The File System

Files in Linux

Stefano Quer and Stefano Scanzio

Dipartimento di Automatica e Informatica

Politecnico di Torino

skenz.it/os stefano.scanzio@polito.it

File System

- The file system is one of the most visible aspects of an OS
- It provides mechanisms to save data (permanently)
- It includes management of
  - Files
  - Direttories
  - Disks and disk partitions

Files

- Information is store for a long period of time
  - independently from
    - Termination of programs/processes, power supply, etc.
- From the logical point of view a file is

1 file

  - A set of correlated information
    - All information (i.e., numbers, characters, images, etc.) are stored in a (electronic) device using a **coding system**
  - Contiguous address space

How is this information encoded?

What is the actual organization of this space?

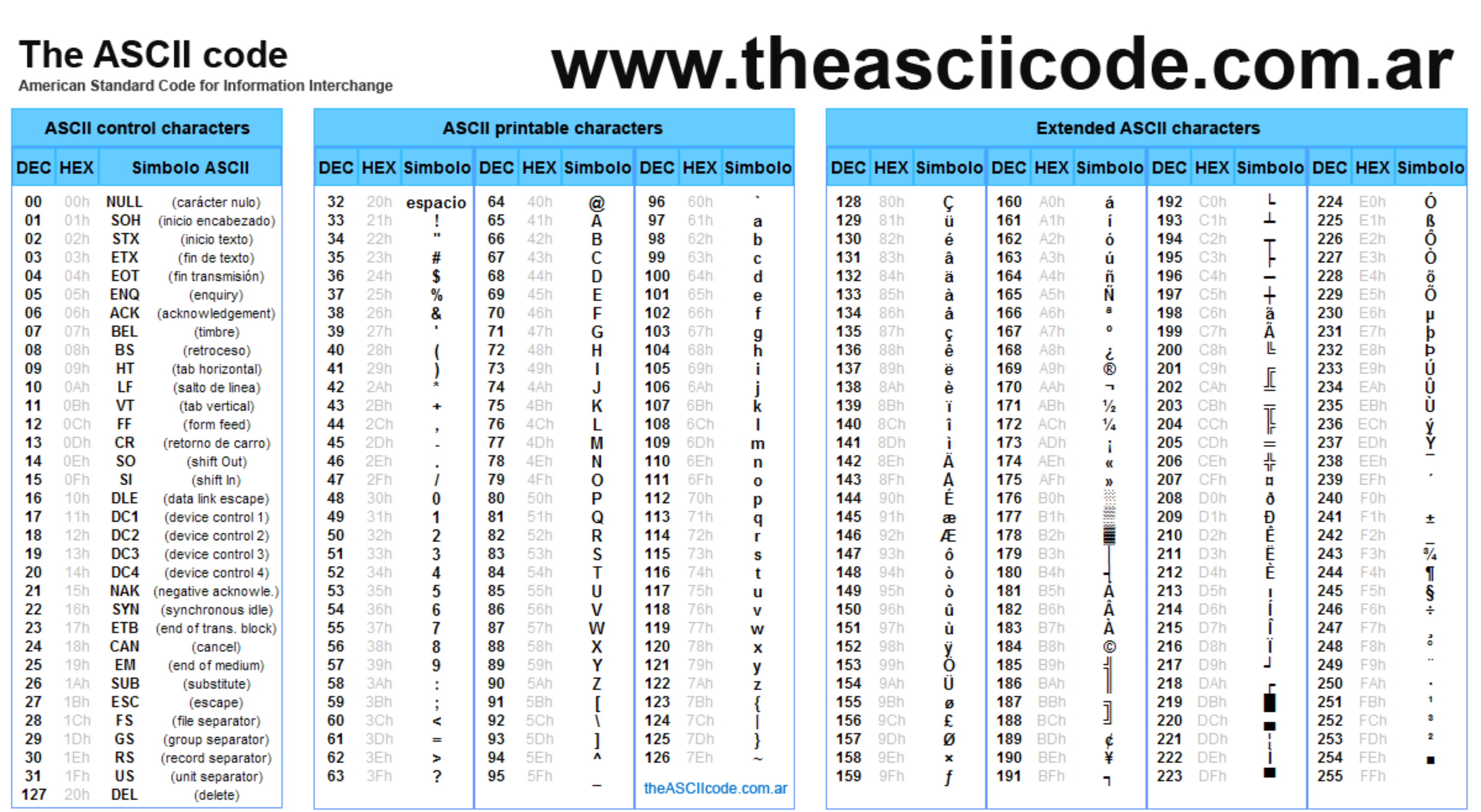
ASCII encoding

128 total characters
32 not printable
96 printable

- De-facto standard
  - ASCII, American Standard Code for Information Interchange
    - Originally based on the English alphabet
    - 128 characters are coded in 7-bit (binary numbers)
  - Extended ASCII (or high ASCII)
    - Extension of ASCII to 8-bit and 255 characters
    - Several versions exist
      - ISO 8859-1 (ISO Latin-1), ISO 8859-2 (Eastern European languages), ISO 8859-5 for Cyrillic languages, etc.

The alphabet of Klingom language is not supported by Extended ASCII

Extended ASCII table



The ASCII code

American Standard Code for Information Interchange

www.theasciicode.com.ar

ASCII control characters

DEC	HEX	Simbolo ASCII	
00	00h	NULL	(carácter nulo)
01	01h	SOH	(inicio encabezado)
02	02h	STX	(inicio texto)
03	03h	ETX	(fin de texto)
