GNU C Compiler e Makefile

gcc - opzioni del preprocessore

-DNOMESIMBOLO definisce il simbolo NOMESIMBOLO senza specificarne un valore. Il simbolo apparirà esistente a partire dall'inizio del file che si sta compilando.

-DNOMESIMBOLO=VALORE definisce il simbolo NOMESIMBOLO assegnandogli il valore VALORE che è una stringa qualunque. Il simbolo apparirà esistente a partire dall'inizio del file che si sta compilando.

> **Feature test macros** (per dettagli su simboli disponibili: man feature_test_macros).
> Un esempio particolare di utilizzo della define è rappresentato dal modo in cui si stabilisce quale standard utilizzare per compilare condizionatamente alcune parti di file di inclusione.
> Si consideri ad esempio il file di inclusione standard <time.h> in cui è contenuto il prototipo della funzione nanosleep(). Tale funzione è definito nello standard POSIX.1b specificate nell'edizione del 1993 (IEEE Standard 1003.1b-1993) quindi nel file di inclusione il prototipo della funzione nanosleep() è protetto da una direttiva al preprocessore che controlla l'esistenza di un simbolo `_POSIX_C_SOURCE` e controlla che quel simbolo abbia valore superiore a `199309L` . Per abilitare la compilazione di quel prototipo occorre definire quel simbolo PRIMA dell'inclusione del file time.h.
> Il modo migliore è dichiarare quel simbolo passandolo come opzione al compilatore gcc, ovvero passando l'opzione
>
> `-D_POSIX_C_SOURCE=199309L`
>
> Analogamente, per utilizzare la funzione `strerror_r()` occorre includere il file di inclusione `<string.h>` ma, poiché quella funzione è definita nelle specifiche IEEE Std 1003.1-2001 (cioè POSIX `200112L`) occorre anche definire il simbolo
>
> `-D_POSIX_C_SOURCE=200112L`
>
> Similmente, per abilitare le sole versioni POSIX dello standard originale (IEEE Std 1003.1) si può usare il simbolo `-D_POSIX_SOURCE=1` oppure `-D_POSIX_C_SOURCE=1`

-D'NOMEMACROCONARGOMENTI=IMPLEMENTAZIONEMACROSUUNICARIGA'
definisce la macro. I due singoli apici servono ad impedire che eventuali parentesi tonde e punti e virgola nella macro siano interpretati rispettivamente come raggruppamenti di comandi e fine del comando.

Esempio:

-D'SALUTA(**stringa**)=printf(" %s\n ", **stringa**); fflush(stdout)'

-UNOMESIMBOLO elimina la definizione del simbolo NOMESIMBOLO, equivale alla direttiva al preprocessore #undef NOMESIMBOLO, ma ha effetto sin dall'inizio dei file che compiliamo.

-IPercorsoDirectoryDegliHeaderFile specifica il percorso relativo o assoluto per raggiungere la directory in cui sono contenuti i file di intestazione dell'utente. L'opzione può essere utilizzata più volte per specificare più di un percorso. Es: **-I**. **-I**/home/studente/include **-I**../directory

gcc - opzioni del compilatore

-E ordina di effettuare la sola fase di preprocessing, mandando sullo standard output il codice sorgente C generato dal preprocessore.

-S ordina di tradurre il file sorgente C tramutandolo nel corrispondente sorgente assembly.

-masm=dialect Usato assieme al flag -S ordina di generare il sorgente assembly secondo il dialetto specificato, cioè Intel (-masm=intel) oppure AT&T (-masm=att). Se non viene usato questo flag, ma solo il flag -S, viene generato assembly nel dialetto AT&T.

-c ordina di **compilare** il modulo specificato e di non procedere al linking.

-o nomefile ordina di mettere il risultato della fase realizzata in un file di nome nomefile. Puo' essere usato per specificare sia il nome di un modulo oggetto generato nella fase di compilazione, ma anche il nome dell'eseguibile generato dalla fase di linking.

-ansi ordina di compilare secondo lo standard ansi, cioà c90.

--std=STANDARD specifica lo standard da usare nella compilazione, ad es: ansi c89 c90 gnu99.

-Wpedantic quando si compila secondo lo standard ansi (c90), ordina di avvisare quando si individuano istruzioni che rappresentano delle estensioni comunemente usate del linguaggio C (ad esempio dichiarazioni di variabili in mezzo al codice) che rendono il codice meno leggibile e percio' potenzialmente pericoloso.

