Compilazione di un programma C

- Per compilare un semplice programma C:
  `gcc -o nomeeseguibile nomesorgente`
- Per eseguirlo basta dare al prompt il nome dell'eseguibile ...

- ... ma con `./` davanti perchè la directory corrente di norma non è nel path

- **Provare a compilare ed eseguire i propri vari hello nella directory `esempio` (`hello.c` già presente, varianti da creare erano suggerite per esercitarsi con `vimtutor`)**

Un processo (in loop infinito)

- Per fare pratica con alcuna operazioni relative ai processi, usiamo un programma C che esegue un loop infinito (`loop.c` nella directory `loop`)
- Per non caricare troppo il processore è inserita una chiamata a sleep nel corpo del loop:

  - **modificare il parametro di `sleep` da `0.001` a `1`**

  - **salvare, compilare `loop.c` ed avviare l'eseguibile risultante**
- A questo punto perdiamo il prompt ... ma abbiamo due possibilità:
  digitare `CTRL-C` per terminare il processo oppure
  digitare `CTRL-Z` per porre il processo in stato Stopped
- Per avviare un processo in background (viene eseguito senza occupare il prompt) digitare `&` alla fine della linea di comando
- Per avviare un processo a bassa priorità: `nice`
- **Quindi digitiamo: `./loop &` oppure `nice ./loop &`**

Elencare i processi: comando ps

- `ps`: visualizza lista di processi. Ha molte possibili opzioni e loro varianti storiche che si mescolano in modo confuso.
  L'output è composto da varie colonne ciascuna con un'intestazione "autoesplicativa". Si possono selezionare le colonne da visualizzare (tra le tante possibili) con l'opzione -o
- Alcuni esempi:
  - `ps` : visualizza i processi relativi alla sessione di lavoro corrente su 4 colonne (PID, TTY, TIME, CMD).
  - `ps -e` : visualizza tutti i processi esistenti (sempre con formato 4 colonne)
  - `ps u -e` : visualizza tutti i processi esistenti con formato esteso
  - `ps aux` : equivalente a `ps u -e`
  - `ps -u nomeutente` : visualizza tutti i processi di un utente
  - `ps u -u nomeutente` : visualizza tutti i processi di un utente con formato esteso

Esempi d’uso di ps

- Una volta avviato e stoppato (con CTRL-Z) il processo loop:

  - `ps` mostra il processo interprete dei comandi bash, il processo loop e ps stesso

  - `ps u` mostra informazioni aggiuntive, p.e. tasso di utilizzo CPU e memoria, occupazione di memoria in pagine (RSS = Resident Set Size, VSZ = Virtual Memory Size), stato del processo (R=Running, S=Sleeping, T=sTopped, Z=Zombie,...), start time, ... purtroppo `man ps` non documenta bene tutto ciò

  - `ps -u studente` compare un processo in più di proprietà di studente (quello che gestisce la connessione SSH)

Elencare i processi: comando top

- Il comando `top` presenta un insieme di informazioni statistiche globali sul sistema e una lista di processi con informazioni disposte su colonne con intestazioni “autoesplicative”
- `top` aggiorna dinamicamente il suo output
- `top` accetta comandi interattivamente, i più semplici sono:
  `h` per avere un help
  `spazio` per aggiornare immediatamente i dati visualizzati
  `s` per stabilire l’intervallo di tempo tra due aggiornamenti
  `O` per scegliere il criterio di ordinamento tra vari possibili
  `R` per invertire il senso di ordinamento (crescente/decrescente)
- Comandi di ordinamento veloci
  - `N` sort tasks by pid (numerically)
  - `P` sort tasks by CPU usage (default)
  - `M` sort tasks by resident memory usage
  - `T` sort tasks by time / cumulative time.
- `q` per uscire

Comando top

- Il `man` di `top` è un po' più informativo di quello di `ps`
- **Cercare nel `man` di `top` il significato delle colonne PR, NI, SHR**
- **Verificare quanti Page Fault hanno avuto i processi e quanto di loro è swappato**
- Soddisfare altre curiosità ...