04	04h	EOT	(fin transmisión)
05	05h	ENQ	(enquiry)
06	06h	ACK	(acknowledgement)
07	07h	BEL	(timbre)
08	08h	BS	(retroceso)
09	09h	HT	(tab horizontal)
10	0Ah	LF	(salto de linea)
11	0Bh	VT	(tab vertical)
12	0Ch	FF	(form feed)
13	0Dh	CR	(retorno de carro)
14	0Eh	SO	(shift Out)
15	0Fh	SI	(shift In)
16	10h	DLE	(data link escape)
17	11h	DC1	(device control 1)
18	12h	DC2	(device control 2)
19	13h	DC3	(device control 3)
20	14h	DC4	(device control 4)
21	15h	NAK	(negative acknowle.)
22	16h	SYN	(synchronous idle)
23	17h	ETB	(end of trans. block)
24	18h	CAN	(cancel)
25	19h	EM	(end of medium)
26	1Ah	SUB	(substitute)
27	1Bh	ESC	(escape)
28	1Ch	FS	(file separator)
29	1Dh	GS	(group separator)
30	1Eh	RS	(record separator)
31	1Fh	US	(unit separator)
127	20h	DEL	(delete)

ASCII printable characters

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
32	20h	espacio	64	40h	@	96	60h	`
33	21h	!	65	41h	A	97	61h	a
34	22h	"	66	42h	B	98	62h	b
35	23h	#	67	43h	C	99	63h	c
36	24h	$	68	44h	D	100	64h	d
37	25h	%	69	45h	E	101	65h	e
38	26h	&	70	46h	F	102	66h	f
39	27h	'	71	47h	G	103	67h	g
40	28h	(	72	48h	H	104	68h	h
41	29h	)	73	49h	I	105	69h	i
42	2Ah	*	74	4Ah	J	106	6Ah	j
43	2Bh	+	75	4Bh	K	107	6Bh	k
44	2Ch	,	76	4Ch	L	108	6Ch	l
45	2Dh	-	77	4Dh	M	109	6Dh	m
46	2Eh	.	78	4Eh	N	110	6Eh	n
47	2Fh	/	79	4Fh	O	111	6Fh	o
48	30h	0	80	50h	P	112	70h	p
49	31h	1	81	51h	Q	113	71h	q
50	32h	2	82	52h	R	114	72h	r
51	33h	3	83	53h	S	115	73h	s
52	34h	4	84	54h	T	116	74h	t
53	35h	5	85	55h	U	117	75h	u
54	36h	6	86	56h	V	118	76h	v
55	37h	7	87	57h	W	119	77h	w
56	38h	8	88	58h	X	120	78h	x
57	39h	9	89	59h	Y	121	79h	y
58	3Ah	:	90	5Ah	Z	122	7Ah	z
59	3Bh	;	91	5Bh	[	123	7Bh	{
60	3Ch	<	92	5Ch	\	124	7Ch	|
61	3Dh	=	93	5Dh	]	125	7Dh	}
62	3Eh	>	94	5Eh	^	126	7Eh	~
63	3Fh	?	95	5Fh	_			

