



# Programmazione di Sistema e di Rete

Matteo Aprile  
Professore: Franco Tommasi

## INDICE

<b>I</b>	<b>Libri di testo consigliati</b>	1
<b>II</b>	<b>Comandi utili</b>	2
II-A	find: trovare tutti i file eseguibili . . . . .	2
II-B	find: trovare file di intestazione del mac come stdio.h . . . . .	2
II-C	lld: per capire che librerie usa il codice	2
II-D	gcc: per vedere tutta la gerarchia di file in una libreria . . . . .	2
II-E	gcc: per vedere il codice con tutti i file importati . . . . .	2
II-F	gcc -g: debugging debole . . . . .	2
II-G	gcc -ggb: debugging forte . . . . .	2
II-H	xattr: usato per i file che entrano in quarantena su MacOS . . . . .	2
<b>III</b>	<b>Variabili di sistema definite in .bashrc</b>	2
III-A	INC . . . . .	2
<b>IV</b>	<b>Introduzione - 23/27.09.22</b>	2
IV-A	System call . . . . .	2
IV-B	Programma Make - 30.09.22 . . . . .	2
IV-C	Direttive di preprocessore - 28.09.22 . . . . .	3
IV-D	Librerie . . . . .	3
IV-E	Creazione librerie . . . . .	4
IV-F	Aggiornamento librerie . . . . .	4
<b>V</b>	<b>System call</b>	5
V-A	Funzioni e system call . . . . .	5
V-B	Capire se una funzione è una system call	6

## I. LIBRI DI TESTO CONSIGLIATI

- Advanced Programming in the Unix Environment, 3th ed, Stevens, Rago
- TCP/IP 1, Stevens (facoltativo)
- Unix Networking Programming the Socket Networking API, Stevens
- The Linux Programing Interface, Kerrisk
- maset
- Gail Guida alla Programmazione in Linux, Simone Piccardi

## II. COMANDI UTILI

A. *find*: trovare tutti i file eseguibili

```
1 $ find . -type f -perm -0100
2 ./standards/makeopt.awk
3 ./standards/makeconf.awk
4 ./proc/awkexample
5 ./systype.sh
6 ./advio/fixup.awk
```

B. *find*: trovare file di intestazione del mac come *stdio.h*

```
1 find /Applications/Xcode.app/ -name stdio.h 2>/dev/
  null
```

C. *ldd*: per capire che librerie usa il codice

```
1 ldd [nomevoci]
```

D. *gcc*: per vedere tutta la gerarchia di file in una libreria

```
1 gcc -H lib.a
```

E. *gcc*: per vedere il codice con tutti i file importati

```
1 gcc -E file.c
```

F. *gcc -g*: debugging debole

```
1 gcc -g -ansi -I../include -Wall -DMACOS -
  D_DARWIN_C_SOURCE ls1.c -o ls1 -L../lib -lapue
```

G. *gcc -g*: debugging forte

```
1 gcc -g -ansi -I../include -Wall -DMACOS -
  D_DARWIN_C_SOURCE ls1.c -o ls1 -L../lib -lapue
```

H. *xattr*: usato per i file che entrano in quarantena su MacOS

```
1 xattr -d (delete) com.apple.quarantine [path sh]
```

## III. VARIABILI DI SISTEMA DEFINITE IN .BASHRC

A. *INC*

```
1 INC="/Applications/Xcode.app/Contents/Developer/
  Platforms/MacOSX.platform/Developer/SDKs/MacOSX.
  sdk/usr/include/"
```

## IV. INTRODUZIONE - 23/27.09.22

A. *System call*

Sono uguali alle funzioni di libreria dal punto di vista sintattico, però cambia il modo di compilarle. Notare che non possono essere usati i nomi delle SC per delle function call.

Per poi poter "raccontare" tra umani le sequenze di bit che vengono mandate ai processori si usa **assembly**.

Sono effettivamente delle chiamate a funzioni ma poi dal codice assembly puoi capire che è una system call dato che ha dei meccanismi specifici.

