Sistemi Operativi

Enrico Favretto

A.A. 2022/2023

Indice

1.1 Lab-01

Terminale: è l'ambiente testuale di interazione con il sistema operativo. Tipicamente usato come applicazione all'interno dell'ambiente grafico ed è possibile avviarne più istanze, è anche disponibile all'avvio.

Shell: All'interno del terminale, l'interazione avviene utilizzando una applicazione specifica in esecuzione al suo interno, comunemente detta Shell. Essa propone un prompt per l'immissione diretta dei comandi da tastiera e fornisce un feedback testuale, è anche possibile eseguire sequenze di comandi preorganizzate contenute in file testuali, chiamati *script* o *batch*. A seconda della modalità alcuni comandi possono avere senso o meno o comportarsi in modo particolare.
L'insieme dei comandi e delle regole di composizione costituisce un linguaggio di programmazione orientato allo scripting.

Bash: Esistono molte shell. La shell **Bash** è una di quelle più utilizzate e molte sono comunque sono molto simili tra di loro, ma hanno sempre qualche differenza.
Tipicamente - almeno in sessioni non grafiche - al login un utent ha associata una shell particolare

POSIX: **P**ortable **O**perating **S**ystem **I**nterface for Uni**x**: è una famiglia di standard IEEE.
Nel caso delle shell in particolare definisce una serie di regole e comportamenti che possono favorire la probabilità
La shell bash soddisfa molti requisiti POSIX, ma presenta alcune differenze ed "estensioni" per agevolare la programmazione.

Comandi Interattivi: La shell attende un input dall'utente e al completamento, di solito premendo INVIO, lo elabora.
Per indicare lattesa mostra allutente un PROMPT (pu essere modificato).
Fondamentalmente si individuano 3 canali:

- Standard input (tipicamente la tastiera), canale 0, detto *stdin*
- Standard output (tipicamente il video), canale 1, detto *stdout*
- Standard error (tipicamente il video), canale 2, detto *stderr*

Struttura generale comandi: Solitamente un comando è identificato da una parola chiave cui possono seguire uno o pi "argomenti" opzionali o obbligatori, accompagnati da un valore di riferimento o meno (in questo caso hanno valore di "flag") e di tipo posizionale o nominale. A volte sono ripetibili.

ls -alh /tmp

gli argomenti nominali sono indicati con un trattino cui segue una voce (stringa alfanumerica) e talvolta presentano una doppia modalità di riferimento: breve (tipicamente voce di un singolo carattere) e lunga (tipicamente un termine mnemonico)

ls -h / ls -help

Termini:

- l'esecuzione dei comandi avviene "per riga" (in modo diretto quella che si immette fino a INVIO)
- un "termine" (istruzione, argomento, opzione, etc.) solitamente una stringa alfanumerica senza spazi
- spaziature multiple sono spesso valide come singole e non sono significative se non per separare i termini
- è possibile solitamente usare gli apici singoli/doppi per forzare una sequenza come singolo termine
- gli spazi iniziali e finali di una riga collassano
- le righe vuote sono ignorate

Commenti: È anche possibile utilizzare dei commenti da passare alla shell. L'unico modo formale è l'utilizzo del carattere # per cui esso e tutto ciò che segue fino al termine della riga viene ignorato

ls -la # this part is a comment

Comandi fondamentali: I comandi possono essere "builtin" (funzionalità intrinseche dell'applicazione shell utilizzata) o "esterni" (applicazioni eseguibili che risiedono su disco).

- clear (*clearing the terminal screen in Linux*)
- pwd (*prints the current working directory path, starting from the root(/)*)
- ls (*list files or directories*)
- cd (*list the contents of the current directory*)
- wc (*creates a new file in your current working directory with the name wc with the output of your ls command*)
- date (*displays and sets the system date and time*)
- cat (*allows us to create single or multiple files, view content of a file, concatenate files and redirect output in terminal or files*)
- echo (*displaying lines of text or string which are passed as arguments on the command line*)

- alias/unalias (*is a way to make a complicated command or set of commands simple/remove the definition for each alias name specified*)
- test (*used as part of the conditional execution of shell commands*)
- read (*attempts to read up to count bytes from file descriptor fd into the buffer starting at buf*)
- file (*tells you the type of a file*)
- chown (*changes the user and/or group ownership of each given file*)
- chmod (*used to manage file system access permissions on Unix and Unix-like systems*)
- cp/mv (*copy a file/directory into another directory/moves a file/directory into another directory*)
- help (*listing all possible commands that are pre-installed in Ubuntu*)
- type (*used to describe how its argument would be translated if used as commands*)
- grep (*allows you to find a string in a file or stream*)
- function (*a reusable block of code*)

Canali Input/Output
Ogni comando lavora su un insieme di canali, come standar output o un file descriptor

Ridirizionamento di base
I canali possono anche essere ridirezionati, ad esempio

- ls 1 >/tmp/out.txt 2 >/tmp/err.txt
- ls nonExistingItem 1 > */tmp/err.txt* 2 > &1
- <: $command < file$ invia l'input al comando (file.txt read-only)
  mail -s "subject" *rcpt* < *context.txt* (anzichè iterattivo)
- <>: come sopra solo che il file.txt è aperto in read-write
- $source > target : command1 > out.txt2 > err.txt$
  ridireziona source su target (source può essere sottointeso, target può essere un canale)
- > |: si comporta come > ma forza la sovrascrittura anche se bloccata nelle configurazioni
- >>: si comporta come > ma opra un append se la destinazione esiste

- <<: here-document, consente all'utente di specificare un terminatore testuale, dopodiche accetta l'input fino alla ricezione di tale terminatore
- <<<: here-string, consente di fornire input in maniera non iterattiva

Esistono molte varianti e possibilità di combinazione dei vari operatori
Un caso molto usato è la soppressione dell'output (utile per gestire solo side-effects, come il codice di ritorno), esempio:
$typecommand 1 > dev/null 2 > \&1$ (per sapere se command esiste)

Ambiente e variabili

- La shell può utilizzare variabili per memorizzare e recuperare i valori
- I valori sono generalmente trattati come stringhe o interi: sono presenti anche semplici array
- Il formato del nome è del tipo: $\hat{}[\_[:alpha:]][\_[:alnum:]]\$$
- Per set (impostare) / get (accedere) al valore di una variabile:
  - Il set si effettua con $[export]variabile = valore$, lo scope è generalmente quello del processo attuale: anteponendo export si rende disponibile anche ai processi figli
  - Il get si effettua con $\$variabile$ o $\${variabile}$
- La shell opera in un ambiente in cui ci sono alcuni riferimenti impostabili e utilizzabili attraverso l'uso delle cosiddette "variabili d'ambiente" con cui si intendono generalmente quelle con un significato particolare per la shell stessa, tra le variabili d'ambiente più comuni vi sono:

  $$SHELL, PATH, TERM, PWD, PS1, HOME$$

  Essendo variabili si impostano e usano come le altre

Variabili di sistema
Alcune variabili sono impostate e/o utilizzate direttamente dal sistema in casi particolari.

