



Datawhale 开源社区

DATAWHALE OPEN SOURCE COMMUNITY

# 深入理解计算机系统 (7)

Computer Systems A Programmer's Perspective

CSAPP

李岳昆、易远哲

realgurk@gmail.com、yuanzhe.yi@outlook.com

2021 年 12 月 14 日



第 I 部分

# 程序的机器级表示-III

过程

数组分配和访问



在大型软件的构建过程中，需要对复杂功能进行切分，过程提供了一种封装代码的方式，它可以隐藏某个行为的具体实现，同时提供清晰简洁的接口定义，在不同的编程语言中，过程的具体实现又是多种多样的。例如 C 语言中的函数，Java 语言中的方法等。

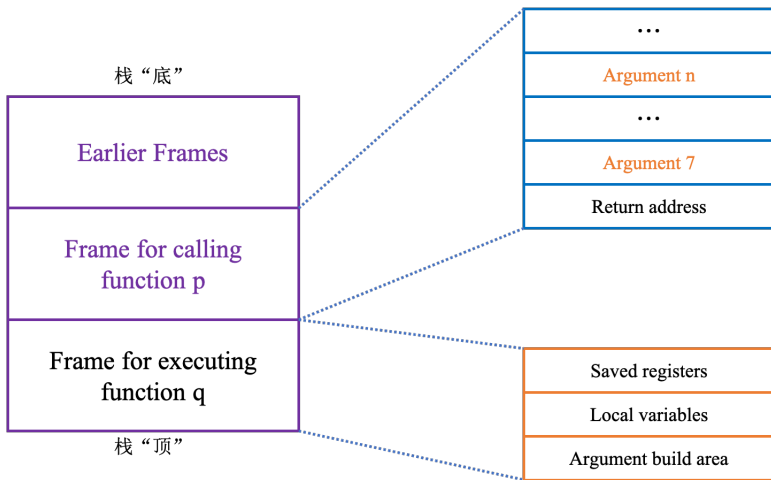
接下来，我们以 C 语言中的函数调用为例，介绍一下过程的机制，为了方便讨论，假设函数 P 调用函数 Q，函数 Q 执行完返回函数 P，这一系列操作包括图中的一个或者多个机制。

```
long P0      ▪传递控制
{
    ...      ▪传递数据
}
long Q0
{
    ...      ▪分配和释放内存
}
```

在函数 P 调用函数 Q 的例子中，当函数 Q 正在执行时，函数 P 以及相关调用链上的函数都会被暂时挂起。



我们先来介绍一下栈帧的概念，当函数执行所需要的存储空间超出寄存器能够存放的大小时，就会借助栈上的存储空间，我们把这部分存储空间称为函数的栈帧。对于函数 P 调用函数 Q 的例子，包括较早的帧、调用函数 P 的帧，还有正在执行函数 Q 的帧，具体如图所示。



以 main 函数调用 multstore 函数为例来解释一下指令 call 和指令 ret 的执行情况

```
1 #include<stdio.h>
2 void multstore(long, long, long *);
3 int main(){
4     long d;
5     mulstore(2, 3, &d);
6     printf("2 * 3 —> %d\n", d);
7     return 0;
8 }
9 long mult2(long a, long b){
10     long s = a * b;
11     return s;
12 }
```



```
1 long mult2(long, long);
2 void multstore(long x, long y, long *dest){
3     long t = mult2(x, y);
4     *dest = t;
5 }
```

- `linux> gcc -Og -o prog main.c mstore.c`
- `linux> objdump -d prog`

节选了相关的部分的反汇编代码，具体如图所示。

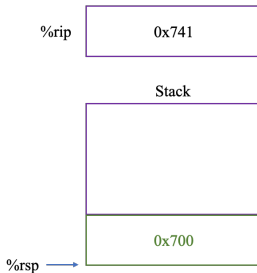
```
000000000000006da<main>:
...
6fb: e8 41 00 00 00  callq   741 <multstore>
700: 48 8b 14 24      mov     (%rsp), %rdx
00000000000000741<multstore>:
...
741: 53              push    %rbx
742: 48 89 d3        mov     %rdx, %rbx
```

```
6fb: e8 41 00 00 00  callq 741 <multstore>
```

```
%rip 0x741
```



