OTHER KINDS OF ARRAYS*

*** Some of the contents are adopted from**
MATLAB® for Engineers, Holly Moore, 3rd Ed., Pearson Inc., 2012.

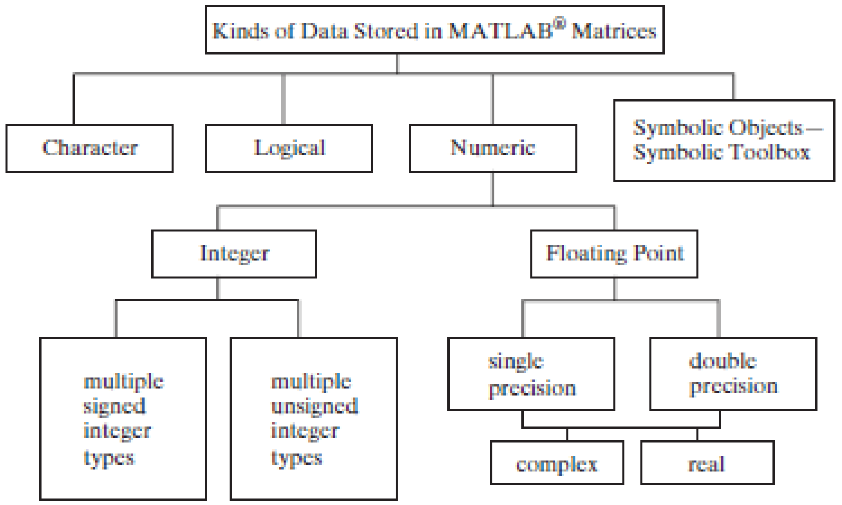
Data Types

- The primary data type (also called a class) in MATLAB is the *array or matrix .*

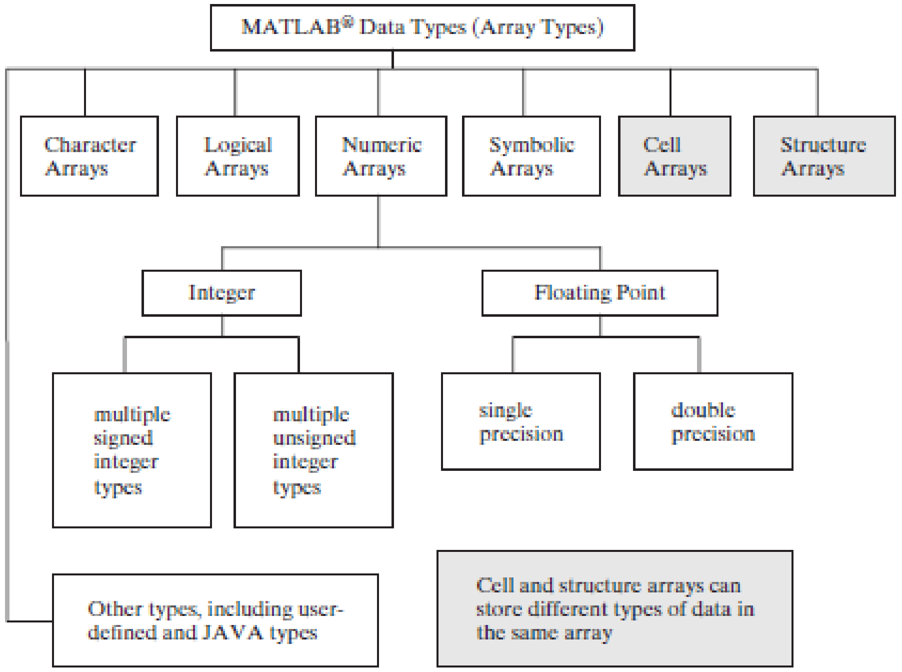
- Within the array, MATLAB supports a number of different secondary data types. Because MATLAB was written in C, many of those data types parallel the data types supported in C. In general, all the data within an array must be the same type.

- However, MATLAB also includes functions to convert between data types, and array types to store different kinds of data in the same array (cell and structure arrays).

Data Types



Kinds of Data Stored in MATLAB® Matrices
Character
Logical
Numeric
Symbolic Objects— Symbolic Toolbox
Integer
Floating Point
multiple signed integer types
multiple unsigned integer types
single precision
double precision
complex
real



MATLAB® Data Types (Array Types)
Character Arrays
Logical Arrays
Numeric Arrays
Symbolic Arrays
Cell Arrays
Structure Arrays
Integer
Floating Point
multiple signed integer types
multiple unsigned integer types
single precision
double precision
Other types, including user-defined and JAVA types
Cell and structure arrays can store different types of data in the same array

Numeric Data Types

- Double-Precision Floating-Point Numbers
- Single-Precision Floating-Point Numbers
- Integers
- Complex Numbers

Double-Precision Floating-Point Numbers

- The default numeric data type in MATLAB is the double-precision floating-point number.

- Recall that when we create a variable such as A , as in

  A = 1;

  the variable is listed in the workspace window and the class is “double”.

  B = 1:10; C=[1,2,3; 4,5,6];
- Notice that the array requires 8 bytes of storage space. Each byte is equal to 8 bits, so the number 1 requires 64 bits of storage space. The variable B requires 80 bytes, 8 for each of the 10 values stored, and C requires 48 bytes, again 8 for each of the 6 values stored.

- You can use the realmax and realmin functions to determine the positive maximum/minumum possible value of a double-precision floating-point number:

  realmax

  ans = 1.7977e+308

  realmin

  ans = 2.2251e-308

Double-Precision Floating-Point Numbers

- If you try to enter a value whose absolute value is greater than realmax, or if you compute a number that is outside this range, MATLAB will assign a value of infinity:

```
x = 5e400
x = Inf
```

- Similarly, if you try to enter a value whose absolute value is less than realmin, MATLAB will assign a value of zero:

```
x = 1e-400
x =0
```

Single-Precision Floating-Point Numbers

Single-precision floating-point numbers use only half the storage space of a double-precision number and store only half the information. Each value requires only 4 bytes, or 4 x 8 = 32 bits, of storage space. When we define D as a single-precision number:

```
D = single(5)
D =5
```

We need to use the single function to change the value 5 (which is double precision by default) to a single-precision number. Similarly, the double function will convert a variable to a double, as in

```
double(D)
```

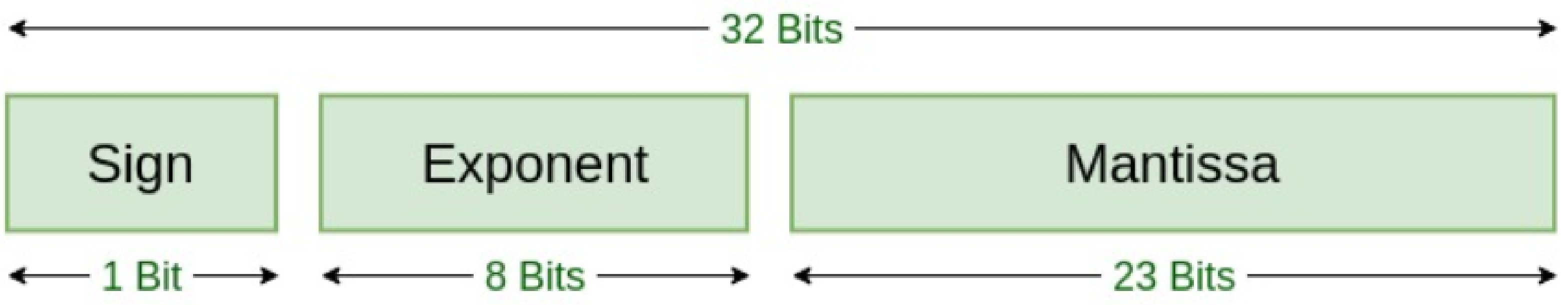
which changes the variable D into a double.

