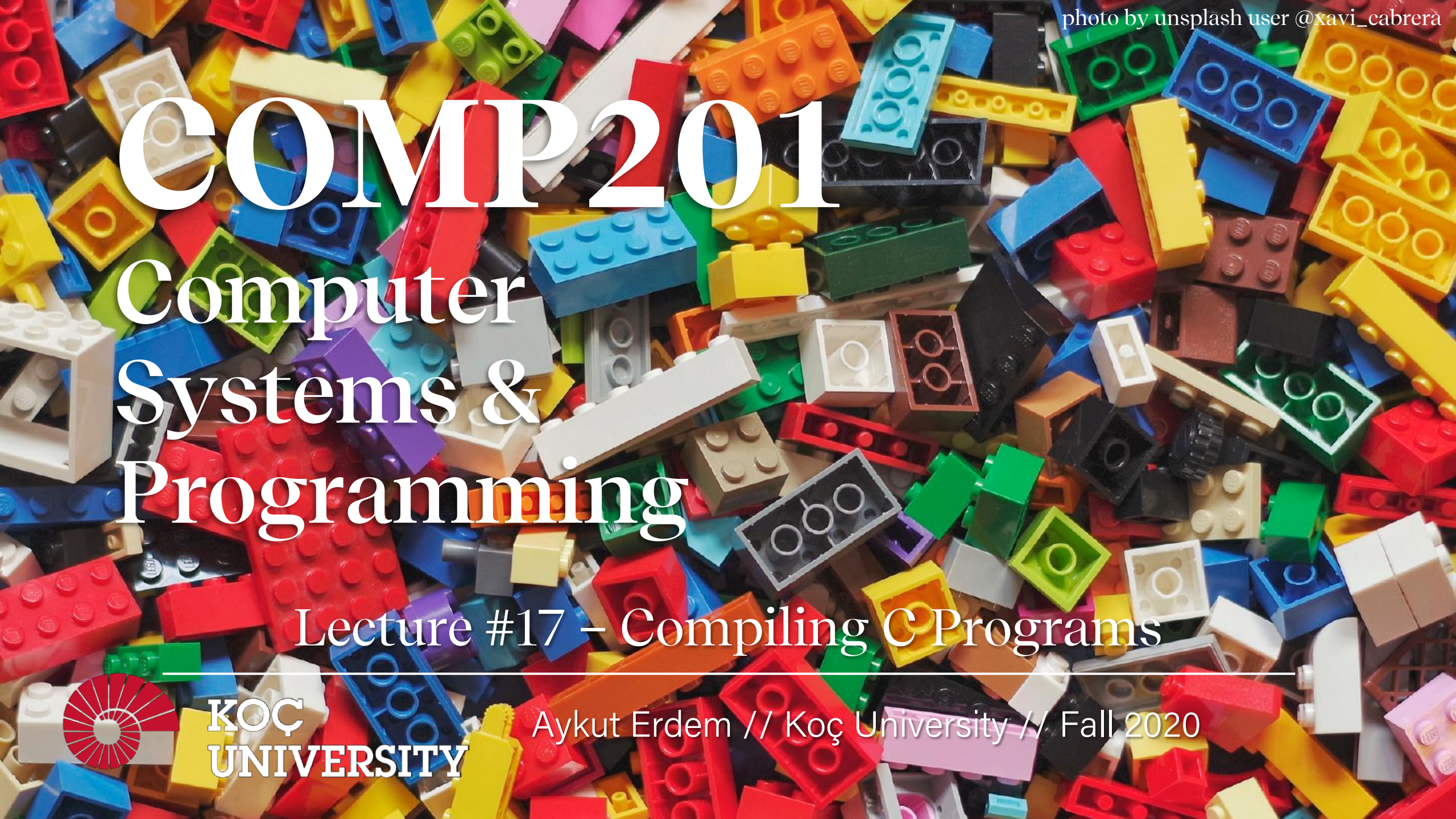



# COMP201

# Computer Systems & Programming

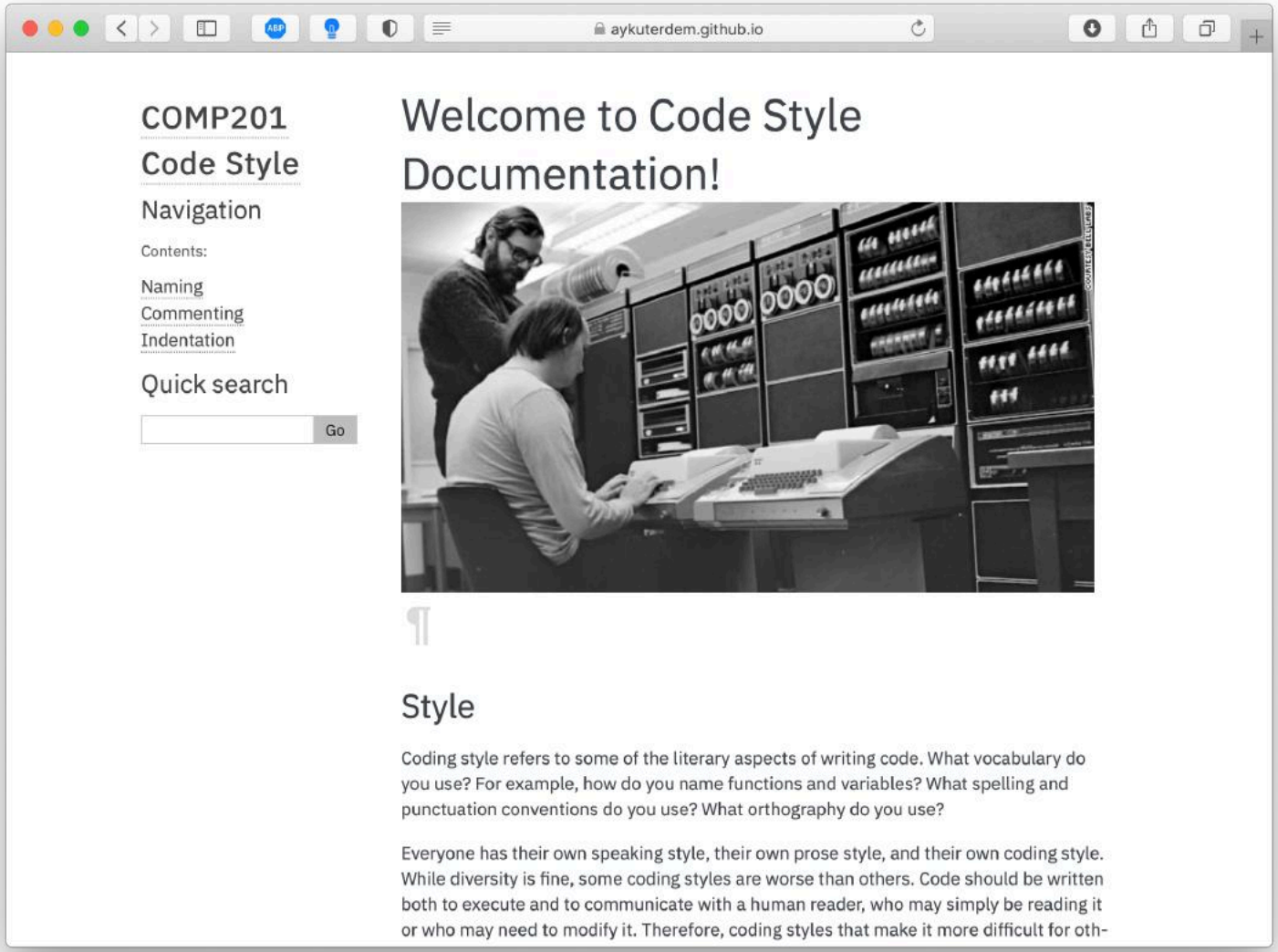
## Lecture #17 – Compiling C Programs

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2020

# COMP201 Coding Style Guide for C Programming

- Our guide serves as a brief introduction to C coding style.
- Following an formal style is very important to write a clean and easy to read code.
- There are many standards out there!




