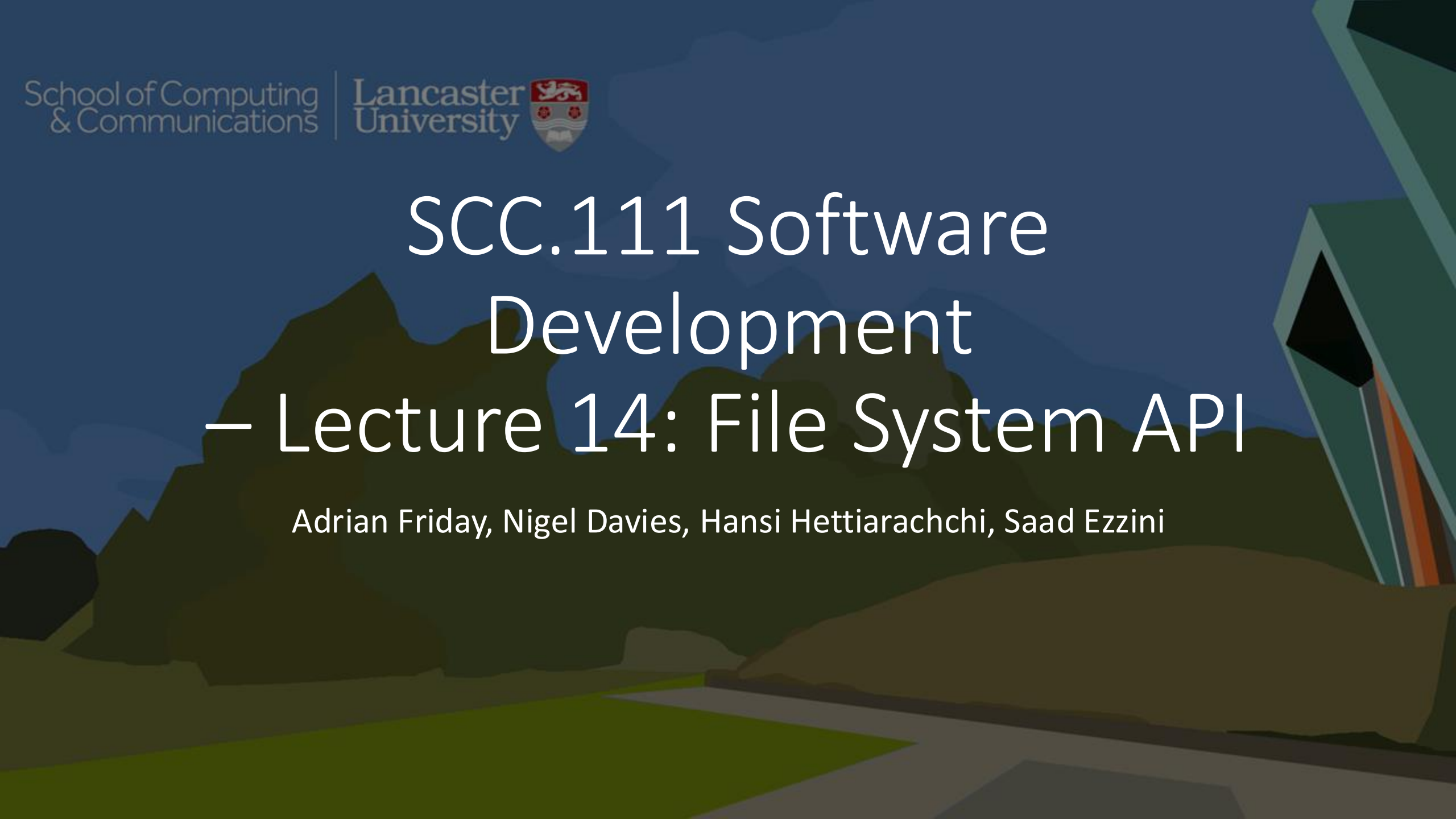
