Lecture 07
Standard I/O

CS211 – Fundamentals of Computer Programming II

Branden Ghena – Winter 2022

Slides adapted from:
Jesse Tov

Northwestern

Administrivia

- Quiz today!
  - Message your friends who didn’t show up to class

  - It’ll be 15 minutes long
  - I’ll stop lecture around 3:00 and hand out quizzes
    - Somebody let me know if I forget

Administrivia

- Some office hours are now in-person
  - All are held in Tech EG20 (Wing E, Ground floor, Room 20)

New: Dark purple office hours are in-person. Light teal office hours remain online.

Administrivia

- Homeworks are getting harder! Remember to start early
  - Homework 4 (next week) will on top of Homework 3 (this week)

- Remember that there will be a break after homework 4
  - Homework 4 due February 03
  - **Nothing due February 10**
  - Homework 5 due February 17

  - There is a break in sight!

Today's Goals

- Explore input and output to files
  - What C library functions allow interacting with files?
  - How do stdin, stdout, and stderr work?
- If we have time:
- Practice dynamic memory allocation with arrays
  - How do we make a dynamically-sized array?

Getting the code for today

```
cd ~/cs211/lec/
```
(or wherever you put stuff)

```
tar -xkvf ~cs211/lec/07_stdio.tgz
cd 07_stdio/
```

Outline

- **File Input and Output in C**
- Standard Input and Output
- Dynamic Arrays

Files

- Collections of data
  - Usually in permanent storage on your computer

- Types of files
  - Regular files
    - Arbitrary data
    - Think of as a big array of bytes (just like memory)

  - Directories
    - Collections of regular files

  - Special files
    - Links, pipes, devices (see CS343)

How do we interact with files?

- Analogy: think of a file as a book
  - Big array of characters (bytes)

1. Open the book, starting at the first page
2. Read from the book
3. Write to the book
4. Change pages (without reading everything in between)
5. Close the book when finished

System calls for interacting with files

1. Open the book, starting at the first page
    - fopen()
2. Read from the book
    - fread()
3. Write to the book
    - fwrite()
4. Change pages (without reading everything in between)
    - fseek()
5. Close the book when finished
    - fclose()

Opening files

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

- `filename` is the string path for the file
  - “/home/brghena/class/cs211/w22/hw/hw01/src/circle.c”
  - “./arguments.c”
  - “arguments.c”

- `mode` specifies what you intend to do with the file
  - “r” - read only (must exist)
  - “w” - write (overwrites if exists)
  - “a” - append (starts writing at end of file if exists)

Open returns a FILE object

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

- Pointer type for an object used to interact with the file
  - A “handle” to the file
- Other file interaction functions will take in a `FILE*` as an argument
  - Don’t need to remember the file path and look it up every time
- `NULL` instead specifies an error attempting to open the file

Reading files

```
size_t fread(void* ptr, size_t size, size_t count, FILE* stream);
```

- `ptr` is a pointer to an array to read into
  - At least `size * count` bytes in length
- `size` is the number of bytes for each element in the array
- `count` is the number of elements to read
- `stream` is the file pointer returned from a previous call to `fopen()`

- Note: nowhere do we specify where to *start* reading
  - Library keeps track of a file offset with the file
  - Updated on each read
    - First read of 100 bytes starts at zero, next starts 100 bytes in

How do we know when we finished the file?

```
size_t fread(void* ptr, size_t size, size_t count, FILE* stream);
```

- Return from read is the count of elements *actually* read
  - Less than `count` means there was either an error or end-of-file was reached

- `feof()` lets you check if end-of-file was reached
- `ferror()` lets you check for particular errors

Writing files looks a lot like reading

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

- Array to write from, size of elements in the array, number of elements to write, and a file pointer

- Returns number of elements *actually* written

- Write occurs at the current file offset

Moving the file offset

```
int fseek(FILE* stream, long int offset, int origin);
```