https://aykuterdem.github.io/classes/comp201/code-style/html/index.html

# Recap

- `struct`
- Generic stack

# Plan for Today

- What really happens in GCC?
- Make and Makefiles

**Disclaimer:** Slides for this lecture were borrowed from
—Gabbi Fisher and Chris Chute's Stanford CS107 class

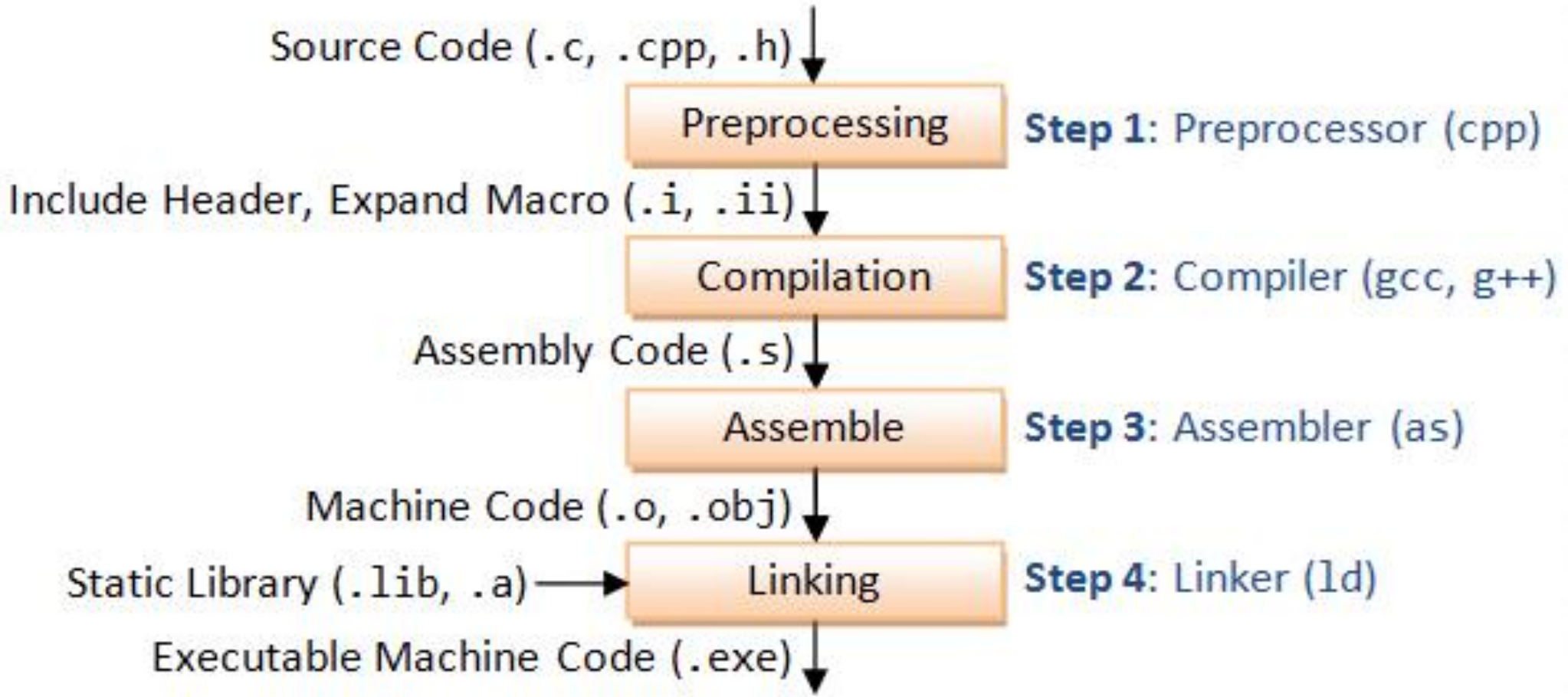
# Lecture Plan

- What really happens in GCC?
  - The Preprocessor
  - The Compiler
  - The Assembler
- Make and Makefiles

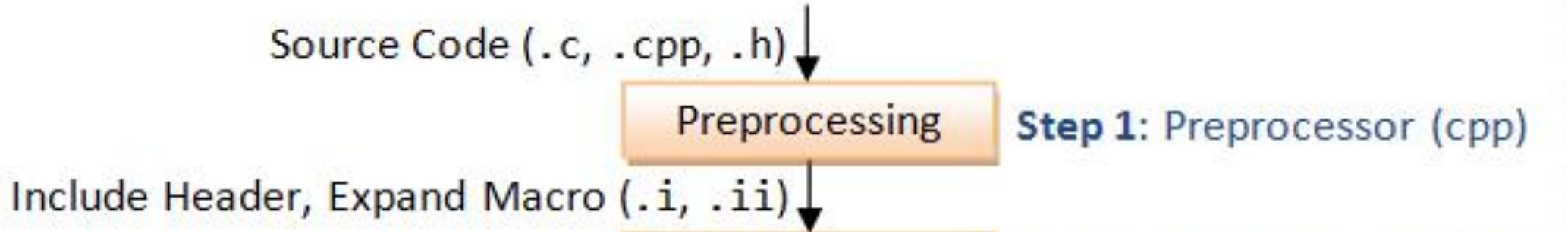
# Compiling a C program with GCC

```
gcc -g -O0 hello.c -o hello
```

# The GNU Compiler Collection (GCC)



# The GNU Compiler Collection (GCC)



# The Preprocessor

`#define`

`#include`

# The Preprocessor – Object Macros

```
#define BUFFER_SIZE 1024

foo = (char *) malloc (BUFFER_SIZE);
```

# The Preprocessor – Object Macros

```
#define BUFFER_SIZE 1024

foo = (char *) malloc (BUFFER_SIZE);

=> foo = (char *) malloc (1024);
```

# The Preprocessor – Function Macros

```
#define min(X, Y) ((X) < (Y) ? (X) : (Y))


y = min(1, 2);
```

# The Preprocessor – Function Macros

```
#define min(X, Y) ((X) < (Y) ? (X) : (Y))

y = min(1, 2);

=> y = ((1) < (2) ? (1) : (2));
```

# The Preprocessor – Imports

```
#include
```

# The Preprocessor – Imports

header.h

```
char *test(void)
```

program.c

```
#include “header.h”

int x;

int main(int argc, char *argv[]) {
  puts(test());
}
```

# The Preprocessor – Imports

header.h

```
char *test(void)
```

program.c

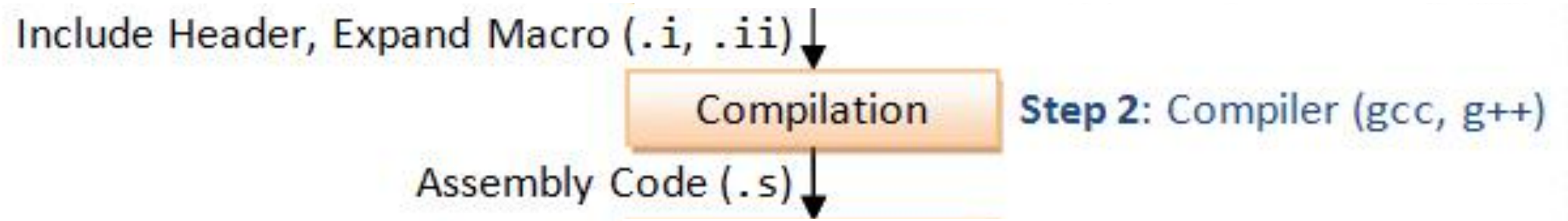
```
char *test(void);

int x;

int main(int argc, char *argv[]) {
  puts(test());
}
```

# The Preprocessor – Demo

```
gcc -E -o hello.i hello.c
```

Preprocess hello.c, store output in hello.i

# The GNU Compiler Collection (GCC)



# The Compiler

- They're too complicated to explain in 5 minutes.

