CSE 333
Lecture 6 - final C details

Hal Perkins

Department of Computer Science & Engineering

University of Washington

Administrivia 1

Exercise 5 (fix and modularize buggy program) posted after section yesterday. Due before class Wed.

Exercise 6 (file I/O) out now, also due before class Wed.

HW1 still due next Thursday

No class Monday - July 4 holiday

Administrivia 2

HW1, due Thursday night

Watch that hashtable.c doesn't violate the modularity of ll.h

Watch for pointers to local (stack) variables - don't store in persistent data

What do you do if one of the test_suite tests fails and it's not obvious why?

Hints: segfault? use gdb (bt, …); make small tests; breakpoints in Verify333

Suggestion from past graders: clean up the "to do" comments, but if you can leave the "step 1", "step 2" markers so they can find things quickly

Extra credit: if you add unit tests, put them in a new file and adjust the makefile

Quiz: what is the late day policy?

If you decide to use a late day, don't tag hw1-final until you are really ready

Administrivia - Code Quality

Code quality ("style") **really** matters - and not just for homework

Rule #0: reader's time is ***much*** more important than writer's

Good comments are essential, clarity/understandibility is critical

Good comments ultimately save writer's time too!

Rule #1: match existing code

Rule #2: use tools. examples:

Compiler warnings: just fix them!

clint style warnings: fix most of them; be sure you understand anything you don't fix and can justify it (ok to have a type as malloc parameter or use readdir, not ok to have spaces instead of tabs or magic numbers instead of #define, etc., ...)

valgrind warnings: fix all of them unless you know why it's not an error (example: reading/printing uninitialized bytes in a debugging tool)

Agenda

Today's topics:

a few final C details

header guards and other preprocessor tricks

extern, static and visibility of symbols

some topics for you to research on your own

an #include problem

What happens when we compile foo.c?

```
typedef void *LinkedList;

// more declarations below
                                ll.h
```

```
                                ht.h
#include "ll.h"

typedef void *HashTable;

// A hypothetical function
LinkedList HTKeyList(HashTable t);
```

```
#include "ll.h"
#include "ht.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 ht.h:1,
                 from foo.c:2:
ll.h:1: error: redefinition of typedef 'LinkedList'
ll.h:1: note: previous declaration of 'LinkedList' was here
```
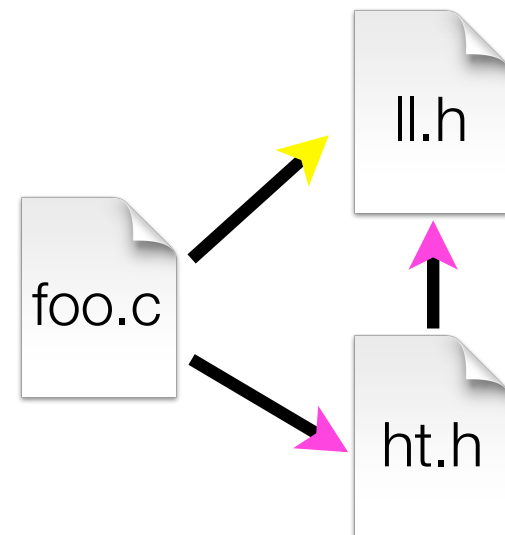
foo.c includes **ll.h** twice!

    2nd time is indirectly via **ht.h**

    so, typedef shows up twice!

    try using cpp to see this

header guards

A commonly used C preprocessor trick to deal with this

    uses macro definition (#define)

    uses conditional compilation (#ifndef and #endif)

```
                                                          ll.h
#ifndef _LL_H_
#define _LL_H_

typedef void *LinkedList;

// more definitions below

#endif  // _LL_H_
```

```
                                                          ht.h
#ifndef _HT_H_
#define _HT_H_

#include "ll.h"

typedef void *HashTable;

// A hypothetical function
LinkedList HTKeyList(HashTable t);

#endif  // _HT_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");
}
```

Be careful of precedence issues; use parenthesis:

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

ODD(5 + 1);

BAD(5 + 1);
```

cpp →

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

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

Conditional Compilation

You can change what gets compiled

```
#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 on the gcc command

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

assert is controlled the same way - #define NDEBUG and asserts expand to "empty" (it's a macro - see assert.h)

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

Namespace problem

If I define a global variable named "counter" in foo.c, is it visible in bar.c?

if you use **external linkage**: yes

the name "**counter**" refers to the same variable in both files

the variable is defined in one file, declared in the other(s)

when the program is linked, the symbol resolves to one location

if you use **internal linkage**: no

the name "**counter**" refers to different variables in each file

the variable must be defined in each file

when the program is linked, the symbols resolve to two locations

External linkage

```
#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

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

bar.c

Some gotchas

Every global (variables and functions) is extern by default

unless you write 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 (i.e., global) declarations in a module's header file

Extern, static functions

```
// 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);

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

Somebody should get fired

C has a second, different use for the word "static"

to declare the extent of a local variable

if you declare a static local variable, then:

the storage for that variable is allocated when the program loads, in either the program's .data or .bss segment

the variable retains its value across multiple function invocations

(see static_extent.c for an example)

Additional C topics

Teach yourself

bit-level manipulation in C (cf CSE 351): **~ | &   <<   >>**

string library functions provided by the C standard library

#include <string.h>

strlen( ), strcp( ), strdup( ), strcat( ), strcmp( ), strchr( ), strstr( ), ...

learn why **strncat** is safer (in the security sense) than **strcat**, etc.

#include <stdlib.h> or #include <stdio.h>

atoi( ), atof( ), sprintf( ), sscanf( )

man pages are your friend!

Additional C topics

Teach yourself

the syntax for function pointers, including passing as args

how to declare, define, and use a function that accepts a variable-lengthed number of arguments (varargs)

unions and what they are good for

what argc and argv are for in main

```c
#include <stdio.h>

int main(int argc, char **argv) {
  int i;

  for (i = 0; i < argc, i++) {
    printf("%d: %s\n", i, argv[i]);
  }
  return 0;
}
```
argv.c

```
bash$ gcc -o argv argv.c
bash$ ./argv
0: ./argv
bash$ ./argv foo bar
0: ./argv
1: foo
2: bar
bash$
```

Additional C topics

Teach yourself:

the difference between pre-increment (++v) and post-increment (v++)

the meaning of the "register" storage class

Might see it in code, but compilers often ignore it these days since they can often do a better job without it

harder: the meaning of the "volatile" storage class

pages 91, 92 of CARM, much more precise in C11

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

Exercise 2

Modify the linked list code from last lecture / exercise 1

add static declarations to any internal functions you impemented 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

See you on Wednesday!

(But wait!!! There's more....)