Slide 1

Administrivia

- Reminder: Homework 7 due today.
- Quiz 4 scores not great overall, though some were very good.

Slide 2

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 3

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 4

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 5

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 6

was not able to get that much memory.

- Example — program to generate and sort "random" numbers, revised.

Slide 7

Multi-Dimensional Arrays in C, Revisited

- Allocating multi-dimensional arrays in C can be ugly — either allocate with fixed size or use VLAs (probably not great for big arrays).
- Now that we know about dynamic memory allocation and more about pointers, can do better. Looking only at 2D arrays for now, two approaches. Both involve representing array as array of arrays/pointers:
- One way is to first allocate array of pointers and then fill it with pointers to dynamically-allocated 1D arrays. (Example code.)
- Another way is to first allocate array of pointers and 1D array big enough to hold whole array, and then fill array of pointers with pointers into this big array. (Example code.)

Slide 8

Minute Essay

- Anything noteworthy about Homework 7?
- If you didn't do well on Quiz 4, what do you think went wrong? To me the way to approach such a problem involves "tracing through" code — here, writing down what values are put in the array by the first loop and then figuring out how they're used in the second loop.
- How are you finding the workload for this class? about right for a 3-credit-hour course? light? heavy?