Makefiles & File I/O

Intro to Computer Systems, Fall 2022

Instructor: Travis McGaha

TAs:

Ali Krema	Andrew Rigas	Anisha Bhatia
Audrey Yang	Craig Lee	Daniel Duan
David LuoZhang	Eddy Yang	Ernest Ng
Heyi Liu	Janavi Chadha	Jason Hom
Katherine Wang	Kyrie Dowling	Mohamed Abaker
Noam Elul	Patricia Agnes	Patrick Kehinde Jr.
Ria Sharma	Sarah Luthra	Sofia Mouchtaris

Poll: Familiarity File I/O?

Upcoming Due Dates

- HW07 (Deque & RPN) Due Friday 11/11 @ 11:59 pm
- Check-in08: Due Monday 11/14 @ 4:59 pm
  - Releases Tomorrow
- HW08 (Disassembler) Due Friday 11/18 @ 11:59 pm
  - Should have everything you need after this lecture (or Monday's lecture)
- **Assignments will very likely take increasingly longer to complete. Please try to not let the work accumulate**

Lecture Outline

- **Makefiles**
- Command Line Args
- File I/O
- Binary files & Endianness

`make`

- `make` is a classic program for controlling what gets (re)compiled and how
  - Many other such programs exist (*e.g.*, `ant`, `maven`, IDE "projects")
- `make` has tons of fancy features, but only two basic ideas:
  1) Scripts for executing commands
  2) Dependencies for avoiding unnecessary work
- To avoid "just teaching `make` features" (boring and narrow), let's focus more on the concepts…

Building Software

- Programmers spend a lot of time "building"
  - Creating programs from source code
  - Both programs that they write and other people write

- Programmers like to automate repetitive tasks
  - Repetitive: gcc -Wall -g -std=c17 -o widget foo.c bar.c baz.c
    - Retype this every time: 
    - Use up-arrow or history:  (still retype after logout)
    - Have an alias or bash script: 
    - Have a Makefile:  (you're ahead of us)

"Real" Build Process

- On larger projects, you can't or don't want to have one big (set of) command(s) that are all run every time you change anything. To do things "smarter," consider:
  1) It could be worse: If `gcc` didn't combine steps for you, you'd need to preprocess, compile, and link on your own (along with anything you used to generate the C files)
  2) Source files could have multiple outputs (*e.g.*, `javadoc`). You may have to type out the source file name(s) multiple times
  3) You don't want to have to document the build logic when you distribute source code; make it relatively simple for others to build
  4) You don't want to recompile everything every time you change something (especially if you have 10^5-10^7 files of source code)

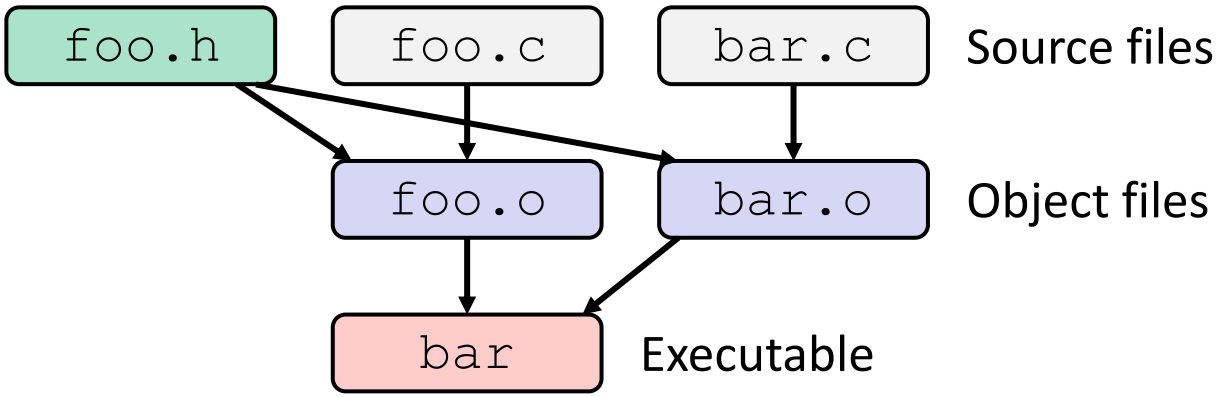
- A script can handle 1-3 (use a variable for filenames for 2), but 4 is trickier

Recompilation Management

- The "theory" behind avoiding unnecessary compilation is a *dependency dag* (**d**irected, **a**cyclic **g**raph)

