Ridirezione dell'output

I comandi visti finora producono output sul terminale. P. es.

```
/home/user1 $ ls -C affari
domande  lettere  prevent
```

L'output di un comando si può, anziché inviare sullo schermo, ***ridirigere*** sul file `f`, facendo seguire il comando da: `>f`. P. es.:

```
/home/user1 $ ls f
ls: File or directory "f" is not found
/home/user1 $ ls -C affari >f
/home/user1 $ ls f
f
/home/user1 $ cat f
domande  lettere  prevent
```

Viceversa l'output di `echo` non deve essere necessariamente ridiretta su un file, come visto finora:

```
/home/user1 $ echo ciao
ciao
```

Nel primo esempio `f` non esisteva, se esiste viene sovrascritto:

```
/home/user1 $ echo ciao >f
/home/user1 $ cat f
ciao
```

L'output di un comando si può invece ***appendere*** al file `f`, facendo seguire il comando da: `>>f`:

```
/home/user1 $ echo bello >>f
/home/user1 $ cat f
ciao
bello
```

Ridirezione dell'input

Il comando `wc` accetta del testo immesso dalla tastiera, finché all'inizio di una nuova riga si immette **`Ctrl D`**, allora scrive sul terminale il n. di righe, parole e caratteri nel testo:

```
/home/user1 $ wc
ciao hello
^D
      1       2      11
```

Si può anche far sì che l'input di un comando provenga, anziché dalla tastiera, da un file `f`, facendo seguire il comando da `<f`:

```
/home/user1 $ echo ciao hello >f
/home/user1 $ wc <f
      1       2      11
```

Ridirezione di input ed output si possono combinare: un comando

- può ricevere il suo input da un file `f` e
- mandare il suo output su un altro `g`

```
/home/user1 $ wc < f > g
/home/user1 $ cat g
      1       2      11
```

Standard input-output e ridirezione

Si è visto un uso di `wc` senza argomenti:

```
/home/user1 $ wc < f
      1       2       11
```

In effetti `wc` ammette come argomento un file che funge da input:

```
/home/user1 $ wc f
      1       2       11    f
```

Molti comandi si comportano come `wc`, ovvero:

- se si omette o non è previsto come argomento un file di input, l'input proviene dalla tastiera;
- l'output appare sul terminale

A livello di codice:

- l'input si legge dal descrittore di file 0 (*standard input*)
- l'output si scrive sul descrittore di file 1 (*standard output*).
- di norma standard input/output associati a tastiera/terminale.
- in presenza di ridirezione, la shell associa standard input/output a file reali opportuni.

→ Il (programmatore del) comando ignora la ridirezione, *che è supportata invece dalla shell*

Pipes

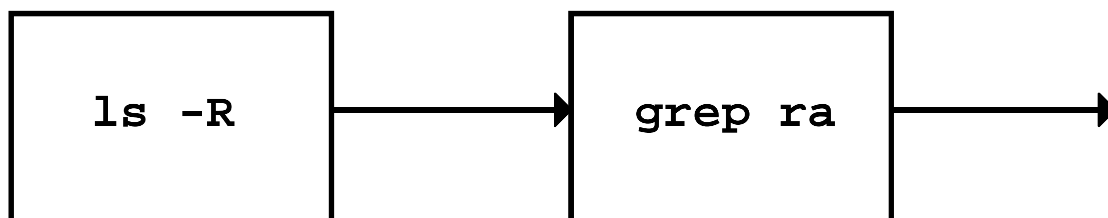
`grep` *chars* cerca nel suo standard input le righe contenenti i caratteri *chars* e le scrive sullo standard output:

```
/home/user1 $ ls -R > tmp/f
/home/user1 $ grep ra < tmp/f
rai
sfera
/home/user1 $ rm tmp/f
```

In casi simili, anziché gestire un file intermedio `tmp/f` per far comunicare due comandi, si può lasciar fare alla shell:

```
/home/user1 $ ls -R | grep ra
rai
sfera
```

Questo meccanismo e il segno | si dicono ***pipe*** (tubo), perché le ***pipeline*** di comandi operano così:



