CSE 374
Programming Concepts & Tools

Brandon Myers

Winter 2015

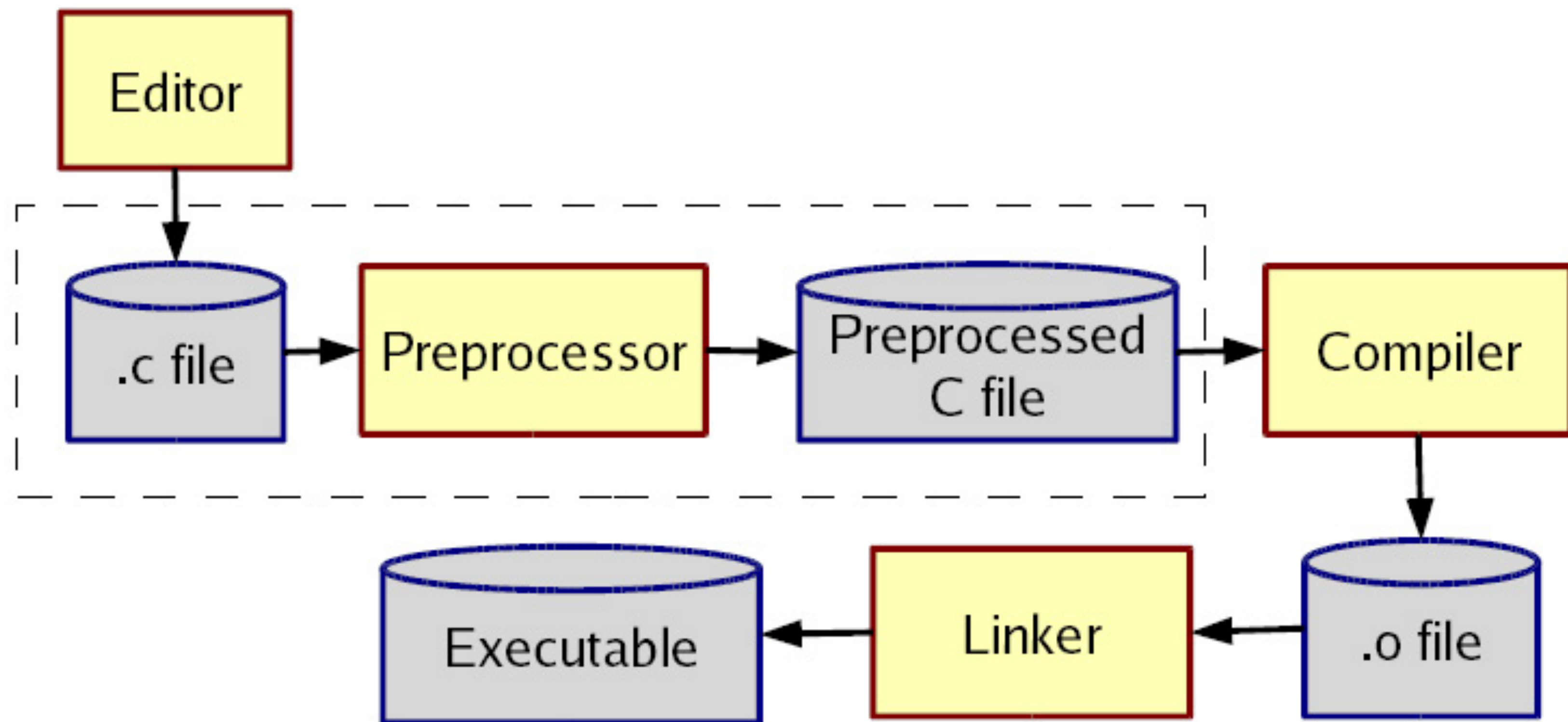
Lecture 17 - Build process, make

(Thanks to Hal Perkins)

C preprocessor summary

- A few easy to abuse features and a bunch of conventions (for overcoming C's limitations).
  - #include (the way you say what other definitions you need; cycles are fine with "the trick")
  - #define (parameterized macros have a few justifiable uses; token-based text replacement)
  - #if... (for showing the compiler less code)

The compilation picture



Where we are

- We are C programmers! Onto tools...
- Today: basics of make
  - in particular, the concepts