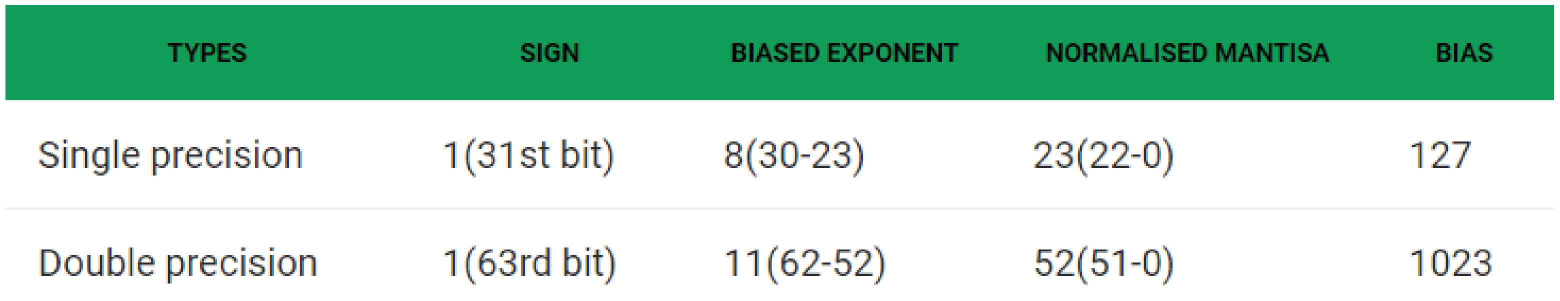
Since single-precision numbers are allocated only half as much storage space, they cannot cover as large a range of values as double-precision numbers. We can use the realmax and realmin functions to show this:

```
realmax('single')
ans =
3.4028e+038
realmin('single')
ans =
1.1755e–038
```

IEEE Standard 754 Floating Point Numbers

Single Precision

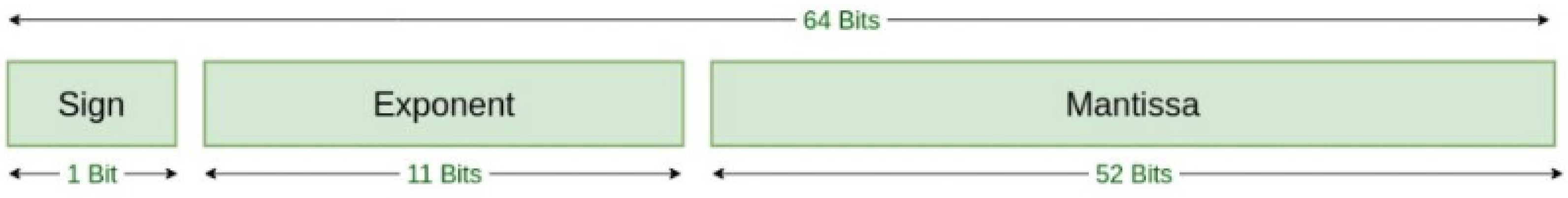


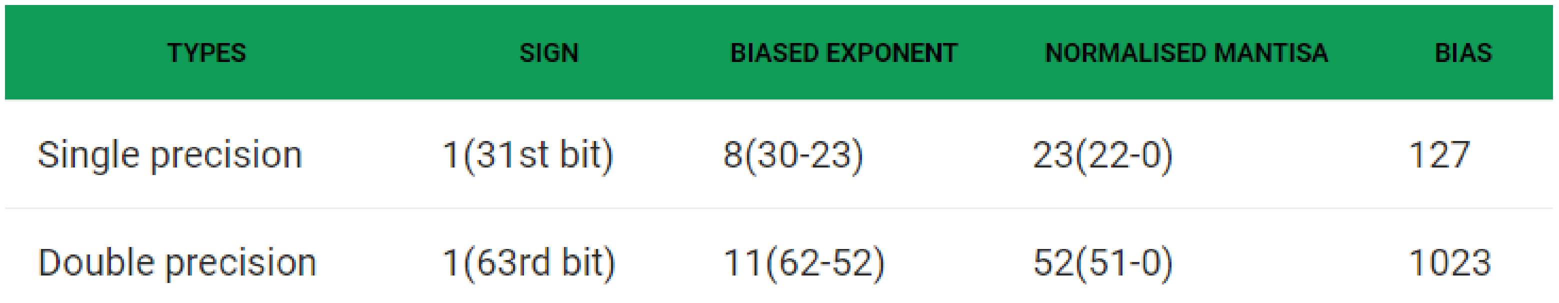


TYPES	SIGN	BIASED EXPONENT	NORMALISED MANTISA	BIAS
Single precision	1(31st bit)	8(30-23)	23(22-0)	127
Double precision	1(63rd bit)	11(62-52)	52(51-0)	1023