- SHELL: contiene il riferimento alla shell corrente (path completo)
- PATH: contiente i percorsi in ordine di priorità in cui sono cercati i comandi (separati da :)
- TERM: contiene il tipo di terminale corrente
- PWD: contiene la cartella corrente
- PS1: contiene il prompt e si possono usare marcatori speciali
- HOME: contiene la cartella principale all'utente corrente

Esecuzione comandi e $PATH
Quando si immette un comando (o una sequenza di comandi) la shell analizza quanto inserito (parsing) e individua le parti che hanno la posizione di comandi da eseguire: se sono interni ne esegue le funzionalit direttamente altrimenti cerca di individuare un corrispondente file eseguibile: questo normalmente cercato nel file-system solo e soltanto nei percorsi definiti dalla variabile PATH a meno che non sia specificato un percorso (relativo o assoluto) nel qual caso viene utilizzato esso direttamente. Dall'ultima osservazione discende che per un'azione abbastanza comune come lanciare un file eseguibile (non installato) nella cartella corrente occorre qualcosa come: ./nomefile

Array

- Definizione: lista=("a" 1 "b" 2 "c" 3) separati da spazi
- Output completo: {$*lista*[@]}
- Accesso singolo: {$*lista*[*x*]}
- Lista indici: {$!*lista*[@]}
- Dimensione: {#*lista*[@]}
- Set elemento: $lista[x] = value$
- Append: $lista+ = (value)$
- Sub-array: $*lista*[@] : *s* : *n*$, dall'indice s di lunghezza n

Variabili $$ e $?
Le variabili $$ e $? non possono essere impostate manualmente.

- $$: contiene il PID del processo attuale
- $?: contiene il codice di ritorno dell'ultimo comando eseguito

Esecuzione comandi e Parsing
La riga dei comandi è elaborata con una serie di azioni "in sequenza" e poi rielaborata più volte, tra le azioni ci sono:

- Sostituzioni speciali della shell
- Sostituzione variabili
- Elaborazione subshell

sono svolte con un ordine di priorità e poi l'intera riga è rielaborata

Concatenazioni Comandi
È possibile concatenare più comandi in un'unica riga in vari modi con effetti differenti

- comand1 ; comand2: concatenazione semplice, esecuzione in sequenza
- comand1 && comand2: concatenazione logica (AND), l'esecuzione procede solo se il comando precedente non fallisce (**codice ritorno 0**)
- comand1 —— comand2: concatenazione logica (OR), l'esecuzione procede solo se il comando precedente non fallisce (**codice riotrno NON 0**)
- comand1 — comand2 : concatenazione con piping

Operatori di piping (pipe): — e —&
La concatenazione con gli operatori di piping cattura l'output di un comando e lo passa in input al successivo

- ls — wc -l: cattura solo stdout
- ls —& wc -l: cattura stdout e stderr

Nota: il comando ls ha un comportamento atipico: il suo output di base è differente a seconda che il comando sia diretto al terminale o a un piping

Subshell
È possibile avviare una subshell, ossia un sotto-ambiente in vari modi, in particolare raggruppando i comandi tra parentesi tonde: (... comandi ...)
Spesso si usa il catturare l'output standard della sequenza che viene sostituito letteralmente e rielaborato, ci sono due modi per farlo: $ (... comandi ...) oppure`*comandi*`

Esempio

```
echo "/tmp" > /tmp/tmp.txt ; ls $(cat /tmp/tmp.txt)
```

i comandi sono eseguiti rispettando la sequenza:

1. `echo "/tmp" > /tmp/temp.txt`: crea un file temporaneo con /tmp
2. `ls $(cat /tmp/tmp.txt)`: è prima eseguita la subshell
   - `cat /tmp/tmp.txt` che genera in stdout `"/tmp"` e poi con sostituzione:
   - `ls /tmp` mostra il contenuto della cartella /tmp

Espansione Aritmetica
La sintassi base per una subshell è da non confondere con l'espansione aritmetica che utilizza le doppie parentesi rotonde, all'interno delle doppie parentesi rotonde si possono rappresentare varie espressioni matematiche inclusi assegnamenti e confronti

```
((a=7))((a++))((a<10))((a = 3>10?1:0))
```

```
b = $((c+a))
```

Confronti Logici
I costrutti fondamentali per i confronti logici sono il comando `test ...` e i ragruppamenti tra parentesi quadre singole e doppie: `[ ... ]`, `[[ ... ]]` Dove `test ...` e `[ ... ]` sono builtin equivalenti mentre `[[ ... ]]` è una coppia di **shell-keywords**
In tutti i casi il blocco di confronto genera il codice in uscita 0 in caso di successo, un valore differente altrimenti

- Built-in: sono sostanzialmente dei comandi il cui corpo dell'esecuzione è incluso nell'applicazione shell direttamente (non eseguibili esterni) e quindi seguono sostanzialmente le regole generali dei comandi
- Shell-keywords: sono gestite come marcatori speciali così che possono "attivare" regole particolari di parsing.

