Espressioni CONDIZIONALI (3)

Alcuni operatori per verificare **condizioni su file** :

-d file — True if file exists and is a directory.
-e file — True if file exists.
-f file — True if file exists and is a regular file.
-h file — True if file exists and is a symbolic link.
-r file — True if file exists and is readable.
-s file — True if file exists and has a size greater than zero.
-t fd — True if file descriptor fd is open and refers to a terminal.
-w file — True if file exists and is writable.
-x file — True if file exists and is executable.
-O file — True if file exists and is owned by the effective user id.
-G file — True if file exists and is owned by the effective group id.
-L file — True if file exists and is a symbolic link.

file1 **-nt** file2 — True if file1 is newer (according to modification date) than file2, or if file1 exists and file2 does not.

file1 **-ot** file2 — True if file1 is older than file2, or if file2 exists and file1 does not.

Nota Bene: Ne esistono altre, per vedere quali guardare man bash nella parte intitolata CONDITIONAL EXPRESSIONS.

Espressioni CONDIZIONALI (4)

Alcuni operatori per verificare **condizioni su stringhe, aritmetiche e altre varie** :

-o optname True if shell option optname is enabled. See the list of options under the description of the **-o** option to the set command.

Operatori su stringhe

-z string True if the length of string is zero.

-n string True if the length of string is non-zero.

string1 **== string2**
string1 **= string2** True if the strings are equal

string1 **!= string2** True if the strings are not equal.

string1 **< string2** True if string1 sorts before string2 lexicographically.

string1 **> string2** True if string1 sorts after string2 lexicographically.

Operatori aritmetici su stringhe

arg1 OP arg2 **OP** is one of **-eq**, **-ne**, **-lt**, **-le**, **-gt**, or **-ge**. These arithmetic binary operators return true if arg1 is equal to, not equal to, less than, less than or equal to, greater than, or greater than or equal to arg2, respectively. Arg1 and arg2 may be positive or negative integers.

Esempio d'uso di Espressioni CONDIZIONALI

Due porzioni di codice bash che fanno la stessa cosa :

```
[[   -e  ${name}  ]]
if  (( $? == 0 )) ; then
        echo "esiste ${name}, lo elimino" ;
        rm -f ${name}
fi ;
```

analogamente

```
if  [[  -e  ${name} ]]  ; then
        echo "esiste ${name}, lo elimino" ;
        rm -f ${name}
fi ;
```

Esempio d'uso di Espressioni CONDIZIONALI

Due porzioni di codice bash che fanno la stessa cosa usando due diversi tipi di espressioni condizionali: [ ] e [[ ]]

```
if [ -d ${name} -a ${#name} -lt 10 ] ; then
        echo "ok ${name}" ;
else
        echo "no ${name}";
fi ;
```

Analogamente

```
if [[ -d ${name} && ${#name} -lt 10 ]] ; then
        echo "ok ${name}" ;
else
        echo "no ${name}";
fi ;
```

Attenzione all'interpretazione di &&
può essere
operatore AND logico in Espressioni CONDIZIONALI
o operatore in sequenza di comandi condizionale

QUI && è AND LOGICO

```
if [[ -e ${name} && ${#name} -lt 10 ]] ; then
        echo “ok ${name}" ;
fi
```

QUI && è operatore in sequenza di comandi condizionale

```
if [[ -e prova.c ]] && gcc -o prova.exe prova.c ; then
        echo “posso eseguire prova.exe"
fi
```

Evitare Errori Più Comuni (1)
METTETE SPAZI PRIMA E DOPO le [[e le]] ed anche PRIMA E DOPO i - delle espressioni

Ad esempio, l'espressione condizionale seguente

```
if  [[-d  /usr/include/stdio.h  ]] ;  then echo esiste ;  fi
```

provoca errore, perché la bash crede che [[-d sia un comando da eseguire, infatti avvisa:

```
[[-d: command not found
```

perché MANCA LO SPAZIO TRA le [[e il -d

Infatti, il parser della bash riconosce le espressioni condizionali prima separando la riga in parole delimitate da spazi. Se mancano gli spazi le espressioni condizionai non vengono riconosciute.

Analogamente per gli spazi prima e dopo le espressioni regolari.
Il comando

```
if  [[  -dFILECHENONESISTE ]] ;  then echo esiste ;  fi
```

esegue e mette in output esiste.

Sembra dire che FILECHENONESISTE esiste perché senza spazi viene valutata (E NON ESEGUITA) la stringa -dFILECHENON ESISTE che essendo non vuota restituisce vero.

Evitare Errori Più Comuni (2)

Cosa posso mettere dentro un if ?

Liste di Comandi o Espressioni condizionali

OK	`if ls prova ; then echo ciao ; else echo fanculo ; fi`
OK, ma strano	`if` ` ` `ls prova` ` ` `; then echo ciao ; else echo fanculo ; fi`
OK	`if [[ -e prova ]] ; then echo ciao ; else echo fanculo ; fi`

Cosa posso mettere dentro le [[ ]] ?

solo gli operatori delle espressioni condizionali, NON dei comandi, se non come command substitution

OK	`if [[ "aaa" == ` ` ` `cat prova` ` ` ` ]] ; then echo ciao ; else echo fanculo ; fi`
OK	`if [[ -e prova ]] ; then echo ciao ; else echo fanculo ; fi`
NOOO	`if [[ ls prova ]] ; then echo ciao ; else echo fanculo ; fi`

Espressioni CONDIZIONALI (5)

NOTA BENE:

RIBADISCO CHE NON POSSO eseguire dei COMANDI all'interno dell'operatore espressione condizionale. Posso mettere solo delle espressioni con operatori di confronto o di condizioni su file oppure delle command substitution.