Alcuni esempi di chiamate e registri:

- **eax** : registro dove metti il **numero della sc**
- **int 0x80**: avvisa il **kernel** che serve chiamare una sc
- **exit()**: chiudere un processo
- **write()**:

```
1 mov edx,4      ; lunghezza messaggio
2 mov ecx,msg    ; puntatore al messaggio
3 mov ebx,1      ; file descriptor
4 mov eax,4      ; numero della sc
5 int 0x80
```

dove nel **file descriptor** indichi a quale file devi mandare l'output. Questo viene usato dato che così non deve cercare il path ogni volta ma lo mantiene aperto riferendosi ad esso tramite il numero.

B. *Programma Make* - 30.09.22

Quando viene avviato verifica la presenza di un file chiamato "Makefile", oppure si usa 'make -f'. In questo file ci sono le **regole di cosa fare per automatizzare delle azioni per un numero n di file**. Se, durante la compilazione di massa, **una di queste da un errore il programma make si interrompe**, per evitare ciò si usa '-i' (ignore).

Il Makefile andrà ad aggiornare una libreria andando a guardare se una delle 3 date di ultima modifica si sono aggiornate.

Andiamo a guardare **cosa contiene Makefile**:

```
1 DIRS = lib intro sockets advio daemons datafiles db
  environ \
2 fileio filedir ipc1 ipc2 proc pty relation
  signals standards \
3 stdio termios threadctl threads printer
  exercises
4
5 all:
6   for i in $(DIRS); do \
7     (cd $$i && echo "making $$i" && $(MAKE) ) ||
8     exit 1; \
9   done
10 clean:
11   for i in $(DIRS); do \
12     (cd $$i && echo "cleaning $$i" && $(MAKE)
13     clean) || exit 1; \
14   done
```

dove:

- **DIRS**: lo si associa alle **stringhe singole** che gli sono state associate
- **all**: nel ciclo for:
  - manda un comando in subshell

- \$\$i: riferimento alla variabile "i" del for + simbolo escape per il Makefile
- \$(MAKE): macro predefinita per i Makefile

La struttura è:

```
1 target: prerequisiti
2 rule
```

dove:

- **target**: è la cosa che si vuole fare, se essendo il primo target, sarà anche quello di default
- **prerequisiti**: file e/o target a loro volta
- **rule**: indica cosa può fare il target

Può capitare che prima di eseguire il Makefile ci sia uno script "configure".

In molti casi si ha un target "clean" che permette di pulire i file .o che sono inutili dopo la compilazione, o comunque qualsiasi tipo di file gli si voglia far eliminare. Questo tipo di target che non rappresentano un file, sono detti "phony" perchè fasulli, dato che non sono file ma sole parole

```
1 file: file.o lib.o
2
3 clean:
4 rm file.o
```

Abbiamo delle variabili automatiche per rendere il lavoro più facile:

- \$@: per riferirsi il target
- \$?: tutti i prerequisiti più recenti del target
- \$^: tutti i prerequisiti del target
- <https://www.gnu.org/software/make/manual/make.html#Automatic-Variables>

Un altro esempio di Makefile è:

```
1 ROOT=..
2 PLATFORM=$(shell $(ROOT)/systype.sh)
3 include $(ROOT)/Make.defines.$(PLATFORM)
4
5 PROGS = getcputc hello ls1 mycat shell1 shell2
6         testerror uidgid
7
8 all:      $(PROGS)
9
10 %:  %.c $(LIBAPUE)
11     $(CC) $(CFLAGS) $@.c -o $@ $(LDFLAGS) $(LDLIBS)
12
13 clean:
14     rm -f $(PROGS) $(TEMPFILES) *.o
15
16 include $(ROOT)/Make.libapue.inc
```

dove:

- ROOT: cwd
- PLATFORM: assumerà in valore del OS: macos/linux
- include: include un file
- PROGS: elenco dei programmi da usare
- %: target con nome variabile, indica un file
- %.c: target con nome variabile ma estensione .c
- \$(CC): indica il compilatore dove cc è un link simbolico a clang
- \$(CFLAGS) indica una macro predefinita dei default vuota che si può usare all'occorrenza

- all: target che prende in carico tutti i programmi che se saranno di tipo .c saranno presi in carico dal target successivo

Per la compilazione dei file, qualsiasi sia il linguaggio, make saprà come compilarlo grazie a tutte le definizioni di default presenti in:

```
1 make -p
```

notare che si può mettere un comando custom nelle rule del target

Nell'eventualità di voler aggiornare un solo file della libreria senza far aggiornare il resto ci basterà usare uno script che compila quel file passato a linea di comando.

```
1 gcc -ansi -I../include -Wall -DMACOS -
  D_DARWIN_C_SOURCE ${1}.c -o ${1} -L../lib -
  lapue
```

### C. Direttive di preprocessore - 28.09.22

Sono delle indicazioni date a gcc prima di iniziare la compilazione.

