这是道哥的第016篇原创

关注+星标公众号，不错过最新文章

一、前言
二、八个示例
1. 开胃菜：修改主调函数中的数据
2. 在被调用函数中，分配系统资源
2.1 错误用法
2.2 正确用法
3. 传递函数指针
4. 指向结构体的指针
5. 函数指针数组
6. 在结构体中使用柔性数组
7. 通过指针来获取结构体中成员变量的偏移量
8. 通过结构体中成员变量的指针，来获取该结构体的指针
三、总结

一、前言

半个月前写的那篇关于指针最底层原理的文章，得到了很多朋友的认可(链接: C语言指针-从底层原理到花式技巧，用图文和代码帮你讲解透彻)，特别是对刚学习C语言的小伙伴来说，很容易就从根本上理解指针到底是什么、怎么用，这也让我坚信一句话：用心写出的文章，一定会被读者感受到！在写这篇文章的时候，我列了一个提纲，写到后面的时候，发现已经超过一万字了，但是提纲上还有最后一个主题没有写。如果继续写下去，文章体积就太大了，于是就留下了一个尾巴。

今天，我就把这个尾巴给补上去：主要是介绍指针在应用程序的编程中，经常使用的技巧。如果之前的那篇文章勉强算是“道”层面的话，那这篇文章就属于“术”的层面。主要通过 8 个示例程序来展示在 C 语言应用程序中，关于指针使用的常见套路，希望能给你带来收获。

记得我在校园里学习C语言的时候，南师大的黄凤良老师花了大半节课的时间给我们解释指针，现在最清楚地记得老师说过的一句话就是：指针就是地址，地址就是指针！

二、八个示例

1. 开胃菜：修改主调函数中的数据

```
// 交换 2 个 int 型数据
void demo1_swap_data(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

void demo1()
{
    int i = 1;
    int j = 2;
    printf("before: i = %d, j = %d \n", i, j);
    demo1_swap_data(&i, &j);
    printf("after:  i = %d, j = %d \n", i, j);
}
```

这个代码不用解释了，大家一看就明白。如果再过多解释的话，好像在侮辱智商。

2. 在被调用函数中，分配系统资源

代码的目的是：在被调用函数中，从堆区分配 size 个字节的空间，返回给主调函数中的 pData 指针。

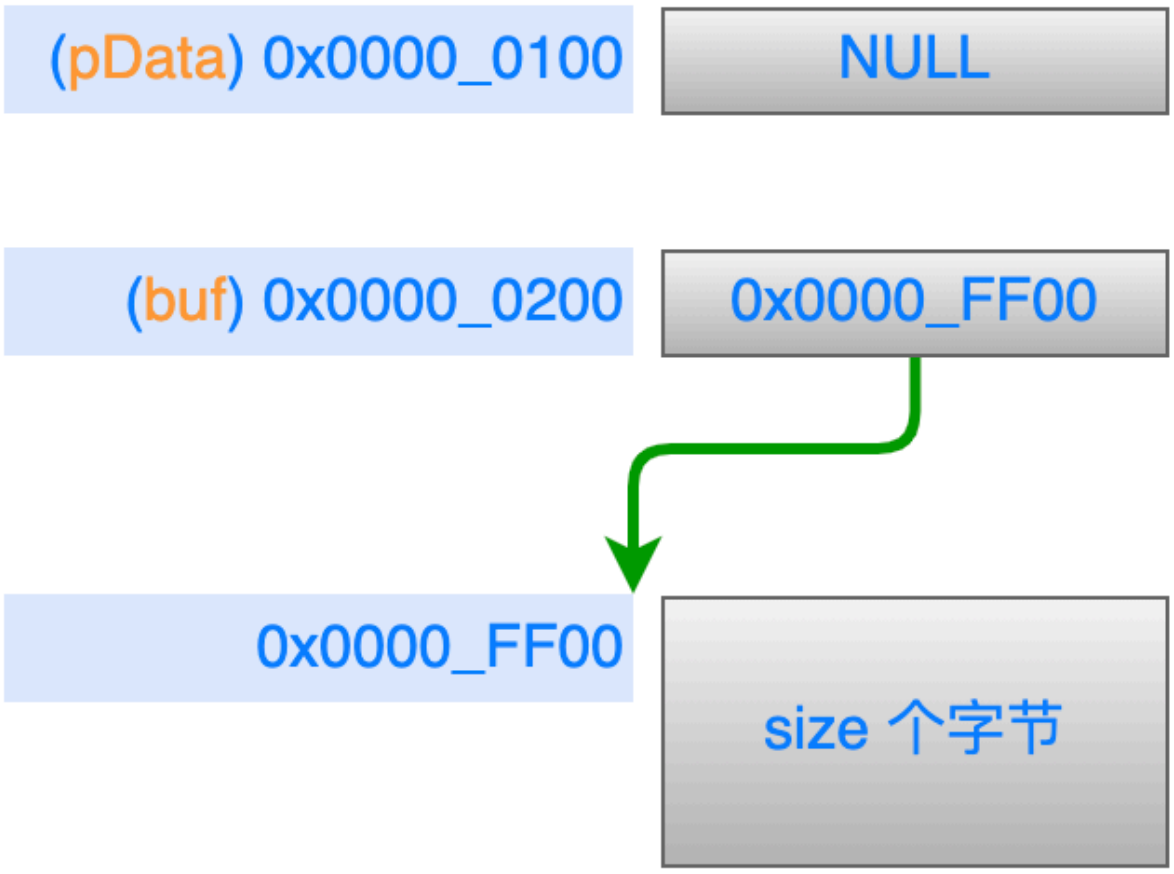
```
void demo2_malloc_heap_error(char *buf, int size)
{
    buf = (char *)malloc(size);
    printf("buf = 0x%x \n", buf);
}

void demo2_malloc_heap_ok(char **buf, int size)
{
    *buf = (char *)malloc(size);
    printf("*buf = 0x%x \n", *buf);
}

void demo2()
{
    int size = 1024;
    char *pData = NULL;

    // 错误用法
    demo2_malloc_heap_error(pData, size);
    printf("&pData = 0x%x, pData = 0x%x \n", &pData, pData);

    // 正确用法
    demo2_malloc_heap_ok(&pData, size);
```

```
    printf("&pData = 0x%x, pData = 0x%x \n", &pData, pData);
    free(pData);
}
```

