

Final C Details, Build Tools

CSE 333 Spring 2019

Instructor: Justin Hsia

Teaching Assistants:

Aaron Johnston

Andrew Hu

Daniel Snitkovskiy

Forrest Timour

Kevin Bi

Kory Watson

Pat Kosakanchit

Renshu Gu

Tarkan Al-Kazily

Travis McGaha

Administrivia

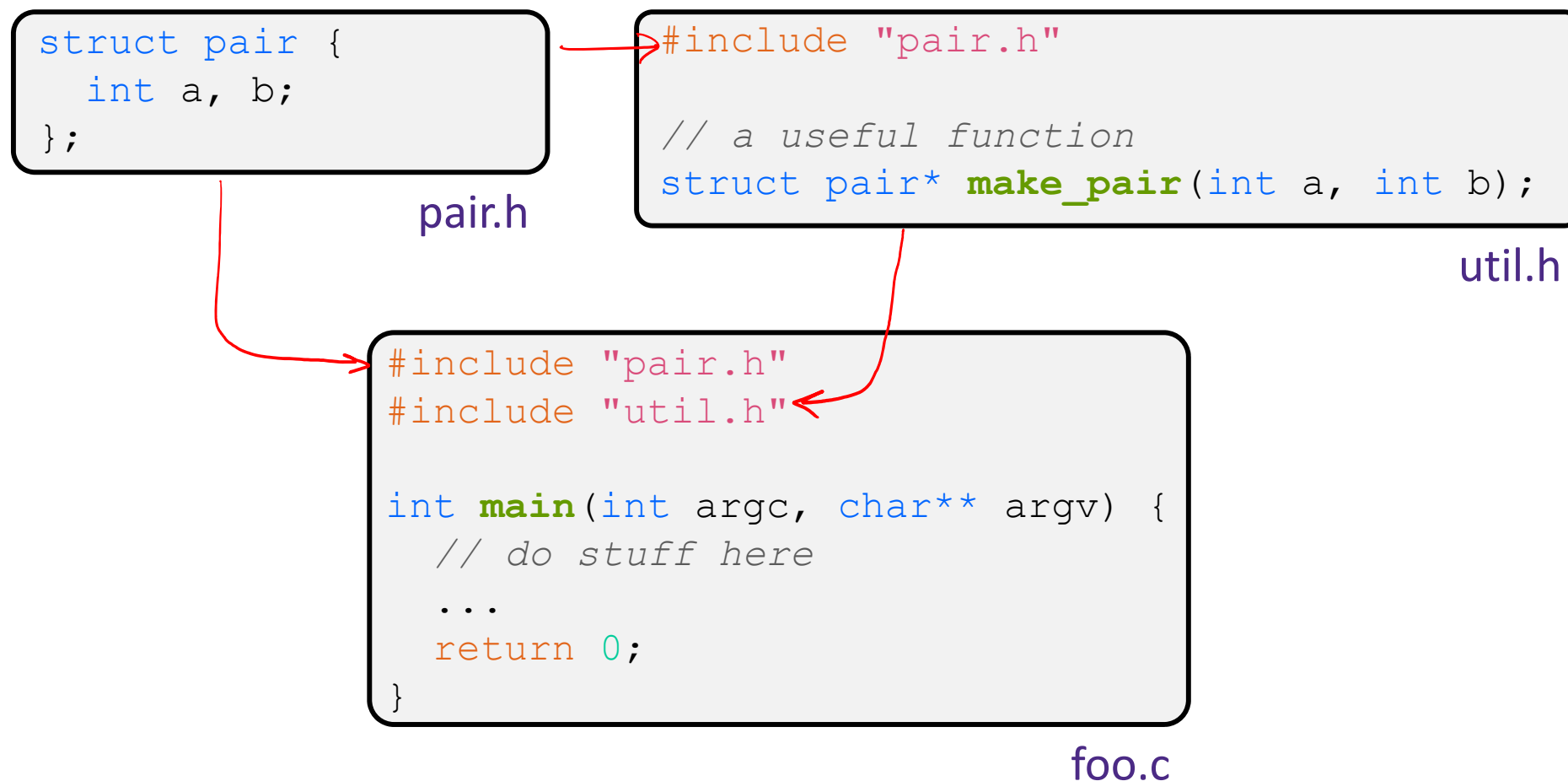
- ❖ Exercise 5 posted yesterday, due Monday
- ❖ Homework 1 due on Thursday (4/18)
 - Watch that `hashtable.c` doesn't violate the modularity of `ll.h`
 - Watch for pointer to local (stack) variables
 - Keep track of types of things – draw memory diagrams!
 - Use a debugger (*e.g.* `gdb`) if you're getting segfaults
 - Advice: clean up “to do” comments, but leave “step #” markers for graders
 - Late days: don't tag `hw1-final` until you are really ready
 - Extra Credit: if you add unit tests, put them in a new file and adjust the Makefile

