

SWE2001: System Program

0x07: Machine-Level Representation of Programs: Procedures

Hojoon Lee

Systems Security Lab @ SKKU

Table of Contents

▶ **Arrays**

- One-dimensional
- Multi-dimensional (nested)
- Multi-level

▶ **Structures**

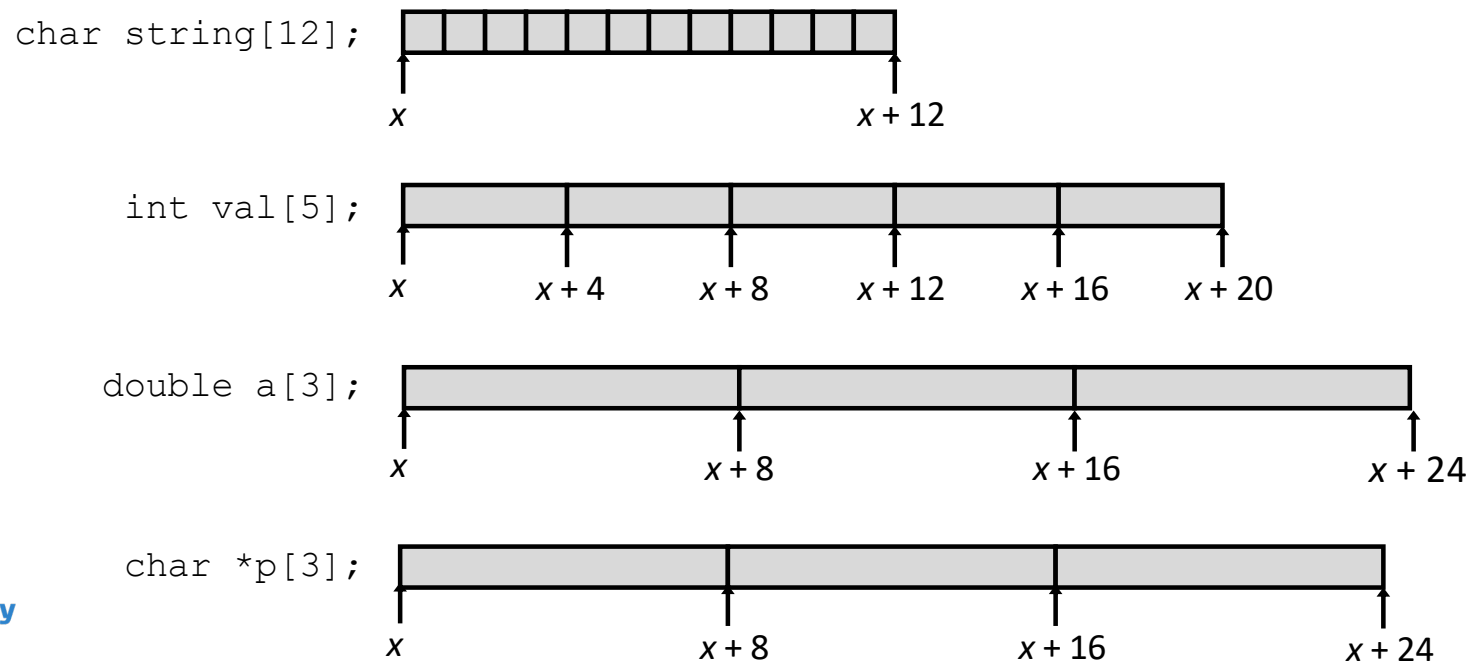
- Allocation
- Access
- Alignment

Array Allocation

Basic Principle

$T \ A[L];$

- Array of data type T and length L
- Contiguously allocated region of $L * \text{sizeof}(T)$ bytes in memory

