

## 跟涛哥一起学嵌入式 21：一个static关键字引发的思考

文档说明	作者	日期
来自微信公众号：宅学部落(armLinuxfun)	王利涛	2019.10.22
嵌入式视频教程淘宝店： <a href="https://wanglitao.taobao.com/">https://wanglitao.taobao.com/</a>		
联系微信：brotau(宅学部落)		

今天有个学员问了一个C语言的static静态变量的细节问题，以前自己也没怎么注意过，感觉挺有意思，就跟大家分享下。

```
lpf@ubuntu: ~/wm
1 #include<stdio.h>
2 static int j;
3 int fun1(void)
4 {
5     static int i = 0;
6     i++;
7     return i;
8 }
9
10 int fun2(void)
11 {
12     j = 0;
13     j++;
14     return j;
15 }
16
17 int main()
18 {
19     int k,i,j;
20     for(k=0;k<10;k++)
21     {
22         i= fun1();
23         j= fun2();
24     }
25     printf("main i= %d j = %d\n",i,j);
26     return 0;
27 }
```

-- INSERT --

► Description of upd

在上面的程序中，分别定义了一个静态变量 *i* 和全局变量 *j*，然后在 *main* 函数中循环调用10次后，再分别打印 *i* 和 *j* 的值，不用纠结地去想，答案分别是 *i*=10, *j*=1。

然后问题就来了：为什么 *i* 的值等于10，而 *j* 的值等于1呢？

全局变量 *j* 的值等于1，这个很好理解：我们每次调用 *fun2* 函数时，都会将全局变量 *j* 重新赋值为0，然后来个 ++ 操作，所以 *j* 的值在调用10次之后依然是 1。

而对于静态变量 *i*，当我们10次调用 *fun1* 时，*fun1* 函数体内的语句 *static int i = 0;* 会不会每次都会将 *i* 初始化为0？答案是不会。

这里就要涉及到普通局部变量和静态变量的区别了：普通局部变量是定义在函数体内的变量，是在栈中存储的。我们可以通过栈指针来访问和修改它，当函数退出时，栈销毁，普通局部变量也就灰飞烟灭，生命周期结束。而静态变量则不同，当一个局部变量使用 *static* 修饰时，我们可以改变这个局部变量的存储方式从栈中迁移到数据段或BSS段中，升级为静态变量，但是这个静态变量的作用域不变，仍然由大括号 {} 决定。

比如下面这段程序，我们定义了一个静态变量 *i*：

```
1  #include <stdio.h>
2
3  int fun1 (void)
4  {
5      static int i = 0;
6      i++;
7      printf ("i = %d\n", i);
8  }
9
10 int main (void)
11 {
12     for (int k = 0; k < 10; k++)
13         fun1 ();
14     return 0;
15 }
```

知乎 @宅学部落

使用ARM交叉编译器编译，然后使用 *readelf* 文件查看其符号表：

```
$ arm-linux-gnueabi-gcc main.c
$ readelf -s a.out
```