Lecture Outline

- ❖ **Header Guards and Preprocessor Tricks**
- ❖ **Visibility of Symbols**
 - `extern, static`
- ❖ **Make and Build Tools**

A Problem with #include

- ❖ What happens when we compile `foo.c`?

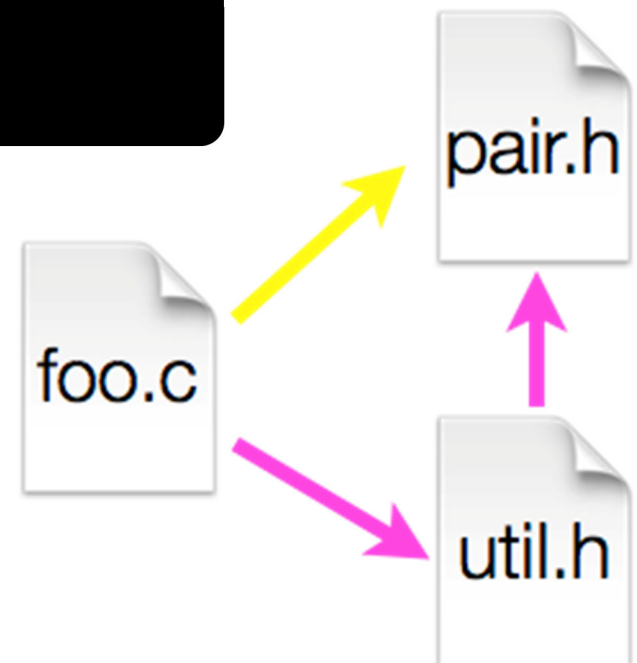


A Problem with #include

- ❖ What happens when we compile `foo.c`?

