Finally … Functions!

ESC101: Fundamentals of Computing

Purushottam Kar

Announcements

- Mid-sem lab exam marks available on Prutor
  - Maximum: 150, Average: 80, Standard deviation: 47
  - See rubric on website carefully before applying for regrading
  - Frivolous regrading requests will simply get penalty negative marks
- Last date for dropping Advanced Track October 12
  - Application must be an email to instructor, mentors, teammates
- Last date for dropping ESC101 course October 12
  - Application must be on standard DoAA course drop form – no email!
- Joint tutorial for B1 and B14 on October 12
  - 12 – 1 PM (same time), L19 - just an arrangement for this week ☺

The Golden Rules of Pointers

The Golden Rules of Pointers

RULE 1: All pointers store addresses, take 8 bytes to store

The Golden Rules of Pointers

RULE 1: All pointers store addresses, take 8 bytes to store

Does not matter whether pointer to variable, to array, to another pointer etc

The Golden Rules of Pointers

RULE 1: All pointers store addresses, take 8 bytes to store

Does not matter whether pointer to variable, to array, to another pointer etc

RULE 2 (Reference): &a gives address of variable a

The Golden Rules of Pointers

RULE 1: All pointers store addresses, take 8 bytes to store

Does not matter whether pointer to variable, to array, to another pointer etc

RULE 2 (Reference): &a gives address of variable a

Does not matter whether variable a is char, long, or even a pointer variable

The Golden Rules of Pointers

RULE 1: All pointers store addresses, take 8 bytes to store

Does not matter whether pointer to variable, to array, to another pointer etc

RULE 2 (Reference): &a gives address of variable a

Does not matter whether variable a is char, long, or even a pointer variable

Special case for static array variables – next slide

The Golden Rules of Pointers

RULE 1: All pointers store addresses, take 8 bytes to store

Does not matter whether pointer to variable, to array, to another pointer etc

RULE 2 (Reference): &a gives address of variable a

Does not matter whether variable a is char, long, or even a pointer variable

Special case for static array variables – next slide

RULE 3: (Dereference): Whenever expression **expr** generates an address, *(**expr**) gives value stored at that address

The Golden Rules of Pointers

RULE 1: All pointers store addresses, take 8 bytes to store

Does not matter whether pointer to variable, to array, to another pointer etc

RULE 2 (Reference): &a gives address of variable a

Does not matter whether variable a is char, long, or even a pointer variable

Special case for static array variables – next slide

RULE 3: (Dereference): Whenever expression **expr** generates an address, ***(expr)** gives value stored at that address

Careful! Value at the address given by **expr** is interpreted as type of **expr**

The Golden Rules of Pointers

RULE 1: All pointers store addresses, take 8 bytes to store

Does not matter whether pointer to variable, to array, to another pointer etc

RULE 2 (Reference): &a gives address of variable a

Does not matter whether variable a is char, long, or even a pointer variable

Special case for static array variables – next slide

RULE 3: (Dereference): Whenever expression **expr** generates an address, *(**expr**) gives value stored at that address

Careful! Value at the address given by **expr** is interpreted as type of **expr**

RULE 4: (Arithmetic): Pointer arithmetic is w.r.t datatype

The Golden Rules of Pointers

RULE 1: All pointers store addresses, take 8 bytes to store

Does not matter whether pointer to variable, to array, to another pointer etc

RULE 2 (Reference): &a gives address of variable a

Does not matter whether variable a is char, long, or even a pointer variable

Special case for static array variables – next slide

RULE 3: (Dereference): Whenever expression **expr** generates an address, *(**expr**) gives value stored at that address

Careful! Value at the address given by **expr** is interpreted as type of **expr**

RULE 4: (Arithmetic): Pointer arithmetic is w.r.t datatype

char* arithmetic w.r.t. 1 byte blocks, int* w.r.t. 4 byte blocks, double* 8 bytes

The Golden Rules of Pointers

RULE 1: All pointers store addresses, take 8 bytes to store

Does not matter whether pointer to variable, to array, to another pointer etc

RULE 2 (Reference): &a gives address of variable a

Does not matter whether variable a is char, long, or even a pointer variable

Special case for static array variables – next slide

RULE 3: (Dereference): Whenever expression **expr** generates an address, *(**expr**) gives value stored at that address

Careful! Value at the address given by **expr** is interpreted as type of **expr**

RULE 4: (Arithmetic): Pointer arithmetic is w.r.t datatype

char* arithmetic w.r.t. 1 byte blocks, int* w.r.t. 4 byte blocks, double* 8 bytes

RULE 5: Name of array points to first element of array

The Golden Rules of Pointers

RULE 1: All pointers store addresses, take 8 bytes to store

Does not matter whether pointer to variable, to array, to another pointer etc

RULE 2 (Reference): &a gives address of variable a

Does not matter whether variable a is char, long, or even a pointer variable

Special case for static array variables – next slide

RULE 3: (Dereference): Whenever expression **expr** generates an address, ***(expr)** gives value stored at that address

Careful! Value at the address given by **expr** is interpreted as type of **expr**

RULE 4: (Arithmetic): Pointer arithmetic is w.r.t datatype

char* arithmetic w.r.t. 1 byte blocks, int* w.r.t. 4 byte blocks, double* 8 bytes

RULE 5: Name of array points to first element of array

Does not matter whether malloc-ed array or static array

The Curious Case of Static Arrays

The Curious Case of Static Arrays

Three types of arrays studied so far

The Curious Case of Static Arrays

Three types of arrays studied so far

- Static arrays of fixed size `int a[10];`

The Curious Case of Static Arrays

Three types of arrays studied so far

- Static arrays of fixed size int a[10];
- Static arrays of variable size int n; scanf("%d",&n); int b[n];

The Curious Case of Static Arrays

Three types of arrays studied so far

- Static arrays of fixed size int a[10];
- Static arrays of variable size int n; scanf("%d",&n); int b[n];
- Dynamic malloc/calloc/realloc-ed arrays int *c = (int*)malloc(n * sizeof(int));

The Curious Case of Static Arra

Three types of arrays studied so far

- Static arrays of fixed size int a[10];
- Static arrays of variable size int n; scanf("%d",&n); int b[n];
- Dynamic malloc/calloc/realloc-ed arrays int *c = (int*)malloc(n * sizeof(int));

The Cur a

a, b, c all point to their respective first elements ☺

Three types of arrays studied so far

- Static arrays of fixed size int a[10];
- Static arrays of variable size int n; scanf("%d",&n); int b[n];
- Dynamic malloc/calloc/realloc-ed arrays int *c = (int*)malloc(n * sizeof(int));

The Cur

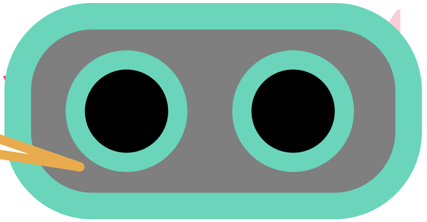
a, b, c all point to their respective first elements ☺

Three types of arrays studied so far

- Static arrays of fixed size int a[10];
- Static arrays of variable size int n; scanf("%d",&n); int b[n];
- Dynamic malloc/calloc/realloc-ed arrays int *c = (int*)malloc(n * sizeof(int));

For static arrays (fixed/variable length) **sizeof()** gives total size of array. However, for malloc-ed arrays, **sizeof(c)** just gives 8, the space required to store the pointer **c** ☹

The Cur a



a, b, c all point to their respective first elements ☺

Three types of arrays studied so far

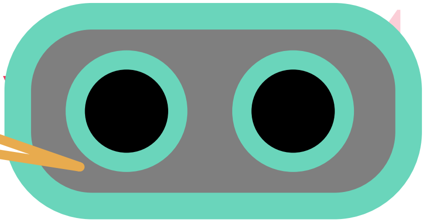
- Static arrays of fixed size int a[10];
- Static arrays of variable size int n; scanf("%d",&n); int b[n];
- Dynamic malloc/calloc/realloc-ed arrays int *c = (int*)malloc(n * sizeof(int));

For static arrays (fixed/variable length) **sizeof()** gives total size of array. However, for malloc-ed arrays, **sizeof(c)** just gives 8, the space required to store the pointer **c** ☹

&c gives us the address where the pointer c is stored ☺

The Cur

a, b, c all point to their respective first elements ☺



Three types of arrays studied so far

- Static arrays of fixed size int a[10];
- Static arrays of variable size int n; scanf("%d",&n); int b[n];
- Dynamic malloc/calloc/realloc-ed arrays int *c = (int*)malloc(n * sizeof(int));

For static arrays (fixed/variable length) **sizeof()** gives total size of array. However, for malloc-ed arrays, **sizeof(c)** just gives 8, the space required to store the pointer **c** ☹

&c gives us the address where the pointer c is stored ☺

&a just gives us the address of first element a[0] again ☹

The Cur

a, b, c all point to their respective first elements ☺

Three types of arrays studied so far

- Static arrays of fixed size int a[10];
- Static arrays of variable size int n; scanf("%d",&n); int b[n];
- Dynamic malloc/calloc/realloc-ed arrays int *c = (int*)malloc(n * sizeof(int));

For static arrays (fixed/variable length) **sizeof()** gives total size of array. However, for malloc-ed arrays, **sizeof(c)** just gives 8, the space required to store the pointer **c** ☹

&c gives us the address where the pointer c is stored ☺

&a just gives us the address of first element a[0] again ☹

&b just gives us the address of first element b[0] again ☹

The Cur

a, b, c all point to their respective first elements ☺

I will hide the location of the pointer a and b from you since I store these pointers secretly in a location called the **symbol table** – no access!!

Three types

- Static arrays of
- Static arrays of variable size int n; scanf("%d",&n); int b[n];
- Dynamic malloc/calloc/realloc-ed arrays int *c = (int*)malloc(n * sizeof(int));

For static arrays (fixed/variable length) **sizeof()** gives total size of array. However, for malloc-ed arrays, **sizeof(c)** just gives 8, the space required to store the pointer **c** ☹

&c gives us the address where the pointer c is stored ☺

&a just gives us the address of first element a[0] again ☹

&b just gives us the address of first element b[0] again ☹

The Cur

a, b, c all point to their respective first elements ☺

I will hide the location of the pointer a and b from you since I store these pointers secretly in a location called the **symbol table** – no access!!