Besides the slides and online Unix docs, the Stanford CSLib notes on Unix Programming Tools has a nice overview of make and other tools:

http://cslibrary.stanford.edu/107/UnixProgrammingTools.pdf

Onto tools

- The language implementation (preprocessor, compiler, linker, standard-library) is hardly the only useful thing for developing software
- The rest of the course:
  - Tools (recompilation managers, version control, profilers; we've already seen a debugger)
  - Linking
  - A taste of C++

make

- make is a classic program for controlling what gets (re)compiled and how. Many other such programs exist (e.g., ant, maven, “projects” in IDEs, ...)
- make has tons of fancy features, but only two basic ideas:
  1. Scripts for executing commands
  2. Dependencies for avoiding unnecessary work
- we will focus on the concepts…

Building software

Programmers spend a lot of time "building" (creating programs from source code)

- Programs they write
- Programs other people write

Programmers automate repetitive tasks. Trivial example:

```
gcc -Wall -g -o widget foo.c bar.c baz.c
```

If you:

- Retype this every time: "shame, shame"
- Use up-arrow or history: "shame" (retype after logout)
- Have an alias or bash script: "good-thinkin"
- Have a Makefile: you're ahead of us

“Real” build process

- On larger projects, you can’t or don’t want to have one big (set of) command(s) that redoes everything every time you change anything
  1. If gcc didn’t combine steps behind your back, you’d need to preprocess and compile each file, then run the linker
  2. If another program (e.g., sed) created some C files, you would need an “earlier” step
  3. If you have other outputs for the same source files (e.g., javadoc), it’s unpleasant to type the source file names multiple times
  4. If you want to distribute source code to be built by other users, you don’t want to explain the build logic to them
  5. If you have 10^5 to 10^7 lines of source code, you don’t want to recompile them all every time you change something
- A simple script handles 1–4 (use a variable for filenames for 3), but 5 is trickier

“Real” build process

- On larger projects, you can’t or don’t want to have one big (set of) command(s) that redoes everything every time you change anything
  1. **If gcc didn’t combine steps behind your back, you’d need to preprocess and compile each file, then run the linker**
  2. If another program (e.g., sed) created some C files, you would need an “earlier” step
  3. If you have other outputs for the same source files (e.g., javadoc), it’s unpleasant to type the source file names multiple times
  4. If you want to distribute source code to be built by other users, you don’t want to explain the build logic to them
  5. If you have 10^5 to 10^7 lines of source code, you don’t want to recompile them all every time you change something
- A simple script handles 1–4 (use a variable for filenames for 3), but 5 is trickier

“Real” build process

- On larger projects, you can’t or don’t want to have one big (set of) command(s) that redoes everything every time you change anything
  1. If gcc didn’t combine steps behind your back, you’d need to preprocess and compile each file, then run the linker
  2. **If another program (e.g., sed) created some C files, you would need an “earlier” step**
  3. If you have other outputs for the same source files (e.g., javadoc), it’s unpleasant to type the source file names multiple times
  4. If you want to distribute source code to be built by other users, you don’t want to explain the build logic to them
  5. If you have 10^5 to 10^7 lines of source code, you don’t want to recompile them all every time you change something
- A simple script handles 1–4 (use a variable for filenames for 3), but 5 is trickier

"Real" build process

- On larger projects, you can't or don't want to have one big (set of) command(s) that redoes everything every time you change anything
  1. If gcc didn't combine steps behind your back, you'd need to preprocess and compile each file, then run the linker
  2. If another program (e.g., sed) created some C files, you would need an "earlier" step
  3. **If you have other outputs for the same source files (e.g., javadoc), it's unpleasant to type the source file names multiple times**
  4. If you want to distribute source code to be built by other users, you don't want to explain the build logic to them
  5. If you have 10^5 to 10^7 lines of source code, you don't want to recompile them all every time you change something
- A simple script handles 1–4 (use a variable for filenames for 3), but 5 is trickier

“Real” build process

- On larger projects, you can’t or don’t want to have one big (set of) command(s) that redoes everything every time you change anything
  1. If gcc didn’t combine steps behind your back, you’d need to preprocess and compile each file, then run the linker
  2. If another program (e.g., sed) created some C files, you would need an “earlier” step
  3. If you have other outputs for the same source files (e.g., javadoc), it’s unpleasant to type the source file names multiple times
  4. **If you want to distribute source code to be built by other users, you don’t want to explain the build logic to them**
  5. If you have 10^5 to 10^7 lines of source code, you don’t want to recompile them all every time you change something
- A simple script handles 1–4 (use a variable for filenames for 3), but 5 is trickier

