Makefile

Introduzione

Struttura

Regole

I makefile

- **Un makefile è un elenco di regole che permettono**
  - Mantenere allineati più file collegati da dipendenze
  - Aggiornare i file che risultano essere obsoleti

- **Le regole contenute in un makefile sono eseguite dal comando `make`**
  - La varsione originale del comando **`make`** è obsoleta e molto meno flessibile delle versioni attuali
  - La versione GNU di **`make`** è chiamata **`gmake`**

- **L'uso più frequente del comando `make` e dei makefile è legata alla ricompilazione condizionale di porzioni di grossi programmi**

make

make [*options*] [*goal...*]

- **Applica le regole di un makefile**
  - Se non specificato, **make** cerca nell'ordine **GNUmakefile, makefile, Makefile**
  - Se non è specificato alcun goal, **make** esegue le regole relative al primo goal
- **Lo stato di uscita può essere**
  - **0** Se ha avuto successo
  - **1** Se esiste almeno un goal da aggiornare
  - **2** Se si sono verificati errori
- **Le principali opzioni sono le seguenti**
  - **-C *dir*** Esegue il comando **cd *dir*** prima di iniziare
  - **-f *file*** Specifica un makefile diverso da quelli di default
  - **-i** Ignora gli errori e procede
  - **-j [*n*]** Esegue ***n*** job in parallelo, Se n è omesso, esegue quanti job possibili
  - **-*n*** Stampa i comandi richiesti per aggiornare il goal senza eseguirli
  - **-q** Ritorna 1 o 0 a seconda che il goal richeda l'aggiornamento di qualche file
  - **-t** Eesegue il comando touch per i file da aggiornare

Struttura di un makefile

- **Un makefile contiene 5 tipi di regole**

- **Regole esplicite (explicit goals)**
  - Specificano come aggiornare un file specifico

- **Regole implicite (implicit goals)**
  - Specificano come aggiornare una classe di file

- **Definizioni di variabili**
  - Assegna un valore ad una variabile

- **Direttive**
  - Indicano quando leggere altri makefile e/o qunado ignorare alcune porzioni del makefile

- **Commenti**
  - Inseriti per chiarezza, iniziano con il carattere **#**

Struttura di un goal

```
target : prerequisites
    commands
```

- **In cui**
  - ***target***
    - Indica i file che saranno costruiti dalla regola
  - ***prerequisites***
    - Indica i file da cui il file target dipendono
  - ***commands***
    - I comandi che costruiscono i file del target a partire dai file dei prerequisiti

Goal espliciti

```
tfile: pfile […]
    commands
```

- **Un goal esplicito nomina in modo esplicito tutti i nomi dei file coinvolti**
  - ***tfile***
    - Il file target
  - ***pfile…***
    - I file prerequisiti

- **I goal espliciti offrono una limitata flessibilità**
  - Si riferiscono a singoli file
  - Devono essere scritti per ogni file anche se l'elaborazione richiesta è identica per più file

Esempio di goal esplicito

- **Goal esplicito**

```
# Compiles a single file

main.o: main.c
    gcc -c main.c -o main.o
```

- **Questo goal svolge le seguenti funzioni**
  - Determina la data di ultima modifica del file prerequisito **`main.c`**
  - Se il target **`main.o`** non esiste o se la sua data di ultima modifica è meno recente di quella del file prerequisito
    - Esegue il comando, cioè compila il sorgente
  - Altrimenti
    - Non esegue alcun comando

Esempio di goal esplicito

- **Goal esplicito**

```
# Removes all object files

clean:
    rm -f *.o
```

- **Questo goal svolge le seguenti funzioni**
  - Dato che non vi sono prerequisiti esegue il comando
  - Il comando è eseguito dalla shell
    - Nella shell avviene l'espansione del glob `*.o`
    - Gli argomenti espansi sono passati al comando `rm`
  - Il comando non crea il file target clean
    - Questo goal sarà sempre eseguito

Goal impliciti

- **Un goal implicito**
  - E' una regola che si riferisce ad una classe
  - Non ad un singolo file
- **Una classe di file**
  - E' identificata dall'estensione del nome dei file che vi appartengono

- **Un nome di file è da vedersi nella forma**

  `[p]stem[s].ext`

- **In cui**
  - ***p*** Prefisso comune a tutti i file della classe
  - ***stem*** Porzione univoca del nome
  - ***s*** Suffisso comune a tutti i file della classe
  - ***ext*** Estensione comune a tutti i file della classe