- Moves to offset for this file descriptor based on origin:
  - SEEK_SET – set to offset (essentially start of file plus offset)
  - SEEK_CUR – current location plus the offset
  - SEEK_END – end of file plus the offset (which can be negative)

- Returns zero if successful
  - Anything else means an error occurred

- `ftell()` gets the current location in a file
  - So you can seek back there later

Closing a file

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

- Closes the file
- Returns zero on success
- It is an error to keep using the file descriptor after it is closed
  - Just like with dynamic memory management

References

- https://www.cplusplus.com/reference/cstdio/
  - Explanation of and links for everything in <stdio.h>

Buffered I/O

- C standard library buffers your interactions to make them more efficient
  - One big write to a file is MUCH faster than many small writes

- Sometimes you want to write to output *right now*
  - `fflush()` guarantees that the buffer is written *now*
  - Otherwise no write is guaranteed until `fclose()` is called

- Example: `printf()` buffers until a newline is reached
  - So a print right before a fault might not appear unless it includes a '\n' 😱

Example: kitten tool

- Command line tool: `cat` – prints out the contents of files
  - Does so very efficiently
- Our program: `kitten` – prints out the contents of one file
  - No efficiency promises
- Writing kitten only requires file I/O mechanisms we’ve discussed!

Live coding: implement kitten

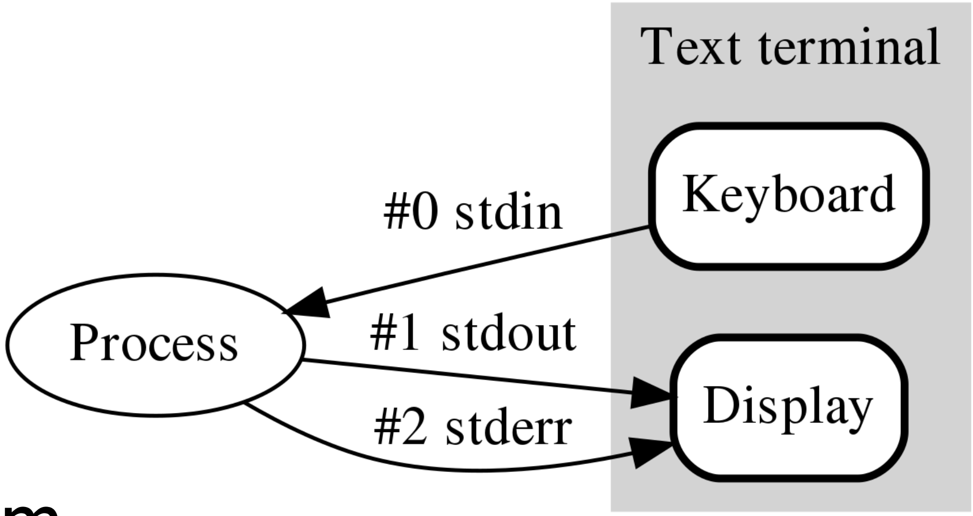
kitten.c

- Requirements
  - Parse argv[] to find file to open
  - Open the file
  - Read in lines from the file repeatedly
    - If end-of-file is reached, break (`feof()`)
    - Print contents of file
  - Handle errors

Outline

- File Input and Output in C
- **Standard Input and Output**
- Dynamic Arrays

How do programs talk to users?



- We glossed over this before
  - `printf()`
  - `scanf()`
- Work through the same file mechanism
  - Three special files created for each program
  - stdin – standard input
  - stdout – standard output
  - stderr – standard error
- `printf()` -> `fprintf(stdout)` -> handle arguments & `fwrite(stdout)`

Standard I/O is a process thing, not a C thing

- You can access them in Python, for instance
  - https://docs.python.org/3/library/sys.html#sys.stdin

`sys.stdin`
`sys.stdout`
`sys.stderr`

File objects used by the interpreter for standard input, output and errors:

- `stdin` is used for all interactive input (including calls to `input()`);
- `stdout` is used for the output of `print()` and expression statements and for the prompts of `input()`;
- The interpreter's own prompts and its error messages go to `stderr`.

