Final C Details, Intro to File I/O

CSE 333 Autumn 2025

Instructors: Naomi Alterman, Chris Thachuk

Teaching Assistants:

Ann Baturytski	Derek de Leuw	Blake Diaz
Rishabh Jain	Chendur Jel Jayavelu	Lucas Kwan
Irene Xin Jie Lau	Nathan Li	Maya Odenheim
Advay Patil	Selim Saridede	Deeksha Vatwani
Angela Wu	Jiexiao Xu	

Administrivia

- Today: C wrapup, File I/O with C standard library

- New exercise 6 posted today, due Wednesday morning
  - Will use concepts from today's lecture on File I/O

- Graded hw0 out later today, hw2 out soon

- And you should be well along on hw1 by now…
  - Slides with hw1 hints will be sent via Ed this afternoon

Lecture Outline

- **Preprocessor tricks**
- Additional C topics
- File I/O with the C standard library
- C Stream Buffering

Other Preprocessor Tricks

- A way to deal with "magic numbers" (constants)

```
int globalbuffer[1000];

void circalc(float rad,
             float* circumf,
             float* area) {
  *circumf = rad * 2.0 * 3.1415;
  *area = rad * rad * 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 * rad * 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

- Preprocessor tricks
- **Additional C topics**
- File I/O with the C standard library
- C Stream Buffering

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`)
  - `union`s and what they are good for
  - `enum`s and what they are good for
  - Pre- and post-increment/decrement
  - Harder: the meaning of the "`volatile`" storage class

Lecture Outline

- Preprocessor tricks
- Additional C topics
- **File I/O with the C standard library**
- C Stream Buffering

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`
    - keyboard → console
    - console (buffered)
    - console (unbuffered)
    - 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`):

  NULL if error!

  - `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)

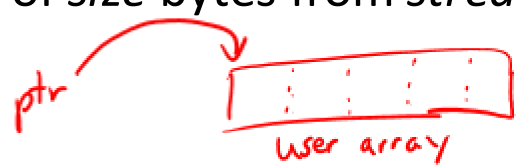
- Some stream functions (complete list in `stdio.h`):
  - `FILE* fopen(filename, mode);`
    - Opens a stream to the specified file in specified file access mode
  - `int fclose(stream);`
    - Closes the specified stream (and file)

tries to move
size * count bytes total

  - `size_t fwrite(ptr, size, count, stream);`

# of elements actually moved (unsigned)

    - 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`):
  - `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
  - `void perror(message);`
    - Prints `message` followed by an error message related to `errno` to `stderr`

global var

error message

① fprintf(stderr, ...); // single, known cause

② perror("my error msg"); // multiple possible causes

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;
  FILE* fout;          ← stream variables
  char readbuf[READBUFSIZE]; ← arbitrarily-sized buffer
  size_t readlen;
                          prints to console, even if you pipe program output
  if (argc != 3) {
    fprintf(stderr, "usage: ./cp_example infile outfile\n");
    return EXIT_FAILURE;       // defined in stdlib.h
  }

  // Open the input file    file must exist when reading
  fin = fopen(argv[1], "rb");  // "rb" -> read, binary mode
  if (fin == NULL) {
    perror("fopen for read failed"); ← prints extra info on source of error
    return EXIT_FAILURE;
  }
  ...   // next slide's code
```

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) {
    // Test to see if we encountered an error while reading
    if (ferror(fin)) {
      perror("fread failed");
      fclose(fin);
      fclose(fout);
      return EXIT_FAILURE;
    }
    ...   // next slide's code
}
```

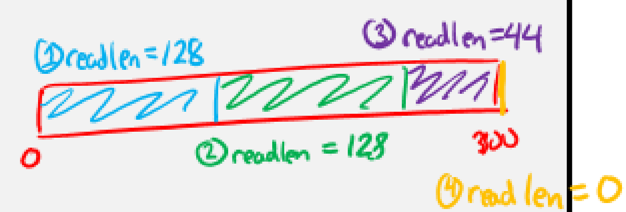
when writing, file created if it doesn't exist

make sure to clean up for every exit path!

# of bytes actually read

(check if error on input stream)

for file of size 300 bytes, fread called 4 times:



C Streams Example

cp_example.c

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

  ...   // two slides ago's code

  ...   // previous slide's code

    if (fwrite(readbuf, 1, readlen, fout) < readlen) {
      perror("fwrite failed");
      fclose(fin);
      fclose(fout);
      return EXIT_FAILURE;
    }
  } // end of while loop

  fclose(fin);
  fclose(fout);

  return EXIT_SUCCESS;
}
```

