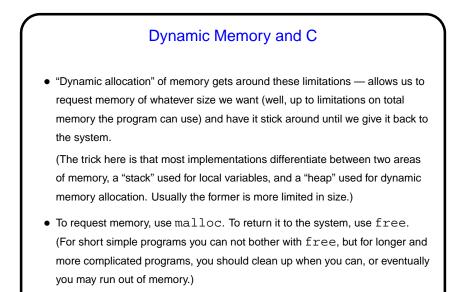


Slide 1

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).



Dynamic Memory and C, Continued
Examples:

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

Book recommends "casting" value returned by malloc. Other references recommend the opposite! But you should check the value — if NULL, system was not able to get that much memory.
Example — program to generate N numbers and sort them.

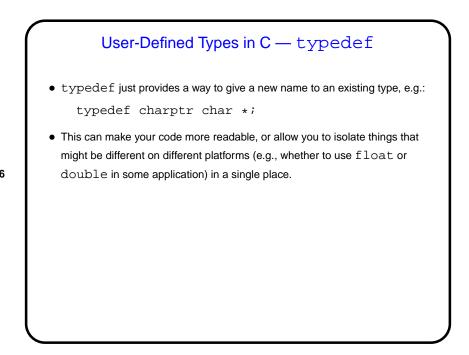
Slide 3

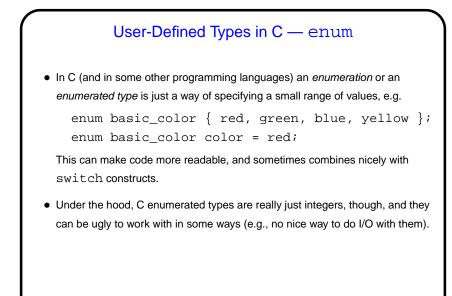


• So far we've only talked about representing very simple types — numbers, characters, text strings, arrays, and pointers. You might ask whether there are ways to represent more complex objects (e.g., a "money" object to represent dollars and cents — useful since floating-point is inexact for decimal fractions).

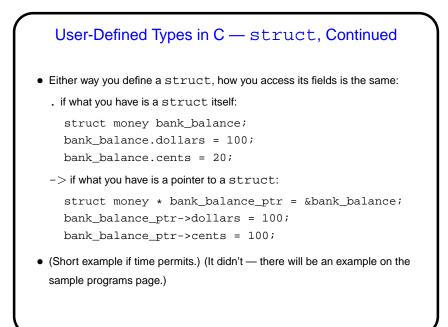
Slide 5

 Most/many programming languages (C included) do let you do that, in various ways ...





User-Defined Types in C — struct • More complex (interesting?) types can be defined with struct, which lets you define a new type as a collection of other types. • One way to define uses typedef: typedef struct { int dollars; Slide 8 int cents; } money; money bank_balance; • Another way doesn't: struct money { int dollars; int cents; }; struct money bank_balance;



Slide 9