IEEE Standard 754 Floating Point Numbers

Double Precision

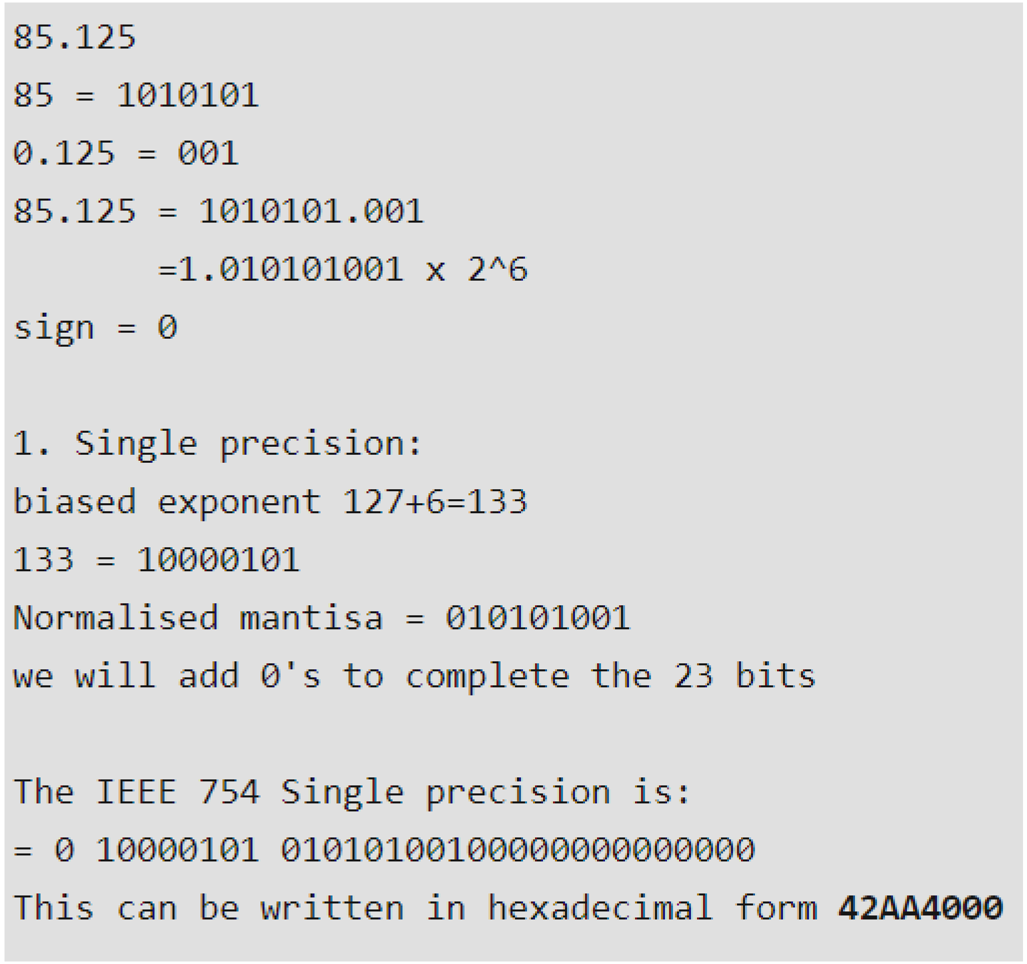




TYPES	SIGN	BIASED EXPONENT	NORMALISED MANTISA	BIAS
Single precision	1(31st bit)	8(30-23)	23(22-0)	127
Double precision	1(63rd bit)	11(62-52)	52(51-0)	1023

Single/Double Precision Example

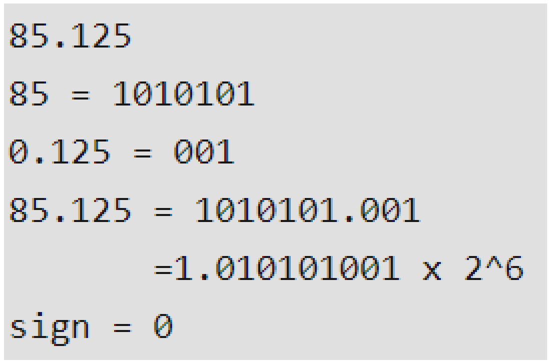
Single Precision



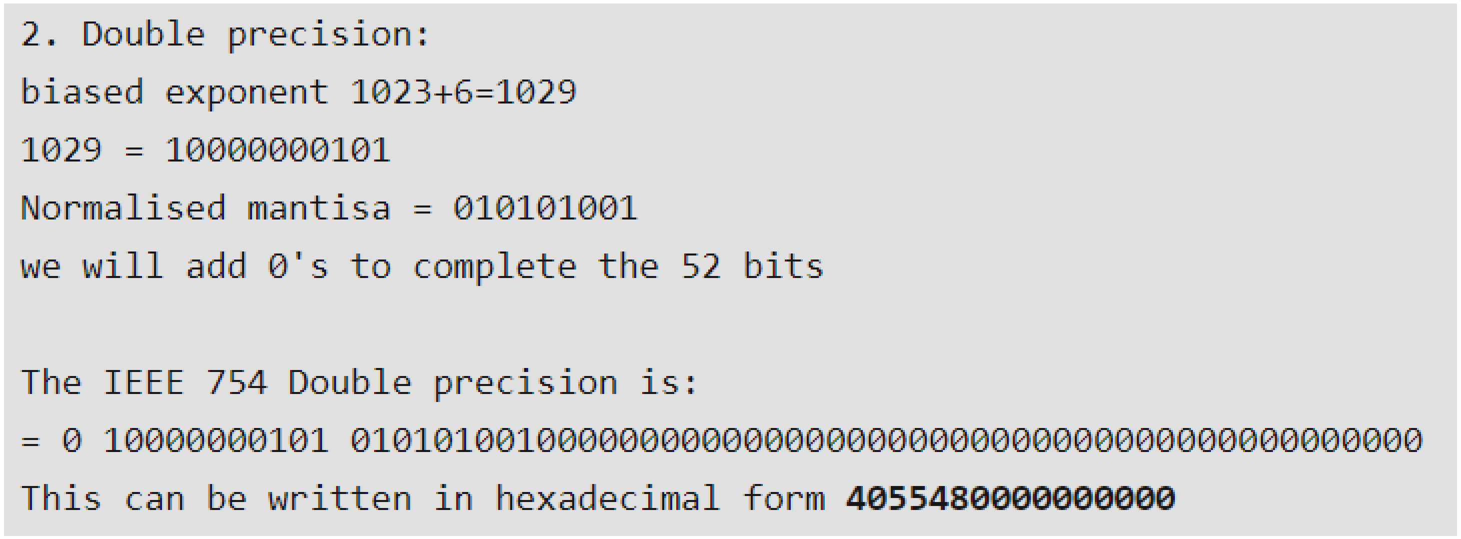
```
85.125
85 = 1010101
0.125 = 001
85.125 = 1010101.001
       =1.010101001 x 2^6
sign = 0

1. Single precision:
biased exponent 127+6=133
133 = 10000101
Normalised mantisa = 010101001
we will add 0's to complete the 23 bits

The IEEE 754 Single precision is:
= 0 10000101 01010100100000000000000
This can be written in hexadecimal form 42AA4000
```

Single/Double Precision Example

Double Precision



```
85.125
85 = 1010101
0.125 = 001
85.125 = 1010101.001
       =1.010101001 x 2^6
sign = 0
```



```
2. Double precision:
biased exponent 1023+6=1029
1029 = 10000000101
Normalised mantisa = 010101001
we will add 0's to complete the 52 bits

The IEEE 754 Double precision is:
= 0 10000000101 0101010010000000000000000000000000000000000000000000
This can be written in hexadecimal form 4055480000000000
```

Single-Precision Floating-Point Numbers

We can demonstrate the effect of round-off error in single-precision versus double- precision problems with an example. Consider the series

$$\Sigma\left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \cdots + \frac{1}{n} + \cdots\right)$$

A series is the sum of a sequence of numbers, and this particular series is called the *harmonic series, represented with the following shorthand notation:*

$$\sum_{n=1}^{\infty} \frac{1}{n}$$

The harmonic series diverges; that is, it just keeps getting bigger as you add more terms together. You can represent the first 10 terms of the harmonic sequence with the following commands:

```
n = 1:10;
harmonic = 1./n
```

You can view the results as fractions if you change the format to rational:

```
format rat
harmonic =
1    1/2    1/3    1/4    1/5    1/6    1/7    1/8    1/9    1/10
```

or you can use the short format, which shows decimal representations of the numbers:

```
format short
harmonic =
1.0000    0.5000    0.3333    0.2500    0.2000    0.1667    0.1429
          0.1250    0.1111    0.1000
```

Single-Precision Floating-Point Numbers

By calculating the partial sums (also called cumulative sums), we can see how the value of the sum of these numbers changes as we add more terms:

```
partial_sum = cumsum(harmonic)
partial_sum =
Columns 1 through 6
1.0000     1.5000     1.8333     2.0833     2.2833     2.4500
Columns 7 through 10
2.5929     2.7179     2.8290     2.9290
```

The cumulative sum (cumsum) function calculates the sum of the values in the array up to the element number displayed.

