

Prendi il makefile e scappa – come scrivere un makefile universale – pt.2

Louise: Sai una cosa? Presto avremo un bambino.

Virgil: Scherzi...

Louise: No! Avremo proprio un bambino: me l'ha detto il dottore, è sicuro. Sarà il mio regalo per Natale.

Virgil: Ma a me bastava una cravatta!

Nel divertentissimo e bel *mockumentary* **Prendi i soldi e scappa** del Maestro **Woody Allen**, il protagonista **Virgil Starkwell** diceva che ha lui bastava una cravatta... a noi invece basta un bel **makefile universale**! Spero che abbiate già letto **la prima parte** di questo articolo... *Non l'avete ancora fatto? Subito a leggerla e poi tornate qua!*



...presto avremo un makefile...

Rieccoci con il nostro *makefile universale*. Ora che avete (ri)letto l'articolo precedente sapete già che questo sarà un'altro post veloce, e non propriamente sul **C**: in questa

seconda parte, come promesso, prenderò il nostro *makefile* per aggiungere una funzionalità (spero) molto interessante: la creazione di una **shared-library** per **Linux** (*una .so, per gli amici*). Vediamo schematicamente quali sono i passi da eseguire (su un Linux della famiglia Debian) per creare e usare una *shared-lib*:

1. Creare una directory per condividere la shared-lib, ad esempio: `"/usr/local/lib/pluto"` 2. Modificare (come vedremo tra poco) il *makefile* per generare la shared-lib (che chiameremo `"libmyutils.so"`) e copiarla in `"/usr/local/lib/pluto"`. 3. Aggiungere in `"/etc/ld.so.conf.d"` un nuovo file `"libmyutils.conf"` che conterrà le seguenti due linee (la prima è solo un commento): `# libmyutils.so default configuration /usr/local/lib/pluto` 4. Rendere disponibile la nuova shared-lib eseguendo: `sudo ldconfig`

E proseguiamo: supponiamo di usare lo stesso progetto dell'altra volta (si chiamava *pluto*). I nostri file sono organizzati (ancora) in una maniera *canonica*, questa volta in quattro *directory* (l'altra volta erano tre): *pluto*, *lib*, *libmyutils* e *include*. La directory aggiunta è *libmyutils*, e contiene i sorgenti della *shared-lib* che vogliamo creare. Nella *directory pluto* troviamo il *main* del progetto e il *makefile*, nella *directory lib* troviamo gli altri sorgenti dell'applicazione *pluto* e, infine, nella *directory include* troviamo gli *header-files* comuni (*ma tutta la struttura era già ben descritta nell'articolo precedente, no? Consultarla per fugare eventuali dubbi*). E ora vediamo il nuovo *makefile*:

```
# variabili
SRCS = $(wildcard *.c)
SRCS_LIB = $(wildcard ../lib/*.c)
SRCS_SHLIB = $(wildcard ../libmyutils/*.c)
OBJS = $(SRCS:.c=.o)
OBJS_LIB = $(SRCS_LIB:.c=.o)
OBJS_SHLIB = $(SRCS_SHLIB:.c=.o)
DEPS = $(SRCS:.c=.d)
DEPS_LIB = $(SRCS_LIB:.c=.d)
DEPS_SHLIB = $(SRCS_SHLIB:.c=.d)
```

```

PATHLIB = /usr/local/lib/pluto
NAMELIB = libmyutils.so
OBS_APP = $(OBS) $(OBS_LIB) # compilatore e linker (ometto
per semplicità la variabile LD)
CC = gcc # flag per il preprocessore di CC per i file oggetto
dei due target (qui ometto CFLAGS)
CPPFLAGS = -I../include -g -O2 -Wall -pedantic -pthread -
DUNA_MIA_DEFINE -MMD -MP -std=c11
CPPFLAGS_SHLIB = -fpic -I../include -g -O2 -Wall -pedantic -
MMD -MP -std=c11 # flag per il linker LD per la creazione dei
due target (qui ometto LDLIBS)
LDFLAGS = -L$(PATHLIB) -lmyutils -pthread
LDFLAGS_SHLIB = -shared -fpic # i due target: shared-lib e
applicazione
all: libmyutils pluto # creazione del target applicazione
"pluto"
pluto: $(OBS_APP) $(CC) $^ -o $@ $(LDFLAGS) # creazione del
target shared-lib "libmyutils.so" con copia nella directory
destinazione
libmyutils: $(OBS_SHLIB) $(CC) $^ -o $(NAMELIB)
$(LDFLAGS_SHLIB) mv $(NAMELIB) $(PATHLIB) # creazione degli
object files per la applicazione
$(OBS_APP): %.o: %.c # target (i file .o) e da chi dipende (i
file .c) $(CC) $(CPPFLAGS) -c $< -o $@ # creazione degli
object files per la shared-lib
$(OBS_SHLIB): %.o: %.c # target (i file .o) e da chi dipende
(i file .c) $(CC) $(CPPFLAGS_SHLIB) -c $< -o $@ # direttive
phony
.PHONY: clean # pulizia progetto ($(RM) è di default "rm -f")
clean: $(RM) $(OBS) $(OBS_LIB) $(OBS_SHLIB) $(DEPS)
$(DEPS_LIB) $(DEPS_SHLIB) pluto # creazione dipendenze
-include $(DEPS) $(DEPS_LIB) $(DEPS_SHLIB)