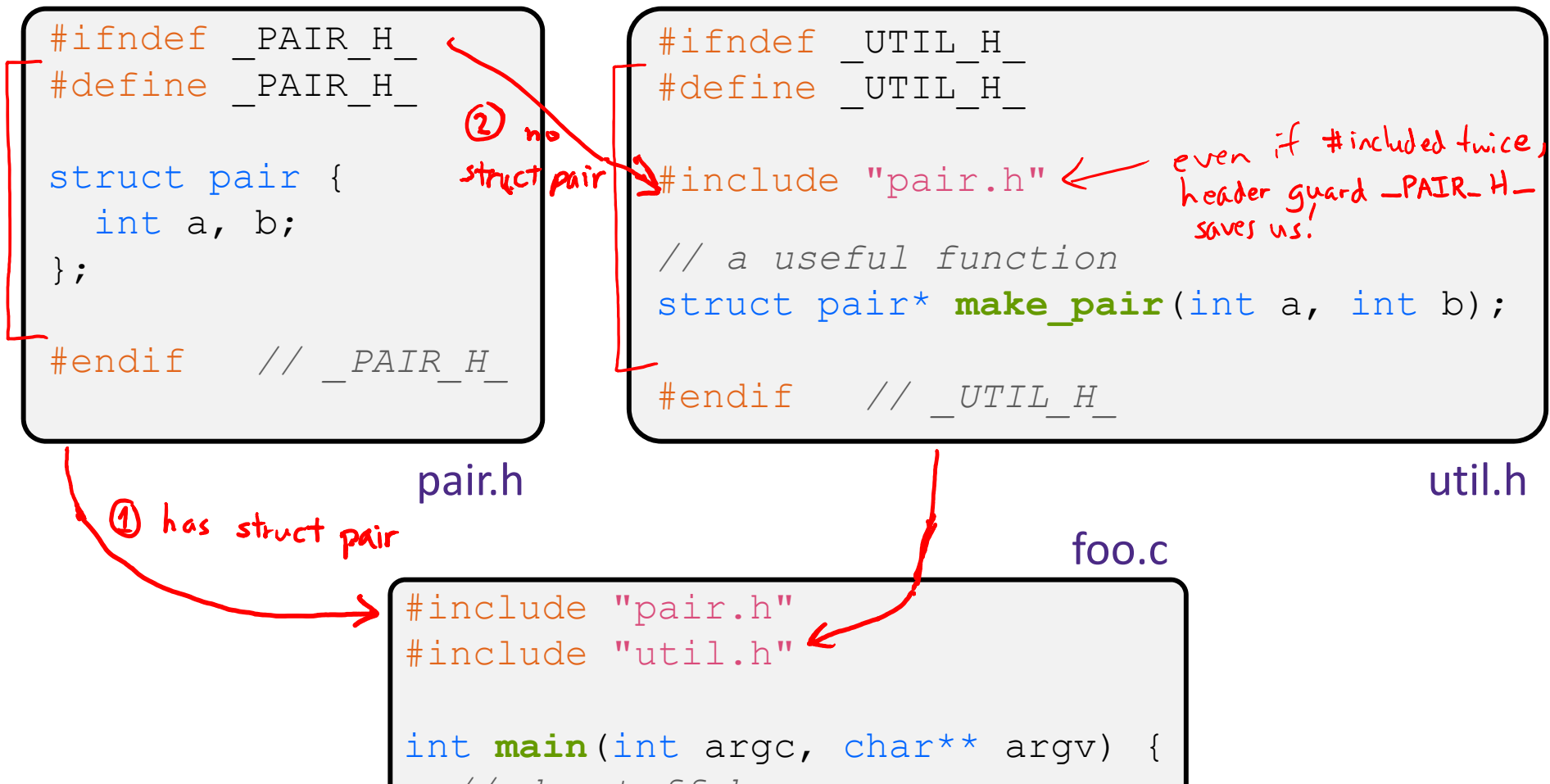
```
bash$ gcc -Wall -g -o foo foo.c
In file included from util.h:1:0,
    from foo.c:2:
pair.h:1:8: error: redefinition of 'struct pair'
  struct pair { int a, b; };
    ^
In file included from foo.c:1:0:
pair.h:1:8: note: originally defined here
  struct pair { int a, b; };
    ^
```

- ❖ `foo.c` includes `pair.h` twice!
 - Second time is indirectly via `util.h`
 - Struct definition shows up twice
 - Can see using `cpp`



Header Guards

- ❖ A standard C Preprocessor trick to deal with this
 - Uses macro definition (`#define`) in combination with conditional compilation (`#ifndef` and `#endif`)



Other Preprocessor Tricks

- ❖ A way to deal with “magic constants”

```
int globalbuffer[1000];

void circalc(float rad,
             float* circumf,
             float* area) {
    *circumf = rad * 2.0 * 3.1415;
    *area = rad * 3.1415 * 3.1415;
}
```

