

# BBM 201

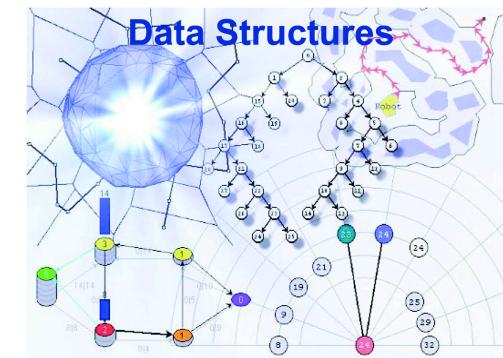
# DATA STRUCTURES

---

Lecture 3:  
Representation of Multidimensional Arrays



2019-2020 Fall



# What is an Array?

- An array is a fixed size sequential collection of elements of identical types.
- A multidimensional array is treated as an array of arrays.
  - Let  $a$  be a  $k$ -dimensional array; the elements of  $a$  can be accessed using the following syntax:

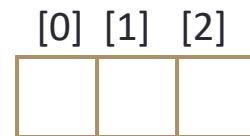
$a[i_1][i_2] \dots [i_k]$

The following loop stores 0 into each location in two dimensional array  $A$ :

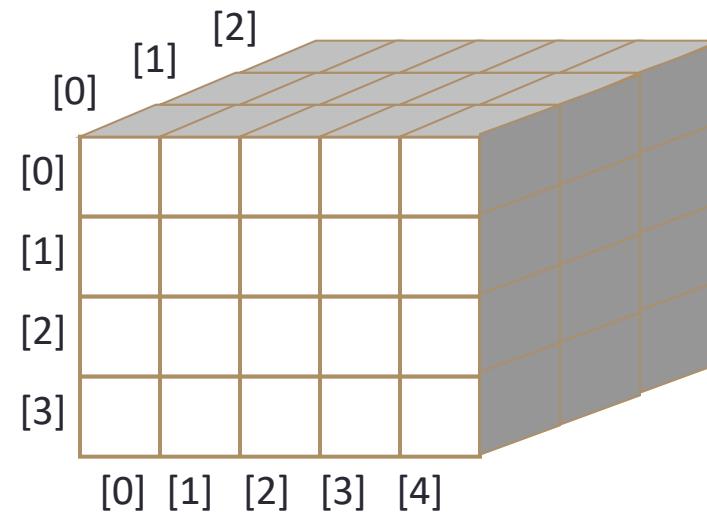
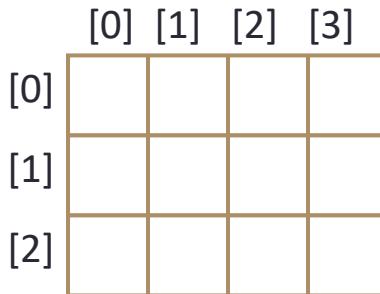
```
int row, column;
int A[3][4];
for (row = 0; row < 3; row++)
{
    for (column = 0; column < 4; column++)
    {
        A[row][column] = 0;
    }
}
```

# Definition of a Multidimensional Array

- One-dimensional arrays are linear containers.



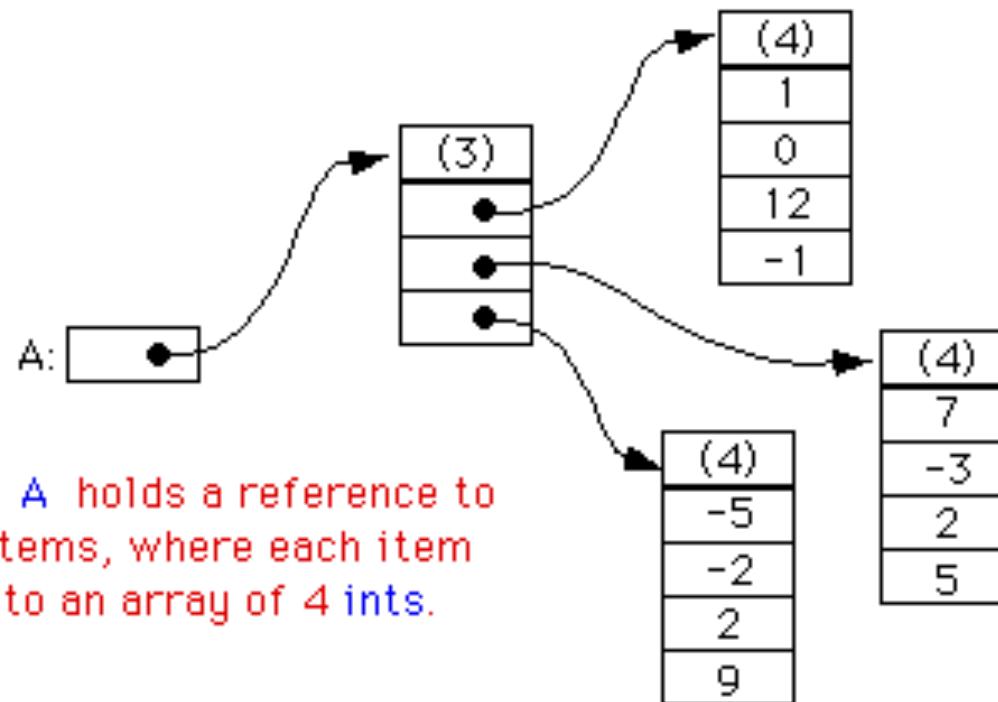
## Multi-dimensional Arrays



# Two-Dimensional Array

A:	1	0	12	-1
	7	-3	2	5
	-5	-2	2	9

If you create an array `A = new int[3][4]`, you should think of it as a "matrix" with 3 rows and 4 columns.