Goal impliciti

```
[tp]%[ts].text: [pp]%[ps].pext [...]
    commands
```

- **I file su cui operare sono definiti in base allo stem**
  - Il carattere `%` (variabile speciale) indica lo stem
  - Quando un file ha match con il target, la variabile speciale `%` assume il valore dello stem
  - Lo stem è sostituito al posto del carattere `%`
    - In onguno dei prerequisiti
    - Per formare i nomi effettivi dei file

- **Esistono regole predefinite per diverse classi di file**
  - Sono raccolte in un file di configurazione globale
  - Possono essere aggiunte nel file **`.makefile`**
    - Presente nella home directory di un utente

Goal impliciti

- **Nei comandi corrispondenti ad un goal implicito**
  - E' necessario riferirsi ai nomi di file effettivi
    - Target
    - Prerequisiti

- **Si usano le seguenti variabili speciali**
  - `$@` Il target
  - `$<` Il primo prerequisito
  - `$?` Tutti i prerequisiti più recenti del target
  - `$^` Tutti i prerequisiti
  - `$*` Lo stem assegnato alla variabile speciale `%`

Esempio di goal implicito

- **Goal implicito**

```
# Compiles all source files

%.o: %.c
    gcc -c -g $< -o $@
```

- **Questo goal svolge le seguenti funzioni**
  - Per ogni file che soddisfa la forma **%.o**
    - Assegna il nome del file allo stem
  - Costruisce il nome effettivo del prerequisito aggiungendo allo stem l'estensione **.c**
  - Assegna il nome del target alla variabile **$@**
  - Assegna il nome del prerequisito alla variabile **$<**
  - Esegue il comando specificato sostituendo alle variabili speciali i valori appena assegnati

Esempio di goal implicito

- **Supponendo di voler creare il file `foo.o`, il goal visto si comporta come segue**
  - Assegna a `%` il valore `foo`
  - Assegna a `$@` il valore `foo.o`
  - Assegna a `$<` il valore `foo.c`
- **La regola che ne risulta è quindi**

```
foo.o: foo.c
    gcc -c -g foo.c -o foo.o
```

- **Infine esegue il comando di compilazione**

Esempio di goal implicito

- La stessa regola può essere scritta come

```
# Compiles all source files
# Uses the stem substitution with the variable $*

%.o: %.c
    gcc -c -g $*.c -o $*.o
```

- La prima forma mostrata è tuttavia da preferirsi
  - Più sintetica
  - Più chiara

Goal speciali

- **I goal speciali sono costrutti utilizzati in situazioni particolari**
  - Per forzare un dato comportamento
  - Esistono tre tipi di goal speciali

- **Phony targets**
  - Realizzano regole con target fittizi

- **Force targets**
  - Realizzano regole di forzatura

- **Empty target**
  - Tengono traccia del momento in cui alcune operazioni sono svolte

Phony targets

```
.PHONY: target [traget...]
```

- **Un phony target**
  - Ha come prerequisiti altri target
  - Deve essere eseguito sempre
- **Nell'esempio precedente**
  - Se il file **`clean`** dovesse esistere
    - Il comando **`rm`** non sarebbe mai eseguito
    - Infatti il target **`clean`** non ha prerequisiti che lo potrebbero rendere obsoleto
  - In alcuni casi questo comportamento è dannoso
- **Una migliore soluzione è quindi la seguente**

```
# Removes all object files
# Using phony targets makes the rule safer

.PHONY: clean
clean:
    rm -f *.o
```

Force targets

FORCE

- **Un force target**
  - E' un target vuoto
  - E' utilizzato come prerequisito
  - Deve essere eseguito sempre
- **È mantenuto come alternativa ai phony targets**
  - Nessun comando crea il file **FORCE**
    - Quindi i goal che lo contengono come prerequisito saranno sempre eseguiti
- **Una versione equivalente ma meno efficiente dell'esempio precedente è**

```
# Removes all object files
# Using FORCE is less efficient than using phony

FORCE:
clean: FORCE
    rm -f *.o
```

Empty targets

- **Un empty target**
  - È usato per tenere traccia del momento in cui un certo goal è estato eseguito
  - Crea effettivamente un file, generalmente vuoto
  - È analogo al meccanismo dei phony targets