Bad code

(littered with magic constants)

```
#define BUFSIZE 1000
#define PI 3.14159265359

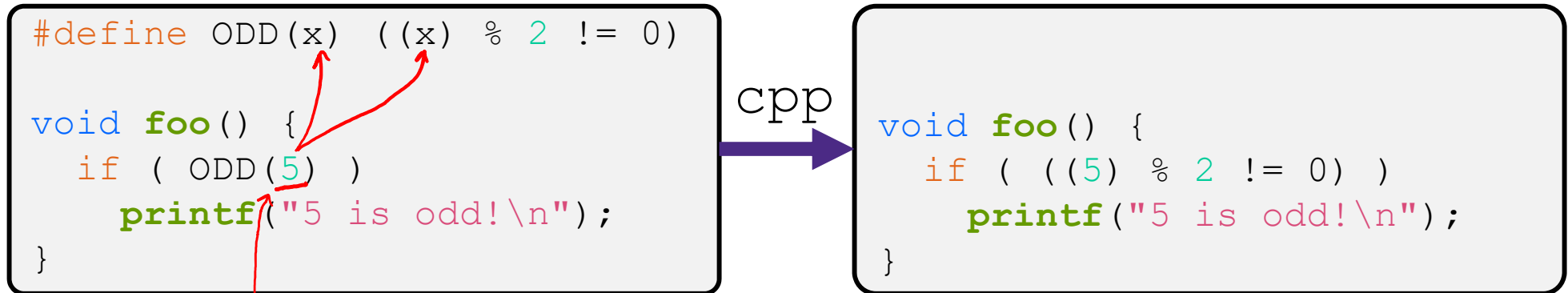
int globalbuffer[BUFSIZE];

void circalc(float rad,
             float* circumf,
             float* area) {
    *circumf = rad * 2.0 * PI;
    *area = rad * PI * PI;
}
```

Better code

Macros

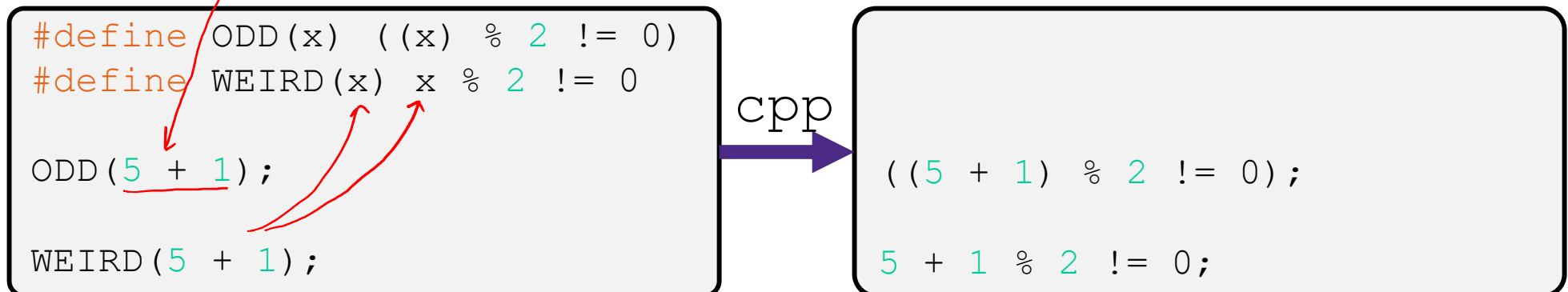
- ❖ You can pass arguments to macros



treated as just text

- ❖ Beware of operator precedence issues!

- Use parentheses



Conditional Compilation

`#ifdef` = "if defined"
`#ifndef` = "if not defined"

- ❖ You can change what gets compiled
 - In this example, `#define TRACE` before `#ifdef` to include debug `printfs` in compiled code

```
#ifdef TRACE
#define ENTER(f) printf("Entering %s\n", f);
#define EXIT(f) printf("Exiting %s\n", f);
#else
#define ENTER(f)
#define EXIT(f)
#endif

// print n
void pr(int n) {
    ENTER("pr");
    printf("\n = %d\n", n);
    EXIT("pr");
}
```

ifdef.c

Defining Symbols

- ❖ Besides `#defines` in the code, preprocessor values can be given as part of the `gcc` command:

```
bash$ gcc -Wall -g -DTRACE -o ifdef ifdef.c
```

