Subscript Binding & Array Categories

- *Static*: subscript ranges are statically bound and storage allocation is static (before run-time)
  - Advantage: efficiency (no dynamic allocation)
  - Example: In C and C++ arrays that include the `static` modifier are *static*
  - **`static int myarray[3] = {2, 3, 4};`**

Subscript Binding & Array Categories

- *Fixed stack-dynamic*: subscript ranges are statically bound, but the allocation is done at declaration time
  - Advantage: space efficiency
  - Example: arrays without static modifier are fixed stack-dynamic
  - `int array[3] = {2, 3, 4};`

Subscript Binding Time

- *Stack-dynamic*: subscript ranges are dynamically bound and the storage allocation is dynamic (done at run-time)
  - Advantage: flexibility (the size of an array need not be known until the array is to be used)
  - Example: In Ada, you can use stack-dynamic arrays as

```
Get(List_Len);
declare
  List: array (1..List_Len) of Integer
begin
...
end;
```

Subscript Binding Time

- *Fixed heap-dynamic*: similar to fixed stack-dynamic: storage binding is dynamic but fixed after allocation (i.e., binding is done when requested & storage is allocated from heap, not stack)
  - Example: In **C/C++,** using malloc/free to allocate/deallocate memory from the heap
  - **Java** has fixed heap dynamic arrays
  - **C#** includes a second array class ArrayList that provides fixed heap-dynamic

Subscript Binding Time

- *Heap-dynamic*: binding of subscript ranges and storage allocation is dynamic and can change any number of times
  - Advantage: flexibility (arrays can grow or shrink during program execution)
  - Examples: Perl, JavaScript, Python, and Ruby support *heap-dynamic* arrays
  - Perl: @states = (“Idaho",“Washington",“Oregon");
  - Python: a = [1.25, 233, 3.141519, 0, -1]

Heterogeneous Arrays

- A *heterogeneous array* is one in which the elements need not be of the same type
- Supported by Perl, Python, JavaScript, and Ruby
- Python example

```
o a = array([12, 3.5, -1, 'two'])
```

Array Initialization

- C-based languages
  - o `int list [] = {1, 3, 5, 7}`
  - o `char *names [] = {“Mike”, “Fred”,“Mary Lou”};`
- Ada
  - o
```
List : array (1..5) of Integer :=
    (1 => 17, 3 => 34, others => 0);
```
- Python
  - o List comprehensions
```
list = [x ** 2 for x in range(12) if x % 3 == 0]
```
    puts `[0, 9, 36, 81]` in `list`

Array Operations

- APL - most powerful array processing operations for vectors and matrices
- Ada allows array assignment but also concatenation
- Python supports array catenation and element membership operations

Array Operations

- Ruby also provides array catenation
- Fortran provides *elemental* operations because they are between pairs of array elements
  - For example, + operator between two arrays results in an array of the sums of the element pairs of the two arrays: `C = A + B`

Memory for Arrays

- For 1D arrays
  - contiguous block of memory with equal amount of space for each element
- Two approaches for multi-dimensional arrays
  - Single block of contiguous memory for all elements
    - Arrays must be rectangular
    - Address of array is starting memory location
  - Implement as arrays of arrays (Java)
    - Jagged arrays are possible
    - Array variable is a pointer (reference)

Implementation of Arrays

- **Access function** maps subscript expressions to an address in the array
  - For `int myarray[5];` what address does `myarray[3]` map to
- Access function for single-dimensioned arrays:

```
address(list[k]) = address (list[lower_bound])
    + ((k-lower_bound) * element_size)
```

Memory Allocation for 2D Array

- Two common ways to organize 2D arrays
  - Row major order (by rows) – used in most languages
  - Column major order (by columns) – used in Fortran