You can modify the pointer c by saying c++ but I will not allow you to say things like a++, b++. I also dont allow you to free/realloc a and b ☺

Three types

Static arra

d",&n); int b[n];

s int *c = (int*)malloc(n * sizeof(int));

For static arrays (fixed/variable length) sizeof() gives total size of array. However, for malloc-ed arrays, sizeof(c) just gives 8, the space required to store the pointer c ☹

&c gives us the address where the pointer c is stored ☺

&a just gives us the address of first element a[0] again ☹

&b just gives us the address of first element b[0] again ☹

2D Arrays in C

2D Arrays in C

```
int mat[3][5];
```

2D Arrays in C

```
int mat[3][5];
```

Declares a matrix (2D array) with 3 rows 5 columns

2D Arrays in C

```
int mat[3][5];
```

Declares a matrix (2D array) with 3 rows 5 columns

Rows numbered 0, 1, 2. Columns numbered 0, 1, 2, 3, 4

2D Arrays in C

```
int mat[3][5];
```

Declares a matrix (2D array) with 3 rows 5 columns

Rows numbered 0, 1, 2. Columns numbered 0, 1, 2, 3, 4

Element at row-index i and column-index j is an int variable

2D Arrays in C

int mat[3][5];

Declares a matrix (2D array) with 3 rows 5 columns

Rows numbered 0, 1, 2. Columns numbered 0, 1, 2, 3, 4

Element at row-index i and column-index j is an int variable

Can access it using several ways
mat[i][j],*(mat[i] + j),*(*(mat + i) + j),(*(mat + i))[j]

2D Arrays in C

int mat[3][5];

Declares a matrix (2D array) with 3 rows 5 columns

Rows numbered 0, 1, 2. Columns numbered 0, 1, 2, 3, 4

Element at row-index i and column-index j is an int variable

Can access it using several ways
mat[i][j],*(mat[i] + j),*(*(mat + i) + j),(*(mat + i))[j]

Careful! **(mat + i +j) ≠ *(*(mat + i) + j) ≠ *(*mat + i + j)

2D Arrays in C

```
int mat[3][5];
```

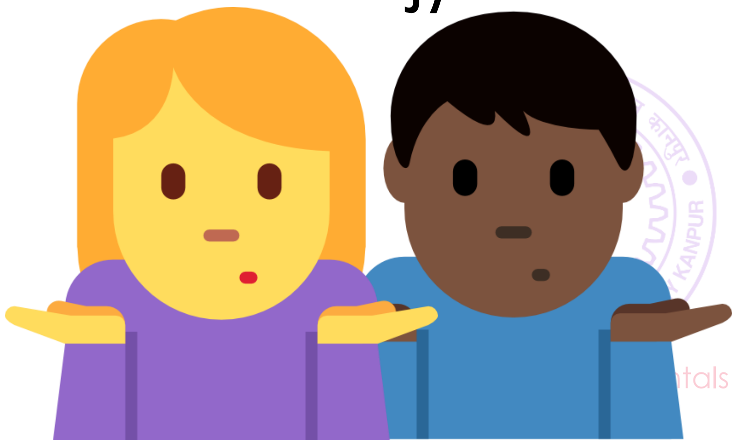
Declares a matrix (2D array) with 3 rows 5 columns

Rows numbered 0, 1, 2. Columns numbered 0, 1, 2, 3, 4

Element at row-index i and column-index j is an int variable

Can access it using several ways
mat[i][j],*(mat[i] + j),*(*(mat + i) + j),(*(mat + i))[j]

Careful! **(mat + i +j) ≠ *(*(mat + i) + j) ≠ *(*mat + i + j)



2D Arrays in C

```
int mat[3][5];
```

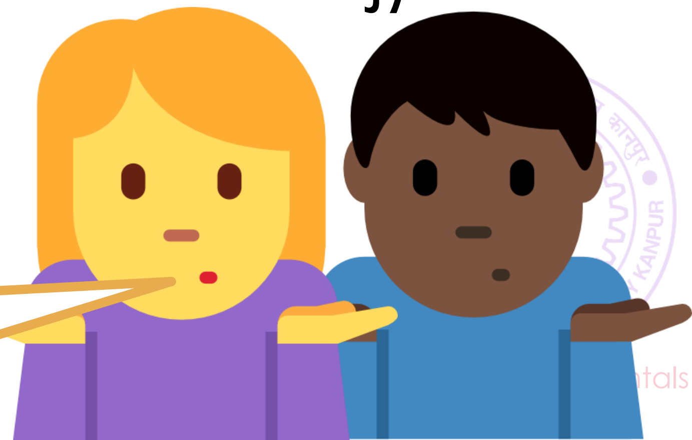
Declares a matrix (2D array) with 3 rows 5 columns

Rows numbered 0, 1, 2. Columns numbered 0, 1, 2, 3, 4

Element at row-index i and column-index j is an int variable

Can access it using several ways
mat[i][j],*(mat[i] + j),*(*(mat + i) + j),(*(mat + i))[j]

Careful! **(mat + i +j) ≠ *(*(mat + i) + j) ≠ *(*mat + i + j)



This looks exactly like the way we access an array of arrays – what is the difference?

2D Arrays in C

int mat[3][5];

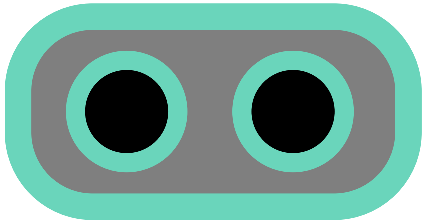
Declares a matrix (2D array) with 3 rows 5 columns

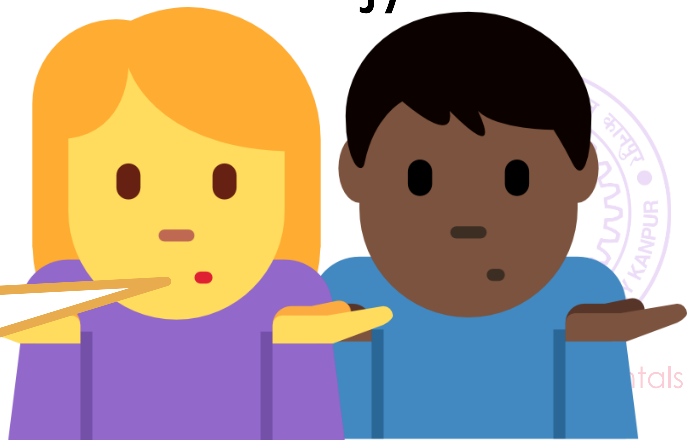
Rows numbered 0, 1, 2. Columns numbered 0, 1, 2, 3, 4

Element at row-index i and column-index j is an int variable

Can access it using several ways
mat[i][j],*(mat[i] + j),*(*(mat + i) + j),(*(mat + i))[j]

Careful! **(mat + i +j) ≠ *(*(mat + i) + j) ≠ *(*mat + i + j)





This looks exactly like the way we access an array of arrays – what is the difference?

2D Arrays in C

int mat[3][5];

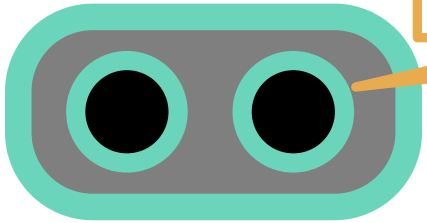
Declares a matrix (2D array) with 3 rows 5 columns

Rows numbered 0, 1, 2. Columns numbered 0, 1, 2, 3, 4

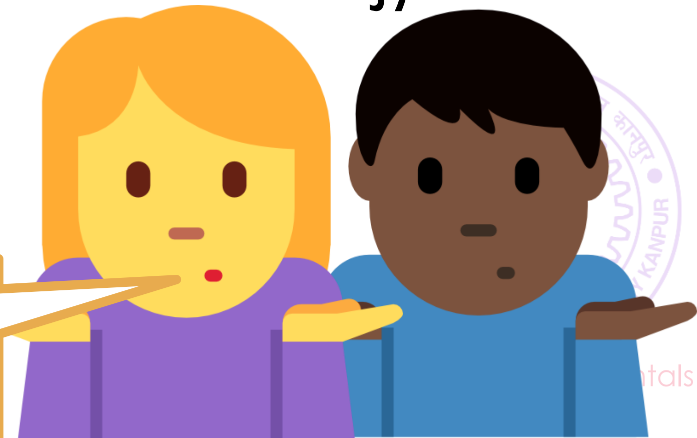
Element at row-index i and column-index j is an int variable

Can access it using several ways
mat[i][j],*(mat[i] + j),*(*(mat + i) + j),(*(mat + i))[j]

Careful! **(mat + i +j) ≠ *(*(mat + i) + j) ≠ *(*mat + i + j)



Not that much actually – let me show you the differences



This looks exactly like the way we access an array of arrays – what is the difference?

2D arrays vs Array of arrays

2D ARRAYS

ARRAY OF ARRAYS

2D arrays vs Array of arrays

2D ARRAYS

Number of elements in each row is the same

ARRAY OF ARRAYS

2D arrays vs Array of arrays

2D ARRAYS

Number of elements in each row is the same

ARRAY OF ARRAYS

Different arrays can have different number of elements – more flexibility

2D arrays vs Array of arrays

2D ARRAYS

Number of elements in each row is the same

All elements of 2D array are located contiguously in memory

ARRAY OF ARRAYS

Different arrays can have different number of elements – more flexibility

2D arrays vs Array of arrays

2D ARRAYS

Number of elements in each row is the same

All elements of 2D array are located contiguously in memory

ARRAY OF ARRAYS

Different arrays can have different number of elements – more flexibility

Elements of a single array are contiguous but different arrays could be located far off in memory

2D arrays vs Array of arrays

2D ARRAYS

Number of elements in each row is the same

All elements of 2D array are located contiguously in memory

Easier to initialize

ARRAY OF ARRAYS

Different arrays can have different number of elements – more flexibility

Elements of a single array are contiguous but different arrays could be located far off in memory

2D arrays vs Array of arrays

2D ARRAYS

Number of elements in each row is the same

All elements of 2D array are located contiguously in memory

Easier to initialize

```
int mat[3][5] = { {1,2}, {3},
{4,5,6},{7,8,9,10,11},{-1,2,3,4}};
```

ARRAY OF ARRAYS

Different arrays can have different number of elements – more flexibility

Elements of a single array are contiguous but different arrays could be located far off in memory

2D arrays vs Array of arrays

2D ARRAYS

Number of elements in each row is the same

All elements of 2D array are located contiguously in memory

Easier to initialize

```
int mat[3][5] = { {1,2}, {3},
{4,5,6},{7,8,9,10,11},{-1,2,3,4}};
```

ARRAY OF ARRAYS

Different arrays can have different number of elements – more flexibility

Elements of a single array are contiguous but different arrays could be located far off in memory

Have to be initialized element by element

2D arrays vs Array of arrays

2D ARRAYS

Number of elements in each row is the same

All elements of 2D array are located contiguously in memory

Easier to initialize

```
int mat[3][5] = { {1,2}, {3},
{4,5,6},{7,8,9,10,11},{-1,2,3,4}};
```

Very convenient ☺

ARRAY OF ARRAYS

Different arrays can have different number of elements – more flexibility

Elements of a single array are contiguous but different arrays could be located far off in memory

Have to be initialized element by element

2D arrays vs Array of arrays

2D ARRAYS

Number of elements in each row is the same

All elements of 2D array are located contiguously in memory

Easier to initialize

int mat[3][5] = { {1,2}, {3}, {4,5,6},{7,8,9,10,11},{-1,2,3,4}};

Very convenient ☺

ARRAY OF ARRAYS

Different arrays can have different number of elements – more flexibility

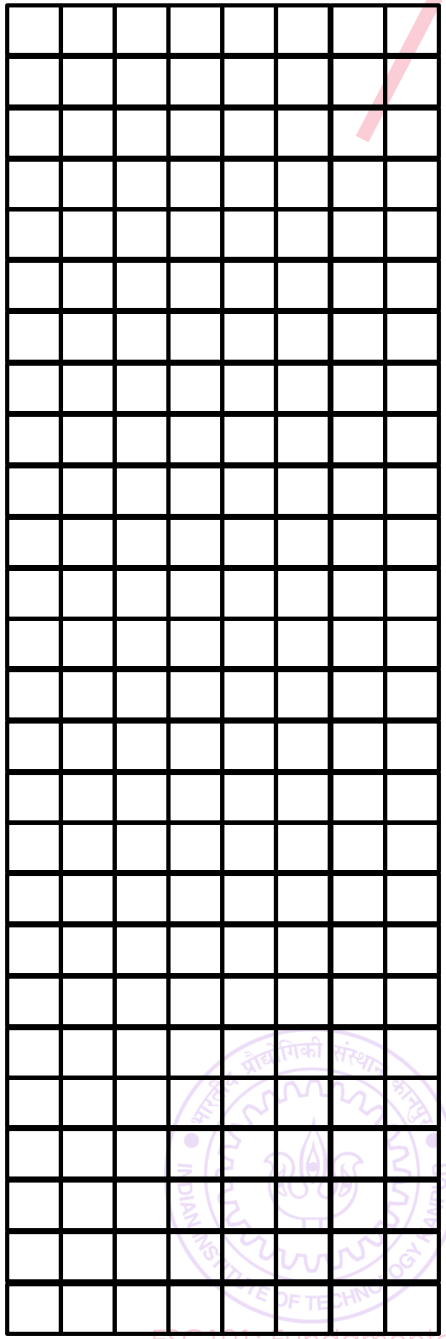
Elements of a single array are contiguous but different arrays could be located far off in memory