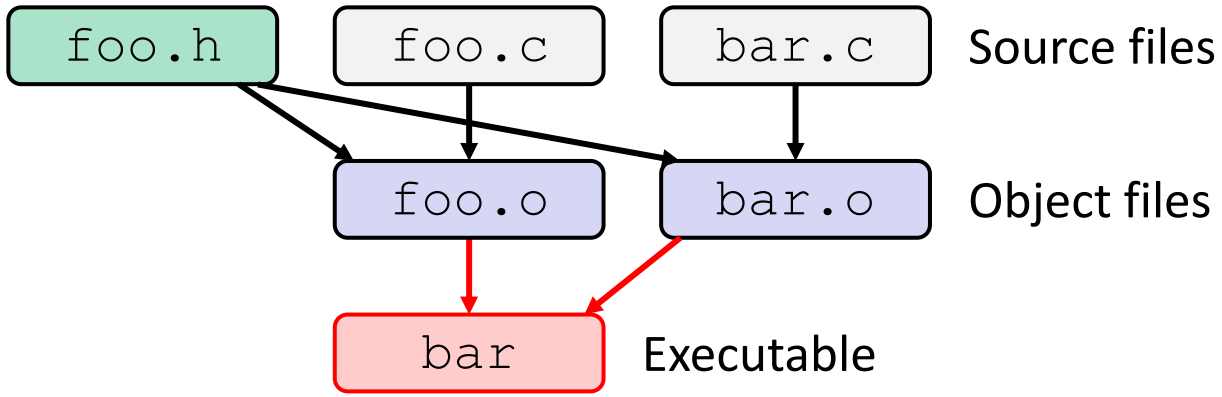
- To create a target t, you need sources $s_1, s_2, \ldots, s_n$ and a command c that directly or indirectly uses the sources
  - If t is newer than every source (file-modification times), assume there is no reason to rebuild it
  - Recursive building: if some source s_i is itself a target for some other sources, see if it needs to be rebuilt…
  - Cycles "make no sense"!

Theory Applied to C



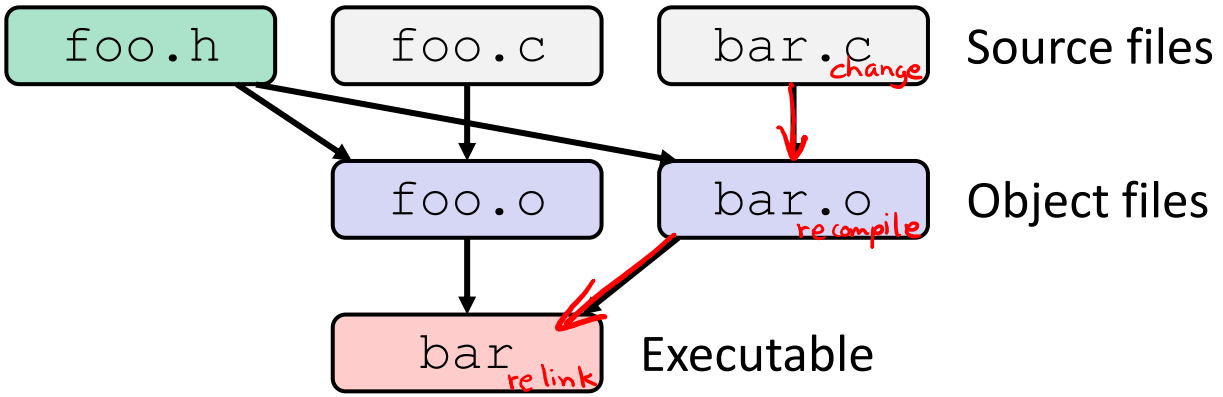
- Compiling a `.c` creates a `.o` – the `.o` depends on the `.c` and all included files (`.h`, recursively/transitively)

Theory Applied to C



- Compiling a `.c` creates a `.o` – the `.o` depends on the `.c` and all included files (`.h`, recursively/transitively)
- Creating an executable ("linking") depends on `.o` files

Theory Applied to C



- If one `.c` file changes, just need to recreate one `.o` file, maybe a library, and re-link
- If a `.h` file changes, may need to rebuild more
- Many more possibilities!