同时还要将返回地址压入栈中。



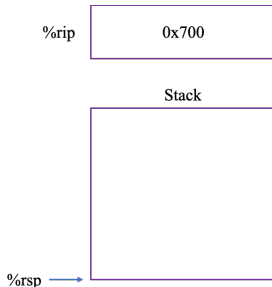
这个返回地址就是函数 `multstore` 调用执行完毕后，下一条指令的地址。

00000000000006da<main>:

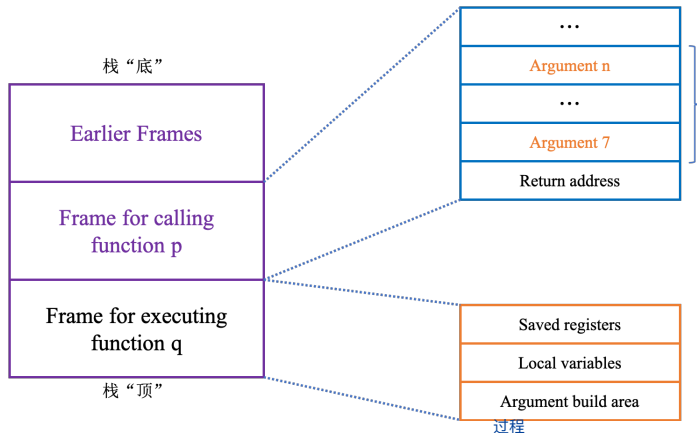
...

**6fb:** e8 41 00 00 00 **callq** 741 <multstore>

→ **700:** 48 8b 14 24 **mov** (%rsp), %rdx



说完了返回地址，再来看一下参数传递，如果一个函数的参数数量大于 6，超出的部分就要通过栈来传递。假设函数 P 有 n 个整型参数，当 n 的值大于 6 时，参数 7 参数 n 需要用到栈来传递。



参数 1 参数 6 的传递可以使用对应的寄存器。

% rdi

Argument #1 - Caller saved

% rsi

Argument #2 - Caller saved

% rdx

Argument #3 - Caller saved

% rcx

Argument #4 - Caller saved

% r8

Argument #5 - Caller saved

% r9

Argument #6 - Caller saved

例如一段代码如下：

## 代码

```
1 void proc(long a1, long *a1p,
2           int a2, long *a2p,
3           short a3, long *a3p,
4           char a4, long *a4p){
5     a1p += a1;
6     a2p += a2;
7     a3p += a3;
8     a4p += a4;
9 }
```

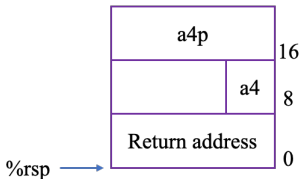


代码中函数有 8 个参数，包括字节数不同的整数以及不同类型的指针，参数 1 到参数 6 是通过寄存器来传递，参数 7 和参数 8 是通过栈来传递。

a1 → %rdi	a1p → %rsi
a2 → %edx	a2p → %rcx
a3 → %r8w	a1p → %r9
a4 → %rsp + 8	
a4p → %rsp + 16	

这里有两点需要注意一下：

- ① 通过栈来传递参数时，所有数据的大小都是向 8 的倍数对齐，虽然变量 a4 只占一个字节，但是仍然为其分配了 8 个字节的存储空间。由于返回地址占用了栈顶的位置，所以这两个参数距离栈顶指针的距离分别为 8 和 16。



- | Operand<br>size(bits) | Argument number |      |      |      |      |      |
|-----------------------|-----------------|------|------|------|------|------|
|                       | 1               | 2    | 3    | 4    | 5    | 6    |
| 64                    | %rdi            | %rsi | %rdx | %rcx | %r8  | %r9  |
| 32                    | %edi            | %esi | %edx | %ecx | %r8d | %r9d |
| 16                    | %di             | %si  | %dx  | %cx  | %r8w | %r9w |
| 8                     | %dil            | %sil | %dl  | %cl  | %r8b | %r9b |

当代码中对一个局部变量使用地址运算符时，我们需要在栈上为这个局部变量开辟相应的存储空间，接下来我们看一个与地址运算符相关的例子。

long caller

```
1 long caller(){
2     long arg1 = 534;
3     long arg2 = 1057;
4     long sum = swap(&arg1, &arg2);
5     long diff = arg1 - arg2;
6     return sum * diff;
7 }
```