theASCIIcode.com.ar

Extended ASCII characters

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
128	80h	Ç	160	A0h	á	192	C0h	└	224	E0h	Ó
129	81h	ü	161	A1h	í	193	C1h	┴	225	E1h	ß
130	82h	é	162	A2h	ó	194	C2h	┬	226	E2h	Ô
131	83h	â	163	A3h	ú	195	C3h	├	227	E3h	Ò
132	84h	ä	164	A4h	ñ	196	C4h	─	228	E4h	õ
133	85h	à	165	A5h	Ñ	197	C5h	┼	229	E5h	Õ
134	86h	å	166	A6h	ª	198	C6h	ã	230	E6h	µ
135	87h	ç	167	A7h	º	199	C7h	Ã	231	E7h	þ
136	88h	ê	168	A8h	¿	200	C8h	╚	232	E8h	Þ
137	89h	ë	169	A9h	®	201	C9h	╔	233	E9h	Ú
138	8Ah	è	170	AAh	¬	202	CAh	╩	234	EAh	Û
139	8Bh	ï	171	ABh	½	203	CBh	╦	235	EBh	Ù
140	8Ch	î	172	ACh	¼	204	CCh	╠	236	ECh	ý
141	8Dh	ì	173	ADh	¡	205	CDh	═	237	EDh	Ý
142	8Eh	Ä	174	AEh	«	206	CEh	╬	238	EEh	¯
143	8Fh	Å	175	AFh	»	207	CFh	¤	239	EFh	´
144	90h	É	176	B0h	░	208	D0h	ð	240	F0h	
145	91h	æ	177	B1h	▒	209	D1h	Ð	241	F1h	±
146	92h	Æ	178	B2h	▓	210	D2h	Ê	242	F2h	‗
147	93h	ô	179	B3h	│	211	D3h	Ë	243	F3h	¾
148	94h	ö	180	B4h	┤	212	D4h	È	244	F4h	¶
149	95h	ò	181	B5h	Á	213	D5h	ı	245	F5h	§
150	96h	û	182	B6h	Â	214	D6h	Í	246	F6h	÷
151	97h	ù	183	B7h	À	215	D7h	Î	247	F7h	¸
152	98h	ÿ	184	B8h	©	216	D8h	Ï	248	F8h	°
153	99h	Ö	185	B9h	╣	217	D9h	┘	249	F9h	¨
154	9Ah	Ü	186	BAh	║	218	DAh	┌	250	FAh	·
155	9Bh	ø	187	BBh	╗	219	DBh	█	251	FBh	¹
156	9Ch	£	188	BCh	╝	220	DCh	▄	252	FCh	³
157	9Dh	Ø	189	BDh	¢	221	DDh	¦	253	FDh	²
158	9Eh	×	190	BEh	¥	222	DEh	Ì	254	FEh	■
159	9Fh	ƒ	191	BFh	┐	223	DFh	▀	255	FFh	

Unicode encoding

- Industrial standard that includes the alphabets for any existing writing system
  - It contains more 110,000 characters
  - It includes more than 100 sets of symbols
- Several implementations exist
  - UCS (Universal Character Set)
  - UTF (Unicode Tranformation Format)
    - UTF-8, groups of 8 bits size (1, 2, 3 or 4 groups)
      - ASCII coded in the first 8 bits
    - UTF-16, groups of 16 bits size (1 or 2 groups)
    - UTF-32, groups of 32 bits size (fixed length)

Textual and binary files

- A file is basically a sequence of bytes written one after the other
  - Each byte includes 8 bits, with possible values 0 or 1
  - As a consequence all files are binary
- Normally we can distinguish between
  - Textual files (or ASCII)
  - Binary files

C sources, C++, Java, Perl, etc.

