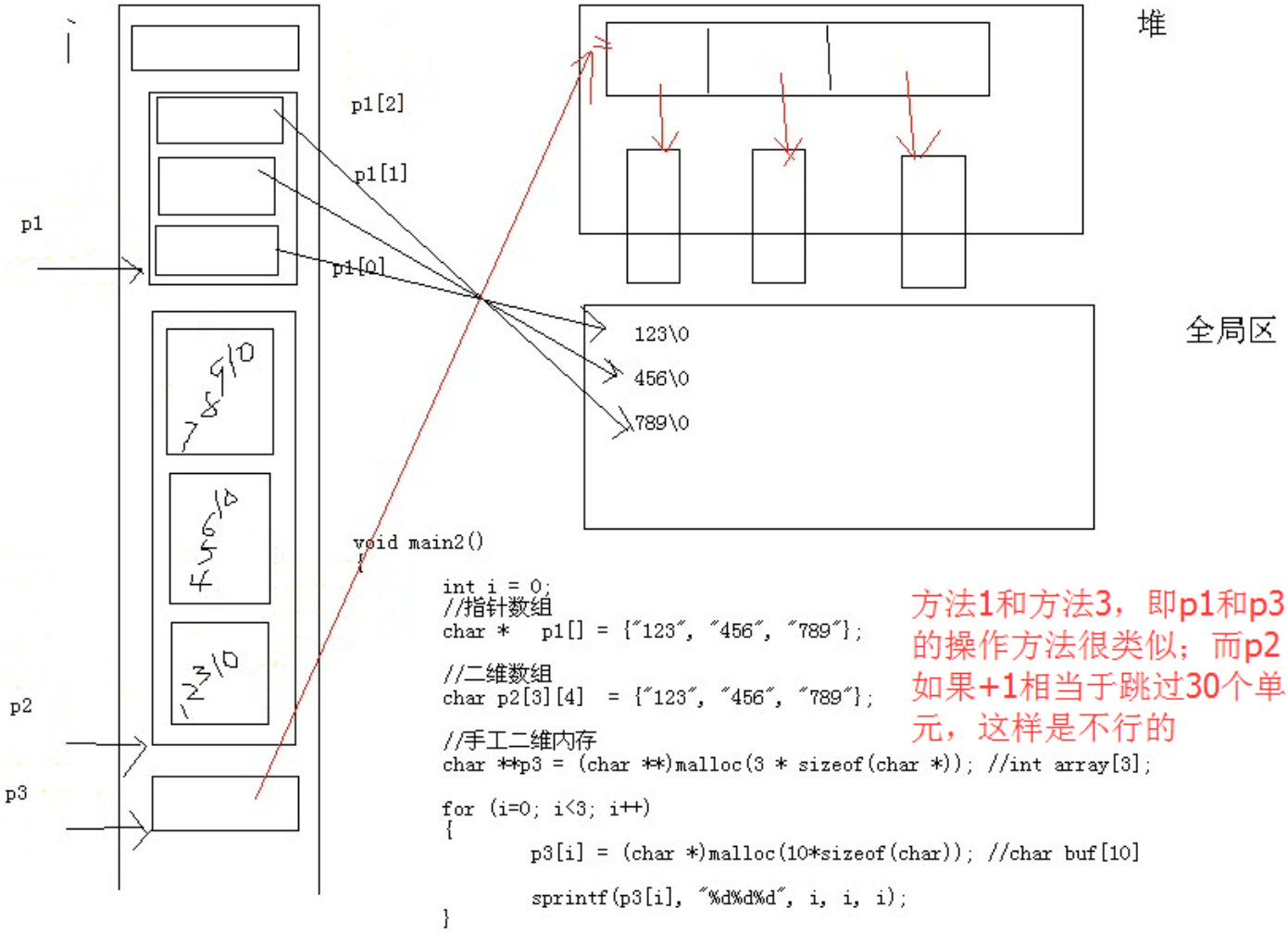
区分名词“指针常量”和“常量指针”的方法，个人总结：中间加个to即可，很明显只能是指针to常量，本身英文也是pointer to const；另一个概念也就很好理解了。

```
void main()
{
    int i = 0;
    int j = 0;
    int num = 4;
    char myBuf[30];
    char tmpBuf[30];
    char myArray[10][30] = { "aaaaa","ccccc","bbbbb","11111" };
    //多维数组名的本质；
    //myArray：编译器只关心有行，每行列; myArray+1会往后跳个单元
    int len1 = sizeof(myArray);
    int len2 = sizeof(myArray[0]);
    int size = len1 / len2;
    printf("len1:%d, len2:%d , size:%d \n", len1, len2, size);
    //打印
    printMyArray(myArray, num);
    //排序
    for (i = 0; i < num; i++)
    {
        for (j = i + 1; j < num; j++)
        {
            if (strcmp(myArray[i], myArray[j]) > 0)
            {
                strcpy(tmpBuf, myArray[i]);
                strcpy(myArray[i], myArray[j]);
                strcpy(myArray[j], tmpBuf);
            }
        }
    }
    //打印
    printf("排序之后：\n");
    for (i = 0; i < num; i++)
    {
        printf("%s\n", myArray[i]);
    }
    return;
}
```