Elencare i processi: comando pstree

- `pstree` presenta in formato “ASCII art” l’albero genealogico di tutti i processi del sistema
- E’ possibile specificare un singolo utente del quale visualizzare il sistema di processi
- Con l’opzione `-p` viene visualizzato anche il PID
- Con l’opzione `-h` si evidenzia la genealogia del processo corrente e con `-H` quella di un certo processo (identificato tramite PID)
- Con l’opzione `-u` si visualizzano i cambi di proprietario nella genealogia

Terminare processi

- Il comando `kill PID` si usa tipicamente per imporre la terminazione di un processo con un certo PID
- In generale `kill` serve per inviare un segnale di un certo tipo (non necessariamente letale) ad un processo
- Non sempre un semplice `kill` è sufficiente a terminare un processo: `kill -9 PID` invia un segnale di terminazione più "convincente"
- `killall nomeeseguibile` si usa per terminare tutti i processi che sono stati lanciati con un certo nome di eseguibile
- Anche `killall` in generale invia un segnale ad un gruppo di processi ed è più letale con l'opzione `-9`

- **Killare tutti i processi loop presenti**

Processi duraturi: il comando nohup

- I processi avviati in un sessione di lavoro (anche in background) vengono terminati quando si esce dalla sessione interattiva
- Per avviare un processo che sopravvive alla sessione interattiva usare: `nohup comando &`
- Il processo avviato proseguirà anche dopo il logout dell'utente, fino alla sua terminazione naturale o forzata
- Ogni eventuale output diretto a schermo sarà salvato in un file denominato `nohup.out`

Operazioni remote

- Il comando `ssh user@hostname` permette di aprire una sessione di lavoro protetta con cifratura su un host remoto
- Si usa l'opzione `-p` per specificare la porta da usare (default 22)

- Il comando `scp` permette di eseguire un `cp` in modalità remota e sicura specificando per uno o entrambi i file anche il nome utente e il nome host remoto (`-P` per specificare la porta da usare)
- **Creare un file di testo denominato `vm-NN.txt` sulla propria macchina e trasferirlo sulla macchina `soa-70` collegandosi come utente `studente`**

Due domande tipiche

- Chi c'è in casa ?
- Il comando `who` elenca gli utenti attualmente collegati alla macchina
- La tal macchina è accesa e raggiungibile ?
- Il comando `ping` causa l'invio di pacchetti "sonda" ad una macchina chiedendo risposta.
  Se la macchina è attiva, raggiungibile e accetta la richiesta, giungono pacchetti di risposta e vengono presentate indicazioni sul tempo di andata e ritorno
- **Pingare la `soa-70` (o un'altra vm), www.unibs.it e www.google.it: verificare la differenza dei tempi di andata e ritorno**
- **Pingare www.microsoft.com e www.linux.org**

Combinare i comandi

- Ogni processo è associato a tre flussi di I/O standard:
  lo standard input dal quale riceve gli ingressi (normalmente la tastiera)
  lo standard output sul quale produce le uscite “regolari” (normalmente il video)
  lo standard error sul quale produce i messaggi di “errore” (normalmente il video)
- E’ possibile combinare i programmi tra di loro collegando lo standard output di uno di essi allo standard input di un altro
- Questo tipo di utilizzo giustifica alcuni comportamenti di default (leggere ingressi da standard input, produrre uscite su standard output) che non avrebbero alcun senso pratico se riferiti a tastiera e video

Pipeline di comandi

- Il carattere | posto tra due comandi serve a connettere lo standard output del primo con lo standard input del secondo
- Esempi:

1) estrarre dalla lista di tutti i processi attivi quelli con un certo nome di eseguibile
2) visualizzare il nome del solo file più grosso in una dir
3) visualizzare il nome del solo file più piccolo in una dir
4) visualizzare e scorrere un file (/etc/passwd) in ordine alfabetico
5) farsi restituire il numero di utenti (numero righe di /etc/passwd)
6) farsi restituire il numero di utenti che usano bash come shell

Ridirezione su/da file

- Un comando seguito da `> nomefile` salva il suo standard output dentro `nomefile` (che viene sovrascritto)
  Esempio: produrre una versione ordinata di /etc/passwd nella propria directory
- Un comando seguito da `>> nomefile` salva il suo standard output in append in coda a `nomefile`
- Un comando seguito da `< nomefile` legge il suo standard input da file
  Esempio: mandarsi un e-mail con contenuto preso da un certo file (è solo un esempio, non funziona)
- Si possono combinare le due ridirezioni:
  `comando <nomefilein >nomefileout`

Ridirezioni evolute

- Il numero `0` indica lo standard input, il numero `1` lo standard output e il numero `2` lo standard error, `&num` indica il dispositivo al quale è associato in un certo momento un dato flusso, quindi:

  `comando 2>nomefile`
  ridirige i messaggi di errore di `comando` sul file `nomefile`

  `comando 2>&1`
  ridirige standard error su standard output (utile se entrambi devono andare a loro volta in una pipe)

  `comando > nomefile 2>&1`
  ridirige sia standard output sia i messaggi di errore di `comando` sul file `nomefile` (forma breve: `comando &>nomefile`)
- Se si vuole ridirigere "nel vuoto" mandare a `/dev/null`

Comando tee

- Il comando tee sdoppia un flusso in due copie: una va allo standard output una viene salvata su file

```
ls -al  | tee listato.txt
```

- Il contenuto della directory corrente viene mostrato a video e anche salvato sul file `listato.txt`
- L'opzione `-a` di `tee` serve per operare in append sul file specificato anzichè sovrascriverlo

La shell

- La shell svolge il ruolo di interprete dei comandi e può essere utilizzata sia in modo interattivo sia tramite programmi (script)
- Esiste una notevole varietà di shell, con molti aspetti comuni ma anche differenze non trascurabili: noi useremo la shell `bash` (riferimento http://tldp.org/LDP/abs/html/index.html)
- Passi principali di interpretazione di una linea di comando:
  - separazione della linea in parole (riconoscimento separatori, se non protetti da quoting, eliminazione spazi duplicati)
  - sostituzione di eventuali alias
  - applicazione di espansioni
  - ridirezione di input e/o output
  - esecuzione
  - attesa del risultato (exit status)

Quoting

- Il carattere backslash \ serve a impedire l'interpretazione di un carattere speciale (ce ne sono vari, p.e. $) o a specificare (come solito) i caratteri non stampabili (`\n` = newline, `\t` = tab, ...)

- `' '` il quoting forte impedisce l'interpretazione di tutto ciò che è incluso tra gli apici

- `" "` il quoting debole impedisce l'interpretazione di ciò che è incluso tra le virgolette tranne i caratteri speciali `$, ` `, \`
- C'e' una sintassi particolare `$'QUALCOSA'` per i caratteri non stampabili, p.e. `$'\t'` dà luogo al carattere tab (vedi `man bash`)
- Usi non semplici possono condurre a rompicapo: perchè non è possibile mettere un apice all'interno di un quoting forte ma è possibile mettere le virgolette all'interno di in un quoting debole ?

