Slide 1

Administrivia

- As noted in e-mail yesterday, Homework 5 on the Web. Due next Wednesday. Be advised that your program, run on one of the classroom/lab machines, should produce exactly the counts given in the writeup; if it doesn't, something is wrong.
- Variance example from last time (including slides) updated to be consistent with what I believe to be the correct definition (which in class I got wrong).

Slide 2

Minute Essay From Last Lecture

- Many people mentioned finding at least one of the Homework 4 problems difficult. That might be a good result?

Slide 3

Arrays and Functions

- As noted previously, you can operate on individual elements of an array as if they were single variables (use them in expressions, assign to them, and pass them to functions); syntax is name of array followed by index in square brackets.
- You can also pass a whole array to a function; syntax on calling side is to just give its name (no index); on function side, follow name with brackets. *Note* that in this case the function actually has access to the array and can change its elements. (Is this an exception to the rule about "pass by value" with copying? Not really — what is being passed is a pointer — but it may appear so.)
- (Trivial example on "sample programs" page.)

Slide 4

Multi-Dimensional Arrays

- Single-dimensional arrays provide a way to represent something like singly-subscripted variables in math. What about variables with multiple subscripts? e.g., matrices? "multi-dimensional arrays"
- C has them (syntax in book), but they're somewhat awkward to work with . . .

Slide 5

Multi-Dimensional Arrays, Continued

- For old-style arrays (i.e., not VLAs), can't really write functions that work with different sizes.
- For VLAs, functions are easier but total size may be limited, and some very cautious programmers avoid VLAs because some compilers allegedly do not support them well.
- Dynamic allocation (making an array of arrays — more later) may be better but is tedious.
- User-defined macros that "fake" multiple dimensions in single-dimensional array also work okay but are tedious.

Slide 6

Sorting and Searching

- Traditional topics in CS1 courses. Arguably not of first importance to people more interested in using computers as tools, but still interesting . . . :
- Both are good examples of problems that can be solved in different ways.
- Both are good examples for introducing the idea of "order of magnitude" for algorithms.
- (But if you actually need to do one of these operations, look first for a library function!)

Slide 7

Sorting — The Problem and Some Solutions

- Problem: Given an array (or list) of elements for which there is a sensible "less than" operator, put them in order.
- Simple solutions include bubble sort, selection sort, insertion sort. Easy to program but not "fast" (more later).
- More-complex but "faster" solutions exist, and two of the best-known use recursion.

Slide 8

Searching — The Problem and Some Solutions

- Problem: Given an array (or list) and an element, search the array for the element.
- Simplest solution is sequential search. Easy to program and works for any array but not "fast".
- Slightly more-complex solution is binary search. "Faster" but requires array to be in order.

Slide 9

Order of Magnitude of Algorithms

- Conventional wisdom (among computer scientists) is to write programs in a way that humans can understand, and let the compiler turn them into something that will run fast.
- One exception is "order of magnitude" of algorithm, however.
- Key idea is to think about how execution time scales with the "problem size".
- Roughly analogous to order of magnitude of numbers — provide a way of grouping into classes in which all members of one class are sort of "the same" but members of different classes are not.
- Typically written using "big-O" notation (e.g., $O(N)$, $O(N^2)$, etc.). Formal definition possible, but informally, $O(f(N))$ means that execution time for problem size N scales as $f(N)$.

Slide 10

Order of Magnitude of Algorithms, Continued

- A key idea — for large enough problem sizes, algorithms with smaller orders of magnitude are faster, though this may not be true for small problem sizes.
- Another key idea — some orders of magnitude (e.g., $O(2^N)$) are sufficiently "big" that solving problems of any non-trivial size is simply not feasible, so "wait until computers get faster" is probably not a good strategy. "Hm!"?

Slide 11

Minute Essay

- Can you think of problems you might want to solve that would require multi-dimensonal arrays?