-Wall avvisa quando individua istruzioni non sbagliate ma potenzialmente pericolose o anche istruzioni che rendono il codice meno leggibile e percio' potenzialmente pericoloso. Non significa però abilitare tutti i possibili warning.

--pedantic è sostanzialmente analogo a -Wpedantic.

-Werror ordina al compilatore di trattare ogni warning come se fosse un errore, interrompendo la compilazione. Viene usato per auto-costringersi ad eliminare i warning.

gcc - opzioni del linker

-LPercorsoDirectoryDelleLibrerie specifica il percorso relativo o assoluto per raggiungere la directory in cui sono contenute le librerie fornite dall'utente. L'opzione può essere utilizzata più volte per specificare più di un percorso. Es: **-L**. **-L**/home/studente/lib

-lnomelibreria specifica di utilizzare la libreria indicata dal nome "ristretto" nomelibreria. Ad esempio, la libreria matematica lib**m**.a viene specificato con **-l**m mentre una eventuale libreria lib**pippo**.a viene specificato con **-lpippo** .

Nota bene: l'opzione **-l** può essere usata più volte in una stessa riga di comando per indicare tante librerie diverse.

Attenzione: nelle versioni più recenti del gcc, **le opzioni -l devono essere collocate obbligatoriamente alla fine della riga di comando, dopo l'elenco dei moduli da collegare**.

-Wl,-soname,*nomelibreria* (tutto attaccato, e le virgole servono) utilizzato quando si sta creando una libreria dinamica, specifica il nome *nomelibreria* da assegnare alla libreria che si sta creando.

-Wl,-rpath,*RunTimeLibDir* (tutto attaccato, e le virgole servono) utilizzato quando si sta creando un eseguibile che deve usare una libreria dinamica collocata fuori dalla directory predefinita. In questo modo si indica la directory *RunTimeLibDir* in cui, nel momento della esecuzione (cioé a run-time) dell'eseguibile, si troverà la libreria dinamica da utilizzare. Mnemonicamente, rpath significa Run-Time Path.

Makefile

Il **file dependency system di Unix** nasce per automatizzare il corretto aggiornamento di più file che hanno delle dipendenze. Viene solitamente utilizzato per automatizzare le operazioni di compilazione e linking di progetti scritti in linguaggio C, ma può essere utilizzato anche per altri scopi.

1. Concetto di dipendenza e albero delle dipendenza.

Supponiamo di avere scritto un programma costituito da due moduli C implementati in due file main.c e funzioni.c, di avere un altro file di intestazione funzioni.h incluso in main.c, e di avere un ulteriore altro file di intestazioni strutture.h incluso sia in funzioni.c che in main.c.
Per compilare il progetto occorre compilare i due moduli main.c e funzioni.c e generare così i due moduli oggetto main.o e funzioni.o. Fatto questo occorre linkare assieme i due moduli main.o e funzioni.o per generare l'eseguibile finale main.exe.

