# Capitolo 0

# Make

Il programma make serve per mantenere aggiornate le versioni dei programmi, e può essere utilizzato un po' per tutti i linguaggi.

Utilizzando delle regole definite in un "makefile", ricostruirà gli eseguibili determinando qualli moduli hanno bisogno di essere ricompilati ogni qualvolta si modifica un file sorgente. Infatti, quando un programma è costituito da un gran numero di file sorgenti, spesso questi file dipendono l'uno dall'altro in modi molto complessi. Quanto le modifiche sono fatte ad un file da cui dipendono altri file, tutti questi file avranno bisogno di essere ricompilati. Spesso determinare esattamente quali file hanno bisogno di essere aggiornati può diventare un compito complesso e impegnativo. L'alternativa è quella di ricompilare tutto senza preoccuparsi se è effettivamente necessario. ma questa soluzione è chiaramente uno spreco di CPU e di tempo.

Il programma make aiuta ad automatizzare il processo di compilazione e di linking del programma durante la fase di sviluppo. Si definiranno le dipendendenze presenti all'interno del programma specificando le regole appropriate, dopodichè quando sarà necessario aggiornare l'eseguibile, make si prenderà cura dei dettagli.

## 12.1 Utilizzo di Make

Quando viene invocato il comando make, le regole di dipendenza sono lette da un file chiamato makefile o Makefile nella directory corrente.

È possibile specificare il nome di un altro file che contiene queste informazioni utilizzando il flag -f.

Un makefile molto semplice ha la forma seguente:

```
target-file:  lista_di_file
TAB commandi_di_costruzione
```

dove `target-file` è il nome del file che dipende dai file elencati nella lista `lista_di_file`.

I comandi da utilizzare per la costruzione del programma devono iniziare con un `TAB` ed includere tutti i comandi da immettere per la costruzione del file `target-file`.

Ad esempio supponiamo che il file eseguibile `usacomplessi` dipenda dai file object `usacomplessi.o complessi.o`. Una linea di dipendeneze corretta per un `makefile` potrebbe essere:

```
usacomplessi:  usacomplessi.o complessi.o
        cc -lm -o usacomplessi complessi.o usacomplessi.o
```

Dando il comando di costruzione precedente, verrà creato `usacomplessi`, ovvero il "target-file".

Spesso saranno necessarie ulteriori linee di dipendenza; ad esempio potremmo specificare come costruire i file .o. In altre parole, i prerequisiti di una linea di idipendenza possono apparire come un "target" in un'altra linea di dipendenza.

I `makefile` possono includere commenti, delimitati dal carattere #. Il carattere \ può essere usato alla fine di una linea per continuare un caomando sulla linea successiva. Se necessario più comandi possono essere utilizzati per costruire uno specifico file "target". Ciascun comando di costruzione dovrà essere collocato sulla propria linea, con un `tab` di indentazione.

Per tenere aggiornato un insieme di sottoprogrammi, `make` confronta la data in cui sono stati modificati i file "target" con la data dei dei file prerequisiti. Ciascun file prerequisito che ha una data più recente che il suo file "target" forza la ricostruzione del file `target`. Ovviamante se il file "target" non esiste questo sarà costruito indipendentemente dalle date dei file prerequisiti.

Di default, verrà costruito il primo target file che appare nel `makefile`. Gli altri target saranno controllati solo se sono un prerequisito del target iniziale. A parte il fatto che il primo target è quello di defaultm l'ordine dei target non conta: `make` li costruirà nell'ordine necessario.

Nella costruzione dei target, le linee di comando sono eseguite una alla volta, ciascuna con la propria subshell. Pertanto in un `makefile` l'effetto di un comando di shell (quale un `cd`), non si estenderà automaticamente oltre il carattere di fine riga.

Si può controllare un `makefiles` utilizzando l'opzione `-n`. Questa opzione fa sì che `make` mostri a video (senza eseguirli) tutti i comandi che sarebbero stati eseguiti per costruire il programma. I comandi saranno scritti sullo standard output.

## 12.2 Regole Interne

Per semplificare un makefile possiamo affidarci ad alcune regole di dipendenza e ad alcuni comandi di costruzione impliciti (dette regole interne).

Un insieme di regole interne consente a make di capire come costruire un file con uncerto suffisso a partire da un file con un altro suffisso.

