Slide 1

Administrivia

- Midterms graded. Most scores very good (!).
- Sample solutions for all homeworks posted.

  If you've turned something in, not a bad idea to have a look even if you got full credit — some full-credit solutions seemed to me to be more complicated than they needed to be.

  If you haven't turned in all assignments, I'm still willing to accept late assignments for *some* credit if you haven't look at the solution.

Slide 2

Why Arrays?

- Suppose you wanted to write a program to count how many times each letter occurs in the program's input. What would you do? Is there an obvious way to solve this with what we've discussed so far?

Slide 3

Why Arrays?, Continued

- You could have a variable for how many A's, one for how many B's, etc., and a huge `switch` construct. But how ugly . . .
- What seems to be needed is something similar to subscripted variables in math — an *array*.
- Other uses abound — e.g., if working with large amounts of input, sometimes you can process elements as you read them (e.g., our program to compute an integer sum), but sometimes it's necessary or at least convenient to have them all in memory at once.

Slide 4

Arrays

- Previously we've talked about how to reserve space for a single number/character and give it a name.
- Arrays extend that by allowing you to reserve space for many elements of the same type (`int`, `float`, etc.) and give a common name to all. You can then reference an individual element via its *index* (similar to subscripts in math).

Slide 5

Arrays in C

- Declaring an array — give its type, name, and how many elements. Examples:

```
int nums[10];
double stuff[N];
```

(The second example assumes N is declared and given a value previously. In old C, it had to be a constant. In newer C, it can be a variable.)

- Referencing an array element — give the array name and an index (ranging from 0 to array size minus 1). Index can be a constant or a variable. Then use as you would any other variable. Examples:

```
nums[0] = 20;
printf("%d\n", nums[0]);
```

(Notice that the second example passes an array element to a function. AOK!)

Slide 6

Example — Variance

- As an example of a calculation where it's necessary (or at least convenient) to have all input values in memory at once, consider computing *variance* of inputs, where variance of $a_0 \cdots a_{n-1}$ is defined as the average of $(a_i - avg)^2$ (avg is the average of the a_i's).
- Unless we can be clever somehow, we can't start computing this sum until we have the average, and computing that requires us to read all the inputs, but then we need to read them again, which might not be possible, so store them . . .

Slide 7

Arrays in C, Continued

- We said if you declare an array to be of size n you can reference elements with indices 0 through $n - 1$. What happens if you reference element -1? n? $2n$?
- Well, the compiler won't complain. At runtime, the computer will happily compute a memory address based on the starting point of the array and the index. If the index is "in range", all is well. If it's not (i.e., it's "out of bounds) . . .

Slide 8

Arrays in C, Continued

- (What happens if you try to access an array with an index that's out of bounds?)
- "Results are unpredictable." Maybe it's outside the memory your program can access, in which case you probably get the infamous "Segmentation fault" error message.

  Almost worse is if it's not — then what's at the computed memory address might be some other variable in your program, which will then be accessed/changed. (This is the essence of the *buffer overflows* you may hear mentioned in connection with security problems.)
- What to do? *Be careful.* (Probably worth noting here that many more-recent languages, for example Java and Python, protect you from such errors by "throwing an exception", which by default crashes your program, but with information about what went wrong.)

Slide 9

Minute Essay

- How did the midterm compare to your expectations (topics, level of difficulty, . . .).