

这是道哥的第015篇原创

关注+星标公众号，不错过最新文章



微信搜一搜

🔍 IOT物联网小镇

## 一、前言

1. 为什么写这篇文章
2. 你能得到什么收获
3. 我的测试环境
  - 3.1 操作系统
  - 3.2 编译器

## 二、问题导入

1. 网友求助代码
2. 期望结果
3. 实际打印结果

## 三、分析问题的思路

1. 打印内存模型
2. 分开打印信息
3. 一步步分析问题本质原因
  - 3.1 打印一个最简单的字符串
  - 3.2 打印一个结构体变量
  - 3.3 测试更简单的结构体变量
  - 3.4 继续打印结构体变量

## 四、C语言中的可变参数

1. 利用可变参数的三个函数示例
  - 示例1：参数类型是 int，但是参数个数不固定
  - 示例2：参数类型是 float，但是参数个数不固定
  - 示例3：参数类型是 char\*，但是参数个数不固定
2. 可变参数的原理
  - 2.1 可变参数的几个宏定义
  - 2.2 可变参数的处理过程
3. printf利用可变参数打印信息
  - 3.1 GNU 中的 printf 代码

## 五、总结

### 1. 为什么写这篇文章

在上周六，我在公众号里发了一篇文章：[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](#)，以[直白的语言](#)、[一目了然的图片](#)来解释指针的底层逻辑，有一位小伙伴对文中的代码进行测试，发现一个比较奇怪的问题。我把发来的测试代码进行验证，思考好久也无法解释为什么会出现那么奇怪的打印结果。

为了整理思路，我到阳台抽根烟。晚上的风很大，一根烟我抽了一半，风抽了一半，可能风也有自己的烦恼。后来一想，烟是我买的，为什么让风来抽？于是我就开始抽风！不对，开始回房间继续抽代码，我就不信，这么简单的 printf 语句，怎么就搞不定？！

于是就有了这篇文章。

### 2. 你能得到什么收获

1. 函数参数的传递机制；
2. 可变参数的实现原理(va\_list)；
3. printf 函数的实现机制；
4. 面对问题时的分析思路。

友情提醒：文章的前面大部分内容都是在记录思考问题、解决问题的思路，如果你对这个过程不感兴趣，可以直接跳到[最后面的第四部分](#)，[用图片清晰的解释了可变参数的实现原理](#)，看过一次之后，保管你能深刻记住。

### 3. 我的测试环境

#### 3.1 操作系统

每个人的电脑环境都是不一样的，包括操作系统、编译器、编译器的版本，也许任何一个小差别都会导致一些奇奇怪怪的现象。不过大部分人都是使用 Windows 系统下的 VS 集成开发环境，或者 Linux 下的 gcc 命令行窗口来测试。

我一般都是使用 Ubuntu16.04-64 系统来测试代码，本文中的所有代码都是在这个平台上测试的。如果你用 VS 开发环境中的 VC 编译器，可能在某些细节上与我的测试结果又出入，但是问题也不大，遇到问题再分析，毕竟[解决问题也是提升自己能力的最快途径](#)。

#### 3.2 编译器

我使用的编译器是 Ubuntu16.04-64 系统自带的版本，显示如下：

```
gcc (Ubuntu 5.4.0-6ubuntu1~16.04.12) 5.4.0 20160609
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.
```

另外，我安装的是 64 位系统，为了编译 32 位的可执行程序，我在编译指令中添加了 -m 选项，编译指令如下：

```
gcc -m32 main.c -o main
```

使用 `file main` 命令来查一下编译得到的可执行文件：

```
main: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=0928728acdeac1688c392a3d31a529978f86787e, not stripped
```

所以，在测试时如果输出结果与预期有一些出入，先检查一下编译器。C 语言本质上都是一些标准，每家编译器都是标准的实现者，只要结果满足标准即可，至于实现的过程、代码执行的效率就各显神通了。

