photo by unsplash user @xavi_cabrera

COMP201

Computer Systems & Programming

Lecture #13 – Compiling C Programs

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Fall 2021

Image: Professor Farnsworth (Futurama)

Good news, everyone!

- Assignment 3 is out today! (due November 25, 23:59)
  - Mustafa will announce an extra office hour to answer your questions

Recap

- `const`
- `struct`
- Generic stack

Recap: `const`

- Use **`const`** to declare global constants in your program. This indicates the variable cannot change after being created.

```
const double PI = 3.1415;
const int DAYS_IN_WEEK = 7;

int main(int argc, char *argv[]) {
      …
      if (x == DAYS_IN_WEEK) {
            …
      }
      …
}
```

Recap: `const`

- Use **`const`** with pointers to indicate that the data that is pointed to cannot change.

```
char str[6];
strcpy(str, "Hello");
const char *s = str;

// Cannot use s to change characters it points to
s[0] = 'h';
```

Recap: `const`

Sometimes we use **const** with pointer parameters to indicate that the function will not / should not change what it points to. The actual pointer can be changed, however.

```
// This function promises to not change str's characters
int countUppercase(const char *str) {
	int count = 0;
	for (int i = 0; i < strlen(str); i++) {
		if (isupper(str[i])) {
			count++;
		}
	}
	return count;
}
```

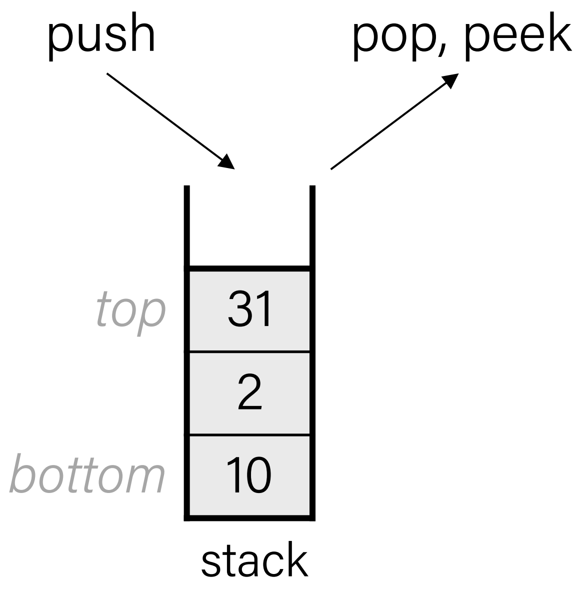
Recap: Structs

A **struct** is a way to define a new variable type that is a group of other variables.

```
typedef struct date {        // declaring a struct type
    int month;
    int day;              // members of each date structure
} date;
…

date today;                          // construct structure instances
today.month = 1;
today.day = 28;

date new_years_eve = {12, 31};  // shorter initializer syntax
```

Recap: Stacks

- A **Stack** is a data structure representing a stack of things.
- Objects can be ***pushed*** on top of or ***popped*** from the top of the stack.
- Only the top of the stack can be accessed; no other objects in the stack are visible.
- Main operations:
  - **`push(value)`**: add an element to the top of the stack
  - **`pop()`**: remove and return the top element in the stack
  - **`peek()`**: return (but do not remove) the top element in the stack



Int vs. Generic Stack Structs

```c
typedef struct int_node {
    struct int_node *next;
    int data;
} int_node;


typedef struct int_stack {
    int nelems;
    int_node *top;
} int_stack;
```

```c
typedef struct int_node {
    struct int_node *next;
    void *data;
} int_node;


typedef struct stack {
    int nelems;
    int elem_size_bytes;
    node *top;
} stack;
```

Int vs. Generic stack_create

```c
int_stack *int_stack_create() {
    int_stack *s =
        malloc(sizeof(int_stack));
    s->nelems = 0;
    s->top = NULL;
    return s;
}
```

```c
stack *stack_create(int elem_size_bytes) {
    stack *s = malloc(sizeof(stack));
    s->nelems = 0;
    s->top = NULL;
    s->elem_size_bytes = elem_size_bytes;
    return s;
}
```

From previous slide:

```c
typedef struct stack {
    int nelems;
    int elem_size_bytes;
    node *top;
} stack;
```

Int vs. Generic stack_push

```
void int_stack_push(int_stack *s, int data) {
    int_node *new_node =
              malloc(sizeof(int_node));
    new_node->data = data;

    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```

From previous slide:

```
typedef struct stack {
    int nelems;
    int elem_size_bytes;
    node *top;
} stack;
```

```
typedef struct node {
    struct node *next;
    void *data;
} node;
```

```
void stack_push(stack *s, const void *data) {
    node *new_node = malloc(sizeof(node));
    new_node->data = malloc(s->elem_size_bytes);
    memcpy(new_node->data,data,s->elem_size_bytes);

    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}


int main() {
    stack *int_stack = stack_create(sizeof(int));
    add_one(int_stack);
}

void add_one(stack *s) {
    int num = 7;
    stack_push(s, &num);
}
```

Int vs. Generic `stack_pop`

```c
int int_stack_pop(int_stack *s) {
  if (s->nelems == 0) {
     error(1,0,"Cannot pop from empty stack");
  }
  int_node *n = s->top;
  int value = n->data;

  s->top = n->next;

  free(n);
  s->nelems--;

  return value;
}
```

From previous slide:

```c
typedef struct stack {
    int nelems;
    int elem_size_bytes;
    node *top;
} stack;
```

```c
typedef struct node {
    struct node *next;
    void *data;
} node;
```

```c
void stack_pop(stack *s, void *addr) {
  if (s->nelems == 0)
      error(1,0,"Cannot pop from empty stack");
  node *n = s->top;
  memcpy(addr, n->data, s->elem_size_bytes);
  s->top = n->next;