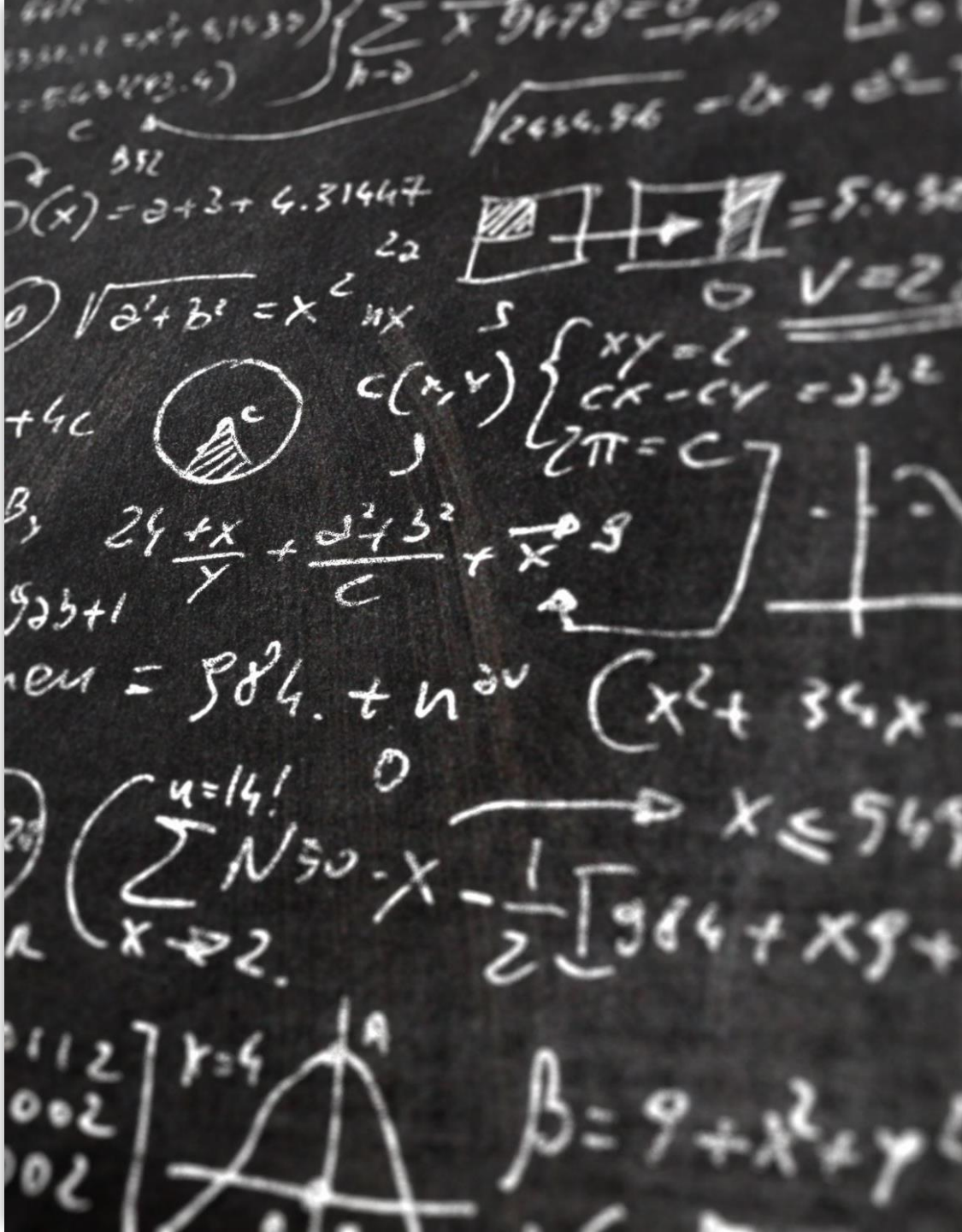