二级指针做输入的三种内存模型

```
int i = 0;

//指针数组
char *   p1[] = {"123", "456", "789"};


//二维数组
char p2[3][4]  = {"123", "456", "789"};

//手工二维内存
char **p3 = (char **)malloc(3 * sizeof(char *)); //int array[3];

for (i=0; i<3; i++)
{
    p3[i] = (char *)malloc(10*sizeof(char)); //char buf[10]

    sprintf(p3[i], "%d%d%d", i, i, i);
}
```

第三种内存模型（手工打造内存模型，通过malloc）：交换指针或交换内容都可以！（第一种由于指向全局区，只能交换指针）—— 训练到极致！背过！



方法1和方法3，即p1和p3的操作方法很类似；而p2如果+1相当于跳过30个单元，这样是不行的

```
char  **getMem(int count)
{
    int i = 0;
    char **tmp = (char **)malloc(count*sizeof(char *));
    for (i=0; i<count; i++)
    {
        tmp[i]  = (char *)malloc(100);
    }
    return tmp;
}
```

1、交换指针：

```
void sortArray(char **myArray, int count)
{
    int i = 0, j = 0;

    char *tmp;
    for (i=0; i<count; i++)
    {
        for (j=i+1; j<count; j++)
        {
            if (strcmp(myArray[i], myArray[j]))
            {
                tmp = myArray[i];
                myArray[i] = myArray[j];
                myArray[j] = tmp;
            }
        }
    }
}
```

2、交换内容：

```
void sortArray02(char **myArray, int count)
{
    int i = 0, j = 0;

    char tmp[200];
    for (i=0; i<count; i++)
    {
        for (j=i+1; j<count; j++)
        {
            if (strcmp(myArray[i], myArray[j]) > 0)
            {
                strcpy(tmp, myArray[i]);
                strcpy(myArray[i], myArray[j]);
                strcpy(myArray[j], tmp);
            }
        }
    }
}
```

第三种内存模型的结束标志：多分配一个字节的内存(count+1)，这样在sortArray()函数中无需再传入count；

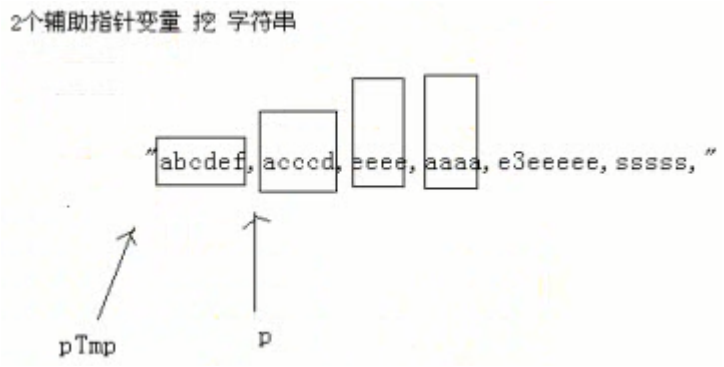
```
char **getMem(int count)
{
    int i = 0;
```

```
    char **tmp = (char **)malloc((count+1) * sizeof(char *));
    for(int = 0; i< count; i++)
    {
       tmp[i] = (char *)malloc(100);
    }
    tmp[count] = '\0';
    return tmp;
}
```