函数 caller 定义了两个局部变量 arg1 和 arg2，函数 swap 的功能是交换这两个变量的值，最后返回二者之和。

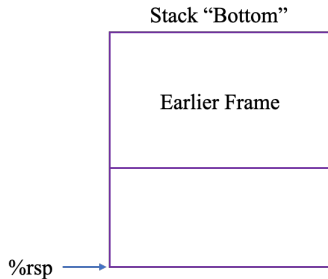
long swap

```
1 long swap(long *xp, long * yp){
2     long x = *xp;
3     long y = *yp;
4     *xp = y;
5     *yp = x;
6     return x + y;
7 }
```

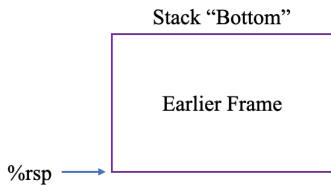
我们通过分析函数 caller 的汇编代码来看一下地址运算符的处理方式。

```
caller:
    subq    $16, %rsp
    movq    $534, (%rsp)
    movq    $1057, 8(%rsp)
    leaq    8(%rsp), %rsi
    movq    %rsp, %rdi
    call    swap
    movq    (%rsp), %rdx
    subq    8(%rsp), %rdx
    imulq   %rdx, %rax
    addq    $16, %rsp
    ret
```

第一条减法指令将栈顶指针减去 16，它表示的含义是在栈上分配 16 个字节的空間，具体如图所示。



根据接着的两条 `mov` 指令，可以推断出变量 `arg1` 和 `arg2` 存储在函数 caller 的栈帧上，接下来，分别计算变量 `arg1` 和 `arg2` 存储的地址，参数准备完毕，执行 `call` 指令调用 `swap` 函数。最后函数 caller 返回之前，通过栈顶指针加上 16 的操作来释放栈帧。



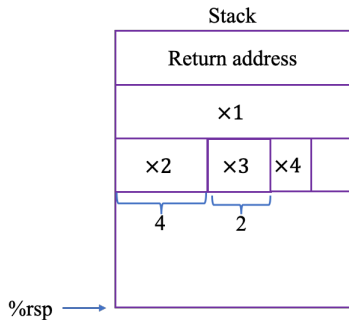


我们再看一个稍微复杂的例子：

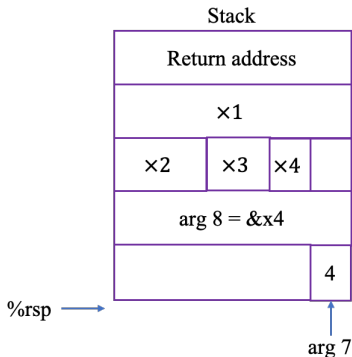
call proc

```
1 long call_proc(){
2     long x1 = 1;
3     int x2 = 2;
4     short x3 = 3;
5     char x4 = 4;
6     proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
7     return (x1 + x2) * (x3 - x4);
8 }
```

根据上面的 C 代码，我们来画一下这个函数的栈帧。根据变量的类型可知 x1 占 8 个字节，x2 占 4 个字节，x3 占两个字节，x4 占一个字节，因此，这四个变量在栈帧中的空间分配如图所示。



由于函数 `proc` 需要 8 个参数，因此参数 7 和参数 8 需要通过栈帧来传递。注意，传递的参数需要 8 个字节对齐，而局部变量是不需要对齐的。



对于 16 个通用寄存器，除了寄存器 `rsp` 之外，其他 15 个寄存器分别被定义为调用者保存和被调用者保存，具体如图所示。

## Caller-saved Register

(调用者保存寄存器)

`%rdi`    `%rsi`    `%rdx`

`%rcx`    `%r8`    `%r9`

`%rax`    `%r10`    `%r11`

## Callee-saved Register

(被调用者保存寄存器)

`%rbx`    `%rbp`    `%r12`

`%r13`    `%r14`    `%r15`

从上面的例子我们可以看到，当函数运行需要局部存储空间时，栈提供了内存分配与回收的机制。在程序执行的过程中，寄存器是被所有函数共享的一种资源，为了避免寄存器的使用过程中出现数据覆盖的问题，处理器规定了寄存器的使用的惯例，所有的函数调用都必须遵守这个惯例。

