last time

course intro / logistics

building C programs (cc = clang/gcc, etc.)

`cc -c` *`file`*`.c` (makes *`file`*`.o`) — compile+assemble

- reads *`file`*`.c` and all the files it #includes

`cc` *`file1.o file2.o`* … `-o executable` — link

- reads .o files + some system files

…`-Lpath -l`*`name`* — libraries:

- static (`lib`*`name`*`.a`): included in executable itself
- dynamic (`lib`*`name`*`.so`): found + loaded at program start — one copy on system

runtime search paths for dynamic libraries

anonymous feedback (1)

Holding class in Mcleod is very hard for most of this class as I know at least DMT2, which a lot of us are in , is all the way in gilmer and is a more than 15 min to get here. I ask to please consider holding class somewhere else on grounds or starting class a few minutes later every day so that everyone has ample time to get here and be prepared to learn.

I'm pretty sure there's not an alternative room (I didn't volunteer for a long walk from Rice...)

disappointed to lose some lecture time, ...

quiz demo

anonymous feedback (2)

I was hoping you could do some introductions to some concepts before diving into the slides. I think it would help clarify things for us before we learn new content due to the large gap we have had since last talking about this subject. Also if you could continue some in class exercises and add examples to the slides that would be very helpful.

Is there a way we could get a C refresher. Are there any good resources to learn memory allocation etc? We did not have good practice with that in cso1

warmup assignment

C exercise

```
int array[4] = {10,20,30,40};
int *p;
p = &array[0];
p += 2;
p[1] += 1;
```

array $=$

A. compile or runtime error

B. $\{10,20,30,41\}$

C. $\{10,20,32,41\}$

D. $\{10,21,30,40\}$

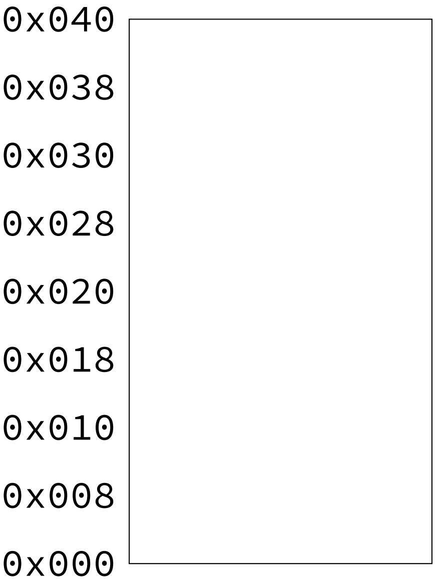
E. $\{12,21,30,40\}$

F. none of these

some avenues for review

- review CSO1 stuff
  - labs 9–12 (of last Spring)
  - `https://www.cs.virginia.edu/~jh2jf/courses/cs2130/spring2023/`
- exercises we've used in the past:
  - implement strsep library function
  - implement conversion from dynamic array to linked list

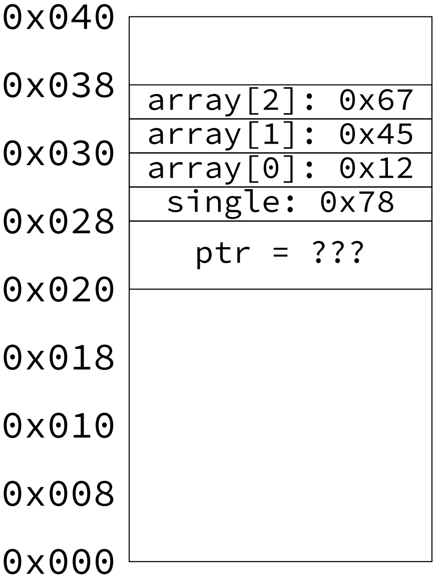
some pointer stuff



```
int array[3]={0x12,0x45,0x67};
int single = 0x78;
int *ptr;
```

some pointer stuff

```
int array[3]={0x12,0x45,0x67};
int single = 0x78;
int *ptr;
```



0x040

0x038

array[2]: 0x67

array[1]: 0x45

0x030

array[0]: 0x12

single: 0x78

0x028

ptr = ???