  free(n->data);
  free(n);
  s->nelems--;
}

int main() {
  stack *intstack = stack_create(sizeof(int));
  for (int i = 0; i < TEST_STACK_SIZE; i++) {
    stack_push(intstack, &i);
  }
  // Pop off all elements
  int popped_int;
  while (intstack->nelems > 0) {
    int_stack_pop(intstack, &popped_int);
    printf("%d\n", popped_int);
  }
}
```

Plan for Today

- What really happens in GCC?
- Make and Makefiles

xkcd.com/303/

Disclaimer: Slides for this lecture were borrowed from
—Gabbi Fisher and Chris Chute's Stanford CS107 class
—Jae Woo Lee's Columbia COMS W3157 class

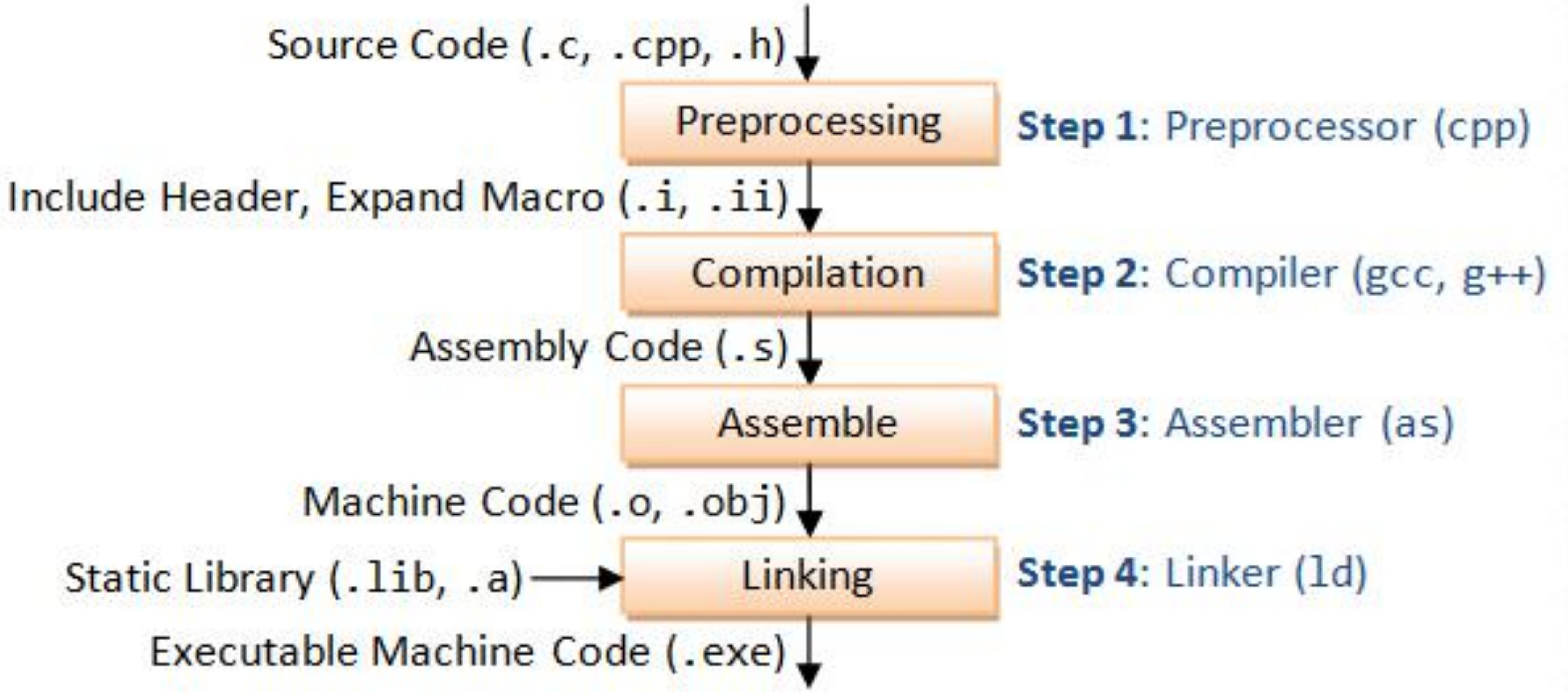
Lecture Plan

- What really happens in Gnu Compiler Collection (`gcc`)?
  - The Preprocessor
  - The Compiler
  - The Assembler
  - The Linker
- Make and Makefiles

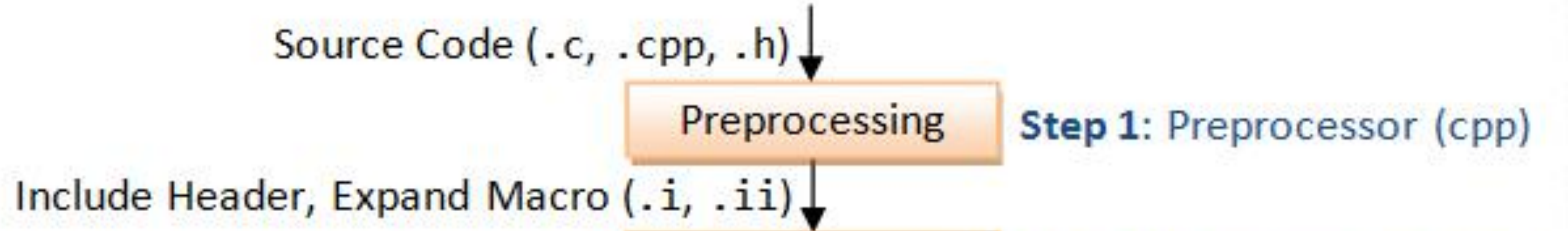
Compiling a C program with GCC

```
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 `#define` directive can be used to set up symbolic replacements in the source.

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))
#define twice(X) (2*(X))


y = min(1,2);



y = twice(1+1);
```

The Preprocessor – Function Macros

```
#define min(X,Y) ((X) < (Y) ? (X) : (Y))
#define twice(X) (2*(X))

y = min(1,2);
=> y = ((1) < (2) ? (1) : (2));

y = twice(1+1);
=> y = (2*(1+1));
```

The Preprocessor – Imports

```
#include
```

The Preprocessor – Imports

<u>header.h</u>

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

<u>program.c</u>

```
#include "header.h"

int x;

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

The `#include` directive just pastes in the text from the given file.