SCC.111 Software Development – Lecture 14: File System API

Adrian Friday, Nigel Davies, Hansi Hettiarachchi, Saad Ezzini

This lecture

- How C interfaces with the operating system
- What is an API
- How to call one
- File system I/O as a case study



So far…

- We've only dealt with variables and state that we create at runtime
- We've relied on the user of our program to input any data
- What if we want our programs to have lasting effects? (e.g. create files to store things persistently, get input from files)

We'd need

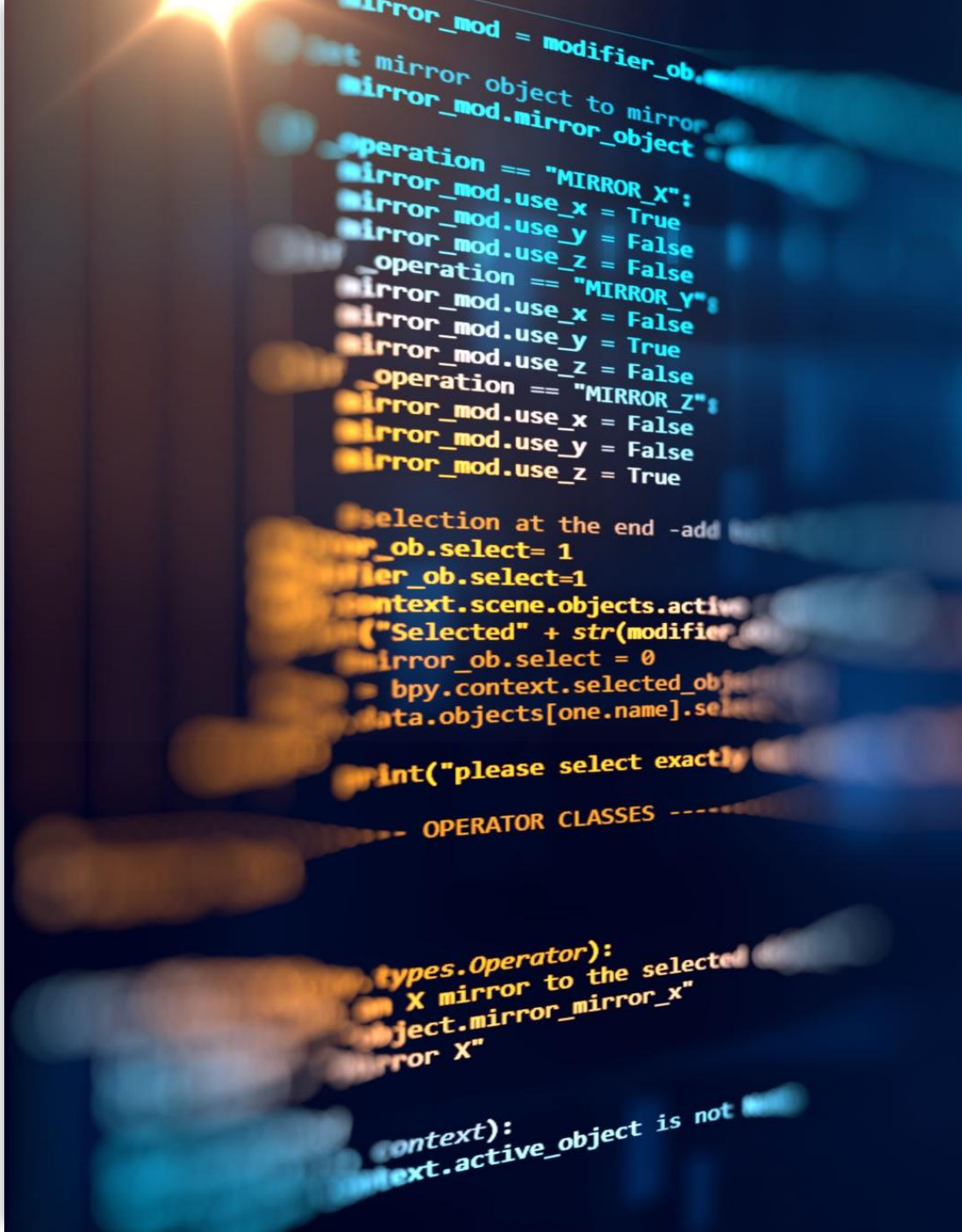
- Some way for our running program to interact with its environment (the operating system or **OS**)
- The ability to get data from long term persistent storage
- To handle any errors (missing files, permissions, read/write data)

We need to use an API

- An **application programming interface (API)** is just a software interface
- In this case, a set of operations defined for interacting with the system in a controlled way
- There are many APIs to access different system features (storage, inter-process or network communication, starting/stopping other processes, …)

What does an API look like?