## 二、问题导入

### 1. 网友求助代码

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

typedef struct
{
    int age;
    char name[8];
} Student;

int main()
{
    Student s[3] = {{1, "a"}, {2, "b"}, {3, "c"}};
    Student *p = &(s[0]);
    printf("%d, %d \n", *s, *p);
}
```

### 2. 期望结果

根据上篇文章的讨论，我们知道：

1. `s` 是一个包含 3 个元素数组，每个元素的类型是结构体 `Student`;
2. `p` 是一个指针，它指向变量 `s`，也就是说指针 `p` 中保存的是变量 `s` 的地址，因为数组名就表示该数组的首地址。

既然 `s` 也是一个地址，它也代表了这个数组中第一个元素的首地址。第一个元素类型是结构体，结构体中第一个变量是 `int` 型，因此 `s` 所代表的那个位置是一个 `int` 型数据，对应到示例代码中就是数字 1。因此 `printf` 语句中希望直接把这个地址处的数据当做一个 `int` 型数据打印出来，期望的打印结果是：1, 1。

这样的分析过程好像是没有问题的。

## 3. 实际打印结果

我们来编译程序，输出警告信息：

```
rm -rf main *.o
gcc -m32 main.c -o main -I./
main.c: In function 'main':
main.c:215:12: warning: format '%d' expects argument of type 'int',
but argument 2 has type 'Student {aka struct <anonymous>}' [-Wformat=]
    printf("%d, %d \n", *s, *p);
               ^
main.c:215:12: warning: format '%d' expects argument of type 'int',
but argument 3 has type 'Student {aka struct <anonymous>}' [-Wformat=]
    printf("%d, %d \n", *s, *p);
               ^
```

警告信息说：printf 语句需要 int 型数据，但是传递了一个 Student 结构体类型，我们先不用理会这个警告，因为我们就是想通过指针来访问这个地址里的数据。

执行程序，看到实际打印结果是：1，97，很遗憾，与我们的期望不一致！

## 三、分析问题的思路

### 1. 打印内存模型

可以从打印结果看，第一个输出的数字是1，与预期符合；第二个输出 97，很明显是字符 'a' 的 ASCII 码值，但是 p 怎么会指到 name 变量的地址里呢？

首先确认 3 个事情：

1. 结构体 Student 占据的内存大小是多少？
2. 数组 s 里的内存模型是怎样的？
3. s 与 指针变量 p 的值是否正确？

把代码改为如下：

```
Student s[3] = {{1, "a"}, {2, "b"}, {3, "c"}};
Student *p = s;

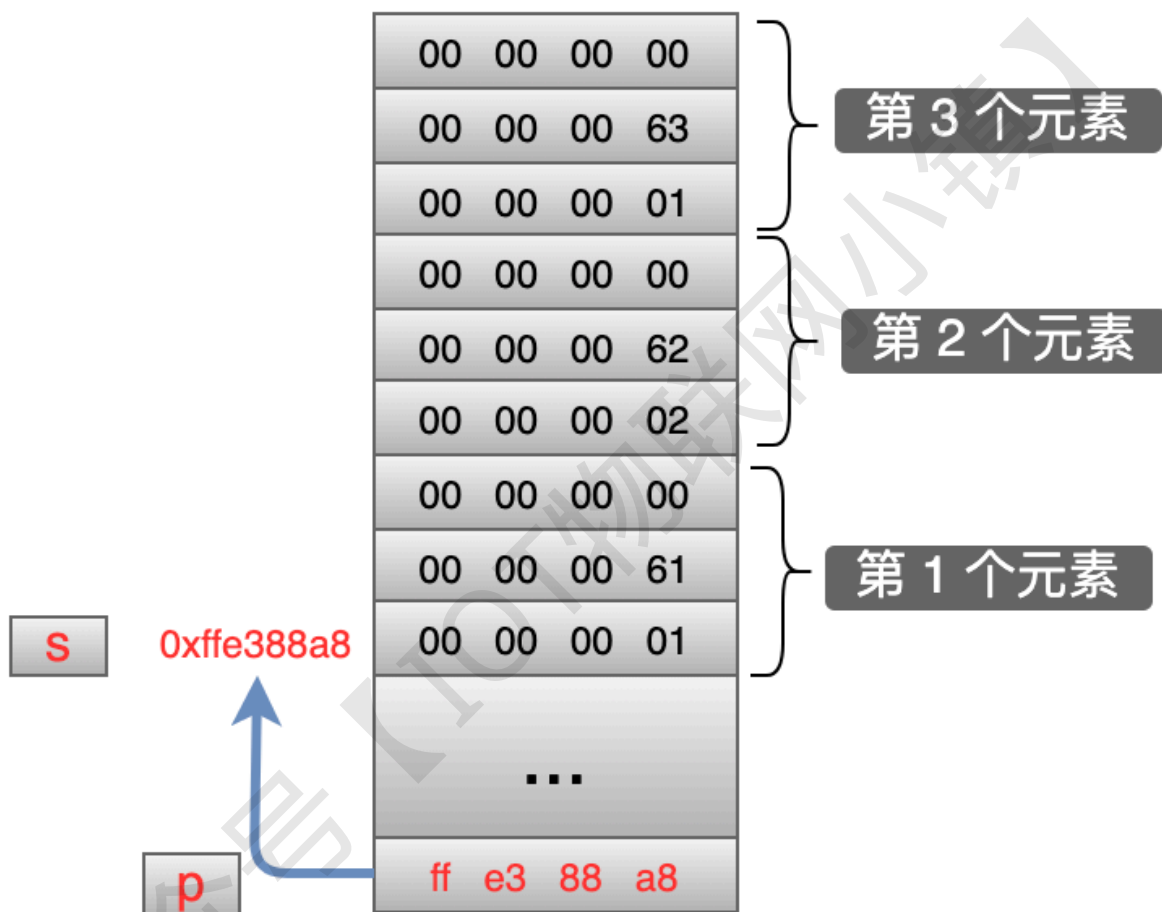
printf("sizeof Student = %d \n\n", sizeof(Student));