Si possono anche avere pipeline di 3 o più comandi:

```
/home/user1 $ ls -R | grep ra | wc -l
      2
```

- tutti i programmi che per default di argomenti comunicano con standard input e output si possono collegare in pipeline
- come per ridirezione, i programmi ignorano associazioni di standard input e output: a quelle necessarie provvede la shell
- in `c|d`, `c` e `d` partono insieme e sono eseguiti *in parallelo* (anche se in genere `d` aspetterà almeno un po' di input da `c`)

Pipeline e `tee`

In una pipeline l'output di un comando che non è l'ultimo non lascia traccia:

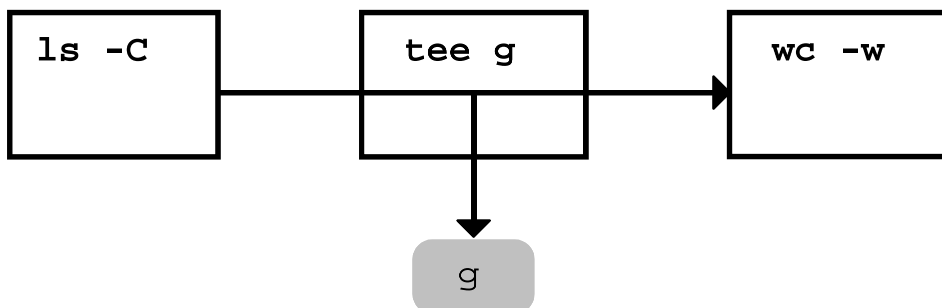
```
/home/user1 $ ls -C
affari    agenda    compiti   f       g      telefoni  tmp
/home/user1 $ ls -C | wc -w
      7
```

Si può però salvare l'output di questo comando sul file `g` interponendo `tee g` dopo il comando:

```
/home/user1 $ ls -C | tee g | wc -w
      7
/home/user1 $ cat g
affari    agenda    compiti   f       g      telefoni  tmp
```

`tee g` funziona come un tubo a T:

- copia il suo standard input sul suo standard output
- devia anche una copia sul file `g`:



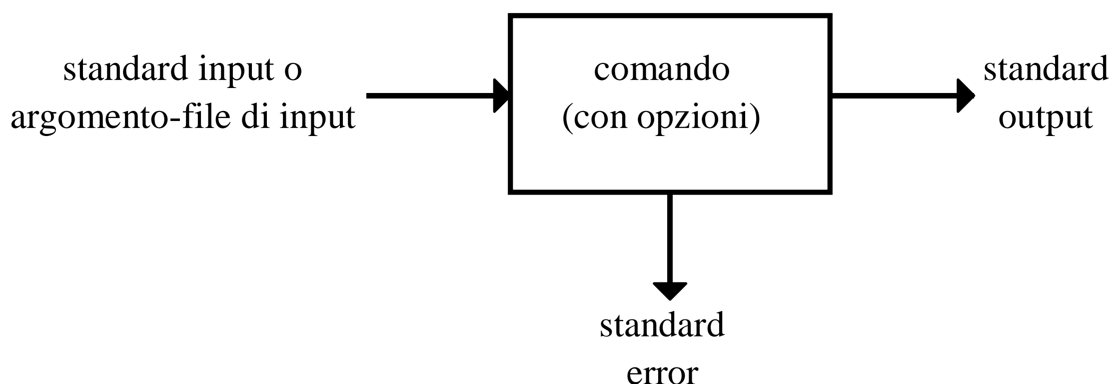
Standard error

I messaggi di errore non sono scritti dai comandi sulla standard input, ma su un (descrittore di) file detto ***standard error*** (descr. 2)

Questo evita che i messaggi di errori vengano ridiretti su un file o a un altro comando e l'utente non li veda subito. P. es.

```
/home/user1 $ ls -C
affari    agenda    compiti   f      g      telefoni  tmp
/home/user1 $ rm f g
/home/user1 $ ls f
ls: File or directory "f" is not found
/home/user1 $ ls f >h
ls: File or directory "f" is not found
/home/user1 $ cat h   NB: h vuoto perché ls f non dà output
/home/user1 $ rm h
```

Dunque il comportamento dei comandi si può raffigurare così:



Esercizio: eseguire tutti gli esempi mostrati su pipe e ridirezione.

