CSE 374: Programming Concepts and Tools

Spring 2026
Instructor: Megan Hazen

Today's Goals

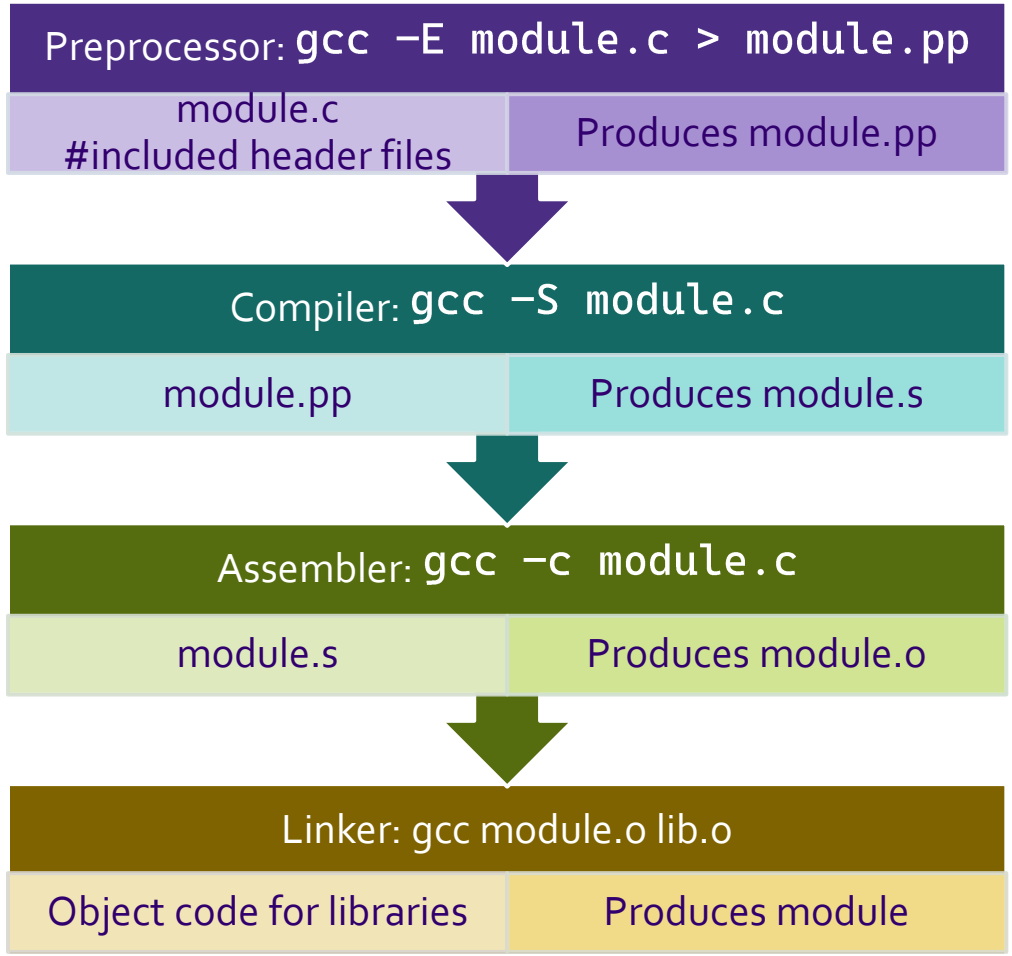
Subject Matter

- Make and Makefiles

Your Goals

- Debug interview scheduling
- Work on your Makefile for HW5

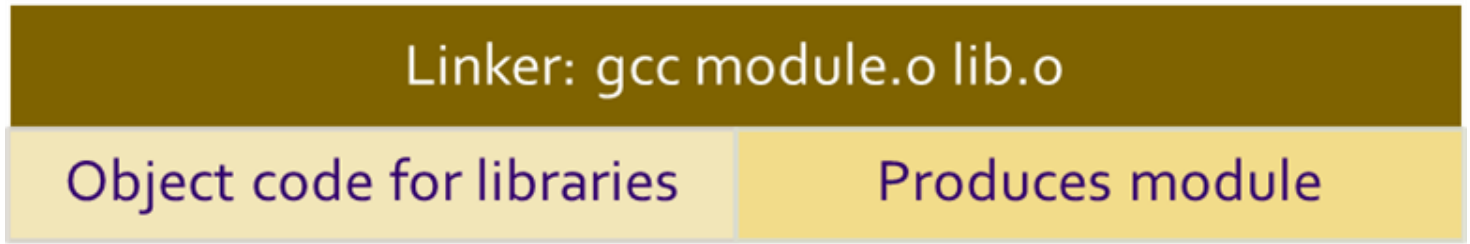
Compiling process



- Each step feeds into the next
- You can stop the process at any time
  - And do: when building a large project you often build object code and link it in separate steps.