0x020

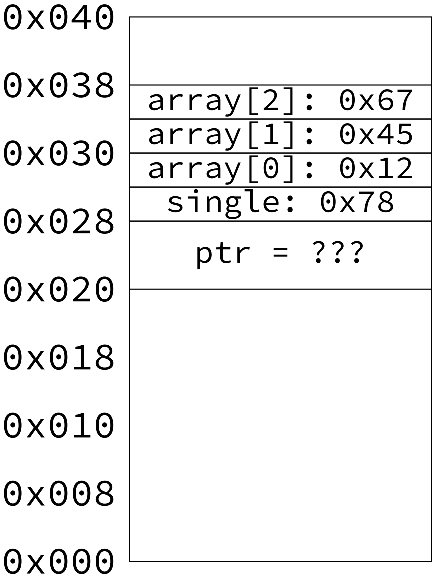
0x018

0x010

0x008

0x000

some pointer stuff



0x040

0x038

array[2]: 0x67

array[1]: 0x45

0x030

array[0]: 0x12

single: 0x78

0x028

ptr = ???

0x020

0x018

0x010

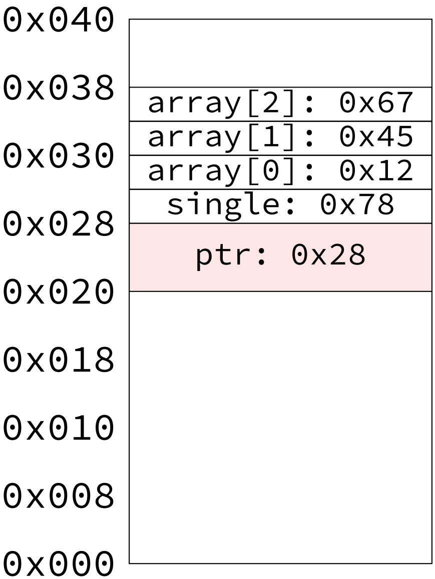
0x008

0x000

```
int array[3]={0x12,0x45,0x67};
int single = 0x78;
int *ptr;
```

~~*ptr = 0xAB;~~ compile error

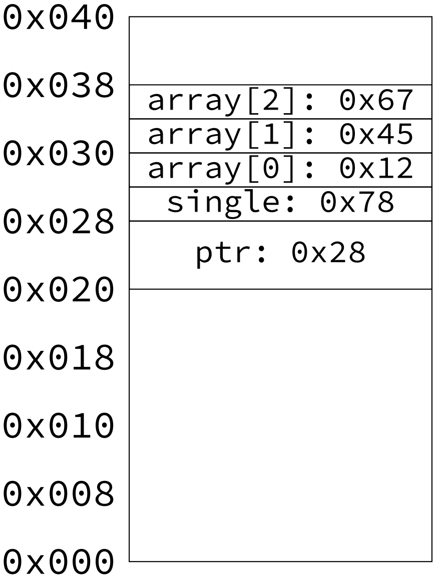
some pointer stuff



```
int array[3]={0x12,0x45,0x67};
int single = 0x78;
int *ptr;

ptr = &single;
ptr = (int*) 0x28;
```

addr. of single

some pointer stuff



0x040

0x038 | array[2]: 0x67

0x030 | array[1]: 0x45

array[0]: 0x12

single: 0x78

0x028 | ptr: 0x28

0x020

0x018

0x010

0x008

0x000

```
int array[3]={0x12,0x45,0x67};
int single = 0x78;
int *ptr;

ptr = &single;
ptr = (int*) 0x28;
```

