Slide 1

Administrivia

- Reminder: Homework 7 was due today; deadline extended to Monday. Please do remember that on the second problem I want more than just your code.
- Next homework to be assigned Monday.
- Quiz 5 moved to next Friday. Topics TBA Monday, but will likely be files or strings or both.

Slide 2

Strings in C — Recap/Review

- Strings in C are arrays of `char`, delimited by special character `'\0'`.
- Print with `printf` and `%s` or `puts`; read with `fgets` or (carefully!) `scanf`.
- Library functions for comparing, copying, etc.
- (Finish palindrome example.)

Slide 3

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) and want to extract from it a command-line argument? You could use `sscanf`, or . . .
- 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;
long n = strtol(argv[1], &endptr, 10);
if (*endptr != '\0')  /* error */
```

- (Example — program to convert command-line arguments to integers and sum.)

Slide 4

Dynamic Memory and C

- 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).

Slide 5

Dynamic Memory and C

- "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 "ragged" arrays (arrays in which rows are of different sizes) and "linked" data structures (later).

Slide 6

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 7

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 recommend the opposite! But you should check the value — if `NULL`, system

Slide 8

was not able to get that much memory.

- Example — program to generate N "random" numbers and sort them.

Slide 9

Minute Essay

- Anything noteworthy about Homework 7?