CSE 374: Lecture 17

Makefiles

Programming Tools

So far: Gcc, Gdb, Valgrind, Git

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

Today: Make & makefiles

https://www.gnu.org/software/make/manual/

Why?

Programmers are lazy

Compilation commands get long

`($gcc -Wall -std=c11 -g -D DEBUG -o demo demo1.c demo2.c)`

Build processes get longer with multiple files

Automating process reduces both typing and errors

Large projects can take hours to compile: Makefiles provide options

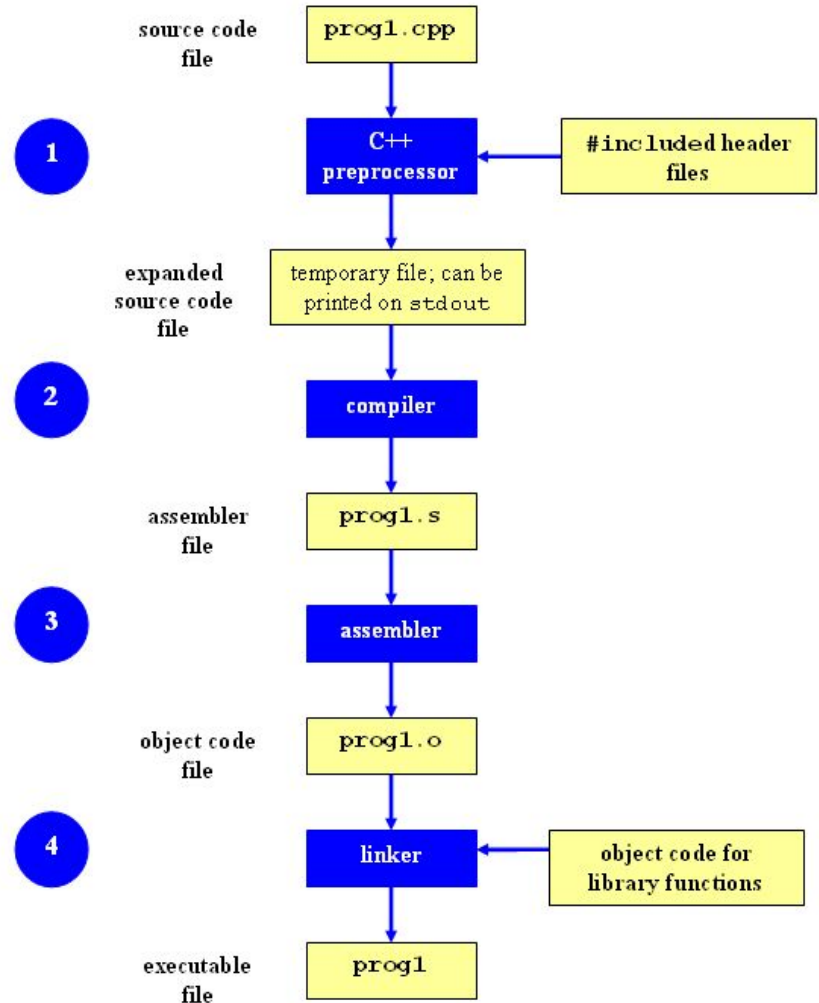Makes automated installation for shipped products possible

What we Make

Compiler actually runs in stages:

a. Preprocessor
b. Compiler
c. Assembler
d. Linker

There are other tools to manage this:

● IDEs
● Projects
● Ant

source code file — `prog1.cpp`

1 — C++ preprocessor ← #included header files

expanded source code file — temporary file; can be printed on `stdout`

2 — compiler

assembler file — `prog1.s`

3 — assembler

object code file — `prog1.o`

4 — linker ← object code for library functions
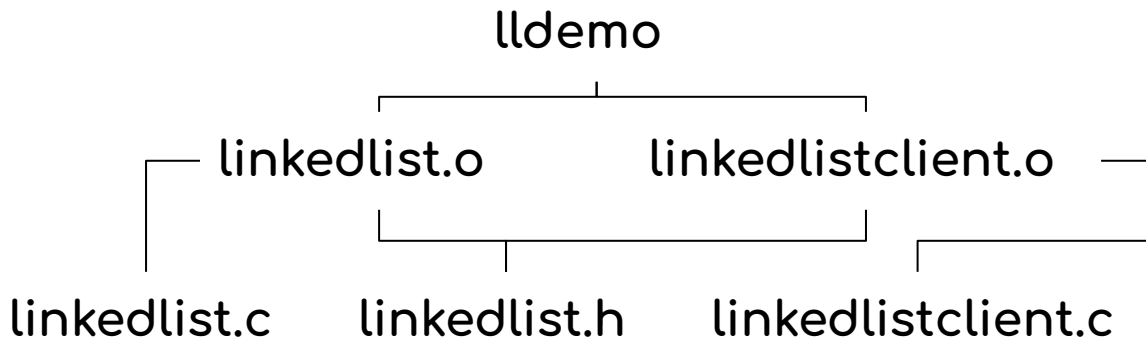
executable file — `prog1`