```

Come avrete notato il nuovo *makefile* presentato è uno stretto parente di quello vecchio e continua ad essere veramente semplice e universale: fa tutto quello che serve, compresa la generazione dei file di dipendenza dagli *header*, e possiamo usarlo per qualsiasi progetto, indipendentemente dal numero di *file* (le *directory lib* e *include* potrebbero essere vuote oppure contenere centinaia di *file*). Possiamo aggiungere e togliere sorgenti e *header* e ricompilare senza modificare una

sola linea del *makefile*, perché lui si adatta automaticamente a quello che trova nelle *directory* del progetto: cosa vogliamo di più?

Però adesso è il caso di evidenziare alcune delle differenze rispetto alla versione originale, fermo restando che le descrizioni dei vari punti fatte nell'articolo precedente restano valide e non è il caso di ripeterle qui. Vediamo, allora, solo dettagli e differenze:

- **# variabili**

Qui ci sono le stesse della versione originale a cui ho aggiunto quelle necessarie a descrivere sorgenti, oggetti e dipendenze relativi alla *shared-lib*: **SRCS_SHLIB**, **OBJS_SHLIB** e **DEPS_SHLIB**. In più ho aggiunto *pathname* (**PATHLIB**) e *name* (**NAMELIB**) della libreria.

- **# compilatore e linker (ometto per semplicità la variabile LD)**

Qui c'è, ancora, il compilatore da usare, ma ho ometto (come evidenziato nel commento) il *linker* per semplificare rispetto alla versione originale.

- **# flag per il preprocessore di CC per i file oggetto dei due target (qui ometto CFLAGS)**

Qui ho messo i *flag extra* da assegnare al preprocessore C e ai programmi che lo utilizzano, e ho messo anche i *flag* aggiuntivi da fornire al compilatore C, per cui ho ometto (come evidenziato nel commento) la variabile **CFLAGS** per semplificare rispetto alla versione originale. Come si nota, questa versione ha due tipi di flag: **CPPFLAGS** e **CPPFLAGS_SHLIB**, visto che i file dell'applicazione e quelli della *shared-lib* si devono compilare con modalità differenti.

- **# flag per il linker LD per la creazione dei due target (qui ometto LDLIBS)**

Qui ho messo i *flag* aggiuntivi da dare al compilatore quando deve invocare il *linker* **ld**, e ho messo anche i nomi delle librerie forniti al compilatore, per cui ho

omesso (come evidenziato nel commento) la variabile **LDLIBS** per semplificare rispetto alla versione originale. Come si nota, questa versione ha due tipi di flag: **LDFLAGS** e **LDFLAGS_SHLIB**, visto che i file dell'applicazione e quelli della *shared-lib* si devono *linkare* con modalità differenti.

