Files in C

- In C, each file is simply a sequential stream of bytes. C imposes no structure on a file.

- A file must first be opened properly before it can be accessed for reading or writing. When a file is opened, a stream is associated with the file.

- Successfully opening a file returns a pointer to (i.e., the address of) a file

File structure

*** FILE - a structure containing the information about a file or text stream needed to perform input or output operations on it, including:**

- **a file descriptor**
- **the current stream position**
- **an end-of-file indicator**
- **an error indicator**
- **a pointer to the stream's buffer, if applicable**

Files in C

- The statement:

  FILE *fptr1, *fptr2 ;

  declares that *fptr1* and *fptr2* are *pointer* variables of type FILE. They will be assigned the address of a file descriptor, that is, an area of memory that will be associated with an input or output stream.

- Whenever you are to read from or write to the file, you must first open the file

Opening Files

- The statement:

```
        fptr1 = fopen ( "mydata", "r" ) ;
```

  would open the file *mydata* for input (reading).

- The statement:

```
        fptr2 = fopen ("results", "w" ) ;
```

  would open the file *results* for output (writing).

- Once the files are open, they stay open

Testing for Successful Open

- If the file was not able to be opened, then the value returned by the *fopen* routine is NULL.

- For example, let's assume that the file *mydata* does not exist. Then:

```
    FILE *fptr1 ;
    fptr1 = fopen ( "mydata", "r") ;
    if (fptr1 == NULL)
    {
                printf ("File 'mydata' did not
open.\n") ;
```

File pointers predefined in stdio.h

Name	Notes
stdin	a pointer to a FILE which refers to the standard input stream, usually a keyboard.
stdout	a pointer to a FILE which refers to the standard output stream, usually a display terminal.
stderr	a pointer to a FILE which refers to the standard error stream, often a display terminal

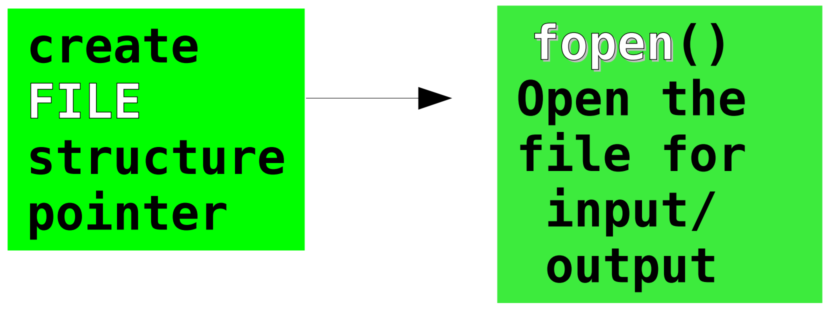
Standard IO

- When a program begins execution, three file streams are already defined and open.
  - **`stdin`**, standard input
  - **`stdout`**, standard output
  - **`stderr`**, standard error
- The first two are sent to “normal” IO. Typically the keyboard and screen.
- The first two are buffered by default. Minimise expensive system calls by sending data in chunks. Can control buffering via the standard function **`setbuf()`**.
- The **`stderr`** stream is reserved for sending error messages. It is typically directed to the screen and is unbuffered.

Functions in stdio.h are divided into two categories: file manipulation and input-output.

Name	Function
fopen()	opens a file for certain types of reading or writing - returns FILE pointer
fclose()	closes a file associated with the FILE * value passed to it
rewind()	positions to beginning of file
fseek()	position to any location within file
feof()	check if end-of-file indicator has been set
ferror()	checks whether an error indicator has been set for a given stream