These streams are regular text files like those returned by the `open()` function. Their parameters are chosen as follows:

Standard I/O is configured by the shell

- When you run a program in command line, the shell attaches a standard input, standard output, and standard error to it

- Defaults
  - stdin - read from terminal
  - stdout - write to terminal
  - stderr - write to terminal

Live coding: kitten upgrades

kitten.c

- Errors should be written to stderr

- Output can be written to stdout directly using `fwrite()`
  - Instead of using `printf()` in a loop to do it for us

Redirecting standard I/O

- Shells by default setup standard I/O to connect to the keyboard and the screen
  - But any file will also work

- Shell I/O redirection commands
  - COMMAND < filename
    - Connect standard input to filename
  - COMMAND > filename
    - Connect standard output to filename (overwrite)
  - COMMAND >> filename
    - Connect standard output to filename (append)

Piping commands

- A command shell desire is to run multiple commands where the output of the first feeds into the second

- COMMAND1 | COMMAND2
  - Connects stdout of COMMAND1 to stdin of COMMAND2

- Example: print out files and sort by size
  - ls –lah | sort –h

Sidebar: super useful command for testing

- **`tee`** `[`*`OPTION`*`]... [`*`FILE`*`]...`
  - Reads from stdin and write to **both** stdout and file

- Example: prints out a list of files and saves results
  - ls –lah | tee results.txt

- I run this with various programs I'm testing, so I can record the results, but also seem them in real-time.

Example: redirection with kitten

- Standard I/O redirection is handled when the process is created
  - So it does not need to be aware of it at all

- Our kitten tool works with redirection automatically!
  - ./kitten arguments.c > OUTPUT_FILE

Break + Thinking Excercise

- Take a look at the `cat` command to see the other flags it supports

```
-A, --show-all
       equivalent to -vET

-b, --number-nonblank
       number nonempty output lines, overrides -n

-e     equivalent to -vE

-E, --show-ends
       display $ at end of each line

-n, --number
       number all output lines

-s, --squeeze-blank
       suppress repeated empty output lines

-t     equivalent to -vT

-T, --show-tabs
       display TAB characters as ^I

-u     (ignored)

-v, --show-nonprinting
       use ^ and M- notation, except for LFD and TAB
```

How hard would these be to implement in `kitten`?

Outline

- File Input and Output in C
- Standard Input and Output
- **Dynamic Arrays**

Dealing with dynamic input

- What if you want to read in data, but you don't know how much data there might be?

- Arrays in C are a fixed size
- But you can `malloc()` as many times as needed
  - Request some memory
  - Use until you run out
  - Request more memory and copy existing values over

  - `realloc()` makes this simple

Example of dynamic memory: read_line()

```
char* read_line(void)
```

- Reads an entire line at a time from stdin
  - Can’t know in advance how many bytes there will be to read

  - Keeps reading in bytes until ‘\n’ character or end-of-file
  - Needs to request more memory until it holds the entire line

- Note: part of the 211 library, not standard C

Live coding: implement read_line()

readline.c

```
char* read_line(void)
```

- Requirements
  - Read from stdin until '\n' or end-of-file (EOF)
    - Could `fread()` or just use `getchar()`
  - Allocate an array to hold the read characters
    - Make sure to end it with a '\0'
  - Returns
    - NULL pointer if EOF was reached immediately
    - Pointer to string otherwise (not including the newline character)

Realloc versus malloc

- We could just `malloc()` and copy ourselves, what does `realloc()` add?