Dependency Tree - *helps decide what to do*

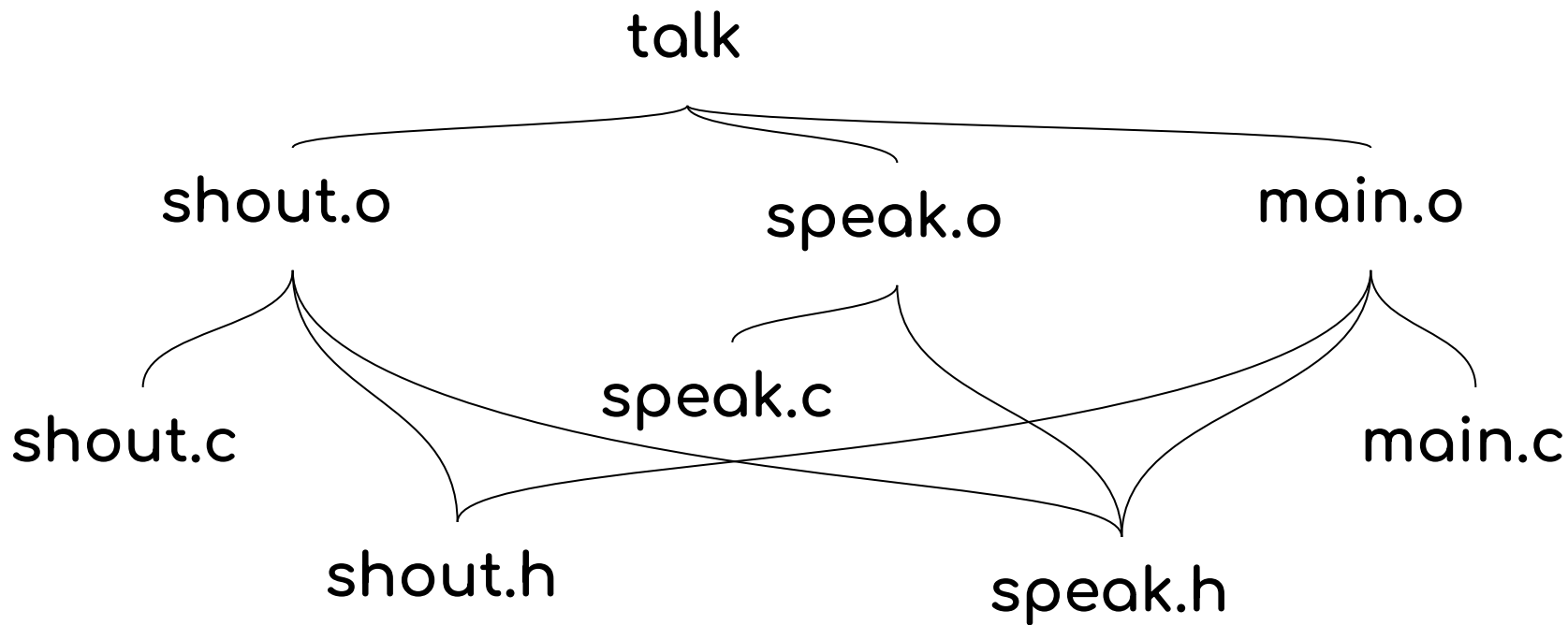Each target T is dependent on one or more sources S
    If any S is newer than T, remake T.

Recursive: If a source is also a target for other sources, must also evaluate its dependencies and possibly remake

Directed-acyclic-graph
(cycles make no sense)

```
                              lldemo
                    ┌────────────┴────────────┐
        ┌── linkedlist.o          linkedlistclient.o ──┐
        │         └──────┬──────────────┘              │
  linkedlist.c     linkedlist.h      linkedlistclient.c
```

Talk Demo

An algorithm to Make targets

➔ Calculate build from dependencies
➔ Have a list of tuples

◆ Each tuple made of Target, Sources, Command to build Target
◆ Recursively determine which targets must be rebuilt
● Usually if one or more sources is newer than the target

➔ Execute all necessary commands
➔ Including re-linking the object files

Make Basics

- Target - output file
- Prerequisites - sources needed for that output
- Recipe - the command needed to generate target.
- More than one command is possible, possibly on multiple lines.
- You don't always need sources

$make uses a Makefile to determine what to do

Makefiles consist of rules in the form:

Must have colon:

```
Target … : prerequisites …

    recipe/command

    ...
```

Make isn't language specific: recipe may be any valid shell command

Unless specifically set otherwise, recipes MUST be indented with TAB not spaces.

