

gcc Tip

- I say always always compile with -Wall six extra keystrokes, or not even that if you remember about the up arrow in bash (shell).
- And then *do something* about warnings almost all indicate a potential problem. (If you can't figure out what, ask! if nothing else asking me by e-mail works though isn't as immediate.)

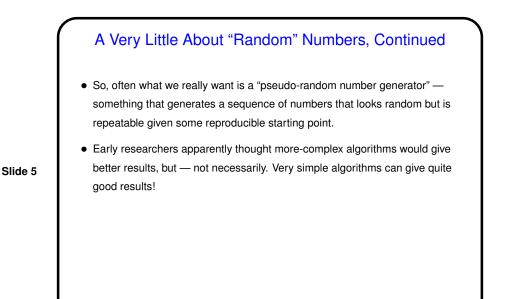
Slide 3

• (The first thing I usually do when students ask why their code doesn't work is to ask them to recompile with this option. It's surprising how often it warns about something that turns out to be the source of the problem!)

A Very Little About "Random" Numbers

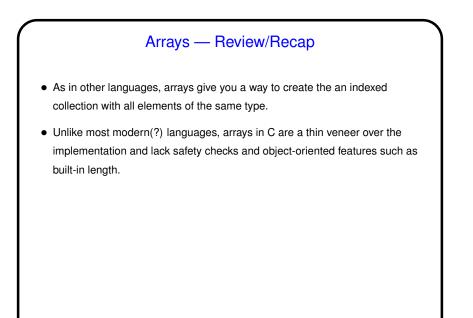
- Homework 4 asks you to work with the library functions srand() and rand(). A few words about what they do...
- First, what we mean by "random" is (I think!) an interesting question with no obvious answer.

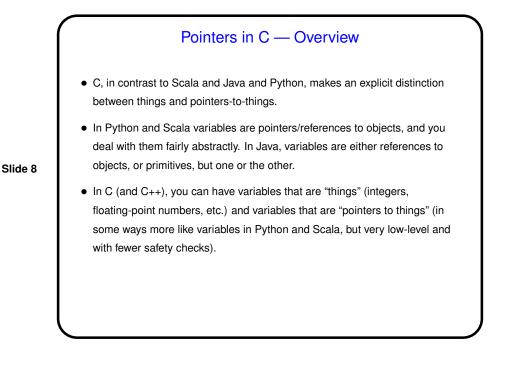
- What's often wanted is something that can't be predicted, and it's not clear we can get that with a system that's deterministic. Further, even if we could, we might not want that, since we often want to be able to repeat a test.
- (Canonical reference discussion in volume 2 of Knuth's *The Art of Computer Programming*. Very mathematical. Other references may be easier.)

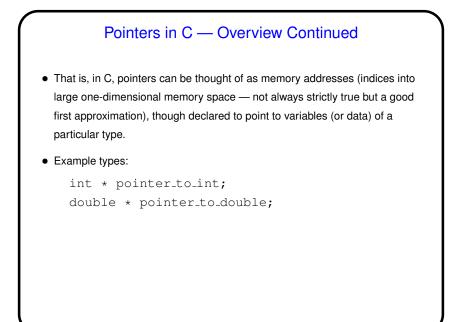


A Very Little About "Random" Numbers, Continued

- Lots of uses for "random" sequences (e.g., so-called "Monte Carlo" methods for simulating things), so many libraries include function(s) to produce them.
- Typical library provides some way to set the starting point (the "seed") and then a function that when called repeatedly produces the sequence srand() and rand() in standard C. Mostly these produce a large range of possible values. (Why is this good?)
- Some libraries also provide functions to map the full range to a smaller one (e.g., to simulate rolling a die). C doesn't, but there are some semi-obvious approaches. The problem on Homework 4 asks you to do a simple comparison of two of them.







```
Pointers in C — Operators
& gets a pointer to something in memory. So for example you could write int x; int * x_ptr = &x;
* "dereferences" a pointer. So for example you could change x above by writing

*x_ptr = 10;

(What do you think happens if x_ptr hasn't been initialized?)
You can also perform arithmetic on pointers (e.g., ++x_ptr) — something not allowed in languages more concerned with safety. Potentially risky but sometimes useful.
```





 In C, all function parameters are passed "by value" — which means that the value provided by the caller is copied to a local storage area in the called function. The called function can change its copy, but changes aren't passed back to the caller.

Slide 11

- An apparent exception is arrays no copying is done, and if you pass an array to a function the function can change its contents (as you would want to do in, say, a sort function). Why "apparent exception"? because really what's being passed to the function is not the array but a pointer! so the copying produces a second pointer to the same actual data.
- This is at least simple and consistent, but has annoying limitations



 A significant potential limitation on functions is that a function can only return a single value. Pointers provide a way to get around this restriction: By passing a pointer to something, rather than the thing itself, we can in effect have a function return multiple things.

- To make this work, typically you declare the function's parameters as pointers, and pass addresses of variables rather than variables.
- (The "sort of" of the title means that this isn't true pass by reference, as it exists in some other languages such as C++, but it can be used to more or less get the same effect.)
- (Example?)

