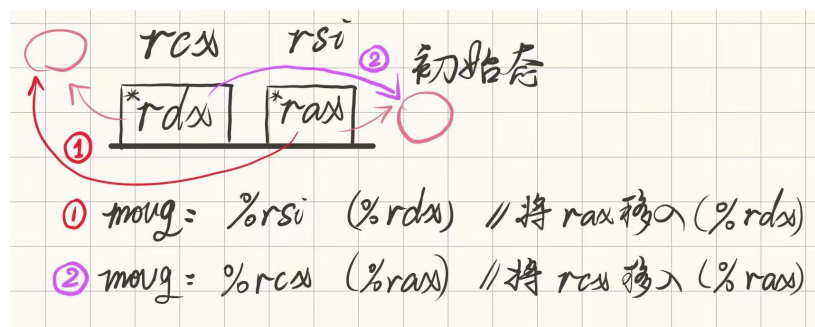


### 3.65

```
#define M 10
void transpose(long A[M][M]) {
    long i,j;
    for (i = 0; i < M; i++)
        for (j = 0; j < i; j++) {
            long t = A[i][j];
            A[i][j] = A[j][i];
            A[j][i] = t;
        }
}
```

汇编代码：

```
.L6:movq  (%rdx), %rcx  # %rcx 保存 %rdx 指向的值
Movq   (%rax),  %rsi   # %rsi 保存 %rax 指向的值
Movq   %rsi,    (%rdx) # %rdx 指向的值变成 %rsi 保存的值
Movq   %rcx,    (%rax)  # %rax 指向的值变成 %rcx 保存的值
Addq   $8,      %rdx    # %rdx 保存的值 += 8
Addq   $120,    %rax    # %rax 保存的值 += 120
Cmpq   %rdi,    %rax    # 比较 %rax 与 %rdi 的大小
Jne     .L6            # 若不相等，重新进入循环
```



A. 哪个寄存器保存着指向数组元素  $A[i][j]$  的指针？

答：%rdx 寄存器，一方面，根据上图的流程分析，我们可以知道最优先被操作的是 %rdx 寄存器，可以看作是  $\text{long } t = A[i][j]$  这一行代码，另外，因为 %rdx 寄存器在循环体内出现了 +8 的操作，这应该是在二重循环中 j++ 而指向每一行的下一个元素的操作；

B. 哪个寄存器保存着指向数组元素  $A[j][i]$  的指针？

答：%rax 寄存器，根据上面的分析，除了 %rdx 外，在循环中出现类似反馈的仅剩 %rax。

C. M 的值为多少？

答：根据上面的分析，+8 代表指针移动到该行的下一个元素，那么 +120 即为移动到下一行，因为这是一个正方形的矩阵，所以有  $8 * M = 120$ ，解得  $M = 15$ 。

### 3.69

```
typedef struct {
    int first;
    a_struct a[CNT];
    int last;
} b_struct;

void test(long i, b_struct *bp)
{
    int n = bp->first + bp->last;
    a_struct *ap = &bp->a[i];
    ap->x[ap->idx] = n;
}
```

```
void test(long i, b_struct *bp)
    i in %rdi, bp in %rsi
1  0000000000000000 <test>:
2      0:  8b 8e 20 01 00 00      mov     0x120(%rsi),%ecx
3      6:  03 0e                        add     (%rsi),%ecx
4      8:  48 8d 04 bf      lea     (%rdi,%rdi,4),%rax
5      c:  48 8d 04 c6      lea     (%rsi,%rax,8),%rax
6     10:  48 8b 50 08      mov     0x8(%rax),%rdx
7     14:  48 63 c9      movslq  %ecx,%rcx

8     17:  48 89 4c d0 10      mov     %rcx,0x10(%rax,%rdx,8)
9     1c:  c3                        retq
```

我们先逐行分析这段代码：

```
%ecx = %rsi + 0x120;    # %ecx = bp + 0x120;
%ecx = %ecx + %rsi;     # %ecx = bp + 0x120 + bp;
%rax = 5 * %rdi;        → # %rax = 5 * i;
%rax = 8 * %rax + %rsi; # %rax = 40 * i + bp;
%rdx = %rax + 8;        # %rdx = 40 * i + bp + 8;
%rcx = %ecx;            # %rcx = bp + 0x120 + bp;
```

```
(8 * %rdx + %rax + 16) = %rcx;
```

```
# %(240 * i + 8 * bp + 64 + 40 * i + bp + 16) = %bp + 0x120 + bp;
```

这题应该考察的是结构体的内存对齐，还有各种指针访问的操作...但是很乱。

Movslq 指令 将 %ecx 传入 %rcx , 应该是用于将 int 数组拓展的

按照前面的分析, &bp -> arr[i] 的计算公式为  $40 * i + \&bp + 8$

那么 int 对齐后占 8 个字节, 猜测 int 数组拓展为了 long 数组, a[i] 的大小为 40, 由于  $0x120[H] = 288[D]$ ,  $(288 - 8) / 40 = 7$ , CNT 的值应该为 7.