- Unsurprisingly, an API we access in C is just
  - One or more **functions** we can call
  - Declared in a **header** (.h) file that we need to include to be able to use them
  - APIs that are included as part of ANSI C can be assumed to be available on any platform
  - *Though in practice owe a lot to the design and APIs of the UNIX operating system*



Recap: Declaring vs. calling functions

```
int main()
{
  int userInput;

  scanf("%d", &userInput);

  print_user_input(userInput);
}
```

The function declaration specifies its ‘interface’

```
void print_user_input(int dataInHere)
{
  // dataInHere is a parameter
  // it has 'local scope' to this function
  printf("User input was %d\n",
        dataInHere);
}
```

The function call, fills in the ‘actual’ values,
But the number and type of parameters must match the function declaration…

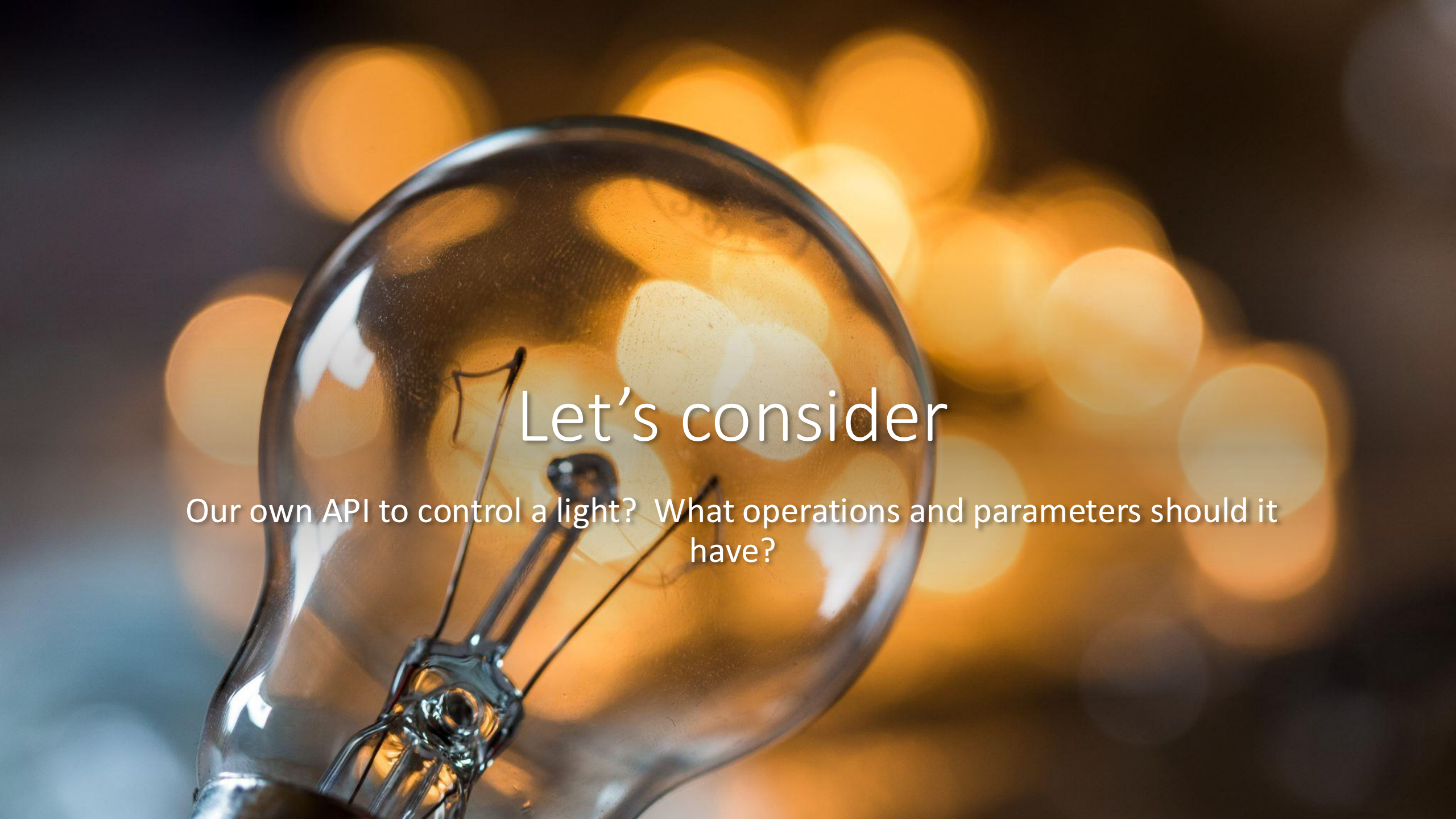
You can think of this as an API

```
void print_user_input(int dataInHere)
```