- `realloc()` can be far more efficient
  - Doesn't have to copy data at all if there is room in the heap to expand

- Also simpler for programmers
  - Can't forget to free the old memory if `realloc()` does it for you

Default string size will change efficiency

- Memory efficiency
  - Pointer returned could have way more memory than characters
  - User might hold on to memory for a while before freeing
  - The less wasted memory, the less memory the program needs

- Runtime speed
  - `malloc()` and `realloc()` are slow
  - The fewer times we call them, the faster the program will run

- Need to pick a sweet spot to balance the two of these
  - Real program: starts at 80 characters, doubles size when reallocating

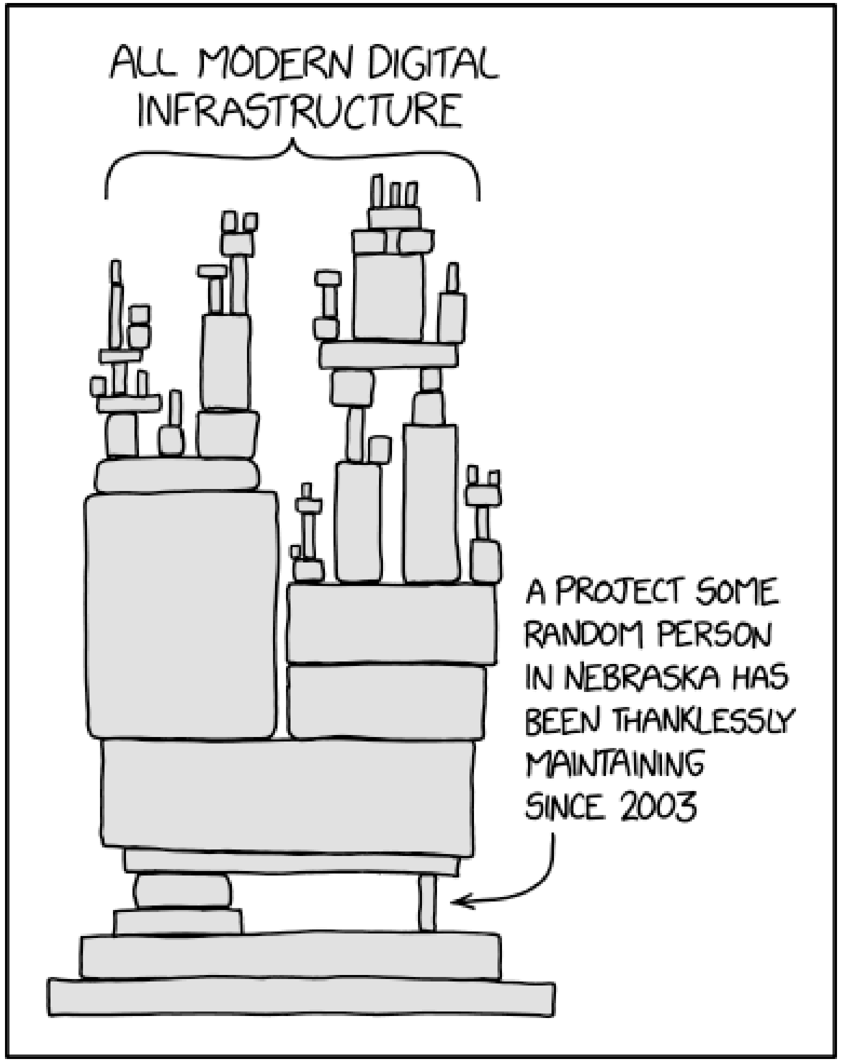
Does efficiency really matter though?

- If you're writing a CS211 homework: **no**

- If you're writing a Javascript interpreter for Firefox,
  - Which has millions of users
  - times hundreds of websites per day for each user
  - times hundreds of lines of code per website
  - and each line of code is read with `read_line()`

  - **YES**

Break + relevant xkcd



https://xkcd.com/2347/

Outline

- File Input and Output in C
- Standard Input and Output
- Dynamic Arrays