Slide 1

Administrivia

- (None?)

Slide 2

Minute Essay From 10/02

- Question for was about uses for repetition. Some answers:
- Solving PDEs; numerical integration.
- Applying a function to each pixel in a photo.
- Doing a calculation at each point in a grid.
- “Guess and check” methods.
- (Others — all interesting!)

Minute Essay From Last Lecture

- About uses for file, several interesting answers along the lines of what I had in mind (programs that work with files), but several people also mentioned using files as a way of breaking up a long program into pieces. (!)

  More in next slides . . .
- About programs for making plots, most people mentioned Excel and/or Matlab, There were also mentions of R, EES, and Minitab.

Uses for Files

- Files are useful if you want to deal with input data from some source other than "the keyboard", or from more than one source.
- Files are also useful if you want to produce output data that can be saved for plotting or other analysis (perhaps as input to another program).
- If uses for files aren't occurring to you, consider that "real" programs such as the ones you likely use frequently (word processors, spreadsheets, etc.) all make use of files. Files are also likely involved if an application "remembers" what you did in previous sessions.
- (Spreadsheets are to me kind of an interesting(?) case in that they bundle together program-like logic, data, and — other things. To me this seems wrong-headed, but they're very popular!)

Slide 5

Files and Program Structure

- All the programs we've written have been single files. Fine for small programs but seems like it might be unwieldy for large ones, no?
- So usually code for large programs is split into multiple files, which can be separately compiled and then "linked" together to form an executable. This also allows reuse of the same function(s) in multiple programs. We'll do examples of both, later.
- "Library" functions build on this idea: Someone writes code for them, typically splitting it into a `.h` file with just declarations of functions (e.g., `stdio.h`) and a `.c` file with code for functions. The code is compiled and turned into something that can be linked into user programs. What's distributed might be just those `.h` files and the compiled code.

Slide 6

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 7

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 8

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).

Arrays in C

Slide 9

- 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!)

Example — Variance

Slide 10

- 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 11

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. (How would it know to?) And 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 12

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, Scala, and Python, protect you from such errors by "throwing an exception", which by default crashes your program, but with information about what went wrong.)

Arrays — Summary

- Arrays are very useful and extend the range of what we can (easily) do.
- However, in C they open up new sources of potential error, and because they're fixed in size (when you create them), I say avoid their use when you easily can.

Slide 13

Minute Essay

- Have you seen arrays before (maybe in Matlab)?
- Either way — can you think of uses?

Slide 14