Un esempio di questo sono gli operatori `< e >` che normalmente valgono come ridirezionamento, ma all'interno di `[[ ... ]]` valgono come operatori relazionali
Tipologia operatori
Le parentesi quadre singole sono POSIX-compilant, mentre le doppie sono un'estensione bash. Nel primo caso gli operatori relazionali tradizionali $(>, <, ...)$ non possono essere usati perchè hanno un altro significato, salvo eventualmente l'utilizzo nelle doppie parentesi quadre, e quindi se ne usano di specifici che hanno però un equivalente più tradizionale nel secondo caso
Gli operatori e le sintassi variano a seconda del tipo di informazioni utilizzate: c'è una distinzione sottile tra confronti per stringhe e per interi. **Confronti tra Interi e Stringhe**

Interi	[...]	[[...]]
$uguale - a$	$-eq$	$==$
$diverso - da$	$-ne$	$!=$
$minore/minore - uguale$	$-lt/-le$	$</<=$
$maggiore/maggiore - uguale$	$-gt/-ge$	$>/>=$

Stringhe	[...]	[[...]]
$uguale - a$	$= o ==$	
$diverso - da$	$!=$	
$minore/minore - uguale$	$/ <$	$<$
$maggiore/maggiore - uguale$	$/ >$	$>$

Per le strighe nell'uguaglianza bisogna lasciare uno spazio prima e dopo
Confronti tra Operatori Unari Esistono alcuni operatori unari ad esempio per verificare se una stringa è vouta, oppure per controllare l'esistenza di file in una cartella

- [[`-f /tmp/prova`]]: è un file?

- [[`-e /tmp/prova`]]: file esiste?
- [[`-d /tmp/prova`]]: è una cartella?

N.B.: è possibile utilizzare sia [e sia [[
Negazione Il carattere ! viene usato per negare i confronti

Script/Bash
È possibile raccogliere sequenze di comandi in un file di testo che può poi essere eseguito:

- Richiamando il tool "bash" e passando il file come argomento

```
bash ../file.sh
```

- Impostando il bit "x" e specificando il percorso completo, o solo il nome se la cartella è in $PATH

```
chmod +x ../file.sh && ../file.sh
```

Esempio:

- **Script**: "bashpid.sh"

```
echo $BASHPID
echo $( $BASHPID)
```

- **CLI**: `chmod +x ./bashpid.sh; echo $BASHPID; ./bashpid.sh`

Elementi particolari
Le righe vuote e i commenti (#) sono ignorati
La prima riga può essere un **metacommento**, detto hash-bang, she-bang, che identifica un'applicazione a cui passare il file stesso come argomento (tipicamete usato per identificare l'interprete utilizzato)
Sono disponibili anche variabili speciali:

- $: lista completa degli argomenti passati allo Script
- $#: numero di argomenti passati allo script
- $i: i-esimo argomento con i da 0 a n

Costrutti e Funzioni
Esistono costrutti come for-loop, if-else, while-loop che possono anche essere scritti su singola riga.
If-Else:

```
if [ $1 -lt 10 ]; then
     echo less than 10
elif [ $1 -ge 20 ]; then
     echo greater than 20
else
     echo between 10 and 20
fi
```

For-loop:

```
for i in ${!lista[@]}; do
     echo ${lista[$i]}
done
```

While-loop:

```
while [[ $i < 10 ]]; do
     echo $i; (( i++ ))
done
```

Funzioni

- `function_name () {`
  `local var1 = 'C' #variabile con scope locale`
  `echo $1 $2 #argomenti normali della funzione`
  `return 44 #codice di ritorno`
  `}`
- `function function_name () {commands;}`
- `function_name arg1 arg2 #chiamata della funzione con due argomenti`

Esercizi

1. Stampare T o F se il valore di input rappresenta un file o una cartella esistente.

   ```
   [-e $DATA] && echo "T" || echo "F"
   ```

2. Stampare "file", "cartella" o "?" a seconda di cosa l'input rappresenta nel fileStystem.

   ```
   [-f $DATA] && echo "file" || ([-d $DATA] && echo "cartella"
                              || echo "?")
   ```

   Concatenazione di if uno dentro l'altro

3. Stampare il risultato di una semplice operazione aritmetica contenuta nel file rappresentato dal valore di input, oppure "?" se non esiste

```
[-f $DATA] && echo $(( $(cat $DATA) )) || echo "?"
```

Controllo se input è un file e computo l'operazione all'interno di esso, tramite (( )), altrimenti ritorna il ?

4. Scrivere uno script che dato un qualunque numero di argomenti li restituisca in ordine inverso.

5. Scrivere uno script che mostri il contenuto della cartella corrente in ordine inverso rispetto all'output generato da "ls", per semplicità assumere l'assenza di spazi

N.B.: Se eseguiti all'interno di uno script, la variabile DATA deve essere esportata con:

```
export DATA = valore
```

L'utilizzo di BASH, tramite CLI o Script, è basilare per poter interagire attraverso comandi con il file-system, con le risorse di sistema e per poter invocare tools e applicazioni.

1.2 Lab-02

Docker: Tecnologia di virtualizzazione a livello del sistema Operativo che consente la creazione, gestione e esecuzione di applicazioni attraverso containers
Containers: sono ambieni leggeri, dinamici ed isolati che vengono eseguiti sopra il kernel Linux

Docker Container	**Virtual Machine**
◦ *Virtualizzazione a livello OS*	• *Virtualizzazione a livello HW*
◦ *Containers condividono il kernel*	• *Ogni VM ha il suo OS*
◦ *Avvio e creazione in secondi*	• *Avvio e creazione in minuti*
◦*Leggere (KB/MB)*	• *Pesanti (GB)*
◦*Utilizzo leggero di risorse*	• *Utilizzo intenso di risorse*
◦*Si distruggono e si rieseguono*	• *Si trasferiscono*
◦*Minore sicurezza*	• *Maggiore sicurezza*
	• *Maggiore controllo*
◦*Basati su immagini (già pronte)*	

Compatibilità

- **Linux**: docker gestisce i containers usando il kernel linux nativo
- **Windows**: docker gestisce i containers usando il kernel linux tramite WSL2 (originariamente **virtualizzato** tramite Hyper-V), gestito da un applicazione
- **Mac**: docker gestisce i container usando il kernel linux virtualizzato tramite xhyve hypervysor, gestito da una applicazione

Immagine: Un'immagine docker è un insieme di istruzioni per la creazione di un container. Essa consente di ragruppare varie applicazioni ed eseguirle, con una certa configurazione, in maniera più rapida attraverso un container
Container: I containers sono invece gli ambienti virtualizzati gestiti da docker, che possono essere creati, avviati, fermati ed eliminati. I container sono **basati** su una immagine

