Parte 9

Compendio shell

[P. Bruegel – Children's games, 1560]

Shell o Terminale

- Nei sistemi UNIX-like è molto usato il concetto di ***Shell (Terminale)***
- In un OS, shell (o terminale) è un programma che permette agli utenti di comunicare col sistema
  - Es. Avviare altri programmi (esecuzione di un nostro eseguibile ./a.out)
- Una delle **componenti principali di un sistema operativo** (insieme al kernel)

Shell testuali o grafiche

- Esistono molti tipi di shell, che si dividono principalmente in ***testuali e grafiche***
  - **Shell grafica (o desktop environment)**: un ambiente grafico da cui è possibile gestire file e programmi
  - **Shell testuale (o terminale)**: un ambiente dotato di interfaccia a riga di comando
    - Tra le più note in sistemi operativi Unix-like vi sono: Bourne Shell (sh) e Bourne-Again Shell (bash) – versione più evoluta

Terminale

- Quando un programma viene mandato in esecuzione da un terminale, esistono ***3 flussi standard*** che vengono aperti automaticamente: ***stdout, stderr e stdin***
- Lo ***stdout*** e lo ***stderr*** del programma sono tipicamente agganciati al dispositivo di uscita del terminale (video)
- Lo ***stdin*** del programma è tipicamente agganciato al dispositivo di ingresso del terminale (tastiera)

Terminale virtuale

- In ambiente grafico, è usato il concetto di ***terminale virtuale (pseudo-terminale)***
  - Applicazione grafica che emula un terminale
- Esempi di terminali virtuali sono le applicazioni grafiche
  - **Unix-like: xterm, Konsole, gnome-terminal**
  - **Windows: cmd.exe**

Terminale virtuale

- Quando viene avviato, lo pseudo-terminale manda in esecuzione un programma di interfaccia a riga di comando (**shell**)
- La finestra grafica dello pseudo-terminale funge da dispositivo di output a cui sono collegati ***stdout e stderr***
- Quando tale finestra è in primo piano, lo pseudo-terminale inoltra i caratteri premuti da tastiera allo ***stdin*** del programma

Shell in ambiente Unix-like

- Entrambe le shell più usate (sh e bash) sono tipicamente installate nella cartella /bin
  - which bash ; which sh
- In molti sistemi GNU/Linux **bash ha sostituito sh** (a volte **dash** in Debian)
  - Il file /bin/sh di fatto è un link al file /bin/bash, per cui bash è eseguita al posto di sh in modo trasparente
  - ls -l /bin/sh

bash

- Si farà riferimento a bash
- Vedremo solo un **sottoinsieme sintetico** delle caratteristiche di bash
- Per la documentazione sintetica ma completa si può far riferimento alle pagine di manuale

  man bash

Spazi negli argomenti

Come passare ad un comando un ***argomento contenente spazi*** per evitare che sia interpretato come due argomenti distinti?

- Es. Per stampare, mediante il comando **cat**, il contenuto del file **Prima Prova.txt** su stdout
- **cat Prima Prova.txt** non funziona

Spazi negli argomenti

2 possibilità:

- Racchiudere le parole dell'argomento tra doppi apici “ ...”
- Far precedere ciascuno spazio da \
  (lo spazio è un carattere speciale)

Quindi:

cat "Prima Prova.txt"

oppure

cat Prima\ Prova.txt

- **Date queste complicazioni, è meglio evitare nomi di file contenenti spazi!**

Utilizzo della history

- Capita spesso di dover rieseguire gli stessi comandi già eseguiti
  - Comando **history**
- E' molto comodo utilizzare i ***tasti cursore*** per richiamarli
- Inoltre ricordatevi della ***ricerca incrementale all'indietro***:
  - Ctrl-R, e poi basta inserire solo una sottostringa della riga di comando precedentemente invocata

Completamento automatico

- Mediante il tasto **TAB** avviene il **completamento automatico**
  - Vale anche per i nomi di file presenti
  - Se possibile, viene completato il comando
  - Altrimenti viene inserito il prefisso più lungo in comune tra i possibili completamenti
- Se si preme due volte di fila il tasto **TAB** viene mostrato l'elenco dei possibili completamenti (se più di uno)

Definizioni

Prima di iniziare, diamo le seguenti definizioni:

- ***Metacarattere***
  | & * ? ; ( ) < > spazio tabulazione
- ***Blank***
  si chiamano così spazio e tabulazione
- ***Parola***
  sequenza di caratteri delimitata da metacaratteri

Token e operatori di controllo

- Parole e metacaratteri sono utilizzati da bash come elementi lessicali (***token***)
  - Unità di base che compongono un comando
- Un altro importante token è l'***operatore di controllo*** che effettua, appunto, operazioni di controllo (es. terminare la linea di comando)
  - Considereremo solo <newline> (a capo) e ; (per terminare un comando)

Linee di comando

- Bash interpreta ed esegue ***linee di comando*** di diversa complessità sintattica
- Nella forma più semplice una linea di comando (chiamata semplicemente comando) è costituita da:
  - una sequenza di parole
  - un operatore di controllo (obbligatorio)

Comando

- In modo un po' ambiguo, la prima parola della linea di comando è chiamata ***comando (semplice)***
- Come primo passo, bash controlla se tale comando è uno dei ***comandi integrati (built-in)*** o si tratta di un ***comando esterno***

Comando integrato

- Si definisce integrato un ***comando che viene eseguito direttamente dalla shell***, senza invocare nessun altro programma esterno
  - **cd** è un esempio di comando integrato
- Non esiste un file eseguibile corrispondente
  - **which cd** non restituisce nulla

Comando esterno

Se il comando è esterno, vi sono due possibilità:

1. ***Se il nome non contiene nessuno slash (/)*** bash cerca una cartella contenente un file con tale nome nella propria lista di directory candidate (variabile d'ambiente **PATH**)
2. ***Se il nome contiene almeno uno slash***, bash assume che sia un percorso verso un file eseguibile (percorso relativo o assoluto, a seconda che lo / sia all'inizio)

Esempio caso 1.

- Se si invoca il comando **ls** bash cerca nella lista di cartelle indicate in **PATH** una cartella contente un file di nome **ls**
- **echo $PATH**
  - Lista di cartelle separate da **:**
- Verificate dove si trova con **which ls**
- Supponendo che tale cartella sia **/usr/bin**, bash manda in esecuzione il programma memorizzato nel file **/usr/bin/ls**

Esempio caso 2.

- Se si invoca il comando

  /home/guest/mio_comando

  bash prova a mandare in esecuzione il programma memorizzato nel file corrispondente: **/home/guest/mio_comando**
- Stessa cosa per le invocazioni degli eseguibili compilati **./a.out** (percorso relativo)

Agire sulla variabile PATH

- E' possibile inserire nella variabile **PATH** una directory personale dove l'utente tiene i propri eseguibili

  export PATH=$PATH:/home/guest/
- ***NB:*** modifica solo il terminale aperto!
- Per rendere permanenti le modifiche, scriverlo nel file **~/.bashrc**

Esecuzione ed argomenti

- In entrambi i casi di comando esterno, se il file viene trovato, **bash** prova a caricarlo in memoria e ad iniziarne l'esecuzione
- Ciascuna delle parole successive sulla linea di comando è interpretata come un ***argomento da passare al comando***
  - In realtà, al comando è passato implicitamente anche il nome del comando stesso, quindi gli argomenti successivi

Esempio

- Se si invoca il comando

  g++ sorgente.cc -o prog

  bash passa le parole
  - g++
  - sorgente.cc
  - -o
  - prog

  come argomenti al programma g++

Lettura argomenti

- Come leggere tali argomenti nei nostri programmi C++?
- E' legale dichiarare la funzione **main** anche nel modo seguente:
  int main(int argc, char *argv[]) ;
- Ossia **main** può prendere in ingresso un intero **argc** seguito da un array di puntatori a carattere **argv**

Lettura argomenti

- In **argc** è memorizzato il numero di elementi validi nell'array **argv**
- Ciascun puntatore a carattere contenuto nell'array **argv** punta ad una stringa contenente uno degli argomenti
  - Incluso il nome stesso del programma!
- **argv** è quindi un array di stringhe accessibili con indice compreso tra **0** e **argc - 1**

Esempio (1)

- Ipotesi: **mio_prog** è l'eseguibile risultante dalla compilazione di un programma C++ in cui il main ha argc ed argv per argomenti
- L'invocazione del seguente comando

  ./mio_prog prova

  comporta l'esecuzione del programma e ...