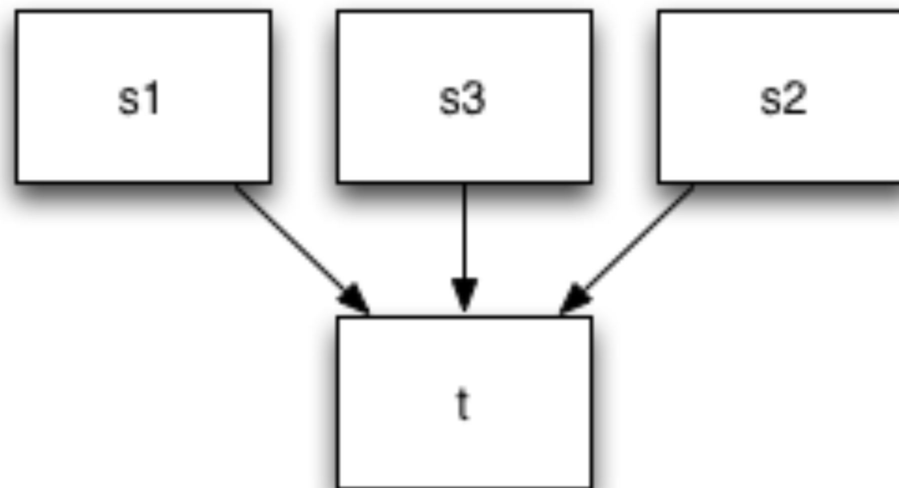
“Real” build process

- On larger projects, you can’t or don’t want to have one big (set of) command(s) that redoes everything every time you change anything
  1. If gcc didn’t combine steps behind your back, you’d need to preprocess and compile each file, then run the linker
  2. If another program (e.g., sed) created some C files, you would need an “earlier” step
  3. If you have other outputs for the same source files (e.g., javadoc), it’s unpleasant to type the source file names multiple times
  4. If you want to distribute source code to be built by other users, you don’t want to explain the build logic to them
  5. **If you have 10^5 to 10^7 lines of source code, you don’t want to recompile them all every time you change something**
- A simple script handles 1–4 (use a variable for filenames for 3), but 5 is trickier