printf("print each byte in s: ");
char *pTmp = p;
for (int i = 0; i < 3 * sizeof(Student); i++)
{
    if (0 == i % sizeof(Student))
        printf("\n");
```

# 公众号【IOT物联网小镇】

```
printf("%x ", *(pTmp + i));  
}  
printf("\n\n");  
  
printf("print value of s and p \n\n");  
printf("s = 0x%x, p = 0x%x \n\n", s, p);  
  
printf("%d, %d \n", *s, *p);
```

我们先画一下数组 **s** 预期的内存模型，如下：



编译、测试，打印结果如下：

```
sizeof Student = 12

print each byte in s:
1 0 0 0 61 0 0 0 0 0 0 0
2 0 0 0 62 0 0 0 0 0 0 0
3 0 0 0 63 0 0 0 0 0 0 0

print value of s and p

s = 0xffe388a8, p = 0xffe388a8

1, 97
```

从打印结果看：

1. 结构体 Student 占据 12 个字节，符合预期。
2. 数组 s 的内存模型也是符合预期的，一共占据 36 个字节。
3. s 与 p 都代表一个地址，打印结果它俩相同，也是符合预期的。

那就见鬼了：既然 s 与 p 代表同一个内存地址，但是为什么用 \*p 读取 int 型数据时，得到的却是字符 'a' 的值呢？

## 2. 分开打印信息

既然第一个 \*s 打印结果是正确的，那么就把这个两个数据分开来打印，测试代码如下：

```
Student s[3] = {{1, "a"}, {2, "b"}, {3, "c"}};
Student *p = s;

printf("%d \n", *s);
printf("%d \n", *p);
```

编译、测试，打印结果如下：

```
1
1
```

打印结果符合预期！也就是说分成两条打印语句是可以正确读取到目标地址里的 int 型数据的，但是在一条语句里就不行！

其实此时，可以判断出大概是 printf 语句的原因了。从现象上看，似乎是 printf 语句在执行过程中打印第一个数字之后，影响到了指针 p 的值，但是具体是怎么影响的说不清楚，而且它是系统里的库函数，肯定不能改变 p 的值。

于是在 google 中搜索关键字："glibc printf bug"，你还别说，真的搜索到很多相关资料，但是浏览了一下，没有与我们的测试代码类似的情况，还得继续思考。

## 3. 一步步分析问题本质原因

## 3.1 打印一个最简单的字符串

既然是因为在 printf 语句中打印 2 个数据才出现问题，那么我就把问题简化，用一个最简单的字符串来测试，代码如下：