`make` Basics

- A makefile contains a bunch of triples:

```
target: sources
← Tab → command
```

  - Colon after target is *required*
  - Command lines must start with a **TAB**, NOT SPACES
  - Multiple commands for same target are executed *in order*
    - Can split commands over multiple lines by ending lines with '\'

- Example:

```
foo.o: foo.c foo.h bar.h
	gcc -Wall -o foo.o -c foo.c
```

Using `make`

```
bash$ make <target>
```

- Defaults:
  - If no `target` specified, will use the first one in the file
  - Will interpret commands in your default shell
- Target execution:
  - Check each source in the source list:
    - If the source is a target in the makefile, then process it recursively
    - If some source does not exist, then error
    - If any source is newer than the target (or target does not exist), run `command` (presumably to update the target)

"Phony" Targets

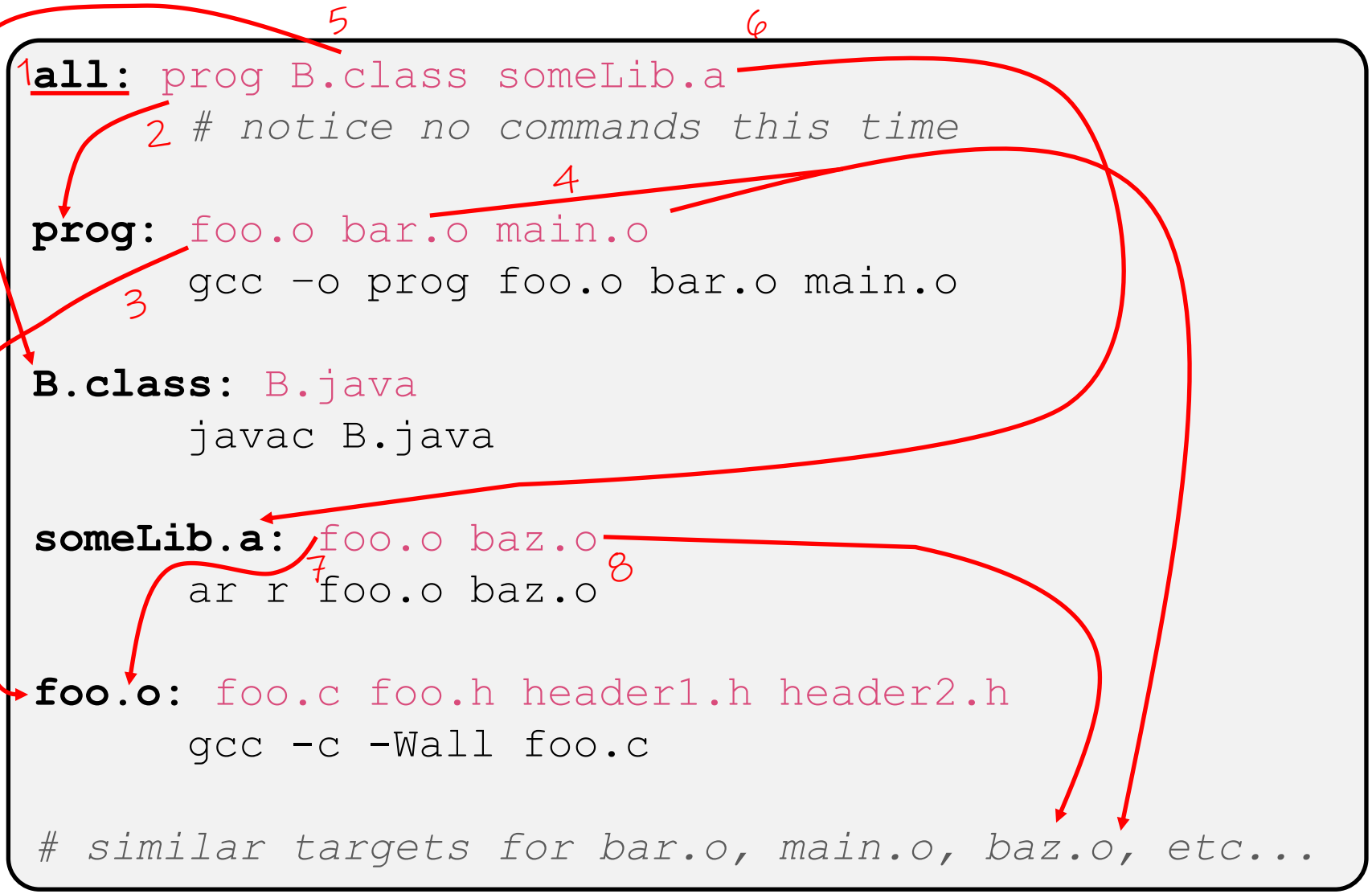
- A make target whose command does not create a file of the target's name (*i.e.*, a "recipe")
  - As long as target file doesn't exist, the command(s) will be executed because the target must be "remade"