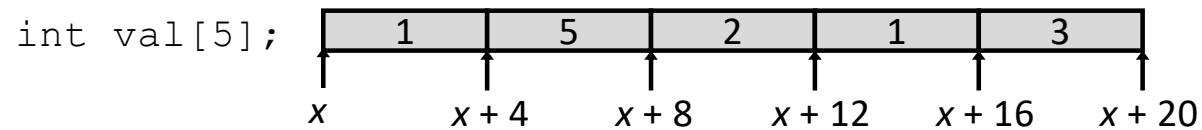


Array Access

► Basic Principle

T **A**[L];

- Array of data type T and length L
- Identifier **A** can be used as a pointer to array element 0: Type T^*



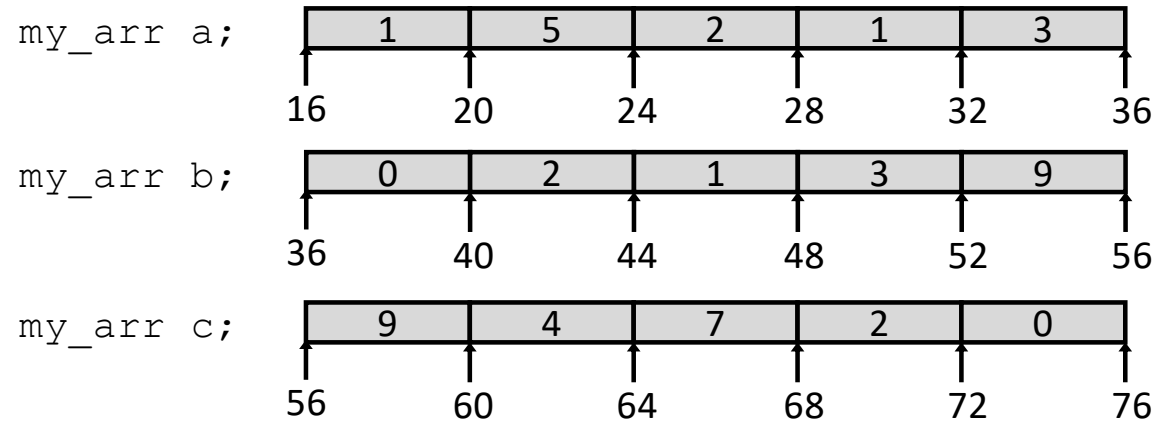
► Reference

Reference	Type	Value
<code>val[4]</code>	<code>int</code>	3
<code>val</code>	<code>int *</code>	x
<code>val+1</code>	<code>int *</code>	$x+4$
<code>&val[2]</code>	<code>int *</code>	$x+8$
<code>val[5]</code>	<code>int</code>	??
<code>*(val+1)</code>	<code>int</code>	5
<code>val + i</code>	<code>int *</code>	$x+4i$

Array Example

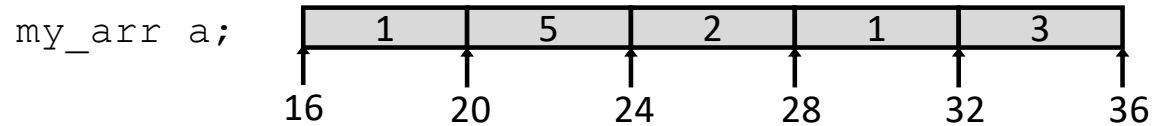
```
#define ZLEN 5
typedef int my_arr[ZLEN];

my_arr a = { 1, 5, 2, 1, 3 };
my_arr b = { 0, 2, 1, 3, 9 };
my_arr c = { 9, 4, 7, 2, 0 };
```



- ▶ Declaration “my_arr a” equivalent to “int a[5]”
- ▶ Example arrays were allocated in successive 20 byte blocks
 - Not guaranteed to happen in general

Array Accessing Example



```
int get_digit  
  (my_arr z, int digit)  
{  
  return z[digit];  
}
```

