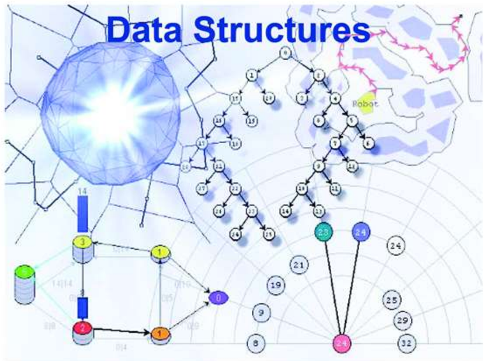
BBM 201
DATA STRUCTURES

Lecture 3:
Representation of Multidimensional Arrays



2015-2016 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:

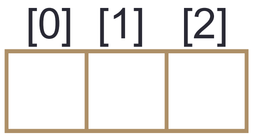
    $a [i_1] [i_2] \ldots [i_k]$

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

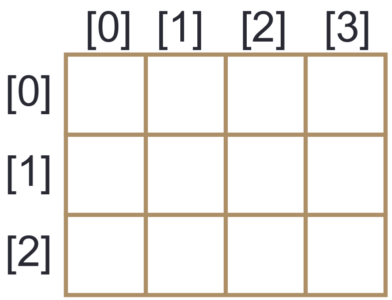
```
int row, column;
 int A[3][4];
 for (row = 0; row < 3; row++)
 {
  for (column = 0; column < 4; column++)
    {
          A[row][column] = 0;
    }
 }
```

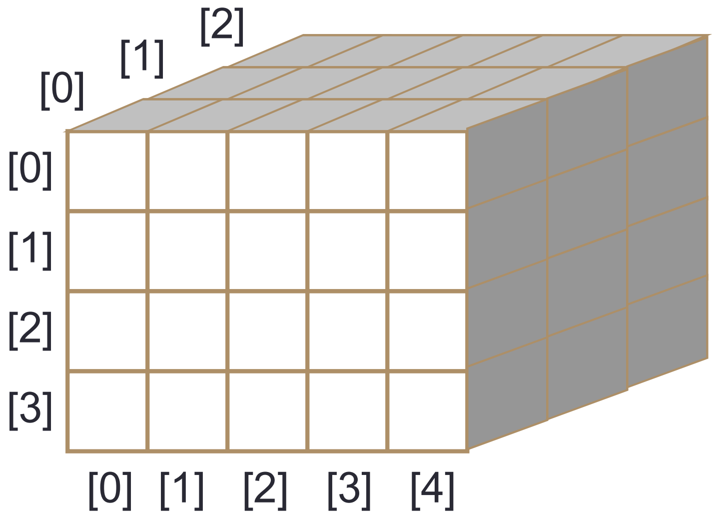
Definition of a Multidimensional Array

- **One-dimensional** arrays are linear containers.



