Final C Details
CSE 333

Instructor: Hannah C. Tang

Teaching Assistants:

Deeksha Vatwani	Hannah Jiang	Jen Xu
Justin Tysdal	Leanna Nguyen	Sayuj Shahi
Wei Wu	Yiqing Wang	Youssef Ben Taleb

Lecture Outline

- **Header Guards and Preprocessor Tricks**
- Visibility of Symbols
  - `extern`, `static`

Yet More hw1 Hints

- Debugging
  - Use a debugger (*e.g.* `gdb`) if you're getting segfaults – fix reality!
  - Write and run little tests to track down problems (don't kill lots of time trying to debug large test_suite code)
  - gdb hint: What if `Verify333` fails? How can you debug it? Answer: look at the `Verify333` macro (`#define`), figure out what function it calls on failure, and put a breakpoint there

- Late days: don't tag `hw1-final` until you are really ready (then check your work – clone repo – and re-read assignment to be sure you didn't miss anything!)

Poll Everywhere

pollev.com/uwcse333

- What are the contents of `foo.c`?

```
struct pair {
  int a, b;
};
```

pair.h

```
#include "pair.h"

// a useful function
struct pair* make_pair(int a, int b);
```

util.h

```
#include "pair.h"
#include "util.h"

int main(int argc, char** argv) {
  // do stuff here
  ...
  return EXIT_SUCCESS;
}
```

foo.c

An #include Problem

- 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`)

```
#ifndef PAIR_H_
#define PAIR_H_


struct pair {
  int a, b;
};


#endif   // PAIR_H_
```

pair.h

```
#ifndef UTIL_H_
#define UTIL_H_

#include "pair.h"

// a useful function
struct pair* make_pair(int a,
                       int b);


#endif   // UTIL_H_
```

util.h

Other Preprocessor Tricks

- A way to deal with "magic numbers" (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

```
#define ODD(x) ((x) % 2 != 0)

void foo() {
  if ( ODD(5) )
    printf("5 is odd!\n");
}
```

cpp

```
void foo() {
  if ( ((5) % 2 != 0) )
    printf("5 is odd!\n");
}
```

- Beware of operator precedence issues!
  - Use parentheses

```
#define ODD(x) ((x) % 2 != 0)
#define WEIRD(x) x % 2 != 0

ODD(5 + 1);

WEIRD(5 + 1);
```

cpp

```
((5 + 1) % 2 != 0);

5 + 1 % 2 != 0;
```

Conditional Compilation

- You can change what gets compiled
  - In this example, `#define TRACE` before `#ifdef` to include debug `printf`s 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
```

- 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
```

Poll Everywhere

pollev.com/uwcse333

- What will happen when we try to compile and run?

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

A. Outputs "333"

B. Outputs "334"

C. Compiler message about EVEN

D. Compiler message about BAZ

E. We're lost…

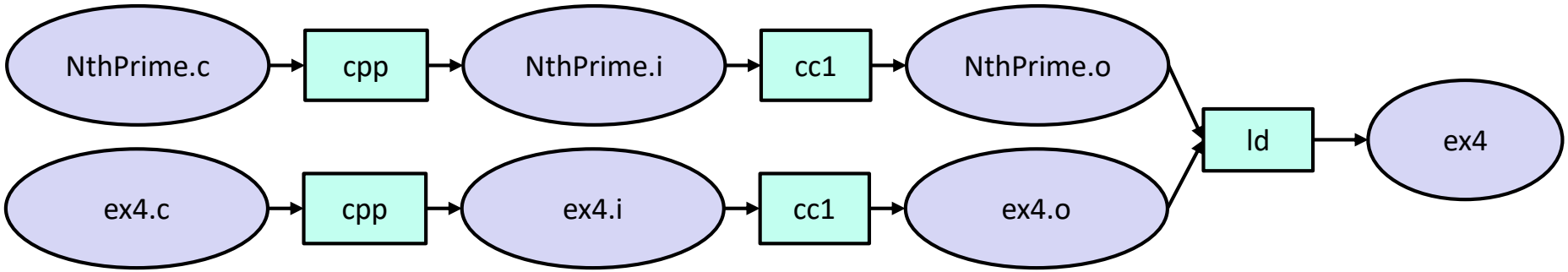
```c
#include <stdio.h>
#ifdef FOO
#define EVEN(x) !(x%2)
#endif
#ifndef DBAR
#define BAZ 333
#endif

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

Poll Everywhere

pollev.com/uwcse333

- Assume `NthPrime.c` and `ex4.c`, as in our most-recently completed exercise



- … except we are now precalculating the first 20 primes.
  - What are the contents of each .o, assuming the following modification to `NthPrime.h`?