Building projects

- Projects may have many steps to building a working application
  - Installing or updating dependencies
  - Changing path variables
  - Compiling multiple modules
  - Producing an executable
- Managing projects may have other steps
  - Removing old versions of the code
  - Running test suites
  - Compiling alternative versions



Programming tools

We already know a lot!

Why Make?

- *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

Make What?

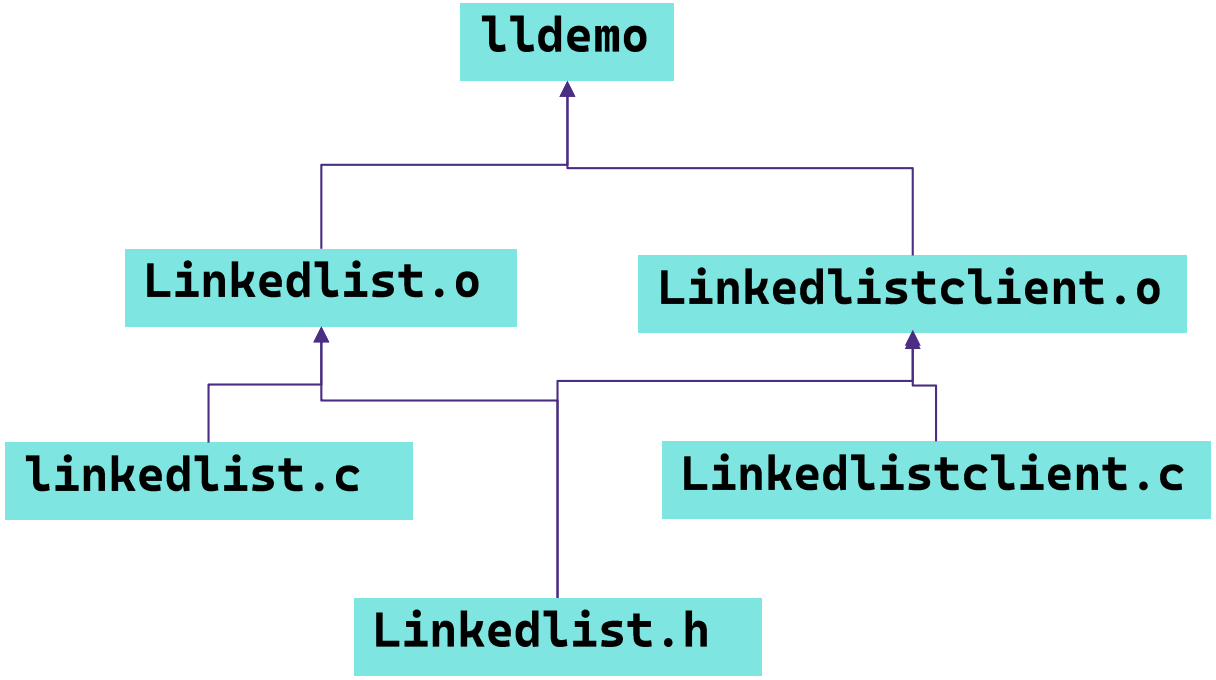
"The make utility automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them. "

- Make works by writing rules to act on a dependency tree
  - Rule say what to do
  - Based on whether their dependencies are fresh.

"Our examples show C programs, since they are most common, but you can use make with any programming language whose compiler can be run with a shell command. Indeed, make is not limited to programs. You can use it to describe any task where some files must be updated automatically from others whenever the others change."

Dependency Tree

- 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: must also evaluate S to see if any S' are newer than S
  - Remake S
- The tree should be a directed acyclic graph



Make Algorithm

- Create rules (tuples) in the form:
  - Target: dependencies, command

```
make_target (Target, dependencies, command):
    for each dependency:
        if dependency OutOfDate
            make_target (dependency)
    Target = execute command
```

Note: Getting the rules

- Create rules (tuples) in the form:
  - Target: dependencies, command
- Make has no knowledge of dependency trees
  YOU must writ the rules with the right dependency lists
  Consider auto-generation:

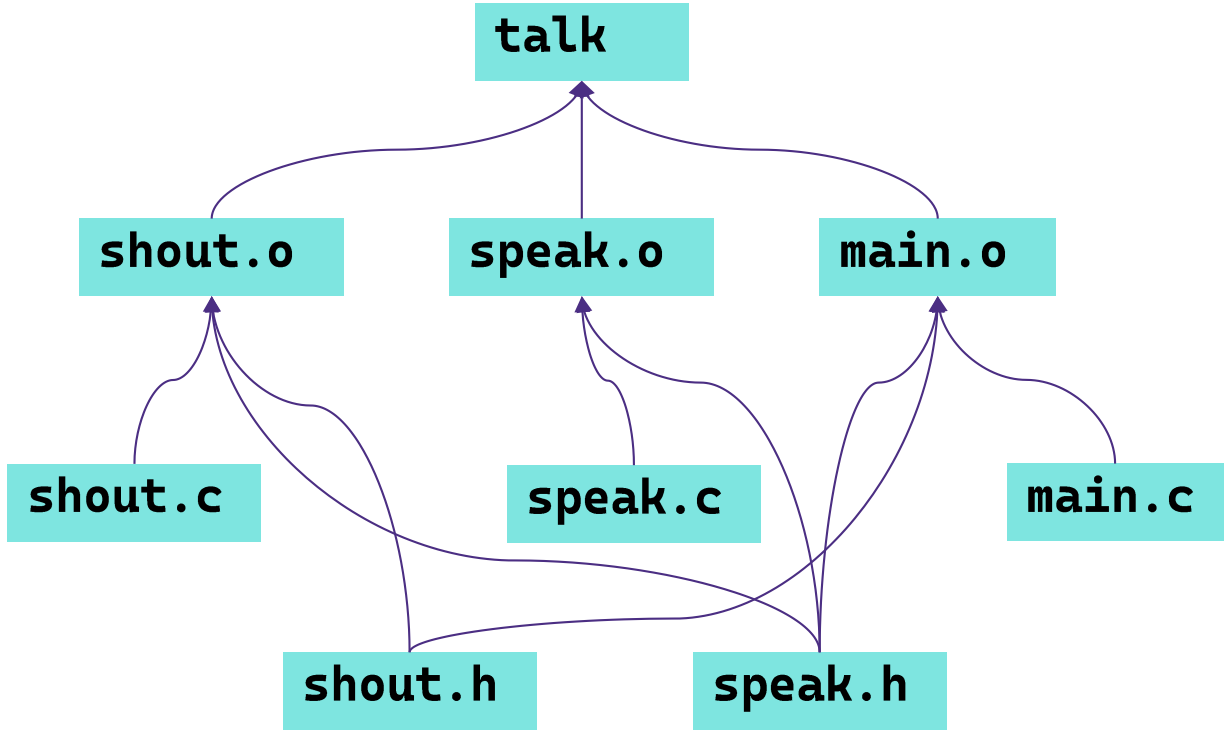
  In C:
  ```
  $gcc -MM target.c
  ```

  Can 'make depend':
  ```
  depend: $(PROGRAM_C_FILES)
  gcc -M $^
  ```

<< More on that shorthand later

Talk Demo

- Talk depend on modules shout, speak, and main.
- Shout depends on the shout module and shout and speak headers.
- Speak depends on the speak module and the speak header.
- Main depends on the main module and both headers.




```
[mh75@calgary cse374]$ gcc -M main.c

main.o: main.c /usr/include/stdc-predef.h
speak.h shout.h
```

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:

```
Target … : prerequisites …
        recipe/command \
        command2
 …
```

Must have a colon:

Must be indented with a TAB

Make Basics Example

Name of target, here it is the executable

List of sources, here they are object files

Recipe, or command, here is compile command

`$make` uses a `Makefile` to determine what to do

`Makefiles` consist of rules in the form:

```
talk: main.o speak.o shout.o
   gcc –std=c11 –o talk main.o speak.o shout.o
```

Special Targets

- 'All'
  - Used to specify the targets that make up a complete build
  - Often first, or default, target
- 'Phony' targets
  - Not actually a target file, but used as a command
  - Will not be called if files by those names exist

`$make` uses a `Makefile` to determine what to do

`Makefiles` consist of rules in the form:

```
all: talk

talk: main.o speak.o shout.o
   gcc –std=c11 –o talk main.o
speak.o shout.o

clean:
    rm –f *.o talk *~
```

Variables

- Variables in Makefiles
  - Similar to bash (can have space around =)
  - 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

```
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
```

Makefile1

Makefile2

Makefile3

demos

Extra special 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)

https://www.gnu.org/software/make/manual/html_node/Wildcard-Function.html#Wildcard-Function

Fancy Stuff (use with care!)

- Implicit rules: 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

Conditional Compilation

```
#ifdef WIN32
// only compiled if WIN32 is defined
int main() {
  …
}
#else
// only compiled if LINUX
int main() {
  …
}
#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).

Practice

What do these rules do?

```
CC = gcc
CFLAGS = -g -Wall -std=c11
all: t9_tests t9_tests_buggy t9_demo
test: t9_tests
    ./t9_tests
test-buggy: t9_tests_buggy
    ./t9_tests_buggy
t9_tests: t9_tests.o t9_lib.o
    $(CC) $(CFLAGS) -o $@ $^
t9_tests_buggy: t9_tests.o t9_lib_buggy.o
    $(CC) $(CFLAGS) -o $@ $^
t9_demo: t9_demo.o t9_lib.o
    $(CC) $(CFLAGS) -o $@ $^
t9_tests.o: t9_tests.c t9_lib.h safe_assert.h
    $(CC) $(CFLAGS) -o $@ -c $<
t9_demo.o: t9_demo.c t9_lib.h
    $(CC) $(CFLAGS) -o $@ -c $<
clean:
    rm -f t9_demo.o t9_tests.o t9_tests
  t9_tests_buggy t9_demo
```