有一部分实在看不懂了, 有些连蒙带猜, 猜测结构体的声明为

```
Struct a_struct {  
    Long item;  
    Long arr[7];  
}
```

### 3.70

```
union ele {  
    struct {  
        long *p;  
        long y;  
    } e1;  
    struct {  
        long x;  
        union ele *next;  
    } e2;  
};
```

这个声明说明联合中可以嵌套结构。

下面的函数(省略了一些表达式)对一个链表进行操作, 链表是以上述联合作为元素的:

```
1 void proc (union ele *up) {  
2     up->_____ = *(<_____>) - _____;  
3 }
```

A. 下列字段的偏移量是多少(以字节为单位):

```
e1.p      _____  
e1.y      _____  
e2.x      _____  
e2.next   _____
```

B. 这个结构总共需要多少个字节?

C. 编译器为 proc 产生下面的汇编代码:

```
void proc (union ele *up)  
up in %rdi  
1 proc:  
2     movq    8(%rdi), %rax  
3     movq    (%rax), %rdx  
4     movq    (%rdx), %rdx  
5     subq    8(%rax), %rdx  
6     movq    %rdx, (%rdi)  
7     ret
```

在这些信息的基础上, 填写 proc 代码中缺失的表达式。提示: 有些联合引用的解释可以有歧义。当你清楚引用指引到哪里的时候, 就能够澄清这些歧义。只有一个答案, 不需要进行强制类型转换, 且不违反任何类型限制。

首先，我们要对联合体 Union 有相关的认识，它的基本性质是所有的成员是共用内存的，底层如此实现的目的是节约内存。

显然，在 Union ele 中定义了两个结构体 e1、e2，它们共用内存，在汇编代码中，每个元素的偏移量取决于它们在联合体中的顺序和大小。

A、下列字段的偏移量是多少（以字节为单位）：

显然，long \*p 定义了一个指向 long 类型的指针 p，它作为结构体的第一个元素，它的偏移量为 0，那么在结构体 e2 中定义的 long x 的偏移量也为 0，在 64 位系统上，指针通常是 8 字节，所以在结构体 e1 中的元素 y 的偏移量应该是 8，同理，e2 的元素指向下一个 union ele 的指针 \*next 的偏移量也是 8。

B、这个结构总共需要多少字节？

经过上述的分析，由于内存共用，两个结构体成员中最大的字节数即为整个 Union 所需要的字节，联合的大小将等于其最大成员的大小。实际上只需要 16 字节即可（在 union 中定义的结构体 e1 与 e2 中，long 类型的一个元素占 8 字节，一个指针占 8 字节，两者都只占用 16 字节，故最大值为 16）

C、在这些信息的基础上，填写 proc 代码中缺失的表达式。提示:有些联合引用的解释可以有歧义。当你清楚引用指引到哪里的时候，就能够澄清这些歧义。只有一个答案，不需要进行强制类型转换，且不违反任何类型限制。

```
void proc (union ele *up)
// up in %rdi
proc:
    Movq 8(%rdi), %rax    // %rax = (%rdi + 8)
    Movq (%rax), %rdx     // %rdx = (%rax)
    Movq (%rdx), %rdx     // %rdx = (%rdx)
    Subq 8(%rax), %rdx    // %rdx = (%rax + 8)
    Movq %rdx, (%rdi)     // (%rdi) = %rdx
    Ret
```

首先 %rax 存储了【指向的联合体 up 中 e2 的成员 next 指针】指向的联合体的 e2 成员】的【next 指针】赋值给【up 指向的联合体 up 的 e2 成员的【next 指针】】

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

## 6.22

假设要求你设计一个每条磁道位数固定的旋转磁盘。你知道每条磁道的位数是由最里层磁道的周长决定的，可以假设它就是中间那个圆洞的周长。因此，如果你把磁盘中间的洞做得大一点，每条磁道的位数就会增大，但是总的磁道数会减少。如果用  $r$  来表示盘面的半径， $r \cdot r$  表示圆洞的半径，那么工取什么值能使这个磁盘的容量最大？

根据题意： $r$  为盘面的半径， $x \times r$  表示圆洞的半径，我们可以假设：

每条磁道位  $B = K1 \times (x \times r)$ ;

磁道数  $T = (1-x) \times r \times K2$ ;

此时的比特总数  $B \times T = K1$  (常数)  $K2$  (常数)  $\times (r^2) \times (1-x) \times x$

根据均值不等式的知识：

当  $x = 1-x$  时， $B \times T$  的值取到最大，最大值  $= K1 \times K2 \times (r^2) / 4$ .