- **Un esempio chiarisce come**
  - Stampare un dato file
  - Se e solo se è stato modificato dall'ultima stampa

```
# Print main.c only if changed since last print
# The file print is an empty target

print: main.c
    lpr -p main.c
    touch print
```

Target multipli

```
target [...]: prerequisites
    commands
```

- **Questa regola**
  - È attivata ogniqualvolta
    - Uno qualsiasi dei target è più vecchio di uno qualsiasi dei prerequisiti
  - Indica implicitamente che i comandi saranno tali da costruire tutti i target

- **Sono generalmente usate quando un comando genera più di un file di uscita**
  - Ad esempio il genratore di parser bison

Esempio di target multipli

- Traget multiplo esplicito

```
# Genrates a specific parser from its grammar definition
# Explicit goal

grammar.tab.c grammar.tab.h: grammar.y
    bison -d grammar.y
```

- Traget multiplo implicito

```
# Genrates a generic parser from its grammar definition
# Implicit goal

%.tab.c %.tab.h: %.y
    bison -d $<
```

Definizioni di variabili

- **All'interno di un makefile è possibile definire variabili generiche**
  - Per facilitare la lettura/scrittura del makefile
  - Per migliorare la manutenibilità del makefile

- **Una variabile si definisce mediante la sintassi**

  `var = value`

- **In cui**
  - ***`var`*** Nome della variabile
  - ***`value`*** Valore della variabile

- **Si accede al valore di una variabile mediante**

  `$(`*`var`*`)`

Definizioni di variabili

- **Alcuni esempi di assegnamenti di variabile**
- ***`var = string`***
  - Assegna una stringa ad una variabile
- ***`var = item1 item2 … itemN`***
  - Assegna una lista di stringhe ad una variabile
- ***`var = $(othervar)`***
  - Copia una variabile in un'altra
- ***`var = prefix$(othervar)suffix`***
  - Costruisce una variabile per concatenazione di parti costanti e altre variabili
- ***`var = $(function args)`***
  - Assegna ad una variabile il risultato di una funzione

Variabili predefinite

- **Alcune variabili predefinite controllano**
  - Il comportamento del programma make
  - I tool di compilazione usati
  - Le opzioni standard per i toool usati

- **Variabili per il programma `make`**
  - **`MAKE`**
    - Nome dell'eseguibile di **`make`**
  - **`MAKEFLAGS`**
    - Flag di invocazione di **`make`**
  - **`MAKECMDGOALS`**
    - Elengo dei goal di **`make`**
  - **`MAKELEVEL`**
    - Livello di ricorsione di **`make`**

Variabili predefinite

- **Comandi di compilazione fondamentali**
  - **`CPP`** Preprocessore **`$(CC) -E`**
  - **`CC`** Compilatore C **`gcc`**
  - **`CXX`** Compilatore C++ **`g++`**
  - **`AS`** Assembler **`as`**
  - **`AR`** Tool di gestione delle librerie **`ar`**

- **Comandi di compilazione aggiuntivi**
  - **`LEX`** Generatore di scanner **`lex`**
  - **`YACC`** Generatore di parser **`yacc`**

- **Altri comandi**
  - **`RM`** Rimozione di file **`rm -f`**

Variabili predefinite

- **Opzioni per i comandi di compilazione fondamentali**
  - **`CPPFLAGS`** Opzioni per il preprocessore
  - **`CFLAGS`** Opzioni di compilazione
  - **`CXXFLAGS`** Opzioni di compilazione
  - **`LDFLAGS`** Opzioni per il linker
  - **`ARFLAGS`** Opzioni per la costruzione di librerie
  - **`ASFLAGS`** Opzioni per l'assembler

- **Opzioni per i comandi di compilazione aggiuntivi**
  - **`LFLAGS`** Opzioni per il generatore di scanner
  - **`YFLAGS`** Opzioni per il generatore di parser

Funzioni

- **Nella definizione di variabili è possibile ricorrere ad alcune funzioni di utilità**
  - Funzioni per la manipolazione di stringhe
  - Funzioni per la manipolazione di nomi di file
  - Funzioni definite dall'utente

- **La chaimata di una funzione ha la forma**

```
$(function [arg,...])
```

- **In cui**
  - ***function*** Nome della funzione
  - ***arg*** Argomenti della funzione