But in reality, `A` holds a reference to an array of 3 items, where each item is a reference to an array of 4 ints.

# Storage Allocation

The storage arrangement shown in this example uses the array subscript, also called the array indices.

Array declaration: int a[3][4];

Array elements:

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

# Array size

- In a matrix which is defined as

$a[upper_0] [upper_1] \dots [upper_{n-1}]$ ,

the number of items is:

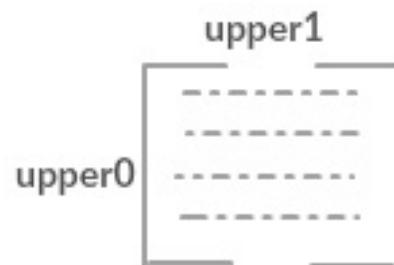
$$\prod_{i=0}^{n-1} upper^i$$

Example: What is the number of items in  $a[20][20][1]$ ?

# Memory Storage

- There are two types of placement for multidimensional arrays in memory:
  - Row major ordering
  - Column major ordering

Example: In an array which is defined as  $A[\text{upper}_0][\text{upper}_1]$ , if the memory address of  $A[0][0]$  is  $\alpha$ , then what is the memory address of  $A[i][0]$  (according to row major ordering)?

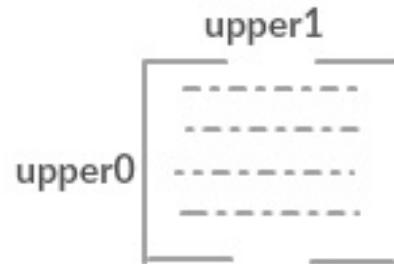


# Memory Storage

- There are two types of placement for multidimensional arrays in memory:
  - Row major ordering
  - Column major ordering

Example: In an array which is defined as  $A[\text{upper}_0][\text{upper}_1]$ , if the memory address of  $A[0][0]$  is  $\alpha$ , then what is the memory address of  $A[i][0]$  (according to row major ordering)?

$$\alpha + i * \text{upper}_1$$

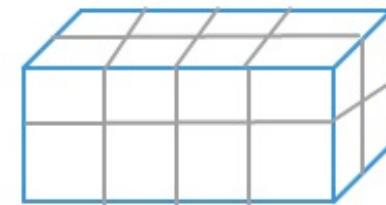


# Memory Storage

- For a three-dimensional array  $A[\text{upper}_0][\text{upper}_1][\text{upper}_2]$  what is the memory storage like?

- Example: char  $y[2][2][4]$

which slice?      which row?      which column?



- What is the memory address of  $y[1][1][3]$  if the memory address of  $y[0][0][0]$   $\alpha$ ?

# Memory Storage

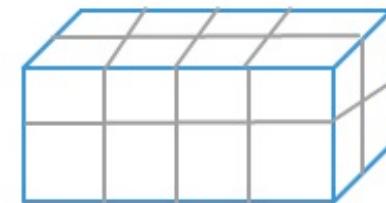
- For a three-dimensional array  $A[\text{upper}_0][\text{upper}_1][\text{upper}_2]$  what is the memory storage like?

- Example: char  $y[2][2][4]$

which slice?

which row?

which column?



- What is the memory address of  $y[1][1][3]$  if the memory address of  $y[0][0][0]$   $\alpha$ ?  $\alpha + 1 * 2 * 4 + 1 * 4 + 3$

# Memory Storage

The memory address of  $a[i][0][0]$  is:

$$\alpha + i * \text{upper}_1 * \text{upper}_2$$

if the memory address of  $a[0][0][0]$  is  $\alpha$ . Therefore, the memory address of  $a[i][j][k]$  becomes:

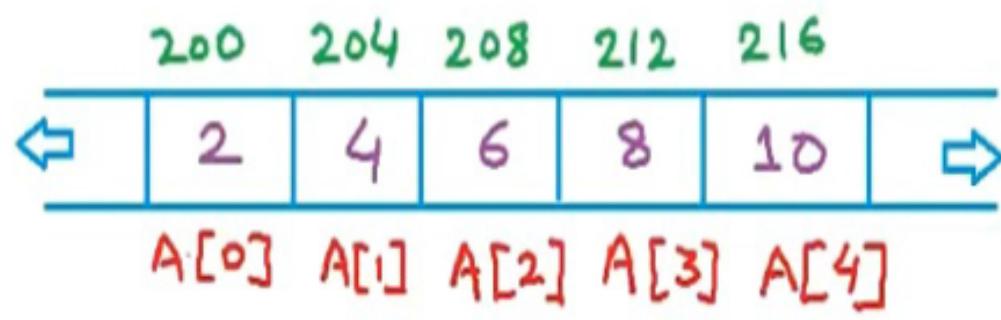
$$\alpha + i * \text{upper}_1 * \text{upper}_2 + j * \text{upper}_2 + k$$

