- Arrays
  - 1/2 dimension, all elements have same type (homogeneous), are ordered (start at a[0] to a[n-1])
    - 1 dim = vector
    - 2 dim = matrix (array of vectors)
    - >2 dim = tensor (arrays of arrays...)
  - a[-2] → makes sense w/ pointers but don't use this
- Declaring array: (type) name[count] = {initialize list}
  - don't need count if you initialize list & vice versa
- Matrix: (type) m[3][3] ⇒ elements m[0][0] to [2][2]
  - C stores memory row by row
- Memory = 1 dim, rows are sequential
- arrays are large, don't copy onto stack → relate to pointers
- name of array = pointer to element 0 of array (base address)

- sizeof operator tells variable's # of bytes
- sizeof(array) = size of array
- sizeof(pointer) = size of pointer
- a[i] == *(a+i)
- if matrix allocated at compile time, laid out contiguously
- if allocated dynamically, it's array of pointers to vectors
  ⇒ matrix = array of pointers

[Searching for biggest/smallest/an item Lecture 7]

- if array ordered, get min & max in constant time
- binary search fastest if ordered

- Strings = arrays of characters, ends in '\0'
  - char s[] = "word"
    char *s = "word" ← most common
    char s[] = {'w', 'o', 'r', 'd'}

- strcmp(): <0 means s<t
  >0 means s>t
  =0 means s==t
- strlen(): # of non-null characters
- strcpy(): copy str if not null (remember to tack '\0' at end
  ↳ might cause buffer-overflows

- strncmp(): checks to make sure we didn't go too few (depends on knowing how large array is)
- strncpy(): must account for null at the end

- Sorting → place into defined order
  - ex) lexicographical, natural order, total & partial ordering
- sorting adds info to our data (can make assertions about before, after, lesser, greater, etc)
- "enumerate all possibe orderings of n objects" ⇒ inefficient & too big "bogosort"
- selection sort
  - find smallest item by looking at all n of them
  - place in first slot
  - if there's anything in that slot, swap it

INSERTION SORT
- array of 1 element = sorted
- array of 2 elements ⇒ check before or after 1st element
- array of 3 elements ⇒ check possible positions

→ not efficient, goes up to vv large # of elements

- Bubble sort
  - if 1st element > 2nd element then swap them
  - if 2nd > 3rd element, swap them too, etc.
  - last element is largest → only look at n-1 pairs
  - no swaps = already sorted
    - to make code efficient, check if sorted already

- Merge Sort
  - split elements into disjoint pairs (a[0], a[1] & a[2], a[3])
    - $\frac{n}{2}$ # of pairs but still n # of elements
  - place elements of each pair in order & call it a run
  - then take pair of runs & merge them into runs of 4
  - can double 1 before we exceed n by $\log n$
  - sort finishes in time proportional to $n \log n$

(SORT TYPES)

$O(n^2)$ sorts	$O(n^{5/3})$	$O(n \log n)$
bubble	shell (good)	merge
insertion		heap
selection		quick (average)
quick (worst one)		

- for any choice of pivots, there are arrays

- Heaps
  - types: min/max, leftist, binomial, fibonacci, brodal, radix, ariel
  - max heap (put largest child & swap w/ parent)
    - single node = heap
    - heap if parent is heap & trees rooted at both children are heaps
    - parent's value (key) > either child's
    - no order among children

- radix sort runs proportional to # of digits * # of records
- better algorithm > faster computer

POINTERS (ptr)

- Pointer = variable, holds memory address
  - points to location of object in memory
- not all pointers contain an address (can be set to NULL) =0
  - NULL for macros can be: ((void *)0), 0, 0L
- memory stored in registers, accessed by address
  - each byte has unique address
  - byte = word of size 4
- pointers point to address they're assigned
- assign pointer the address of a variable using &
  - ↳ multiple pointers can point to same address

process = dereference or indirection (use * operator)

- useful for manipulating values of several variables w/ call by ref
- EX)
```
int foo = 13;
int *bar = &foo; // foo's address stored in pointer bar
```
- pointers have addresses ⇨ EX) `int **barz = &bar`
  `// bar's address in ptr barz`
- use ptr for dynamic data & large #s of it returns
- passing by value = duplicates passed values onto stack
- passing by reference = duplicates ptr onto stack
  - ↳ returns multiple values, passes large data quickly
- pointer arithmetic = same
  - ↳ EX new) array[i] == *(a+i) of ptr

(int) 1000 1001 1002 1003 1004 1005 1006 1007 1008

arr[0] arr[1] arr[2]

↳ 4 bytes per int → sizeof()

- adding/subtracting int w/ ptr ✓
- subtracting 2 ptr ✓
- adding 2 ptr X
- * or / ptrs X } don't do these

- arrays can be written using ptrs
  - declaring array allocates function on stack
  - global array in data area
  - dynamic array dec. to get ptr allocates it on heap

- str = ptr to array of chars
  ↳ can be indexed, passed by ref

- multi dim array → Ex) int a[3][3] = {{0, 1, 2},
  {3, 4, 5},
  {6, 7, 8}};
  - can be of any dim
  - Ex) int a[2][3][2] = {{{0,1}, {2,3}},
    {{4,5}, {6,7}}};

- function ptrs
  - points to executable code in memory (not data value)
  - deref func ptr yields referenced func ⇒ use ()