Review of data types:

We have a finite amount of memory for data storage. As programmers, we must decide how much memory we should allocate for each variable.

The amount of memory we allocate will determine how accurate the results will be. We cannot store numbers with infinite precision.

In our decimal number system, if you allocate 4 digits to store number data, you can only store the numbers 0 to 9999. You can permute 0-9 in 4 digits only 10000 ways.

In the binary number system, if you allocate 4 binary digits (bits) to store number data, you can only store the numbers 0 to 15. You can permute 0-1 in 4 digits only 16 ways.

binary		decimal
0000	=	0
1111	=	15

If you allocate 8 bits (1 byte) to store number data, you can store 256 different pieces of information.

binary		decimal
11111111	=	255

In RAM, information is stored in capacitors. If a capacitor is charged, that represents 1. If a capacitor is discharged, that represents 0. There are only 2 states that the capacitor can be in. The 1s and 0s are just our abstract representation of the state of capacitors.

Characters are given 1 byte (8 bits) for information storage. We can create 256 different characters.

Beyond MAE10:

Matlab/Octave is designed to be very convenient when working with matrices. Much of the world's computing power is dedicated to solving sets of linear equations, **Ax** = **b**, where **A** is a known NxN array, **x** is a Nx1 array, and **b** is a known Nx1 array. Weather forecasting, graphics rendering, and many engineering problems require solving the equation **Ax** = **b**. A lot of time and effort has gone into trying to solve **Ax** = **b** as efficiently as possible.

Let's look at a simple example:

2 apples and 1 pear cost $4
1 apple and 2 pears cost $5
How much does each apple and pear cost?

If we say apple = $\mathbf{x_1}$, pear = $\mathbf{x_2}$, then we can write the following set of linear equations,

$2x_1 + 1x_2 = 4$
$1x_1 + 2x_2 = 5$

or

$$\underbrace{\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}}_{x} = \underbrace{\begin{bmatrix} 4 \\ 5 \end{bmatrix}}_{b}$$

By simple substitution we can calculate that $\mathbf{x_1}$ = 1 and $\mathbf{x_2}$ = 2. However, what if we had 50 different foods (50 different $\mathbf{x_i}$)? This problem would be straight-forward to solve, but it would take a very long time.

Fortunately, Matlab/Octave allows you to solve **Ax = b** easily by multiplying **b** with the inverse of **A**.

```
> A = [2 1; 1 2];
> b = [4 ; 5];
> x = inv(A)*b;
> disp(x)
   1.00000
   2.00000
```

FINAL THOUGHTS FOR THE COURSE:

The past 60 years has seen computers becoming smaller, faster, cheaper, more numerous, and more user-friendly. In this relatively short period of human history, we have gone from a hand-full of people developing only a few giant metal behemoths that weighed tons and took up entire rooms, to almost everyone having powerful cell phones that fit comfortably into their pocket.

The trend of smaller, faster, cheaper, more numerous, and more user-friendly computers likely will continue well into the future.

HOWEVER, the basic building blocks of computer programming – variables, arrays, data types, Boolean logic, if statements, loops, functions, and everything else you learned in this course – have remained. Every high-level computer language that is used today (Fortran, C, C++, Java, Matlab, Mathematica, etc) still contains these basic building blocks.

It is possible that computer programs will become more user-friendly and may feature many shortcuts for commonly-used functions or commands. For example, Matlab/Octave offers many built-in functions such as **mean( )** and **std( )** that calculate the average and standard deviation of an array, respectively. These two programs also allow for much easier array manipulation than other languages, such as Fortran.

If you choose a career that involves computer programming, the skills that you learned in MAE10 will be relevant to your job. Good luck!