The memory address of  $a[i_0] [i_1] [i_2] \dots [i_{n-1}]$  is:

$$\alpha + \sum_{j=0}^{n-1} i_j a_j \left\{ \begin{array}{l} a_j = \prod_{k=j+1}^{n-1} \text{upper}_k \quad 0 \leq j \leq n-1 \\ a_{n-1} = 1 \end{array} \right.$$

# Pointers and Multi-dimensional Arrays

int A[5]



int \*P = A;

Print P // 200

Print \*P // 2

Print \*(P+2) // 6

int A[5]



int \*p = A;

Print A // 200

Print \*A // 2

Print \*(A+2) // 6

\*(A+i) is same as A[i]

(A+i) is same as &A[i]

int A[5]



int \*p = A;

Print A // 200

Print \*A // 2

Print \*(A+2) // 6

\*(A+i) is same as A[i]

P = A; ✓

(A+i) is same as &A[i]

A = P; X

int A[5]

A[0] }  
A[1]  
:  
;

	200	204	208	212	216	
	2	4	6	8	10	
	A[0]	A[1]	A[2]	A[3]	A[4]	

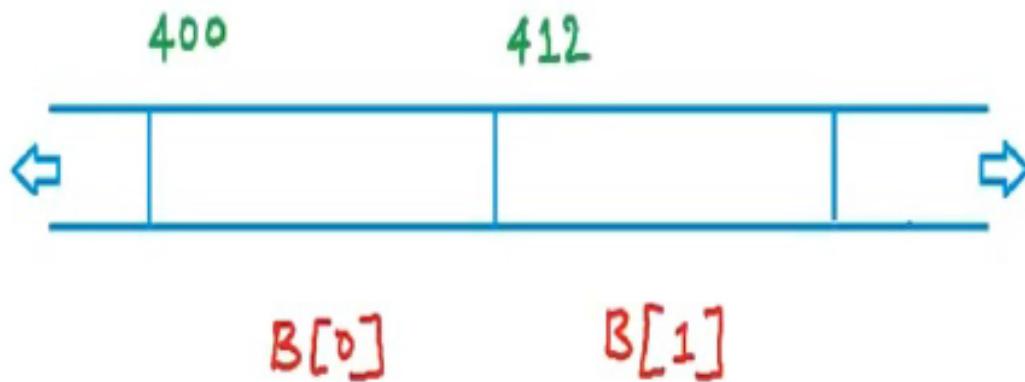
int B[2][3]

B[0] }  
B[1] }  
      → 1-D arrays  
      of 3 integers

`int B[2][3]`

`B[0]`  
`B[1]`

} → 1-D arrays  
of 3 integers

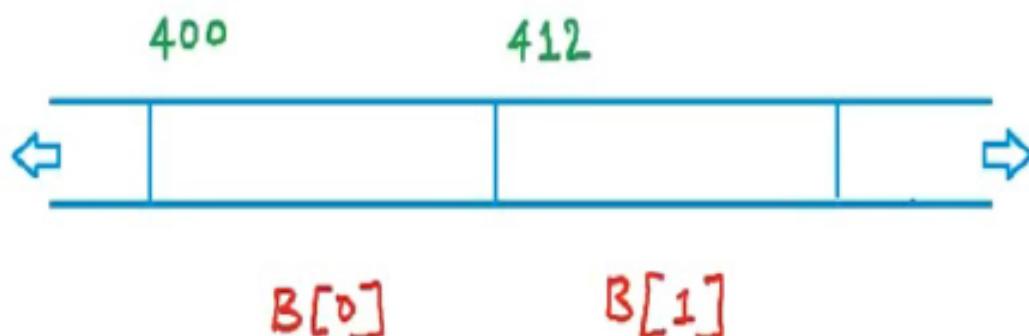


`int *p = B; X`

↓  
will return a pointer  
to 1-D array of 3 integers

`int B[2][3]`

`B[0]`      } → 1-D arrays  
`B[1]`      of 3 integers



`int *P = B; X`

↓  
will return a pointer  
to 1-D array of 3 integers

`int (*P)[3] = B; ✓`

```
int B[2][3]
```

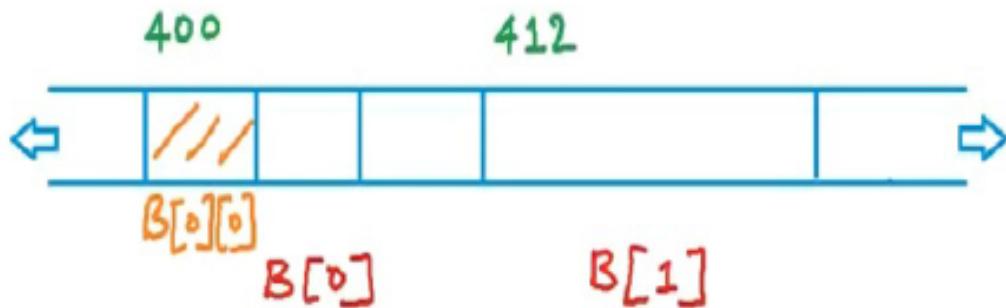
B[0] } → 1-D arrays  
B[1]    of 3 integers

```
int (*P)[3] = B;
```

