Islamic University of Technology (IUT)

Organization of Islamic Cooperation (OIC)

Department of Electrical and Electronic Engineering (EEE)

COURSE NO : EEE 4416
LAB NO : 02 (PART C)
TOPIC : CELL ARRAY, LOGICAL ARRAY

Cell Array

- A cell array in MATLAB is a versatile data structure that can hold values of varying data types within the same array.
- Each element in a cell array is known as a cell, which acts like a pointer to an individual data item.
- Each cell points to a distinct array, and these arrays may differ in type, size, or structure.
- Cell arrays are especially useful for storing collections of strings, as they allow each string to have a different length. [Example shown at the end]

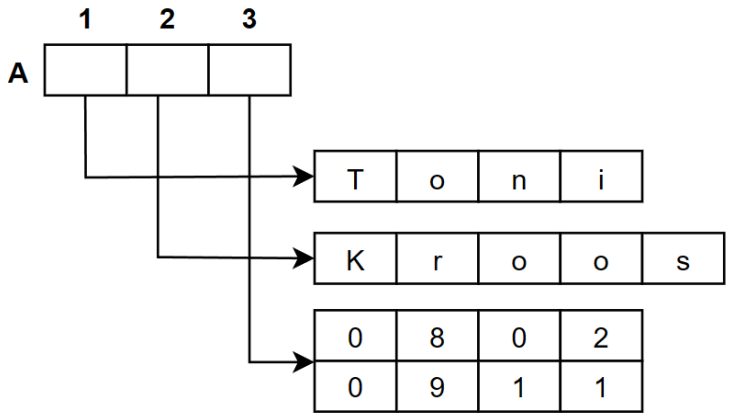
✓ A numeric array can only contain numbers, while a character array can only store characters. If you need to store both numbers and characters in a single array, then you need to look into a different structure – that is cell array.

```
>> a=[20, 'a']

a =

    '□a'

>> a={20, 'a'}

a =

  1×2 cell array

    {[20]}    {'a'}
```

- Example

A = { 'Toni', 'Kroos', [0 8 0 2; 0 9 1 1] }

This creates the memory shown in the diagram below. The cell array **A** contains three items: a 1-by-4 character array, a 1-by-5 character array, and a 2-by-4 double array.



Creating cell arrays

Method 1: Enclose a series of values within braces **{ }**

Example:

```
A = { 'Toni', 'Kroos', [0 8 0 2; 0 9 1 1] }

City = { 'Chattogram'  'Dhaka';   'Khulna'    'Sylhet'; 'Rajshahi'  'Rangpur' }
```

Here, row and column indices now refer to cell position and not to the underlying data. For example, if we ask for row-2 and column-1, we will get the whole cell contents 'Sylhet'.

City(2,2)

```
ans = 1×1 cell array

  {'Sylhet'}
```

Method 2: Use the **cell()** function to pre-allocate an entire cell array and then assign each cell to its desired value.

Example:

```
A = cell(1,4);
A{1} = 'Madeira'
A{2} = 'Manchester'
A{3} = 'Madrid'
A{4} = 'Turin'
```

```
A = 1×4 cell
'Madeira'    'Manchester' 'Madrid'     'Turin'
```

Cell arrays will grow dynamically if you use an index that is larger than the current size of the cell array. For example:

```
A{5} = 'Manchester again'
```

```
A = 1×5 cell
'Madeira'    'Manchester' 'Madrid'     'Turin'      'Manchester again'
```

Content Indexing

Use braces **{ }** - which is called ***content indexing***- to retrieve the data a cell points to.

For example:

```
cellArray = { 'Toni', 'Kroos', [0 8 0 2; 0 9 1 1] }
```

Here,

- **cellArray{1}** is the entire 1-by-4 character array containing 'Toni'
- **cellArray {1}(3)** is the single character 'n' -- that is, the 3rd value in the 1-by-4 character array 'Toni'. *Notice here 1 is enclosed by curly brackets {} and 3 is enclosed by the first bracket ().*
- **cellArray {2}** is the entire 1-by-5 character array containing 'Kroos'
- **cellArray {2}(1)** is the single character 'K' -- that is, the 1st value in the 1-by-5 character array 'Kroos'
- **cellArray {3}** is the entire 2-by-4 double array, [0 8 0 2; 0 9 1 1]
- **cellArray {3}(2,3)** is the single value 1, which is the element on row 2 and column 3 of the array
- **cellArray {1}(2,4)** would generate an error message because cell 1 does not point to a 2-by-4 array

Note the use of braces **{ }** to access a cell and the use of parentheses **( )** to access individual elements in the arrays each cell points to.