40:	00000000	0	FILE	LOCAL	DEFAULT	ABS	/usr/lib/gcc-cross/arm-li
41:	000102c8	0	NOTYPE	LOCAL	DEFAULT	11	\$a
42:	00010528	0	NOTYPE	LOCAL	DEFAULT	14	\$a
43:	00000000	0	FILE	LOCAL	DEFAULT	ABS	crtstuff.c
44:	00020f14	0	OBJECT	LOCAL	DEFAULT	20	__JCR_LIST__
45:	00010370	0	NOTYPE	LOCAL	DEFAULT	13	\$a
46:	00010370	0	FUNC	LOCAL	DEFAULT	13	deregister_tm_clones
47:	00010394	0	NOTYPE	LOCAL	DEFAULT	13	\$d
48:	000103a0	0	NOTYPE	LOCAL	DEFAULT	13	\$a
49:	000103a0	0	FUNC	LOCAL	DEFAULT	13	register_tm_clones
50:	000103cc	0	NOTYPE	LOCAL	DEFAULT	13	\$d
51:	00021024	0	NOTYPE	LOCAL	DEFAULT	23	\$d
52:	000103d8	0	NOTYPE	LOCAL	DEFAULT	13	\$a
53:	000103d8	0	FUNC	LOCAL	DEFAULT	13	__do_global_dtors_aux
54:	000103fc	0	NOTYPE	LOCAL	DEFAULT	13	\$d
55:	00021028	1	OBJECT	LOCAL	DEFAULT	24	completed.9905
56:	00020f10	0	NOTYPE	LOCAL	DEFAULT	19	\$d
57:	00020f10	0	OBJECT	LOCAL	DEFAULT	19	__do_global_dtors_aux_fin
58:	00010400	0	NOTYPE	LOCAL	DEFAULT	13	\$a
59:	00010400	0	FUNC	LOCAL	DEFAULT	13	frame_dummy
60:	00010430	0	NOTYPE	LOCAL	DEFAULT	13	\$d
61:	00020f0c	0	NOTYPE	LOCAL	DEFAULT	18	\$d
62:	00020f0c	0	OBJECT	LOCAL	DEFAULT	18	__frame_dummy_init_array_
63:	00021028	0	NOTYPE	LOCAL	DEFAULT	24	\$d
64:	00000000	0	FILE	LOCAL	DEFAULT	ABS	main.c
65:	00010530	0	NOTYPE	LOCAL	DEFAULT	15	\$d
66:	00010438	0	NOTYPE	LOCAL	DEFAULT	13	\$a
67:	00010474	0	NOTYPE	LOCAL	DEFAULT	13	\$d
68:	0002102c	4	OBJECT	LOCAL	DEFAULT	24	i.4673
69:	0001047c	0	NOTYPE	LOCAL	DEFAULT	13	\$a
70:	0002102c	0	NOTYPE	LOCAL	DEFAULT	24	\$d
71:	00000000	0	FILE	LOCAL	DEFAULT	ABS	elf-init.oS
72:	000104c0	0	NOTYPE	LOCAL	DEFAULT	13	\$a
73:	00010518	0	NOTYPE	LOCAL	DEFAULT	13	\$d
74:	00010520	0	NOTYPE	LOCAL	DEFAULT	13	\$a
75:	00000000	0	FILE	LOCAL	DEFAULT	ABS	crtstuff.c

我们会看到一个叫 i.4673 的变量保存在BSS段中，这就说明了，当我们使用static修饰一个局部变量时，它的存储方式会发生变化，有栈中迁移到数据段或BSS段中。

还需要注意的是，当我们使用static修饰一个局部变量时，如果我们不初始化，默认值是0；而普通局部变量如果不初始化，默认值则是一个随机值。

如果我们使用static定义一个静态变量时对其进行初始化，这个初始化语句只有第一次执行才有效。这也解释了为什么我们多次调用fun1时，i 的值不会重新初始化为0，而是保存上一次函数退出时的值。我们接着再看一个例子：

知乎 @宅学部落



```

1 #include <stdio.h>
2
3 int fun1 (int arg)
4 {
5     static int i = arg;
6     i++;
7     printf ("i = %d\n", i);
8 }
9
10 int main (void)
11 {
12     for (int k = 0; k < 10; k++)
13         fun1 (k);
14     return 0;
15 }

```

知乎 @宅学部落

在上面的程序中，我们在调用fun1时，使用一个变量 arg 来给静态变量 i 进行初始化。编译这个程序，你会发现编译错误：

```

error: initializer element is not constant
static int i = arg;

```

这是另外一个需要注意的地方：static静态变量初始化语句需要使用常量进行初始化。

为什么static修饰的局部变量需要常量才能初始化呢？其实这个也很好理解：static修饰的静态变量，是存储在数据段或BSS段中的，这两个段中的变量在编译阶段就要给它们分配存储空间，然后初始化。这跟函数内的局部变量在运行时才给它们分配存储空间是不同的。在编译阶段，因为数据段或BSS段的变量需要一个确定的值来初始化(要么是0，要么是指定的常量值)，当static静态变量也要保存到这块区域时，因此必须也要用一个常量来初始化。