-D define

-U undefine

- ❖ `assert` can be controlled the same way – defining `NDEBUG` causes `assert` to expand to “empty”
 - It’s a macro – see `assert.h`

```
bash$ gcc -Wall -g -DNDEBUG -o faster useassert.c
```

Peer Instruction Question

- ❖ What will happen when we try to compile and run?
 - Vote at <http://PollEv.com/justinh>

```
bash$ gcc -Wall -DFOO -DBAR -o condcomp condcomp.c
bash$ ./condcomp
```

FOO and BAR are defined

A. Output "333"

B. Output "334"

C. Compiler message about EVEN

D. Compiler message about BAZ

E. We're lost...

```
#include <stdio.h>
#ifdef FOO ← yes
#define EVEN(x) !(x%2)
#endif
#ifndef DBAR ← no
#define BAZ 333
#endif

int main(int argc, char** argv) {
    int i = EVEN(42) + BAZ;
    printf("%d\n", i);
    return 0;
}
```

evaluates to !0 = 1

!(42%2) ← 1

333

Lecture Outline

- ❖ Header Guards and Preprocessor Tricks
- ❖ **Visibility of Symbols**
 - **extern, static**
- ❖ Make and Build Tools

Namespace Problem

- ❖ If we define a global variable named “counter” in one C file, is it visible in a different C file in the same program?
 - Yes, if you use external linkage
 - The name “counter” refers to the same variable in both files
 - The variable is *defined* in one file and *declared* in the other(s)
 - When the program is linked, the symbol resolves to one location
 - No, if you use internal linkage
 - The name “counter” refers to a different variable in each file
 - The variable must be *defined* in each file
 - When the program is linked, the symbols resolve to two locations

External Linkage

this program will print:

1
(b): counter = 2
2

- ❖ `extern` makes a *declaration* of something externally-visible

```
#include <stdio.h>

// A global variable, defined and
// initialized here in foo.c.
// It has external linkage by
// default.
int counter = 1;

int main(int argc, char** argv) {
    printf("%d\n", counter);
    bar();
    printf("%d\n", counter);
    return 0;
}
```

foo.c

```
#include <stdio.h>

// "counter" is defined and
// initialized in foo.c.
// Here, we declare it, and
// specify external linkage
// by using the extern specifier.
extern int counter;

void bar() {
    counter++;
    printf("(b): counter = %d\n",
           counter);
}
```

bar.c

Internal Linkage

this program will print:

```
1
(b): counter = 101
1
```

- ❖ `static` (in the global context) restricts a definition to visibility within that file

```
#include <stdio.h>

// A global variable, defined and
// initialized here in foo.c.
// We force internal linkage by
// using the static specifier.
static int counter = 1;

int main(int argc, char** argv) {
    printf("%d\n", counter);
    bar();
    printf("%d\n", counter);
    return 0;
}
```

foo.c

```
#include <stdio.h>

// A global variable, defined and
// initialized here in bar.c.
// We force internal linkage by
// using the static specifier.
static int counter = 100;

void bar() {
    counter++;
    printf("(b): counter = %d\n",
           counter);
}
```

bar.c

Function Visibility

```
// By using the static specifier, we are indicating
// that foo() should have internal linkage. Other
// .c files cannot see or invoke foo().
```

```
static int foo(int x) {
    return x*3 + 1;
}
```

```
// Bar is "extern" by default. Thus, other .c files
// could declare our bar() and invoke it.
```

```
int bar(int x) {
    return 2*foo(x);
}
```

*bar() can invoke foo() because
in same file*

bar.c

```
#include <stdio.h>
```

```
extern int bar(int x); // "extern" is default, usually omit
```

not explicitly needed, but indicates that definition is elsewhere

```
int main(int argc, char** argv) {
    printf("%d\n", bar(5));
    return 0;
}
```

main.c

Linkage Issues

- ❖ Every global (variables and functions) is `extern` by default
 - Unless you add the `static` specifier, if some other module uses the same name, you'll end up with a collision!
 - Best case: compiler (or linker) error
 - Worst case: stomp all over each other

- ❖ It's good practice to:
 - Use `static` to “defend” your globals
 - Hide your private stuff!
 - Place external declarations in a module's header file
 - Header is the public specification

Static Confusion...

