```
###########################################################
#           cs315 Week 10
#
#   ->  2 dimensional arrays (or matrix)
#
###########################################################

* Multidimensional arrays
    * arrays so far have been 1 dimensional (flat)
        --> need 1 index to access an element
    * multidimensional arrays use multiple indices to access elements
    * we will focus on 2-dimensional arrays (matrices)
        --> Need 2 indices to access an element
    * Note: 'dimension' here refers to the size of the array  not 'dimension' in the mathematical/spatial sense
    * Important:
        --> 1 dimensional arrays needed a base address and length to be complete
        --> 2 dimensional arrays need a base address  height  and width to be complete

* Matrices
    * represented as a 2 dimensional array
    * size of a matrix is specified with HEIGHT first and WIDTH second
        --> Ex. (7, 12) specifies a matrix which is 7 rows high and 12 columns wide
    * indices are given with the ROW first and COLUMN second
        --> Ex. (2, 5) is the index for row 2 and column index 5

* Storage orders
    * row major - stores array as a sequence of rows
        --> used in Java, C/C++, Python, etc.
    * column major -  stores array as a sequence of columns
        --> used in Fortran, MATLAB, etc.
    --> Ex: 4 X 3 matrix

        +-----+-----+-----+
        | 17  | 21  | 32  |
        +-----+-----+-----+
        | 47  | 51  | 68  |
        +-----+-----+-----+
        | 72  | 89  | 90  |
        +-----+-----+-----+
        | 104 | 117 | 121 |
        +-----+-----+-----+

    * when stored as row major, memory is:
        --> 17, 21, 32, 47, 51, 68, 72, 89, 90, 104 117 121
    * when stored as column major, memory is:
        --> 17, 47, 72, 104, 21, 51, 89, 117, 32, 68, 90, 121

* Address calculation
    * address calculations are affected by storage order
```

--> storage order must be known when calculating addresses
* the same general equation is used to calculate addresses for both storage orders but use of the variables differs
* indices here are 0 indexed
    --> (0 0) is the upper left element
    --> for a M x N matrix (M-1, N-1) is the bottom right element

    --> equation: i = b + s * (e*k + n')
    --> variables:

variable	row major	column major
b	base address	base address
s	element size	element size
e	width (number of columns)	height (number of rows)
k	row index	column index
n'	column index	row index

--> example: if the matrix above is a matrix of words whose base address if 0x10040000 calculate the address of (2, 1) (element value 89)
- ~ row major:
    * e = 3
    * k = 2
    * n' = 1
    * i = 0x1004 0000 + 4 * (3*2 + 1) = 0x1004 0000 + 2810 = 0x10040000 + 0x1C = 0x1004 001C

- ~ column major:
    * e = 4
    * k = 1
    * n' = 2
    * i = 0x1004 0000 + 4 * (4*1 + 2) = 0x1004 0000 + 2410 = 0x10040000 + 0x18 = 0x1004 0018

Reminder: Do not mix hexadecimal and decimal arithmetic!

* Note:
    --> for 1 dimensional arrays address calculation is i = b + s*n
    --> for 2 dimensional arrays n becomes (e*k + n')
    --> all multidimensional arrays are stored in a 1 dimensional memory therefore address calculations for multidimensional arrays must eventually be reduced to a 1 dimensional address calculation