Esempio (2)

… la memorizzazione dei seguenti valori in argc ed argv

- argc = 2
- argv[0] = Indirizzo di una stringa contenente “./mio_prog”
- argv[1] = Indirizzo di una stringa contenente “prova”

Programma

- Scrivere un programma che stampi su **stdout** tutti gli argomenti ricevuti in ingresso
- Soluzione in *stampa_argomenti.cc*

Cosa può essere eseguito?

- Torniamo ai comandi esterni...
- Si è visto che se il file che viene trovato contiene un ***programma eseguibile***, quest'ultimo viene mandato in esecuzione
- E' l'unico tipo di file che può essere eseguito?
- ***No, ci sono anche gli script ...***
  - ***Programmi scritti in linguaggio interpretato***

Script

- Si è visto che bash è un ***interprete dei comandi di un linguaggio***
- Ma, dato un linguaggio, possiamo utilizzarlo per scrivere programmi!
- Uno ***shell script*** è un programma nel linguaggio di una shell
- Vediamo due dei modi in cui può essere eseguita bash

Modalità di esecuzione bash

bash può essere eseguita in:

1) modalità interattiva:

i comandi sono letti da **stdin**

2) modalità non interattiva:

i comandi sono letti da un file (**script**)

Modalità interattiva

- bash presenta un prompt (invito) all'utente (**$** o **>**) che immette le linee di comando
- Le linee di comando sono terminate mediante l'operatore di controllo **<newline>**
- L'operatore di controllo **<newline>** è equivalente a **;**
- Di fatto in modalità interattiva un comando non è invocato finché non si preme invio

Modalità non interattiva

- La modalità non interattiva è utilizzata tipicamente per eseguire **script**
- Per far partire l'esecuzione di uno **script** basta immettere il suo nome nella linea di comando, come un eseguibile
- Sarà ***implicitamente lanciata una seconda shell non interattiva*** che lo eseguirà e poi terminerà

Script

- ***Uno script è un file di testo che contiene una sequenza di comandi da eseguire in un linguaggio interpretato***
- Tipicamente si usa il **suffisso .sh** per i nomi degli script shell
- Vedremo ora pochissimi elementi di sintassi di uno script shell

Esempio

Un file di testo contenente:

ls

./mio_prog

fa eseguire a bash prima il comando **ls**, poi l'eseguibile **mio_prog** (cercandolo nella cartella da cui si lancia lo script)

E' equivalente a: **ls ; ./mio_prog**

Diritti di esecuzione

- Per essere eseguito da un certo utente mediante invocazione, uno script deve avere i **diritti di esecuzione** per tale utente
- Fare riferimento ad esempio alle **slide sui comandi di base del corso di introduzione a Linux** per i dettagli sui diritti

  http://informatica.scienze.unimo.it/corso_linux/primi_passi.pdf
- Uno dei modi per dare i diritti (a tutti) è: **chmod a+x nome_script**

Programma

- Scrivere ed eseguire uno script che stampi su **stdout**

  Sto per eseguire il programma

  e quindi esegua il programma di stampa degli argomenti *stampa_argomenti.cc*
- Per stampare il messaggio, utilizzare il comando **echo**
- Usare le pagine di manuale se necessario
- Soluzione in *primo_script.sh*

Variabili

- Anche bash – come tutti i linguaggi di scripting e di programmazione – fa uso di variabili
- Una variabile è una ***entità che memorizza valori***
  - Fondamentalmente stringhe o numeri
- Nel linguaggio di bash ***non è obbligatorio dichiarare il tipo di una variabile***

Variabili

- Per ***dichiarare una variabile*** è sufficiente:

  nome_variabile=[valore]
- Il valore è opzionale, se assente alla variabile si assegna la *stringa nulla*
- La stessa sintassi può essere utilizzata anche per aggiornare il valore di una variabile (***fase di assegnamento***)

Utilizzo di una variabile

- Se ci si riferisce al ***valore contentuto nella variabile*** (suo utilizzo in operazioni) si deve utilizzare l'espansione:

  $nome_variabile

  oppure

  ${nome_variabile}

- La seconda forma evita possibili interpretazioni errate dei caratteri che identificano la variabile

Espansione

- Si parla di espansione perchè, nel punto in cui appare questo costrutto, viene ***inserito (espanso) testualmente il valore del parametro*** durante l'interpretazione della linea di comando interessata
- In modalità interattiva, provare a definire una variabile **var** e a stamparne il valore su **stdout** usando **echo**