Using File Input/Output


create FILE structure pointer
fopen() Open the file for input/ output

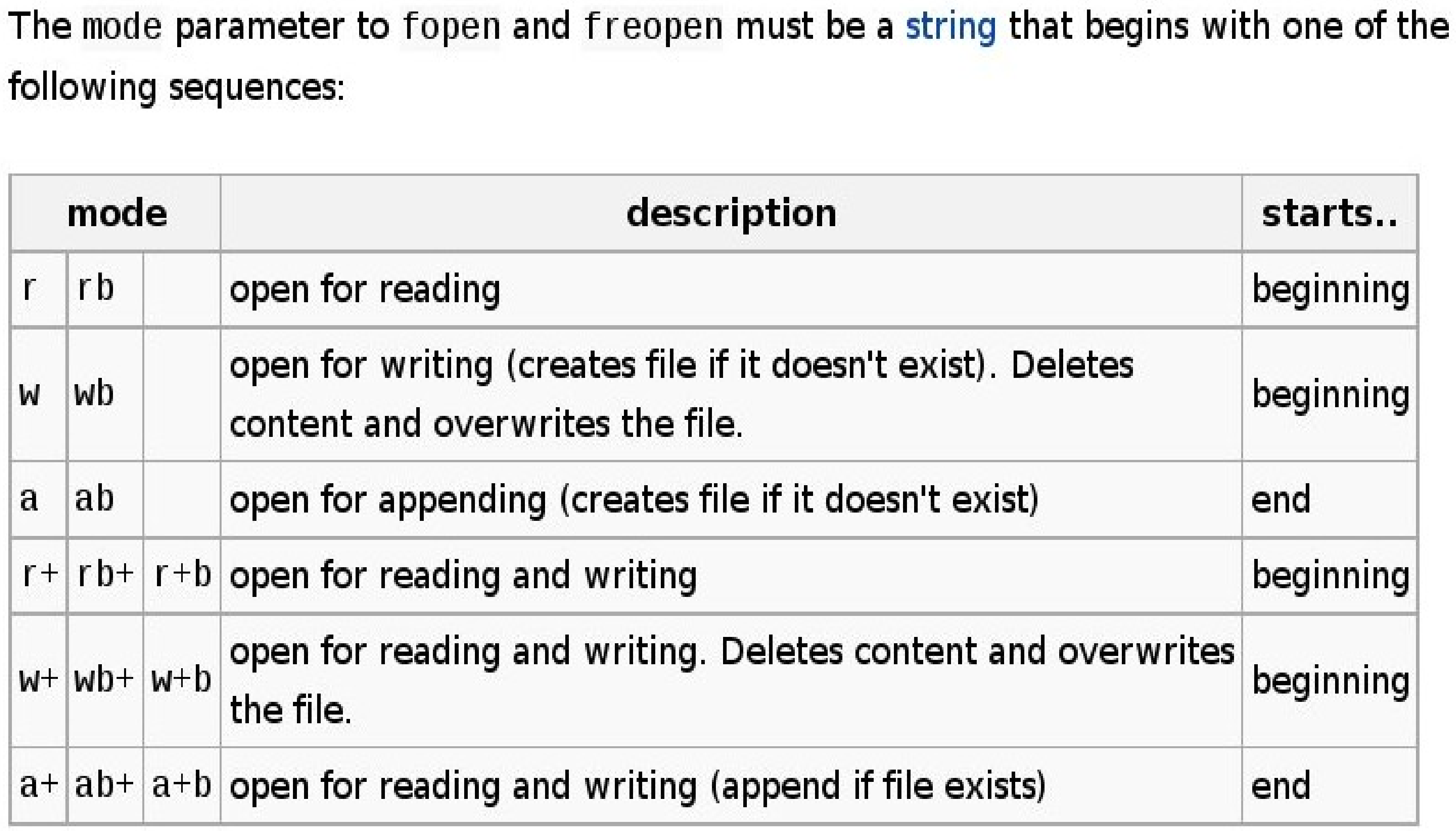
`fopen()`

- A file is referred to by a file-pointer. This is a pointer to a structure **`typedef`** called **`FILE`**.
- The **`FILE`** structure is only ever accessed by a pointer. It hides its members behind abstract type-name, and is manipulated solely by standard IO functions.
- To open a file, call **`fopen()`**.

```
•  if ( (fp = fopen("direct.txt", "wb")) ==
   NULL)
•          {
•                  fprintf(stderr, "Error opening
   file.");
•                  exit(1);
•          }
•
```