IA32

```
# %rdi = z  
# %rsi = digit  
movl (%rdi,%rsi,4), %eax # z[digit]
```

- Register `%rdi` contains starting address of array
- Register `%rsi` contains array index
- Desired digit at $\text{\%rdi} + 4 * \text{\%rsi}$
- Use memory reference $(\text{\%rdi}, \text{\%rsi}, 4)$

Array Loop Example

```
void zincr(my_arr z) {  
    size_t i;  
    for (i = 0; i < ZLEN; i++)  
        z[i]++;  
}
```

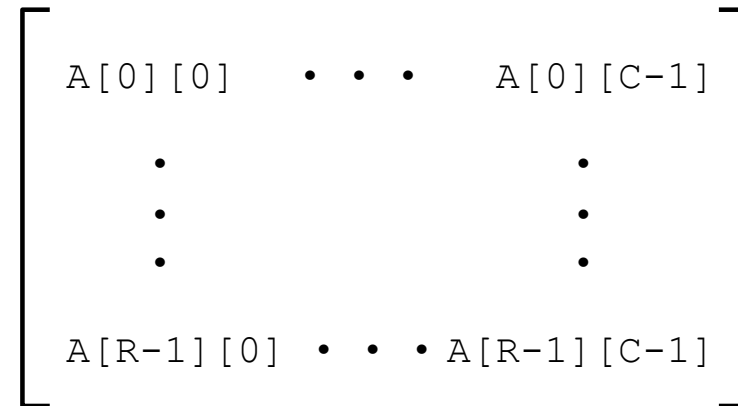
```
# %rdi = z  
movl    $0, %eax           # i = 0  
jmp     .L3                # goto middle  
.L4:                          # loop:  
    addl    $1, (%rdi,%rax,4) # z[i]++  
    addq    $1, %rax         # i++  
.L3:                          # middle  
    cmpq    $4, %rax         # i:4  
    jbe     .L4              # if <=, goto loop  
rep; ret
```