↓  
will return a pointer  
to 1-D array of 3 integers

```
Print B or &B[0] // 400
```

```
Print *B or B[0] or &B[0][0] // 400
```



```
int B[2][3]
```

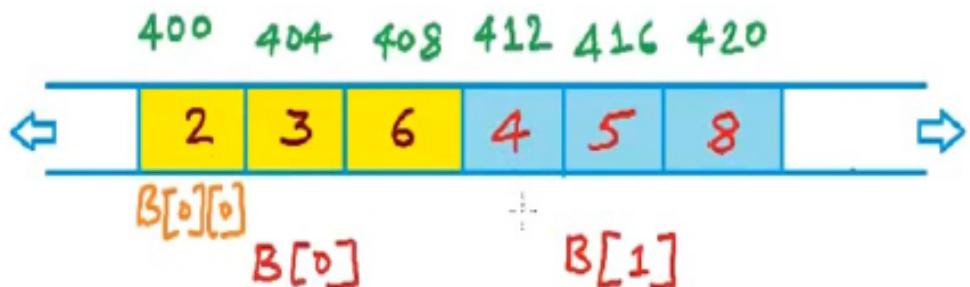
B[0] } → 1-D arrays  
B[1]    of 3 integers

```
int (*P)[3] = B;
```

```
Print B or &B[0] // 400
```

```
Print *B or B[0] or &B[0][0] // 400
```