The name of the function to call (case must match)

The type and order of any parameters

Anything returned
(*here nothing*)

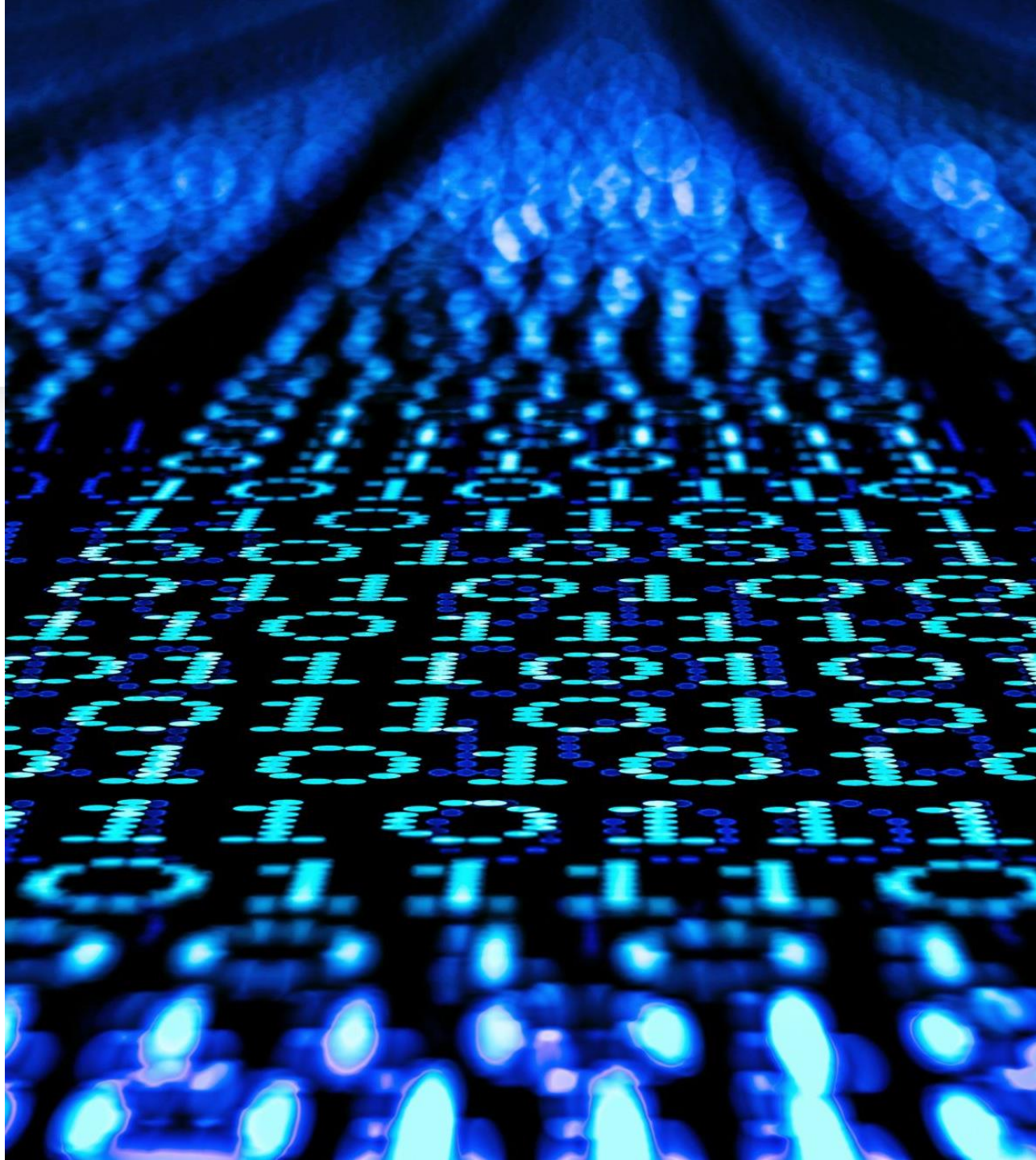


Let’s consider

Our own API to control a light? What operations and parameters should it have?