- **# i due target: shared-lib e applicazione**

Qui ci sono gli obiettivi di creazione: nel nostro caso la libreria *libmyutils* e l'applicazione *pluto*: il comando *make* senza argomenti esegue entrambi gli obiettivi (la parola chiave è *all*), ma si può bypassare questo eseguendo, ad esempio, *make libmyutils* che crea solo la *shared-lib*.

- **# creazione del target applicazione "pluto"**

Qui si mette il comando per *linkare* i file oggetto creati e produrre il file eseguibile finale. Notare che con la direttiva **-L** contenuta in **LDFLAGS** indichiamo al *linker* dove si trova la libreria *libmyutils.so*. Notare anche che questa direttiva serve solo a livello *linker*, mentre, a livello esecuzione delle applicazioni che usano la nostra *shared-lib*, servono i passi della lista descritta all'inizio del post (in particolare i passi 3 e 4).

- **# creazione del target shared-lib "libmyutils.so" con copia nella directory destinazione**

Qui ci sono le istruzioni per la creazione della *shared-lib libmyutils.so* e per spostarla (col comando Linux *mv*) nella directory destinazione.

- **# creazione degli object files per la applicazione**

Qui si mette il comando per compilare ogni sorgente dell'applicazione e creare il file oggetto corrispondente, attivando (attraverso le variabili definite precedentemente) tutte le opzioni del compilatore che ci servono. Notare il *trucco* necessario per diversificare la creazione di questi oggetti rispetto agli oggetti della *shared-lib*: attraverso la direttiva **"\$(OBJJS_APP): %.o: %.c"** si dice al comando

make che solo questi oggetti si compilano con il comando della linea successiva che usa i flag **CPPFLAGS**.

▪ **# creazione degli object files per la shared-lib**

Qui si mette il comando per compilare ogni sorgente della *shared-lib* e creare il file oggetto corrispondente, attivando (attraverso le variabili definite all'inizio) tutte le opzioni del compilatore che ci servono. Notare il *trucco* necessario per diversificare la creazione di questi oggetti rispetto agli oggetti dell'applicazione: attraverso la direttiva **"\$(OBS_SHLIB): %.o: %.c"** si dice al comando *make* che solo questi oggetti si compilano con il comando della linea successiva che usa i flag **CPPFLAGS_SHLIB**.

E concludo con tre ultime note generiche e (spero) interessanti:

1. Sicuramente, avrete notato che il *path* scelto per la *shared-lib* è **/usr/local/lib/pluto** e questa non è una scelta casuale: **/usr/local** è una delle *directory* usate su **Linux** per aggiungere pacchetti non tipici della distribuzione base, quindi *Software* che si installa a parte o, come nel nostro caso, creato e aggiunto localmente: **/usr/local** è, in pratica, una *directory ombra* della **root**, e contiene tutte le versioni *locali* delle *directory* di sistema (*bin*, *etc*, *include*, *lib*, ecc.). Nel nostro caso la libreria la aggiungiamo in **/usr/local/lib**, mentre un (eventuale) *header* si dovrebbe mettere **/usr/local/include**.
2. Si noti che per compilare e *linkare* la *shared-lib* si usano due direttive fondamentali: **"-fpic"** (in compilazione e *link*) e **"-shared"** (solo in *link*). E ribadisco: **"-fpic"** si usa sia in compilazione che in *link*: questo è un dettaglio che molte delle guide che si trovano in rete omettono e può essere una possibile causa di strani malfunzionamenti di una *shared-lib*.
3. Nell'esempio ho usato un flag, **"-pthread"** (in

compilazione e *link*) che si usa solo per applicazioni *multithread*, quindi avrei potuto ometterlo per questo esempio generico, ma ho voluto evidenziare che anche questo *flag* si usa sia in compilazione che in *link*, un particolare che molti dimenticano.

E qua finiscono le differenze e i dettagli. Credo che per oggi possa bastare... fatevi un piccolo progetto di prova (ad esempio usando funzioni semivuote che scrivono solo "*Ciao, sono la funzione xyz*") e provate il nuovo *makefile universale*: scoprirete che è veramente facilissimo da usare!

Ciao e al prossimo post!

<https://italiancoders.it/prendi-il-makefile-e-scappa-come-scrivere-un-makefile-universale-pt-2/>