2.1 错误用法

刚进入被调用函数 demo2_malloc_heap_error 的时候，形参 buff 是一个 char* 型指针，它的值等于 pData 变量的值，也就是说 buff 与 pData 的值相同(都为 NULL)，内存模型如图：

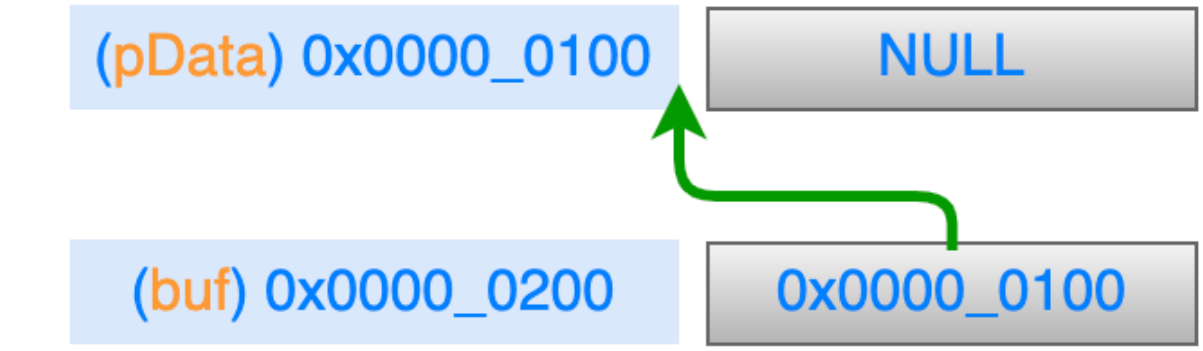
在被调用函数中执行 malloc 语句之后，从堆区申请得到的地址空间赋值给 buf，就是说它就指向了这个新的地址空间，而 pData 里仍然是NULL，内存模型如下：



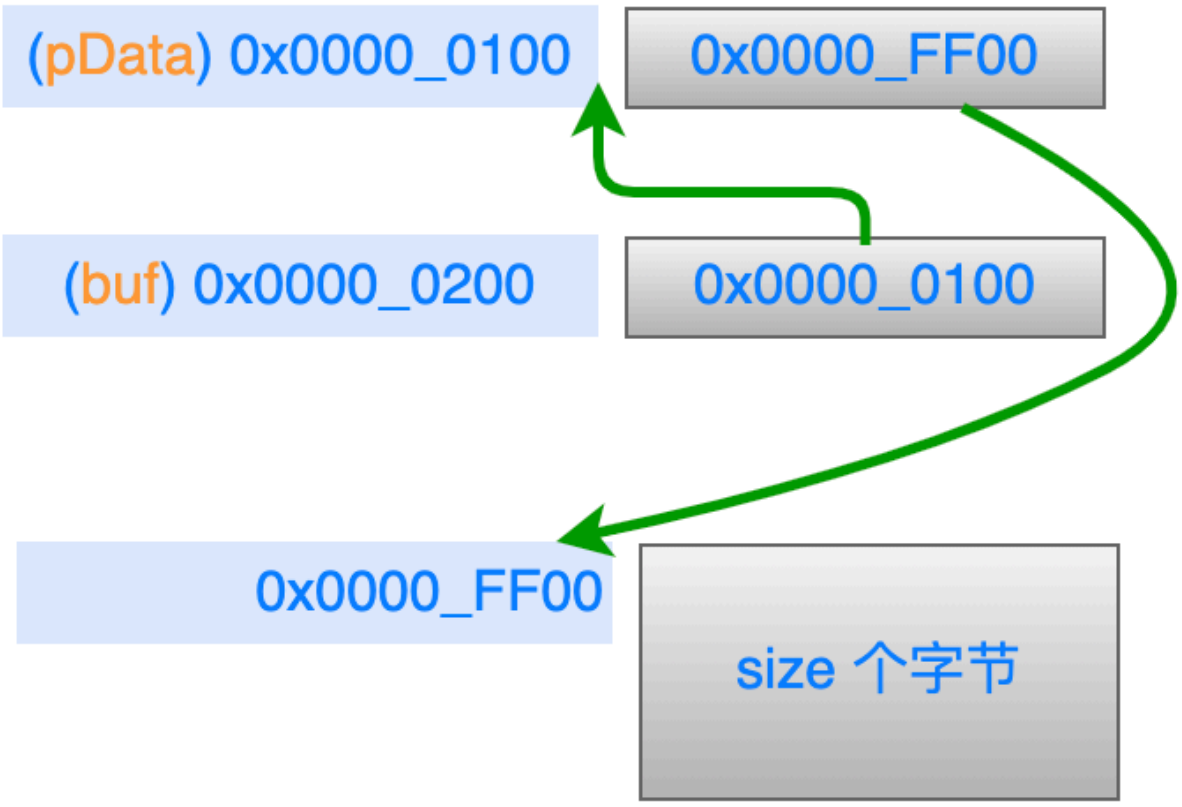
从图中可以看到，pData 的内存中一直是 NULL，没有指向任何堆空间。另外，由于形参 buf 是放在函数的栈区的，从被调函数中返回的时候，堆区这块申请的空间就被泄漏了。

