Make e Makefile

http://idefix.mi.infn.it/~palombo/didattica/Lab-TNDS/CorsoLab/LezioniFrontali/Intro4.pdf

Fernando Palombo

A cosa serve un Makefile

- Il comando UNIX make cerca all'interno della directory dove viene eseguito questo comando un file di nome makefile (o anche Makefile) il quale istruisce il sistema operativo su come compilare un insieme di più file.

- C/C++ permettono la **compilazione separata** : il codice viene scritto in più file .cc (con generalmente i corrispondenti file .hh) i quali vengono compilati separatamente. Il comando make, utilizzando le istruzioni contenute nel makefile, permette di compilare e linkare tutti i file in modo simultaneo, estremamente pratico e veloce.

- L'utilizzo di make e la scrittura del makefile è molto semplice e allo stesso tempo è uno strumento molto potente. Più sono i file da compilare più l'utilizzo del makefile è necessario. Ma è anche comodo e utile con pochi file.

- Per illustrare il problema e l'utilità del comando make consideriamo un esempio (ricerca di zeri con il metodo della bisezione). Il codice si trova in "http://www.mi.infn.it/~palombo/didattica/Lab-TNDS/CorsoLab/Applicazioni-Web/Zeri/Bisezione/"

A cosa serve il makefile

- In questa directory Bisezione ci sono tre file .cc e due file .hh. Per creare l'eseguibile (che qui chiamo bisez) io posso dare questo comando:

  g++ -o bisez bisezione.cc f.cc main.cc

- Eseguendo questo comando potete controllare che nella vostra directory è apparso il file **bisez** (l'eseguibile). **Non vengono creati i file oggetto** ! Col comando ./bisez voi runnate il programma ed avete i vostri risultati

- Potrei compilare i file anche così:

  g++ –c bisezione.cc f.cc main.cc

  Questo comando compila i file .cc creando i file oggetto (file .o).
  Posso creare l'eseguibile bisez dai file oggetto:

  g++ -o bisez bisezione.o f.o main.o

  (posso anche scrivere g++ bisezione.o f.o main.o –o bisez)

- Si noti che il primo modo dove il programmatore usa un solo comando è inefficiente. Se anche uno solo dei file , f.cc, è stato cambiato, il comando dato ricompila sempre di tutti e tre i files .cc

A cosa serve il makefile

- Poiché main.cc e bisezione.cc non sono cambiati il compilatore potrebbe usare i file oggetto main.o e bisezione.o creati in precedenza.

- È chiaro che se si hanno molti file da compilare il problema si complica enormemente. Vediamo come possiamo risolverlo.

- Da ogni file .cc compilandolo ho un file .o e dai file .o costruisco l'eseguibile. C'è quindi una dipendenza dell'eseguibile dai file oggetto e una dipendenza dei file oggetto dai file sorgente .cc

- Quando si crea o si altera un file .cc o compilando si crea un file oggetto .o o quando linkando più file oggetti si crea l'eseguibile, ognuno di questi processi avviene in un determinato tempo (timestamp).

- Controllando il timestamp (basterebbe digitare ls –lat) posso controllare se l'eseguibile sia stato creato prima di qualche file oggetto da cui dipende l'eseguibile o se qualche file oggetto sia stato creato prima di qualche cambiamento nel corrispondente file sorgente .cc

Makefile

- Il makefile tiene conto di queste dipendenze tra file .cc, .o ed eseguibile e della sequenza temporale dei timestamp per decidere quali file deve rigenerare

- Il Makefile è un file senza estensione che l'utente deve scrivere nella stessa directory dei file sorgente.

- il Makefile contiene una o più regole. Ogni regola contiene:
  - un target (tipicamente un file)
  - le dipendenze (cioè i file da cui dipende il target)
  - il comando da eseguire per costruire (o aggiornare) il target

- La sintassi di una regola è:

  <target>: <linea delle dipendenze>

  <TAB> <linea di comando>

- Si noti i due punti tra il target e la linea delle dipendenze e si **NOTI** il <TAB> prima della linea di comando

Makefile

- Facciamo un esempio:

```
bisezione.o: bisezione.cc bisezione.hh f.hh
        g++ -c bisezione.cc
```

- Dopo il target ed i due punti appaiono i file da cui dipende il target. Questa linea è usata da make per decidere (sulla base dei timestamp) quando il target deve essere ricostruito. Se per esempio il target bisezione.o ha un timestamp precedente a quello di bisezione.cc o a quello di bisezione.hh o a quello di f.hh allora vuol dire che bisogna ricreare il file bisezione.o e make passa alla lina di comando e riaggiorna il file oggetto di bisezione.cc
- Un primo esempio di Makefile completo per il nostro progetto Bisezione è:

```
bisez: main.o bisezione.o f.o
        g++ -o bisez main.o  bisezione.o f.o
main.o: main.cc  bisezione.hh f.hh
        g++ -c main.cc
bisezione.o: bisezione.cc bisezione.hh f.hh
        g++ -c bisezione.cc
f.o: f.cc f.hh
        g++ -c f.cc
```

Makefile con Dummy Target

- Si noti che per ogni target di tipo .o file le dipendenze sono quasi sempre i corrispondente file .cc e .hh e gli altri file .hh (dell'utente) che sono inclusi nel file .cc (per esempio in main.cc abbiamo f.hh e bisezione.hh)