Multi-dimensional Arrays



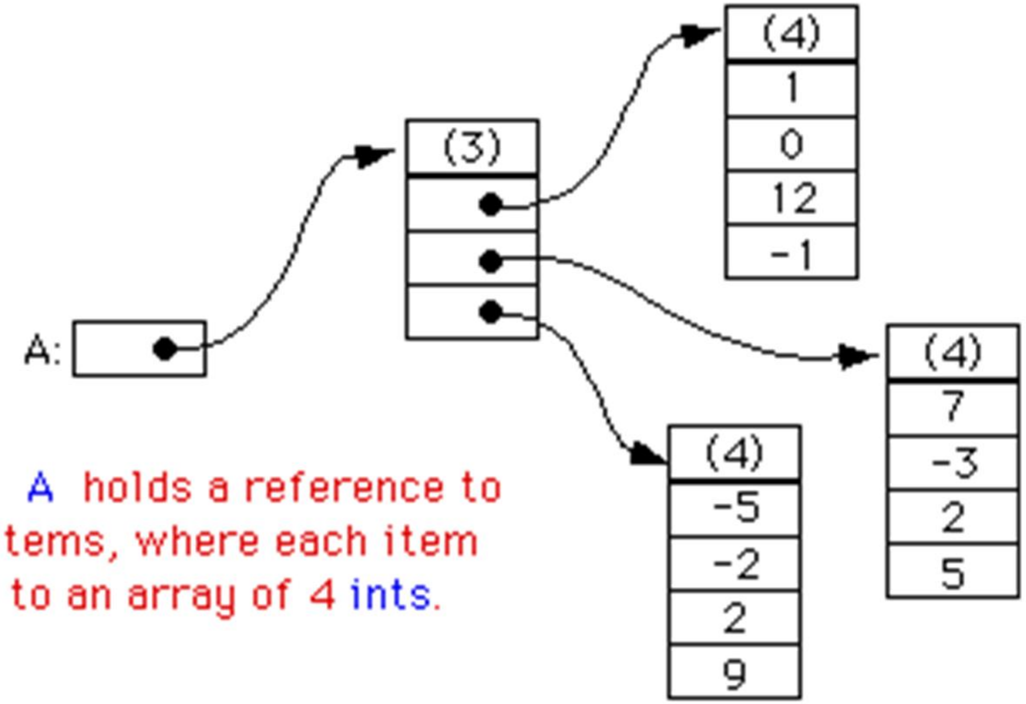


Two-Dimensional Array

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

A:

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



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[$upper_0$] [$upper_1$]…[$upper_{n-1}$],

  the number of items is:

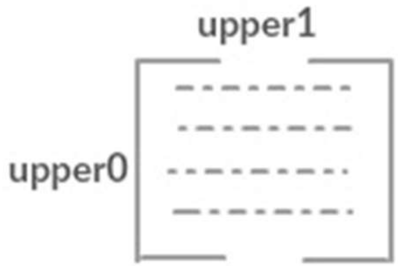
$$\prod_{i=0}^{n-1} upper^i$$

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 $A[upper_0][upper_1]$, if the memory address of A[0][0] is α, then what is the memory address of A[i][0] (according to row major ordering)?

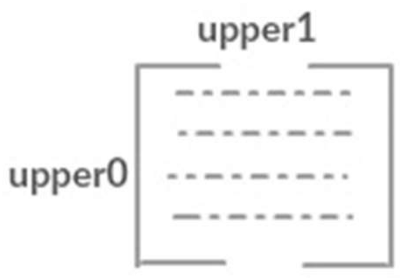


Memory Storage

- 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[$upper_0$][$upper_1$], if the memory address of A[0][0] is α, then what is the memory address of A[i][0] (according to row major ordering)?

$$\alpha + i * upper_1$$

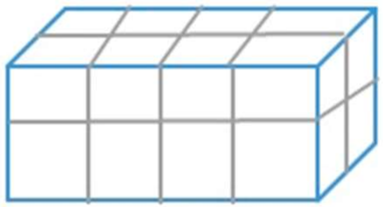


Memory Storage

- For a three-dimensional array A[$upper_0$][$upper_1$][$upper_2$] what is the memory storage like?

- <u>Example:</u> 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] α?

Memory Storage

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

$$\alpha + i * upper_1 * upper_2$$

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

$$\alpha + i * upper_1 * upper_2 + j * upper_2 + k$$

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

$$\alpha + \sum_{j=0}^{n-1} i_j a_j \begin{cases} a_j = \prod_{k=j+1}^{n-1} upper_k & 0 \le j \le n-1 \\ a_{n-1} = 1 \end{cases}$$

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

P = A; ✓

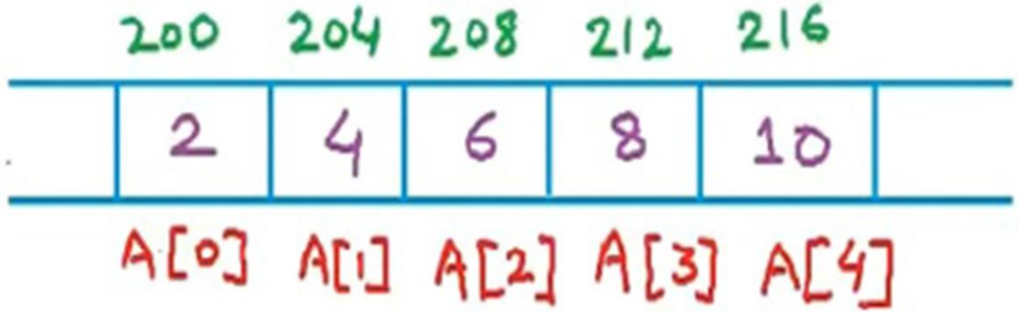
A = P; ✗

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

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

int A[5]