Executables, Word, Excel, etc.

Remark:
The UNIX/Linux kernel does not distinguish between binary and textual files

Textual files (or ASCII)

- Files consisting of data encoded in ASCII
  - Sequence of 0 and 1, which (in groups of 8 bit) codify ASCII symbols

- Textual files are usually "line-oriented"
  - Newline: go to the next line
    - UNIX/Linux and Mac OSX
      - Newline = 1 character
      - Line Feed (go to next line, LF, 10_{10})
    - Windows
      - Newline = 2 characters
      - Line Feed (go to next line, LF, 10_{10})
        + Carriage Return (go to beginning of the line, CR, 13_{10})

Binary Files

- A sequence of 0 and 1, not “byte-oriented”
- The smallest unit that can be read/write is the bit
  - Non easy the management of the single bit
  - They usually include every possible sequence of 8 bits, which do not necessarily correspond to printable characters, new-line, etc.

Why are binary files used?

- Advantages
  - Compactness (smaller average dimension)
    - Examples: Number 100000_{10} occupies 6 characters, (i.e., 6 bytes) in the Text/ASCII format, and 4 bytes if coded in an integer (short)
  - Ease of editing the file
    - An integer always occupies the same space
  - Ease of positioning on the file
    - Fixed record structure
- Drawbacks
  - Limited portability
  - Impossibility to use a standard editor

Example

“ciao”

‘c’ ‘i’ ‘a’ ‘o’

99_{10} 105_{10} 97_{10} 111_{10}

01100011_{2} 01101001_{2} 01100100_{2} 01101111_{2}

String
Textual or binary file

“231”

‘2’ ‘3’ ‘1’

50_{10} 51_{10} 49_{10}

00110010_{2} 00110011_{2} 00110001_{2}

Integer number
Textual file

“231”

“231_{10}”

11100111_{2}

Integer number
Binary file

Serialization

- Process of translating a structure (e.g., C struct) into a storable format
  - Using serialization, a struct can be stored or transmitted (on the network) as a single entity
  - When the sequence of bits is read, it is done in accordance with the serialization process, and the struct is reconstructed in an identical manner
- Many languages support serialization using R/W operations on a file
  - Java, Python, Objective-C, Ruby, etc.

Example

```
struct mys {
  int id;
  long int rn;
  char n[L], c[L];
  int mark;
} s;
```

Text:
Single fields
Characters on 8 bits (ASCII)

```
1 100000 Romano Antonio 25

† Romano ÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌ
Antonio ÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌ

??? †?R?o?m?a?n?o???ÌÌÌÌÌÌÌÌÌÌÌÌÌ
ÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌ
ÌÌA?n?t?o?n?i?o???ÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌ
ÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌ
```

Binary:
Serialization
Ctr on 8 bits (ASCII)

Binary:
Serialization
Ctr on 16 bits (UNICODE)
N.B. File dimension

ISO C Standard Library

- I/O operations with ANSI C can be performed through different categories of functions
  - Character by character
  - Row by row
  - Formatted I/O
  - Binary I/O
  - Read examples
    - https://www.skenz.it/cs/c_language/file_reading_1
  - Write examples
    - https://www.skenz.it/cs/c_language/file_writing_1
  - Binary I/O examples
    - https://www.skenz.it/cs/c_language/write_and_read_a_binary_file

ISO C Standard Library

- Standard I/O is “fully buffered”
  - The I/O operation is performed only when the I/O buffer is full
  - The “flush” operation indicates the actual write of the buffer to the I/O

```
#include <stdio.h>

void setbuf (FILE *fp, char *buf);

int fflush (FILE *fp);
```

Standard error is never buffered

For concurrent processes, use:
setbuf (stdout, 0);
fflush (stdout);