Cell Indexing

Use parentheses **( )** - which is called ***cell indexing***, in the rare case where you need to do something to the cell pointer.

Possible uses of cell indexing

Example case:

```
A = { 'Wraith', 'Octane', [1 9 4 5; 0 9 1 1] }
```

```
A = 1×3 cell
```

	1	2	3
1	'Wraith'	'Octane'	[1,9,4,5;0,9,1,1]

1. *Access a cell as a cell (returns a 1x1 cell):*

```
cell_element = A(2)
```

Output:

cell_element = 1×1 cell array
{'Octane'}

2. *Assign another cell into that position:*

```
A(2) = {'Caustic'} % replaces cell at index 2 with a new string
```

Output:

```
A = 1×3 cell
```

	1	2	3
1	'Wraith'	'Caustic'	[1,9,4,5;0,9,1,1]

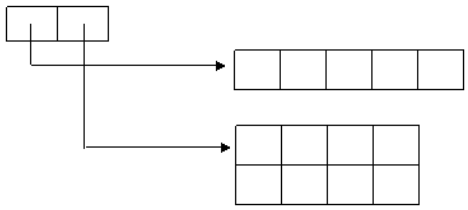
3. *Delete a cell:*

```
A(2) = [];  % deletes the second cell
```

Output:

```
A = 1×2 cell
```

	1	2
1	'Wraith'	[1,9,4,5;0,9,1,1]

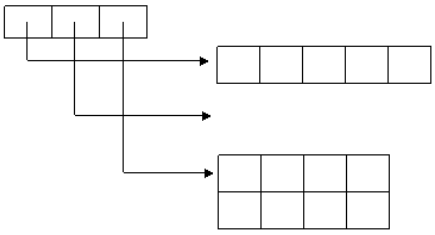


But, A{2} = []; does not modify the cell array, but it deletes the data that was being pointed to by cell 2.

```
A{2} = []
```

```
A = 1×2 cell
```

	1	2
1	'Wraith'	[]



4. *Extract multiple cells:*

```
A(1:2)
```

```
ans = 1×2 cell
```

	1	2
1	'Wraith'	[]

5. Replace multiple cells:

```
A(1:3) = {'Optimus', 'Prime', magic(2)}
```

```
A = 1×3 cell
```

	1	2	3
1	'Optimus'	'Prime'	[1,3;4,2]

Cell Array Functions

num2cell() - Convert a numeric scalar or array into a cell array

```
numArray = [10, 20; 30, 40]
```

```
numArray = 2×2
    10    20
    30    40
```

```
cellArray = num2cell(numArray)
```

```
cellArray = 2×2 cell
```

	1	2
1	10	20
2	30	40

```
% cellArray is now a 2x2 cell array where each cell contains a number
```

cellstr() - Convert text string(s) into a cell array of strings

```
text = 'Optimus Prime'
```

```
text = 'Optimus Prime'
```

```
cellStr = cellstr(text)
```

```
cellStr = 1×1 cell array
    {'Optimus Prime'}
```

```
% cellStr is a 1x1 cell array containing 'Optimus Prime'
```

cell2mat() - Convert a cell array containing numeric data into a numeric matrix

```
C = {[1 2]; [3 4]; [5 6]}
```

```
C = 3×1 cell
```

	1
1	[1,2]
2	[3,4]
3	[5,6]

```
mat = cell2mat(C)
```

```
mat = 3×2
     1     2
     3     4
     5     6
```

char() - Convert a cell array of strings into a character array (2D char matrix)

```
player = {'Ronaldo', 'Benzema', 'Bale'}
```

```
player = 1×3 cell
'Ronaldo'    'Benzema'    'Bale'
```

```
playerCharArray = char(player)
```

```
playerCharArray = 3×7 char array
    'Ronaldo'
    'Benzema'
    'Bale   '
```

playerCharArray is a 3x6 char array with padded spaces

iscell() - Check if a variable is a cell array

```
x = {1, 2, 3}
```

```
x = 1×3 cell
```

	1	2	3
1	1	2	3

```
result = iscell(x)   % Returns true (1)
```

```
result = logical
   1
```

```
y = [1, 2, 3]
```

```
y = 1×3
     1     2     3
```

```
result2 = iscell(y)  % Returns false (0)
```

```
result2 = logical
   0
```

iscellstr() - Check if a variable is a cell array of strings

```
a = {'one', 'two', 'three'}
```
```
a = 1×3 cell
'one'        'two'        'three'
```
```
result = iscellstr(a) % Returns true (1)
```
```
result = logical
   1
```

```
b = {1, 2, 3}
```
```
b = 1×3 cell
```

	1	2	3
1	1	2	3

```
result2 = iscellstr(b) % Returns false (0)
```
```
result2 = logical
   0
```

