Multi Dimensional Arrays

Array Declaration v.s.
Two Dimensional Array Declaration

Array	Two Dimensional Array
int a[6]	int b[2][3]
int a[6] = {1,2,3,4,5,6}	int b[2][3] = {{1,2,3} , {4,5,6}} ~~int b[2][3] = {1,2,3,4,5,6} → warning~~

int a[ ] = {1,2,3,4,5,6}	int b[ ][3] = {{1,2,3} , {4,5,6}}
	int b[ ][3] = {{1,2} , {3,4} , {5,6}}
	~~int b[ ][3] = {1,2,3,4,5,6} → warning~~

- ✓ A 1D array is a contiguous block of memory (the name of array behaves like a constant pointer and points to the very first element of the array).
- ✓ The same is true for 2D arrays, array name serves as a constant pointer, and points to the first element of the first row
- ✓ A 2D array of size m by n is defined as arr[m][n] - a 2D array is stored in the memory where entries in row 0 are stored first followed by row 1 and so on. Here m represent the number of rows and n represents the columns i.e. **2-D arrays are represented as a contiguous block of m blocks each with size n**
- ✓ A 2D array can be imagined as a matrix or table of rows and columns or as an array of one dimensional arrays.

int A[2][3] :

A[0]			A[1]		
A[0][0]	A[0][1]	A[0][2]	A[1][0]	A[1][1]	A[1][2]

- Multi-dimensional arrays are initialized the same way as one-dimensional array e.g. :

~~int arr[2][3] = { 1,2,3,4,5,6} → warning~~

- For better readability, a multi-dimensional array can be initialized using sub-aggregate grouping by adding braces accordingly:

```
int arr[2][3] = {
                  {1,2,3},
                  {4,5,6}
                };
```

```
int arr[2][3] = {
                    {1, 2},
                    {3, 4, 5}
};
```

arr has two elements, each of which is an array of three elements. Since only the first two elements of the first row are specified, the third element is zero.

```
1   2   0
3   4   5
```

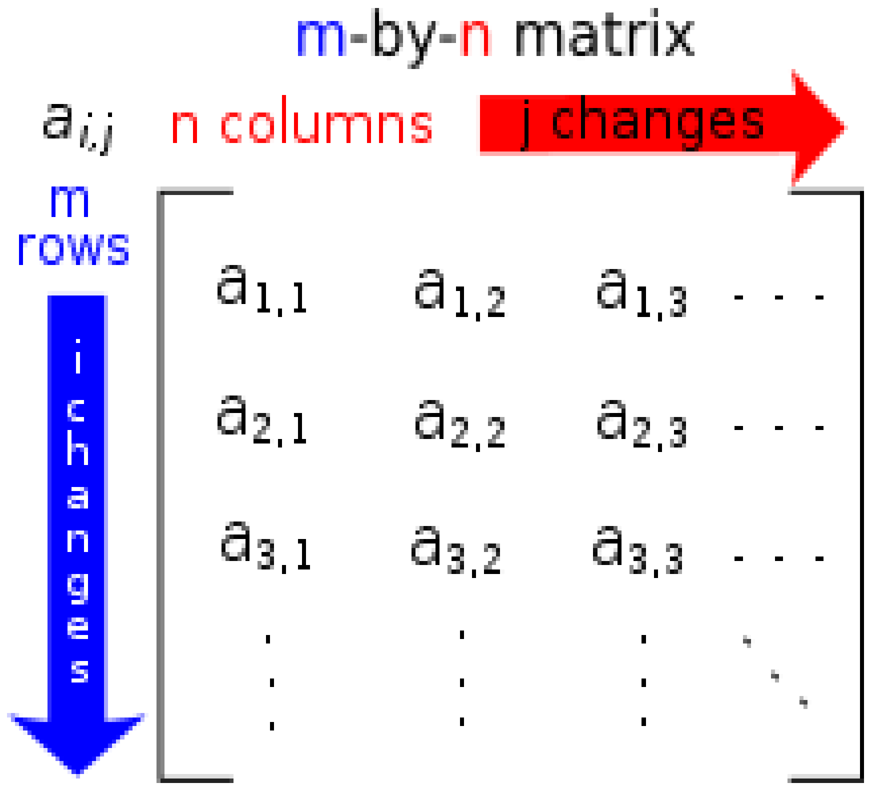
Print all elements of 2D Array

```
#include <stdio.h>
#define M 3
#define N 3
int  main()
{
   int i,j;
   int arr[M][N] =
   {
      {1,2,3},
      {4,5,6},
      {6,7,8}
   };
   for(i=0; i<M; i++)
   {
      for(j=0; j<N; j++)
      {
         printf("%-3d", arr[i][j]);
      }
      putchar('\n');
   }
   return 1;
}
```