2.2 正确用法

刚进入被调用函数 demo2_malloc_heap_error 的时候，形参 buf 是一个 char* 型的二级指针，就是说 buf 里的值是另一个指针变量的地址，在这个示例中 buf 里的值就是 pData 这个指针变量的地址，内存模型如下：



在被调用函数中执行 malloc 语句之后，从堆区申请得到的地址空间赋值给 *buf，因为 buf = &pData，所以 *buf 就相当于是 pData，那么从堆区申请得到的地址空间就赋值 pData 变量，内存模型如下：



从被调函数中返回之后，pData 就正确的得到了一块堆空间，别忘了使用之后要主动释放。

3. 传递函数指针

从上篇文章中我们知道，函数名本身就代表一个地址，在这个地址中存储着函数体中定义的一连串指令码，只要给这个地址后面加上一个调用符(小括号)，就进入这个函数中执行。在实际程序中，函数名常常作为函数参数来进行传递。

熟悉C++的小伙伴都知道，在标准库中对容器类型的数据进行各种算法操作时，可以传入用户自己的提供的算法函数(如果不传入函数，标准库就使用默认的)。

下面是一个示例代码，对一个 int 行的数组进行排序，排序函数 demo3_handle_data 的最后一个参数是一个函数指针，因此需要传入一个具体的排序算法函数。示例中有 2 个候选函数可以使用：

1. 降序排列: demo3_algorithm_decend；
2. 升序排列: demo3_algorithm_ascend;

```
typedef int BOOL;
#define FALSE 0
#define TRUE  1

BOOL demo3_algorithm_decend(int a, int b)
{
    return a > b;
}

BOOL demo3_algorithm_ascend(int a, int b)
{
    return a < b;
}

typedef BOOL (*Func)(int, int);
void demo3_handle_data(int *data, int size, Func pf)
{
    for (int i = 0; i < size - 1; ++i)
    {
        for (int j = 0; j < size - 1 - i; ++j)
        {
            // 调用传入的排序函数
            if (pf(data[j], data[j+1]))
            {
                int tmp = data[j];
                data[j] = data[j + 1];
                data[j + 1] = tmp;
            }
        }
    }
}

void demo3()
{
    int a[5] = {5, 1, 9, 2, 6};
    int size = sizeof(a)/sizeof(int);
    // 调用排序函数，需要传递排序算法函数
    //demo3_handle_data(a, size, demo3_algorithm_decend); // 降序排列
    demo3_handle_data(a, size, demo3_algorithm_ascend);   // 升序排列
```

```
    for (int i = 0; i < size; ++i)
        printf("%d ", a[i]);
    printf("\n");
}
```

这个就不用画图了，函数指针 pf 就指向了传入的那个函数地址，在排序的时候直接调用就可以了。

4. 指向结构体的指针

在嵌入式开发中，指向结构体的指针使用特别广泛，这里以智能家居中的一条控制指令来举例。在一个智能家居系统中，存在各种各样的设备(插座、电灯、电动窗帘等)，每个设备的控制指令都是不一样的，因此可以在每个设备的控制指令结构体中的最前面，放置所有指令都需要的、通用的成员变量，这些变量可以称为指令头(指令头中包含一个代表命令类型的枚举变量)。

当处理一条控制指令时，先用一个通用命令(指令头)的指针来接收指令，然后根据命令类型枚举变量来区分，把控制指令强制转换成具体的那个设备的数据结构，这样就可以获取到控制指令中特定的控制数据了。

本质上，与 Java/C++ 中的接口、基类的概念类似。

```
// 指令类型枚举
typedef enum _CMD_TYPE_ {
    CMD_TYPE_CONTROL_SWITCH = 1,
    CMD_TYPE_CONTROL_LAMP,
} CMD_TYPE;

// 通用的指令数据结构(指令头)
typedef struct _CmdBase_ {
    CMD_TYPE cmdType; // 指令类型
    int deviceId;     // 设备 Id
} CmdBase;

typedef struct _CmdControlSwitch_ {
    // 前 2 个参数是指令头
    CMD_TYPE cmdType;
    int deviceId;

    // 下面都有这个指令私有的数据
    int slot;  // 排插上的哪个插口
    int state; // 0:断开, 1:接通
} CmdControlSwitch;

typedef struct _CmdControlLamp_ {
    // 前 2 个参数是指令头
    CMD_TYPE cmdType;
    int deviceId;

    // 下面都有这个指令私有的数据
    int color;      // 颜色
    int brightness; // 亮度
} CmdControlLamp;
```

```
// 参数是指令头指针
void demo4_control_device(CmdBase *pcmd)
{
    // 根据指令头中的命令类型，把指令强制转换成具体设备的指令
    if (CMD_TYPE_CONTROL_SWITCH == pcmd->cmdType)
    {
        // 类型强制转换
        CmdControlSwitch *cmd = pcmd;
        printf("control switch. slot = %d, state = %d \n", cmd->slot, cmd-
>state);
    }
    else if (CMD_TYPE_CONTROL_LAMP == pcmd->cmdType)
    {
        // 类型强制转换
        CmdControlLamp * cmd = pcmd;
        printf("control lamp.   color = 0x%x, brightness = %d \n", cmd-
>color, cmd->brightness);
    }
}

void demo4()
{
    // 指令1：控制一个开关
    CmdControlSwitch cmd1 = {CMD_TYPE_CONTROL_SWITCH, 1, 3, 0};
    demo4_control_device(&cmd1);

    // 指令2：控制一个灯泡
    CmdControlLamp cmd2 = {CMD_TYPE_CONTROL_LAMP, 2, 0x112233, 90};
    demo4_control_device(&cmd2);
}
```