Have to be initialized element by element

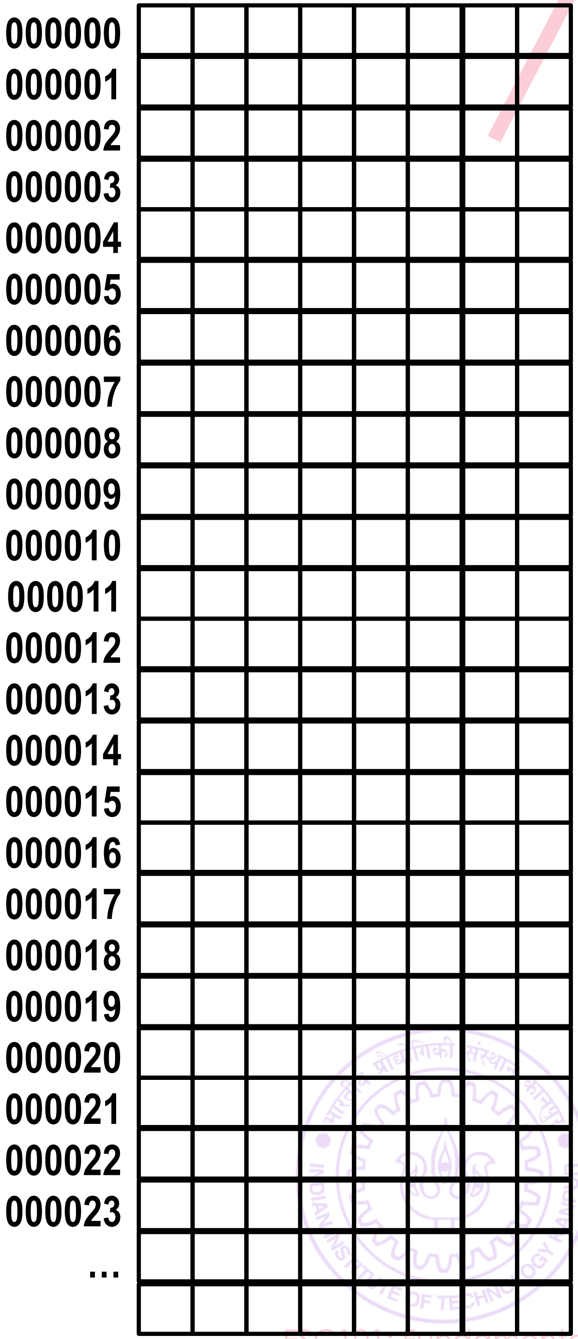
More power, responsibility

Memory layout of 2D arrays

Memory layout of 2D arrays

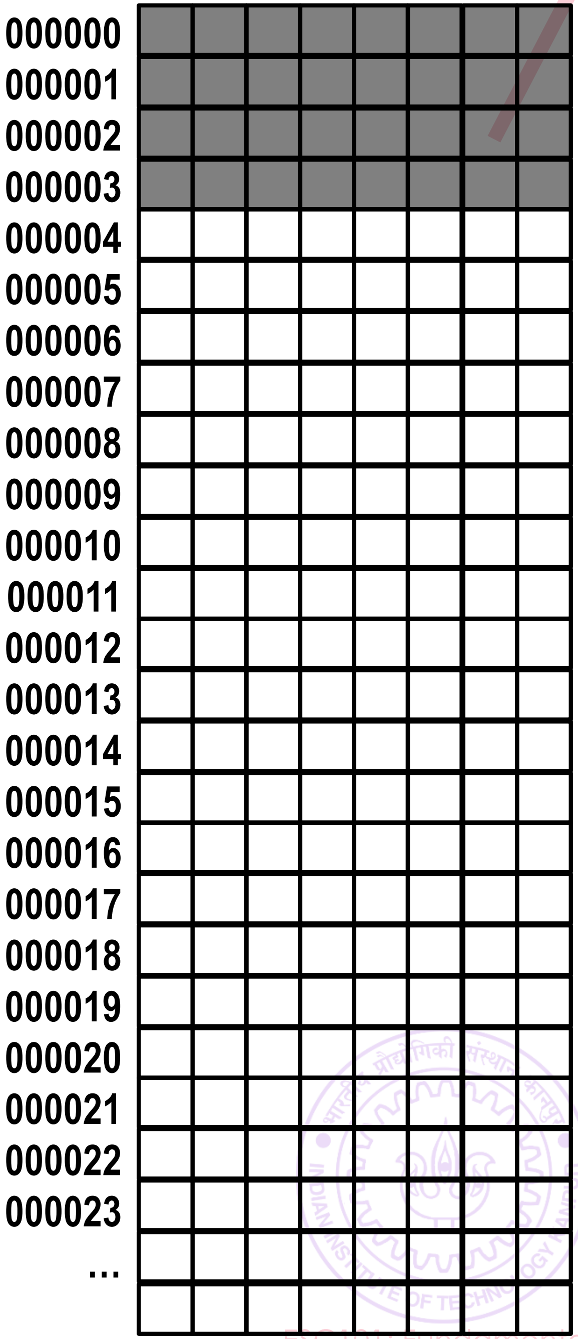


Memory layout of 2D arrays



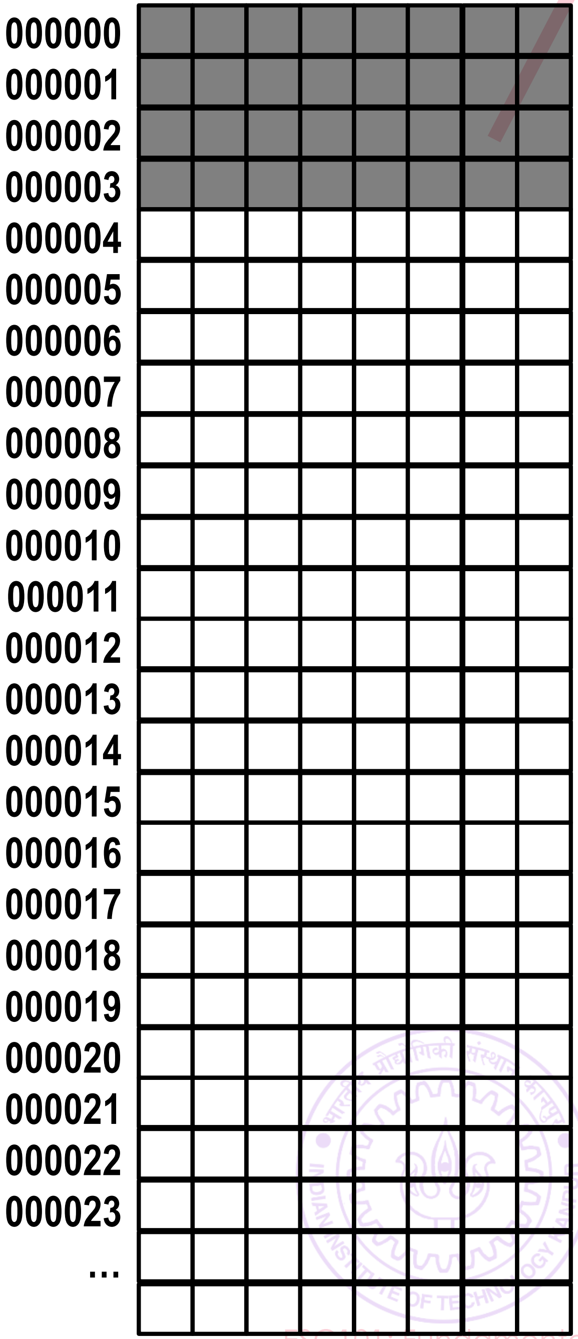
000000
000001
000002
000003
000004
000005
000006
000007
000008
000009
000010
000011
000012
000013
000014
000015
000016
000017
000018
000019
000020
000021
000022
000023
...

Memory layout of 2D arrays



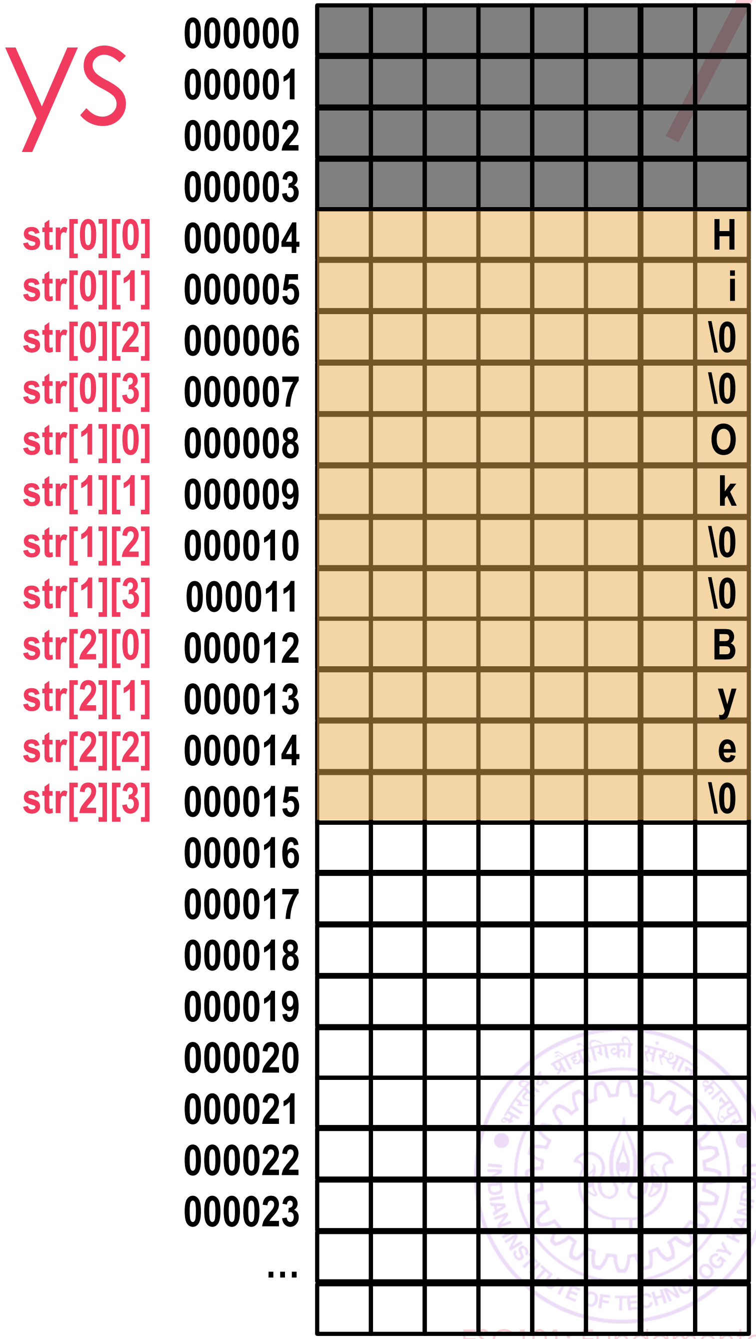
Memory layout of 2D arrays

```
char str[3][4] = {"Hi","Ok","Bye"};
```