接下来，我们看一个栈保存寄存器数值的例子。

```
void P(long x, long y){  
    long u = Q(y);  
    long v = Q(x);  
    return u + v;  
}
```

P:

x in %rdi, y in %rsi

pushq %rbp

pushq %rbx

subq \$8, %rsp

movq %rdi, %rbp → Save x

movq %rsi, %rdi

call Q

...

popq %rbx

popq %rbp

ret

- 由于函数 Q 需要使用寄存器 rdi 来传递参数，因此，函数 P 需要保存寄存器 rdi 中的参数 x；保存参数 x 使用了寄存器 rbp，根据寄存器使用规则，寄存器 rbp 被定义为被调用者保存寄存器，所以便有了开头的这条指令 `pushq %rbp`，至于 `pushq %rbx` 也是类似的道理。
- 在函数 P 返回之前，使用 `pop` 指令恢复寄存器 rbp 和 rbx 的值。由于栈的规则是后进先出，所以弹出的顺序与压入的顺序相反。

最后，我们再来看一个递归调用的例子。

```

long rfact(long n){
    long result;
    if(n <= 1){
        result = 1;
    }else {
        result = n* rfact( n - 1 );
    }
    return result ;
}

```

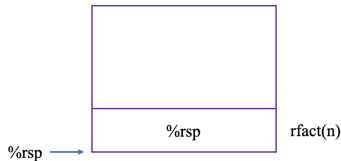
```

rfact:
    pushq    %rbx
    movq     %rdi, %rbx
    movl     $1, %eax
    cmpq     $1, %rdi
    jle      .L35
    leaq     -1(%rdi), %rdi
    call     rfact
    imulq    %rbx, %rax
.L35:
    popq     %rbx
    ret

```

这段代码是关于  $N$  的阶乘的递归实现，我们假设  $n=3$  时，看一些汇编代码的执行情况。由于使用寄存器 `rbx` 来保存  $n$  的值，根据寄存器使用惯例，首先保存寄存器 `rbx` 的值。

`pushq %rbx`  $\rightarrow$  Save `%rbx`

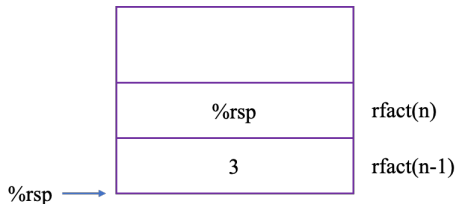


由于  $n=3$ ，所以跳转指令 `jle` 不会跳转到 L35 处执行。

指令 `leaq` 是用来计算  $n-1$ ，然后再次调用该函数。



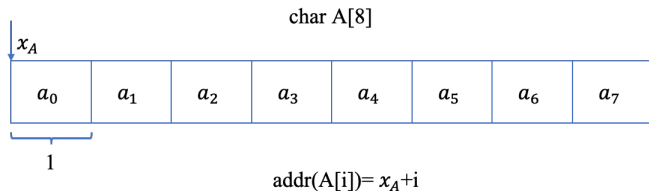
注意，此时寄存器 `rbx` 内保存的值是 3，指令 `pushq` 执行完毕后，栈的状态如图所示。



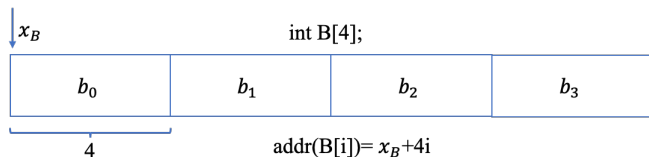
继续执行，直到  $n=1$  时，程序跳转到 L35 处，执行 pop 操作。

可以看出，递归调用一个函数本身与调用其他函数是一样的，每次函数调用都有它自己私有的状态信息，栈分配与释放的规则与函数调用返回的顺序也是匹配的，不过当  $N$  的值非常大时，并不建议使用递归调用，至于原因应该是一目了然了。

