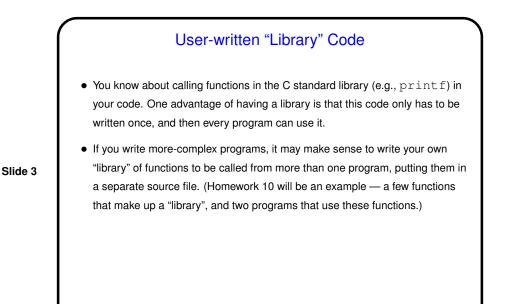


Slide 1

Minute Essay From Last Lecture

- At least a few people found Homework 8 interesting or fun. More than one said it turned out to be not as intimidating as they thought. Good!
- About the quiz question, no responses really stood out, except that a few people I think are still having trouble figuring out, for "write some code" questions, what's being asked for a complete program, or a function that returns a value, or a function that prints something, or what. I *think* if you read such questions carefully it should be clear, but maybe pay more conscious attention to what your code is supposed to do in the way of input and output?



User-written "Library" Code, Continued
How to "package" these library functions — at least two possibilities.
One is to put them in a .h file and use #include to include it in every program that uses the functions.
Another way (which is pretty much how the standard library functions are packaged) is to have a .h file containing declarations and a companion .c file with definitions. You then #include the .h file in programs that use the functions, and use "separate compilation" ...



- For large programs it's often better to split up code into more than one source file for readability if nothing else.
- How then to make the executable? A good way is to compile each .c file separately (with gcc -c) and then use gcc to "link" the resulting .o ("object code") files to produce the executable. (Note too that while gcc names the executable a .out by default, it will call it something else if you say to.)
- Sounds complicated? well, not as simple as compiling a single . c file, but ...

A Little About make

- Motivation: Most programming languages allow you to compile programs in pieces ("separate compilation"). This makes sense when working on a large program: When you change something, just recompile parts that are affected.
- Idea behind make have computer figure out what needs to be recompiled and issue right commands to recompile it.
- (Caveat: make is a UNIXworld thing. I feel sure there's something analogous for people developing software under Windows but am not sure what it is!)

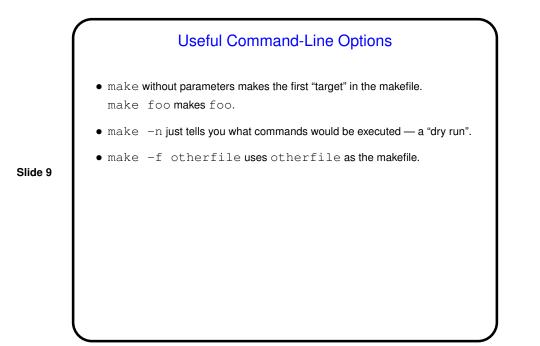


Makefiles
First step in using make is to set up "makefile" with "rules" describing how files that make up your program (source, object, executable, etc.) depend on each other and how to update the ones that are generated from others. Normally call this file Makefile or makefile.
(Example: Makefile.v1 in "makefile example" on sample programs page — rename to Makefile to easily use.)
When you type make, make figures out (based on files' timestamps) which files need to be recreated and how to recreate them.

Slide 7

Slide 8

Defining Rules
Define dependencies for a rule by giving, for each "target", list of files it depends on.
Also give the list of commands to be used to recreate target. *NOTE!*: Lines containing commands must start with a tab character. Alleged paraphrase from an article by Brian Kernighan on the origins of UNIX:
The tab in makefile was one of my worst decisions, but I just wanted to do something quickly. By the time I wanted to change it, twelve (12) people were already using it, and I didn't want to disrupt so many people.



Variables in Makefiles
You can also define variables; define with, e.g., CFLAGS = -Wall and reference as \$ (CFLAGS).
One good use is options to be used for all compiles.
Another good use is to specify lists of files.