Manipolazione di stringhe

- **`$(subst from,to,text)`**
  - Sostituisce la stringa ***`from`*** con la stringa ***`to`*** nella stringa ***`text`***

- **`$(patsubst patt,repl,text)`**
  - Sostituisce il pattern ***`patt`*** con la stringa ***`repl`*** nella stringa ***`text`***
  - Il pattern si basa sul carattere `%` usato per gli stem
  - La stringa ***`repl`*** può contenere il carattere `%`

- **`$(strip text)`**
  - Rimuove gli spazi all'inizio e alla fine della stringa ***`text`***
  - Sostituisce le sequenze di più spazi con spazi singoli

Manipolazione di stringhe

- **`$(findstring key,text)`**
  - Cerca la stringa ***`key`*** nella stringa ***`text`***
  - Ritorna ***`key`*** se presente, una stringa vuota altrimenti

- **`$(filter patt[...],list)`**
  - Ritorna le parole presenti nella lista ***`list`*** che soddisfano uno dei pattern ***`patt`***

- **`$(filter-out patt[...],text)`**
  - Ritorna le parole presenti nella lista ***`list`*** che non soddisfano uno dei pattern ***`patt`***

- **`$(sort list)`**
  - Ordina le parole nella lista ***`list`*** e ritorna la lista ordinata ottenuta

Manipolazione di stringhe

- **`$(word n,`*`text`*`)`**
  - Ritorna la parola ***`n`***-esima nel testo (o lista) ***`text`***

- **`$(wordlist `*`start`*`,`*`end`*`,`*`text`*`)`**
  - Ritorna le parole dalla posizione ***`start`*** alla posizione ***`end`*** nel testo (o lista) ***`text`***

- **`$(words `*`text`*`)`**
  - Ritorna il numero di parole nel testo ***`text`***

Manipolazione di nomi di file

- **`$(dir names)`**
  - Ritorna la direcory dei file nella lista ***`names`***
- **`$(notdir names)`**
  - Ritorna il nome privo di directory dei file nella lista ***`names`***
- **`$(suffix names)`**
  - Ritorna le estensioni dei file nella lista ***`names`***
- **`$(basename names)`**
  - Ritorna tutto tranne l'estensione dei file nella lista ***`names`***
- **`$(wildcard glob)`**
  - Esegue l'espansione del glob ***`glob`***

Manipolazione di nomi di file

- **`$(addsuffix suffix,`*`names`*`)`**
  - Aggiunge il suffisso ***`suffix`*** ai file nella lista ***`names`***

- **`$(addprefix prefix,`*`names`*`)`**
  - Aggiunge il prefisso ***`prefix`*** ai file nella lista ***`names`***

- **`$(join list1,list2)`**
  - Concatena ogni elemento di list1 con il corrispondente elemento di list2

Funzioni definite dall'utente

- **Una funzione definita dall'utente**
  - È una variabile
  - Ha come valore una espressione complessa
  - L'espressione contiene le variabili speciali **`$(1)`**, **`$(2)`**
    - Indicano i parametri formali

- **La chiamata di una funzione ha la sintassi**

  `$(call` ***`var`*****`,`*****`param`**`[,…])`

- **In cui**
  - ***`var`*** Nome della funzione, ovvero della variabile
  - ***`param`*** Valori dei parametri attuali

Funzioni definite dall'utente

- Inverte una lista di due parole

```
# Reverses a two-words list
reverse = $(2) $(1)

# Example
result = $(call reverse hello,world)

# The call produces:
#   $(1) = hello
#   $(2) = world
# The result is:
#   result = world hello
```

Funzioni definite dall'utente

- Cerca nel path un eseguibile e ritorna il percorso

```
# Return the pathname of an executable lookin in PATH
exefind = $(firstword \
            $(wildcard \
              $(addsufix $(1),\
                $(subst :, ,$(PATH)\
           ))))

# Example
PATH = .:/usr/bin:/bin:/sbin
result = $(call exefind ls)

# The call produces:
#   $(1) = ls
# Intermediate functions return (innermost first)
#   $(subst ...)       = . /usr/bin /bin /sbin
#   $(addsuffix ... ) = ./ls /usr/bin/ls /bin/ls /sbin/ls
#   $(wildcard ...)    = /bin/ls
# The result is:
#   result = $(firstword ...) = /bin/ls
```