Multidimensional (Nested) Arrays

Declaration

`T A[R][C];`

- 2D array of data type T
- R rows, C columns
- Type T element requires K bytes



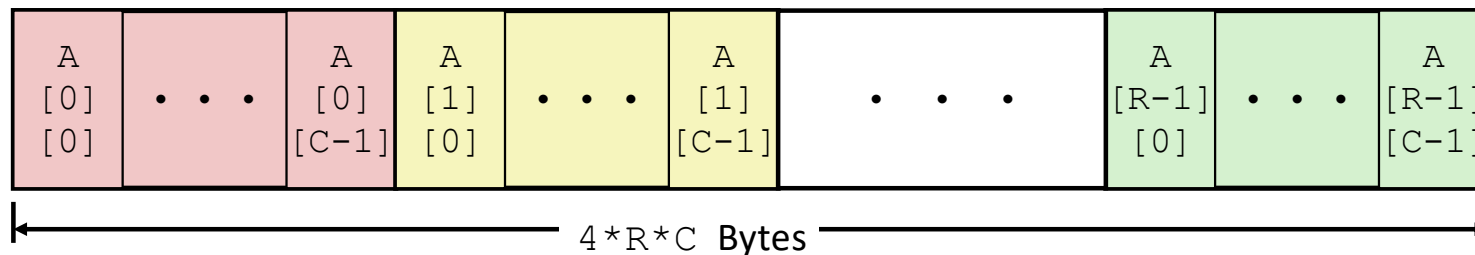
Array Size

- $R * C * K$ bytes

Arrangement

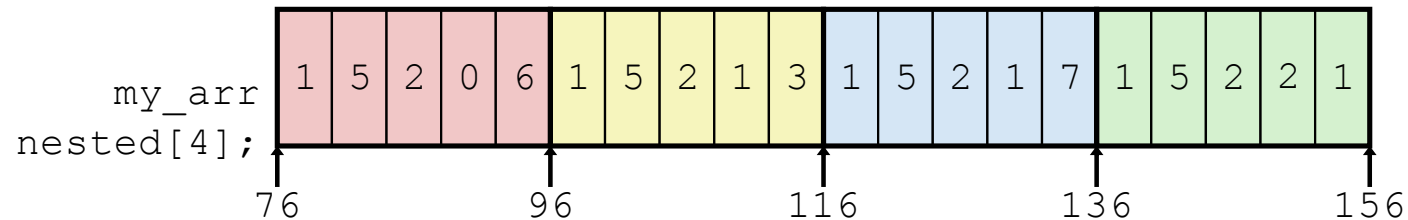
- Row-Major Ordering

`int A[R][C];`



Nested Array Example

```
#define PCOUNT 4
my_arr nested[PCOUNT] =
    {{1, 5, 2, 0, 6},
     {1, 5, 2, 1, 3 },
     {1, 5, 2, 1, 7 },
     {1, 5, 2, 2, 1 }};
```



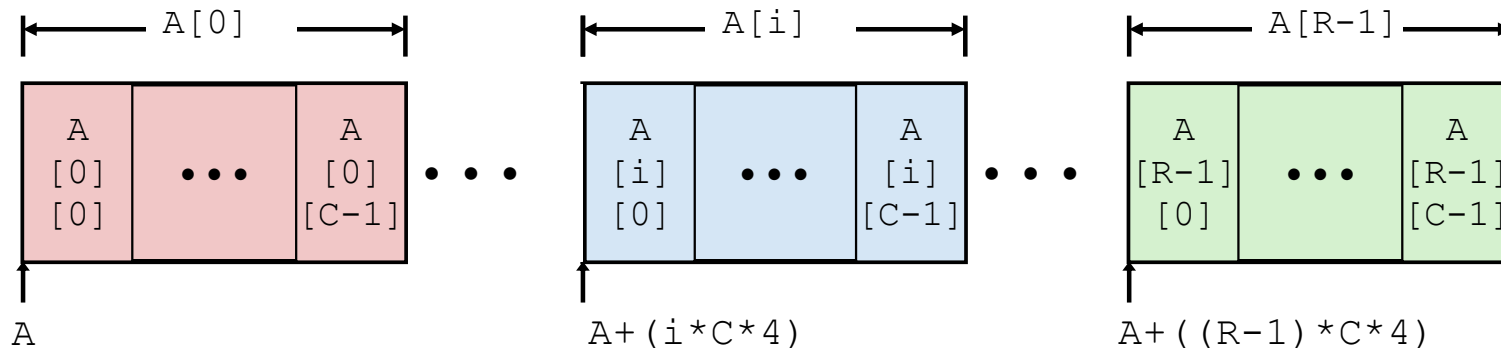
- ▶ “`my_arr nested[4]`” equivalent to “`int nested[4][5]`”
 - Variable **nested**: array of 4 elements, allocated contiguously
 - Each element is an array of 5 **int**’s, allocated contiguously
- ▶ “Row-Major” ordering of all elements in memory

Nested Array Row Access

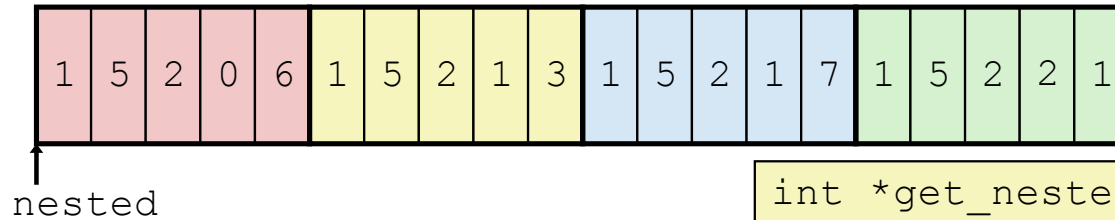
► Row Vectors

- $\mathbf{A}[i]$ is array of C elements
- Each element of type T requires K bytes
- Starting address $\mathbf{A} + i * (C * K)$

```
int A[R][C];
```