¯_(ツ)_/¯

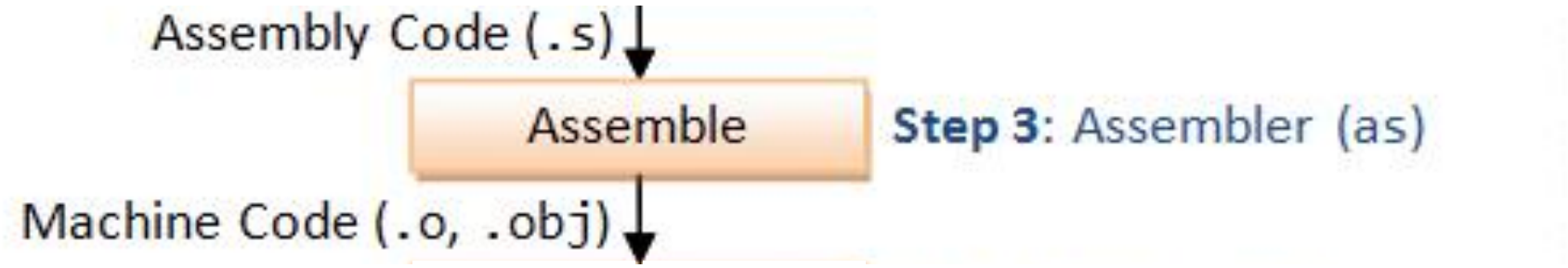
- **It's important to know that they parse source code and compile it into assembly code. You will learn more about this in the second part the course.**

# The Compiler – Demo

```
gcc -S hello.i
```

Compile preprocessed .i code into assembly instructions

# The GNU Compiler Collection (GCC)



# The Assembler – Demo

```
as -o hello.o hello.s
```

Assemble object code from hello.s

# The Assembler – ELF

**ELF: the Executable and Linkable Format**

# The Assembler – ELF

**ELF: the Executable and Linkable Format**

Cross-platform, used across multiple operating systems to represent components (object code) of a program. This comes in handy for linking and execution across different computers.

# The Assembler – ELF

**ELF: the Executable and Linkable Format**

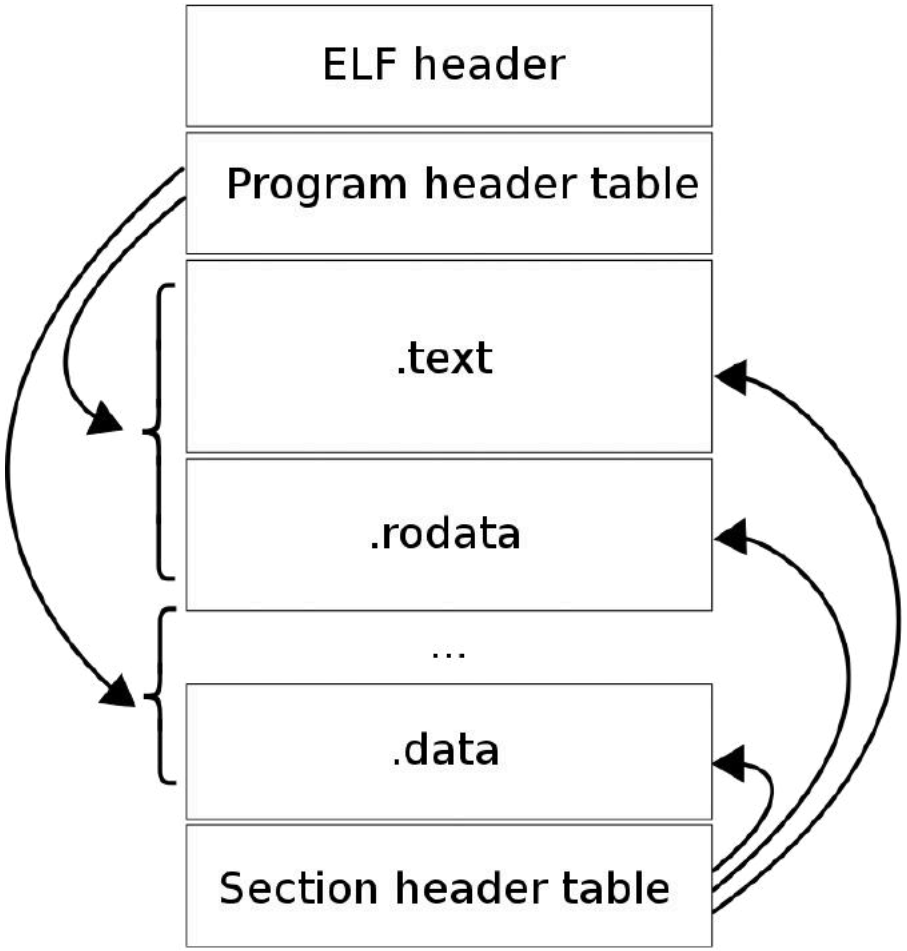
```
readelf -e hello.o
```

Actually read hello.o!
“-e” flag is for printing headers out only

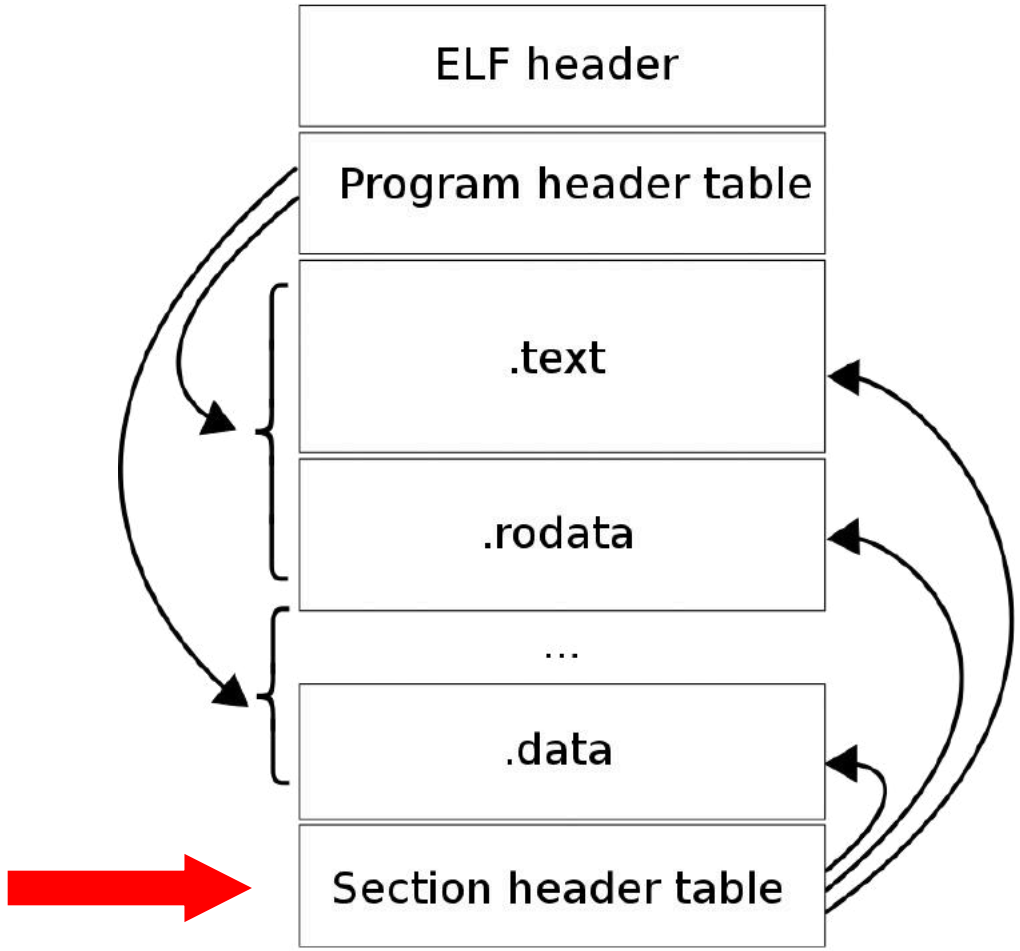
# The Assembler – ELF

| Section | Contents | Code Example |
|---|---|---|
| `.text` | Executable code (x86 assembly) | `mov -0x8(%rbp),%rax` |
| `.data` | Any global or static vars that have a pre-defined value and can be modified | `int val = 3` (as global var) |
| `.rodata` | Variables that are only read (never written) | `const int a = 0;` |
| `.bss` | All uninitialized data; global variables and static variables initialized to zero or or not explicitly static int i; initialized in source code | `static int i;` |
| `.comment` | Comments about the generated ELF (details such as compiler version and execution platform) | |