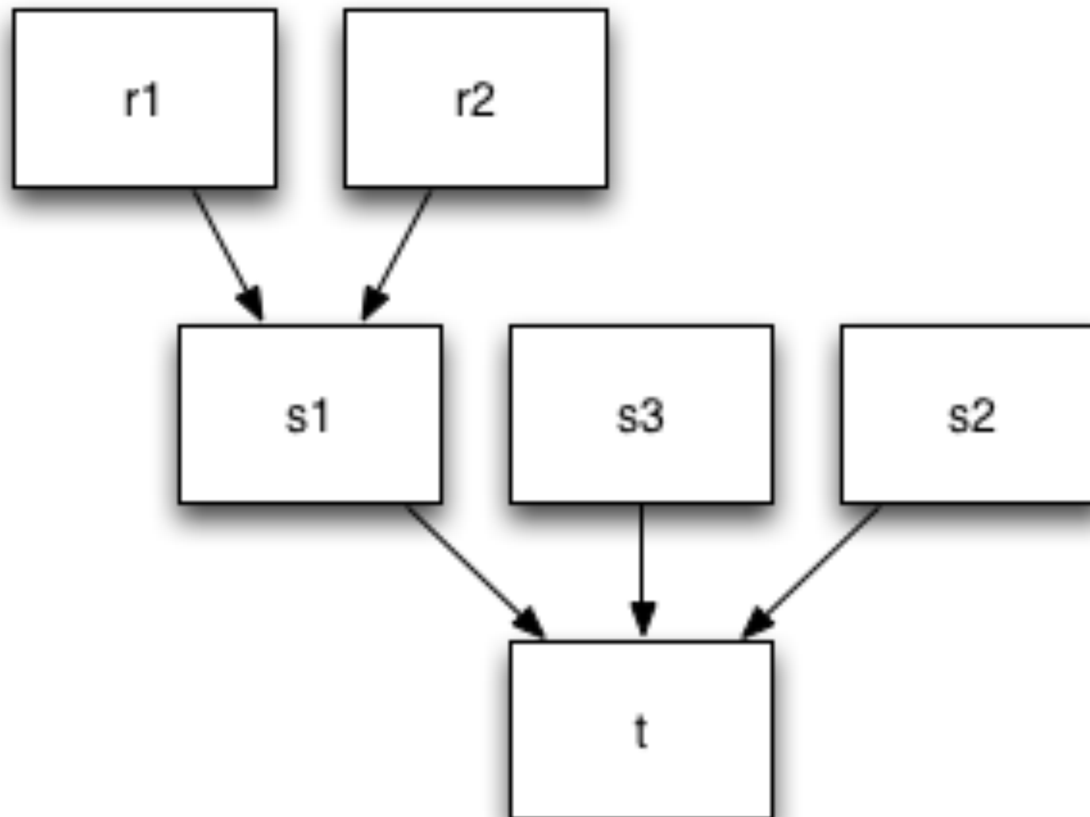
Recompilation management

- The "theory" behind avoiding unnecessary compilation is a "dependency dag" (**d**irected, **a**cyclic **g**raph):
- To create a target t, you need sources $s_1, s_2, \ldots, s_n$ and a command c (that directly or indirectly uses the sources)
- If t is newer than every source (file-modification times), assume there is no reason to rebuild it
- Recursive building: If some source s_i is itself a target for some other sources, see if it needs to be rebuilt…
- Cycles "make no sense"