What is a file?

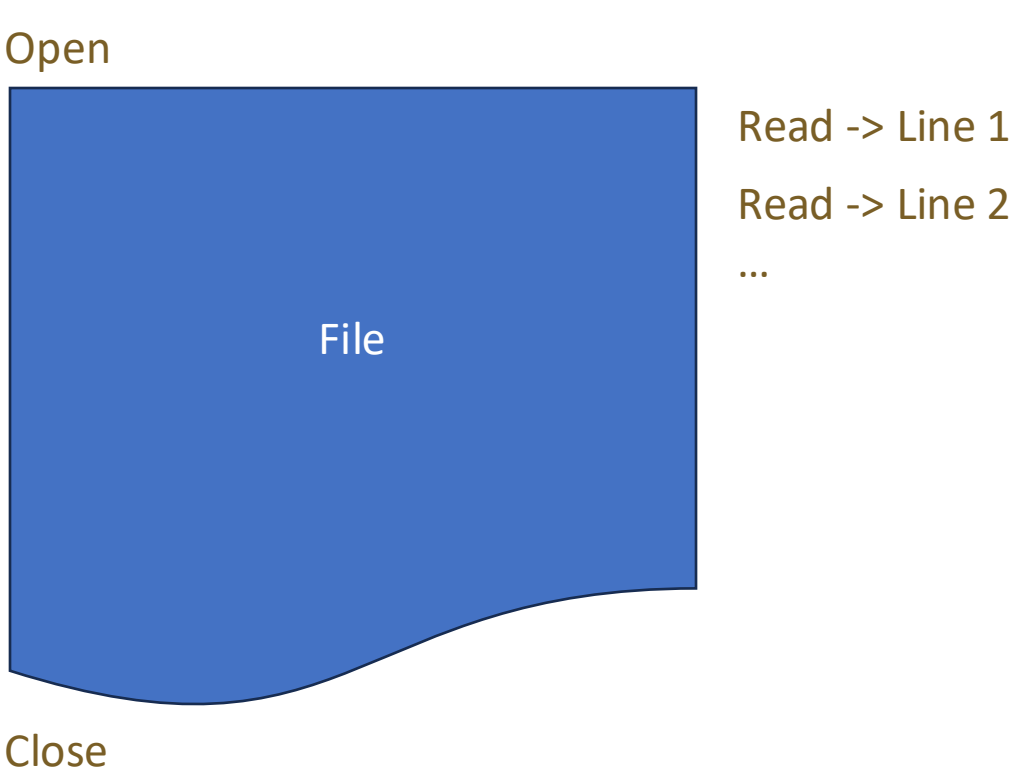
- A **file** can be informally defined as a collection of (typically related) data, which can be logically viewed as a stream of bytes (i.e. characters). A file is the smallest unit of storage in the Unix file system.

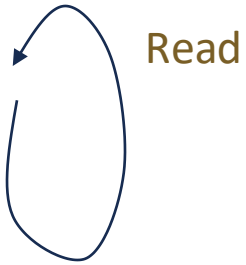


A file

- Provides *ideally* persistent storage of data
- Either text or binary format (text files contain lines terminated with an end of line marker, e.g. ‘\n’ – *some platform differences here!*)
- Can access serially (lines of data read in turn until the end of the file is reached) or random access (jumping or ‘seeking’ to the data we want)
- We need to open a file to access it, and close it afterwards

Serial access





Conceptually, there's a 'file pointer' that moves from the start to the end of the file as we read data from it

File API case study

1. Character streams…

- Processes input and output at a character stream level…
- int getchar(void)
  - Returns the next input character each time it is called, or **EOF** when it encounters end of file. The symbolic constant EOF is defined in <stdio.h>.
  - We would likely need to redirect input to our program to make this refer to f a file, e.g. `prog < infile`
  - Or, `otherprog | prog`
- int putchar(int)
  - is used for output: putchar(c) puts the character c on the standard output, which is by default the screen. putchar returns the character written, or EOF if an error occurs. Also, `prog > outfile`

Quick example:

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

// lower: convert input to lower case

int main()
{
  int c;

  while ((c = getchar()) != EOF)
    putchar(tolower(c));

  return 0;
}
```

More generally, what other operations make sense for files?

File system rules

- Before it can be read or written, a file has to **be opened** by the library function fopen
- **fopen** takes a file name, does some housekeeping with the operating system, and returns a *pointer (FILE *)* to be used in subsequent reads or writes
- This pointer, the *file pointer*, points to a structure that contains information about the file (e.g. where to read from)
- Users *don't need to know the details*, because the definitions obtained from <stdio.h> include a structure declaration called FILE

2. Opening a file

```
// Declare a pointer to some FILE structure
FILE *fp;

// The API call to open a file
FILE *fopen(char *name, char *mode);
```

All being well, the file pointer is ‘valid’. But it could be ‘NULL’, why?

… and closing it again…

```
// Reading and writing characters
int getc(FILE *fp)
int putc(int c, FILE *fp)

// And, eventually...
fclose(FILE *fp);
```

*Note: these are the declarations, you don't put the type information (int, FILE *) etc. when you call it!*

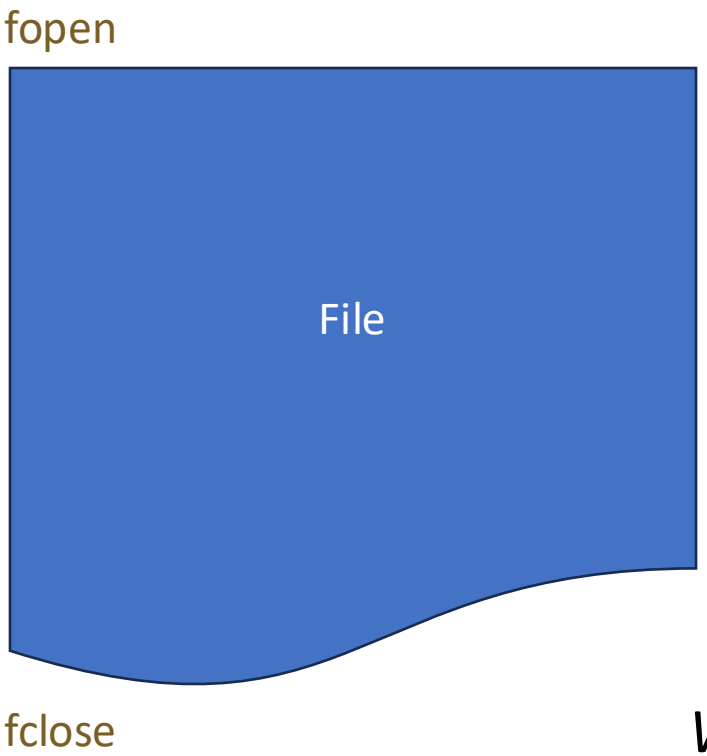
Example, copying a file:

```
/* filecopy: copy file ifp to file ofp */

void filecopy(FILE *ifp, FILE *ofp)
{
  int c;

  while ((c = getc(ifp)) != EOF)
    putc(c, ofp);
}
```

```
int main()
{
  FILE *inFile = fopen("input", "r"),
       *outFile = fopen("output", "w");

  if (inFile && outFile) {
    filecopy(inFile, outFile);

    fclose(outFile);
    fclose(inFile);
  }

  return 0;
}
```

3. Working with lines of text (text files)

```
char *fgets(char *line, int maxline, FILE *fp)
```




```
char line[80];
```


```
while (fgets(line, 80, fp) != NULL) {
  // Do something with line
}
```

What happens if the line of text in the file is longer than 80?

Note: others include fprintf, fscanf, fputs…

4. Reading and writing ‘records’ (binary files)