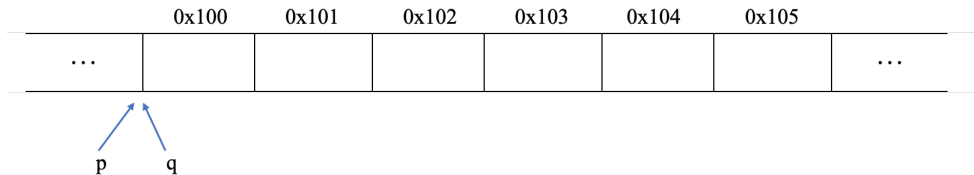
首先看几个数组的例子，数组 A 是由 8 个 char 类型的元素组成，每个元素的大小是一个字节。假设数组 A 的起始地址是  $x_A$ ，那么数组元素  $A[i]$  的地址就是  $x_{A+i}$ 。



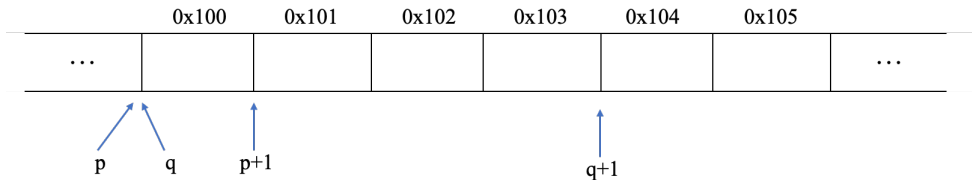
我们再来看一个 int 类型的数组，数组 B 是由 4 个整数组成，每个元素占 4 个字节，因此数组 B 总的大小为 16 个字节。假设数组 B 的起始地址是  $x_B$ ，那么数组元素  $B[i]$  的地址就是  $x_B + 4i$ 。



在 C 语言中，允许对指针进行运算，例如，我们声明了一个指向 char 类型的指针 p，和一个指向 int 类型的指针 q。为了方便理解，我们还是把内存抽象成一个大的数组。假设指针 p 和指针 q 都指向 0x100（内存地址）处。

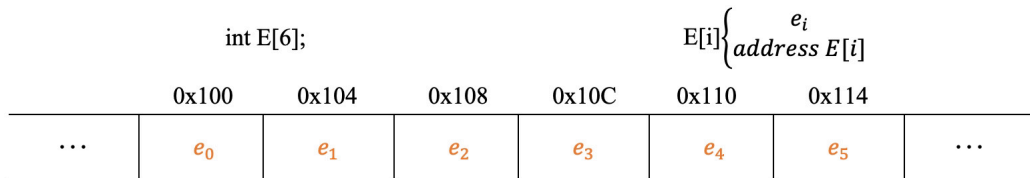


现在分别对指针  $p$  和指针  $q$  进行加一的操作，指针  $p$  加 1 指向  $0x101$  处，而指针  $q$  加 1 后指向  $0x104$  处。

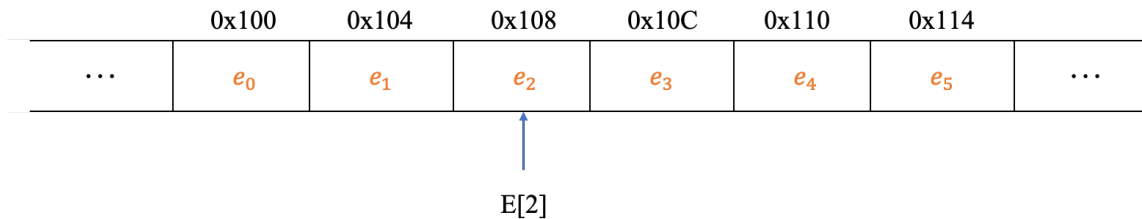


虽然都是对指针进行加一的运算，但是得到的结果却不同。这是因为对指针进行运算时，计算结果会根据该指针引用的数据类型进行相应的伸缩。

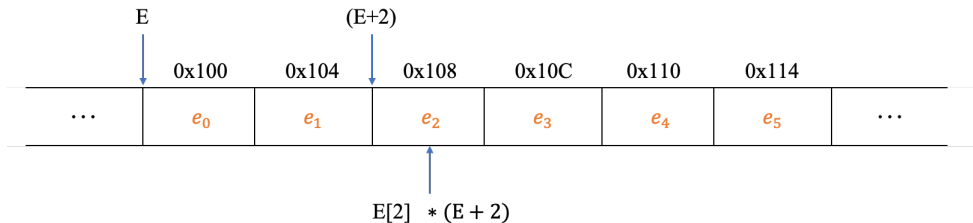
接下来，我们看一个例子，我们定义了一个数组  $E$ ，假设这个数组存放在内存中，对于数组的每一个元素都有两个属性，一个属性是它存储的内容，另外一个属性是它的存储地址，说白了就是它是啥，放在哪儿。对于元素的存储地址，可以通过取地址运算符来获得，具体如图所示。



