C Lecture 7

File I/O

File I/O

- To date, we've used various I/O functions without really exploring where they come from.

- We're going to take a slightly more in-depth look at how C does I*nput* and O*utput* here.

- Our context is "writing and reading to files"…

    - but *everything* in Linux is a file, so this also applies to the other I/O we've done.

- All I/O functions are declared in `stdio.h` (and implemented in libc)

"Stream I/O"

- C provides several ways to think about I/O. In this course, we will consider two kinds, both "stream I/O" models.

- Stream I/O considers everything as a "stream" of "characters".

- Functions can take some characters out of a stream (reading).

- Functions can also put some characters into a stream (writing).

- A stream can represent any file (or file-like thing - such as the special stdio, stdin, stderr files for handling user input/output).

Text Stream I/O

This is some text that is in the input stream.\n More lines of text...

This is some text that is in the input stream.\n More lines of text...

fgets(…)

fgets takes characters from the stream,
until it reaches a newline

More lines of text... Continue on here until \n We just continue the

After fgets returns, the stream no longer contains the characters it took.

(If the stream represents a file, then the file itself isn't changed - we've just "moved on in the file" past those characters.)

ASCII

- The "characters" in the stream are encoded in the format known as ASCII (American Standard Code for Information Interchange).

- This is also what is used for your `char` variables.

- It assigns values from 0 to 127 to different symbols (including the Latin letters, lowercase and uppercase).

ASCII cannot represent non-Latin alphabets. More modern languages than C use other codes - Unicode, for example - which can represent many more characters, alphabets, etc.

NULL = \0 character

space character

newline character

ASCII Symbol Table

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	(NUL)	(SOH)	(STX)	(ETX)	(EOT)	(ENQ)	(ACK)	(BEL)	(BS)	(HT)	(LF)	(VT)	(FF)	(CR)	(SO)	(SI)
(16)+	(DLE)	(DC1)	(DC2)	(DC3)	(DC4)	(NAK)	(SYN)	(ETB)	(CAN)	(EM)	(SUB)	(ESC)	(FS)	(GS)	(RS)	(US)
(32)+	(SP)	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
(48)+	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
(64)+	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
(80)+	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
(96)+	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
(102)+	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	(DEL)

Codes in () are non-printing characters (most of them were originally designed to control automatic typewriters). The relevant ones for our use are indicated.

Opening a File

Name of file to open

- `FILE * fopen(char[] filename, char[] mode)`

- A `FILE` is a special structured type which represents the stream of data flowing to or from a file.

- fopen will try to open a file with the path and name we specify, and return a pointer to a `FILE` representing it.

- We also need to specify how we are going to access the file.

The special file pointers stdout, stdin and stderr are automatically opened when a program starts and correspond to the same concepts you met in the Bash part of the course.

File access modes

File mode	Intent	Equivalent Bash operator	Effect of adding a +
`"r"`	Just read the file	<	`"r+"` - read and write the file (from the start)
`"w"`	Just overwrite the file	>	`"w+"` - overwrite the file, but allow reading too
`"a"`	Just write stuff to end of file	>>	`"a+"` - start writing to end of file (but allow reading too)

```
FILE * fp = fopen("myfile.txt", "r");
```

Reading from a File

- `char * fgets(char * string, int len, FILE * file);`
- `int fscanf(FILE * file, char * format, …)`

Any number of pointers to variables to match format

- fscanf returns an int, which is the number of variables it successfully put values into.
- (This can be less than the number requested, if it was unable to interpret some of the stream in the requested way.)
- fscanf returns `EOF` if they reach the end of a file (and therefore there's nothing more to read).

fscanf(stdin, "%d is an integer\n", &myint) is identical to scanf("%d is an integer\n", &myint)

fscanf

900 6 8.000\n980 h 8.100\n577 86 6.456\n8 77 8.100\n900 6 8.000\n

900 6 8.000\n980 h 8.100\n577 86 6.456\n8 77 8.100\n900 6 8.000\n

fscanf(fp,"%d %d %f\n",…)

format matches next characters in stream
(fscanf returns 3)

980 h 8.100\n577 86 6.456\n8 77 8.100\n900 6 8.000\n7 34 4.4359\n

fscanf(fp,"%d **%d** %f\n",…)

format can't match second conversion specifier
('h' is not interpretable as an int)

h 8.100\n577 86 6.456\n8 77 8.100\n900 6 8.000\n7 34 4.4359\n789

stream is left at point of first "non-matching" characters.
fscanf returns 1; first variable (pointer) passed to fscanf is assigned value 980 (as an int), the other two are *unchanged*.

