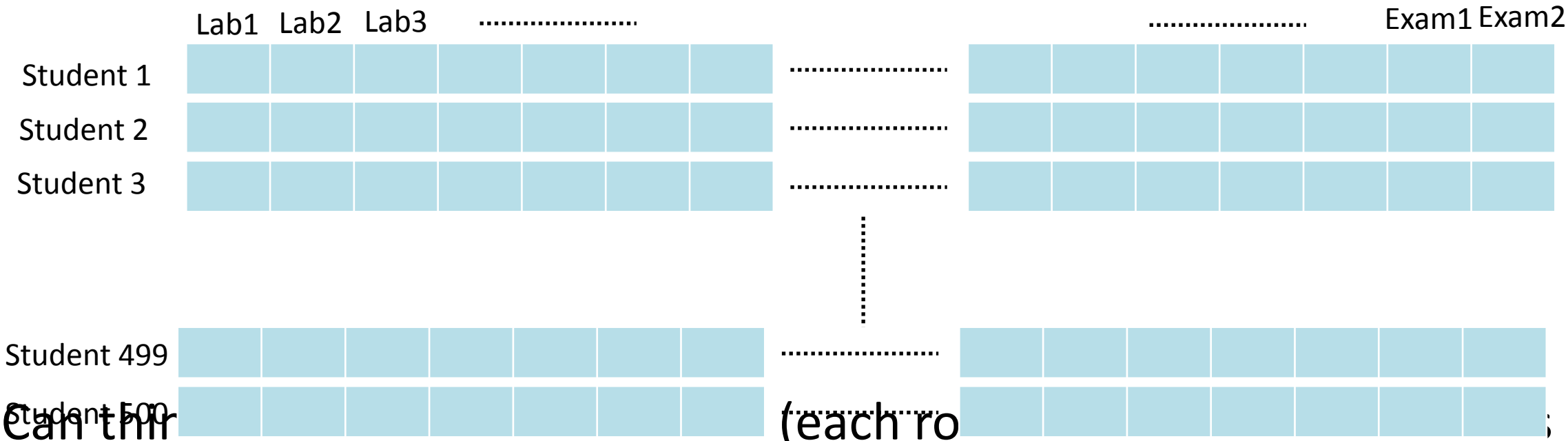
Multidimensional arrays

ESC101: Fundamentals of Computing

Nisheeth

Multi-dimensional Array: Example

- Marks of all ESC101 students in various labs/quizzes/exams



- Can think of it as a 2D array (each row is a 1D array)
- A 2D array is equivalent to a matrix (rows and columns)
- Can also have 3D or higher-dimensional arrays

Multi-dimensional Array in C

Declaration of a 2D array:

```
double  mat[5][6];
```

```
int mat[5][6];
```

```
float mat[5][6];
```

mat is a 5 X 6 matrix of doubles (or ints or floats). It has 5 rows, each row has 6 columns, each entry is of the type double (or int or float in the other two examples).

2.1	1.0	-0.11	-0.87	31.5	11.4
-3.2	-2.5	1.678	4.5	0.001	1.89
7.889	3.333	0.667	1.1	1.0	-1.0
-4.56	-21.5	1.0e7	-1.0e-9	1.0e-15	-5.78
45.7	26.9	-0.001	1000.09	1.0e15	1.0

Declaration of Multi-dimensional Array

number of rows

number of columns

Two-dim array `type array_name[size1][size2];`

Size of dimension 1

Size of dimension 2

Size of dimension 3

Three-dim array `type array_name[size1][size2][size3];`

Size of dimension 1

Size of dimension 2

Size of dimension 2

N-dim array `type array_name[size1][size2][size3]... [sizeN]`

Size of dimension N

Accessing Elements of a 2D Array (Printing)

- (i,j)th member of mat: mat[i][j] (mathematics: mat(i,j)).
- The row and column index start at 0 (not 1).
- The following program prints the input matrix mat[5][6].

```
int i,j;
    for (i=0; i < 5; i=i+1) {      /* prints the ith row i = 0..4. */
      for (j=0; j < 6; j = j+1) {  /* In each row, prints each of
         printf("%f ", mat[i][j]);    the six columns  j=0..5 */
      }
      printf("\n");  /* prints a newline after each row */
    }
```

Accessing Element of a 2D Array (Reading)

- Code for reading the matrix mat[5][6] from the terminal.
- The address of the i,j th matrix element is &mat[i][j].
- This works without parentheses since the array indexing operator [] has higher precedence than &.

```
int i,j;
   for (i=0; i < 5; i=i+1) {
       for (j=0; j < 6; j = j+1) {
         scanf("%f ", &mat[i][j]);
       }
    }
```

/* read the ith row i = 0..4. */

/* In each row, read each of the six columns j=0..5 */

scanf with %f option will skip over whitespace.