Print B+1 // 400 + 12 = 412  
or  
&B[1]



```
int B[2][3]
```

B[0] } → 1-D arrays  
B[1] } of 3 integers

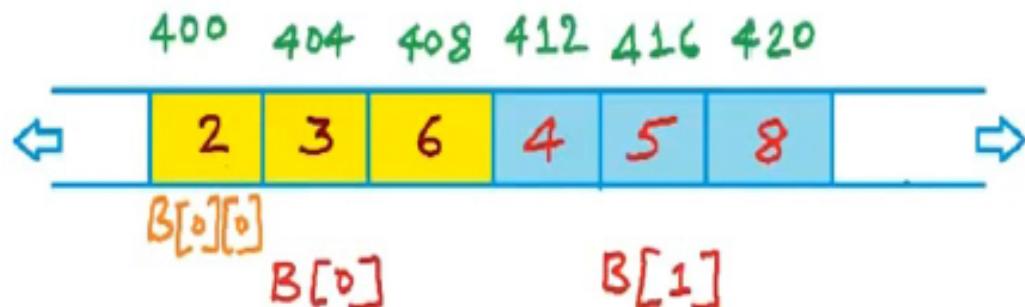
```
int (*P)[3] = B;
```

Print B or &B[0] // 400

Print \*B or B[0] or &B[0][0] // 400

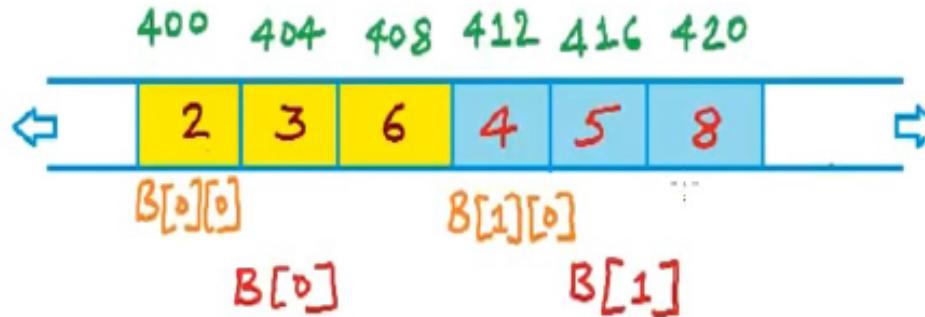
Print B+1 or &B[1] // 412

Print \*(B+1) or B[1] or &B[1][0] // 412



```
int B[2][3]
```

$B[0]$  } → 1-D arrays  
 $B[1]$     of 3 integers



```
int (*P)[3] = B;
```

Print B or &B[0] // 400

Print \*B or B[0] or &B[0][0] // 400

Print B+1 or &B[1] // 412

Print \*(B+1) or B[1] or &B[1][0] // 412

Print \*(B+1)+2 or B[1]+2 or &B[1][2] // 420

→ returning int \*

```
int B[2][3]
```

B[0] } → 1-D arrays  
B[1]      of 3 integers

```
int (*P)[3] = B;
```

Print B or &B[0] // 400

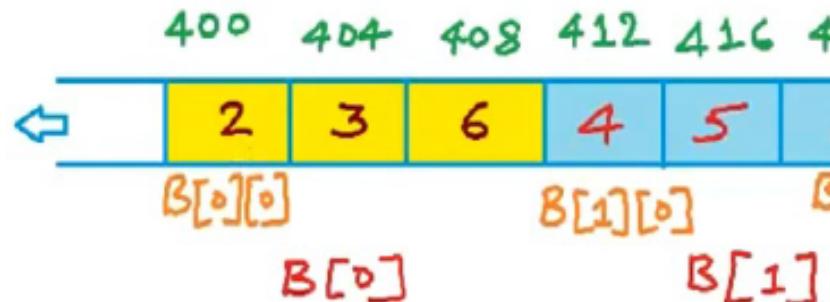
Print \*B or B[0] or &B[0][0] // 400

Print B+1 or &B[1] // 412

Print \*(B+1) or B[1] or &B[1][0] // 412

Print \*(B+1)+2 or B[1]+2 or &B[1][2] // 420

Print \*(\*B+1)  
B → int (\*)[3]  
B[0] → int \*



```
int B[2][3]
```

B[0] } → 1-D arrays  
B[1] of 3 integers

```
int (*P)[3] = B;
```

Print B or &B[0] // 400

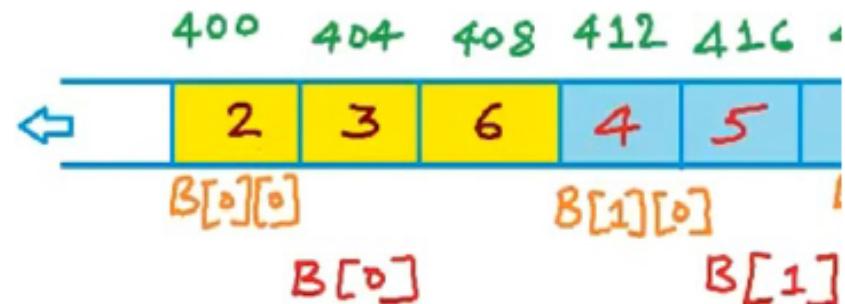
Print \*B or B[0] or &B[0][0] // 400

Print B+1 or &B[1] // 412

Print \*(B+1) or B[1] or &B[1][0] // 412

Print \*(B+1)+2 or B[1]+2 or &B[1][2] // 420

Print \*(\*B + 1)  
↓  
&B[0][1]



```
int B[2][3]
```

B[0]      } → 1-D arrays  
B[1]      of 3 integers

```
int (*P)[3] = B;
```

Print B or &B[0] // 400

Print \*B or B[0] or &B[0][0] // 400

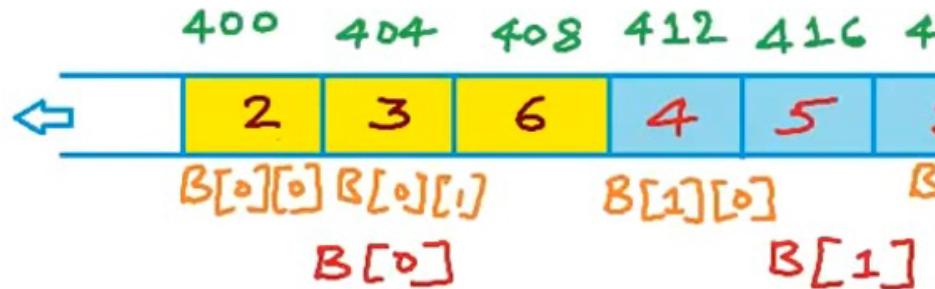
Print B+1 or &B[1] // 412

Print \*(B+1) or B[1] or &B[1][0] // 412

Print \*(B+1)+2 or B[1]+2 or &B[1][2] // 420

Print \*(\*B+1) // 3

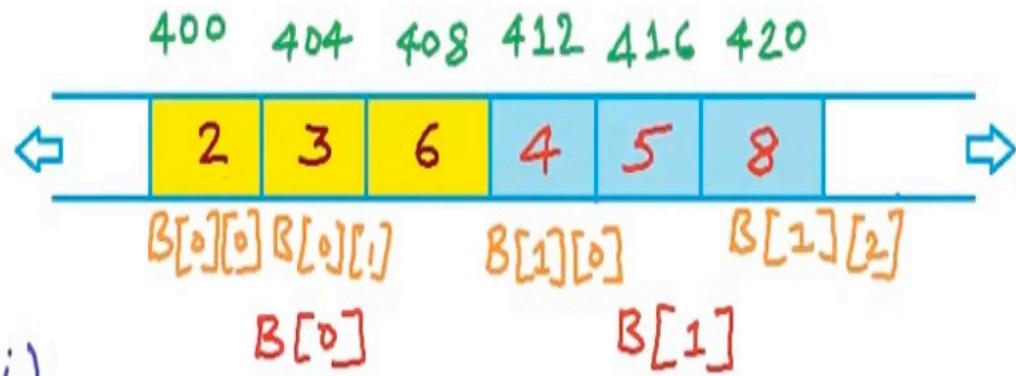
↓  
B[0][1]



int B[2][3]

