last time

main themes

- automating building / incremental compilation
- sharing machines
- parallelism and concurrency
- networks
- modern processors

(start) logistics

two sections

Skadron's section [12:30p] and Reiss's [2p]

same labs, homeworks, [probably] quizzes

tentatively mostly separate lectures

(some exceptions when we have travel/other conflicts)

labs

attend lab in person and get checked off by TA, *or*

(most labs) submit something to submission site and we'll grade it

- submit to submission site? don't care if you attend the lab
- more strict about submissions without checkoffs
- in-person lab checkoff of incomplete lab at least 50% credit

some labs will basically require attendance

- or contact me for other arrangements if you can't (sick, etc.)
- logistically won't work otherwise — e.g. code review

if can't make lab in-person (example: sick)

- let me know, can arrange late/alternate checkoff

lab collaboration and submissions

please collaborate on labs!

when working with others on lab and submitting code files

please indicate who you worked with in those files

- via comment or similar

lab space

if labs are full, might kick out students from ‘wrong’ lab section

homeworks

several homework assignments

done individually

generally due on Fridays

(tentative dates on schedule)

homework/lab automatic testing

some homeworks/labs have automatic testing

with some delay after you submit

- usually 10s of minutes
- depending on assignment, number of submissions in queue
- if you submit very early, testing program might not be setup yet

when testing program doesn't understand/can't test something, left for manual grading ("not yet graded")

intention is that testing results are not surprises
if you did some manual testing (no hidden requirements, etc.)

if you think testing program made a mistake,
please submit regrade request

warmup assignment

first homework

write C function to split a string into array of strings

- with dynamic memory allocation

write C program to call function using input/command-line arguments

write Makefile for it (next topic, next week's lab)

quizzes

released evening after Thursday lecture

- starting *next* week

due 15 minutes before lecture on Tuesdays

about lecture and/or lab from the prior week

5–6ish questions

individual, open book, open notes, open Internet

quizzes and work/comments

quizzes will have place for comments/work

will be used to do grading

delay: about 1 week after quiz is due

please use so we can give partial credit

if you find possible error in quiz question
please make your best guess about was meant
and explain what you did in the comemnts

on help on quiz questions

I and the TAs won't answer quiz questions...

but we will answer questions about the lecture material, etc.

(and TAs (not you) are responsible for knowing
what they can't answer
but we'd prefer you don't try to test those limits)

going over past quizzes

have in past gone over quiz Qs in lecture

- either when a lot missed it or
- on request in lecture

also fine office hour/Piazza question

readings

in lieu of textbook, have readings

mostly written by Prof Tychnoveich (now at UIUC) with edits by me

on website; should be indicated with corresponding lecture

readings often link to alternative/supplemental readings on topic

lecture + assignment sync

generally:
quiz after lecture and/or lab coverage
labs after lecture coverage
homework after lab coverage

means homework (and sometimes quiz)
may be relatively delayed from lecture coverage

exams

1 final exam

- likely in-person
- see official exam schedule

no midterms — instead:

- quizzes count a lot

development enviroment

we will test via something like SSH into portal

- officially supported environment

no restrictions re: IDEs

- but make sure you test/know how to run from command line

many students had success with VSCode + its SSH support

some notes on VSCode

I don't use VSCode (I use vim via SSH+tmux...)

but many of our TAs do; their advice:...

use SSH support to run on portal (dept machine)

- tutorial in last semester's CS 2130 lab (linked off main course website)

install Microsoft's C/C++ extension

- set C standard in settings as 'gnu17' or similar

install Microsoft's Makefile Tools extension

getting help

office hours — calendar will be posted on website

- mix of in-person and remote, indicated on calendar
- remote OH will use Discord + online queue
- in-person OH may or may not — indicated on whiteboard, probably

Piazza

- use private questions if homework code, etc.

emailing me (preferably with '3130' in subject)

collaboration (1)

labs — you can/should work with other students

- everyone should understand the work submitted
- we may ask questions/etc. to check on occassion

homeworks — individual

- write your own code / do not share your code
- can ask/look up *conceptual* questions of others
- others includes other students, Q&A sites, code generation tools, etc.
- **cite** any sources you use (comments in code)

collaboration (2)

quizzes — individual

but open book+notes+etc.

can/should have help reviewing lecture/readings/etc.