# The Assembler – ELF



# The Assembler – ELF

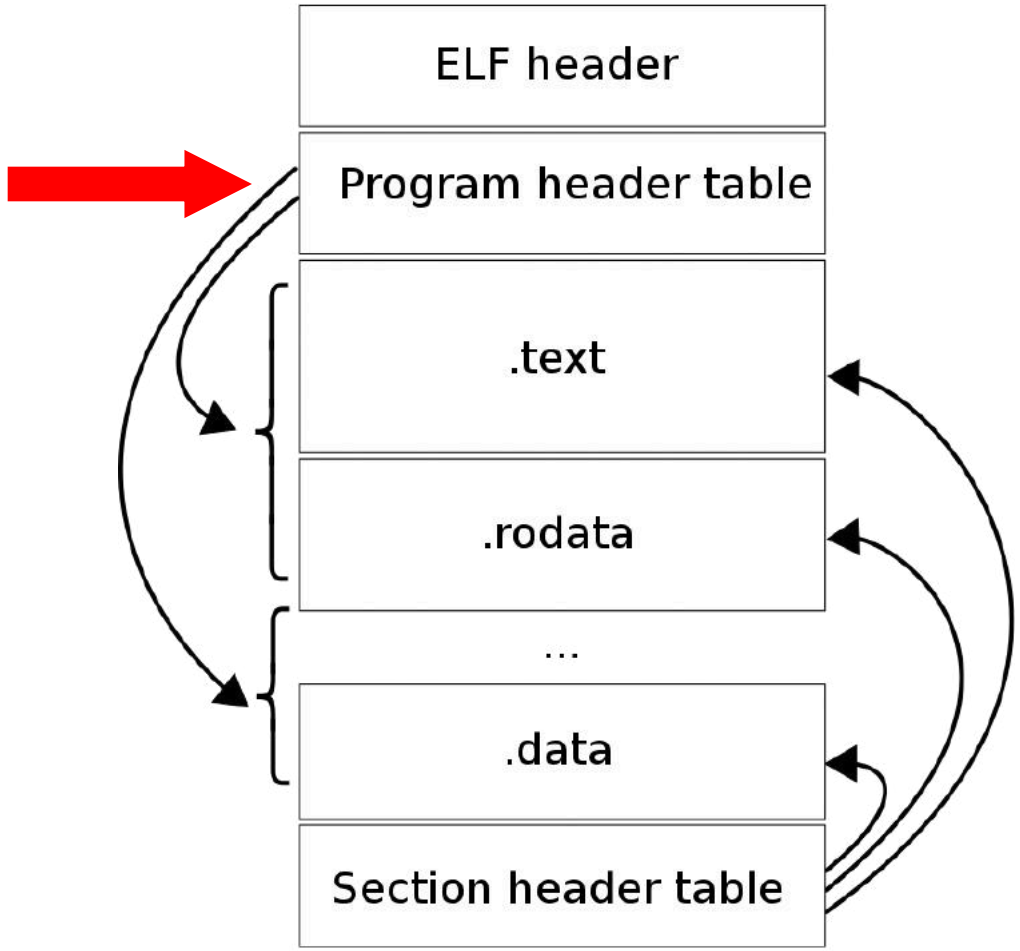


# The Assembler

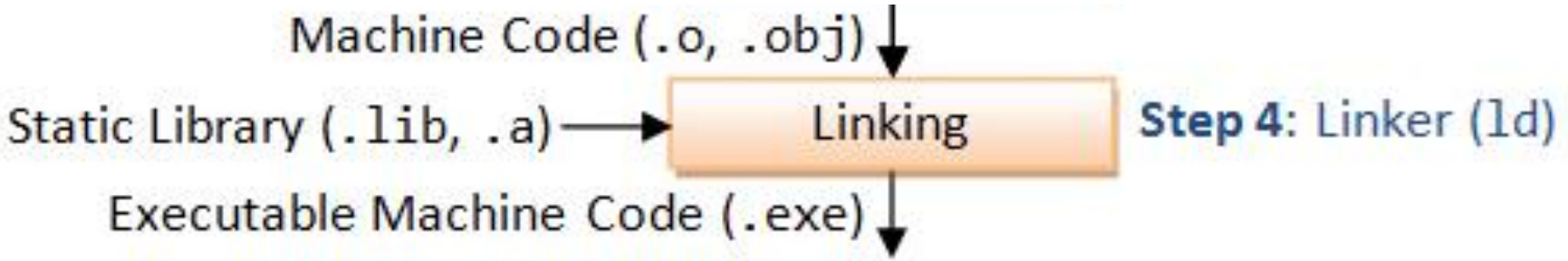
```
nm hello.o
```

Dump the variables and functions in hello and see what sections they belong to!

# The Assembler – ELF



# The GNU Compiler Collection (GCC)



# The Linker – Shared vs. Static Libraries

## Static Linking

1. When your program uses static linking, the machine code of external functions used in your program is copied into the executable.
2. A static library has file extension of ".a" (archive file) in Unix.

## Dynamic Linking

1. When your program is dynamically linked, only an offset table is created in the executable. The operating system loads the machine code needed for external functions during execution—a process known as dynamic linking.
2. A shared library has file extension of ".so" (shared objects) in Unix.

# The Linker

```
ld --dynamic-linker /lib64/ld-linux-x86-64.so.2 hello.o
   -o  hello -lc --entry main
```

1. **`--dynamic-linker`** is used to specify the linker we must use to load stdlib.
2. **`-lc`** tells the linker to link to the standard C library.
3. **`--entry main`** specifies the entry point of the program (the method “main”).

**Note: You may not get this command working, because it will be slightly different on different Linux distributions**

# Finally…

```
./hello
```

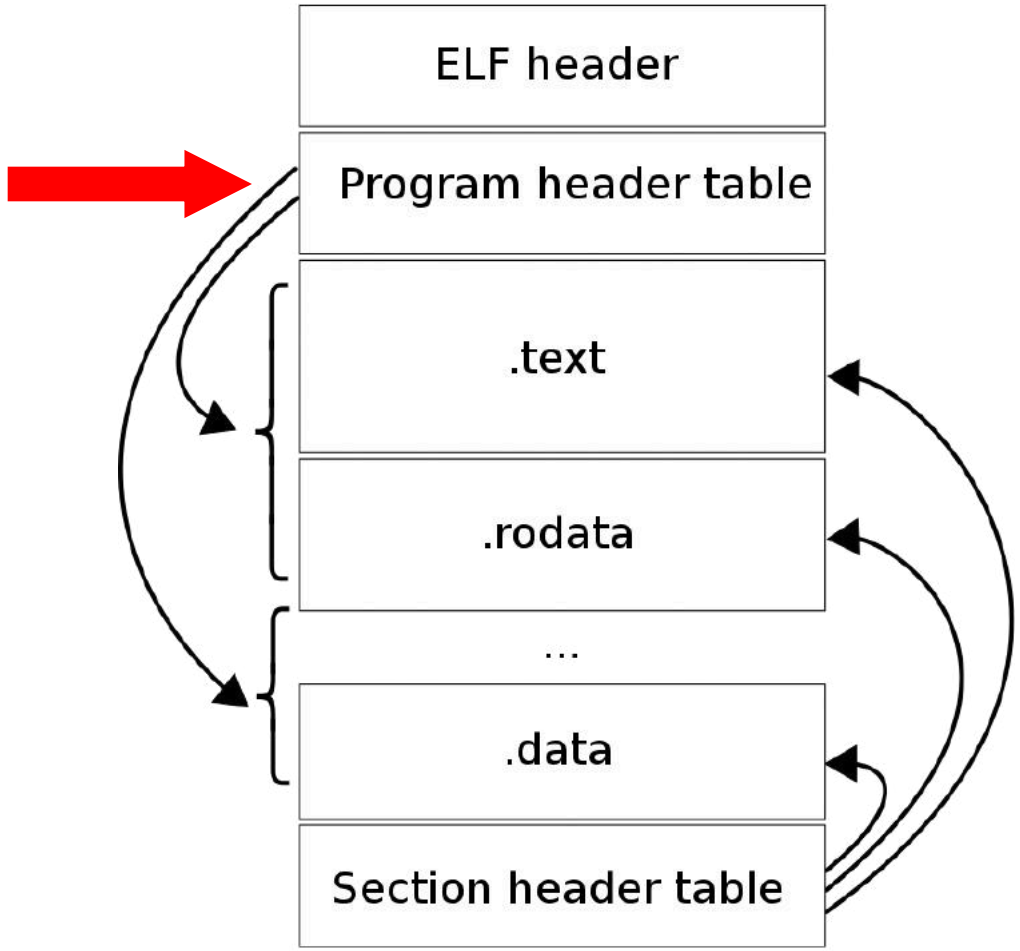
(Run your executable!)