通常我们习惯使用数组引用的方式来访问数组中的元素，例如可以使用图中的表达式来访问数组中的元素。



除此之外，还有另外一种方式，具体如图所示，其中表达式  $E+2$  表示数组第二个元素的存储地址，大写字母  $E$  表示数组的起始地址 (第 0 个元素)，此处加 2 的操作与指针加 2 的运算类似，也是与数据类型相关。



指针运算符  $*$  可以理解成从该地址处取数据，指针是 C 语言中最难理解的部分，我们理解了内存地址的概念之后，可以发现指针其实就是地址的抽象表述。



	A[0][0]	A[0][1]	A[0][2]	A[1][0]	A[1][1]	A[1][2]		A[4][0]	A[4][1]	A[4][2]	
...	$a_{00}$	$a_{01}$	$a_{02}$	$a_{10}$	$a_{11}$	$a_{12}$	...	$a_{40}$	$a_{41}$	$a_{42}$	...

Diagram illustrating a 1D array  $A$  with elements  $a_{00}, a_{01}, a_{02}, a_{10}, a_{11}, a_{12}, a_{40}, a_{41}, a_{42}$ . The elements are grouped into three sets:  $A[0]$  (containing  $a_{00}, a_{01}, a_{02}$ ),  $A[1]$  (containing  $a_{10}, a_{11}, a_{12}$ ), and  $A[4]$  (containing  $a_{40}, a_{41}, a_{42}$ ). The indices  $A[0][0], A[0][1], A[0][2], A[1][0], A[1][1], A[1][2], A[4][0], A[4][1], A[4][2]$  are shown above the corresponding elements.

无论用何种方式来理解，数组元素在内存中的存储位置都是一样的。下面我们来看一下数组元素的地址是如何计算的，对于数组 D 任意一个元素可以通过图中的计算公式来计算地址。

$$\begin{aligned} &T \ D[R][C]; \\ &\&D[i][j]=x_D+L (C \cdot i + j) \end{aligned}$$

其中， $X_D$  表示数组的起始地址；L 表示数据类型 T 的大小，如果 T 是 int 类型，L 就等于 4，T 是 char 类型，L 就等于 1；在具体的示例中，C、i、j 都是常数。

根据图中的计算公式，对于  $5 \times 3$  的数组 A，其任意元素的地址可以  $X_a + 4*(3i+j)$  来计算。

```
int A[5][3];
&A[i][j]=xA+4 ( 3 · i + j)
```

假设数组起始地址  $X_a$  在寄存器 `rdi` 中，索引值 `i` 和 `j` 分别在寄存器 `rsi` 和 `rdx` 中，我们可以用图中的汇编代码将  $A[i][j]$  的值复制到寄存器 `eax` 中，具体如图所示。

$x_A$  in %ordi,  $i$  in %rsi,  $j$  in %rdx

leaq	(%rsi, %rsi, 2)	%rax	compute $3 \cdot i$
leaq	(%rdi, %rax, 4)	%rax	compute $x_A + 12 \cdot i$
movl	(%rax, %rdx, 4)	%eax	Read from $M[x_A + 12i + 4]$

接下来，我们看一下编译器对定长多维数组的优化。首先使用以下方式将数据类型 `fix_matrix` 声明为 `16*16` 的整型数组。

- `#define N 16`
- `typedef int fix_matrix[N][N];`

通过 define 声明将 N 与常数 16 关联到一起，之后的代码中就可以使用 N 来代替常数 16，当需要修改数组的长度时，只需要简单的修改 define 声明即可。

```
1 #define N 16
2 typedef int fix_matrix[N][N]
3 int matrix(fix_matrix A, fix_matrix B, long i, long k){
4     long j;
5     int result = 0;
6     for(j = 0; j < N; j++){
7         result += A[i][j] * B[j][k];
8     }
9     return result;
10 }
```