Memory layout of 2D arrays

```
char str[3][4] = {"Hi","Ok","Bye"};
```

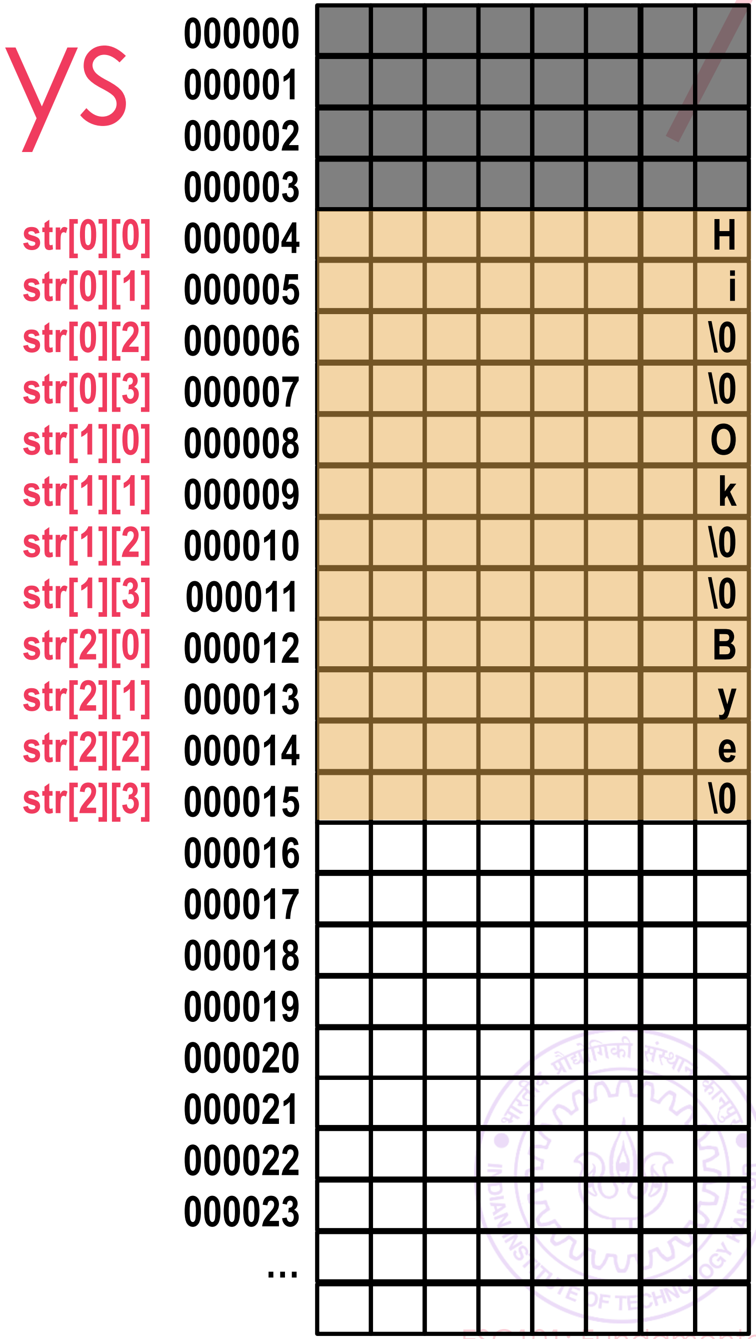


	000000	
	000001	
	000002	
	000003	
str[0][0]	000004	H
str[0][1]	000005	i
str[0][2]	000006	\0
str[0][3]	000007	\0
str[1][0]	000008	O
str[1][1]	000009	k
str[1][2]	000010	\0
str[1][3]	000011	\0
str[2][0]	000012	B
str[2][1]	000013	y
str[2][2]	000014	e
str[2][3]	000015	\0
	000016	
	000017	
	000018	
	000019	
	000020	
	000021	
	000022	
	000023	
	...	

Memory layout of 2D arrays

char str[3][4] = {"Hi","Ok","Bye"};

Location of the str pointer not shown



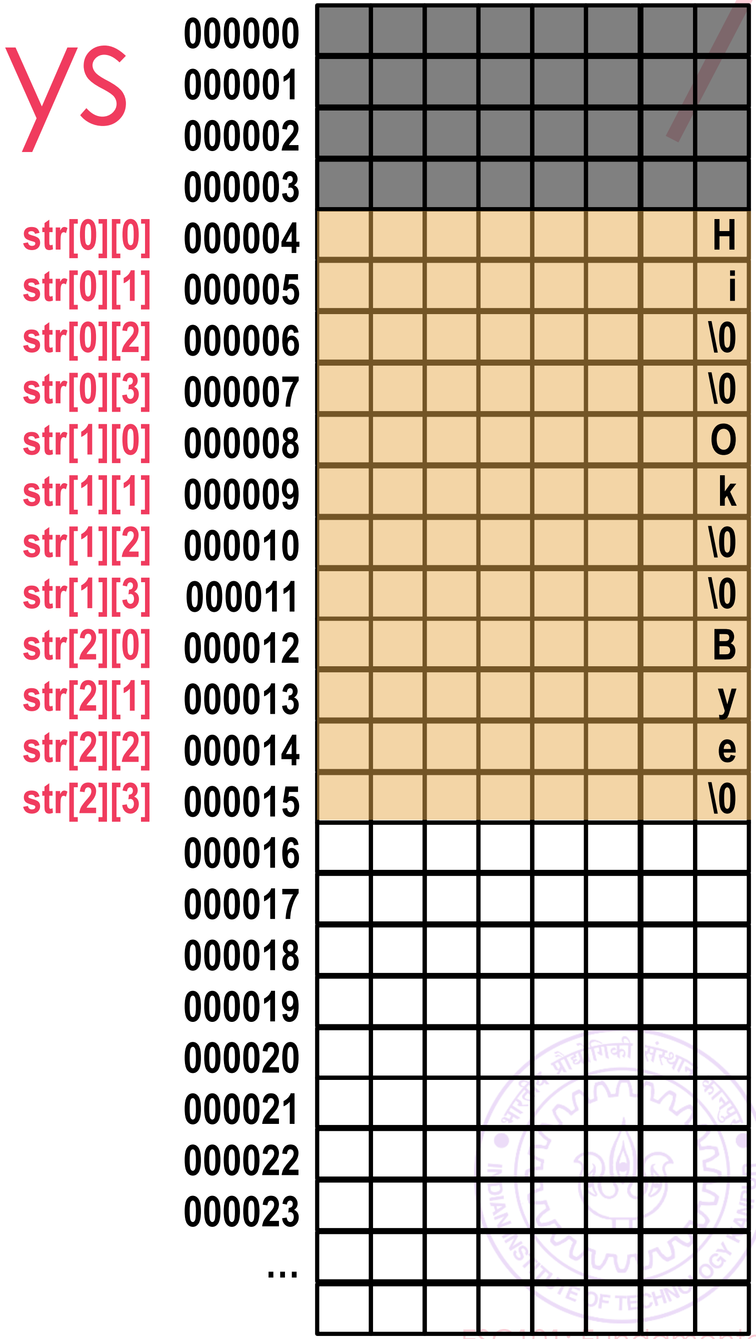
	000000	
	000001	
	000002	
	000003	
str[0][0]	000004	H
str[0][1]	000005	i
str[0][2]	000006	\0
str[0][3]	000007	\0
str[1][0]	000008	O
str[1][1]	000009	k
str[1][2]	000010	\0
str[1][3]	000011	\0
str[2][0]	000012	B
str[2][1]	000013	y
str[2][2]	000014	e
str[2][3]	000015	\0
	000016	
	000017	
	000018	
	000019	
	000020	
	000021	
	000022	
	000023	
	...	

Memory layout of 2D arrays

```
char str[3][4] = {"Hi","Ok","Bye"};
```

Location of the str pointer not shown

First all elements of row 0 stored in continuous sequence



	000000	
	000001	
	000002	
	000003	
str[0][0]	000004	H
str[0][1]	000005	i
str[0][2]	000006	\0
str[0][3]	000007	\0
str[1][0]	000008	O
str[1][1]	000009	k
str[1][2]	000010	\0
str[1][3]	000011	\0
str[2][0]	000012	B
str[2][1]	000013	y
str[2][2]	000014	e
str[2][3]	000015	\0
	000016	
	000017	
	000018	
	000019	
	000020	
	000021	
	000022	
	000023	
	...	

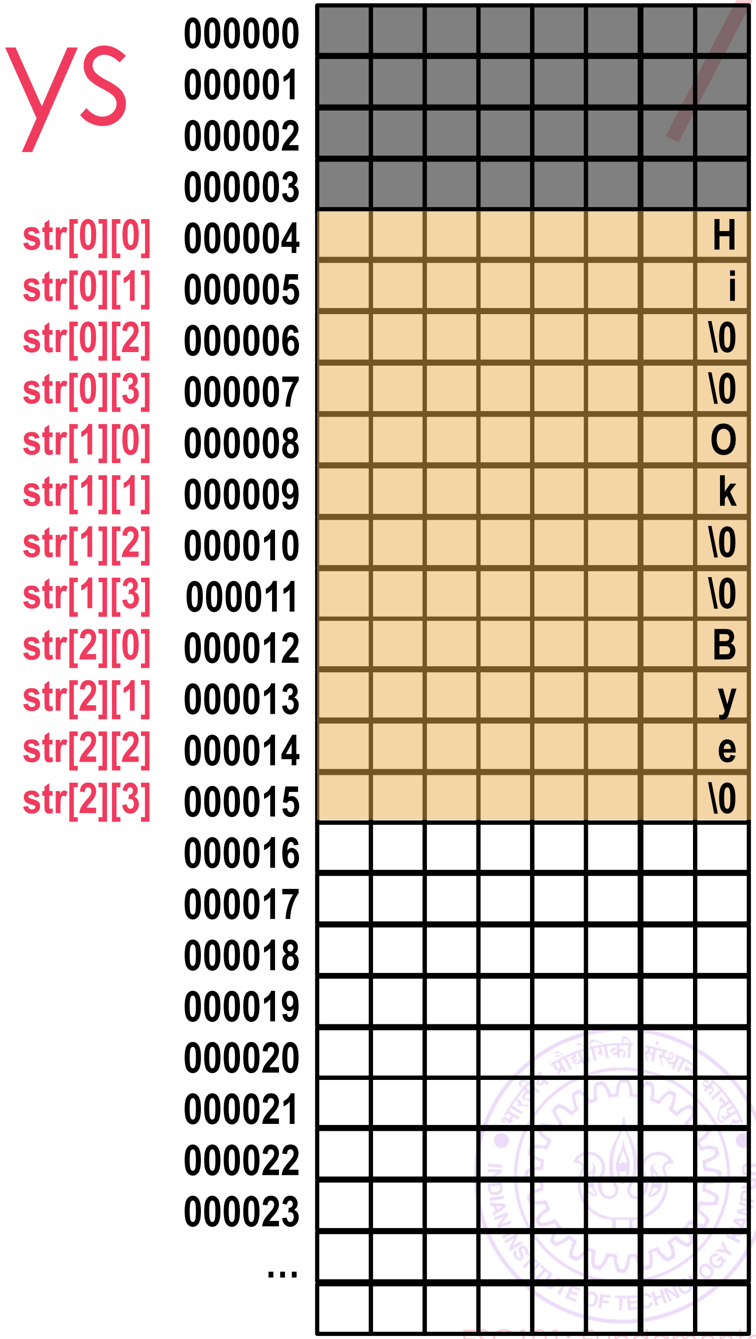
Memory layout of 2D arrays

char str[3][4] = {"Hi","Ok","Bye"};

Location of the str pointer not shown

First all elements of row 0 stored in continuous sequence

Then without breaking sequence, all elements of row 1 stored and so on



	000000	
	000001	
	000002	
	000003	
str[0][0]	000004	H
str[0][1]	000005	i
str[0][2]	000006	\0
str[0][3]	000007	\0
str[1][0]	000008	O
str[1][1]	000009	k
str[1][2]	000010	\0
str[1][3]	000011	\0
str[2][0]	000012	B
str[2][1]	000013	y
str[2][2]	000014	e
str[2][3]	000015	\0
	000016	
	000017	
	000018	
	000019	
	000020	
	000021	
	000022	
	000023	
	...	