Soluzione

var=prova

echo $var

Provare **echo var** vs. **echo $var** vs. **echo ${var}**

Variabili d'ambiente

- Una variabile dichiarata in modalità interattiva o dentro uno script ***non è visibile in un'altra shell → Variabile locale***
- Definiamo ***variabile di ambiente*** una ***variabile globale*** che viene ereditata da ogni nuova shell avviata
- Definiamo ***ambiente di una shell*** l'insieme delle variabili d'ambiente visibili in tale shell
- Comando **env** per visualizzarle

Variabile PATH

- Un esempio importante di variabile d'ambiente predefinita è **PATH**
- Contiene la ***lista delle directory in cui la shell cerca gli eseguibili***
- Ciascuna directory è separata dalle altre mediante il carattere :
- Modificando **PATH** possiamo cambiare le directory in cui bash cerca gli eseguibili

Parametri posizionali

Oltre alle variabili, vi sono due tipi di parametri: ***posizionali e speciali***

- ***Parametri posizionali***: contengono gli argomenti con cui uno script è invocato. Sono identificati mediante numeri interi.

  Es. All'interno di uno script:

  echo $1

  stampa il primo argomento passato

  echo $0 *cosa fa?*

- ***$# dà il numero dei parametri passati***

Parametri speciali

- ***Parametri speciali***
- Ve ne sono vari
- Uno di questi è **?**, che contiene lo **stato di uscita** dell'ultimo comando eseguito

  Es. Il comando

  echo $?

  stampa lo **stato di uscita** dell'ultimo comando eseguito

Stato di uscita

- Lo stato di uscita di un comando è usato in bash per determinare, in base a delle convenzioni, ***se il comando ha avuto successo o ha fallito***
  - Il valore **0** è utilizzato come indicazione di **successo**, mentre tutti i valori **diversi da 0** sono considerati indicazione di **fallimento**
    - *Lo abbiamo visto in C++?*
- Provare: **cat file_esistente ; echo $?**
  cat file_inesistente ; echo $?

Stato di uscita di un programma C++

- Come facciamo ad indicare ***lo stato di uscita in un programma C/C++***?
  - *Nel main*: mediante il valore passato all'istruzione **return**
  - *In qualsiasi punto del programma:* mediante il valore passato alla funzione **exit**
- Modificare **stampa_argomenti.cc** in modo da far ritornare stati di uscita diversi
  - Se vengono passati meno di 2 argomenti, ritorna lo stato di errore 1

Comandi composti

- Continuiamo il nostro excursus sui comandi shell
- Esistono anche i ***comandi composti***
- Come esempio vediamo solo il comando **if ... then ... else ... fi**
- Per illustrarlo consideriamo prima il **comando integrato test**

Comando test

Il comando **test** prende per argomento una ***espressione condizionale*** e ritorna:

- stato **0** se l'espressione è **vera**
- stato **1** se l'espressione è **falsa**

ATT: è il contrario della convenzione logica usata per C/C++!

Espressioni condizionali

Esempi di espressioni condizionali in shell:

- **string1 == string2** o **string1 != string2**

  Verifica che le due stringhe siano rispettivamente uguali oppure diverse

- **arg1 OP arg2** (con arg1 e arg2 interi)

  OP può essere: **-eq, -ne, -lt, -le, -gt, -ge**

  L'espressione è vera se **arg1** è uguale, diverso, minore, minore-uguale, maggiore, maggiore-uguale rispetto ad **arg2**

Esempi

- **test $v == $g**
- Ritorna 0 se le stringhe memorizzate nelle variabili v e g sono uguali

- **test $i -lt 5**
- Ritorna 0 se il valore memorizzato nella variabile i è minore di 5

Forma sintetica

- In modo più sintetico, il comando test si può scrivere nella forma

  [espressione_condizionale]

  anziché

  test espressione_condizionale
- Bisogna lasciare un blank dopo [e prima di]

Esempio

- Stampare lo stato ritornato da **test** per una espressione vera e per una espressione falsa

Espressione vera

```
[ 2 -lt 3 ]
echo $?
0
```

Espressione falsa

```
[ 3 -lt 2 ]
echo $?
1
```

Comando composto if

- Sintassi:

  if <comando> then
  <sequenza_di_comandi_1>
  [else
  <sequenza_di_comandi_2>]
  fi

- Il comando **if** controlla lo stato di uscita di <comando> e *se è uguale a 0, esegue <sequenza_di_comandi_1>*

Comando richiesto!
In C++ bastava
la condizione...