Dependency DAG



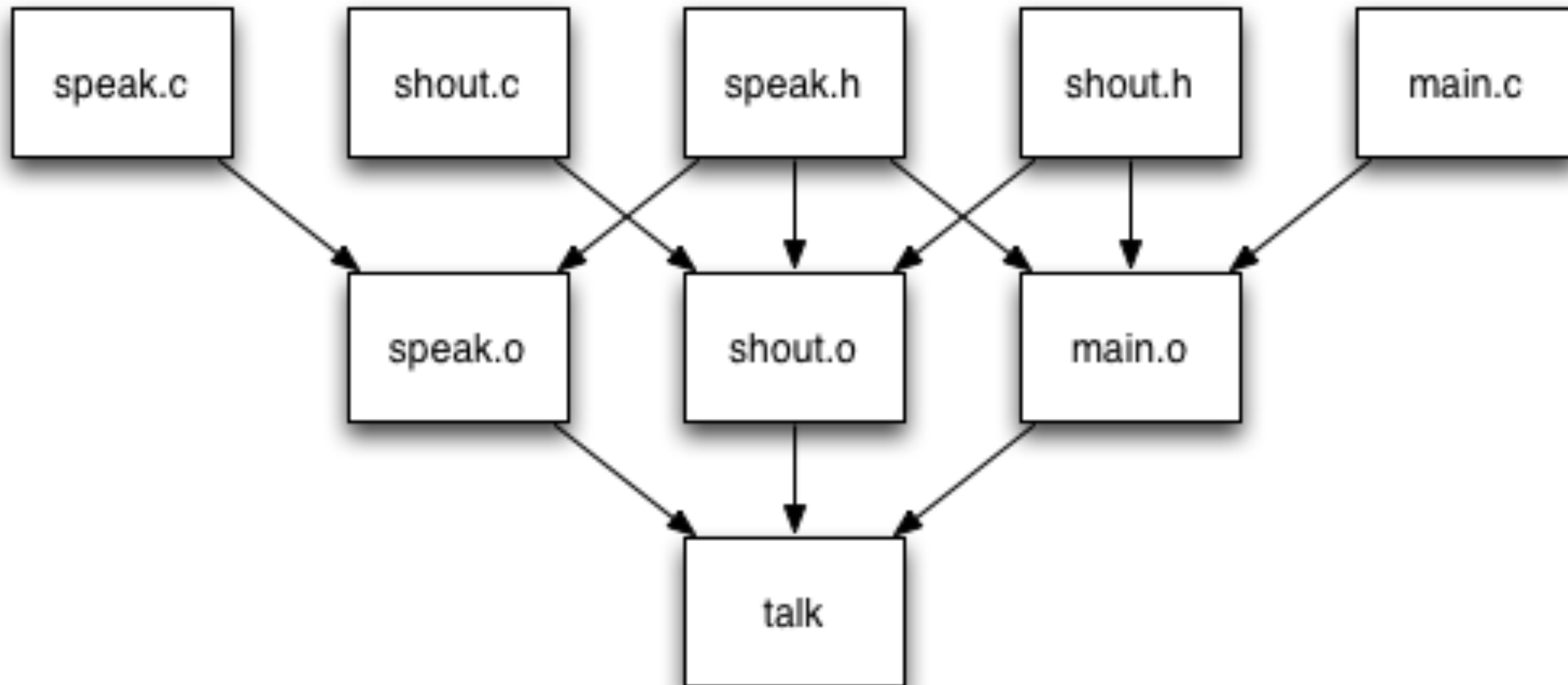
Dependency DAG



Theory applied to C

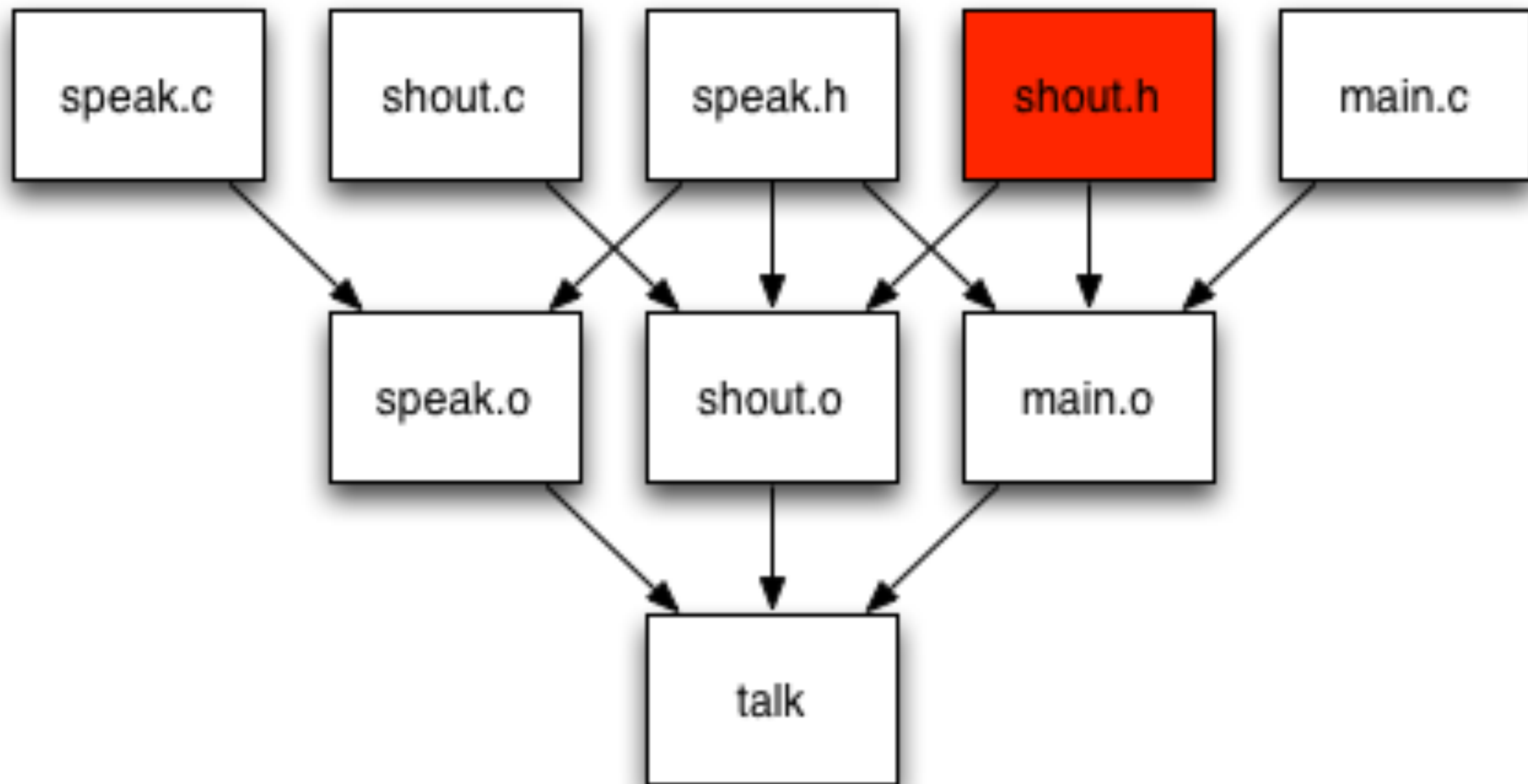
- Here is what we need to know today for C (still need to talk more about linking in a future lecture)
  - Compiling a .c creates a .o – the .o depends on the .c and all included files (.h files, recursively/ transitively)
  - Creating an executable ("linking") depends on .o files
  - So if one .c file changes, just need to recreate one .o file and relink
  - If a header file changes, may need to rebuild more
  - Of course, this is only the simplest situation