The Preprocessor – Imports

header.h

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

program.c

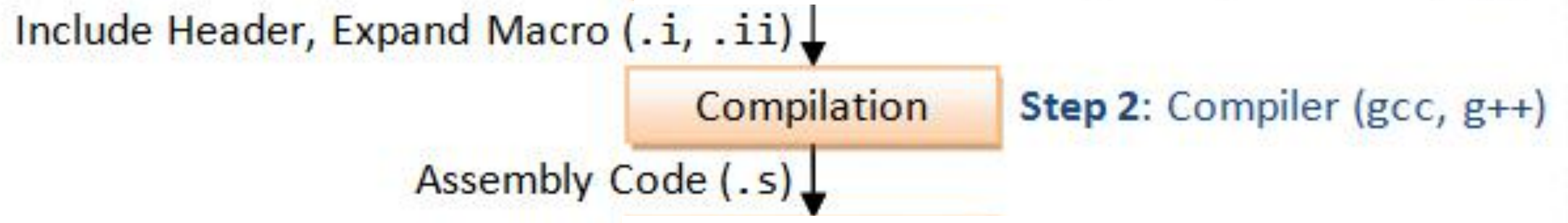
```c
char *test(void);

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)_/¯

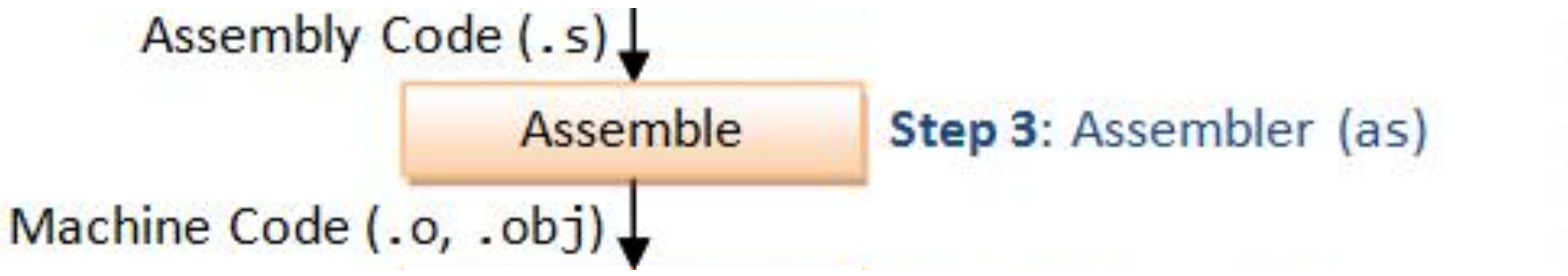
- **It's important to know that they parse source code and compile it into assembly code.** **You will learn more about assembly in the second part the course.**

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

a rare depiction of an Elf made by Tolkien

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

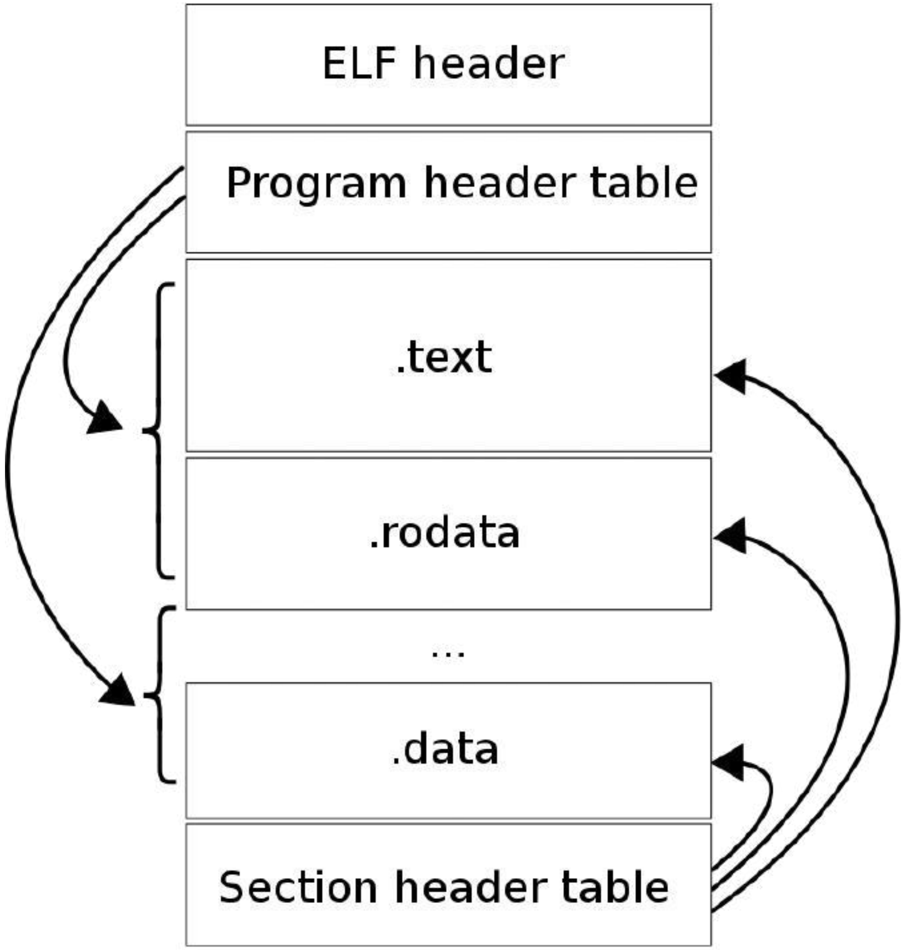
```
readelf -e hello.o
```

Actually read `hello.o`!
"`-e`" flag is for printing headers out only

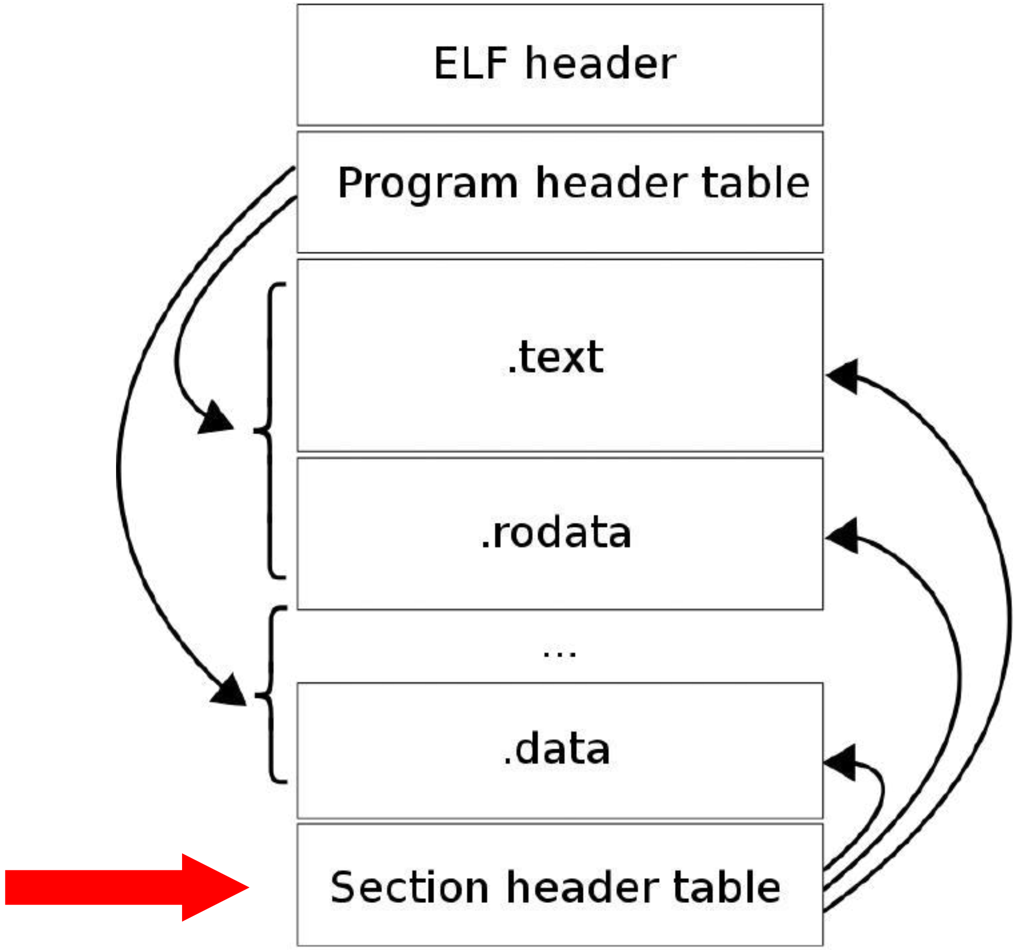
The Assembler – ELF

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

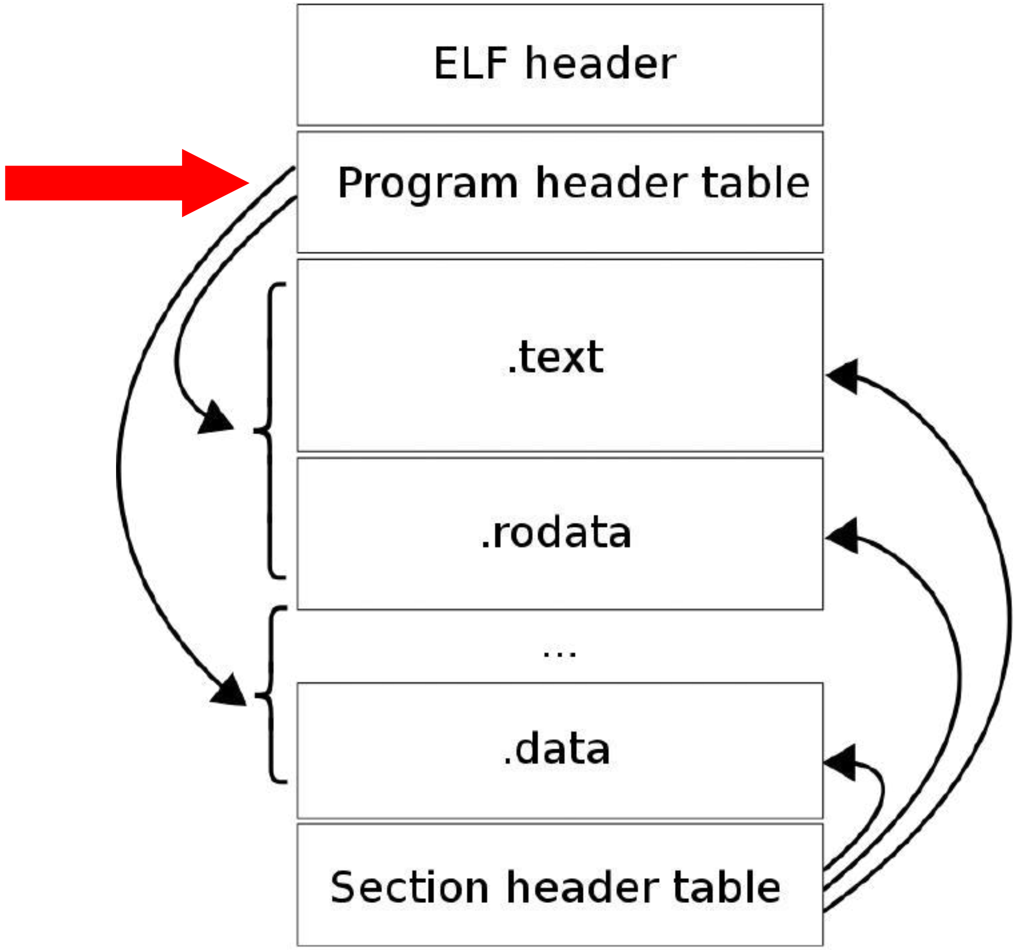


The Assembler

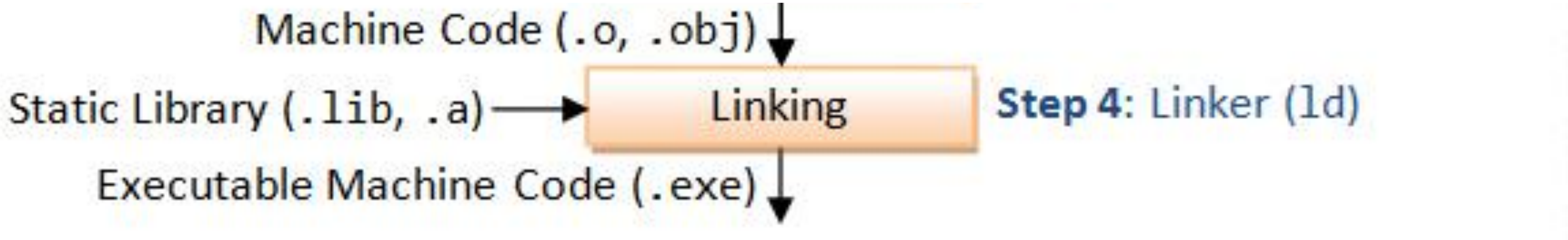
```
nm hello.o
```

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
```