Assign elements to 2D Array from STDIN

```
#include <stdio.h>
#define M 3
#define N 3
int  main()
{
   int i,j;
   int arr[M][N] =
   {
      {1,2,3},
      {4,5,6},
      {6,7,8}
   };
   printf("Enter %d elements to 2D array:", M*N);
   for(i=0; i<M; i++)
   {
      for(j=0; j<N; j++)
      {
         scanf("%d", &arr[i][j]);
      }
   }
   return 1;
}
```

Matrix
מטריצה



m-by-n matrix
$a_{i,j}$
n columns
j changes
m rows
i changes
$a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,1}$ $a_{3,2}$ $a_{3,3}$

http://en.wikipedia.org/wiki/Matrix_(mathematics)

- מטריצה מסדר n על m (n-ו m טבעיים) היא מערך דו-ממדי שבו n שורות ו- m עמודות.
- רכיבי המטריצה הם בדרך כלל מספרים $a_{i,j}$
- את הרכיבים מסמנים בזוג אינדקסים
- מטריצה ריבועית היא מטריצה שבה מספר השורות שווה למספר העמודות כלומר m=n .

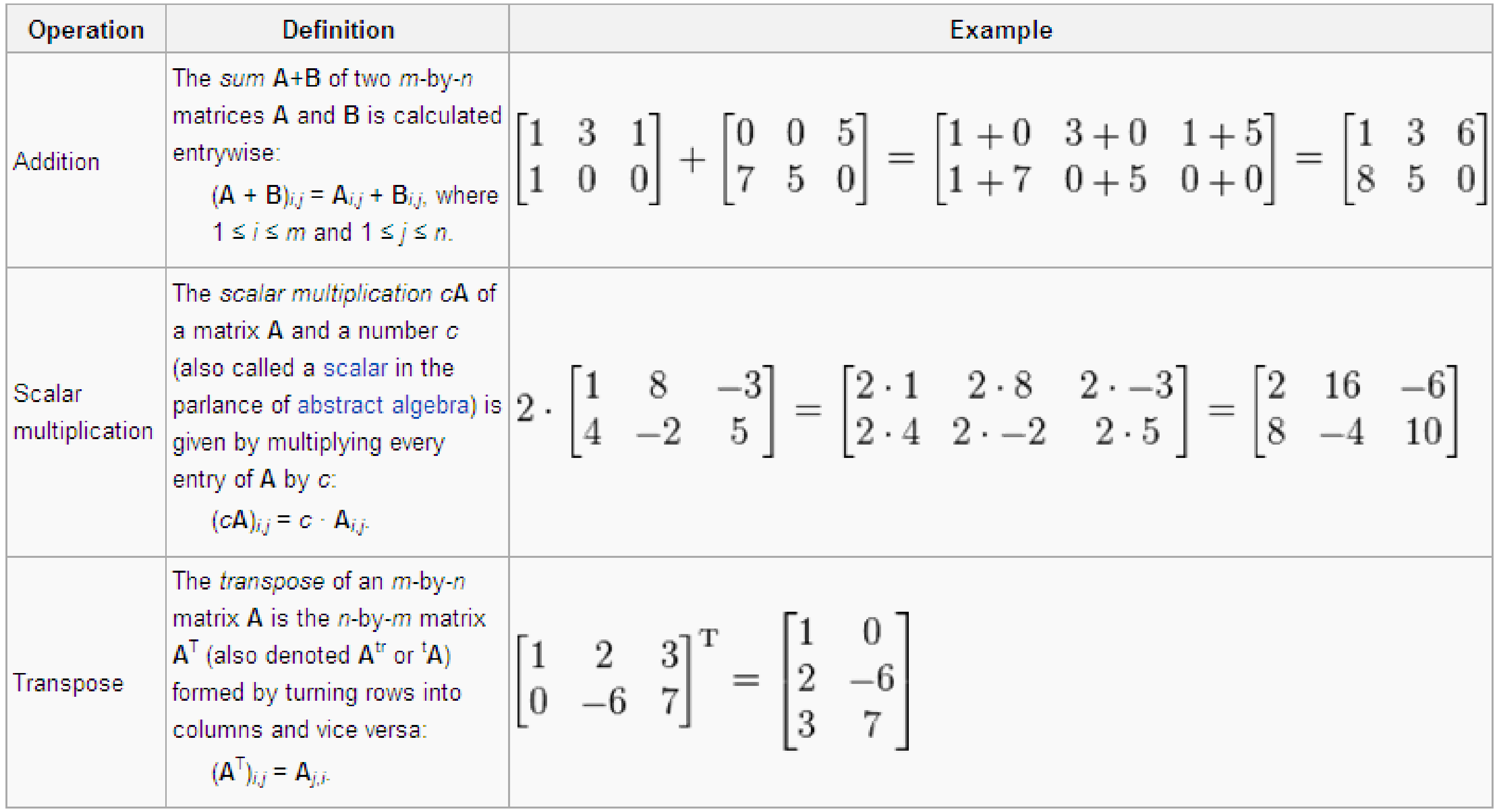
$$\mathbf{A} = \begin{bmatrix} 9 & 13 & 5 \\ 1 & 11 & 7 \\ 3 & 7 & 2 \\ 6 & 0 & 7 \end{bmatrix}.$$