Espansioni

- brace expansion: lista di varianti tra parentesi graffe
  `echo a{d,c,b}e` produce in output
  `ade ace abe`
- tilde expansion: salvo usi particolari, il carattere ~ viene espanso nel nome della home directory
- command substitution: `$(comando)` o `` `comando` `` queste espressioni vengono sostituite dall'output di comando
- arithmetic expansion: `$((espressione_aritmetica))` viene sostituito dal valore dell'espressione
- word splitting: dopo le espansioni di cui sopra si ripete la separazione in parole dei risultati che contengano separatori
- filename expansion: alcuni caratteri speciali sono usati per creare espressioni sintetiche riferite a più nomi di file

Filename expansion

- `*` indica qualunque stringa (inclusa quella nulla):
  - `rm -rf *` rimuove TUTTO nella dir corrente
  - `vi *.txt` apre con vi tutti i file .txt
  - `mv tesi* ..` sposta nella directory superiore tutti i file il cui nome inizia con tesi
- `?` indica qualunque singolo carattere
  - `rm ?.dat` rimuove p.e. a.dat, Z.dat, 1.dat, ...
- `[ ]` le parentesi quadre con un insieme di caratteri indicano uno qualunque dei caratteri dell'insieme (usi anche esotici)
  - `[aeiou]*` qualunque file il cui nome inizia per vocale minuscola
  - `*[A-Z]` qualunque file il cui nome finisce per lettera maiuscola

Istruzioni per lo sviluppo degli esercizi

- Creare una directory `esercizi` dentro `/home/studente`
- Ogni esercizio deve essere salvato dentro una sottodirectory di `/home/studente/esercizi` con nome opportuno, che sarà indicato per ogni esercizio
- Ogni esercizio deve contenere un commento con i nomi degli autori
- Il nome della prima sottodirectory dove inserire il primo esercizio è: `/home/studente/esercizi/hello`

Hello world

- Un primo script originale:

```
#!/bin/bash
#Autori:P.Baroni,N.Gatta,A.Mauro,PMartinelli
echo Hello World
```

- La prima riga serve a far identificare il file come script della shell bash, e si può usare per specificare un qualunque altro interprete
- In generale il carattere `#` indica un commento fino a fine riga
- Inserire sempre una riga di commento con i nomi degli autori in ogni programma sviluppato !
- Scrivere con un editor il primo script (nella directory `hello`) ed eseguirlo

Primi passi

- Script di più righe: i comandi non devono terminare con ; a meno di metterli su una stessa riga
- Di solito si inserisce un comando per ogni riga
- Esercizio: Write a script that upon invocation shows the time and date, lists all logged-in users, and gives the system uptime. The script then saves this information to a logfile.
- Salvare questo esercizio nella directory `tempo`

Script di shell: le variabili

- L’uso delle variabili negli script di shell è “spregiudicato”: le variabili non hanno tipo e per creare una variabile basta eseguire un assegnamento:

  ```
  nomevar=valore
  ```

  N.B. NON CI DEVONO ESSERE SPAZI PRIMA O DOPO =

- Per inserire da tastiera il valore di una variabile anziche' assegnarla da programma basta fare:

  ```
  read nomevar
  ```

Script di shell: le variabili

- Le variabili così definite si intendono globali
- Il valore è per default una stringa (eventualmente risultato di espressioni, espansioni, etc.)
- Se la stringa contiene solo caratteri numerici la shell la può trattare come numero intero
- In generale la shell non è adatta ad elaborazioni numeriche se non estremamente elementari

Script di shell: le variabili

- Per riferire il valore di una variabile si deve scrivere: `$nomevar`
- Si può anche scrivere `${nomevar}qualcosa` p.e. se serve indicare il valore di `nomevar` seguito da `qualcosa` senza spazi intermedi
- I puristi suggeriscono di scrivere i riferimenti a valori di variabili sempre tra `" "` (quoting debole) per proteggere eventuali caratteri speciali nel nome e nel valore della variabile nonché per evitare errori dovuti a variabili aventi come valore la stringa vuota