5. 函数指针数组

这个示例在上篇文章中演示过，为了完整性，这里再贴一下。

```
int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }
int mul(int a, int b) { return a * b; }
int divide(int a, int b) { return a / b; }

void demo5()
{
    int a = 4, b = 2;
    int (*p[4])(int, int);
    p[0] = add;
    p[1] = sub;
    p[2] = mul;
    p[3] = divide;
    printf("%d + %d = %d \n", a, b, p[0](a, b));
    printf("%d - %d = %d \n", a, b, p[1](a, b));
    printf("%d * %d = %d \n", a, b, p[2](a, b));
```

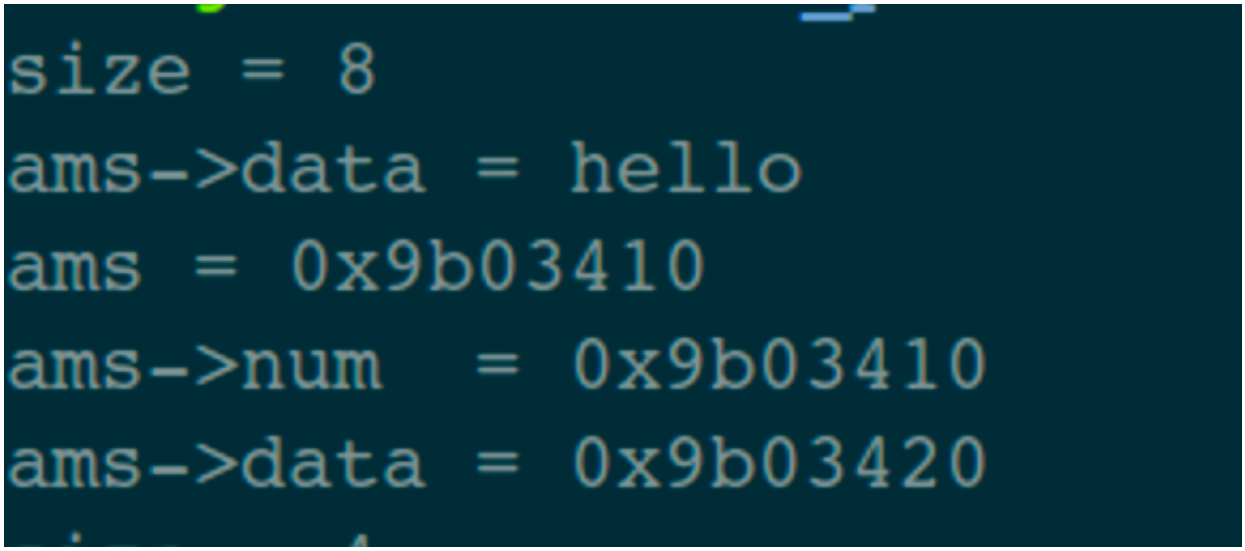
```
    printf("%d / %d = %d \n", a, b, p[3](a, b));
}
```

6. 在结构体中使用柔性数组

先不解释概念，我们先来看一个代码示例：

```
// 一个结构体，成员变量 data 是指针
typedef struct _ArraryMemberStruct_NotGood_ {
    int num;
    char *data;
} ArraryMemberStruct_NotGood;

void demo6_not_good()
{
    // 打印结构体的内存大小
    int size = sizeof(ArraryMemberStruct_NotGood);
    printf("size = %d \n", size);

    // 分配一个结构体指针
    ArraryMemberStruct_NotGood *ams = (ArraryMemberStruct_NotGood
*)malloc(size);
    ams->num = 1;

    // 为结构体中的 data 指针分配空间
    ams->data = (char *)malloc(1024);
    strcpy(ams->data, "hello");
    printf("ams->data = %s \n", ams->data);

    // 打印结构体指针、成员变量的地址
    printf("ams = 0x%x \n", ams);
    printf("ams->num  = 0x%x \n", &ams->num);
    printf("ams->data = 0x%x \n", ams->data);

    // 释放空间
    free(ams->data);
    free(ams);
}
```