לא ריבועית

$$\begin{bmatrix} 2 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

ריבועית

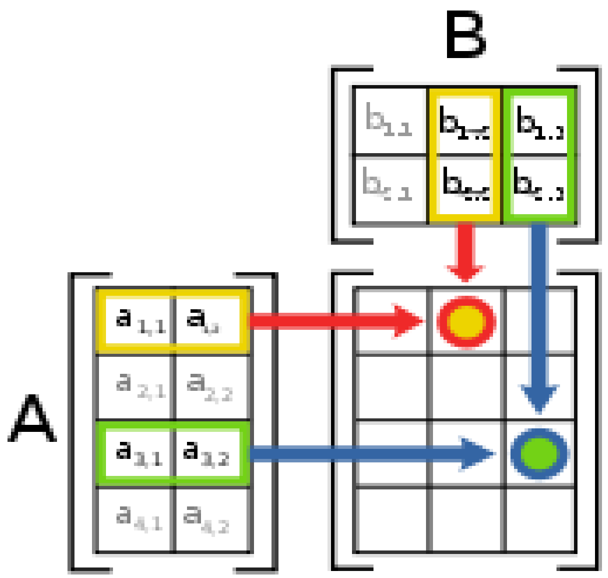
פעולות שניתן לבצע על מטריצות



Operation	Definition	Example
Addition	The *sum* **A**+**B** of two *m*-by-*n* matrices **A** and **B** is calculated entrywise: $(\mathbf{A}+\mathbf{B})_{i,j} = \mathbf{A}_{i,j} + \mathbf{B}_{i,j}$, where $1 \le i \le m$ and $1 \le j \le n$.	$\begin{bmatrix} 1 & 3 & 1 \\ 1 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 5 \\ 7 & 5 & 0 \end{bmatrix} = \begin{bmatrix} 1+0 & 3+0 & 1+5 \\ 1+7 & 0+5 & 0+0 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 6 \\ 8 & 5 & 0 \end{bmatrix}$
Scalar multiplication	The *scalar multiplication* $c\mathbf{A}$ of a matrix **A** and a number *c* (also called a scalar in the parlance of abstract algebra) is given by multiplying every entry of **A** by *c*: $(c\mathbf{A})_{i,j} = c \cdot \mathbf{A}_{i,j}$.	$2 \cdot \begin{bmatrix} 1 & 8 & -3 \\ 4 & -2 & 5 \end{bmatrix} = \begin{bmatrix} 2\cdot 1 & 2\cdot 8 & 2\cdot -3 \\ 2\cdot 4 & 2\cdot -2 & 2\cdot 5 \end{bmatrix} = \begin{bmatrix} 2 & 16 & -6 \\ 8 & -4 & 10 \end{bmatrix}$
Transpose	The *transpose* of an *m*-by-*n* matrix **A** is the *n*-by-*m* matrix $\mathbf{A}^{\mathrm{T}}$ (also denoted $\mathbf{A}^{\mathrm{tr}}$ or ${}^{t}\mathbf{A}$) formed by turning rows into columns and vice versa: $(\mathbf{A}^{\mathrm{T}})_{i,j} = \mathbf{A}_{j,i}$.	$\begin{bmatrix} 1 & 2 & 3 \\ 0 & -6 & 7 \end{bmatrix}^{\mathrm{T}} = \begin{bmatrix} 1 & 0 \\ 2 & -6 \\ 3 & 7 \end{bmatrix}$

http://en.wikipedia.org/wiki/Matrix_(mathematics)

