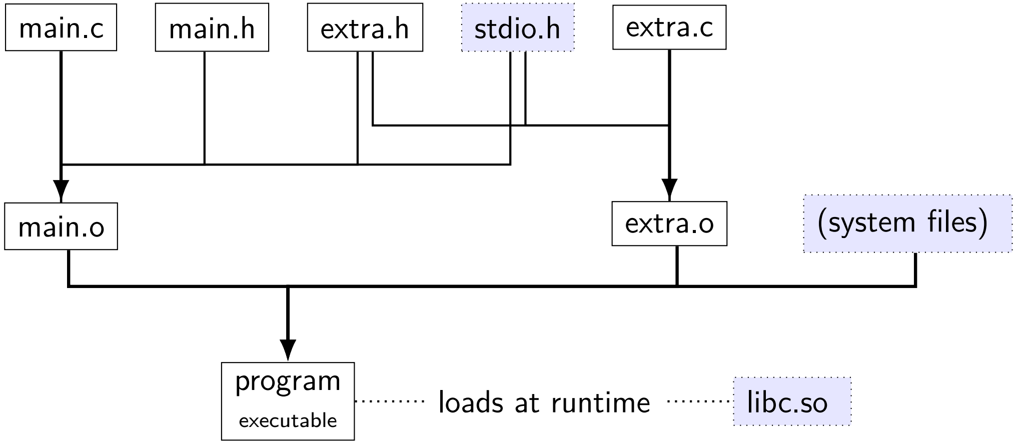
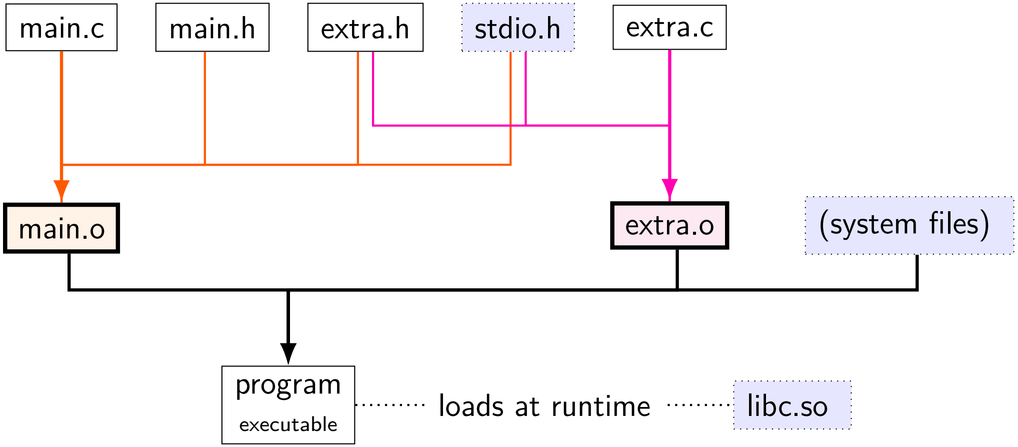
files in building C programs [dynamic linking]

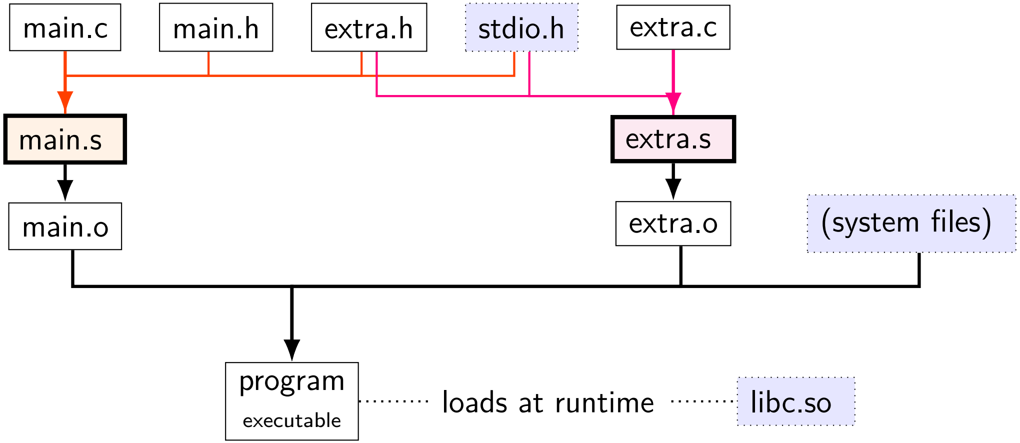


files in building C programs [dynamic linking]