Gestione dei containers

- `docker run [options] <image>`: crea un nuovo container da un'immagine
- `docker container ls [options]`: mostra i containers attivi (`[-a]` li mostra tutti)
- `docker exec [options] <container> <command>`: esegue il comando al'interno del container
- `docker stats`: mostra le statistiche di utilizzo dei containers

- `docker <command> --help`: aiuto su specifico comando

Parametri opzionali

- `--name <nome>`: assegna un nome specifico al container
- `-d`: detach mode, cioè scollega il container
- `-ti`: esegue il container in modalità interattiva, per collegarsi `docker attach <container>`, scollegarsi `Crtl-P,Crtl-Q`
- `--rm`: elimina container all'uscita
- `--hostname <nome>`: imposta l'hostname nel container
- `--workdir <path>`: imposta la cartella di lavoro nel container
- `--network host`: collega il container alla rete locale, la modalità host non funziona a causa della VM sottostante
- `--privileged`: esegue il container con i privilegi dell'host

Gestione delle immagini
È possibile creare nuove immagini delle docker

- `docker images`: mostra le immagini salvate
- `docker rmi <imageID>`: elimina immagine se non in uso
- `docker search <keyword>`: cerca un'mmagine nella repository di docker
- `docker commit <container> <repository/imageName>`: crea una nuova immagine dai cambiamenti nel container

Dockerfile
I dockerfile sono dei documenti testuali che raccolgono una serie di comandi necessari alla creazione di una nuova immagine. Ogni nuova immagine sarà generata a partire da un'immagine di base, come Ubuntu o l'immagine minimale 'scratch'. La creazione a partire da un docker file viene gestita attraverso del caching che ne permette la ricompilazione rapida in caso di piccoli cambiamenti.

Esempio:
```
FROM ubuntu:22.04
RUN apt-get update && apt-get install build-essential nano -y
RUN mkdir /home/labOS
CMD cd /home/labOS && bash
docker build -t /labos/ubuntu - < dockerfile
```

Gestione dei volumi
Docker salva i file persistenti su bind mount o su dei volumi. Sebbene i bind mount siano strettamente collegati con il filesystem dell'host OS, consentendo dunque una facile comunicazione con il containers, i volumi sono ormai lo standard in quanto indipendenti, facili da gestire e pi in linea con la filosofia di docker.

Sintassi comandi

- `docker volume create <volumeName>`: crea un nuovo volume
- `docker volume ls`: mostra i volumi esistenti
- `docker volume inspect <volumeName>`: esamina volume
- `docker volume rm <volumeName>`: rimuove il volume
- `docker run -v <volume>:<path/in/container> <image>`: crea un nuovo container con il volume specificato montato nel percorso specificato
- `docker run -v <pathHost>:<path/in/container> <image>`: crea un nuovo container con un bind mount specificato mostrato nel percorso specificato

GCC
GCC = Gnu Compiler Collection, insiemi di strumenti open-source che costituisce lo standard per la creazione di eseguibili Linux. Supporta diversi linguaggi e consente la modifica dei vari passaggi intermedi per una completa personalizzazione dell'eseguibile
Compilazione
Gli strumenti gcc possono essere chiamati singolarmente:

- `gcc -E <sorgente.c> -o <preProcessed.ii|i>`
- `gcc -S <preProcessed.ii|i> -o <assembly.asm|.s>`
- `gcc -c <assembly.asm|.s> -o <objectFile.obj|.o>`
- `gcc <objectFile.obj|.o> -o <executable.out>`

l'input di ogni comando può essere il file sorgente, e l'ultimo comando è in grado di creare direttamente l'eseguibile, inoltre l'assembly e il codice macchina generato dipendono dall'architettura di destinazione

Make
Make tool: è uno strumento della collezione GNU che può essere usato per gestire la compilazione automatica e selettiva di grandi e piccoli progetti. Consente di specificare delle dipendenze tra i vari file, per esempio consentendo solo la compilazione delle librerie in cui i sorgenti sono stati modificati

Può anche essere usato per gestire un deployment di una applicazione, assumendo alcune delle capacità di uno scipt bash
Makefile: Make può eseguire dei makefiles i quali contengono tutte le direttive utili alla compilazione di un'applicazione

```
make -f makefile
```

in alternativa, il comando make senza argomenti processerà il 'makefile' presente nella cartella di lavoro.
Makefiles secondari possono essere inclusi nel makefile principale con delle direttive specifiche, consentendo una gestione più articolata di grandi progetti

Target, prerequisite and recipes

- Una **ricetta** o **regola** una lista di comandi bash che vengono eseguiti indipendentemente dal resto del makefile.
- I **target** sono generalmente dei files generati da uno specifico insieme di regole. Ogni target può specificare dei prerequisiti.
- **Prerequisiti**: ovvero degli altri file che devono esistere affinchè le regole di un target vengano eseguite. Un prerequisito pu essere esso stesso un target!

L'esecuzione di un file make inizia specificando uno o pi target `make -f makefile target1 ...` e prosegue a seconda dei vari prerequisiti.

```
target:  prerequisite
     recipe/rule
     recipe/rule
     ...
```

Esempio:

```
target1:  target2 target3
     rule(3)
     rule(4)
     ...

target2:  target3
     rule(1)

target3:
     rule(2)
```

Sintassi
Un makefile è un file di testo plain in cui righe vuote e parti di testo dal il carattere # fino alla fine della riga non in una ricetta (considerato un commento)
Le ricette/rule devono iniziare con il carattere TAB
Una ricetta che inizia con TAB@ non viene visualizzata in output, altrimenti i comandi sono visualizzati e poi eseguiti, una riga con un singolo TAB è una ricetta vuota.

Esistono costrutti più complessi per necessità particolari.

