CSE220 – 2D Arrays in MIPS

- 2D Arrays are stored in memory as contiguously allocated bytes, identical to the way 1D arrays are stored.
- To correctly access the elements in the array, one must know the number of columns and rows of the array.
- Additionally, there are many different approaches for laying out the data values in memory. The two use approaches are
  - Row-major order
  - Column-major order

- Suppose we have this 3×5, 2D array, where char ‘a’ is stored at index [0][0]:
  
  a b c d e
  f g h i j
  k l m n o

  - If we use row-major order, elements are stored in memory like this:
    
    a b c d e f g h i j k l m n o

  - If we use column-major order, elements are stored in this order:
    
    a f k b g l c h m d i n e j o

- The approach used, row-major or column-major, is usually dependent on the HLL.
- At the assembly level, we have full control of memory and place all data into memory ourselves. Therefore we get to choose. In this course we will use the more commonly used approach: row-major order.
- Assuming we use row-major order to store elements in a 2D array, the memory address of element [i][j] is
  
  \[ \text{addr} = \text{base_addr} + i \times \text{size_of_a_row_in_bytes} + j \times \text{size_of_a_column_in_bytes} \]

  - This translates into:
    
    \[ \text{addr} = \text{base_addr} + i \times \text{num_columns} \times \text{elem_size_in_bytes} + j \times \text{elem_size_in_bytes} \]

    \[ \text{addr} = \text{base_addr} + \text{elem_size_in_bytes} \times (i \times \text{num_columns} + j) \]

- Visually, in our 32-bit architecture, a 3 x 4 integer array would be stored in memory as shown below. Note that each rectangle is the size of an integer (4 bytes).
To calculate the address of \( a[i][j] \): 
\[
\text{(Base address of } a \text{) } + 
(4 \text{ bytes per int} \times i \times (\text{num of columns}) + 
(4 \text{ bytes per int} \times j
\]

**Ex:** address of \( a[1][2] \) = (Base address of \( a \)) + 4*1*4 + 4*2  
= (Base address of \( a \)) + 24

**Ex:** address of \( a[2][3] \) = (Base address of \( a \)) + 4*2*4 + 4*3  
= (Base address of \( a \)) + 44

### Declaring 2D Arrays in MIPS

- Since 2D arrays are contiguous elements in memory, they can declared and initialized as such.
  
  **Ex:** `alphabet_array`: `.byte  a, b, c, d, e, f, g, h, i, j, k, l, m, n, o`  
  
  **Ex:** `a_array`: `.word  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11`

- They can also be declared using the total number of bytes for the array without initialization
  
  **Ex:** `alphabet_array`: `.space 15`  
  \#2 = 3*5*(size of char)  
  
  **Ex:** `a_array`: `.space 60`  
  \#2 = 3*4*(size of int)

- Declarations can be split into multiple statements. The assembler will place them contiguously in memory
  
  **Ex:** `alphabet_array`: `.byte a, b, c, d, e`  
  
  `.byte f, g, h, i, j`  
  
  `.byte k, l, m, n, o`
Example Program

# Demonstrates traversing a 2D array of words, storing a value at each
# cell of the array.

.data
matrix: .space 60 # 5 x 3 matrix of 4-byte words
rows:   .word 5
columns: .word 3

.text
la $t0, matrix
lw $t1, rows    # number of rows
lw $t2, columns # number of columns
li $t3, 0  # i, row counter
li $t9, 65  # value to store in array

row_loop:
  li $t4, 0  # j, column counter
col_loop:
    # Although this array traversal could be implemented by simply
    # adding 4 to a starting address (e.g., matrix's address), the
    # point here is to show the arithmetic of computing the address
    # of an element in a 2D array.
    # addr = base_addr + i * num_columns * elem_size_in_bytes + j * elem_size_in_bytes
    # addr = base_addr + elem_size_in_bytes * (i * num_columns + j)
    mul $t5, $t3, $t2 # i * num_columns
    add $t5, $t5, $t4 # i * num_columns + j
    sll $t5, $t5, 2   # 4*(i * num_columns + j)
    # Mult by 4 b/c we have an array of 4-byte words
    add $t5, $t5, $t0 # base_addr + 4*(i * num_columns + j)
    sw $t9, 0($t5)
    addi $t4, $t4, 1 # j++
    addi $t9, $t9, 1 # generate next value to save
    blt $t4, $t2, col_loop
col_loop_done:
addi $t3, $t3, 1 # i++
blt $t3, $t1, row_loop
row_loop_done:
  # Finish the program here