

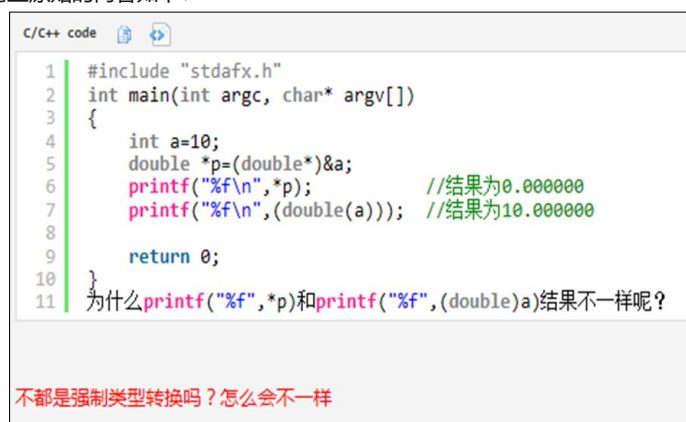
## 案例 3：数据类型转换案例

### 1 题目：

对于以下 C 语言程序，为什么第 6 行和第 7 行语句打印出来的结果不同？

```
1 #include <stdio.h>
2 int main ()
3 {
4     int a = 10;
5     double *p = (double*) &a;
6     printf("%f\n", *p);
7     printf("%f\n", (double)a);
8     return 0;
9 }
```

网上原贴的内容如下：



```
C/C++ code
1 #include "stdafx.h"
2 int main(int argc, char* argv[])
3 {
4     int a=10;
5     double *p=(double*)&a;
6     printf("%f\n",*p);           //结果为0.000000
7     printf("%f\n",(double(a))); //结果为10.000000
8
9     return 0;
10 }
11 为什么printf("%f",*p)和printf("%f",(double)a)结果不一样呢？

不都是强制类型转换吗？怎么会不一样
```

### 2 知识点：

该案例涉及数据的表示、数据类型转换以及 ABI 定义的过程调用约定、缓冲区溢出攻击防范等知识点。

通过案例机器级代码的展示，可以启发学生多从机器级代码理解程序的行为。

### 3 问题分析与讨论要点

P 是一个 double 类型的指针，指向变量 int a 的地址。因此，第一个 printf 将变量 a 首地址开始的 8 个字节的存储内容按照浮点数解释并打印。而第二个 printf 将变量 a 做强制类型转换，也就是说，打印的是将整数 10 强制类型转换为浮点数 10.00...00 打印出来。第一次打印，a 本身的机器数没有变化，第二次打印，因为强制类型转换，机器数发生了变化。根据以上原则，下面进行具体分析。

批注 [kk1]: 小写 p

#### 3.1.1 linux x86 平台运行结果分析

指针 p 取得变量 a 的地址，由于指针所指类型为 double，因而地址为 &a ~ (&a+7) 共 8 个连续字节组成的 64 位被解释为一个双精度浮点数。IA-32 采用小端模式，8 个字节中的低 32 位，即 &a ~ (&a+3) 4 个字节中存储的内容为 int 型变量 a，即这 4 个字节对应的机器数为十六进制的 00 00 00 0Ah；而高位的 4 个字节 (&a+4) ~ (&a+7) 不同的编译器有不同的处理方式，即有不同的填充规则。

##### (1) 高 4 字节填充 0

如果高 4 字节填充 0，则 8 个字节的机器数为 0000 0000 0000 000Ah。按照 IEEE 754 双精度浮点数解释时，最高符号为 0，为正数；11 位阶码 E 全 0，属于非规格化数，指数按规定为 -1023；52 位尾数只在最后 4 位有非零值“1010”。因而其值可如下计算：