Se consideriamo ancora il makefile visto in precedenza, possiamo osservare come ci sia una unica linea di dipendenza che specifica come costruire l'eseguibile a partire dai file oggetto: non abbiamo specificato da nessuna parte come costruire questi file .o e non compaiono come target in nessuna linea del makefile. Perciò make cercherà nella directory corrente quale codice sorgente potrà essere utilizzato per costruire i file oggetto utilizzando i comandi di costruzione predefiniti.

Nel nostro esempio i comandi di default che saranno utilizzati (per trasformare un sorgente .c in un file oggetto .o) saranno della forma:

```
cc -O -c complessi.c
cc -O -c usacomplessi.c
```

Ovviamente se questo non è ciò che desideriamo dovremo modificare il makefile per inserirci le regole di dipendenza ed i comandi di costruzione desiderati.

## 12.3 Macro

Il comando make supporta un meccanismo sofisticato di macro. In modo simile alle variabili di shell di Unix, le macro possono essere utilizzate per sostituire un certo insieme di sequenze di caratteri con altre sequenze. Nel nostro makefile potremo specificare una macro con una riga del tipo:

```
nomemacro = nuovastringa
```

La stringa nuovastringa conterrà tutti i caratteri che vogliamo sulla linea fino ad un eventuale carattere di commento (o alla fine della riga stessa).

Le macro possono essere prolungate su linee successive utilizzando il carattere di continuazione \.

Dopo la definizione di una macro, make rimpiazzerà ogni occorrenza dei caratteri $(nomemacro) presenti nel makefile con la stringa nuovastringa assegnata.

Ad esempio potremmo modificare il nostro makefile di esempio come segue:

```
LDFLAGS = -lm
usacomplessi:  usacomplessi.o complessi.o
        cc $(LDFLAGS) -o usacomplessi complessi.o usacomplessi.o
```

In questo caso abbiamo definito una macro che serve per passare alcuni parametri al compilatore. Inserendoli in una macro è così possibile cambiarli in un unico posto piuttosto che in ciascun singolo comando di compilazione.

## 12.4 Completamento dell'esempio

La regola generale di compilazione per trasformare un sorgente C in un file oggetto è la seguente:

```
.c.o:
          $(CC) -c $(CFLAGS) -o $@ $<
```

Analizziamo nel dettaglio questa regola:

- la riga `.c.o` indica che la regola successiva descriverà come passare dal suffisso .c al suffisso .o
- la macro $(CC) ha un valore di default che è `cc`. Se non vogliamo utilizzare il compilatore C di deafult potremo quindi semplicemente aggiungere una definizione di macro del tipo: `CC = gcc`
- la macro `$(CFLAGS)` verrà utilizzata per passare al compilatore opportuni parametri di compilazione. Ad esempio potremo definire: `CFLAGS = -Wall -O2`
- la variabile `$@` contiene il nome del file target di questa regola. Quindi se la nostra regola ad esempio è utilizzata per costruire automaticamente il file `complessi.o` partendo dal file `complessi.c`, `$@` varrà `complessi.o`
- la variabile `$<` contiene il nome del primo file di dipendenza. In una regola implicita questo file è la prima dipendenza aggiunta automaticamante dalla regola. Nell'esempio della costruzione automatica del file `complessi.o` partendo dal file `complessi.c`, `$<` varrà `complessi.c`

Compresa quindi questa regola potremo aggiungere al nostro `makefile` la definizione `CFLAGS = -Wall -O2`, così ogni volta che dovremo compilare un sorgente C per ottenere un file oggetto utilizzeremo questi parametri di compilazione.

Non avremo da aggiungere altro alle regole potrendo sfruttare le regole di compilazione automatica.

Potremo poi aggungere una regola per rimuovere tutti i file intermedi che non servono più una volta terminata la compilazione. Tradizionalmente questa regola si chiama `clean` ed avrà la forma seguente:

```
clean:
        rm -f a.out
        rm -f *.o *~ core
```

Questa regola sarà invocata passando il suo nome come argomento del comando `make`, ovvero dando il comando: `make clean`. Il risultato sarà l'eleiminazione di tutti i file .o, dei file che terminano in ~, dei file `core` ed `a.out`.