再对sortArray进行相应修改，去掉count参数。（这里指修改sortArray02，01和02一样）

```
void sortArray02(char **myArray)
{
    int i = 0, j = 0;

    char tmp[200];
    for (i=0; myArray[i]!=NULL; i++)
    {
        for (j=i+1;  myArray[j]!=NULL; j++)
        {
            if (strcmp(myArray[i], myArray[j]) > 0)
            {
                strcpy(tmp, myArray[i]);
                strcpy(myArray[i], myArray[j]);
                strcpy(myArray[j], tmp); //交换是buf的内容
            }
        }
    }
}
```

关于作业：两个辅助指针挖字符串



```
char *p=NULL, *pTmp = NULL;
int tmpcount = 0;

//1 p和ptmp初始化
p = buf1;
pTmp = buf1;

do
{
    //2 检索符合条件的位置 p后移  形成差值 挖字符串
    p = strchr(p, c);
    if (p != NULL)
    {
        if (p-pTmp > 0)
        {
            strncpy(buf2[tmpcount], pTmp,  p-pTmp);
            buf2[tmpcount][p-pTmp]  = '\0';  //把第一行数据变成 C风格字符串
            tmpcount ++;
            //3重新 让p和ptmp达到下一次检索的条件
            pTmp = p = p + 1;
        }
    }
    else
    {
        break;
    }
} while (*p!='\0');
```

主函数：

```
char **p = NULL;  //char buf[10][30]   等价于下面的语句
p = (char **)malloc(10 * sizeof(char *)); // char * array[10]
if (p == NULL)
{
    return;
}
for (i=0; i<10; i++)
{
    p[i] = (char *)malloc(30 * sizeof(char));
}

//释放内存
for (i=0; i<10; i++)
{
    free(p[i]);
}
free(p);
```

被调函数：

```
int spitString2(const char *buf1, char c, char **myp /*in*/, int *count)
```

.....

```
do
{
    //2 检索符合条件的位置 p后移  形成差值 控字符串
    p = strchr(p, c);
    if (p != NULL)
    {
        if (p-pTmp > 0)
        {
            strncpy(myp[tmpcount], pTmp,  p-pTmp);
            myp[tmpcount][p-pTmp] = '\0';  //把第一行数据变成 C风格字符串
            tmpcount ++;
            //3重新 让p和ptmp达到下一次检索的条件
            pTmp = p = p + 1;
        }
    }
    else
    {
        break;
    }
} while (*p!='\0');
```

多维数组名的本质

char myArray[10][30]，myArray是一个指针变量，是一个常量指针。

多维数组做函数参数退化问题

```
void f(int a[5]) ====》 void f(int a[]); ===》 void f(int* a);
void g(int a[3][5])====》 void g(int a[][5]); ====》 void g(int (*a)[5]);
技术指演过程  *(*(a+1) +j ) a[j][j]
```

第1种和第3种二级指针做函数参数退化问题

```
Chsr * p[3] = {"aaaa", "bbb","cccc"};
Int printArray(char *p[3])==➔Int printArray(char *p[])==➔Int printArray(char **p)
```

等价关系

数组参数	等效的指针参数
一维数组 char a[30]	指针 char*
指针数组 char *a[30]	指针的指针 char **a
二维数组 char a[10][30]	数组的指针 char(*a)[30]

数组类型、数组指针类型、数组指针类型变量（压死初学者的三座大山）

```
//03、数组类型、数组指针类型、数组指针类型变量
typedef int MyTypeArray[5];
MyTypeArray a; //int a[5];
int intArray[3][5];

{
    typedef int (*MyPTypeArray)[5];
    MyPTypeArray  myArrayPoint ;
    myArrayPoint = &a;
    (*myArrayPoint)[0] = 1; //通过一个数组指针变量去操作数组内存
}
                //  二者等价
{
    int (*myArrayVar)[5]; //告诉编译给我开辟4个字节的内存 ‘
    myArrayVar = &a;
    (*myArrayVar)[1] = 2;
    myArrayVar = intArray
}
```

