Arrays

- Ordered collections of elements, accessed by position
- Single or multi-dimensional
- Typically some uniformity to nature of storage (e.g. same types or references)
- May or may not support re-sizing of arrays

Syntax

- Array syntax often a highly recognizable aspect of language, particularly for multi-dimensional arrays
- Subscripting syntax usually needs to be distinct from that of function calls (lisp unusual in this regard)
- Declaration usually (not always) identifies size, and (for statically typed languages) stored data type
- Indexing typically 0-based or 1-based

Storage

- Storage might be on stack or in heap
- If on stack, size might be determined statically or at time of call
- Resizable arrays usually require storage in heap
- Cells generally stored sequentially, and of uniform size (e.g. same stored type, or cells contain references/pointers to actual storage for the cell data somewhere in heap)
- Uniform size allows compiler to compute cell offsets efficiently, especially for iteration through cells

Storage of multi-dim arrays

- To generate sequence of all cells in the array, compiler must select an ordering across the rows/columns
- Row-major order: store elements of row 0 first (in sequence) then elements of row 1, then row 2, etc
- Column-major order: store elements of column 0 first, then column 1, then column 2, etc
- Sparse arrays also a possibility: for very large arrays where only a small percent of cells actually used, only store those cells that contain data, each cell in a row (or column etc) contains some form of link to next used cell in that row (column etc)

Passing, copying, returning arrays

- Array assignment may do shallow or deep copy
- When passing arrays as parameters or using as return values, again may be either shallow copy or deep
- Deep copies significantly increase use of stack space and time required to perform call/return/copy
- Shallow copies can introduce possibilty of unexpected side effects

Initialization, assignment

- May or may not be initialized at time of creation
- May or may not generate warnings about use of uninitialized cells
- Assignment may be possible to single cells, entire row, entire array, etc depending on language
- Bounds testing (access to out-of-range cells) may or may not be automatically supported, or may be compiler option
- Does overflow/underflow of cell data affect adjacent cells?

Array resizing

- Some languages support implicit or explicit resizing of arrays (generally only if arrays stored in heap)
- Implicit resizing: if user accesses out-of-bounds element then array is automatically resized to make that “in-bounds”
- Explicit resizing: user must explicitly request/specify new size
- Mechanism for resizing might move existing content to new (big enough) space elsewhere, or may take linked-list approach to join together chunks of array space

Slices and subranges

- Language might support accessing subrange of array in single HLL instruction (e.g. copy elements 17 through 27 of one array to positions 2 through 12 of another array)
- Language might also support slices: specifying a subset of the array indices and taking ALL elements that match that position.

  E.g. for a 20x30 array, arr[][3] might specify all the elements in column 3, or arr[7][] might specify all the elements in row 7.

  Could be extended to multiple dimensions, e.g. for three dimensional array, arr[2][4][] might refer to all elements in row 2 and column 3