Open and close a file

```
#include <stdio.h>

FILE *fopen (char *path, char *type);

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

- Access methods
  - r, rb, w, wb, a, ab r+, r+b, etc.
  - The UNIX kernel does not make any difference between textual files (ASCII) and binary files
    - The “b” option has no effect, e.g. “r”==“rb”, “w”==“wb”, etc.

I/O character by character

```
#include <stdio.h>

int getc (FILE *fp);
int fgetc (FILE *fp);

int putc (int c, FILE *fp);
int fputc (int c, FILE *fp);
```

- Returned values
  - A character on success
  - EOF on error, or when the end of the file is reached
- The function
  - **getchar** is equivalent to **getc (stdin)**
  - **putchar** is equivalent to **putc (c, stdout)**

I/O row by row

```
#include <stdio.h>

char *gets (char *buf);
char *fgets (char *buf, int n, FILE *fp);

int puts (char *buf);
int fputs (char *buf, FILE *fp);
```

- Returned values
  - buf (gets/fgets), or a non-negative value (puts/fputs) in the case of success
  - NULL (gets/fgets), or EOF for errors or when the end of file is reached (puts/fputs)
- Lines must be delimited by "new-line"

Formatted I/O

```
#include <stdio.h>

int scanf (char format, …);
int fscanf (FILE *fp, char format, …);

int printf (char format, …);
int fprintf (FILE *fp, char format, …);
```

- High flexibility in data manipulation
  - Formats (characters, integers, reals, etc.)
  - Conversions

Binary I/O

```
#include <stdio.h>

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

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

- Each I/O operation (single) operates on an aggregate object of specific size
  - With getc/putc it would be necessary to iterate on all the fields of the struct
  - With gets/puts it is not possible, because both would terminate on NULL bytes or new-lines

Binary I/O

```
#include <stdio.h>

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

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

- Returned values
  - Number of objects written/read
  - If the returned value does not correspond to the parameter nObj
    - An error has occurred
    - The end of file has been reached

ferror and feof can be used to distinguish between the two cases

Binary I/O

```
#include <stdio.h>

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

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

- Often used to manage binary files
  - serialized R/W (single operation for the whole struct)
  - Potential problems in managing different architectures
    - Data format compatibility (e.g., integers, reals, etc.)
    - Different offsets for the fields of the struct

POSIX Standard Library

- I/O in UNIX can be entirely performed with only **5** functions
  - open, read, write, lseek, close
- This type of access
  - Is part of POSIX and of the Single UNIX Specification, but not of ISO C
  - It is normally defined with the term "unbuffered I/O", in the sense that each read or write operation corresponds to a system call

System call open()

- In the UNIX kernel a "file descriptor" is a non-negative integer
- Conventionally (also for shells)
  - Standard input
    - 0 = STDIN_FILENO
  - Standard output
    - 1 = STDOUT_FILENO
  - Standard error
    - 2 = STDERR_FILENO

These descriptors are defined in the headers file **unistd.h**

System call open()

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open (const char *path, int flags);

int open (const char *path, int flags,
  mode_t mode);
```

- It opens a file defining the permissions
- Returned values
  - The descriptor of the file on success
  - -1 on error

System call open()

```
int open (
   const char *path,
   int flags,
   mode_t mode
);
```

- It can have 2 or 3 parameters
  - The **mode** parameter is optional
- **Path** indicates the file to open
- **Flags** has multiple options
  - Can be obtained with the OR bit-by-bit of constants defined in the header file **fcntl.h**
  - One of the following three constants is mandatory

O_RDONLY	open for read-only access
O_WRONLY	open for write-only access
O_RDWR	open for read-write access

System call open()

```
int open (
   const char *path,
   int flags,
   mode_t mode
);
```

- Optional constants

O_CREAT	creates the files if not exist
O_EXCL	error if O_CREAT is set and the file exists
O_TRUNC	remove the content of the file
O_APPEND	append to the file
O_SYNC	each write waits that the physical write operation is finished before continuing
…	

System call open()