Iniziano tutte con '#':

- **#include**: serve ad includere delle librerie di sistema (<lib.h>) oppure di librerie fatte da noi e non in directory standard ("lib.h")
- **#define**:
  - permette di creare delle "macro", che vanno a sostituire una stringa con un'altra (es: #define BUFLLEN), può capitare che debbano essere definite delle macro prima che si compili il programma, in questi casi si usa scrivere es: '-DMACOS'
  - permette di creare delle "function like macro" (es: #define ABSOLUTE\_VALUE(x) (((x<0)?-(x):(x)))
- **#ifdef**, **#ifndef**, **#endif**: usata per far accadere qualcosa nel caso un macro sia stata definita

```
1 #ifdef VAR
2 print("hello");
3 #endif
```

Per evitare che più file includano lo stesso si usano degli #ifndef in tutto il codice, in modo da evitare doppie definizioni.

### D. Librerie

Durante la fase di compilazione creiamo dei file oggetto (.o) per ogni file in cui è scritta la descrizione delle funzioni di libreria (.c)

```
1 gcc -c bill.c
```

Si andrà poi a creare il prototipo della funzione (.h).

In fine tramite il linker si andranno ad unire tutti i file per crearne uno unico con tutte le definizioni delle funzioni incluse nelle librerie, di sistema e non, importate. Si vanno quindi a sciogliere tutti i riferimenti incrociati.

```
1 gcc -o program program.o bill.o
```

Per quanto riguarda le **funzioni di sistema** NON abbiamo il file sorgente ma abbiamo direttamente l'eseguibile. In compenso abbiamo un **file di libreria**, cioè un insieme di file oggetto linkati in un unico file, dove c'è il codice oggetto di tutte le funzioni.

Abbiamo **2 tipi di librerie**:

- **statiche**: è una **collezione di file oggetto** che hanno il codice compilato delle funzioni e che verranno **linkati al momento della compilazione**. Il programma che si crea sarà possibile essere eseguito solo sullo stesso OS.

Il **problema si ha nell'aggiornamento delle librerie al momento della scoperta di un bug**. Una volta corretto servirà ricevere la versione corretta per poter aggiornare il programma.

- **dynamic**: ricordano il concetto di plug-in, quindi **viene invocato a runtime e caricato in memoria** (es: aggiornamenti dei OS). **L'eseguibile non viene toccato la correzione avviene solo nella libreria**.

Il requisito maggiore è che chi si passa il codice debba avere lo stesso OS dell'altro utente. Notare che **non cambia il prototipo** dato che sennò bisognerà ricompilare l'intero programma.

In generale le **librerie statiche sono molto pericolose** infatti alcuni OS le aboliscono **per le questioni di sistema**. Su linux si ha come libreria statica 'lib.c' che è la libreria con le funzioni più usate in c. Per macos è stata abolita.

Per compilare con la versione dinamica non servono opzioni, per la statica si usa:

```
gcc -static
```

#### E. Creazione librerie

Per costruire una **libreria statica per MacOS**:

- 1) costruiamo il **file oggetto**:

```
gcc -c libprova.c
```

- 2) costruiamo la **libreria** (con ar=archive, c=create se lib.a non esiste):

```
ar rcs libprova.a libprova.o
```

- 3) costruire il **codice** che usa la libreria (con -Wall=verbose warning, -g=debugging, -c=create del file):

```
gcc -Wall -g -c useprova.c
```

- 4) **linker** per risolve le chiamate incrociate (con -L.=dove prendere la libreria, -l[nomelib]=usare la libreria):

```
gcc -g -o useprova useprova.o -L. -lprova
```

Per capire che librerie usa il codice si usa:

```
otool -L [nomecodice]
```

Per costruire una **libreria statica per Linux**:

- 1) costruiamo il **file oggetto**:

```
gcc -fPIC -Wall -g -c libprova.c
```

- 2) costruiamo la **libreria** (con 0.0=versione della libreria):

```
gcc -g -shared -Wl,-soname,libprova.so.0 -o  
libprova.so.0.0 libprova.o -lc
```