Target Speciali
Se non viene passato alcun target viene eseguito quello di default: il primo disponibile. Esistono poi dei target con un significato e comportamento speciale

- `.INTERMEDIATE e .SECONDARY`: hanno come prerequisiti i target intermedi, nel primo caso sono poi rimossi, nel secondo sono mantenuti a fine esecuzione
- `.PHONY`: ha come prerequisiti i target che non corrispondono a dei files, o comunque da eseguire "sempre" senza verificare l'eventuale file omonimo

In un target, il simbolo % sostituisce una qualunque stringa, in un prerequisito corrisponde alla stringa sostituita nel target
Esempio:

```
all:  ...
      rule

.SECONDARY: target1 ..

.PHONY: target2 ..

%.s :  %.c
      #prova.s:  prova.c
      #src/h.s:  src/h.c
```

Variabile utenti e automaitche

- **Utente**: Le variabili utente si definiscono con la sintassi `nome:=valore` oppure `nome=valore` e vengono usate con `$(nome)`, inoltre possono essere sovrascritte da riga di comando cone `make nome=value`
- **Automatiche**: Le variabili automatiche possono essere usate all'interno delle regole per riferirsi ad elementi specifici relativi al target corrente. La variabile SHELL può essere usata per specificare la shell di riferimento per l'esecuzione di makefile, esempio SHELL=/bin/bash abilita bash anzichè SH

Funzioni speciali

- `$(eval ...)`:consente di creare nuove regole make dinamiche
- `$(shell ...)`: cattura l'output di un comando shell
- `$(wildcard *)`: restituisce un elenco di file che corrispondono alla stringa specificata

Esempio:

```
LATER=hello
PWD=$(shell pwd)
OBJ_FILES:=$(wildcard *.o)

target1:
	echo $(LATER) #hello
	$(eval LATER+= world)
	echo $(LATER) #hello world
```

Esempio:

```
all:  main.out
	@echo "Application compiled"

%.s:  %.c
	gcc -S $< -o $@

%.out:  %.s
	mkdir -p build
	gcc $< -o build/$@

clean:
	rm -rf build *.out *.s

.PHONY: clean

.SECONDARY: make.s
```

Docker, GCC e make possono essere utilizzati per la gestione delle varie applicazioni in C. Ognuno di questi strumenti non è indispensabile ma permette di creare un flusso di lavoro coerente e strutturato

1.3 Lab-03

C Fundamentals

- **Tipi e Casting**

  C è un linguaggio debolmente e tipizzato che utilizza 8 tipi fondamentali. È possibile fare il casting tra tipi differenti:

  ```
  float a = 3.5;
  int b = (int)a
  ```

  La grandezza delle variabili è dipendente dall' architettura di riferimento i valori massimi per ogni tipo cambiano a seconda se la variabile è signed o unsigned.

$void = 0byte$	$float = 4bytes$
$char = 1byte$	$long = 8bytes$
$short = 2bytes$	$double = 8bytes$
$int = 4bytes$	$longdouble = 16bytes$

- **sizeof**: Si tratta di un operatore che elabora il tipo passato come argomento (tra parentesi) o quello dellespressione e restituisce il numero di bytes occupati in memoria

- **Puntatori**
  C si evolve attorno all'uso di puntatori, ovvero degli alias per zone di memorie condivise tra diverse variabili/funzioni. L'uso di puntatori è abilitato da due operatori: * ed &.

  - **Variabili**:
    : ha significati diversi a seconda se usato in una dichiarazione o in una assegnazione
    `int *pointer; // crea un puntatore ad intero`
    `int valore = *(pointer) // ottiene un valore puntato`
    &: ottiene l'indirizzo di memoria in cui è collocata una certa variabile.
    `long whereIsValore = &valore`
  - **Funzioni**: C consente anche di creare dei puntatori a delle funzioni: puntatori che possono contenere l'indirizzo di funzioni differenti
    `float (*punt)(float,float)`
    `ret_type (*ptnName)(argType, argType,....)`

- **main.c**

  A parte in casi particolari l'applicazione deve avere una funzione "main" che è utilizzata come punto di ingresso, il valore di ritorno è un int che rappresenta il codice di uscita dell'applicazione (0 se omesso).

Quando viene invocata riceve in input il numero di argomenti, `int argc`, in cui è incluso il nome dell'eseguibile e la lista degli argomenti, `char * argv[]`, come vettore di stringhe (vettore di vettori di caratteri)
Esempio:

```
gcc main.c -o main.o
./main.o arg1 arg2
```

In questo esempio il valore di argc = 3, e argv è della forma "0–./main.o; 1– arg1; 2– arg2"

- **printf/fprintf**

```
int printf(const char *format, ...)
int fprintf(FILE *stream, const char *format, ...)
```

Inviano dati sul canale stdout (printf) o su quello specificato (fprintf) secondo il formato indicato. Il formato una stringa contenente contenuti stampabili (testo, a capo,) ed eventuali segnaposto identificabili dal formato generale:

```
%[flags][width][.precision][length]specifier
```

Esempio: %d (intero con segno), %c (carattere), %s (stringa), ...
Ad ogni segnaposto deve corrispondere un ulteriore argomento del tipo corretto

- **Direttive**

Il compilatore, nella fase di preprocessing, elabora tutte le direttive presenti nel sorgente. Ogni direttiva viene introdotta con "#" e pu essere di vari tipi:

`#include <lib>`	*copia il contenuto del file lib nel file corrente*
`#include "lib"`	*come sopra ma cerca prima nella cartella corrente*
`#define VAR VAL`	*crea una costante VAR con il contenuto VAL e sostituisce ogni occorrenza di VAR con VAL*
`#define MUL(A,B) A*B`	*dichiara una funzione con parametri A e B, queste funzioni hanno una sintassi limitata*
`#ifdef, #ifndef, #if, #else, #endif`	*rende l'inclusione di parte di codice dipendente da una condizione*

Macro possono essere passate a GCC con -D NAME=VALUE
Esempio:

```
#include <stdio.h>
#define ITER 5
#define POW(A) A*A
```

```
int main(int argc, char **argv) {
#ifdef DEBUG
      printf("%d\n, argc");
      printf("%s\n", argv[0]);
#endif
      int res = 1;
      for (int i = 0; i < ITER; i++){
            res *= POW(argc);
      }
      return res;
}
gcc main.c -o main.out -D DEBUG=0
gcc main.c -o main.out -D DEBUG=1
./main.out 1 2 3 4
```
Danno lo stesso risultato

```
#include <stdlib.h>
#include <stdio.h>
#define DIVIDENDO 3
int division(int var1, int var2, int * result){
      *result = var1/var2;
      return 0;
}
int main(int argc, char * argv[]){
      float var1 = atof(argv[1]);
      float result = 0;
      division((int)var1,DIVIDENDO,(int *)&result);
      printf("%d \n",(int)result);
      return 0;
}
```

- **Librerie standard**:

  Librerie possono essere usate attraverso la direttiva #include. Tra le pi importanti vi sono:

  - stdio.h: FILE, EOF, stderr, stdout, stdin, ...
  - stdlib.h: atof(),atoi(),...
  - string.h: memset(), memcpy(), strncat(), ...
  - math.h: sqrt(), sin(), cos(),...
  - unistd.h: read(), write(), fork(), ...
  - fcntl.h: creat(), open(), ...

