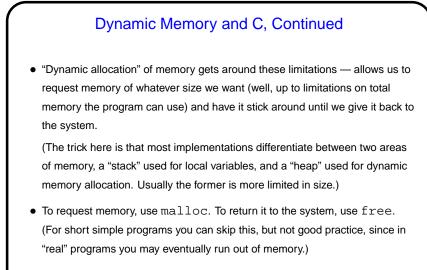


Dynamic Memory and C
With the C89 standard, you had to decide when you compiled the program how big to make things, particularly arrays — a significant limitation.
Variable-length arrays in C99 standard help with that, but don't solve all related problems:

In many 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



• Python and Scala hide most of this from you — allocating space for objects is automatic/hidden, and space is reclaimed by automatic garbage collection.

	Dynamic Memory and C, Continued
• Sim	nple examples:
	<pre>int * nums = malloc(sizeof(int) * 100);</pre>
	char * some_text = malloc(sizeof(char) *
	20);
	<pre>free(nums);</pre>
tho	ugh it's better style/practice to write
	<pre>int * nums = malloc(sizeof(*nums) * 100);</pre>
	char * some_text = malloc(sizeof(*some_text)
	* 20);
	<pre>free(nums);</pre>
<ul> <li>Sor</li> </ul>	ne books/resources recommend "casting" value returned by malloc.
Oth	er references recommend the opposite! But you should check the value —
if N	ULL, system was not able to get that much memory.



Slide 5

## Function Pointers You know from more-abstract languages that there are situations in which it's useful to have method parameters that are essentially code. Some languages make that easy (functions are "first-class objects") and others don't, but almost all of them provide some way to do it, since it's so useful — e.g., providing a "less-than" function for a generic sort. In C, you do this by explicitly passing a pointer to the function.

