

In questo secondo articolo su `make` vedremo nuove tecniche per semplificare i nostri `makefile` e renderli sempre più flessibili ed adattabili ad ogni esigenza del progetto

Rodolfo Giometti

r.giometti@oltrelinux.com

Ingegnere Informatico specializzato in automazione e robotica, lavora come libero professionista. Attualmente si trova a Pavia, dove collabora con Alessandro Rubini.



GNU Make: gestire progetti complessi (2)

Eccoci di nuovo a parlare di `make`. In questo articolo finiremo la trattazione degli argomenti iniziati nel precedente e, in particolare, continueremo ad occuparci di come si possa rendere sempre più versatile e compatto il `makefile` scritto per il piccolo progetto presentato la scorsa volta.

Un piccolo richiamo

Nello scorso articolo abbiamo presentato una serie di programmi scritti in C tra i quali sussistevano varie dipendenze. Nella *figura 1* riporto lo schema delle dipendenze e vi invito a riprendere il precedente articolo per vedere il codice dei vari programmi che, per ovvie esigenze di spazio, non riporto. Vi invito inoltre a dare un'occhiata al `makefile` che avevamo ottenuto la volta scorsa giusto per fare mente locale su quello che abbiamo già detto.

La definizione delle dipendenze

Eravamo rimasti alla definizione del `makefile` come:

```
TARGET = calc
OBJECTS = calc.o fact.o comb.o sum.o
```

```
$(TARGET): $(OBJECTS)
```

```
calc.o : calc.c calc.h func.h
```

```
fact.o : fact.c calc.h
```

```
comb.o : comb.c calc.h
```

```
sum.o : sum.c calc.h
```

Vediamo ora come possiamo semplificarlo ulteriormente e, nello specifico, vediamo come possiamo fare in modo che se modifichiamo, aggiungendo o

togliendo un file dalla definizione della variabile `OBJECTS`, `make` automaticamente esegua i passi giusti per la compilazione senza dover aggiungere ulteriori comandi.

Ad esempio, supponiamo che nel nostro progetto dobbiamo, ad un certo punto, aggiungere un nuovo file `dummy.c` definito come segue:

```
void dummy(void)
{
    /* Do nothing! */
}
```

Con il `makefile` presentato sopra dovremmo aggiungere il nome del nostro nuovo file alla definizione di `OBJECTS` e poi aggiungere la linea:

```
dummy.o : dummy.c
```

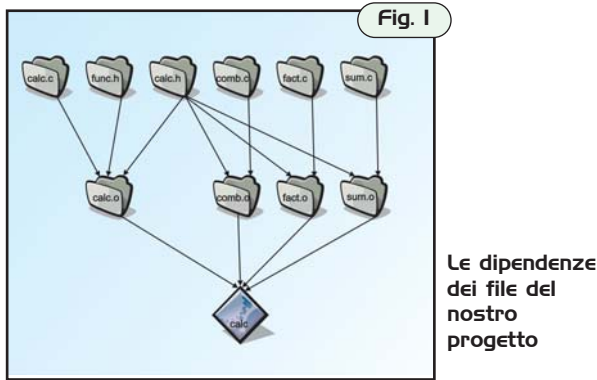
C'è un modo più veloce e compatto per fare questa cosa. Infatti le dipendenze di un file possono essere specificate anche in linee successive, in particolare le dipendenze del file `calc.o` possono essere specificate come:

```
calc.o : calc.c
```

```
calc.o : calc.h
```

```
calc.o : func.h
```

Inoltre `make`, grazie alle *implicit rules*, inizia da solo, dato un file target, a cercare le sue dipendenze facendo delle "prove": quando cerca di generare il file `calc.o` lui controlla se, ad esempio, esiste un file `calc.c` e, se lo trova, esegue la regola relativa alla compilazione di un programma C. Detto questo, possiamo dire che le tre regole sopra citate sono equivalenti a:



```
calc.o : calc.h
calc.o : func.h
```

in quanto la dipendenza rimossa è già implementata all'interno di make. Nota questa caratteristica il nostro makefile diventa:

```
TARGET = calc
OBJECTS = calc.o fact.o comb.o sum.o
```

```
$(TARGET) : $(OBJECTS)
```

```
calc.o : calc.h func.h
fact.o : calc.h
comb.o : calc.h
sum.o : calc.h
```