- Two arguments:
  1. The file name. eg, **`myfile.txt`**
  2. The file mode. **`"r", "w", "rb", "wb"`**
- Return value: Pointer to file if successful, NULL if

Fopen Mode Parameter



The `mode` parameter to `fopen` and `freopen` must be a string that begins with one of the following sequences:

mode			description	starts..
`r`	`rb`		open for reading	beginning
`w`	`wb`		open for writing (creates file if it doesn't exist). Deletes content and overwrites the file.	beginning
`a`	`ab`		open for appending (creates file if it doesn't exist)	end
`r+`	`rb+`	`r+b`	open for reading and writing	beginning
`w+`	`wb+`	`w+b`	open for reading and writing. Deletes content and overwrites the file.	beginning
`a+`	`ab+`	`a+b`	open for reading and writing (append if file exists)	end

`fclose()`

- To close a file, pass the file pointer to **`fclose()`**.
- General form:

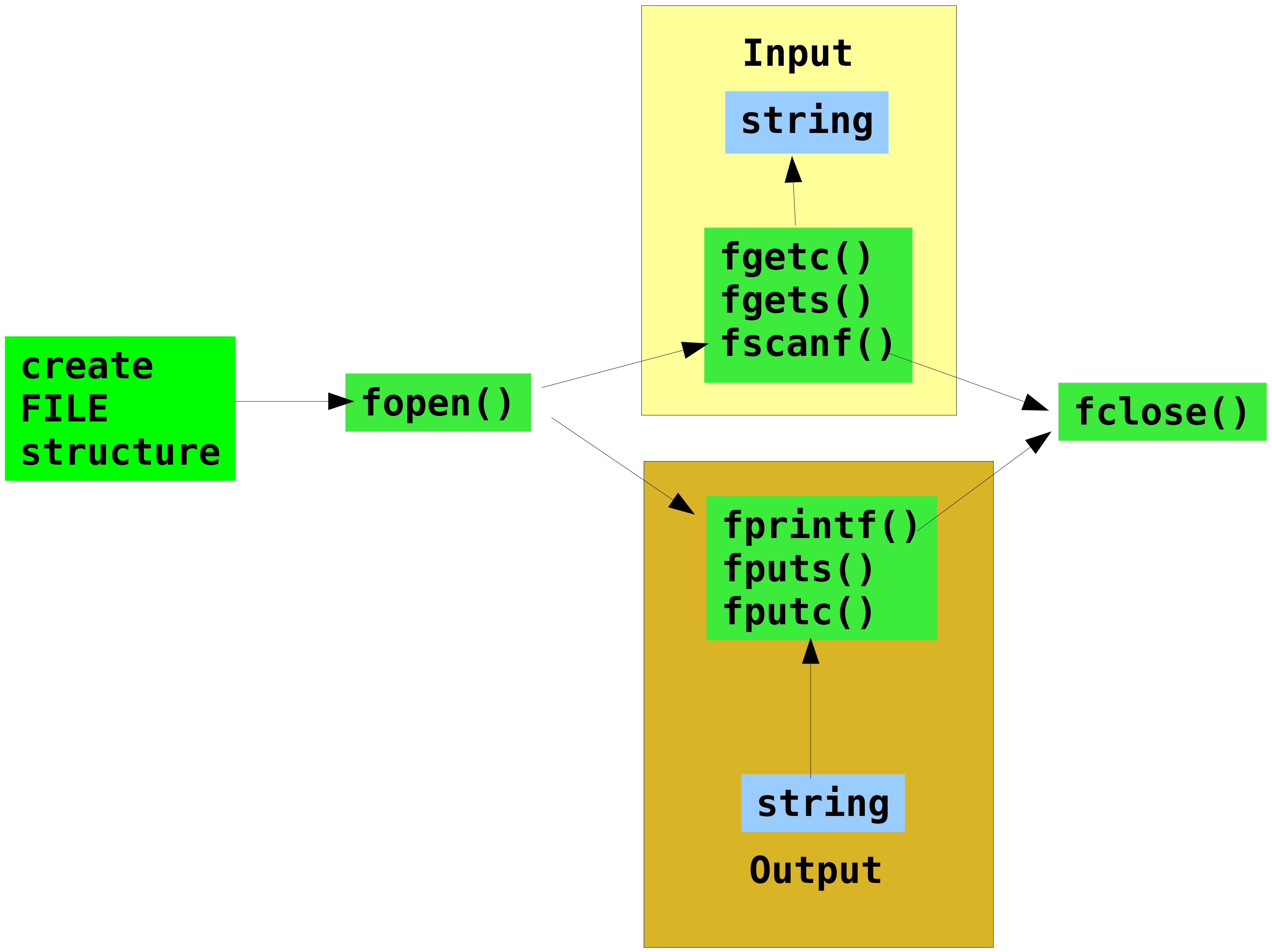
```
int fclose(FILE *fp);
```

- **`fclose()`** breaks the connection with the file and frees the file pointer.
- Good practice to free file pointers when a file is no longer needed as most OSs have a limit on the number of files a program may have open at any given time.
- Note, **`fclose()`** is called automatically for each open file when the program terminates.

Sequential File Operations

- Once a file is open, operations on the file (reading and writing) usually work through the file sequentially – from the beginning to the end.
- There are four basic types of file IO:
  - Character by character.
  - Line by line.
  - Formatted IO.
  - Binary IO.

Text File I/O


create FILE structure
fopen()
Input
string
fgetc()
fgets()
fscanf()
fclose()
fprintf()
fputs()
fputc()
string
Output

Character Input

- Character input functions:
  - **fgetc() returns one character from a file**
  - **fgets() gets a string from the file (ending at newline or end-of-file)**
  - **fscanf()   works like the original scanf function**

