Homework Week8

3.65

通过函数的逻辑可以看出，`i` 表示行数，`j` 表示列数。

`i` 为 `0` 到 `M - 1` 的每一行时，将 `j` 从 `0` 到 `i - 1` 的每个 `(i, j)` 对确定的位置与其对称位置 `(j, i)` 交换。

二维数组是行优先，右移一列使地址 `+ sizeof(long)`，下移一行使地址 `+ sizeof(long) * M`。

`(i, j)` 和 `(j, i)` 确定一对对称的位置，内循环每一次循环后 `j++`，前者右移一列，后者下移一行。

在汇编代码中则分别对应 `6 addq $8, %rdx` 和 `7 addq $120, %rax`。前者是 `j++` 时右移一列的 `(i, j)` 索引，后者是 `j++` 时下移一行的 `(j, i)` 索引。

所以 `sizeof(long)` 为 `8`，`sizeof(long) * M` 为 `8 * M`，即 `120`。

解得 `M == 15`，`%rdx` 是 `(i, j)` 的地址，`%rax` 是 `(j, i)` 的地址

A. `%rdx`　　B. `%rax`　　C. `M == 15`

3.69

容易看出 `mov 0x120(%rsi), %ecx` 与 `add (%rsi), %ecx` 实现了 `int n = bp -> first + bp -> last;`，将 `n` 存入 `%ecx`。可以得出 `bp -> first` 与 `bp -> last` 相差 `288` 字节。去掉 `bp -> first` 本身，**`a` 数组可能占用 `284` 字节。**

`lea (%rdi, %rdi, 4), %rax` 与 `lea (%rsi, %rax, 8), %rax` 计算 `i * 40 + bp` 存入 `%rax`。猜测是寻找 `bp -> a[i]` 的地址，该地址中错误包括了 `bp -> first` 的内存。由索引 `i` 乘以 `40` 可以得出结论，**每个 `a[i]` 占用 `40` 字节。**

`mov 0x8(%rax), %rdx` 中 `0x8` 本应是去除 `bp -> first` 的偏移量，但实际上 `int` 类型占 `4` 字节而非 `8` 字节，所以可能是为了数据对齐额外填充了 `4` 字节。这说明 **`a_struct` 中的第一个元素（可能是 `idx` 或 `x[0]`）是 `8` 字节类型**，存储在 `%rdx` 中。

同时，上文推测的 `a` 数组占用 `284` 字节应额外去掉数据对齐浪费的 `4` 字节，**`a` 数组实际为 `280` 字节**，每个 `a[i]` 占用 `40` 字节，所以 **`a` 数组实际上有 `7` 个元素，`CNT == 7`。**

`movslp %ecx, %rcx` 将 `n` 从 `int` 强转为 `long`，说明 `n` 即将赋值至的 `x[ap -> idx]` 是 `8` 字节类型，也就是说 **`x` 数组是 `8` 字节类型的数组。**

`mov %rcx, 0x10(%rax, %rdx, 8)` 将 `n` 存入 `x[ap -> idx]`。

以下分析 `mov` 的目标操作数：

1. 操作数是地址，其值为 `0x10 + %rax + %rdx * 8`
2. `%rax` 是错误包括了 `bp -> first`（及数据对齐的额外空间）的 `bp -> a[i]` 的错误地址
3. `0x10` 是偏移量，其中 `0x8` 是针对 `bp -> first` 的偏移，另外的 `0x8` 则应当是针对 `ap -> idx` 的偏移，也就是说 **`idx` 是 `a_struct`中的第一个元素。**
4. `%rdx * 8` 也印证了 `%rdx` 是索引，即 `idx`。

5. 由于 `a[i]` 占用 `40` 字节，去掉 `8` 字节的 `idx` （上文中得出结论，该结构的第一个元素是 8 字节类型）后，`x` 数组占用 `32` 字节，由于 `x` 是 `8` 字节类型的数组，所以 **`x` 有 `4` 个元素。**

A. `CNT == 7`　　B. 如下

```
typedef struct {
    long idx;
    long x[4];
} a_struct
```

3.70

A.

字段	偏移量
e1.p	0
e1.y	8
e2.x	0
e2.next	8

B.
`union` 类型占用等于其成员的最大的占用，而其两个成员都为 `8 + 8 == 16` 字节，所以 `union` 类型占用 `16` 字节。

C.
第 `2` 行的 `movq` 会对地址做解引用，所以第一行的源操作数应当是指针，偏移量为 `8` 的指针变量即 `e2.next`，将其存储的另一 `ele` 联合的指针存入 `%rax`。

第 `3` 行的 `movq` ，源操作数 `%rax` 是指针并对其解引用，将 `%rax` 处（`*(up -> e2.next)` 的第一个元素）存储的某个数值存入 `%rdx`，可能为 `e1.p` 或 `e2.x`。

第 `4` 行的 `movq` 与第一行类似，会对 `%rdx` 做解引用，因此 `rdx` 的内容是一个指针，结合第二行，说明 `%rdx` 是 `e1.p`，而该行则是对自身做解引用，得到一个 `long` 变量。

第 `5` 行的 `subq`，将 `%rdx` 减去 `%rax + 8` 处（`*(up -> e2.next)` 的第二个元素）的数值，所以此时的 `%rax + 8` 是一个数字，即 `e1.y` 的值，得到 `%rdx - (up -> e2.next -> e1.y)`，存入 `%rdx`。

第 `6` 行的 `movq` ，目的操作数是 `%rdi` 的解引用，即 `%rdi` 所存储的指针指向的元素，即 `up` 的第一个元素，源操作数 `%rdx` 是一个 `long` 变量，所以 `%rdi` 指向的是一个 `long` 变量，即 `up -> e2.x`。

综上，可以将这个过程用高级语言表示为

```
void proc(union ele *up) {
    ele *up_next = up -> e2.next;
    long *p1 = up_next -> e1.p;
    long temp = *p1;
    temp = temp - up_next -> e1.y;
```

```
    up -> e2.x = temp;
}
```

将上述代码压缩如下

```
void proc(union ele *up) {
    up -> e2.x = *(up -> e2.next -> e1.p) - up -> e2.next -> e1.y;
}
```