在我的电脑上，打印结果如下：



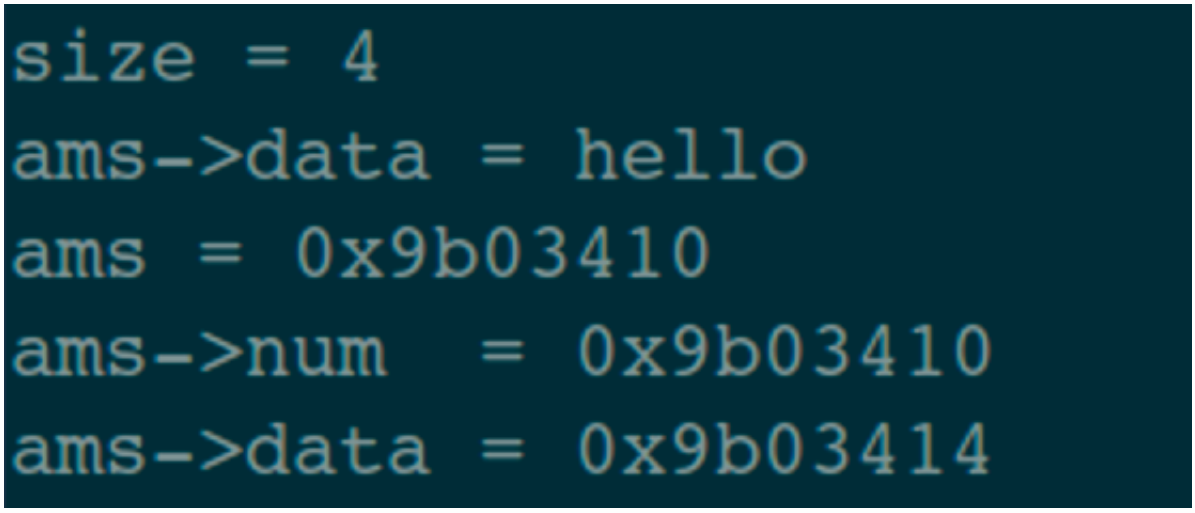
可以看到：该结构体一共有 8 个字节(int 型占 4 个字节，指针型占 4 个字节)。

结构体中的 data 成员是一个指针变量，需要单独为它申请一块空间才可以使用。而且在结构体使用之后，需要先释放 data，然后释放结构体指针 ams，顺序不能错。
这样使用起来，是不是有点麻烦?

于是，C99 标准就定义了一个语法：flexible array member(柔性数组)，直接上代码(下面的代码如果编译时遇到警告，请检查下编译器对这个语法的支持):

```
// 一个结构体，成员变量是未指明大小的数组
typedef struct _ArraryMemberStruct_Good_ {
    int num;
    char data[];
} ArraryMemberStruct_Good;

void demo6_good()
{
    // 打印结构体的大小
    int size = sizeof(ArraryMemberStruct_Good);
    printf("size = %d \n", size);

    // 为结构体指针分配空间
    ArraryMemberStruct_Good *ams = (ArraryMemberStruct_Good *)malloc(size +
1024);

    strcpy(ams->data, "hello");
    printf("ams->data = %s \n", ams->data);

    // 打印结构体指针、成员变量的地址
    printf("ams = 0x%x \n", ams);
    printf("ams->num  = 0x%x \n", &ams->num);
    printf("ams->data = 0x%x \n", ams->data);

    // 释放空间
    free(ams);
}
```

打印结果如下：



与第一个例子中有下面几个不同点：

1. 结构体的大小变成了 4；
2. 为结构体指针分配空间时，除了结构体本身的大小外，还申请了 data 需要的空间大小；
3. 不需要为 data 单独分配空间了；
4. 释放空间时，直接释放结构体指针即可；