以上就是我们使用static关键字去修饰一个静态变量时，需要注意的一些细节。接下来我们就要思考了：当我们在函数体内去定义一个静态变量时，编译器到底是如何处理它的，或者说生成的指令代码到底是什么样的？我们以下的代码为例：

```

1  #include <stdio.h>
2
3  int fun1 (void)
4  {
5      static int i = 0;
6      i++;
7      printf ("i = %d\n", i);
8  }
9
10 int main (void)
11 {
12     for (int k = 0; k < 10; k++)
13         fun1 ();
14     return 0;
15 }

```

知乎 @宅学部落

交叉编译上面的程序，然后再反汇编，生成汇编代码：

```

$ arm-linux-gnueabi-gcc main.c
$ arm-linux-gnueabi-objdump -D a.out > 1.s

```

分析生成的汇编文件1.s，找到fun1函数的实现：

```

00010438 <fun1>:
10438: e92d4800    push    {fp, lr}
1043c: e28db004    add fp, sp, #4
10440: e59f302c    ldr r3, [pc, #44] ; 10474 <fun1+0x3c>
10444: e5933000    ldr r3, [r3]
10448: e2833001    add r3, r3, #1
1044c: e59f2020    ldr r2, [pc, #32] ; 10474 <fun1+0x3c>
10450: e5823000    str r3, [r2]
10454: e59f3018    ldr r3, [pc, #24] ; 10474 <fun1+0x3c>
10458: e5933000    ldr r3, [r3]
1045c: e1a01003    mov r1, r3
10460: e59f0010    ldr r0, [pc, #16] ; 10478 <fun1+0x40>
10464: ebffff9d    bl 102e0 <printf@plt>
10468: e1a00000    nop ; (mov r0, r0)
1046c: e1a00003    mov r0, r3
10470: e8bd8800    pop {fp, pc}
10474: 0002102c    andeq r1, r2, ip, lsr #32
10478: 00010530    andeq r0, r1, r0, lsr #32

```

知乎 @宅学部落

分析fun1的反汇编代码，我们可以看到，当我们多次调用fun1函数时，并没有每次都将变量i赋值为0，在函数体内压根就没有这样的指令，而是一上来就对i做++操作，i变量存储在0002102C这个地址，而这个地址在哪里呢？在我们的BSS段空间内：

```

24 Disassembly of section .data:
25
26 00021020 <__data_start>:
27     21020:    00000000    andeq    r0, r0, r0
28
29 00021024 <__dso_handle>:
30     21024:    00000000    andeq    r0, r0, r0
31
32 Disassembly of section .bss:
33
34 00021028 <__bss_start>:
35     21028:    00000000    andeq    r0, r0, r0
36
37 0002102c <.4673>:
38     2102c:    00000000    andeq    r0, r0, r0
39
40 Disassembly of section .comment:
41

```

知乎 @宅学部落

通过以上分析，我们可以得出结论：当我们在一个函数体内使用static定义一个静态局部变量时，在编译阶段，遇到static int i = 0;这样的语句，编译器会将该变量存储在数据段或BSS段中。而且这个初始化语句只有一次有效，阅后即焚。当我们多次调用fun1时，我们在函数体内并没有找到这条语句的汇编指令，这说明编译器在首次编译后，然后就可能把它当作一个声明语句来处理了。

C语言博大精深，任何一个细节细细品味，都能牵涉出一系列自己想不到的知识来，进而能不断更新和完善我们的知识体系。感谢这位学员的问题，让我们对C语言的语法理解又加深了一层。

专注嵌入式、Linux精品教程：<https://wanglitao.taobao.com/>

嵌入式技术教程博客：<http://zhaixue.cc/>

联系 QQ：3284757626

嵌入式技术交流QQ群：475504428

微信公众号：宅学部落(armlinuxfun)