For 2-D array

$$\begin{aligned}B[i][j] &= *(B[i]+j) \\&= *(*(B+i)+j)\end{aligned}$$



## Pointers and multi-dimensional arrays

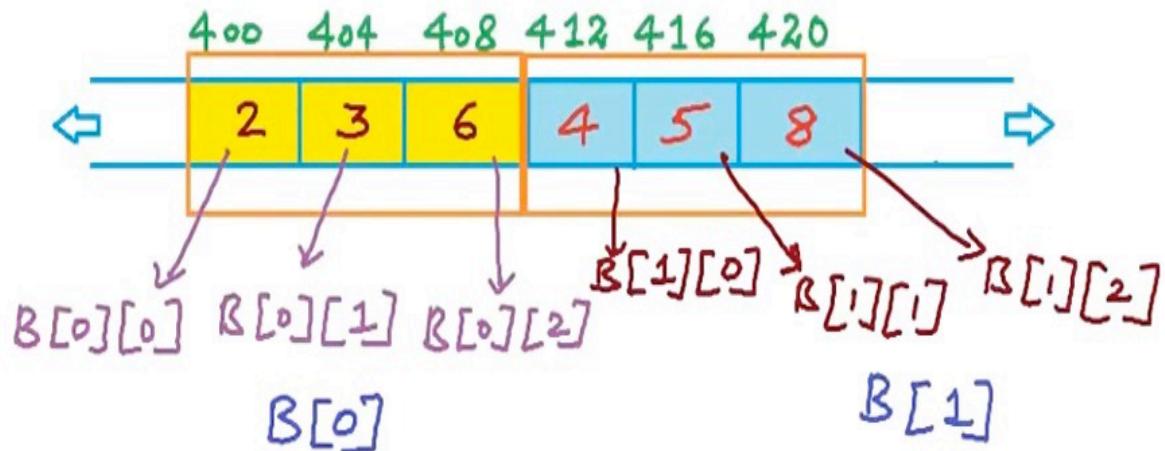
int B[2][3]

int (\*P)[3] = B; ✓

↓  
declaring

pointer to 1-D  
array of 3 integers

int \*P = B; X



## Pointers and multi-dimensional arrays

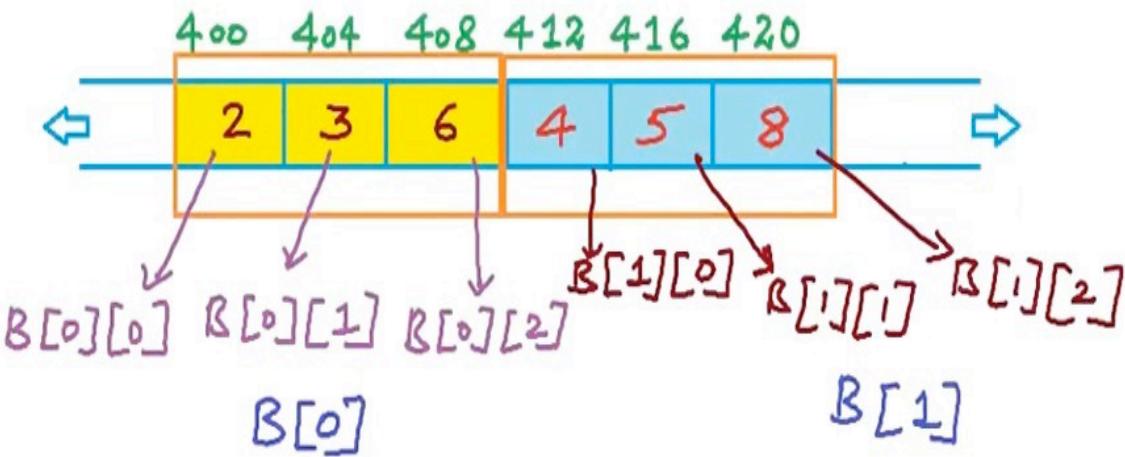
```
int B[2][3]
```

```
int (*P)[3] = B; ✓
```

```
Print B //400
```

```
Print *B //400
```

```
Print B[0] //400
```



## Pointers and multi-dimensional arrays

```
int B[2][3]
```

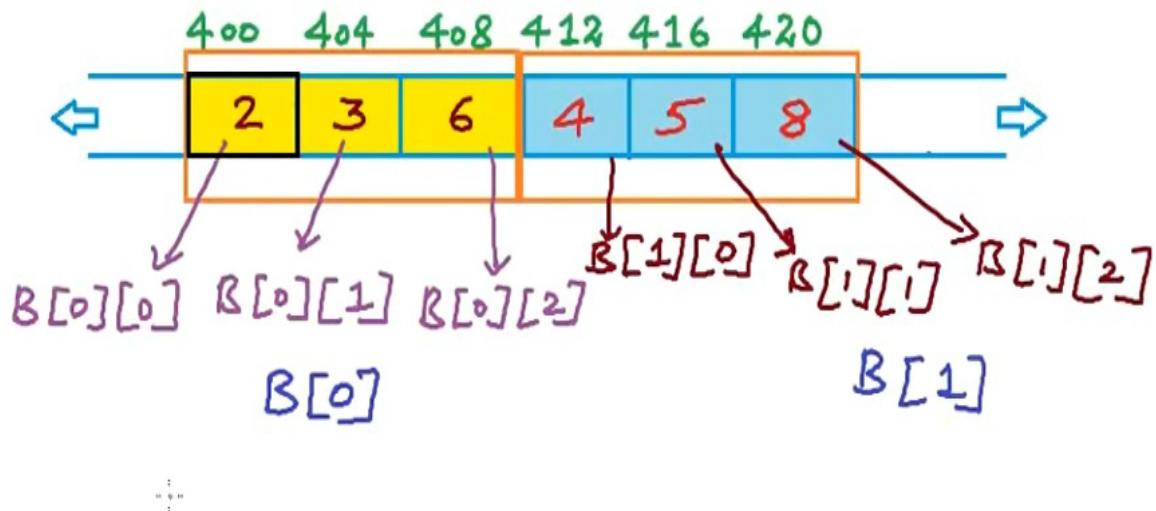
```
int (*P)[3] = B; ✓
```

```
Print B //400
```

```
Print *B //400 }
```

