Data Storage:

Computers are made of many small parts, including transistors, capacitors, resistors, magnetic materials, etc. Somehow they have to store information in these materials both temporarily (RAM, or “memory”) and permanently (hard drives).

In previous lectures, we have said that variables and arrays store values. The values actually are stored in memory. But what does that mean physically? How can we store information in physical materials?

Memory consists of (among other things) many, many capacitors. Capacitors are able store charge and “slowly” release charge over time. We can think of the capacitors being in one of two states, charged or discharged. Alternatively, we can think of each capacitor as being in a state 1 or a state 0.

Hard drives contain a magnetic material that can stay permanently magnetized. Each small section of the material will be either magnetized or unmagnetized. It is convenient to think of these small sections as being in a state 1 or state 0.

By combining many of these “1s” and “0s” we can store a lot of information. The 1s and 0s are called binary digits, or bits. A group of 8 bits makes a byte.

Since there is a finite amount of material in a computer, there is a finite amount of data that can be stored in a computer both in the short-term and long-term.

Each time you create a variable in memory, a certain amount of memory is allocated for that variable based on its data type (or class).

Data Types (Classes):

There are many different data types. Thus far, we have focused on three major data types: floating-point double precision number, character, and logical.

(1A) Floating point numbers – Double Precision

By default, Matlab/Octave assumes all numerical data are double precision floating-point numbers. Matlab/Octave allocates 8 bytes of memory for each variable and array element created.

Using the **realmax** and **realmin** commands, we can see the maximum and minimum value for double precision floating point numbers.

The maximum value that can be stored is ~1.7977e+308
The minimum value that can be stored is ~ 2.2251e-308

We run into problems when we try to store numbers that are too big or too small.

In m-file:
```
a = 1e400
b = 1e-400
a*b
(1e200/1e-200) * 1e-200
```

Output:
```
a = Inf
b = 0
ans = NaN
ans = Inf
```

A double precision floating-point variable can store ~16 digits of precision.

In m-file:
```
format long
a = 1.1;
b = 1.100000000000001;
c = b-a;   d = c*1e15;
fprintf('b   = %28.25f\n',b)
fprintf('a   = %28.25f\n',a)
fprintf('b-a = %28.25f\n',c)
fprintf('(b-a)*1e15 = %28.25f\n',d)
```

Output:
```
b   =  1.10000000000000009769962617
a   =  1.1000000000000000888178420
b-a =  0.00000000000000008881784197
(b-a)*1e15 =  0.8881784197001252323389053
```

We get 0.888178... instead of 1, an 11% error. Watch out! Errors can accumulate.

In double precision arrays, each element is allocated 8 bytes. A 2x5 array is allocated 80 bytes.

(1B) Floating point numbers – Single Precision

In the not-too-distant past, programmers had to worry about not having enough memory for very large codes and often found it useful to use single precision floating point numbers. Matlab/Octave allocates half the memory as a double precision number (4 bytes instead of 8 bytes), but this comes at a cost – the range of numbers you can store is smaller and the precision is worse (8 digits). Nowadays, lack of memory is an issue only if your code is long and does many computations.

The maximum value that can be stored is ~3.4028e+038
The minimum value that can be stored is ~ 1.1755e-038

You can create single precision variables and arrays with the **single( )** command.

In m-file:
```
a = 1.1;
fprintf('a   = %28.25f\n',a)
b = single(a);
fprintf('b   = %28.25f\n',b)
```

Output:
```
a   =  1.1000000000000000888178420
b   =  1.1000000238418579101562500
```

Error accumulation can become significant much faster.

In single precision arrays, each element is allocated 4 bytes. A 2x5 array is allocated 40 bytes.

(2) Integers

Integers usually are used for counting. When creating integers, you get to decide the amount of storage space to allocate for the integer and whether it is *signed* (both positive and negative values are allowed) or *unsigned* (only positive values are allowed).

To create a *signed* integer, use the **int*( )** command, where * can be 8, 16, 32, or 64. These numbers correspond to the amount of bits of memory you allocate for the integer (8 bits = 1 byte. Why 8 bits in a byte? Why 12 items in a dozen? Why 3 strikes and you're out? It is something that people decided on somewhat arbitrarily.).

For an *unsigned* integer that has 8 bits of memory allocated for it,
The maximum value = 255
The minimum value = 0

Here is why… a short lesson in binary.

Binary is a number system that is 'base 2' meaning each digit can only have the values of 0 and 1 (two numbers). We are used to the decimal number system, or 'base 10' number system, in which each digit can have a value from 0 to 9 (ten numbers).

Example: Decimal number system
6304 has four digits in the decimal number system. You can express number as,

$6*10^3 + 3*10^2 + 0*10^1 + 4*10^0$

binary	decimal		binary	Decimal
0	0		1000	8
1	1		1010	10
10	2		10000	16
11	3		100000	32
100	4		1000000	64
101	5		1111111	127
110	6		10000000	128
111	7		11111111	255

Example: Binary number system
14 has two digits in the decimal number system. What is this number in the binary number system?

$1*2^3 + 1*2^2 + 1*2^1 + 2^0 = 1110$ (This is not one thousand, one hundred, ten)

Kilobyte = 2^10 = 1,024 ~ 10^3 bytes
Megabyte = 2^20 = 1,048,576 ~ 10^6 bytes
Gigabyte = 2^30 = 1,073,741,824 ~ 10^9 bytes
Terabyte = 2^40 = 1,099,511,627,776 ~ 10^{12} bytes

With 8 bits, we can store 256 unique pieces of information; that is, there are 256 unique ways to arrange the 1s and 0s.