- Generalmente scopo del Makefile è di creare un eseguibile. Questo target è spesso il primo target. Il comando make apre il file Makefile ed esegue i comandi dal primo target
- Il Makefile può contenere più file eseguibili ed è possibile dare il comando make in modo da eseguire i comandi per uno specifico target. Per esempio nel makefile precedente potrei digitare make bisezione.o crea solo il target bisezione.o

- I target nel nostro esempio precedente sono tutti file. Ci sono alcuni casi in cui questo non è vero. Questi si chiamano **dummy target**. Un dummy target che viene usato (quasi) sempre è **make clean**

Makefile ; Dummy target

```
bisez: main.o bisezione.o f.o
        g++ -o bisez main.o  bisezione.o f.o
main.o: main.cc  bisezione.hh f.hh
        g++ -c main.cc
bisezione.o: bisezione.cc bisezione.hh f.hh
        g++ -c bisezione.cc
f.o: f.cc f.hh
        g++ -c f.cc
clean:
        rm -f  bisez  *.o   *~
```

- make arriva al target clean ed esegue i comandi (come se dovesse creare il file target clean). Nel nostro esempio il comando rm –f rimuove l'eseguibile, tutti i file oggetto e tutti i file che terminano con una ~ (e che sono file di backup dell'editor). Questo comando è molto utile perché permette di rimuove tutti questi file quando c'è il sospetto che qualcosa sia sbagliato. In questo modo si forza Makefile a ricreare tutti i file target.
- Usualmente prima si esegue make clean per ripulire e poi make per ricreare l'eseguibile

Macro nel Makefile

- L'uso di macro nel makefile permette di non ripetere più volte lo stesso testo e di rendere il makefile facilmente modificabile . Definizione di una macro : NOME_MACRO = stringa di testo

- Nel makefile ogni $(NOME_MACRO) viene sostituito con "stringa di testo". Le parentesi tonde possono anche essere graffe.

- Per convenzione il nome della macro è scritta con caratteri maiuscolo o con underscore

- In genere le macro sono scritte in testa al makefile. Esempi di comuni macro:
  CC = g++ nome del compilatore utilizzato
  DEBUG = -g è il flag per chiedere il debugging
  LFLAGS = -Wall $(DEBUG) (-Wall stampa tutti i warnings, $DEBUG degugging nel link)
  CFLAGS = -Wall –c $(DEBUG) flag usati in compilazione include –Wall, -c per la compilazione e il debugging in compilazione. Si noti la macro di una macro

Makefile con Macro

```
CC = g++
OBJS = main.o  bisezione.o f.o

bisex:  $(OBJS)
        $(CC) –o  bisez $(OBJS)

main.o: main.cc  bisezione.hh f.hh
        $(CC) –c main.cc

bisezione.o: bisezione.cc
        $(CC)  -c bisezione.cc
f.o: f.cc
        $(CC)  -c f.cc
clean:
        rm -f  bisez  *.o   *~
```

Makefile con Macro

Se volessi aggiungere la stampa di tutti i warning e il degugging in fase di compilazione e di link potrei aggiungere altre macro :

```
CC = g++
LFLAGS = -Wall  -g
CFLAGS = -Wall  –c  -g
OBJS = main.o  bisezione.o f.o

bisex:  $(OBJS)
        $(CC)  –o  bisez  $(LFLAGS)  $(OBJS)

main.o: main.cc  bisezione.hh f.hh
        $(CC)   $(CFLAGS) main.cc

bisezione.o: bisezione.cc f.hh
        $(CC)   $(CFLAGS)  bisezione.cc
f.o: f.cc f.hh
        $(CC)  $(CFLAGS)  f.cc
clean:
        rm -f  bisez  *.o   *~
```

Makefile con Macro e Commenti

Per rendere più leggibile il Makefile è opportuno aggiungere linee commenti (che sono preceduta da #) per dire cosa fa il Makefile e per spaziare le diverse regole. Esempio

```
#
# Makefile for zero's search with Bisection method
# Use of g++ compiler  version ….
#
CC = g++
LFLAGS = -Wall  -g
CFLAGS = -Wall  –c  -g
OBJS = main.o  bisezione.o f.o
#
bisez:  $(OBJS)
            $(CC)  –o  bisez  $(LFLAGS)  $(OBJS)
#
main.o: main.cc  bisezione.hh f.hh
            $(CC)   $(CFLAGS) main.cc
# ecc ecc ecc
```

Makefile

- Se i file che stiamo compilando per creare l'eseguibile hanno bisogno di un package (tipo ROOT) allora bisogna istruire il Makefile dove trovare le librerie e gli header file del package.

- Volendo usare ROOT i comandi per avere il PATH alla sua libreria e include file sono: root-config –libs e root-config –cflags

- Inserisco nel makefile le due macro:
  INCS = `root-config --cflags`
  LIBS = `root-config --libs`
  e nel comando per compilare il file .cc aggiungerò: $(INCS)
  mentre nel comando che crea l'eseguibile aggiungerò: $(LIBS)

- Per sintassi del make più avanzate (tipo uso e ridefinizione di regole implicite, variabili automatiche ed altro ancora si veda il manuale gnu del make:
  http://www.gnu.org/s/make/manual/make.pdf