Memory layout of 2D arrays

char str[3][4] = {"Hi","Ok","Bye"};

Location of the str pointer not shown

First all elements of row 0 stored in continuous sequence

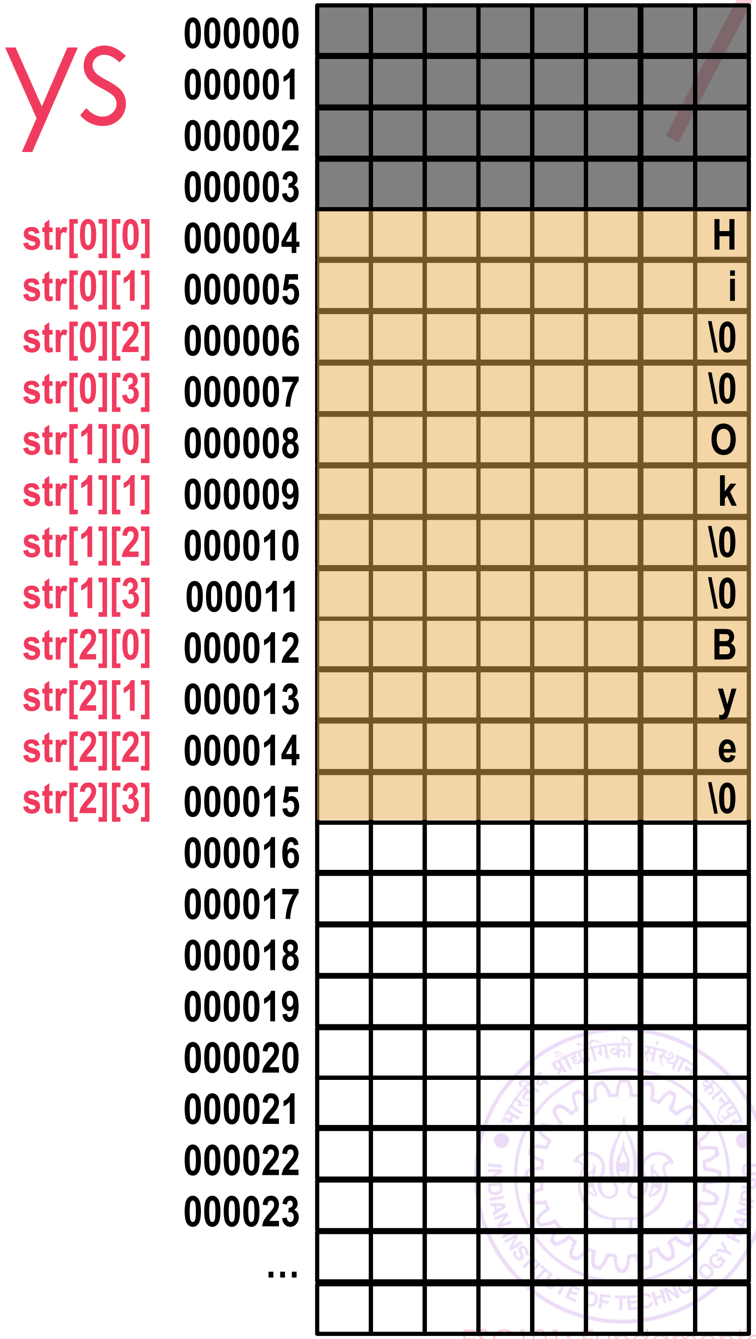
Then without breaking sequence, all elements of row 1 stored and so on

char* ptr = *str; // ptr points to str[0][0]

ptr += 4; // ptr now points to str[1][0]

ptr += 4; // ptr now points to str[2][0]

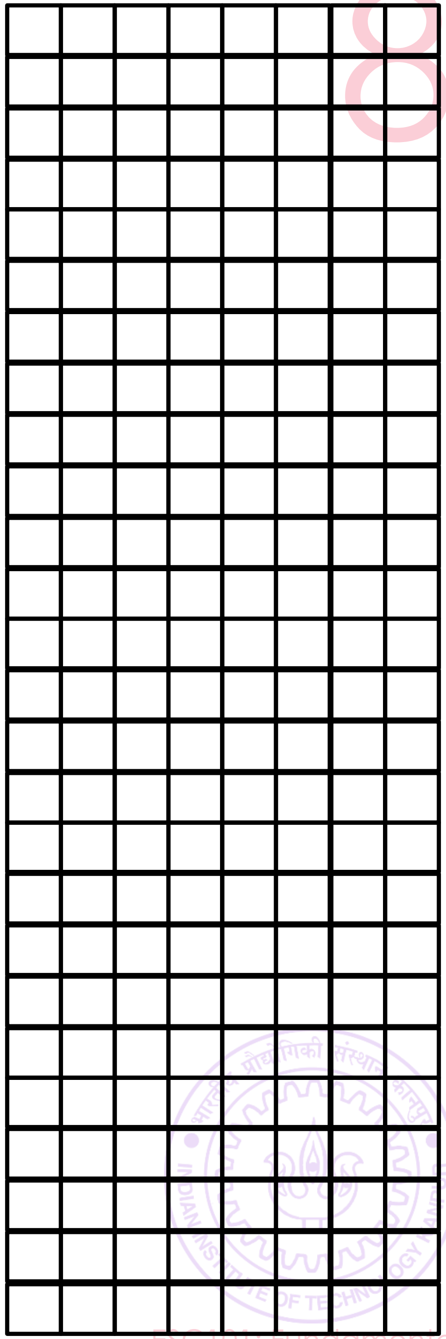
ptr += 1; // ptr now points to str[2][1]



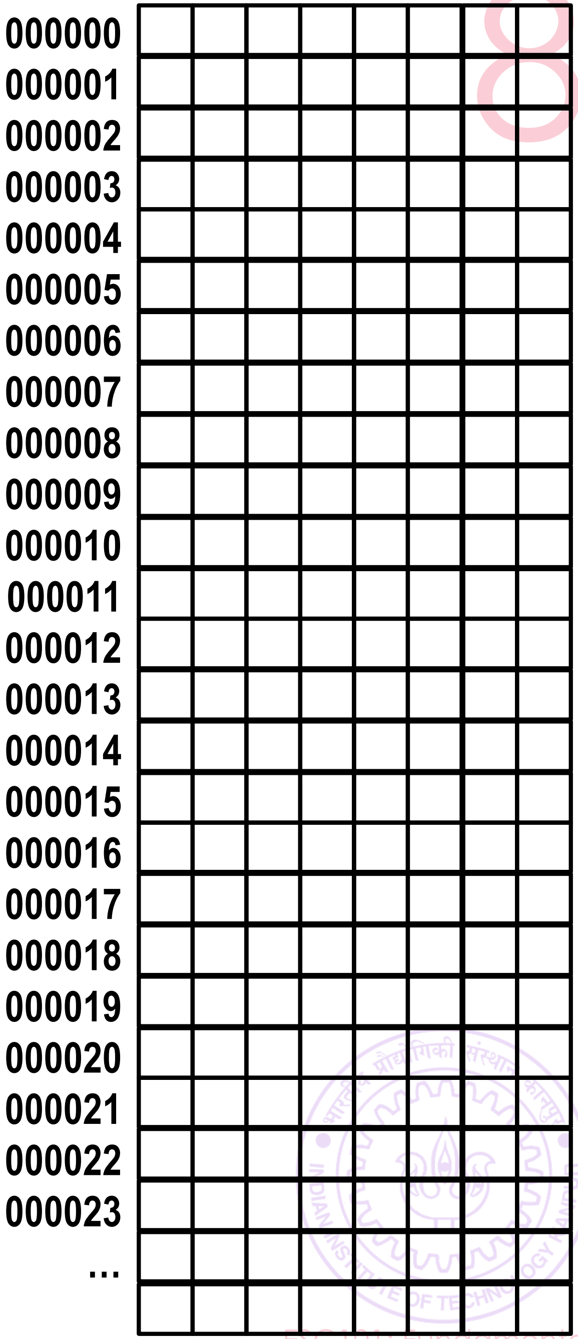
	000000	
	000001	
	000002	
	000003	
str[0][0]	000004	H
str[0][1]	000005	i
str[0][2]	000006	\0
str[0][3]	000007	\0
str[1][0]	000008	O
str[1][1]	000009	k
str[1][2]	000010	\0
str[1][3]	000011	\0
str[2][0]	000012	B
str[2][1]	000013	y
str[2][2]	000014	e
str[2][3]	000015	\0
	000016	
	000017	
	000018	
	000019	
	000020	
	000021	
	000022	
	000023	
	…	

