Strings in C:

- Arbitrary length
- Special value terminator
- Access past the end
- No array bounds checking
- Why do banks limit you to 12-character passwords?
    - Multi-faceted problem
    - Strings are fixed length
        - `char str[10] = "Hi people";`
            - Allows for 9 characters + null terminator
            - But nothing stops us from accessing past the end
                - `str[10] = '@';`
            - Could be anything past the array
            - If the null terminator is lost, we have no idea how long the string is
                - `str[9] = 'x";`
                - Will keep going until it happens to find a zero byte \0
    - An initializer fills the rest of the array with 0
        - `char stri[8] = "Hello!"`
        - H e l l o ! /0 /0
        - What if we want a longer array?
- You don't know in advance how long an input string will be
    - From the user, a file, network, etc…
- How much space do you need?
    - How do you "allocate" enough space?
- Initializing an array of characters
    - `char mystr[100] = "some string";`
    - Allocates space for 100 characters
    - Fills the first however many with that string
    - Fills the rest with \0s
    - Shorthand for longer initializer
    - Use if you want to manipulate the string
    - `char* mystr = "some string";`
        - Use pointers if you don't care about manipulating the string
            - Static segment is read only
        - Creates a pointer variable
        - Puts "some string" in the static data segment (constants)
        - Makes that pointer point to the static data segment

String Functions:

- `<string.h>` has many functions
    - `strlen()`
        - Has to look through the whole string for \0 every time
        - Avoid putting in a loop
    - `strcmp()`

        - Compares two strings and returns a comparison value
        - < 0 means a comes before b
        - > 0 means a comes after b
        - 0 means a is equal to b
        - Case sensitive
            - Hello and hello will not return equal
        - `int streq(const char* a, const char* b) {`
            - `return strcmp(a,b) == 0;`
        - `}`
            - Const means read-only pointer, I won't try to change the data it points to
        - `if(streq(command, "print")) {`
            - `// print stuff…`
            - `} else…`
- String manipulation functions:
    - `strcpy(a,b)` copies the string from b into the memory of a
    - `strcat(a,b)` copies the string from b into the memory AFTER a
        - Concatenation
    - String manipulation in C is hard and should try to avoid

Files:

- File paradigm
    - A big array of bytes
    - Each byte has a position within the files, which starts from 0
    - But unlike an array, the file has a current position
    - It starts at 0 and every time you read or write something, it moves ahead
- What's in a file?
    - The meaning of the data in the file is up to you
    - A file format defines the structure and meaning of data in the file
        - All files are sequences of 0s and 1s, grouped into bytes
            - Text files are like a big string, they hold text and nothing else
            - Binary files are pretty much anything else
                - You can interact with the information
    - File extensions are not magical
        - Always enable file extension display
        - Extension is just part of the file's name, it lets you know what type of file it is
        - Has no effect on its contents
            - Can't just rename file to change file type
- Opening and closing files in C
    - You can open files like this (make sure to include stdio.h)
        - `FILE f* = fopen(name, mode);`
        - Name: name of the file

    - Mode: one of the following
        - r, w: for reading or writing text files
        - “rb”, “wb”: for reading or writing binary files
        - “r+”, “rb”: for reading AND writing files at the same time
    - Fopen may return null
- Close files like this:
    - `fclose(f);`
    - DON’T FORGET TO CLOSE THEM
    - If you don’t close them after changing them, your changes may or may not actually end up in the file
- What is `FILE*` exactly?
    - Pointer to data in the file
        - Pointers are also used to point to objects
        - You can’t do anything with a file pointer except pass it to functions which expect them as arguments
        - Treat as a black box
- Reading and writing text files
    - `fgets(“buffer, sizeof(buffer), f);`
    - `fprintf(f, “hello!\n”);`
    - Because stdin and stdout and stderr are `FILE”`s too
        - `print(“hi”)` is short for `fprintf(stoud,”hi”);`
    - How is the console a file?
        - Not stored in file system
        - Created in real time as user types
        - Everything’s a file
- `ftell(f)`
    - Gives you the current file position (distance from the beginning)
    - Measured in bytes
- `feof(f)`
    - Tells you if you are at the end of file (EOF)
    - Commonly used with text files
        - Reading the lines from a file:
            - `while(!eof(f)) read_a_line(f);`
- Moving around
    - Move around the file with `fseek(f, offset, how)`
        - Offset depends on what you pass to “how”
            - `seek_set` sets the position
                - Offset 4, go to 4
            - `seek_cur` relative movement from current position
                - Offset of 4, add 4 to current position
            - `seek_end` relative movement from end position
                - Works like a negative offset
            - Can't go off ends of file

- Can combine `fseek` and `ftell` to figure out how long a file is
- `fseek(f, 0, SEEK_END);`
- `len = ftell(f);`
- `fseek(f, 0, SEEK_SET));`