Multi-line commands may \ Have lines split with \ backslashes

Make Basics

$make uses a Makefile to determine what to do

Makefiles consist of rules in the form:

```
talk: main.o speak.o shout.o

    gcc -Wall -std=c11 -g -o talk main.o speak.o shout.o
```

Target is final executable:

Sources are the object files of all the different C programs.

Recipe, or command, is gcc shell command

Special Rules

'Phony' targets

Targets are not actually files, but often used commands

A phony target may have no dependencies:

```
clean:
    rm -f *.o talk *~
```

(Phony targets will never be called if files by those names exist, so must be forced. Refer to manual:
https://www.gnu.org/software/make/manual/html_node/Phony-Targets.html)

'All' is a special phony target that just specifies what to make in a complete build.

Often the first 'default' target

```
# phony all specifies every \
  target to make.
all: talk
talk: main.o speak.o shout.o
    gcc -Wall -std=c11 -g -o talk
main.o speak.o shout.o
```

Variables

- You can define variables in Makefiles
  - Set defaults at top of file
  - Reduce repetitive typing
  - Change variables at command line
  - Reuse Makefiles on new projects
  - Use conditionals to choose variable settings

make -f MakefileV2_RenameToMakefile CFLAGS=-Wall

```
CC = gcc
CFLAGS = -Wall
foo.o: foo.c foo.h bar.h
    $(CC) $(CFLAGS) -c foo.c -o foo.o


make CFLAGS=-g


EXE=
ifdef WINDIR # defined on Windows
  EXE=.exe
endif
widget$(EXE): foo.o bar.o
    $(CC) $(CFLAGS) -o widget$(EXE)\
      foo.o bar.o


OBJFILES = foo.o bar.o baz.o
widget: $(OBJFILES)
    gcc -o widget $(OBJFILES)


clean:
    rm $(OBJFILES) widget
```

Extra Characters

➤ **In commands (short list):**
- ○ **$@ for target**
- ○ **$^ for all sources**
- ○ **$< for left-most source**

➤ **Examples:**
- ○ **widget$(EXE): foo.o bar.o**
  $(CC) $(CFLAGS) -o
  $@ $^
- ○ **foo.o: foo.c foo.h bar.h**
  $(CC) $(CFLAGS) -c $<

Also use wild cards (ex. *.o), but you need to be careful.

Use the 'wildcard' function for precision.

`$(wildcard *.o)`

Fancy Stuff *(use with care!)*

Implicit rules:

Make automatically applies rules to common types of files

```
n.o is made automatically from n.c with a recipe of
the form '$(CC) $(CPPFLAGS) $(CFLAGS) -c'.
```

Pattern rules:

Define new implicit rules by using '%' as a type of wildcard

```
%.o : %.c
    $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@

%.class: %.java
    javac $<      # Note we need $< here
```

Commands can be any valid shell command, including shell scripts

Repeating targets can add dependencies (useful for automatic target generation)

Suffix rules:
Old form of pattern rules using only suffixes

Dependency Generation

Make has no knowledge of dependency trees

    If you make a mistake in your source list make can't fix it.

    Consider auto-generation:

In C:

```
$gcc -MM target.c
```

Can 'make depend':

```
depend: $(PROGRAM_C_FILES)
    gcc -M $^
```

Summary

★ $make uses Makefiles to encode build processes
  ○ Automate process
    ■ For shipment?
    ■ Convenience
    ■ Reliability
  ○ Reduce unnecessary rebuilds
  ○ Provide build options
★ $make relies on tuples of [Target(s), Source(s), Command(s)]
★ $make relies on timestamps and shell commands
  ○ Language independent
★ Many convenient additional variations
  ○ Use with care - can obfuscate meaning

Problem of multiple 'main' functions

```c
//sample.c
#ifdef WIN32
int main() {
    //in this case only this main()
       will be compiled.
}
#endif

#ifdef LINUX
int main() {
  //another main for linux platform
}
#endif

# sample Makefile
ifdef WINDIR # defined on Windows
   CFLAGS += -D WIN32
endif
```

You would not use two 'main' functions, because main is always the single entry point.

(Note: It works in Java because we can define one 'main' for each class namespace. We don't have the same concept of namespaces in C.)

Your code could define two mains, and choose one at pre-process time.

You could also include code that was chosen with a compiler flag (such as `#ifdef DEBUG`).