## 6.23

根据书上的知识：访问一个磁盘扇区内容时，

访问的平均时间 = 平均寻道时间 + 平均旋转延迟 + 平均传送时间

该扇区的内容了。一个扇区的传送时间依赖于旋转速度和每条磁道的扇区数目。因此，我们可以粗略地估计一个扇区以秒为单位的平均传送时间如下

$$T_{\text{avg transfer}} = \frac{1}{\text{RPM}} \times \frac{1}{(\text{平均扇区数} / \text{磁道})} \times \frac{60\text{s}}{1\text{min}}$$

我们可以估计访问一个磁盘扇区内容的平均时间为平均寻道时间、平均旋转延迟和平均传送时间之和。例如，考虑一个有如下参数的磁盘：

参数	值
旋转速率	7200RPM
$T_{\text{avg seek}}$	9 ms
每条磁道的平均扇区数	400

对于这个磁盘，平均旋转延迟(以 ms 为单位)是

$$T_{\text{avg rotation}} = 1/2 \times T_{\text{max rotation}} = 1/2 \times (60\text{s}/7200 \text{ RPM}) \times 1000 \text{ ms/s} \approx 4 \text{ ms}$$

平均传送时间是

$$T_{\text{avg transfer}} = 60/7200 \text{ RPM} \times 1/400 \text{ 扇区} / \text{磁道} \times 1000 \text{ ms/s} \approx 0.02 \text{ ms}$$

总之，整个估计的访问时间是

$$T_{\text{access}} = T_{\text{avg seek}} + T_{\text{avg rotation}} + T_{\text{avg transfer}} = 9 \text{ ms} + 4 \text{ ms} + 0.02 \text{ ms} = 13.02 \text{ ms}$$

这个例子说明了一些很重要的问题：

- 访问一个磁盘扇区中 512 个字节的时间主要是寻道时间和旋转延迟。访问扇区中的第一个字节用了很长时间，但是访问剩下的字节几乎不用时间。
- 因为寻道时间和旋转延迟大致相等，所以将寻道时间乘 2 是估计磁盘访问时间的简单而合理的方法。

根据本周的 Test 中的题目（书上的笔记是做 Test 时打的草稿）  
我们可以轻松地得出：

**【其实主要记住那三条公式就可以了】**

在

$$T(\text{AvgSeek}) = 4 \text{ ms}$$

$$\text{旋转速率} = 15000 \text{ RPM}$$

$$\text{平均扇区数/磁道} = 800$$

的条件下：

$$T(\text{AvgRotation}) = 1/2 \times 1/15000 \times (60\text{s})/(1\text{min}) * 1000 \text{ ms/s} = 2\text{ms}$$

$$T(\text{AvgTransfer}) = 1/15000 \times 1/800 \times (60\text{s})/(1\text{min}) * 1000 \text{ ms/s} = 5 \times 10^{-3}\text{ms}$$

$$T(\text{Access, 访问时间}) = 4 \text{ ms} + 2 \text{ ms} + 5 \times 10^{-3} \text{ ms} = 6.005 \text{ ms}$$

## 6.24

在最佳的情况下，无需进行磁头的寻找，只需要等待磁盘旋转到正确的扇区。  
根据给定的信息，我们可以计算出在最佳情况下的总访问时间：

$$T(\text{Transfe}) = T(\text{Rotation}) = T(\text{AvgRotation}) = 2 \text{ ms};$$

$$T(\text{Access}) = T(\text{AvgSeek}) + T(\text{AvgRotation}) + T(\text{Transfer})$$

$$= 4 \text{ ms} + 2 \text{ ms} + 2 \text{ ms}$$

$$= 8\text{ms}$$

因此，最佳情况下读取整个文件所需的时间为 8 ms.

B. 在随机情况下，每次访问新的逻辑块都需要进行寻道，因此需要加上平均寻道时间和平均旋转时间。

在：

$$T(\text{AvgSeek}) = 4 \text{ ms};$$

$$T(\text{AvgRotation}) = 2 \text{ ms};$$

$$\text{逻辑块数量} = 4000;$$

的情况下：

计算出在随机情况下：

$$\begin{aligned}T(\text{Access}) &= 4000 \times [T(\text{AvgSeek}) + T(\text{AvgRotation})] \\&= 4000 \times (4 \text{ ms} + 2 \text{ ms}) = 4000 \times 6 \text{ ms} \\&= 24000 \text{ ms} \\&= 24 \text{ s}\end{aligned}$$

因此，在随机情况下读取文件需要 24 s.