

# 源码与汇编代码交织

有位同学问了一个很好的问题，我来试图重复下这个问题。

Q: 如图所示，从 unsigned char 通过数据类型转换到 long 为什么要这么写，我能写成下面的形式吗？

```
1 movzbq (%rdi), %rax
2 movq %rax, (%rsi)
```

下面给出了有错误解释的代码：

```
movb $0xF, (%ebx)      Cannot use %ebx as address register
movl %rax, (%rsp)      Mismatch between instruction suffix and register ID
movw (%rax), 4(%rsp)   Cannot have both source and destination be memory references
movb %al, %sil         No register named %sil
movq %rax, $0x123      Cannot have immediate as destination
movl %eax, %rdx        Destination operand and incorrect size
movb %si, 8(%rbp)      Mismatch between instruction suffix and register ID
```

3.4 这个练习给你更多经验，关于不同的数据传送指令，以及它们与 C 语言的数据类型和转换规则的关系。

src_t	dest_t	指令	注释
long	long	movq(%rdi), %rax movq %rax, (%rsi)	读 8 个字节 存 8 个字节
char	int	movsbl(%rdi), %eax movl %eax, (%rsi)	将 char 转换成 int 存 4 个字节
char	unsigned	movsbl(%rdi), %eax movl %eax, (%rsi)	将 char 转换成 int 存 4 个字节
unsigned char	long	movzbl(%rdi), %eax movq %rax, (%rsi)	读一个字节并零扩展 存 8 个字节
int	char	movl(%rdi), %eax movb %al, (%rsi)	读 4 个字节 存低位字节
unsigned	unsigned char	movl(%rdi), %eax movb %al, (%rsi)	读 4 个字节 存低位字节
char	short	movsbw(%rdi), %eax movw %ax, (%rsi)	读一个字节并符号扩展 存 2 个字节

3.5 逆向工程是一种理解系统的好方法。在此，我们想要逆转 C 编译器的效果，来确定什么样的 C 代码...

当我看到这个问题的时候，我觉得问的很棒，于是我开始思考(时间滴答滴答流逝了...

我当时的回答是：我觉得不行，但是我又没有证据说为什么不行，于是我通过以下方法试图还原作者当时的想法。我写了下面一段 c 程序，并将其保存为 test.c

```

1 #include <stdio.h>
2 int main()
3 {
4     unsigned char c = 'c';
5     long l = (long)c;
6     return 0;
7 }

```

很明显，这是一段 c 代码，我想是否有办法能够看到这段代码的汇编呢？我想到了 `objdump` 工具。于是我做了下面的操作

```

1 man objdump

```

我查到了通过 `-S` 参数可以使得交织汇编和源代码，欣喜若狂地执行下面指令

```

1 gcc -O0 -g -o test test.c
2 objdump -S test > test.asm

```

接着使用 vim 打开 `test.asm` 文件，我看到了下面的内容：

```

1 #include <stdio.h>
2 int main()
3 {
4     1129:      f3 0f 1e fa      endbr64
5     112d:      55                push   %rbp
6     112e:      48 89 e5          mov    %rsp,%rbp
7     unsigned char c = 'c';
8     1131:      c6 45 f7 63      movb   $0x63,-0x9(%rbp)
9     long l = (long)c;
10    1135:      0f b6 45 f7      movzbl -0x9(%rbp),%eax
11    1139:      48 89 45 f8      mov    %rax,-0x8(%rbp)
12    return 0;
13    113d:      b8 00 00 00 00    mov    $0x0,%eax
14 }

```

哦吼，这就是我想要的，我们看到 `long l = (long)c;` 下面的两句汇编实际上就是执行了和问题图中一样的操作，很明显 `eax` 是 32 位寄存器。上面的实验是在 `amd64` 上做的，刚好我还有一台 `arm64` 的 `ubuntu` 机器，做了下面操作：

```
parallels@jpzhu-db-edu:~$ cat /proc/version
Linux version 5.4.0-126-generic (build@bos02-arm64-060) (gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.1)) #142-Ubuntu SMP Fri Aug 26 12:15:55 UTC 2022
parallels@jpzhu-db-edu:~$ cat test.c
#include <stdio.h>

int main()
{
    unsigned char c = 'c';
    long l = (long)c;
    return 0;
}
parallels@jpzhu-db-edu:~$ gcc -O0 -g -o test test.c
parallels@jpzhu-db-edu:~$ objdump -S test > test.asm
parallels@jpzhu-db-edu:~$ vim test.asm
```

我得到了如下输出，`arm64` 下的 `ldrb` 指令也是扩展为 32 位后，在 `store` 回去，`gcc` 编译出的结果看上去都是经历了 `unsigned char -> int -> long` 的过程。

```
1 #include <stdio.h>
2 int main()
3 {
4 71c: d10043ff      sub     sp, sp, #0x10
5     unsigned char c = 'c';
6 720: 52800c60      mov     w0, #0x63                // #99
7 724: 39001fe0      strb    w0, [sp, #7]
8     long l = (long)c;
9 728: 39401fe0      ldrb    w0, [sp, #7]
10 72c: f90007e0      str     x0, [sp, #8]
11     return 0;
12 730: 52800000      mov     w0, #0x0                 // #0
13 }
```

于是我通过真实的 `gcc` 编译出的汇编代码，接受了如图所示的这种写法。

那么还有一个问题，为什么 `gcc` 在编译代码的时候要选择这种方法呢？这也是当时这位提出问题的学生追问我的。我再次陷入了思考……

我检索到了 [https://en.wikipedia.org/wiki/Sign\\_extension](https://en.wikipedia.org/wiki/Sign_extension)，我发现在 `zero extension` 部分，通常情况下，在 64 位的机器上，都是先扩展到 32 位，同时高位的 32 位也会被置零，这是一个传统。我想这个传统恐怕是为了最大程度和 32 位机器兼容。