A questo punto, allora, per aggiungere il nostro file `dummy.c` non occorre che aggiungiamo la linea relativa alla sua dipendenza, perché questa è implicita e quindi il makefile diventa:

```
TARGET = calc
OBJECTS = calc.o fact.o comb.o sum.o dummy.o
```

```
$(TARGET) : $(OBJECTS)
```

```
calc.o : calc.h func.h
fact.o : calc.h
comb.o : calc.h
sum.o : calc.h
```

Le dipendenze automatiche

Bene, abbiamo ridotto proprio all'osso il nostro makefile, ma possiamo fare di più! Come vedete, nelle dipendenze esplicite elencate non sono rimasti che riferimenti ai file di tipo include. A questo punto, "tiro fuori il coniglio dal cilindro" ed introduco un parametro speciale del compilatore C che permetterà di semplificare ulteriormente il nostro makefile. Questa funzionalità è valida solo per i

programmi C, ma serve per introdurre il concetto di *variabile automatica*.

Tenete presente che queste "funzionalità speciali" di alcuni programmi, create appositamente per poter essere utilizzate in associazione con make, permettono di aumentare la flessibilità di utilizzo di entrambe le applicazioni.

Se lanciate il compilatore con l'opzione `-M` allora questo non effettuerà nessuna operazione di compilazione, ma creerà una lista che elenca le dipendenze del file passato; ma non solo, questa lista sarà in formato makefile! Ecco allora che questa funzionalità può essere utilizzata per automatizzare la gestione delle dipendenze.

Facciamo un po' di esempi, vediamo come si possono generare le dipendenze di un file `.c`:

```
$ cc -M comb.c
comb.o: comb.c calc.h
```

Come vedete abbiamo un output in formato makefile e se poi diamo il comando:

```
$ cc -M calc.c fact.c comb.c sum.c dummy.c
calc.o: calc.c /usr/include/stdio.h /usr/include/features.h \
/usr/include/sys/cdefs.h /usr/include/gnu/stubs.h \
/usr/lib/gcc-lib/i386-linux/2.95.4/include/stddef.h \
/usr/include/bits/types.h /usr/include/bits/pthreadtypes.h \
/usr/include/bits/sched.h /usr/include/libio.h \
/usr/include/_G_config.h /usr/include/wchar.h \
/usr/include/bits/wchar.h /usr/include/gconv.h \
/usr/lib/gcc-lib/i386-linux/2.95.4/include/stdarg.h \
/usr/include/bits/stdio_lim.h \
/usr/include/bits/sys_errlist.h \
/usr/include/stdlib.h /usr/include/sys/types.h \
/usr/include/time.h \
/usr/include/endian.h /usr/include/bits/endian.h \
/usr/include/sys/select.h /usr/include/bits/select.h \
/usr/include/bits/sigset.h /usr/include/bits/time.h \
/usr/include/sys/sysmacros.h /usr/include/alloca.h \
/usr/include/string.h calc.h func.h
fact.o: fact.c calc.h
comb.o: comb.c calc.h
sum.o: sum.c calc.h
dummy.o: dummy.c
```

Ecco fatto che abbiamo generato tutte, ma proprio tutte, le dipendenze per il nostro progetto! Si noti come si siano generate anche le dipendenze relative ai file di sistema (come `stdio.h`) che prima non erano considerate.

Il carattere `\` serve per dire a make che la linea corrente prosegue in quella successiva e ad evitare quindi messaggi di errore, infatti una dipendenza di un file deve stare su una sola linea.