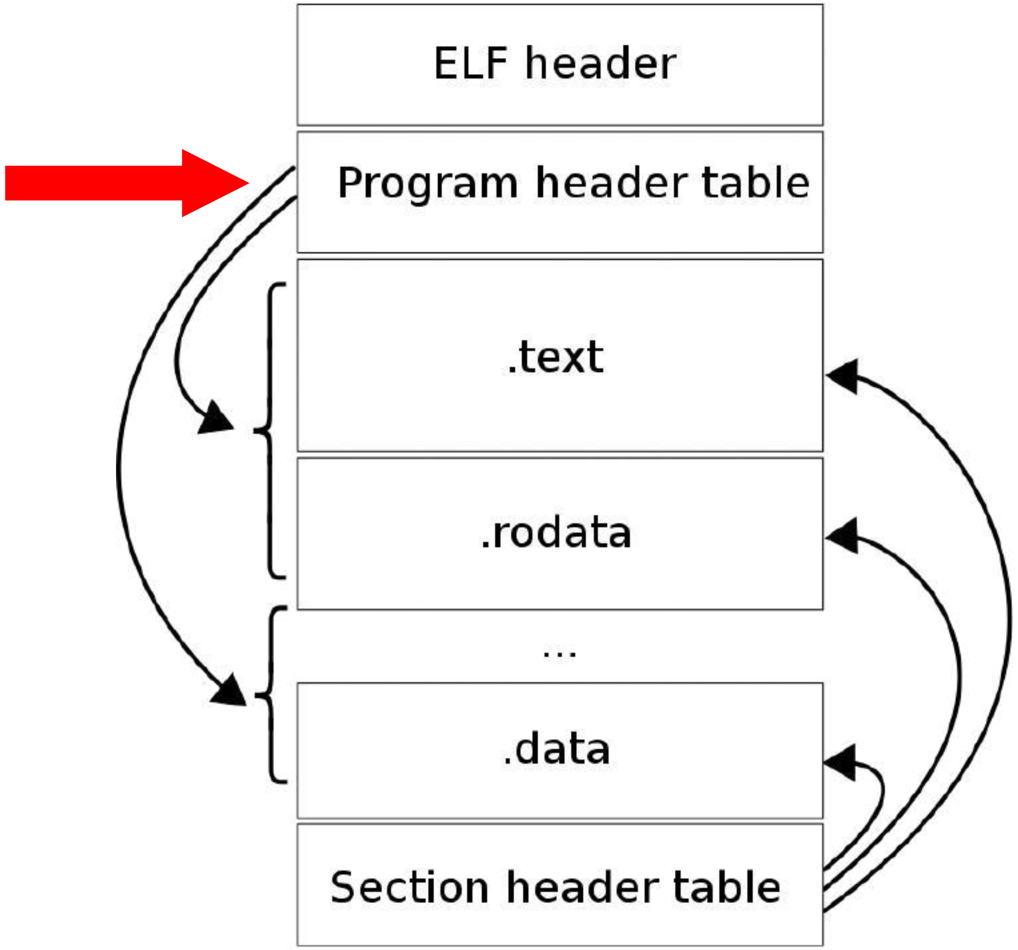
(Run your executable!)

The Executable

```
nm hello
```

Let's prove to ourselves linking did something…

The Assembler – ELF



Finally… (Really!)

```
./hello
```

(Run your executable!)