The only problem with this process is that the values in harmonic keep getting smaller and smaller. Eventually, when n is big enough, 1./n is so small that the computer can't distinguish it from zero. This happens much more quickly with single precision than with double-precision representations of numbers. We can demonstrate this property with a large array of *n -values:*

```
n = 1:1e7;
harmonic = 1./n;
partial_sum = cumsum(harmonic);
```

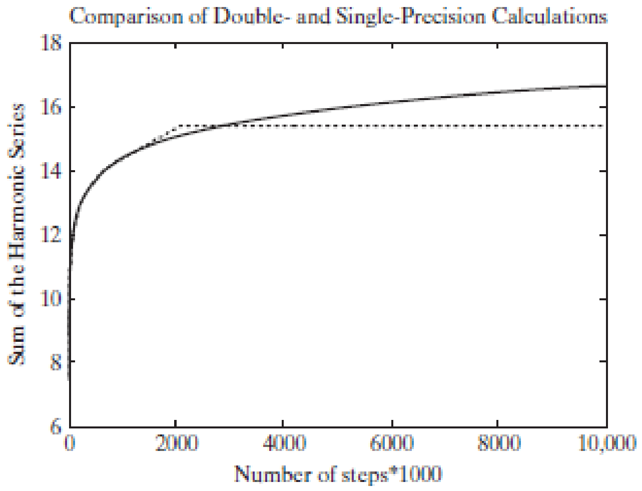
Single-Precision Floating-Point Numbers

We can select every thousandth value with the following code:

```
m = 1000:1000:1e7;
partial_sums_selected = partial_sum(m);
plot(partial_sums_selected)
```

Now we can repeat the calculations, but change to single-precision values. You may need to clear your computer memory before this step, depending on how much memory is available on your system. The code is

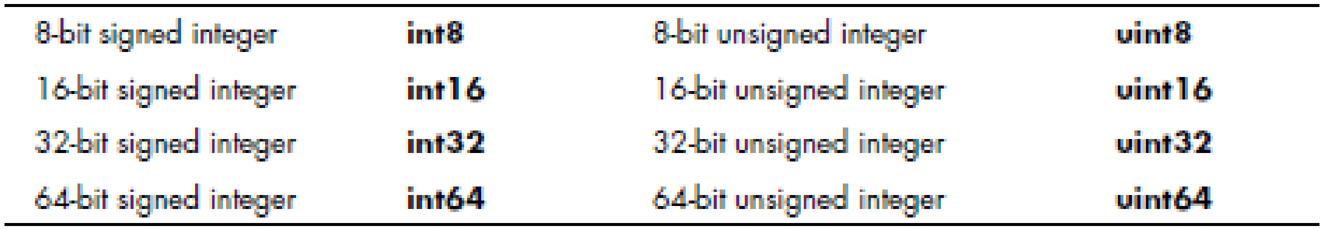
```
n = single(1:1e7);
harmonic = 1./n;
partial_sum = cumsum(harmonic);
m = 1000:1000:1e7;
partial_sums_selected = partial_sum(m);
hold on
plot(partial_sums_selected,':')
```



The solid line represents the partial sums calculated with double precision. The dashed line represents the partial sums calculated with single precision. The single-precision calculation levels off, because we reach the point where each successive term is so small that the computer sets it equal to zero. We haven't reached that point yet for the double-precision values.

Integers

- Traditionally, integers are used as counting numbers. For example, there can’t be 2.5 people in a room, and you can’t specify element number 1.5 in an array.
- Eight different types of integers are supported by MATLAB. They differ in how much storage space is allocated for the type and in whether the values are signed or unsigned.
- The more storage space, the larger the value of an integer number you can use.



8-bit signed integer	**int8**	8-bit unsigned integer	**uint8**
16-bit signed integer	**int16**	16-bit unsigned integer	**uint16**
32-bit signed integer	**int32**	32-bit unsigned integer	**uint32**
64-bit signed integer	**int64**	64-bit unsigned integer	**uint64**

Integers

- Since 8 bits is 1 byte, when we assign E as an int8 with the code

  ```
  E = int8(10)
  E =10
  ```

  it requires only 1 byte of storage. You can determine the maximum value of any of the integer types by using the intmax function. For example, the code

  ```
  intmax('int8')
  ans =127
  ```

  indicates that the maximum value of an 8-bit signed integer is 127.

- The four signed-integer types allocate storage space to specify whether the number is plus or minus. The four unsigned-integer types assume that the number is positive and thus do not need to store that information, leaving more room to store numerical values.

  The code

  ```
  intmax('uint8')
  ans =255
  ```

  reveals that the maximum value of an 8-bit unsigned integer is 255.

Complex Numbers

- The default storage type for complex numbers is double; however, twice as much storage room is needed, because both the real and imaginary components must be stored:

  F = 5+3i;

- Thus, 16 bytes (= 128 bits) are required to store a double complex number.

- Complex numbers can also be stored as singles or integers as the following code illustrates:

  G = int8(5+3i);

Character and String Data

In addition to storing numbers, MATLAB can store character information. Single quotes are used to identify a string and to differentiate it from a variable name. When we type the string

H ='Holly';

a 1 x 5 character array is created. Each letter is a separate element of the array, as is indicated by the code

H(5)

ans = y

Any string represents a character array in MATLAB.

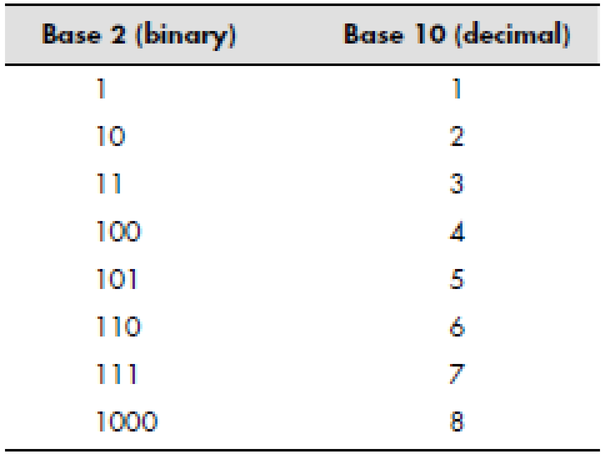
K = 'MATLAB is fun'

becomes a 1x13 character array. Notice that the spaces between the words are counted as characters.

All information in computers is stored as a series of zeros and ones. There are two major coding schemes to do this: ASCII and EBCDIC. You can think of the series of zeros and ones as a binary, or base-2, number. In this sense, all computer information is stored numerically. Every base-2 number has a decimal equivalent.

Character and String Data

When we ask MATLAB to change a character to a double, the number we get is the decimal equivalent in the ASCII coding system. Thus, we may have



Base 2 (binary)	Base 10 (decimal)
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8

```
double('a')
ans =
97
```

Conversely, when we use the char function on a double, we get the character represented by that decimal number in ASCII—for example,

```
char (98)
ans =
b
```

If we try to create a matrix containing both numeric and character information, MATLAB converts all the data to character information:

```
['a',98]
ans =
ab
```

Character and String Data

The character b is equivalent to the number 98. Not all numbers have a character equivalent. If this is the case they are represented as a blank in the resulting character array

```
['a',3]
ans =
a
```

Although this result looks like it has only one character in the array, check the workspace window. You'll find that the size is a 1 x 3 character array. If we try to perform mathematical calculations with both numeric and character information, MATLAB converts the character to its decimal equivalent:

```
'a' + 3
ans =
100
```

Since the decimal equivalent of 'a' is 97, the problem is converted to 97 + 3 = 100

Symbolic Data

The symbolic toolbox uses symbolic data to perform symbolic algebraic calculations. One way to create a symbolic variable is to use the sym function:

```
L = sym('x^2-2')
L =
x^2-2
```

The storage requirements of a symbolic object depend on how large the object is.