Vediamo allora come poter utilizzare questa nuova funzionalità. Se modifichiamo il nostro makefile così:

```
TARGET = calc
OBJECTS = calc.o fact.o comb.o sum.o dummy.o

all : .depend $(TARGET)

.depend :
$(CC) $(CFLAGS) -M calc.c fact.c comb.c sum.c
dummy.c > .depend

include .depend

$(TARGET): $(OBJECTS)
```

Otteniamo:

```
$ make
Makefile:9: .depend: No such file or directory
cc -M calc.c fact.c comb.c sum.c dummy.c > .depend
cc -c -o calc.o calc.c
cc -c -o fact.o fact.c
cc -c -o comb.o comb.c
cc -c -o sum.o sum.c
cc -c -o dummy.o dummy.c
cc calc.o fact.o comb.o sum.o dummy.o -o calc
```

Il nostro progetto è stato correttamente ricompilato! Notate come in questo caso sia stata eseguita la generazione automatica delle dipendenze. Sicuramente non vi sarà sfuggito quel *"No such file or directory"* nell'output del comando appena dato. All'atto dell'invocazione di **make** il file **.depend** non esiste ancora e questo viene creato durante l'elaborazione del makefile.

Per evitare di ottenere tale messaggio possiamo ricorrere ad una operazione condizionale e cioè ad una operazione che ci permette di eseguire un certo comando se una data condizione è vera o meno. La nostra direttiva **include** diventa allora:

```
ifeq ($wildcard .depend, .depend)
include .depend
endif
```

L'operazione condizionale viene eseguita dal costrutto **ifeq endif** che è un po' simile a quelli usati all'interno del preprocessore C. In pratica se l'espressione **\$wildcard .depend** è uguale a **.depend** allora si esegue l'**include** altrimenti no. Come si legge l'espressione appena citata? Il comando **\$wildcard** non fa altro che restituire la lista di tutti i file nelle directory di lavoro che corrispondono all'espressione che segue, è un po' come il comando **ls**, poiché l'espressione è proprio il nome di un file (**.depend**, appunto), senza caratteri jolly, allora il risultato è **.depend** o la stringa vuota (che indica che il file non esiste).

Per come funziona, nello specifico, il comando **wildcard** rimando i lettori alle pagine del manuale.

Ecco quindi che, una volta aggiunta la direttiva **ifeq**, se eseguiamo il comando di prima otteniamo il giusto funzionamento:

```
$ make
cc -M calc.c fact.c comb.c sum.c dummy.c > .depend
[...]
```

Manipolare le variabili

Facciamo il punto della situazione, il nostro makefile è:

```
TARGET = calc
OBJECTS = calc.o fact.o comb.o sum.o dummy.o

all : .depend $(TARGET)

.depend :
$(CC) $(CFLAGS) -M calc.c fact.c comb.c
sum.c dummy.c >> .depend

ifeq ($wildcard .depend, .depend)
include .depend
endif

$(TARGET): $(OBJECTS)

clean:
rm -f *.o $(TARGET) .depend
```

Se, da un lato, abbiamo automatizzato la generazione delle dipendenze, dall'altro però, abbiamo tirato in ballo nuovamente la lista dei file sorgente. Quello che vogliamo ora è fare in modo tale da specificare una sola volta i file da processare e ottenere che **make** riesca a ricavarsi da solo tutti i nomi dei file che derivano da essi.

Per ottenere questo risultato possiamo allora riscrivere il nostro makefile come segue:

```
TARGET = calc
SRCS = calc.c fact.c comb.c sum.c dummy.c

all : .depend $(TARGET)

.depend :
$(CC) $(CFLAGS) -M $(SRCS) >> .depend

ifeq ($wildcard .depend, .depend)
include .depend
endif

$(TARGET): $(SRCS:.c=.o)
```

Come si vede ho rimosso la lista dei file oggetto ed ho aggiunto quella dei file sorgenti (più logico), in questo modo il comando delle dipendenze automatiche diventa:



```
.depend :  
$(CC) $(CFLAGS) -M $(SRCS) >> .depend
```

La parte più interessante, però, è questa:

```
$(TARGET) : $(SRCS:.c=.o)
```

Con l'espressione `$(SRCS:.c=.o)` si dice a **make** di considerare i nomi nella variabile `SRCS` e sostituire a tutti l'estensione `.c` con `.o`, ecco che allora questa linea diventa equivalente a quella che avevamo prima:

```
OBJECTS = calc.o fact.o comb.o sum.o dummy.o  
...  
$(TARGET) : $(OBJECTS)
```

In questo modo abbiamo allora eliminato dal nostro makefile qualsiasi informazione doppia, cioè ora per inserire o togliere un nuovo file al progetto basta modificare solo il valore della variabile `SRCS` e il tutto continua a funzionare correttamente!