Linking Multiple Files and Library

```
gcc -c myfile1.c
gcc -c myfile2.c
gcc -g myfile1.o myfile2.o -lm -o myprogram
```

Using Multiple Functions

```c
program.c

int add(int x, int y);

int main(int argc, char **argv)
{
    int sum;
    sum = add(1, 2);
    printf("%d\n", sum);
    return 0;
}

int add(int x, int y)
{
    return x + y;
}
```

- function declaration (also called a prototype)
- a function must have been seen before it's called
- enables compiler to do type-checking

Using Multiple Files

myadd.h (called a header file)

```
#ifndef _MYADD_H_
#define _MYADD_H_

int add(int x, int y);

#endif
```

myadd.c

```
#include "myadd.h"
int add(int x, int y)
{
    return x + y;
}
```

main.c

```
#include "myadd.h"

int main(int argc, char **argv)
{
    int a = 1;
    int b = 2;

    c = add(a,b);

    printf("%d + %d = %d", a, b, c);
}
```

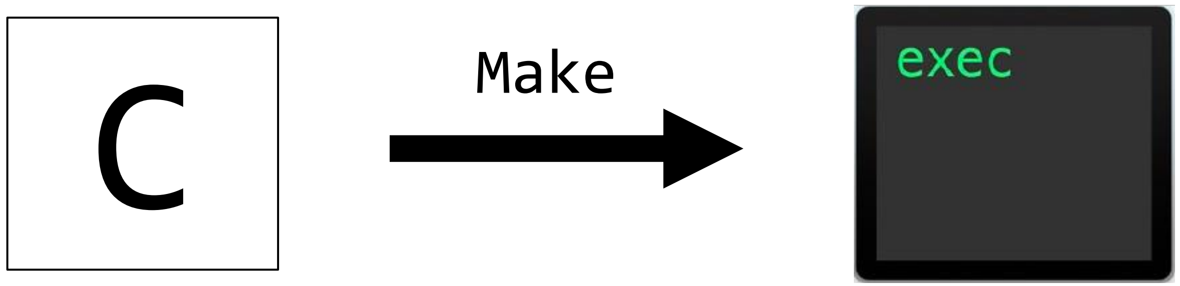
Lecture Plan

- 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?

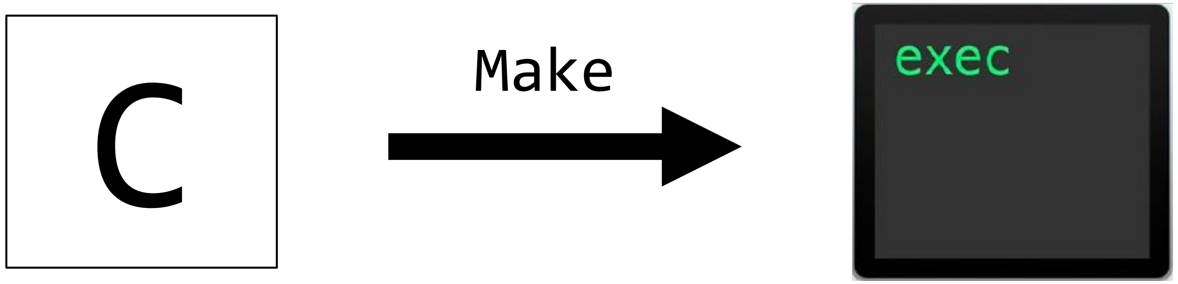
Main Idea

- 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.”

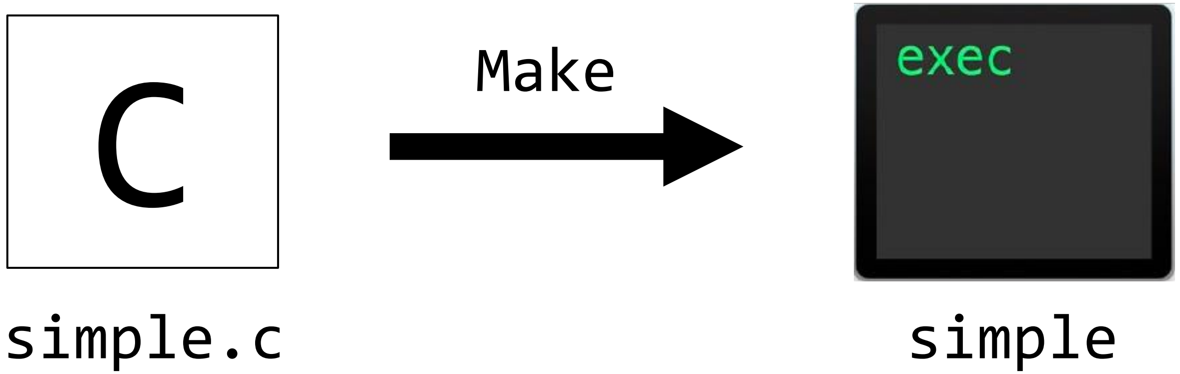
  - GNU Make Docs



What is Make?

Example

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

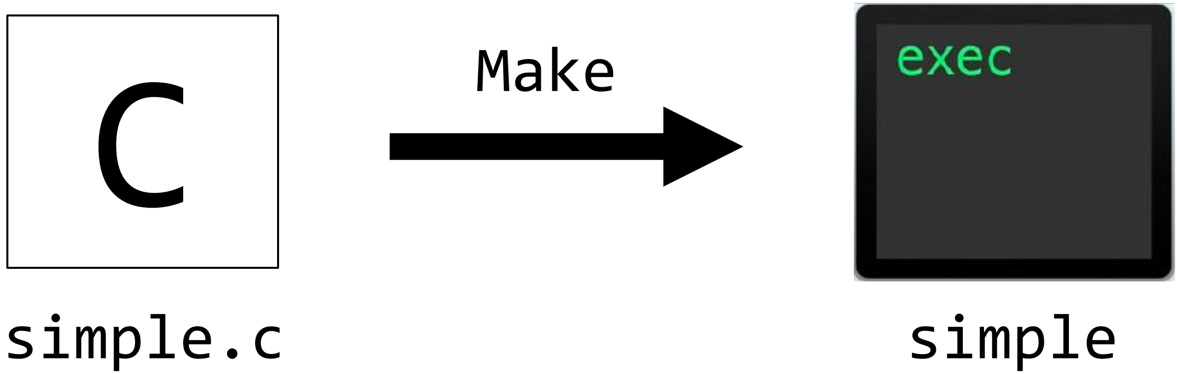


What is Make?

Example

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

Makefile Demo



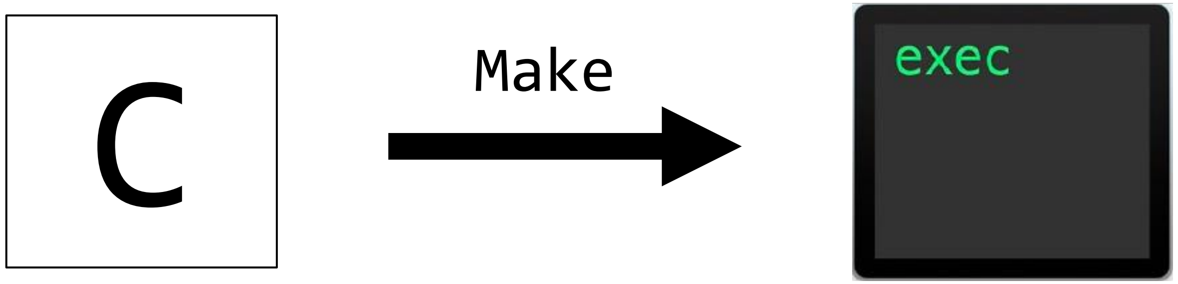
What is Make?