Dependency DAG for C

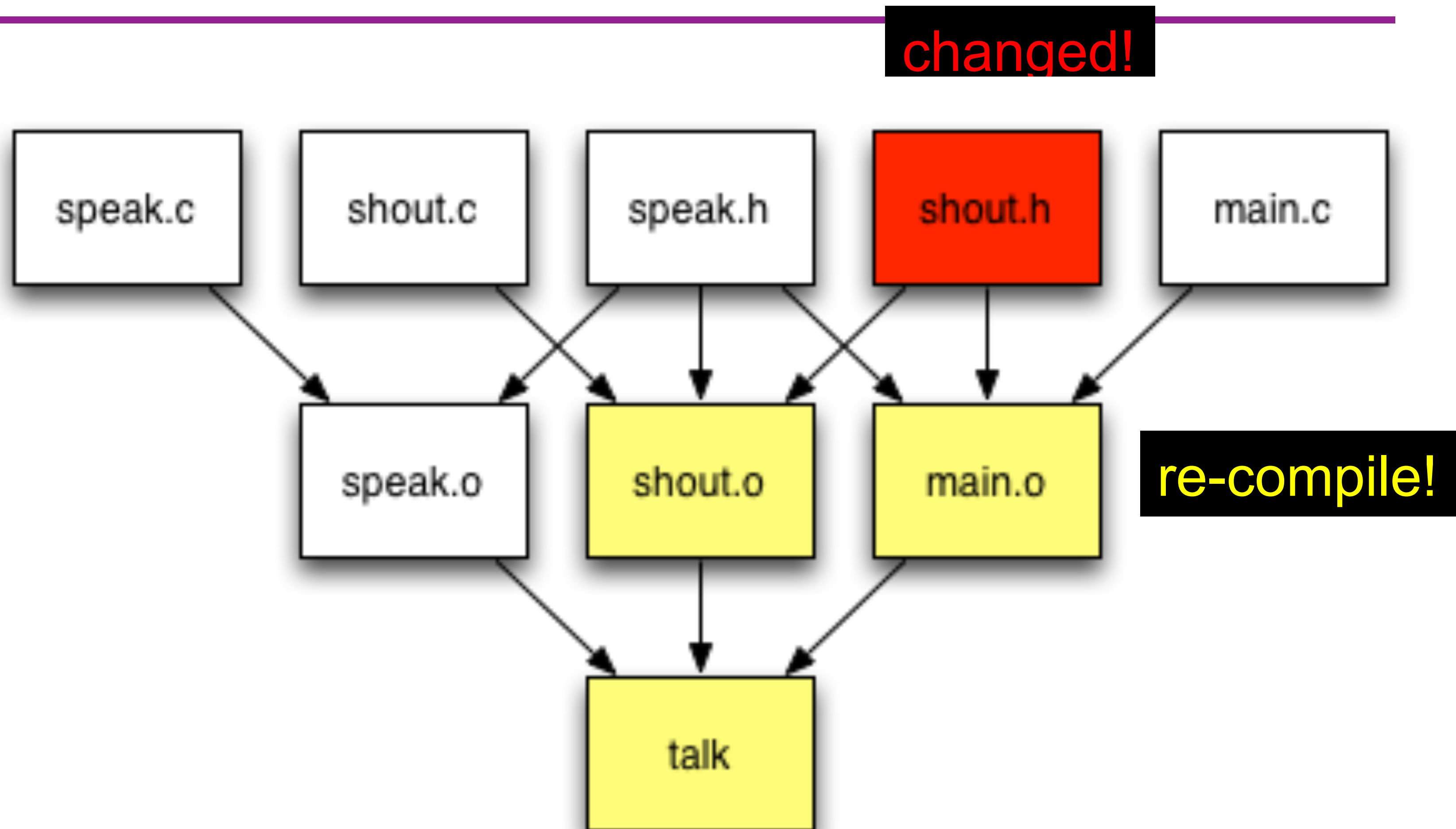


Dependency DAG for C

changed!