Writing to a File

- `int fputs(char * string, FILE * file);`
- `int fprintf(FILE * file, char * format, …);`

Any number of values to match format

- fputs and fprintf both return an int - for fprintf, this is the number of characters it wrote.
- If they fail to write to the file, they instead return the special value `EOF`.
- Unlike puts, fputs does not add a '\n' to output.

```
fprintf(file, "%d is an integer\n", myint);
```

fprintf(stdout, "%d is an integer\n", myint) is identical to printf("%d is an integer\n", myint)

Streams and Buffers

- Strictly, writes to a stream are not immediately reflected in the file they represent.

- (Physical media like hard disks, or optical disks, takes real time to write stuff.)

- Instead, the writes are "queued up" in a buffer of things needing to be written.

- Periodically, the buffer is emptied, and its contents actually committed to the file.

Closing a File

- `int fflush(FILE * fp);`

- fflush makes sure that all the data we've written so far has actually been committed to the file itself (it flushes the buffer immediately).

- `int fclose(FILE * fp);`

- fclose does a flush, and then removes the connection between fp and the file it's attached to. If successful, it returns 0.

- After closing a file, the file pointer *cannot* be used for I/O until assigned a new file with an fopen.

NEVER fclose stdin,stdout or stderr! (fflush can be useful to ensure stuff is printed immediately).

General I/O Functions

- In general, I/O functions are written in the form objectverbsubject.

- Where object can be **f**, for files, **s**, for strings, or omitted, for “default/terminal”.

- (There are other options, such as **sn** for ”string, along with a max length”, but we leave those for the documentation.)

- And subject is **f** for “use formatted values”, **s** for “use a string”, or **c** for “use a single character”.

- sprintf, e.g., uses a format string to convert values to text, and stores the result in a string.

- (There are some violations of these general rules, especially for the get and put families, and some combinations do not exist, so check the documentation before using.)

Stream I/O Example

```c
#include <stdio.h>

int main(void){

        FILE * fp = fopen("testfile.txt","w");

        fprintf(fp, "Number %d, %f", 5, 5.0);

        fflush(fp); //ensure everything written to file
        fclose(fp); //although fclose would do this for us

        fp = fopen("testfile.txt", "r");

        int a;
        double b;

        fscanf(fp, "Number %d, %lf", &a, &b);

        fclose(fp);

        printf("Double in file was %f\n", b);

        return 0;

 }
```

(Strictly unneeded here, as we fclose immediately after)

(ASCII) text representation of text and numeric values

```
hexdump -C testfile.txt
00000000  4e 75 6d 62 65 72 20 35  2c 20 35 2e 30 30 30 30  |Number 5, 5.0000|
00000010  30 30                                             |00|
00000012
```

text '5' not value 5

text '5.000' not double

Binary Stream I/O

- Rather than opening a file as if it were text, we may want to access the bytes in a file directly.
    - This can be more efficient than text stream I/O
    - Also doesn't lose precision (consider floating point values).

I/O type	Text Stream	Binary Stream
Pros	Human readable	Exact representation of variables
Cons	Precision loss Often larger than needed	Opaque Less portable

Opening, Closing Binary Streams

- We can still use fopen and fclose to work with Binary Stream I/O.

- There's just one change: the file mode must have a "b" added to it, to ensure we directly read precisely what's in the file.

- So, "r" -> "rb", "w+" -> "wb+" and so on.

```
FILE * fp = fopen("myfile.txt", "rb");
```

The C standard allows text streams to do "conversion" between the file and the stream itself. On Linux (and other POSIX) systems, no such conversion occurs, but other platforms, including Windows, do. For portability, *always* explicitly turn on binary mode for binary I/O.

Reading, Writing Binary

- `long fread(void * start, long size, long num, FILE * file);`
- `long fwrite(void * start, long size, long num, FILE * file);`

•fread reads `size*num` bytes from `file`, and inserts them into memory starting at `start`. (It returns how many it actually wrote).