是不是用起来简单多了？！这就是柔性数组的好处。

从语法上来说，柔性数组就是指结构体中最后一个元素个数未知的数组，也可以理解为长度为 0，那么就可以让这个结构体称为可变长的。

前面说过，数组名就代表一个地址，是一个不变的地址常量。在结构体中，数组名仅仅是一个符号而已，只代表一个偏移量，不会占用具体的空间。

另外，柔性数组可以是任意类型。这里示例大家多多体会，在很多通讯类的处理场景中，常常见到这种用法。

7. 通过指针来获取结构体中成员变量的偏移量

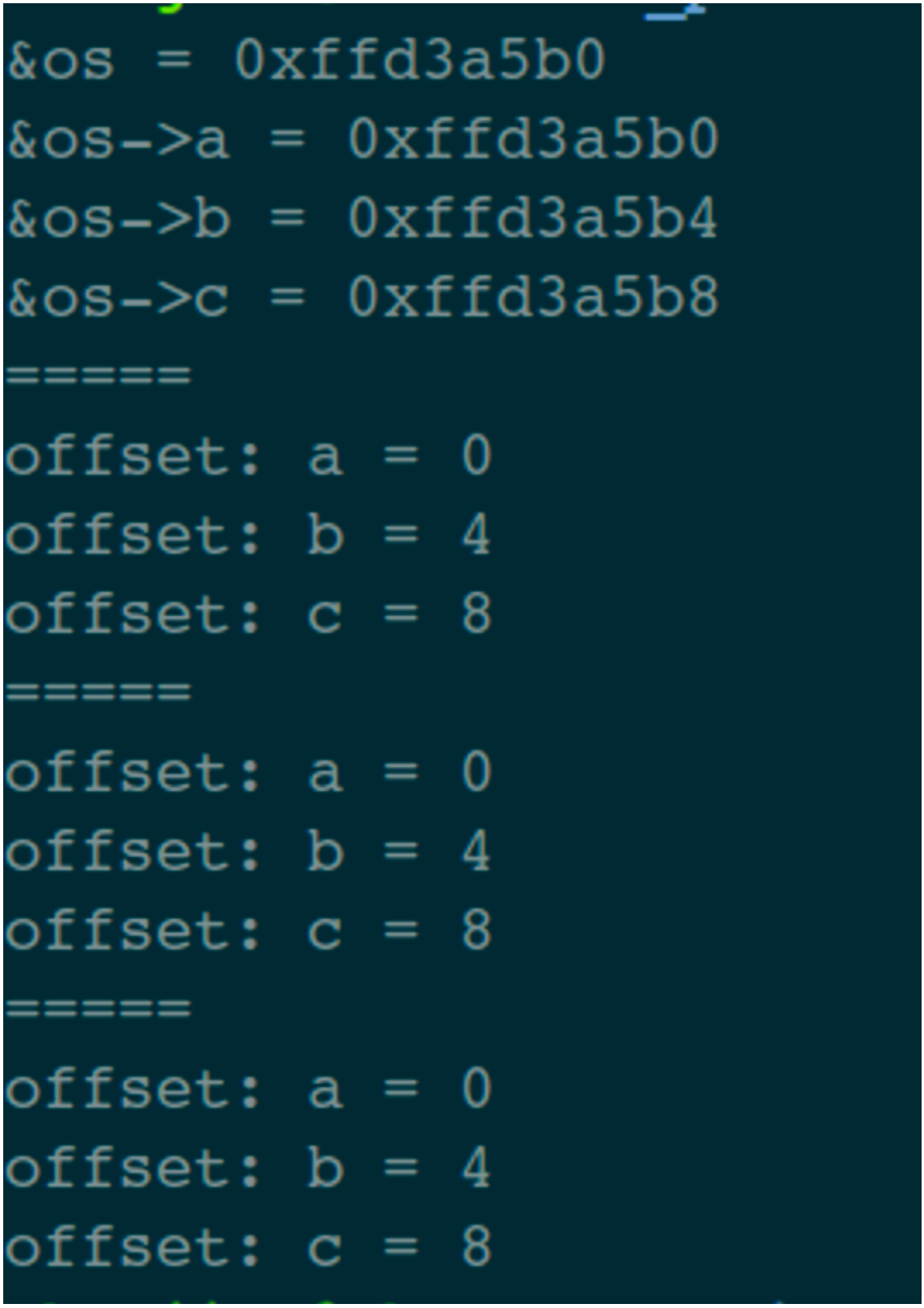
这个标题读起来似乎有点拗口，拆分一下：在一个结构体变量中，可以利用指针操作的技巧，获取某个成员变量的地址、距离结构体变量的开始地址、之间的偏移量。

在 Linux 内核代码中你可以看到很多地方都利用了这个技巧，代码如下：

```
#define offsetof(TYPE, MEMBER) ((size_t) &(((TYPE*)0)->MEMBER))

typedef struct _OffsetStruct_ {
    int a;
    int b;
    int c;
} OffsetStruct;

void demo7()
{
    OffsetStruct os;
    // 打印结构体变量、成员变量的地址
```

```
    printf("&os = 0x%x \n", &os);
    printf("&os->a = 0x%x \n", &os.a);
    printf("&os->b = 0x%x \n", &os.b);
    printf("&os->c = 0x%x \n", &os.c);
    printf("===== \n");
    // 打印成员变量地址，与结构体变量开始地址，之间的偏移量
    printf("offset: a = %d \n", (char *)&os.a - (char *)&os);
    printf("offset: b = %d \n", (char *)&os.b - (char *)&os);
    printf("offset: c = %d \n", (char *)&os.c - (char *)&os);
    printf("===== \n");
    // 通过指针的强制类型转换来获取偏移量
    printf("offset: a = %d \n", (size_t) &((OffsetStruct*)0)->a);
    printf("offset: b = %d \n", (size_t) &((OffsetStruct*)0)->b);
    printf("offset: c = %d \n", (size_t) &((OffsetStruct*)0)->c);
    printf("===== \n");
    // 利用宏定义来得到成员变量的偏移量
    printf("offset: a = %d \n", offsetof(OffsetStruct, a));
    printf("offset: b = %d \n", offsetof(OffsetStruct, b));
    printf("offset: c = %d \n", offsetof(OffsetStruct, c));
}
```

先来看打印结果：



前面 4 行的打印信息不需要解释了，直接看下面这个内存模型即可理解。

下面这个语句也不需要多解释，就是把两个地址的值进行相减，得到距离结构体变量开始地址的偏移量，注意：需要把地址强转成 char* 型之后，才可以相减。

```
printf("offset: a = %d \n", (char *)&os.a - (char *)&os);
```