Comando composto if

- Se presente, il ramo **else** è eseguito se lo stato di uscita di <comando> è diverso da 0
- Così come in C/C++, l'indentazione serve solo per leggibilità
- **A delimitare i blocchi sono le parole chiave if, then, else e fi**
- Ricordare che **ogni comando va terminato con ; o <newline>** !!!

Esercizio

- Scrivere ed eseguire uno **script** che:
  - prenda in ingresso 2 parametri
  - stampi su **stdout** “Minore”, “Uguale” o “Maggiore” a seconda che il primo argomento che gli viene passato sia rispettivamente minore, uguale o maggiore del secondo
- Ricordatevi dei diritti di esecuzione …
- Soluzione in *confronta.sh*

Esercizio

- Cosa succede se *confronta.sh* viene invocato senza parametri?
- Estendi il programma *confronta.sh* perchè controlli di avere ricevuto in ingresso i due parametri richiesti (non di più e non di meno)

- Soluzione in *confronta-completo.sh*

Esercizio

- Dato *stampa_argomenti_esteso.cc* (già usato per testare gli stati di uscita) scrivere uno script che lo richiami e stampi
  - “Tutto OK” se lo stato di uscita del programma è uguale a 0
  - “Qualcosa è andato male” se lo stato di uscita è diverso da 0
- In entrambi i casi deve stampare anche il valore dello stato di uscita

Soluzione

- Soluzione in *controlla_stato.sh*
- Soluzione alternativa

```
if ./stampa_argomenti ; then
        echo Tutto OK, stato $?
else v=$?
        echo Qualcosa è andato male,
        echo stato $v
fi
```

Domanda

- La seguente soluzione è corretta?

```
if stampa_argomenti ; then
        echo Tutto OK, stato $?
else
        echo Qualcosa è andato male,
        echo stato $?
fi
```

Risposta

- No, perché, lo stato di uscita stampato dal secondo comando **echo**

  echo stato $?

  è di fatto lo stato di uscita del precedente comando **echo**, ossia 0, e non dell'invocazione a **stampa_argomenti**
  - Se **stampa_argomenti** ritorna un valore diverso da 0, viene invece stampato 0

Nota

- Notare che **anche in modalità interattiva la bash è un interprete di un linguaggio di programmazione**
- Si può quindi immettere gli script dalla riga di comando una riga alla volta – o su una sola riga a patto di terminare tutti i comandi intermedi con ; e non con <newline>
  - *Incluso il comando composto **if then else fi**!*

Stdout, stderr, stdin

Quando manda in esecuzione un comando, un eseguibile o uno script, bash ***aggancia*** **stdout, stderr** e **stdin** del comando/eseguibile/script al proprio **stdout, stderr e stdin**

- ***l'output*** del comando/eseguibile/script appare nella finestra dello pseudo-terminale
- ***l'input*** dello pseudoterminale (quindi dalla tastiera) viene dato in ingresso al comando/eseguibile/script

Redirezione

- Questo comportamento di default si può modificare attraverso la ***redirezione***
- **Stdout, stderr e stdin** possono essere **redirezionati su un qualunque file**
  - il file in questione prende le veci del dispositivo associato al flusso redirezionato
- ***Scopi primari della redirezione***:
  - prendere ingressi da file
  - scrivere uscite su file
  - scrivere errori su file diversi da quelli dell'uscita

Operatori di redirezione: >

- Redirige lo standard output su un file qualunque

  ls > ls.output
- Il file **ls.output** è aperto in scrittura, quindi se non esiste viene creato, mentre, se esiste, il suo precedente contenuto è **perso**
- ***NOTA: > redirige solo stdout, NON stderr***
  - gli errori continuate a vederli sul terminale..

    ls * pippo1 > ls.output

    ls: pippo1: No such file or directory

Esercizio

- Mediante i comandi visti finora (senza realizzare un programma in C/C++) creare un file di testo contenente la stringa “Ciao mondo”
  - **Usare comandi shell e redirezione**
- Utilizzare il comando **cat** per verificare il successo dell'operazione, visualizzando il contenuto del file