```
char aa[] = "abcd";  
char *pc = aa;  
printf("%d, %d \n", *pc, *pc);
```

编译、执行，打印结果为："97, 97"，非常正确！这就说明 printf 语句在执行时没有改变指针变量的指向地址。

## 3.2 打印一个结构体变量

既然在字符串上测试没有问题，那么问题就出在结构体类型上了。那就继续用结构体变量来测试，因为上面的测试代码是结构体变量的数组，现在我们把数组的影响去掉，只对单独的一个结构体变量进行测试：

```
Student s = {1, "a"};  
  
printf("%d \n", s);  
printf("%d, %d \n", s, s);
```

注意：这里的 s 是一个变量，不是数组了，所以打印时就不需要用 \* 操作符了。编译、执行，输出结果：

```
1  
1, 97
```

输出结果与之前的错误一样，至此可以得出结论：问题的原因至少与数组是没有关系的！

现在测试的结构体中有 2 个变量：age 和 name，我们继续简化，只保留 int 型数据，这样更容易简化问题。

## 3.3 测试更简单的结构体变量

测试代码如下：



```
typedef struct _A
{
    int a;
    int b;
    int c;
}A;

int main()
{
    A a = {10, 20, 30};
    printf("%d %d %d \n", a, a, a);
}
```

编译、执行，打印结果为：10 20 30，把3个成员变量的值都打印出来了，太诡异了！好像是在内存中，从第一个成员变量开始，自动递增然后获取 int 型数据。

于是我就把后面的两个参数 a 去掉，测试如下代码：

```
A a = {10, 20, 30};
printf("%d %d %d \n", a);
```

编译、执行，打印结果仍然为：10 20 30！这个时候我快疯掉了，主要是时间太晚了，我不太喜欢熬夜。

于是大脑开始偷懒，再次向 google 寻求帮助，还真的找到这个网

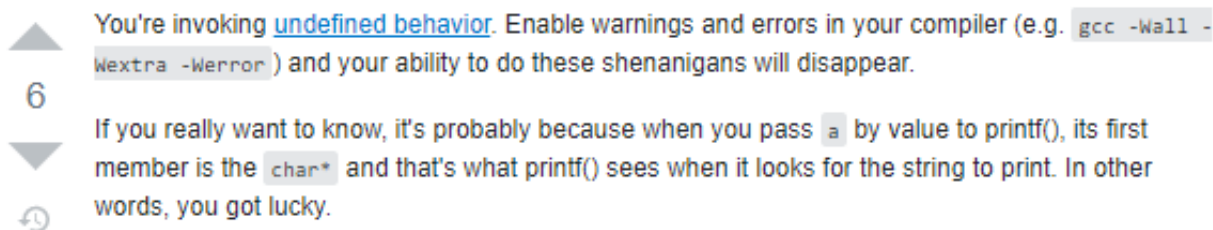
页：<https://stackoverflow.com/questions/26525394/use-printfs-to-print-a-struct-the-structs-first-variable-type-is-char>。感兴趣的小伙伴可以打开浏览一下，其中有下面这两段话

说明了重点：

- ▲ What you're doing is undefined behaviour...
- 1 As for why it works, it's down to the nature of how variable length arguments work in C. Functions like `printf` rely on the format string to tell them what type of argument has been passed. In your first example you've indicated that you're passing a string (`%s`) but you've actually passed a struct. The `printf` function will look for a `char*` pointer on the stack, and because your struct starts with a string it will find one! If you swap the order of `s` and `len` around and then call `printf` with a `%s` you'll either get garbage output or an access violation.

The same rule applies to your second example where you have swapped the order but this time you're trying to output an integer. You've passed the entire structure to the `printf` call which then looks for a `int` on the stack. Because the struct starts with an `int` you got lucky! If you'd not swapped the field order around you'd have seen a number other than 10.





一句话总结：用 `printf` 语句来打印结构体类型的变量，结果是 `undefined behavior`！什么是未定义行为，就是说发生任何状况都是可能的，这个就要看编译器的实现方式了。

看来，我已经找到问题的原因了：原来是因为我的知识不够扎实，不知道打印结构体变量是未定义行为。

补充一点心得：

1. 我们在写程序的时候，因为脑袋中掌握的大部分知识都是正确的，因此编写的代码大部分也都是与预期符合的，不可能故意去写一些稀奇古怪的代码。就比如打印结构体信息，一般正常的思路都是把结构体里面的成员变量，按照对应的数据类型来打印输出。
2. 但是偶尔也会犯低级错误，就像这次遇到的问题一样：直接打印一个结构体变量。因为发生错误了，所以才了解到原来直接打印结构体变量，是一个未定义行为。当然了，这也是一个获取知识的途径。

追查到这里，似乎可以结束了。但是我还是有点不死心，既然是未定义的行为，那么为什么每次打印输出的结果都错的这么一致呢？既然是由编译器的实现决定的，那么我使用的这个 `gcc` 版本内部是怎么来打印结构体变量的呢？

于是我继续往下查...

### 3.4 继续打印结构体变量

刚才的结构体 `A` 中的成员都是 `int` 型，每个 `int` 数据在内存中占据 4 个字节，所以刚才打印出的数据恰好是跨过 4 个字节。如果改成字符串型，打印时是否也会跨过 4 个字节，于是把测试代码改成下面这样：