- 3) costruire il **link simbolico per aggiornare le librerie** senza aggiornare gli eseguibili e senza cambiare il nome del programma:

```
ln -sf libprova.so.0.0 libprova.so.0
```

- 4) **linker** per risolve le chiamate:

```
ln -sf libprova.so.0 libprova.so
```

Per capire che librerie usa il codice si usa:

```
ldd [nomecodice]
```

#### F. Aggiornamento librerie

Su **Linux** il sistema **andra' a prendere direttamente una libreria dinamica**, per evitare ciò e far trovare la nostra, basterà **impostare una variabile di ambiente**:

```
LD_LIBRARY_PATH='pwd' ldd useprova
```

Tipicamente la libreria viene distribuita nelle directory di sistema andandola ad "installare".

Su **MacOS** la libreria dinamica è un **dylib**:

```
gcc -dynamiclib libprova.c -o libprova.dylib
```

Quindi eseguendo il programma **trovera' la libreria controllando nella directory corrente** e quindi non serve creare la variabile di ambiente come su Linux.

i file di intestazione del mac come stdio.h per cercarla uso:

```
find /Applications/Xcode.app/ -name stdio.h 2>/dev/  
null
```

## V. SYSTEM CALL

## A. Funzioni e system call

Se prendiamo un funzionamento più semplice del comando "ls" potrebbe essere:

```
1 #include "apue.h"
2 #include <dirent.h>
3
4 int
5 main(int argc, char *argv[])
6 {
7     DIR          *dp;
8     struct dirent *dirp;
9
10    if (argc != 2)
11        err_quit("usage: lsl directory_name");
12
13    if ((dp = opendir(argv[1])) == NULL)
14        err_sys("can't open %s", argv[1]);
15    while ((dirp = readdir(dp)) != NULL)
16        printf("%s\n", dirp->d_name);
17
18    closedir(dp);
19    exit(0);
20 }
```

dove abbiamo che:

- **DIR**: struttura dati
  - **struct dirent**: **tipo struttura** che contiene al suo interno diversi tipi di variabili.
- Per capire se è una funzione di sistema lanciamo:

```
1 grep -rw "struct dirent" $INC
```

seguiamo il percorso:

```
1 /Applications/Xcode.app/Contents/Developer/
  Platforms/MacOSX.platform/Developer/SDKs/
  MacOSX.sdk/usr/include//sys/dirent.h:struct
  dirent {
```

```
1 #ifndef _SYS_DIRENT_H
2 #define _SYS_DIRENT_H
3
4 #include <sys/_types.h>
5 #include <sys/cdefs.h>
6
7 #include <sys/_types/_ino_t.h>
8
9
10 #define __DARWIN_MAXNAMLEN      255
11
12 #pragma pack(4)
13
14 #if !__DARWIN_64_BIT_INO_T
15 struct dirent {
16     ino_t d_ino;                /* file
17     number of entry */
18     __uint16_t d_reclen;        /* length of
19     this record */
20     __uint8_t d_type;           /* file type
21     , see below */
22     __uint8_t d_namlen;         /* length of
23     string in d_name */
24     char d_name[__DARWIN_MAXNAMLEN + 1]; /*
25     name must be no longer than this */
26 };
27 #endif /* !__DARWIN_64_BIT_INO_T */
28
29 #pragma pack()
30
31 #define __DARWIN_MAXPATHLEN      1024
```

```
28 #define __DARWIN_STRUCT_DIRENTRY { \
29     __uint64_t d_ino;          /* file number of
30     entry */ \
31     __uint64_t d_seekoff;      /* seek offset (
32     optional, used by servers) */ \
33     __uint16_t d_reclen;       /* length of this
34     record */ \
35     __uint16_t d_namlen;       /* length of string
36     in d_name */ \
37     __uint8_t d_type;          /* file type, see
38     below */ \
39     char d_name[__DARWIN_MAXPATHLEN]; /*
40     entry name (up to MAXPATHLEN bytes) */ \
41 }
42
43 #if __DARWIN_64_BIT_INO_T
44 struct dirent __DARWIN_STRUCT_DIRENTRY;
45 #endif /* __DARWIN_64_BIT_INO_T */
46
47 #if !defined(_POSIX_C_SOURCE) || defined(
48     _DARWIN_C_SOURCE)
49 #define d_fileno      d_ino          /*
50     backward compatibility */
51 #define MAXNAMLEN     __DARWIN_MAXNAMLEN
52 /*
53 * File types
54 */
55 #define DT_UNKNOWN      0
56 #define DT_FIFO         1
57 #define DT_CHR          2
58 #define DT_DIR          4
59 #define DT_BLK          6
60 #define DT_REG          8
61 #define DT_LNK         10
62 #define DT_SOCK        12
63 #define DT_WHT         14
64
65 /*
66 * Convert between stat structure types and
67 * directory types.
68 */
69 #define IFTODT(mode)    (((mode) & 0170000) >>
70     12)
71 #define DTTOIF(dirtype) ((dirtype) << 12)
72 #endif
73 #endif /* _SYS_DIRENT_H */
```

