ESD Electronic Systems Design

Synthesis Verification Testing Power Communication

Building, Debugging, Documenting

Dipartimento di Informatica

Università di Verona, Italy

Agenda

- Building
  - GNU Make
- Debugging
  - GNU Debugger
- Documenting
  - Doxygen

GNU Make

Compilazione

Build chain

- Una build chain automatica è utile per compilare:
  - Decine/centinaia di file sorgenti
  - Rapidamente
    - Senza ri-eseguire continuamente il compiler
    - Senza ri-compilare file già compilati
  - Facilmente
    - Compilazione semplificata anche per terze persone
  - Su varie piattaforme

GNU Make

- Build chain tradizionale dell'ambiente UNIX
- Richiede un file di configurazione
  - I.e., il **Makefile**
- Basato sul concetto di **dipendenza**
- Basato sul concetto di **timestamp**

- Svantaggi
  - Poco portabile
  - Sensibile a spazi e indentazioni

Dipendenza e Timestamp

- Dipendenza
  - Ogni target dipende da (è costruito a partire da) una lista di altri target, che devono essere costruiti prima di esso
    - Un file .o dipende da un .c
    - Un eseguibile dipende da un insieme di file .o
- Timestamp
  - Un target è considerato *out-of-date* se il tempo della sua ultima modifica è precendete a quello di almeno una delle sue dipendenze
  - Quando un target è out-of-date, il target è ricompilato
    - Quando un file .c è modificato, il corrispondente .o è ri-compilato e l'eseguibile ri-linkato

Makefile

- Il Makefile è il file che contiene tutte le informazioni necessarie alla compilazione

- Specifica:
  - Variabili di configurazione
    - che compilatore e linker usare
    - Flag da passare al compilatore e linker
    - Indirizzi di cartelle contenenti header e librerie
    - Etc..
  - Direttive di compilazioni (istruzioni per la compilazione)

```
<target>: <file sorgente principale> <altri file…>
\tab     <comandi di compilazione o linking>
```

- Nota: headers che dipendono da altri headers possono essere «aggiornati» usando `touch`

Makefile

- Variabili:
  - Assegnamento:

    `VARNAME=VALUELIST`
    - Assegna una lista di valori, viene fatto a tempo di lettura della variabile (portabile)

    `VARNAME:=VALUELIST`
    - Assegna una lista di valori **immediatamente** (GNU)

    `VARNAME+=VALUELIST`
    - Aggiunge la lista di valori a quelli assegnati precedentemente
  - Lettura:

    `$(VARNAME)`

Makefile

- Nomi comuni per le variabili
  - `CC` il compilatore C
    - E.g., gcc oppure clang
  - `CFLAGS` flag per il compilatore C
  - `CXX` il compilatore C++
    - E.g., `g++` oppure `clang++`
  - `CXXFLAGS` flag per il compilatore C++
  - `LD` il linker
  - `LDFLAGS` flag per il linker
  - `LDLIBS` librerie da linkare

Makefile

- Macro predefinite:
  - `$@` il target
  - `$<` la prima dipendenza
  - `$^` tutte le dipendenze
  - `$?` I file modificati
  - `%` il file corrente

Makefile: target speciali

- Primo target
  - Target di default
  - Viene eseguito quando make è chiamato senza parametri
- Nomi tipici di target
  - all compila e linka tutti i target
  - clean cancella tutti i file generati
  - install installa tutti i file generati
  - help stampa una lista di target disponibili
  - doc genera la documentazione
- `.PHONY`
  - Colleziona come dipendenze i target che non sono nomi di file generati (e.g., all, clean, help, etc..)

Makefile: esecuzione

- Eseguire make con Makefile, usando il target di default
  `make`
- Eseguire make in parallelo (opzionale: spcificare il numero massimo di thread in parallelo)
  `make -jN`
    - Massimo N thread in parallelo
  - E.g. per avere massimo 4 thread: `make -j4`
  - `make -j`
    - Nessun limite massimo di thread in parallelo
- Eseguire make con MyMake come file di configurazione:
  `make -f MyMake`
- Eseguire make per costruire un target specifico (e.g., app.x)
  `make app.x`

Makefile: esempio 1

- Files:
  - Implementation: `main.c com.c set.c`
  - Headers: `com.h set.h`, located into include.
- Auxiliary libraries:
  - Standard C math library.
  - Custom `libsupport.so` located into `lib`