Example

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

Makefile Demo

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



simple.c simple

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

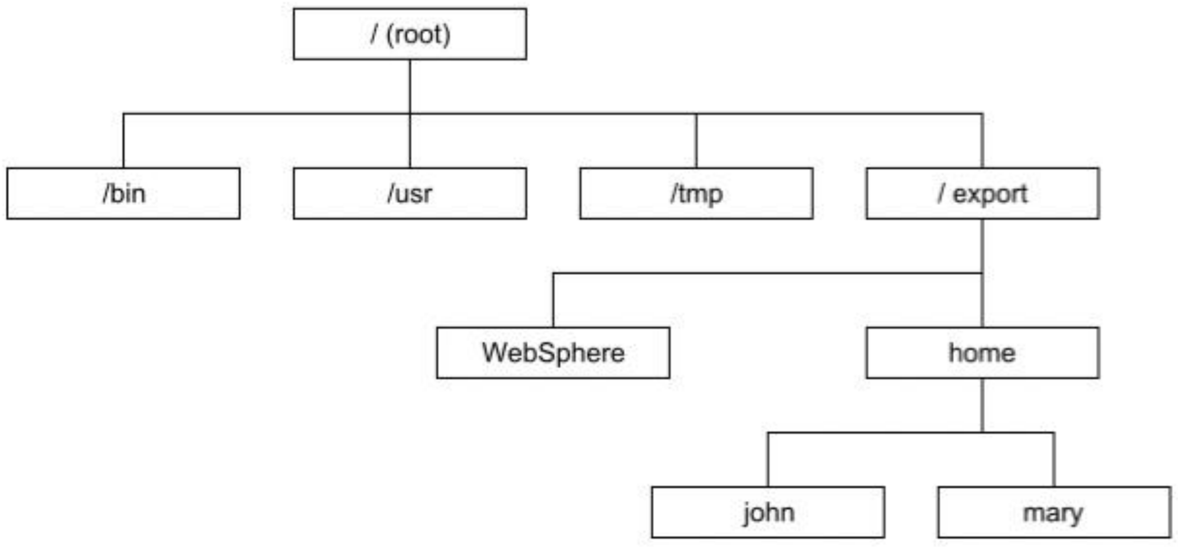
- *Rule:* Tells Make how to get to a ***target*** from ***source files***

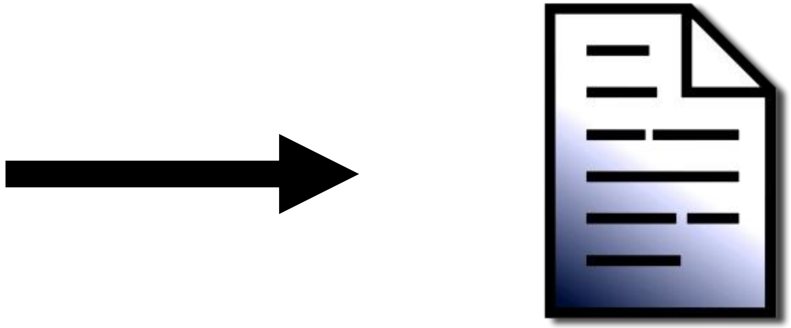
```
target: dependencies...
        commands
        …
```

"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



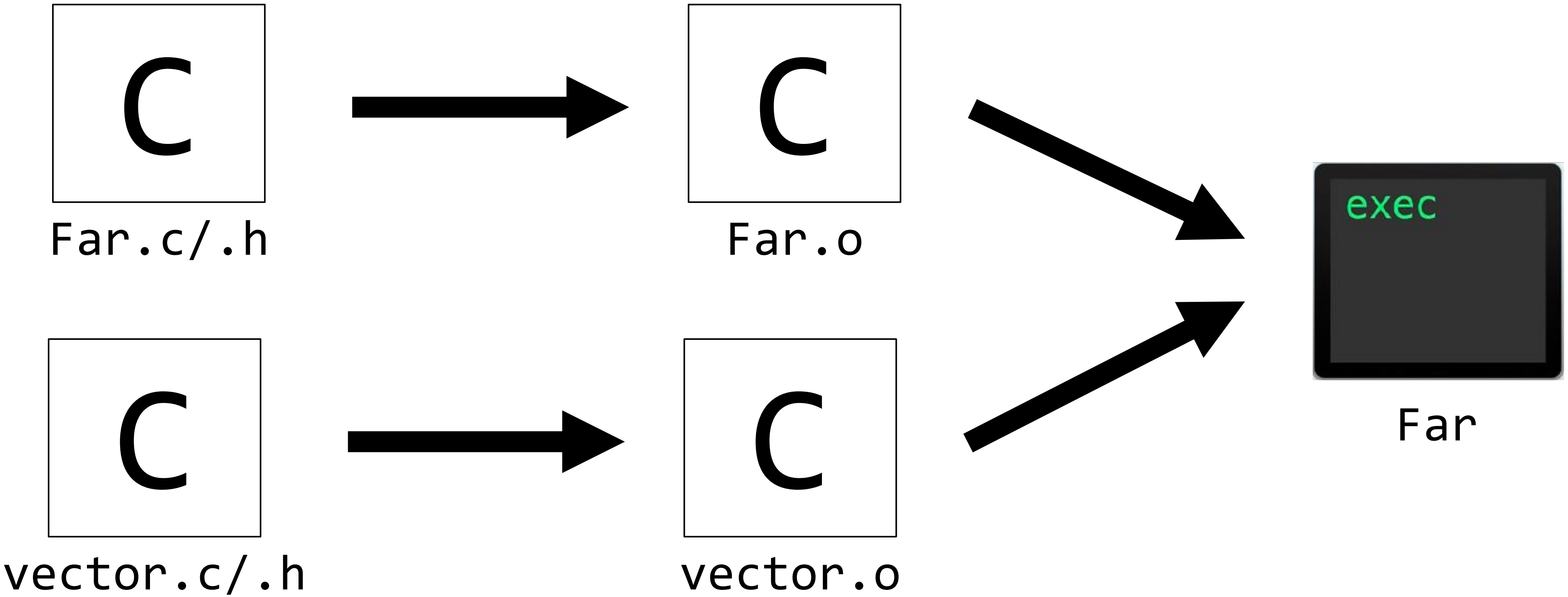


`all_files.ark`

Realistic Example

File Archiver

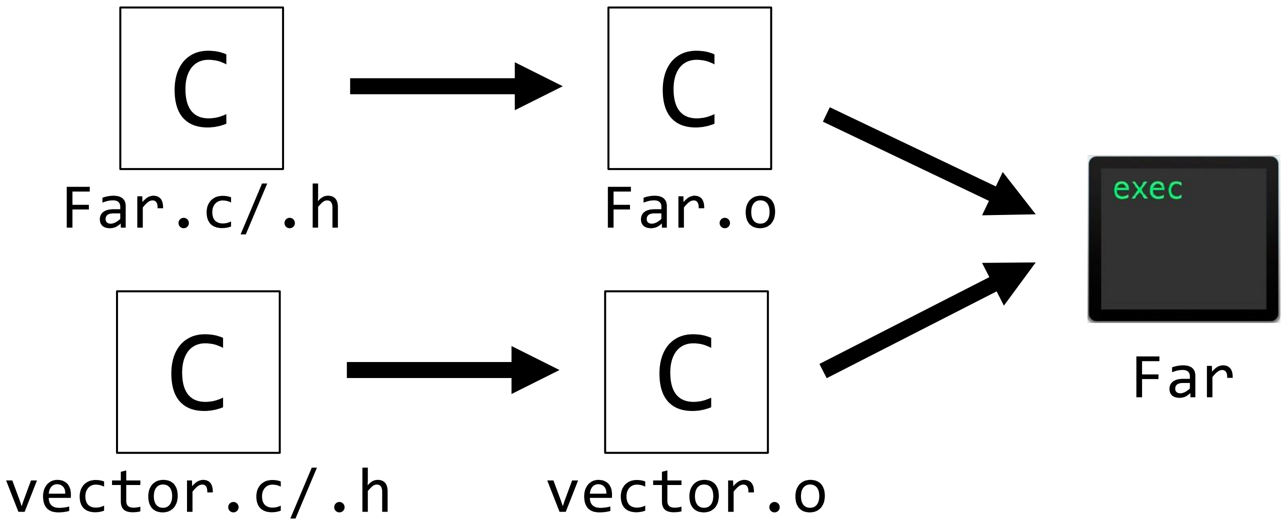
- Target file: `Far` (an executable)
- Source files: `Far.c Far.h vector.c vector.h`



What is Make?

