Preprocessor and Linking

CSE 333 Autumn 2019

Instructor: Hannah C. Tang

Teaching Assistants:

Dao Yi	Farrell Fileas	Lukas Joswiak
Nathan Lipiarski	Renshu Gu	Travis McGaha
Yibo Cao	Yifan Bai	Yifan Xu

Poll Everywhere

pollev.com/cse333

About how long did Exercise 4 take?

A. 0-1 Hours
B. 1-2 Hours
C. 2-3 Hours
D. 3-4 Hours
E. 4+ Hours
F. I prefer not to say

Administrivia

- HW 1 due on Thursday (10/10) @ 9pm – 
  - Watch that `HashTable` doesn't violate the modularity of `LinkedList`
  - Watch for pointer to local (stack) variables
  - ***Draw memory diagrams*!**
  - Use a debugger (*e.g.* `gdb`) and valgrind
  - Please leave "STEP #" markers for graders!
  - Remember to tag `hw1-final` (we'll figure out late days)
  - Extra Credit: if you add unit tests, put them in a new file and adjust the Makefile

Administrivia

- Exercise 5 out today, due Wednesday morning
- *No exercise due Friday!* Exercise 6 will be released on Friday (10/11) and due the following Monday (10/14)

Lecture Outline

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

A Problem with `#include`

- What happens when we compile `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 0;
}
```

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

Preprocessor state

PAIR_H ✓
UTIL_H ✓

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

pair.h

```
#ifndef PAIR_H_
#define PAIR_H_

struct pair {
  int a, b;
};

#endif   // PAIR_H_
```

util.h

```
#ifndef UTIL_H_
#define UTIL_H_

#include "pair.h"

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

#endif   // UTIL_H_
```

empty file

foo.c

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

int main(int argc, char** argv) {
  // do stuff here
```

Other Preprocessor Tricks: Constants

- Create tokens 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

Other Preprocessor Tricks: 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;
```

Other Preprocessor Tricks: Defining Tokens at Build Time

- Besides `#define`s in the code, preprocessor tokens 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
```

which is why we reimplemented assert as Verify333 (we want debug symbols AND assertions)

Other Preprocessor Tricks: Defining Tokens at Build Time

- 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

Putting it Together: Preprocessor Example

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

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

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

int main(int argc, char** argv) {
  int32_t i = EVEN(42) + BAZ;
  printf("%d\n", i);
  return 0;
}
```

condcomp.c

Lecture Outline

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

Global Variables Thus Far …

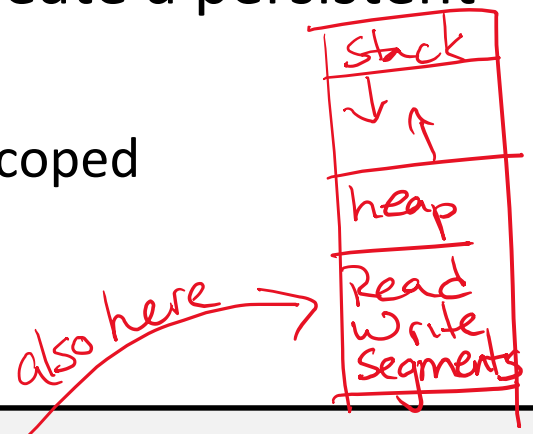
- The storage for global variables is allocated when the program loads, in either the `.data` or `.bss` segment
- Retains its value across multiple function invocations


```
int global_count = 1;

void foo() {
  global_count++;
  printf("global_count: %d\n", global_count);
}

int main(int argc, char** argv) {
  foo();
  printf("global_count: %d\n", global_count);
  global_count++;
  return 0;
}
```

globals.c

Static Thus Far …

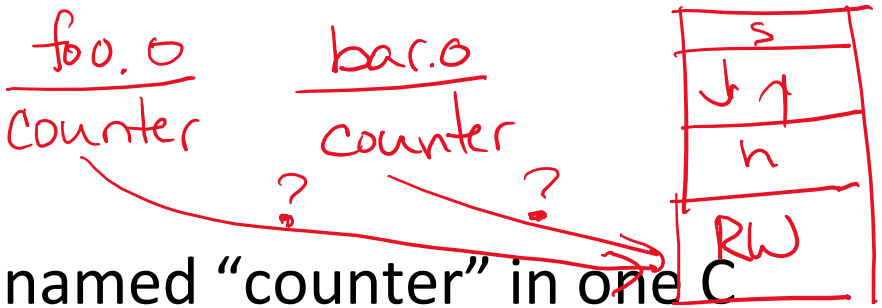
- C uses for the word "`static`" to create a persistent (sometimes *locally scoped*) variable
  - Just like global variables, except locally scoped