Ridirezione in generale

n<*file* — *file* aperto in lettura con descrittore *n*

```
/home/user1 $ sort 5<f        inutile: sort non usa il descrittore 5
```

n>*file* — *file* aperto in scrittura con descrittore *n*

```
/home/user1 $ sort 8>g        inutile: sort non usa il descrittore 8
```

n<>*file* — *file* aperto in lettura e scrittura con descrittore *n*
n omesso: *file* aperto come stdin in lettura e stdout in lettura

n>>*file* — *file* aperto in modo *append*, con descrittore *n*

&>*file* — ridirige standard output ed error insieme su file

>&*file* — ridirige standard output ed error insieme su file

n<&*m* — il descrittore *n* (default 0) diventa una copia
del descrittore *m*, associato a un file aperto in lettura

```
/home/user1 $ sort 5<f 0<&5      legge da stdin
```

n>&*m* — il descrittore *n* (default 1) diventa copia del descrittore *m* →
l'output diretto al descrittore *n* andrà al file descritto da *m*

```
/home/user1 $ sort 8>g 1>&8      scrive su stdout
/home/user1 $ ls >dirlist 2>&1   2 (stderr) associato a 1
/home/user1 $ ls >&dirlist       equivalente
/home/user1 $ ls 2>&1 >dirlist   problema!
```

Nell'ultimo esempio, 2 (stderr) diventa una copia di 1 (stdout)
prima che questo sia ridiretto su `dirlist`

→ i messaggi di errore andranno sul terminale

Valutazione dell'uscita di un comando

Nella riga di comando, un comando racchiuso tra accenti ` `` `
viene sostituito con l'uscita che esso produce. P. es.

```
/home/user1 $ echo date
date
/home/user1 $ echo `date`
Wed Nov 18 02:30:21 EST 1992
```

La shell: comandi composti

I comandi possono essere *terminati* da un `;`
Ciò consente di creare comandi composti:

```
//home/user1 $ date; who
Tue Nov 17 21:55:09 EST 1992
utente1      tty1          Nov 17 20:14
utente2      tty0          Nov 17 20:13
```

Proviamo a mettere un comando composto in pipeline con `wc`:

```
/home/user1 $ date; who | wc
Tue Nov 17 22:01:57 EST 1992
      1      5       39
```

L'effetto non è quello voluto: solo `who` è andato in pipe con `wc`.
La ragione è che | *lega più forte di* `;` (come $\times$ e + in aritmetica).
Occorrono dunque delle parentesi:

```
/home/user1 $ (date; who) | wc
      2     11       68
```

Esecuzione in *background*

Normalmente, finché non termina un comando, la shell non ne considera un altro.
P.es. `sleep 10` aspetta 10 secondi prima di terminare:

```
/home/user1 $ sleep 10
/home/user1 $                compare solo dopo 10 sec
```

Per eseguire invece un comando in ***background***, si usa il terminatore `&`:

- la shell fa partire un *processo* che esegue il comando,
- viene scritto sul terminale il *numero* di questo processo,
- il prompt ritorna senza attendere che il comando sia terminato
- quando termina, il comando avverte con un messaggoio

```
/home/user1 $ sleep 10 &
3651
/home/user1 $                compare quasi subito
```

Per eseguire due comandi *in parallelo*, si termina il primo con `&`

```
/home/user1 $ (sleep 10; date) & date -u
11156
Tue Nov 17 21:19:40 GMT 1992
/home/user1 $ Tue Nov 17 22:19:50 EST 1992
```

Cioè:

`(sleep 10; date)` va in background,
`date -u` produce subito il suo output in GMT,
ricompare il prompt `/home/user1 $`
dopo 10 secondi termina `(sleep 10; date)`
e scrive la data accanto al prompt.

Controllo dei processi

`exec` *cmd* è un comando di shell: esso fa sì che il processo che esegue la shell cessi di eseguirla e passi a eseguire *cmd*;
se la shell è quella di login, terminato *cmd*, l'utente si trova "fuori"

`kill` *process-id* fa terminare il processo *process-id*.