附网友文章：C语言数组做函数参数退化为指针的技术推演

程序员眼中的二维数组，在物理内存中是线性存储的！！一般32位地址均为4字节，如char *或int *等。

http://www.cnblogs.com/zhanggaofeng/p/5377768.html

```
//数组做函数参数退化为指针的技术推演
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

//一维数组做函数参数退化为指针的技术推演
void printfA(char * strarr[3]);
//计算机中，数组都是线性存储，二维数组元素也是一个个的排列的
//例如： 1,2,3,4,5，6,7,8,9   像这组数据 我们可以认为是一维数组  int a[9]={1,2,3,4,5,6,7,8,9};
//也可以认为是二维数组 int b[3][3]={1,2,3,4,5,6,7,8,9};
//所以计算机并不清楚数组名的步长是多少 就是 a+1移动多少个字节 或者 b+1 移动多少个字节
//这就需要程序员去告诉计算机 数组名的步长是多少
//对于本题 void printfA(char * strarr[3]); 数组做函数参数 该数组是一个一维数组  数组元素类型是 char *
//那么数组strarr的步长应该是 sizeof(char *) 也就是4个字节
//这么来说 我们只需要告诉计算机  你跳4个字节 就OK了
//所以技术推演为printfB

void printfB(char * strarr[]);
//因为计算机根本不关心你有多少个元素 是3个 还是30个 与计算机没关系 是程序员需要关心的（这就是数组越界问题）
//函数参数 char * strarr[] 同样告诉计算机 我是一个一维数组 数组元素是 char * 类型
//那么数组strarr的步长还是 sizeof(char *) 也就是4个字节
//那么我们继续推演  既然计算机只需要确定 该数组每次移动的步长是 4个字节就好
//那么void printfC(char ** strarr);这么写也是可以的 strarr是个指针 strarr里的值指向一个类型是 char *的变量
//步长只与指针的值有关 ，因此strarr的 步长是 sizeof(char *) 也就是4个字节
//所以C语言的开发人员就做了优化 printfC （与我没关系 设计C语言的就是这么优化的）

void printfC(char ** strarr);
//char * strarr[3]做参数退化为char ** strarr) 有2个好处
//好处1：复制一个一维数组char * strarr[3] 比复制一个指针char ** strarr 会耗费更多的内存空间
//char * strarr[3] 需要耗费 sizeof(char *) * 3 =12 个字节的内存空间 ；
//而char ** strarr需要耗费 sizeof(char **) = 4 个字节的内存空间  ；
//节约了内存 和创建 数组时的资源消耗
//好处2：减少了无用解析 ；对于char * strarr[3]  元素个数3 没用，
//这是个一维数组 这个信息没用 ，因为遍历数组的时候从首地址开始遍历， 只要给计算机个首地址就行
//计算机从首地址向后遍历  无需知道他是什么  只需要知道每次的步长是多少就好了
```

```
//二维数组做函数参数退化为指针的技术推演

void printfD(int arr[3][4]);
//对于二维数组，C语言编译器同样需要知道 数组名arr的步长 就是在遍历的时候 每次计算机改移动多少个字节
//那么首先  我们应该确定数组名arr 到底是个什么类型
//数组名 是数组首元素的指针 （ 我自己的推论 ） 那么二维数组 可以想象成一维数组
//只是这个一维数组的每个元素比较特殊，还是一个一维数组
//那么根据推论  数组名arr的类型是一个一维数组的指针   一维数组是这样定义的  typedef int Myarr[4];
//一维数组指针的类型定义是这样的 typedef int (* PMyarr)[4];
//一维数组指针的变量是这么定义的int (* pmyarr)[4];
//所以数组名arr的类型是 int (* PMyarr)[4];  因为指针的步长与指针所指向的内存空间相关
//arr指向的是一个typedef int Myarr[4]类型的数组，这个数组有4个元素，每个元素都是int类型
//由此得出arr这个一维数组指针的步长是  sizeof(int)*4 = 16;
//由 一维数组的推演可知  数组元素的个数对C语言编译器并不重要 二位数组的元素可以看作一维数组
//推演出 void printfD(int arr[][4]);

void printfE(int arr[][4]);
//又因为  C语言编译器  只需要知道  首地址  和步长  所以  可以用 int (*p)[4] 来代替 int arr[][4]

void printfF(int(*arr)[4]);
//综合以上分析，导出结论：数组做函数参数退化为指针，指针的类型就是数组名的类型

void main()
{
    char * strarr[3] = { "123", "456", "789" };
    int arr[3][4] = { 0 };
    system("pause");
}
```