Example

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

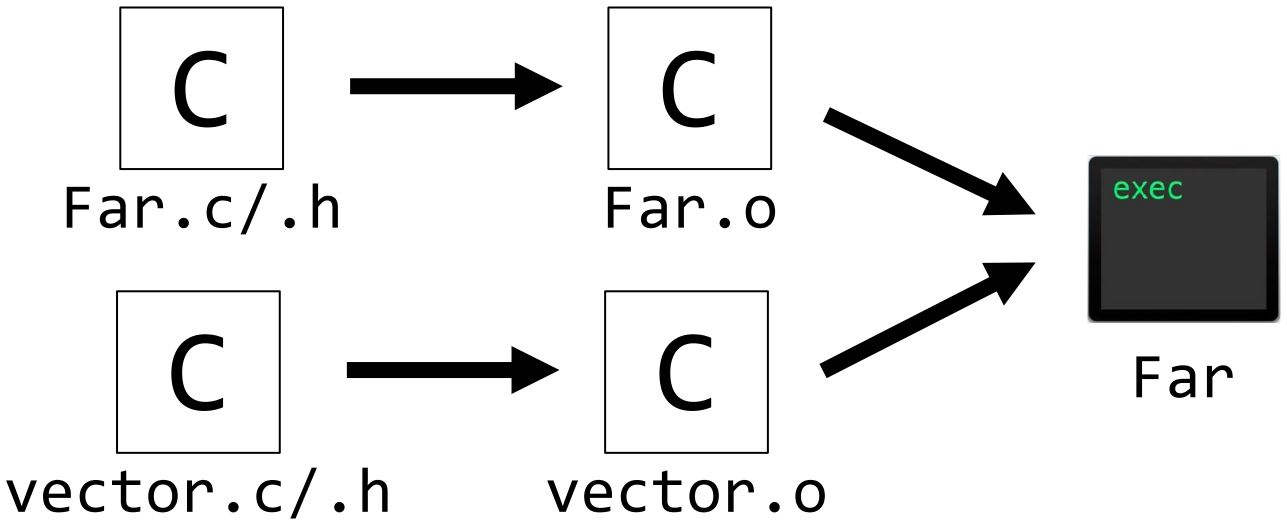


What is Make?

Example

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

Makefile Demo



What is Make?

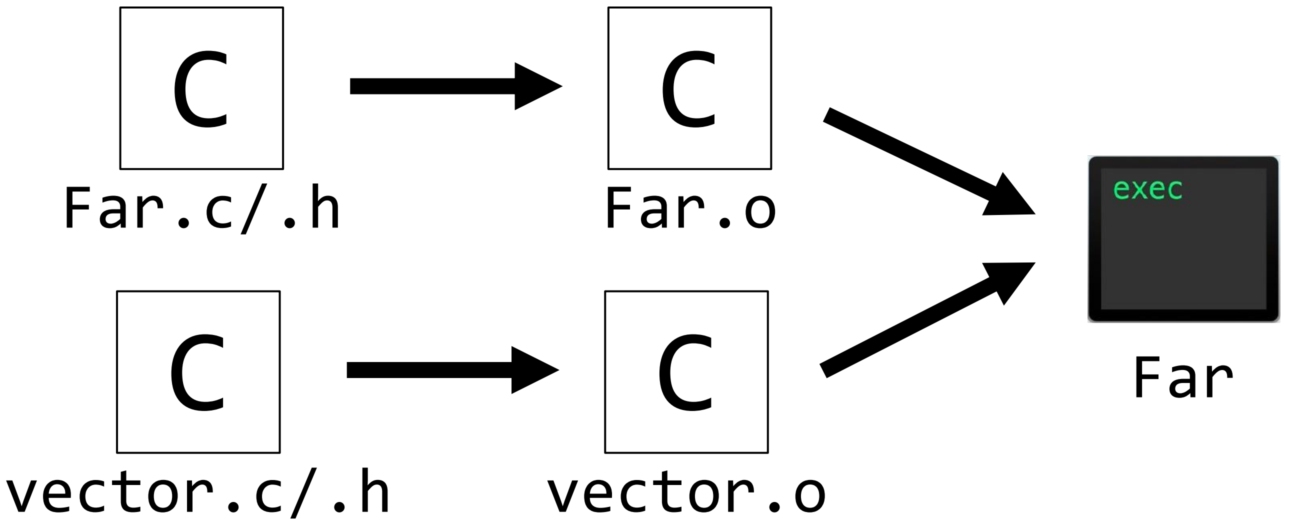
Example

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

$@: The file name of the target of the rule
$^ : The names of all the prerequisites, with spaces between them