A[0]
A[1] } → int
⋮

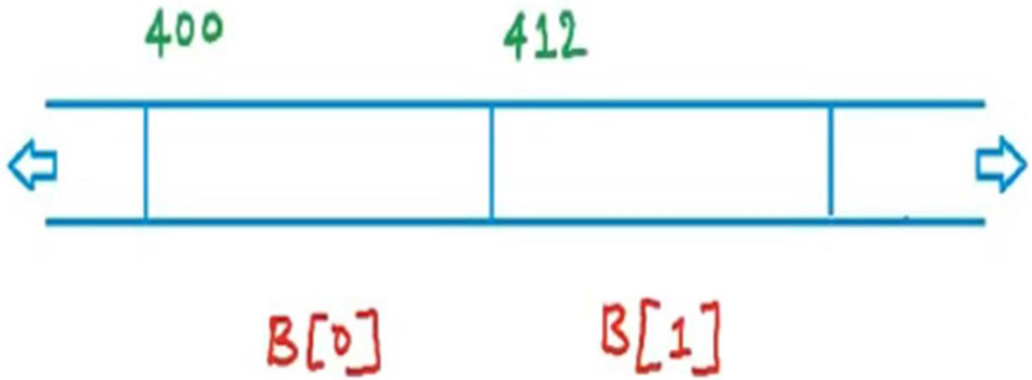


int B[2][3]

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; ✗

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

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

$\left.\begin{array}{l} B[0] \\ B[1] \end{array}\right\} \rightarrow$ 1-D arrays of 3 integers

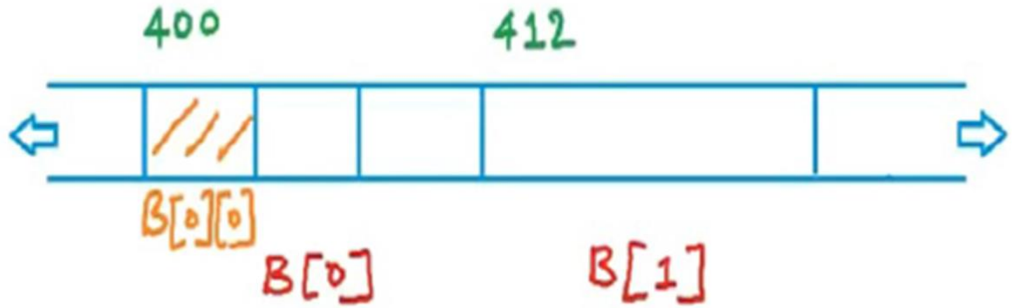

```c
int *p = B;   X
```

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

```c
int (*p)[3] = B;   ✓
```

int B[2][3]

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




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

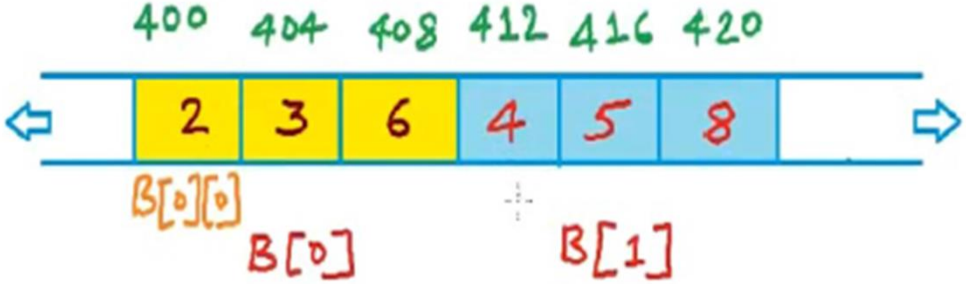
↓ will return a pointer to 1-D array of 3 integers

Print B or &B[0] // 400

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

int B[2][3]

B[0]
B[1] } → 1-D arrays 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] // 400+12 = 412

int B[2][3]

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

int (*P)[3] = B;

