# CSE 374
# Programming Concepts & Tools

Hal Perkins

Spring 2022

Lecture 9a – C: File I/O

(slides courtesy of CSE 333)

# File I/O in C - a brief introduction

- C's `stdio` library defines the notion of a stream
  - A way of reading or writing a sequence of characters to and from a device
  - Can be either text or binary; Linux does not distinguish
  - Is buffered by default; the library reads ahead of your program, and output is buffered before write to device
  - 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

- Some stream functions defined in `<stdio.h>`
  - See online reference links for details

```
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 memory location *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 Functions

- Formatted I/O stream functions (more in in stdio.h):

```
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

# Error Checking/Handling

- Some error functions (complete list in `stdio.h`):

```
void perror(message);
```

  - Prints message and error message related to errno to stderr

```
int ferror(stream);
```

  - Checks if the error indicator associated with the specified stream is set

```
void clearerr(stream);
```

  - Resets error and eof indicators for the specified stream

# C Streams Example (in file 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];   // space for input data
  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) {
    fprintf(stderr, "%s -- ", argv[1]);
    perror("fopen for read failed");
    return EXIT_FAILURE;
  }
  ...  // next slide's code
```

# C Streams Example (cont.)

```
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) {
    fprintf(stderr, "%s -- ", argv[2]);
    perror("fopen for write failed");
    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");
      return EXIT_FAILURE;
    }
  }
  ...   // next slide's code
}
```

# C Streams Example (concl.)

```
int main(int argc, char** argv) {
  ...   // code from previous 2 slides

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

  fclose(fin);
  fclose(fout);

  return EXIT_SUCCESS;
}
```

# 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 (not needed except for special applications)
  - 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

- Input data from disk files (but not keyboard) is also typically buffered by `stdio`
  - When a file is opened or first read, usually read a disk block (often 1024 or 4096 bytes) into `stdio` memory buffer
  - As program reads data, `stdio` copies data from its buffer to user program as requested
  - When no more data available in memory to satisfy the next user input request, read next block from disk and continue

# Why Buffer?

- Performance – avoid disk accesses
  - Group many small reads or writes into a single larger I/O operation
- Disk Latency = 😱😱😱 (Jeff Dean from LADIS '09)
  - Actual numbers change over time, but relative magnitudes stay similar
- Convenience – nicer API than the native "read a block at a time" API provided by Linux

### 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 |

# Controlling Buffering

- Why not buffer?
  - Reliability – ensure data is written to device now, not later (in case power fails, etc.)
  - Performance – avoid extra data copying for high-volume / high-performance jobs
    - (No, our CSE 374 programs don't really qualify)

- Controlling C's buffering
  - Explicitly disable with `setbuf(stream, NULL)`
    - But potential performance problems if lots of small I/O operations require actual disk accesses each time
  - Use direct Linux read/write system calls
    - More complex, harder to use, but avoids user-level buffering
    - (but Linux also buffers its own disk accesses for performance, which can also be controlled if needed)