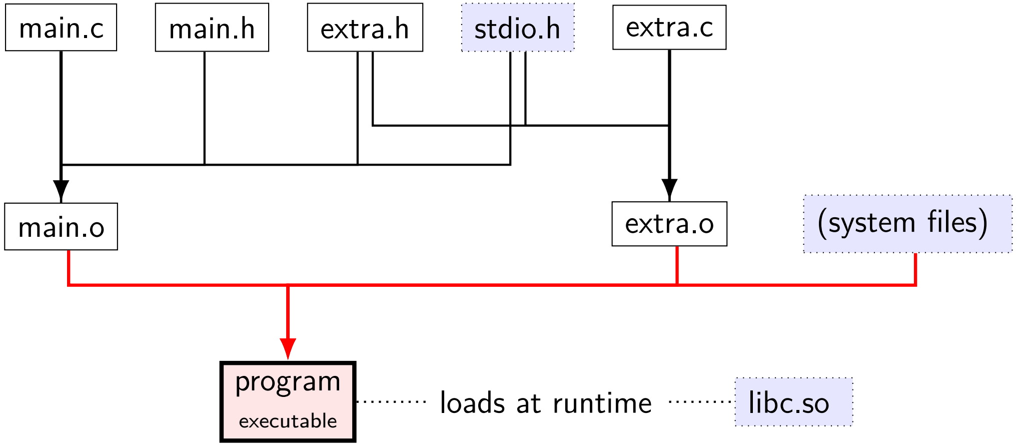
```
clang -c main.c
clang -c extra.c
```

files in building C programs [dynamic linking]



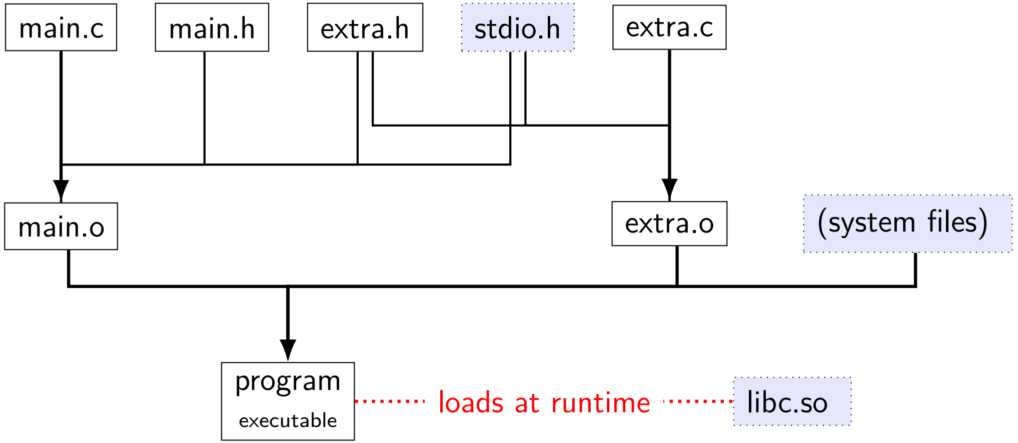
```
clang -S -c main.c
clang -S -c extra.c
```

files in building C programs [dynamic linking]



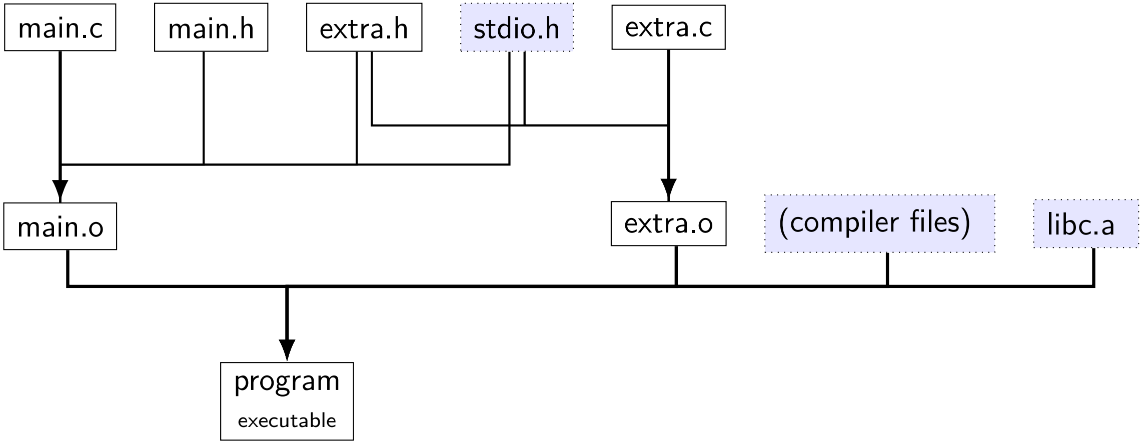
```
clang -o program main.o extra.o
```

files in building C programs [dynamic linking]



```
./program ...
```

files in building C programs [static linking]