# The Executable

```
nm hello
```

Let’s prove to ourselves linking did something…

# The Assembler – ELF



# Finally… (Really!)

```
./hello
```

(Run your executable!)

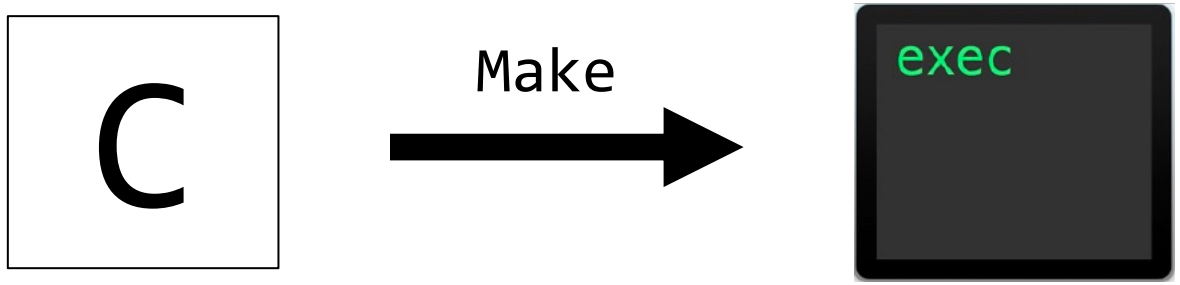
# Lecture Plan

- What really happens in GCC?
- Make and Makefiles
  - Overview of Make
  - Makefiles from scratch
  - Template for your Makefiles

# What is Make?

**Main Idea**

- You write the “recipe”
- Make builds target



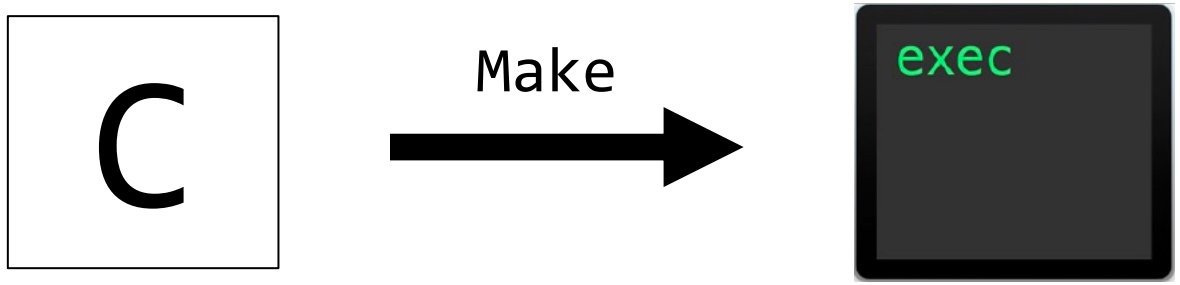
# What is Make?

## Main Idea

- You write the “recipe”
- Make builds target

## Definition

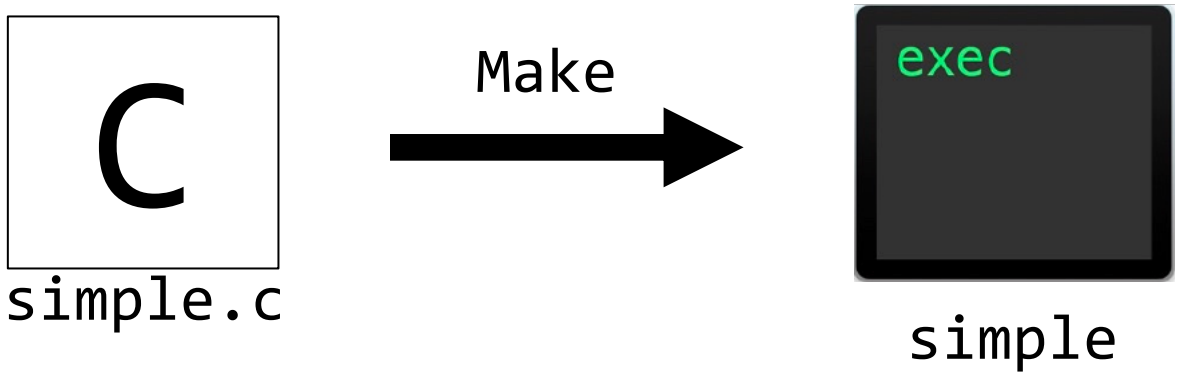
- “GNU Make is a tool which *controls the generation of executables*... from the program's source files.”
- - GNU Make Docs



# What is Make?

## Example

- *Target:* `simple`
- *Ingredients:* `simple.c`
- *Recipe:* `gcc -o simple simple.c`

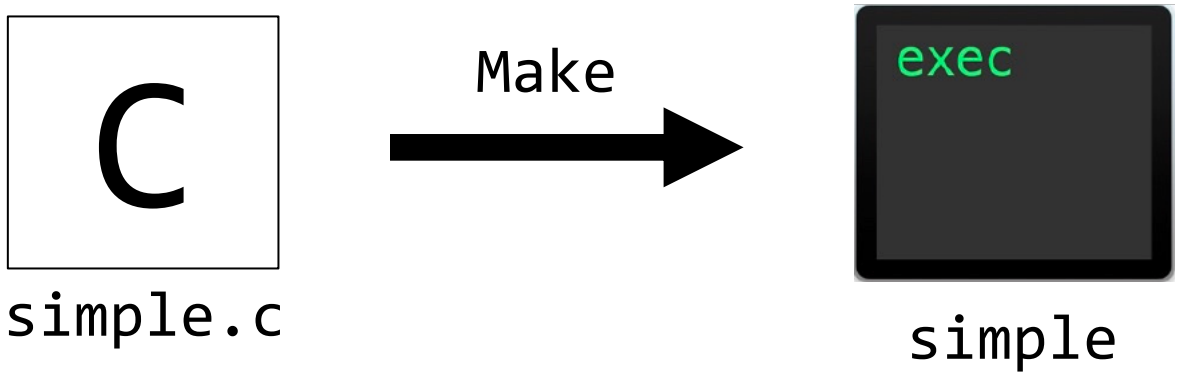


# What is Make?

## Example

- *Target:* `simple`
- *Ingredients:* `simple.c`
- *Recipe:* `gcc -o simple simple.c`

## Makefile Demo



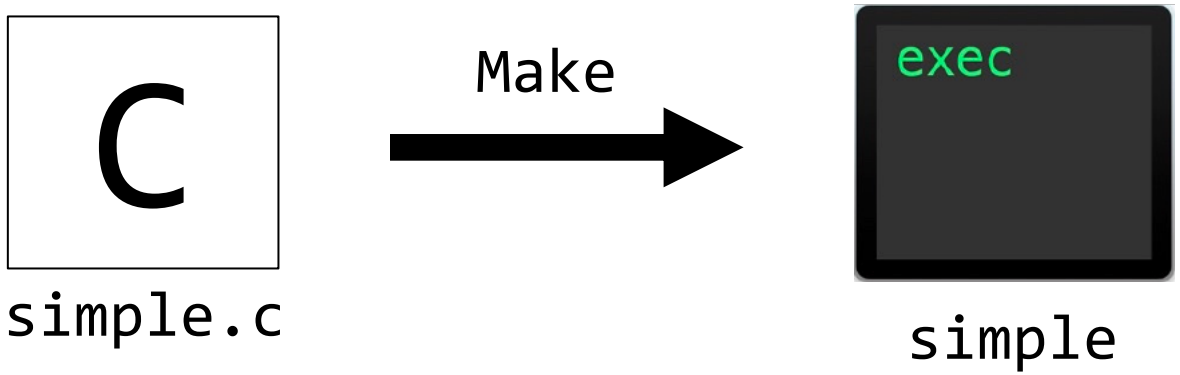
# What is Make?