```
int open (
   const char *path,
   int flags,
   mode_t mode
);
```

- **Mode** specifies access permissions

S_I[RWX]USR	rwx --- ---
S_I[RWX]GRP	--- rwx ---
S_I[RWX]OTH	--- --- rwx

When a file is created, actual permissions are obtained from the **umask** of the user owner of the **process**

System call read()

```
#include <unistd.h>

int read (int fd, void *buf, size_t nbytes);
```

- Read from file **fd** a number of bytes equal to **nbytes**, storing them in **buf**
- Returned values
  - number of read bytes on success
  - -1 on error
  - 0 in the case of EOF

System call read()

```
#include <unistd.h>

int read (int fd, void *buf, size_t nbytes);
```

- The returned value is lower that **nbytes**
  - If the end of the file is reached before **nbytes** bytes have been read
  - If the **pipe** you are reading from does not contain **nbytes** bytes

System call write()

```
#include <unistd.h>

int write (int fd, void *buf, size_t nbytes);
```

- Write **nbytes** bytes from **buf** in the file identified by descriptor **fd**
- Returned values
  - The number of written bytes in the case of success, i.e., normally **nbytes**
  - -1 on error

System call write()

```
#include <unistd.h>

int write (int fd, void *buf, size_t nbytes);
```

- Remark
  - write writes on the system buffer, not on the disk
    - fd = open (file, O_WRONLY | O_SYNC);
  - O_SYNC forces the sync of the buffers, but only for ext2 file systems

Examples: File R/W

```
float data[10];
if ( write(fd, data, 10*sizeof(float))==(-1) ) {
   fprintf (stderr, "Error: Write %d).\n", n);
  }
}
```

writing of the vector data (of float)

```
struct {
  char name[L];
  int n;
  float avg;
} item;
if ( write(fd,&item,sizeof(item)))==(-1) ) {
   fprintf (stderr, "Error: Write %d).\n", n);
  }
}
```

Writing of the serialized struct item (with 3 fields)

System call lseek()

```
#include <unistd.h>

off_t lseek (int fd, off_t offset, int whence);
```

- The current position of the file offset is associated to each file
  - The system call lseek assigns the value **offset** to the file offset
  - The **offset** value is expressed in bytes

System call lseek()

```
#include <unistd.h>

off_t lseek (int fd, off_t offset, int whence);
```

- whence specifies the interpretation of offset
  - If whence==SEEK_SET
    - The offset is evaluated from the beginning of the file
  - If whence==SEEK_CUR
    - The offset is evaluated from the current position
  - If whence==SEEK_END
    - The offset is evaluated from the end of the file

The value of **offset** can be positive or negative

It is possible to leave "holes" in a file (filled with zeros)

System call lseek()

```
#include <unistd.h>

off_t lseek (int fd, off_t offset, int whence);
```

- Returned values
  - new offset on success
  - -1 on error

System call close()

```
#include <unistd.h>

int close (int fd);
```

- Returned values
  - 0 on success
  - -1 on error
- All the open files are closed automatically when the process terminates

Example: File R/W

```
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#define BUFFSIZE 4096

int main(void) {
  int nR, nW, fdR, fdW;
  char buf[BUFFSIZE];
  fdR = open (argv[1], O_RDONLY);
  fdW = open (argv[2], O_WRONLY | O_CREAT | O_TRUNC,
                       S_IRUSR | S_IWUSR);
  if ( fdR==(-1) || fdW==(-1) ) {
    fprintf (stdout, “Error Opening a File.\n“);
    exit (1);
  }
```

Example : File R/W

```
while ( (nR = read (fdR, buf, BUFFSIZE)) > 0 ) {
    nW = write (fdW, buf, nR);
    if ( nR!=nW )
      fprintf (stderr,
        "Error: Read %d, Write %d).\n", nR, nW);
  }

  if ( nR < 0 )
    fprintf (stderr, "Write Error.\n");

  close (fdR);
  close (fdW);

  exit(0);
}
```

Error check on the last reading operation

This program works indifferently on text and binary files