So it really doesn't matter whether the entire input is given in 5 rows of 6 doubles in a row or all 30 doubles in a single line, etc..

Accessing Element of a 2D Array (Reading)

```
int i,j;
   for (i=0; i < 5; i=i+1) {
       for (j=0; j < 6; j = j+1) {
         scanf("%f ", &mat[i][j]);
       }
    }
```

/* read the ith row i = 0..4. */

/* In each row, read each of the six columns j=0..5 */

Could I change declaration to mat[6][5]? Would it mean the same? Or mat[10][3]?

That would NOT be correct. It would change the way elements of mat are addressed. We will discuss this in detail later.

Multi-dimensional Array

- Easy to think of it as an array of arrays
- It means: An array in which each element is another array
- Can think of the 2D array below as containing 5 1D arrays

Array 1 →
Array 2 →
Array 3 →
Array 4 →
Array 5 →

2.1	1.0	-0.11	-0.87	31.5	11.4
-3.2	-2.5	1.678	4.5	0.001	1.89
7.889	3.333	0.667	1.1	1.0	-1.0
-4.56	-21.5	1.0e7	-1.0e-9	1.0e-15	-5.78
45.7	26.9	-0.001	1000.09	1.0e15	1.0

Multi-dimensional Array: Declaration

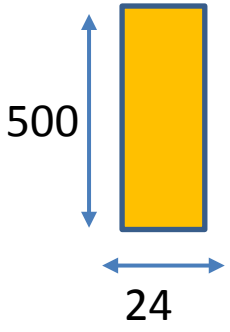
- We declare any multi-dimensional array as follows

type array_name[size1][size2]…[sizeK];

- Some examples

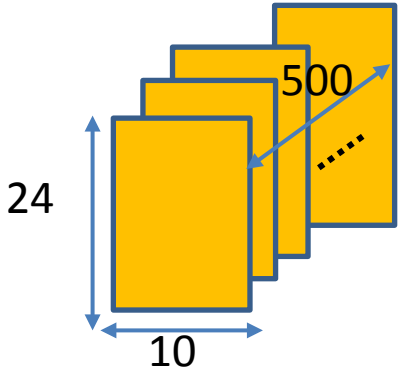
int arr[500][24];

Two-dim array
(dim1 = 500, dim2 = 24)
Can think of it as 500 one-dim arrays of size 24 each



int arr[500][24][10];

Three-dim array
(dim1 = 500, dim2 = 24, dim3 = 10)
Can think of it as 500 two-dim arrays of size 24x10 each



int arr[][24];

IMPORTANT: No need to specify the size of the first dimension (number of rows in 2D arrays). Must specify the sizes of the remaining dimensions (columns in case of the 2D array)

Accessing Elements of a Multi-dim Array

Keep in mind this basic picture of a 2D array whose name is a and which has 3 rows and 4 columns

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

For a 2D array, a[i][j] gives the element at row i and column j (i, j start with 0)

Likewise, for 3D array, a[i][j][k] gives the element at index i in dim 1, index j in dim 2 and index k in dim 3 and column j (i, j, and k start with 0)

Elements of higher-dimensional (>3) arrays are also accessed in a similar manner

Multi-dimensional Array: Initialization

- Declaration + init. of a 2D (3 rows, 4 columns) array of integers

```
int a[3][4] = {
  {-2, 1, 4, 3} ,   /*  row 0 */
  {-3 5, 7, -5} ,   /*  row 1 */
  {8, 2, 10, 6}   /*  row 2 */
};
```

Need not specify no. of rows

Must specify no. of columns

```
int a[][4] = {
  {-2, 1, 4, 3} ,   /*  row 0 */
  {-3, 5, 7, -5} ,   /*  row 1 */
```

If want to initialize later (not with declaration), then must do it one element at a time

- Values given row-wise (comma separated, row-1, row-2, so on)
- Values in each row must be enclosed in curly braces {}
- Can also initialize like this

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

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

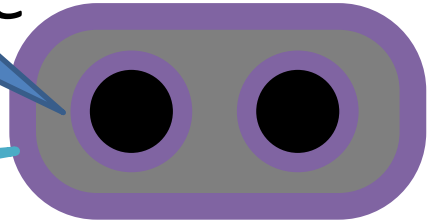
Both these are correct but less common ways (may also be a bit confusing)