- *e.g.*, target `clean` is a convention to remove generated files to "start over" from just the source

```
clean:
	rm foo.o bar.o baz.o widget *~
```

- *e.g.*, target `all` is a convention to build all "final products" in the makefile
  - Lists all of the "final products" as sources

"all" Example




```
all: prog B.class someLib.a
    # notice no commands this time

prog: foo.o bar.o main.o
    gcc -o prog foo.o bar.o main.o

B.class: B.java
    javac B.java

someLib.a: foo.o baz.o
    ar r foo.o baz.o

foo.o: foo.c foo.h header1.h header2.h
    gcc -c -Wall foo.c

# similar targets for bar.o, main.o, baz.o, etc...
```

Makefile Writing Tips

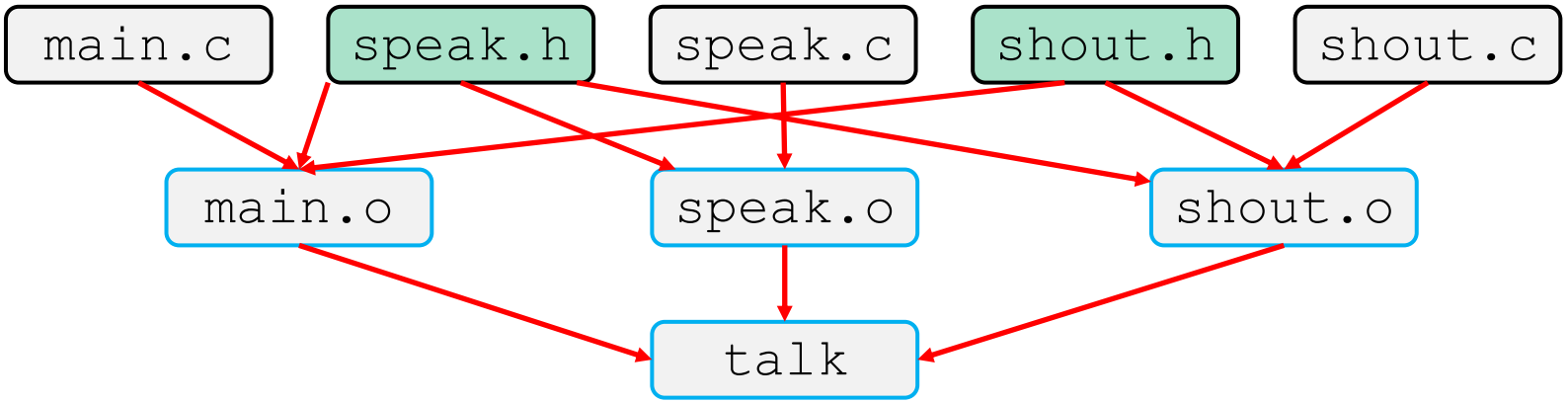
- *When creating a Makefile, first draw the dependencies!!!!*

- C Dependency Rules:
  - `.c` and `.h` files are never targets, only sources.
  - Each `.c` file will be compiled into a corresponding `.o` file
    - Header files will be implicitly used via `#include`
  - Executables will typically be built from one or more `.o` file

- Good Conventions:
  - Include a `clean` rule
  - If you have more than one "final target," include an `all` rule
  - The first/top target should be your singular "final target" or `all`

Writing a Makefile Example

- "`talk`" program (find files on web with lecture slides)



main.c

```
#include "speak.h"
#include "shout.h"

int main(int argc, char** argv) {…
```