Memory Allocation for 2D Array

```
int A[2][3] = { {1, 2, 3}, {4, 5, 6} };
```

- Row major order

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \qquad = 1\ 2\ 3\ 4\ 5\ 6$$

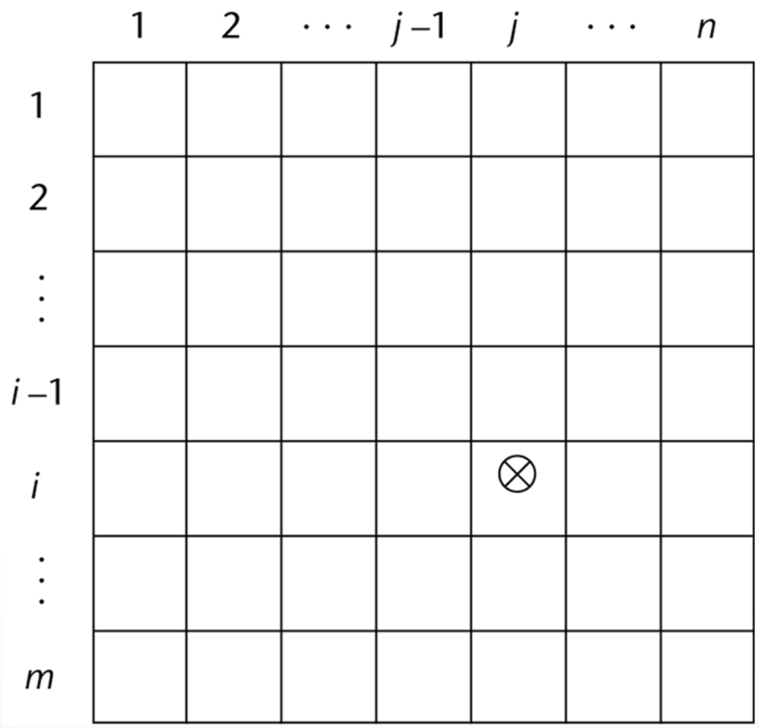
- Column major order

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \qquad = 1\ 4\ 2\ 5\ 3\ 6$$

- [Two-dimensional array indexing exercise]()

Locating an Element in a 2-D Array

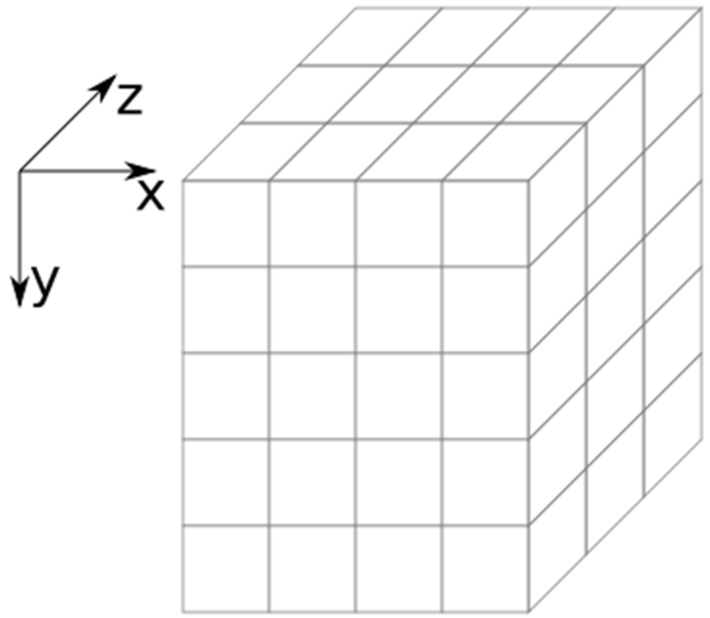
- General format
  Location (a[i,j]) = address of a [0,0] + ((i * n) + j) * element_size



Locating an Element in a 3D Array

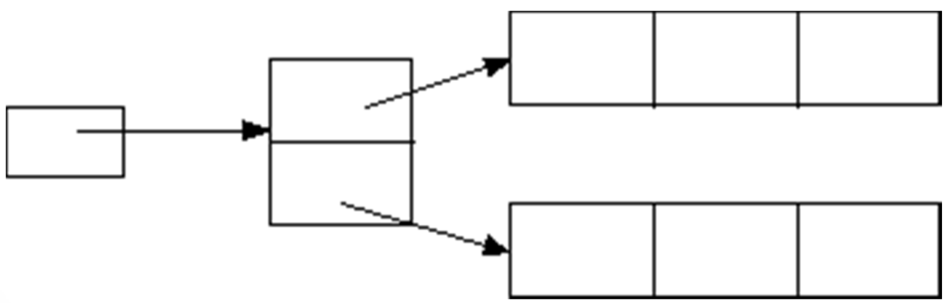
- General format
  Location (a[i,j,k]) = address of a [0,0] +
  ((i*m*n) + ((j * n) + k) * elem_size

4 x 5 x 3 Array



Multidimensional Arrays in Java

- Java implements multi-dimensional arrays as arrays of arrays



Compile-Time Descriptors

Array
Element type
Index type
Index lower bound
Index upper bound
Address

Single-dimensioned array

Multidimensioned array
Element type
Index type
Number of dimensions
Index range 1
⋮
Index range *n*
Address

Multi-dimensional array

Rectangular and Jagged Arrays

- A rectangular array is a multi-dimensional array
  - all rows have the same number of elements
  - all columns have the same number of elements
- A jagged matrix has rows with varying number of elements
  - Possible when multi-dimensioned arrays actually appear as arrays of arrays
- C, C++, C# and Java support jagged arrays
- Fortran, Ada, and C# support rectangular arrays

Pointer Arithmetic in C and C++

- `float stuff[100];`
- `float *p;`
- `p = stuff;`

- `*(p+5)` is equivalent to `stuff[5] and p[5]`
- `*(p+i)` is equivalent to `stuff[i]` and `p[i]`

Slices

- A *slice* is some substructure of an array; nothing more than a referencing mechanism