Multi-dim. Array: Storage in Memory

- Let's look at a simple example of a

```
int a[3][4] = {
        {-2, 1, 4, 3} ,   /*  row 0 */
        {0, 5, 7, -5} ,   /*  row 1 */
        {8, 2, 10, 6}   /*  row 2 */
    };
```

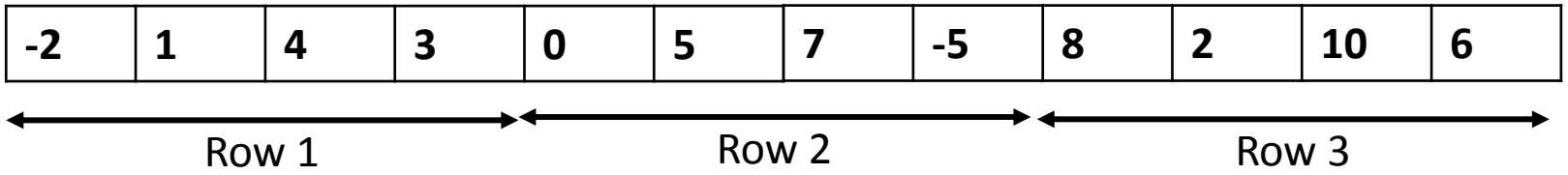
More on storage of arrays/multi-dim arrays when we study pointers



Basically, the 2D array is "flattened" row-wise and then stored in memory

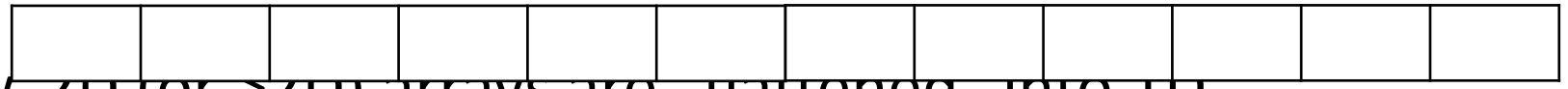
- First all the element of the first row are stored sequentially
- Then all the elements of the next row..
- Then all the elements of the row after..
- .. And so on..

This example is for 2D arrays. But 3D arrays are also stored similarly (all rows of a its first 2D array one by one, then repeat the same for the third dimension



Why Number of Columns Required?

- The memory of a computer is in form of a 1D array!



- As we saw, 2D (or >2D) arrays are "flattened" into 1D
- Row-Major order is a common way to flatten(used in C)

mat[3][5]

- In case of [illegible], where the next row starts.

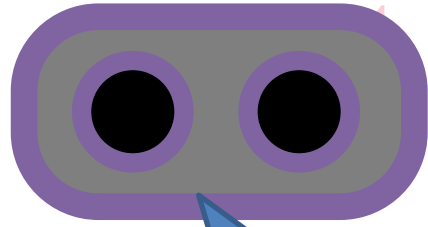
0,0	0,1	0,2	0,3	0,4	1,0	1,1	1,2	1,3	1,4	2,0	2,1	2,2	2,3	2,4

Tip 1: For 2D array mat[M][N], mat[i][j] is stored in memory at location i*N + j from start of mat (note: 0 <= i <= M, 0 <= j <= N)

Tip 2: For K-D array $arr[N_1][N_2]...[N_K]$, $arr[i_1][i_2]...[i_K]$ will be stored at the following location from start of arr

$$i_K + N_K*(i_{K-1} + N_{K-1}*(i_{K-2} + (\ldots + N_2*i_1) \ldots))$$

A look at 3D arrays..



- Declaration + init. of a 3D (dims = 2, 3, 4) array of integers

```
int x[2][3][4] =
 {
  { {0,1,2,3}, {4,5,6,7}, {8,9,10,11} },
  { {12,13,14,15}, {16,17,18,19}, {20,21,22,23} }
 };
```

Can leave blank

	12	13	14	15
0	1	2	3	19
4	5	6	7	23
8	9	10	11	

Another 3D array example: pressure values at each (x,y,z) co-ordinate of a room

Let us think of it as "array of arrays"": An array with 2 elements, each of which is a 3x4 array (and each of these 3x4 arrays can be thought of an array with 3 elements, each of which is a 1D 4 element array ☺)

Can leave blank

Correct but less common way (may also be a bit confusing)

```
int x[2][3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23};
```