file extensions

name		
`.c`		C source code
`.h`		C header file
`.s`	(or `.asm`)	assembly file
`.o`	(or `.obj`)	object file (binary of assembly)
(none)	(or `.exe`)	executable file
.a	(or `.lib`)	statically linked library [collection of .o files]
.so	(or `.dll` or `.dylib`)	dynamically linked library ['shared object']

static libraries

Unix-like *static* libraries: libfoo.a

internally: archive of .o files with index

create: `ar rcs libfoo.a file1.o file2.o …`

- 'archive' utility `ar` and not normal C compiler

use: `cc … -o program -L/path/to/lib … -lfoo`

- no space between `-l` and library name
- `cc` could be clang, gcc, clang++, g++, etc.
- `-L/path/to/lib` not needed if in standard location

shared libraries

Linux *shared* libraries: libfoo.so

create:

- compile .o files with `-fPIC` (position independent code)
- then: `cc -shared … -o libfoo.so`

use: `cc …-o program -L/path/to/lib …-lfoo`

- `cc` = C compiler (clang, gcc, etc.)

shared libraries

Linux *shared* libraries: libfoo.so

create:

compile .o files with `-fPIC` (position independent code)
then: `cc -shared … -o libfoo.so`

use: `cc …-o program` *`-L/path/to/lib`* `…-lfoo`

`cc` = C compiler (clang, gcc, etc.)

`-L...` sets path *only when making executable*

runtime path set separately

finding shared libraries (1)

```
$ ls
libexample.so  main.c
$ clang -o main main.c -lexample
/usr/bin/ld: cannot find -lexample
clang: error: linker command failed with exit code 1 (use -v to see
$ clang -o main main.c -L. -lexample
$ ./main
./main: error while loading shared libraries:
    libexample.so: cannot open shared object file: No such
    file or directory
```

finding shared libraries (1)

```
$ ls
libexample.so  main.c
$ clang -o main main.c -lexample
/usr/bin/ld: cannot find -lexample
clang: error: linker command failed with exit code 1 (use -v to see
$ clang -o main main.c -L. -lexample
$ ./main
./main: error while loading shared libraries:
    libexample.so: cannot open shared object file: No such
    file or directory
```

```
$ LD_LIBRARY_PATH=. ./main
```

or

```
$ export LD_LIBRARY_PATH=.
$ ./main
```

or

```
$ clang -o main main.c -L. -lexample -Wl,-rpath .
$ ./main
```

finding shared libraries (1)

`cc …-o program -L/path/to/lib …-lfoo`

- on Linux: `/path/to/lib` only used to create program
- program contains `libfoo.so` *without full path*

Linux default: `libfoo.so` expected to be in /usr/lib, /lib, and other 'standard' locations

possible overrides:

- LD_LIBRARY_PATH environment variable
- paths specified with `-Wl,-rpath=/path/to/lib` when creating executable

libraries and command line

when linking against libraries use:

```
clang -o executable foo.o bar.o -lName
```

rather than

~~`clang -o executable -lName foo.o bar.o`~~

by default, linker processes files in order

might only grab things that previous files needed from library

(especially for static libraries)

exercise (incremental compilation)

program built from main.c + extra.c

main.c, extra.c both include extra.h, stdio.h

```
clang -c main.c                        # command 1
clang -c extra.c                       # command 2
clang -o program main.o extra.o        # command 3
```

What commands need to be rerun if...

Question A: ...main.c changes?

Question B: ...extra.h changes?

make

`make` — Unix program for “making” things...

