C Lecture 7

File I/O

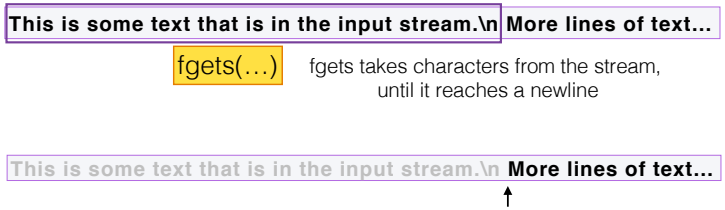
File I/O

- To date, we've used various I/O functions without really exploring the whole subject of input-output.
- We're going to take a slightly more in-depth look at how C does input and output here.
- Our context is "writing and reading to files"…
  - but *everything* we do I/O to in Linux is a file (including keyboard and terminal), 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"

- In this course, we will consider two kinds, of "stream I/O" ("text" and "binary").
- You can think of a stream as a logical interface between your program and a "file".
- Functions can stream characters out of a file (reading) or stream them into a file (writing).
- A stream can connect your program to any file (or file-like thing - such as the special STDIO, STDIN, STDERR files for handling user input/output).

Text Stream I/O



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 – but we've "moved on in the file" past those characters.

ASCII

- The "characters" in a text 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).
- A Text I/O stream, therefore, is a sequence of ASCII symbols.

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.

ASCII: American Standard Code for Information Interchange.

NULL = \0 character

space character

newline character

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Opening a File

```c
#include <stdio.h>

int main(void)
{
    char mytext[100];

    FILE *file_ptr;
    file_ptr = fopen("myFile.txt", "r");
    fgets(mytext,sizeof(mytext),file_ptr);
    puts(mytext);
    return 0;
}
```

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

fopen(char[] filename, char[] mode) returns a pointer to type file

The mode specifies how we are going to access the file (read, write, append…)

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)

Truncates (erases contents!) if it opens an existing file. Otherwise creates a new file

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

Reading from a File - fgets

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

```c
#include <stdio.h>

int main(void)
{
    char mytext[100];
    char *p1;

    FILE *file_ptr;
    file_ptr = fopen("myFile.txt", "r");
    p1 = fgets(mytext,sizeof(mytext),file_ptr);
    puts(mytext);
    puts(p1);
//  puts(fgets(mytext,sizeof(mytext),file_ptr));
    return 0;
}
```

Reading from a File - fscanf

- `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 it reaches 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)

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". sscanf() printf()
- (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 "formatted values", **s** for "to string", or **c** for "single character". getc()
- 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.)

fscanf() Example

```c
#include <stdio.h>

int main(void)
{
    char mytext[100];
    int i = 0,j = 0;
    float a = 0.0;

    FILE *file_ptr;
    file_ptr = fopen("myFile.txt", "r");
    puts(fgets(mytext,sizeof(mytext),file_ptr));
    puts(fgets(mytext,sizeof(mytext),file_ptr));
    fscanf(file_ptr,"%d %f \n", &i, &a);
//  j = fscanf(file_ptr,"%d %f \n", &i, &a);

    printf("Read in %d %f %d \n",i,a, j );

    return 0;
}
```

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 are connected to.
- (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).

Text Stream I/O Example

```c
#include <stdio.h>

int main(void)
{
    FILE *fp;
    fp = fopen("testfile.txt", "w");

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

    fflush(fp); //ensure everything written - not needed in this example
    fclose(fp); //fclose flushes buffer before closing file

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

    int a;
    float b;

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

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

    return 0;
}
```

```
mac-db2016:lecture7 Dave$ hexdump -C testfile.txt
00000000  4e 75 6d 62 65 72 20 35  20 35 2e 30 30 30 30 30  |Number 5 5.00000|
00000010  30                                                |0|
00000011
```

Binary Stream I/O

- Rather than opening a file and streaming the contents as if it were text, we may want to stream the bytes to/from a file directly.
- A file created this way is called a binary file. Bits in text files represent ASCII characters, but the bits in binary files represent the actual custom data.
  - Binary stream I/O can be more efficient than text stream I/O
  - Also doesn't lose precision of floating point values.
  - But you can't just read a binary file because you don't know how to interpret the contents (is the first byte part of an integer? Or a character? Or what?)

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");
```

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);
```

Binary Stream I/O Example

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

int main(void)
{
    FILE *fp;
    fp = fopen("testfile.bin", "wb");

    char str[] = "Example String";
    int a[5] = {0,1,2,3,4};

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

    char str2[] = "00000000000000";
    int b[5] = {0,0,0,0,0};

    fp = fopen("testfile.bin", "rb");
    fread(str2, sizeof(char),strlen(str2),fp);
    fread(b, sizeof(int), 5, fp);

    printf("Read in %s %d,%d,%d,%d,%d\n", str2, b[0],b[1],b[2],b[3],b[4]);

    return 0;
}
```