- ❖ C has a *different* use for the word “`static`”: to create a persistent *local* variable
 - The storage for that variable is allocated when the program loads, in either the `.data` or `.bss` segment (*Static Data*)
 - Retains its value across multiple function invocations

```
void foo() {
    static int count = 1; // persists
    printf("foo has been called %d times\n", count++);
}

void bar() {
    int count = 1; // re-initialized each time
    printf("bar has been called %d times\n", count++);
}

int main(int argc, char** argv) {
    foo(); foo(); bar(); bar(); return 0;
} 1 times  2 times  1 times  1 times
```

static_extent.c

Additional C Topics

❖ Teach yourself!

- **man pages** are your friend!
- String library functions in the C standard library
 - `#include <string.h>`
 - `strlen()`, `strcpy()`, `strdup()`, `strcat()`, `strcmp()`, `strchr()`, `strstr()`, ...
 - `#include <stdlib.h>` or `#include <stdio.h>`
 - `atoi()`, `atof()`, `sprintf()`, `scanf()`
- How to declare, define, and use a function that accepts a variable-number of arguments (`varargs`)
- `unions` and what they are good for
- `enums` and what they are good for
- Pre- and post-increment/decrement
- Harder: the meaning of the “`volatile`” storage class

Lecture Outline

- ❖ Header Guards and Preprocessor Tricks
- ❖ Visibility of Symbols
 - `extern, static`
- ❖ **Make and Build Tools**

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=c11 -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)

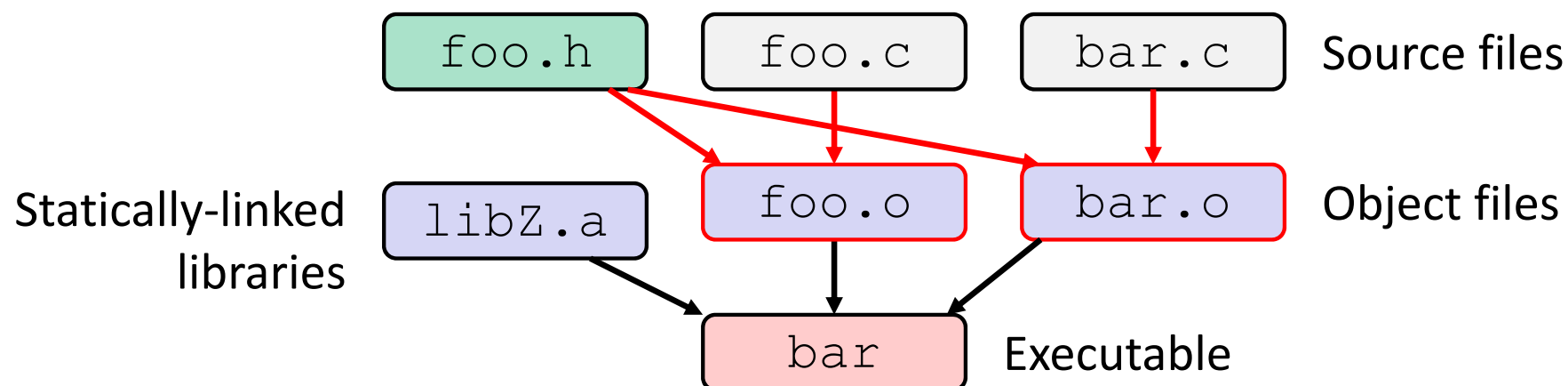
“Real” Build Process

- ❖ On larger projects, you can't or don't want to have one big (set of) command(s) that redoes everything every time you change anything:
 - 1) 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) If source files have multiple outputs (*e.g.* javadoc), you'd 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
 - 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