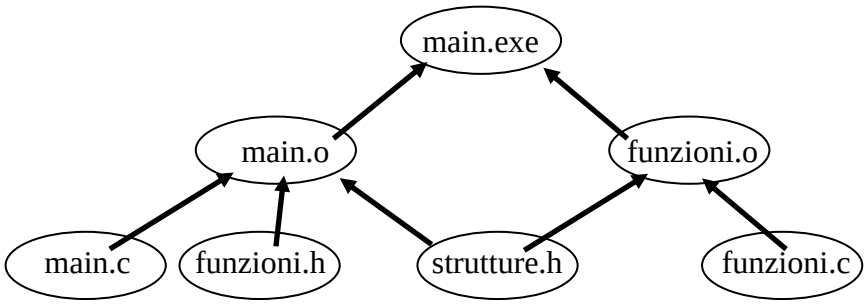

Figura 1: albero delle dipendenza dell'esempio

Da questa descrizione origina il **concetto di dipendenza** (dependency) di un file da un altro. Un file ***target1*** dipende da un altro file ***dipendenza1*** quando il contenuto del file ***target1*** dipende dal contenuto del file ***dipendenza1***. Una modifica del file ***dipendenza1*** causa la necessità di effettuare delle operazioni per rigenerare il contenuto del file ***target1***. Possiamo rappresentare graficamente una dipendenza come una freccia che parte dal file dipendenza e arriva al file target, collocato graficamente più in alto della sua dipendenza. Uno stesso file target può dipendere da più dipendenze.
Nelle dipendenze di un target vanno considerati anche i file di inclusione inclusi, con una direttiva al preprocessore, nei moduli C. Nell'esempio citato, poiché il file funzioni.h è incluso nel file main.c, la modifica del file funzioni.h appare agli occhi del compilatore come una modifica del file main.c e questo causa la necessità di ricompilare il file main.c.
In sostanza, **un modulo oggetto .o dipende sia dal file .c che implementa quel modulo ma anche dai file di intestazione .h che sono inclusi in quel file .c.**

Poiché un file target può essere a sua volta una dipendenza di un altro target, la rappresentazione grafica di tutte le dipendenze diventa un albero detto albero delle dipendenze (dependency tree). Si noti che se il file target1 è a sua volta una dipendenza di un altro file target2, allora la modifica del file dipendenza1 causa la necessità di rigenerare il file target1 e questo causa la necessità di rigenerare il contenuto del file target2 utilizzando il contenuto del file target1. In pratica, l'insieme delle dipendenze causa il fatto che la modifica di un file dipendenza comporti una serie di operazioni per rigenerare tutti i target che, direttamente o indirettamente, dipendono da quel file dipendenza. In termini grafici, la modifica di un file causa la necessità di rigenerare tutta la parte di albero che, seguendo le frecce verso l'alto, segue il file modificato.

Nell'esempio citato, supponiamo di avere effettuato una iniziale compilazione di tutti i moduli e il linking generando l'eseguibile main.exe. Se, successivamente, io modificassi il file funzioni.c, dovrei ricompilare il modulo funzioni.c generando un nuovo modulo oggetto funzioni.o e, a questo punto, dovrei linkare assieme il nuovo modulo oggetto funzioni.o con il vecchio modulo oggetto main.o per generare un muovo eseguibile main.exe. Si noti che, nel caso descritto di modifica del solo file funzioni.c, non sarebbe necessario ricompilare il file main.c perché il vecchio modulo main.o corrisponderebbe ancora al file main.c che non è stato modificato. In questo senso,

2. Condizione di rigenerazione di un target.

E' importante notare che la necessità di rigenerare un target dipende dal fatto che sia vera una delle due seguenti condizioni:

1. il file target non esiste, quindi deve essere generato ex-novo.
2. il file target esiste ma è più vecchio di uno dei suoi file di dipendenza, quindi non corrisponde più al contenuto di quel file di dipendenza. Quindi, se la data di ultima modifica di una dipendenza è più recente della data di ultima modifica del target, allora il target deve essere rigenerato.

Si noti che la rigenerazione di un target rende necessaria la rigenerazione in cascata verso l'alto di tutti i target che nell'albero delle dipendenze "dipendono" (seguendo le frecce) da quel target.

3. Struttura del Makefile: Make Rules

Il file **Makefile** di un progetto è un file di testo che contiene un insieme di regole (**make rules**) che nel loro complesso :

1. descrivono la struttura dell'albero delle dipendenze del progetto;
2. descrivono le regole di generazione dei vari target di ogni dipendenza del progetto.

Ciascuna make rule ;

1. indica uno o più **target** a cui la regola si riferisce (**target list**),
2. elenca tutte le dipendenze da cui quel target dipende (**dependency list**),
3. ed elenca le operazioni (**command list**) che devono essere svolte quando una delle dipendenze è più recente del target oppure quando il file target non esiste.

Solitamente, l'operazione descritta genera il target, ma non è indispensabile. Esistono infatti target fittizi, inseriti proprio per svolgere sempre alcune operazioni.

Le regole devono essere così formate;

```
target_list :   dependency_list
  TAB           command1
                ...
  TAB           commandN
RIGA_VUOTA
```

4. Attenzione alla sintassi nei Makefile

Nella scrittura di un Makefile occorre rispettare, per ragioni di compatibilità storiche, alcune inquietanti regole sintattiche.

- Il # indica una riga commentata.
- Ciascuna dependency list e ciascun comando termina dove termina la riga.
- In ciascuna regola, le dipendenze sono separate tra loro da uno o più spazi bianchi o tab.
- Ogni comando della command list deve iniziare con un carattere TAB.
- Ciascuna regola deve essere separata dalla successiva da una riga vuota.
- Il Makefile deve terminare con una riga vuota (o meglio, una andata a capo \n).