- legitimate questions for office hours

don't ask other students, stack overflow, gen AI tools, etc. the quiz questions

don't try to find exactly the quiz question on stack overflow

feedback

anonymous feedback on Canvas

would appreciate feedback (esp. when I can do something)

(but not a good way to ask for regrades, etc.)

late policy

no late quizzes

one quiz dropped (unconditionally)

90% credit for 0–72 hours late homeworks

for labs that allow submission only

- lab submission due time is 11:59am the next day
- 90% credit for 0–24 hours late

no late lab checkoffs except by special arrangement

excused lateness

special circumstances?

- illness, emergency, etc.

contact me, we'll figure something out

please don't attend lab/etc. sick!

attendance

- I won't take attendance in lecture
- I will attempt to have lecture recordings
  - sometimes there may be issues with the recording

some avenues for review

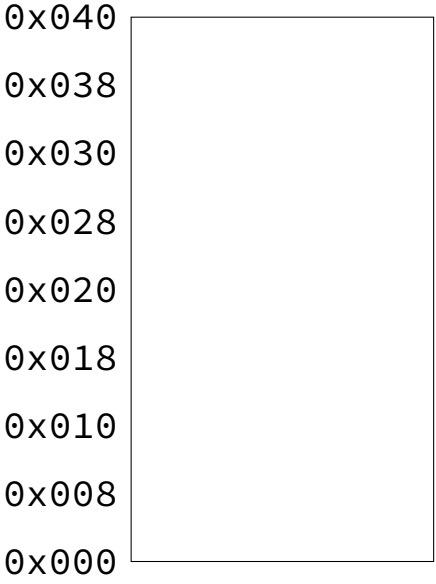
review CSO1 stuff

- labs 9–12 (of last Fall)
  `https://researcher111.github.io/uva-cso1-F23-DG/`

exercises we've used in the past:

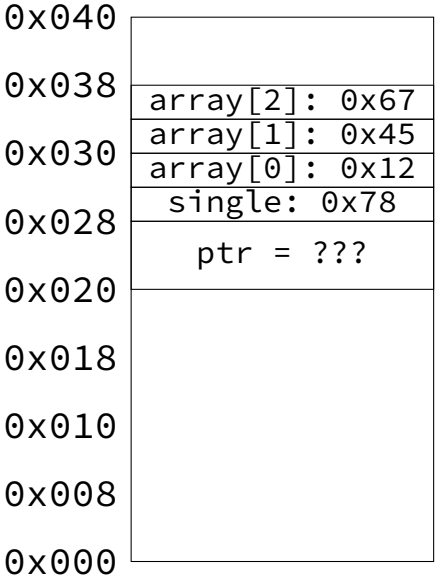
- implement strsep library function
- implement conversion from dynamic array to linked list

some pointer stuff



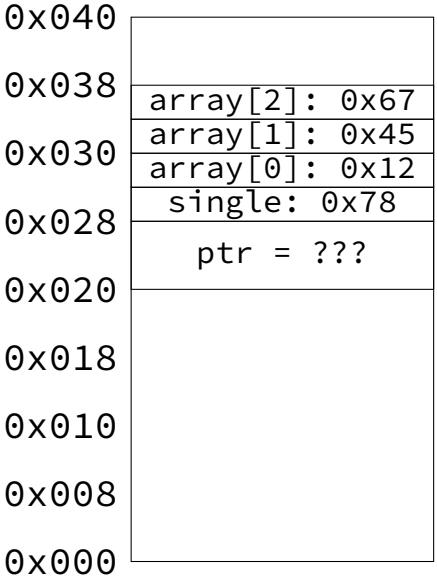

```
int array[3]={0x12,0x45,0x67};
int single = 0x78;
int *ptr;
```