```
#define NUM_PRECALC 20
int16_t kPrecalculated[NUM_PRECALC] =
    {2, 3, 5, 7, /* etc */ };

int16_t NthPrime(int16_t n);
```

Lecture Outline

- Header Guards and Preprocessor Tricks
- **Visibility of Symbols**
  - `extern`, `static`

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

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

```c
#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 EXIT_SUCCESS;
}
```

foo.c

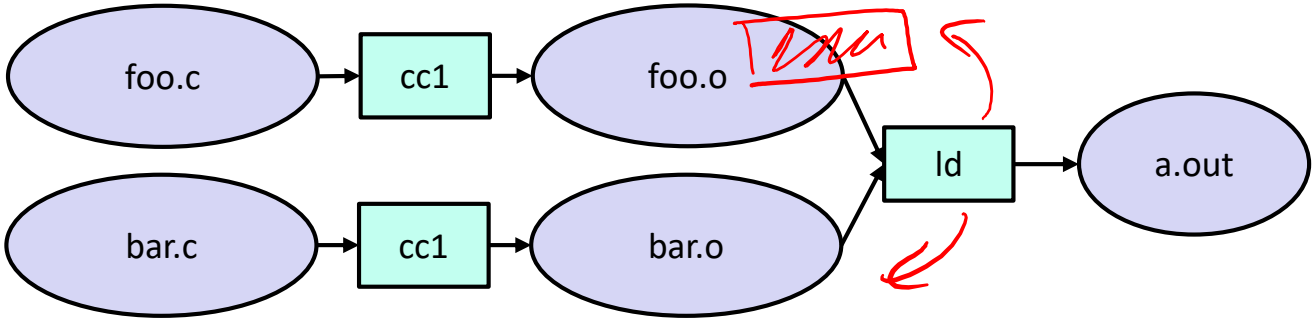
```c
#include <stdio.h>

// "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

External Linkage




```c
#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 EXIT_SUCCESS;
}
```

foo.c

```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

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

```c
#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 EXIT_SUCCESS;
}
```

foo.c

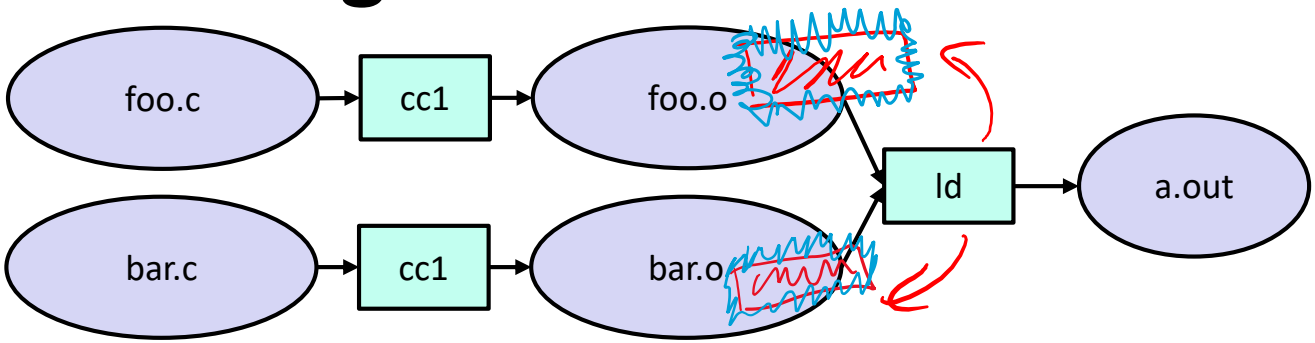
```c
#include <stdio.h>