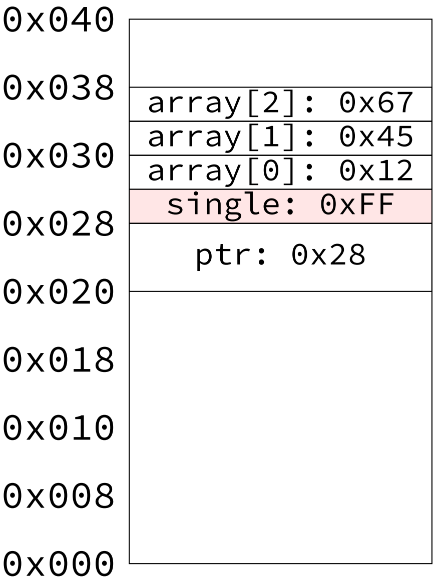
addr. of single

~~ptr = 0x28;~~ compile error

~~ptr = (int*) single;~~

pointer to unknown place

some pointer stuff



0x040

0x038

array[2]: 0x67

array[1]: 0x45

0x030

array[0]: 0x12

single: 0xFF

0x028

ptr: 0x28

0x020

0x018

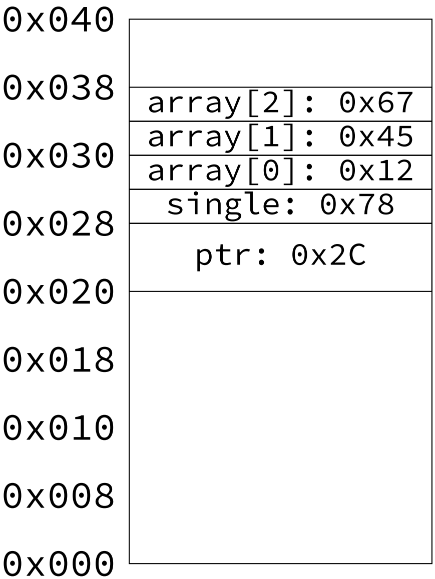
0x010

0x008

0x000

```
int array[3]={0x12,0x45,0x67};
int single = 0x78;
int *ptr;
ptr = &single;

*ptr = 0xFF;
```

some pointer stuff



0x040

0x038

array[2]: 0x67

array[1]: 0x45

0x030

array[0]: 0x12

single: 0x78

0x028

ptr: 0x2C

0x020

0x018

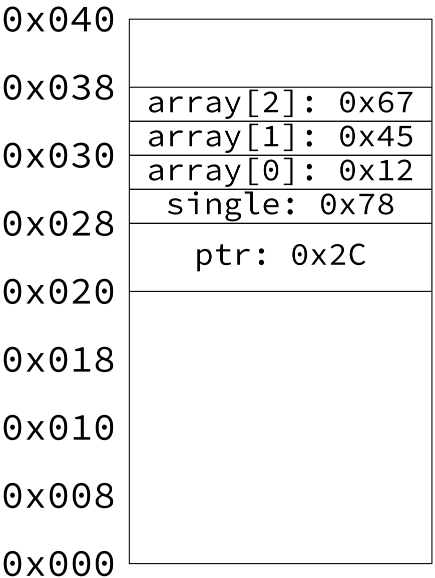
0x010

0x008

0x000

```
int array[3]={0x12,0x45,0x67};
int single = 0x78;
int *ptr;

ptr = array;
ptr = &array[0];
ptr = (int*) 0x2C;
```

some pointer stuff



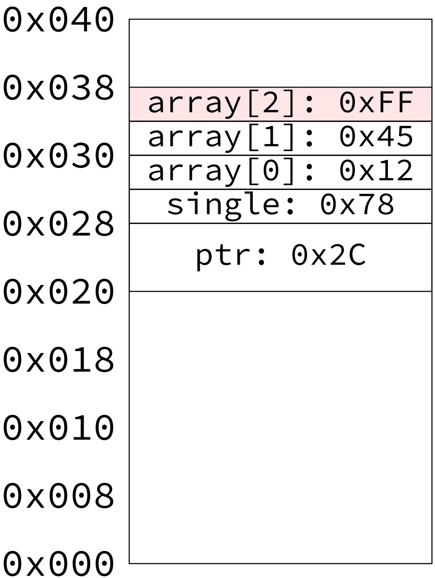
```
int array[3]={0x12,0x45,0x67};
int single = 0x78;
int *ptr;

ptr = array;
ptr = &array[0];
ptr = (int*) 0x2C;
```