speak.c

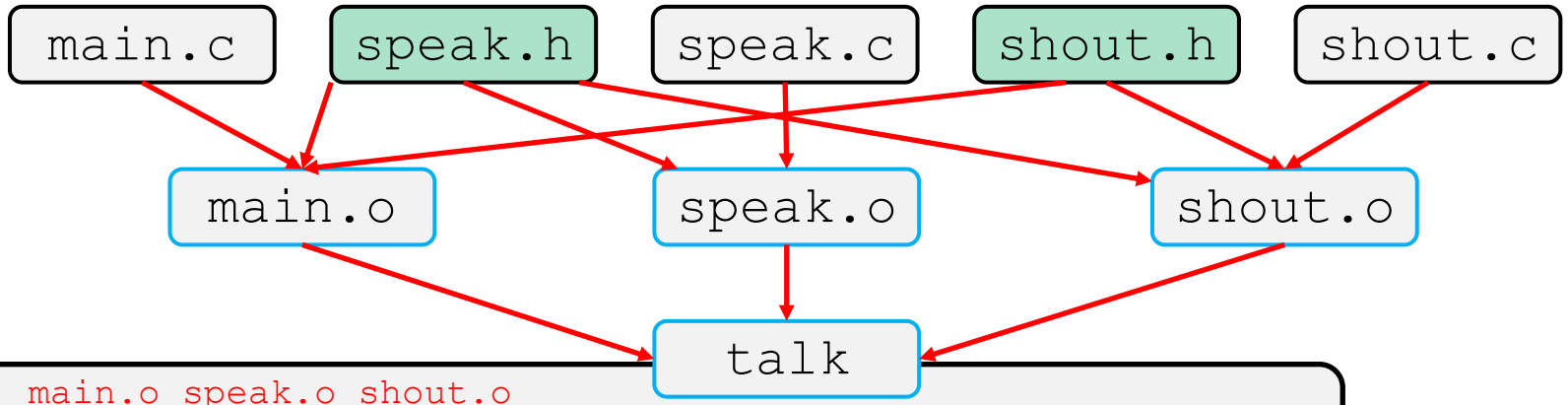
```
#include "speak.h"
...
```

shout.c

```
#include "speak.h"
#include "shout.h"
...
```

Writing a Makefile Example

- "`talk`" program (find files on web with lecture slides)




```
talk: main.o speak.o shout.o
        gcc -g -Wall -o talk main.o speak.o shout.o

main.o: main.c speak.h shout.h
        gcc -g -Wall -c main.c

speak.o: speak.c speak.h
        gcc -g -Wall -c speak.c

shout.o: shout.c speak.h shout.h
        gcc -g -Wall -c shout.c

clean:
        rm talk *.o
```

`make` Variables

- You can define variables in a makefile:
  - All values are strings of text, no "types"
  - Variable names are case-sensitive and can't contain '`:`', '`#`', '`=`', or whitespace
- Example:

```
CC = gcc
CFLAGS = -Wall -std=c17
OBJFILES = foo.o bar.o baz.o
widget: $(OBJFILES)
	$(CC) $(CFLAGS) -o widget $(OBJFILES)
```

- Advantages:
  - Easy to change things (especially in multiple commands)
    - It's common to use variables to hold lists of filenames
  - Can also specify/overwrite variables on the command line: (*e.g.*, `make CC=clang CFLAGS=-g`)

Revenge of the Funny Characters

- Special variables:
  - **$@** for target name
  - **$^** for all sources
  - **$<** for left-most source
  - Lots more! – see the documentation

- Examples:

```
# CC and CFLAGS defined above
widget: foo.o bar.o
        $(CC) $(CFLAGS) -o $@ $^
foo.o: foo.c foo.h bar.h
        $(CC) $(CFLAGS) -c $<
```

And more…

- There are a lot of "built-in" rules – see documentation
- There are "suffix" rules and "pattern" rules
  - Example:

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

- Remember that you can put *any* shell command – even whole scripts!
- You can repeat target names to add more dependencies
- Often this stuff is more useful for reading makefiles than writing your own (until some day…)

Lecture Outline

- Makefiles
- **Command Line Args**
- File I/O
- Binary files & Endianness

Executing a C program

- The Command Line: receives commands from a user in the form of lines of text

- The "Terminal" in ubuntu runs a command line for us to run various commands.
  - Commands you may have seen before:
    - `$ cd ~/Desktop`
    - `$ tar -xvf hw6.tar`
    - `$ java -jar Pennsim.jar`

$ often used to indicate that the terminal is ready for another command

Following strings specify the inputs/options for the command/program

First string specifies the program/command name

  - Can also run user written programs:
    - `$ ./my_program`
    - `$ ./test_suite`

C Syntax: `main`

Advantages: Simple (terminal takes chars) & flexible (can take in any number of args)

Disadvantages: Input checking & data conversion needed.

- To get command-line arguments in `main`, use:

```
int main(int argc, char* argv[])
```

= char** argv

C String

- What does this mean?
  - `argc` contains the number of strings on the command line (the executable name counts as one, plus one for each argument).
  - `argv` is an array containing *pointers* to the arguments as strings (more on pointers later)

Arrays in C are not objects, don't have '.length'

Should we treat as string or number?