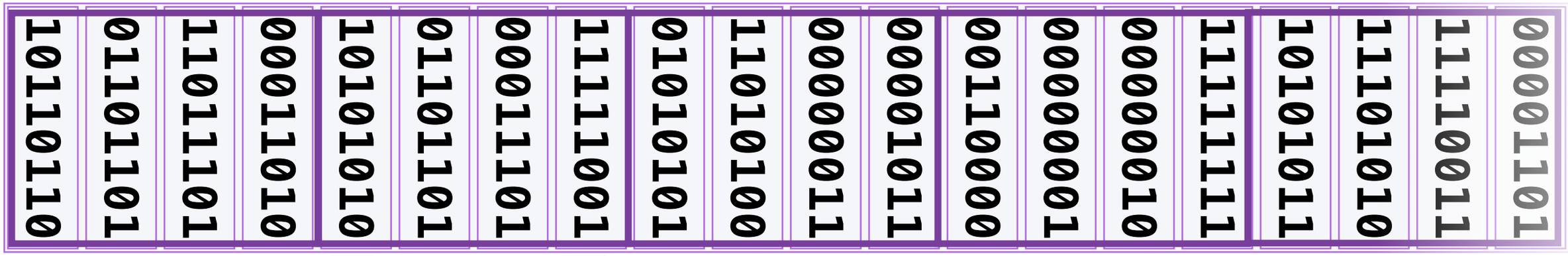
•fwrite does the same thing, but takes *from* `start`, and writes *into* `file`.

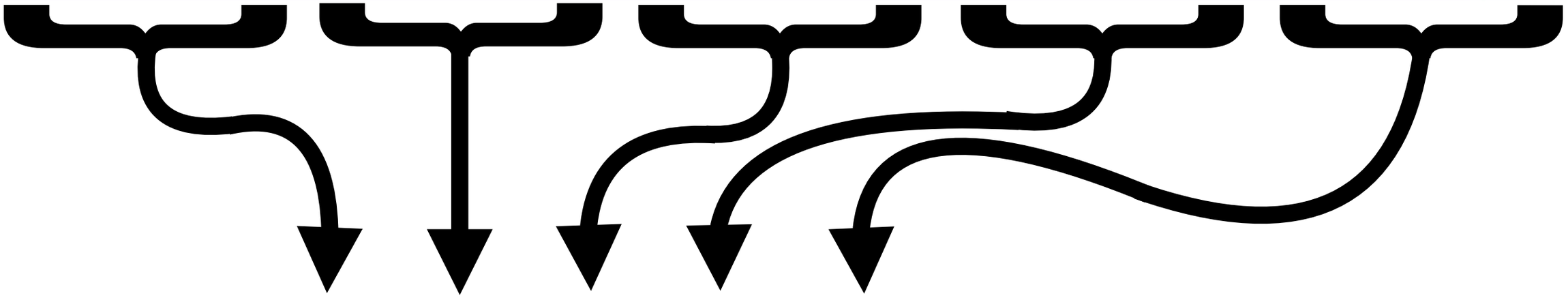
```
int a[5];
fread(a,sizeof(int),5,fp);
```

Reading, Writing Binary

fp

```
int a[5];
fread(a,sizeof(int),5,fp);
```





a

a[0] a[1] a[2] a[3] a[4]

Here, for example a[0] might be set to 450719158 (binary 00011010110111010110110110110110)

Binary Stream I/O Example

```c
#include <stdio.h>
#include <string.h>

int main(void){

        char str[] = "Example string";

        int a[5] = {0,1,2,3,4};

        FILE * fp = fopen("testfile.bin","wb");

        fwrite(str,sizeof(char),strlen(str), fp);
        fwrite(a,sizeof(int),5, fp);

        fclose(fp);

        return 0;
}
```

Binary version of text… is still text

```
 hexdump -C testfile.bin
00000000  45 78 61 6d 70 6c 65 20  73 74 72 69 6e 67 00 00  |Example string..|
00000010  00 00 01 00 00 00 02 00  00 00 03 00 00 00 04 00  |................|
00000020  00 00                                            |..|
```

Direct binary values of int type (don't look like text)

Additional note: moving around in files

- `int fseek(FILE * fp, long int offset, int whence);`

- fseek, for a file which we can move around in, moves us to a new location. The new position is *offset* bytes after the position described by *whence*. (So, use negative values to move "backward").

- *whence* can be set to 3 specific values (#DEFINE d in **stdio.h**):
  - **SEEK_SET** (the start of the file) - eg fseek(fp, 0, SEEK_SET) start of file
  - **SEEK_CUR** (the current position we are in the file)
  - **SEEK_END** (the end of the file) - eg fseek(fp, 0, SEEK_END) end of file

- fseek returns **0** if successful, and any other number if failed.

- (You can't, for example, "seek" in a terminal IO stream like stdin.)

Additional note: moving around in files

- `long int ftell(FILE * fp);`

- ftell returns a *long int* value equal to the number of bytes "into" the file we currently are. (It will return **-1** if it can't: for example, it makes no sense to ask where we are in *stdin*.)