- Return values:
  - On success: the next character in the input stream.
  - On error: `EOF`.
  - On end-of-file: `EOF`.
- If return value is `EOF`, can determine what caused it by calling either `feof()` or `ferror()`.

Character Output

- Character output functions:
  - fputc() writes one character to a file
  - fputs() writes a string to a file
  - fprintf() enables printf output to be written to any file
- **`putchar(c)`** is equivalent to **`putc(c, stdout)`**.
- Return values:
  - On success: the character that was written.
  - On error: **`EOF`**.

Example

```
FILE *fp;
int c;

fp = fopen(“myfile.txt”, “r”);
if (fp == NULL)
  exit(1);

while((c = getc(fp)) != EOF)
  putc(c, stdout);

fclose(fp);
```

Formatted IO

```
int fprintf(FILE *fp, const char *format, ...);
int fscanf(FILE *fp, const char *format, ...);
```

- These functions are generalisations of **`printf()`** and **`scanf()`**, respectively.
- In fact, **`printf()`** and **`scanf()`** are equivalent to

```
fprintf(stdout, format, arg1, arg2, ...);
fscanf(stdin, format, arg1, arg2, ...);
```

Line (string) Input

- Read a line from a file:
  ```
  char *fgets(char *buf, int max, FILE *fp);
  ```
- Returns after one of the following:
  - Reads (at most) **`max-1`** characters from the file.
  - Reads a **`\n`** character.
  - Reaches end-of-file.
  - Encounters an error.
- Return values:
  - On success: pointer to **`buf`**. Note, **`fgets()`** automatically appends a **`\0`** to the end of the string.
  - On end-of-file: **`NULL`**.
  - On error: **`NULL`**.
- Use **`feof()`** or **`ferror()`** to determine if an error has occurred.

Line Output

- Character strings may be written to file using
```
int fputs(const char *str, FILE *fp);
```
- Not actually line output. It does not automatically append a **`\n`** and consecutive calls may print strings on the same line.
- Return values:
  - On success: zero.
  - On error: **`EOF`**.