return value from fread

something wrong if didn't write all requested bytes

close streams when done with them!

Lecture Outline

- Preprocessor tricks
- Additional C topics
- 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
  - At 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`**`()`)

Buffering Example

```
int main(int argc, char** argv) {
  FILE* fout = fopen("test.txt", "wb");


  // write "hi" one char at a time
  if (fwrite("h", sizeof(char), 1, fout) < 1) {
    perror("fwrite failed");
    fclose(fout);
    return EXIT_FAILURE;
  }

  if (fwrite("i", sizeof(char), 1, fout) < 1) {
    perror("fwrite failed");
    fclose(fout);
    return EXIT_FAILURE;
  }

  fclose(fout);
  return EXIT_SUCCESS;
}
```

buffered_hi.c

C stdio buffer

'h'	'i'	...

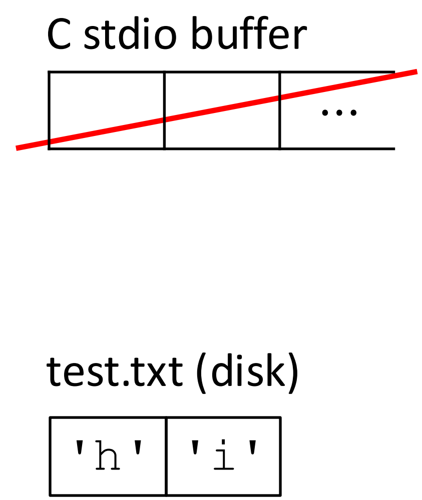
test.txt (disk)

'h'	'i'

No Buffering Example

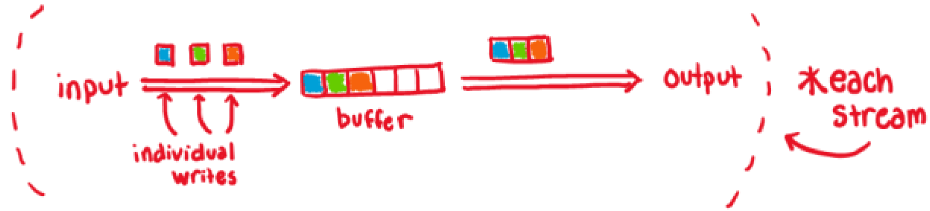
```c
int main(int argc, char** argv) {
  FILE* fout = fopen("test.txt", "wb");
  setbuf(fout, NULL); // turn off buffering

  // write "hi" one char at a time
  if (fwrite("h", sizeof(char), 1, fout) < 1) {
    perror("fwrite failed");
    fclose(fout);
    return EXIT_FAILURE;
  }

  if (fwrite("i", sizeof(char), 1, fout) < 1) {
    perror("fwrite failed");
    fclose(fout);
    return EXIT_FAILURE;
  }

  fclose(fout);
  return EXIT_SUCCESS;
}
```

unbuffered_hi.c



Why Buffer?

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



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

orders of magnitude still accurate!

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"*)

many small writes large writes

- When is buffering faster? Slower?

We Need To Go Deeper…

Preview for next lecture!

- So far we've seen the C standard library to access files
  - Use a provided `FILE*` *stream* abstraction
  - `fopen()`, `fread()`, `fwrite()`, `fclose()`, `fseek()`

- These are convenient and portable
  - They are buffered*
  - They are implemented using lower-level OS calls

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