~~`ptr = array[0];`~~ compile error

~~`ptr = (int*) array[0];`~~

pointer to unknown place

some pointer stuff



0x040

0x038

array[2]: 0xFF

array[1]: 0x45

0x030

array[0]: 0x12

single: 0x78

0x028

ptr: 0x2C

0x020

0x018

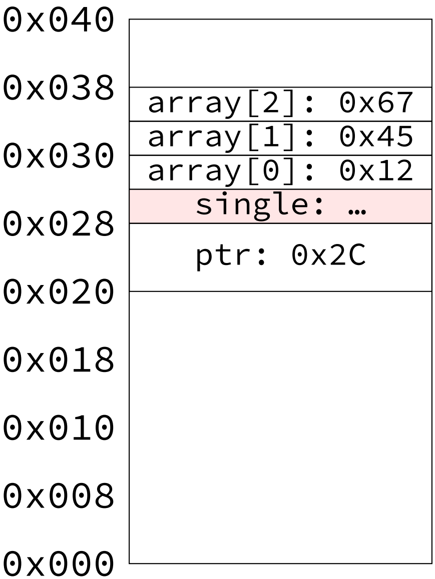
0x010

0x008

0x000

```
int array[3]={0x12,0x45,0x67};
int single = 0x78;
int *ptr;
ptr = &array[0];

ptr[2] = 0xFF;
*(ptr + 2) = 0xFF;

int *temp1; temp1 = ptr + 2;
*temp1 = 0xFF;

int *temp2; temp2 = &ptr[2];
*temp2 = 0xFF;
```

some pointer stuff



0x040

0x038

array[2]: 0x67

0x030

array[1]: 0x45

array[0]: 0x12

single: …

0x028

ptr: 0x2C

0x020

0x018

0x010

0x008

0x000

```
int array[3]={0x12,0x45,0x67};
int single = 0x78;
int *ptr;

void change_arg(int *x) {
    *x = compute_some_value();
}
…
change_arg(&single);
```

make

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

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

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

make rules

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

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

after colon: prerequisite(s)

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 -c main.c
```

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

after colon: prerequisite(s)

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 -c main.c
```

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

after colon: prerequisite(s)

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 -c main.c
```

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

after colon: prerequisite(s)

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 -c main.c
```

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

after colon: prerequisite(s)

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 -c main.c
```

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

after colon: prerequisite(s)

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 -c main.c
```

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

after colon: prerequisite(s)

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 -o program main.o extra.o

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

main.o: main.c main.h extra.h
►          clang -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 dependencies, 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?

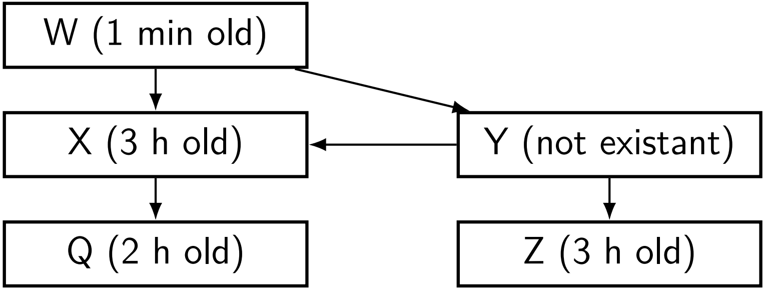
```
W: X Y
▶    buildW
X: Q
▶    buildX
Y: X Z
▶    buildY
```