Le variabili automatiche

Abbiamo sin qui parlato molto di *implicit rules* (o regole implicite), ma non abbiamo ancora visto come queste vengono definite o, più interessante, come se ne possono definire di nuove.

Vediamo un semplice esempio; supponiamo di avere una serie di file Postscript e di voler scrivere un makefile per passare velocemente dal formato Postscript al formato PDF. Per ottenere questo risultato, andrà usato il comando `ps2pdf` il quale prende come primo argomento il file Postscript da convertire e come secondo argomento il nome del file PDF da generare.

Il nostro makefile allora sarebbe:

```
all : doc1.pdf doc2.pdf doc3.pdf  
  
doc1.pdf : doc1.ps  
    ps2pdf doc1.ps doc1.pdf  
  
doc2.pdf : doc2.ps  
    ps2pdf doc2.ps doc2.pdf  
  
doc3.pdf : doc3.ps  
    ps2pdf doc3.ps doc3.pdf
```

che, se eseguito, darà questo output:

```
$ make  
ps2pdf doc1.ps doc1.pdf  
ps2pdf doc2.ps doc2.pdf  
ps2pdf doc3.ps doc3.pdf
```

È chiaro che anche in questo caso se dobbiamo inserire

uno o più nuovi documenti la cosa diventa alquanto noiosa poiché dovremo riscrivere una serie di comandi per ogni nuovo file.

Quello che vogliamo ottenere è un po' quello che ricercavamo nei precedenti paragrafi e cioè avere un makefile nel quale basta modificare solo un punto per far processare più di un file. Nello specifico ci piacerebbe poter modificare solo la lista dei file da convertire ed ottenere un makefile che continui a funzionare bene.

Per arrivare a questo occorre, innanzi tutto, introdurre il concetto di *variabile automatica*.

Una variabile automatica non è altro che una variabile che viene automaticamente definita da make quando esegue una particolare regola. Se noi modifichiamo il nostro makefile come segue:

```
doc1.pdf : doc1.ps  
    ps2pdf $^ $@  
  
doc2.pdf : doc2.ps  
    ps2pdf $^ $@  
  
doc3.pdf : doc3.ps  
    ps2pdf $^ $@
```

Diciamo a make che il file `doc1.pdf` si ottiene dal file `doc1.ps` eseguendo il comando `ps2pdf` con il primo parametro uguale al valore specificato nelle dipendenze della regola stessa e come secondo parametro il valore del suo target. In pratica queste variabili ci permettono di eseguire dei riferimenti ai valori specificati nella regola che le contiene.

Se rilanciamo make otteniamo sempre:

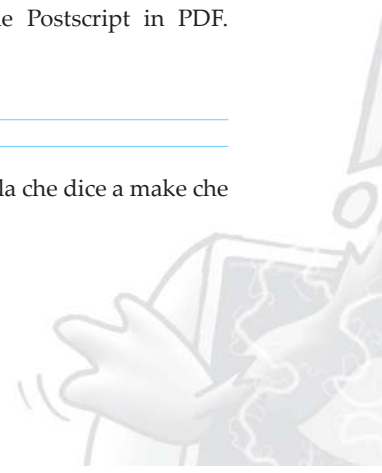
```
$ make  
ps2pdf doc1.ps doc1.pdf  
ps2pdf doc2.ps doc2.pdf  
ps2pdf doc3.ps doc3.pdf
```

All'interno di make esistono altre variabili automatiche, ma non le riporterò in questo articolo e rimando il lettore più curioso al manuale di make.