Binary IO

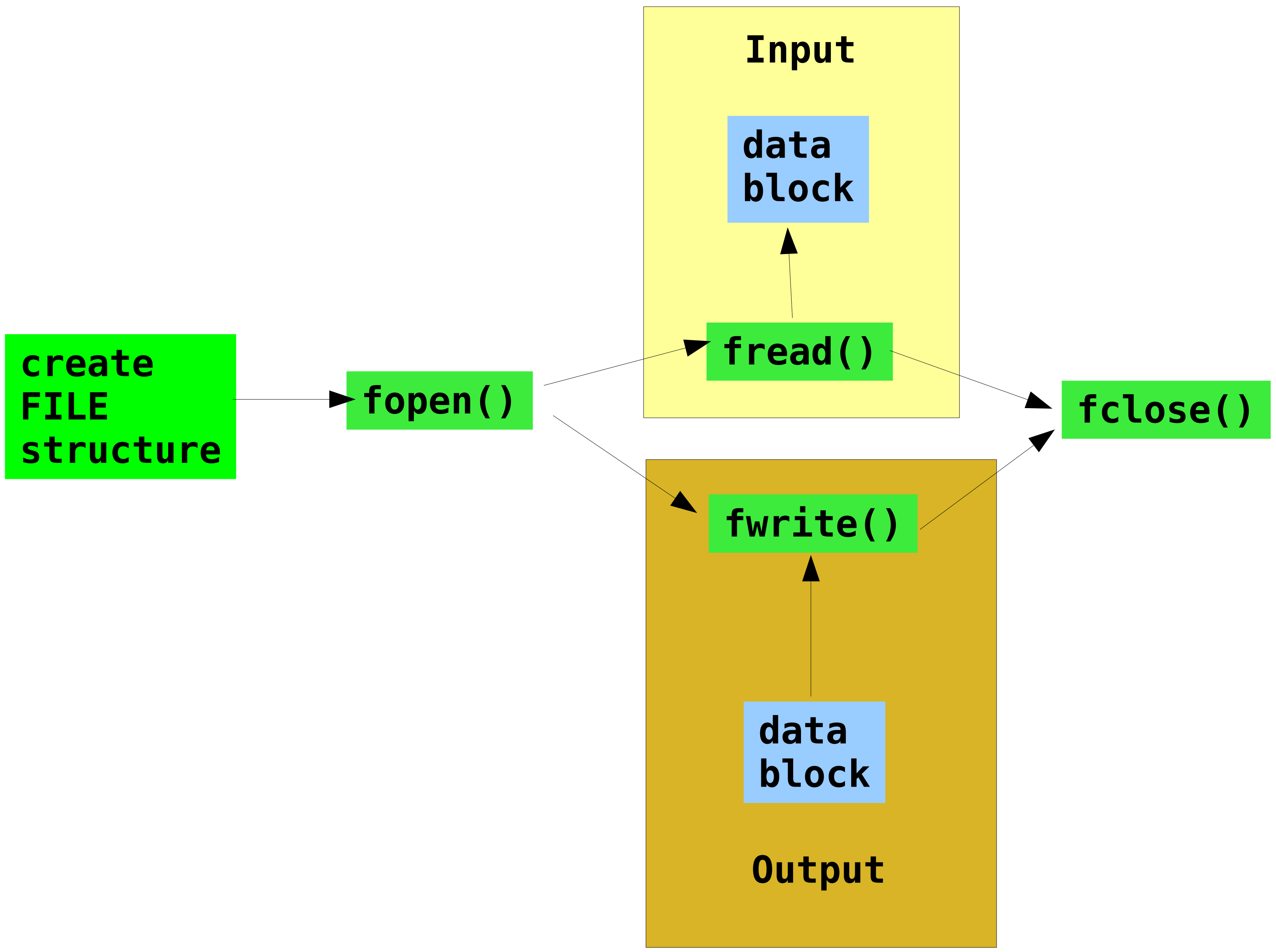
- When reading and writing binary files, may deal with objects directly without first converting them to character strings.
- Direct binary IO provided by

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

- Can pass objects of any type. For example,

```
struct Astruct mystruct[10];
fwrite(&mystruct, sizeof(Astruct), 10, fp);
```