- Cell arrays are especially useful for storing collections of strings, as they allow each string to have a different length.

```
% Case 1: Error
city = ['Chattogram';'Feni';'Dhaka';'Khulna']
```
```
Error using vertcat
Dimensions of arrays being concatenated are not consistent.
```

```
% Case 2:
% If we use a regular character array (matrix), all strings need to have the same length, padded with spaces
city = ['Chattogram'; 'Feni      '; 'Dhaka     '; 'Khulna    ' ]
```
```
city = 4×10 char array
    'Chattogram'
    'Feni      '
    'Dhaka     '
    'Khulna    '
```

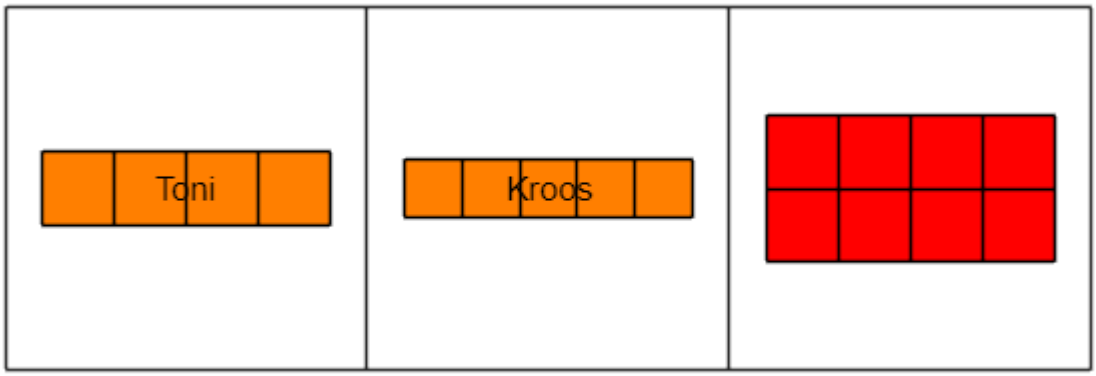
```
% Case 3: Let's store each string in its natural length without padding or trimming.
city = {'Chattogram';'Feni';'Dhaka';'Khulna'}
```
```
city = 4×1 cell
'Chattogram'
'Feni'
'Dhaka'
'Khulna'
```

cellplot() function is useful for visualizing the structure of a cell array. It creates a visual plot of a cell array.

```
A = { 'Toni', 'Kroos', [0 8 0 2; 0 9 1 1] }
```
```
A = 1×3 cell
```

	1	2	3
1	'Toni'	'Kroos'	[0,8,0,2;0,9,1,1]

```
cellplot(A)
```



Logical Array

The **logical** data type in MATLAB represents Boolean true/false values: **true** (1), **false** (0). It's essential for conditional testing, indexing, and control flow. Relational operators produce logical arrays.

Relational Operators

A relational operator produces a value that depends on the relation between the values of its two operands. It compares two numbers by determining whether a comparison statement (e.g., 5 < 8) is true or false.

Operator	Description	Example	Result
==	equal	5 == 5	true
~=	not equal	4 ~= 3	true
<	less than	2 < 7	true
<=	less than or equal to	3 <= 3	true
>	greater than	9 > 10	false
>=	greater than or equal to	5 >= 8	false

When two numbers are compared, the result is 1 (logical true) if the comparison, according to the relational operator, is true, and 0 (logical false) if the comparison is false.

Run the following in the live script:

```
3 == 35-32
x = (45*47 > 2105) + 9
x = 45*47 > 2105 + 9
```

Output:

```
ans = logical
   1
x = 10
x = logical
   1
```

The parentheses have an important effect here. If we omit them, we get a different answer, as the addition operator is executed before the greater-than operator, because the precedence of + is higher than the precedence of >, so it is carried out first.

Relational Expressions on arrays:

- If two scalars are compared, the result is a scalar 1 or 0.
- If two arrays are compared (only arrays of the same size can be compared), the comparison is done element-by-element, and the result is a logical array of the same size with 1s and 0s according to the outcome of the comparison at each address.
- If a scalar is compared with an array, the scalar is compared with every element of the array, and the result is a logical array with 1s and 0s according to the outcome of the comparison of each element.

```
A = [1 4 7; 2 5 8; 3 6 9];
L1 = A > 5
```

```
L1 = 3×3 logical array
   0   0   1
   0   0   1
   0   1   1
```

```
[4 -1 7 5 3] > [5 -9 6 5 -3]
[4 -1 7 5 3] ~= [5 -9 6 5 -3]
```

```
ans = 1×5 logical array
   0   1   1   0   1
```

```
ans = 1×5 logical array
   1   1   1   0   1
```

```
>> y=(6<10)+(7>8)+(5*3==60/4)
```

Using relational operators in math expression.