## Example

- *Target:* `simple`
- *Ingredients:* `simple.c`
- *Recipe:* `gcc -o simple simple.c`

## Makefile Demo

```
simple: simple.c
    gcc -o simple simple.c
```



# So is Make just a shorter GCC?

**No!**

- More general
- Any target, any shell command

# So is Make just a shorter GCC?

**No!**

- More general
- Any target, any shell command

**Makefile Demo**

# So is Make just a shorter GCC?

**No!**

- More general
- Any target, any shell command

**Makefile Demo**

```
clean:
    rm -rf simple
```

**Usage:**

```
make clean
```

# So is Make just a shorter GCC?

## Advantages of Make

- *General:* Not just for compiling C source files
- *Fast:* Only rebuilds what's necessary
- *Shareable:* End users just call "make"

# Makefiles

## Makefile

- *Makefile:* A list of *rules*.
- *Rule:* Tells Make the ***commands*** to build a ***target*** from 0 or more ***dependencies***

```
target: dependencies...
    commands
    …
```

# Makefiles

## Makefile

- *Makefile:* A list of *rules*.
- *Rule:* Tells Make the ***commands*** to build a ***target*** from 0 or more ***dependencies***

```
target: dependencies...
    commands
    …
```

Must indent with '\t', not spaces

# Makefiles

## Makefile = List of Rules

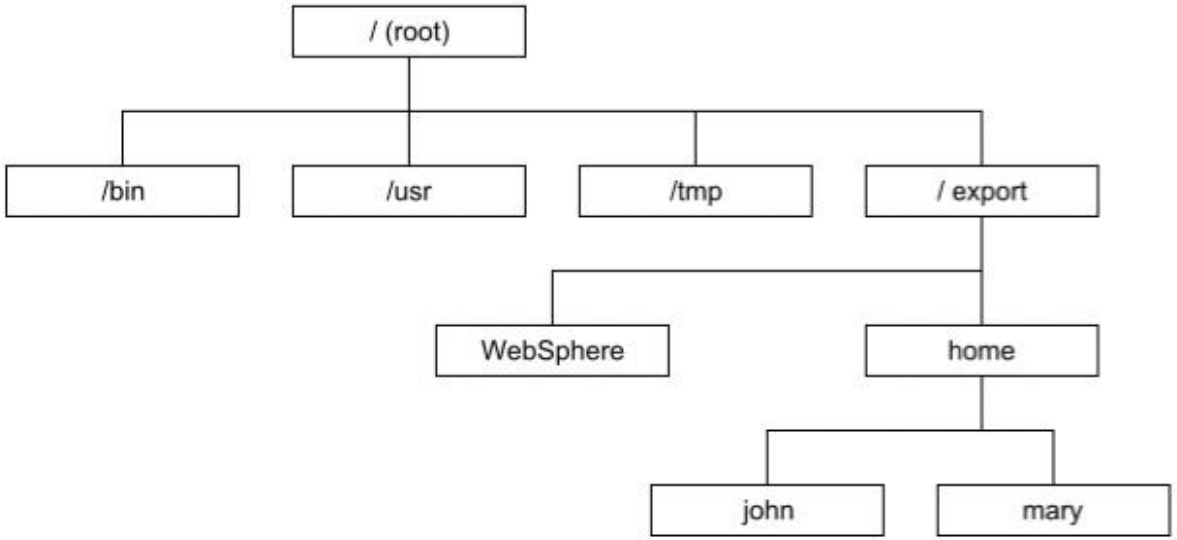
- *Rule:* Tells Make how to get to a ***target*** from ***source files***

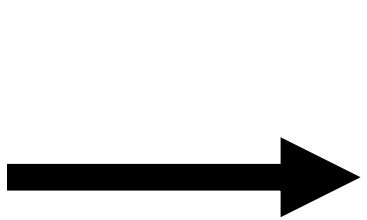
```
target: dependencies...
    commands
    …
```

"If dependencies have changed or don't exist, rebuild them…
Then execute these commands."

# Realistic Example

- Like Zip
- Traverses FS tree, builds a list of files
- Don’t know length ahead of time? Need growable data structure