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
```



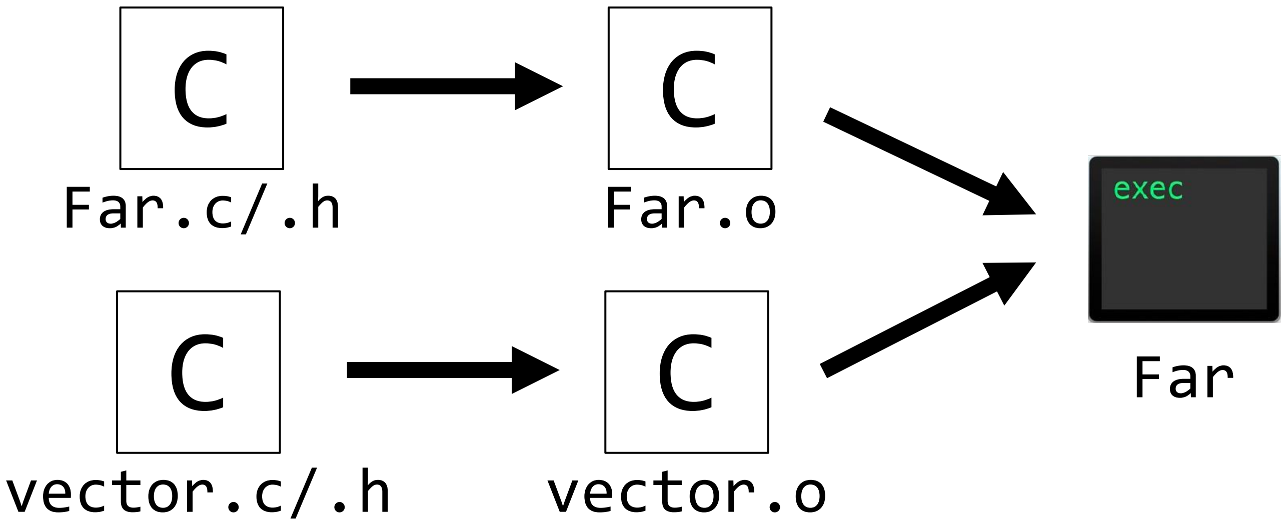
What is Make?

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?

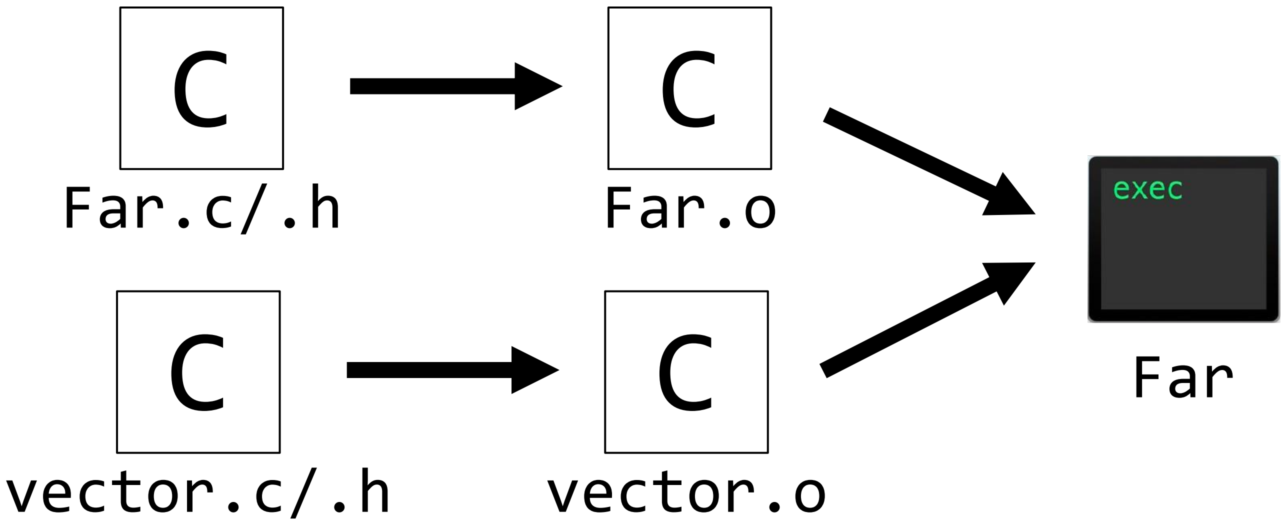
Example

- *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`

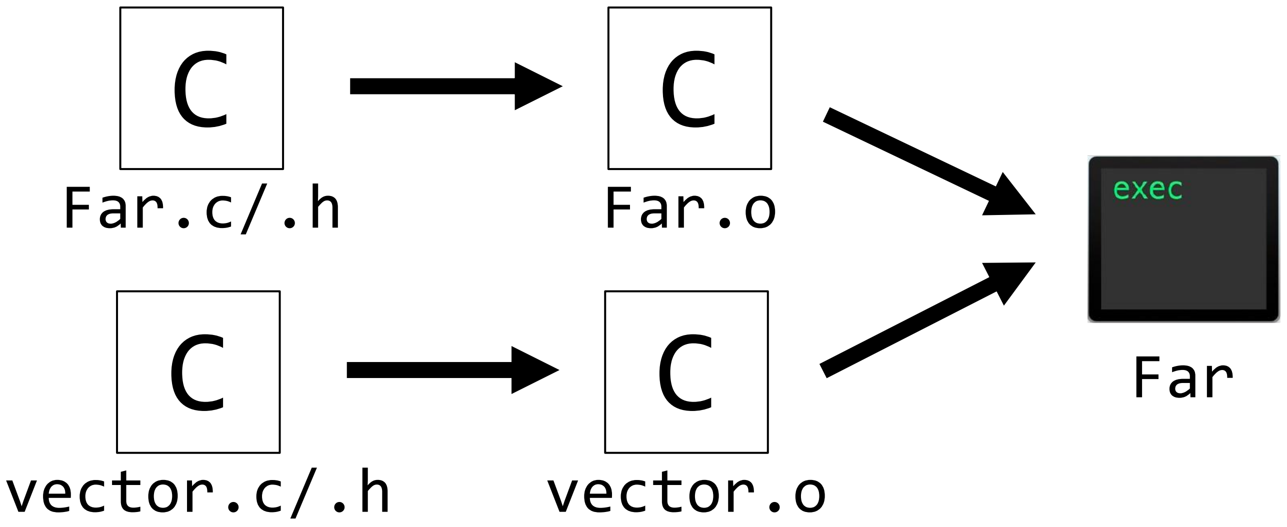
Good Test Problem!

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.
#
PROGRAMS = hello

all:: $(PROGRAMS)

# It is likely that default C compiler is already gcc, but explicitly
# set, just to be sure
CC = gcc

# The CFLAGS variable sets compile flags for gcc:
#  -g          compile with debug information
#  -Wall       give verbose compiler warnings
#  -O0         do not optimize generated code
#  -std=gnu99  use the GNU99 standard language definition
CFLAGS = -g -Wall -O0 -std=gnu99

# The LDFLAGS variable sets flags for linker
#  -lm   says to link in libm (the math library)
LDFLAGS = -lm

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

.PHONY: clean all

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

Example – Source Files

myadd.h (called a header file)

```
#ifndef _MYADD_H_
#define _MYADD_H_

int add(int x, int y);

#endif
```

myadd.c

```
#include "myadd.h"
int add(int x, int y)
{
    return x + y;
}
```

main.c

```
#include "myadd.h"

int main(int argc, char **argv)
{
    int a = 1;
    int b = 2;

    c = add(a,b);

    printf("%d + %d = %d", a, b, c);
}
```

Example – Makefile

```
# This Makefile should be used as a template for future Makefiles.
# It’s heavily commented, so hopefully you can understand what each
# line does.

# We’ll use gcc for C compilation and g++ for C++ compilation
CC  = gcc
CXX = g++

# Let’s leave a place holder for additional include directories
INCLUDES =

# Compilation options:
# -g for debugging info and -Wall enables all warnings
CFLAGS   = -g -Wall $(INCLUDES)
CXXFLAGS = -g -Wall $(INCLUDES)

# Linking options:
# -g for debugging info
LDFLAGS = -g

# List the libraries you need to link with in LDLIBS
# For example, use "-lm" for the math library
LDLIBS =

# The 1st target gets built when you type "make".
# It’s usually your executable.  ("main" in this case.)
#
# Note that we did not specify the linking rule.
# Instead, we rely on one of make’s implicit rules:
#
#     $(CC) $(LDFLAGS) <all-dependent-.o-files> $(LDLIBS)
#
# Also note that make assumes that main depends on main.o,
# so we can omit it if we want to.
```

```
main: main.o myadd.o

# main.o depends not only on main.c, but also on myadd.h because
# main.c includes myadd.h.  main.o will get recompiled if either
# main.c or myadd.h get modified.
#
# make already knows main.o depends on main.c, so we can omit main.c
# in the dependency list if we want to.
#
# make uses the following implicit rule to compile a .c file into a .o
# file:
#
#     $(CC) -c $(CFLAGS) <the-.c-file>
#
main.o: main.c myadd.h

# And myadd.o depends on myadd.c and myadd.h.
myadd.o: myadd.c myadd.h

# Always provide the "clean" target that removes intermediate files.
# What you remove depend on your choice of coding tools
# (different editors generate different backup files for example).
#
# And the "clean" target is not a file name, so we tell make that
# it’s a "phony" target.
.PHONY: clean
clean:
        rm -f *.o a.out core main

# "all" target is useful if your Makefile builds multiple programs.
# Here we’ll have it first do "clean", and rebuild the main target.
.PHONY: all
all: clean main
```

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
  - The Linker
- Make and Makefiles
  - Overview of Make
  - Makefiles from scratch
  - Template for your Makefiles

xkcd.com/303/

Next Time: Assembly language