Equal to 1 since 6 is smaller than 10.

Equal to 0 since 7 is not larger than 8.

Equal to 1 since 5*3 is equal to 60/4.

```
y =
     2
>> b=[15 6 9 4 11 7 14]; c=[8 20 9 2 19 7 10];
>> d=c>=b
d =
     0     1     1     0     1     1     0
```

Define vectors b and c.

Checks which c elements are larger than or equal to b elements.

Assigns 1 where an element of c is larger than or equal to an element of b.

```
>> b == c
ans =
     0     0     1     0     0     1     0
>> b~=c
ans =
     1     1     0     1     1     0     1
>> f=b-c>0
f =
     1     0     0     1     0     0     1
>> A=[2 9 4; -3 5 2; 6 7 -1]
A =
     2     9     4
    -3     5     2
     6     7    -1
>> B=A<=2
```

Checks which b elements are equal to c elements.

Checks which b elements are not equal to c elements.

Subtracts c from b and then checks which elements are larger than zero.

Define a 3×3 matrix A.

Checks which elements in A are smaller than or equal to 2. Assigns the results to matrix B.

Built-in Constructors:

```
T = true(3)     % 3×3 all true
F = false(2,4)  % 2×4 all false
```

```
T = 3×3 logical array
   1   1   1
   1   1   1
   1   1   1
F = 2×4 logical array
   0   0   0   0
   0   0   0   0
```

Numeric Conversion:

```
B = [0, -2, 5, 0];
L = logical(B)
```

```
L = 1×4 logical array
   0   1   1   0
```

Logical Operators

A logical operator produces a value that depends on the truth of its two operands.

Operator	Name	Example	Description
&	AND	A & B	Operates on two operands (A and B). If **both** are true, the result is true (1); otherwise, false (0).
|	OR	A | B	Operates on two operands (A and B). If **either** or **both** are true, the result is true (1); otherwise, false (0).
~	NOT	~A	Operates on one operand (A). Returns the opposite: true (1) if A is false, and false (0) if A is true.

```
X = [10, 20, 30];
(X>=15)
(X==10)
(X>=15) | (X==10)
```

```
ans = 1×3 logical array
   0   1   1
ans = 1×3 logical array
   1   0   0
ans = 1×3 logical array
```

1 1 1

Logical Indexing

Logical indexing is a powerful MATLAB feature that uses Boolean arrays (true/false) to select elements of another array. Instead of specifying numeric subscripts, you supply a logical mask of the same size.

Creating Logical Masks

A logical mask is an array of true (1) and false (0) values generated by relational or logical operators.

```
A = [3, 7, 2, 9, 5];   % Numeric array
mask_gt5 = A > 5;      % [false, true, false, true, false]
mask_even = mod(A,2) = = 0; % [false, false, true, false, false]
mask_comp = (A > 2) & (A < 8);  % true for elements between 2 and 8
```

Extracting Elements

Use a logical mask to index the original array and extract only the elements where the mask is true.

```
selected = A(mask_gt5);   % [7, 9]
evens = A(mask_even);      % [2]
```

Key point: The resulting array is a column vector containing all selected elements in linear order.

Modifying Elements

Logical indexing also allows in-place modification of selected entries.

```
A(mask_gt5) = 0;   % Set all values > 5 to zero
% A becomes [3, 0, 2, 0, 5]
```

You can assign any compatible array:

```
A(mask_even) = [20];  % Assigns 20 to each even element
```

Multi-Dimensional Arrays

Logical indexing extends to matrices and higher dimensions. Create a mask of the same size and apply it.

```
M = magic(4);
mask = M > 10;
largeVals = M(mask);    % All entries > 10
```

Indexing returns a column vector of all qualifying entries.

Combining with find

Use find to obtain the numeric indices of true entries.

```
idx = find(M>10);  % Linear indices
[row, col] = find(M>10);  % Row, column subscripts
```

This is useful for mapping positions or iterating over specific entries.

Wherever possible, use built-in array operations instead of for loops for speed. Clear large arrays when done: clear A B Use help for any function: help reshape

Logical Functions

Function	What it does
any(X)	true if any element of X is true
all(X)	true if all elements of X are true
nnz(X)	Number of nonzero (true) elements
find(X)	Indices of true elements
isnan(X)	true where X is NaN
isfinite(X)	true where X is finite
ismember(A,B)	true where elements of A are also in B

Run the following code:

```
v = [1, NaN, 3, Inf]
finiteMask = isfinite(v)    % [1 0 1 0]
nanIdx      = find(isnan(v))% 2
```