Administrivia

- Reminder: Homework 3 due today.

- Homework 4 will be on the Web tomorrow; due next week.

Slide 1

Arrays and Pointers — A Correction/Clarification

- I've said that C, unlike higher-level languages, doesn't try to detect out-of-bounds array accesses or attempts to dereference null or invalid pointers. I've also said what often happens with arrays. But in fact . . .

- The C standard is fairly specific about what things programmers should not do (at least if they're trying to write portable code). What it doesn't say is what should happen if you break the rules! the behavior I've described is typical but not guaranteed. Caveat programmer!

Slide 2

I/O in C — Some Very Basic Functions

- `getchar` gets one character and returns it as an `int`. The special value `EOF` indicates end of input. ("End of input"? control-D from terminal, more in next sidebar.)

- `putchar` writes out one character.

Slide 3

- We could use this to write a very simple program that simply copies its input to its output . . .

I/O in C, Continued

- You already know about a function to write output to "standard output", `printf`. Many options, allowing a lot of control over what's printed.

- How about input? Counterpart of `printf` is `scanf`, as described last time. Simple to use, though error detection is somewhat crude, and reading text strings can be hazardous.

Slide 4

- One way to work with files is I/O redirection. Is there something more general? Yes . . . .

Sidebar — Input/Output Redirection in UNIX/Linux

- In programming classes I talk about "reading from standard input" rather than "reading from the keyboard", and "writing to standard output" (or "writing to standard error") rather than "writing to the screen".

- What's the difference?

Slide 5

I/O Redirection, Continued

- `stdin` (standard input) can come from keyboard, file, or from another program or shell script.

- `stdout` and `stderr` (standard output, error) can go to terminal or file (overwrite or append), separately or together.

Slide 6

Slide 7

I/O Redirection, Continued

- For example — to redirect output of `ls` to `ls.out`, type

  `ls >ls.out`

  (Overwrites `ls.out`. To append, replace $>$ with $>>$.)

  To also redirect any error messages, append `2>&1`.

- To redirect input, use $<$`infile`.

Slide 8

Streams

- C's notion of file I/O is based on the notion of a *stream* — a sequence of characters/bytes. Streams can be *text* (characters arranged into lines separated by something platform-dependent) or *binary* (any kind of bytes). UNIX/Linux doesn't make a distinction, but some other operating systems do.

- An input stream is a sequence of characters/bytes coming into your program (think of characters being typed at the console).

- An output stream is a sequence of characters/bytes produced by your program (think of characters being printed to the screen, including special characters such as the one for going to the next line).

Streams in C

- In C, streams are represented by the type FILE * — i.e., a pointer to a
  FILE, which is something defined in stdio.h.

- A few streams are predefined — stdin for standard input, stdout for
  standard output, stderr) for standard error (also output, but distinct from
  stdout so you can separate normal output from error messages if you
  want to).

- To create other streams ...

Slide 9

Creating Streams in C

- To create a stream connected with a file — fopen.

- Parameters, from its man page:
  - First parameter is the name of the file, as a C string.
  - Second parameter is how we want to access the file – read or write,
    overwrite or append — plus a b for binary files, also a string.
  - Return value is a FILE * — a somewhat mysterious thing, but one we
    can pass to other functions. If NULL, the open did not succeed. (Can you
    think of reasons this might happen?)

Slide 10

Working With Streams in C

Slide 11

- To read from an input stream — `fscanf`, almost identical to `scanf`. To write to an output stream — `fprintf`, almost identical to `printf`. `fgetc` and `fputc` may also be useful.

- When done with a stream, `fclose` to tidy up. (Particularly important for output files, which otherwise may not be completely written out.)

Reading Text Strings

Slide 12

- Getting text-string input is surprisingly tricky. `scanf` (or `fscanf`) seems like an obvious choice, but:
    - it can't read a string that includes blanks, and
    - it has no nice way to limit the number of characters read to the size of the array being read into.
    .

- Getting a whole line is probably better. `gets()` is an obvious/simple choice for reading from standard input, but it also has no way to limit how much is read. `fgets()` is better. (Look at its `man` page.)

  (Also notice `puts()` — simple way to write out a text string.)

One More Way to Get Input — Command-Line Arguments

Slide 13

- Now that we know about arrays, pointers, and text strings, we can talk about command-line arguments. What are they? text that comes after the name of the program on the command line (e.g., when you write `gcc -Wall myprogram.c`, there are are two command-line arguments), possibly modified by the shell (e.g., for filename wildcards).

- Most programming languages provide a way to access this text, often via some sort of argument to the main function/method. In C, the arguments are passed to `main` as an array of text strings. So if you define `main` as

    `int main(int argc, char * argv[]) { .... }`

    `argc` is the number of arguments, plus one, and `argv` is an array of strings containing the arguments.

    (Examples as time permits.)

Minute Essay

Slide 14

- Can you think of situations in which I/O redirection would be useful?

Minute Essay Answer

Slide 15

- There are several, possibly many. One is for program testing — you put the input in a file, run the program with input redirected to come from that file, and capture the output. If you later change the program, you can easily determine whether it still produces the same results, by capturing output again and comparing (e.g., with `diff`) to the old output.