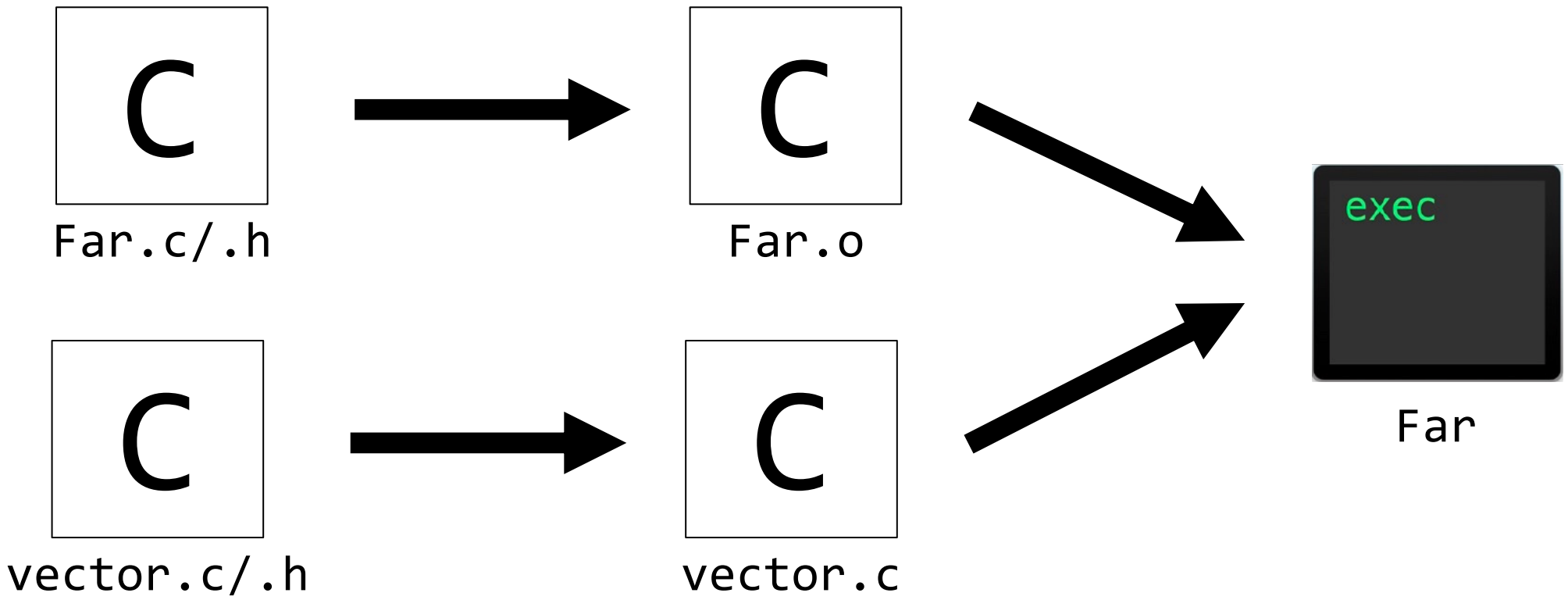



`all_files.ark`

# Realistic Example

**File Archiver**

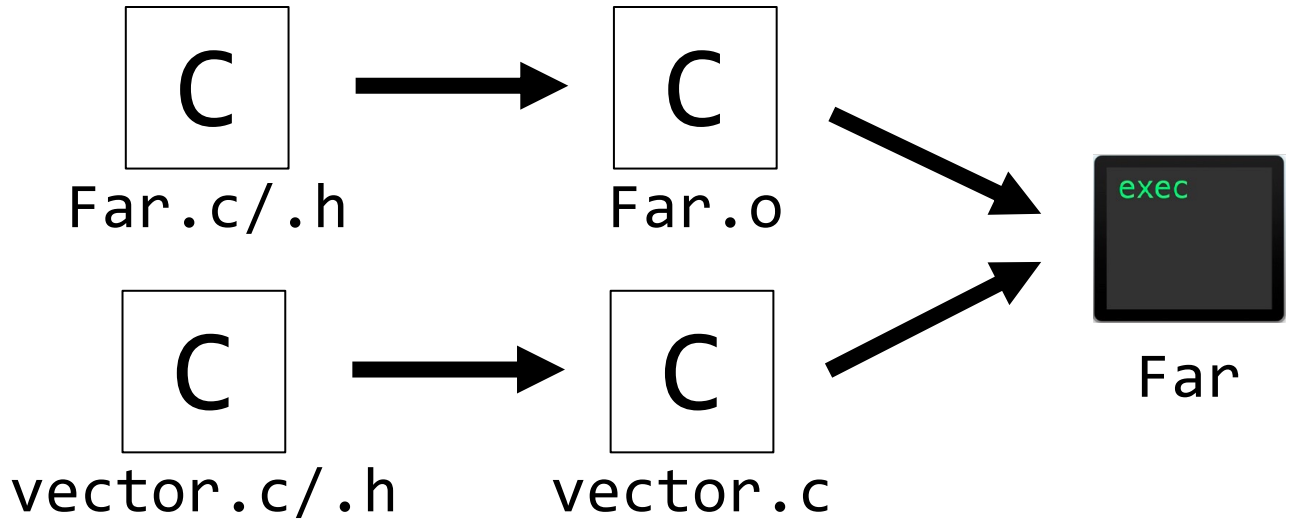
- Target file: `Far` (an executable)
- Source files: `Far.c Far.h vector.c vector.h`



# What is Make?

**Example**

- *Target:* `Far`
- *Ingredients:* `Far.o, vector.o`
- *Recipe:* `gcc -o Far Far.o vector.o`

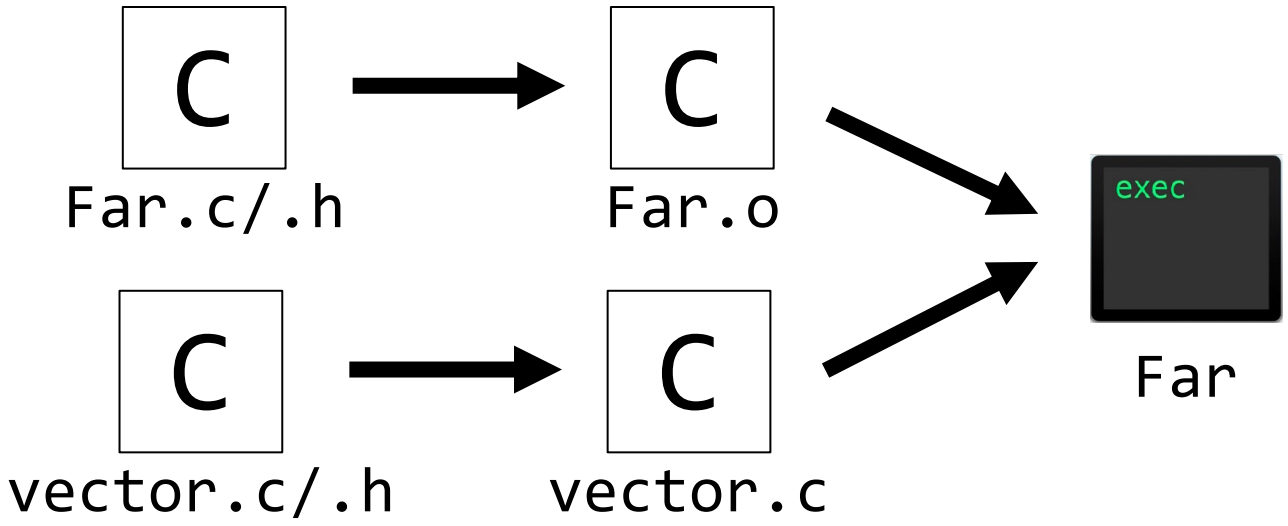


# What is Make?

## Example

- *Target:* `Far`
- *Ingredients:* `Far.o, vector.o`
- *Recipe:* `gcc -o Far Far.o vector.o`

## Makefile Demo



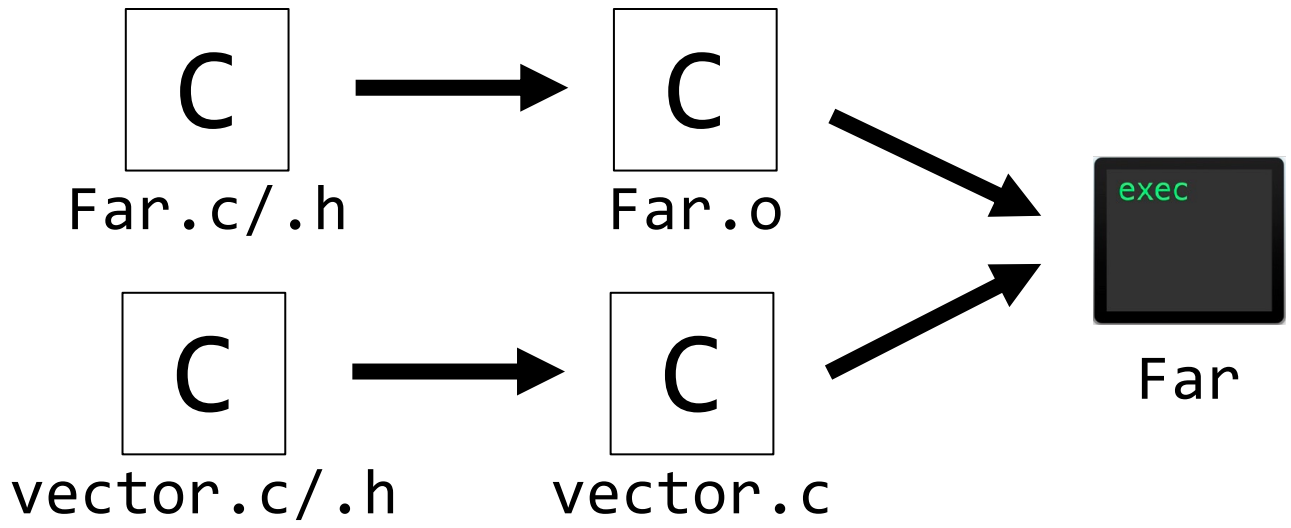
# What is Make?

## Example

- *Target:* `Far`
- *Ingredients:* `Far.o, vector.o`
- *Recipe:* `gcc -o Far Far.o vector.o`

## Makefile Demo

```
CC=gcc
CFLAGS=-g -std=c99 -pedantic –Wall

all: Far

Far: Far.o vector.o
     ${CC} ${CFLAGS} $^ -o $@
Far.o: Far.c Far.h vector.h
     ${CC} ${CFLAGS} -c Far.c
vector.o: vector.c vector.h
     ${CC} ${CFLAGS} -c vector.c
clean:
     ${RM} Far.o vector.o Far
```



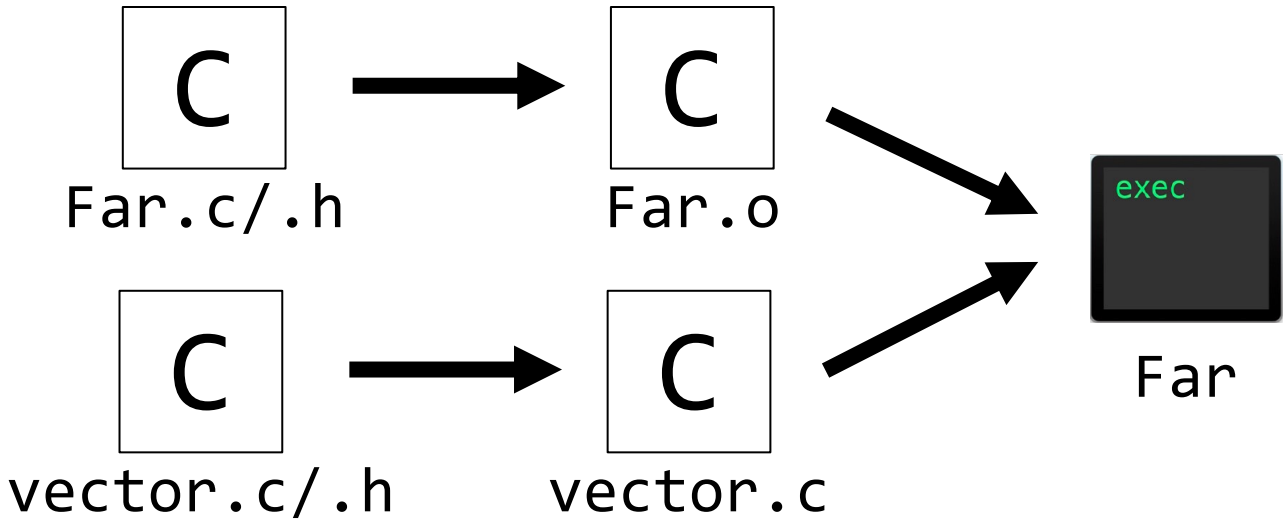
# What is Make?