Matrix Multiplication
כפל מטריצות



$$\begin{bmatrix} 2 & 3 & 4 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1000 \\ 1 & 100 \\ 0 & 10 \end{bmatrix} = \begin{bmatrix} 3 & 2340 \\ 0 & 1000 \end{bmatrix}.$$

$$[\mathbf{AB}]_{i,j} = A_{i,1}B_{1,j} + A_{i,2}B_{2,j} + \cdots + A_{i,n}B_{n,j} = \sum_{r=1}^{n} A_{i,r}B_{r,j}$$

Borders of a Sub Matrix

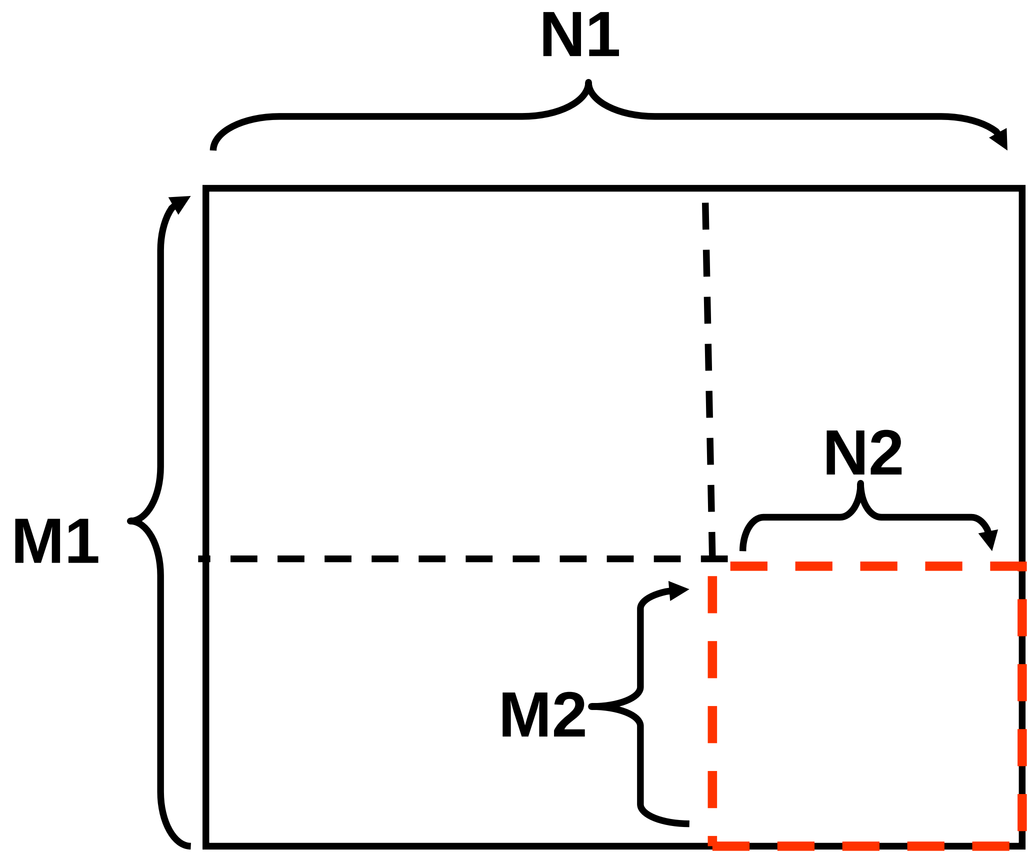
B[M2][N2]

4	5
7	3

is B sub a of A?

A[M1][N1]

1	2	1	1	1
1	4	5	2	4
2	7	3	4	5
0	1	6	7	3



התא הראשון של תת המטריצה , אסור שיעבור את הגבול הבא :

Borders : [M1-M2] , [N1-N2] = [4-2][5-2]=[2][3]

תוכנית אשר מדפיסה את האלכסון הראשי והמשני במטריצה ריבועית

```c
#include <stdio.h>
#define M 3
#define N 3
int  main()
{
   int i;
   int arr[M][N] =
   {
      {1,2,3},
      {4,5,6},
      {6,7,8}
   };
   printf("---------\n");
   for(i=0; i<M; i++)
   {
      printf("|%-3d|%-3d|\n", arr[i][i], arr[i][N-i-1]);
   }
   printf("---------\n");
   return 1;
}
```