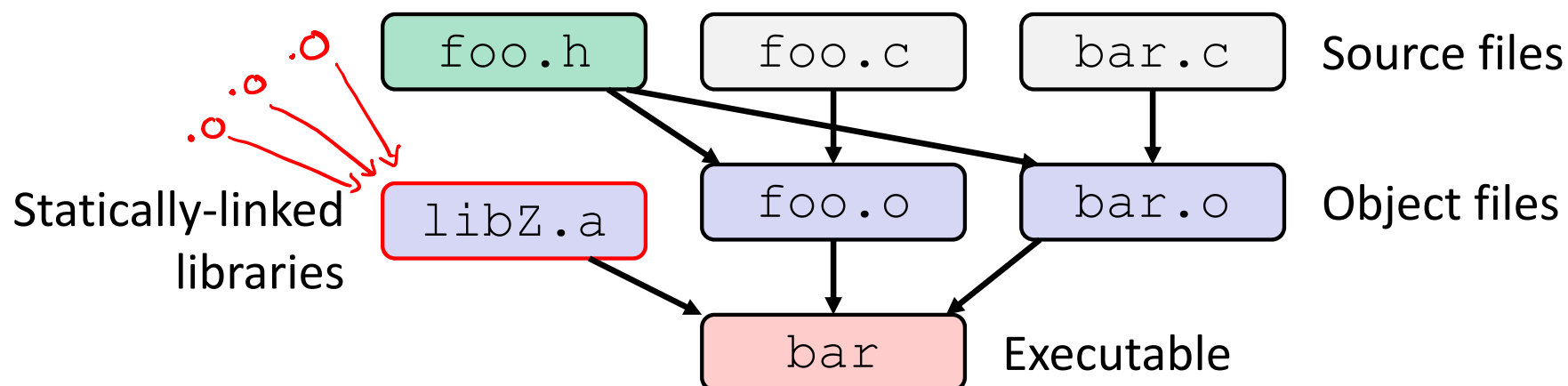
- ❖ The “theory” behind avoiding unnecessary compilation is a *dependency dag* (directed, acyclic graph)
- ❖ To create a target t , you need sources s_1, s_2, \dots, s_n and a command c that directly or indirectly uses the sources
 - It t is newer than every source (file-modification times), assume there 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



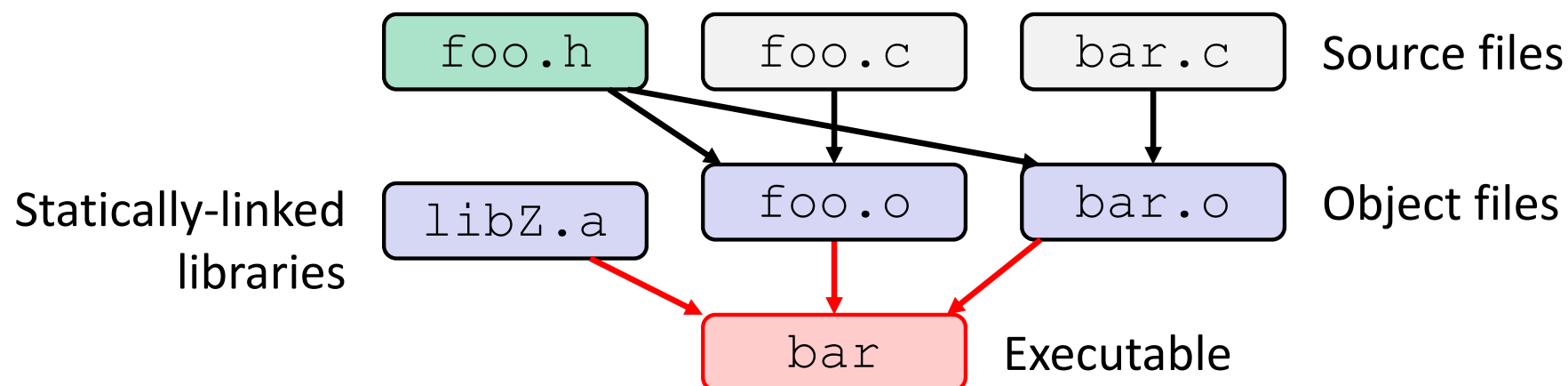
- ❖ Compiling a `.c` creates a `.o` – the `.o` depends on the `.c` and all included files (`.h`, recursively/transitively)