Se ora osserviamo il makefile vediamo che i comandi da eseguire per generare i vari file PDF sono sempre gli stessi e quindi ci piacerebbe poter semplificare ulteriormente il nostro makefile, questo è possibile utilizzando una nuova regola ad-hoc per convertire i file Postscript in PDF. Modificando il makefile così:

```
%.pdf : %.ps  
    ps2pdf $^ $@
```

definiamo appunto una nuova regola che dice a make che



un file con estensione **.pdf** si ottiene dal file con lo stesso nome, ma con estensione **.ps** lanciando il programma **ps2pdf** con primo parametro il file Postscript e come secondo il file PDF. In questo caso quindi il carattere **%** è un carattere jolly.

Ancora, se lanciamo **make** otteniamo, ancora una volta:

```
$ make
ps2pdf doc1.ps doc1.pdf
ps2pdf doc2.ps doc2.pdf
ps2pdf doc3.ps doc3.pdf
```

Proprio quello che volevamo!

Manipolare le variabili

Bene, abbiamo appena visto come poter definire una o più variabili e come sia possibile utilizzare il contenuto delle stesse. Ora voglio mostrarvi altre situazioni in cui l'uso delle variabili è utile e come si può agire per modificarle in maniera automatica.

Riprendiamo il **makefile** che abbiamo ottenuto fino a questo punto:

```
all : doc1.pdf doc2.pdf doc3.pdf

%.pdf : %.ps
    ps2pdf $^ $@

clean :
    rm *.pdf
```

Supponiamo di volerlo modificare in modo tale che non sia necessario dire a **make** quali sono i file da convertire, ma che questo lo “deduca” in maniera automatica. Per fare ciò modifichiamo il **makefile** come segue:

```
SRCS = doc1.ps doc2.ps doc3.ps

all : $(SRCS: %.ps= %.pdf)

%.pdf : %.ps
    ps2pdf $^ $@
```

Dal punto di vista del risultato non è cambiato nulla, ma se ora definiamo la variabile **SRCS** come:

```
SRCS = $(wildcard *.ps)
```

otteniamo proprio quello che volevamo!

Il comando **wildcard**, come abbiamo visto, funziona un po' come il comando **ls** e quindi quando eseguiamo il comando **make** la variabile **SRCS** viene inizializzata con la lista di tutti i file con estensione **.ps** presenti nella directory corrente, infatti:

```
$ make
ps2pdf doc1.ps doc1.pdf
ps2pdf doc2.ps doc2.pdf
ps2pdf doc3.ps doc3.pdf
$ cp doc3.ps doc4.ps
$ make
ps2pdf doc4.ps doc4.pdf
```

Vediamo altri esempi. Torniamo ai file C del precedente articolo e vediamo come sia possibile gestire la definizione della variabile **DEBUG** che ci permette di controllare la fase di debugging del nostro progetto. Riprendiamo il file **calc.h**:

```
#include <stdio.h>

#ifdef DEBUG
#define PDEBUG(fmt, args...) fprintf(stderr, fmt, ## args)
#else
#define PDEBUG(fmt, args...)
#endif

#define START_NUMBER 3
#define MAX_NUMBER 15
```

Da qui vediamo che per ottenere le stampe di debugging tramite la macro **PDEBUG** occorre definire appunto la variabile **DEBUG**; per fare questo possiamo allora modificare il nostro **makefile** come segue:

```
# DEBUG = yes

SRC = calc.c fact.c comb.c sum.c
TARGET = calc
CFLAGS = -Wall

ifeq ($(DEBUG),yes)
    CFLAGS += -g -gdb -DDEBUG
endif

all: .depend $(TARGET)

$(TARGET): $(SRC:.c=.o)

depend .depend dep:
    $(CC) $(CFLAGS) -M $(SRC) > $@

ifeq (.depend,$(wildcard .depend))
include .depend
endif
```

In questo modo abbiamo ottenuto che, lasciando tutto così com'è, otteniamo una normale compilazione:

```
$ make
cc -Wall -M calc.c fact.c comb.c sum.c > .depend
[...]
```

Se togliamo il commento dalla prima linea (togliendo il carattere **#**) allora otteniamo:



```
$ make
cc -Wall -g -gdb -DDEBUG -M calc.c fact.c comb.c
sum.c > .depend
[...]
```