Layout of arrays of arrays

Layout of arrays of arrays

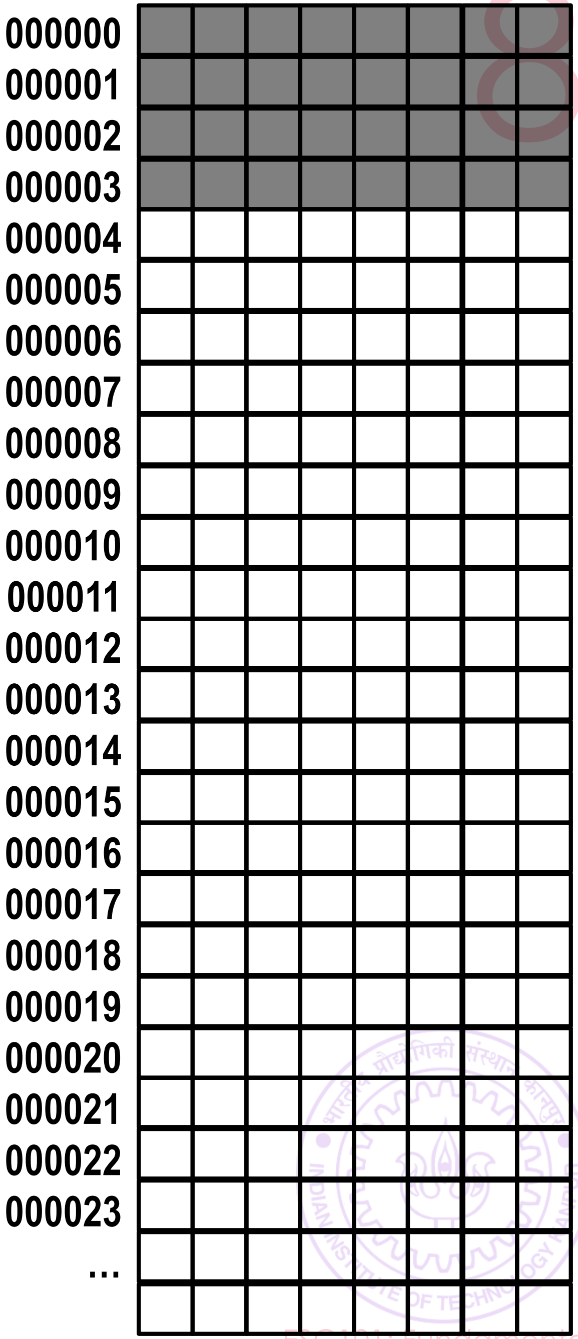


Layout of arrays of arrays



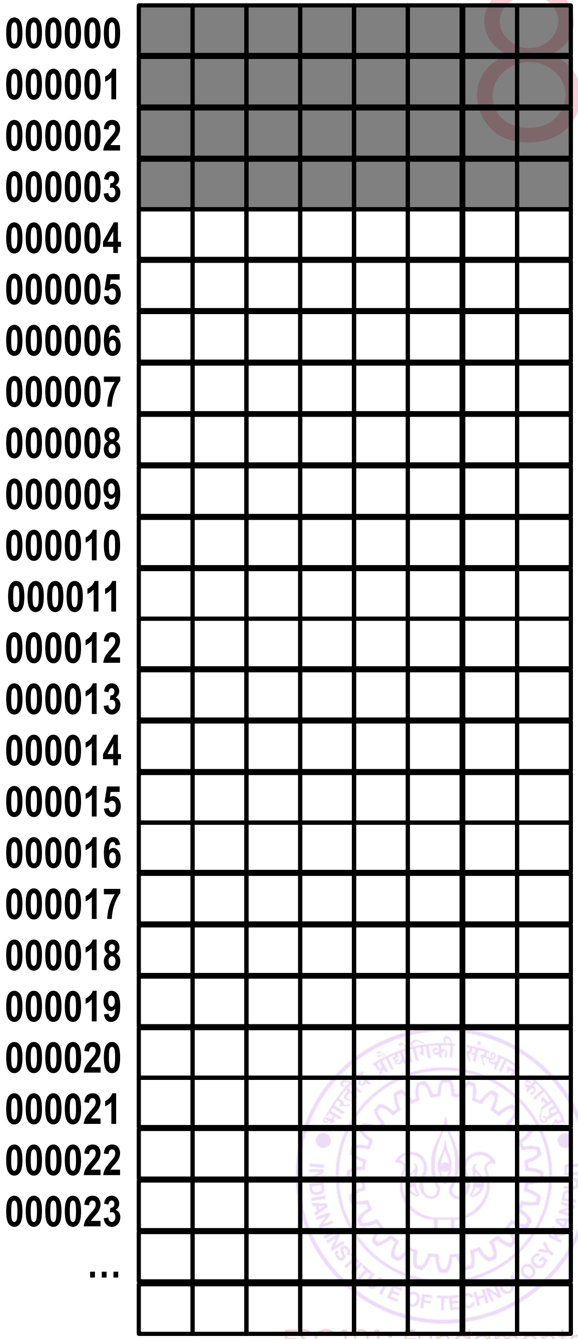
000000
000001
000002
000003
000004
000005
000006
000007
000008
000009
000010
000011
000012
000013
000014
000015
000016
000017
000018
000019
000020
000021
000022
000023
...

Layout of arrays of arrays



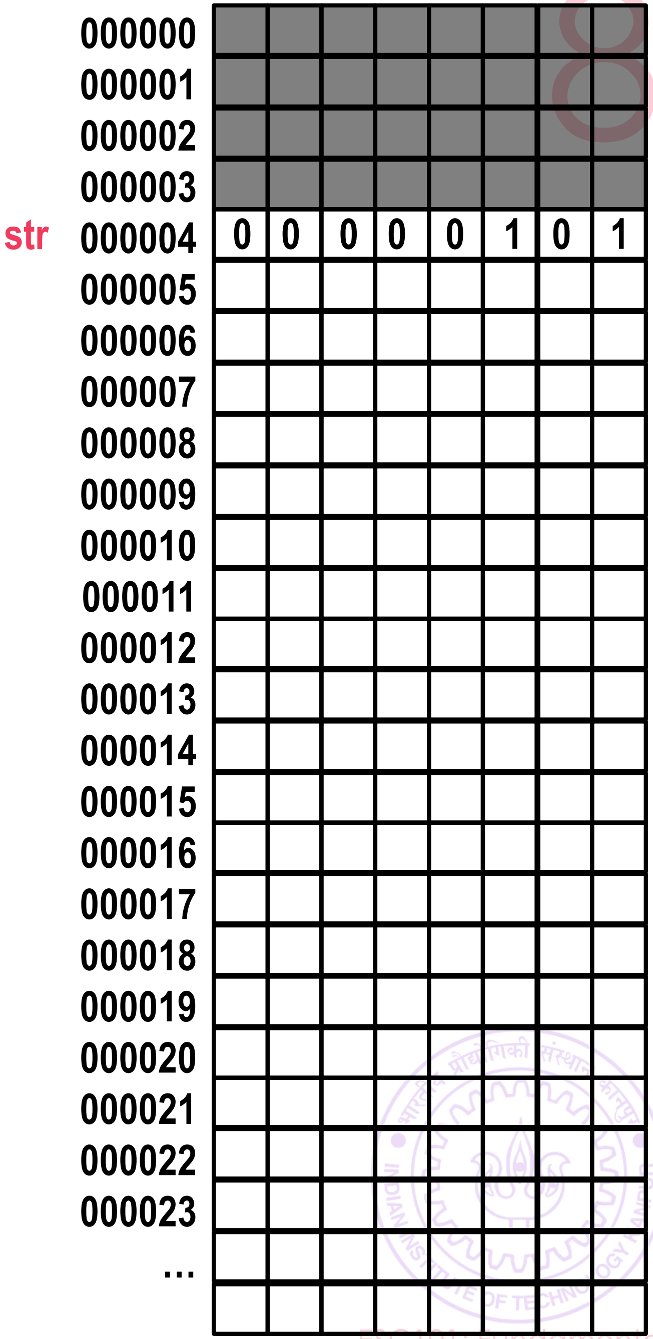
Layout of arrays of arrays

```
char **str = (char**)malloc(3*sizeof(char*));
str[0] = (char*)malloc(4*sizeof(char));
str[1] = (char*)malloc(4*sizeof(char));
```



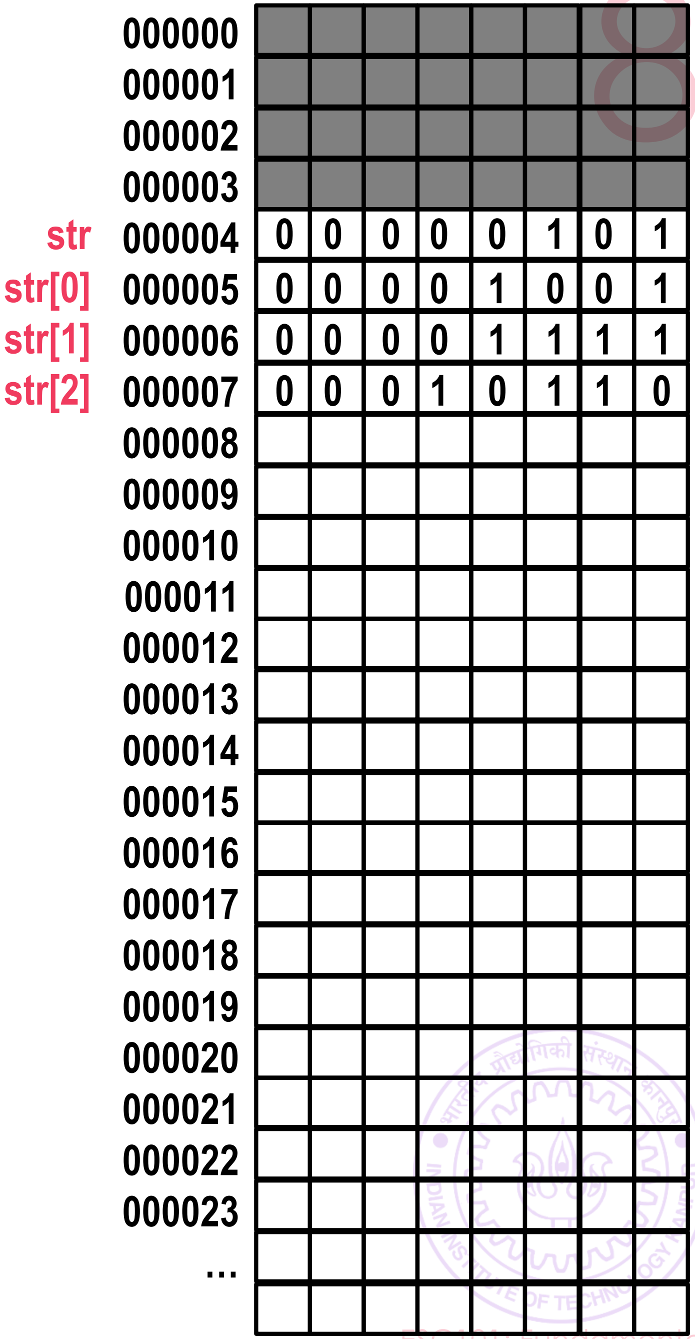
Layout of arrays of arrays

```
char **str = (char**)malloc(3*sizeof(char*));
str[0] = (char*)malloc(4*sizeof(char));
str[1] = (char*)malloc(4*sizeof(char));
```



Layout of arrays of arrays

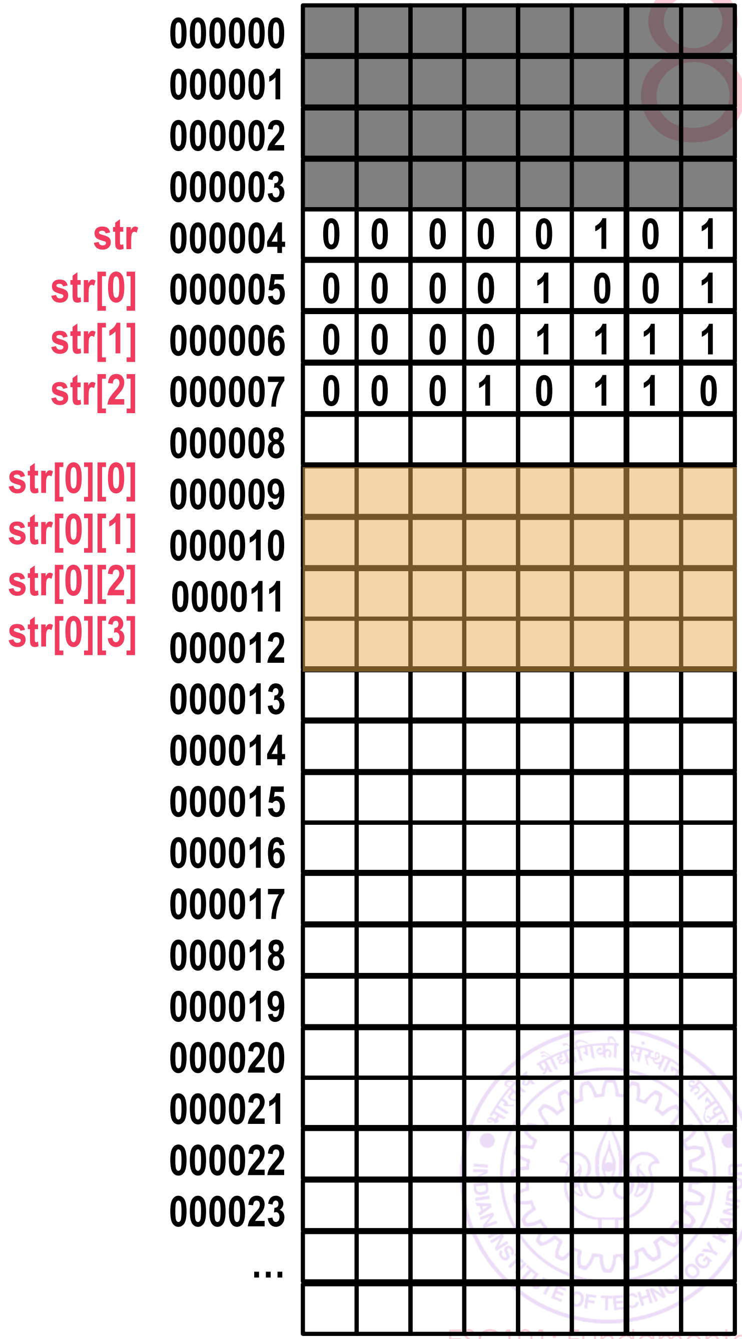
```
char **str = (char**)malloc(3*sizeof(char*));
str[0] = (char*)malloc(4*sizeof(char));
str[1] = (char*)malloc(4*sizeof(char));
```



	000000								
	000001								
	000002								
	000003								
str	000004	0	0	0	0	0	1	0	1
str[0]	000005	0	0	0	0	1	0	0	1
str[1]	000006	0	0	0	0	1	1	1	1
str[2]	000007	0	0	0	1	0	1	1	0
	000008								
	000009								
	000010								
	000011								
	000012								
	000013								
	000014								
	000015								
	000016								
	000017								
	000018								
	000019								
	000020								
	000021								
	000022								
	000023								
	...								