```
typedef struct _B
{
    int a;
    char b[12];
}B;

int main()
{
    B b = {10, "abcdefghg"};
    printf("%d %c %c \n", b);
}
```

编译、执行，打印结果如下：

果然如此：字符 a 与数字 10 之间跨过 4 个字节，字符 e 与 a 之间也是跨过 4 个字节。那就说明 printf 语句在执行时可能是按照 int 型的数据大小(4个字节)为单位，来跨越内存空间，然后再按照百分号%后面的字符来读取内存地址里的数据。

那就来验证这个想法是否正确，测试代码如下：

```
Student s = {1, "aaa"};
char *pTmp = &s;
for (int i = 0; i < sizeof(Student); i++)
{
    printf("%x ", *(pTmp + i));
}

printf("\n");
printf("%d, %x \n", s);
```

编译、执行，打印结果为：

```
1 0 0 0 61 61 0 0 0 0 0
1, 616161
```

输出结果确实如此：数字 1 之后的内存中存放的是 3 个字符 'a'，第二个打印数据格式是 %x，所以就按照整型数据来读取，于是得到十六进制的 616161。

至此，我们也知道了 gcc 这个版本中，是如何来操作这个“undefined behavior”的。但是事情好像还没有结束，我们都知道：在调用系统中的 printf 语句时，传入的参数个数和类型不是固定的，那么 printf 中是如何来动态侦测参数的个数和类型的呢？

## 四、C语言中的可变参数

在 C 语言中实现可变参数需要用到这下面这几个数据类型和函数(其实是宏定义)：

1. va\_list
2. va\_start
3. va\_arg
4. va\_end

处理动态参数的过程是下面这 4 个步骤：

1. 定义一个变量 va\_list arg;
2. 调用 va\_start 来初始化 arg 变量，传入的第二个参数是可变参数(三个点)前面的那个变量;
3. 使用 va\_arg 函数提取可变参数：循环从 arg 中提取每一个变量，最后一个参数用来指定提取的

# 公众号【IOT物联网小镇】

数据类型。比如：如果格式化字符串是 %d，那么就从可变参数中提取一个 int 型的数据，如果格式化字符串是 %c，就从可变参数中提取一个 char 型数据；

4. 数据处理结束后，使用 va\_end 来释放 arg 变量。

文字表达起来好像有点抽象、复杂，先看一下下面的 3 个示例，然后再回头看一下上面这 4 个步骤，就容易理解了。

## 1. 利用可变参数的三个函数示例


示例1：参数类型是 int，但是参数个数不固定

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

void my_printf_int(int num,...)
{
    int i, val;
    va_list arg;
    va_start(arg, num);
    for(i = 0; i < num; i++)
    {
        val = va_arg(arg, int);
        printf("%d ", val);
    }
    va_end(arg);
    printf("\n");
}

int main()
{
    int a = 1, b = 2, c = 3;
    my_printf_int(3, a, b, c);
}
```