Print B or &B[0] // 400

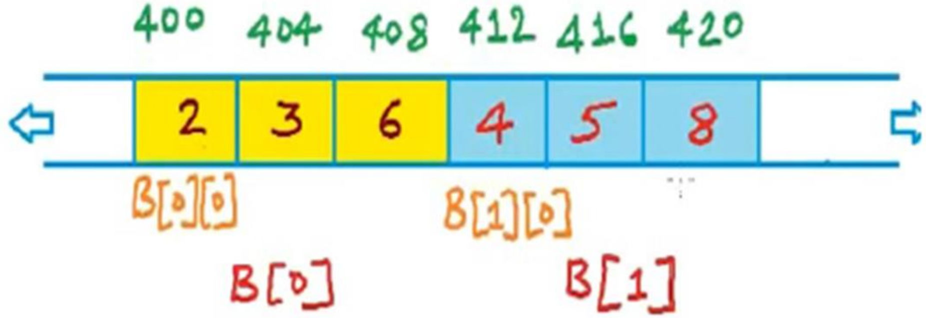
Print *B or B[0] or &B[0][0] // 400

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

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

int B[2][3]

B[0]
B[1]
→ 1-D arrays 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

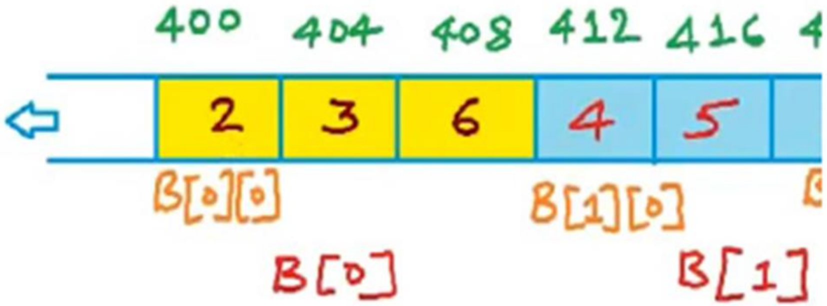
Print *(B+1) or B[1] or &B[1][0] // 412

→ returning int*

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

int B[2][3]

B[0]
B[1] } → 1-D arrays 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

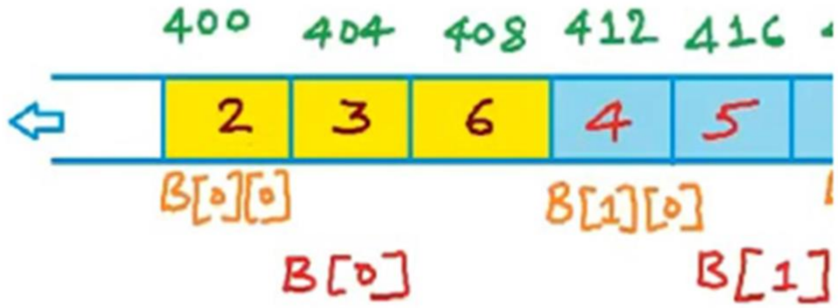
Print *(B+1)+2 or B[1]+2 or &B[1][2] // 420

Print $*(\underbrace{*B}_{B[0]}+1)$

B → int (*)[3]
B[0] → int *

int B[2][3]

B[0]
B[1]
} → 1-D arrays 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 $*(\underbrace{*B+1}_{\&B[0][1]})$

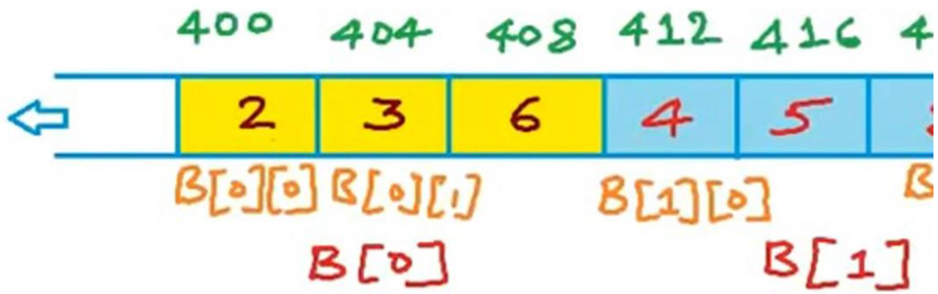
B → int (*)[3]

B[0] → int *

int B[2][3]