Output:

```
---------
|1  |3  |
|5  |5  |
|8  |6  |
---------
```

Check if Matrix ‘a’ is sub of Matrix ‘b’

version1

```
#include <stdio.h>
#define M1 3
#define N1 5
#define M2 2
#define N2 2
int  main() {
 int i, j, ii, jj, STOP=0, FOUND=0;
 int a[M1][N1]={{1,2,3,4,5},
                {4,5,6,6,7},
                {6,7,8,9,1}};
 int b[M2][N2] ={{6,7},
                 {9,1}};
 for(i=0; (i<=M1-M2) && !FOUND; i++)
   for(j=0; (j<=N1-N2) && !FOUND; j++, FOUND=!STOP)
     for(ii=0, STOP=0; (ii<M2) && !STOP; ii++)
       for(jj=0; (jj<N2) && !STOP; jj++)
          if(a[i+ii][j+jj] != b[ii][jj])
             STOP=1;
 if(FOUND)
  printf("Found!\n");
 else
  printf("Not found\n");
 return 1;
}
```

version2

```
#include <stdio.h>
#define M1 3
#define N1 5
#define M2 2
#define N2 2
int check(int a[M1][N1],int b[M2][N2], int i, int j) {
  int ii, jj;
  for(ii=0; ii<M2; ii++)
      for(jj=0; jj<N2; jj++)
         if(a[i+ii][j+jj] != b[ii][jj])    return 0;
  return 1;
}
int isSub(int a[M1][N1],int b[M2][N2]) {
  int i,j;
  for(i=0; i<=M1-M2; i++)
   for(j=0; j<=N1-N2; j++)
     if(check(a,b,i,j))    return 1;
  return 0;
}
int main() {
 int a[M1][N1]={{1,2,3,4,5}, {4,5,6,6,7},{6,7,8,9,1}};
 int b[M2][N2] ={{6,7}, {9,1}};
 if(isSub(a,b))   printf("Found!\n");
 else  printf("Not found\n");
  return 1;
}
```

Find the Biggest sub square matrix having Zero elements only.

```
#include <stdio.h>
#define M 4
#define N 6
int check(int a[M][N],int i, int j,int k) {
  int ii,jj;
  for(ii=0; ii<k; ii++)
    for(jj=0; jj<k; jj++)
       if(a[i+ii][j+jj])
          return 0;
  return 1;
}
int main() {
 int i,j,k,STOP=0
 int a[M][N] = {{0,1,0,1,1,0},
                         {0,0,0,1,1,0},
                         {0,0,0,1,1,0},
                         {0,0,0,1,1,0}};
 for(k=(M>N?N:M); (k>0) && !STOP; k--)
     for(i=0; (i<=M-k) && !STOP; i++)
         for(j=0; (j<=N-k) && !STOP; j++)
             STOP=check(a,i,j,k);
 if(STOP) printf("\nMax size is %d start at (%d,%d)", ++k,--i,--j);
 else printf("Not found!");
 return 1;
}
```

(ii,jj): on the sub matrix

K: maximum size

(i,j): on the biggest matrix

```
0,1,0,1,1,0
0,0,0,1,1,0
0,0,0,1,1,0
0,0,0,1,1,0
```

Arrays of Strings

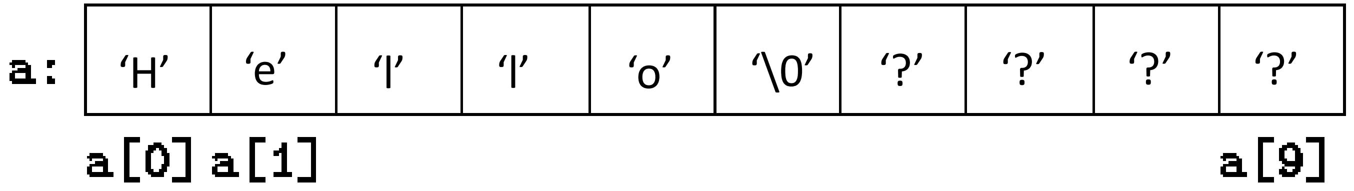
- An array of strings is a special form of a two-dimensional array.

✓ The size of the left index determines the number of strings.