dove vediamo che se la variabile "\_\_\_DARWIN\_64\_BIT\_INO\_T" è stata definita avremo che la struttura di struct dirent è:

```
1 #define __DARWIN_STRUCT_DIRENTRY { \
2     __uint64_t d_ino;          /* file number of
3     entry */ \
4     __uint64_t d_seekoff;      /* seek offset (
5     optional, used by servers) */ \
6     __uint16_t d_reclen;       /* length of this
7     record */ \
8     __uint16_t d_namlen;       /* length of string
9     in d_name */ \
10    __uint8_t d_type;          /* file type, see
11    below */ \
12    char d_name[__DARWIN_MAXPATHLEN]; /*
13    entry name (up to MAXPATHLEN bytes) */ \
14 }
```

- **if**: esegue un controllo sugli args. Notiamo che "err\_quit" non è una funzione di sistema da:

```
1 grep -rw "err_quit" $INC
```

infatti non restituisce nulla. Deve allora essere una funzione di libreria creata da noi quindi non presente nella directory standard.

La funzione andrà a dare un messaggio di errore e poi esce dal programma.

- **opendir**: serve ad aprire una directory andandola a caricare nella RAM.
- **while**: leggiamo la directory e la inseriamo nella struttura che poi sarà richiamata tramite:

```
1 dirp->d_name
```

dove "d\_name" è il nome dello slot in cui è contenuto il nome del file.

- **exit**: restituisce l'exit code del programma

### B. Capire se una funzione è una system call

Andiamo a **vedere se è una funzione o una system call tramite "man"**, lo si capisce tramite la dicitura in alto alla pagina del manuale:

- **Library Functions Manual**
- **System Calls Manual**

Abbiamo anche **esempi più particolari**, come fork, dove è indicata come system call ma in realtà le richiama ma in prima persona.

Potremo trovare i simboli di una libreria tramite:

```
1 nm lib.a
```

che ci fa vedere, per ogni file oggetto, i simboli associati per ogni funzione.

Le system call le troveremo in "\$INC/sys/syscall.h"

file descriptor: per accedere a d un file si fa tramite il path che visita e ricerca in tutte le cartelle del path che rimanderanno ad un certo blocco. Per non fare tutto ciò ogni volta ci si conserva un identificativo per quel file. si dà sempre il più piccolo numero intero disponibile. viene conservato fino alla chiusura del file.

ogni processo ha, se lanciato da bash, 3 file desc usati 0, 1, 2.

```
1 #include "apue.h"
2
3 #define BUFSIZE 4096
4
5 int
6 main(void)
7 {
8     int    n;
9     char   buf[BUFSIZE];
10
11     while ((n = read(STDIN_FILENO, buf, BUFSIZE)) >
12            0)
13         if (write(STDOUT_FILENO, buf, n) != n)
14             err_sys("write error");
15
16     if (n < 0)
17         err_sys("read error");
18
19     exit(0);
20 }
```

questo è un esempio semplificato del comando "cat"