编译、执行，打印结果如下：



1 2 3

示例2：参数类型是 float，但是参数个数不固定

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

void my_printf_float (int n, ...)
{
    int i;
    double val;
```

```
va_list vl;
va_start(vl,n);
for (i = 0; i < n; i++)
{
    val = va_arg(vl, double);
    printf("%.2f ",val);
}
va_end(vl);
printf ("\n");
}

int main()
{
    float f1 = 3.14159, f2 = 2.71828, f3 = 1.41421;
    my_printf_float (3, f1, f2, f3);
}
```

编译、执行，打印结果如下：



3.14 2.72 1.41

示例3：参数类型是 **char\***，但是参数个数不固定

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

void my_printf_string(char *first, ...)
{
    char *str = first;
    va_list arg;
    va_start(arg, first);
    do
    {
        printf("%s ", str);
        str = va_arg(arg, char*);
    } while (str != NULL );
    va_end(arg);
    printf("\n");
}

int main()
{
    char *a = "aaa", *b = "bbb", *c = "ccc";
    my_printf_string(a, b, c, NULL);
}
```

编译、执行，打印结果如下：

注意：以上这3个示例中，虽然传入的**参数个数是不固定的**，但是**参数的类型都必须是一样的**！

另外，处理函数中必须能够知道传入的参数有多少个，处理 int 和 float 的函数是通过**第一个参数**来判断的，处理 char\* 的函数是通过**最后一个可变参数NULL**来判断的。

## 2. 可变参数的原理

### 2.1 可变参数的几个宏定义

```
typedef char *    va_list;

#define va_start   _crt_va_start
#define va_arg     _crt_va_arg
#define va_end     _crt_va_end

#define _crt_va_start(ap,v)  ( ap = (va_list)_ADDRESSOF(v) + _INTSIZEOF(v) )

#define _crt_va_arg(ap,t)    ( *(t *)((ap += _INTSIZEOF(t)) - _INTSIZEOF(t)) )

#define _crt_va_end(ap)      ( ap = (va_list)0 )
```

注意：va\_list 就是一个 char\* 型指针。

### 2.2 可变参数的处理过程

我们以刚才的示例 my\_printf\_int 函数为例，重新贴一下：

```
void my_printf_int(int num, ...) // step1
{
    int i, val;
    va_list arg;
    va_start(arg, num);          // step2
    for(i = 0; i < num; i++)
    {
        val = va_arg(arg, int); // step3
        printf("%d ", val);
    }
    va_end(arg);                 // step4
    printf("\n");
}

int main()
{
    int a = 1, b = 2, c = 3;
    my_printf_int(3, a, b, c);
}
```

# 公众号【IOT物联网小镇】

## Step1: 函数调用时

C语言中函数调用时，参数是**从右到左、逐个压入到栈中**的，因此在进入 my\_printf\_int 的函数体中时，栈中的布局如下：



## Step2: 执行 va\_start

```
va_start(arg, num);
```

把上面这语句，带入下面这宏定义：

```
#define _crt_va_start(ap,v) ( ap = (va_list)_ADDRESSOF(v) + _INTSIZEOF(v) )
```

宏扩展之后得到：

```
arg = (char *)num + sizeof(num);
```

# 公众号【IOT物联网小镇】

结合下面的图来分析一下：首先通过 `_ADDRESSOF` 得到 `num` 的地址 `0x01020300`，然后强转成 `char*` 类型，再然后加上 `num` 占据的字节数(4个字节)，得到地址 `0x01020304`，最后把这个地址赋值给 `arg`，因此 `arg` 这个指针就指向了栈中数字 1 的那个地址，也就是第一个参数，如下图所示：



## Step3: 执行 `va_arg`

```
val = va_arg(arg, int);
```

把上面这语句，带入下面这宏定义：

```
#define _crt_va_arg(ap,t)    ( *(t *)((ap += _INTSIZEOF(t)) - _INTSIZEOF(t)) )
```