Dependency DAG for C



A program

- What would a program (e.g., a shell script) that did this for you look like? It would take:
  - a bunch of triples: ***target***, ***sources***, ***commands*** for getting the target file from source files
  - a "current target to build"
- It would compute what commands needed to be executed, in what order, and do it (it would detect cycles and give an error)
- This is exactly what programs like make, ant, and build tools integrated into IDEs do!

make basics

The “triples” are typed into a “makefile” like this:

```
target: sources
        command
```

Example:

```
foo.o: foo.c foo.h bar.h
	gcc -Wall -o foo.o -c foo.c
```

TAB

Syntax gotchas:

- The colon after the target is required
- Command lines must start with a ***TAB NOT SPACES***
- You can actually have multiple commands (executed in order); if *one command* spans lines you must end the previous line with \
- Which shell-language interprets the commands? (Typically bash; to be sure, set the SHELL variable in your makefile.)

Using make

At the prompt:

```
prompt% make -f nameOfMakefile aTarget
```

Defaults:

- If no -f specified, use a file named Makefile
- If no target specified, use the first one in the file

Building software that uses make

- Open source usage: You can download a tarball, extract it, type make (four characters) and everything should work
- Actually, there's typically a "configure" step first, for finding things like "where is the compiler" that generates the Makefile (but we won't get into that)
  - The mantra: ./configure; make; make install
  - many READMEs or INSTALLs boil down to these three commands

HW 6

- Build your own memory management library (malloc and free)!
- Two parts
  - part I: skeleton code (header files) checked into git repository due 2/26
  - part II: fully working memory manager due 3/5
- assignment appears tomorrow 2/19

HW 6 Project partners

- If you haven't yet, find a project partner **now**
- Use today after class or the discussion board, or Friday in class at the latest

- 80 students → 40 pairs → everyone *must* have a partner

- *One partner* in every pair submits a text file to *"HW6 - Project partners"* in the dropbox (directions are in the dropbox)
  - partner choices due by Saturday night 2/21
  - no late days on this; 1% of project score
- Staff will send out git repository info by Sunday

Midterm common errors

Midterm common shell errors

- stdin/out vs command line arguments vs exit code
- The documentation will describe what is what
  - c1 | c2 # stdout of c1 into *stdin* of c2
  - c2 `c1` # use the stdout of c1 as *arguments* to c2
  - v=`c1` # stdout of c1 assigned to variable v
  - c1; v=$? # exit code of c1 assigned to variable v
  - if cmd; then… # if uses exit code of the provided command to determine true/false
  - if [[-f $f]]; then… # the [[-f $f]] is just the test command

Midterm #2c

- “i before e except after c”. Lines containing ie (except for cie) or cei. For example, it should match re**cei**ve, s**ie**ve, th**ie**f, BUT NOT match currencies.
- `((^|[^c])ie)|cei`

Midterm #3a

```
alice    12277   0.0  0.1  2472836 6896   ??  S     Sat07PM   0:44.05 /usr/bin/top
alice    13275   0.0  0.0  2497564 3564   ??  S     Sat07PM   0:07.55 /usr/bin/display
root     274     0.0  0.1  2497520 10072  ??  Ss    Sat07PM   0:14.29 /usr/sbin/sshd
bob      10273   0.0  0.1  2490020  3321 ??  S     Sat07PM   0:03.11 /usr/bin/find

…
```

the .* is greedy; limit which characters can match

```
ps aux | grep alice | sed 's/.*([0-9]+)/\1/'
ps aux | grep alice | sed 's/[^ ]+[ ]+([0-9]+)/\1/'

# if you are replacing whole line with \1,
# then need to match whole line
ps aux | grep alice | sed 's/[^ ]+[ ]+([0-9]+)/\1/'
ps aux | grep alice | sed 's/[^ ]+[ ]+([0-9]+).*/\1/'
```