Slide 1

Administrivia

- Homework 6 deadline extended to Friday. Possibly one more easier/shorter homework to be assigned Friday.

Slide 2

Pointers in C — Review/Recap

- Many programming languages provide something like pointers. Unlike some more-recent languages, C allows you to have both pointer variables and non-pointer variables.
- To a first approximation, C pointers are just memory addresses — i.e., numbers — but they are declared to point to variables (or data) of a particular type. Example:

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

Slide 3

Pointers in C — Operators and Arithmetic

- & gets a pointer to something in memory.
- `*` “dereferences” a pointer.
- Can display value of pointer using `printf` with `%p` and “casting” pointer to type `void*`. Sometimes interesting in exploring how variables are laid out in memory (implementation-dependent).
- C also permits doing some arithmetic operations on pointers (addition and subtraction). Adding n to a pointer that points to *type* advances it n times the size of *type*. Can also compare pointers, but probably best not to rely on anything but `==` and `!=`.

Slide 4

Converting Text Strings to Numeric Types

- You know about `scanf` (and `fscanf` for converting text input to numeric types. But what if you have a text string (e.g., a command-line argument)?
- Functions `strtol` and `strtod` can help. `atoi` and `atof` can also be used but do not provide any kind of error checking.

  Usage example (to convert the first command-line argument, if the second parameter to main is `argv`):

```
char *endptr;int n = strtol(argv[1],
&endptr, 10);if (*endptr != '\0')  /* error
*/
```

Dynamic Memory and C

Slide 5

- With the old C standard, you had to decide when you compiled the program how big to make things, particularly arrays — a significant limitation.
- Variable-length arrays help with that, but don't solve all related problems:

  In most implementations, space is obtained for them on "the stack", an area of memory that's limited in size.

  You can return a pointer from a function, *but* not to one of the function's local variables (because these local variables cease to exist when you return from the function).

Dynamic Memory and C

Slide 6

- "Dynamic allocation" of memory gets around these limitations — allows us to request memory of whatever size we want (well, up to limitations on total memory the program can use) and have it stick around until we give it back to the system.

  (How this helps — most implementations differentiate between two areas of memory, a "stack" used for local variables, and a "heap" used for dynamic memory allocation. Usually the former is more limited in size.)
- Dynamic memory allocation also needed to build "linked" data structures (next time, briefly?).

Slide 7

Dynamic Memory and C, Continued

- To request memory, use `malloc`.
- To return it to the system, use `free`. (For short simple programs you can probably get away with skipping `free` since the operating system will probably clean up after you, but for longer and more complicated programs, you should clean up when you can, or eventually you may run out of memory.)

Slide 8

Dynamic Memory and C, Continued

- Examples:

  ```
  int * nums = malloc(sizeof(int) * 100);
  char * some_text = malloc(sizeof(char) *
  20);
  ```

  or better:

  ```
  int * nums = malloc(sizeof(*nums) * 100);
  char * some_text = malloc(sizeof(some_text)
  * 20);
  ```

  and then

  ```
  free(nums); free(some_text);
  ```

- Book recommends “casting” value returned by `malloc`. Other references

Slide 9

recommend the opposite! But you should check the value — if `NULL`, system was not able to get that much memory.

Slide 10

Multidimensional Arrays Revisited

- Multidimensional arrays are easy to declare:

```
int matrix[100][200];
```

- The messy part comes when you try to pass one of these to a function, though as with 1D arrays, VLAs do help. (Without them, there's really no way to specify *at runtime* all dimensions. The old-C way is to fix and specify all but the first dimension — e.g., for a 2D array, fix the number of columns.)
- Another way is to represent them as "arrays of arrays" — i.e., arrays of pointers. Could do this as what textbook calls "ragged arrays" or by building list of pointers into one big 1D array.

Minute Essay

- None — quiz.