B[0]
B[1] } → 1-D arrays 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

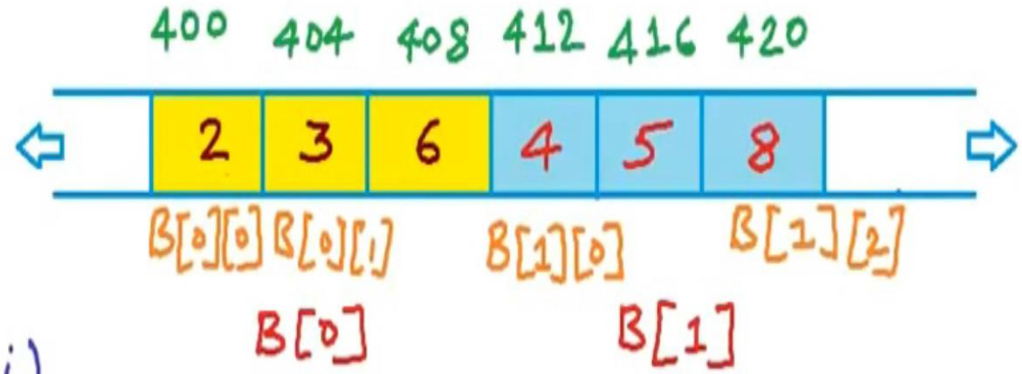
Print $\underbrace{*(*B+1)}_{B[0][1]}$ // 3

int B[2][3]

For 2-D array

$B[i][j] = *(B[i]+j)$

$= *(*(B+i)+j)$



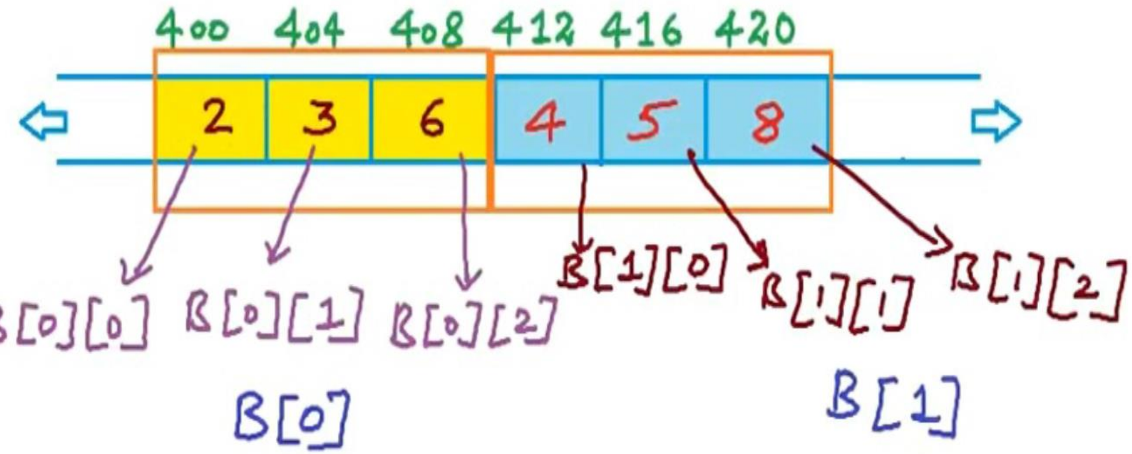
Pointers and multi-dimensional arrays

int B[2][3]