- Example: `$ ./my_program hello 87`
  - `argc = 3`
  - `argv[0]="./my_pogram", argv[1]="hello", argv[2]="87"`

Command Line Args Demo

- Code on the course website: **`print_args.c`**

Lecture Outline

- Makefiles
- Command Line Args
- **File I/O**
- Binary files & Endianness

Files Purpose

- So far, we have talked about memory, which is often a type of "volatile storage"
  - Volatile storage: requires power to maintain the data values. Loss of power = loss of data
  - Program memory is also deallocated at the end of the program. To get those values again, the program to compute them must be run again

- Files: a type of permanent/long-term "non-volatile" storage
  - Non-volatile: retains data when the power is turned off
  - Long-term: holds data that exists beyond the lifetime of a program

File: Examples

- You've already been interacting with files (maybe not through programs yet though)

- Program files: (.c/.asm/.obj/etc.) are modified and persist between program executions.
  - While these contain information about how a program is setup, it doesn't contain all of program memory, which will change as the program executes

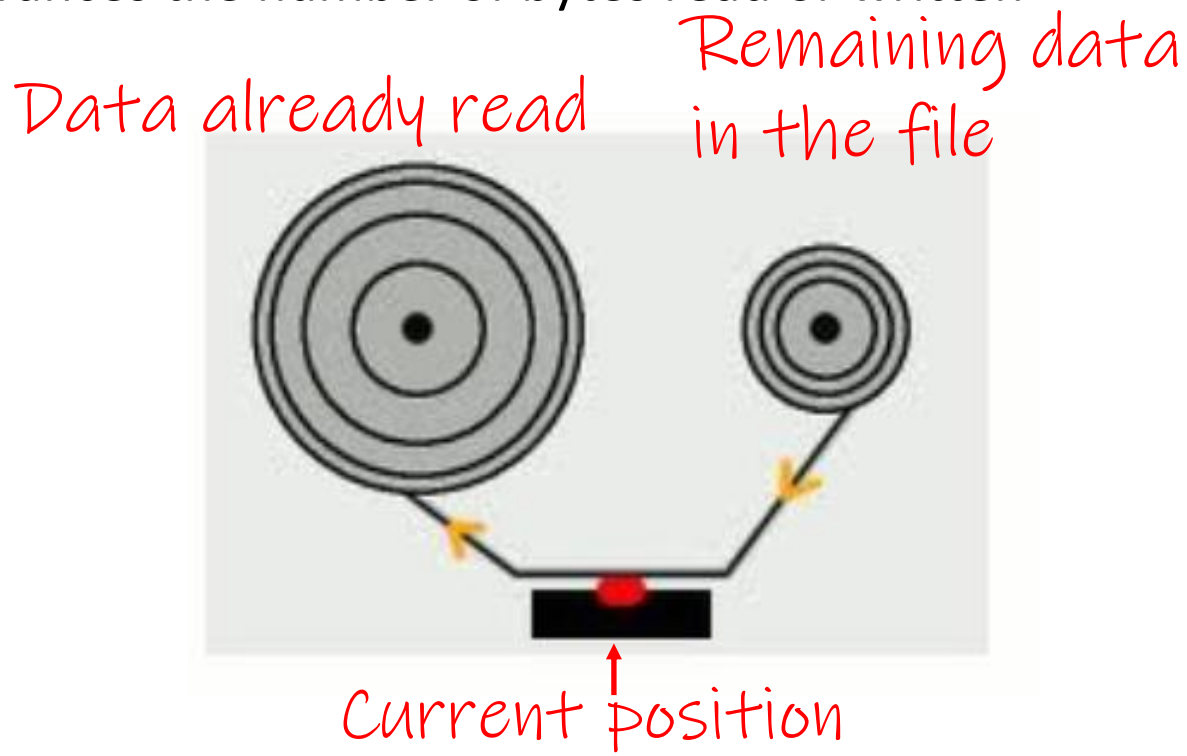
- Editors (sublime/IntelliJ/vim/PowerPoint/etc.) are programs that read and modify files based on user input

File Interface Metaphor: Tape Drives

- Programs usually interact with files following a similar file interface:
- Functions that model a sequential access device like magnetic tape drives