Logical Data

Logical arrays are usually the result of logical operations. For example,

```
x = 1:5;
y = [2,0,1,9,4];
z = x>y
```

returns

```
z =
0 1 1 0 1
```

We can interpret this to mean that x > y is false for elements 1 and 3, and true for elements 2, 3, and 5. These arrays are used in logical functions and usually are not even seen by the user. For example,

```
find(x>y)
ans =
2 3 5
```

tells us that elements 2, 3, and 5 of the x array are greater than the corresponding elements of the y array. Thus, we don’t have to analyze the results of the logical operation ourselves.

Sparse Arrays

- Both double-precision and logical arrays can be stored either in full matrices or as sparse matrices. Sparse matrices are “sparsely populated,” which means that many or most of the values in the array are zero. (Identity matrices are examples of sparse matrices.)

- If we store double-precision sparse arrays in the full-matrix format, every data value takes 8 bytes of storage, be it a zero or not. The sparse-matrix format stores only the nonzero values and remembers where they are—a strategy that saves a lot of computer memory.

- For example, define a 1000 x 1000 identity matrix, which is a one-million element matrix:

  N = eye(1000);

- At 8 bytes per element, storing this matrix takes 8 MB. If we convert it to a sparse matrix, we can save some space. The code to do this is

  P = sparse(N);

- Notice in the workspace window that array P requires only 16,004 bytes! Sparse matrices can be used in calculations just like full matrices.

Multidimensional Arrays

- Suppose you would like to combine the following four two-dimensional arrays into a three-dimensional array:

```
x = [1,2,3;4,5,6];
y = 10*x;
z = 10*y;
w = 10*z;
```

- You need to define each page separately:

```
my_3D_array(:,:,1) = x;
my_3D_array(:,:,2) = y;
my_3D_array(:,:,3) = z;
my_3D_array(:,:,4) = w;
```

- When you call up my_3D_array , using the code

```
my_3D_array
```

Multidimensional Arrays

The result is

my_3D_array

my_3D_array(:,:,1) =

1 2 3

4 5 6

my_3D_array(:,:,2) =

10 20 30

40 50 60

my_3D_array(:,:,3) =

100 200 300

400 500 600

my_3D_array(:,:,4) =

1000 2000 3000

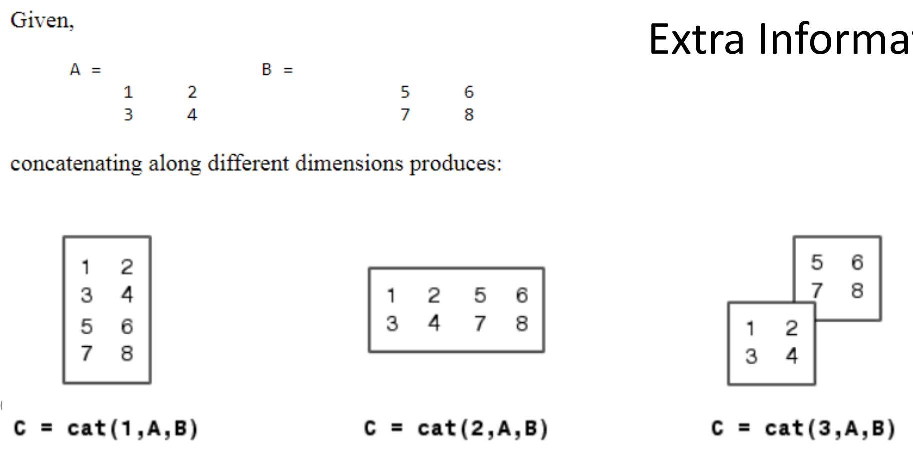
4000 5000 6000

An alternative approach is to use the cat function. When you concatenate a list you group the members together in order, which is what the cat function does.

The first field in the function specifies which dimension to use to concatenate the arrays, which follow in order. For example, to create the array we used in the previous example the syntax is

cat(3,x,y,z,w)

Extra Information:



Given,

```
A =                     B =
       1      2                    5      6
       3      4                    7      8
```

concatenating along different dimensions produces:

`C = cat(1,A,B)`   `C = cat(2,A,B)`   `C = cat(3,A,B)`

Character Arrays

We can create two-dimensional character arrays only if the number of elements in each row is the same. Thus, a list of names such as the following won't work, because each name has a different number of characters:

```
Q = ['Holly';'Steven';'Meagan';'David';'Michael';'Heidi']
??? Error using ==> vertcat
All rows in the bracketed expression must have the same number
of columns.
```

The char function "pads" a character array with spaces, so that every row has the same number of elements:

```
Q = char('Holly','Steven','Meagan','David','Michael','Heidi')
Q =
Holly
Steven
Meagan
David
Michael
Heidi
```

Q is a 6 x 7 character array. Notice that commas are used between each string in the char function.

Character Arrays

Let us assume that the array R contains test scores for the students in the character array Q:

```
R = [98;84;73;88;95;100]
R =
98
84
73
88
95
100
```

If we try to combine these two arrays, we'll get a strange result, because they are two different data types:

```
table = [Q,R]
table =
Holly b
Steven T
Meagan I
David X
Michael_
Heidi d
```

Character Arrays

The double-precision values in R were used to define characters on the basis of their ASCII equivalent. When doubles and chars are used in the same array, MATLAB converts all the information to chars.

The num2str (number to string) function allows us to convert the double R matrix to a matrix composed of character data:

S = num2str(R)

S =

98

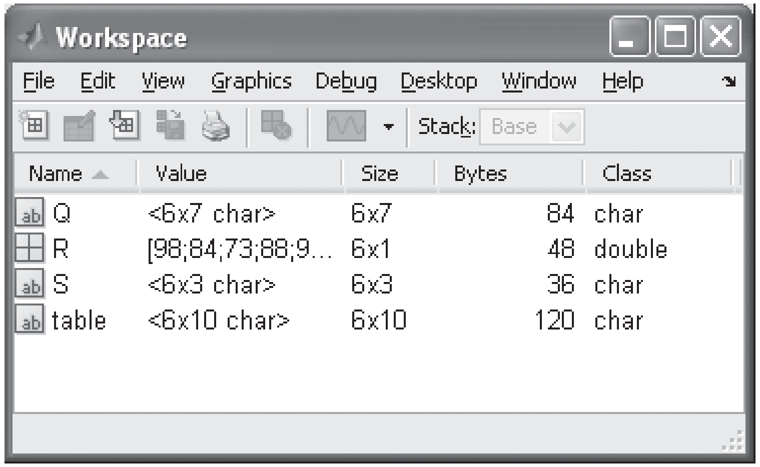
84

73

88

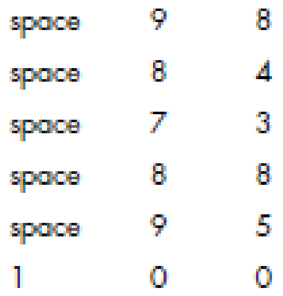
95

100



Name	Value	Size	Bytes	Class
Q	<6x7 char>	6x7	84	char
R	[98;84;73;88;9...	6x1	48	double
S	<6x3 char>	6x3	36	char
table	<6x10 char>	6x10	120	char

R and S look alike, but if you check the workspace window, you'll see that R is a 6 x 1 double array and S is the 6 x 3 char array shown below.



space	9	8
space	8	4
space	7	3
space	8	8
space	9	5
1	0	0

Character Arrays

Now we can combine Q , the character array of names, with S , the character array of scores:

```
table = [Q,S]
table =
Holly      98
Steven     84
Meagan     73
David      88
Michael    95
Heidi     100
```