Soluzione

- **echo Ciao mondo > out.txt**
- **cat out.txt**

Sul comando cat

- Se invocato senza nessun argomento, il comando **cat** stampa su **stdout** quello che legge da **stdin**, finché non incontra il marcatore **EOF**
- Provare ad utilizzarlo in questa modalità per scrivere lo stdout in un file di testo inserendo i caratteri da tastiera
  - Ricordare che **EOF** viene generato attraverso la combinazione di tasti [Ctrl+d]

Soluzione

```
cat > out.txt
Ciao mondo
^D

cat out.txt
Ciao mondo
```

Redirezione programma C++

- *raddoppia.cc*
- Esempio di programma in C++ che scrive su **stdout** i numeri interi che legge da **stdin** moltiplicati per 2, fino a che non incontra **EOF**
- Eseguire il programma redirigendo l'output su un file di testo e verificare il risultato

Soluzione

- // sorgente *raddoppia.cc*

```
main()
{ int a ;
   while(cin>>a)
   cout<<2*a ;
}
```

- ---------- Esecuzione da bash:

```
./raddoppia > out.txt
2 4 5 ^D
cat out.txt
4 8 10
```

Operatori di redirezione: 2>

- **Redirige lo standard error su un file**

  ls 2> ls.error
- ***NOTA: 2> redirige solo stderr, NON stdout!***
  - l'output continuate a vederlo sul terminale...
  - ls * pippo1 2> ls.error
  - ... output directory ...
  - cat ls.error
  - ls: pippo1: No such file or directory

Separare stdout e stderr

- Si ottiene utilizzando i simboli di redirezione > e 2> nello stesso comando

  ls * > ls.out 2> ls.err

- La redirezione proposta **separa l'output del programma dalla stampa degli errori**

Esercizio

- Scrivere un programma in C++ che legga dei numeri interi da tastiera e stampi su **stdout** i numeri dispari e su **stderr** quelli pari, fino a che non incontra **EOF**
- Eseguire il programma redirigendo lo **stdout** su un file (es. out.txt) e lo **stderr** su un altro file (err.txt)

Soluzione

```
main()   //sorgente pari_dispari.cc
{ int a ;
  while(cin>>a)
     if (a%2) cout << a << endl ;
     else cerr << a << endl ;
}
```

- ----------Esecuzione da bash:

  pari_dispari > out.txt 2> err.txt
  2 3 4 5 8 9 ^D
  cat out.txt   3 5 9
  cat err.txt   2 4 8

Redirezione in append: >>, 2>>

- ***Redirigono lo standard output/error su un file qualunque, senza sovrascriverlo***
- Gli operatori **>, 2>** sovrascrivono il contenuto del file di redirezione
- Gli operatori **>>, 2>>** aggiungono il contenuto in coda al file di redirezione (se il file non esiste, lo creano in scrittura)

Operatori di redirezione: <

- Redirige lo standard input su un file qualunque
- ***NOTA: < redirige solo stdin, NON stdout o stderr***
  - output ed errori continuate a vederli sul terminale...

./pari_dispari < input.txt
Legge gli argomenti dal file **input.txt**

Pipe e pipeline

- La filosofia UNIX supporta il concetto di ***separazione*** dei compiti fra programmi
  - Un singolo programma deve fare una cosa sola e farla bene
- Programmi più complessi possono essere costruiti a partire da programmi più semplici
- ***Il meccanismo che permette di "concatenare" più programmi è denominato pipe***

Definizione di pipe

- L'**operatore di pipe** è contrassegnato dal simbolo |
  - comando1 | comando2 | ... | comandoN
- **Significato**:
  - lo **stdout** del comando1 viene dato in pasto allo **stdin** del comando2
  - lo **stdout** del comando2 viene dato allo **stdin** del comando3
  - lo **stdout** del comando(N-1) viene dato allo **stdin** del comandoN

Esempio

- Proviamo a risolvere il seguente problema:
  - Convertire in maiuscolo lo standard input
  - Usiamo un programma C++
  - **conv_maiusc.cc** : legge un carattere alla volta e lo trasforma in maiuscolo usando la funzione **toupper()**

Esempio

Convertire in maiuscolo l'output del comando **ls** con **conv_maiusc**:

- **ls | ./conv_maiusc**

Che cosa è successo?