Makefile: esempio 1

```
# Configurazione:
CC:= gcc
LD:= gcc
CFLAGS:= -c -Wall -Iinclude
LDFLAGS:= -Llib
LDLIBS:= -lsupport -lm

# Sorgenti:
SRCS:= main.c com.c set.c

# File oggetto:
OBJS:= $(SRCS:.c=.o)
```

Makefile: esempio 1

```
# Target di default:
all: myapp.x

myapp.x: $(OBJS)
  @echo Linking $@
  @$(LD) $(LDFLAGS) -o $@ $^ $(LDLIBS)

clean:
    @rm *.o myapp.x
# Compilazione che utilizza pattern matching:
%.o: %.c
    @echo $@
    @$(CC) $(CFLAGS) -o $@ $<

.PHONY: all clean
```

Makefile: esempio 1

- Vantaggi
  - Molto corto
  - Molto riusabile
- Svantaggi
  - Non ben ottimizzato:
    Non cattura le dipendenze relative agli header file

Makefile: esempio 2

```
# Fine tuned compiling:
main.o: main.c include/com.h include/set.h
	@echo $@
	@$(CC) $(CFLAGS) -o $@ $<
set.o: set.c include/set.h
	@echo $@
	@$(CC) $(CFLAGS) -o $@ $<
com.o: com.c include/com.h include/set.h
	@echo $@
	@$(CC) $(CFLAGS) -o $@ $<
```

Makefile: esempio 2

- Vantaggi
  - Cattura tutte le dipendenze
  - Estremamente sicuro, anche in caso di compilazione parallela
- Svantaggi
  - Lungo da scrivere
  - Non può essere riusato in altri progetti

GNU Debugger

Debugging

Debugging

- Verifica e debugging
  - Una delle fasi più complesse durante lo sviluppo SW
  - L'attività che richiede più tempo (70-80%)
- Fondamentale usare strumenti per
  - Semplificare il debugging del codice
  - Velocizzare la correzione di bug
  - Evitare comportamenti strani dovuti a metodi bufferizzati nel caso di errari di segmentazione
- Un debugger permette:
  - Sospensione dell'esecuzione di un programma in un punto specifico
  - Eseguire un programma passo dopo passo
  - Ispezione i valori delle variabili a run-time

GNU Debugger

- GDB (The GNU Debugger)
  - Libero e open-source
  - Debugger standard usato con gcc
  - Richiede di arricchire l'eseguibile
    - gcc -g compila con informazioni di debug
    - gcc -ggdb come -g ma esplicitamente per gdb
  - Strumento da linea di comando
    - DDD è la più famosa interfaccia grafica
      - Non più mantenuta
    - cgdb è un'interfaccia testuale basata su *curses*

GDB: Comandi base

- Caricare un eseguibile

```
gdb my_exec.x
gdb --args  my_exec.x <program args>
```

- Chiudere gdb

```
quit / q
```

- Assegnare eventuali argomenti all'eseguibile

```
set args <arguments>
```

- Aprire l'help di un argomento (e.g., di un comando)

```
help <argomento>
```

GDB Breakpoints

- Breakpoint
  - Marcatura di un punto nel codice sorgente
  - L'esecuzione del programma viene sospesa quando viene raggiunto un breakpoint
  - Utile per ispezionare un punto del programma a run-time
  - Sintassi:

```
break / b <file:line>
break / b <methodName>
delete <numero> (rimuove il breakpoint #<numero>
```

GDB Espressioni

- Monitoraggio: Watch point
  - Specifica di un **espressione**
    - Non un punto nel codice
  - Utile per controllare l'evoluzione di una variabile
  - Sintassi:

```
watch <espressione>
```

- Valutazione: Print
  - Stampa il risultato di un espressione
    - Utilizza il valore corrente delle variabili
  - Utile per monitorare variabili
  - Sintassi:

```
print / p <espressione>
```

GDB: Ispezione dello Stack

- Backtrace
  - Stampa la sequenza di chiamate a funzione
  - Permette di capire l'ordine di esecuzione
  - Sintassi:

```
backtrace / bt
```

- Frame
  - Stampa informazioni sullo stack di metodi corrente
  - Sintassi:

```
frame / f                 descrizione corta
info frame / info f       descrizione lunga
```