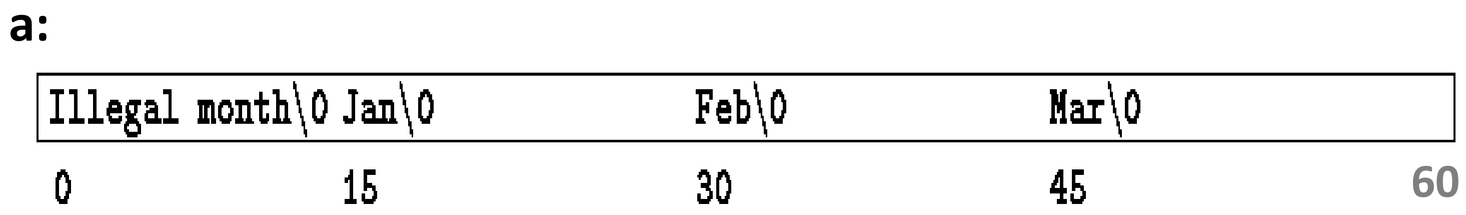
✓ The size of the right index specifies the maximum length of each string.

- For example, the following declares an array of 2 strings, each having a maximum length of 10 characters

```
char string_array[2][10] = {"Hello", "Students"};
char a[10] = “Hello”;
```



```
char a[4][15] = { "Illegal month", "Jan", "Feb", "Mar" };
```



```
#include <stdio.h>

int main() {
    char a[4][15] = { "Illegal month", "Jan", "Feb", "Mar" };
    int i,j;

    for(i=0;i<4;i++,putchar('\n'))
      for(j=0;j<15;j++)
       printf("%c ", a[i][j]);

    return 1;
}
```

```
#include <stdio.h>

int main() {
    char a[4][15] = { "Illegal month", "Jan", "Feb", "Mar" };
    int i,j;

    for(i=0;i<4;i++,putchar('\n'))
      for(j=0;j<15;j++)
       printf("%c ", *(*(a+i) + j));
    return 1;
}
```

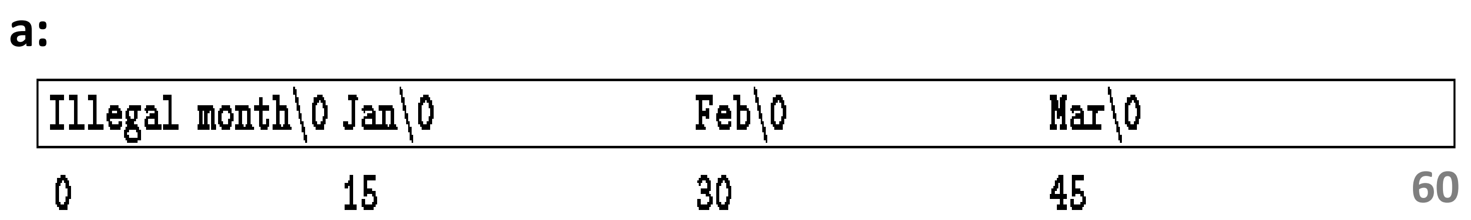
Multi-Dimensional Array vs. Array of Pointers

two-dimensional array $\longrightarrow$ int a[10][20] ⇨ 200 int-sized locations have been allocated

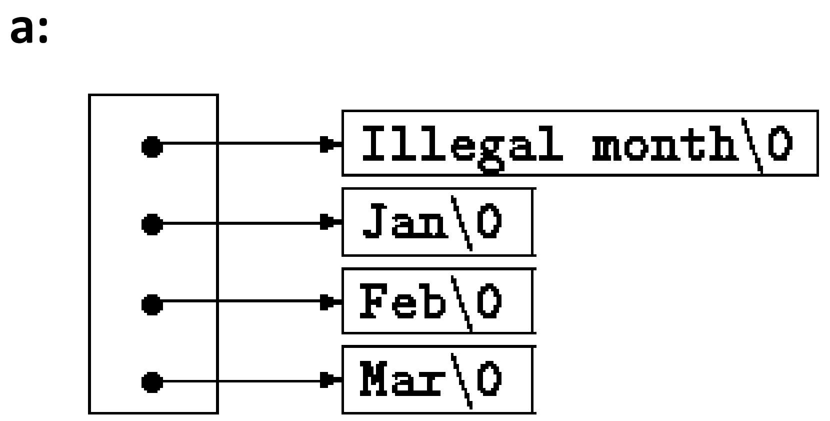
Array of pointers $\longrightarrow$ int *b[10] ⇨ 10 uninitialized pointers have been allocated. Initialization must be done explicitly. Assuming that each element of b does point to a twenty-element array, then there will be 200 ints set aside, plus ten cells for the pointers.