下面这条语句需要好好理解：

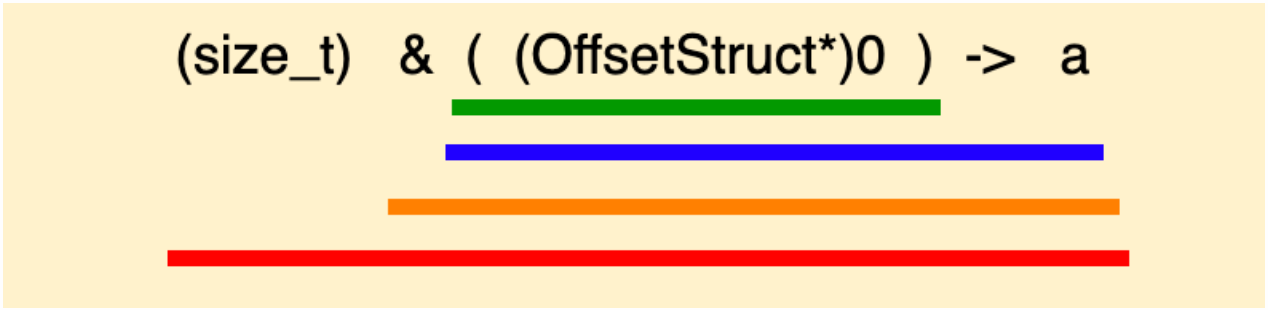
```
printf("offset: a = %d \n", (size_t) &((OffsetStruct*)0)->a);
```

数字 0 看成是一个地址，也就是一个指针。上篇文章解释过，指针就代表内存中的一块空间，至于你把这块空间里的数据看作是什么，这个随便你，你只要告诉编译器，编译器就按照你的意思去操作这些数据。

现在我们把 0 这个地址里的数据看成是一个 OffsetStruct 结构体变量(通过强制转换来告诉编译器)，这样就得到了一个 OffsetStruct 结构体指针(下图中绿色横线)，然后得到该指针变量中的成员变量 a(蓝色横线)，再然后通过取地址符 & 得到 a 的地址(橙色横线)，最后把这个地址强转成 size_t 类型(红色横线)。

因为这个结构体指针变量是从 0 地址开始的，因此，成员变量 a 的地址就是 a 距离结构体变量开始地址的偏移量。

上面的描述过程，如果感觉拗口，请结合下面这张图再读几遍：



上面这张图如果能看懂的话，那么最后一种通过宏定义获取偏移量的打印语句也就明白了，无非就是把代码抽象成宏定义了，方便调用：

```
#define offsetof(TYPE, MEMBER) ((size_t) &(((TYPE*)0)->MEMBER))

printf("offset: a = %d \n", offsetof(OffsetStruct, a));
```

可能有小伙伴提出：获取这个偏移量有什么用啊？那就请接着看下面的示例 8。

8. 通过结构体中成员变量的指针，来获取该结构体的指针

标题同样比较拗口，直接结合代码来看：

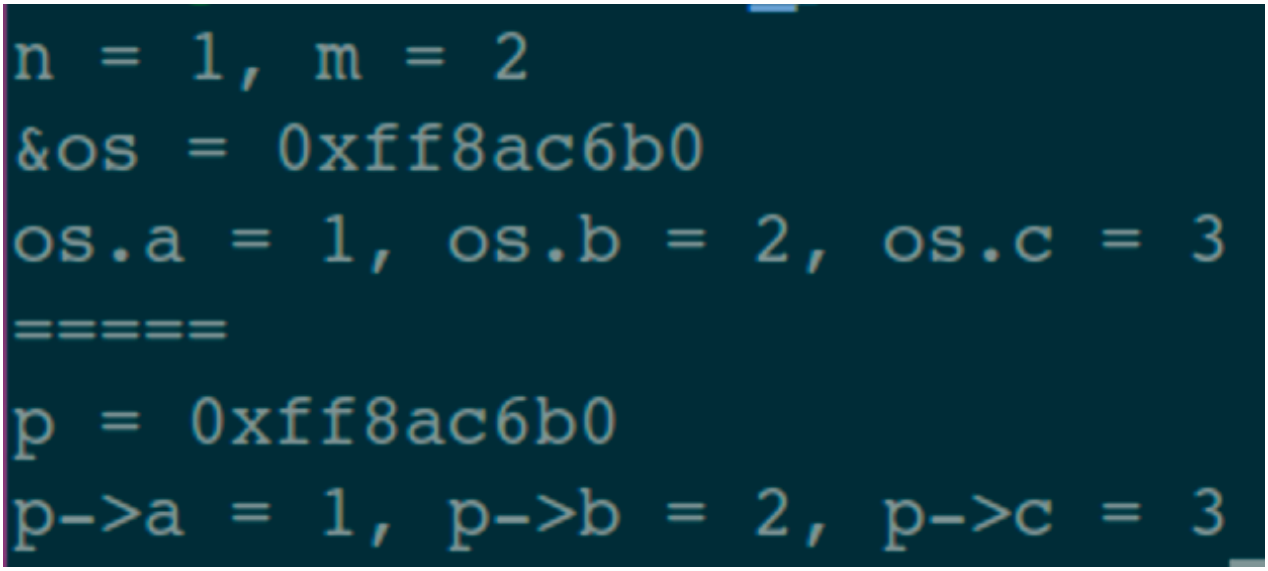
```
typedef struct _OffsetStruct_ {
    int a;
    int b;
    int c;
} OffsetStruct;
```

假设有一个 OffsetStruct 结构体变量 os，我们只知道 os 中成员变量 c 的地址(指针)，那么我们想得到变量 os 的地址(指针)，应该怎么做？这就是标题所描述的目的。