**Example**

- *Target:* `Far`
- *Ingredients:* `Far.o, vector.o`
- *Recipe:* `gcc -o Far Far.o vector.o`

**Good Test Problem!**

Suppose I update `Far.c`,
Then call `make Far`.



# What is Make?

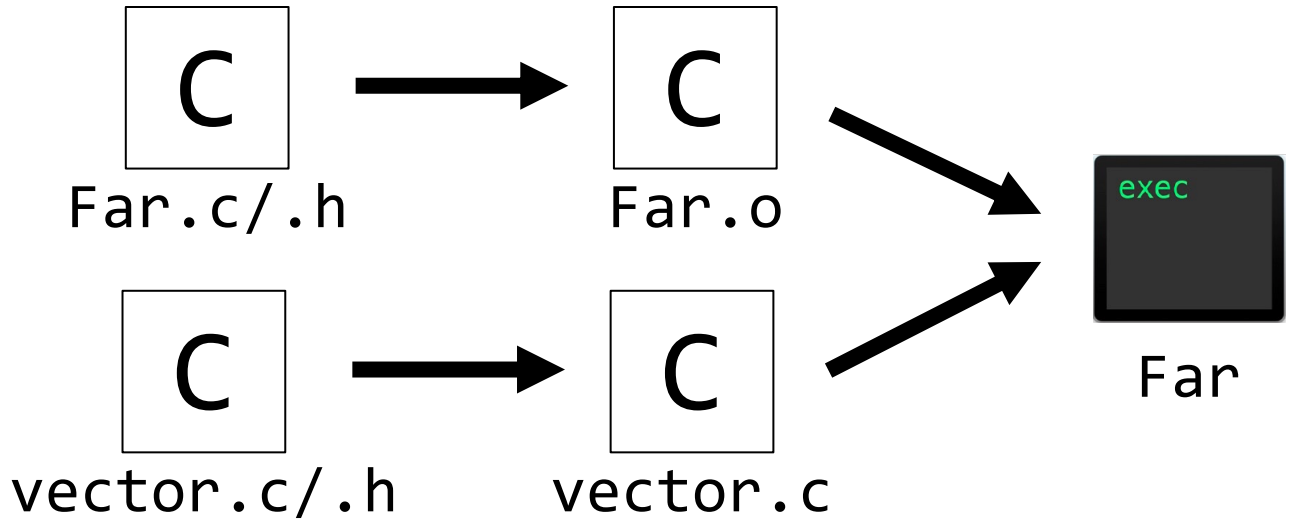
**Example**

- *Target:* `Far`
- *Ingredients:* `Far.o, vector.o`
- *Recipe:* `gcc -o Far Far.o vector.o`

**Good Test Problem!**

Suppose I update `Far.c`,
Then call `make Far`.

*Which commands does Make run?*



# What is Make?

## Example

- *Target:* `Far`
- *Ingredients:* `Far.o, vector.o`
- *Recipe:* `gcc -o Far Far.o vector.o`

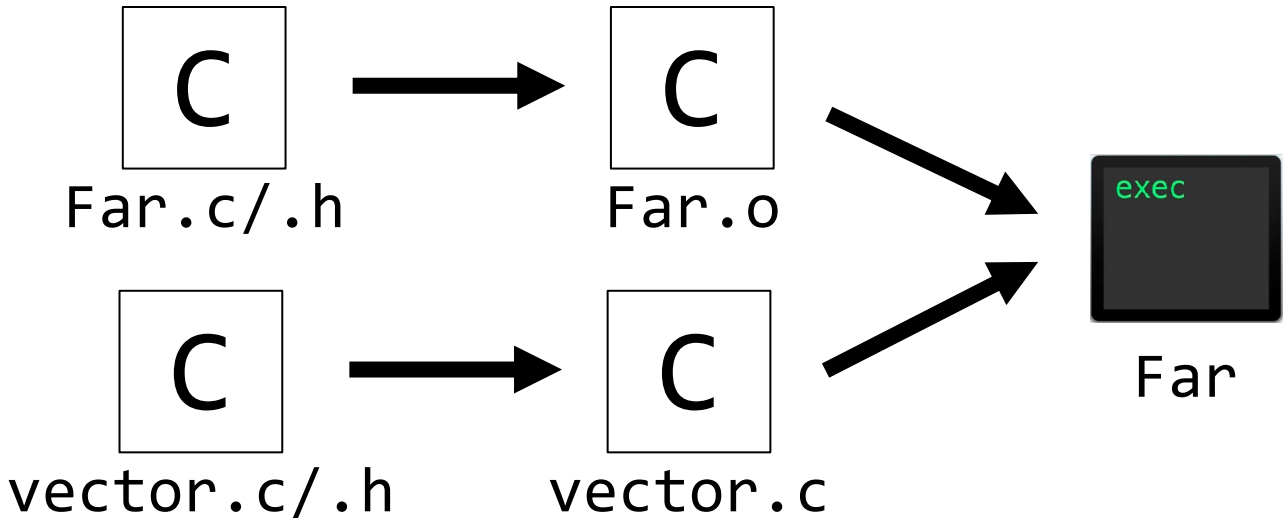
## Good Test Problem!

Suppose I update `Far.c`,
Then call `make Far`.

*Which commands does Make run?*

***Answer:***

*gcc -g -std=c99 -pedantic -Wall -c Far.c*
*gcc -g -std=c99 -pedantic -Wall Far.o vector.o -o Far*



# Takeaways

**Takeaways from File Archiver Example**

- Recursive rules
- Bigger projects practically *need* Make (or another build system) ▪
- Makefile variables (*e.g.,* `CC` and `CFLAGS`)
- Target need not be a file! (*e.g.,* `clean`)

# Generic Makefile

**Reusable Makefile**

- Any simple project
- Main program and its header
- Can be easily extended to include libraries
- Feel free to copy-paste

# Generic Makefile

```
# A simple makefile for building a program composed of C source files.
# Refer to CS107 guide to Make for background info on makefiles
#
PROGRAMS = hello

all:: $(PROGRAMS)

CC = gcc
FLAGS = -Wl -vervose -g3 -O0 -std=gnu99 -Wall $$warnflags
export warnflags = -Wfloat-equal -Wtype-limits -Wpointer-arith -Wshadow -Winit-self -fno-diagnostics-show-option
LDFLAGS =
LDLIBS =

$(PROGRAMS): %:%.c
	$(CC) $(CFLAGS) $(LDFLAGS) $^ $(LDLIBS) -o $@

clean::
	rm -f $(PROGRAMS) *.o

.PHONY: clean all
```

# Make Takeaways

## In The Wild

- Will see very complex makefiles — Don't be intimidated
- Will see other build systems (*e.g.,* CMake) — Same idea as Make
- Will see Make for other languages — Same source -> executable mapping

## References

- https://www.gnu.org/software/make/
- https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_makefiles.html Good Makefile examples/templates.

# Recap

- What really happens in GCC?
  - The Preprocessor
  - The Compiler
  - The Assembler
- Make and Makefiles
  - Overview of Make
  - Makefiles from scratch
  - Template for your Makefiles