File Interface Metaphor: Tape Drives

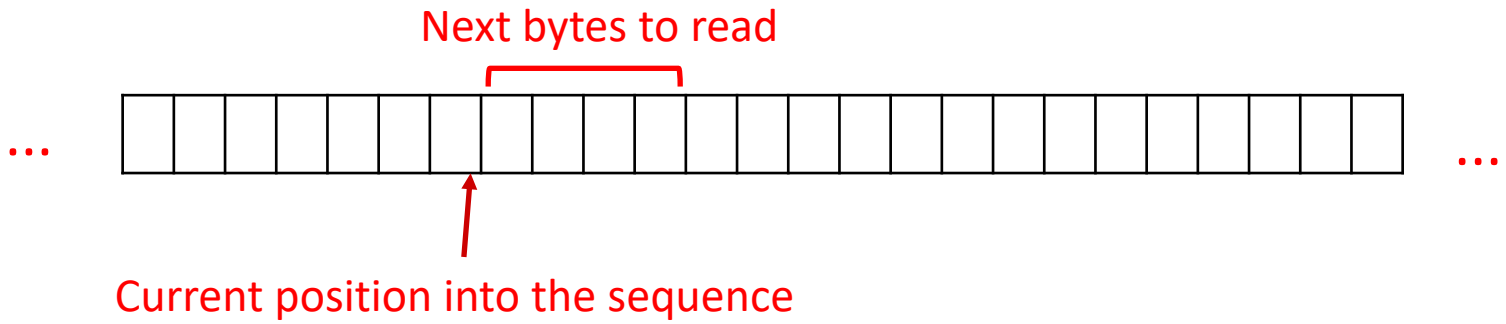
- Open a file for reading or writing
  - (usually starting at the beginning of the file)
- Read/Write the file
  - Each read/write advances the number of bytes read or written
- Rewind: start at beginning again
- others



File Interface Metaphor: Streams

- Another (more modern) abstraction is to think of I/O in terms of "streams"

- Stream:
  - A **sequence** of bytes that flows **to** and **from** a device
  - We do not have access to whole file at once (some files are too big to fit inside of memory easily)



This ends up working sort of like an iterator over the file.
Where we can read current data, and/or insert new data

C Stream Functions (1 of 3)

- Some stream functions (complete list in `stdio.h`):

Returns NULL on error

Do we create a new file if it doesn't exist?

Are we reading the file?

Are we writing the file?

- `FILE* fopen(filename, mode);`
  - Opens a stream to the specified file in specified file access mode
- `int fclose(stream);`

a **`FILE*`** returned by fopen

  - Closes the specified stream (and file)
- `int fprintf(stream, format, ...);`
  - Writes a formatted C string
    - Like `printf(...);` but for files
- `int fscanf(stream, format, ...);`
  - Reads data and stores data matching the format string

C Stream Functions (2 of 3)

- Some stream functions (complete list in `stdio.h`):

Pointer to the start of elements in memory to write to file

Size of an element

Number of elements

FILE*

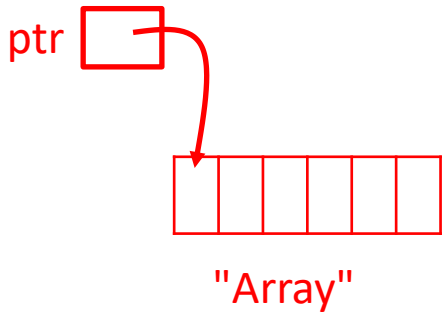
```
size_t fwrite(ptr, size, count, stream);
```

- Writes an array of *count* elements of *size* bytes from *ptr* to *stream*

```
size_t fread(ptr, size, count, stream);
```

- Reads an array of *count* elements of *size* bytes from *stream* to *ptr*

Returns number of elements actually read/written



C Stream Functions (3 of 3)

- Some stream functions (complete list in `stdio.h`):

```
int fgetc(FILE* stream);
```

  - Reads one character (one byte)

```
int fputc(char c, FILE* stream);
```

  - Prints one character (one byte)

```
char* fgets(char* str, int n, FILE* stream);
```

  - Reads a string from the strearm into the string **`str`**. Reads N characters or until a newline character (or end of file).

C Stream Error Checking/Handling

- Some error functions (complete list in `stdio.h`):
  - `int ferror(stream);`
    - Checks if the error indicator associated with the specified stream is set
  - `int clearerr(stream);`
    - Resets error and EOF indicators for the specified stream
  - `void perror(message);`
    - Prints `message` followed by an error message related to `errno` to `stderr`

Global variable

Extra information

Terminal input/output

- C defines three file streams for terminal input/output
  - Defined in <stdio.h>
  - Opened at program start by default

  - **`stdin`**: standard input (console)
  - **`stdout`**: standard output (console, for normal output)
  - **`stderr`**: standard error (console, for error output)

- The following are equivalent:

```
printf("Hello World!\n");
```

```
fprintf(stdout, "Hello World!\n");
```