Il Makefile per l'esempio citato è così fatto:

```
all :           main.exe
riga vuota
main.exe :      main.o  funzioni.o
TAB             gcc -ansi -Wpedantic -o main.exe main.o funzioni.o
riga vuota
main.o :        main.c  funzioni.h strutture.h
TAB             gcc -c -ansi -Wpedantic main.c
riga vuota
funzioni.o :    funzioni.c  strutture.h
TAB             gcc -c -ansi -Wpedantic funzioni.c
riga vuota
clean:
TAB             -rm    main.exe *.o                    <- del  - ne parliamo dopo
NEWLINE
```

(scaricabile in http://www.cs.unibo.it/~ghini/didattica/sistemioperativi/MAKEFILE_ESEMPINOTEVOLI/simplemake.tgz)

5. Ordine delle Regole e costruzione dell'albero delle dipendenze

L'ordine delle regole scritte nel Makefile è importantissimo, infatti:

- Il make costruisce l'albero delle dipendenze a partire dalla prima regola che incontra leggendo il file delle regole Makefile.
- Il target trovato nella prima regola letta costituisce la radice dell'albero delle dipendenze.
- Ciascuna dipendenza nella dependency list della prima regola viene appesa come nodo figlio della radice dell'albero.
- Per ciascuna dipendenza (nodo figlio e foglia) appena aggiunta si cercano nel makefile le regole che hanno quella dipendenza come proprio target. Per ciascuna regola trovate, si leggono le dipendenze e le si aggiunge come figli del proprio target.
- Si continua utilizzando le foglie come candidate target, cercando regole che hanno come target le candidate target e, se le si trova, aggiungendo come foglie le dipendenze delle candidate target.
- Ci si ferma quando nelle foglie ho solo delle dipendenze che non compaiono come target in nessuna regola.

6. Esecuzione delle lista dei comandi mediante visita Bottom-Up dell'albero delle dipendenze

L'albero delle dipendenze viene visitato partendo dai livelli più bassi. Per ciascun nodo N si legge la data di ultima modifica del file N e la si confronta con la data di ultima modifica del file presente nel nodo padre P. Se il file del nodo padre P non esiste oppure se ha data di ultima modifica più vecchia del nodo figlio N allora viene eseguita la lista di comandi della regola che ha il padre come target.

7. Esecuzione del make.

La creazione dell'albero delle dipendenze e l'esecuzione deIle liste di comandi viene effettuata eseguendo il comando ***make*** passandogli come argomento l'opzione ***-f*** seguita dal nome del file che contiene le regole da seguire

make -f nomeMakefile

Se non viene specificata l'opzione -f allora il comando make cerca nella directory corrente un file di nome Makefile (o anche makefile, nei sistemi recenti) per usarlo come contenitore delle regole.
Durante l'esecuzione, il file make stampa sullo stdout i comandi delle liste mentre li esegue.
Aggiungendo l'opzione -n ottengo che il make stampi i comandi che dovrebbe eseguire ma senza eseguirli veramente.

Posso ordinare di eseguire il make specificando esplicitamente il target che voglio sia usato come root dell'albero delle dipendenze. In pratica specifico quale target voglio ottenere. Verrà visitato solo il sottoalbero appeso al target che ho specificato.
Nell'esempio specificato, potrei volere ottenere solo il modulo oggetto funzioni.o nel qual caso dovrei eseguire il comando:

make -f nomeMakefile **funzioni.o**

8. Best Practices #1 nella scrittura dei Makefile: all come Prima Regola

La prima regola stabilisce quale è il target che costituisce la radice dell'albero.
Molto spesso, si definisce come prima regola una regola che come target specifica **all**, come dipendenze specifica il/i target reali **e che ha una lista di comandi vuota**. Ciò serve a indicare quali sono i target da ottenere.

```
all :    main.exe
```