```
void foo() {
  static int count = 1;
  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 0;
}
```

static_extent.c

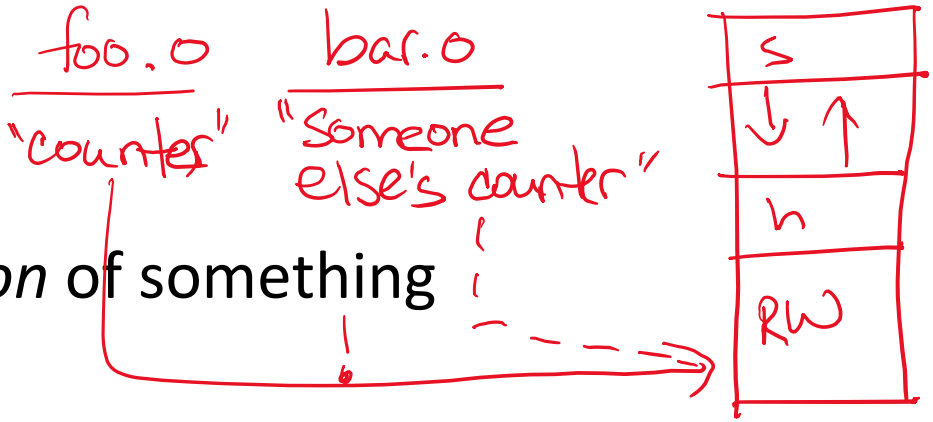
Namespace Problem



- 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




```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 0;
}
```

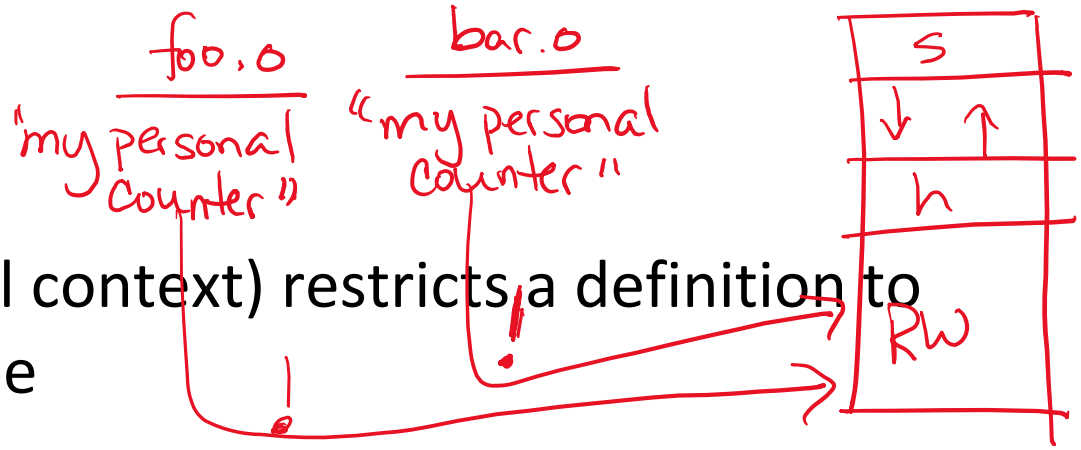
foo.c

```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("(bar): counter = %d\n",
         counter);
}
```

bar.c

extern is the "variable's version" of a function prototype!

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 0;
}
```

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("(bar): counter = %d\n",
         counter);
}
```

bar.c

What About Functions?

- Can use function declarations -- not definitions -- to bypass linkage issues

Function Visibility

no specifier

bar.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 exported by default (ie, neither "extern" nor
// "static").  Thus, other .c files could declare
// our bar() and invoke it.
int bar(int x) {
  return 2*foo(x);
}
```

"my personal foo"

"my bar that's available if others want it"

main.c

```
#include <stdio.h>

extern int bar (int x);

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

"somebody else's bar"

Linkage Issues

- Every global – both variable definitions and function definitions – are *exported* 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" things you want to keep private in the .c
  - Place external *declarations* in a module's header file
    - Header is the public specification
  - Never *define* variables or functions in the header, even if you use `static`

Additional C Topics

- **man pages** are your friend!

- Other fun topics to read about:
  - 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