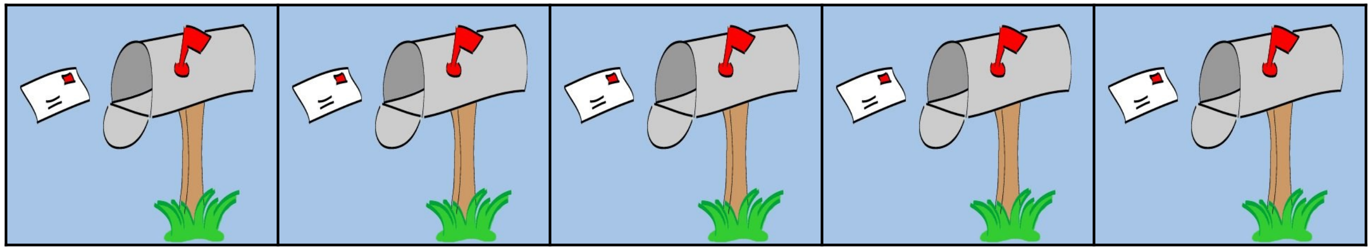
We could also use the disp function to display the results:

```
disp([Q,S])
Holly      98
Steven     84
Meagan     73
David      88
Michael    95
Heidi     100
```

Cell Arrays

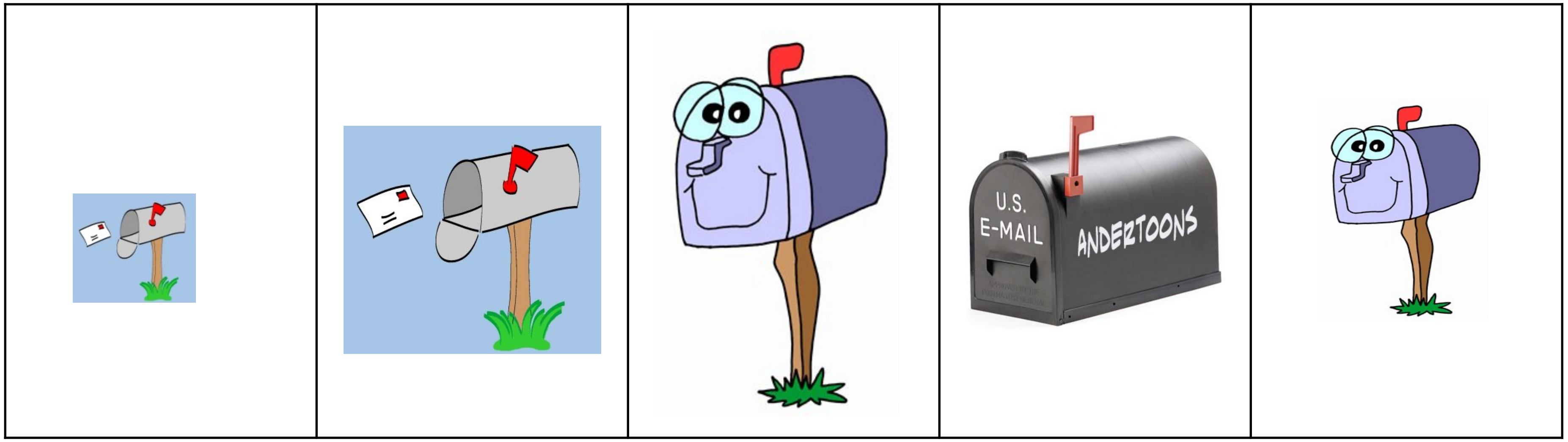
- So far we could only save arrays and matrices that contain elements of the same type (numbers, characters ,etc.)

```
x = [ 1,          4,          5,    2,     -1]
y = ['ankara']
```



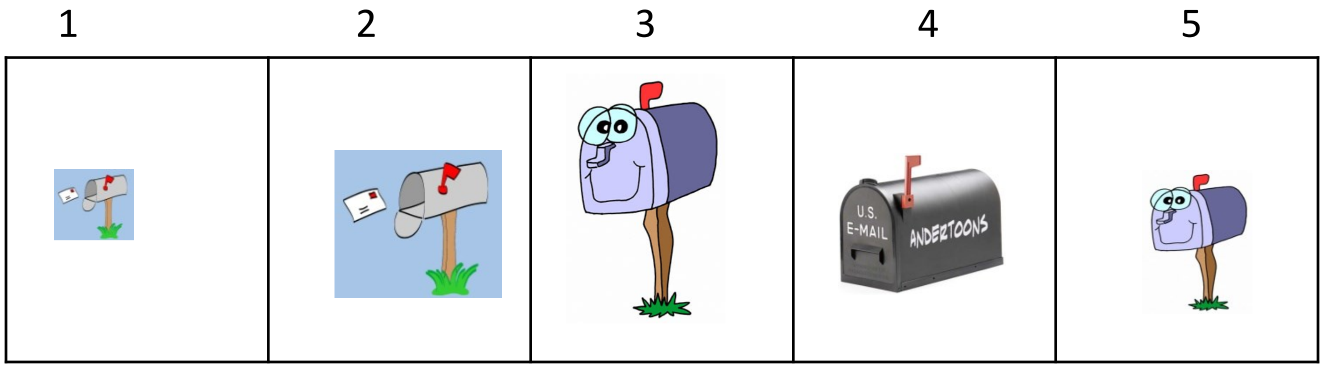
Cell Arrays

- Sometimes we want to store elements of different types within the same container
- **Cells and structures enable us to do just that!**

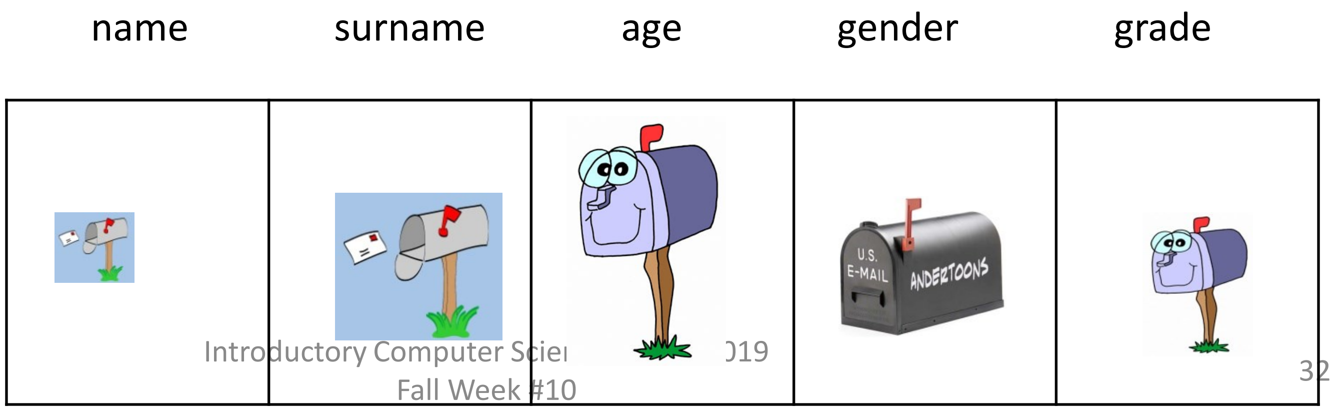


Cell vs Structure

- Cells use **addresses**



- Structures use **names**



Cell Array Creation

- One way to create a cell (**ce**

Note that Matlab shows that x is a 2*2 array but does not show the content of all cells

```
>> x(1, 1) = { [1 2 3; 4 5 6;
>> x(1, 2) = { 2.5 };
>> x(2, 1) = {'ankara'};
>> x(2, 2) = { [10 : -2 : 0]
>> x
x =

    [3x3 double]    [    2.5000]
    'ankara'        [1x6 double]
```

- Alternatively the following statement creates the same cell array (**content addressing**)

```
x{1, 1} =  [1 2 3; 4 5 6; 7 8 9] ;
x{1, 2} =  2.5 ;
x{2, 1} = 'ankara' ;
x{2, 2} =  [10 : -2 : 0] ;
```

Cell Array Creation

- The two commands:

```
x(i, j) = { y }
```

and

```
x{i, j} = ( y )
```

…have the same outcome.