In tal caso, per eseguire le regole specificate nel Makefile potrò eseguire

make -f nomeMakefile **all**

9. Target fittizi (.phony).

È possibile specificare target che non sono file e che hanno come scopo solo l'esecuzione di una sequenza di azioni.
Questi target non specificano nessuna dipendenza e NON devono comparire come prima regola, in modo da essere eseguiti solo se vengono passati come argomento al comando make esplicitamente.
Poiché il target fittizio non è un file (non esiste il file) quel target fittizio serve a provocare l'esecuzione del comando in ogni caso.
Ad esempio, un classico target fittizio usato assai spesso è :

```
clean:          NESSUNA DIPENDENZA
        rm  *exe  *.o  *~
```

che serve ad eliminare tutti gli eseguibili e i moduli oggetto generati ed anche i file temporanei degli editor. Poiché non c'è nessuna regoola che crea un file chiamato clean, il comando rm verrà eseguito ogni volta che invoco

make clean

10. Controindicazioni nell'uso dei target fittizi.

Esistono due controindicazioni nell'uso dei target fittizi:

1. Mascheramento da parte di file esistenti: Se per sbaglio nella directory viene creato un file chiamato con lo stesso nome del target fittizio allora, poiché la regola fittizia non ha dipendenze e il file c'è già, quel file non ha necessità di essere aggiornato e quindi la lista dei comandi non verrà mai eseguita.
2. **Non trattato.** Spreco di tempo. In caso di regole implicite (**che qui non vengono trattate**) il make spreca tempo, perchè cerca di risolvere il target fittizio applicandogli le regole implicite.

11. Best Practices #2 nella scrittura dei Makefile: esplicitare i target fittizi con .phony

Nel file delle regole, è bene esplicitare quali sono i target fittizi elencandoli come dipendenze di un target predefinito denominato **.phony**. Ad esempio:
.PHONY **:** clean

```
clean:          NESSUNA DIPENDENZA
                rm  *exe  *.o  *~
```

Per ciascun target elencato come dipendenze di .phony il make non cerca l'esistenza di un file e non cerca di applicare al target le regole implicite (non trattate qui).

12. Variabili "del" make.

Dentro il file delle regole è possibile dichiarare delle variabili "del" make, inserendole all'inizio del Makefile stesso, nel senso che hanno validità solo dentro il Makefile, ovvero possono essere utilizzate solo nel resto del Makefile. Si noti che **il Makefile NON è uno script**, quindi alle variabili definite all'inizio del Makefile devono essere assegnati dei valori costanti, e non posso assegnare dei valori ottenuti invocando degli altri comandi.
Ad esempio, all'inizio del Makefile posso scrivere:

```
PERCORSO=/usr/local/lib
```

ma NON POSSO scrivere

```
PERCORSO=`pwd`
```

inoltre, **all'inizio del Makefile** non posso nemmeno eseguire comandi, ad esempio non posso eseguire:

```
cd  /usr/local/lib
```

In generale, infatti, posso eseguire dei comandi **SOLO dentro** la lista di comandi inserita in ciascuna regola.

Esiste un caso in cui serve definire una variabile.
I comandi specificati nelle liste di comandi del Makefile sono eseguiti in una shell sh, non in una bash. La shell sh è simile ma non uguale alla bash. Alcune differenze possono causare problemi. In particolare: a) in sh NON ESISTONO le espressioni condizionali con le DOPPIE PARENTESI QUADRE [[ ]] ma solo quelle con le semplici parentesi quadre [ ]. Inoltre 2) nel comando built-in echo della sh non esiste l'opzione -e.
Se voglio essere assolutamente sicuro di eseguire i comandi proprio in una bash, devo inserire nel Makefile una prima riga in cui definisco la variabile

```
SHELL=/bin/bash
```

13. Best Practices #3 nella scrittura dei Makefile: indicare comandi di cui non conta il risultato

Il make termina con un errore quando uno dei comandi di una lista dei comandi viene eseguito e restituisce un errore (error code >0). Se di un comando non interessa il risultato, ma si vuole impedire che il make termini anche se quel comando termina con un errore, è possibile mettere un - davanti al comando. Il make interpreta quel - come l'ordine di non considerare un eventuale errore restituito dal comando.
Ad esempio, quando eseguo il comando clean, non vorrei preoccuparmi se i file da cancellare esistono oppure no, In altre parole, non mi interessa il risultato restituito da rm. Potrei perciò informare il make che non mi frega nulla del risultato di rm mettendogli davanti un bel - .