Nested Array Row Access Code



```
int *get_nested_zip(int
index)
{
    return nested[index];
}
```

```
# %rdi = index
leaq (%rdi,%rdi,4),%rax # 5 * index
leaq nested(,%rax,4),%rax # nested + (20 * index)
```

► Row Vector

- **nested[index]** is array of 5 int's
- Starting address **nested+20*index**

► Machine Code

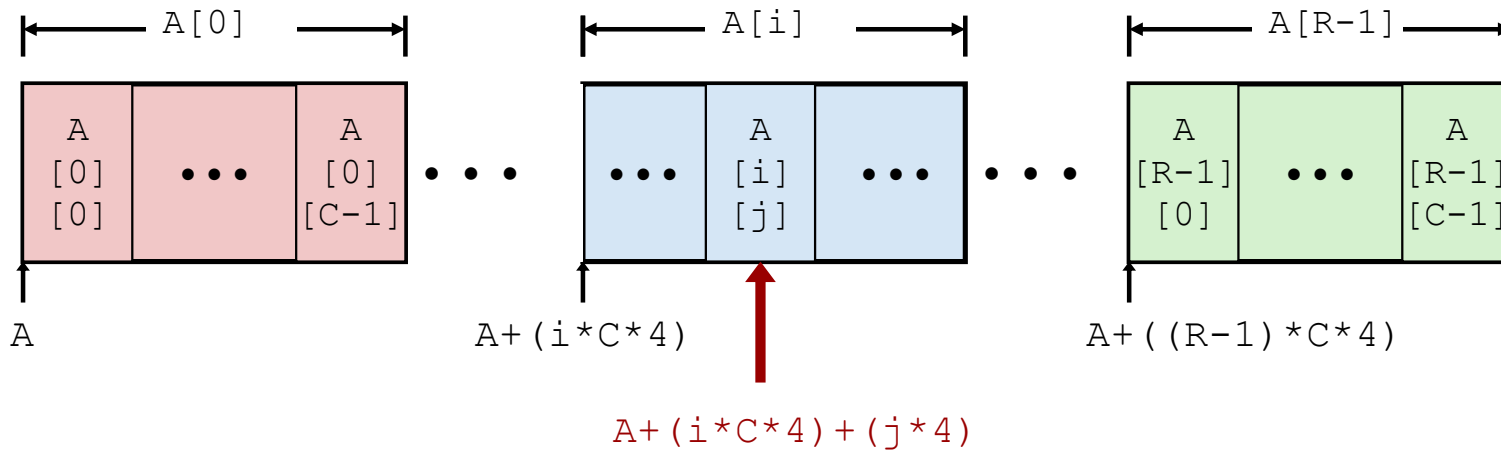
- Computes and returns address
- Compute as **nested + 4*(index+4*index)**

Nested Array Element Access

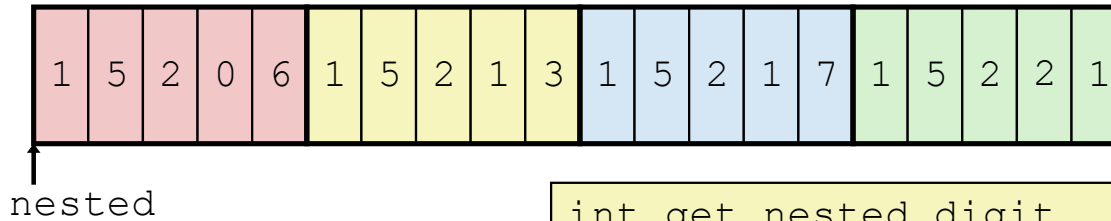
► Array Elements

- $A[i][j]$ is element of type T , which requires K bytes
- Address $A + i * (C * K) + j * K = A + (i * C + j) * K$

```
int A[R][C];
```