Theory Applied to C



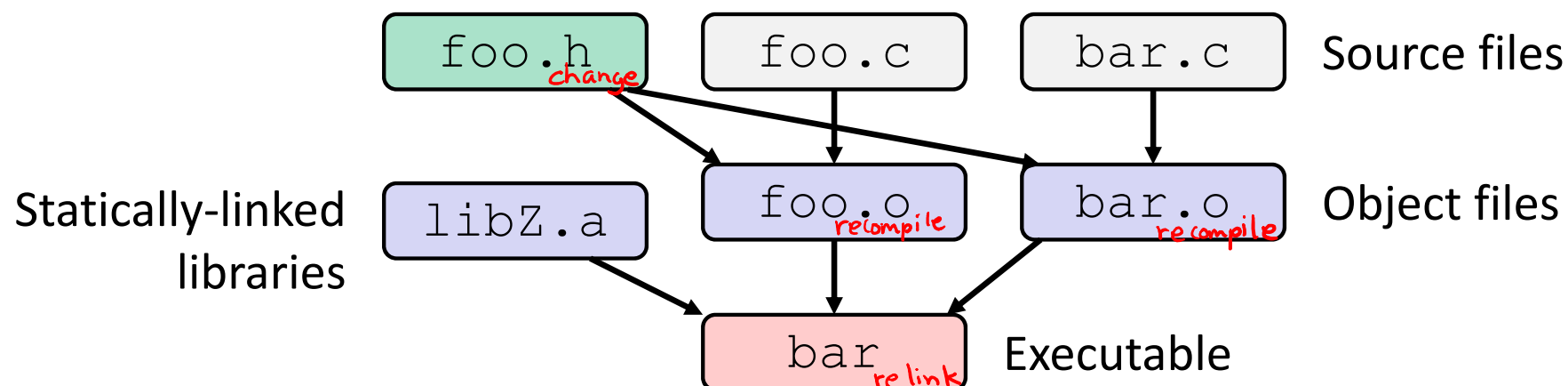
- ❖ Compiling a `.c` creates a `.o` – the `.o` depends on the `.c` and all included files (`.h`, recursively/transitively)
- ❖ An archive (library, `.a`) depends on included `.o` files

Theory Applied to C



- ❖ Compiling a `.c` creates a `.o` – the `.o` depends on the `.c` and all included files (`.h`, recursively/transitively)
- ❖ An archive (library, `.a`) depends on included `.o` files
- ❖ Creating an executable (“linking”) depends on `.o` files and archives
 - Archives linked by `-L<path> -l<name>`
(*e.g.* `-L. -lfoo` to get `libfoo.a` from current directory)

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!

Extra Exercise #1

- ❖ Write a program that:
 - Prompts the user to input a string (use `fgets()`)
 - Assume the string is a sequence of whitespace-separated integers (*e.g.* "5555 1234 4 5543")
 - Converts the string into an array of integers
 - Converts an array of integers into an array of strings
 - Where each element of the string array is the binary representation of the associated integer
 - Prints out the array of strings

Extra Exercise #2

- ❖ Modify the linked list code from Lecture 5 Extra Exercise #1
 - Add static declarations to any internal functions you implemented in `linkedlist.h`
 - Add a header guard to the header file
 - Write a `Makefile`
 - Use Google to figure out how to add rules to the `Makefile` to produce a library (`liblinkedlist.a`) that contains the linked list code