clean **:**

```
	-rm   *exe   *.o  *~
```

Provare cosa capita quando in un Makefile compaiono le seguenti righe. Assicurarsi, prima, che tra i nomi e i valori delle variabili d'ambiente non compaia la stringa VAFF:

```
all:
	echo env1PATH
	env | grep PATH
	echo env1VAFF
	-env | grep VAFF
	echo env1VAFF
	env | grep VAFF
	echo env1VAFF
	env | -grep VAFF
```

14. **Conclusioni.**

Il Makefile per l'esempio di progetto descritto all'inizio.

```
CFLAGS=-ansi -Wpedantic
TARGET=main.exe
OBJECTS=main.o funzioni.o

all :           ${TARGET}
riga vuota
main.exe :      main.o  funzioni.o
TAB             gcc ${CFLAGS} -o ${TARGET}  main.o funzioni.o
riga vuota
main.o :        main.c  funzioni.h strutture.h
TAB             gcc -c   ${CFLAGS}  main.c
riga vuota
funzioni.o :    funzioni.c  strutture.h
TAB             gcc -c  ${CFLAGS}  funzioni.c
riga vuota
.PHONY:         clean
riga vuota
clean:
TAB             -rm    ${TARGET}  ${OBJECTS}   *~   core
NEWLINE
```

15. **Alcuni Inquietanti Dettagli a volte importanti**

15.1 **In che directory viene eseguito il make e come cambiare tale directory.**

Il make viene eseguito nella directory in cui lancio il comando make stesso, quindi nella directory corrente. In questa directory corrente, il make cerca il file Makefile, In questa stessa directory il make esegue la lista dei comandi specificati dalle regole scritte nel Makefile.
Qualora sia necessario far eseguire il make all'interno di una directory *nomedirectory* diversa dalla directory corrente, occorre specificare nella riga di comando del make l'opzione -C *nomedirectory* oppure l'opzione --directory=*nomedirectory*. In tal modo, il make cambia la propria directory corrente, entrando in quella specificata dal nome *nomedirectory* , prima di cercare il file Makefile. Dopo avere letto il file Makefile, il make esegue le regole nella nuova directory specificata.
Al termine dell'esecuzione del comando make, la shell in cui abbiamo invocato il comando make si troverà ancora nella directory corrente originale.
Tale possibilità viene utilizzata per creare Makefile ricorsivi, ad esempio i Makefile per compilare moduli del kernel Linux.

15.2 **In che tipo di shell vengono eseguiti i comandi specificati nel Makefile.**

I comandi specificati nelle liste di comandi del Makefile potrebbero essere eseguiti in una shell sh o dash, non in una bash. Dipende dalla configurazione del sistema operativo.
Le shell sh e dash sono molto simili alla bash ma non sono esattamente uguali alla bash. Alcune differenze possono causare problemi.
In particolare:
in sh e dash NON ESISTONO le espressioni condizionali con le DOPPIE PARENTESI QUADRE [[ ]] ma solo quelle con le semplici parentesi quadre [ ].
Il comando built-in echo della sh e dash è diverso dal comando built-in echo della bash, infatti nel comando echo della sh e dash NON ESISTONO le opzioni **-e** e -E che invece esistono nel comando echo della bash.
Se voglio essere assolutamente sicuro di eseguire i comandi proprio in una bash, devo inserire nel Makefile una prima riga in cui definisco la variabile

```
SHELL=/bin/bash
```

Come verifica, provare ad eseguire il comando make in una directory in cui esiste un file Makefile con il seguente contenuto:

```
# SHELL=/bin/bash

all :
        ps
        echo -e "ciao \t gatto"
        if  [[  -f  Makefile  ]]  ; then  echo "esiste" ; fi
```

Accadranno degli errori dovuti al fatto che i comandi vengono eseguiti in una sh, ed il comando ps farà vedere che tra i processi è in esecuzione proprio una shell sh..
Poi decommentare la prima riga e ri-eseguire il comando per verificare che i comandi vengono lanciati in una bash e che non accadono errori.