...by running commands based on what’s changed

what commands? based on *rules* in *makefile*

(text file called `makefile` or `Makefile` (no extension))

make rules

```
main.o: main.c main.h extra.h
▶           clang -Wall -c main.c
```

before colon: target(s) (file(s) generated/updated)

after colon: prerequisite(s) (also known as dependencies)

following lines prefixed by a tab character: command(s) to run

`make` runs commands if any prereq modified date after target

make rules

```
main.o: main.c main.h extra.h
►       clang -Wall -c main.c
```

before colon: *target(s)* (file(s) generated/updated)

after colon: prerequisite(s) (also known as dependencies)

following lines prefixed by a tab character: command(s) to run

`make` runs commands if any prereq modified date after target

make rules

```
main.o: main.c main.h extra.h
►          clang -Wall -c main.c
```

before colon: target(s) (file(s) generated/updated)

after colon: *prerequisite(s)* (also known as dependencies)

following lines prefixed by a tab character: command(s) to run

`make` runs commands if any prereq modified date after target

make rules

```
main.o: main.c main.h extra.h
►           clang -Wall -c main.c
```

before colon: target(s) (file(s) generated/updated)

after colon: prerequisite(s) (also known as dependencies)

following lines prefixed by a *tab* character: command(s) to run

`make` runs commands if any prereq modified date after target

make rules

```
main.o: main.c main.h extra.h
►           clang -Wall -c main.c
```

before colon: target(s) (file(s) generated/updated)

after colon: prerequisite(s) (also known as dependencies)

following lines prefixed by a tab character: *command(s) to run*

`make` runs commands if any prereq modified date after target

make rules

```
main.o: main.c main.h extra.h
►           clang -Wall -c main.c
```

before colon: target(s) (file(s) generated/updated)

after colon: prerequisite(s) (also known as dependencies)

following lines prefixed by a tab character: command(s) to run

`make` runs commands *if any prereq modified date after target*

make rules

```
main.o: main.c main.h extra.h
►           clang -Wall -c main.c
```

before colon: target(s) (file(s) generated/updated)

after colon: prerequisite(s) (also known as dependencies)

following lines prefixed by a tab character: command(s) to run

`make` runs commands if any prereq modified date after target

…*after making sure prerequisites up to date*

make rule chains

```
program: main.o extra.o
►          clang -Wall -o program main.o extra.o

extra.o: extra.c extra.h
►          clang -Wall -c extra.c

main.o: main.c main.h extra.h
►          clang -Wall -c main.c
```

to *make* `program`, first...

update main.o and extra.o if they aren't

running make

“`make` *`target`*”

> look in `Makefile` in current directory for rules
> check if *`target`* is up-to-date
> if not, rebuild it (and prerequisites, if needed) so it is

“`make` *`target1`* *`target2`*”

> check if both target1 and target2 are up-to-date
> if not, rebuild it as needed so they are

“`make`”

> if “*`firstTarget`*” is the first rule in Makefile,
> same as ‘`make` *`firstTarget`*”

exercise: what will run?

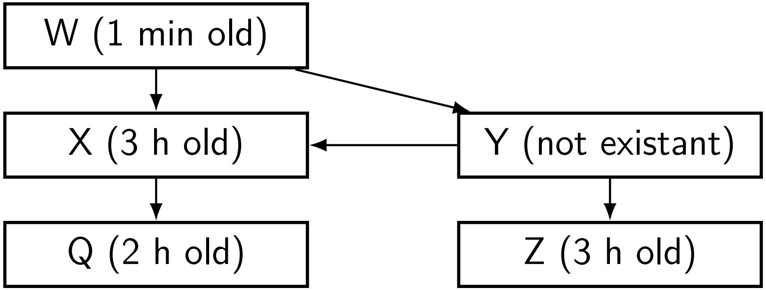
```
W: X Y
▶    buildW
X: Q
▶    buildX
Y: X Z
▶    buildY
```

W	modified 1 minute ago
X	modified 3 hours ago
Y	does not exist
Z	modified 1 hour ago
Q	modified 2 hours ago

exercise: “make W” will run what commands?

A. none
B. `buildY` only
C. `buildW` then `buildY`
D. `buildY` then `buildW`
E. `buildX` then `buildY` then `buildW`
F. `buildX` then `buildW`
G. something else

explanation



first: to make W, need X, Y up to date

to make X up to date:

need Q up to date ✓

then build X if less recent than Q (yes) ✓

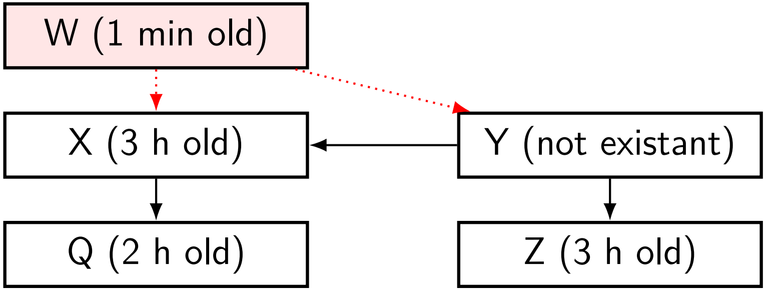
---

to make Y up to date: need X up to date ✓

need Z up to date ✓

then build Y if less recent than X (yes) or Z (yes) ✓

explanation



first: to make W, *need X, Y up to date*

to make X up to date:

need Q up to date ✓

then build X if less recent than Q (yes) ✓

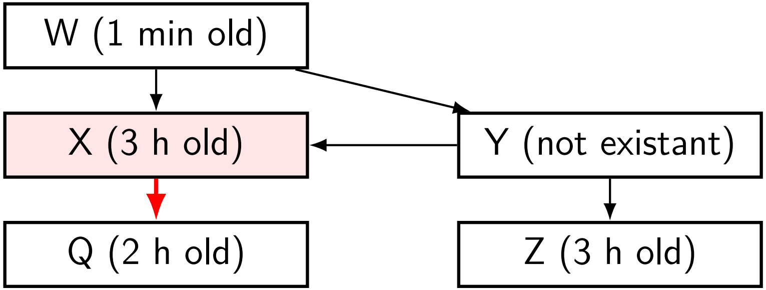
---

to make Y up to date: need X up to date ✓

need Z up to date ✓

then build Y if less recent than X (yes) or Z (yes) ✓

explanation



first: to make W, need X, Y up to date

to make X up to date:

need Q up to date ✓

then build X if less recent than Q (yes) ✓

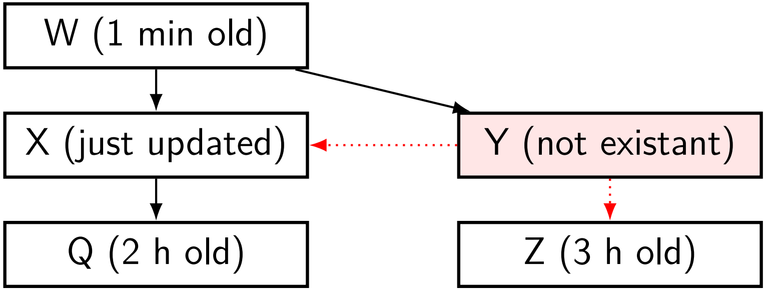
---

to make Y up to date: need X up to date ✓

need Z up to date ✓

then build Y if less recent than X (yes) or Z (yes) ✓

explanation



first: to make W, need X, Y up to date

to make X up to date:

need Q up to date ✓

then build X if less recent than Q (yes) ✓

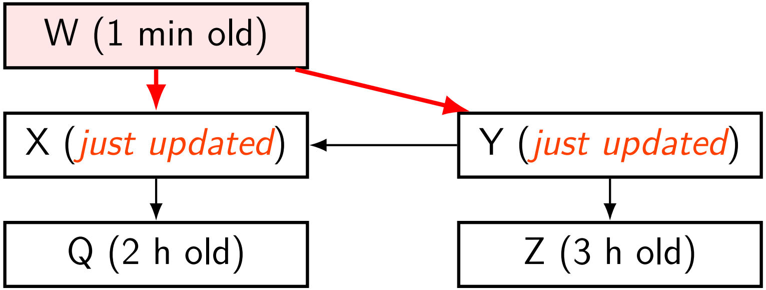
---

to make Y up to date: need X up to date ✓

need Z up to date ✓

then build Y if less recent than X (yes) or Z (yes) ✓

explanation



first: to make W, need X, Y up to date

to make X up to date:

need Q up to date ✓

then build X if less recent than Q (yes) ✓

---

to make Y up to date: need X up to date ✓

need Z up to date ✓

then build Y if less recent than X (yes) or Z (yes) ✓

‘phony’ targets (1)

common to have Makefile targets that aren’t files

```
all: program1 program2 libfoo.a
```

“`make all`” effectively shorthand for “`make program1 program2 libfoo.a`”

no actual file called “all”

‘phony’ targets (2)

sometimes want targets that don’t actually build file

example: “`make clean`” to remove generated files

```
clean:
►           rm --force main.o extra.o
```

but what if I create...

```
clean:
▶           rm --force main.o extra.o

all: program1 program2 libfoo.a
```

Q: if I make a file called “all” and then “make all” what happens?