If we want the ability to create negative integers, we can use one of the 8 bits to store the sign (plus or minus). The other seven bits allow us to store numbers. The minimum and maximum values for an *unsigned* integer that has 8 bits of memory allocated for it is -128 and 127 (256 different numbers).

As we increase the number of bits to 16 for an unsigned integer,

0000000000000000	0
0111111111111111	32767
1000000000000001	32768
1111111111111111	65536

Just like with floating point numbers, the more memory we allocate for an integer variable the higher the number we can store in it.

In integer arrays, each element is allocated N bits, where N is 8, 16, 32, or 64. A 2x5 array is allocated 10xN bits.

(3A) Character variables

A computer stores all data as 1s and 0s. This is true for character data as well. Although computers only can understand data as numbers, data associated with character variables are treated differently than data associated with numerical variables. If data were stored in a character variable, the 1s and 0s in memory will mean something different than if the data were stored in a numerical variable (such as a double precision floating-point variable).

All characters (such as 's' or 'A' or '%' or '5') have an integer associated with them. For example, the character 'A' is associated with 65, while the character '4' is associated with 52. The ASCII (American Standard Code for Information Interchange) table lists the integer associated with each character. (http://www.asciitable.com/)

(Note: If someone says they want a document in "ASCII format," this means is they want "plain" text with no formatting such as bold, italics, or underscoring. ASCII format allows the user to easily import the file into his own applications without issues. Notepad creates ASCII text, as does MSWord when you can save a file as "text only.")

Characters are allocated 1 byte (8 bits) of memory, just like when integers are created using the **int8( )** or **uint8( )** commands. We are able to store 256 different characters (notice how the ASCII table only goes from 0 to 255).

If we stored the number 4 as a signed integer that is allocated 8 bits for data storage in a variable **A**,

```
A = int8(4)
```

The number 4 would be stored as 00000100. However, if we stored the character 4 in a variable called **B**,

```
A = '4'
```

The character 4 would be stored as 00110100 (the number 52) since the number 52 is the ASCII code for the character '4'. We can convert a character to its ASCII code, and vice versa, in the following manner:

In m-file:

```
A = int8(4)      % The signed integer variable A stores the number 4
B = '4'          % The character variable B stores the character '4'
C = double(A)    % The double precision floating-point variable C contains the number 4
D = double(B)    % The double precision floating-point variable D contains the number
                 %     associated with the character stored in B.
F = char(105)    % The character variable F contains the character associated with the
                 %    number 105, which is 'i'
```

After execution:

```
A = 4
B = 4
C = 4
D = 52
F = i
```

Look at the amount of memory allocated for each variable using the **whos** command,

```
 Attr   Name  Size    Bytes   Class
 ==== ====   ====  ====  =====
            A      1x1     1     int8
            B      1x1     1     char
            C      1x1     8     double
            D      1x1     8     double
            F      1x1     1     char

Total is 5 elements using 19 bytes
```

Strings of characters are stored in arrays. Each character is stored in one element.

In m-file:

```
mystring = 'hello'
mystring2 = 'world'
superstring = [mystring , mystring2]
disp( superstring(2) )
```

After execution:

```
mystring = hello
mystring2 = world
superstring = helloworld
e
```

Look at the amount of memory allocated for each array using the **whos** command.

```
Variables in the current scope:
   Attr Name          Size          Bytes    Class
 ==== ====         ====         =====  =====
     mystring        1x5             5      char
     mystring2       1x5             5      char
     superstring     1x10           10      char
```

(3B) 2D Character Arrays

You can use the **char( )** function to create 2D character arrays.

In m-file:

```
names = char('Paul' , 'Phillip' , 'Rick' , 'Susan')
a = names(1,5)
```

After execution:

```
names =
Paul
Phillip
Rick
Susan
a =
```

The array **names** is automatically made into a 4x7 array. The **char( )** function automatically pads the character array with spaces so that every row has the same number of elements. Blank spaces will be inserted after names shorter than 7 characters (3 blank spaces after ‘Paul’).

(4) Other data types

There are a couple other data types that are useful to know.

Logical:

Logical variables only can have two values, true or false. These true or false states are represented by the numbers of 1 and 0. We have discussed logical variables earlier in the course.

In m-file:

```
a = 34
b = 55
c = a > b
d = a < b
f = [23, 45, -12, 97, 65, 9]
g = [65, 44,  32, -10, 34, 0]
h = f > g
```

After execution:

```
a =  34
b =  55
c = 0
d =  1
f =    23   45  -12   97   65    9
g =   65   44   32  -10   34    0
h =   0   1   0   1   1   1
```

The logical function **find( )** returns the array elements contains true (1) values.

In m-file:

```
f = [23, 45, -12, 97, 65, 9];
g = [65, 44,  32,-10, 34, 0];
find(f  > g)
h = f  > g
find(h)
```

After execution:

```
ans =   2   4   5   6
h =   0   1   0   1   1   1
ans =   2   4   5   6
```

Complex:

Complex numbers are found in many areas of science including optics, signal processing, control systems, fluid mechanics, and quantum mechanics. A complex number has a real and imaginary part.

B + Ci

where B is the real part and C is the imaginary part.

In m-file:

```
a = 3.0 + 2.0i
b = a^2
```

After execution:

```
a = 3 + 2i
b = 5 + 12i
```

Double precision complex numbers are allocated 8x2 (16) bytes each.