Linking, File I/O

CSE 333 Winter 2020

Instructor: Justin Hsia

Teaching Assistants:

Andrew Hu, Austin Chan, Brennan Stein, Cheng Ni, Cosmo Wang, Diya Joy, Guramrit Singh, Mengqi Chen, Pat Kosakanchit, Rehaan Bhimar, Renshu Gu, Travis McGaha, Zachary Keyes

Administrivia

- Exercise 5 posted yesterday, due Wednesday
- Exercise 6 posted today, also due Wednesday

- Homework 1 due next Thursday (1/23)
  - 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!
  - 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

- **Visibility of Symbols**
  - `extern`, `static`
- File I/O with the C standard library
- C Stream Buffering

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
  - Works slightly differently for variables and functions…

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

bar.c

Internal Linkage

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

foo.c

```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;
}
```

bar.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);
}
```

Function Visibility

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 "extern" by default.  Thus, other .c files
// could declare our bar() and invoke it.
int bar(int x) {
  return 2*foo(x);
}
```

main.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;
}
```

Linkage Issues

STYLE TIP

- 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

```
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

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

Lecture Outline

- Visibility of Symbols
  - `extern`, `static`
- **File I/O with the C standard library**
- C Stream Buffering

This is essential material for the next part of the project (hw2)!

File I/O

- We'll start by using C's standard library
  - These functions are part of `glibc` on Linux
  - They are implemented using Linux system calls (POSIX)

- C's `stdio` defines the notion of a stream
  - A sequence of characters that flows **to** and **from** a device
    - Can be either *text* or *binary*; Linux does not distinguish
  - Is *buffered* by default; `libc` reads ahead of your program
  - Three streams provided by default: `stdin`, `stdout`, `stderr`
    - You can open additional streams to read and write to files
  - C streams are manipulated with a `FILE*` pointer, which is defined in `stdio.h`

C Stream Functions (1 of 2)

- Some stream functions (complete list in `stdio.h`):
  - `FILE* fopen(filename, mode);`
    - Opens a stream to the specified file in specified file access mode
  - `int fclose(stream);`
    - Closes the specified stream (and file)
  - `int fprintf(stream, format, ...);`
    - Writes a formatted C string
      - `printf(...);` is equivalent to `fprintf(stdout, ...);`
  - `int fscanf(stream, format, ...);`
    - Reads data and stores data matching the format string

C Stream Functions (2 of 2)

- Some stream functions (complete list in `stdio.h`):
  - `FILE* fopen(filename, mode);`
    - Opens a stream to the specified file in specified file access mode
  - `int fclose(stream);`
    - Closes the specified stream (and file)
  - `size_t fwrite(ptr, size, count, stream);`
    - Writes an array of *count* elements of *size* bytes from *ptr* to *stream*
  - `size_t fread(ptr, size, count, stream);`
    - Reads an array of *count* elements of *size* bytes from *stream* to *ptr*

C Stream Error Checking/Handling

- Some error functions (complete list in `stdio.h`):
  - `void perror(message);`
    - Prints `message` followed by an error message related to `errno` to `stderr`
  - `int ferror(stream);`
    - Checks if the error indicator associated with the specified stream is set
  - `int clearerr(stream);`
    - Resets error and EOF indicators for the specified stream

C Streams Example

cp_example.c

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#define READBUFSIZE 128

int main(int argc, char** argv) {
  FILE *fin, *fout;
  char readbuf[READBUFSIZE];
  size_t readlen;

  if (argc != 3) {
    fprintf(stderr, "usage: ./cp_example infile outfile\n");
    return EXIT_FAILURE;        // defined in stdlib.h
  }

  // Open the input file
  fin = fopen(argv[1], "rb");  // "rb" -> read, binary mode
  if (fin == NULL) {
    perror("fopen for read failed");
    return EXIT_FAILURE;
  }

  ...
```

C Streams Example

cp_example.c

```
int main(int argc, char** argv) {

  ...   // previous slide's code

  // Open the output file
  fout = fopen(argv[2], "wb");  // "wb" -> write, binary mode
  if (fout == NULL) {
    perror("fopen for write failed");
    fclose(fin);
    return EXIT_FAILURE;
  }

  // Read from the file, write to fout
  while ((readlen = fread(readbuf, 1, READBUFSIZE, fin)) > 0) {
    if (fwrite(readbuf, 1, readlen, fout) < readlen) {
      perror("fwrite failed");
      fclose(fin);
      fclose(fout);
      return EXIT_FAILURE;
    }
  }
  ...   // next slide's code
}
```

C Streams Example

cp_example.c