Review session for Final Exam:

(1) Array manipulation:

In mfile.m:

```
x = [3,7,4,6,7]
y = x(2:4)
a = [ 0:2:6  ;  1:3:10 ;  3, 5, 9, 11]
b = a(1:2,1:2)
c = a(2:3,3:4)
d = [b , c]
```

At command line:

```
> mfile
x =   3   7   4   6   7
y =   7   4   6
a =    0    2    4    6
       1    4    7   10
       3    5    9   11
b =   0   2
      1   4
c =    7   10
       9   11
d =    0    2    7   10
       1    4    9   11
```

(2) Order of operations – PEMDAS

In mfile.m:

```
a = [3 , 6, 2, -1];
b = a(2)-a(3)*a(4)^a(1)/a(3) + a(3)
```

At command line:

```
> mfile
b =  9
```

(3A) Logical expressions and if statements

In mfile.m:

```
a = 20;  b = 10;
a>b | (a==20 & b>5)
if( a>b | (a==20 & b>5) )
   disp('hi')
   if(b>a)
      disp('cheese')
   elseif(a==20)
      disp('railroad')
   else
      disp('yak')
   end
elseif( b>5 )
   disp('bye')
else(b==10)
   disp('hola')
end
```

At command line:

```
> mfile
ans =  1
hi
railroad
```

(3B) Switch and case

In mfile.m:

```
animal = 'ape';
horse = 'zebra';
switch(animal)
   case{horse,'goat'}
      disp('hi')
   case{'ape','snake'}
      disp('bye')
   otherwise
      disp('hola')
end
```

At command line:

```
> mfile
bye
```

(4) For and while loops

In mfile.m:

```
a = [3 , 1, 1, -1 ; 4, 3, 2, 1];
summy = 0;
for i=1:2
   for j=1:4
      summy = summy + a(i,j);
   end
end
avg = summy / numel(a)

summy = 0;
i=1;
while(i<=2)
   j = 1;
   while(j<=4)
      summy = summy + a(i,j);
      j = j + 1;
   end
   i = i + 1;
end
avg = summy / numel(a)
```

At command line:

```
> mfile
avg =  1.7500
avg =  1.7500
```

(5) Input/output

In mfile.m:

```
a = [3 , 1, 1, -1 ; 4, 3, 2, 1];
fprintf('%7.2f %7.2f %7.2f %7.2f\n', a')
fileid = fopen('output.txt','w');
fprintf(fileid,'%7.2f %7.2f %7.2f %7.2f\n', a')
fclose(fileid);

fileid = fopen('output.txt');
b = fscanf(fileid,'%f', [4,2])
fclose(fileid);
fprintf('%7.2f %7.2f %7.2f %7.2f\n', b')
```

At command line:

```
> mfile
   3.00    1.00    1.00   -1.00
   4.00    3.00    2.00    1.00
   3.00    4.00    1.00    3.00
   1.00    2.00   -1.00    1.00
```

(6) Functions

In mfile.m:

```
cheese = @(z) z^2
a = 2;  b = 5;
w = 20; t = 40;
x = 100;  y = 200;
[c,d] = funky(a,b)
disp(x)
disp(y)
disp(cheese(a))
disp(cheese(b))
```

In funky.m:

```
function [x,y] = funky(w,t)
x = w+t;
y = w-t;
endfunction
```

At command line:

```
> mfile
c =  7
d = -3
 100
 200
 4
25
```

(7) Plotting

In mfile.m:

```
x = (0:1:20);
y = x.^2;
z = x.^3

%plot(x,y,'-ro' , x,z,'-gx')
%xlabel('my xlabel')

%plot(x,y,'-ro')
%hold on
%plot(x,z,'-gx')
%xlabel('my xlabel')
%hold off

%subplot(1,2,1)
%plot(x,y,'-ro')
%subplot(1,2,2)
%plot(x,y,'-gx')
```

(8) Data types

```
a = 55;          %This is double precision by default
single(a)
int8(a)
int64(a)
char(a)
```

(9) Binary

Binary code – This is a base 2 counting system.
01010101 = 64 + 16 + 4 + 1 = 85

85 is the decimal (base 10) representation of 01010101 in the binary (base 2) counting system.