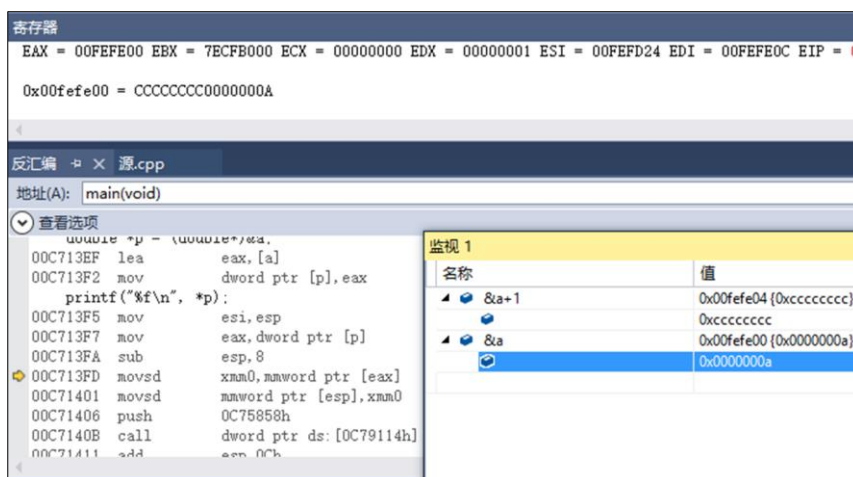
$$+0.\underbrace{0000\dots0000}_{48\text{个}0}101_2 \times 2^{-1023_{10}}$$

按上式计算的十进制数值非常小，而 “%f” 格式默认输出小数点后 6 位十进制小数，故此  
时，第一个 printf 打印的 6 位均为 0。下图用 “.320%f” 打印，可见该数值的有效数字远在小  
数点后 300 多位。

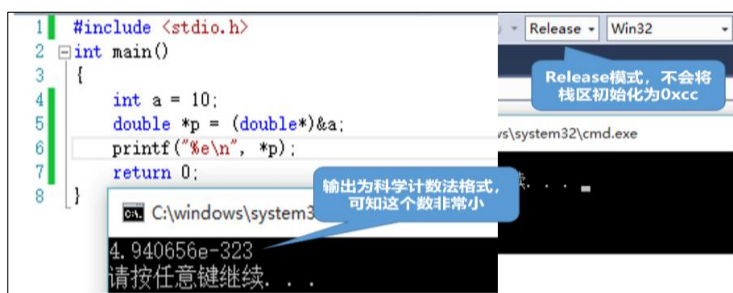
```
#include "stdafx.h"

int main(int argc, char* argv)
{
    int a = 10;
    double *p = (double*)6;
    printf("%f\n", *p);
    printf("%f\n", (double)10.000000);
    return 0;
}
```

下图为 Windows 系统 VS 开发环境、Release 模式下的运行结果。在 Release 模式下编译器也用 0 对栈帧初始化，此时打印结果为机器数 0000 0000 0000 000A 对应的浮点数值。