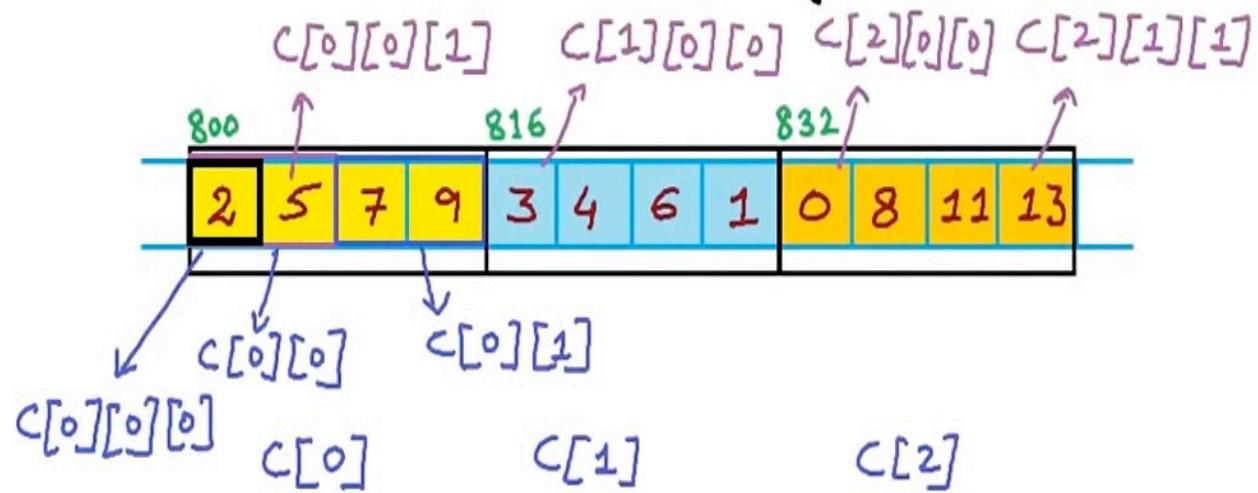
```
Print B[0] //400 }
```

```
Print &B[0][0] //400
```



## Pointers and multi-dimensional arrays

int  $c[3][2][2]$



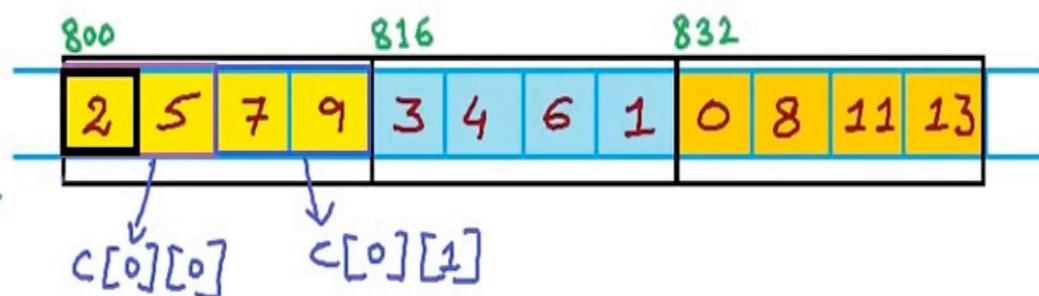
## Pointers and multi-dimensional arrays

```
int C[3][2][2]
```

```
int (*P)[2][2] = C; ✓
```

Print  $\hookrightarrow$   $\text{int}(*\text{[2][2]})$        $C[0]$        $C[1]$        $C[2]$   
 $// 800$

Print  $*C$  or  $C[0]$  or  $\&C[0][0]$        $// 800$   
 $\downarrow$   
 $\text{int}(*\text{[2]})$