15.3 **Come utilizzare le variabili d'ambiente della shell nelle liste di comandi del Makefile.**
All'interno della lista di comandi da eseguire per ottenere un target, posso utilizzare, oltre alle variabili del make, anche le variabili della bash (in generale della shell) in cui i comandi verranno eseguiti.
Se definisco una variabile del make con lo stesso nome di una variabile della shell, nascondo la variabile della shell (se però lancio il make con l'opzione -e oppure --environment-overrides allora la variabile della shell nasconde la variabile del make).
NB: Se premetto un solo carattere $ davanti al nome della variabile, posso selezionare sia i nomi delle variabili del make ed anche il nome delle variabili della shell. Invece, per indicare solo il nome di una variabile della shell, devo usare DUE caratteri $ invece che uno solo, ad esempio per usare la variabile PATH devo scrivere $${PATH} invece che ${PATH}. In particolare, per indicare la variabile che contiene il risultato dell'ultimo comando restituito $? devo utilizzare NON UNO BENSI' **DUE caratteri $** prima del ?. Quindi per usare la variabile $? devo scriverla così **$$**? Analogamente, per visualizzare la variabile che contiene il PID della shell corrente $$ devo utilizzare il doppio dei $, cioè devo scrivere la variabile così **$$$$**
Per le altre variabili della shell, è sempre bene utilizzare anche la coppia di parentesi graffe, ad es: $${PATH}.

```
target: dependancy list
        grep stringa nomefile ; if $$? ; then echo $${PATH} ; else echo $${USER} ; fi
```

15.4 **Attenzione alla differenza tra variabili del make e variabili della shell, anche quando sono definite DENTRO la lista dei comandi. Occorre indicarle in modo diverso.**

```
# Makefile
VarDelMake=CIAO
all:
        for name in ` ls ` ; do  echo $${name}  $${PATH}  ${VarDelMake} ; done
```

Notare che uso le variabile del make con UN SOLO DOLLARO, mentre uso le variabili della shell con DUE DOLLARI.

15.5 **Ciascuna riga di comando della lista di comandi 1) viene lanciata in una propria shell separata che è figlia della shell in cui esegue il make, e 2) può contenere più comandi i quali saranno eseguiti come figli di quella shell figlia.**
La lista di comandi di una regola è costituita da tante righe, ciascuna delle quali inizia con un TAB e termina dove finisce la riga, cioè dove c'è l'EOL. In ciascuna di queste righe di comandi, posso specificare una sequenza di comandi. Questa riga con tanti comandi viene eseguita in una shell figlia della shell del make, quindi ciascuna riga di comandi comincia la sua esecuzione in un ambiente nuovo, che non risente delle azioni svolte dalle altre righe di comandi.

Facciamo un esempio pratico: nella directory **/tmp** viene lanciato il comando make, quindi la directory corrente vista dal make è **/tmp**. In quella directory c'è un Makefile con il seguente contenuto: vediamo che succede

```
# Makefile
all:
        pwd;  cd / ; pwd   <- la prima riga visualizza /tmp  poi si sposta in /  e visualizza  /
        pwd                <-la seconda riga visualizza ancora /tmp  nonostante nella riga
                             precedente ci si sia spostati in  /
```

La morale è che ogni riga di comandi viene eseguita in un ambiente di esecuzione nuovo che è una copia dell'ambiente del make. **I comandi di una stessa riga risentono delle**

operazioni svolte solo dai comandi già eseguiti in quella stessa riga di comandi, non nelle righe precedenti.

15.6 **Andare a capo in una singola lista di comandi, sfruttando \ come nelle macro**

Nel Makefile, se una riga di comandi della lista di comandi da eseguire è troppo lunga, posso andare a capo riga inserendo un carattere \ come ultimo carattere prima dell'EOL. In tal caso, la riga di comandi prosegue nella riga successiva. E' come nella macro in C.
Invece non posso sperare di far proseguire la riga inserendo dei doppi apici.

```
all:
# ok
	gcc -c -ansi -Wall   \
	      main.c  -lm
# ok
	gcc -c -ansi -Wall   \
main.c  -lm
# ok
	pwd ; ls ;  cd /root/  ;  \
	   pwd ; ls ;  cd
#
# NO, causa errore
	echo "Questo provoca
	   un errore sintattico"
```

15.7 **Ordine di definizione delle variabili del make e annidamento delle variabili**

Nel Makefile, può accadere che una variabile del make contenga, nel suo contenuto, il contenuto di una seconda variabile del make. Ebbene, questa seconda variabile del make può essere definita sia prima che dopo avere definito la prima variabile, senza provocare nessuna differenza nel contenuto finale della prima variabile.
Ciò accade perché il make prima legge tutte le regole e poi assegna i valori alle variabili.