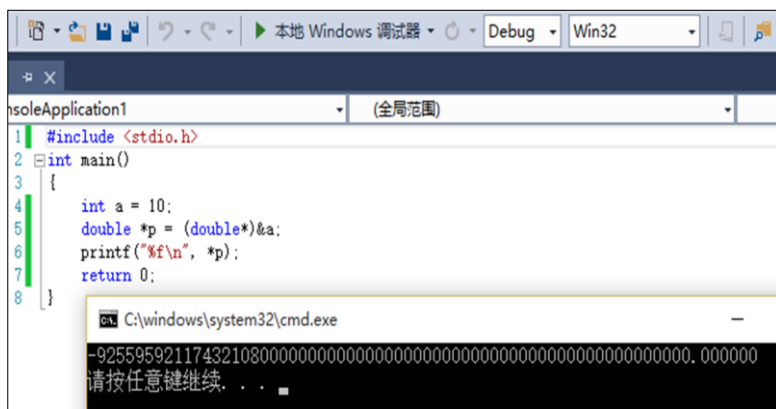


如果用 %e 打印，可以输出浮点数的科学记数法表示的十进制值，从下图可以看出，输出的数值非常小。

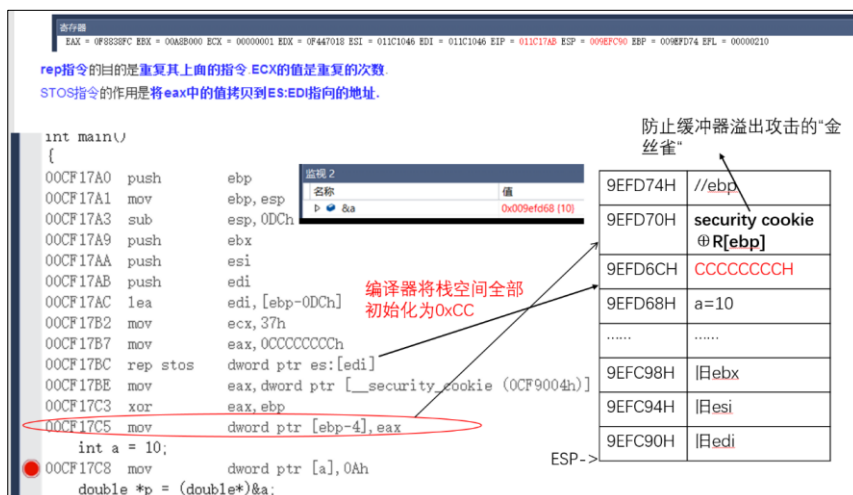


## (2) 高 4 字节填充 CCh

不同的编译器对上述高 4 字节填充值，或者说对栈帧的初始化可能不同，填充的值并不一定为 0。下图为 Windows 系统 VS 开发环境、Debug 模式下的运行结果。在 Debug 模式下，编译器会用“CCh”（int 3，断点设置）对栈帧进行初始化，此时第一个 printf 打印的结果为机器数 CCCC CCCC 0000 000A 对应的浮点数值。



下图同样显示了 Debug 模式下会对栈用“CCh”进行初始化，并且 Windows 下 VS 开发工具采用的缓冲区溢出攻击防范方法采用了“设置金丝雀值”的方式。



### (3) 高 4 字节填充随机值

下图是 Linux、GCC 下执行的结果，每次打印数值都不相同。原因在于采用了缓冲区溢出攻击防范措施“栈随机化”，使得每次栈低位置都发生改变，导致对应机器数的高 32 位（是栈区的某个地址）每次都不相同。

```
环境 1 $ cat test2.c
#include <stdio.h>
int main()
{
    int a=10;
    double *p=(double*)&a;
    printf("%e\n\n", *p);
    return 0;
}
环境 1 $ gcc test2.c -o test2
环境 1 $ for i in `seq 5`; do ./test2 ; done
-1.873005e+00
-1.131204e-02
-1.196463e-01
-8.468896e-03
-7.310276e-01
```

下图是在 mac 机上运行的结果，高四字节的内容也是变化的

```
[conjee@Yilis-MacBook-Pro Downloads % ./a.out
a 0 0 0 f0 19 8f bd
-3.535811e-12
10.000000
[conjee@Yilis-MacBook-Pro Downloads % ./a.out
a 0 0 0 f0 d9 fa bd
-3.907381e-10
10.000000
```

### (3) 变量 p 与 a 的地址观察

下图展示了 x86 平台下变量 a 和 p 在栈帧中分配的位置顺序，图中 a 和 p 连续存放。

```
1 #include <stdio.h>
2 int main()
3 {
4     int a=10;
5     double *p=(double*)&a;
6     printf("Scientific: %e\n", *p);
7     printf("Machine: %08x %08x\n", *(&a+1), a);
8     printf("Address: %p\n\n", &a);
9     return 0;
10 }
(gdb) b 6
Breakpoint 1 at 0x8048436: file test.c, line 6.
(gdb) r
Starting program: /home/tangruize/临时/a.out
Breakpoint 1, main () at test.c:6
6     printf("Scientific: %e\n", *p);
(gdb) p &a
$1 = (int *) 0xffffcc68
(gdb) p &a+1
$2 = (int *) 0xffffcc6c
(gdb) p &p
$3 = (double **) 0xffffcc6c
(gdb)
```

+6c	p : &a= 0xffffcc68
+68	a : 0xa
	.....
+8	p : &a= 0xffffcc68
+4	a : 0xa
ESP	0x8048500 (指向 "%f\n" 的指针)
	这里R[esp]=0xffffcc60

疑问：为什么局部变量的地址在内存空间（比0xc0000000更大）？

批注 [kk2]: “栈顶”??

