# CS 241: Systems Programming Lecture 21. Binary and Formatted I/O

Spring 2020
Prof. Stephen Checkoway

# Review from last time

```c
// Open and close a file.
FILE *fopen(char const *path, char const *mode);
int fclose(FILE *stream);

// Read or write a character.
int fgetc(FILE *stream);
int fputc(int ch, FILE *stream);

// Read or write a string.
char *fgets(char *str, int size, FILE *stream);
int fputs(char const *str, FILE *stream);

// Formatted output.
int fprintf(FILE *stream, char const *format, ...);
```

# re-reading a file

```
void rewind(FILE *stream);
```

- resets the file back to the start of the stream
- NOTE: no return value
    - zero out and check errno for problems

Actually is an alias for…

# Changing location in a file

```
int fseek(FILE *stream, long offset, int whence);
int fseeko(FILE *stream, off_t offset, int whence);
```

- Return 0 on success, -1 on failure (`errno`)

Reposition location in stream

- `offset` is number of bytes added to position specified by `whence`
  - `SEEK_SET` - start of the file
  - `SEEK_CUR` - current position
  - `SEEK_END` - end of the file

# Getting location in file

```
long ftell(FILE *stream);
off_t ftello(FILE *stream);
```

- returns current offset on success
- returns -1 and sets errno on failure

`fseeko` and `ftello` are specified by POSIX but not C

`off_t` is an integral type representing file sizes (often 8 bytes)

How can we get the size of a file to which we have an open `FILE *stream`?

A.
```
off_t size = ftello(stream);
```

B.
```
fseeko(stream, 0, SEEK_SET);
off_t size = ftello(stream);
```

C.
```
fseeko(stream, 0, SEEK_CUR);
off_t size = ftello(stream);
```

D.
```
fseeko(stream, 0, SEEK_END);
off_t size = ftello(stream);
```

E.
```
off_t pos = ftello(stream);
fseeko(stream, 0, SEEK_END);
off_t size = ftello(stream);
fseeko(stream, pos,
       SEEK_SET);
```

```c
int get_file_size(char const *path, off_t *size) {
  FILE *fp = fopen(path, "rb");
  if (!fp)
    return -1;

  int ret = -1;
  if (fseeko(fp, 0, SEEK_END) == 0) {
    if ((*size = ftello(fp)) != -1)
      ret = 0;
  }
  int err = errno;
  fclose(fp);
  errno = err;
  return ret;
}
```

```c
int get_file_size(char const *path, off_t *size) {
  FILE *fp = fopen(path, "rb");
  if (!fp)
    return -1;

  int ret = -1;
  if (fseeko(fp, 0, SEEK_END) == 0) {
    if ((*size = ftello(fp)) != -1)
      ret = 0;
  }
  int err = errno;
  fclose(fp);
  errno = err;
  return ret;
}
```

Returning the size via a pointer parameters

```c
int get_file_size(char const *path, off_t *size) {
  FILE *fp = fopen(path, "rb");
  if (!fp)
    return -1;

  int ret = -1;
  if (fseeko(fp, 0, SEEK_END) == 0) {
    if ((*size = ftello(fp)) != -1)
      ret = 0;
  }
  int err = errno;
  fclose(fp);
  errno = err;
  return ret;
}
```

Returning the size via a pointer parameters

fclose() might change errno

```c
int main(int argc, char *argv[argc]) {
  for (int i = 1; i < argc; ++i) {
    off_t size;
    if (get_file_size(argv[i], &size) == -1) {
      perror(argv[i]);
    } else {
      intmax_t s = size; // Can't print off_t directly.
      printf("%s: %jd\n", argv[i], s);
    }
  }
  return 0;
}
```

# Formatted input

```
int scanf(const char *format, ...);
```

- input analog to `printf()`
- reads input from stdin
- uses format string to determine types
- arguments must be **pointers**
  - common error
- Stops when
  - format string is done
  - input mismatch
- returns # of successfully matched items
- returns EOF on EOF (not 0)

# Example

```c
#include <stdio.h>
int main(void) {
  int pairs = 0;
  int x, y;
  while (scanf(" (%d ,%d )", &x, &y) == 2)
    ++pairs;
  printf("Read in %d valid pairs.\n", pairs);
  return 0;
}
```

Spaces in the format match white space characters, the %d skips white space so `(1,2)` are `(  3  ,   4      )` both valid, but `(0,1(2,3(4,5` gives 3 valid pairs!

# scanf format string interpretation

White space matches 0 or more white space characters in the input

Ordinary characters are matched against non-whitespace

Conversion specifications: e.g., `%8lx`

- `%` to indicate start (like printf)
- * indicates not to store the value
- number for field width
- hh, h, l: size of storage character
- conversion character (see printf)
- Most conversion specifiers skip white space (all but `%[…]`)

Assume we have an integer variable x, how do I read in a decimal value from `stdin` and assign it to x?

```
int x;
```

```
A. scanf("&x");

B. scanf("%d", x);

C. scanf("%d", &x);

D. scanf("%d", *x);

E. scanf("%x", &x);
```

## scanf family

```c
int fscanf(FILE *stream, const char *format, ...);

int sscanf(const char *str, const char *format, ...);
```

# Character ranges

`%s` matches a sequence of non-whitespace characters

`%[chars]` matches a range of characters, which can include whitespace

```
char html_tag[32];
sscanf(line, " <%31[^>]>", html_tag);
```

Assume we have a char array `word`, how do I read in a word of text from `stdin`?

```
char word[16];
```

```
A. scanf("&word");

B. scanf("%15s", word);

C. scanf("%16s", word);

D. scanf("%s", &word);

E. scanf("%s", *word);
```

## Useful input technique

fgets() / sscanf() pairing

- Read a line using fgets()
- Parses data using sscanf() from line

Always does bounds checking

# Binary data

```
size_t fread(void *ptr, size_t size, size_t nitems,
             FILE *stream);
size_t fwrite(void const *ptr, size_t size,
              size_t nitems, FILE *stream);
```

- Read/write `nitems` number of `size` sized objects
- Returns the number of objects read/written which will be less than `nitems` for EOF or an error
- Must use `feof()` or `ferror()` to determine which occurred

```
int x = 42;
float y[8];
size_t num = fread(y, sizeof(float), 8, stream);
num = fwrite(&x, sizeof(int), 1, stream);
```

## Aside: Printing to a string

```
int snprintf(char *str, size_t size,
             char const *format, ...);
```

- Writes at most size-1 bytes into str and null terminates
- Returns number of bytes that are printed (or would be printed if the string were large enough), negative on error

```
char message[100];
snprintf(message, sizeof message, "%s %d",
         some_string, some_int);
/* or */
size_t size = snprintf(0, 0, "%s %d", str, x);
char *message = malloc(size + 1);
snprintf(message, size+1, "%s %d", str, x);
```

# DANGER: Format String Attacks

Don't just print arbitrary users' strings

- `printf(str);`

If the attacker sets the value of line they can

- Cause it to reveal other program data by printing it from the program stack (e.g., `"%x%x%x"`)
- Can cause it to change program data by using "%n" which stores # of chars printed so far

Use one of these instead

- `printf("%s", str);`
- `fputs(str, stdout);`

# In-class exercise

https://checkoway.net/teaching/cs241/2020-spring/exercises/Lecture-21.html

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