

Final C Details

CSE 333 Spring 2020

Instructor: Hal Perkins

Teaching Assistants:

Ramya Challa

Greg Guo

Travis McGaha

Cosmo Wang

Haoran Yu

Mengqui Chen

Zachary Keyes

Arjun Singh

Yifan Xu

Velocity Yu

John Depaszthory

CJ Lin

Guramrit Singh

Robin Yang

Administrivia

- ❖ Today: C wrapup, Makefiles
- ❖ Exercise 5 posted yesterday afternoon; due Monday morning

Administrivia

- ❖ HW1 due Thursday night
 - Write and run little tests to track down problems (don't kill lots of time debugging 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
- ❖ Remember: the only supported systems for the class are the Allen School Linux machines (workstations, attus, home VM). You should be working on those systems and the projects you build **must** work there.
 - We do not have the cycles to try to support other Unix-like things or chase bugs due to configuration or software differences (including file transfers to/from Windows systems)
 - “Bug” reports caused by other configurations that do not identify the other system are not complete or appropriate

Administrivia

❖ Homework 1 due on Thursday

- Advice: be *sure* to read headers carefully while implementing
- Advice: use `git add/commit/push` often to save your work
 - But don't use gitlab to copy files so you can edit them on one system and run on another – just work in CSE Linux environment (VM or remotely)
- Watch that `HashTable.c` doesn't violate the modularity of `LinkedList.h` (i.e., don't mess with private implementation stuff)
- Watch for pointers to local (stack) variables (`0x7fff...` addresses)
- Keep track of types of things – draw memory diagrams
- Use a debugger (e.g. `gdb`) if you're getting segfaults – fix reality!
- Advice: leave “step #” markers to help graders navigate
- 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`

An #include Problem

❖ 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;  
}
```

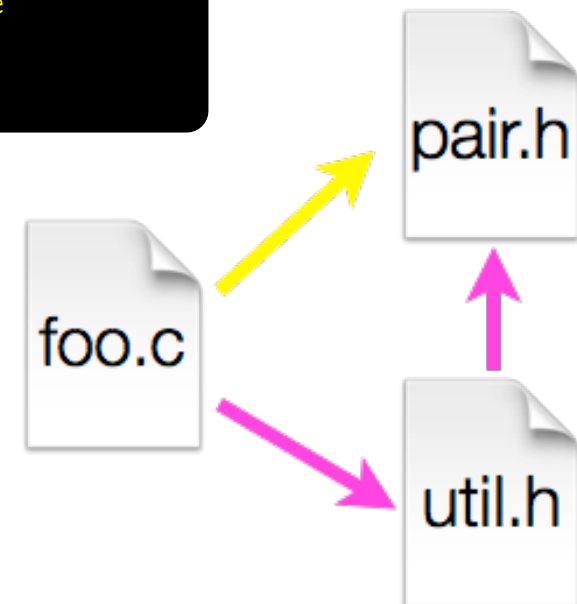
foo.c

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 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
```

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

```
#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

- ❖ `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.c

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

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

```
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

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