接下来，我们看一下如何使用汇编代码访问数组元素。由于编译器对相关的操作进行了优化，因此，这段汇编代码有些晦涩难懂。

**martix:**

salq      \$6, %rdx

```
addq    %rdx, %rdi
```

```
leaq    (%rsi, %rcx, 4), %rcx
```

```
leaq    1024(%rcx), %rsi
```

```
movl    $0 , %eax
```

.L7:

• • •



```
salq    $6, %rdx
addq    %rdx, %rdi
leaq    (%rsi, %rcx, 4), %rcx
leaq    1024(%rcx), %rsi
movl    $0, %eax
```

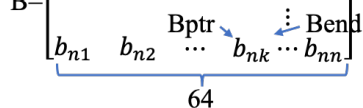
$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{i1} & a_{i2} & \cdots & a_{in} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

为了方便表述，这里我们引入三个指针来记录这三个地址，接下来，我们介绍一下循环的实现。

```
.L7:
    movl    (%rdi), %edx
    imull   %rcx, %edx
    addl    %edx, %eax
    addq    $4, %rdi
    addq    $64, %rcx
    cmpq    (%rsi), %rcx
    jne     .L7
    rep; ret
```

计算完成之后，分别移动指针 Aptr 和 Bptr 指向下一个元素，由于 int 类型占 4 个字节，对寄存器 rdi 加 4 的这个操作，对应于移动指针 Aptr 指向数组 A 的下一个元素。由于数组 B 一行元素的数量为 16 个，每个元素占 4 个字节，因此相邻列元素的地址相差为 64 个字节，对寄存器 rcx 进行加 64 的操作对应移动指针 Bptr 指向数组 B 的下一个元素。

判断循环结束的条件是：指针 Bptr 指针与指针 Bend 是否指向同一个内存地址，如果二者不相等，继续跳转到 L7 处执行，如果二者相等，循环结束。

$$B = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1k} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2k} & \cdots & b_{2n} \\ \vdots & & & & & \\ b_{n1} & b_{n2} & \cdots & b_{nk} & \cdots & b_{nn} \end{bmatrix}$$


通过这段汇编代码，我们可以发现，编译器使用了很巧妙的方式来计算数组元素的地址，这些优化方法显著的提升了程序的执行效率。

在 C89 的标准中，程序员在使用变长数组时需要使用 malloc 这类函数，为数组动态分配存储空间。在 ISO C99 的标准中，引入了变长数组的概念，因此，我们可以通过下列代码的方式来声明一个变长数组。

## 变长数组

```
1 int A[expr1][expr2];
2 long var_ele(long n, int A[n][n], long i, long k){
3     return A[i][j];
4 }
```

它可以作为一个局部变量，也可以作为函数的参数，当变长数组作为函数参数时，参数 `n` 必须在数组 `A` 之前。

变长数组元素的地址计算与定长数组类似，不同点在于新增了参数  $n$ ，需要使用乘法指令来计算  $n$  乘以  $i$ 。

<code>var_ele:</code>		
<code>imulq</code>	<code>%rdx, %rdi</code>	compute $n * i$
<code>leaq</code>	<code>(%rsi, %rdi, 4), %rax</code>	compute $x_A + 4(n * i)$
<code>movl</code>	<code>(%rax, %rcx, 4), %eax</code>	Read from $M[x_A + 4(n * i) + 4j]$
<code>ret</code>		

还是矩阵  $A$  和矩阵  $B$  内积的例子，如果采用变长数组来存储矩阵  $A$  和矩阵  $B$ ，与定长数组相比  $C$  代码的实现几乎没有差别。

```
int var_mat(long n, int A[n][n], int B[n][n], long i, long k)
{
    long j;
    int result = 0;
    for (j=0; j <N; j++){
        result += A[i][j] * B[j][k];
    }
    return result;
}
```

不过对比二者的汇编代码，可以发现编译器采用了不同的优化方法。

```
.L7:(fixed)      .L24:(variable)
movl    (%rdi), %edx      movl    (%rsi, %rdx, 4), %r8x
imull   (%rcx), %edx      imull   (%rcx), %r8d
addl    %edx, %eax        addl    %r8d, %eax
addq    $4, %rdi          addq    $1, %rdx
addq    $64, %rcx         addq    %r9, %rcx
cmpq    (%rsi), %rcx      cmpq    %rdi, %rdx
jne .L7              jne .L24
rep; ret
```

无论是采用何种优化方法, 都显著的提高了程序的性能。