```
size_t fread(void *ptr, size_t size, size_t nobj, FILE *fp);
```

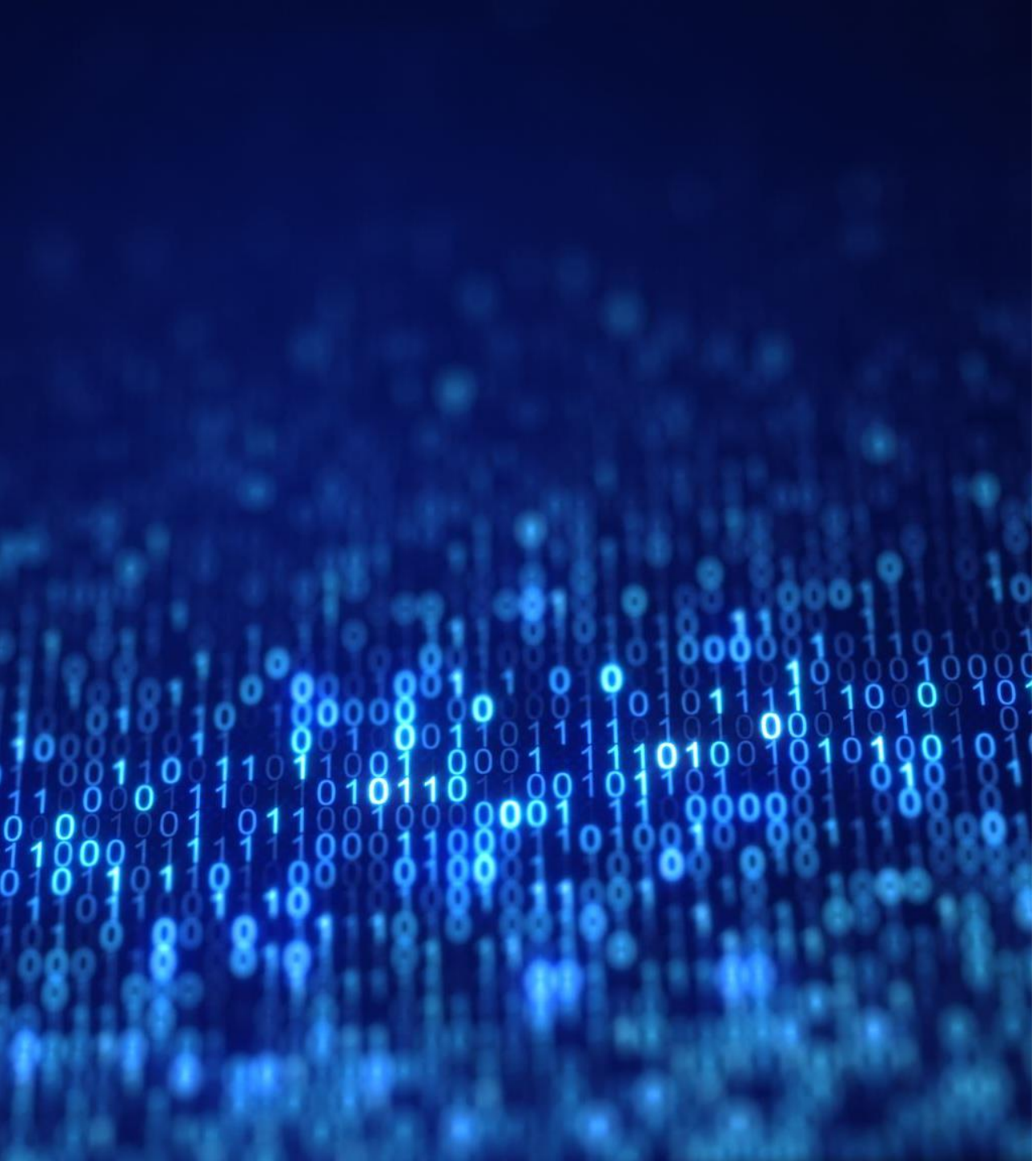
- fread reads from stream into the array ptr at most nobj objects of size size
- fread returns the number of objects read; this may be less than the number requested
- feof and ferror must be used to determine status

- There’s also:

```
size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *fp);
```

To call API functions, we just need to follow the rules ☺

1. Include the *correct* header file (to define the API functions)
2. Get the order and type of the parameters right for each function you call
3. Remember to call them in the right order (open before read/write, close – *but only if open was successful!)*
4. Open the file correctly (read, write, append, text, binary…)
5. Pay close attention to subtleties (are we providing a pointer or an array, for example…) – *check the book/man page*
6. Check return codes, you're interacting with real files!



Summary

- Discussed APIs and interacting with file system
- Basic file I/O using characters
- More advanced string based I/O