Q: same with “clean” and “make clean”?

marking phony targets

```
clean:
▶           rm --force main.o extra.o

all: program1 program2 libfoo.a

.PHONY: all clean
```

special .PHONY rule says “ ‘all’ and ‘clean’ not real files”

(not required by POSIX, but in every `make` version I know)

conventional targets

common convention:

target name	purpose
(default), `all`	build everything
`install`	install to standard location
`test`	run tests
`clean`	remove generated files

redundancy (1)

```
program: main.o extra.o
▶     clang -Wall -o program main.o extra.o

extra.o: extra.c extra.h
▶     clang -Wall -o extra.o -c extra.c
main.o: main.c main.h extra.h

▶     clang -o main.o -c main.c
```

what if I want to run `clang` with `-fsanitize=address` instead of `-Wall`?

what if I want to change `clang`to `gcc`?

variables/macros (1)

```
CC = gcc
CFLAGS = -Wall -pedantic -std=c11 -fsanitize=address
LDFLAGS = -Wall -pedantic -fsanitize=address
LDLIBS = -lm

program: main.o extra.o
▶       $(CC) $(LDFLAGS) -o program main.o extra.o $(LDLIBS)

extra.o: extra.c extra.h
▶       $(CC) $(CFLAGS) -o extra.o -c extra.c

main.o: main.c main.h extra.h
▶       $(CC) $(CFLAGS) -o main.o -c main.c
```

aside: conventional names

chose names CC, CFLAGS, LDFLAGS, etc.

not required, but conventional names (incomplete list follows)

CC	C compiler
CFLAGS	C compiler options
LDFLAGS	linking options
LIBS or LDLIBS	libraries

variables/macros (2)

$@: target
$<: first dependency
$^: all dependencies

```
CC = gcc
CFLAGS = -Wall
LDFLAGS = -Wall
LDLIBS = -lm

program: main.o extra.o
▶       $(CC) $(LDFLAGS) -o $@ $^ $(LDLIBS)

extra.o: extra.c extra.h
▶       $(CC) $(CFLAGS) -o $@ -c $<

main.o: main.c main.h extra.h
▶       $(CC) $(CFLAGS) -o $@ -c $<
```

aside: `$^` works on GNU make (usual on Linux), but not portable.

aside: make versions

multiple implementations of `make`

for stuff we've talked about so far, no differences

most common on Linux: GNU make

will talk about 'pattern rules', which aren't supported by some other make versions

- older, portable, (in my opinion less intuitive) alternative: suffix rules

pattern rules

```
CC = gcc
CFLAGS = -Wall
LDFLAGS = -Wall
LDLIBS = -lm

program: main.o extra.o
▶       $(CC) $(LDFLAGS) -o $@ $^ $(LDLIBS)

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

extra.o: extra.c extra.h
main.o: main.c main.h extra.h
```

built-in rules

'make' has the 'make .o from .c' rule built-in already, so:

```
CC = gcc
CFLAGS = -Wall
LDFLAGS = -Wall
LDLIBS = -lm

program: main.o extra.o
▶       $(CC) $(LDFLAGS) -o $@ $^ $(LDLIBS)

extra.o: extra.c extra.h
main.o: main.c main.h extra.h
```

(don't actually need to write supplied rule!)

built-in rules

'make' has the 'make .o from .c' rule built-in already, so:

```
CC = gcc
CFLAGS = -Wall
LDFLAGS = -Wall
LDLIBS = -lm

program: main.o extra.o
▶ $(CC) $(LDFLAGS) -o $@ $^ $(LDLIBS)

extra.o: extra.c extra.h
main.o: main.c main.h extra.h
```

note: built-in rules not allowed on the make lab

(don't actually need to write supplied rule!)

writing Makefiles?

error-prone to write all .h dependencies

`-MM` (and related) options to `gcc` or `clang`

- outputs make rule
- ways of having make run this + use output

Makefile generators

- other programs that write Makefiles

other build systems

alternatives to writing Makefiles:

other make-ish build systems

- ninja, scons, bazel, maven, xcodebuild, msbuild, …

tools that generate inputs for make-ish build systems

- cmake, autotools, qmake, …

backup slides

suffix rules

```
CC = gcc
CFLAGS = -Wall
LDFLAGS = -Wall

program: main.o extra.o
▶       $(CC) $(LDFLAGS) -o $@ $^

.c.o:
▶       $(CC) $(CFLAGS) -o $@ -c $<

extra.o: extra.c extra.h
main.o: main.c main.h extra.h
.SUFFIXES: .c .o
```