- **Struct e Unions**: Structs permettono di aggregare diverse variabili, mentre le unions permettono di creare dei tipi generici che possono ospitare uno di vari tipi specificati.

Esempio:

```
struct Books {
      char author[50];
      char title[50];
      int bookID;
} book1, book2;
union Result{
      int intero;
      float decimale;
} result1, result2;
```

- **Typedef**: consente la definizione di nuovi tipi di variabili o funzioni
  Esempio:

```
typedef unsigned int intero;
typedef struct Books{
...
} bookType;
intero var = 22; //= unsigned int var = 22;
bookType book1; //= struct Books book1;
```

- **Exit**: `void exit(int status)`
  Il processo è terminato restituendo il valore status come codice di uscita. Si ottiene lo stesso effetto se all'interno della funzione main si ha return status. La funzione non ha un valore di ritorno proprio perchè non sono eseguite ulteriori istruzioni dopo di essa.
  Il processo chiamante è informato della terminazione tramite un segnale apposito.

- **Vettori e Stringhe**

  - **Vettori**:
    I vettori sono sequenze di elementi omogenei (tipicamente liste di dati dello stesso tipo, ad esempio liste di interi o di caratteri), si realizzano con un puntatore al primo elemento della lista.
    Esempio: Con int arr[4] = {2, 0, 2, 1} si dichiara un vettore di 4 interi inizializzandolo: sono riservate 4 aree di memoria consecutive di dimensione pari a quella richiesta per ogni singolo intero (tipicamente 2 bytes, quindi 4*2=8 in tutto)
    Esempio: `char str[7] = {'c', 'i', 'a', 'o', 56,57,0}` : 7*1 = 7 bytes
    str è dunque un puntatore a char (al primo elemento) e si ha che:
    `str[n] corrisponde a *(str+n)`
    e in particolare
    `str[0] corrisponde a *(str+0)=*(str)=*str`
  - **Stringhe**:

Le stringhe in C sono vettori di caratteri, ossia puntatori a sequenze di bytes, la cui terminazione è definita dal valore convenzionale 0 (zero).
Un carattere tra apici singoli equivale all'intero del codice corrispondente.
In particolare un vettore di stringhe è un vettore di vettore di caratteri e dunque:

```
char c; carattere
char * str; vettore di a caratteri o stringa
char **strarr; vettore di vettore di caratteri o vettore
di stringhe
```

Si comprende quindi la segnatura della funzione main con **argv.
C supporta l'uso di stringhe che, tuttavia, corrispondono a degli array di caratteri.
Gli array sono generalmente di dimensione statica e non possono essere ingranditi durante l'esecuzione del programma. Per array dinamici dovranno essere usati costrutti particolari (come malloc).
Le stringhe, quando acquisite in input o dichiarate con la sintassi "stringa", terminano con il carattere '\0' e sono dunque di grandezza `str_len+1`
Sebbene ci siano diversi modi per dichiarare ed inizializzare una stringa, questi hanno comportamenti diversi:

```
char string[] = ciao; // writable string in the stack
char * string2 = ciao; // READ-ONLY string
string[2] = a;
string2[2] = a; //Segmentation fault!
```

Dato che le stringhe sono riferite con un puntatore al primo carattere non ha senso fare assegnamenti e confronti diretti, ma si devono usare delle funzioni. La libreria standard string.h ne definisce alcune come ad esempio:

* `char * strcat(char *dest, const char *src)`: aggiunge src in coda a dest
* `char * strchr(const char *str, int c)`: cerca la prima occorrenza di c in str
* `int strcmp(const char *str1, const char *str2)`: confronta str1 con str2
* `size_t strlen(const char *str)`: calcola la lunghezza di str
* `char * strcpy(char *dest, const char *src)`: copia la stringa src in dst
* `char * strncpy(char *dest, const char *src, size_t n)`: copia n caratteri dalla stringa src in dst

1.4 Lab-04

File
In unix ci sono due modi per interagre con i file:

- **Streams**:forniscono strumenti come la formattazione dei dati, bufferizzazione, ecc
  Utilizzando gli streams, un file è descritto da un puntatore a una struttura di tipo FILE (definita in stdio.h). I dati possono essere letti e scritti in vari modi (un carattere alla volta, una linea alla volta, ecc.) ed essere interpretati di conseguenza.
  Manipolazione con streams

  `FILE *fopen(const char *filename, const char *mode)`
  Restituisce un FILE pointer per gestire il "filename" nella modalità specificata:

  - r: read
  - w: write
  - r+ read adn write
  - w+: read and write, create or overwrite
  - a: write at the end
  - a+: read and write at the end

  `int fclose(FILE *stream)`: chiusura del file che ritorna un intero con lo stato della chiusura
  Lettura del file:

  - `int fgetc(FILE *stream)`: restituisce un carattere dallo stream
  - `char *fgets(char *str, int n, FILE *strem)`: ritorna una stringa da stream e la salva in str, si ferma quando sono stati letti n-1 caratteri, o viene letto un "\n", oppure raggiunta la fine Inserisce il carattere di terminazione.
  - `int fscanf(FILE *stream, const char *format)`: legge da stream dei dati, salvando ogni dato nelle variabili fornite, seguendo la stringa format
  - `int feof(FILE *strem)`: restituisce 1 se lo stream è arrivato alla fine del file.

  Scrittura del file:

  - `int fputc(int chr, FILE *stream)`: scrive un singolo carattere char su stream
  - `int fputs(const char *str, FILE *stream)`: scrive una stringa str su file

– `int fprintf(FILE *stream, const char *format)`:scrive il contenuto di alcune variabili su stream seguendo la stringa format

Flush e Rewind

Seguendo l'immagine, il contenuto di un file viene letto e scritto con degli streams (dei buffer) di dati. Come tali, è comprensibile come queste operazioni non siano immediate: i dati vengono scritti sul buffer e solo successivamente scritti sul file. Il **flush** è l'operazione che trascrive il file dallo stream. Questa operazione avviene quando:

– Il programma termina con un return dal main o con exit().