The most advantage of array of pointers is that the rows of the array may be of different lengths.

```
char a[4][15] = { "Illegal month", "Jan", "Feb", "Mar" }:
```



```
char *a[] = { "Illegal month", "Jan", "Feb", "Mar" }:
```



```c
#include <stdio.h>
int  main()
{
  int i,j;
   int arr[2][6] = {
                  {1,2,3,4,5,6},
                  {7,8,9,10,11,12}
                 };

   printf("%p %p %p %p %p\n",arr,&arr[0],&arr[1],&arr[0][0],&arr[1][0]);
   printf("%p %p\n",arr[0],arr[1]);

   for(i=0;i<2;i++, putchar('\n'))
     for(j=0;j<6;j++)
        printf("%d ",*(*arr + i*6 + j));

   return 1;
}
```

Output:

```
0xbf9cf488 0xbf9cf488 0xbf9cf4a0 0xbf9cf488 0xbf9cf4a0
0xbf9cf488 0xbf9cf4a0
1 2 3 4 5 6
7 8 9 10 11 12
```

arr

1	2	3	4	5	6
7	8	9	10	11	12

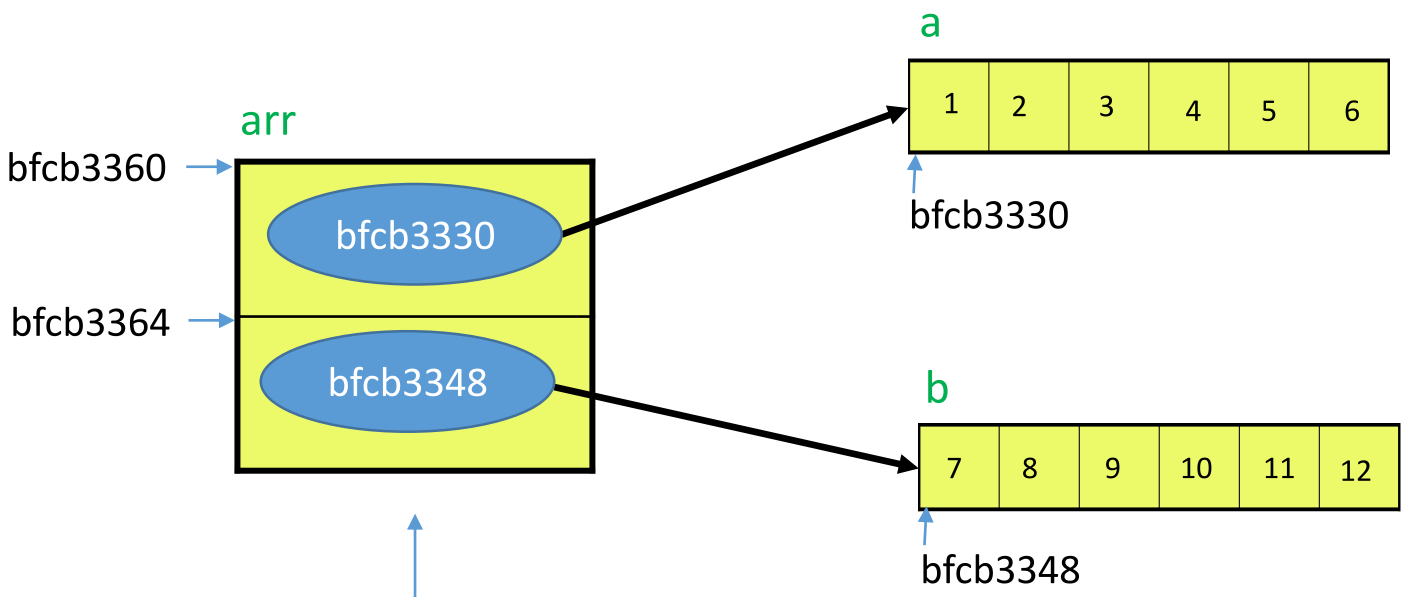
arr[i][j] = *(arr[i] + j) = *(*(arr+i) + j)) = *(*arr + i*6 + j)

```
matrix[0][0] = *(*(matrix))
matrix[i][j] = *((*(matrix)) + (i * COLS + j))
matrix[i][j] = *(*(matrix + i) + j)
matrix[i][j] = *(matrix[i] + j)
matrix[i][j] = (*(matrix + i))[j]
&matrix[i][j] = ((*(matrix)) + (i * COLS + j))
```

```c
#include <stdio.h>
int  main()
{
   int a[6] = {1,2,3,4,5,6};
   int b[6] = {7,8,9,10,11,12};

   int *arr[2];
   arr[0]=a;
   arr[1]=b;

   printf("%p %p %p %p\n",a,&a[0],b,&b[0]);
   printf("%p %p %p\n",arr,&arr[0],&arr[1]);
   printf("%p %p\n",arr[0],arr[1]);

   return 1;
}
```