Both store the content of the variable y inside the (i, j) element of cell array x.

Cell Array Creation

- If you assign a cell to an existing variable that is not a cell Matlab will report an error!

- Example:

```
>> x = 1;
>> x(1, 1) = {1 : 4}
??? Subscripted assignment dimension
  mismatch.
```

- How can you correct this ?

```
>> clear x
```

Cell Array Manipulation

- Cell array manipulation is a natural extension to the standard array manipulation we have used so far

- Example #1:

```
x =

    [3x3 double]    [    2.5000]
    'ankara'        [1x6 double]

>> y(1, 1) = { [10 20 30; 40 50 60; 70 80 90] };
>> y(2, 1) = {'istanbul'};
>> y(1, 2) = { 5.5 };
>> y(2, 2) = { [0 : 2 : 10] };
>> y
y =

    [3x3 double]    [    5.5000]
    'istanbul'      [1x6 double]


>>z = [x; y]
 z =

    [3x3 double]    [    2.5000]
    'ankara'        [1x6 double]
    [3x3 double]    [    5.5000]
    'istanbul'      [1x6 double]
```

Cell Array Manipulation

```
>>a = z([1, 3], :)

a =

    [3x3 double]    [2.5000]
    [3x3 double]    [5.5000]

>>size(a)
ans =

     2     2
```

Retrieving cell array content

- There are two different ways to retrieve the content of a cell array x :

  - **( ) identify the cells without looking at their content (cell indexing)**
    - Example: x(1,2)

  - **{ } access the content of the cells (content addressing)**
    - Example: x{1,2}

Retrieving cell array content

```
>> x(1, 1) = { [1 2 3; 4 5 6; 7 8 9] };
>> x(1, 2) = {'ankara' };
>> x(2, 1) = { [1 : 10] };
>> x(2, 2) = { 'A' };
```

- How can we retrieve the content of a cell within a cell array?
  - Option #1: apply content addressing

```
>> b = x{1, 1}
b =
     1     2     3
     4     5     6
     7     8     9


>> c = x{1, 2}
c =
   ankara
```

```
>> class(b)
double
```

```
>> class(c)
char
```

Retrieving cell array content

```
>> x(1, 1) = { [1 2 3; 4 5 6; 7 8 9] };
>> x(1, 2) = {'ankara' };
>> x(2, 1) = { [1 : 10] };
>> x(2, 2) = { 'A' };
```

- How can we retrieve the content of a cell within a cell array?
  - Option #2: apply cell indexing

```
>> b = x(1, 1)
b =
       [3x3 doub
>> c = x(1, 2)
c =
     'ankara'
```

```
>> class(b)
cell
```

```
>> class(c)
cell
```

Cell Arrays

- Unlike the numeric, character, and symbolic arrays, the cell array can store different types of data inside the same array. Each element in the array is also an array. For example, consider these three different arrays:

  A = 1:3;

  B = ['abcdefg'];

  C = single([1,2,3;4,5,6]);

- We have created three separate arrays, all of a different data type and size. A is a double, B is a char, and C is a single. We can combine them into one cell array by using curly brackets as our cell-array constructor:

  my_cellarray = {A,B,C} returns

  my_cellarray =

  [1x3 double] 'abcdefg' [2x3 single]

- To save space, large arrays are listed just with size information. You can show the entire array by using the celldisp function:

  celldisp(my_cellarray)

  my_cellarray{1} = 1 2 3

  my_cellarray{2} = abcdefg

  my_cellarray{3} = 1 2 3

- The indexing system used for cell arrays is the same as that used in other arrays. You may use either a single index or a row-and-column indexing scheme.

Cell Arrays

- There are two approaches to retrieving information from cell arrays: You can use parentheses, as in

  ```
  my_cellarray(1)
  ans =
  [1x3 double]
  ```

  which returns a result as new cell array. An alternative is to use curly brackets, as in

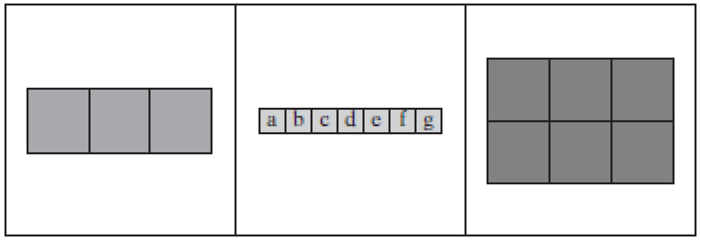
  ```
  my_cellarray{1}
  ans =
  1 2 3
  ```

- In this case the answer is a double. To access a particular element inside an array stored in a cell array, you must use a combination of curly brackets and parentheses:

  ```
  my_cellarray{3}(1,2)
  ans =
  2
  ```

- Cell arrays can become quite complicated. The cellplot function is a useful way to view the structure of the array graphically, as shown in

  ```
  cellplot(my_cellarray)
  ```



Structure Arrays

- Structure arrays are similar to cell arrays. Multiple arrays of differing data types can be stored in structure arrays, just as they can in cell arrays. Instead of using content indexing, however, each matrix stored in a structure array is assigned a location called a *field.*

- For example, using the three arrays from the previous section on cell arrays,

  ```
  A = 1:3;
  B = ['abcdefg'];
  C = single([1,2,3;4,5,6]);
  ```

  we can create a simple structure array called my_structure :

  ```
  my_structure.some_numbers = A
  ```

  which returns

  ```
  my_structure =
  some_numbers: [1 2 3]
  ```

- The name of the structure array is my_structure . It has one field, called some_numbers. We can now add the content in the character matrix B to a second field called some_letters:

  ```
  my_structure.some_letters = B
  my_structure =
  some_numbers: [1 2 3]
  some_letters: 'abcdefg'
  ```

Structure Arrays

- Finally, we add the single-precision numbers in matrix C to a third field called some_more_numbers:

```
my_structure.some_more_numbers = C
my_structure =
some_numbers: [1 2 3]
some_letters: 'abcdefg'
some_more_numbers: [2x3 single]
```

- We can add more content to the structure, and expand its size, by adding more matrices to the fields we've defined:

```
my_structure(2).some_numbers = [2 4 6 8]
my_structure =
1x2 struct array with fields:
some_numbers
some_letters
some_more_numbers
```

Structure Arrays

- You can access the information in structure arrays by using the matrix name, field name, and index numbers. The syntax is similar to what we have used for other types of matrices.

  ```
  my_structure(2)
  ans =
  some_numbers: [2 4 6 8]
  some_letters: []
  some_more_numbers: []
  ```