Demo: copy file program

- Well Written file posted on website as **`copy_file.c`**

- Things to do when dealing with C stream I/O:
  - Eventually we will hit the end of file, need to handle that
  - Must ask for an amount of bytes/elements to be read.
    Best practice is to request for a chunk of bytes/elements at a time (e.g. 100 or so)

Other Functions

- Many other functions not covered in lecture (not enough time). Feel free to look up others and use them

- Some examples:
  - `int feof(FILE* f);`
    - check for end of file
  - `void rewind(FILE *f);`
    - start back at the beginning of file
  - `long ftell(FILE* f);`
    - gives the current position into the file
  - `int fseek(FILE* f, long offset, int whence);`
    - Reposition where we are in the file

Lecture Outline

- Makefiles
- Command Line Args
- File I/O
- **Binary files & Endianness**

Binary files & Serialization

- So far this lecture has implicitly assumed we are working with files that hold text (characters)
- Binary files also exist where data isn't stored as characters. (.obj files are an example)
- Some data/data-structures make more sense to be stored in binary through a process called **serialization**.

Serialization Example:

- Posted on course website
  - read_floats.c
  - write_floats.c

- Notes:
  - Don't have to read/write an array, can read/write only one "element"
  - Trying to open these files in an editor will not be readable

Endianness

- In other architectures, there is one byte at each address location
  - For multi-byte data, how do we order it in memory?
  - Data should be kept together, but what order should it be?
  - Example, store the 4-byte (32-bit) int:
    0x A1 B2 C3 D4

- The order of the bytes in memory is called endianness
  - Big endian vs little endian

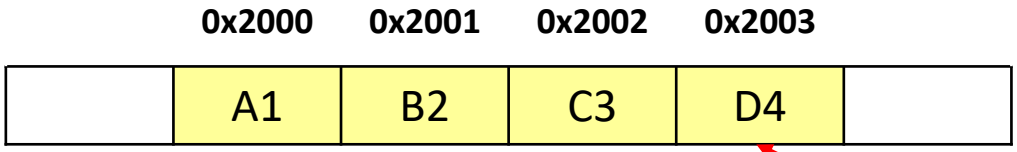
Endianness

- Consider our example 0x A1 B2 C3 D4

Most significant Byte

Least significant Byte

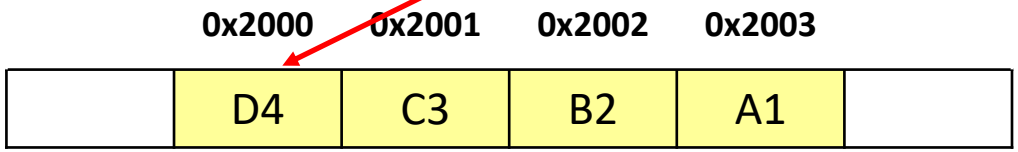
- Big endian
  - Least significant byte has highest address
  - Looks the most like what we would read
  - The standard for storing information on files/the network



- Little Endian
  - Least significant byte has lowest address
  - What your VM probably uses

Least significant Byte

Note how the hex digits within a byte are still in the same order



Poll Everywhere

pollev.com/tqm

- If we have the following int which is four bytes. on a big-endian machine, how would this be stored in memory?

```
int num = 0xCADEDADA;
```

A.

CA	DE	DA	DA

B.

DA	DA	DE	CA

C.

AC	ED	AD	AD

D.

AD	AD	ED	AC

E. I'm not sure

Poll Everywhere

pollev.com/tqm

- If we have the following int which is four bytes. on a big-endian machine, how would this be stored in memory?

```
int num = 0xCADEDADA;
```

A.

CA	DE	DA	DA

B.

DA	DA	DE	CA

C.

AC	ED	AD	AD

D.

AD	AD	ED	AC

E. I'm not sure

Endianness: Why it matters

- Since machines may store things in different byte orderings, it causes problems when they share files or communicate over the network.
- A standard ordering is used for storing binary data, big endian (often called Network ordering).
- Need to make sure that we store bytes in network byte ordering when we serialize data

Endianness functions

- There are some functions out there that convert byte orderings
  - `htons()` -> Host to Network short (16 bits)
    - Converts from Host byte ordering to network byte ordering
  - `ntohs()` -> Network to Host short (16 bits)
    - Converts from network byte ordering to host byte ordering

- "Network byte order" is big endian. Your "host" machine is little endian

- More info in <arpa/inet.h>
  - Variants also exist for 32 bit and 64 bit conversion