```
int main(int argc, char** argv) {

  ...   // two slides ago's code

  ...   // previous slide's code

  // Test to see if we encountered an error while reading
  if (ferror(fin)) {
    perror("fread failed");
    fclose(fin);
    fclose(fout);
    return EXIT_FAILURE;
  }

  fclose(fin);
  fclose(fout);

  return EXIT_SUCCESS;
}
```

Lecture Outline

- Visibility of Symbols
  - `extern`, `static`
- File I/O with the C standard library
- **C Stream Buffering**

Buffering

- By default, `stdio` uses buffering for streams:
  - Data written by `fwrite()` is copied into a buffer allocated by `stdio` inside your process' address space
  - As some point, the buffer will be "drained" into the destination:
    - When you explicitly call `fflush()` on the stream
    - When the buffer size is exceeded (often 1024 or 4096 bytes)
    - For `stdout` to console, when a newline is written (*"line buffered"*) or when some other function tries to read from the console
    - When you call `fclose()` on the stream
    - When your process exits gracefully (`exit()` or `return` from `main()`)

Why Buffer?

- Performance – avoid disk accesses
  - Group many small writes into a single larger write

  - Disk Latency = 😱😱😱 (Jeff Dean from LADIS '09)

Numbers Everyone Should Know

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

- Convenience – nicer API
  - We'll compare C's `fread()` with POSIX's `read()`

Why NOT Buffer?

- Reliability – the buffer needs to be flushed
  - Loss of computer power = loss of data
  - "Completion" of a write (*i.e.* return from `fwrite()`) does not mean the data has actually been written
    - What if you signal another process to read the file you just wrote to?
- Performance – buffering takes time
  - Copying data into the `stdio` buffer consumes CPU cycles and memory bandwidth
  - Can potentially slow down high-performance applications, like a web server or database (*"zero-copy"*)
- When is buffering faster? Slower?

Disabling C's Buffering

- Explicitly turn off with `setbuf(stream, NULL)`
- Use POSIX APIs instead of C's
  - No buffering is done at the user level
  - We'll see these soon
- But… what about the layers below?
  - The OS caches disk reads and writes in the file system *buffer cache*
  - Disk controllers have caches too!

Extra Exercise #1

- Modify the linked list code from Lecture 4 Extra Exercise #3
  - Add static declarations to any internal functions you implemented in `linkedlist.h`
  - Add a header guard to the header file

Extra Exercise #2

- Write a program that:
  - Uses `argc`/`argv` to receive the name of a text file
  - Reads the contents of the file a line at a time
  - Parses each line, converting text into a `uint32_t`
  - Builds an array of the parsed `uint32_t`'s
  - Sorts the array
  - Prints the sorted array to `stdout`

- Hint: use `man` to read about `getline`, `sscanf`, `realloc`, and `qsort`

```
bash$ cat in.txt
1213
3231
000005
52
bash$ ./extra1 in.txt
5
52
1213
3231
bash$
```

Extra Exercise #3

- Write a program that:
  - Loops forever; in each loop:
    - Prompt the user to input a filename
    - Reads a filename from `stdin`
    - Opens and reads the file
    - Prints its contents to `stdout` in the format shown:

```
00000000 50 4b 03 04 14 00 00 00 00 00 9c 45 26 3c f1 d5
00000010 68 95 25 1b 00 00 25 1b 00 00 0d 00 00 00 43 53
00000020 45 6c 6f 67 6f 2d 31 2e 70 6e 67 89 50 4e 47 0d
00000030 0a 1a 0a 00 00 00 0d 49 48 44 52 00 00 00 91 00
00000040 00 00 91 08 06 00 00 00 c3 d8 5a 23 00 00 00 09
00000050 70 48 59 73 00 00 0b 13 00 00 0b 13 01 00 9a 9c
00000060 18 00 00 0a 4f 69 43 43 50 50 68 6f 74 6f 73 68
00000070 6f 70 20 49 43 43 20 70 72 6f 66 69 6c 65 00 00
00000080 78 da 9d 53 67 54 53 e9 16 3d f7 de f4 42 4b 88
00000090 80 94 4b 6f 52 15 08 20 52 42 8b 80 14 91 26 2a
000000a0 21 09 10 4a 88 21 a1 d9 15 51 c1 11 45 45 04 1b
... etc ...
```

- Hints:
  - Use `man` to read about `fgets`
  - Or, if you're more courageous, try `man 3 readline` to learn about `libreadline.a` and Google to learn how to link to it