Layout of arrays of arrays

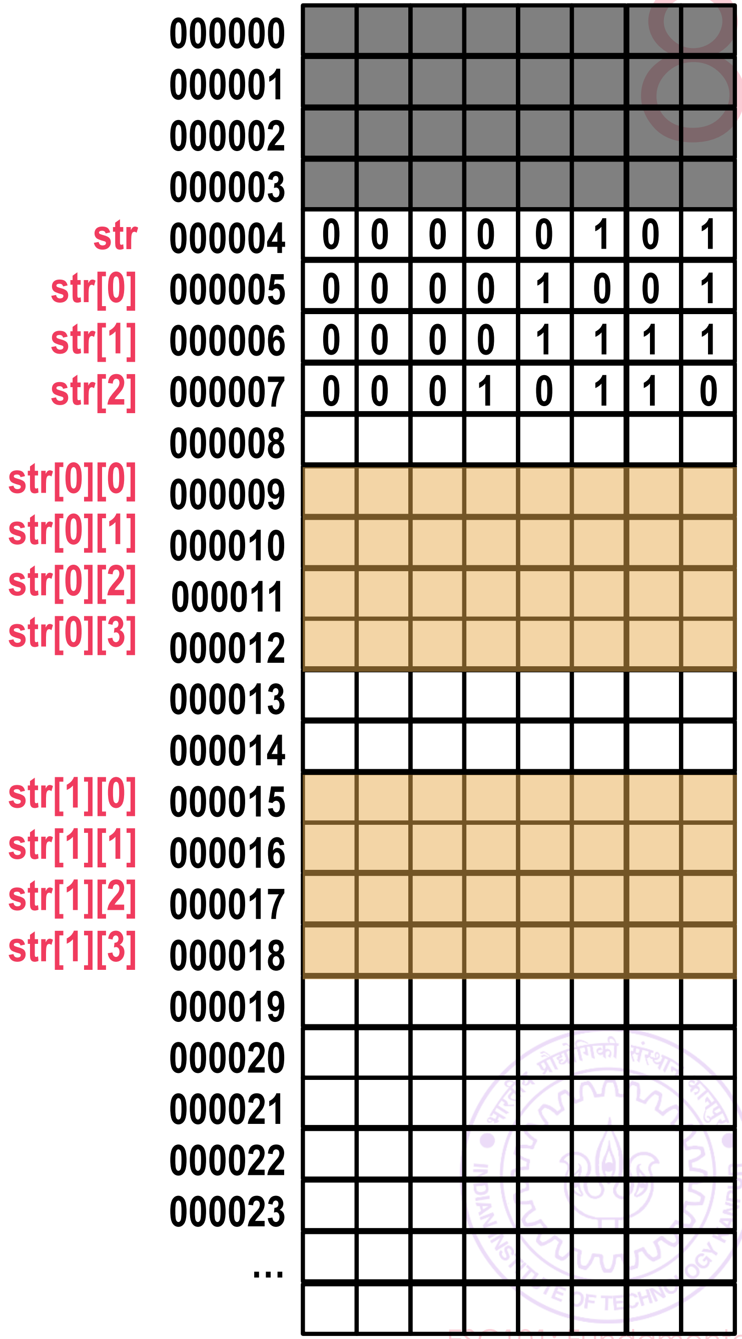
```
char **str = (char**)malloc(3*sizeof(char*));
str[0] = (char*)malloc(4*sizeof(char));
str[1] = (char*)malloc(4*sizeof(char));
```



	000000								
	000001								
	000002								
	000003								
str	000004	0	0	0	0	0	1	0	1
str[0]	000005	0	0	0	0	1	0	0	1
str[1]	000006	0	0	0	0	1	1	1	1
str[2]	000007	0	0	0	1	0	1	1	0
	000008								
str[0][0]	000009								
str[0][1]	000010								
str[0][2]	000011								
str[0][3]	000012								
	000013								
	000014								
	000015								
	000016								
	000017								
	000018								
	000019								
	000020								
	000021								
	000022								
	000023								
	...								

Layout of arrays of arrays

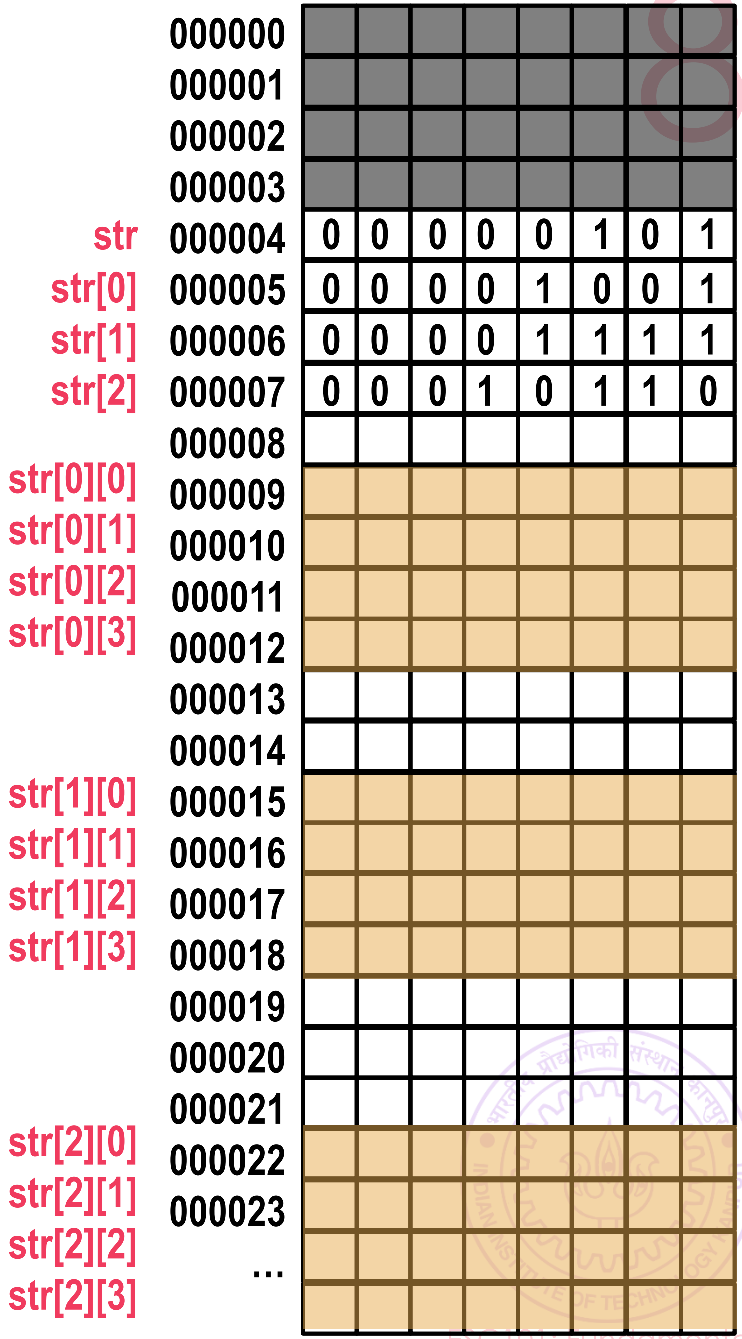
```
char **str = (char**)malloc(3*sizeof(char*));
str[0] = (char*)malloc(4*sizeof(char));
str[1] = (char*)malloc(4*sizeof(char));
```



	000000								
	000001								
	000002								
	000003								
str	000004	0	0	0	0	0	1	0	1
str[0]	000005	0	0	0	0	1	0	0	1
str[1]	000006	0	0	0	0	1	1	1	1
str[2]	000007	0	0	0	1	0	1	1	0
	000008								
str[0][0]	000009								
str[0][1]	000010								
str[0][2]	000011								
str[0][3]	000012								
	000013								
	000014								
str[1][0]	000015								
str[1][1]	000016								
str[1][2]	000017								
str[1][3]	000018								
	000019								
	000020								
	000021								
	000022								
	000023								
	...								

Layout of arrays of arrays

```
char **str = (char**)malloc(3*sizeof(char*));
str[0] = (char*)malloc(4*sizeof(char));
str[1] = (char*)malloc(4*sizeof(char));
```

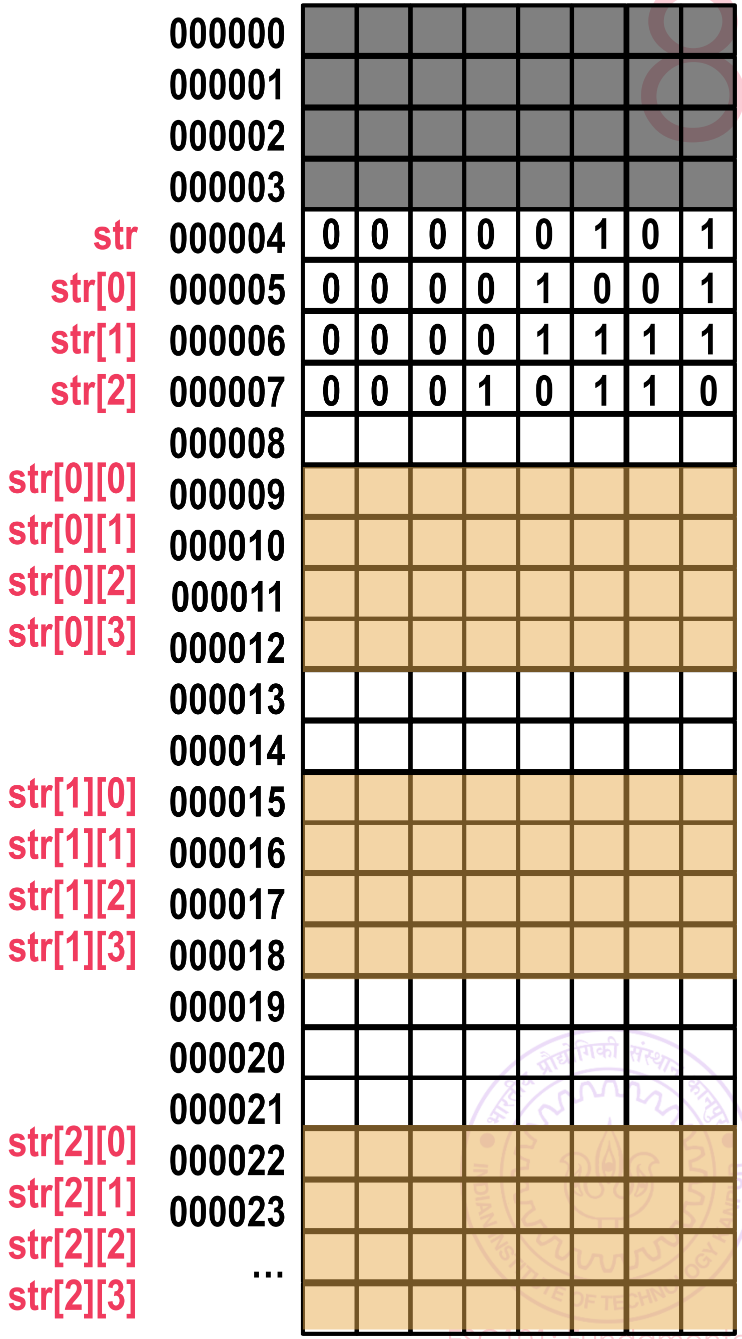


	000000								
	000001								
	000002								
	000003								
str	000004	0	0	0	0	0	1	0	1
str[0]	000005	0	0	0	0	1	0	0	1
str[1]	000006	0	0	0	0	1	1	1	1
str[2]	000007	0	0	0	1	0	1	1	0
	000008								
str[0][0]	000009								
str[0][1]	000010								
str[0][2]	000011								
str[0][3]	000012								
	000013								
	000014								
str[1][0]	000015								
str[1][1]	000016								
str[1][2]	000017								
str[1][3]	000018								
	000019								
	000020								
	000021								
str[2][0]	000022								
str[2][1]	000023								
str[2][2]	...								
str[2][3]									