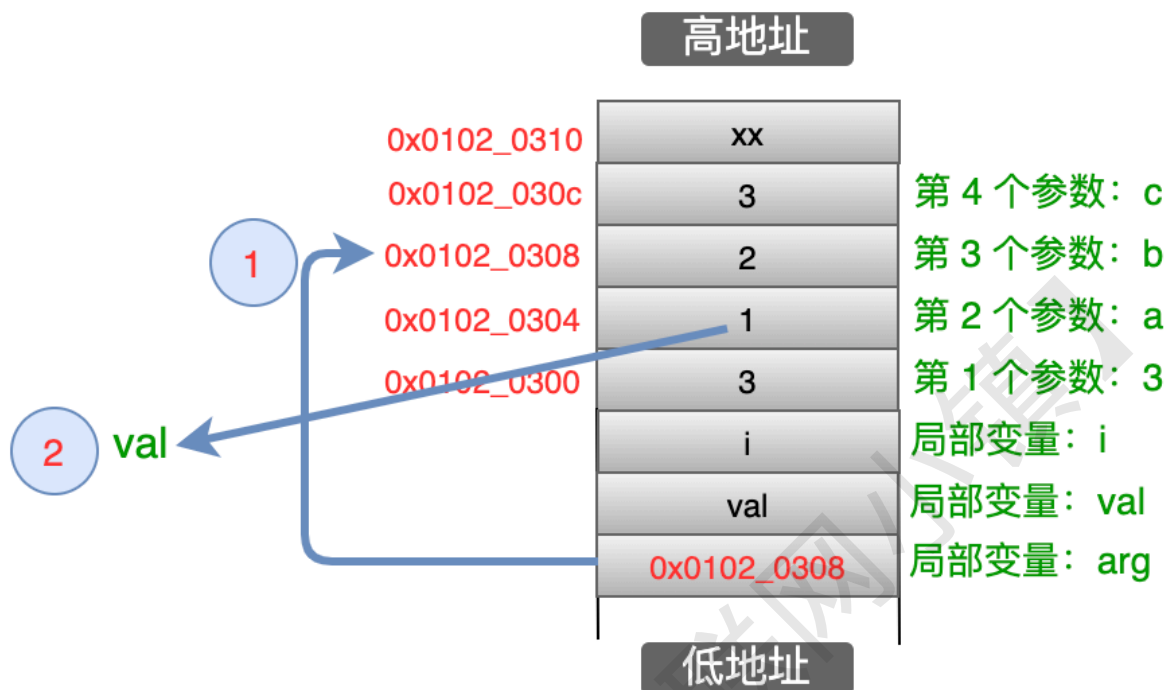
宏扩展之后得到：

```
val = ( *(int *)((arg += _INTSIZEOF(int)) - _INTSIZEOF(int)) )
```



# 公众号【IOT物联网小镇】

结合下面的图来分析一下：先把 **arg** 自增 **int** 型数据的大小(4个字节)，使得 **arg** = 0x01020308；然后再把这个地址(0x01020308)减去4个字节，得到的地址(0x01020304)里的这个值，强转成 **int** 型，赋值给 **val**，如下图所示：



简单理解，其实也就是：得到当前 **arg** 指向的 **int** 数据，然后把 **arg** 指向位于高地址处的下一个参数位置。

**va\_arg** 可以反复调用，直到获取栈中所有的函数传入的参数。

## Step4: 执行 **va\_end**

```
va_end(arg);
```

把上面这语句，带入下面这宏定义：

```
#define _crt_va_end(ap)    ( ap = (va_list)0 )
```

宏扩展之后得到：

```
arg = (char *)0;
```

这就好理解了，直接**把指针 arg 设置为空**。因为栈中的所有动态参数被提取后，**arg** 的值为 0x01020310(最后一个参数的上一个地址)，如果不设置为 **NULL** 的话，下面使用的话就得到未知的结果，为了防止误操作，需要设置为**NULL**。

## 3. printf利用可变参数打印信息

理解了 C 语言中可变参数的处理机制，再来思考 **printf 语句的实现机制** 就很好理解了。

## 3.1 GNU 中的 printf 代码

```
__printf (const char *format, ...)
{
    va_list arg;
    int done;

    va_start (arg, format);
    done = vfprintf (stdout, format, arg);
    va_end (arg);

    return done;
}
```