  - Cambiare frame:

```
frame <numero> / f <numero>
```

GDB: Esecuzione

- Eseguire un programma dall'inizio:
  `run / r [<argomenti>]`
- Continuare l'esecuzione dal punto raggiunto
  `continue / c`
- Eseguire la prossima riga *atomicamente*
  `next / n`
- Eseguire la prossima istruzione
  - Eventualmente entrando in metodi/funzioni
    `step / s`
- Ripeti l'ultimo comando
  `\return`

Doxygen

Documentazione

Documentazione

- Tipi di documentazione
  - Per gli utenti
    - Manuali, guide, tutoria, etc…
    - Documentazione di API (nel caso di librerie)
  - Per gli sviluppatori
    - Documentazione di API
    - Documentazione dell'implementazione
    - Altro
      - Use cases, UML, specifiche, etc…

Documentazione del codice sorgente

- Commenti delle API
  - Generazione automatica della documentazione
  - Richiede l'utilizzo di un formato preciso
    - Linguaggi per la generazione della documentazione disponibili per ogni linguaggio
  - Obiettivo: generare la documentazione «ufficiale» del progetto

- Commenti dei dettagli implementativi
  - Immersi nel codice sorgente
  - Possono utilizzare semplice linguaggio naturale
  - Obiettivo: rendere il codice sorgente comprensibile a terze parti o «riletture» future

Doxygen

- Tool per la generazione automatica della documentazione di API
  - Multi-piattaforma
  - Libero ed open-source
- Supporta diversi linguaggi di programmazione
  - C, C++, Java
- Supporta diversi formati di output
  - ***HTML***, Latex, RTF

Usare Doxygen

- Generare un file di configurazione (Doxyfile)

```
Doxygen -g
```

- Configurare il Doxyfile

```
emacs Doxyfile
```

- Eseguire doxygen con il Doxyfile di default

```
Doxygen
```

- Eseguire doxygen con un file di configurazione specifico

```
doxygen <confi file>
```

Configurazione di Doxygen

- La configurazione avviene **settando parametri nel Doxyfile**

- Parametri principali
  - PROJECT_NAME (e.g., MyLib)
    - Nome del progetto
  - OUTPUT_DIRECTORY (e.g. doc)
    - Directory dove salvare la documentazione
  - INPUT / FILE_PATTERNS (e.g. src)
    - Lista dei file (o directory) di ingresso
  - RECURSIVE (e.g. YES)
    - Se YES, allora visita le cartelle ricorsivamente
  - EXCLUDE / EXCLUDE_PATTERNS (*.java)
    - File da escludere
  - GENERATE_* (e.g. GENERATE_HTML)
    - Specifica del formato di uscita

Formato dei commenti

- I commenti in doxygen devono rispettare una forma particolare:

```
/** Commento parserizzato da Doxygen */
/// Commento parserizzato da Doxygen
```

  - Altri formati di commento vengono ignorati

```
/* Commento ignorato da Doxygen */
// Commento ignorato da Doxygen
```

- Informazioni speciali per la documentazione vengono date tramite *tag* speciali nella forma:

```
@tag
\tag
```

Tag principali di Doxygen

`@brief <commento>`

- Un breve commento sulla parte di codice seguente
- Utilizzato prima di blocchi di codice

`@param <nome parametro> <commento>`

- <commento> spiega il significato/utilizzo del parametro <nome parametro>
- Utilizzato per commentare dichiarazioni di metodi

`@return <comment>`

- Commenta il comportamento del valore di ritorno di un metodo
- Utilizzato per commentare dichiarazioni di metodi

`@throw <nome eccezione> <commento>`

- Commenta eccezioni lanciate da metodi (non per C)
- Utilizzato per commentare dichiarazioni di metodi

Tag di documentazione globale

- Informazioni sul contenuto di un file

```
/** @file
 * <comment >
 */
```

- Raggruppare metodi con una connessione logica

```
/** @name <groupName>*/
/*@{ */
<methodsWithTheirDocumentation>
/*@} */
```

Esempio

```
/** @name List accessors. */
/*@{ */

/** @brief Gets the element at given position.
 * Linear complexity.
 * @param l The list.
 * @param pos The position.
 * @return The stored element.
 */
void * getListElement( List * l, int pos );

/*@} */
```