– `fprintf()` inserisce una nuova riga.

– `int fflush(FILE *stream)` viene invocato.

– `void rewind(FILE *stream)` viene invocato.

– `fclose()` viene invocato

rewind consente inoltre di ripristinare la posizione della testina all'inizio del file.

Esempio 1:

```
#include <stdio.h>
FILE *ptr; //Declare stream file
ptr = fopen("filename.txt","r+"); //Open
int id;
char str1[10], str2[10];
while (!feof(ptr)){ //Check end of file
      //Read int, word and word
      fscanf(ptr,"%d %s %s", &id, str1, str2);
      printf("%d %s %s\n",id,str1,str2);
}
printf("End of file\n");
fclose(ptr); //Close file
```

Esempio 2:

```
#include <stdio.h>
#define N 10
FILE *ptr;
ptr = fopen("fileToWrite.txt","w+");
fprintf(ptr,"Content to write"); //Write content to file
rewind(ptr); // Reset pointer to begin of file
char chAr[N], inC;
fgets(chAr,N,ptr); // store the next N-1 chars from ptr in
chAr
printf("%d %s",chAr[N-1], chAr);
do{
      inC = fgetc(ptr); // return next available char or EOF
```

```
     printf("%c",inC);
}while(inC != EOF);
printf("\n");
fclose(ptr);
```

- **File descriptor** interfaccia di basso livello costituita dalle system call messe a disposizione dal kernel. Un file è descritto da un semplice intero (**file descriptor**) che punta alla rispettiva entry nella file table del sistema operativo. I dati possono essere letti e scritti soltanto un buffer alla volta di cui spetta al programmatore stabilire la dimensione.
  Un insieme di system call permette di effettuare le operazioni di input e output mantenendo un controllo maggiore su quanto sta accadendo a prezzo di un'interfaccia meno amichevole.

  Per accedere al contenuto di un file bisogna creare un canale di comunicazione con il kernel, aprendo il file con la system call open la quale localizza li-node del file e aggiorna la file table del processo.
  A ogni processo è associata una tabella dei file aperti di dimensione limitata, dove ogni elemento della tabella rappresenta un file aperto dal processo ed individuato da un indice intero (il "file descriptor")
  I file descriptor 0, 1 e 2 individuano normalmente standard input (0=stdin), output (1=stdout) ed error (2=stderr) (aperti automaticamente).
  Il kernel gestisce l'accesso ai files attraverso due strutture dati:

  - La **la tabella dei files attivi** che contiene una copia dell'i-node di ogni file aperto (per efficienza)
  - La **tabella dei files aperti** che contiene un elemento per ogni file aperto e non ancora chiuso
    Questo elemento contiene:
    * I/O pointer: posizione corrente nel file
    * i-node pointer: Puntatore a inode corrispondente

    La tabella dei file aperti può avere più elementi corrispondenti allo stesso file

  Apertura e chiusura file
  `int open(const char *pathname, int flags[, mode_t mode]);`

  flags: interi che definiscono l'apertura del file:

  - **O_RDONLY, O_WRONLY, O_RDWR**: almeno uno è obbligatorio
  - **O_CREAT, O_EXCL**: crea il file se non esiste, come creat però fallisce se il file esiste già
  - **O_APPEND**: apre il file in append
  - **O_TRUNC**: cancella il contenuto del file (se usato in scrittura)

mode: interi per i privilegi da assegnare al nuovo file, **S_IRUSR, S_IWUSR, S_IXUSR, S_IRWXU, S_IRGRP, , S_IROTH**

`int close(int fd)`

Lettura e scrittura file

- `ssize_t read (int fd, void *buf, size_t count);`
  legge dal file e salva nel buffer buf, fino a count bytes di dati dal file associato con il file descriptor fd
- `ssize_t write(int fd, const void *buf, size_t count);`
  scrive sul file associato al file descriptor fd fino a count bytes di dati dal buffer 'buf'
- `off_t lseek(int fd, off_t offset, int whence);`
  riposiziona l'offset del file a seconda dell'argomento offset partendo da una certa posizione whence. Tre posizioni fondamentali: **SEEK_SET** (inizio file), **SEEK_CUR** (posizione attuale), **SEEk_END** (fine file)

Esempio 1:

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
//Open new file in Read only
int openedFile = open("filename.txt", O_RDONLY);
char content[10];
int canRead;
do{
     //Read 9B to content
     bytesRead = read(openedFile,content,9);
     content[bytesRead]=0;
     printf("%s",content);
} while(bytesRead > 0);
close(openedFile);
```

Esempio 2:

```
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
//Open file (create it with user R and W permissions)
int openFile = open("name.txt", O_RDWR | O_CREAT, S_IRUSR |
S_IWUSR);
char toWrite[] = "Professor";
write(openFile, "hello world\n", strlen("hello world\n"));
//Write to file
lseek(openFile, 6, SEEK_SET); // move I/O pointer
write(openFile, toWrite, strlen(toWrite)); //Write to file
```

```
close(openFile);
```

Canali standard

- I canali standard (in/err/out), sono rappresentati con strutture stream (stdin, stderr, stdout) e macro (STDIN_FILENO, STDERR_FILENO, STDOUT_FILENO)
- La funzione `fileno` restituisce l'indice di uno stream per cui:
  - `fileno(stdin) = STDIN_FILENO` (= 0)
  - `fileno(stdout) = STDOUT_FILENO` (= 1)
  - `fileno(stderr) = STDERR_FILENO` (= 2)
- `isatty(stdin) == 1` con esecuzione interattiva, 0 altrimenti
- `printf("ciao")` ≡ `fprintf(stdout, "ciao")`
- **Esempio**