下面代码中的宏定义 container_of 同样是来自于 Linux 内核中的(大家平常没事时多挖掘，可以发现很多好东西)。

```
#define container_of(ptr, type, member) ({ \
     const typeof( ((type *)0)->member ) *__mptr = (ptr); \
     (type *)( (char *)__mptr - offsetof(type,member) );})

void demo8()
{
    // 下面 3 行仅仅是演示 typeof 关键字的用法
    int n = 1;
    typeof(n) m = 2;  // 定义相同类型的变量m
    printf("n = %d, m = %d \n", n, m);

    // 定义结构体变量，并初始化
    OffsetStruct os = {1, 2, 3};

    // 打印结构体变量的地址、成员变量的值(方便后面验证)
    printf("&os = 0x%x \n", &os);
    printf("os.a = %d, os.b = %d, os.c = %d \n", os.a, os.b, os.c);

    printf("===== \n");

    // 假设只知道某个成员变量的地址
    int *pc = &os.c;
    OffsetStruct *p = NULL;

    // 根据成员变量的地址，得到结构体变量的地址
    p = container_of(pc, OffsetStruct, c);

    // 打印指针的地址、成员变量的值
    printf("p = 0x%x \n", p);
    printf("p->a = %d, p->b = %d, p->c = %d \n", p->a, p->b, p->c);
}
```

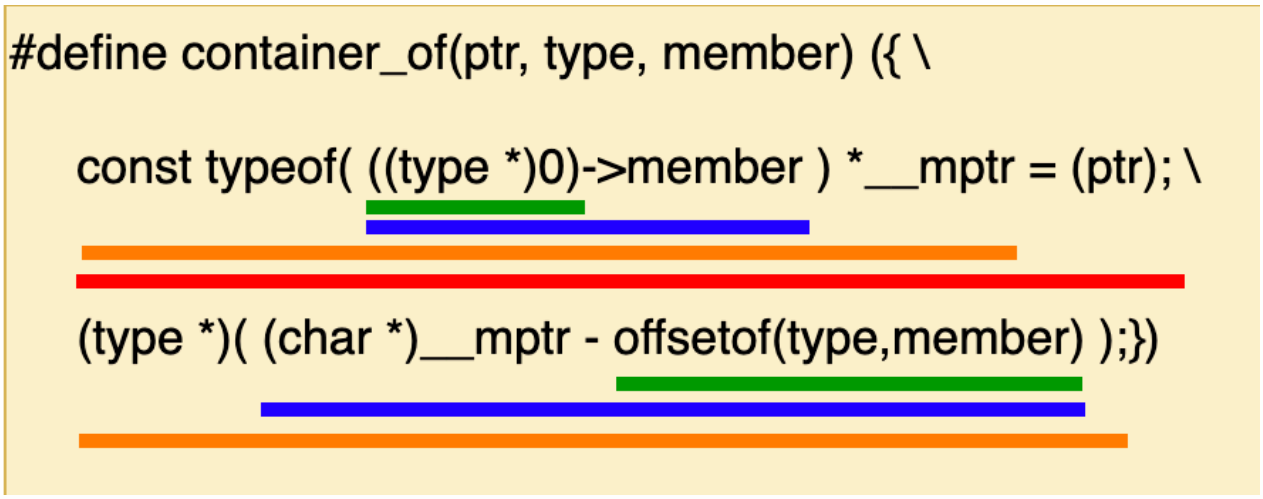
先看打印结果：



首先要清楚宏定义中参数的类型：

1. ptr: 成员变量的指针；
2. type: 结构体类型；
3. member：成员变量的名称；

这里的重点就是理解宏定义 `container_of`，结合下面这张图，把宏定义拆开来进行描述：



宏定义中的第 1 条语句分析：

1. 绿色横线：把数字 0 看成是一个指针，强转成结构体 type 类型；
2. 蓝色横线：获取该结构体指针中的成员变量 member;
3. 橙色横线：利用 typeof 关键字，获取该 member 的类型，然后定义这个类型的一个指针变量 __mptr；
4. 红色横线：把宏参数 ptr 赋值给 __mptr 变量；

宏定义中的第 2 条语句分析：

5. 绿色横线：利用 demo7 中的 offset 宏定义，得到成员变量 member 距离结构体变量开始地址的偏移量，而这个成员变量指针刚才已经知道了，就是 __mptr;
6. 蓝色横线：把 __mptr 这个地址，减去它自己距离结构体变量开始地址的偏移量，就得到了该结构体变量的开始地址；
7. 橙色横线：最后把这个指针(此时是 char* 型)，强转成结构体 type 类型的指针；

三、总结

上面这 8 个关于指针的用法掌握之后，再去处理子字符、数组、链表等数据，基本上就是熟练度和工作量的问题了。
希望大家都能用好指针这个神器，提高程序程序执行效率。

面对代码，永无bug；面对生活，春暖花开！祝您好运！

原创不易，如果这篇文章有帮助，请转发、分享给您的朋友，道哥在此表示感谢！

【原创声明】

> 作者：道哥(公众号: IOT物联网小镇)
> 知乎：道哥
> B站：道哥分享
> 掘金：道哥分享
> CSDN：道哥分享

我会把十多年嵌入式开发中的项目实战经验进行输出总结！

如果觉得文章不错，请转发、分享给您的朋友，您的支持是我持续写作的最大动力！

长按下图二维码关注，每篇文章都有干货。

推荐阅读

[1] C语言指针-从底层原理到花式技巧，用图文和代码帮你讲解透彻
[2] 一步步分析-如何用C实现面向对象编程
[3] 原来gdb的底层调试原理这么简单
[4] 生产者和消费者模式中的双缓冲技术
[5] 关于加密、证书的那些事
[6] 深入LUA脚本语言，让你彻底明白调试原理
[7] 一个printf(结构体变量)引发的血案