Si noti come make sostituisca i valori corretti della variabile **CFLAGS** quando la variabile **DEBUG** viene impostata al valore **yes**, infatti il costrutto:

```
ifeq ($(DEBUG),yes)
    CFLAGS += -g -gdb -DDEBUG
endif
```

non fa altro che aggiungere al valore corrente di **CFLAGS** i nuovi parametri di compilazione, questo grazie all'operatore **+=** che è del tutto simile a quello utilizzato, ad esempio, nel linguaggio C. Si noti, però, che ho usato la parola simile e non uguale perché una differenza c'è. Dovete sapere che make non sostituisce il valore di una variabile immediatamente (a meno che non lo si forzi a farlo), ma lo fa quando questa viene usata. Facciamo un esempio, se scrivo il makefile:

```
C = $(B) 3
B = $(A) 2
A = 1

all :
    echo $(C)
```

e lo eseguo:

```
$ make
echo 1 2 3
1 2 3
```

come vedete ottengo la stampa corretta della variabile **C**, questo perché make ha sostituito i valori della variabili solo quando queste sono state utilizzate.

Questo metodo di funzionamento viene implementato in make con l'uso di variabili di diverso tipo; nello specifico queste vengono chiamate *variabili immediate*, quelle cioè il cui valore viene assegnato alla definizione e le *variabili differite* e cioè quelle il cui valore viene assegnato quando vengono utilizzate.

Utilizzando il segno **=** definisco variabili differite, ma se voglio definire variabili immediate devo usare il segno **:=**. Nell'esempio fatto prima, allora, sostituendo tutti i segni **=** con **:=**, ottengo:

```
$ make
echo 3
3
```

dato che quando vado a definire la variabile **C** le altre due

ancora non esistono, quindi ad esse viene sostituita la stringa vuota. Avere a disposizione questi due tipi di variabili è comodo, ma alle volte può essere un problema; se, ad esempio, nel makefile del nostro progetto sostituisco la linea:

```
CFLAGS += -g -gdb -DDEBUG
```

con:

```
CFLAGS = $(CFLAGS) -g -gdb -DDEBUG
```

ottengo:

```
$ make DEBUG=yes
Makefile:16: *** Recursive variable `CFLAGS'
references itself (eventually). Stop.
```

Infatti make si trova una definizione ricorsiva che potrebbe definire **CFLAGS** infinite volte e quindi restituisce un errore.

Un modo per ovviare a questo potrebbe essere quello di sostituire la definizione di **CFLAGS** così:

```
CFLAGS := $(CFLAGS) -g -gdb -DDEBUG
```

in questo modo si forza l'assegnamento immediato e si evitano le ricorsioni infinite. Quest'ultimo esempio è equivalente all'uso di **+=**.

Comunque la gestione delle variabili immediate e differite è molto più complessa da come l'ho presentata: rimando i più interessati ad approfondire l'argomento sulle pagine del manuale.

Le funzioni di manipolazione delle stringhe

Sempre sull'argomento di come sia possibile manipolare le variabili di make, vorrei dire due parole anche sulle funzioni di elaborazione delle stringhe, molto importanti perché, sostanzialmente, un makefile manipola stringhe. Non presenterò tutte le funzioni, vedremo solo le più importanti. Per prima vediamo la funzione, **filter** che dati una stringa ed un pattern, toglie tutte le parole della stringa che non corrispondono al pattern dato, ad esempio:

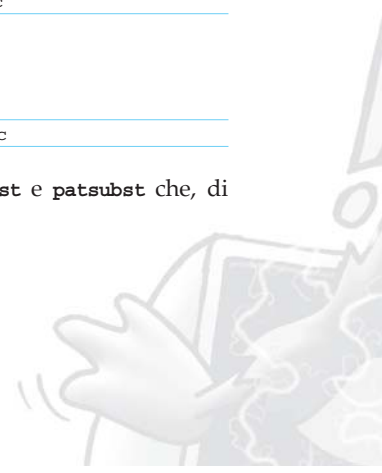
```
SRCS = calc.c comb.c fact.c sum.c calc.h func.h

calc : $(SRCS)
    cc $(filter %.c, $(SRCS)) -o calc
```