int (*P)[3] = B; ✓

declaring pointer to 1-D array of 3 integers

int *p = B; X

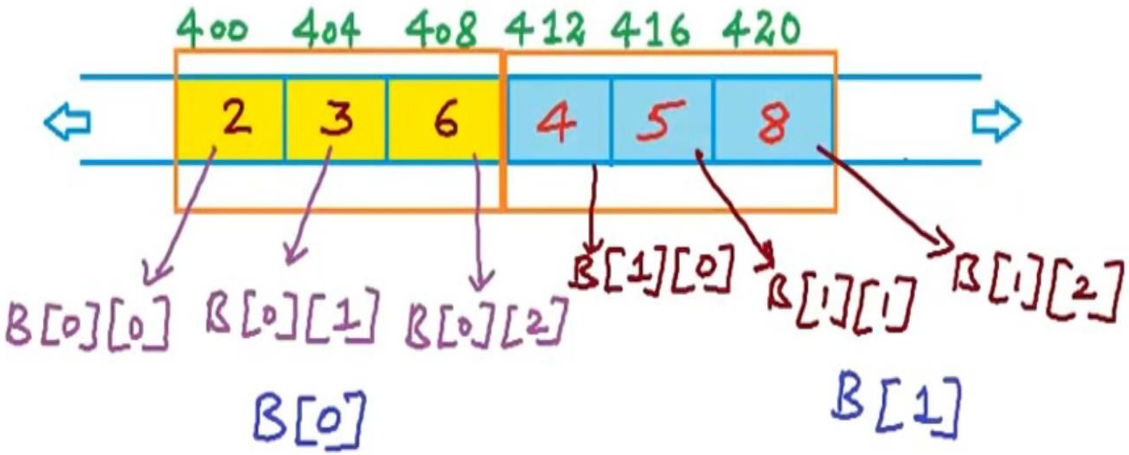


Pointers and multi-dimensional arrays

```
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]

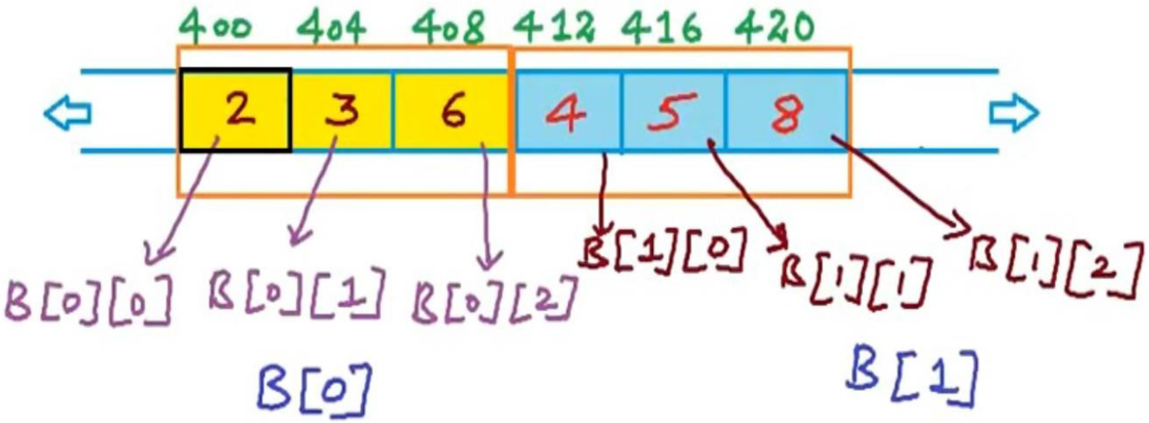
int (*P)[3] = B; ✓

Print B //400

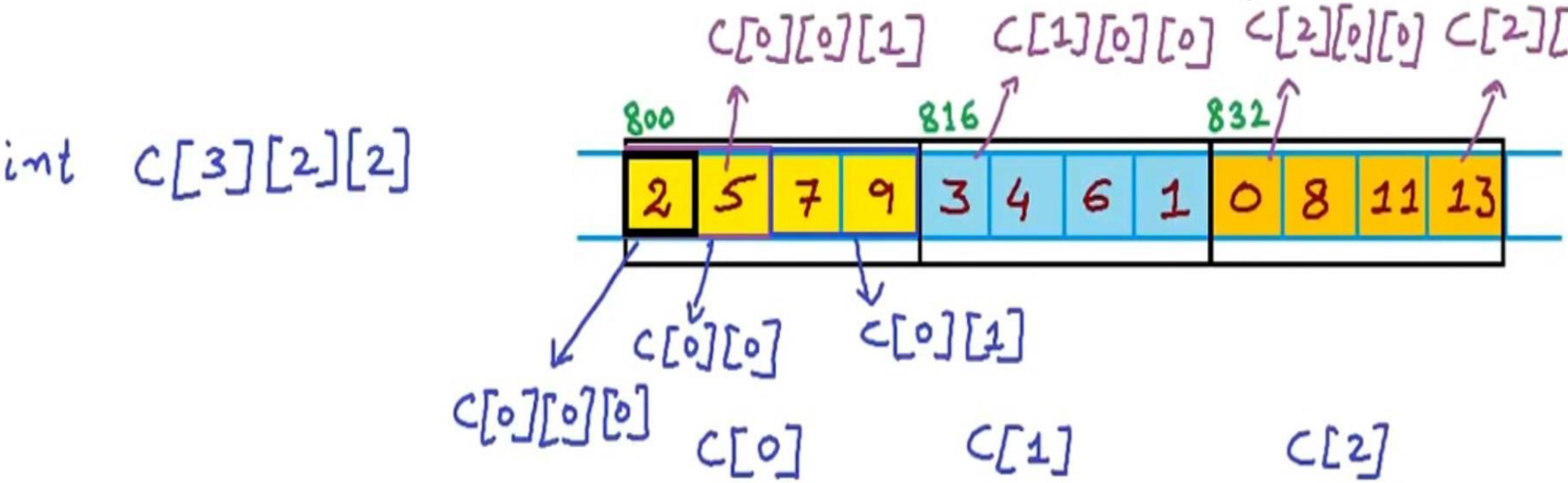
Print *B //400

Print B[0] //400

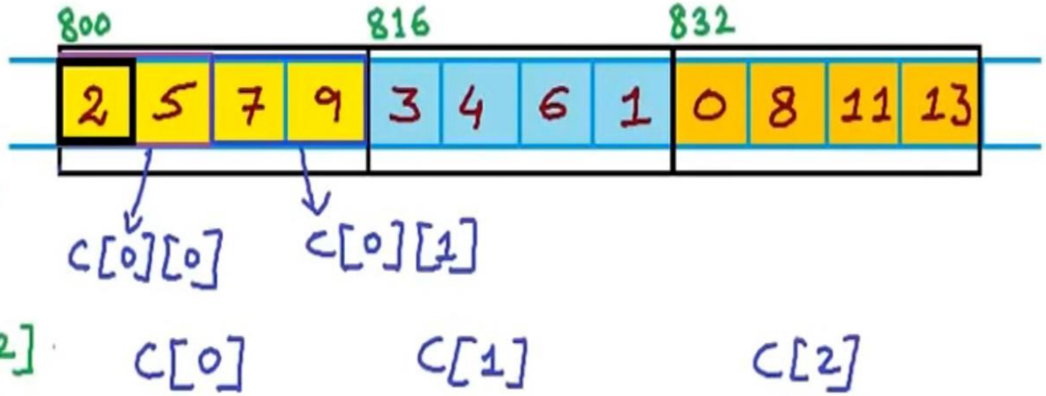
Print &B[0][0] //400