some pointer stuff




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

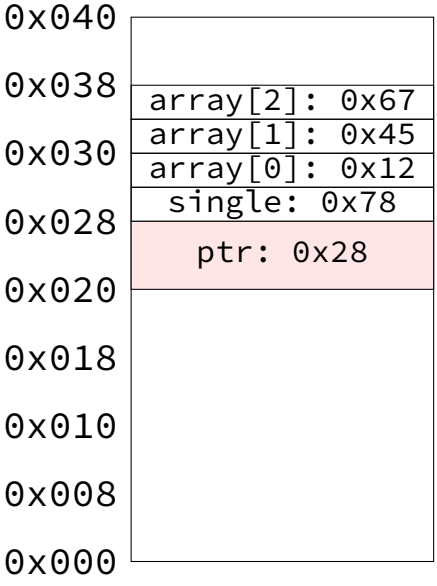
some pointer stuff




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

~~`*ptr = 0xAB;`~~ runtime error

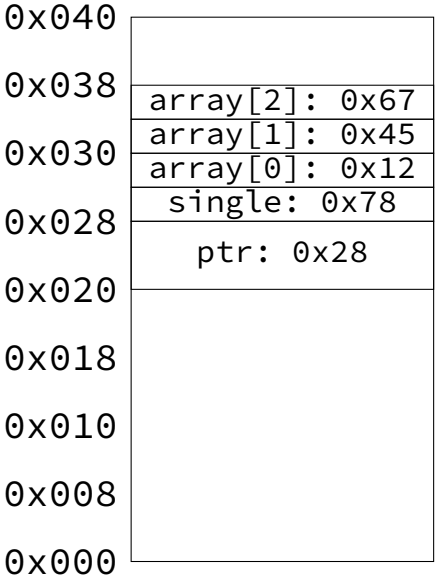
some pointer stuff




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

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

addr. of single

some pointer stuff



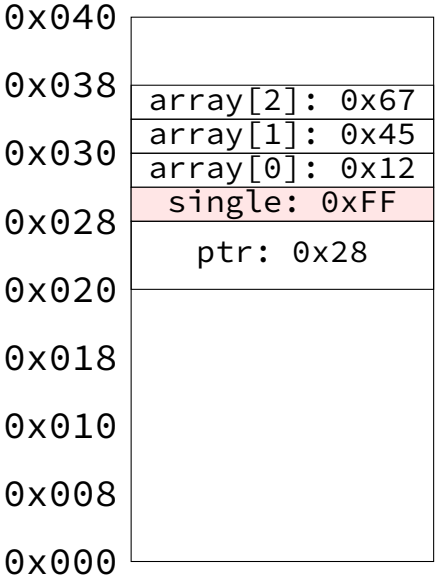

```
int array[3]={0x12,0x45,0x67};
int single = 0x78;
int *ptr;

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

~~`ptr = 0x28;`~~ compile error

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

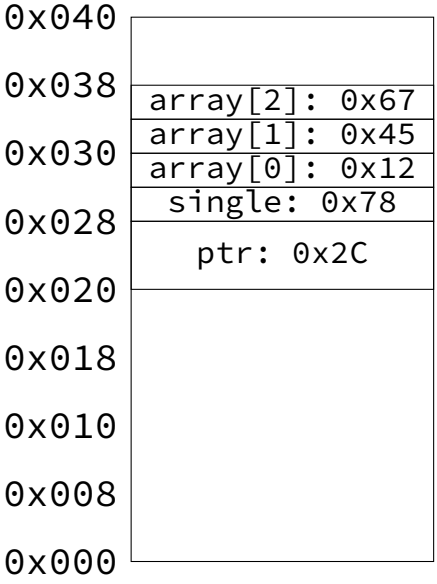
pointer to unknown place

some pointer stuff



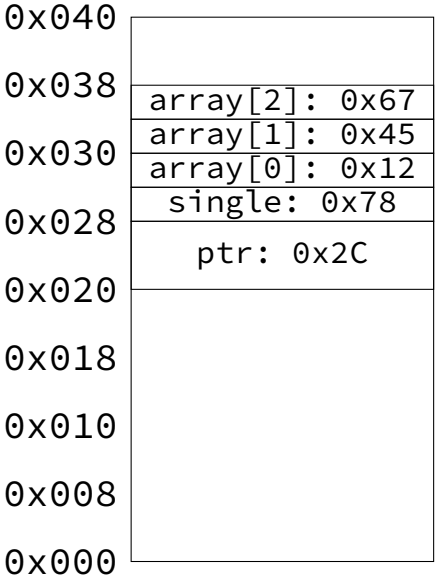