Layout of arrays of arrays

```
char **str = (char**)malloc(3*sizeof(char*));
str[0] = (char*)malloc(4*sizeof(char));
str[1] = (char*)malloc(4*sizeof(char));
```

Element within a single array always stored in sequence



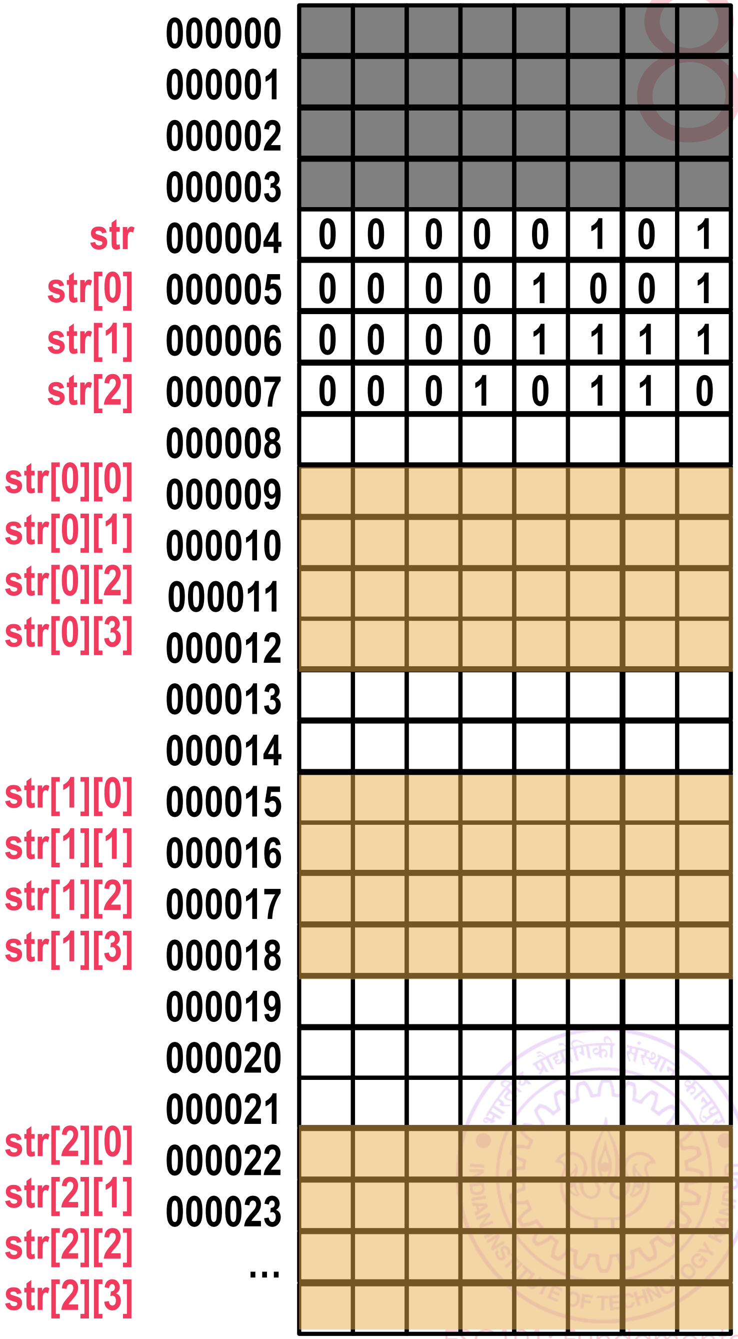
	000000								
	000001								
	000002								
	000003								
str	000004	0	0	0	0	0	1	0	1
str[0]	000005	0	0	0	0	1	0	0	1
str[1]	000006	0	0	0	0	1	1	1	1
str[2]	000007	0	0	0	1	0	1	1	0
	000008								
str[0][0]	000009								
str[0][1]	000010								
str[0][2]	000011								
str[0][3]	000012								
	000013								
	000014								
str[1][0]	000015								
str[1][1]	000016								
str[1][2]	000017								
str[1][3]	000018								
	000019								
	000020								
	000021								
str[2][0]	000022								
str[2][1]	000023								
str[2][2]	...								
str[2][3]									

Layout of arrays of arrays

```
char **str = (char**)malloc(3*sizeof(char*));
str[0] = (char*)malloc(4*sizeof(char));
str[1] = (char*)malloc(4*sizeof(char));
```

Element within a single array always stored in sequence

Different arrays may be stored far away from each other



	000000								
	000001								
	000002								
	000003								
str	000004	0	0	0	0	0	1	0	1
str[0]	000005	0	0	0	0	1	0	0	1
str[1]	000006	0	0	0	0	1	1	1	1
str[2]	000007	0	0	0	1	0	1	1	0
	000008								
str[0][0]	000009								
str[0][1]	000010								
str[0][2]	000011								
str[0][3]	000012								
	000013								
	000014								
str[1][0]	000015								
str[1][1]	000016								
str[1][2]	000017								
str[1][3]	000018								
	000019								
	000020								
	000021								
str[2][0]	000022								
str[2][1]	000023								
str[2][2]	...								
str[2][3]									

Mr C takes a Math Lesson

Mr C takes a Math Lesson

Mathematics is full of functions - we define more powerful functions using simple functions

Mr C takes a Math Lesson

Mathematics is full of functions - we define more powerful functions using simple functions

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

Mr C takes a Math Lesson

Mathematics is full of functions - we define more powerful functions using simple functions

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

sin(x) itself can be defined w.r.t addition, factorial, division

Mr C takes a Math Lesson

Mathematics is full of functions - we define more powerful functions using simple functions

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

sin(x) itself can be defined w.r.t addition, factorial, division

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

Mr C takes a Math Lesson

Mathematics is full of functions - we define more powerful functions using simple functions

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

sin(x) itself can be defined w.r.t addition, factorial, division

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

Factorial can be defined in terms of multiplication ☺

Mr C takes a Math Lesson

Mathematics is full of functions - we define more powerful functions using simple functions

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

sin(x) itself can be defined w.r.t addition, factorial, division

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

Factorial can be defined in terms of multiplication ☺

Multiplication can be defined in terms of addition ☺

Mr C takes a Math Lesson

Mathematics is full of functions - we define more p functions using simple functions

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

sin(x) itself can be defined w.r.t addition, factorial, division

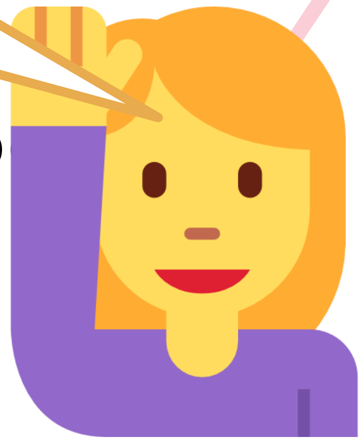
$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

Factorial can be defined in terms of multiplication ☺

Multiplication can be defined in terms of addition ☺

Mr C tak

Although we can write tan(x) in terms of addition, subtraction, multiplication and division, we almost never do that. We always write tan(x) = sin(x)/cos(x)



Mathematics is full of functions - we define more p functions using simple functions

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

sin(x) itself can be defined w.r.t addition, factorial, division

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

Factorial can be defined in terms of multiplication ☺

Multiplication can be defined in terms of addition ☺

Mr C tak

Although we can write tan(x) in terms of addition, subtraction, multiplication and division, we almost never do that. We always write tan(x) = sin(x)/cos(x)

Mathematics is full of functions - we define more p functions using simple functions

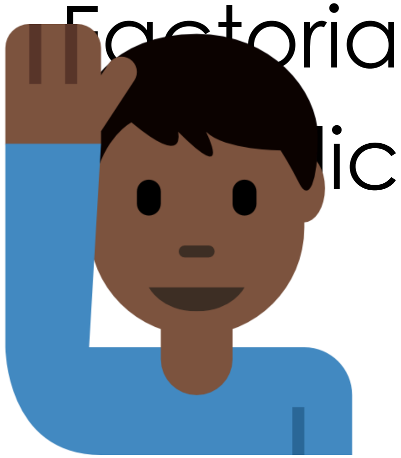
$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

sin(x) itself can be defined w.r.t addition, factorial, division

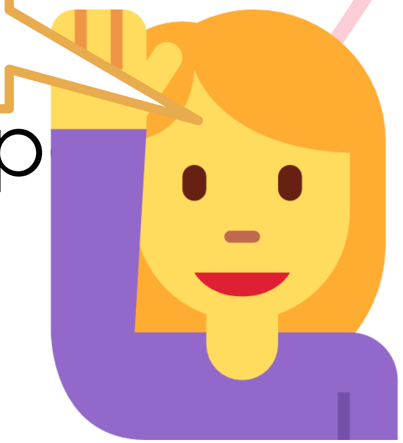
$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

Factorial can be defined in terms of multiplication ☺

lication can be defined in terms of addition ☺



Mr C tak



Although we can write tan(x) in terms of addition, subtraction, multiplication and division, we almost never do that. We always write tan(x) = sin(x)/cos(x)

Mathematics is full of functions - we define more p functions using simple functions

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

sin(x) itself can be defined w.r.t addition, factorial, division

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

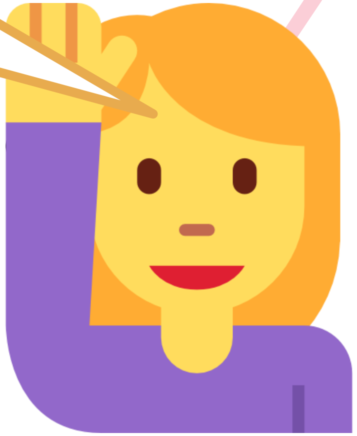
Facto ltiplication ☺

f addition ☺

Helps us write much cleaner math expressions, as well as we do not make mistakes very often

Mr C tak

Although we can write tan(x) in terms of addition, subtraction, multiplication and division, we almost never do that. We always write tan(x) = sin(x)/cos(x)



Mathematics is full of functions - we define more p
functions using simple functions

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

sin(x) itself can be defined w.r.t addition, factorial, division

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

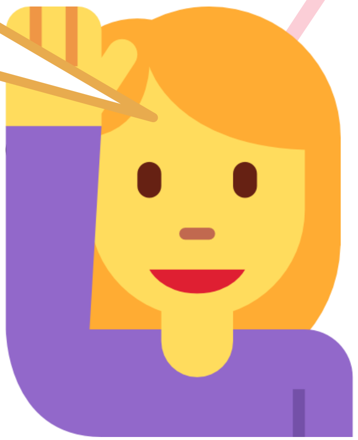
Facto ltiplication ☺

f addition ☺

Helps us write much cleaner math expressions, as well as we do not make mistakes very often

Mr C tak

Although we can write tan(x) in terms of addition, subtraction, multiplication and division, we almost never do that. We always write tan(x) = sin(x)/cos(x)



Mathematics is full of functions - we define more p
functions using simple functions

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

sin(x) itself can be defined w.r.t addition, factorial, division

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

Facto ltiplication ☺

f addition ☺

Helps us write much cleaner math expressions, as well as we do not make mistakes very often

I too allow you to write your own functions for your comfort and to make your code easier to read and easier to debug!