Nested Array Element Access Code



```
int get_nested_digit  
    (int index, int dig)  
{  
    return nested[index][dig];  
}
```

```
leaq    (%rdi,%rdi,4), %rax    # 5*index  
addl    %rax, %rsi             # 5*index+dig  
movl    nested(,%rsi,4), %eax  # M[nested + 4*(5*index+dig)]
```

▶ Array Elements

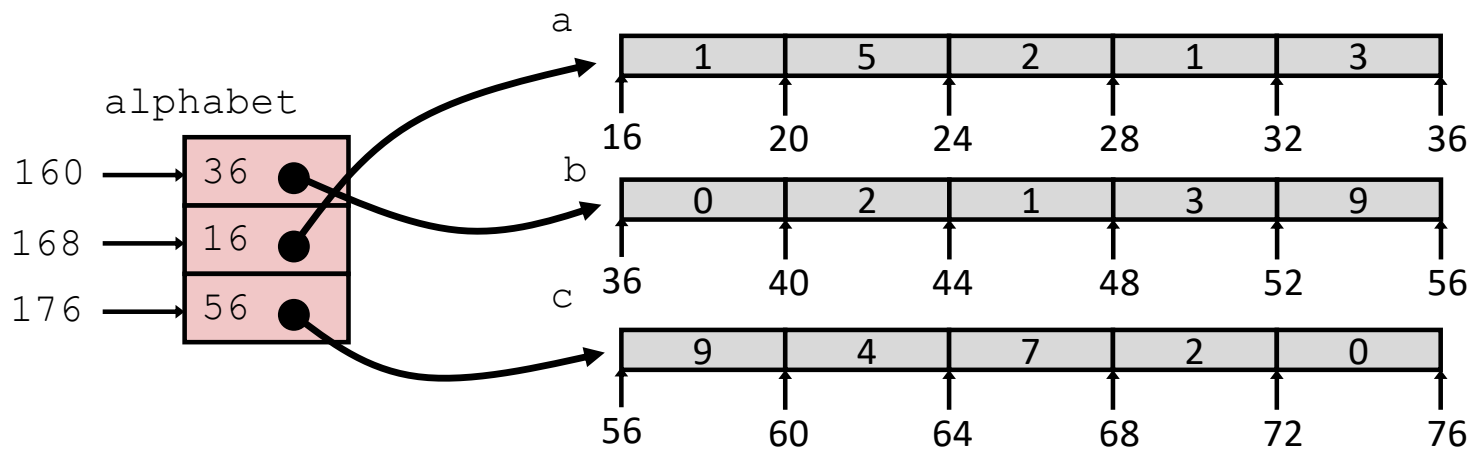
- `nested[index][dig]` is `int`
- Address: `nested + 20*index + 4*dig`
 - $= \text{nested} + 4*(5*\text{index} + \text{dig})$

Multi-Level Array Example

```
my_arr a = { 1, 5, 2, 1, 3 };  
my_arr b = { 0, 2, 1, 3, 9 };  
my_arr c = { 9, 4, 7, 2, 0 };
```

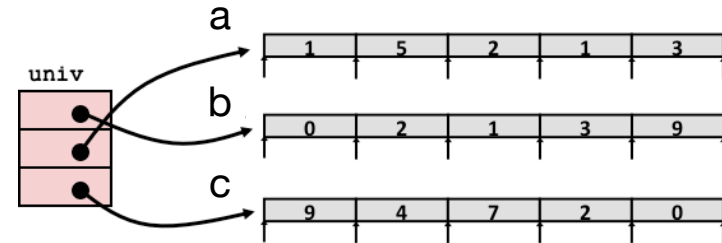
```
#define UCOUNT 3  
int *alphabet[UCOUNT] = {a, b, c};
```

- ▶ Variable alphabet denotes array of 3 elements
- ▶ Each element is a pointer
 - 8 bytes
- ▶ Each pointer points to array of int's