```
#include <stdio.h>
#include <unistd.h>
void main() {
      printf("stdin:  stdin ->_flags = %hd, STDIN_FILENO =
         %d\n", stdin->_flags, STDIN_FILENO
      );
      printf("stdout:  stdout->_flags = %hd, STDOUT_FILENO =
         %d\n",stdout->_flags, STDOUT_FILENO
      );
      printf("stderr:  stderr->_flags = %hd, STDOUT_FILENO =
         %d\n", stderr->_flags, STDERR_FILENO
      );
}
```

Piping con bash

Normalmente un'applicazione eseguita da ash ha accesso ai canali standard, se le applicazioni sono usate in un'operazione di piping (`ls | wc -l`) allora l'output dell'applicazione sulla sinistra diventa l'input dell'applicazione sulla destra e verranno eseguite **parallelamente**
Esempio:

```
gcc src.c -o pip.out
echo "hi how are you" | ./pip.out
```

src.c

```
#define MAXBUF 10
#include <stdio.h>
```

```
#include <string.h>
int main() {
   char buf[MAXBUF];
   fgets(buf, sizeof(buf), stdin); // may truncate!
   printf("%s\n", buf);
   return 0;
}
```

```
ls /tmp | ./inv.out
```

Code:

```
#include <stdio.h>
int main() {
   int c, d;
   // loop into stdin until EOF (as CTRL+D)
   // read from stdin
   while ((c = getchar()) != EOF) {
      d = c;
      if (c >= 'a' && c <= 'z') d -= 32;
      if (c >= 'A' && c <= 'Z') d += 32;
      putchar(d); // write to stdout
   };
   return (0);
}
```

1.5 Lab-05

Architettura

Kernel Unix: il kernel è l'elemento di base di un sistema Unix-like, ovvero il nucleo del sistema operativo. Il kernel è incaricato della gestione delle risorse essenziali (CPU, mem, periferiche)
Ad ogni boot il sistema verifica lo stato delle periferiche, monta la prima partizione (roo file system) in read-only e carica il kernel in memoria.
Il kernel lancia il primo programma (systemd) che a seconda della configurazione voluta, ilizializza il sistema di conseguenza
Il resto delle operazioni, tra cui l'interazione con l'utente, vengono gestite con programmi eseguiti dal kernel
Kernel & Mem Virtuale
I programmi utilizzati dall'utente che vogliono accedere alle periferiche chedono al kernel di farlo per loro
L'interazione tra programmi ed il resto del sistema viene mascherata da alcune caratteristiche intrinseche ai processori, come la gestione del HW e della MemVirtuale
Ogni programma vede se stesso come **unico processore della CPU** e npn gli è dunque possibile disturbare l'azione degli altri programmi, $\Rightarrow$ stabilità sistemi Unix
Privilegi
Nei sistemi Unix-like ci sono due livelli di privilegi:

- **User space**: ambiente in cui vengono eseguiti i programmi
- **Kernel space**: ambiente in cui viene eseguito il kernel

System calls
Le interfacce con cui i programmi accedono al hw si chiamano **system calls**, cioè chiamate al sistema che il kernel esegue nel **kernel space** retituendo i risultati al programma chiamante nello user space
Le chiamate restituiscono -1 in caso di errore e settano la varaibile globale `errno`. Errori validi sono numeri positivi e seguono lo standard POSIX, che definisce gli alias
Librerie di sistema
Utilizzando il comando di shell `ldd` su di un eseguibile si possono visualizzare

1.6 Script

1.6.1 lez-01

1.0) Esempio prova

```
 nargs=$#
while [[ $1 != "" ]]; do
      echo "ARG=$1"
      shift
done
```

1.1) Scrivere uno script che dato un qualunque numero di argomenti li restituisca in output in ordine inverso.

```
lista=()
while [[ $1 != "" ]]; do
      lista=( "$1" "$lista[@]" )
      shift
done
echo "$lista[@]"
```

1.2) Scrivere uno script che mostri il contenuto della cartella corrente in ordine inverso rispetto all'output generato da "ls" (che si pu usare ma senza opzioni). Per semplicità , assumere che tutti i file e le cartelle non abbiano spazi nel nome.

```
dati=($( ls ))
lista=()
for i in $!dati[@]; do
      lista=( "$dati[$i]" "$lista[@]" )
done
echo $( ls )
echo "$lista[@]"
```

1.6.2 lez-02

2.1) Creare un makefile con una regola help di default che mostri una nota informativa, una regola backup che crei un backup di una cartella appendendo .bak al nome e una restore che ripristini il contenuto originale.

```
SHELL := /bin/bash
FOLDER := /tmp

help:
      @echo "make -f makefile_backup backup FOLDER=<path>"

backup:
      @echo "Backup of folder $(FOLDER) as $(FOLDER).bak..." ;
```

```
	sleep 2s
	@[[ -d $(FOLDER).bak ]] && echo "?Error" || cp -rp $(FOLDER)
$(FOLDER).bak

restore:  $(FOLDER).bak
	@echo "Restore of folder $(FOLDER) from $(FOLDER).bak..." ;
	sleep 2s
	@[[ -d $(FOLDER) ]] && echo "?Error" || cp -rp $(FOLDER).bak
$(FOLDER)

.PHONY: help backup restore
```

1.6.3 lez-03

3.1) Scrivere un'applicazione che data una stringa come argomento ne stampa a video la lunghezza, ad esempio: `./lengthof "Questa frase ha 28 caratteri"`
deve restituire a video il numero 28

```
#include <stdio.h>
int main(int argc, char **argv){
	int code = 0;
	int len = 0;
	char *p;
	printf("%d\n", argc);
	if(argc != 2) {
		fprintf(stderr,"Usage:  %s <stringa>\n", argv[0]);
		code = 2;
	}else {
		p = argv[1]; //Copy pointer to first argument
		while (*p != 0 ){ //Check if character is termination
			p++; //Move to next char
			len++; //Increase length count
		}
		printf("%d\n", len);
	}
	return code;
}
```

3.2) Scrivere un'applicazione che definisce una lista di argomenti validi e legge quelli passati alla chiamata verificandoli e memorizzando le opzioni corrette, restituendo un errore in caso di un'opzione non valida.
Non ho voglia di copiarlo

3.3) Realizzare funzioni per stringhe `char *stringrev(char * str)` (inverte ordine caratteri) e `int stringpos(char * str, char chr)` (cerca chr in str e restituisce la posizione)
(Non completo, controllare stringrev, da segmentation fault)

1.6.4 lez-04