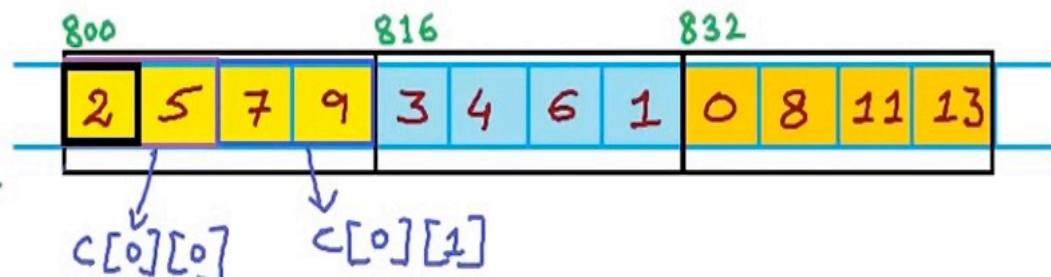


## Pointers and multi-dimensional arrays

int C[3][2][2]

int (\*P)[2][2] = C; ✓

Print C // 800  
    ↳ int (\*)[2][2]      C[0]      C[1]      C[2]  
Print \*C or C[0] or &C[0][0]



$$\begin{aligned} C[i][j][k] &= * (C[i][j] + k) = * (* (c[i] + j) + k) \\ &= * (* (* (c + i) + j) + k) \end{aligned}$$

## Pointers and multi-dimensional arrays

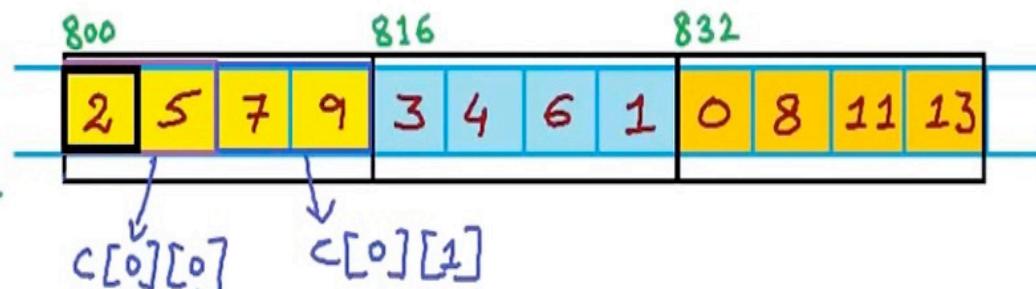
int C[3][2][2]

int (\*P)[2][2] = C; ✓

Print C // 800

Print \*C or C[0] or &C[0][0]

Print \*(C[0][1] + 1) or C[0][1][1] // 9



## Pointers and multi-dimensional arrays

int C[3][2][2]

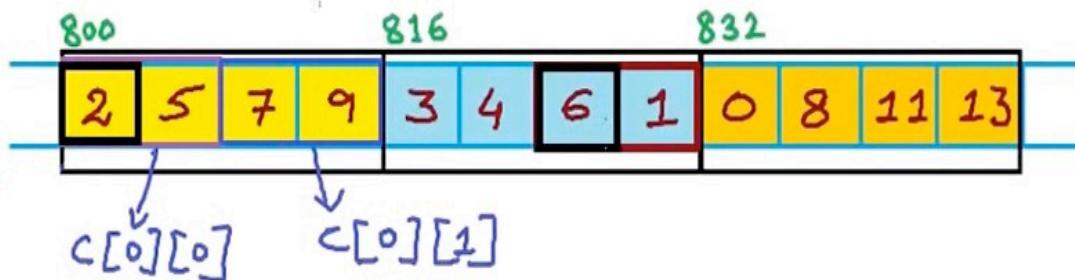
int (\*P)[2][2] = C; ✓

Print C // 800  
    ↳ int (\*)[2][2]      C[0]      C[1]      C[2]

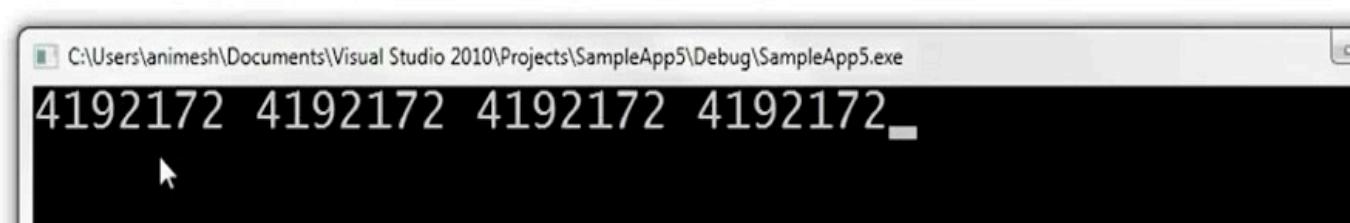
Print \*C or C[0] or &C[0][0] // 800

Print \*(C[0][1] + 1) or C[0][1][1] // 9

Print \*(C[1] + 1) or C[1][1] or &C[1][1][0] // 824



```
// Pointers and multi-dimensional arrays
#include<stdio.h>
int main()
{
    int C[3][2][2]={{ {2,5},{7,9} },
                    {{3,4},{6,1} },
                    {{0,8},{11,13}}};
    printf("%d %d %d %d", C, *C, C[0], &C[0][0]);
}
```



# References

- BBM 201 Notes by Mustafa Ege
- Lecture Videos: [www.mycodeschool.com/videos/pointers-and-arrays](http://www.mycodeschool.com/videos/pointers-and-arrays)