Element Access in Multi-Level Array

```
int get_alphabet_digit
(size_t index, size_t digit)
{
    return alphabet[index][digit];
}
```



```
salq    $2, %rsi                # 4*digit
addq     alphabet(,%rdi,8), %rsi # p = alphabet[index] + 4*digit
movl     (%rsi), %eax            # return *p
ret
```

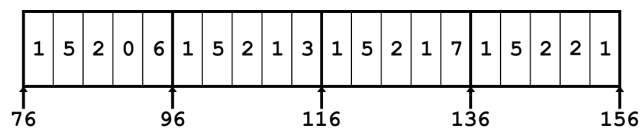
► Computation

- Element access $\text{Mem}[\text{Mem}[\text{alphabet} + 8 * \text{index}] + 4 * \text{digit}]$
- Must do two memory reads
 - First get pointer to row array
 - Then access element within array

Array Element Accesses

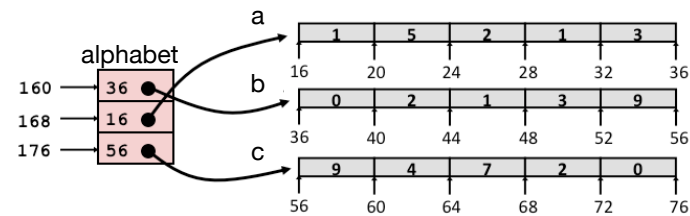
Nested array

```
int get_nested_digit
(size_t index, size_t digit)
{
    return nested[index][digit];
}
```



Multi-level array

```
int get_alphabet_digit
(size_t index, size_t digit)
{
    return alphabet[index][digit];
}
```



Accesses looks similar in C, but address computations very different:

$\text{Mem}[\text{nested} + 20 * \text{index} + 4 * \text{digit}]$

$\text{Mem}[\text{Mem}[\text{alphabet} + 8 * \text{index}] + 4 * \text{digit}]$

N X N Matrix Code

- ▶ **Fixed dimensions**
 - Know value of N at compile time
- ▶ **Variable dimensions, explicit indexing**
 - Traditional way to implement dynamic arrays
- ▶ **Variable dimensions, implicit indexing**
 - Now supported by gcc

```
#define N 16
typedef int fix_matrix[N][N];
/* Get element a[i][j] */
int fix_ele(fix_matrix a,
           size_t i, size_t j)
{
    return a[i][j];
}
```

```
#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element a[i][j] */
int vec_ele(size_t n, int *a,
           size_t i, size_t j)
{
    return a[IDX(n,i,j)];
}
```

```
/* Get element a[i][j] */
int var_ele(size_t n, int a[n][n],
           size_t i, size_t j) {
    return a[i][j];
}
```

16 X 16 Matrix Access

► Array Elements

- Address $A + i * (C * K) + j * K$
- $C = 16, K = 4$

```
/* Get element a[i][j] */  
int fix_ele(fix_matrix a, size_t i, size_t j) {  
    return a[i][j];  
}
```

```
# a in %rdi, i in %rsi, j in %rdx  
salq    $6, %rsi                # 64*i  
addq    %rsi, %rdi               # a + 64*i  
movl    (%rdi,%rdx,4), %eax      # M[a + 64*i + 4*j]  
ret
```

n X n Matrix Access

- ▶ Array Elements
 - Address $A + i * (C * K) + j * K$
 - $C = n, K = 4$
 - Must perform integer multiplication

```
/* Get element a[i][j] */  
int var_ele(size_t n, int a[n][n], size_t i, size_t j)  
{  
    return a[i][j];  
}
```

```
# n in %rdi, a in %rsi, i in %rdx, j in %rcx  
imulq    %rdx, %rdi          # n*i  
leaq     (%rsi,%rdi,4), %rax  # a + 4*n*i  
movl     (%rax,%rcx,4), %eax  # a + 4*n*i + 4*j  
ret
```