// 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

Internal Linkage




```c
#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 EXIT_SUCCESS;
}
```

foo.c

```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

Poll Everywhere

pollev.com/uwcse333

- Coming back to our `NthPrime.h`, `NthPrime.c`, and `ex4.c`, example, do either of these successfully link?

```
#define NUM_PRECALC 20
static int16_t
kPrecalculated[NUM_PRECALC] =
    {2, 3, 5, 7, /* etc */ };

int16_t NthPrime(int16_t n);
```

NthPrime.static.h

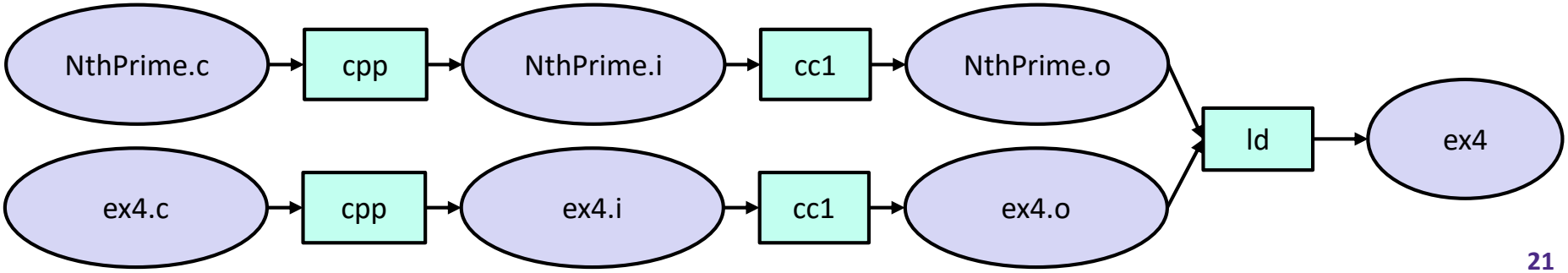
```
extern int16_t kPrecalculated[];



int16_t NthPrime(int16_t n);
```

NthPrime.extern.h

- *(you can assume NthPrime.c and ex4.c are unchanged)*



Closing Thoughts

- Don't do either of these with an unchanged NthPrime.c!

```
#define NUM_PRECALC 20
static int16_t
kPrecalculated[NUM_PRECALC] =
    {2, 3, 5, 7, /* etc */ };

int16_t NthPrime(int16_t n);
```

NthPrime.static.h

```
extern int16_t kPrecalculated[];



int16_t NthPrime(int16_t n);
```

NthPrime.extern.h

- Static variable in the header: every file that #includes the header will have its own private copy of the variable
  - Unnecessary data duplication!
- Extern variable in the header: the accompanying .c file must define the extern'ed variable for successful linkage

Function Visibility

bar.c

```c
// 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);
}
```

main.c

```c
#include <stdio.h>

extern int bar(int x); // "extern" is default, usually omit.
                       // should be in .h file, but effect is same

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

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
  - Retains its value across multiple function invocations
  - Confusing! Don't use!! (But you may see it ☹)

```
void foo() {
  static int count = 1;  // static var, not auto!!
  printf("foo has been called %d times\n", count++);
}

void bar() {
  int count = 1;
  printf("bar has been called %d times\n", count++);
}

int main(int argc, char** argv) {
  foo(); foo(); bar(); bar(); return EXIT_SUCCESS;
}
```

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(), sprint(), sscanf()
  - 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

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