darà:

```
$ make
cc calc.c comb.c fact.c sum.c -o calc
```

Altre due utili funzioni sono **subst** e **patsubst** che, di



solito, vengono utilizzate per creare una serie di direttive **include** data una lista di directory separate da due punti (:), ad esempio se voleste utilizzare il contenuto della vostra variabile `PATH` per creare una serie di direttive **include** per il vostro compilatore il makefile sarebbe:

```
SRCS = calc.c comb.c fact.c sum.c calc.h func.h
INCLUDE_LIST = $(patsubst %, -I%, $(subst :, , $(PATH)))

calc : $(SRCS)
cc $(filter %.c, $(SRCS)) $(INCLUDE_LIST) -o calc
```

che, se eseguito, darà:

```
$ echo $PATH
/usr/bin:/bin:/usr/bin/X11:/usr/games
$ make
cc calc.c comb.c fact.c sum.c -I/usr/bin -I/bin
-I/usr/bin/X11 -I/usr/games -o calc
```

Definire le variabili da linea di comando

Ancora molte cose ci sarebbero da dire su `make`, ma prima di chiudere questa lunga lezione vorrei parlare di come si possono definire le variabili direttamente da linea di comando. Abbiamo visto che commentando o decommentando la prima linea del nostro makefile (la direttiva `DEBUG = YES`) era possibile diversificare la fase di compilazione del nostro progetto. Il fatto, però, di dover tutte le volte editare il makefile è scoccante, a noi piacerebbe poter dire a `make`, direttamente da linea di comando, se deve compilare con le opzioni di debug o meno. Tutto questo è possibile, invocando `make` così:

```
$ make DEBUG=yes
cc -Wall -ggdb -DDEBUG -M calc.c fact.c comb.c
sum.c > .depend
[...]
```

In questo modo `make`, prima di eseguire il makefile, si preoccupa di definire la variabile `DEBUG`. In questo modo vedete che impostare delle variabili diventa molto più semplice e comodo.

Proviamo a modificare ancora il makefile (riporto solo le prime linee):

```
DEBUG = no

SRC = calc.c fact.c comb.c sum.c
TARGET = calc
CFLAGS = -Wall
```

e diamo di nuovo il comando di prima:

```
$ make DEBUG=yes
cc -Wall -ggdb -DDEBUG -M calc.c fact.c comb.c
sum.c > .depend
[...]
```

Otteniamo lo stesso risultato nonostante la variabile `DEBUG` sia stata definita all'interno del makefile; questo perché `make` privilegia la definizione delle variabili dalla linea di comando.

Questa caratteristica, di per sé, potrebbe essere una penalizzazione; supponiamo di voler definire i flag di compilazione dalla linea di comando, faremmo:

```
$ make CFLAGS="-O2" DEBUG=yes
cc -O2 -M calc.c fact.c comb.c sum.c > .depend
[...]
```

Come vedete in questo caso abbiamo perso completamente i flag di debugging! Questo accade perché `make` considera **solo** le impostazioni dalla linea di comando; per ovviare a questo occorre usare la direttiva **override** per la variabile `CFLAGS`, in questo modo:

```
ifeq ($(DEBUG),yes)
override CFLAGS += -ggdb -DDEBUG
endif
```

Così diciamo a `make` che deve utilizzare le impostazioni da linea di comando, ma senza sovrascrivere quelle presenti nel makefile, infatti ora otteniamo:

```
$ make CFLAGS="-O2" DEBUG=yes
cc -O2 -ggdb -DDEBUG -M calc.c fact.c comb.c
sum.c > .depend
[...]
```

che è appunto quello che volevamo.

Termina qui questo ciclo di articoli, anche se ci sarebbe stato ancora altro da dire su `make`: purtroppo lo spazio è tiranno!

Spero di aver attirato la vostra curiosità ed il vostro interesse su questo stupendo strumento di sviluppo: vi ho presentato molti esempi sulla compilazione di codice C (deformazione professionale), ma vi ricordo che `make` può essere utilizzato in molti altri campi in cui, in generale, ci siano da generare uno o più file a partire da altri, tenendo conto delle dipendenze tra di essi.