```
int array[3]={0x12,0x45,0x67};
int single = 0x78;
int *ptr;
ptr = &single;

*ptr = 0xFF;
```

some pointer stuff




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

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

some pointer stuff



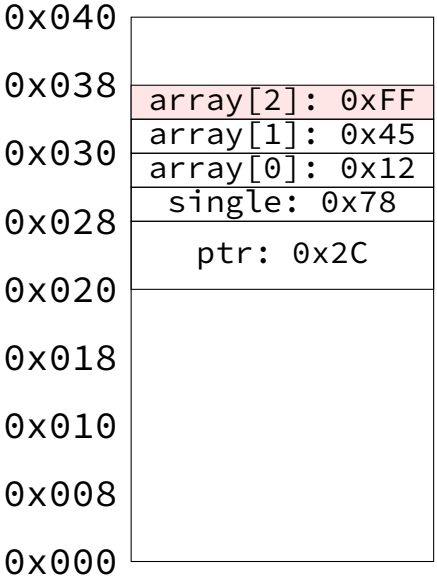

```
int array[3]={0x12,0x45,0x67};
int single = 0x78;
int *ptr;

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

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

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

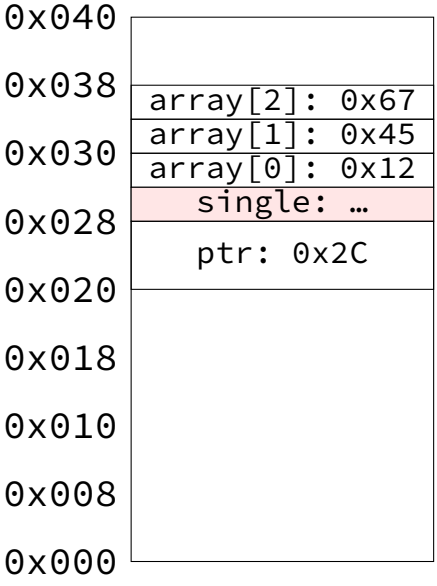
pointer to unknown place

some pointer stuff




```
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




