

# CS 241: Systems Programming

## Lecture 20. File I/O in C

Spring 2020

Prof. Stephen Checkoway

# Streams

C's view of Input/Output

Sequence of bytes

Physical I/O characteristics are concealed (it's an **abstraction**)

- Files
- Terminal
- Network
- Devices

# Unix I/O

Unix treats all I/O as reading or writing a file

- mice
- printer
- keyboard
- networking
- screen
- disk files

Lower level I/O will be covered later (file descriptors)

# File pointers

C standard library uses **file pointers** to associate a file with a **stream**

```
FILE *stdin;
```

Treat as opaque

- You can't manipulate the **FILE** structure's members directly, must use functions

# Buffering

Output data is stored in a buffer (an array) when writing until there is "enough" data to write to the device

## Buffering types

- ▶ Unbuffered: data is written to device immediately
- ▶ Line buffered: data is written after each newline
- ▶ Fully (or block) buffered: data is written in blocks once the block is full

```
int fflush(FILE *file);
```

# Standard file pointers in Unix

`stdin` — Line buffered if connected to a terminal; otherwise fully buffered

`stdout` — Line buffered if connected to a terminal; otherwise fully buffered

`stderr` — Unbuffered

Recall redirection and pipelines

- `./a.out < input.txt > output.txt`
- `./a.out | filter1 | filter2 > output.txt`

# Opening files as streams

```
FILE *fopen(char const *filename, char const *mode);
```

- ▶ `NULL` on error, `errno` set to indicate error

Mode:

- ▶ "`r`" reading, at beginning
- ▶ "`r+`" read/write, at beginning
- ▶ "`w`" write, create/truncate
- ▶ "`w+`" read/write, create/truncate
- ▶ "`a`" write, create, always at end
- ▶ "`a+`" read/write, create, always at end
- ▶ In addition to `+`, there are also modifiers `b` for binary streams and `x` for eXclusive (`fopen(path, "wx")` fails if path already exists)

If we want to read the contents of a text file into memory, modify it, and then write it back to the same file, which call to `fopen()` should we use?

A. `FILE *fp = fopen(path, "r+");`

B. `FILE *fp = fopen(path, "w+");`

C. `FILE *fp = fopen(path, "a+");`

D. `FILE *fp = fopen(path, "rb");`

E. `FILE *fp = fopen(path, "wx");`



# Stream I/O single char

```
int getchar();    // gets a char from stdin
```

```
int getc(FILE *stream); // macro
```

```
int fgetc(FILE *stream); // actual function
```

```
int putchar(int c); // writes a char to stdin
```

```
int putc(int c, FILE *stream); // macro
```

```
int fputc(int c, FILE *stream); // function
```

# Stream I/O multiple chars

// Reads a line (up to a maximum size)

```
char *fgets(char *str, size_t size, FILE *stream);
```

// Writes str to stdout and appends a newline

```
int puts(char const *str);
```

// Writes str to file but does not append a newline

```
int fputs(char const *str, FILE *stream);
```

Analogous to `puts ( )` vs. `fputs ( )`, there's a function

```
char *gets(char *str);
```

that reads a line from `stdin` and stores it in `str`.

**This function should never be used under any circumstance!**

Why not?

- A. Including the function was a mistake by the C designers
- B. There's no bounds checking on the input
- C. A too-long line may crash the program
- D. A too-long line may let an attacker take control of the program
- E. All of the above

# Checking for EOF/error

`int feof(FILE *stream);` // returns nonzero if stream is at the end

`int ferror(FILE *stream);` // returns nonzero if stream had an error

```
#include <stdio.h>
```

```
int main(int argc, char *argv[argc]) {  
    FILE *input = fopen(argv[1], "r");  
    FILE *output = fopen(argv[2], "w");  
    char str[1024];  
  
    while (fgets(str, sizeof str, input) != 0) {  
        if (fputs(str, output) == EOF)  
            break;  
    }  
    if (ferror(input) || ferror(output))  
        return 1;  
    return 0;  
}
```

# Error information

```
#include <stdio.h>
```

```
#include <errno.h>
```

```
extern int errno; // libc funcs set this on failure
```

```
char *strerror(int errnum); // human-readable error string
```

```
void perror(char const *str); // prints error on stderr
```

perror(str) is (essentially)

```
if (str != 0 && str[0] != 0)
```

```
    fprintf(stderr, "%s: %s\n", str, strerror(errno));
```

```
else
```

```
    fprintf(stderr, "%s\n", strerror(errno));
```

# Exit values

When errors occur, may want to terminate program

```
void exit(int status);
```

`EXIT_SUCCESS` — value 0, c99 standard

`EXIT_FAILURE` — some value other than 0, (usually 1) c99 standard

BSD has tried to standardize other values

- `/usr/include/syssexits.h`

# Closing a stream

```
int fclose(FILE *stream);
```

- Returns 0 if successful
- **EOF** on error (see `errno`)

Can close **stdin**, **stdout**, **stderr** if unneeded

- There is a limit to the number of files allowed to be open at once



```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[argc]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s FILE\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    FILE *fp = fopen(argv[1], "w");
    if (!fp) {
        perror(argv[1]);
        exit(EXIT_FAILURE);
    }
    fputs("Created for CS 241\n", fp);
    fclose(fp);
    return EXIT_SUCCESS;
}
```

# In-class exercise

<https://checkoway.net/teaching/cs241/2020-spring/exercises/Lecture-20.html>

Grab a laptop and a partner and try to get as much of that done as you can!