Pointers and multi-dimensional arrays



Pointers and multi-dimensional arrays



int C[3][2][2]

int (*P)[2][2] = C; ✓

Print C //800

↳ int (*) [2][2]

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

↓

int (*)[2]

Pointers and multi-dimensional arrays

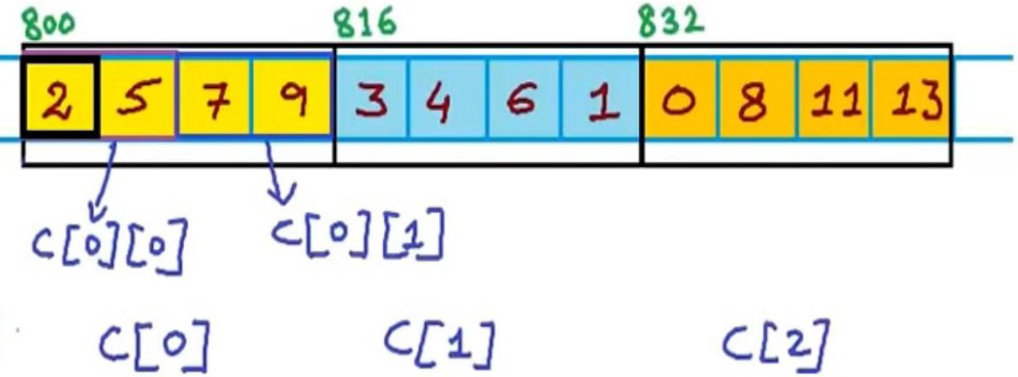
int c[3][2][2]

int (*p)[2][2] = c; ✓

Print c //800

↳ int (*) [2][2]

Print *c or c[0] or &c[0][0]



$$c[i][j][k] = *(c[i][j]+k) = *(*(c[i]+j)+k)$$