Slice Examples

- Fortran 95

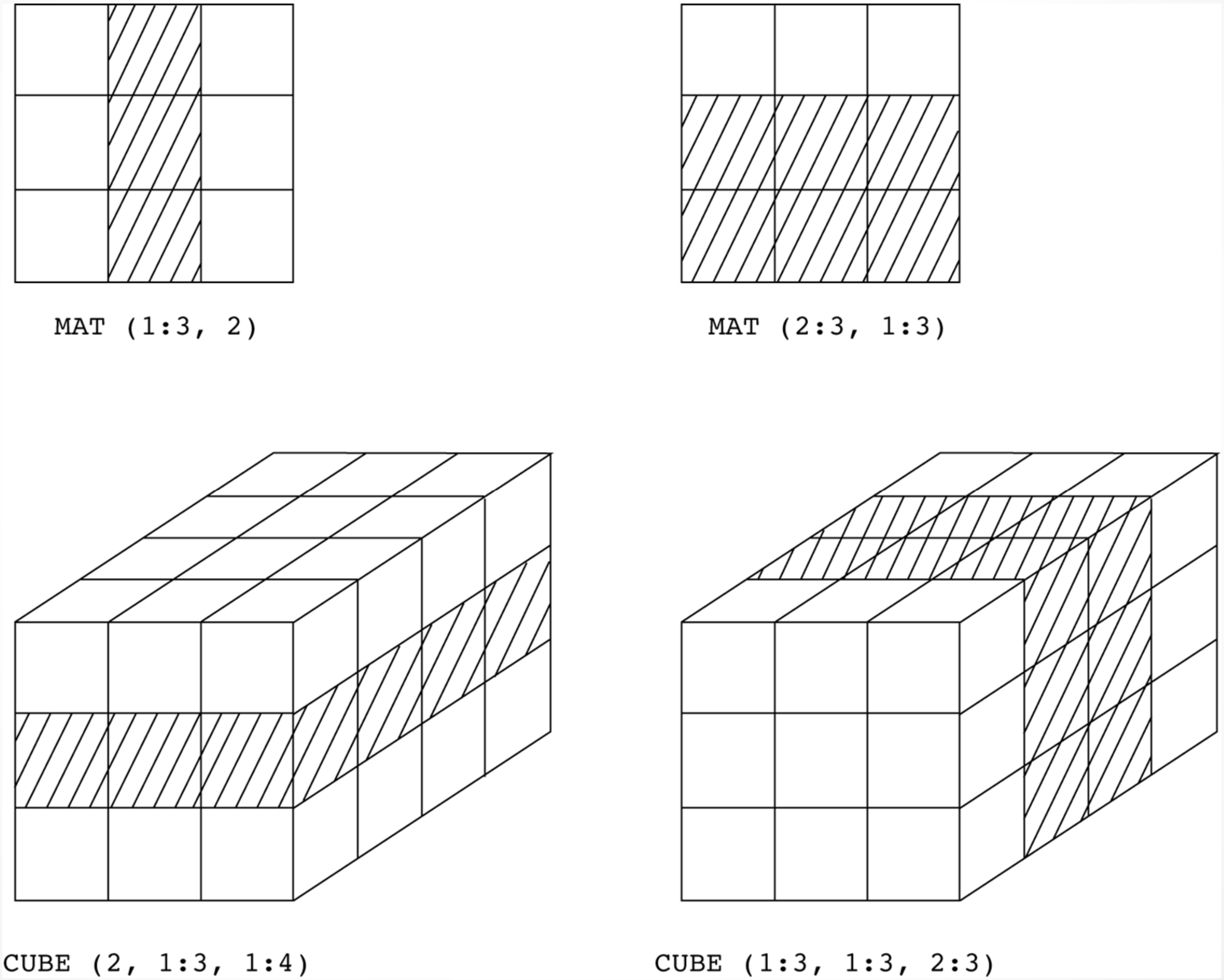
```
Integer, Dimension (10) :: Vector
Integer, Dimension (3, 3) :: Mat
Integer, Dimension (3, 3) :: Cube
```

`Vector (3:6)` is a four element array

- Ruby supports slices with the `slice` method

`list.slice(2, 2)` returns the third and fourth elements of `list`

Slices Examples in Fortran 95



Associative Arrays

- An *associative array* is an unordered collection of data elements that are indexed by an equal number of values called *keys*
  - User-defined keys must be stored
- Built-in type in Perl, Python, Ruby, and Lua
  - In Lua, they are supported by tables

Associative Arrays in Perl

- Names begin with `%`; literals are delimited by parentheses

```
%hi_temps = ("Mon" => 77, "Tue" =>
  79, “Wed” => 65, …);
```

- Subscripting is done using braces and keys

```
$hi_temps{"Wed"} = 83;
```

  - Elements can be removed with `delete`

```
  delete $hi_temps{"Tue"};
```

Other Languages

- Ruby has hashes
  - `ht = {key1=> value1, …}`
  - use `ht[key1]` to access
- Python has dictionary type
  - `ht = {key1 : value1, …}`
  - use `ht[key1]` to access
- In C++, Java provide library classes
- In C, need user-defined type