Program example: calculate an address in a column-major matrix of words
```
    #  $t0 -  base address (b)
    #  $t1 -  height (e)
```

```
    #  $t2 -  width
    #  $t3 -  row index (n')
    #  $t4 -  column index (k)
    #  $t5 -  index address (i, to be calculated)
    #  words are 4 bytes each, therefore s = 4

-->
    mul $t5  $t1  $t4    # $t5 <-- e*k
    add $t5  $t5  $t3    # $t5 <-- e*k + n'
    sll $t5  $t5  2      # $t5 <-- s*(e*k + n')
    add $t5  $t0  $t5    # $t5 <-- b + s*(e*k + n')
<--
    # $t5 is now the address of element ($t3  $t4)


    +-------------------------------------+
    | base addresses are in HEX (base 16) |
    +-------------------------------------+


Assume array base address for the following >>=>> 0x1000 BC0C

    1) address of 9th element if each array element takes 1 byte
        in 1 dimensional array (regular array) = array base address + size * index
                size * index in decimal = 1 * 8 = 8
                size * index in base 16 = 0x0000 0008

                         1
                0x1000 BC0C
            +             8
            ---------------
                0x1000 BC1(20)

        %16     0x1000 BC14     <-- result


    2) address of 9th element if each array element takes 2 byte
        in 1 dimensional array (regular array) = array base address + size * index
                size * index in decimal = 2 * 8 = 16
                size * index in base 16 = 0x0000 0010

                0x1000 BC0C
            +            10
            ---------------
                0x1000 BC1C     <-- result


    3) address of 9th element if each array element takes 4 byte
        in 1 dimensional array (regular array) = array base address + size * index
```

```
        size * index in decimal = 4 * 8 = 32
        size * index in base 16 = 0x0000 0020

        0x1000 BC0C
    +            20
    ---------------
        0x1000 BC2C     <-- result


4) address of 9th element if each array element takes 8 byte
    in 1 dimensional array (regular array) = array base address + size * index
        size * index in decimal = 8 * 8 = 0x0000 0064
        size * index in base 16 = 40

        0x1000 BC0C
    +            40
    ---------------
        0x1000 BC4C     <-- result


5) if 18 bytes structure is being stored (each element is 1 byte). The address of the next element that would be in * word boundary *
    in 1 dimensional array (regular array) = array base address + size * index

    0x1000 BC0C:    _x_|_x_|_x_|_x_
    0x1000 BC10:    _x_|_x_|_x_|_x_
    0x1000 BC14:    _x_|_x_|_x_|_x_
    0x1000 BC18:    _x_|_x_|_x_|_x_
    0x1000 BC1C:    _x_|_x_|_w_|_w  <<< 'w' <--> waste
    0x1000 BC20:    ___|___|___|___
                     ^
        start of word boundary

    * we waste 2 bytes and store the next element in the start of word boundary

    20 is divisible by 4 (18 < 20 and 20 % 4 = 0)
    20 in base 16 = 0x0000 0014

                 1
        0x1000 BC0C
    +            14
        -----------
        0x1000 BC2(16)

    %16     0x1000 BC20     <-- result


6) in 2 dimensional array of integers (4 bytes)

Given:
```

```
50 rows     <-->    height
100 columns <-->    width

Essentially, in a simple language:
    row major = base address + size * (width * row index + column index)     <-- in row major, we are concerned about number of rows to skip
                        ^        ^
            hex (base 16)   base 16 addition

    column major = base address + size * (height * column index + row index) <-- in column major, we are concerned about number of columns to skip
                        ^           ^
            hex (base 16)   base 16 addition

Order of numbers:
        (10,      15)
         ^         ^
    row index  column index

a) address of (10, 15) in row major:
    0x1000 BC0C + 4 * (100 * 10  + 15)
    0x1000 BC0C + 4 * (1015)
    0x1000 BC0C + 4060

    4060 in base 16 = 0x000 0FDC

    4060| 12 (C)    ^
     253| 13 (D)    ^
      15| 15 (F)    ^   rewrite from bottom to up
       0|

               1 1
        0x1000 BC0C
    +           FDC
        -----------
        0x1000 C(27)E(24)

%16     0x1000 CBE8     <-- result


b) address of (10, 15) in column major:
    0x1000 BC0C + 4 * (50 * 15  + 10)
    0x1000 BC0C + 4 * (760)
    0x1000 BC0C + 3040

    3040 in base 16 = BE0

    3040| 0         ^
     190| 14 (E)    ^
      11| 11 (B)    ^   rewrite from bottom to up
       0|
```

```
                 1
          0x1000 BC0C
    +             BE0
          -----------
          0x1000 C(23)EC

%16       0x1000 C7EC     <-- result
```