Ad esempio, se considero il seguente Makefile

```
VAR1=var1
VAR0=${VAR3}
VAR3=${VAR1}_${VAR2}
VAR2=var2

all:
	echo  ${VAR0}
```

ed eseguo il comando make, ottengo il seguente output

```
var1_var2
```

16. **Makefile Ricorsivi**

Potremmo trovarci in una situazione in cui il nostro progetto è dislocato in tante directory separate.
In ciascuna directory posso collocare un Makefile che permette di svolgere il compito in quella specifica directory.
Nella directory principale del progetto devo collocare un **Makefile** in cui il target principale **all:** ha il compito di ordinare di:

1. **andare in ciascuna directory** dove devo svolgere qualche compito;
   1.1. lanciare il make per il Makefile **in quella specifica** directory.

Dopo avere eseguito tutti i Makefile nelle sottodirectory del progetto, il makefile nella directory principale dovrà

2. eseguire il compito di mettere assieme tutti i files generati e creare il risultato finale del progetto. Occorre perciò creare un target diverso dal target principale che esegue il compito di generare il risultato finale.

Un esempio potrebbe essere questo:

```
# NOTARE che nel target principale non metto nessuna dipendenza
# per costringere il make ad eseguire tutti i comandi che specifico sotto
# almeno fino a che non capita un errore in una sottodirectory
# NOTARE anche che PER CIASCUNA RIGA DI COMANDI,
#                  il make riparte dalla DIRECTORY CORRENTE

all:
                cd 1 ; make                     NB: Potevo anche scrivere solo   make -C 1
                cd 2 ; make                     NB: Potevo anche scrivere solo   make -C 2
                make main.exe

main.exe:       main.o 1/obj1.o 2/obj2.o
                gcc ${CFLAGS} -o main.exe main.o 1/obj1.o  2/obj2.o ${LIBRARIES}

main.o:         main.c
                gcc -c ${CFLAGS} main.c

.PHONY: clean

clean:
                - cd 1 ; make clean
                - cd 2 ; make clean
                - rm -f main.o main.exe
```

Notare che metto in una stessa riga, il comando di cambio di directory ed il comando make. Infatti, ogni comando lanciato dal make viene eseguito nella directory in cui viene lanciato il make.
Se scrivessi *cd 1* **e poi a capo riga scrivessi** *make* **il comando make verrebbe eseguito ancora nella directory principale.**

Un esempio completo di semplice progetto con Makefile ricorsivo è disponibile a questo link:
http://www.cs.unibo.it/~ghini/didattica/sistemioperativi/MAKEFILE_RICORSIVO.tgz

Un piccolo limite del Makefile proposto è che occorre elencare esplicitamente tutte le directory in cui occorre entrare.
Il Makefile potrebbe essere modificato per automatizzare la ricerca delle directory in cui operare.

17. **Makefile Ricorsivi con automatizzazione della ricerca delle sottodirectory**

Ipotesi: supponiamo che, in tutte le directory figlie direttamente contenute nella directory corrente, sia stato scritto un file di regole Makefile da far interpretare.

Allora, posso scrivere un Makefile che utilizza un comando for per entrare in tutte queste directory e lanciare il comando make in ciascuna delle directory.
Notare che il cambio della directory corrente lo faccio fare al make con l'opzione -C per non dovermi occupare di tornare alla directory corrente originale al termine dell'esecuzione di ciascun make.

Un esempio potrebbe essere questo:

```
# NOTARE che nel target principale non metto nessuna dipendenza
# per costringere il make ad eseguire tutti i comandi che specifico sotto
# almeno fino a che non capita un errore in una sottodirectory
# NOTARE anche che PER CIASCUNA RIGA DI COMANDI,
#                  il make riparte dalla DIRECTORY CORRENTE

all:
                for DIRNAME in `find ./ -mindepth 1 -maxdepth 1 -type d -print` ; do if
! make -C ${DIRNAME}  ; then exit $? ; fi ; done
                make main.exe


main.exe:       main.o 1/obj1.o 2/obj2.o
                gcc ${CFLAGS} -o main.exe main.o 1/obj1.o  2/obj2.o ${LIBRARIES}

main.o:         main.c
                gcc -c ${CFLAGS} main.c

.PHONY: clean

clean:
                for DIRNAME in `find ./ -mindepth 1 -maxdepth 1 -type d -print` ; do if
! make -C ${DIRNAME} clean ; then exit $? ; fi ; done
                - rm -f main.o main.exe
```