Cioe' **non posso eseguire** un comando cosi' quello che segue perche' il comando specificato tra le parentesi quadre semplicemente NON VIENE ESEGUITO poiche' **all'interno delle espressioni condizionali non e' previsto che venga eseguito un comando**. Diverso è il caso delle command substitution.

NO !!!!!!!!!! ERRORE !!!!!!!

```
if  [[   ls nomefile.txt  ]] ;  then echo "il file esiste" ; else echo "il file non esiste" ; fi
```

Se occorre eseguire un comando e valutarne il risultato, NON DEVO METTERE il comando all'interno di una espressione condizionale ma devo metterlo semplicemente come condizione di un if, cosi':

SI

```
if   ls nomefile.txt  ;  then echo "il file esiste" ; else echo "il file non esiste" ; fi
```

File Descriptors

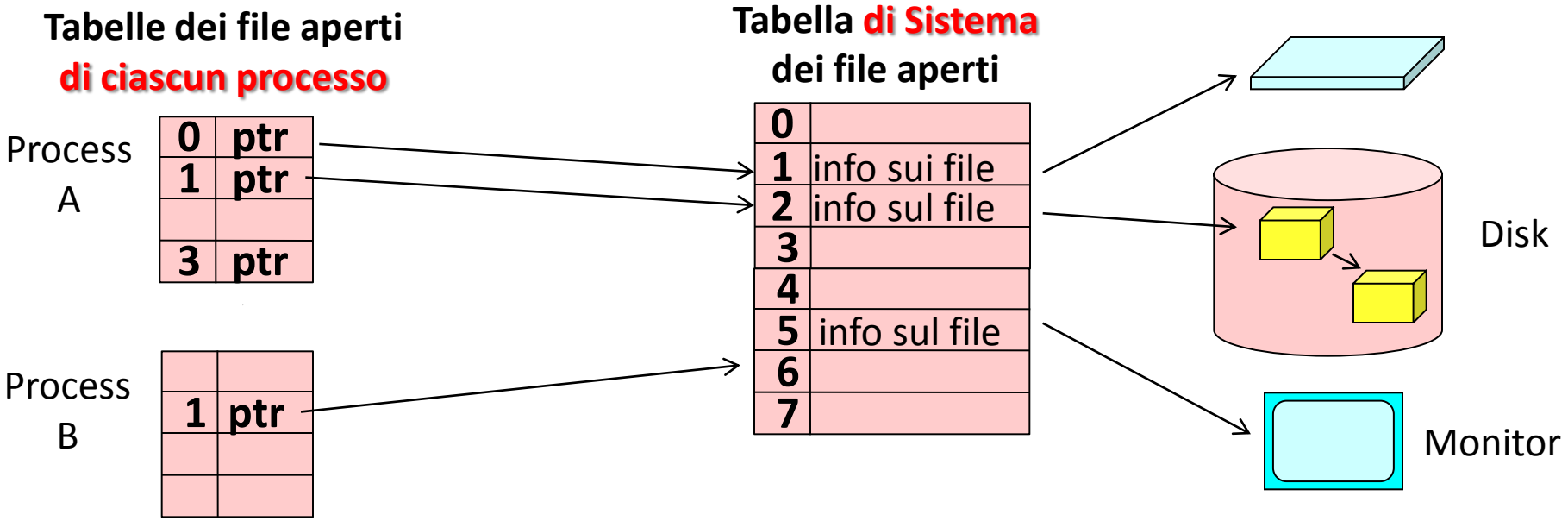
Il **file descriptor** è un'astrazione per permettere l'accesso ai file.
Ogni file descriptor e' rappresentato da un integer.
Il concetto di file descriptor esiste sia usando linguaggio C ed anche in bash, perché è un concetto fornito dal sistema operativo.

```
#define MAXSIZE 100
#define nomefile "/home/vic/piffero.txt"
void main(void) {
        int fd, ris; char buffer[MAXSIZE];
        fd = open( nomefile, O_RDONLY );
        if( fd < 0 ) {     perror("open failed: ");  exit(1); }

        // letture fino a raggiungere fine file
        do {
                ris=read( fd, buffer, MAXSIZE );
                if( ris<0 ) {     perror("read failed: ");  exit(1); }
        } while( ris>0);

        close(fd):
}
```

File Descriptors e Tabelle dei File Aperti

Il **file descriptor** è un'astrazione per permettere l'accesso ai file. Ogni file descriptor e' rappresentato da un integer.

Ogni processo ha la propria **file descriptor table** che contiene (**indirettamente**) le informazioni sui file attualmente utilizzati (aperti) dal processo stesso. In particolare, la file descriptor table contiene un file descriptor (un intero) per ciascun file usato dal processo.
Per ciascuno di questi file descriptor, la tabella del processo punta ad una tabella di sistema che contiene le informazioni sui tutti i file attualmente aperti dal processo stesso.
Si noti che due diversi processi possono avere due diversi file descriptor con stesso valore, ma che fanno riferimento a file diversi.



In POSIX ogni processo è inizialmente associato a 3 fd standard
stdin (0), stdout (1), stderr (2)