Final C Details
CSE 333 Autumn 2018

Instructor: Hal Perkins

Teaching Assistants:
Tarkan Al-Kazily  Renshu Gu  Travis McGaha
Harshita Neti     Thai Pham       Forrest Timour
Soumya Vasisht  Yifan Xu
Administrivia

- Today: C wrapup, start file I/O & system calls
- Exercise 6 posted today, due Wednesday morning
- Wed.: More system calls, overview of POSIX (system) library
- 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
Discussion board and email hints

Please help your readers (both for cse333 and elsewhere)

- Use descriptive titles and provide enough context in the question so readers don’t need to go on a treasure hunt
  - Not great: Subject: About exercise
  - Only ε better: Subject: exercise
  - Much better: Subject: is sizeof(type) ok in ex3?

- Please don’t post screenshots of text
  - Hard to read and/or require opening an extra window
  - If it’s text, copy and paste the text(!) (drag to select in terminal or dialog boxes)
  - Images are fine if they actually are relevant to the posting

- Your readers thank you for your help 😊
Lecture Outline

- Header Guards and Preprocessor Tricks
- Visibility of Symbols
  - extern, static
An `#include` Problem

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

```c
#include "pair.h"

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

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

int main(int argc, char** argv) {
    // do stuff here
    ...
    return 0;
}
```

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

#include "pair.h"
```
An \#include Problem

- What happens when we compile \texttt{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; };
    ^
```

- \texttt{foo.c} \texttt{includes} \texttt{pair.h} twice!
  - Second time is indirectly via \texttt{util.h}
  - Struct definition shows up twice
    - Can see using \texttt{cpp}
Header Guards

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

```c
#ifndef _PAIR_H_
#define _PAIR_H_

struct pair {
    int a, b;
};

#endif  // _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_
```

```c
pair.h
```

```c
util.h
```
Other Preprocessor Tricks

- A way to deal with “magic constants”

```c
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)

```c
#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

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

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

- Beware of operator precedence issues!
  - Use parentheses

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

ODD(5 + 1);
WEIRD(5 + 1);
```

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

```c
ODD(5 + 1);
WEIRD(5 + 1);
```

```c
((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` statements in compiled code

```c
#ifndef 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
  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
  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;
}
```

```
#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);
}
```

foo.c     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 0;
}
```

```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);
}
```

```
```
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);
}

#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;
}
Linkage Issues

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

- It’s good practice to:
  - Use \texttt{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 😞)

```c
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 0;
}
```

*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