Script di shell: variabili speciali

- $0, $1, $2, ... rappresentano i valori degli argomenti sulla linea di comando dello script: $0 è il nome dello script stesso, $1 il primo parametro della linea di comando, etc.
- $* indica l'intero insieme di parametri (a partire da 1)
- $# indica il numero di parametri (escluso lo 0)
- $? indica l'exit status dell'ultimo comando eseguito (si puo' provarlo da tastiera)
- Esiste un set di variabili predefinite di uso vario (http://tldp.org/LDP/abs/html/internalvariables.html)
- Ogni shell può inoltre "ereditare" delle variabili d'ambiente dalla shell genitrice

Operazioni su stringhe

- In generale l'assegnazione di un valore ad una variabile è considerata un'operazione su una stringa
- La concatenazione tra stringhe si ottiene semplicemente per affiancamento delle stesse:

```
NOME1=pluto
NOME2=pippo
NOME3=$NOME1$NOME2
echo $NOME3                  cosa apparirà ?
echo ${NOME3}paperino        cosa apparirà ?
echo {$NOME3}paperino        cosa apparirà ?
echo $NOME3paperino          cosa apparirà ?
NOME3=$NOME1 $NOME2          cosa apparirà ?
NOME3="$NOME1 $NOME2"        cosa apparirà ?
NOME3='$NOME1 $NOME2'        cosa apparirà ?
```

Operazioni sui numeri

- Che valore ha NUM dopo l'assegnazione seguente ?
  `NUM=4+3`

- Per trattare espressioni numeriche:
  `let NUM=4+3` (occhio agli spazi, ora vale 7)

  `let NUM="4 + 4"` (il quoting debole protegge gli spazi, ora vale 8)

  `NUM=\`expr 4 + 5\`` (in alternativa il comando expr valuta espressioni numeriche, in questo caso gli spazi servono !)

  `NUM=$(expr 4 + 5)` (forma equivalente)

"Compiti" a casa

- Tenersi allenati con la lettura di man e l'uso di file di testo
- Produrre un file di testo di nome `DaCercare` che contiene alcune righe ognuna contenente un numero
- Capire come usare `grep` per trovare all'interno di tutti i file creati nella directory `testi` (compito della lezione precedente) tutte le righe che contengono uno dei numeri presenti nel file `DaCercare`
- Identificare l'ulteriore opzione da usare per fare quanto sopra richiesto senza mostrare il nome del file che contiene ciascuna riga trovata

"Compiti" a casa

- Sono proposti due esercizi per chi vuole fare un po' di pratica in autonomia con la programmazione di shell
- Nel primo sono indicate variante che prevedono l'uso del costrutto if (introdotto nei lucidi della lezione successiva)
- Il secondo richiede necessariamente l'uso del costrutto if
- Provare a portarsi avanti sui lucidi della lezione successiva non può far male

Verifica numero righe di un file

- Scrivere uno script di shell denominato `verificarighe.sh` che inserisce in una variabile il numero di righe di un file di testo prefissato (per esempio `istruzioni.txt`) e mostra all'utente tale numero
- VARIANTE CON IF: informa l'utente se il numero è maggiore, minore o uguale ad una soglia prefissata (per esempio `10`)

- Modificare quindi il programma in modo che sia il nome del file sia il valore della soglia siano inseribili interattivamente dall'utente.
- VARIANTE CON IF: si controlli se il file è un "regular file" e se è leggibile prima di procedere, dando un messaggio di errore in caso contrario

Verifica presenza stringa

- Scrivere uno script di shell denominato `verificastringa.sh` che deve ricevere all'utente come argomenti della linea di comando due stringhe di caratteri `STR1` e `STR2` e si comporta come segue:

  1) se `STR1` è il nome di un "regular file" esistente e leggibile procede ai punti successivi altrimenti dà un messaggio di errore ed esce

  2) verifica se la stringa `STR2` compare nel file di nome `STR1`

  3) di conseguenza scrive a video un messaggio che indica se la stringa e' presente oppure no. In caso affermativo scrive anche il numero di righe di `STR1` in cui compare `STR2`