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

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

some avenues for review

review CSO1 stuff

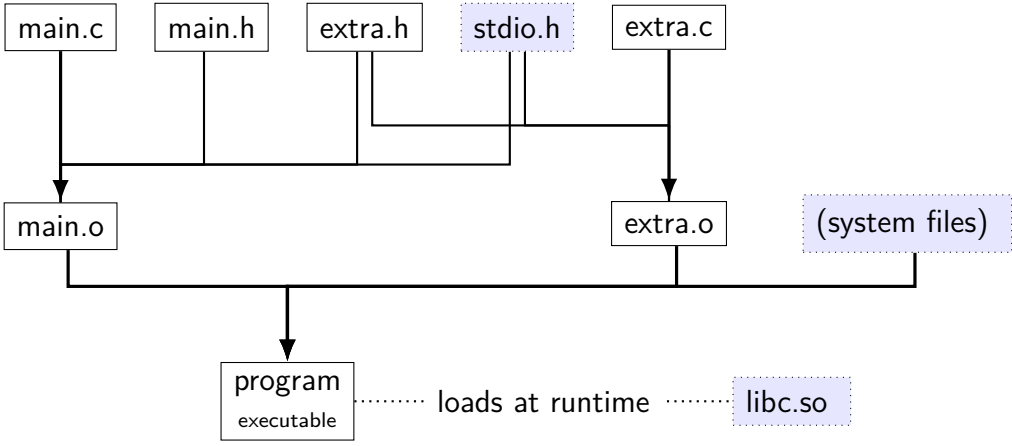
- labs 9–12 (of last Fall)
  `https://researcher111.github.io/uva-cso1-F23-DG/`

exercises we've used in the past:

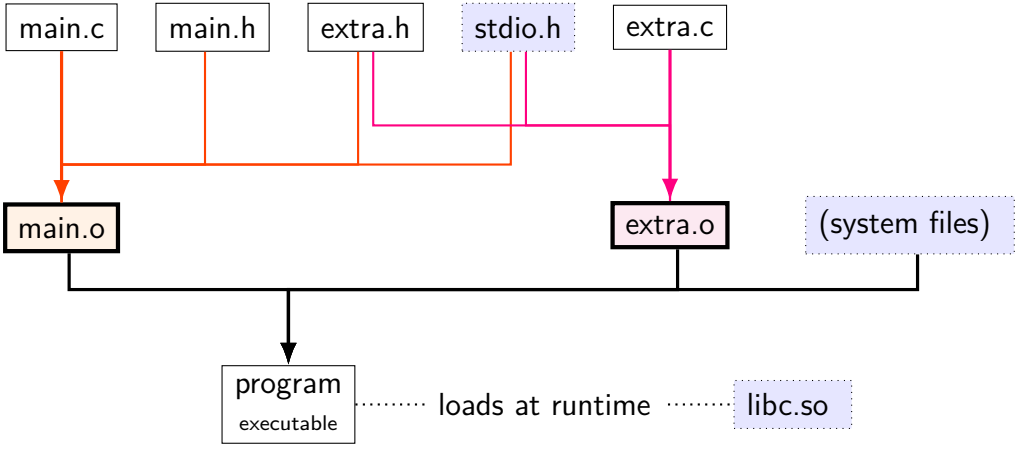
- implement strsep library function
- implement conversion from dynamic array to linked list

Skadron posted some C refreshers from the old OS course in Canvas > Files

files in building C programs [dynamic linking]

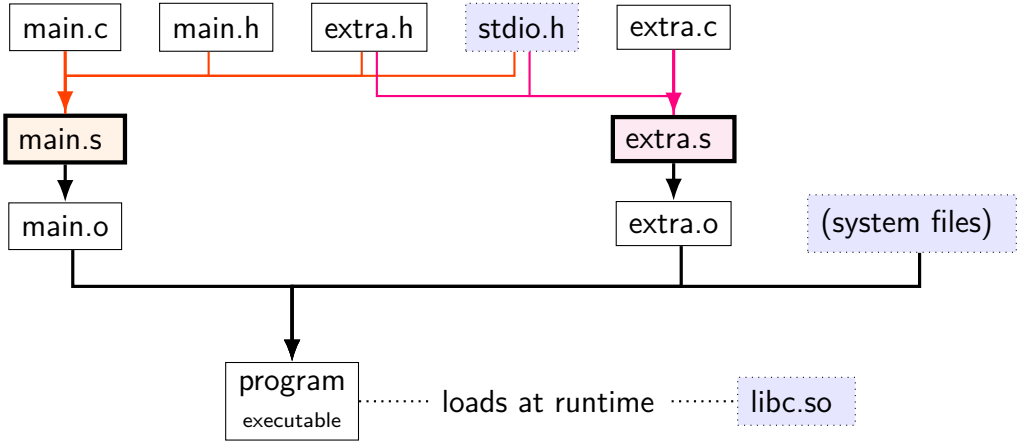


files in building C programs [dynamic linking]



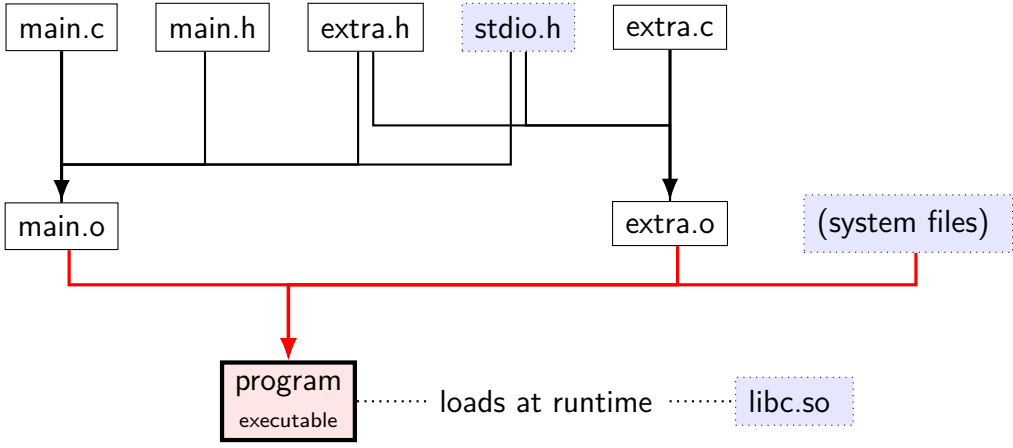

