

Slide 1

### Administrivia

- Reminder: Homework 4 due next week.
- Sample solution for Homework 3 posted. (So if you weren't happy with your answer — “couldn't this be simpler?” — compare with mine?)

Slide 2

### Minute Essay From Last Lecture

- About seconds per hour, etc., in Homework 2, no clear majority: Some used a calculator (or the equivalent — one mention of Python!), other a search engine.
- About Homework 3, several mentions of difficulty checking for non-integer input. Before searching the Web, consider reviewing examples on “sample programs” on course Web site — I mean for them to be useful in doing the assignments.
- Also several comments about the first problem — yes, it's just a specialized sort, and can be done that way (see one of my sample solutions), or with conditionals.

### Minute Essay From Last Lecture, Continued

- Other interesting comments:
  - “15 minutes on the logic, 2 hours debugging” (syntax). Sounds about right for a new-to-you language?
  - “Hurt my soul” not to be able to just use bubble sort on the first problem. :-)

Slide 3

### gcc Tip

- I say always always compile with `-Wall` — six extra keystrokes, and 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.)
- (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 — or not! — how often it warns about something that turns out to be the source of the problem!)
- If you want to be really thorough, add `-pedantic` (flags nonstandard usage — such as nested functions).

Slide 4

Slide 5

### 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.)
- (Aside: In the process of getting TAOCP published, Knuth got curious about typesetting via computer and started a side project that eventually produced  $\text{\TeX}$ . Classic example of a side project that turned into much more!)

Slide 6

### A Very Little About “Random” Numbers, Continued

- So, often what we really want is a “pseudo-random number generator” — something that generates a sequence of numbers that looks random but is repeatable given some reproducible starting point.
- Early researchers apparently thought more-complex algorithms would give better results, but — not necessarily. Very simple algorithms can give quite good results!

Slide 7

### 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.

Slide 8

### Arrays — Review/Recap

- As in other languages, arrays give you a way to create the an indexed collection with all elements of the same type.
- Unlike most modern(?) languages, arrays in C are a thin veneer over the implementation and lack safety checks and object-oriented features such as built-in length.

### Pointers in C — Overview

Slide 9

- C, in contrast to Scala and Java and Python, makes an explicit distinction between things and pointers-to-things.
- In Python and Scala variables are pointers/references to objects, and you deal with them fairly abstractly. In Java, variables are either references to objects, or primitives, but one or the other.
- In C (and C++), you can have variables that are “things” (integers, floating-point numbers, etc.) and variables that are “pointers to things” (in some ways more like variables in Python and Scala, but very low-level and with fewer safety checks).

### Pointers in C — Overview Continued

Slide 10

- That is, in C, pointers can be thought of as memory addresses (indices into large one-dimensional memory space — not always strictly true but a good first approximation), though declared to point to variables (or data) of a particular type.

- Example types:

```
int * pointer_to_int;  
double * pointer_to_double;
```

## 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.

Slide 11

## Parameter Passing in C — Review

- 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.
- 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 . . .

Slide 12

### Pass By Reference (Sort Of)

Slide 13

- 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?)

### Pointers Versus Arrays

Slide 14

- In almost all contexts arrays and pointers are interchangeable.
- In particular, if you declare the type of a function parameter to be a pointer, you can pass it an array, and vice versa.

## Strings in C — Overview

Slide 15

- C has a data type `char`, used for much the same purposes as characters in other language, *but* with a smaller minimum range (enough to represent 7-bit ASCII but not Unicode).
- C “strings” are null-terminated arrays of characters and can be worked with as arrays or using pointers. There are standard library functions for doing (some) things with characters and strings.
- (Examples next time.)

## Minute Essay

Slide 16

- Do you remember to compile with `-Wall`? and if so, do you try to fix anything being warned about? I ask because I got a fair amount of code for Homework 3 that gave warnings ...
- Any questions — about pointers, strings, anything else?