`kill -9` *process-id* dovrebbe funzionare nei casi più ostinati

`nice` *comando* abbassa la priorità di esecuzione di *comando*;
`nice` -*n comando* abbassa la priorità di *comando* al crescere di *n*;
solo il super user può usare *n* negativo

`ps` mostra i processi associati al terminale da cui è invocato
per ogni processo mostra il comando in esecuzione e il *process-id*.

`ps -a` mostra tutti i processi richiesti più di frequente.

Esercizio:

Creare dei processi con `&` e distruggerli con `kill`.

La shell: metacaratteri

Alcuni caratteri hanno un significato speciale per la shell e si dicono perciò ***metacaratteri***, p.es.: `> < | * ? [] ; &`

Talvolta però si può volerli usare come caratteri.
P.es. per scrivere `a*` sul terminale, `echo a*`, non va bene:

```
/home/user1 $ echo a*
affari agenda            NB: * è stato interpretato come jolly dalla shell
```

Per proteggere i metacaratteri dall'interpretazione di shell si può:

- racchiuderli tra apici

```
/home/user1 $ echo 'a*'
a*
/home/user1 $ echo '\'
\
```

- farli precedere da un *backslash* \

```
/home/user1 $ echo a\*
a*
/home/user1 $ echo \'
'
```

In effetti qualsiasi carattere così trattato resterà indisturbato.

Tra `''` e \ si può perfino battere un `Return` e non averlo interpretato come segnale che il comando è pronto:

```
/home/user1 $ echo 'ciao
> caro'
ciao                     il Return è stato interpretato come carattere
caro
/home/user1 $ echo ciao\
> caro
ciaocaro                 il Return è stato ignorato ma non interpretato
```

Il segno > alla seconda riga sopra si dice ***prompt secondario*** e indica che la shell aspetta altro input per concludere il comando.

Creare comandi di shell

Supponiamo di usare spesso il comando `ls | wc -l`
per contare i file nella directory corrente.

Per battere meno tasti, si può scrivere il comando in un file `nls`:

```
/home/user1 $ echo 'ls | wc -l' > nls
/home/user1 $ cat nls
ls | wc -l
```

e rendere il file `nls` ***eseguibile***

```
/home/user1 $ chmod +x nls
```

ora `nls` è un vero comando, equivalente a `ls | wc -l`:

```
/home/user1 $ nls
      8
```

Ecco ciò che accade:

- se si chiede alla shell di eseguire un file testo,
- essa genera un processo che esegue (una copia della) shell, detta ***subshell***,
- questa copia esegue i comandi contenuti nel file testo, questo si dice dunque ***programma di shell*** o ***shell script***

Programmi di shell: parametri e argomenti

Supponiamo di voler abbreviare il comando `chmod +x nls` visto prima, con `cx`. Creiamo uno script `cx`:

```
/home/user1 $ echo 'chmod +x $1' >cx
/home/user1 $ chmod +x cx
```

Se si invoca `cx` con un argomento, la subshell che esegue `cx` sostituisce il `$1` con l'argomento:

```
/home/user1 $ echo echo ciao > saluto
/home/user1 $ cat saluto
echo ciao
/home/user1 $ saluto
saluto: cannot execute
/home/user1 $ cx saluto
/home/user1 $ saluto
ciao
```

In generale:

- se un comando invocato con *n* argomenti è uno script,
  la subshell che lo esegue sostituisce `$`*i* con l'*i*-esimo argomento
- `$`*i* si dice ***parametro (posizionale)*** dello script

P.es, per dare 5 argomenti a `cx`:

```
/home/user1 $echo 'chmod +x $1 $2 $3 $4 $5' >cx
```

Ma questo è scomodo e non consente di andare oltre `$9`.

Per fortuna si può usare il parametro `$*` che viene sostituito da tutti gli argomenti con cui è chiamato il programma di shell.

I parametri posizionali `$1 $2 ...` non possono essere modificati da un programma di shell.

Variabili di shell

Una variabile di shell è definita da una stringa (che non sia `1 2`...)