$$= *(*(*(c+i)+j)+k)$$

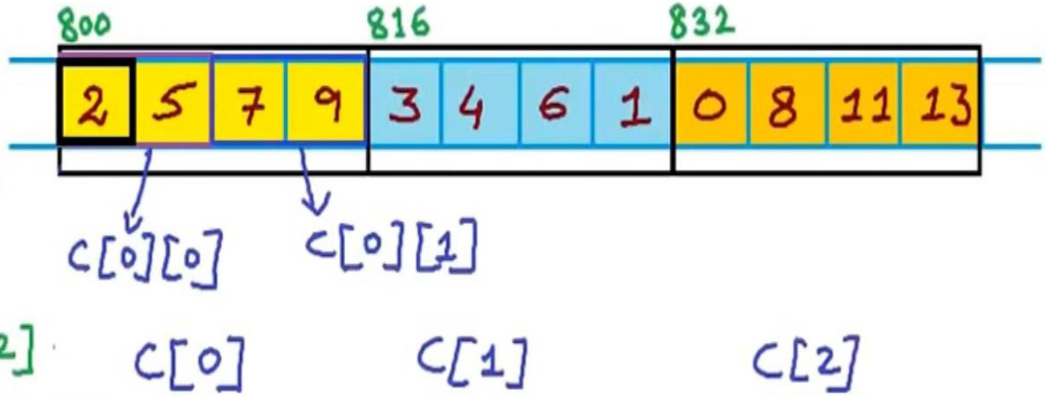
Pointers and multi-dimensional arrays

int C[3][2][2]

int (*P)[2][2] = C; ✓

Print C //800

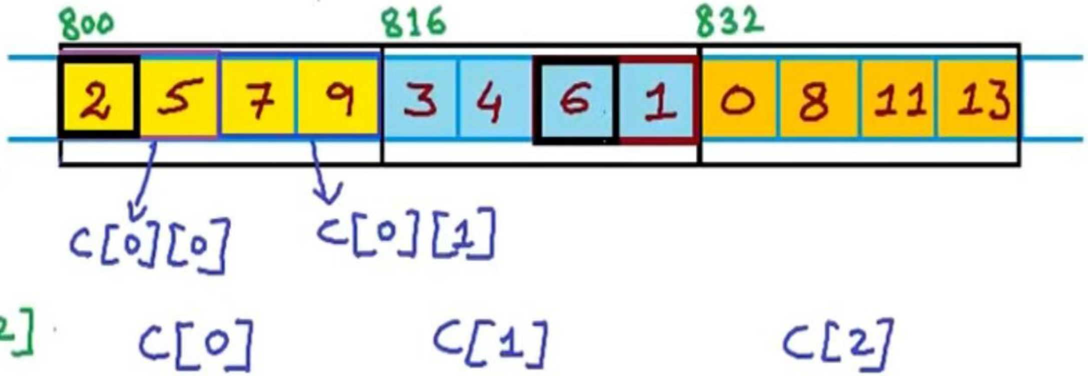
↳ int (*)[2][2]



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

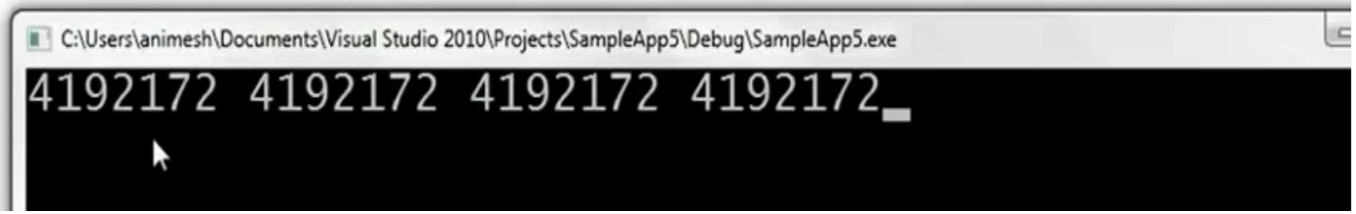
↳ int (*)[2][2]

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

```c
// 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