另外，linux 下可以关闭栈随机化：

用 root 权限执行：

```
# echo 0 > /proc/sys/kernel/randomize_va_space
```

运行结果，就永远是-nan

```
mars@mars-QiTianM428-A688
-nan
mars@mars-QiTianM428-A688
-nan
mars@mars-QiTianM428-A688
-nan
mars@mars-QiTianM428-A688
-nan
mars@mars-QiTianM428-A688
-nan
mars@mars-QiTianM428-A688
-nan
```

### 3.1.2 RISC-V 平台运行结果分析

下图为案例在 RISC-V 平台上反汇编的结果。

```
0000000000000624 <main>:
624: 1101      addi sp,sp,-32
626: ec06      sd ra,24(sp)
628: e822      sd s0,16(sp)
62a: 1000      addi s0,sp,32
62c: 47a9      li a5,10
62e: fef42223  sw a5,-28(s0)
632: fe440793  addi a5,s0,-28
636: fef43423  sd a5,-24(s0)
63a: fe843783  ld a5,-24(s0)
63e: 239c      fld fa5,0(a5)
640: e20785d3  fmv.x.d a1,fa5
644: 00000517  auipc a0,0x0
648: 07450513  addi a0,a0,116 # 6b8 <__libc_csu_fini+0x4>
64c: f05ff0ef  jal ra,550 <printf@plt>
650: 4781      li a5,0
652: 853e      mv a0,a5
654: 60e2      ld ra,24(sp)
656: 6442      ld s0,16(sp)
658: 6105      addi sp,sp,32
65a: 8082      ret
```

从这个例子来看，a 和 p 在栈帧中是连续存放的，a 的地址确实比 p 更小，但是 RISC-V 的相关手册没有规定必须这样存放，换一个编译器 a 和 p 的位置关系可能就会发生变化，所以与 a 和 p 的位置相关的结论，并没有普适性。

### 3.1.3 ARM 平台运行结果分析

下图为案例在某一款 ARM64-v8 平台上反汇编的结果。两条划线处指令可以看出为什么两个强制类型转化打印的值不同。第一条划线处指令“ldr d0,[x0]”执行后，d0 中是变量 a 值存放位置首地址开始的 8 个字节的内容。而第二条划线处指令“scvtf”则将定点数强制类型转换为浮点数，产生不同的机器码。

```
126 00000000004005b4 <main>:
127 4005b4: a9be7bfd stp x29, x30, [sp, #-32]!
128 4005b8: 910003fd mov x29, sp
129 4005bc: 52800140 mov w0, #0xa // #10
130 4005c0: b90017e0 str w0, [sp, #20]
131 4005c4: 910053e0 add x0, sp, #0x14
132 4005c8: f9000fe0 str x0, [sp, #24]
133 4005cc: f9400fe0 ldr x0, [sp, #24]
134 4005d0: fd400000 ldr d0, [x0]
135 4005d4: 90000000 adrp x0, 4000000 <__abi_tag-0x254>
136 4005d8: 9118c000 add x0, x0, #0x630
137 4005dc: 97ffffb5 bl 4004b0 <printf@plt>
138 4005e0: b94017e0 ldr w0, [sp, #20]
139 4005e4: 1e620000 scvtf d0, w0
140 4005e8: 90000000 adrp x0, 4000000 <__abi_tag-0x254>
141 4005ec: 9118c000 add x0, x0, #0x630
142 4005f0: 97ffffb5 bl 4004b0 <printf@plt>
143 4005f4: 52800000 mov w0, #0x0 // #0
144 4005f8: a8c27bfd ldp x29, x30, [sp], #32
145 4005fc: d65f03c0 ret
146
147 Disassembly of section .fini:
148
```

此外，从上图中的反汇编结果可见，a 在[sp,#20]处，而 p 在[sp,#24]处，该例中 p 接着 a 后面存放。

以下为另一个 ARM64-v8 平台运行的结果分析。源程序如下图所示：

```
#include <stdio.h>
int main()
{
    int a = 10;
    double *p = (double *) &a;
    printf("%e\n", *p);
    printf ("&a : %.8x, &p : %.8x\n", &a, &p);
    printf ("*(&a + 1): %.8x, a: %.8x\n", *(&a + 1), a);
    printf("done\n\n");
    return 0;
}
```

反汇编结果如下:

#### 反汇编

0000000004006c4 <main>:

```
4006c4: a9be7bfd      stp x29, x30, [sp, #-32]!
4006c8: 910003fd      mov x29, sp
4006cc: 52800140      mov w0, #0xa // #10
4006d0: b9001fe0      str w0, [sp, #28] //int a 存在栈偏移 28 地址中
4006d4: 910073e0      add x0, sp, #0x1c
4006d8: f9000be0      str x0, [sp, #16]
4006dc: f9400be0      ldr x0, [sp, #16]
4006e0: fd400000      ldr d0, [x0] //从栈偏移 28 地址中直接取出 64 位 double 给*p
4006e4: 90000000      adrp x0, 400000 <__abi_tag-0x278>
4006e8: 911da000      add x0, x0, #0x768
4006ec: 97ffffa5 bl 400580 <printf@plt>
4006f0: 910043e1      add x1, sp, #0x10 //&p 地址
4006f4: 910073e0      add x0, sp, #0x1c //&a 地址
4006f8: aa0103e2      mov x2, x1
4006fc: aa0003e1      mov x1, x0
400700: 90000000      adrp x0, 400000 <__abi_tag-0x278>
400704: 911dc000      add x0, x0, #0x770
400708: 97ffff9e bl 400580 <printf@plt>
40070c: 910073e0      add x0, sp, #0x1c
400710: 91001000      add x0, x0, #0x4//&a + 1 地址
400714: b9400000      ldr w0, [x0]
400718: b9401fe1      ldr w1, [sp, #28]
40071c: 2a0103e2      mov w2, w1

400720: 2a0003e1      mov w1, w0
400724: 90000000      adrp x0, 400000 <__abi_tag-0x278>
400728: 911e2000      add x0, x0, #0x788
40072c: 97ffff95 bl 400580 <printf@plt>
400730: 90000000      adrp x0, 400000 <__abi_tag-0x278>
400734: 911ea000      add x0, x0, #0x7a8
400738: 97ffff8e bl 400570 <puts@plt>
40073c: 52800000      mov w0, #0x0 // #0
400740: a8c27bfd      ldp x29, x30, [sp], #32
400744: d65f03c0      ret
```



执行结果如下：

执行结果	-1.371609e+88
for i in {0..10}; do ./main ; done	&a : d23b935c, &p : d23b9350
-3.165019e+94	*(&a + 1): d23b9470 , a: 0000000a
&a : d38e579c, &p : d38e5790	done
*(&a + 1): d38e58b0 , a: 0000000a	-7.642088e+241
done	&a : f226eadc, &p : f226ead0
	*(&a + 1): f226ebf0 , a: 0000000a
	done
-1.592717e+51	-5.402705e+202
&a : ca91060c, &p : ca910600	&a : ea060d7c, &p : ea060d70
*(&a + 1): ca910720 , a: 0000000a	*(&a + 1): ea060e90 , a: 0000000a
done	done
-1.151678e+270	-6.663222e+294
&a : f8016f8c, &p : f8016f80	&a : fd24dc9c, &p : fd24dc90
*(&a + 1): f80170a0 , a: 0000000a	*(&a + 1): fd24ddb0 , a: 0000000a
done	done
-3.195228e+288	-2.592242e+279
&a : fbd4facc, &p : fbd4fac0	&a : f9f2466c, &p : f9f24660
*(&a + 1): fbd4fbe0 , a: 0000000a	*(&a + 1): f9f24780 , a: 0000000a
done	done
-1.060789e+232	-4.790435e+15
&a : f01b539c, &p : f01b5390	&a : c33103cc, &p : c33103c0
*(&a + 1): f01b54b0 , a: 0000000a	*(&a + 1): c33104e0 , a: 0000000a
done	done

上图中的执行结果可见，a 和 p 在栈帧中所分配的位置顺序，没有编译优化时间间隔 0xC。

而在-O2 优化时，就只间隔 0x4，连续存放了，如下图所示：

```
[#131#root@node1 /home/guoyang/Ecological]
$ gcc -O2 main.c -o main
[#132#root@node1 /home/guoyang/Ecological]
$ gcc main.c -o main^C
[#132#root@node1 /home/guoyang/Ecological]
$ ./main
-1.124711e+255
&a : f4e32d04, &p : f4e32d08
*(&a + 1): f4e32d04 , a: 0000000a
done
```

因此，一定是 a 所在的地址比 p 所在的地址更小吗？这个不一定，取决于编译器。

案例 3 的另一个教学目的，就是让学生了解非静态局部变量的分配顺序在 C 标准中没有规定，比较其地址大小等操作属于“未定义行为”。