Ad essa si può assegnare un valore con l'operatore `=`

```
/home/user1 $ aff=/home/user1/affari
```

E si può fare riferimento a questo valore premettendo un `$`:

```
/home/user1 $ echo $aff
/home/user1/affari
```

e usarlo anche nel lato destro di un'assegnazione:

```
/home/user1 $ aff=$aff/lettere
/home/user1 $ echo $aff
/home/user1/affari/lettere
```

D'ora in poi `$aff` si può usare al posto del suo valore:

```
/home/user1 $ ls -C $aff
fiat  ibm   rai
```

Variabili di shell, cont.

Alcune variabili sono necessarie per la shell e per altri programmi che la shell manda in esecuzione.
I loro nomi sono maiuscoli per convenzione. P.es.

Nome	***Valore***
$HOME	la home directory dell'utente
$PS1	il prompt dell'utente
$PATH	sequenza di directory separate da : quando la shell riceve un comando *cmd*, cerca un file *cmd* nelle directory di $PATH; se non lo trova risponde *cmd*: not found

I valori di queste variabili vengono assegnati al momento del login:

- da uno script prefissato,
  (`.profile` sotto la shell di Bourne o `.login` sotto la c-shell, listatelo con `ls -a`)
- per default dal sistema (`/etc/profile`)

e possono essere visualizzati con **`set`** (tutti) o uno alla volta con:

```
/home/user1 $ echo $HOME
/home/user1
/home/user1 $ echo $PATH
.:/bin:/usr/bin
```

Il `.` iniziale di `$PATH` indica che i comandi vanno cercati innanzitutto nella directory corrente.

Questi valori possono essere cambiati durante la sessione:

```
/home/user1 $ PATH=$HOME/bin:$PATH
/home/user1 $ echo $PATH
/home/user1/bin:.:/bin:/usr/bin
```

Così `$HOME/bin` sarà la prima dir dove la shell cerca i comandi.

Variabili di shell, cont.

Le variabili sono “private” della shell che le crea con un’assegnazione.

Ma variabili come `HOME`, `PATH` sono necessarie anche a molti programmi attivati dalla shell.

Affinché la shell *esporti* il valore di queste variabili ad altri programmi, occorre il comando `export`:

```
export HOME PATH
```

Per lo stesso motivo, solo la shell può modificare le sue variabili:

```
/home/user1 $ x=Ciao; echo $x
Ciao
/home/user1 $ sh                invoca un’altra shell
/home/user1 $ echo $x           in questa shell x non è definita
/home/user1 $ x=Addio; echo $x
Addio
/home/user1 $ ^D
/home/user1 $ echo $x           qui x è ancora Ciao
Ciao
/home/user1 $ export x; sh
/home/user1 $ echo $x           ora x è Ciao anche qui
Ciao
```

Il comando di shell .

Ciò che vale per le shell interattive vale anche
per i programmi di shell, in quanto eseguiti da subshell.

Programma `chngpath` per premettere a `PATH` la directory `mia`:

```
/home/user1 $ echo $PATH
/bin:/usr/bin
/home/user1 $ echo 'PATH=mia:$PATH' >chngpath
/home/user1 $ chmod + x chngpath
/home/user1 $ cat chngpath
PATH=mia:$PATH
```

Adesso proviamo a eseguire `chngpath`:

```
/home/user1 $ chngpath; echo $PATH
/bin:/usr/bin
```

Come si vede `chngpath` non ha effetto, perché la subshell che esegue `chngpath` cambia solo la sua copia privata di `$PATH`.

Per farlo funzionare, invece:

```
/home/user1 $ . chngpath
/home/user1 $ echo $PATH
mia:/bin:/usr/bin
```

Premettendo il `.` al comando `chngpath`:

- la shell **non** genera una subshell che esegue `chngpath`, ma:
- la standard input della shell diventa il file `chngpath`, per cui i comandi nel file sono eseguiti come se battuti alla tastiera (tra l'altro ciò rende inutile che il file `chngpath` sia eseguibile),
- è la stessa shell che assegna un nuovo valore alla sua variabile `PATH`, come desiderato

Rovescio della medaglia: i parametri posizionali `$1`... non possono essere usati in un file eseguito premettendo `.`