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 ✓

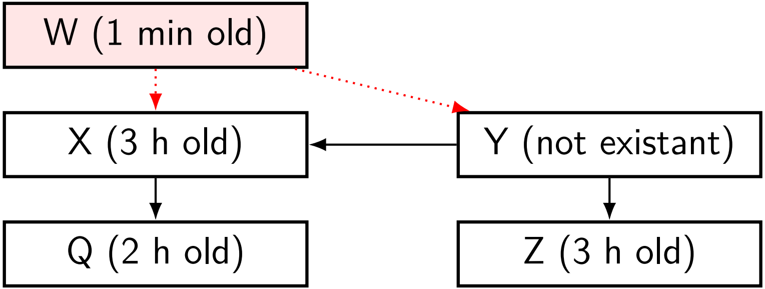
then build X if less recent than Q (yes) ✓

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 ✓

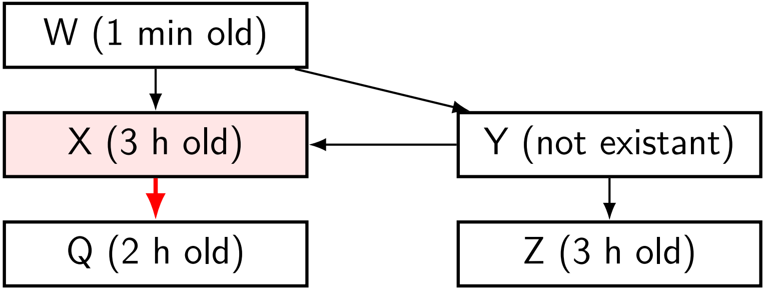
then build X if less recent than Q (yes) ✓

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) ✓

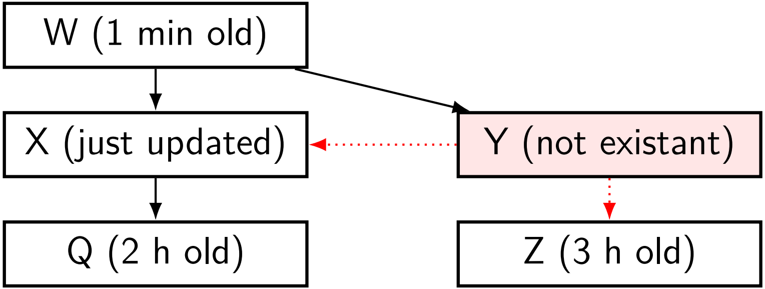
---

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 ✓

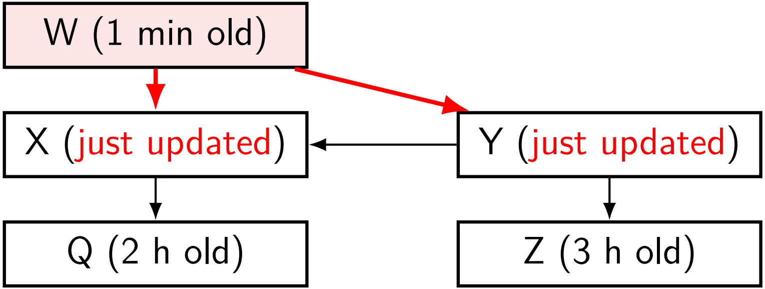
then build X if less recent than Q (yes) ✓

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



W (1 min old)

X (just updated)

Y (just updated)

Q (2 h old)

Z (3 h old)

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 -o program main.o extra.o

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

what if I want to change 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
```

variables/macros (2)

```
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 $<
```

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
```

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

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!)

writing Makefiles?

error-prone to automatically all .h dependencies

`-M` option to `gcc` or `clang`

- outputs Make rule
- ways of having make run this

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, …