Output:

```
0xbfcb3330 0xbfcb3330 0xbfcb3348 0xbfcb3348
0xbfcb3360 0xbfcb3360 0xbfcb3364
0xbfcb3330 0xbfcb3348
```



Array of Pointers to Arrays
(Array of Arrays)

Passing Two-Dimensional Array to a Function in C

While passing a two dimensional array to a function, a one should either use square bracket syntax or pointer to an array syntax but not double pointer.

A called function does not allocate space for the passed array because it does not create a local copy of the array rather it uses the original one that has been passed to it. Hence, it does not need to know the overall size and the number of rows can be omitted. However, the width of the array is still important because the compiler must know the number of elements contained by one row in order to increment the pointer to point to the next row. So the column dimension must be specified.

```
#include <stdio.h>
void printMat(int arr[][3])
{
   int i,j;
   for(i=0; i<2; i++,putchar('\n'))
      for(j=0; j<3; j++)
         printf("%d ",arr[i][j]);
}
int  main() {
   int arr[][3] = {
      {1,2,3},
      {4,5,6}
   };
   printMat(arr);
   return 0;
}
```

```
#include <stdio.h>
void printMat(int (*arr)[3])
{
   int i,j;
   for(i=0; i<2; i++)
   {
      for(j=0; j<3; j++)
         printf("%d ",(*arr)[j]);

      putchar('\n');
      arr++;
    }
}
int  main() {
   int arr[][3] = {
      {1,2,3},
      {4,5,6}
   };
   printMat(arr);
   return 0;
}
```

Double Pointer and Two Dimensional Arrays

```c
#include <stdio.h>
int  main()
{
   int matrix[][3] =
   {
      {1,2,3},
      {4,5,6}
   };

   int** pmat = (int **)matrix;

   printf("%x\n", &matrix[0][0]);
   printf("%x\n", &pmat[0][0]);

   return 0;
}
```

?

printf("%x\n", pmat[0][0]);
Will cause dump.why?

Output:

```
bfa71104
1
```

2D array ‘matrix’ and double pointer ‘pmat’ have different types and they points to different locations in memory.

```c
#include <stdio.h>
void printMat(int **pa) {
   int i,j;
   for(i=0; i<2; i++)
   {
      for(j=0; j<3; j++)
      {
         printf("%d ",**pa);
         (*pa)++;
      }
      putchar('\n');
      pa++;
   }
}
int  main(){
   int arr[][3] ={
      {1,2,3},
      {4,5,6}
   };
   printMat(arr);
   return 0;
}
```

Warning: expected ‘int **’ but argument is of type ‘int (*)[3]

✗

```c
#include <stdio.h>
void printMat(int **pa){
   int i,j;
   for(i=0; i<2; i++)
   {
      for(j=0; j<3; j++)
      {
         printf("%d ",**pa);
         (*pa)++;
      }
      putchar('\n');
      pa++;
   }
}
int  main(){
   int arr[][3] ={
      {1,2,3},
      {4,5,6}
   };
   printMat((int**)arr);
   return 0;
}
```

Segmentation fault (core dumped)

✗

```c
#include <stdio.h>
void printMat(int **pa){
   int i,j;
   for(i=0; i<2; i++)
   {
      for(j=0; j<3; j++)
      {
         printf("%d ",**pa);
         (*pa)++;
      }
      putchar('\n');
      pa++;
   }
}
int  main() {
   int arr[][3] ={
      {1,2,3},
      {4,5,6}
   };
   int *pa[2];
   pa[0]=arr[0];
   pa[1]=arr[1];
   printMat(pa);
   return 0;
}
```

✓

Dynamic allocation of 2D array

```
#include <stdio.h>
#include <stdlib.h>

void matrixAllocate(int*** pMat, int m, int n)
{
   int i=0;

   *pMat = (int**)malloc(m*sizeof(int*));

   for (i=0; i<m; i++)
      (*pMat)[i] = (int*)malloc(n*sizeof(int));
}

int  main()
{
   int i, j, m=2, n=3;
   int** matrix;
   matrixAllocate(&matrix,m,n);

   return 0;
}
```

END