- la shell fa partire due processi: **ls** e **conv_maiusc**
- la shell fa puntare lo **stdout** di **ls** allo **stdin** di **conv_maiusc**
- **ls**, non accorgendosi di niente, scrive dati sul suo **stdout**
- **conv_maiusc**, non accorgendosi di niente, elabora dati dal suo **stdin**

Pipeline

- L'espressione shell:

  command1 | command2 | ... | commandN

  prende il nome di ***pipeline***
- Lo **stato di uscita** associato alla pipeline è lo stato di uscita dell'ultimo comando:
  - **== 0: tutto ok**
  - **> 0: errore**

Riassumendo: pipeline internals

Cosa fa la shell quando legge una pipeline da riga di comando?

- **STEP 1**: per ciascuna pipeline, fa partire un processo shell
- **STEP 2**: la shell collega lo **stdin** del processo successivo con lo **stdout** del processo precedente

Esercizio

- Eseguire in pipeline il programma che legge un numero intero e lo stampa raddoppiato, fornendo l'input mediante il comando **echo** (senza passare attraverso un file di testo)

Soluzione: **echo 1 2 3 4 | raddoppia**

Composizione comandi

- Utilizzando le pipeline si può risolvere in modo efficiente e brillante il precedente *problema della conversione in maiuscolo* (**conv_maiusc**)
- Ad esempio, componendo in pipeline col il comando **tr** per ottenere il risultato in un'unica riga

  ls | tr '[a-z]' '[A-Z]'
  oppure
  echo proVA | tr '[:lower:]' '[:upper:]'

Operatore di controllo &

- Come anticipato, vi sono altri operatori di controllo oltre **<newline>** e **;**
- In aggiunta vediamo solo l'operatore **&**
- Si manda in esecuzione il comando in ***background***
  - Ossia lo si fa partire e si riprende subito il controllo del terminale

Esempi uso &

- Considerate l'invocazione di un editor di testo che apre una propria finestra
  - Ad esempio **kate, gedit o emacs**
- Se lo si invoca da riga di comando il terminale rimane bloccato fino alla chiusura dell'editor
- Tuttavia l'editor non interagisce mai con l'utente mediante il terminale

Esempi uso &

- In generale, quando si hanno ***programmi che non utilizzano il terminale*** può essere pratico non sprecare il terminale stesso (e doverne magari usare un altro)
- Se si fa partire il programma con l'operatore **&** il controllo torna subito al terminale

Non sempre si può usare &

- Ovviamente un programma mandato in esecuzione in background può non funzionare correttamente se il suo **stdin/stdout/stderr** è collegato al terminale ed il programma usa questo canale di comunicazione
- Esempi sono l'**editor vi** (**vim**) o la maggior parte dei programmi che abbiamo scritto finora

Note conclusive

- Il **linguaggio di bash** ha la stessa potenza espressiva di un qualsiasi altro linguaggio di programmazione
- E' un linguaggio interpretato, con i pro ed i contro di questo tipo di linguaggi
- Uno **script** è un programma a tutti gli effetti
- *Si possono inserire commenti facendoli precedere dal carattere* **#**

Intestazione corretta di uno script shell

- Prima riga dello script

  #! /bin/sh

  specifica il programma in grado di interpretarli – in questo caso **/bin/sh**
- I nostri script funzionano lo stesso perchè invocati dalla shell stessa, che ne interpreta il contenuto come comandi shell

Intestazione corretta di uno script shell

- **Non sarebbe così per altri script**
  - Es. script perl **prova.pl**
- Uno **script perl** può essere invocato
  - Semplicemente con **./prova.pl** se nella sua prima riga c'è l'intestazione che indica dove si trova l'interprete perl

    #! /usr/bin/perl
  - Invocando esplicitamente il compilatore perl e dandogli come argomento il nome del file

    perl ./prova.pl

Documentazione

- Per **bash**, ed in generale per le altre applicazioni che vedremo, esistono **varie fonti di documentazione**
- La documentazione è quasi sempre disponibile gratuitamente sul **sito di riferimento** (quasi sempre esistente) o in generale in **rete**

Tipi di documento

Quattro categorie principali di documenti

- **Tutorial**
  - Rivolto ai nuovi utenti di uno strumento; insegna passo passo come utilizzarlo (spesso tramite esempi pratici)
- **Spiegazione completa**
  - Manuali, ..

Tipi di documento

- **Aiuto specifico per un compito**
  - Howto, Frequently Asked Questions (FAQ)
- **Riferimento veloce**
  - Manuali di riferimento, man pages in UNIX, help nel DOS

- Il **sito di riferimento** di bash è

  http://www.gnu.org/software/bash/