Binary File I/O



Binary File Input

```
/* Read the data into array[SIZE]. */

    if (fread(array, sizeof(int), SIZE, fp) != SIZE)
    {
        fprintf(stderr, "Error reading file.");
        exit(1);
    }
```

Binary File Output

```
/* Save array[SIZE] to the file. */

    if (fwrite(array, sizeof(int), SIZE, fp) != SIZE)
    {
        fprintf(stderr, "Error writing to file.");
        exit(1);
    }
```

Random File Operations

- IO is not confined to sequential motion through a file. May also shift the file position back and forth to any specified location.
- Three functions:

```
long ftell(FILE *fp);
int fseek(FILE *fp, long offset, int from);
void rewind(FILE *fp);
```

- Operate differently on text files as to binary files.

Where are you in the file? - ftell()

```
/* Rewind the stream. */

     rewind(fp);

     printf("\n\nAfter rewinding, the position is back at %ld",
             ftell(fp));
```

Seek to a specific position in file

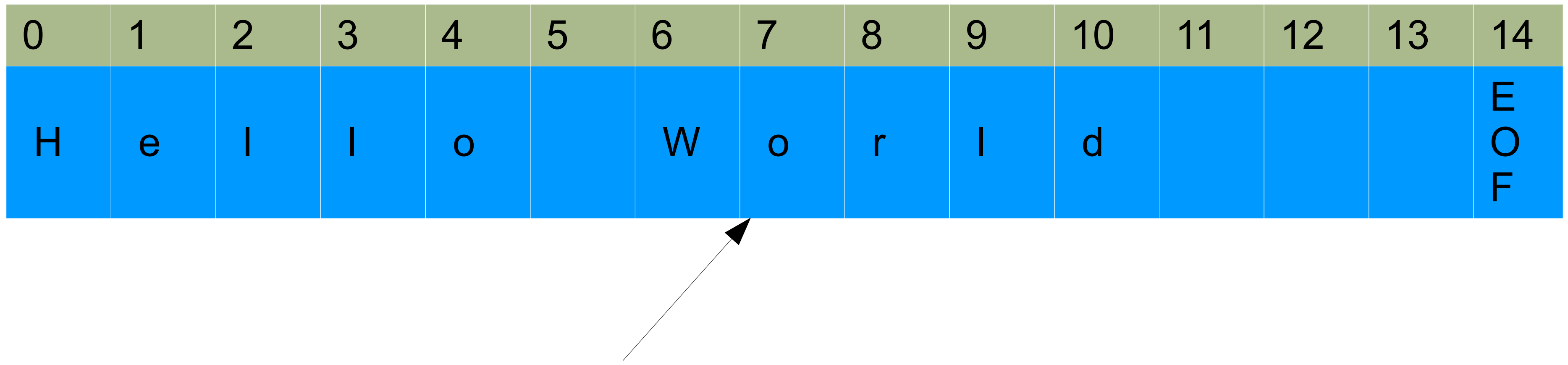
```
Function prototype

    int fseek(FILE *stream_pointer, long offset, int origin);

The fseek function moves the file pointer associated with the
stream to a new location that is offset bytes from origin

Argument meaning:

    * stream_pointer is a pointer to the stream FILE structure
of which the position indicator should be changed;
    * offset is a long integer which specifies the number of
bytes from origin where the position indicator should be
placed;
    * origin is an integer which specifies the origin position.
It can be:
          o SEEK_SET: origin is the start of the stream
          o SEEK_CUR: origin is the current position
          o SEEK_END: origin is the end of the stream
```

Position to a single integer

```
/* Move the position indicator to the specified element. */

     if ( (fseek(fp, (offset*sizeof(int)), SEEK_SET)) != NULL)
     {
         fprintf(stderr, "\nError using fseek().");
         exit(1);
     }

     /* Read in a single integer. */

     fread(&data, sizeof(int), 1, fp);
```

fseek example

File Contents - characters stored in file



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
H	e	l	l	o		W	o	r	l	d				EOF

```
fseek(fp, (7*sizeof(char)), SEEK_SET)
```