Multi-dim. Array: Incomplete Initialization

- Consider a 2D array. It is okay if the number of elements initialized in each row is less than the number of columns (can initialize them later or never)

```
int a[3][4] = {
   {-2} ,              /*  row 0 */
   {-3, 5} ,           /*  row 1 */
   {8, 2, 10, 6}       /*  row 2 */
};
```

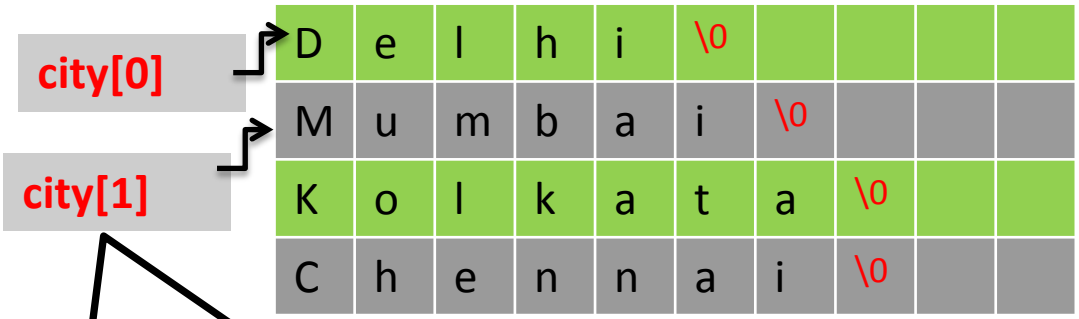
-2	0	0	0
-3	5	0	0
8	2	10	6

- If left uninitialized, the remaining unspecified values in each row will be set to 0 (note: For 1D arrays too, the uninitialized elements are set to 0)

Array of Strings (= 2D Array of Char)

- Another example of a multi-dimensional array

```
const int num_cities = 4;
const int name_length = 10;
char city[num_cities][name_length] = {
        {'D','e','l','h','i','\0'},
        {'M','u','m','b','a','i','\0'},
        {'K','o','l','k','a','t','a','\0'},
        {'C','h','e','n','n','a','i','\0'}
};
```



String array shortcut to directly access a full string: city[0] is the first string, city[1] is the second string, and so on..

Array with 4 elements. Each element is a char array

Array of Strings: Another Way

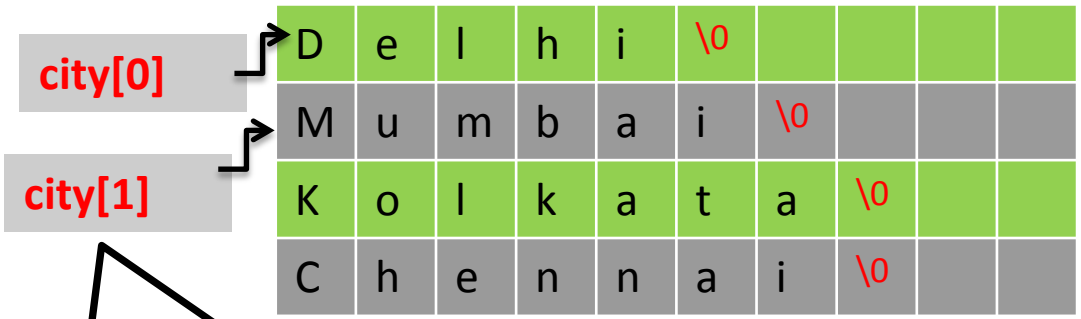
- Array of strings can also be declared/initialized as

```
const int name_length = 10;
char city[][name_length] = {
    {"Delhi"},
    {"Mumbai"},
    {"Kolkata"},
    {"Chennai"}
};
```

Recall that we can leave it blank

Each row directly defined as a string instead of a char array ending with \0

These curly braces around each string are not needed



String array shortcut to directly access a full string: city[0] is the first string, city[1] is the second string, and so on..

Array with 4 elements. Each element is a string

Reading and Printing Array of Strings

- Write a program that reads and displays the name of few cities of India

```
int main(){
  const int ncity = 4;
  const int lencity = 10;
  char city[ncity][lencity];
  int i;

  for (i=0; i<ncity; i++){
    scanf("%s", city[i]);
  }

  for (i=0; i<ncity; i++){
    printf("%d %s\n", i, city[i]);
  }
  return 0;
}
```

INPUT
Delhi
Mumbai
Kolkata
Chennai

city[0]

city[1]

D	e	l	h	i	\0				
M	u	m	b	a	i	\0			
K	o	l	k	a	t	a	\0		
C	h	e	n	n	a	i	\0		

OUTPUT
0 Delhi
1 Mumbai
2 Kolkata
3 Chennai