```
clang -c main.c
clang -c extra.c
```

files in building C programs [dynamic linking]



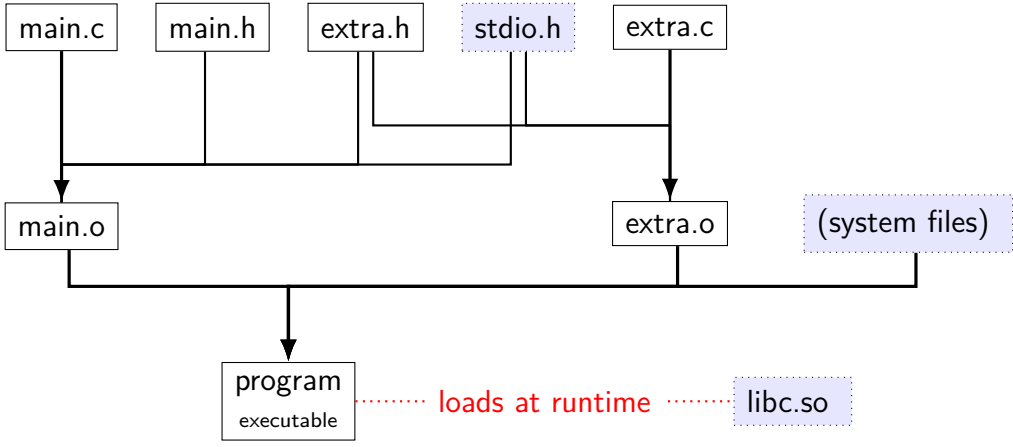

```
clang -S -c main.c
clang -S -c extra.c
```

files in building C programs [dynamic linking]



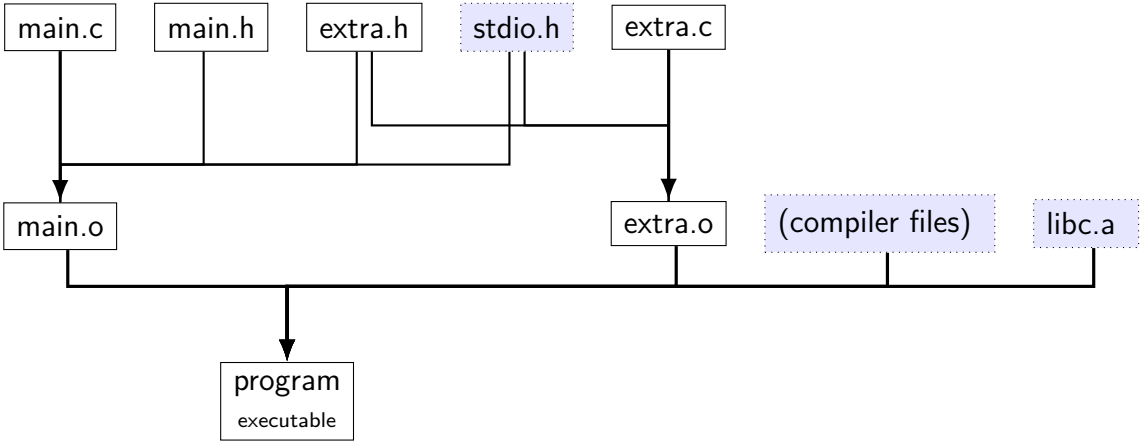

```
clang -o program main.o extra.o
```

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 …`

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`

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`

`-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

‘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`, `LDFALGS`, 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)

```
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 $<
```

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

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
```

aside: these rules work on GNU make (usual on Linux), but less portable than suffix rules.

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
▶       $(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!)

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

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