可见，系统库中的 printf 也是这样来处理动态参数的，vfprintf 函数最终会调用系统函数 sys\_write，把数据输出到 stdout 设备上(显示器)。

vfprintf 函数代码看起来还是有点复杂，不过稍微分析一下就可以得到其中的大概实现思路：

1. 逐个比对格式化字符串中的每一个字符；
2. 如果是普通字符就直接输出；
3. 如果是格式化字符，就根据指定的数据类型，从可变参数中读取数据，输出显示；

以上只是很**粗略**的思路，实现细节肯定复杂的多，需要考虑各种细节问题。下面是 2 个简单的示例：

```
void my_printf_format_v1(char *fmt, ...)
{
    va_list arg;
    int d;
    char c, *s;

    va_start(arg, fmt);
    while (*fmt)
    {
        switch (*fmt) {
            case 's':
                s = va_arg(arg, char *);
                printf("%s", s);

                break;

            case 'd':
                d = va_arg(arg, int);
                printf("%d", d);
                break;

            case 'c':
                c = (char) va_arg(arg, int);
                printf(" %c", c);
                break;
        }
        ++fmt;
    }
}
```

# 公众号【IOT物联网小镇】

```
default:
    if ('%' != *fmt || ('s' != *(fmt + 1) && 'd' != *(fmt + 1) &&
        'c' != *(fmt + 1)))
        printf("%c", *fmt);
        break;
    }
    fmt++;
}
va_end(arg);
}

int main()
{
    my_printf_format_v1("age = %d, name = %s, num = %d \n",
        20, "zhangsan", 98);
}
```

编译、执行，输出结果：

```
age = 20, name = zhangsan, num = 98
```

完美！但是再测试下面代码(把格式化字符串最后面的 num 改成 score):

```
my_printf_format_v1("age = %d, name = %s, score = %d \n",
    20, "zhangsan", 98);
```

编译、执行，输出结果：

```
Segmentation fault (core dumped)
```

因为普通字符串 score 中的字符 s 被第一个 case 捕获到了，所以发生错误。稍微改进一下：

```
void my_printf_format_v2(char *fmt, ...)
{
    va_list arg;
    int d;
    char c, lastC = '\\0', *s;

    va_start(arg, fmt);
    while (*fmt)
    {
        switch (*fmt) {
            case 's':
                if ('%' == lastC)
                {
                    s = va_arg(arg, char *);
                    printf("%s", s);
                }
                else
                {

```

# 公众号【IOT物联网小镇】

```
        printf("%c", *fmt);
    }
    break;

    case 'd':
        if ('%' == lastC)
        {
            d = va_arg(arg, int);
            printf("%d", d);
        }
        else
        {
            printf("%c", *fmt);
        }
        break;

    case 'c':
        if ('%' == lastC)
        {
            c = (char) va_arg(arg, int);
            printf(" %c", c);
        }
        else
        {
            printf("%c", *fmt);
        }

        break;
    default:
        if ('%' != *fmt || ('s' != *(fmt + 1) && 'd' != *(fmt + 1) &&
        'c' != *(fmt + 1)))
            printf("%c", *fmt);
        break;
    }
    lastC = *fmt;
    fmt++;
}
va_end(arg);
}

int main()
{
    my_printf_format_v2("age = %d, name = %s, score = %d \n",
        20, "zhangsan", 98);
}
```

编译、执行，打印结果：

```
age = 20, name = zhangsan, score = 98
```

## 五、总结

我们来复盘一下上面的分析过程，开头的第一个代码本意是测试关于指针的，结果到最后一直分析到 C 语言中的可变参数问题。可以看出，分析问题-定位问题-解决问题是一连串的思考过程，把这个过程走一遍之后，理解才会更深刻。

我还有另外一个感受：如果我没有写公众号，就不会写这篇文章；如果不写这篇文章，就不会研究的这么较真。也许在中间的某个步骤，我就会偷懒对自己说：理解到这个层次就差不多了，不用继续深究了。所以说以文章的形式来把自己的思考过程进行输出，是技术提升是非常有好处的，也强烈建议各位小伙伴尝试一下这么做。

而且，如果这些思考过程能得到你们的认可，那么我就会更有动力来总结、输出文章。因此，如果这篇总结对你能有一丝丝的帮助，请转发、分享给你的技术朋友，在此表示衷心的感谢！

另外，文中如果有错误，非常欢迎在留言区一起讨论。或者添加我的个人微信，这样沟通就更及时、有效率。

祝你好运！

### 【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。



# 公众号【IOT物联网小镇】

转载：欢迎转载，但未经作者同意，必须保留此段声明，必须在文章中给出原文连接。

## 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](#)