- **BUFSIZE**: macro di preprocessore
- **read**: system call con parametri: **STDIN\_FILENO**: file descriptor per dire da dove si vuole leggere, **buf**: indirizzo di dove sono i dati dell'array, **BUFSIZE**: quando deve leggere. n sarà il numero di char che ha letto la read dato che potrebbe leggere meno byte letti di quelli richiesti se il file ne contiene di meno. read si ricorda una "posizione nel file" che gli permette di non leggere sempre i primi n byte ma di ricominciare da dove ha lasciato
- **write**: richiede gli stessi valori di read tranne per **STDOUT\_FILENO** e ritorna il numero byte effettivamente letti

per capire quando è **STDIN\_FILENO**:

```
1 $ grep -rw "STDIN_FILENO" $INC
2 /Applications/Xcode.app/Contents/Developer/Platforms
  /MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/
  include/unistd.h:#define STDIN_FILENO 0
  /* standard input file descriptor */
3 /Applications/Xcode.app/Contents/Developer/Platforms
  /MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/
  include/asl.h: * asl_log_descriptor(c, m,
  ASL_LEVEL_NOTICE, STDIN_FILENO,
  ASL_LOG_DESCRIPTOR_READ);
```

abbiamo che un processo per eseguire un programma esegue prima una fork e poi con exec esegue il programma prima di fare la fork il child chiude il file 1 e quando fa una open il file prenderà la 1 che sarà presa quindi dal file nel quale stiamo indirizzando. questo è il sistema sul quale gioca tutto UNIX infatti avviene anche con le pipe "—". questo quindi permette di creare programmi complessi unendo tanti piccoli programmi specializzati in un'unica funzione ognuno fatto da diversi

i child ereditano i file descriptor dei parent quindi non è necessario che il programma corrente faccia una open dei file descriptor

open va quindi a restituire il numero di riferimento, dopo aver preso il path ed averlo associato al minor numero di disponibile.

notare che si usa ./mycat & outputfile.txt che aspetterà allo standard di input da scrivere dato che **STDIN\_FILENO** è stato già aperto.

in pratica noi andiamo a **REINDIRIZZARE** lo standard output in un file creato da noi ma in realtà stiamo scrivendo nel file già aperto: 1

un file è una serie di meccanismi: apri, leggi, scrivi, chiudi. il che è applicabile a file, cartelle, stampanti ecc solo che per ogni "tipo" i 4 meccanismi si adeguano a ciò che il caso particolare deve fare.

siamo onel capitolo unbuffered i/o.

le system call rappresentano una barriera tra kernel e programmi. quindi il sistema entra in una modalità in cui tutto è possibile detta kernel mode.

stdliob è una libreria che ha il meccanismo di avere delle prestazioni migliori se ciò che deve essere scritto lo consegna tutto insieme e con qualche info aggiuntiva per facilitare il lavoro. ci sarà allora una piccola riflessione prima di mandare roba al kernel, quindi si va a prendere pianopiano i byte fino ad

un certo tetto. quindi ha un buffered i/o che permette di avere dei buffer che una volta riempiono sono mandati al kernel.

img

system call getpid() usata per prendere i process id dell'attuale processo.

prendiamo shell1.c

```
1 #include "apue.h"
2 #include <sys/wait.h>
3
4 int
5 main(void)
6 {
7     char    buf[MAXLINE];    /* from apue.h */
8     pid_t   pid;
9     int     status;
10
11     printf("%s "); /* print prompt (printf requires
12                    %% to print %) */
13     while (fgets(buf, MAXLINE, stdin) != NULL) {
14         if (buf[strlen(buf) - 1] == '\n')
15             buf[strlen(buf) - 1] = 0; /* replace
16             newline with null */
17
18         if ((pid = fork()) < 0) {
19             err_sys("fork error");
20         } else if (pid == 0) { /* child */
21             execlp(buf, buf, (char *)0);
22             err_ret("couldn't execute: %s", buf);
23             exit(127);
24         }
25
26         /* parent */
27         if ((pid = waitpid(pid, &status, 0)) < 0)
28             err_sys("waitpid error");
29         printf("%s ");
30     }
31     exit(0);
32 }
```

crea uno shell dal quale eseguire programmi.

- fgets: funzione della stdio che legge la riga che dai prima di dare invio e la mette in un buffer. restituisce una stringa con un carattere alla fine
- MAXLINE: proviene da una nostra libreria

```
1 $ grep -rw "MAXLINE" include/
2 Binary file include/apue.h.gch matches
3 include/apue.h:#define MAXLINE 4096
   /* max line length */
```

- stdin: presente in stdio ed è una struttura file che definisce uno standard input tramite un puntatore di un "file"
- if 1: permette di avere un null dove prima avevamo