- Notice that some_letters and some_more_numbers are empty matrices, because we didn't add information to those fields. To access just a single field, add the field name:

  ```
  my_structure(2).some_numbers
  ans = 2 4 6 8
  ```

- Finally, if you want to know the content of a particular element in a field, you must specify the element index number after the field name:

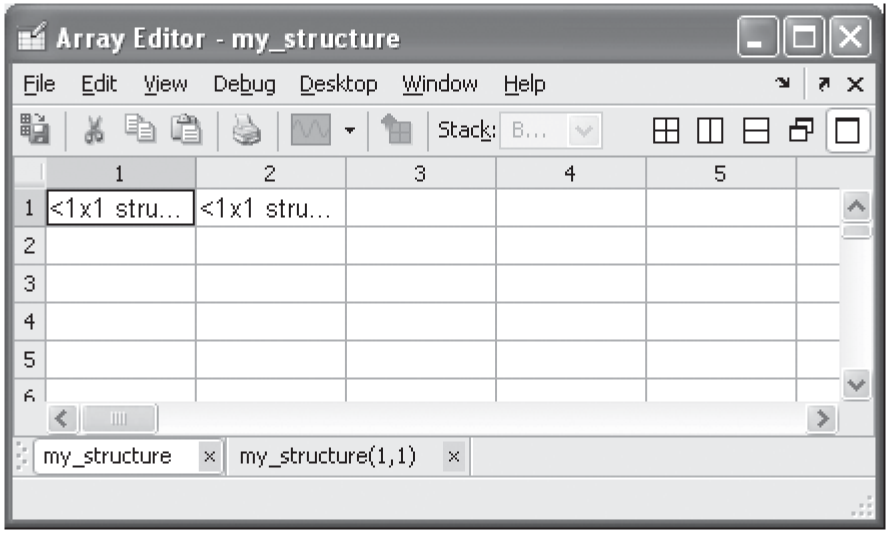
  ```
  my_structure(2).some_numbers(2)
  ans = 4
  ```

- The disp function displays the contents of structure arrays. For example,

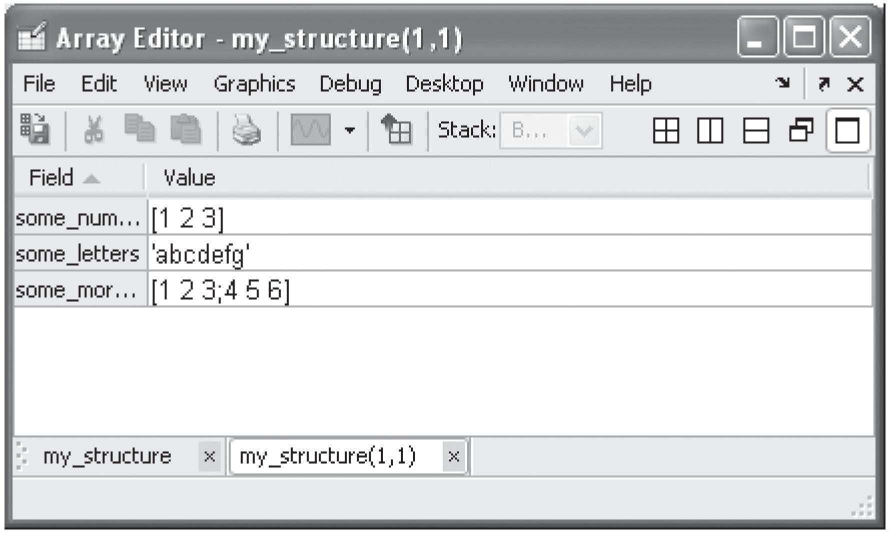
  disp(my_structure(2).some_numbers(2)) returns **4.**

Structure Arrays

- You can also use the array editor to access the content of a structure array (and any other array, for that matter). When you double-click the structure array in the workspace window, the array editor opens.
- If you double-click one of the elements of the structure in the array editor, the editor expands to show you the contents of that element


Array Editor - my_structure
File Edit View Debug Desktop Window Help
Stack: B...
my_structure
my_structure(1,1)

	1	2	3	4	5
1	<1x1 stru...	<1x1 stru...			
2					
3					
4					
5					
6					


Array Editor - my_structure(1,1)
File Edit View Graphics Debug Desktop Window Help
Stack: B...
my_structure
my_structure(1,1)

Field	Value
some_num...	[1 2 3]
some_letters	'abcdefg'
some_mor...	[1 2 3;4 5 6]

Summary

Special Characters	
{ }	cell-array constructor
' '	string data (character information)
abc	character array
⊞	numeric array
◧	symbolic array
✓	logical array
▧	sparse array
{ }	cell array
⊧	structure array

Commands and Functions	
cumsum	finds the cumulative sum of the members of an array
double	changes an array to a double-precision array
eye	creates an identity matrix
format rat	converts the display format to rational numbers (fractions)
int16	16-bit signed integer
int32	32-bit signed integer
int64	64-bit signed integer
int8	8-bit signed integer
intmax	determines the largest integer that can be stored in MATLAB®
intmin	determines the smallest integer that can be stored in MATLAB®
num2str	converts a numeric array to a character array
realmax	determines the largest real number that can be expressed in MATLAB®
realmin	determines the smallest real number that can be expressed in MATLAB®
single	changes an array to a single-precision array
sparse	converts a full-format matrix to a sparse-format matrix
squeeze	removes singleton dimensions from multidimensional arrays
str2num	converts a character array to a numeric array
uint16	16-bit unsigned integer
uint32	32-bit unsigned integer
uint64	64-bit unsigned integer
uint8	8-bit unsigned integer

Structure Array Example

Structure name

Structure field

```
>> guest.name_surname       = 'Ali Veli';
>> guest.gender             = 'Male';
>> guest.age                = 30;           % in years
>> guest.special_food       = 'none';
>> guest
guest =


          gender: 'Male'
    name_surname: 'Ali Veli'
    special_food: 'none'
             age: 30
```

Structure Array Example

- Adding more guests…

```
>> guest(2).name_surname= 'Ayse Dogan';
>> guest(2).gender      = 'Female';
>> guest(2).age         = 15;
>> guest(2).special_food= 'baklava';

>> guest(3).name_surname= 'Selma Cicek';
>> guest(3).gender      = 'Female';
>> guest(3).age         = 45;
>> guest(3).special_food= 'gozleme';

>> guest

guest =

1x3 struct array with fields:

    gender
    name_surname
    special_food
    age
```

Retrieving structure content

- Retrieving a specific guest

```
>> x = guest(2)

x =

          gender: 'Female'
    name_surname: 'Ayse Dogan'
    special_food: 'baklava'
             age: 15
```

Retrieving structure content

- Retrieving information on a specific guest

```
>> x.gender

ans =

Female
>> guest(2).gender

ans =

Female
```

Retrieving structure content

- We can simultaneously retrieve a field from several guests

```
>> guest.age

ans =

    30


ans =

    15


ans =

    45
>> ages = [guest.age]

ages =

    30    15    45
```

Retrieving structure content

- We can simultaneously retrieve a field from several guests

```
>> guest.name_surname

ans =

Ali Veli


ans =

Ayse Dogan


ans =

Selma Cicek

>> names = [guest.name_surname]

names =

Ali VeliAyse DoganSelma Cicek

>> names_cell = {guest.name_surname}

names_cell =

    'Ali Veli'    'Ayse Dogan'    'Selma Cicek'
```